SAMS

畅销全球的
经典C++教程

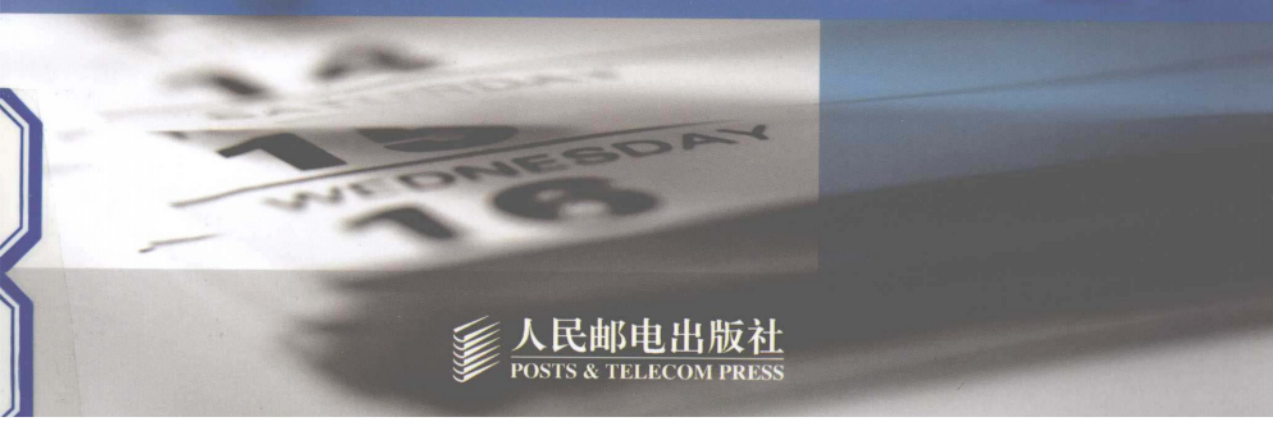
Sams Teach Yourself C++
in One Hour a Day
Seventh Edition

中文版
累计销量
超50000册

全面涵盖C++11新标准
帮助读者编写高效的C++应用程序

21天学通C++（第7版）

[美] Siddhartha Rao 著
袁国忠 译



人民邮电出版社
POSTS & TELECOM PRESS

对没有任何编程经验的新人和有其他语言编程经验的人来说，这是一本卓越的C++入门图书。

—— 独立评论人

21天学通C++（第7版）

倾听来自读者的真实评论，选择最合适的C++教程

本书非常适合初学者，内容很详实，对一些比较抽象的概念会做一些比喻，书中对一些容易弄错或者不容易注意到的问题都特别提出来。书中更为重要的思想是，它不断提醒哪些习惯是优秀的程序员应该有的，哪些习惯容易出问题。总之，这是非常经典的一这书。

—— 当当读者“行动如虫”

C++的经典图书很多，但是这一本书非常有特色……以我的亲身经历，对那些没有任何编程经验的人来说，这本书真的不错。

—— 中国亚马逊读者“后皇嘉树”

这本书完全面向零基础的C++初学者，示例代码很多，讲的也很详细，内容也很全面。

—— 当当读者“风般飘过”

内容浅显易懂，例子简单明了……实在不失为一本好书。

—— 当当读者“小白大帝”

我见过的最好的C++教材，内容有趣，通俗易懂。

—— 中国亚马逊读者“星涛晦暝”

读者可通过 www.ptpress.com.cn 或 http://vdisk.weibo.com/s/gvYGx 下载本书的所有源代码。

美术编辑：王建国

分类建议：计算机 / 程序设计 / C++
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-29624-5

ISBN 978-7-115-29624-5
定价：59.00 元

21天学通C++（第7版）

[美] Siddhartha Rao 著
袁国忠 译

图书在版编目（CIP）数据

21天学通C++ : 第7版 / (美) 罗奥 (Rao, S.) 著 ; 袁国忠译. — 北京 : 人民邮电出版社, 2012.12
ISBN 978-7-115-29624-5

Ⅰ. ①2… Ⅱ. ①罗… ②袁… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2012)第235664号

21 天学通 C++（第 7 版）

◆ 著 [美] Siddhartha Rao
译 袁国忠
责任编辑 傅道坤

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京昌平百善印刷厂印刷

◆ 开本：787×1092 1/16
印张：29.5
字数：870 千字 2012 年 12 月第 1 版
印数：1–3 500 册 2012 年 12 月北京第 1 次印刷

著作权合同登记号 图字：01-2012-5025 号
ISBN 978-7-115-29624-5

定价：59.00 元

读者服务热线：(010)67132692 印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

内容提要

本书通过大量短小精悍的程序，详细而全面地阐述了C++基本概念和技术以及C++11新增的功能，包括管理输入/输出、循环和数组、面向对象编程、模板、使用标准模板库以及lambda表达式等。这些内容被组织成结构合理、联系紧密的章节，每章都可在1小时内阅读完毕；每章都提供了示例程序清单，并辅以示例输出和代码分析，以阐述该章介绍的主题。为加深读者对所学内容的理解，每章末尾都提供了常见问题及其答案以及练习和测验。读者可对照附录D提供的测验和练习答案，了解自己对所学内容的掌握程度。

本书是针对C++初学者编写的，不要求读者有C语言方面的背景知识，可作为高等院校教授C++课程的教材，也可供初学者自学C++时使用。

作者简介

Siddhartha Rao 是全球领先的企业软件提供商 SAP AG 的技术专家。作为 SAP Product Security India 的负责人，其主要职责包括招募产品安全领域的专家以及制定软件开发最佳实践，以保持 SAP 软件的全球竞争力。作为一位 Microsoft Visual C++ MVP，他深信 C++11 有助于编写速度更快、更简洁、更高效的 C++应用程序。

Siddhartha 酷爱旅游，不放过任何一次探索新文化的机会。例如，本书就是在 4 个不同的国度创作而成的，其中包括法国布列塔尼一个面朝大西洋的奇特村庄。他期待着您对这部全球之作提出宝贵的建议。

献词

谨将本书献给我的父母和我的妹妹，他们是我坚强的后盾。

致谢

当我为了编写本书而通宵达旦地忙碌时，是我的朋友承担起了我的后勤工作，对此深表谢意。谢谢本书的所有编辑人员，正是他们的勤劳付出，才让本书出现在各位读者的书架上。

前　言

对 C++来说，2011 是个很特别的年份。在这一年，C++11 终于获批成为新标准，它新增了一些可提高编程效率的关键字和结构，让您能够编写更优质的代码。本书旨在帮助您循序渐进地学习 C++11，其中的章节经过仔细编排，从实用的角度介绍这种面向对象的编程语言的基本知识。读者只需每天花 1 小时，在学完本书后，就能掌握 C++11。

学习 C++的最佳方式是动手实践。本书包含丰富的代码示例，有助于读者提高编程技能，请务必亲自动手尝试这些代码。这些代码片段都使用了（在本书编写时）最新版本的编译器进行了测试，具体地说是 Microsoft Visual C++ 2010 和 GNU C++编译器 4.6 版，它们都支持大量的 C++11 功能。

针对的读者

本书从最基本的 C++知识开始介绍，读者只需具备学习 C++的愿望及了解工作原理的好奇心即可；虽然具备一些 C++知识会有所帮助，但这并非必需的。本书也可供熟悉 C++但想了解 C++11 新增功能的读者参考；如果你是专业程序员，第 3 部分“学习标准模板库”可以帮助你创建更优质、更实用的 C++应用程序。

本书内容

读者可根据自己对 C++的熟练程度，阅读感兴趣的部分。本书包含 5 部分。

- 第 1 部分“基本知识”，引导读者编写一些简单的 C++应用程序，并介绍一些在 C++的未妥协类型安全变量的代码中最常见的关键字。
- 第 2 部分“C++面向对象编程基础”，介绍类的概念，您将学习 C++如何支持封装、抽象、继承和多态等重要的面向对象编程原则。第 9 章将介绍 C++11 新增的移动构造函数，而第 12 章将介绍移动赋值运算符。这些功能有助于避免不必要的复制步骤，从而提升应用程序的性能。第 14 章是一个跳板，助您编写功能强大的 C++通用代码。
- 第 3 部分“学习标准模板库”，将帮助您使用 STL string 类和容器编写高效而实用的 C++代码。您将了解到，使用 std::string 可安全而轻松地拼接字符串，您不再需要使用 C 风格字符串（char*）。您可使用 STL 动态数组和链表，而无需自己编写这样的类。
- 第 4 部分“再谈 STL”，专注于算法，您将学习如何通过迭代器对 vector 等容器进行排序。在这部分，您将发现，通过使用 C++11 新增的关键字 auto，可极大地简化冗长的迭代器声明。第 22 章将介绍 C++11 新增的 lambda 表达式，这可极大地简化使用 STL 算法的代码。
- 第 5 部分“高级 C++概念”，阐述智能指针和异常处理等 C++功能。对 C++应用程序来说，这些功能并非必需的，但可极大地提高应用程序的稳定性和品质。在这部分的最后，简要地介绍了有助于编写杰出 C++应用程序的最佳实践。

本书体例

本书使用了下述提供更多信息的元素：

注意 提供与读者阅读的内容相关的信息。

C++11

突出 C++11 新增的功能。要使用这些功能，可能需要使用较新的编译器版本。

警告 提醒读者注意在特定情况下可能出现的问题或副作用。

提示 提供 C++编程最佳实践。

应该	不应该
提供当前章介绍的基本原理的摘要。	提供一些有用的信息。

本书使用不同的字体来区分代码和正文，全书都用特殊字体呈现代码、命令以及与编程相关的术语。

目　录

第1章 绪 论

欢迎使用本书！通过阅读本章，您将迈出成为高级C++程序员的第一步。

在本章中，您将学习：

- 为何C++是软件开发的标准；
- 输入、编译和链接第一个C++程序；
- C++11新增的功能。

1.1 C++简史

编程语言旨在让人更容易使用计算资源。C++并非一种新语言，但仍被广泛采用，并在不断改进。2011年，最新的C++标准获得了ISO标准委员会的批准，名为C++11。

1.1.1 与C语言的关系

C++最初由Bjarne Stroustroup于1979年在贝尔实验室开发，被设计为C语言的继任者。C语言是一种过程型语言，程序员使用它定义执行特定操作的函数，而C++是一种面向对象的语言，实现了继承、抽象、多态和封装等概念。C++支持类，而类包含成员数据以及操作数据的成员方法（方法类似于C语言中的函数）。其结果是，程序员需要考虑数据以及要用它们来做什么。一直以来，C++编译器都支持C语言，这具有向后与既有代码兼容的优势，但也存在缺点，那就是编译器非常复杂，因为随着C++的发展，编译器既要实现所有的新功能，又要向程序员提供这种向后兼容的功能。

1.1.2 C++的优点

C++是一种中级编程语言，这意味着使用它既可以高级编程方式编写应用程序，又可以低级编程方式编写与硬件紧密协作的库。在很多程序员看来，C++既是一种高级语言，让他们能够开发复杂的应用程序，又提供了极大的灵活性，让开发人员能够控制资源的使用和可用性，从而最大限度地提高性能。

虽然有更新的编程语言面世，如Java以及其他基于.NET的语言，但C++始终深受欢迎并在不断发展。较新的语言因提供了某些功能（如通过垃圾收集管理内存）让一些程序员钟爱有加，但在需要精确控制应用程序的性能时，他们还是会选择C++。当前，常常使用C++编写Web服务器，并使用HTML、Java或.NET编写前端应用程序。

1.1.3 C++标准的发展历程

经过多年的发展，C++得到了广泛接受和采纳，但存在多种不同版本，因为有很多不同的编译

器，它们风格各异。鉴于C++广受欢迎，且不同的版本之间存在差异，这导致了众多互操作性和移植方面的问题，需要对其进行标准化。

1998年，第一个C++标准获得了ISO标准委员会的批准，这就是ISO/IEC 14882:1998。2003年进行了修订，即为ISO/IEC 14882:2003。最新的C++标准于2011年8月获批，其官方名称为C++11（ISO/IEC 14882:2011），它包含一些雄心勃勃的改进。

注意

网上的很多文档仍称这个C++标准为C++0x。人们最初预期新标准将于2008或2009年获批，因此用x表示获批的年份。但提议的新标准到2011年才获批，因此将其称为C++11。换句话说，C++11是新的C++0x。

1.1.4 哪些人使用C++程序

无论你是什么人，做什么工作（无论是经验丰富的程序员，还是将计算机用于特定目的的人），都可能经常会用到C++应用程序和库。C++常用于开发操作系统、设备驱动程序、办公软件、Web服务器、基于云的应用程序和搜索引擎，甚至用于编写新编程语言编译器。

1.2 编写C++应用程序

当您在计算机上启动Notepad或VI时，实际上是命令处理器运行该程序的可执行文件。可执行文件是可运行的成品，应按程序员期望的那样做。

1.2.1 生成可执行文件的步骤

要创建可在操作系统中运行的可执行文件，第一步是编写一个C++程序。创建C++应用程序的基本步骤如下。

1. 使用文本编辑器编写C++代码。
2. 使用C++编译器对代码进行编译，将代码转换为包含在目标文件中的机器语言版本。
3. 使用链接程序链接编译器的输出，生成一个可执行文件（如Windows中的.exe文件）。

您在编程时创建的是文本文件，但微处理器无法处理这样的文件。在编译过程中，C++代码（通常包含在.CPP文本文件中）被转换为处理器能够处理的字节码。编译器每次转换一个代码文件，生成一个扩展名为.o或.obj的目标文件，并忽略这个CPP文件可能对其他文件中代码的依赖。解析这些依存关系的工作由链接程序负责。除将各种目标文件组合起来外，链接程序还建立依存关系，如果链接成功，则创建一个可执行文件，供程序员执行和分发。

1.2.2 分析并修复错误

大多数复杂应用程序很少能够一次通过编译并完美地运行，由众多程序员合作开发的应用程序尤其如此。无论使用什么语言（包括C++）编写，庞大或复杂的应用程序都需要运行很多次，以分析问题和发现Bug。修复Bug后，重新生成程序，再重复上述过程。因此，除编写、编译和链接等三个步骤外，开发过程通常还包括调试步骤。在这个步骤中，程序员使用工具（如监视点）和调试功能（如逐行执行应用程序）对应用程序中的异常和错误进行分析。

1.2.3 集成开发环境

很多程序员都喜欢使用集成开发环境（Integrated Development Environment，IDE）。集成开

发环境让您能够在一个统一的用户界面中完成输入、编译和链接步骤，它还提供了调试功能，让您能够更轻松地发现错误并解决问题。

提示

有很多免费的C++ IDE和编译器。在Windows和Linux系统中，最流行的分别是Microsoft Visual C++学习版和GNU C++编译器g++。如果您使用的是Linux系统，要使用g++编译器来开发C++应用程序，可安装免费的Eclipse IDE。

警告

在编写本书时，还没有全面支持C++11标准的编译器，但前述编译器支持C++11的很多主要功能。

应该	不应该
可使用简单的文本编辑器（如Notepad或gedit）来创建源代码，也可使用IDE。 保存文件时，务必使用扩展名.cpp。	不要使用富文本编辑器，因为它们经常会给您编写的代码添加标记。 不要使用扩展名.c，因为很多编译器都将这种文件视为C语言代码，而不是C++代码。

1.2.4 编写第一个C++应用程序

了解工具和步骤后，该编写第一个C++应用程序了，它在屏幕上打印Hello World!。

如果您使用的是Windows操作系统和Microsoft Visual C++学习版，可采取如下步骤。

1. 选择菜单“文件”>“新建”>“项目”，以新建一个项目。
2. 选择类型“Win32控制台应用程序”，并取消选择复选框“预编译头”。
3. 将项目命名为Hello，并用程序清单1.1所示的代码替换Hello.cpp中自动生成的内容。

如果您使用的是Linux操作系统，使用简单的文本编辑器（我使用的是Ubuntu上的gedit）创建一个包含程序清单1.1所示内容的CPP文件。

程序清单1.1 Hello Wrold程序（Hello.cpp）

```
1:  #include <iostream>
2:
3:  int main()
4:  {
5:    std::cout << "Hello World!" << std::endl;
6:    return 0;
7:  }
```

这个应用程序很简单，只是使用std::cout在屏幕上打印一行消息。std::endl命令cout换行。该应用程序退出时向操作系统返回零。

注意

默读程序时，知道特殊字符和关键字的发音可能会有所帮助。

例如，对于#include，可读作hash-include、sharp-include或pound-include，这取决于您以前的背景。

同样，对于std::cout，可读作standard-c-out。

警告

别忘了，魔鬼隐藏在细节中，这意味着您必须准确地输入程序清单中的代码。编译器对代码的要求非常严格；语句必须以；结尾，如果您错误地输入了:，就会陷入一片混乱。

1.2.5 生成并执行第一个C++应用程序

如果您使用的是Microsoft Viusual C++学习版，可在该IDE中按Ctrl + F5直接运行程序。这将编译、链接并执行应用程序。也可依次执行如下步骤。

1. 右击项目并选择“生成”，准备生成可执行文件。
2. 在命令提示符中，切换到可执行文件所属的文件夹（通常是项目文件夹中的Debug文件夹）。
3. 输入可执行文件的名称以运行它。

在Microsoft Visual C++中编写的程序与图1.1极其相似。

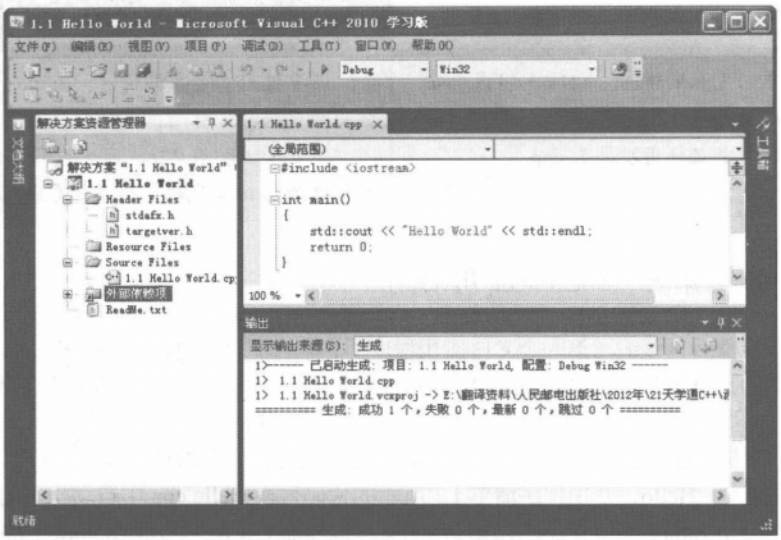


图1.1 在Visual C++ 2010学习版中创建的简单C++程序“Hello World”

如果您使用的是Linux系统，可使用如下命令行调用g++编译器和链接程序。

```
g++ -o hello Hello.cpp
```

该命令行让g++编译C++文件Hello.cpp，并创建一个名为hello的可执行文件。在Linux和Windows系统中，分别执行命令.\hello或Hello.exe，这将返回如下输出。

```
Hello World!
```

祝贺您踏上了学习有史以来最流行、最强大的编程语言之一的道路！

C++ ISO标准的意义

正如您看到的，通过遵守标准，可在多种平台（操作系统）中编译和执行程序清单1.1所示的代码——前提条件是有遵守标准的C++编译器。因此，如果要编写让Windows用户和Linux用户都能运行的软件，通过在编程中遵守标准（不使用编译器或平台特有的语义）是一种获取更多用户的简单方式，这让您无需针对要支持的每种环境进行编程。当然，编写不需要在操作系统级进行大量交互的应用程序时，这种方法的效果最佳。

1.2.6 理解编译错误

编译器的要求非常苛刻，但优秀的编译器会相当明确地指出错误在什么地方。如果您在编译程序清单1.1所示的应用程序时遇到问题，错误消息将与下面的内容极其相似（这是故意省略第

5 行末尾的分号导致的错误）：

```
hello.cpp(6): error C2143: syntax error : missing ';' before 'return'
```

这条错误消息来自 Visual C++编译器，对错误进行了详细描述。它指出了包含错误的文件的名称，在哪一行遗漏了分号（这里是第 6 行），还使用错误编号（这里是 C2143）对错误本身进行了描述。在这个例子中，虽然遗漏分号的是第 5 行，但错误消息却指出错误发生在第 6 行，这是因为对编译器来说，只有等它返回语句后，才能确定在返回前，前一条语句必须结束。如果您可在第 6 行开头添加分号，程序将通过编译！

注意

不同于 VBScript 等语言，在 C++中语句分行并不能自动结束语句。
在 C++中，一条语句可跨越多行。

1.3 C++11 新增的功能

如果您是经验丰富的 C++程序员，可能发现程序清单 1.1 所示的基本程序没有任何变化。虽然 C++11 可以与以前的 C++版本兼容，但仍然做了大量工作让这种语言使用起来更容易。

auto 让您能够定义这样的变量，即编译器将自动推断其类型，这简化了变量声明，同时又不影响类型安全。Lambda 函数是没有名称的函数，让您能够编写紧凑的函数对象，而无需提供冗长的类定义，从而极大地减少了代码。C++11 让程序员能够编写可移植的多线程 C++应用程序，同时确保它们遵守标准。这些应用程序支持并行执行范式，在用户升级到多核 CPU 以改善硬件配置时，其性能将相应地提升。

本书将讨论 C++11 所做的众多改进，这里列举的只是其中几项。

1.4 总结

在本章中，您学习了如何编写、编译、链接和执行第一个 C++程序。本章还简要地介绍了 C++的发展历程，并演示了如何在不同的操作系统中使用不同的编译器对同一个程序进行编译，从而证明了标准的重要意义。

1.5 问与答

问：可以忽略编译器发出的警告消息吗？

答：在有些情况下，编译器会发出警告消息。警告与错误的不同之处在于，相关代码行的语法是正确的，能够通过编译，但可能有更佳的编写方式。优秀编译器在发出警告的同时提供修复建议。

修复建议可能是一种更安全的编程方式，也可能让应用程序能够处理非拉丁语字符和符号。您应该留意警告，并相应地改进应用程序。除非您确定警告是误报，否则不要对其视而不见。

问：解释型语言与编译型语言有何不同？

答：诸如 Windows Script 等语言是解释型的，不需要编译。解释型语言使用解释器，解释器直接读取脚本文件（代码）并执行指定的操作。因此，要在计算机上执行脚本，必须安装解释器。在运行阶段，解释器在微处理器和代码之间充当翻译，因此性能通常会受到影响。

问：什么是运行错误？它与编译错误有何不同？

答：执行应用程序时发生的错误称为运行错误。在较旧的 Windows 版本中，您可能遇到过臭名昭

著的“非法访问（Access Violation）”错误，它就是运行错误。最终用户不会遇到编译错误，这种错误表明程序存在语法问题，禁止程序员生成可执行文件。

1.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

1.6.1 测验

1. 解释器和编译器有何不同？
2. 链接器的作用是什么？
3. 正常的开发周期包括哪些步骤？
4. C++11 标准如何更好地支持多核 CPU？

1.6.2 练习

1. 阅读下面的程序，在不运行它的情况下猜测其功能。

```
 1: #include <iostream>
 2: int main()
 3: {
 4:   int x = 8;
 5:   int y = 6;
 6:   std::cout << std::endl;
 7:   std::cout << x - y << " " << x * y << x + y;
 8:   std::cout << std::endl;
 9:   return 0;
10: }
```

2. 输入练习 1 中的程序，然后编译并链接它。它的功能是什么？与您的猜测相符吗？
3. 下面的程序存在什么样的错误？

```
1: include <iostream>
2: int main()
3: {
4:     std::cout << "Hello Buggy World \n";
5:     return 0;
6: }
```

4. 修复练习 3 中程序的错误，重新编译、链接并运行它。它的功能是什么？

第 2 章
C++程序的组成部分

C++程序由类、函数、变量及其他元素组成。本书的大部分内容将对这些组成部分进行深入解释，但为了解程序是如何组合在一起的，必须分析一个完整的工作程序。

在本章中，您将学习：

- C++程序的组成部分；
- 各部分如何协同工作；
- 函数及其用途；
- 基本输入输出操作。

2.1 Hello World 程序的组成部分

在第 1 章中，您编写的第一个 C++程序只是将句子 Hello World 打印到屏幕上，虽然如此，却包含了 C++程序最重要的基本构件。本节将以程序清单 2.1 为例，对所有程序都包含的重要部分进行分析。

程序清单 2.1 HelloWorldAnalysis.cpp：分析 C++程序

```
 1:  // Preprocessor directive that includes header iostream
 2: #include <iostream>
 3:
 4: // Start of your program: function block main()
 5: int main()
 6: {
 7:    /* Write to the screen */
 8:    std::cout << "Hello World" << std::endl;
 9:
10:    // Return a value to the OS
11:    return 0;
12: }
```

可将这个 C++程序划分为两个部分：以#打头的预处理器编译指令以及以 int main()打头的程序主体。

注意

第 1、4、7 和 10 行以//或/*打头，它们都是注释，将被编译器忽略。这些注释是供人类阅读的。注释将在下一节更详细地讨论。

2.1.1 预处理器编译指令#include

顾名思义，预处理器是一个在编译前运行的工具。预处理器编译指令是向预处理器发出的命令，总是以符号#打头。在程序清单 2.1 中，第 2 行的#include<filename>让预处理器获取指定文件（这里是 iostream）的内容，并将它们放在编译指令所处的位置。iostream 是一个标准头文件，这里包含它是因为第

8行使用了std::cout将Hello World打印到屏幕上，而该头文件包含std::cout的定义。换句话说，编译器之所以能够编译包含std::cout的第8行，是因为我们在第2行指示预处理器包含了std::cout的定义。

注意

在专业人员编写的C++应用程序中，包含的文件并非都是标准头文件。编写复杂的应用程序时，通常将其代码放在多个文件中，这使得需要在某些文件中包含其他文件。因此，如果需要在FileA中使用FileB中定义的元素，就需要在前者中包含后者。为此，通常在FileA中使用如下include语句：

```
#include "...relative path to FileB\FileB"
```

这里使用引号而不是尖括号来包含自己创建的头文件。尖括号（<>）通常用于包含标准头文件。

2.1.2 程序的主体——main()

预处理器编译指令的后面是程序的主体——main()函数，执行C++程序时总是从这里开始。声明main()时，总是在它前面加上int，这是一种标准化约定，表示main()函数的返回类型为int。

注意

在很多C++应用程序中，都使用了类似于下面的main()函数变种：

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

这符合标准且可以接受，因为其返回类型为int。括号内的内容是提供给程序的参数。该程序可能允许用户执行时提供命令行参数，如：

```
program.exe /DoSomethingSpecific
```

其中/DoSomethingSpecific是操作系统传递给程序的参数，以便在main中进行处理。

下面来看第8行，它实际执行该程序的功能：

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

cout（控制台输出，读作see-out）是将Hello World打印到屏幕上的语句。cout是在标准名称空间中定义的一个流（因此这里使用了std::cout），这里使用流插入运算符<<将文本Hello World放到这个流中。std::endl用于换行，将其插入流中相当于插入回车。每次需要将新实体插入流中时，都使用了流插入运算符。

C++流的优点是，将类似的语义用于另一种类型的流时，将执行不同的操作，如插入到文件而不是控制台。因此，流的用法非常直观，使用过一种流（如将文本写入控制台的cout）后，其他流（如帮助将文本文件写入磁盘的fstream）使用起来就非常容易。

流将在第27章更详细地讨论。

注意

实际文本（包括引号）“Hello World”被称为字符串字面量。

2.1.3 返回值

在C++中，除非明确声明了不返回值，否则函数必须返回一个值。main()也是函数，且总是返回一个整数。这个值返回给操作系统，根据应用程序的性质，这可能很有用，因为大多数操作系统都提供了查询功能，让您能够获悉正常终止的应用程序的返回值。在很多情况下，一个应用程序被另一个应用程序启动，而父应用程序（启动者）想知道子应用程序（被启动者）是否成功地完成了其任务。程序员可使用main()的返回值向父应用程序传递成功或错误状态。

注意

根据约定，程序员在程序运行成功时返回0，并在出现错误时返回-1。然而，返回值为整数，程序员可利用整个整数范围，指出众多不同的成功或失败状态。

警告

C++区分大小写，如果将int写成了Int、将void写成了Void或将std::cout写成了Std::Cout，程序将不能通过编译。

2.2 名称空间的概念

在这个程序中，使用的是 std::cout 而不是 cout，原因在于 cout 位于标准（std）名称空间中。

那么什么是名称空间呢？

假设调用 cout 时没有使用名称空间限定符，且编译器知道 cout 存在于两个地方，编译器应调用哪个呢？当然，这会导致冲突，进而无法通过编译。这就是名称空间的用武之地。名称空间是给代码指定的名称，有助于降低命名冲突的风险。通过使用 std::cout，可命令编译器调用名称空间 std 中独一无二的 cout。

注意

> 您使用 std（读作 standard）名称空间来调用获得 ISO 标准委员会批准，并在该名称空间中声明的函数、流和工具。

很多程序员发现，使用 cout 和 std 名称空间中的其他功能时，在代码中添加 std 限定符很繁琐。为避免添加该限定符，可使用声明 using namespace，如程序清单 2.2 所示。

程序清单 2.2 using namespace 声明

```
 1: // Pre-processor directive
 2: #include <iostream>
 3:
 4: // Start of your program
 5: int main()
 6: {
 7:    // Tell the compiler what namespace to search in
 8:    using namespace std;
 9:
10:    /* Write to the screen using std::cout */
11:    cout << "Hello World" << endl;
12:
13:    // Return a value to the OS
14:    return 0;
15: }
```

▼ 分析：

请注意第 8 行。通过告诉编译器您要使用名称空间 std，在第 11 行使用 cout 和 endl 时，就无需显式地指定名称空间了。

程序清单 2.3 是程序清单 2.2 的更严谨版本，它没有包含整个名称空间，而只包含要使用的元素。

程序清单 2.3 关键字 using 的另一种用法

```
 1: // Pre-processor directive
 2: #include <iostream>
 3:
 4: // Start of your program
 5: int main()
 6: {
 7:    using std::cout;
 8:    using std::endl;
 9:
10:    /* Write to the screen using cout */
11:    cout << "Hello World" << endl;
12:
13:    // Return a value to the OS
14:    return 0;
15: }
```

▼ **分析：**

在程序清单 2.3 中，使用第 7～8 行替换了程序清单 2.2 的第 8 行。两者的差别在于，前者让您能够在不显式指定名称空间限定符 std::的情况下使用名称空间 std 中的所有元素，而后者让您能够在不显式指定名称空间限定符 std::的情况下使用 std::cout 和 std::endl。

2.3 C++代码中的注释

在程序清单 2.3 中，第 1、4、10 和 13 行包含口语（这里为英语）文本，但不会影响程序编译，也不会影响程序的输出。这些行被称为注释。注释会被编译器忽略，程序员使用它们来对代码进行解释，因此使用人类能够明白的语言编写。

C++支持下面两种风格的注释。

- //指出当前行为注释，例如：

```
// This is a comment
```

- /*和*/表示它们之间的文本为注释，即便这些文本跨越多行：

```
/* This is a comment
and it spans two lines */
```

注意

程序员为何要对自己编写的代码进行解释呢？这看起来好像有点奇怪，但程序越大，合作开发同一个模块的程序员越多，编写易于理解的代码就越重要。必须使用清晰的注释对代码的功能以及为何要这样做进行解释，这很重要。

应该	不应该
务必添加注释，对程序中复杂算法和复杂部分的工作原理进行解释。 务必以其他程序员能够理解的方式编写注释。	不要使用注释来解释显而易见的代码。 别忘了，不要因为可以添加注释，就编写晦涩难懂的代码。 别忘了，修改代码时，可能需要相应地更新注释。

2.4 C++函数

C++函数与 C 语言函数相同。函数让您能够将应用程序划分成多个功能单元，并按您选择的顺序调用。函数被调用时，通常将一个值返回给调用它的函数。最著名的函数无疑是 main()，它被编译器视为 C++应用程序的起点，必须返回一个整数。

程序员通常需要编写自己的函数。程序清单 2.4 是一个简单的应用程序，它使用一个函数在屏幕上显示内容，而该函数使用各种参数调用了 std::cout。

程序清单 2.4 声明、定义和调用函数，该函数演示了 std::cout 的一些功能

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: // Function declaration
 5: int DemoConsoleOutput();
 6:
 7: int main()
 8: {
 9:     // Call i.e. invoke the function
10:     DemoConsoleOutput();
```

```
11:
12:     return 0;
13: }
14:
15: // Function definition
16: int DemoConsoleOutput()
17: {
18:     cout << "This is a simple string literal" << endl;
19:     cout << "Writing number five: " << 5 << endl;
20:     cout << "Performing division 10 / 5 = " << 10 / 5 << endl;
21:     cout << "Pi when approximated is 22 / 7 = " << 22 / 7 << endl;
22:     cout << "Pi more accurately is 22 / 7 = " << 22.0 / 7 << endl;
23:
24:     return 0;
25: }
```

▼ 输出：

```
This is a simple string literal
Writing number five: 5
Performing division 10 / 5 = 2
Pi when approximated is 22 / 7 = 3
Pi more accurately is 22 / 7 = 3.14286
```

▼ 分析：

这里需要注意的是第 5、10 和 15～25 行。第 5 行为函数声明，它告诉编译器您要创建一个函数，该函数名为 DemoConsoleOutput()，返回类型为 int。正是因为该声明，编译器才会编译第 10 行，并假定后面会有函数定义（即函数的实现），这里是第 15～25 行。

这个函数演示了 cout 的各种功能。它不仅能够像前面显示 Hello World 那样显示文本，还能显示简单算术运算的结果。第 21 和 22 行都试图显示 pi 的值（22/7），但后者更精确，因为 22.0/7 让编译器将结果视为实数（在 C++中为 float），而不是整数。

注意到该函数被声明为返回一个整数（这里返回的是 0）。由于它不做任何决策，因此没必要返回其他值。同样，main()也返回 0。鉴于 main()将其所有任务都交给了函数 DemoConsoleOutput()去完成，更明智的做法是，在 main()中返回该函数的返回值，如程序清单 2.5 所示。

程序清单 2.5　使用函数的返回值

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: // Function declaration and definition
 5: int DemoConsoleOutput()
 6: {
 7:     cout << "This is a simple string literal" << endl;
 8:     cout << "Writing number five: " << 5 << endl;
 9:     cout << "Performing division 10 / 5 = " << 10 / 5 << endl;
10:     cout << "Pi when approximated is 22 / 7 = " << 22 / 7 << endl;
11:     cout << "Pi more accurately is 22 / 7 = " << 22.0 / 7 << endl;
12:
13:     return 0;
14: }
15:
16: int main()
17: {
18:     // Function call with return used to exit
19:     return DemoConsoleOutput();
20: }
```

▼ **分析：**

该应用程序的输出与前一个程序清单相同，但编写方式存在细微的差别。首先，由于在 main()前面（第 5 行）定义了该函数，因此无需声明该函数。较新的 C++编译器将其视为函数声明和定义。另外，main()也更简短。第 19 行调用函数 DemoConsoleOutput()，并将该函数的返回值作为应用程序的返回值。

注意

在函数无需做任何决策，也无需返回成功/失败状态时，可将其返回类型声明为 void:

```
void DemoConsoleOutput()
```

这个函数不能返回值。对于返回类型为 void 的函数，不能通过执行它来做决策。

函数可以接受参数，可以递归，可以包含多条返回语句，可以重载，还可声明为内联的，在这种情况下，编译器将展开函数调用。这些概念都将在第 7 章更详细地介绍。

2.5 使用 std::cin 和 std::cout 执行基本输入输出操作

计算机让用户能够以各种方式与其运行的应用程序交互，还让应用程序能够以众多方式与用户交互。用户通过键盘和鼠标与应用程序交互。您可以在屏幕上以文本和复杂图形的方式显示信息，使用打印机将信息打印到纸上，还可将信息存储到文件系统中，供以后使用。本节讨论最简单的 C++输入输出方式：使用控制台读写信息。

要将简单的文本数据写入到控制台，可使用 std::cout（读作 standard see-out）；要从控制台读取文本和数字，可使用 std::cin（读作 standard see-in）。事实上，在程序清单 2.1 中，在屏幕上显示 Hello World 时，您就使用过 cout：

```
8:    std::cout << "Hello World" << std::endl;
```

在这条语句中，cout 的后面依次为插入运算符<<（帮助将数据插入输出流）、要插入的字符串字面量 Hello World、使用 std::endl（读作 standard end-line）表示的换行符。

cin 的用法也很简单，但它用于输入，因此需要指定要将输入数据存储到其中的变量：

```
std::cin >> Variable;
```

因此，cin 后面依次为提取运算符>>（从输入流中提取数据）以及要将数据存储到其中的变量。如果需要将用户输入存储到两个变量中——每个变量包含用空格分隔的数据，可使用一条语句完成这项任务：

```
std::cin >> Variable1 >> Variable2;
```

请注意，cin 可用于从用户那里获取文本输入和数字输入，如程序清单 2.6 所示。

程序清单 2.6 使用 cin 和 cout 显示用户的数字输入和文本输入

```
 1: #include <iostream>
 2: #include <string>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    // Declare a variable to store an integer
 8:    int InputNumber;
 9:
10:    cout << "Enter an integer: ";
11:
12:    // store integer given user input
13:    cin >> InputNumber;
14:
```

```
15:    // The same with text i.e. string data
16:    cout << "Enter your name: ";
17:    string InputName;
18:    cin >> InputName;
19:
20:    cout << InputName << " entered " << InputNumber << endl;
21:
22:    return 0;
23: }
```

▼ **输出：**

```
Enter an integer: 2011
Enter your name: Siddhartha
Siddhartha entered 2011
```

▼ **分析：**

第 8 行声明了一个名为 InputNumber 的变量，用于存储类型为 int 的数据。第 10 行使用 cout 让用户输入一个数字，而第 13 行使用 cin 将输入的数字存储在该 int 变量中。接下来重复相同的操作，存储用户的名字，当然这不能存储在 int 变量中，而是存储在 string 变量中，如第 17～18 行所示。第 2 行包含了<string>，原因是后面在 main()中使用了类型 string。最后，第 20 行使用一条 cout 语句显示输入的名字和数字以及连接文本，输出为 Siddhartha entered 2011。

这是一个非常简单的示例，演示了 C++中输入输出的基本原理。如果您不清楚变量的概念，不用担心，第 3 章将详细阐述。

2.6 总结

本章介绍了简单 C++程序的基本组成部分。您明白了 main()是什么，了解了名称空间，学习了控制台输入输出的基本知识。现在，您能够在自己编写的任何程序中使用这些知识了。

2.7 问与答

问：#include 的作用是什么？

答：这是一个预处理器编译指令。预处理器在您调用编译器时运行。该指令使得预处理器将#include 后面的<>中的文件读入程序，其效果如同将这个文件输入到源代码中的这个位置。

问：//注释和/*注释之间有何不同？

答：//注释到行尾结束；/*注释到*/结束。//注释也被称为单行注释，/*注释通常被称为多行注释。请记住，即使是函数的结尾也不能作为/*注释的结尾，必须加上注释结尾标记*/，否则将出现编译错误。

问：什么情况下需要命令行参数？

答：需要让用户修改程序的行为时。例如，Linux 命令 ls 和 Windows 命令 dir 都显示当前目录（文件夹）的内容，要查看另一个目录中的文件，需要使用命令行参数指定相应的路径，如 ls/或 dir\。

2.8 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。

请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

2.8.1 测验

1. 声明 Int main()有何问题？
2. 注释可以超过一行吗？

2.8.2 练习

1. 查错：输入下面的程序并编译它。它为什么不能通过编译？如何修复？

```
1: #include <iostream>
2: void main()
3: {
4:       std::Cout << Is there a bug here?";
5: }
```

2. 修复练习 1 中的错误，然后重新编译、链接并运行它。
3. 修改程序清单 2.4，以演示减法（使用-）和乘法（使用*）。

第3章
使用变量和常量

变量让程序员能够将数据临时存储一段时间，而常量让程序员能够定义不允许修改的东西。
在本章中，您将学习：

- 如何使用 C++11 关键字 auto 和 constexpr；
- 如何声明和定义变量与常量；
- 如何给变量赋值以及操纵这些值；
- 如何将变量的值显示到屏幕上。

3.1 什么是变量

在探索编程语言为何需要使用变量前，先来看看计算机的组成及其工作原理。

3.1.1 内存和寻址概述

所有计算机、智能手机及其他可编程设备都包含微处理器和一定数量的临时存储空间，这种临时存储器被称为随机存取存储器（RAM）。另外，很多设备还让您能够将数据永久性地存储到硬盘等存储设备中。微处理器负责执行应用程序，在此过程中，它从 RAM 中获取要执行的应用程序以及相关联的数据，这包括显示到屏幕上的数据以及用户输入的数据。

RAM 类似于宿舍里成排存物柜的存储区域，每个存物柜都有编号，即地址。要访问特定的内存单元，如内存单元 578，需要使用指令要求处理器从这里获取值或将值写入到这里。

3.1.2 声明变量以访问和使用内存

下面的示例将帮助您明白变量是什么。假设您要编写一个程序，它将用户提供的两个数字相乘。用户被要求依次提供被乘数和乘数，而您需要存储它们，以便以后将它们相乘。您还可能需要存储乘法运算的结果，供以后使用，这取决于您要使用这个结果做什么。如果显式地指定用于存储这些数字的内存单元的地址（如 578），既慢又容易出错，因为这需要避免不小心覆盖原有的数据，以后还需避免覆盖您存储的数据。

使用 C++等语言编程时，您只需定义用于存储这些值的变量。定义变量非常简单，其语法如下：

```
variable_type variable_name;
```

或

```
variable_type variable_name = initial_value;
```

变量类型向编译器指出了变量可存储的数据的性质，编译器将为变量预留必要的空间。变量名由程序员选择，它替代了变量值在内存中的存储地址，但更友好。除非给变量赋初值，否则无法确保相应内存单元的内容是什么，这对程序可能不利。因此，初始化虽然是可选的，但对变量初始化通常是一个不错的编程习惯。程序清单 3.1 将用户提供的两个数字相乘，演示了如何在程序中声明、初始化和使用变量。

程序清单 3.1 使用变量存储数字及其相乘的结果

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: int main ()
 5: {
 6:    cout << "This program will help you multiply two numbers" << endl;
 7:
 8:    cout << "Enter the first number: ";
 9:    int FirstNumber = 0;
10:    cin >> FirstNumber;
11:
12:    cout << "Enter the second number: ";
13:    int SecondNumber = 0;
14:    cin >> SecondNumber;
15:
16:    // Multiply two numbers, store result in a variable
17:    int MultiplicationResult = FirstNumber * SecondNumber;
18:
19:    // Display result
20:    cout << FirstNumber << " x " << SecondNumber;
21:    cout << " = " << MultiplicationResult << endl;
22:
23:    return 0;
24: }
```

▼ 输出：

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 24
51 x 24 = 1224
```

▼ 分析：

这个应用程序要求用户输入两个数字，将它们相乘并显示结果。应用程序要使用用户输入的数字，必须将其存储到内存中。第 9 和 13 行声明了变量 FirstNumber 和 SecondNumber，用于临时存储用户输入的整数。第 10 和 14 行使用 std::cin 获取用户输入，并将其存储到两个整型变量中。第 21 行的 cout 语句用于将结果显示到控制台。

下面进一步分析其中的一个变量声明：

```
9:    int FirstNumber = 0;
```

这行代码声明了一个变量，其类型为 int（表示整型），名称为 FirstNumber，并将该变量的初始值设置为零。

使用汇编语言编程时，需要显式地要求处理器将被乘数存储到特定的位置，如 578，而 C++让您能够使用更友好的概念（如变量 FirstNumber）来访问内存单元，以检索和存储数据。将变量 FirstNumber 关联到内存单元的工作由编译器负责，它还负责为您完成相关的簿记工作（book Keeping）。

这样，程序员就可使用对人类友好的名称，把将变量关联到地址以及创建 RAM 访问指令的工作留给编译器去做。

警告

为编写易于理解和维护的代码，给变量指定合适的名称很重要。
变量名可包含数字和字母，但不能以数字打头。变量名不能包含空格和算术运算符（+、-等）。要拉长变量名，可使用下划线。
另外，变量名不能是保留的关键字，例如，将变量命名为 return 将导致程序无法通过编译。

3.1.3 声明并初始化多个类型相同的变量

在程序清单 3.1 中，变量 FirstNumber、SecondNumber 和 MultiplicationResult 属于同一种类型——都是整型，但在三行中分别声明。如果您愿意，可简化这三个变量的声明，在一行代码中完成，如下所示：

```
int FirstNumber = 0, SecondNumber = 0, MultiplicationResult = 0;
```

注意

正如您看到的，在 C++中，可同时声明多个类型相同的变量，还可在函数开头声明变量。然而，需要时再声明变量通常是更好的选择，因为这让代码更容易理解——变量的声明离使用它的地方不远时，别人更容易了解变量的类型。

警告

存储在变量中的数据被存储在内存中。计算机关闭或应用程序终止时，这样的数据将丢失，除非程序员显式地将其存储到硬盘等永久性存储介质中。
将数据存储到磁盘文件将在第 27 章讨论。

3.1.4 理解变量的作用域

常规变量的作用域很明确，只能在作用域域内使用它们，如果您在作用域外使用它们，编译器将无法识别，导致程序无法通过编译。在作用域外面，变量是未定义的实体，编译器对其一无所知。

为让读者更好地理解变量的作用域，程序清单 3.2 将程序清单 3.1 所示的程序重新组织成了一个函数——MultiplyNumbers()，它将两个数字相乘并返回结果。

程序清单 3.2 使用变量存储数字及其相乘的结果

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

void MultiplyNumbers ()
{
    cout << "Enter the first number: ";
    int FirstNumber = 0;
    cin >> FirstNumber;

    cout << "Enter the second number: ";
    int SecondNumber = 0;
    cin >> SecondNumber;

    // Multiply two numbers, store result in a variable
    int MultiplicationResult = FirstNumber * SecondNumber;

    // Display result
    cout << FirstNumber << " x " << SecondNumber;
    cout << " = " << MultiplicationResult << endl;
}
int main ()
{
    cout << "This program will help you multiply two numbers" << endl;
```

```
24:
25:     // Call the function that does all the work
26:     MultiplyNumbers();
27:
28:     // cout << FirstNumber << " x " << SecondNumber;
29:     // cout << " = " << MultiplicationResult << endl;
30:
31:     return 0;
32: }
```

▼ **输出：**

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 24
51 x 24 = 1224
```

▼ **分析：**

程序清单 3.2 的功能与程序清单 3.1 相同，输出也相同。唯一的差别在于，将工作交给了函数 MultiplyNumbers()去完成，并在 main()中调用它。请注意，不能在函数 MultiplyNumbers()外面使用变量 FirstNumber 和 SecondNumber。如果您取消对 main()中第 28 或 29 行的注释，将出现编译错误，而错误很可能是标识符未声明（undeclared identifier）。

这是因为变量 FirstNumber 和 SecondNumber 的作用域为局部，被限定在声明它的函数内，这里为 MultiplyNumbers()。局部变量只能在这样的范围内使用，即从声明它的语句开始到当前函数的末尾。标识函数结束的花括号（}）也限定了函数内部声明的变量的作用域。函数结束后，将销毁所有局部变量，并归还它们占用的内存。

编译时，在 MultiplyNumbers()内部声明的变量在该函数结束时不再存在，如果在 main()中使用它们，程序将无法通过编译，因为在 main()中这些变量未声明。

警告

> 如果您在 main()声明另一组同名变量，就不能指望它们的值与您在 MultiplyNumbers()中赋给同名变量的值相同。
>
> 编译器将 main()中声明的变量视为独立的实体，即便它们与另一个函数中声明的变量同名，因为这些变量的作用域不同。

3.1.5 全局变量

在程序清单 3.2 中，如果变量是在函数 MultiplyNumbers()外部而不是内部声明的，则在函数 main()和 MultiplyNumbers()中都可使用它们。程序清单 3.3 演示了全局变量，它们是程序中作用域最大的变量。

程序清单 3.3 使用全局变量

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: // three global integers
 5: int FirstNumber = 0;
 6: int SecondNumber = 0;
 7: int MultiplicationResult = 0;
 8:
 9: void MultiplyNumbers ()
10: {
11:     cout << "Enter the first number: ";
12:     cin >> FirstNumber;
```

```
13:
14:     cout << "Enter the second number: ";
15:     cin >> SecondNumber;
16:
17:     // Multiply two numbers, store result in a variable
18:     MultiplicationResult = FirstNumber * SecondNumber;
19:
20:     // Display result
21:     cout << "Displaying from MultiplyNumbers(): ";
22:     cout << FirstNumber << " x " << SecondNumber;
23:     cout << " = " << MultiplicationResult << endl;
24: }
25: int main ()
26: {
27:     cout << "This program will help you multiply two numbers" << endl;
28:
29:     // Call the function that does all the work
30:     MultiplyNumbers();
31:
32:     cout << "Displaying from main(): ";
33:
34:     // This line will now compile and work!
35:     cout << FirstNumber << " x " << SecondNumber;
36:     cout << " = " << MultiplicationResult << endl;
37:
38:     return 0;
39: }
```

▼ **输出：**

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 19
Displaying from MultiplyNumbers(): 51 x 19 = 969
Displaying from main(): 51 x 19 = 969
```

▼ **分析：**

程序清单 3.3 在两个函数中显示了乘法运算的结果，而变量 FirstNumber、SecondNumber 和 MultiplicationResult 都不是在这两个函数内部声明的。这些变量为全局变量，因为声明它们的第 5～7 行不在任何函数内部。注意到第 23 和 36 行使用了这些变量并显示它们的值。尤其要注意的是，虽然 MultiplicationResult 的值是在 MultiplyNumbers()中指定的，但仍可在 main()中使用它。

警告

不分青红皂白地使用全局变量通常是一种糟糕的编程习惯。

全局变量可在任何函数中赋值，因此其值可能出乎意料，在不同函数模块由小组中的不同程序员编写时尤其如此。

要像程序清单 3.3 那样在 main()中获取乘法运算的结果，一种更妥善的方式是，让 MultiplyNumbers()将结果返回给 main()。

3.2 编译器支持的常见 C++变量类型

在本书前面的大多数示例中，定义的变量类型都是 int（整型），然而 C++编译器支持很多基本变量类型，可供程序员选择。选择正确的变量类型犹如根据要做的工作选择正确的工具一样重要！十字螺钉与普通螺帽不匹，同样，无符号整型变量也不能用于存储负值！表 3.1 列出了各种变量类型及其可存储的数据的特征。要编写高效而可靠的 C++程序，这些信息非常重要。

表 3.1 变量类型

类型	值	类型	值
bool	true/false	int（16 位）	–32768～32767
char	256 个字符值	int（32 位）	–2147483648～2147483647
unsigned short int	0～65535	unsigned int（16 位）	0～65535
short int	–32768～32767	unsigned int（32 位）	0～4294967295
unsigned long int	0～4294967295	float	1.2e–38～3.4e38
long int	–2147483648～2147483647	double	2.2e–308～1.8e308

接下来的几小节将更详细地介绍一些重要的类型。

3.2.1 使用 bool 变量存储布尔值

C++提供了一种专为存储布尔值 true 和 false 而创建的类型，其中 true 和 false 都是保留的 C++关键字。对于取值为 ON 或 OFF、有或没有、可用或不可用等设置和标记，非常适合使用这种类型的变量来存储。

下面是一个声明并初始化布尔变量的例子：

```
bool AlwaysOnTop = false;
```

下面是一个结果为布尔值的表达式：

```
bool DeleteFile = (UserSelection == "yes");
// evaluates to true only if UserSelection contains "yes", else to false
```

3.2.2 使用 char 变量存储字符

char 变量用于存储单个字符，下面是一个声明示例：

```
char UserInput = 'Y'; // initialized char to 'Y'
```

请注意，表示内存空间容量大小的单位是位和字节。位的取值为 0 或 1，而字节可以包含字符的数字表示。因此像前面的示例那样使用字符数据时，编译器将把字符转换为可存储到内存中的数字表示。美国信息交换标准（ASCII）对拉丁字符 A～Z、a～z、数字 0～9、一些特殊键击（如 DEL）和一些特殊字符（如空格）的数字表示进行了标准化。

如果您查看附录 E 的 ASCII 码表，赋给变量 UserInput 的字符 Y 的 ASCII 码为 89，因此编译器将在分配给 UserInput 的内存空间中存储 89。

3.2.3 有符号整数和无符号整数的概念

符号表示正或负。您在计算机中使用的所有数字都以位和字节的方式存储在内存中。1 字节的内存单元包含 8 位，每位都要么为 0，要么为 1（即存储这两个值之一），因此 1 字节的内存单元可以有 2^8（即 256）个不同的取值。同样，16 位的内存单元可以有 2^{16}（65536）个不同的取值。

如果这些取值是无符号的（即为正数），则 1 个字节的可能取值为 0～255，而 2 个字节的可能取值为 0～65535。从表 3.1 可知，类型 unsigned short 的取值范围为 0～65535，因为它占用 16 位内存。因此，使用位和字节表示正值非常容易，如图 3.1 所示。

在这种空间中如何表示负数呢？一种方式是将 1 位用作符号位，指出其他位包含的值是正还是负，如图 3.2 所示。符号位必须是最高有效位（most-sigificant-bit，MSB），因为最低有效位（least-significant-bit）需要用于表示小于 2 的数字。当 MSB 包含符号信息时，假定 0 表示正，1 表示负，而其他位包含绝对值。

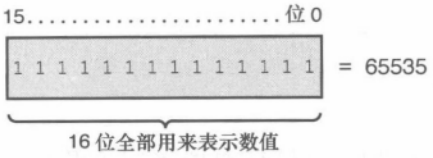


图 3.1 占用 16 位内存的 unsigned short 变量

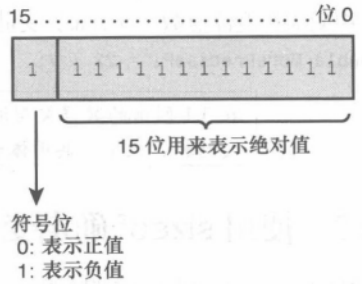


图 3.2 占用 16 位内存的 signed short 变量

因此，占用 8 位的有符号数的取值范围为-127～127，而占用 16 位的有符号数的取值范围为-32768～32768。如果查看表 3.1，将发现类型 short 的取值范围为-32768～32768。

3.2.4 有符号整型 short、int、long 和 long long

这些类型的长度各不相同，因此取值范围也各不相同。int 可能是使用得最多的类型，在大多数编译器中，其长度都是 32 位。应根据变量可能存储的最大值给它指定合适的类型。

声明有符号类型的变量非常简单，如下所示：

```
short int SmallNumber = -100;
int LargerNumber = -70000;
long PossiblyLargerThanInt = -70000; //on some platforms, long is an int
long long LargerThanInt = -70000000000;
```

3.2.5 无符号整型 unsigned short、unsigned int、unsigned long 和 unsigned long long

不同于相应的有符号类型，无符号整型变量不能包含符号信息，因此，它们的最大取值为相应有符号类型的两倍。

声明无符号类型变量也很简单，如下所示：

```
unsigned short int SmallNumber = 255;
unsigned int LargerNumber = 70000;
// on some platforms, long is int
unsigned long PossiblyLargerThanInt = 70000;
unsigned long long LargerThanInt = 70000000000;
```

注意 如果预期变量的取值不会为负数，就应将其类型声明为无符号的。因此，如果您要存储苹果的数量，不要使用 int 变量，而应使用 unsigned int 变量，后者的最大取值为前者的两倍。

警告 对于银行应用程序中用于存储账户余额的变量，将其类型声明为无符号的就可能不合适。

3.2.6 浮点类型 float 和 double

您可能在学校学过，浮点数就是实数，可以是正，也可以是负，还可以包含小数值。因此，如果要使用 C++变量存储 pi（22/7）的值，就应将其声明为浮点类型。

声明浮点类型变量的方式与程序清单 3.1 中声明 int 变量的方式相同。要声明一个可存储小数值的 float 变量，可像下面这样做：

```
float Pi = 3.14;
```

要声明双精度浮点数（double）变量，可像下面这样做：

```
double MorePrecisePi = 22 / 7;
```

注意 表 3.1 列出的数据类型通常被称为 POD（Plain Old Data）。POD 还包含聚合数据类型（结构、枚举、共用体和类）。

3.3 使用 sizeof 确定变量的长度

变量长度指的是：程序员声明变量时，编译器将预留多少内存，用于存储赋给该变量的数据。变量的长度随类型而异，C++提供了一个方便的运算符——sizeof，可用于确定变量的长度（单位为字节）或类型。

sizeof 的用法非常简单。要确定 int 变量的长度，可调用 sizeof 并给它传递参数 int：

程序清单 3.4 演示了如何获悉各种标准 C++变量类型的长度。

```
cout << "Size of an int: " << sizeof (int);
```

程序清单 3.4 获悉标准 C++变量类型的长度

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:     using namespace std;
 6:     cout << "Computing the size of some C++ inbuilt variable types" << endl;
 7:
 8:     cout << "Size of bool: " << sizeof(bool) << endl;
 9:     cout << "Size of char: " << sizeof(char) << endl;
10:     cout << "Size of unsigned short int: " << sizeof(unsigned short) << endl;
11:     cout << "Size of short int: " << sizeof(short) << endl;
12:     cout << "Size of unsigned long int: " << sizeof(unsigned long) << endl;
13:     cout << "Size of long: " << sizeof(long) << endl;
14:     cout << "Size of int: " << sizeof(int) << endl;
15:     cout << "Size of unsigned long long: "<< sizeof(unsigned long long)<<
endl;
16:     cout << "Size of long long: " << sizeof(long long) << endl;
17:     cout << "Size of unsigned int: " << sizeof(unsigned int) << endl;
18:     cout << "Size of float: " << sizeof(float) << endl;
19:     cout << "Size of double: " << sizeof(double) << endl;
20:
21:     cout << "The output changes with compiler, hardware and OS" << endl;
22:
23:     return 0;
24: }
```

▼ 输出：

```
Computing the size of some C++ inbuilt variable types
Size of bool: 1
Size of char: 1
Size of unsigned short int: 2
Size of short int: 2
Size of unsigned long int: 4
Size of long: 4
Size of int: 4
Size of unsigned long long: 8
Size of long long: 8
Size of unsigned int: 4
Size of float: 4
Size of double: 8
The output changes with compiler, hardware and OS
```

▼ 分析：

程序清单 3.4 的输出指出了各种类型的长度（单位为字节），这是针对我使用的平台（编译器、操作系统和硬件）而言的。具体地说，这是在 64 位系统中以 32 位模式（使用 32 位编译器进行编译）运行该程序得到的结果。如果使用 64 位编译器进行编译，结果可能不同。我之所以使用 32 位编译器，是因为这样该应用程序在 32 位和 64 位系统上都能运行。输出表明，无符号类型和相应的有符号类型的长度相同，唯一的差别在于，后者的 MSB 包含符号信息。

注意

输出中的长度单位为字节。给对象分配内存时，其长度将是一个重要参数，在程序员动态地给对象分配内存时尤其如此。

C++11

使用 auto——编译器的类型推断功能

在有些情况下，根据赋给变量的初值，很容易知道其类型。例如，如果将变量的初值设置成了 true，就可推断其类型为 bool。在 C++11 中，可不显式地指定变量的类型，而使用关键字 auto：

```
auto Flag = true;
```

这将指定变量 Flag 的类型的任务留给了编译器。编译器检查赋给变量的初值的性质，再确定将变量声明为什么类型最合适。就这里而言，显然初始值 true 最适合存储到类型为 bool 的变量中。因此，编译器认为变量 Flag 的最佳类型为 bool，并在内部将 Flag 的类型视为 bool，程序清单 3.5 证明了这一点。

程序清单 3.5 使用关键字 auto 依靠编译器的类型推断功能

```
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     auto Flag = true;
7:     auto Number = 2500000000000;
8:
9:     cout << "Flag = " << Flag;
10:    cout << " , sizeof(Flag) = " << sizeof(Flag) << endl;
11:    cout << "Number = " << Number;
12:    cout << " , sizeof(Number) = " << sizeof(Number) << endl;
13:
14:    return 0;
15: }
```

▼ 输出：

```
Flag = 1 , sizeof(Flag) = 1
Number = 2500000000000 , sizeof(Number) = 8
```

▼ 分析：

在第 6 和 7 行声明变量 Flag 和 Number 时，没有将其类型分别指定为 bool 和 long long，而使用了关键字 auto。这让编译器去决定变量的类型，而编译器将根据初始值来确定合适的类型。接下来，使用 sideof 来检查编译器选择的类型是否符合预期，从程序清单 3.5 的输出可知，确实符合预期。

注意

使用 auto 时必须对变量进行初始化，因为编译器需要根据初始值来确定变量的类型。如果将变量的类型声明为 auto，却不对其进行初始化，将出现编译错误。

乍一看，auto 并非什么了不起的功能，但在变量类型非常复杂时，它可让编程容易得多。假设您使用 std::vector 声明了一个名为 MyNumbers 的动态整数数组：

```
std::vector<int> MyNumbers;
```

要访问或遍历该数组中的元素并显示它们，可使用如下代码：

```
for ( vector<int>::const_iterator Iterator = MyNumbers.begin();
      Iterator < MyNumbers.end();
      ++Iterator )
   cout << *Iterator << " ";
```

std::vector 和 for 循环都还没有介绍，因此即便上述代码看起来像天书，也不用担心。其功能如下：对于矢量中的每个元素——从 begin()开始，到 end()前面的一个元素结束，都使用 cout 显示其值。第 1 行代码非常复杂，它声明变量 Iterator，并将其初始值设置为 begin()返回的值。这个变量的类型为 vector<int>::const_iterator，这对程序员来说，学习和书写起来都非常复杂。程序员无需熟记这一点，而可依赖于 begin()的返回类型，将上述 for 循环简化为如下所示：

```
for( auto Iterator = MyNumbers.begin();
     Iterator < MyNumbers.end();
     ++Iterator )
   cout << *Iterator << " ";
```

注意到现在第 1 行有多紧凑。编译器检查 Iterator 的初始值——begin()返回的值，并将该变量的类型设置为该返回值的类型。这简化了 C++编码工作，在大量使用模板时尤其如此。

3.4 使用 typedef 替换变量类型

C++允许您将变量类型替换为您认为方便的名称，为此可使用关键字 typedef。在下面的示例中，程序员想给 unsigned int 指定一个更具描述性的名称——STRICTLY_POSITIVE_INTEGER：

```
typedef unsigned int STRICTLY_POSITIVE_INTEGER;
STRICTLY_POSITIVE_INTEGER PosNumber = 4532;
```

编译时，第 1 行告诉编译器，STRICTLY_POSITIVE_INTEGER 就是 unsigned int。以后编译器再遇到已定义的类型 STRICTLY_POSITIVE_INTEGER 时，就会将它替换为 unsigned int 并继续编译。

注意 涉及语法烦琐的复杂类型，如使用模板的类型时，typedef（类型替换）特别方便。

3.5 什么是常量

假设您要编写一个程序，计算圆的面积和周长，其公式如下：

```
Area = Pi * Radius * Radius;
Circumference = 2 * Pi * Radius of circle
```

在这些公式中，Pi 为常量，其值为 22/7。您希望在整个程序中，Pi 的值都不变；您也不希望无意间将错误的值赋给 Pi，如错误地复制/粘贴或查找/替换。C++让您能够将 Pi 定义为声明后就不能修改的常量，换句话说，定义常量后，就不能修改它的值。在 C++中，给常量赋值会导致编译错误。

因此，在 C++中，常量类似于变量，只是不能修改。与变量一样，常量也占用内存空间，并使用名称标识为其预留的空间的地址，但不能覆盖该空间的内容。在 C++中，常量可以是：

- 字面常量；
- 使用关键字 const 声明的常量；
- 使用关键字 constexpr 声明的常量表达式（C++11 新增的）；
- 使用关键字 enum 声明的枚举常量；
- 使用#define 定义的常量（已摒弃，不推荐）。

3.5.1 字面常量

再来看一下程序清单 3.1——将两个数相乘的简单程序。其中声明了一个名为 FirstNumber 的 int 变量：

```
9:    int FirstNumber = 0;
```

将这个 int 变量的初始值设置成了零。这个零是代码的一部分，将编译到应用程序中，且不可修改，因此称为字面常量。字面常量可以是任何类型：布尔型、整型、字符串等。在您编写的第一个 C++ 程序（程序清单 1.1）中，您使用了如下代码显示 Hello World：

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

Hello World 就是一个字符串字面常量。

3.5.2 使用 const 将变量声明为常量

从实用和编程的角度看，最重要的 C++常量类型是在变量类型前使用关键字 const 声明的。通用的声明方式类似于下面这样：

```
const type-name constant-name;
```

来看一个简单的应用程序，它显示常量 Pi 的值，如程序清单 3.6 所示。

程序清单 3.6 声明一个名为 Pi 的常量

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:    using namespace std;
 6:
 7:    const double Pi = 22.0 / 7;
 8:    cout << "The value of constant Pi is: " << Pi << endl;
 9:
10:    // Uncomment next line to fail compilation
11:    // Pi = 345;
12:
13:    return 0;
14: }
```

▼ 输出：

```
The value of constant Pi is: 3.14286
```

▼ 分析：

请注意常量 Pi 的声明（第 7 行）。这里使用了关键字 const 来告诉编译器，Pi 是一个类型为 double 的常量。第 11 行试图给一个常量赋值，如果取消对该行的注释，将出现编译错误，指出不能给常量赋值。因此，常量是一种确保某些数据不能修改的强大方式。

注意

> 如果变量的值不应改变，就应将其声明为常量，这是一种良好的编程习惯。通过使用关键字 const，程序员可确保数据不变，避免应用程序无意间修改该常量。
> 在多位程序员合作开发时，这特别有用。

声明在编译期间长度固定的静态数组时，常量很有用。程序清单 4.2 提供了一个示例，演示了如何使用 int 常量指定数组长度。

C++11

3.5.3 使用 constexpr 声明常量

在 C++11 之前，C++就支持常量表达式的概念，只是没有关键字 constexpr。在程序清单 3.5 中，22.0/7 是一个常量表达式，C++11 之前的编译器也支持它。然而，C++11 之前的编译器不允许定义在编译阶段计算的函数。在 C++11 中，可以编写下面这样的代码：

```
constexpr double GetPi() {return 22.0 / 7;}
```

还可将 GetPi()与另一个常量一起使用，如下所示：

```
constexpr double TwicePi() {return 2 * GetPi();}
```

乍一看，const 和 constexpr 之间的差别很小，但从编译器和应用程序的角度看，关键字 Constexpy 提供了优化应用程序的可能性。对于第二条语句，如果使用 const，将在运行阶段执行计算，但使用遵守 C++11 的编译器时，将在编译阶段计算该表达式的值，这提高了应用程序的运行速度。

注意 编写本书时，Microsoft Visual C++学习版还不支持关键字 constexpr，但 GNU 的 g++编译器支持。

3.5.4 枚举常量

在有些情况下，变量只能有一组特定的取值。例如，彩虹不能包含青绿色，指南针的方位不能为“左”。在这些情况下，需要定义这样一种变量，即其可能取值由您指定。为此，可使用关键字 enum 来声明枚举常量。

例如，下面的枚举常量包含彩虹的颜色：

```
enum RainbowColors
{
   Violet = 0,
   Indigo,
   Blue,
   Green,
   Yellow,
   Orange,
   Red
};
```

下面的枚举常量包含基本方位：

```
enum CardinalDirections
{
   North,
   South,
   East,
   West
};
```

可使用枚举常量来指定变量的类型，这样声明的变量只能取指定的值。因此，如果要声明一个变量，用于存储彩虹的颜色，可以像下面这样做：

```
RainbowColors MyWorldsColor = Blue; // Initial value
```

上述代码声明了常量 MyWorldColor，其类型为 RainbowColors。这个枚举常量只能取 RainbowColors 中指定的值，而不能取其他值。

注意 声明枚举常量时，编译器将把枚举值（Voilet 等）转换为整数，每个枚举值都比前一个大 1。您可以指定起始值，如果没有指定，编译器认为起始值为 0，因此 North 的值为 0。如果愿意，还可通过初始化显式地给每个枚举量指定值。

程序清单 3.7 演示了如何使用枚举常量来存储 4 个基本方位，并对第一个方位进行了初始化。

程序清单 3.7 使用枚举值指示基本方位

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: enum CardinalDirections
 5: {
 6:    North = 25,
 7:    South,
 8:    East,
 9:    West
10: };
11:
12: int main()
13: {
14:     cout << "Displaying directions and their symbolic values" << endl;
15:     cout << "North: " << North << endl;
16:     cout << "South: " << South << endl;
17:     cout << "East: " << East << endl;
18:     cout << "West: " << West << endl;
19:
20:    CardinalDirections WindDirection = South;
21:    cout << "Variable WindDirection = " << WindDirection << endl;
22:
23:    return 0;
24: }
```

▼ 输出：

```
Displaying directions and their symbolic values
North: 25
South: 26
East: 27
West: 28
Variable WindDirection = 26
```

▼ 分析：

这里将 4 个基本方位定义为枚举常量，并将第一个常量（North）的值设置为 25（第 6 行），这自动将随后的常量分别设置为 26、27 和 28，如输出所示。第 20 行创建了一个类型为 CardinalDirections 的变量，并将其初始值设置为 South。第 21 行显示该变量时，编译器显示的是 South 对应的整数值——26。

提示

> 您可能该看看程序清单 6.4 和程序清单 6.5，它们使用 enum 列举一个星期的各天，并使用条件处理指出用户选择的那天是根据哪颗星星命名的。

3.5.5 使用#define 定义常量

首先也是最重要的是，编写新程序时，不要使用这种常量。这里介绍使用#define 定义常量，只是为了帮助您理解一些旧程序，它们使用下面的语法定义常量：

```
#define Pi 3.14286
```

这是一个预处理器宏，让预处理器将随后出现的所有 Pi 都替换为 3.14286。预处理器将进行文本替换，而不是智能替换。编译器既不知道也不关心常量的类型。

警告

> 使用#define 定义常量的做法已被摒弃，因此不应采用这种做法。

3.6 给变量和常量命名

给变量命名的方式有很多，还有很多不同的约定。有些程序员喜欢在变量名开头用几个字符指出变量的类型，例如：

```
bool bIsLampOn = false;
```

其中 b 是程序员添加的前缀，指出变量的类型为 bool。这种表示法称为匈牙利表示法，最初由微软发明并倡议。然而，C++是一种强类型安全语言，编译器根据类型定义而不是名称前缀来获悉变量的类型。因此，当前强烈推荐程序员不要采用匈牙利表示法。变量名必须易于理解，哪怕这会导致变量名更长些。在这个示例中，假定该布尔变量表示车前灯开关，下面的变量名将更好些：

```
bool IsHeadLampOn = false;
```

这两个变量名都比下面的变量名更好：

```
bool b = false;
```

应不惜一切代价避免使用非描述性变量名。

应该	不应该
务必给变量指定描述性名称，那怕这会导致变量名很长。 务必确保变量名阐述了变量的用途。 务必站在别人的角度想一想，如果他从未见过您编写的代码，能明白变量名的含义吗？ 务必了解团队是否遵循特定的命名约定，如果是，请遵循这些约定。	不要使用太短或只有一个字符的变量名。 不要在变量名中使用只有您自己明白的怪异缩写。 不要将保留的 C++关键字用作变量名，因为这将导致程序无法通过编译。

3.7 不能用作常量或变量名的关键字

有些单词被 C++保留，不能用作变量名。对 C++编译器来说，这些关键字有特殊含义。关键字包括 if、while、for、main 等。表 3.2 和附录 B 列出了 C++定义的关键字。您的编译器可能还保留了其他单词，有关完整的关键字列表，请参阅编译器手册。

表 3.2 C++关键字

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void

续表

delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	
另外，下列单词被保留			
And	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

3.8 总结

在本章中，您了解到内存用于临时存储变量和常量的值。您了解到变量的长度取决于其类型，并可使用运算符 sizeof 来确定。您学习了各种变量类型，如 bool、int 等，知道它们用于存储不同类型的数据。选择正确的变量类型至关重要，如果选择的类型太短，可能导致回绕（wrapping）错误或溢出。您学习了 C++11 新增的关键字 auto，它让编译器根据变量的初始值确定其类型。

您还学习了各种常量，其中最重要的是使用关键字 const 和 enum 定义的常量。

3.9 问与答

问：既然可以使用常规变量代替常量，为何还要定义常量？

答：通过声明常量（尤其是使用关键字 const 时），可告诉编译器，其值是固定的，不允许修改。这样，编译器将确保不给常量赋值，即便另一位程序员接手了您的工作，不小心试图覆盖常量的值。因此，在知道变量的值不应改变时，应将其声明为常量，这是一个不错的编程习惯，可提高应用程序的质量。

问：为何应给变量赋初值？

答：如果不初始化，就无法知道变量包含的初始值。在这种情况下，初始值将是给变量预留的内存单元的内容。下面的语句使得创建变量 MyFavoriteNumber 后，就将指定的初始值 0 写入到为该变量预留的内存单元：

```
int MyFavoriteNumber = 0;
```

有时候，需要根据变量的值（通常是核实它不为零）做条件处理，如果不对变量进行初始化，这样的逻辑将不可靠，因为未赋值或初始化的变量包含的内容是随机的。

问：C++为何提供变量类型 short int、int 和 long int？为何不始终使用取值范围最大的变量类型呢？

答：C++用于编写各种应用程序，其中很多运行在计算能力和内存资源都很有限的设备上。例如，老式手机的计算能力和内存都有限。在这种情况下，程序员通过选择合适的变量类型，可节省内存并提高速度。如果编写的是常规台式机或高端智能手机程序，选择不同整型带来的性能提升或内存节省将很小，有时甚至可以忽略不计。

问：为何不应频繁地使用全局变量？全局变量在应用程序的任何地方都可用，可避免在函数之间传递值，从而节省一些时间，这种说法对吗？

答：可在应用程序的任何地方读取全局变量的值以及给它赋值，这是个问题，因为在应用程序的

任何地方都可修改它们。假设您与其他几位程序员合作开发一个项目，并将变量声明为全局的。如果队友不小心在其代码中修改这些变量——即便是在另一个.CPP 文件中，都将影响代码的可靠性。因此，为确保代码的稳定性，不要为节省几秒钟甚至几分钟而不分青红皂白地使用全局变量。

问：在 C++中，可以声明无符号整型变量，其取值只能是零或正整数。如果一个 unsigned int 变量的值为零，将其减 1 的结果如何？

答：这将导致环绕。如果将值为零的 unsigned int 变量减 1，它将环绕到可存储的最大值！从表 3.1 可知，unsigned short 变量的取值范围为 0~65535。为演示这一点，下面的代码声明了一个 unsigned short 变量，将其初始化为 0，并减 1：

```
unsigned short MyShortInt = 0;  // Initial Value
MyShortInt = MyShortInt - 1; // Decrement by 1
std::cout << MyShortInt << std::endl; // Output: 65535!
```

这不是 unsigned short 的问题，而是使用它的方式有问题；在变量的取值可能为负时，就不应将其类型指定为 unsigned int、unsigned short 或 unsigned long。如果 MyShortInt 被用于指定动态分配的字节数，允许在它为零时减 1，将导致分配 64KB 的内存！更糟糕的是，如果在访问内存单元是将 MyShortInt 用作索引，很可能访问外部内存单元，进而导致应用程序崩溃！

3.10 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

3.10.1 测验

1. 有符号整型和无符号整型有何不同？
2. 为何不应使用#define 来声明常量？
3. 为何要对变量进行初始化？
4. 给定如下枚举类型，QUEEN 的值是多少？

```
enum YOURCARDS {ACE, JACK, QUEEN, KING};
```

5. 下述变量名有何问题？

```
int Integer = 0;
```

3.10.2 练习

1. 修改测验题 4 中的枚举类型 YOURCARDS，让 QUEEN 的值为 45。
2. 编写一个程序，证明 unsigned int 和 int 变量的长度相同，且它们都比 long 变量短。
3. 编写一个程序，让用户输入圆的半径，并计算其面积和周长。
4. 在练习 3 中，如果将面积和圆周存储在 int 变量中，输出将有何不同？
5. 查错：下面的语句有何错误？

```
auto Integer;
```

第 4 章
管理数组和字符串

在前几章中，您声明存储单个 int、char 或字符串的变量。然而，您可能想声明一组对象，如 20 个 int 变量或一组 Cat 对象。

在本章中，您将学习：

- 什么是数组以及如何声明和使用它们；
- 什么是字符串以及如何使用字符数组来表示字符串；
- std::string 简介。

4.1 什么是数组

array 的字典定义与数组的概念很接近。韦氏字典指出，array 是一组元素，它们形成一个整体，如一组太阳能电池板。

数组具有如下特点：

- 数组是一系列元素；
- 数组中所有元素的类型都相同；
- 这组元素形成一个完整的集合。

在 C++中，数组让您能够按顺序将一系列相同类型的数据存储到内存中。

4.1.1 为何需要数组

假设您要编写一个程序，它让用户输入 5 个整数并显示出来。为此，一种方式是声明 5 个独立的 int 变量，并使用它们来存储和显示值。声明类似于下面这样：

```
int FirstNumber = 0;
int SecondNumber = 0;
int ThirdNumber = 0;
int FourthNumber = 0;
int FifthNumber = 0;
```

采用这种方式时，如果用户希望这个程序存储并显示 500 个整数，您将需要声明 500 个 int 变量。只要有足够的耐心和时间，这还是可行的。然而，如果用户要求存储并显示 500000 个整数，您该怎么办呢？

您应采取正确而聪明的方式，声明一个包含 5 个 int 元素的数组，并将每个元素都初始化为零，如下所示：

```
int MyNumbers [5] = {0};
```

这样，当您被要求支持 500000 个整数时，便可以快速扩大数组，如下所示：

```
int ManyNumbers [500000] = {0};
```

要定义一个包含 5 个字符的数组，可以这样做：

```
char MyCharacters [5];
```

这样的数组被称为静态数组，因为在编译阶段，它们包含的元素数以及占用的内存量都是固定的。

4.1.2 声明和初始化静态数组

在前一小节，您声明了一个名为 MyNumbers 的数组，它包含 5 个类型为 int 的元素（即整数），这些元素都被初始化为 0。在 C++中，数组声明遵循如下简单的语法：

```
element-type array-name [number of elements] = {optional initial values}
```

在声明数组时，还可像下面这样分别初始化每个元素，这里将 5 个元素分别初始化为不同的整数：

```
int MyNumbers [5] = {34, 56, -21, 5002, 365};
```

可将数组的所有元素都初始化为相同的值，如下所示：

```
int MyNumbers [5] = {100}; // initialize all integers to 100
```

也可只初始化部分元素，如下所示：

```
int MyNumbers [5] = {34, 56}; // initialize first two elements
```

可将数组长度（即数组包含的元素数）定义为常量，并在数组定义中使用该常量：

```
const int ARRAY_LENGTH = 5;
int MyNumbers [ARRAY_LENGTH] = {34, 56, -21, 5002, 365};
```

需要在多个地方访问并使用数组的长度（如遍历数组中的元素）时，这很有用。这样就无需在每个地方修改数组的长度，而只需修改 const int 声明中的初始值。

注意 如果您只初始化数组的部分元素，有些编译器可能将您忽略的元素初始化为零。

如果知道数组中每个元素的初始值，可不指定数组包含的元素数：

```
int MyNumbers [] = {2011, 2052, -525};
```

上述代码创建一个数组，它包含 3 个 int 元素，这些元素的初始值分别为 2011、2052 和-525。

注意 前面声明的所有数组都是静态数组，因为它们的长度在编译阶段就已确定。这种数组不能存储更多的数据；同时，即便有部分元素未被使用，它们占据的内存也不会减少。

4.1.3 数组中的数据是如何存储的

想想书架上放在一起的图书吧，这就是一个一维数组，因为它只沿一个维度延伸，这个维度就是元素数。每本书都是一个数组元素，而书架就像为存储这些图书而预留的内存，如图 4.1 所示。

这里给图书从零开始编号，这并非错误。后面您将看到，在 C++中，数组索引从零而不是 1 开始。类似于书架上的 5 本图书，包含 5 个整数的数组 MyNubers 类似于图 4.2。

注意到这个数组占用的内存空间包含 5 块，每块的大小都相同。块大小取决于数组存储的数据类型，这里是 int。您可能还记得，第 3 章研究了 int 变量的长度，因此编译器为数组 MyNumbers 预留的内存量为 sizeof(int)* 5。一般而言，编译器为数组预留的内存量为（单位为字节）：

```
Bytes consumed by an array = sizeof(element-type) * Number of Elements
```

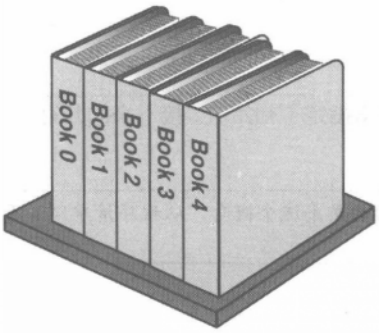


图 4.1 书架上的图书：一维数组

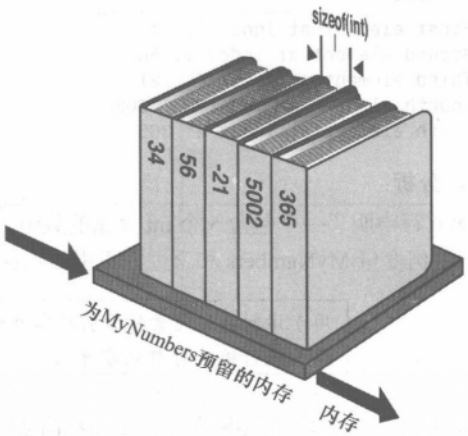


图 4.2 内存中包含 5 个整数的数组 MyNumbers

4.1.4 访问存储在数组中的数据

要访问数组中的元素，可使用从零开始的索引。这些索引之所以被称为从零开始的，是因为数组中第一个元素的索引为零。因此，存储在数组 MyNumbers 中的第一个整数值为 MyNumbers[0]，第二个为 MyNumbers[1]，依此类推。第 5 个元素为 MyNumbers[4]，换句话说，数组中最后一个元素的索引总是比数组长度少 1。

被要求访问索引为 N 的元素时，编译器以第一个元素（索引为零）的内存地址为起点，加上偏移量 N*sizeof(element)，即向前跳 N 个元素，到达包含第 N+1 个元素的地址。C++编译器不会检查索引是否在数组的范围内，您可从只包含 10 个元素的数组中取回索引为 1001 的元素，但这样做将给程序带来安全和稳定性方面的风险。访问数组时，确保不超越其边界是程序员的职责。

警告

访问数组时，如果超越其边界，结果将是无法预料的。在很多情况下，这将导致程序崩溃。应不惜一切代价避免访问数组时超越其边界。

程序清单 4.1 演示了如何声明一个 int 数组、初始化其元素并将元素的值显示到屏幕上。

程序清单 4.1 声明一个 int 数组并访问其元素

```
 0:  #include <iostream>
 1:
 2:  using namespace std;
 3:
 4:  int main ()
 5:  {
 6:     int MyNumbers [5] = {34, 56, -21, 5002, 365};
 7:
 8:     cout << "First element at index 0: " << MyNumbers [0] << endl;
 9:     cout << "Second element at index 1: " << MyNumbers [1] << endl;
10:     cout << "Third element at index 2: " << MyNumbers [2] << endl;
11:     cout << "Fourth element at index 3: " << MyNumbers [3] << endl;
12:     cout << "Fifth element at index 4: " << MyNumbers [4] << endl;
13:
14:     return 0;
15:  }
```

▼ **输出：**

```
First element at index 0: 34
Second element at index 1: 56
Third element at index 2: -21
Fourth element at index 3: 5002
Fifth element at index 4: 365
```

▼ **分析：**

第 6 行声明了一个包含 5 个 int 元素的数组，并给每个元素指定了初始值。接下来的几行代码使用 cout、数组变量 MyNumbers 和合适的索引显示这些整数。

注意 用于访问数组元素的索引从零开始，为帮助读者熟悉这个概念，从程序清单 4.1 开始，给代码行编号时都从零开始。

4.1.5 修改存储在数组中的数据

在前一个程序清单中，并未将用户定义的数据输入到数组中。给数组元素赋值的语法与给 int 变量赋值的语法很像。

```
int AnIntegerValue;
AnIntegerValue = 2011;
```

例如，将 2011 赋给 int 变量的代码类似于下面这样：

```
MyNumbers [3] = 2011; // Assign 2011 to the fourth element
```

程序清单 4.2 演示了如何使用常量指定数组的长度，还演示了如何在程序执行期间给数组元素赋值。

程序清单 4.2 给数组元素赋值

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int ARRAY_LENGTH = 5;
 6:
 7:    // Array of 5 integers, initialized to zero
 8:    int MyNumbers [ARRAY_LENGTH] = {0};
 9:
10:    cout << "Enter index of the element to be changed: ";
11:    int nElementIndex = 0;
12:    cin >> nElementIndex;
13:
14:    cout << "Enter new value: ";
15:    cin >> MyNumbers [nElementIndex];
16:
17:    cout << "First element at index 0: " << MyNumbers [0] << endl;
18:    cout << "Second element at index 1: " << MyNumbers [1] << endl;
19:    cout << "Third element at index 2: " << MyNumbers [2] << endl;
20:    cout << "Fourth element at index 3: " << MyNumbers [3] << endl;
21:    cout << "Fifth element at index 4: " << MyNumbers [4] << endl;
22:
23:    return 0;
24: }
```

▼ 输出：

```
Enter index of the element to be changed: 2
Enter new value: 2011
First element at index 0: 0
Second element at index 1: 0
Third element at index 2: 2011
Fourth element at index 3: 0
Fifth element at index 4: 0
```

▼ 分析：

第 8 行声明数组时，使用了 int 常量 ARRAY_LENGTH，该常量之前被初始化为 5。这是一个静态数组，其长度在编译期间是固定的。编译器将 ARRAY_LENGTH 替换为 5，认为 MyArray 是一个包含 5 个元素的 int 数组。第 10～12 行询问用户要设置哪个数组元素，并将索引存储到 int 变量 ElementIndex 中。第 15 行使用这个 int 变量来修改数组的内容。输出表明，用户要修改索引为 2 的元素，而实际修改的是第 3 个元素，因为索引从零开始，您必须习惯这一点。

注意

在数组包含 5 个 int 元素时，很多 C++新手将第 5 个值赋给索引为 5 的元素。这超出了数组的边界，因为编译后的代码将试图访问数组的第 6 个元素，这不在定义的范围内。这种错误被称为篱笆柱（fence-post）错误。之所以叫这个名字，是因为建造篱笆时，需要的篱笆柱数总是比部分数多 1。

警告

程序清单 4.2 遗漏了一些必不可少的代码：没有检查用户输入的索引是否在数组的边界内。实际上，该程序应检查 ElementIndex 是否为 0～4，如果不是，则拒绝修改数组。由于缺少这种检查，用户将被允许输入超越数组边界的值。在最糟糕的情况下，这将导致应用程序崩溃。执行检查将在第 6 章介绍。

使用循环遍历数组元素

按顺序处理数组及其元素时，应使用循环进行遍历。要了解如何使用 for 循环高效地插入或访问数组元素，请参阅程序清单 6.10。

应该	不应该
务必初始化数组，否则其元素将包含未知值。 使用数组时，务必确保在其边界内。	在包含 N 个元素的数组中，不要使用索引 N 来访问第 N 个元素。 别忘了，使用索引零访问的是第一个数组元素。

4.2 多维数组

到目前为止，读者看到的数组都类似于书架上的图书，书架越长，可放的书越多，书架越短，可放的书越少。也就是说，长度是决定书架容量的唯一维度，因此是一维的。如果要使用数组模拟图 4.3 所示的太阳能电池板，该如何办呢？不同于书架，太阳能电池板沿两个维度延伸：长度和宽度。

正如您在图 4.3 中看到的，6 块太阳能电池板以两维方式排列，组成两行、三列。从某种意义上说，

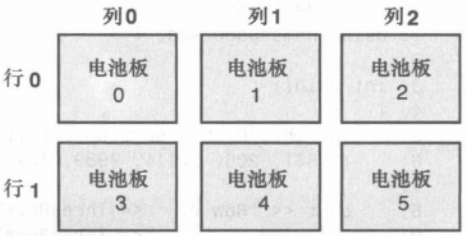


图 4.3 屋顶的一组太阳能电池板

可将这种布局视为一个包含两个元素的数组，其中每个元素本身是一个包含三块电池板的数组，换句话说，这是一个由数组组成的数组。在 C++中，可模拟二维数组，但并不限于二维数组，根据您的需求和应用程序的性质，还可在内存中模拟多维数组。

4.2.1 声明和初始化多维数组

在 C++中，要声明多维数组，可指定每维包含的元素数。因此，要声明一个 int 二维数组，以表示图 4.3 所示的电池板，可以像下面这样做：

```
int SolarPanelIDs [2][3];
```

在图 4.3 中，给每块电池板指定了一个 ID，这 6 块电池板的 ID 为 0～5。如果要以这样的方式初始化相应的 int 数组，可以像下面这样做：

```
int SolarPanelIDs [2][3] = {{0, 1, 2}, {3, 4, 5}};
```

正如您看到的，初始化语法与初始化两个一维数组的语法类似。这是一个包含两行的二维数组，而不是两个数组。如果该数组包含三行、三列，则初始化语法将类似于下面这样：

```
int ThreeRowsThreeColumns [3][3] = {{-501, 206, 2011}, {989, 101, 206}, {303,
456, 596}};
```

注意

虽然 C++让您能够模拟多维数组，但存储数组的内存是一维的。编译器将多维数组映射到内存，而内存只沿一个方向延伸。如果您愿意，也可像下面这样初始化数组 SolarPanelIDs，其效果相同：

```
int SolarPanelIDs [2][3] = {0, 1, 2, 3, 4, 5};
```

然而，前面的初始化方法更佳，因为它让您更容易将多维数组想象为数组的数组。

4.2.2 访问多维数组中的元素

可将多维数组视为由数组组成的数组。因此，对于包含三行、三列的二维 int 数组，可将其视为一个包含 3 个元素的数组，其中每个元素都是一个包含 3 个 int 元素的数组。

在需要访问该数组中的 int 时，可使用第一个下标指出该 int 所属的数组，并使用第二个下标指出该 int。请看下面的数组：

```
int ThreeRowsThreeColumns [3][3] = {{-501, 206, 2011}, {989, 101, 206}, {303,
456, 596}};
```

初始化方式让您能够将其视为三个数组，其中每个数组包含三个 int。其中，值为 206 的元素的位置为[0][1]，值为 456 的元素的位置为[2][2]。程序清单 4.3 演示了如何访问该数组中的 int 元素。

程序清单 4.3 访问多维数组中的元素

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int ThreeRowsThreeColumns [3][3] = \
 6:    {{-501, 206, 2011}, {989, 101, 206}, {303, 456, 596}};
 7:
 8:    cout << "Row 0: " << ThreeRowsThreeColumns [0][0] << " " \
 9:                     << ThreeRowsThreeColumns [0][1] << " " \
10:                     << ThreeRowsThreeColumns [0][2] << endl;
```

```
11:
12:
13:     cout << "Row 1: " << ThreeRowsThreeColumns [1][0] << " " \
14:                       << ThreeRowsThreeColumns [1][1] << " " \
15:                       << ThreeRowsThreeColumns [1][2] << endl;
16:
17:     cout << "Row 2: " << ThreeRowsThreeColumns [2][0] << " "\
18:                       << ThreeRowsThreeColumns [2][1] << " " \
19:                       << ThreeRowsThreeColumns [2][2] << endl;
20:
21:     return 0;
22: }
```

▼ 输出：

```
Row 0: -501 206 2011
Row 1: 989 101 206
Row 2: 303 456 596
```

▼ 分析：

注意到访问元素时将每行视为一个数组，从第 1 行开始（其索引为 0），到第 3 行结束（其索引为 2）。由于每行都是一个数组，因此第 10 行访问第 1 行的第 3 个元素.

注意

在程序清单 4.3 中，随着数组包含的元素或维度数的增加，代码长度将激增。在专业开发环境中，这种代码实际上不可行。

程序清单 6.14 演示了一种更高效的多维数组访问方式，它使用嵌套 for 循环来访问多维数组中的所有元素。使用 for 循环时，代码更短且不容易出错；另外，程序长度也不受数组包含的元素数的影响。

4.3 动态数组

假设要在应用程序中存储医院的病历，程序员将无法知道需要处理的病历数上限。就小医院而言，为稳妥起见，程序员可对上限做合理的假设。在这种情况下，程序员将预留大量的内存，进而降低系统的性能。

为减少占用的内存，可不使用前面介绍的静态数组，而使用动态数组，并在运行阶段根据需要增大动态数组。C++提供了 std::vector，这是一种方便且易于使用的动态数组，如程序清单 4.4 所示。

程序清单 4.4 创建 int 动态数组并动态地增大其容量

```
 0: #include <iostream>
 1: #include <vector>
 2:
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:     vector<int> DynArrNums (3);
 8:
 9:     DynArrNums[0] = 365;
10:     DynArrNums[1] = -421;
11:     DynArrNums[2]= 789;
12:
13:     cout << "Number of integers in array: " << DynArrNums.size() << endl;
14:
15:     cout << "Enter another number for the array" << endl;
```

```
16:    int AnotherNum = 0;
17:    cin >> AnotherNum;
18:    DynArrNums.push_back(AnotherNum);
19:
20:    cout << "Number of integers in array: " << DynArrNums.size() << endl;
21:    cout << "Last element in array: ";
22:    cout << DynArrNums[DynArrNums.size() - 1] << endl;
23:
24:    return 0;
25: }
```

▼ **输出：**

```
Number of integers in array: 3
Enter another number for the array
2011
Number of integers in array: 4
Last element in array: 2011
```

▼ **分析：**

由于还未介绍矢量和模板，如果不明白程序清单 4.4 中的语法，也不用担心。请尝试将输出与代码关联起来。从输出可知，数组的初始长度为 3，这与第 7 行的矢量声明一致。在知道这一点的情况下，第 15 行仍让用户输入第 4 个数字，而第 18 行使用 push_back()将这个数字压入到矢量中。这个矢量动态地调整其长度，以存储更多数据。输出证明了这一点：矢量的长度变成了 4。访问矢量中的数据时，语法与访问静态数组类似。第 22 行访问最后一个元素，其位置是在运行阶段计算得到的。索引从零开始，而 size()返回矢量包含的元素数，因此最后一个元素的索引为 size()-1。

注意

要使用动态数组类 std::vector，需要包含头文件 vector，如程序清单 4.4 的第 1 行所示：

```
#include <vector>
```

矢量将在第 17 章更详细地介绍。

4.4 C 风格字符串

C 风格字符串是一种特殊的字符数组。您在前面编写代码时使用过字符串字面量，它们就是 C 风格字符串：

```
std::cout << "Hello World";
```

这与下面使用数组的方式等价：

```
char SayHello[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '\0'};
std::cout << SayHello << std::endl;
```

请注意，该数组的最后一个字符为空字符‘\0’。这也被称为字符串结束字符，因为它告诉编译器，字符串到此结束。这种 C 风格字符串是特殊的字符数组，因为总是在最后一个字符后加上空字符‘\0’。您在代码中使用字符串字面量时，编译器将负责在它后面添加‘\0’。

在数组中间插入‘\0’并不会改变数组的长度，而只会导致将该数组作为输入的字符串处理将到这个位置结束，程序清单 4.5 演示了这一点。

注意

‘\0’ 看起来像两个字符。使用键盘输入它时，确实需要输入两个字符，但反斜杆是编译器能够理解的特殊转义编码，\0 表示空，即它让编译器插入空字符或零。

您不能将其写做 ‘0’，因为它表示字符 0，其 ASCII 编码为 48。

要获悉字符 0 和其他字符的 ASCII 编码，请参阅附录 E。

程序清单 4.5 分析 C 风格字符串中的终止空字符

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    char SayHello[] = {'H','e','l','l','o',' ','W','o','r','l','d','\0'};
 6:    cout << SayHello << endl;
 7:    cout << "Size of array: " << sizeof(SayHello) << endl;
 8:
 9:    cout << "Replacing space with null" << endl;
10:    SayHello[5] = '\0';
11:    cout << SayHello << endl;
12:    cout << "Size of array: " << sizeof(SayHello) << endl;
13:
14:    return 0;
15: }
```

▼ 输出：

```
Hello World
Size of array: 12
Replacing space with null
Hello
Size of array: 12
```

▼ 分析：

第 10 行将“Hello World”中的空格替换为终止空字符。这样，该数组包含两个终止空字符，但只有第一个发挥了作用。将空格替换为空字符后，显示时字符串被截短为 Hello。第 7 和 12 行的 sizeof()的输出表明，数组的长度没变，虽然显示的字符串发生了很大变化。

警告

在程序清单 4.5 中，如果在第 5 行声明并初始化字符数组时忘记添加‘\0’，则打印该数组时，Hello World 后面将出现垃圾字符，这是因为 std::cout 只有遇到空字符后才会停止打印，即便这将跨越数组的边界。

在有些情况下，这种错误可能导致程序崩溃，进而影响系统的稳定性。

C 风格字符串充斥着危险，程序清单 4.6 演示了这一点。

程序清单 4.6 分析 C 风格字符串中的终止空字符

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Enter a word NOT longer than 20 characters:" << endl;
 6:
 7:    char UserInput [21] = {'\0'};
 8:    cin >> UserInput;
 9:
10:    cout << "Length of your input was: " << strlen (UserInput) << endl;
11:
12:    return 0;
13: }
```

▼ 输出：

```
Enter a word NOT longer than 20 characters:
Don'tUseThisProgram
Length of your input was: 19
```

▼ 分析：

输出说明了这种危险。该程序请求用户输入数据时不要超过 20 个字符，因为第 7 行声明了一个字符数组，用于存储用户输入，其长度是固定的（静态的），为 21 个字符。由于最后一个字符必须是终止空字符'\0'，因此该数组最多可存储 20 个字符。第 10 行使用了 strlen 来计算该字符串的长度。strlen 遍历该字符数组，直到遇到表示字符串末尾的终止空字符，并计算遍历的字符数。cin 在用户输入的末尾插入终止空字符。strlen 的这种行为非常危险，因为如果用户输入的文本长度超过了指定的上限，strlen 将跨越字符数组的边界。程序清单 6.2 演示了如何实现相关的检查，以免写入数组时跨越其边界。

4.5 C++字符串：使用 std::string

无论是处理文本输入，还是执行拼接等字符串操作，使用 C++标准字符串都是效率最高的方式。

警告

使用 C 语言编写的应用程序经常使用 strcpy 等字符串复制函数、strcat 等拼接函数，还经常使用 strlen 来确定字符串的长度；具有较强 C 语言背景的 C++程序员编写的应用程序亦如此。

这些 C 风格字符串作为输入的函数非常危险，因为它们会寻找终止空字符，如果程序员没有在字符数组末尾添加空字符，这些函数将跨越字符数组的边界。

C++提供了 std::string，这是一种功能强大而安全的字符串操作方式，如程序清单 4.7 所示。不同于字符数组（C 风格字符串实现），std::string 是动态的，在需要存储更多数据时其容量将增大。

程序清单 4.7 使用 std::string 初始化字符串、存储用户输入、复制和拼接字符串以及确定字符串的长度

```
 0: #include <iostream>
 1: #include <string>
 2:
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:     string Greetings ("Hello std::string!");
 8:     cout << Greetings << endl;
 9:
10:     cout << "Enter a line of text: " << endl;
11:     string FirstLine;
12:     getline(cin, FirstLine);
13:
14:     cout << "Enter another: " << endl;
15:     string SecLine;
16:     getline(cin, SecLine);
17:
18:     cout << "Result of concatenation: " << endl;
19:     string Concat = FirstLine + " " + SecLine;
20:     cout << Concat << endl;
21:
22:     cout << "Copy of concatenated string: " << endl;
23:     string Copy;
24:     Copy = Concat;
25:     cout << Copy << endl;
```

```
26:
27:    cout << "Length of concat string: " << Concat.length() << endl;
28:
29:    return 0;
30: }
```

▼ **输出：**

```
Hello std::string!
Enter a line of text:
I love
Enter another:
C++ strings
Result of concatenation:
I love C++ strings
Copy of concatenated string:
I love C++ strings
Length of concat string: 18
```

▼ **分析：**

请尝试将输出与代码关联起来，现在暂时不要管其中的新语法。该程序首先显示一个字符串，该字符串在第 7 行被初始化为“Hello std::string”。接下来，它让用户输入两行文本，并将它们分别存储在变量 Firstline 和 Secline 中，如第 12 和 16 行所示。字符串拼接非常简单，看起来很像算术加法运算，如第 19 行所示。这里还在两行之间添加了一个空格。复制也很简单，只需赋值即可，如第 24 行所示。第 27 行对字符串调用 length()，以确定其长度。

注意

要使用 C++字符串，需要包含头文件 string:

```
#include <string>
```

如程序清单 4.7 的第 1 行所示。

要详细了解 std::string 的各种函数，请参阅第 16 章。由于还您未学习类和模板，请跳过该章不熟悉的部分，重点理解示例程序的要点。

4.6 总结

本章介绍了数组的基本知识：数组是什么及其用途。您学习了如何声明和初始化数组以及如何读写数组元素。您了解到，避免超越数组边界至关重要，这被称为缓冲区溢出。将输入用作索引前应对其进行检查，这有助于避免跨越数组边界。

动态数组让程序员无需在编译阶段考虑其最大长度，使用动态数组可更好地管理内存，以免分配过多的内存，而又不使用它们。

您还了解到，C 风格字符串是特殊的 char 数组，用终止空字符‘\0’标识末尾。更重要的是，您了解到，C++提供了更佳的选择——std::string，它包含一些方便的函数，让您能够判断字符串的长度、拼接字符串等。

4.7 问与答

问：为何要不怕麻烦，去初始化静态数组的元素？

答：数组不同于其他类型的变量，除非进行初始化，否则它将包含无法预测的值，因为内存保留最后一次操作时的内容。通过初始化数组，可确保内存包含确定的值。

问：需要基于前一个问题所说的原因初始化动态数组的元素吗？

答：实际上，不需要。动态数组相当聪明，无需将其元素初始化为默认值，除非应用程序要求数组包含特定的初始值。

问：在可以选择的情况下，您会使用需要以空字符结尾的 C 风格字符串吗？

答：除非有人拿枪指着您的头。C++ std::string 更安全，并提供了很多功能，任何优秀的程序员都应避免使用 C 风格字符串。

问：计算字符串长度时，包括末尾的空字符吗？

答：不包括。字符串“Hello World”的长度为 11，这包括其中的空格，但不包括末尾的空字符。

问：如果要用 char 数组标识 C 风格字符串，应将数组声明为多长？

答：这是 C 风格字符串最复杂的地方之一。数组的长度应比它可能包含的最长字符串长 1，以便在末尾包含空字符。如果 char 数组可能存储的最长字符串为“Hello World”，则应将该数组的长度声明为 12（11+1）。

4.8 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 中的答案。在继续学习下一章之前，请务必弄懂这些答案。

4.8.1 测验

1. 对于程序清单 4.1 中的数组 MyNumbers，第一个元素和最后一个元素的索引分别是多少？
2. 如果需要让用户输入字符串，该使用 C 风格字符串吗？
3. 在编译器看来，‘\0’表示多少个字符？
4. 如果忘记在 C 风格字符串末尾添加终止空字符，使用它的结果将如何？
5. 根据程序清单 4.4 中矢量的声明，尝试声明一个包含 char 元素的动态数组。

4.8.2 练习

1. 声明表示国际象棋棋盘的数组；该数组的类型应为枚举，该枚举定义了棋盘方格中的棋子。
2. 查错：下面的代码段有什么错误？

```
int MyNumbers[5] = {0};
MyNumbers[5] = 450; // Setting the 5th element to value 450
```

3. 查错：下面的代码段有什么错误？

```
int MyNumbers[5];
cout << MyNumbers[3];
```

第5章
使用表达式、语句和运算符

从本质上说，程序是一组按顺序执行的命令。这些命令为表达式和语句，使用运算符执行特定的计算或操作。

在本章中，您将学习：

- 什么是语句；
- 什么是语句块（复合语句）；
- 什么是运算符；
- 如何执行简单的算术运算和逻辑运算。

5.1 语句

无论是口头语言还是编程语言，它们都是由按顺序执行的语句组成的。下面来分析您学习的第一条重要语句：

```
cout << "Hello World" << endl;
```

这条语句使用 cout 在屏幕（控制台）上显示文本。在 C++中，所有语句都以分号（;）结尾，分号界定了语句的边界。这就像您在书写英语时在句末添加句号（.）一样。可以接着分号开始下一条语句，但出于方便和可读性考虑，通常每条语句放在一行中。换句话说，下面一行代码实际上包含两条语句：

```
cout << "Hello World" << endl; cout << "Another hello" << endl; // One line, two
➥statements
```

注意 编译器通常不考虑空白，这包括空格、制表符、换行符、回车等，但字符串字面量中的空格将导致输出不同。

下面的代码非法：

```
cout << "Hello
     World" << endl; // new line in string literal not allowed
```

这样的代码通常会导致错误——编译器指出第一行缺少引号（"）和结束语句的分号（;）。如果出于某种原因，要将一条语句放到两行中，可在第一行末尾添加反斜杆（\）：

```
cout << "Hello \
     World" << endl; // split to two lines is OK
```

对于前面的语句，另一种书写方式是将字符串字面量分成两个：

```
cout << "Hello "
   "World" << endl; // two string literals is also OK
```

编译器注意到两个相邻的字符串字面量后，将把它们拼接成一个。

注意 在文本元素很长或表达式由很多变量组成，导致语句很长，大多数显示器无法完全显示时，将语句划分成多行很有帮助。

5.2 复合语句（语句块）

可使用花括号（{}）将多条语句组合在一起，以创建复合语句（语句块）：

```
{
   int Number = 365;
   cout << "This block contains an integer and a cout statement" << endl;
}
```

语句块通常将众多语句组合在一起，指出它们属于同一条语句。编写 if 语句或循环时，语句块特别有用，这将在第 6 章介绍。

5.3 使用运算符

运算符是 C++提供的工具，让您能够使用数据：对其进行变换、处理甚至根据数据做决策。

5.3.1 赋值运算符（=）

赋值运算符的用法比较直观，本书一直在使用它：

```
int MyInteger = 101;
```

上述语句使用赋值运算符将一个 int 变量初始化为 101。赋值运算符将左边的操作数的值（左值）替换为右边的操作数的值（右值）。

5.3.2 理解左值和右值

左值通常是内存单元。在前面的示例中，变量 MyInteger 实际上指向一个内存单元，属于左值。另一方面，右值可以是内存单元的内容。

因此，所有的左值都可用作右值，但并非所有的右值都可用作左值。为更好地理解这一点，请看下面的示例，这行代码不合理，不能通过编译：

```
101 = MyInteger;
```

5.3.3 加法运算符（+）、减法运算符（-）、乘法运算符（*）、除法运算符（/）和求模运算符（%）

可对两个操作数执行算术运算：使用+相加、使用-相减、使用*相乘、使用/相除、使用%求模：

```
int Num1 = 22;
int Num2 = 5;
int addition = Num1 + Num2; // 27
int subtraction = Num1 - Num2; // 17
int multiplication = Num1 * Num2; // 110
int division = Num1 / Num2; // 4
int modulo = Num1 % Num2; // 2
```

除法运算符（/）返回两个数相除的结果。然而，如果两个操作数都是整数，结果将不包含小数，因为根据定义，整数不能包含小数。求模运算符（%）返回除法运算的余数，只能用于整数。程序清

单 5.1 是一个简单的应用程序，演示了如何对用户输入的两个数字执行各种算术运算。

程序清单 5.1 演示如何对用户输入的整数执行算术运算

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:    cout << "Enter two integers:" << endl;
6:    int Num1 = 0, Num2 = 0;
7:    cin >> Num1;
8:    cin >> Num2;
9:
10:   cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
11:   cout << Num1 << " - " << Num2 << " = " << Num1 - Num2 << endl;
12:   cout << Num1 << " * " << Num2 << " = " << Num1 * Num2 << endl;
13:   cout << Num1 << " / " << Num2 << " = " << Num1 / Num2 << endl;
14:   cout << Num1 << " % " << Num2 << " = " << Num1 % Num2 << endl;
15:
16:  return 0;
17: }
```

▼ 输出：

```
Enter two integers:
365
25
365 + 25 = 390
365 - 25 = 340
365 * 25 = 9125
365 / 25 = 14
365 % 25 = 15
```

▼ 分析：

这个程序的大部分代码的含义都是不言自明的。最有趣的代码可能是使用求模运算符（%）的那行，它返回 Num1（365）与 Num2（25）相除的余数。

5.3.4 递增运算符（++）和递减运算符（--）

有时需要将变量加 1，尤其是控制循环的变量：每次执行循环时，都需要将这种变量的值递增或递减。

为帮助您完成这种任务，C++提供了递增运算符（++）和递减运算符（--）。

这些运算符的使用语法如下：

```
int Num1 = 101;
int Num2 = Num1++; // Postfix increment operator
int Num2 = ++Num1; // Prefix increment operator
int Num2 = Num1--; // Postfix decrement operator
int Num2 = --Num1; // Prefix decrement operator
```

从上述示例代码可知，使用递增和递减运算符的方式有两种：放在操作数的前面或放在操作数的后面。放在操作数前面时，称为前缀递增或递减运算符；而放在操作数后面时，称为后缀递增或递减运算符。

5.3.5 前缀还是后缀

首先需要理解前缀和后缀之间的差别，这样才能选择合适的方式。使用后缀运算符时，先将右值

赋给左值，再将右值递增或递减。这意味着在上述所有使用后缀运算符的代码中，Num2 都为 Num1 的旧值（执行递增或递减前的值）。

前缀运算符的行为完全相反，即先将右值递增或递减，再将结果赋给左值。在所有使用后缀运算符的代码中，Num2 的值都与 Num1 的值相同。程序清单 5.2 演示了将前缀和后缀递增和递减运算符用于一个 int 变量的结果。

程序清单 5.2 前缀运算符和后缀运算符之间的差别

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int MyInt = 101;
 6:    cout << "Start value of integer being operated: " << MyInt << endl;
 7:
 8:    int PostFixInc = MyInt++;
 9:    cout << "Result of Postfix Increment = " << PostFixInc << endl;
10:    cout << "After Postfix Increment, MyInt = " << MyInt << endl;
11:
12:    MyInt = 101; // Reset
13:    int PreFixInc = ++MyInt;
14:    cout << "Result of Prefix Increment = " << PreFixInc << endl;
15:    cout << "After Prefix Increment, MyInt = " << MyInt << endl;
16:
17:    MyInt = 101;
18:    int PostFixDec = MyInt--;
19:    cout << "Result of Postfix Decrement = " << PostFixDec << endl;
20:    cout << "After Postfix Decrement, MyInt = " << MyInt << endl;
21:
22:    MyInt = 101;
23:    int PreFixDec = --MyInt;
24:    cout << "Result of Prefix Decrement = " << PreFixDec << endl;
25:    cout << "After Prefix Decrement, MyInt = " << MyInt << endl;
26:
27:    return 0;
28: }
```

▼ 输出：

```
Start value of integer being operated: 101
Result of Postfix Increment = 101
After Postfix Increment, MyInt = 102
Result of Prefix Increment = 102
After Prefix Increment, MyInt = 102
Result of Postfix Decrement = 101
After Postfix Decrement, MyInt = 100
Result of Prefix Decrement = 100
After Prefix Decrement, MyInt = 100
```

▼ 分析：

结果表明，后缀运算符和前缀运算符的差别在于：在第 8 和 18 行，被赋值的左值包含对 MyInt 执行递增或递减运算前的值。另一方面，在第 13 和 23 行，左值包含对 MyInt 执行递增或递减运算后的值。这是最重要的差别，选择合适的运算符时必须牢记这一点。

在下面的语句中，使用前缀还是后缀运算符对结果没有影响：

```
MyInt++; // Is the same as…
++MyInt;
```

这是因为没有将原来的值赋给其他变量，这两种情形的最终结果都是将 MyInt 递增。

注意

您经常会听到前缀运算符的性能更高还是后缀运算符更高的争论。换句话说，++MyInt 优于 MyInt++。

至少从理论上说确实如此，因为使用后缀运算符时，编译器需要临时存储初始值，以防需要将其赋给其他变量。就整型变量而言，这对性能的影响几乎可以忽略不计，但对某些类来说，这种争论也许有意义。

明智地选择数据类型，以免溢出

诸如 short、int、long、unsigned short、unsigned int、unsigned long 等数据类型的容量有限，如果算术运算的结果超出了选定数据类型的上限，将导致溢出。

就拿 unsigned short 来说吧，它占用 16 位内存，因此取值范围为 0～65535。usigned short 变量的值为 65535 后，如果再加 1，将导致溢出，结果为 0。这很像汽车的里程表，如果它只支持 5 位数字，则里程超过 99999 公里（或英里）后，里程表将发生机械溢出。

在这种情况下，将计数器的变量类型指定为 unsigned short 不合适。要支持大于 65535 的数字，程序员应使用数据类型 unsigned int。

数据类型 signed short 的取值范围为–32768～32768，如果这种变量的值已经是 32768，则将其加 1 的结果为最小的负数——这取决于编译器。

程序清单 5.3 演示了执行算术运算时可能不小心导致的溢出错误。

程序清单 5.3 演示有符号整型变量和无符号整型变量溢出的负面影响

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    unsigned short UShortValue = 65535;
 6:    cout << "Incrementing unsigned short " << UShortValue << " gives: ";
 7:    cout << ++UShortValue << endl;
 8:
 9:    short SignedShort = 32767;
10:    cout << "Incrementing signed short " << SignedShort<< " gives: ";
11:    cout << ++SignedShort << endl;
12:
13:    return 0;
14:
```

▼ 输出：

```
Incrementing unsigned short 65535 gives: 0
Incrementing signed short 32767 gives: -32768
```

▼ 分析：

正如您看到的，无意的溢出导致应用程序的行为不可预测。如果使用 unsigned short 变量来指定要分配的内存量，则在您原本要分配 65356 字节内存时，实际上请求的却是零字节。

5.3.6 相等运算符（==）和不等运算符（!=）

经常需要在执行操作前检查条件是否满足，相等运算符（==，操作数相等）和不等运算符（!=，操作数不相等）可帮助您完成这种检查。

相等性检查的结果为布尔值，即 true 或 false：

```
int MyNum = 20;
bool CheckEquality = (MyNum == 20); // true
bool CheckInequality = (MyNum != 100); // true

bool CheckEqualityAgain = (MyNum == 200); // false
bool CheckInequalityAgain = (MyNum != 20); // false
```

5.3.7 关系运算符

除相等性检查外，您可能还想检查变量与特定值之间的大小关系。为帮助您进行这种检查，C++提供了关系运算符，如表 5.1 所示。

表 5.1 系运算符

运算符名称	描述
小于（<）	如果左边的操作数小于右边的操作数（Op1<Op2），则结果为 true，否则结果为 false
大于（>）	如果左边的操作数大于右边的操作数（Op1>Op2），则结果为 true，否则结果为 false
小于等于（<=）	如果左边的操作数小于或等于右边的操作数，则结果为 true，否则结果为 false
大于等于（>=）	如果左边的操作数大于或等于右边的操作数，则结果为 true，否则结果为 false

如表 5.1 所示，比较运算的结果总是布尔值，即要么为 true，要么为 false。下面的示例代码演示了如何使用表 5.1 所示的关系运算符：

```
int MyNum = 20; // sample integer value
bool CheckLessThan = (MyNum < 100);  // true
bool CheckGreaterThan = (MyNum > 100); // false
bool CheckLessThanEqualTo = (MyNum <= 20); // true
bool CheckGreaterThanEqualTo = (MyNum >= 20); // true
bool CheckGreaterThanEqualToAgain = (MyNum >= 100); // false
```

程序清单 5.4 演示了这些运算符的作用——将结果显示在屏幕上。

程序清单 5.4 演示相等运算符和关系运算符

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Enter two integers:" << endl;
 6:    int Num1 = 0, Num2 = 0;
 7:    cin >> Num1;
 8:    cin >> Num2;
 9:
10:    bool Equality = (Num1 == Num2);
11:    cout << "Result of equality test: " << Equality << endl;
12:
13:    bool Inequality = (Num1 != Num2);
14:    cout << "Result of inequality test: " << Inequality << endl;
15:
16:    bool GreaterThan = (Num1 > Num2);
17:    cout << "Result of " << Num1 << " > " << Num2;
18:    cout << " test: " << GreaterThan << endl;
19:
```

```
20:    bool LessThan = (Num1 < Num2);
21:   cout << "Result of " << Num1 << " < " << Num2 << " test: " << LessThan <<
      ➥endl;
22:
23:    bool GreaterThanEquals = (Num1 >= Num2);
24:    cout << "Result of " << Num1 << " >= " << Num2;
25:    cout << " test: " << GreaterThanEquals << endl;
26:
27:    bool LessThanEquals = (Num1 <= Num2);
28:   cout << "Result of " << Num1 << " <= " << Num2;
29:   cout << " test: " << LessThanEquals << endl;
30:
31:    return 0;
32: }
```

▼ 输出：

```
Enter two integers:
365
-24
Result of equality test: 0
Result of inequality test: 1
Result of 365 > -24 test: 1
Result of 365 < -24 test: 0
Result of 365 >= -24 test: 1
Result of 365 <= -24 test: 0
```

再次运行的输出：

```
Enter two integers:
101
101
Result of equality test: 1
Result of inequality test: 0
Result of 101 > 101 test: 0
Result of 101 < 101 test: 0
Result of 101 >= 101 test: 1
Result of 101 <= 101 test: 1
```

▼ 分析：

这个程序显示各种运算的结果。请注意用户提供的两个整数相同时的输出，运算符==、>=和<=的结果相同。

相等运算符和关系运算符只有两种可能的结果，这使得它们非常适合用于做决策，还非常适合用作循环的条件表达式，确保只要条件为 true，就不断执行循环。有条件地执行和循环将在第 6 章更详细地介绍。

5.3.8 逻辑运算 NOT、AND、OR 和 XOR

逻辑 NOT 运算用运算符!表示，用于单个操作数。表 5.2 是逻辑 NOT 运算的真值表，这种运算将提供的布尔标记反转。

表 5.2　　辑 NOT 运算的真值表

操作数	NOT 运算的结果
False	True
True	False

AND、OR 和 XOR 等运算需要两个操作数。仅当两个操作数都为 true 时，逻辑 AND 运算的结果

才为 true。表 5.3 说明了逻辑 AND 运算的结果。

表 5.3 辑 AND 运算的真值表

Operand1	Operand2	Operand1 AND Operand2 的结果
False	False	False
True	False	False
False	True	False
True	True	True

逻辑 AND 运算用运算符&&表示。

只要有一个操作数为 true，逻辑 OR 运算的结果就为 true，如表 5.4 所示。

表 5.4 辑 OR 运算的真值表

Operand1	Operand2	Operand1 OR Operand2 的结果
False	False	False
True	False	True
False	True	True
True	True	True

逻辑 OR 运算用运算符||表示。

逻辑 XOR（异或）运算与逻辑 OR 运算稍有不同，有且只有一个操作数为 true 时，这种运算的结果才为 true，如表 5.5 所示。

表 5.5 辑 XOR 运算的真值表

Operand1	Operand2	Operand1 XOR Operand2 的结果
False	False	False
True	False	True
False	True	True
True	True	False

C++提供了按位 XOR 运算，用运算符^表示。这个运算符对操作数相应的各位执行 XOR 运算。

5.3.9 使用 C++逻辑运算 NOT（!）、AND（&&）和 OR（||）

请看下面的句子：

- 如果明天下雨且没有公交车，我就不能去上班；
- 如果折扣很高或奖金创纪录，我就能买下那辆车。

在编程中，您也需要使用这样的逻辑结构，根据运算的结果决定程序的后续流程。C++提供了逻辑运算符 AND 和 OR，您可在条件语句中使用它们，根据条件改变程序的流程。

程序清单 5.5 演示了逻辑运算符 AND 和 OR 的工作原理。

程序清单 5.5 分析 C++逻辑运算符&&和||

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
```

```
 4: {
 5:    cout << "Enter true(1) or false(0) for two operands:" << endl;
 6:    bool Op1 = false, Op2 = false;
 7:    cin >> Op1;
 8:    cin >> Op2;
 9:
10:    cout << Op1 << " AND " << Op2 << " = " << (Op1 && Op2) << endl;
11:    cout << Op1 << " OR " << Op2 << " = " << (Op1 || Op2) << endl;
12:
13:    return 0;
14: }
```

▼ **输出：**

```
Enter true(1) or false(0) for two operands:
1
0
1 AND 0 = 0
1 OR 0 = 1
```

再次运行的输出：

```
Enter true(1) or false(0) for two operands:
1
1
1 AND 1 = 1
1 OR 1 = 1
```

▼ **分析：**

该程序演示了逻辑运算符 AND 和 OR 的工作原理，但没有演示如何使用它们来做决策。程序清单 5.6 演示了如何使用条件语句和逻辑运算符根据变量的值执行不同的代码行。

程序清单 5.6　在 if 语句中使用逻辑 NOT 和 AND 运算符（!和&&）进行条件处理

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Use boolean values(0 / 1) to answer the questions" << endl;
 6:    cout << "Is it raining? ";
 7:    bool Raining = false;
 8:    cin >> Raining;
 9:
10:    cout << "Do you have buses on the streets? ";
11:    bool Buses = false;
12:    cin >> Buses;
13:
14:    // Conditional statement uses logical AND and NOT
15:    if (Raining && !Buses)
16:      cout << "You cannot go to work" << endl;
17:    else
18:      cout << "You can go to work" << endl;
19:
20:    if (Raining && Buses)
21:      cout << "Take an umbrella" << endl;
22:
23:    if ((!Raining) && Buses)
24:       cout << "Enjoy the sun and have a nice day" << endl;
25:
26:    return 0;
27: }
```

▼ 输出：

```
Use boolean values(0 / 1) to answer the questions
Is it raining? 1
Do you have buses on the streets? 1
You can go to work
Take an umbrella
```

再次运行的输出：

```
Use boolean values(0 / 1) to answer the questions
Is it raining? 1
Do you have buses on the streets? 0
You cannot go to work
```

最后一次运行的输出：

```
Use boolean values(0 / 1) to answer the questions
Is it raining? 0
Do you have buses on the streets? 1
You can go to work
Enjoy the sun and have a nice day
```

▼ 分析：

程序清单 5.6 所示的程序使用了还未介绍的 if 条件语句，请尝试根据输出理解这种语句的行为。第 15 行包含条件表达式(Raining && !Buses)，可将其读作“下雨且没有公交车”。这个表达式使用了逻辑 AND 运算符将没有公交车（对有公交车执行逻辑 NOT 运算）和下雨关联起来。

注意 如果您想更详细地了解 if 语句，可快速浏览第 6 章。

程序清单 5.7 演示了如何将逻辑运算符 NOT 和 OR（!和||）用于条件处理。

程序清单 5.7 使用逻辑运算符 NOT 和 OR 帮助判断您能否购买那辆梦寐以求的汽车

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "Answer questions with 0 or 1" << endl;
 6:     cout << "Is there a deep discount on your favorite car? ";
 7:     bool Discount = false;
 8:     cin >> Discount;
 9:
10:     cout << "Did you get a fantastic bonus? ";
11:     bool FantasticBonus = false;
12:     cin >> FantasticBonus;
13:
14:     if (Discount || FantasticBonus)
15:         cout << "Congratulations, you can buy that car!" << endl;
16:     else
17:         cout << "Sorry, waiting a while is a good idea" << endl;
18:
19:     return 0;
20: }
```

▼ 输出：

```
Answer questions with 0 or 1
Is there a deep discount on your favorite car? 0
Did you get a fantastic bonus? 1
Congratulations, you can buy that car!
```

再次运行的输出：

```
Answer questions with 0 or 1
Is there a deep discount on your favorite car? 0
Did you get a fantastic bonus? 0
Sorry, waiting a while is a good idea
```

最后一次运行的输出：

```
Answer questions with 0 or 1
Is there a deep discount on your favorite car? 1
Congratulations, you can buy that car!
```

▼ 分析：

第 14 行使用了一条 if 语句，而第 16 行是与之配套的 else 语句。在条件(Discount || FantasticBonus)为 true 时，将执行第 15 行的语句。这个表达式包含逻辑运算符||，只要您喜欢的汽车的折扣很高，该表达式就为 true。当该表达式为 false 时，将执行 else 语句后面的语句（第 17 行）。

5.3.10 按位运算符 NOT（～）、AND（&）、OR（|）和 XOR（^）

逻辑运算符和按位运算符之前的差别在于，按位运算符返回的并非布尔值，而是对操作数对应位执行指定运算的结果。C++让您能够执行按位 NOT、OR、AND 和 XOR（异或）运算，它们分别使用～将每位取反、使用|对相应位执行 OR 运算、使用&对相应位执行 AND 运算、使用^对相应位执行 XOR 运算。其中后三个运算符对变量与选择的数字（通常是位掩码）执行相应的运算。

在整数的每位都表示特定标记的状态时，有些按位运算很有用。例如，32 位的整数可用于表示 32 个布尔标记。程序清单 5.8 演示了按位运算符的用法。

程序清单 5.8 对整数的各位执行 NOT、AND、OR 和 XOR 运算，以演示按位运算符的用法

```
 0: #include <iostream>
 1: #include <bitset>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Enter a number (0 - 255): ";
 7:    unsigned short InputNum = 0;
 8:    cin >> InputNum;
 9:
10:    bitset<8> InputBits (InputNum);
11:    cout << InputNum << " in binary is " << InputBits << endl;
12:
13:    bitset<8> BitwiseNOT = (~InputNum);
14:    cout << "Logical NOT ¦" << endl;
15:    cout << "~" << InputBits  << " = " << BitwiseNOT << endl;
16:
17:    cout << "Logical AND, & with 00001111" << endl;
18:    bitset<8> BitwiseAND = (0x0F & InputNum);// 0x0F is hex for 0001111
19:    cout << "0001111 & " << InputBits  << " = " << BitwiseAND << endl;
20:
21:    cout << "Logical OR, ¦ with 00001111" << endl;
22:    bitset<8> BitwiseOR = (0x0F ¦ InputNum);
23:    cout << "00001111 ¦ " << InputBits  << " = " << BitwiseOR << endl;
24:
25:    cout << "Logical XOR, ^ with 00001111" << endl;
26:    bitset<8> BitwiseXOR = (0x0F ^ InputNum);
27:    cout << "00001111 ^ " << InputBits  << " = " << BitwiseXOR << endl;
28:
29:    return 0;
30: }
```

▼ 输出：

```
Enter a number (0 - 255): 181
181 in binary is 10110101
Logical NOT ¦
~10110101 = 01001010
Logical AND, & with 00001111
0001111 & 10110101 = 00000101
Logical OR, ¦ with 00001111
00001111 ¦ 10110101 = 10111111
Logical XOR, ^ with 00001111
00001111 ^ 10110101 = 10111010
```

▼ 分析：

这个程序使用了一种还未介绍过的数据类型——bitset，旨在简化二进制数据的显示。这里使用std::bitset完全是为了方便显示，而没有其他任何目的。第10、13、17和22行将一个整数赋给了一个bitset对象，以便使用它来显示该整数的二进制表示。运算是对整数执行的。首先，请关注输出，它显示了用户输入的整数181的二进制表示，然后依次显示了将按位运算符～、&、|和^用于该整数的结果。第14行使用按位运算NOT对各位取反。这个程序还演示了运算符&、|和^的工作原理，它们对两个操作数的相应位执行相应运算，从而获得最终的结果。只要结合使用这里的结果与前面介绍的真值表，您就能明白其中的工作原理。

注意 要更深入地了解如何在C++中操作位标记，请参阅第25章，它详细介绍了std::bitset。

5.3.11 按位右移运算符（>>）和左移运算符（<<）

移位运算符将整个位序列向左或向右移动，其用途之一是将数据乘以或除以2^n。

下面的移位运算符使用示例将变量乘以2：

```
int DoubledValue = Num << 1; // shift bits one position left to double value
```

下面的移位运算符使用示例将变量除以2：

```
int HalvedValue = Num >> 2; // shift bits one position right to halve value
```

程序清单5.9演示了如何使用移位运算符将一个整数乘以或除以2^n。

程序清单5.9 使用按位右移运算符（>>）和左移运算符（<<）分别计算整数的1/4和1/2以及2倍和4倍

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "Enter a number: ";
 6:     int Input = 0;
 7:     cin >> Input;
 8:
 9:     int Half = Input >> 1;
10:     int Quarter = Input >> 2;
11:     int Double = Input << 1;
12:     int Quadruple = Input << 2;
13:
14:     cout << "Quarter: " << Quarter << endl;
15:     cout << "Half: " << Half << endl;
16:     cout << "Double: " << Double << endl;
17:     cout << "Quadruple: " << Quadruple << endl;
```

```
18:
19:     return 0;
20: }
```

▼ **输出：**

```
Enter a number: 16
Quarter: 4
Half: 8
Double: 32
Quadruple: 64
```

▼ **分析：**

输入的数字为 16，其二进制表示为 1000。第 9 行将它向右移 1 位，结果为 0100，即 8，这相当于将其减半。第 10 行向右移两位，从 1000 变成了 00100，即 4。第 11 和 12 行的左移运算符的效果完全相反。向左移 1 位时结果为 10000，即 32，向左移动两位的结果为 100000，即 64，相当于将数字分别翻了一番和两番！

注意

移位运算符不会旋转值。另外，对有符号数执行按位运算的结果随实现而异，在有些编译器上，向左移位时，并不会导致最低有效位的值变成最高有效位的值，而是将最低有效位设置为零。

5.3.12 复合赋值运算符

复合赋值运算符将运算结果赋给左边的操作数。

请看下面的代码：

```
int Num1 = 22;
int Num2 = 5;
Num1 += Num2; // Num1 contains 27 after the operation
```

其中最后一行代码与下面的代码等效：

```
Num1 = Num1 + Num2;
```

因此运算符+=的作用如下：将两个操作数相加，再将结果赋给左边的操作数（Num1）。表 5.6 列出了众多复合赋值运算符，并说明了其工作原理。

表 5.6　　合赋值运算符

运算符	用法	等效于
加法赋值运算符	Num1 += Num2;	Num1 = Num1 + Num2;
减法赋值运算符	Num1 -= Num2;	Num1 = Num1 - Num2;
乘法赋值运算符	Num1 *= Num2;	Num1 = Num1 * Num2;
除法赋值运算符	Num1 /= Num2;	Num1 = Num1 / Num2;
求模赋值运算符	Num1 %= Num2;	Num1 = Num1 % Num2;
按位左移赋值运算符	Num1 <<= Num2;	Num1 = Num1 << Num2;
按位右移赋值运算符	Num1 >>= Num2;	Num1 = Num1 >> Num2;
按位 AND 赋值运算符	Num1 &= Num2;	Num1 = Num1 & Num2;
按位 OR 赋值运算符	Num1 |= Num2;	Num1 = Num1 | Num2;
按位 XOR 赋值运算符	Num1 ^= Num2;	Num1 = Num1 ^ Num2;

程序清单 5.10 演示了这些运算符的效果。

程序清单 5.10 使用按位右移运算符（>>）和左移运算符（<<）分别计算整数的 1/4 和 1/2 以及 2 倍和 4 倍

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:   cout << "Enter a number: ";
 6:   int Value = 0;
 7:   cin >> Value;
 8:
 9:   Value += 8;
10:   cout << "After += 8, Value = " << Value << endl;
11:   Value -= 2;
12:   cout << "After -= 2, Value = " << Value << endl;
13:   Value /= 4;
14:   cout << "After /= 4, Value = " << Value << endl;
15:   Value *= 4;
16:   cout << "After *= 4, Value = " << Value << endl;
17:   Value %= 1000;
18:   cout << "After %= 1000, Value = " << Value << endl;
19:
20:   // Note: henceforth assignment happens within cout
21:   cout << "After <<= 1, value = " << (Value <<= 1) << endl;
22:   cout << "After >>= 2, value = " << (Value >>= 2) << endl;
23:
24:   cout << "After ¦= 0x55, value = " << (Value ¦= 0x55) << endl;
25:   cout << "After ^= 0x55, value = " << (Value ^= 0x55) << endl;
26:   cout << "After &= 0x0F, value = " << (Value &= 0x0F) << endl;
27:
28:   return 0;
29:}
```

▼ 输出：

```
Enter a number: 440
After += 8, Value = 448
After -= 2, Value = 446
After /= 4, Value = 111
After *= 4, Value = 444
After %= 1000, Value = 444
After <<= 1, value = 888
After >>= 2, value = 222
After ¦= 0x55, value = 223
After ^= 0x55, value = 138
After &= 0x0F, value = 10
```

▼ 分析：

在整个程序中，不断使用各种复合赋值运算符修改 value 的值。每次运算都使用了 value，并将结果赋给 value。因此，第 9 行将用户输入的值 440 加上 8，并将结果（448）赋给 value。接下来，第 11 行将 448 减去 2，并将结果（446）赋给 value。

5.3.13 使用运算符 sizeof 确定变量占用的内存量

这个运算符指出特定类型或变量占用的内存量，单位为字节。sizeof 的用法如下：

```
sizeof (variable);
```

或

```
sizeof (type);
```

注意

> sizeof(...)看起来像函数调用，但它并不是函数，而是运算符。有趣的是，程序员不能定义这个运算符，因此不能重载它。
> 第 12 章将更详细地介绍如何定义自己的运算符。

程序清单 5.11 演示了如何使用 sizeof 来确定一个数组占用的内存量。另外，您可能想查看程序清单 3.4，了解如何使用 sizeof 来确定常见变量类型占用的内存量。

程序清单 5.11　使用 sizeof 确定包含 100 个 int 元素的数组占用的内存量（单位为字节）以及每个元素占用的内存量

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "Use sizeof of determine memory occupied by arrays" << endl;
 6:     int MyNumbers [100] = {0};
 7:
 8:     cout << "Bytes occupied by an int: " << sizeof(int) << endl;
 9:     cout << "Bytes occupied by array MyNumbers: " << sizeof(MyNumbers) <<
       ➥endl;
10:     cout << "Bytes occupied by each element: " << sizeof(MyNumbers[0]) <<
       ➥endl;
11:
12:     return 0;
13: }
```

▼ 输出：

```
Use sizeof of determine memory occupied by arrays
Bytes occupied by an int: 4
Bytes occupied by array MyNumbers: 400
Bytes occupied by each element: 4
```

▼ 分析：

该程序演示了如何使用 sizeof 来确定包含 100 个 int 元素的数组占用了多少字节的内存，结果为 400 字节。该程序还表明，每个元素占用的内存为 4 字节。

在需要动态地给 N 个对象（尤其是您自己创建的类型）分配内存时，sizeof 很有用。您可以使用 sizeof 确定每个对象占用的内存量，再使用运算符 new 动态地分配内存。

动态内存分配将在第 8 章详细介绍。

5.3.14　运算符优先级

您可能在学校学过算术运算顺序口诀 BODMAS（Brackets Orders Division Multiplication Addition Subtraction，先括号，后乘除，再加减），它指出了复杂算术表达式的计算顺序。

在 C++中，假设使用运算符编写了如下表达式：

```
int MyNumber = 10 * 30 + 20 - 5 * 5 << 2;
```

MyNumber 的值是多少呢？这可没有猜测的空间，C++标准非常严格地指定了各种运算的执行顺序。这种顺序被称为运算符优先级，如表 5.7 所示。

表 5.7 运算符优先级

等级	名称	运算符
1	作用域解析运算符	::
2	成员选择、下标、函数调用、后缀递增和后缀递减	.、->、()、++、--
3	sizeof、前缀递增和递减、求补、逻辑 NOT、单目加和减、取址和解除引用、new、new[]、delete、delete[]、类型转换、sizeof()	++、--、^、!、+、-、&、()
4	用于指针的成员选择	.*、->*
5	乘、除、求模	*、/、%
6	加、减	+、-
7	移位（左移和右移）	<<、>>
8	不等关系	<、<=、>、>=
9	相等关系	==、!=
10	按位 AND	&
11	按位 XOR	^
12	按位 OR	|
13	逻辑 AND	&&
14	逻辑 OR	||
15	条件运算符	?:
16	赋值运算符	=、*=、/=、%=、+=、-=、<<=、>>=、&=、|=、^=
17	逗号运算符	,

再来看看前面的复杂表达式：

```
int MyNumber = 10 * 30 + 20 - 5 * 5 << 2;
```

计算这个表达式的结果时，需要使用表 5.7 所示的运算符优先级。由于乘法和除法的优先级高于加法和减法，而加法和减法的优先级高于移位，因此可将上述表达式简化为如下所示：

```
int MyNumber = 300 + 20 - 25 << 2;
```

由于加法和减法的优先级高于移位，因此可进一步简化为如下所示：

```
int MyNumber = 295 << 2;
```

最后，您执行移位运算。由于左移一位翻一番，左移两位翻两番，因此该表达式的结果为 295*4，即 1180。

警告

使用括号可让代码易于理解。

前述表达式实际上编写得很糟糕。对编译器来说，这个表达式很容易理解，但编写代码时，还应确保它对人来说也容易理解。

因此，将这个表达式写成下面这样要好得多：

```
int MyNumber = ((10 * 30) - (5 * 5) + 20) << 2; // leave
little room for doubt
```

应该	不应该
务必使用括号让代码和表达式易于理解。	不要编写必须依靠运算符优先级表才能理解的复杂表达式；应确保代码对人来说也易于理解。
务必使用正确的变量类型，确保它不会溢出。	不要将++Variable 与 Variable++混为一谈，以为它们等效。用于赋值时，它们的效果不同。
所有的左值（如变量）都可用作右值，但并非所有的右值都可用作左值（如“Hello World”），务必要明白这一点。	

5.4 总结

在本章中，您学习了 C++语句、表达式和运算符是什么。您学习了如何在 C++中执行加、减、乘、

除等基本的算术运算，还大致了解了 NOT、AND、OR 和 XOR 等逻辑运算。您学习了 C++逻辑运算符!、&&和||以及按位运算符～、&、|和^，前者可用于条件语句中，而后者以每次一位的方式操作数据。

您学习了运算符优先级，知道使用括号让代码对其他程序员来说易于理解很重要。您还大致了解了整型溢出，知道避免这种情形发生很重要。

5.5 问与答

问：既然 unsigned short 占用的内存更少，为何有些程序还使用 unsigned int？

答：unsigned short 变量的最大取值通常为 65535，如果这种变量的值已经是 65535，再递增将溢出，变成零。为避免这种行为，设计良好的程序在不确定变量的取值将远低于 65535 时，将其数据类型声明为 unsigned int。

问：我需要将一个数字除以 3 再翻倍，为此使用了如下代码。这些代码有问题吗？

```
int result = Number / 3 << 1;?
```

答：有问题！为何不添加括号，让这行代码对其他程序员来说更容易理解呢？添加注释也没有害处呀。

问：我的应用程序需要将整数值 5 和 2 相除，为此我编写了如下代码，但执行后，result 的值却为 2。请问有问题吗？

```
int Num1 = 5, Num2 = 2;
int result = Num1 / Num2;
```

答：没有任何问题。int 变量不能包含小数，因此这种运算的结果为 2，而不是 2.5。如果您希望结果为 2.5，应将所有变量的数据类型都改为 float 或 double。这些类型的变量用于存储浮点数（小数）。

5.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

5.6.1 测验

1. 编写将两个数相除的应用程序时，将变量声明为哪种数据类型更合适，int 还是 float？
2. 32/7 的结果是多少？
3. 32.0/7 的结果是多少？
4. sizeof(...)是函数吗？
5. 我需要将一个数翻倍，再加上 5，然后再翻倍，下面的代码是否正确？
   ```
   int Result = number << 1 + 5 << 1;
   ```
6. 如果两个操作数的值都为 true，对其执行 XOR 运算的结果是什么？

5.6.2 练习

1. 使用括号改善问题 5 中的代码，使其更清晰。
2. 下述代码导致 result 的值为多少？
   ```
   int Result = number << 1 + 5 << 1;
   ```
3. 编写一个程序，让用户输入两个布尔值，并显示对其执行各种按位运算的结果。

第 6 章
控制程序流程

大多数应用程序都需要在不同的情形（或用户输入）下以不同的方式执行。为让应用程序能够做出不同的反应，需要编写条件语句，在不同的情形下执行不同的代码片段。

在本章中，您将学习：

- 如何根据特定的条件改变程序的行为；
- 如何使用循环重复执行一系列代码；
- 如何在循环中更好地控制流程。

6.1 使用 if...else 有条件地执行

本书前面介绍的程序都按顺序从上到下执行，且执行每行代码，不跳过任何代码行。但在大多数应用程序中，很少按从上到下的顺序依次执行每行代码。

假设您要编写一个程序，在用户按 m 键时将两个数相乘，而在用户按其他任何键时将这两个数相加。

如图 6.1 所示，并非程序每次执行时，都会执行所有的代码。如果用户按 m 键，将执行将两个数相乘的代码，否则将执行相加的代码。无论在什么情形下，都不可能同时执行这两部分代码。

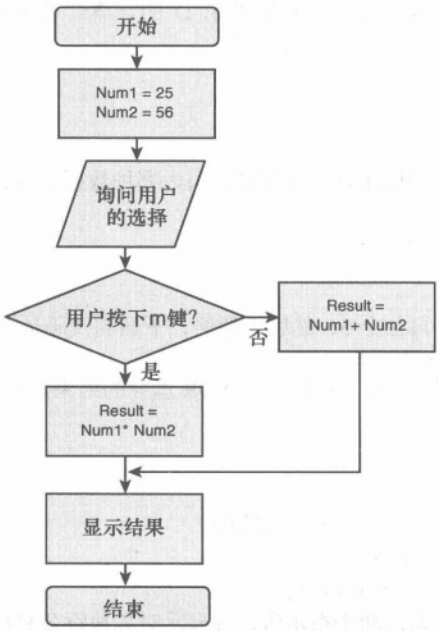


图 6.1　一个根据用户输入进行条件处理的示例

6.1.1 使用 if...else 进行条件编程

在 C++中，使用 if...else 有条件地执行代码，这种结构类似于下面这样：

```
if (conditional expression)
   Do something when expression evaluates true;
else    // Optional
   Do something else when condition evaluates false;
```

因此，在用户输入 m 时将两个数相乘，否则将两个数相加的 if...else 结构类似于下面这样：

```
if (UserSelection == 'm')
   Result = Num1 * Num2;  // multiply
else
   Result = Num1 + Num2;  // add
```

注意

> 在 C++中，表达式的结果为 true 意味着不为 false，而 false 为零。因此，在条件语句中，只要表达式的结果不为零（负数或正数），就被视为结果为 true。

程序清单 6.1 演示了 if...else 结构。它让用户决定要将两个数相乘还是相加，因此使用条件处理来生成所需的结果。

程序清单 6.1 根据用户输入决定将两个正数相乘还是相加

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Enter two integers: " << endl;
 6:    int Num1 = 0, Num2 = 0;
 7:    cin >> Num1;
 8:    cin >> Num2;
 9:
10:    cout << "Enter \'m\' to multiply, anything else to add: ";
11:    char UserSelection = '\0';
12:    cin >> UserSelection;
13:
14:    int Result = 0;
15:    if (UserSelection == 'm')
16:       Result = Num1 * Num2;
17:    else
18:       Result = Num1 + Num2;
19:
20:    cout << "Result is: " << Result << endl;
21:
22:    return 0;
23: }
```

▼ 输出：

```
Enter two integers:
25
56
Enter 'm' to multiply, anything else to add: m
Result is: 1400
```

再次运行的输出：

```
Enter two integers:
25
56
Enter 'm' to multiply, anything else to add: a
Result is: 81
```

▼ 分析：

注意到第 15 行包含 if，而第 17 行包含 else。这些代码告诉编译器，如果 if 后面的表达式（Userselection == 'm'）为 true，则执行乘法运算；如果该表达式为 false，则执行加法运算。如果用户输入的字符为 m（区分大小写）时，表达式（Userselection == 'm'）将为 true，否则将为 false。因此，这个简单的程序模拟了图 6.1 所示的流程图，演示了如何让应用程序在不同的情形下采取不同的行动。

注意

if...else 结构的 else 部分是可选的。如果在表达式为 false 时不执行任何操作，可以不使用这部分。

警告

在程序清单 6.1 中，如果第 15 行为下面这样：

```
15:    if (UserSelection == 'm');
```

这个 if 结构将毫无意义，因为它在同一行以空语句（分号）结束。务必小心避免这样做，因为 if 没有配套的 else 并不会导致编译错误。

有些优秀的编译器在“控制语句为空”时会发出警告。

6.1.2 有条件地执行多条语句

如果要在满足（或不满足）条件时执行多条语句，需要将它们组合成一个语句块。包含在大括号（{}）内的多条语句被视为语句块，例如：

```
if (condition)
{
   // condition success block
   Statement 1;
   Statement 2;
}
else
{
   // condition failure block
   Statement 3;
   Statement 4;
}
```

这样的语句块也被称为复合语句。

第 4 章解释了使用静态数组时跨越其边界带来的危险。这种问题在字符数组中最明显。将字符串写入或复制到字符数组中时，务必检查数组是否足够大，是否能够存储该字符串。程序清单 6.2 演示了如何执行这种重要的检查，以免缓冲区溢出。

程序清单 6.2 将字符串复制到 char 数组中之前，检查数组的容量

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    char Buffer[20] = {'\0'};
 7:
 8:   cout << "Enter a line of text: " << endl;
 9:   string LineEntered;
10:   getline (cin, LineEntered);
11:
12:   if (LineEntered.length() < 20)
13:   {
14:       strcpy(Buffer, LineEntered.c_str());
```

```
15:       cout << "Buffer contains: " << Buffer << endl;
16:   }
17:
18:   return 0;
19: }
```

▼ **输出：**

```
Enter a line of text:
This fits buffer!
Buffer contains: This fits buffer!
```

▼ **分析：**

注意到将字符串复制到缓冲区中前，第 12 行检查字符串是否比缓冲区短。另外，这条 if 语句的特殊之处在于，如果条件为 true，将执行第 13～16 行的语句块（也叫复合语句）。

警告

注意到 if(condition)的行尾没有分号。这是有意为之，旨在确保条件为 true 时将执行 if 后面的语句。

下面的语句能够通过编译，但得不到所需的结果，因为 if 子句后面的分号导致它到此结束，这意味着没有条件处理，后面的语句总是会执行。

```
if(condition);
   statement;
```

6.1.3 嵌套 if 语句

经常需要检查一系列不同的条件，且很多条件依赖于前一个条件是否满足。为满足这种需求，C++允许您对 if 语句进行嵌套。

嵌套 if 语句类似于下面这样：

```
if (expression1)
{
   DoSomething1;
   if(expression2)
      DoSomething2;
   else
      DoSomethingElse2;
}
else
    DoSomethingElse1;
```

假设有一个类似于程序清单 6.1 所示的应用程序，用户可通过按 d 或 m 键，让应用程序执行除法或乘法运算。执行除法运算前，必须核实除数不为零。因此，除检查用户输入外，在用户要求程序执行除法运算时，还必须核实除数不为零。为此，可使用嵌套 if 语句，如程序清单 6.3 所示。

程序清单 6.3 使用嵌套 if 语句执行乘法或除法运算

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Enter two numbers: " << endl;
 6:    float Num1 = 0, Num2 = 0;
 7:    cin >> Num1;
 8:    cin >> Num2;
 9:
10:    cout << "Enter 'd' to divide, anything else to multiply: ";
```

```
11:    char UserSelection = '\0';
12:    cin >> UserSelection;
13:
14:    if (UserSelection == 'd')
15:    {
16:       cout << "You want division!" << endl;
17:       if (Num2 != 0)
18:      {
19:          cout << "No div-by-zero, proceeding to calculate" << endl;
20:        cout << Num1 << " / " << Num2 << " = " << Num1 / Num2 << endl;
21:      }
22:       else
23:          cout << "Division by zero is not allowed" << endl;
24:    }
25:    else
26:    {
27:       cout << "You want multiplication!" << endl;
28:      cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
29:    }
30:
31:    return 0;
32: }
```

▼ 输出：

```
Enter two numbers:
45
9
Enter 'd' to divide, anything else to multiply: m
You want multiplication!
45 x 9 = 405
```

再次运行的输出：

```
Enter two numbers:
22
7
Enter 'd' to divide, anything else to multiply: d
You want division!
No div-by-zero, proceeding to calculate
22 / 7 = 3.14286
```

最后一次运行的输出：

```
Enter two numbers:
365
0
Enter 'd' to divide, anything else to multiply: d
You want division!
Division by zero is not allowed
```

▼ 分析：

这是运行程序三次得到的输出，每次提供的输入都不同。正如您看到的，程序每次的执行路径都不同。相比于程序清单 6.1，这个程序有很多地方不同。

- 为更好地处理小数，将输入存储到了 float 变量中，执行除法运算时这很重要。
- if 条件与程序清单 6.1 中不同，不再检查用户按的是否是 m 键，而在第 14 行使用了表达式 (UserSelection == 'd')，该表达式在用户输入 d 时为 true。如果用户输入了 d，则执行除法运算。
- 鉴于这个程序将两个数相除，且除数由用户输入，因此必须核实除数不为零。这是在第 17 行使用嵌套的 if 语句实现的。

需要根据多个条件执行不同任务时，嵌套 if 语句很有用，这个程序演示了这一点。

提示

这里使用制表符对嵌套语句进行了缩进，这是可选的，但可极大地改善嵌套 if 语句的可读性。

也可组合使用多个 if...else 结构。程序清单 6.4 所示的程序让用户输入星期几，并使用一组 if...else 结构告诉用户它是以哪个星星命名的。

程序清单 6.4 指出一个星期的各天是以哪个星星命名的

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

int main()
{
   enum DaysOfWeek
   {
      Sunday = 0,
      Monday,
      Tuesday,
      Wednesday,
      Thursday,
      Friday,
      Saturday
   };

   cout << "Find what days of the week are named after!" << endl;
   cout << "Enter a number for a day (Sunday = 0): ";

   int Day = Sunday;   // Initialize to Sunday
   cin >> Day;

   if (Day == Sunday)
      cout << "Sunday was named after the Sun" << endl;
   else if (Day == Monday)
     cout << "Monday was named after the Moon" << endl;
   else if (Day == Tuesday)
      cout << "Tuesday was named after Mars" << endl;
   else if (Day == Wednesday)
      cout << "Wednesday was named after Mercury" << endl;
   else if (Day == Thursday)
     cout << "Thursday was named after Jupiter" << endl;
   else if (Day == Friday)
      cout << "Friday was named after Venus" << endl;
   else if (Day == Saturday)
      cout << "Saturday was named after Saturn" << endl;
   else
      cout << "Wrong input, execute again" << endl;

   return 0;
}
```

▼ 输出：

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 5
Friday was named after Venus
```

再次运行的输出：

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 9
Wrong input, execute again
```

▼ 分析：

第 22～37 行的 if-else-if 结构检查用户输入并生成相应的输出。第二次运行的输出表明，如果用户输入的不是 0～6，即不对应于一个星期的任何一天，程序将指出这一点。这种结构的优点是，非常适合用于检查互斥的条件，即星期一不可能是星期二，而无效输入不与一个星期的任何一天对应。另一个有趣的地方是，在 if 语句中使用了第 5 行声明的枚举常量 DaysOfWeek。原本可以将用户输入与整数（如 0 表示星期天等）进行比较，但通过使用枚举常量 Sunday，代码的可读性更强。

6.1.4 使用 switch-case 进行条件处理

switch-case 让您能够将特定表达式与一系列常量进行比较，并根据表达式的值时执行不同的操作。在这种结构中，经常会使用 C++新增的关键字 switch、case、default 和 break。

switch-case 结构的语法如下：

```
switch(expression)
{
case LabelA:
   DoSomething;
   break;

case LabelB:

   DoSomethingElse;
   break;

// And so on…
default:
   DoStuffWhenExpressionIsNotHandledAbove;
   break;
}
```

上述代码计算 expression 的值，并将其与每个 case 标签进行比较。每个 case 标签都必须是常量，如果 expression 的值与 case 标签相等，就执行标签后面的代码。如果 expression 与 LabelA 不相等，将把 expression 与 LabelB 进行比较。如果它与 LabelB 相同，就执行 DoSomethingElse。将不断重复这个过程，直到遇到 break。这是您首次见到 break，它导致程序退出当前代码块。break 并非必不可少，但如果省略它，将不断与后面的标签进行比较，这并非您希望的。default 也是可选的，它用于执行 expression 不与 switch-case 中的任何标签匹配时应执行的操作。

提示

switch-case 结构非常适合与枚举常量结合使用。关键字 enum 在第 3 章介绍过。

程序清单 6.5 使用了 switch-case 结构，它与程序清单 6.4 等效，指出一个星期的各天是以哪个星星命名的，也使用了枚举常量。

程序清单 6.5 指出一个星期的各天是以哪个星星命名的

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    enum DaysOfWeek
 6:    {
 7:       Sunday = 0,
```

```
 8:       Monday,
 9:       Tuesday,
10:       Wednesday,
11:       Thursday,
12:       Friday,
13:       Saturday
14:    };
15:
16:    cout << "Find what days of the week are named after!" << endl;
17:    cout << "Enter a number for a day (Sunday = 0): ";
18:
19:    int Day = Sunday;   // Initialize to Sunday
20:    cin >> Day;
21:
22:    switch(Day)
23:    {
24:    case Sunday:
25:       cout << "Sunday was named after the Sun" << endl;
26:       break;
27:
28:    case Monday:
29:       cout << "Monday was named after the Moon" << endl;
30:       break;
31:
32:    case Tuesday:
33:       cout << "Tuesday was named after Mars" << endl;
34:       break;
35:
36:    case Wednesday:
37:       cout << "Wednesday was named after Mercury" << endl;
38:       break;
39:
40:    case Thursday:
41:       cout << "Thursday was named after Jupiter" << endl;
42:       break;
43:
44:    case Friday:
45:       cout << "Friday was named after Venus" << endl;
46:       break;
47:
48:    case Saturday:
49:       cout << "Saturday was named after Saturn" << endl;
50:       break;
51:
52:    default:
53:       cout << "Wrong input, execute again" << endl;
54:       break;
55:    }
56:
57:    return 0;
58: }
```

▼ 输出：

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 5
Friday was named after Venus
```

再次运行的输出：

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 9
Wrong input, execute again
```

▼ 分析：

第22～55行的switch-case结构根据用户输入的整数（存储在变量Day中）生成不同的输出。用户输入数字5时，应用程序将switch表达式（Day，其值为5）与标签进行比较，并跳过前4个标签后面的代码，因为这些标签为Sunday（0）～Thursday（4），它们都与5不相等。到达标签Friday后，由于switch表达式的值（5）与枚举常量Friday相等，因此执行该标签后面的代码，并在到达break后退出swtich结构。第二次运行时提供的值无效，因此到达default后执行它后面的代码，显示一条让用户再执行一次的消息。

这个程序使用的是switch-case，其输出与使用if-else-if结构的程序清单6.4相同。然而，switch-case版本的结构化程度更高，可能非常适合不仅仅将一行文本显示在屏幕上的情形（在这种情形下，可将代码放在大括号内，组成语句块）。

6.1.5 使用运算符?：进行条件处理

C++提供了一个有趣且功能强大的运算符——条件运算符，它相当于紧凑的if-else结构。

条件运算符也叫三目运算符，因为它使用三个操作数：

```
(conditional expression evaluated to bool) ? expression1 if true : expression2
if false;
```

可使用这个运算符获得两个数字中较大的那个：

```
int Max = (Num1 > Num2)? Num1 : Num2; // Max contains greater of Num1 and Num2
```

程序清单6.6演示了如何使用运算符?:进行条件处理。

程序清单6.6 指出一个星期的各天是以哪个星星命名的

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "Enter two numbers" << endl;
 6:     int Num1 = 0, Num2 = 0;
 7:     cin >> Num1;
 8:     cin >> Num2;
 9:
10:     int Max = (Num1 > Num2)? Num1 : Num2;
11:     cout << "The greater of " << Num1 << " and " \
12:         << Num2 << " is: " << Max << endl;
13:
14:     return 0;
15: }
```

▼ 输出：

```
Enter two numbers
365
-1
The greater of 365 and -1 is: 365
```

▼ 分析：

需要注意的是第10行。它包含一条非常紧凑的语句，该语句判断输入的两个数字那个更大，与下述使用if-else的代码等效：

```
int Max = 0;
if (Num1 > Num2)
   Max = Num1;
else
   Max = Num2;
```

使用条件运算符可节省几行代码！但不应将节省代码放在首位。有些程序员很喜欢条件运算符，而有些不喜欢。使用条件运算符时，确保代码易于理解至关重要。

应该	不应该
务必将常量或枚举用作 case 标签，以提高代码的可读性。 务必提供 default 标签，除非完全没有必要。 务必在每条 case 语句中包含 break。	不要包含两个标签相同的 case 语句，这既不符合逻辑，也无法通过编译。 务必不要使用没有 break 的 case 语句，也不要依赖于 case 语句的顺序，这会让 swtich-case 结构过于复杂。另外，如果以后不小心调整了 case 语句的顺序，代码可能不再可行。 使用条件运算符（?:）时，不要使用复杂的条件和表达式。

6.2 在循环中执行代码

至此，您知道如何让程序在变量包含不同的值时执行不同的操作。例如，当用户按 m 键时，程序清单 6.2 执行乘法运算，否则执行加法运算。然而，如果用户不希望程序就此结束，而要再执行一次（甚至 5 次）乘法或加法运算，该如何办呢？在这种情况下，您需要重复执行现有的代码。

为此，您需要使用循环。

6.2.1 不成熟的 goto 循环

顾名思义，goto 将指令指针移到代码的特定位置，您可使用它回过头去再次执行特定的语句。

goto 语句的语法如下：

```
SomeFunction()
{
JumpToPoint: // Called a label
   CodeThatRepeats;

   goto JumpToPoint;
}
```

这里声明了一个名为 JumpToPoint 的标签，并使用 goto 跳转到这个地方，如程序清单 6.7 所示。除非给 goto 语句指定在特定情况下将为 false 的执行条件，或者重复执行的代码中包含在特定条件下将被执行的 return 语句，否则 goto 命令和标签之间的代码将无休止地执行下去，导致程序永不结束。

程序清单 6.7 使用 goto 语句询问用户是否想重复计算

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5: JumpToPoint:
 6:    int Num1 = 0, Num2 = 0;
 7:
 8:    cout << "Enter two integers: " << endl;
 9:    cin >> Num1;
10:    cin >> Num2;
11:
12:    cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
```

```
13:     cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
14:
15:     cout << "Do you wish to perform another operation (y/n)?" << endl;
16:     char Repeat = 'y';
17:     cin >> Repeat;
18:
19:     if (Repeat == 'y')
20:         goto JumpToPoint;
21:
22:     cout << "Goodbye!" << endl;
23:
24:     return 0;
25: }
```

▼ **输出：**

```
Enter two integers:
56
25
56 x 25 = 1400
56 + 25 = 81
Do you wish to perform another operation (y/n)?
y
Enter two integers:
95
-47
95 x -47 = -4465
95 + -47 = 48
Do you wish to perform another operation (y/n)?
n
Goodbye!
```

▼ **分析：**

程序清单 6.7 和程序清单 6.1 的主要差别在于，要让用户再次输入一组数字，并查看加法或乘法运算的结果，需要再次运行程序清单 6.1，而程序清单 6.7 不需要这样，它询问用户是否想再执行一次运算。实际实现这种重复的代码位于第 20 行，它在用户输入表示 yes 的 y 时执行 goto 语句。执行第 20 行的 goto 语句将导致程序跳转到第 5 行声明的标签 JumpToPoint 处，这相当于重新启动程序。

警告

不推荐使用 goto 语句来编写循环，因为大量使用 goto 语句将导致代码的执行流程无法预测，即不按特定的顺序从一行跳转到另一行；在有些情况下，也可能导致变量的状态无法预测。糟糕地使用 goto 语句将导致意大利面条式代码。要避免使用 goto 语句，可使用接下来将介绍的 while、do...while 和 for 循环。

这里介绍 goto 语句只是为了帮助您理解使用这种语句的代码。

6.2.2 while 循环

C++关键字 while 可帮助您完成程序清单 6.7 中 goto 语句完成的工作，但更优雅。while 循环的语法如下：

```
while(expression)
{
   // Condition evaluates to true
   StatementBlock;
}
```

只要 expression 为 true，就将执行该语句块。因此，必须确保 expression 在特定条件下将为 false，否则 while 循环将永不停止。

程序清单 6.8 与程序清单 6.7 等效，但使用 while 而不是 goto 让用户能够重复计算。

程序清单 6.8 使用 while 循环让用户能够重复计算

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    char UserSelection = 'm';   // initial value
 6:
 7:    while (UserSelection != 'x')
 8:    {
 9:       cout << "Enter the two integers: " << endl;
10:       int Num1 = 0, Num2 = 0;
11:       cin >> Num1;
12:       cin >> Num2;
13:
14:       cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
15:       cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
16:
17:       cout << "Press x to exit(x) or any other key to recalculate" << endl;
18:       cin >> UserSelection;
19:    }
20:
21:    cout << "Goodbye!" << endl;
22:
23:    return 0;
24: }
```

▼ 输出：

```
Enter the two integers:
56
25
56 x 25 = 1400
56 + 25 = 81
Press x to exit(x) or any other key to recalculate
r
Enter the two integers:
365
-5
365 x -5 = -1825
365 + -5 = 360
Press x to exit(x) or any other key to recalculate
x
Goodbye!
```

▼ 分析：

第 7～19 行的 while 循环包含了该程序的大部分逻辑。while 循环检查表达式(UserSelection != 'x')，仅当该表达式为 true 时才继续执行后面的代码。为确保第一次循环能够进行，第 5 行将 char 变量 UserSelection 初始化为'm'。需要确保该变量不为'x'，否则将导致第一次循环不会进行，应用程序退出，而不做任何有意义的工作。第一次循环非常简单，但第 17 行询问用户是否想再次执行计算。第 18 行读取用户输入，这将影响 while 计算的表达式的结果，确定程序继续执行还是就此终止。第一次循环结束后，将跳转到第 7 行，计算 while 语句中表达式的值，如果用户按的不是 x 键，将再次执行循环。如果用户在循环末尾按了 x 键，下次计算第 7 行的表达式时，结果将为 false，这将退出 while 循环，并在显示再见消息后结束应用程序。

注意

循环也叫迭代，while、do...while 和 for 语句也被称为迭代语句。

6.2.3 do...while 循环

在有些情况（如程序清单 6.8 所示的情况）下，您需要将代码放在循环中，并确保它们至少执行一次。此时 do...while 循环可派上用场。

do...while 循环的语法如下：

```
do
{
   StatementBlock; // executed at least once
} while(condition); // ends loop if condition evaluates to false
```

注意到包含 while(expression)的代码行以分号结尾，这不同于前面介绍的 while 循环。在 while 循环中，如果包含 while(expression)的代码行以分号结尾，循环将就此结束，变成一条空语句。

程序清单 6.9 演示了如何使用 do...while 循环来确保语句至少执行一次。

程序清单 6.9 使用 do...while 循环重复执行代码块

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    char UserSelection = 'x';   // initial value
 6:    do
 7:    {
 8:       cout << "Enter the two integers: " << endl;
 9:       int Num1 = 0, Num2 = 0;
10:       cin >> Num1;
11:       cin >> Num2;
12:
13:       cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
14:       cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
15:
16:       cout << "Press x to exit(x) or any other key to recalculate" << endl;
17:       cin >> UserSelection;
18:    } while (UserSelection != 'x');
19:
20:    cout << "Goodbye!" << endl;
21:
22:    return 0;
23: }
```

▼ 输出：

```
Enter the two integers:
654
-25
654 x -25 = -16350
654 + -25 = 629
Press x to exit(x) or any other key to recalculate
m
Enter the two integers:
909
101
909 x 101 = 91809
909 + 101 = 1010
Press x to exit(x) or any other key to recalculate
x
Goodbye!
```

▼ 分析：

这个程序的行为和输出与前一个程序很像。实际上，唯一的差别在于，第 6 行包含关键字 do，而第 18 行使用了 while。将按顺序执行每行代码，直到达到第 18 行的 while。到达第 18 行后，while 计算表达式(UserSelection != 'x')的值。如果该表达式为 true，即用户没有按 x 退出，将重复执行循环。如果该表达式为 false，即用户按了 x 键，将退出循环，显示再见消息，并结束应用程序。

6.2.4 for 循环

for 语句是一种更复杂的循环，因为它允许您指定执行一次的初始化语句（通常用于初始化计数器）、检查退出条件（通常使用计数器）并在每次循环末尾执行操作（通常是将计数器递增或修改其值）。

for 循环的语法如下：

```
for (initial expression executed only once;
     exit condition executed at the beginning of every loop;
     loop expression executed at the end of every loop)
{
    DoSomeActivities;
}
```

for 循环让程序员能够定义并初始化一个计数器变量，在每次循环开头检查退出条件，在循环末尾修改计数器变量的值。

程序清单 6.10 演示了一种使用 for 循环访问数组元素的高效方式。

程序清单 6.10 使用 for 循环填充和显示静态数组的元素

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int ARRAY_LENGTH = 5;
 6:    int MyInts[ARRAY_LENGTH] = {0};
 7:
 8:    cout << "Populate array of " << ARRAY_LENGTH << " integers" << endl;
 9:
10:    for (int ArrayIndex = 0; ArrayIndex < ARRAY_LENGTH; ++ArrayIndex)
11:    {
12:       cout << "Enter an integer for element " << ArrayIndex << ": ";
13:       cin >> MyInts[ArrayIndex];
14:    }
15:
16:    cout << "Displaying contents of the array: " << endl;
17:
18:    for (int ArrayIndex = 0; ArrayIndex < ARRAY_LENGTH; ++ArrayIndex)
19:       cout << "Element " << ArrayIndex << " = " << MyInts[ArrayIndex] <<
endl;
20:
21:    return 0;
22: }
```

▼ 输出：

```
Populate array of 5 integers
Enter an integer for element 0: 365
Enter an integer for element 1: 31
```

```
Enter an integer for element 2: 24
Enter an integer for element 3: -59
Enter an integer for element 4: 65536
Displaying contents of the array:
Element 0 = 365
Element 1 = 31
Element 2 = 24
Element 3 = -59
Element 4 = 65536
```

▼ 分析：

程序清单 6.10 包含两个 for 循环，分别位于第 10 和 18 行。第一个 for 循环帮助填充一个 int 数组的元素，而另一个帮助显示该数组的元素。这两个 for 循环的语法相同，都声明了索引变量 ArrayIndex，用于访问数组中的元素。在每次循环末尾，这个变量都递增，以便在下一次循环时访问数组中的下一个元素。在 for 语句中，中间的表达式为退出条件，它将 ArrayIndex 与 ARRAY_LENGTH 进行比较，检查在每次循环末尾递增后，ArrayIndex 是否还在数组边界内。这也确保了 for 循环不会跨越数组边界。

注意

用于帮助访问集合（如数组）元素的变量（如程序清单 6.10 中的 ArrayIndex）也被称为迭代器。

在 for 语句中声明的迭代器的作用域为 for 循环，因此在程序清单 6.10 中，第二个 for 循环中声明的 ArrayIndex 实际上是个新变量。

然而，初始化语句、条件表达式以及修改变量的语句都是可选的，for 语句可以不包含这些部分，如程序清单 6.11 所示。

程序清单 6.11 使用 for 循环（省略了修改变量的语句）根据用户的请求重复执行计算

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     // without loop expression (third expression missing)
 6:     for(char UserSelection = 'm'; (UserSelection != 'x'); )
 7:    {
 8:         cout << "Enter the two integers: " << endl;
 9:         int Num1 = 0, Num2 = 0;
10:         cin >> Num1;
11:         cin >> Num2;
12:
13:         cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
14:         cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
15:
16:         cout << "Press x to exit or any other key to recalculate" << endl;
17:         cin >> UserSelection;
18:     }
19:
20:     cout << "Goodbye!" << endl;
21:
22:     return 0;
23: }
```

▼ 输出：

```
Enter the two integers:
56
25
```

```
56 x 25 = 1400
56 + 25 = 81
Press x to exit or any other key to recalculate
m
Enter the two integers:
789
-36
789 x -36 = -28404
789 + -36 = 753
Press x to exit or any other key to recalculate
x
Goodbye!
```

▼ 分析：

这与使用 while 循环的程序清单 6.8 相同，唯一的差别在于第 8 行使用了 for 循环。这个 for 循环有趣的地方在于，它只包含初始化表达式和条件表达式，而省略了在每次循环末尾修改变量的语句。

注意

在 for 循环的初始化表达式中，可初始化多个变量。对于程序清单 6.11 所示的 for 循环，如果在其中初始化多个变量，将类似于下面这样：

```
for (int Index = 0, AnotherInt = 5; Index < ARRAY_LENGTH;
➥++Index, --AnotherInt)
```

注意到新增的变量被初始化为 5。

有趣的是，还可使用循环表达式在每次循环时都将其递减。

6.3 使用 continue 和 break 修改循环的行为

在有些情况下（尤其是使用大量条件处理大量参数的复杂循环中），无法编写有效的循环条件，而需要在循环中修改程序的行为。在这种情况下，contiune 和 break 可提供帮助。

continue 让您能够跳转到循环开头，跳过循环块中后面的代码。因此，在 while、do...while 和 for 循环中，continue 导致重新评估循环条件，如果为 true，则重新进入循环块。

注意

在 for 循环中遇到 continue 时，将在评估条件前执行循环表达式（for 语句中的第三个表达式，通常用于递增计数器）。

break 退出循环块，即结束当前循环。

警告

程序员通常的预期是，如果循环条件满足，将执行循环中的所有代码。contiune 和 break 改变了这种行为，可能导致代码不直观。

因此，应慎用 continue 和 break，仅当不使用它们就无法正确而高效地编写循环时才用。

6.3.1 不结束的循环——无限循环

while、do...while 和 for 循环都包含一个条件表达式，循环在它为 false 时结束。如果您指定的条件总是为 true，循环就不会结束。

无限 while 循环类似于下面这样：

```
while(true)   // while expression fixed to true
{
   DoSomethingRepeatedly;
}
```

无限 do…while 循环类似于下面这样：

```
do
{
   DoSomethingRepeatedly;
} while(true);   // do…while expression never evaluates to false
```

而无限 for 循环类似于下面这样：

```
for (;;)   // no condition supplied = unending for
{
   DoSomethingRepeatedly;
}
```

这种循环看似奇怪，但确实有用武之地。假设操作系统需要不断检查 USB 端口是否连接了设备，只要操作系统在运行，这种活动就不应停止。在这种情况下，就应使用永不结束的循环。这种循环也叫无限循环，因为它们将不断执行下去，直到永远。

6.3.2 控制无限循环

如果要结束无限循环（假设前述示例中的操作系统需要关闭），可插入一条 break 语句（通常放在 if(condition)代码块中）。

下面是一个使用 break 退出无限 while 循环的例子：

```
while(true)   // while expression fixed to true
{
   DoSomethingRepeatedly;
   if(expression)
      break;  // exit loop when expression evaluates to true
}
```

下面是一个使用 break 退出无限 do…while 循环的例子：

```
do
{
   DoSomethingRepeatedly;
   if(expression)
      break;  // exit loop when expression evaluates to true
} while(true);   // do…while expression never evaluates to false
```

下面是一个使用 break 退出无限 for 循环的例子：

```
for (;;)   // no condition supplied = unending for
{
   DoSomethingRepeatedly;
   if(expression)
      break;  // exit loop when expression evaluates to true
}
```

程序清单 6.12 演示了如何使用 continue 和 break 来指定无限循环的退出条件。

程序清单 6.12 使用 continue 进入下一次循环，并使用 break 退出无限 for 循环

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    for(;;)   // an infinite loop
 6:    {
 7:       cout << "Enter two integers: " << endl;
 8:       int Num1 = 0, Num2 = 0;
 9:       cin >> Num1;
10:       cin >> Num2;
11:
```

```
12:       cout << "Do you wish to correct the numbers? (y/n): ";
13:       char ChangeNumbers = '\0';
14:       cin >> ChangeNumbers;
15:
16:       if (ChangeNumbers == 'y')
17:          continue;  // restart the loop!
18:
19:       cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
20:       cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
21:
22:       cout << "Press x to exit or any other key to recalculate" << endl;
23:       char UserSelection = '\0';
24:       cin >> UserSelection;
25:
26:       if (UserSelection == 'x')
27:          break;   // exit the infinite loop
28:    }
29:
30:    cout << "Goodbye!" << endl;
31:
32:    return 0;
33: }
```

▼ 输出：

```
Enter two integers:
560
25
Do you wish to correct the numbers? (y/n): y
Enter two integers:
56
25
Do you wish to correct the numbers? (y/n): n
56 x 25 = 1400
56 + 25 = 81
Press x to exit or any other key to recalculate
r
Enter two integers:
95
-1
Do you wish to correct the numbers? (y/n): n
95 x -1 = -95
95 + -1 = 94
Press x to exit or any other key to recalculate
x
Goodbye!
```

▼ 分析：

相比于程序清单 6.11 的 for 循环，第 5 行的 for 循环的不同之处在于，它是一个无限 for 循环，没有包含每次循环迭代前评估的条件表达式。换句话说，如果没有 break 语句，该循环（进而是该应用程序）将永远不会结束。相比于您在本书前面看到的其他输出，这里的输出的不同之处在于，在执行加法和乘法运算前，用户可修改其输入的整数。这种逻辑是使用 continue 实现的；如第 16 和 17 行所示，程序根据指定的条件决定是否执行 continue。被询问是否要修改数字时，如果用户按 y 键，第 16 行的条件将为 true，进行执行后面的 continue。遇到 continue 后，将跳转到循环开头处执行，让用户输入两个整数。同样，在循环末尾询问用户是否想退出时，第 26 行检查用户输入是否是'x'，如果是，则执行 break 语句，结束循环。

注意

在程序清单 6.12 中，使用了空语句 for(;;)来创建无限循环。您也可以使用其他类型的循环来生成相同的输出，为此可将该语句替换为 while(true)或 do…while(true);。

应该	不应该
在循环逻辑至少需要执行一次时，务必使用 do...while 循环。 使用 while、do...while 或 for 循环时，务必指定明确的条件表达式。 在循环包含的语句块中，务必缩进代码以提高可读性。	不要使用 goto 语句。 不要不分青红皂白地使用 continue 和 break。 除非万不得已，否则不要编写使用 break 的无限循环。

6.4 编写嵌套循环

就像本章开头介绍的嵌套 if 语句一样，经常需要在一个循环内嵌套另一个循环。假设有两个 int 数组，而您想将 Array1 的每个元素与 Array2 的每个元素相乘，则通过使用嵌套循环，这种编程工作将很容易完成。第一个循环遍历 Array1，第二个循环遍历 Array2，且位于第一个循环内部。

程序清单 6.13 表明，您可在任何类型的循环内部嵌套任何类型的循环。

程序清单 6.13 使用嵌套循环将一个数组的每个元素与另一个数组的每个元素相乘

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int ARRAY1_LEN = 3;
 6:    const int ARRAY2_LEN = 2;
 7:
 8:    int MyInts1[ARRAY1_LEN] = {35, -3, 0};
 9:    int MyInts2[ARRAY2_LEN] = {20, -1};
10:
11:    cout << "Multiplying each int in MyInts1 by each in MyInts2:" << endl;
12:
13:    for(int Array1Index = 0; Array1Index < ARRAY1_LEN; ++Array1Index)
14:       for(int Array2Index = 0; Array2Index < ARRAY2_LEN; ++Array2Index)
15:          cout << MyInts1[Array1Index] << " x " << MyInts2[Array2Index] \
16:          << " = " << MyInts1[Array1Index] * MyInts2[Array2Index] << endl;
17:
18:    return 0;
19: }
```

▼ 输出：

```
Multiplying each int in MyInts1 by each in MyInts2:
35 x 20 = 700
35 x -1 = -35
-3 x 20 = -60
-3 x -1 = 3
0 x 20 = 0
0 x -1 = 0
```

▼ 分析：

第 13 和 14 行是两个嵌套的 for 循环。第一个 for 循环遍历数组 MyInts1，而第二个 for 循环遍历数组 MyInts2。第一个 for 循环每次迭代时，都执行第二个 for 循环。第二个 for 循环遍历数组 MyInts2，在每次迭代中，都将数组 MyInts2 中的当前元素与数组 MyInts1 中索引为 Array1Index 的元素相乘。因

此，对于 MyInts1 中的每个元素，第二个循环都遍历 MyInts2 的所有元素。其结果是，首先将 MyInts1 的第一个元素（偏移量为 0）与数组 MyInts2 的每个元素相乘；然后，将 MyInts1 的第二个元素与数组 MyInts2 的每个元素相乘；最后，将 MyInts1 的第三个元素与数组 MyInts2 的每个元素相乘。

注意

出于方便以及将重点放在循环上的考虑，在程序清单 6.13 中，对数组进行了初始化。但也可以前面的示例（如程序清单 6.10）那样，让用户输入整数，并使用它们来填充 int 数组。

6.4.1 使用嵌套循环遍历多维数组

第 4 章介绍了多维数组。实际上，在程序清单 4.3 中，您遍历了一个三行、三列的二维数组。当时的做法是，分别访问数组中的每个元素，每个元素占一行代码。这种方式的可扩展性不强，如果数组变大了，除修改各维的长度外，还需添加大量的代码。然而，使用循环可改变这一点，如程序清单 6.14 所示。

程序清单 6.14 使用嵌套循环遍历二维 int 数组的元素

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int MAX_ROWS = 3;
 6:    const int MAX_COLS = 4;
 7:
 8:    // 2D array of integers
 9:    int MyInts[MAX_ROWS][MAX_COLS] = { {34, -1, 879, 22},
10:                                       {24, 365, -101, -1},
11:                                       {-20, 40, 90, 97} };
12:
13:    // iterate rows, each array of int
14:    for (int Row = 0; Row < MAX_ROWS; ++Row)
15:    {
16:       // iterate integers in each row (columns)
17:       for (int Column = 0; Column < MAX_COLS; ++Column)
18:       {
19:          cout << "Integer[" << Row << "][" << Column \
20:               << "] = " << MyInts[Row][Column] << endl;
21:       }
22:    }
23:
24:    return 0;
25: }
```

▼ 输出：

```
Integer[0][0] = 34
Integer[0][1] = -1
Integer[0][2] = 879
Integer[0][3] = 22
Integer[1][0] = 24
Integer[1][1] = 365
Integer[1][2] = -101
Integer[1][3] = -1
Integer[2][0] = -20
Integer[2][1] = 40
Integer[2][2] = 90
Integer[2][3] = 97
```

▼ 分析：

第 14～22 行包含遍历二维 int 数组所需的两个 for 循环。二维数组实际上是数组的数组。注意到第一个 for 循环访问行，每行都是一个 int 数组；而第二个 for 循环访问数组中的每个元素，即列。

注意

程序清单 6.14 使用大括号将嵌套的 for 循环括起来了，这只是为了改善可读性。即便没有大括号，嵌套的循环也不会有问题，因为循环语句只是一条语句，而不是复合语句，需要必须用大括号括起。

6.4.2 使用嵌套循环计算斐波纳契数列

著名的斐波纳契数列以 0 和 1 打头，随后的每个数字都是前两个数字之和。因此斐波纳契数列的开头类似于下面这样：

0, 1, 1, 2, 3, 5, 8, …

程序清单 6.15 演示了如何创建斐波纳契数列：用户想要多长就多长——长度只受限于 int 变量可存储的最大值。

程序清单 6.15 使用嵌套循环计算斐波纳契数列

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int NumsToCal = 5;
 6:    cout << "This program will calculate " << NumsToCal \
 7:         << " Fibonacci Numbers at a time" << endl;
 8:
 9:    int Num1 = 0, Num2 = 1;
10:    char WantMore = '\0';
11:    cout << Num1 << " " << Num2 << " ";
12:
13:    do
14:    {
15:       for (int Index = 0; Index < NumsToCal; ++Index)
16:       {
17:          cout << Num1 + Num2 << " ";
18:
19:          int Num2Temp = Num2;
20:          Num2 = Num1 + Num2;
21:          Num1 = Num2Temp;
22:       }
23:
24:       cout << endl << "Do you want more numbers (y/n)? ";
25:       cin >> WantMore;
26:    }while (WantMore == 'y');
27:
28:    cout << "Goodbye!" << endl;
29:
30:    return 0;
31: }
```

▼ 输出：

```
This program will calculate 5 Fibonacci Numbers at a time
0 1 1 2 3 5 8
Do you want more numbers (y/n)? y
```

```
13 21 34 55 89
Do you want more numbers (y/n)? y
144 233 377 610 987
Do you want more numbers (y/n)? y
1597 2584 4181 6765 10946
Do you want more numbers (y/n)? n
Goodbye!
```

▼ 分析：

第 13 行的外部 do...while 循环基本上是一个询问循环，询问用户是否要生成更多的数字。第 15 行的内部 for 循环计算并显示接下来的 5 个斐波纳契数。第 19 行将 Num2 赋给一个临时变量，以便第 21 行将这个值赋给 Num1。如果不将 Num2 的原始值存储到临时变量中，第 20 行修改 Num2 的值后，就无法将其赋给 Num1 了。如果用户按 y 键，将使用 Num1 和 Num2 存储的新值再次执行内部 for 循环，这都要归功于这 3 行代码。

6.5 总结

本章介绍的全部内容都旨在避免代码只能从上到下执行：如何编写可提供不同执行路径的条件语句以及如何使用循环重复执行代码。您学习了 if...else 结构以及如何使用 switch-case 语句来处理不同的情形（即变量包含不同的值）。

为帮助您理解循环，介绍了 goto 语句，但同时警告您不要使用它，因为使用它创建的代码难以理解。您学习了如何使用 while、do...while 和 for 结构编写循环，学习了如何让循环无休止地迭代下去（即无限循环）以及如何使用 continue 和 break 更好地控制它们。

6.6 问与答

问：如果在 switch-case 语句中省略了 break，结果将如何？

答： break 让程序能够退出 switch 结构，如果没有它，将继续评估后面的 case 语句。

问：如何退出无限循环？

答： 使用 break 退出当前循环；使用 return 退出当前函数模块。

问：我编写了一个类似于 while(Integer)的循环，如果 Integer 的值为-1，这个 while 循环会执行吗？

答： 理想情况下，while 循环表达式应为布尔值 true 或 false，否则这样解读：零表示 false，非零表示 true。由于-1 不是零，因此该 while 条件为 true，循环将执行。如果希望仅当 Integer 为正数时才执行循环，可编写表达式 while(Integer>0)。这种规则适用于所有的条件语句和循环。

问：有与 for(;;)等效的空 while 语句吗？

答： 没有，while 语句必须有配套的条件表达式。

问：我通过复制并粘贴将 do...while(exp);改成了 while(exp);，这会导致问题吗？

答： 会出大问题！while(exp);合法，却是一个空 while 循环，因为 while 后面是一条空语句（分号），即便后面有语句块亦如此。后面的语句块将执行一次，但它位于循环外面。复制并粘贴代码时务必小心。

6.7 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

6.7.1 测验

1. 既然不缩进也能通过编译，为何要缩进语句块、嵌套 if 语句和嵌套循环？
2. 使用 goto 可快速解决问题，为何要避免使用它？
3. 可编写计数器递减的 for 循环吗？这样的 for 循环是什么样的？
4. 下面的循环有何问题？

```
for (int Counter=0; Counter==10; ++Counter)
   cout << Counter << " ";
```

6.7.2 练习

1. 编写一个 for 循环，以倒序方式访问数组的元素。
2. 编写一个类似于程序清单 6.13 的嵌套 for 循环，但以倒序方式将一个数组的每个元素都与另一个数组的每个元素相加。
3. 编写一个程序，像程序清单 6.15 那样显示斐波纳契数列，但让用户指定每次显示多少个。
4. 编写一个 switch-case 结构，指出用户选择的颜色是否出现在彩虹中。请使用枚举常量。
5. 查错：下面的代码有何错误？

```
for (int Counter=0; Counter=10; ++Counter)
   cout << Counter << " ";
```

6. 查错：下面的代码有何错误？

```
int LoopCounter = 0;
while(LoopCounter <5);
{
   cout << LoopCounter << " ";
   LoopCounter++;
}
```

7. 查错：下面的代码有何错误？

```
cout << "Enter a number between 0 and 4" << endl;
int Input = 0;
cin >> Input;

switch (Input)
{
case 0:
case 1:
case 2:
case 3:
case 4:
   cout << "Valid input" << endl;
default:
   cout << "Invalid input" << endl;
}
```

第 7 章
使用函数组织代码

到目前为止，本书的示例程序都使用 main()实现所有功能。对小型应用程序来说，这完全可行，但随着程序越来越大、越来越复杂，除非使用函数，否则 main()将越来越长。

函数让您能够划分和组织程序的执行逻辑。通过使用函数，可将应用程序的内容划分成依次调用的逻辑块。

函数是子程序，可接受参数并返回值，要让函数执行其任务，必须调用它。在本章中，您将学习：

- 为何需要编写函数；
- 函数原型和函数定义；
- 给函数传递参数以及从函数返回值；
- 重载函数；
- 递归函数；
- C++11 lambda 函数。

7.1 为何需要函数

假设要编写一个应用程序，让用户输入圆的半径并计算其周长和面积。为此，一种方式是将所有逻辑都放在 main()中。另一种方式是将该应用程序划分成逻辑块，具体地说是两个逻辑块，它们分别根据半径计算面积和周长，如程序清单 7.1 所示。

程序清单 7.1 两个根据半径分别计算圆的面积和周长的函数

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: const double Pi = 3.14159;
 4:
 5: // Function Declarations (Prototypes)
 6: double Area(double InputRadius);
 7: double Circumference(double InputRadius);
 8:
 9: int main()
10: {
11:    cout << "Enter radius: ";
12:    double Radius = 0;
13:    cin >> Radius;
14:
15:    // Call function "Area"
16:    cout << "Area is: " << Area(Radius) << endl;
17:
18:    // Call function "Circumference"
```

```
19:    cout << "Circumference is: " << Circumference(Radius) << endl;
20:
21:    return 0;
22:  }
23:
24: // Function definitions (implementations)
25: double Area(double InputRadius)
26: {
27:    return Pi * InputRadius * InputRadius;
28: }
29:
30: double Circumference(double InputRadius)
31: {
32:    return 2 * Pi * InputRadius;
33: }
```

▼ **输出：**

```
Enter radius: 6.5
Area is: 132.732
Circumference is: 40.8407
```

▼ **分析：**

乍一看，这不过是换汤不换药，但您将意识到，通过将计算面积和周长的代码放到不同的函数中，将有助于提高可重用性，因为可根据需要重复地调用这些函数。这里的 main()（它本身也是一个函数）相当简洁，它将工作委托给 Area 和 Circumference 等函数去完成，并在第 16 和 19 行分别调用它们。

该程序演示了使用函数进行编程涉及的如下内容。

- 第 6 和 7 行声明了函数原型，这样在 main()中使用 Area 和 Circumference 时，编译器知道它们是什么。
- 在 main()中，第 16 和 19 行调用了函数 Area()和 Circumference()。
- 第 25～28 行定义了函数 Area()，而第 30～33 行定义了函数 Circumference()。

7.1.1 函数原型是什么

再来看一下程序清单 7.1 的第 6 和 7 行：

```
double Area(double InputRadius);
double Circumference(double InputRadius);
```

图 7.1 说明了函数原型的组成部分。

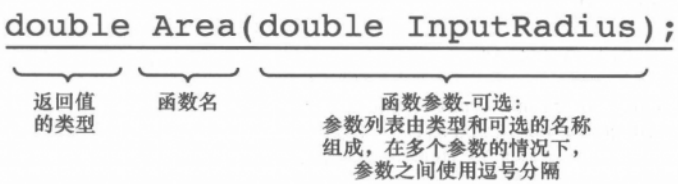


图 7.1 函数原型的组成部分

函数原型指出了函数的名称（Area）、函数接受的参数列表（一个名为 InputRadius 的 double 参数）以及返回值的类型（double）。

如果没有函数原型，编译器遇到 main()中的第 16 和 19 行时，将不知道 Area 和 Circumference 是什么。函数原型告诉编译器，Area 和 Circumference 是函数，它们接受一个类型为 double 的参数，并返回一个类型为 double 的值。这样，编译器将意识到这些语句是合法的，而链接器负责将函数调用与实现关联起来，并确保程序执行时将触发它们。

注意

函数可接受用逗号分隔的多个参数，但只能有一种返回类型。
编写不需要返回任何值的函数时，可将其返回类型指定为 void。

7.1.2 函数定义是什么

函数的最基本部分——实现——被称为函数定义。下面来分析函数 Area 的定义：

```
25: double Area(double InputRadius)
26: {
27:     return Pi * InputRadius * InputRadius;
28: }
```

函数定义总是由一个语句块组成。除非返回类型被声明为 void，否则函数必须包含一条 return 语句。就这里而言，函数 Area 需要返回一个值，因为其返回类型并不是 void。语句块是包含在左大括号和右大括号（{}）内的语句，在函数被调用时执行。Area()使用输入参数 InputRadius 来计算圆的面积，该参数包含调用者以实参方式传递的半径。

7.1.3 函数调用和实参是什么

如果函数声明中包含形参（parameter），调用函数时必须提供实参（argument），它们是函数的形参列表要求的值。下面来分析程序清单 7.1 中对函数 Area 的调用：

```
16:     cout << "Area is: " << Area(Radius) << endl;
```

其中 Area(Radius)是函数调用，而 Radius 是传递给函数 Area 的实参。执行到 Area(Radius)处时，将跳转到函数 Area 处，使用传递给它的半径计算圆的面积。函数 Area 执行完毕时，将返回一个 double 值。然后，通过 cout 语句将这个返回的 double 值显示到屏幕上。

7.1.4 编写接受多个参数的函数

假设要编写一个计算圆柱面积的程序，如图 7.2 所示。

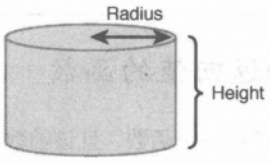


图 7.2 圆柱

计算圆柱面积的公式如下：

```
Area of Cylinder = Area of top circle + Area of bottom circle + Area of Side
                 = Pi *  Radius^2 + Pi * Radius ^2 + 2 * Pi * Radius * Height
                 = 2 * Pi * Radius^2 + 2 * Pi * Radius * Height
```

计算圆柱面积时，需要两个变量——半径和高度。编写计算圆柱表面积的函数时，至少需要在函数声明的形参列表中指定两个形参。为此，需要用逗号分隔形参，如程序清单 7.2 所示。

程序清单 7.2 接受两个参数以计算圆柱表面积的函数

```
0: #include <iostream>
1: using namespace std;
2:
3: const double Pi = 3.14159;
4:
```

```
5: // Declaration contains two parameters
6: double SurfaceArea(double Radius, double Height);
7:
8: int main()
9: {
10:    cout << "Enter the radius of the cylinder: ";
11:    double InRadius = 0;
12:    cin >> InRadius;
13:    cout << "Enter the height of the cylinder: ";
14:    double InHeight = 0;
15:    cin >> InHeight;
16:
17:    cout << "Surface Area: " << SurfaceArea(InRadius, InHeight) << endl;
18:
19:    return 0;
20: }
21:
22: double SurfaceArea(double Radius, double Height)
23: {
24:    double Area = 2 * Pi * Radius * Radius + 2 * Pi * Radius * Height;
25:    return Area;
26: }
```

▼ 输出：

```
Enter the radius of the cylinder: 3
Enter the height of the cylinder: 6.5
Surface Area: 179.071
```

▼ 分析：

第 6 行是函数 SurfaceArea 的声明，其中包含两个用逗号分隔的形参——Radius 和 Height，它们的类型都是 double。第 22～26 行是函数 SurfaceArea 的定义，即实现。正如您看到的，使用输入参数 Radius 和 Height 计算了表面积，将其存储在 Area 中，再将 Area 返回给调用者。

注意

函数形参类似于局部变量，它们只在当前函数内部可用。因此，在程序清单 7.2 中，函数 SurfaceArea 的形参 Radius 和 Height 只在函数 SurfaceArea 内部可用，而在该函数外部不可用。

7.1.5 编写没有参数和返回值的函数

如果将显示 Hello World 的工作委托给一个函数，且该函数不做别的，则它就不需要任何参数（因为它除了显示 Hello World 外，什么也不做），也无需返回任何值（因为您不指望这样的函数能提供在其他地方有用的东西）。程序清单 7.3 演示了一个这样的函数。

程序清单 7.3 没有参数和返回值的函数

```
0: #include <iostream>
1: using namespace std;
2:
3: void SayHello();
4:
5: int main()
6: {
7:    SayHello();
8:    return 0;
9: }
10:
11: void SayHello()
```

```
12: {
13:     cout << "Hello World" << endl;
14: }
```

▼ 输出：

```
Hello World
```

▼ 分析：

注意到第 3 行的函数原型将函数 SayHello 的返回类型声明为 void，即不返回任何值。因此，在第 11～14 行的函数定义中，没有 return 语句。在 main()中，第 7 行的函数调用没有将返回值赋给任何变量，也没有将其用于任何表达式中，因为该函数不返回任何值。

7.1.6 带默认值的函数参数

在本书前面的示例中，您都将 Pi 声明为常量，没有给用户提供修改它的机会。然而，用户可能希望其精度更高或更低。如果编写一个函数，在用户没有提供的情况下，将 Pi 设置为您选择的值呢？

为解决这种问题，一种方式是给函数 Area()新增一个表示 Pi 的参数，并将其默认值设置为您选择的值。对程序清单 7.1 所示的函数 Area()做这样的修改后，结果将如下：

```
double Area(double InputRadius, double Pi = 3.14);
```

请注意第二个参数 Pi 及其默认值 3.14。对调用者来说，这个参数是可选的，因此仍可使用下面的语法来调用 Area：

```
Area(Radius);
```

这里省略了第二个参数，因此它将使用默认值 3.14。然而，如果用户提供了 Pi 值，您就可在调用 Area 时指定它，如下所示：

```
Area(Radius, Pi); // User defined Pi
```

程序清单 7.4 演示了如何编写参数包含默认值的函数，这种默认值可被用户提供的值覆盖。

程序清单 7.4 计算圆面积的函数，其第二个参数为 Pi，该参数的默认值为 3.14

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: // Function Declaration (Prototype)
 4: double Area(double InputRadius, double Pi = 3.14);  // Pi with default value
 5:
 6: int main()
 7: {
 8:    cout << "Enter radius: ";
 9:    double Radius = 0;
10:    cin >> Radius;
11:
12:    cout << "Pi is 3.14, do you wish to change this (y / n)? ";
13:    char ChangePi = 'n';
14:    cin >> ChangePi;
15:
16:    double CircleArea = 0;
17:    if (ChangePi == 'y')
18:    {
19:       cout << "Enter new Pi: ";
20:       double NewPi = 3.14;
21:       cin >> NewPi;
22:       CircleArea = Area (Radius, NewPi);
23:    }
```

```
24:    else
25:       CircleArea = Area(Radius); // Ignore 2nd param, use default value
26:
27:    // Call function "Area"
28:    cout << "Area is: " << CircleArea << endl;
29:
30:    return 0;
31: }
32:
33: // Function definitions don't specify default values again
34: double Area(double InputRadius, double Pi)
35: {
36:    return Pi * InputRadius * InputRadius;
37: }
```

▼ **输出：**

```
Enter radius: 1
Pi is 3.14, do you wish to change this (y / n)? n
Area is: 3.14
```

再次运行的输出：

```
Enter radius: 1
Pi is 3.14, do you wish to change this (y / n)? y
Enter new Pi: 3.1416
Area is: 3.1416
```

▼ **分析：**

从上述输出可知，两次运行时用户输入的半径相同，都是1。然而，第二次运行时，用户修改了Pi的精度，因此计算得到的面积稍有不同。两次运行时调用的是同一个函数，如第22和25行所示。第25行调用Area时没有指定第二个参数Pi，因此将使用默认值3.14，这是在第4行的声明中指定的。

注意 可以给多个参数指定默认值，但这些参数必须位于参数列表的末尾。

7.1.7 递归函数——调用自己的函数

在有些情况下，可让函数调用它自己，这样的函数称为递归函数。递归函数必须有明确的退出条件，满足这种条件后，函数将返回，而不再调用自己。

警告 如果没有退出条件或存在bug，递归函数可能不断调用自己，直到栈溢出后才停止，导致应用程序崩溃。

计算斐波纳契数列时，递归函数很有用，如程序清单7.5所示。该数列的开头两个数为0和1：

```
F(0) = 0
F(1) = 1
```

随后的每个数都是前两个数之和。计算第n个数（n>1）的公式如下：

```
Fibonacci(n) = Fibonacci(n - 1) + Fibonacci(n - 2)
```

因此斐波纳契数列如下：

```
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8, and so on.
```

程序清单 7.5 使用递归函数计算斐波纳契数列中的数字

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int GetFibNumber(int FibIndex)
 4: {
 5:    if (FibIndex < 2)
 6:       return FibIndex;
 7:    else // recursion if FibIndex >= 2
 8:       return GetFibNumber(FibIndex - 1) + GetFibNumber(FibIndex - 2);
 9: }
10:
11: int main()
12: {
13:    cout << "Enter 0-based index of desired Fibonacci Number: ";
14:    int Index = 0;
15:    cin >> Index;
16:
17:    cout << "Fibonacci number is: " << GetFibNumber(Index) << endl;
18:    return 0;
19: }
```

▼ 输出：

```
Enter 0-based index of desired Fibonacci Number: 6
Fibonacci number is: 8
```

▼ 分析：

函数 GetFibNumber 是在第 3～9 行定义的，这是一个递归函数，因为它在第 8 行调用了自己。请注意第 5 和 6 行的退出条件：如果索引小于 2，该函数将不再递归。鉴于该函数调用自己时降低了 FibIndex 的值，因此递归到一定程度后将满足递归条件，从而停止递归。

7.1.8 包含多条 return 语句的函数

在函数定义中，并非只能有一条 return 语句。您可以在函数的任何地方返回，如果愿意，还可包含多条 return 语句，如程序清单 7.6 所示。这可能是糟糕的编程方式，也可能不是，这取决于应用程序的逻辑和需求。

程序清单 7.6 在同一个函数中使用多条 return 语句

```
 0: #include <iostream>
 1: using namespace std;
 2: const double Pi = 3.14159;
 3:
 4: void QueryAndCalculate()
 5: {
 6:    cout << "Enter radius: ";
 7:    double Radius = 0;
 8:    cin >> Radius;
 9:
10:    cout << "Area: " << Pi * Radius * Radius << endl;
11:
12:    cout << "Do you wish to calculate circumference (y/n)? ";
13:    char CalcCircum = 'n';
14:    cin >> CalcCircum;
15:
16:    if (CalcCircum == 'n')
17:       return;
```

```
18:
19:     cout << "Circumference: " << 2 * Pi * Radius << endl;
20:     return;
21: }
22:
23: int main()
24: {
25:     QueryAndCalculate ();
26:
27:     return 0;
28: }
```

▼ **输出：**

```
Enter radius: 1
Area: 3.14159
Do you wish to calculate circumference (y/n)? y
Circumference: 6.28319
```

再次执行的输出：

```
Enter radius: 1
Area: 3.14159
Do you wish to calculate circumference (y/n)? n
```

▼ **分析：**

函数 QueryAndCalculate 包含多条 return 语句：一条位于第 17 行，另一条位于第 20 行。这个函数询问用户是否也想计算周长，如果用户按 n 表示 no，程序将使用 return 语句退出；否则将接着计算周长，然后返回。

警告

在同一个函数中使用多条 return 语句要谨慎。相对于有多个返回点的函数，从顶部开始并在末尾返回的函数要容易理解得多。

在程序清单 7.6 中，使用了多条 return 语句。这很容易避免，只需修改 if 条件，使其检查用户输入的是否是'y'（Yes）即可：

```
cout << "Circumference: " << 2 * Pi * Radius << endl;
```

7.2 使用函数处理不同类型的数据

并非只能每次给函数传递一个值，还可将数组传递给函数。您可创建两个名称和返回类型相同，但参数不同的函数。您可创建这样的函数，即其参数不是在函数内部创建和销毁的；为此可使用在函数退出后还可用的引用，这样可在函数中操纵更多数据或参数。在本节中，您将学习将数组传递给函数、函数重载以及按引用给函数传递参数。

7.2.1 函数重载

名称和返回类型相同，但参数不同的函数被称为重载函数。在应用程序中，如果需要使用不同的参数调用具有特定名称和返回类型的函数，重载函数将很有用。假设您需要编写一个应用程序，它计算圆和圆柱的面积。计算圆面积的函数需要一个参数——半径，而计算圆柱面积的函数除需要圆柱的半径外，还需要圆柱的高度。这两个函数需要返回的数据类型相同，都是面积。C++让您能够定义两个重载的函数，它们都叫 Area，都返回 double，但一个接受半径作为参数，另一个接受半径和高度作为参数，如程序清单 7.7 所示。

程序清单 7.7 使用一个重载函数来计算圆或圆柱的面积

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: const double Pi = 3.14159;
 4:
 5: double Area(double Radius);   // for circle
 6: double Area(double Radius, double Height); // overloaded for cylinder
 7:
 8: int main()
 9: {
10:    cout << "Enter z for Cylinder, c for Circle: ";
11:    char Choice = 'z';
12:    cin >> Choice;
13:
14:    cout << "Enter radius: ";
15:    double Radius = 0;
16:    cin >> Radius;
17:
18:    if (Choice == 'z')
19:    {
20:       cout << "Enter height: ";
21:       double Height = 0;
22:       cin >> Height;
23:
24:       // Invoke overloaded variant of Area for Cyclinder
25:       cout << "Area of cylinder is: " << Area (Radius, Height) << endl;
26:    }
27:    else
28:       cout << "Area of cylinder is: " << Area (Radius) << endl;
29:
30:    return 0;
31: }
32:
33: // for circle
34: double Area(double Radius)
35: {
36:    return Pi * Radius * Radius;
37: }
38:
39: // overloaded for cylinder
40: double Area(double Radius, double Height)
41: {
42:    // reuse the area of circle
43:    return 2 * Area (Radius) + 2 * Pi * Radius * Height;
44: }
```

▼ 输出：

```
Enter z for Cylinder, c for Circle: z
Enter radius: 2
Enter height: 5
Area of cylinder is: 87.9646
```

再次执行的输出：

```
Enter z for Cylinder, c for Circle: c
Enter radius: 1
Area of cylinder is: 3.14159
```

▼ 分析：

第 5 和 6 行声明了两个重载的 Area 函数的原型，一个接受一个参数——圆半径，另一个接受两个

参数——圆柱的半径和高度。这两个函数同名，都叫 Area，返回类型相同，都是 double，但参数不同，因此它们是重载的。第 34～44 行是这两个重载的函数的定义，一个根据半径计算圆的面积，另一个根据半径和高度计算圆柱的面积。有趣的是，由于圆柱的面积由两个圆（顶圆和底圆）的面积和侧面的面积组成，因此用于圆柱的重载版本可重用用于圆的版本，如第 43 行所示。

7.2.2 将数组传递给函数

显示一个整数的函数类似于下面这样：

```
void DisplayInteger(int Number);
```

显示整型数组的函数的原型稍微不同：

```
void DisplayIntegers(int[] Numbers, int Length);
```

第一个参数告诉函数，输入的数据是一个数组，而第二个参数指出了数组的长度，以免您使用数组时跨越边界，如程序清单 7.8 所示。

程序清单 7.8 接受数组作为参数的函数

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: void DisplayArray(int Numbers[], int Length)
 4: {
 5:    for (int Index = 0; Index < Length; ++Index)
 6:       cout << Numbers[Index] << " ";
 7:
 8:    cout << endl;
 9: }
10:
11: void DisplayArray(char Characters[], int Length)
12: {
13:    for (int Index = 0; Index < Length; ++Index)
14:       cout << Characters[Index] << " ";
15:
16:    cout << endl;
17: }
18:
19: int main()
20: {
21:    int MyNumbers[4] = {24, 58, -1, 245};
22:    DisplayArray(MyNumbers, 4);
23:
24:    char MyStatement[7] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
25:    DisplayArray(MyStatement, 7);
26:
27:    return 0;
28: }
```

▼ 输出：

```
24 58 -1 245
H e l l o !
```

▼ 分析：

这里有两个重载的函数，它们都叫 DisplayArray：一个显示 int 数组的元素，另一个显示 char 数组的元素。在第 22 和 25 行，分别使用 int 数组和 char 数组调用了这两个函数。注意到第 24 行声明并初始化 char 数组时，故意在末尾添加了空字符（这是一种最佳实践，也是一种良好的习惯），虽然在这个应用程序中，没有在 cout 语句中将该数组用作字符串（cout << MyStatement;）。

7.2.3 按引用传递参数

再来看一下程序清单 7.1 中根据半径计算圆面积的函数：

```
24: // Function definitions (implementations)
25: double Area(double InputRadius)
26: {
27:     return Pi * InputRadius * InputRadius;
28: }
```

其中，参数 InputRadius 包含的值是在 main()中调用函数时复制给它的：

```
15:     // Call function "Area"
16:     cout << "Area is: " << Area(Radius) << endl;
```

这意味着函数调用不会影响 main()中的变量 Radius，因为 Area()使用的是 Radius 包含的值的拷贝。有时候，您可能希望函数修改的变量在其外部（如调用函数）中也可用，为此，可将形参的类型声明为引用。下面的 Area()函数计算面积，并以参数的方式按引用返回它：

```
// output parameter Result by reference
void Area(double Radius, double& Result)
{
   Result = Pi * Radius * Radius;
}
```

注意到该 Area()函数接受两个参数。别遗漏了第二个形参 Result 旁边的&，它告诉编译器，不要将第二个实参复制给函数，而将指向该实参的引用传递给函数。返回类型变成了 void，因为该函数不再通过返回值提供计算得到的面积，而按引用以输出参数的方式提供它。程序清单 7.9 演示了如何按引用返回值，该程序计算圆的面积。

程序清单 7.9 以引用参数（而不是返回值）的方式提供圆的面积

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: const double Pi = 3.1416;
 4:
 5: // output parameter Result by reference
 6: void Area(double Radius, double& Result)
 7: {
 8:     Result = Pi * Radius * Radius;
 9: }
10:
11: int main()
12: {
13:     cout << "Enter radius: ";
14:     double Radius = 0;
15:     cin >> Radius;
16:
17:     double AreaFetched = 0;
18:     Area(Radius, AreaFetched);
19:
20:     cout << "The area is: " << AreaFetched << endl;
21:     return 0;
22: }
```

▼ 输出：

```
Enter radius: 2
The area is: 12.5664
```

▼ **分析：**

注意到第 18 行调用函数 Area 时提供了两个参数，其中第二个参数将包含结果。由于 Area 的第二个参数是按引用传递的，因此 Area()中第 8 行使用的变量 Result，与 main()中第 17 行声明的 double AreaFetched 指向同一个内存单元。因此，在 main()中，可以使用 Area()中第 8 行计算得到的结果——第 20 行将其显示到屏幕上。

注意

使用 return 语句时，函数只能返回一个值。因此，如果函数需要执行影响众多值的操作，且需要在调用者中使用这些值，则按引用传递参数是让函数将修改结果提供给调用模块的方式之一。

7.3 微处理器如何处理函数调用

在微处理器级，函数调用是如何实现的呢？虽然确切地了解这一点不是非常重要，但您可能发现它很有趣。了解这一点有助于明白 C++为何支持本节后面将介绍的内联函数。

函数调用意味着微处理器跳转到属于被调用函数的下一条指令处执行。执行完函数的指令后，将返回到最初离开的地方。为实现这种逻辑，编译器将函数调用转换为一条供微处理器执行的 CALL 指令，该指令指出了接下来要获取的指令所在的地址，该地址归函数所有。编译函数本身时，编译器将 return 语句转换为一条供微处理器执行的 RET 指令。

遇到 CALL 指令时，微处理器将调用函数后将执行的指令的位置保存到栈中，再跳转到 CALL 指令包含的内存单元处。

理解栈

栈是一种后进先出的内存结构，很像堆叠在一起的盘子，您从顶部取盘子，这个盘子是最后堆叠上去的。将数据加入栈被称为压入操作；从栈中取出数据被称为弹出操作。栈增大时，栈指针将不断递增，始终指向栈顶，如图 7.3 所示。

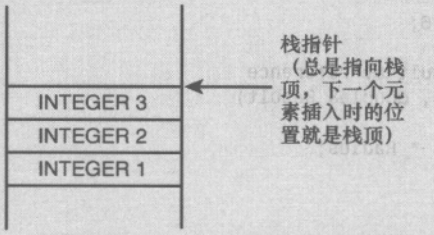


图 7.3 包含三个整数的栈的可视化表示

栈的性质使其非常适合用于处理函数调用。函数被调用时，所有局部变量都在栈中实例化，即被压入栈中。函数执行完毕时，这些局部变量都从栈中弹出，栈指针返回到原来的地方。

该内存单元包含属于函数的指令。微处理器执行它们，直到到达 RET 语句（与您编写的 return 语句对应的微处理器代码）。RET 语句导致微处理器从栈中弹出执行 CALL 指令时存储的地址。该地址包含调用函数中接下来要执行的语句的位置。这样，微处理器将返回到调用函数，从离开的地方继续执行。

7.3.1 内联函数

常规函数调用被转换为 CALL 指令，这会导致栈操作、微处理器跳转到函数处执行等。听起来在幕后发生了很多事情，但在大多数情况下速度都很快。然而，如果函数非常简单，类似于下面这样又如何呢？

```
double GetPi()
{
   return 3.14159;
}
```

相对于实际执行 GetPi()所需的时间，执行函数调用的开销可能非常高。这就是 C++编译器允许程序员将这样的函数声明为内联的原因。程序员使用关键字 inline 发出请求，要求在函数被调用时就地展开它们：

```
inline double GetPi()
{
   return 3.14159;
}
```

同样，只执行将数字翻倍等简单操作的函数也非常适合声明为内联的。程序清单 7.10 演示了一种这样的情形。

程序清单 7.10　将把整数翻倍的函数声明为内联的

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: // define an inline function that doubles
 4: inline long DoubleNum (int InputNum)
 5: {
 6:    return InputNum * 2;
 7: }
 8:
 9: int main()
10: {
11:    cout << "Enter an integer: ";
12:    int InputNum = 0;
13:    cin >> InputNum;
14:
15:    // Call inline function
16:    cout << "Double is: " << DoubleNum(InputNum) << endl;
17:
18:    return 0;
19: }
```

▼ 输出：

```
Enter an integer: 35
Double is: 70
```

▼ 分析：

第 4 行使用了关键字 inline。编译器通常将该关键字视为请求，请求将函数 DoubleNum 的内容直接放到调用它的地方（第 16 行），以提高代码的执行速度。

将函数声明为内联的会导致代码急剧膨胀，在声明为内联的函数做了大量复杂处理时尤其如此。应尽可能少用关键字 inline，仅当函数非常简单，需要降低其开销时（如前面所示），才应使用该关键字。

注意

大多数较新的 C++编译器都提供了各种性能优化选项。有些提供了优化大小或速度的选项，如 Microsoft C++编译器。为内存弥足金贵的设备和外设开发软件时，优化代码的大小至关重要。优化代码大小时，编译器可能拒绝众多的内联请求，因为这会让代码急剧膨胀。

优化速度时，编译器通常会寻找并利用合理的内联机会，为您完成内联工作，即便您没有显式地请求这样做。

C++11

7.3.2 lambda 函数

本节旨在简要地介绍一个对初学者来说不那么容易理解的概念，请快速浏览，尽力学习这个概念，即便不能完全掌握，也不用失望，第 22 章将深入讨论 lambda 函数。

如果您在编程时经常使用 STL 算法对包含在 STL 容器（如 std::vector）中的数据进行排序或处理，lambda 函数将很有用。通常，排序算法要求您提供一个二元谓词（它被实现为类中的一个运算符），这导致编码工作非常烦琐。遵守 C++11 标准的编译器让您能够编写 lambda 函数，从而极大地简化代码，如程序清单 7.11 所示。

程序清单 7.11 使用 lambda 函数对数组中的元素进行排序并显示它们

```
 0: #include <iostream>
 1: #include <algorithm>
 2: #include <vector>
 3: using namespace std;
 4:
 5: void DisplayNums(vector<int>& DynArray)
 6: {
 7:    for_each (DynArray.begin(), DynArray.end(), \
 8:            [](int Element) {cout << Element << " ";} );// lambda
 9:
10:    cout << endl;
11: }
12:
13: int main()
14: {
15:    vector<int> MyNumbers;
16:    MyNumbers.push_back(501);
17:    MyNumbers.push_back(-1);
18:    MyNumbers.push_back(25);
19:    MyNumbers.push_back(-35);
20:
21:    DisplayNums(MyNumbers);
22:
23:    cout << "Sorting them in descending order" << endl;
24:
25:    sort (MyNumbers.begin(), MyNumbers.end(), \
26:         [](int Num1, int Num2) {return (Num2 < Num1); } );
27:
28:    DisplayNums(MyNumbers);
29:
30:    return 0;
31: }
```

▼ 输出：

```
501 -1 25 -35
Sorting them in descending order
501 25 -1 -35
```

▼ 分析：

第 15～19 行将几个整数压入到一个动态数组中，这个动态数组是使用 C++标准模板库中的 std::vector 表示的。函数 DisplayNums 使用 STL 算法遍历数组的每个元素，并显示其值。为此，它在第 8 行使用了一个 lambda 函数，而不是冗长的一元函数谓词。第 25 行使用 std::sort 时，也以 lambda 函数的方式提供了一个二元谓词（第 26 行），这个函数在第二个数比第一个数小时返回 true，这相当于将集合按升序排列。

lambda 函数的语法如下：

```
[optional parameters](parameter list){ statements; }
```

7.4 总结

在本章中，您学习了模块化编程的基本知识。您了解到，使用函数可改善代码的结构，还有助于重用您编写的算法。您了解到，函数可接受参数并返回值；参数可以有调用者可覆盖的默认值，还可按引用传递参数。您学习了如何将数组传递给函数，还学习了如何编写名称和返回类型相同，但参数列表不同的重载函数。

最后，简要地介绍了 lambda 函数是什么。lambad 函数是 C++11 新增的，有望改变 C++应用程序的编写方式，尤其是使用 STL 时。

7.5 问与答

问：如果递归函数不终止，结果将如何？

答：程序将不断执行下去。程序不断执行下去也许不是坏事，因为 while(true)和 for(;;)循环也会导致这种后果。然而，递归函数调用将占用越来越多的栈空间，而栈空间有限，终将耗尽。最终，应用程序将因栈溢出而崩溃。

问：既然将函数声明为内联的可提高执行速度，为何不将所有函数都声明为内联的？

答：这样看情况而定。然而，如果将在多个地方使用的函数声明为内联的，将在调用它的每个地方放置其内容，这将导致代码急剧膨胀。另外，根据性能设置，大多数较新的编译器都能判断应内联哪些函数，进而为程序员这样做。

问：可给函数的所有参数都提供默认值吗？

答：绝对可以，在合理的情况下也推荐这样做。

问：我有两个函数，它们都叫 Area，其中一个接受半径作为参数，另一个接受高度作为参数，但我希望一个返回 float 值，另一个返回 double 值。这可行吗？

答：重载函数时，函数必须同名，且返回类型相同。就这里的情形而言，编译器将报错，因为它要求这两个函数的名称不同。

7.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

7.6.1 测验

1. 在函数原型中声明的变量的作用域是什么？
2. 传递给下述函数的值有何特征？

```
int Func(int &SomeNumber);
```

3. 调用自己的函数被称为什么?
4. 我声明了两个函数，它们的名称和返回类型相同，但参数列表不同，这被称为什么?
5. 栈指针指向栈的顶部、中间还是底部?

7.6.2 练习

1. 编写两个重载的函数，它们分别使用下述公式计算球和圆柱体的体积：

```
Volume of sphere = (4 * Pi * Radius * Radius * Radius) / 3
Volume of a cylinder = Pi * Radius * Radius * Height
```

2. 编写一个函数，它将一个 double 数组作为参数。
3. 查错：下述代码有什么错误?

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

const double Pi = 3.1416;

void Area(double Radius, double Result)
{
   Result = Pi * Radius * Radius;
}

int main()
{
   cout << "Enter radius: ";
   double Radius = 0;
   cin >> Radius;

   double AreaFetched = 0;
   Area(Radius, AreaFetched);

   cout << "The area is: " << AreaFetched << endl;
   return 0;
}
```

4. 查错：下述函数声明有什么错误?

```
double Area(double Pi = 3.14, double Radius);
```

5. 编写一个返回类型为 void 的函数，在提供了半径的情况下，它能帮助调用者计算圆的周长和面积。

第 8 章
阐述指针和引用

C++最大的优点之一是，您既可使用它来编写不依赖于机器的高级应用程序，又可使用它来编写与硬件紧密协作的应用程序。事实上，C++让您能够在字节和比特级调整应用程序的性能。要编写高效地利用系统资源的程序，理解指针和引用是必不可少的一步。

在本章中，您将学习：

- 什么是指针；
- 什么是自由存储区；
- 如何使用运算符 new 和 delete 分配和释放内存；
- 如何使用指针和动态分配编写稳定的应用程序；
- 什么是引用；
- 指针和引用的区别；
- 什么情况下使用指针，什么情况下使用引用。

8.1 什么是指针

简单地说，指针是存储内存地址的变量。就像 int 变量用于存储整数值一样，指针变量用于存储内存地址，如图 8.1 所示。

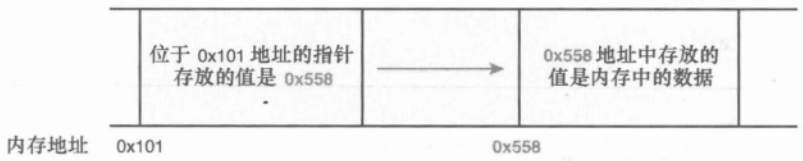


图 8.1 指针的可视化表示

因此，指针是一个变量，与所有变量一样，指针也占用内存空间（在图 8.1 中，其地址为 0x101）。指针的特殊之处在于，指针包含的值（这里为 0x558）被解读为内存地址，因此指针是一种指向内存单元的特殊变量。

8.1.1 声明指针

作为一种变量，指针也需要声明。通常将指针声明为指向特定的类型，如 int，这意味着指针包含的地址对应的内存单元存储了一个整数。也可将指针声明为指向一个内存块，这种指针被称为 void 指针。

作为一种变量，指针与所有变量一样，也需要声明：

```
PointedType * PointerVariableName;
```

与大多数变量一样，除非对指针进行初始化，否则它包含的值将是随机的。您不希望访问随机的内存地址，因此将指针初始化为 NULL。NULL 是一个可以检查的值，且不会是内存地址：

```
PointedType * PointerVariableName = NULL; // initializing value
```

因此，声明 int 指针的代码如下：

```
int *pInteger = NULL; //
```

注意 与您学过的所有数据类型一样，除非对指针进行初始化，否则它包含的将是垃圾值。对指针来说，这种垃圾值非常危险，因为指针包含的值被视为地址。未初始化的指针可能导致程序访问非法内存单元，进而导致程序崩溃。

8.1.2 使用引用运算符（&）获取变量的地址

在编程语言中，变量让您能够处理内存中的数据，这一概念在第 3 章详细阐述过。指针也是变量，但是一种特殊的变量，只用于存储内存地址。

如果 Varname 是一个变量，&VarName 将是存储该变量的内存的地址。

因此，如果您使用下面这种您非常熟悉的语法声明了一个 int 变量：

```
int Age = 30;
```

&Age 将是存储该变量的值（30）的内存的地址。程序清单 8.1 显示了存储变量值的内存的地址。

程序清单 8.1 获取 int 变量和 double 变量的地址

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     int Age = 30;
 6:     const double Pi = 3.1416;
 7:
 8:     // Use & to find the address in memory
 9:     cout << "Integer Age is at: 0x" << hex << &Age << endl;
10:     cout << "Double Pi is located at: 0x" << hex << &Pi << endl;
11:
12:     return 0;
13: }
```

▼ 输出：

```
Integer Age is at: 0x0045FE00
Double Pi is located at: 0x0045FDF8
```

▼ 分析：

注意到第 9 和 10 行使用了引用运算符（&）获取了变量 Age 和常量 Pi 的地址。作为一种约定，显示十六进制数时，应加上文本 0x。

注意 您知道，变量占用的内存量取决于其类型。在第 3 章，程序清单 3.4 使用 sizeof()证明了 int 变量占用的内存为 4 字节（在笔者的系统上，使用笔者的编译器）。前面的输出表明，int 变量 Age 的地址为 0x0045FE00，而 sizeof(int)为 4，因此 0x0045FE00-0x0045FE04 的 4 字节内存由 int 变量 Age 占用。

注意 引用运算符（&）也叫地址运算符。

8.1.3 使用指针存储地址

您知道如何声明指针以及如何获取变量的地址，还知道指针是用于存储内存地址的变量。现在该将它们关联起来，使用指针存储使用引用运算符（&）获取的地址。

假设您声明了一个某种类型的变量：

```
// Declaring a variable
Type VariableName = InitialValue;
```

要将该变量的地址存储到一个指针中，需要声明一个同样类型的指针，并使用引用运算符（&）将其初始化为该变量的地址：

```
// Declaring a pointer to the same type and initializing to address
Type* Pointer = &Variable;
```

因此，如果您使用自己非常熟悉的语法声明了一个 int 变量：

```
int Age = 30;
```

可像下面这样声明一个 int 指针来存储变量 Age 的地址：

```
int* pInteger = &Age; // Pointer to integer Age
```

程序清单 8.2 演示了如何使用指针来存储使用引用运算符（&）获取的地址。

程序清单 8.2 声明并初始化指针

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int Age = 30;
6:     int* pInteger = &Age;  // pointer to an int, initialized to &Age
7:
8:     // Displaying the value of pointer
9:     cout << "Integer Age is at: 0x" << hex << pInteger << endl;
10:
11:    return 0;
12: }
```

▼ 输出：

```
Integer Age is at: 0x0045FE00
```

▼ 分析：

该程序清单的输出与前一个程序清单相同，因为它们都显示变量 Age 的内存地址。不同之处在于，这里先将该地址赋给了一个指针（第 6 行），再在第 9 行使用 cout 显示该指针的值（地址）。

注意

> 在您的输出中，地址可能不同。事实上，即使在同一台计算机上，每次运行该应用程序时输出的变量地址都可能不同。

知道如何将地址存储到指针变量中后，就很容易想象得到，可将不同的内存地址赋给指针变量，让它指向不同的值，如程序清单 8.3 所示。

程序清单 8.3 给指针重新赋值，使其指向另一个变量

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
```

```
 4: {
 5:    int Age = 30;
 6:
 7:    int* pInteger = &Age;
 8:    cout << "pInteger points to Age now" << endl;
 9:
10:    // Displaying the value of pointer
11:    cout << "pInteger = 0x" << hex << pInteger << endl;
12:
13:    int DogsAge = 9;
14:    pInteger = &DogsAge;
15:    cout << "pInteger points to DogsAge now" << endl;
16:
17:    cout << "pInteger = 0x" << hex << pInteger << endl;
18:
19:    return 0;
20: }
```

▼ 输出：

```
pInteger points to Age now
pInteger = 0x002EFB34
pInteger points to DogsAge now
pInteger = 0x002EFB1C
```

▼ 分析：

这个程序表明，同一个 int 指针（pInteger）可指向任何 int 变量。第 7 行将该指针初始化为&Age，因此它包含变量 Age 的地址。第 14 行将&DogsAge 赋给了该指针，因此它指向包含 DogsAge 的内存单元。相应地，输出表明这个指针的值（即它指向的地址）发生了变化，因为 int 变量 Age 和 DogsAge 在内存中的存储位置不同——分别是 0x002EFB34 和 0x002EFB1C。

8.1.4 使用解除引用运算符（*）访问指向的数据

有了包含合法地址的指针后，如何访问这个地方，即如何获取或设置这个地方的数据呢？答案是使用解除引用运算符（*）。基本上，如果有合法的指针 pData，要访问它包含的地址处存储的值，可使用*pData。程序清单 8.4 演示了这种运算符（*）。

程序清单 8.4 使用解除引用运算符（*）来访问整数值

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int Age = 30;
 6:    int DogsAge = 9;
 7:
 8:    cout << "Integer Age = " << Age << endl;
 9:    cout << "Integer DogsAge = " << DogsAge << endl;
10:
11:    int* pInteger = &Age;
12:    cout << "pInteger points to Age" << endl;
13:
14:   // Displaying the value of pointer
15:    cout << "pInteger = 0x" << hex << pInteger << endl;
16:
17:    // Displaying the value at the pointed location
18:    cout << "*pInteger = " << dec << *pInteger << endl;
```

```
19:
20:     pInteger = &DogsAge;
21:     cout << "pInteger points to DogsAge now" << endl;
22:
23:     cout << "pInteger = 0x" << hex << pInteger << endl;
24:     cout << "*pInteger = " << dec << *pInteger << endl;
25:
26:     return 0;
27: }
```

▼ **输出：**

```
Integer Age = 30
Integer DogsAge = 9
pInteger points to Age
pInteger = 0x0025F788
*pInteger = 30
pInteger points to DogsAge now
pInteger = 0x0025F77C
*pInteger = 9
```

▼ **分析：**

像程序清单 8.3 一样，该程序清单也修改了指针存储的地址，还将解除引用运算符（*）用于指针变量 pInteger，以打印存储在这两个地址处的值。在第 18 和 24 行，使用解除引用运算符（*）访问了 pInteger 指向的整数。由于第 20 行修改了 pInteger 包含的地址，因此第 24 行使用该指针访问的是变量 DogsAge，即显示 9。

将解除引用运算符（*）用于该指针时，应用程序从它存储的地址开始，取回内存中 4 个字节的内容（因为该指针指向的是 int 变量，而 sizeof(int)为 4），因此指针包含的地址必须合法。第 11 行将指针初始化为&Age，确保它包含合法的地址。如果指针未初始化，它所在的内存单元将包含随机值，此时对其解除引用通常会导致非法访问（Access Violation），即访问应用程序未获得授权的内存单元。

注意

> 解除引用运算符（*）也叫间接运算符。

程序清单 8.4 使用指针读取它指向的内存单元中的值。程序清单 8.5 演示了将*pInteger 用作左值（即给它赋值，而不是读取其值）的情况。

程序清单 8.5　使用指针和解除引用运算符操纵数据

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     int DogsAge = 30;
 6:     cout << "Initialized DogsAge = " << DogsAge << endl;
 7:
 8:     int* pAge = &DogsAge;
 9:     cout << "pAge points to DogsAge" << endl;
10:
11:     cout << "Enter an age for your dog: ";
12:
13:     // store input at the memory pointed to by pAge
14:     cin >> *pAge;
15:
16:     // Displaying the address where age is stored
17:     cout << "Input stored using pAge at 0x" << hex << pAge << endl;
18:
19:     cout << "Integer DogsAge = " << dec << DogsAge << endl;
```

```
20:
21:    return 0;
22: }
```

▼ 输出：

```
Initialized DogsAge = 30
pAge points to DogsAge
Enter an age for your dog: 10
Input stored using pAge at 0x0025FA18
Integer DogsAge = 10
```

▼ 分析：

这里的关键步骤是第 14 行，它将用户提供的整数存储到指针 pAge 指向的地方。虽然存储输入时使用的是指针 pAge，但第 19 行显示变量 DogsAge 时，显示的却是使用指针存储的值，这是因为第 8 行初始化了 pAge，使其指向 DogsAge。pAge 指向存储 DogsAge 的内存单元，使用 pAge 和 DogsAge 中的一个修改该内存单元的内容时，另一个将受到影响。

8.1.5 将 sizeof()用于指针的结果

您知道，指针是包含内存地址的变量。因此无论指针指向哪种类型的变量，其内容都是一个地址——一个数字。在特定的系统中，存储地址所需的字节数是固定的。因此，将 sizeof()用于指针时，结果取决于编译程序时使用的编译器和针对的操作系统，与指针指向的变量类型无关，程序清单 8.6 演示了这一点。

程序清单 8.6 指向不同变量类型的指针的长度相同

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int Age = 30;
 6:    double Pi = 3.1416;
 7:    char SayYes = 'y';
 8:
 9:    // initialize pointers to addresses of the variables
10:    int* pInt = &Age;
11:    double* pDouble = &Pi;
12:    char* pChar = &SayYes;
13:
14:    cout << "sizeof fundamental types -" << endl;
15:    cout << "sizeof(int) = " << sizeof(int) << endl;
16:    cout << "sizeof(double) = " << sizeof(double) << endl;
17:    cout << "sizeof(char) = " << sizeof(char) << endl;
18:
19:    cout << "sizeof pointers to fundamental types -" << endl;
20:    cout << "sizeof(pInt) = " << sizeof(pInt) << endl;
21:    cout << "sizeof(pDouble) = " << sizeof(pDouble) << endl;
22:    cout << "sizeof(pChar) = " << sizeof(pChar) << endl;
23:
24:    return 0;
25: }
```

▼ 输出：

```
sizeof fundamental types -
sizeof(int) = 4
sizeof(double) = 8
```

```
sizeof(char) = 1
sizeof pointers to fundamental types -
sizeof(pInt) = 4
sizeof(pDouble) = 4
sizeof(pChar) = 4
```

▼ **分析：**

输出表明，虽然 sizeof(char)为 1 字节，而 sizeof(double)为 8 字节，但 sizeof(pointer)总是为 4 字节。这是因为不管指针指向的内存单元是 1 字节还是 8 字节，存储指针所需的内存量都相同。

注意 程序清单 8.6 的输出表明，将 sizeof 用于指针的结果为 4 字节，但在您的系统上结果可能不同。这里的输出是使用 32 位编译器编译代码时得到的，如果您使用的是 64 位编译器，并在 64 位系统上运行该程序，可能发现将 sizeof 用于指针的结果为 64 位，即 8 字节。

8.2 动态内存分配

如果在程序中使用下面这样的数组声明：

```
int Numbers[100]; // a static array of 100 integers
```

程序将存在下面两个问题。

1. 这限制了程序的容量，无法存储 100 个以上的数字。
2. 如果只需存储 1 个数字，却为 100 个数字预留存储空间，这将降低系统的性能。

导致这些问题的原因是，数组的内存分配是静态和固定的。

要编写根据用户需要使用内存资源的应用程序，需要使用动态内存分配。这让您能够根据需要分配更多内存，并释放多余的内存。为帮助您更好地管理应用程序占用的内存，C++提供了两个运算符——new 和 delete。指针是包含内存地址的变量，在高效地动态分配内存方面扮演了重要角色。

8.2.1 使用 new 和 delete 动态地分配和释放内存

您使用 new 来分配新的内存块。通常情况下，如果成功，new 将返回指向一个指针，指向分配的内存，否则将引发异常。使用 new 时，需要指定要为哪种数据类型分配内存：

```
Type* Pointer = new Type; // request memory for one element
```

需要为多个元素分配内存时，还可指定要为多少个元素分配内存：

```
Type* Pointer = new Type[NumElements]; // request memory for NumElements
```

因此，如果需要给整型分配内存，可使用如下语法：

```
int* pNumber = new int;  // get a pointer to an integer
int* pNumbers = new int[10];  // get a pointer to a block of 10 integers
```

注意 new 表示请求分配内存，并不能保证分配请求总能得到满足，因为这取决于系统的状态以及内存资源的可用性。

使用 new 分配的内存最终都需使用对应的 delete 进行释放：

```
Type* Pointer = new Type;
delete Pointer;  // release memory allocated above for one instance of Type
```

这种规则也适用于为多个元素分配的内存：

```
Type* Pointer = new Type[NumElements];
delete[] Pointer; // release block allocated above
```

注意 对于使用 new[...]分配的内存块，需要使用 delete[]来释放；对于使用 new 为单个元素分配的内存，需要使用 delete 来释放。

不再使用分配的内存后，如果不释放它们，这些内存仍被预留并分配给您的应用程序。这将减少可供其他应用程序使用的系统内存量，甚至降低您的应用程序的执行速度。这被称为内存泄露，应不惜一切代价避免这种情况发生。

程序清单 8.7 演示了如何动态地分配和释放内存。

程序清单 8.7 使用解除引用运算符（*）访问使用 new 分配的内存，并使用 delete 释放它

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    // Request for memory space for an int
 6:    int* pAge = new int;
 7:
 8:    // Use the allocated memory to store a number
 9:    cout << "Enter your dog's age: ";
10:    cin >> *pAge;
11:
12:    // use indirection operator* to access value
13:    cout << "Age " << *pAge << " is stored at 0x" << hex << pAge << endl;
14:
15:    delete pAge; // release memory
16:
17:    return 0;
18: }
```

▼ 输出：

```
Enter your dog's age: 9
Age 9 is stored at 0x00338120
```

▼ 分析：

第 6 行运算符 new 请求为一个整型分配内存，而您打算使用它来存储用户输入的小狗年龄。new 返回一个指针，因此将其赋给了一个指针变量。第 10 行使用 cin 和解除引用运算符(*) 将用户输入的年龄存储在新分配的内存中。第 13 行使用解除引用运算符(*) 显示存储的值，还显示了内存的地址。在第 13 行，pAge 包含的地址与第 6 行的 new 返回的地址相同，这个地址始终未变。

警告

> 不能将运算符 delete 用于任何包含地址的指针，而只能用于 new 返回的且未使用 delete 释放的指针。
>
> 因此，程序清单 8.6 所示的指针虽然包含有效地址，但不应使用 delete 来释放它们，因为这些地址并不是由 new 返回的。

对于使用 new[...]为一系列元素分配的内存，应使用 delete[]来释放，如程序清单 8.8 所示。

程序清单 8.8 使用 new[...]分配内存，并使用 delete[]释放它们

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Enter your name: ";
 7:    string Name;
 8:    cin >> Name;
 9:
10:    // Add 1 to reserve space for a terminating null
```

```
11:    int CharsToAllocate = Name.length() + 1;
12:
13:    // request for memory to hold copy of input
14:    char* CopyOfName = new char [CharsToAllocate];
15:
16:    // strcpy copies from a null-terminated string
17:    strcpy(CopyOfName, Name.c_str());
18:
19:    // Display the copied string
20:    cout << "Dynamically allocated buffer contains: " << CopyOfName << endl;
21:
22:    // Done using buffer? Delete
23:    delete[] CopyOfName;
24:
25:    return 0;
26: }
```

▼ **输出：**

```
Enter your name: Siddhartha
Dynamically allocated buffer contains: Siddhartha
```

▼ **分析：**

其中最重要的代码行是第 14 和 23 行，它们分别使用了运算符 new[...]和 delete[]。相比于程序清单 8.7，这个程序清单的不同之处在于，为多个元素而不是单个元素分配内存块。对于为一系列元素分配的内存块，使用完毕后必须使用 delete[]来释放。第 11 行计算需要为多少个字符分配内存时，将用户输入的字符数加 1，以便能够存储对 C 风格字符串来说至关重要的终止空字符。第 4 章解释了为何需要终止空字符。实际的复制工作是在第 17 行使用 strcpy 完成的，它将对 std::string Name 调用 c_str()得到的结果作为输入，将其复制到 char 缓冲区 CopyOfName 中。

注意

运算符 new 和 delete 分配和释放自由存储区中的内存。自由存储区是一种内存抽象，表现为一个内存池，应用程序可分配（预留）和释放其中的内存。

8.2.2 将递增和递减运算符（++和--）用于指针的结果

指针包含内存地址。例如，程序清单 8.3 的 int 指针包含 0x002EFB34——int 在内存中的地址。int 本身长 4 字节，因此占用 0x002EFB34-0x002EFB37 的内存。将递增运算符用于该指针后，它指向的并不是 0x002EFB35，因为指向 int 中间毫无意义。

如果您对指针执行递增或递减运算，编译器将认为您要指向内存块中相邻的值（并假定这个值的类型与前一个值相同），而不是相邻的字节（除非值的长度刚好是 1 字节，如 char）。

因此，对于程序清单 8.3 中的指针 pInteger，对其执行递增运算将导致它增加 4 字节，即 sizeof(int)。将++用于该指针相当于告诉编译器，您希望它指向下一个 int，因此递增后该指针将指向 0x002EFB38。同样，将该指针加 2 将导致它向前移动两个 int，即 8 字节。在本章后面，您将看到指针和数组索引之间的关系。

使用运算符--将指针递减的效果类似：将指针包含的地址值减去它指向的数据类型的 sizeof。

将指针递增或递减的结果

将指针递增或递减时，其包含的地址将增加或减少指向的数据类型的 sizeof（并不一定是 1 字节）。这样，编译器将确保指针不会指向数据的中间或末尾，而只会指向数据的开头。

如果声明了如下指针：

```
Type* pType = Address;
```

则执行++pType 后，pType 将包含（指向）Address + sizeof(Type)。

程序清单 8.9 演示了对指针递增和添加偏移量的结果。

程序清单 8.9 根据需要动态地分配内存，并研究对指针递增和添加偏移量的结果

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:   cout << "How many integers you wish to enter? ";
 6:   int InputNums = 0;
 7:   cin >> InputNums;
 8:
 9:   int* pNumbers = new int [InputNums]; // allocate requested integers
10:   int* pCopy = pNumbers;
11:
12:   cout<<"Successfully allocated memory for "<<InputNums<< " integers"<<endl;
13:   for(int Index = 0; Index < InputNums; ++Index)
14:   {
15:      cout << "Enter number "<< Index << ": ";
16:      cin >> *(pNumbers + Index);
17:   }
18:
19:   cout << "Displaying all numbers input: " << endl;
20:   for(int Index = 0, int* pCopy = pNumbers; Index < InputNums; ++Index)
21:      cout << *(pCopy++) << " ";
22:
23:    cout << endl;
24:
25:    // done with using the pointer? release memory
26:    delete[] pNumbers;
27:
28:    return 0;
29: }
```

▼ 输出：

```
How many integers you wish to enter? 2
Successfully allocated memory for 2 integers
Enter number 0: 789
Enter number 1: 575
Displaying all numbers input:
789 575
```

再次运行的输出：

```
How many integers you wish to enter? 5
Successfully allocated memory for 5 integers
Enter number 0: 789
Enter number 1: 12
Enter number 2: -65
Enter number 3: 285
Enter number 4: -101
Displaying all numbers input:
789 12 -65 285 -101
```

▼ 分析：

这个程序询问用户想输入多少个整数，再在第 9 行相应地分配内存。注意到第 10 行保存了该地址的拷贝，以便第 26 行使用它和 delete 来释放该内存块。这个程序演示了使用指针和动态内存分配相比于静态数组的优势：在用户需要存储较少的数字时，该应用程序占用的内存较少；在用户需要存储较多的数字时，占用的内存也较多，但从不浪费系统资源。由于内存是动态分配的，对可存储的整数数量没有限制，而只受限于可用的系统资源。第 13～17 行的 for 循环让用户输入数字，然后使用第 16 行的

表达式将其存储到相邻的位置。这个表达式给指针增加从零开始的偏移量（Index），从而将用户提供的值存储到内存的合适位置，而不覆盖前一个值。换句话说，表达式(pNumber + Index)返回一个指针，指向内存中的第 Index + 1 个整数（即 Index 为 1 时，返回的指针将指向第二个整数）。因此，cin 语句中的表达式*(pNumber + Index)访问的是第 Index + 1 个整数。第 20 和 21 行的 for 循环与此类似，它显示前一个循环存储的值。这条 for 语句执行了多项初始化任务，其中之一是创建指针的拷贝并存储到 pCopy 中；第 21 行递增该拷贝以便显示值。

第 10 行之所以创建指针的拷贝，是因为第二个循环使用递增运算符（++）修改了该指针。必须保留 new 最初返回的指针，因为第 26 行调用 delete[]时，必须提供它，而不能随便提供一个值。

8.2.3 将关键字 const 用于指针

第 3 章介绍过，通过将变量声明为 const 的，可确保变量的取值在整个生命周期内都固定为初始值。这种变量的值不能修改，因此不能将其用作左值。

指针也是变量，因此也可将关键字 const 用于指针。然而，指针是特殊的变量，包含内存地址，还可用于修改内存中的数据块。因此，const 指针有如下三种。

- 指针指向的数据为常量，不能修改，但可以修改指针包含的地址，即指针可以指向其他地方。

```
int HoursInDay = 24;
const int* pInteger = &HoursInDay; // cannot use pInteger to change
HoursInDay
int MonthsInYear = 12;
pInteger = &MonthsInYear; // OK!
*pInteger = 13; // Compile fails: cannot change data
int* pAnotherPointerToInt = pInteger; // Compile fails: cannot assign const
to non-const
```

- 指针包含的地址是常量，不能修改，但可修改指针指向的数据。

```
int DaysInMonth = 30;
// pInteger cannot point to anything else
int* const pDaysInMonth = &DaysInMonth;
*pDaysInMonth = 31; // OK! Value can be changed
int DaysInLunarMonth = 28;
pDaysInMonth = &DaysInLunarMonth; // Compile fails: Cannot change address!
```

- 指针包含的地址以及它指向的值都是常量，不能修改（这种组合最严格）。

```
int HoursInDay = 24;
// pointer can only point to HoursInDay
const int* const pHoursInDay = &HoursInDay;
*pHoursInDay = 25; // Compile fails: cannot change pointed value
int DaysInMonth = 30;
pHoursInDay = &DaysInMonth; // Compile fails: cannot change pointer value
```

将指针传递给函数时，这些形式的 const 很有用。函数参数应声明为最严格的 const 指针，以确保函数不会修改指针指向的值。这让函数更容易维护，在时过境迁和人员更迭后尤其如此。

8.2.4 将指针传递给函数

指针是一种将内存空间传递给函数的有效方式，其中可以包含值，也可以包含结果。将指针作为函数参数时，确保函数只能修改您希望它修改的参数很重要。例如，如果函数根据以指针方式传入的半径计算圆的面积，就不应允许它修改半径。为控制函数可修改哪些参数以及不能修改哪些参数，可使用 const 指针，如程序清单 8.10 所示。

程序清单 8.10 以指针方式将 Pi 和半径传递给计算圆面积的函数时，使用 const 限定相应的参数

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: void CalcArea(const double* const pPi, // const pointer to const data
 4:               const double* const pRadius, // i.e. nothing can be changed
 5:               double* const pArea) //change pointed value, not address
 6: {
 7:    // check pointers before using!
 8:    if (pPi && pRadius && pArea)
 9:       *pArea = (*pPi) * (*pRadius) * (*pRadius);
10: }
11:
12: int main()
13: {
14:    const double Pi = 3.1416;
15:
16:    cout << "Enter radius of circle: ";
17:    double Radius = 0;
18:    cin >> Radius;
19:
20:    double Area = 0;
21:    CalcArea (&Pi, &Radius, &Area);
22:
23:    cout << "Area is = " << Area << endl;
24:
25:    return 0;
26: }
```

▼ 输出：

```
Enter radius of circle: 10.5
Area is = 346.361
```

▼ 分析：

第 3～5 行演示了两种 const 指针，pRadius 和 pPi 被声明为“指向 const 数据的 const 指针”，因此不能修改指针包含的地址，也不能修改它指向的数据。pArea 显然是用于存储的参数，因为不能修改该指针的值（地址），但可修改它指向的数据。第 8 行在使用函数的指针参数前检查其有效性。在调用者不小心将这三个参数之一设置为 NULL 指针时，您不希望函数计算面积，因为这将导致应用程序崩溃。

8.2.5 数组和指针的类似之处

在程序清单 8.9 中，通过递增指针来访问内存中的下一个整数，这是不是与数组索引很像。当您声明下面的 int 数组时：

```
int MyNumbers[5];
```

编译器将分配固定数量的内存，用于存储 5 个整数；同时向您提供一个指向数组中第一个元素的指针，而指针由您指定的数组名标识。换句话说，MyNumbers 是一个指针，指向第一个元素（MyNumber[0]），程序清单 8.11 演示了这种关系。

程序清单 8.11 数组变量是指向第一个元素的指针

```
 0: #include <iostream>
 1: using namespace std;
 2:
```

```
 3: int main()
 4: {
 5:    // Static array of 5 integers
 6:    int MyNumbers[5];
 7:
 8:    // array assigned to pointer to int
 9:    int* pNumbers = MyNumbers;
10:
11:    // Display address contained in pointer
12:    cout << "pNumbers = 0x" << hex << pNumbers << endl;
13:
14:    // Address of first element of array
15:    cout << "&MyNumbers[0] = 0x" << hex << &MyNumbers[0] << endl;
16:
17:    return 0;
18: }
```

▼ 输出：

```
pNumbers = 0x004BFE8C
&MyNumbers[0] = 0x004BFE8C
```

▼ 分析：

这个程序表明，可将数组变量赋给类型与之相同的指针，如第 9 行所示，这证明了数组与指针类似。第 12 和 15 行表明，存储在指针中的地址与数组第一个元素在内存中的地址相同。这个程序表明，数组是指向其第一个元素的指针。

要访问第二个元素，可使用 MyNumbers[1]，也可通过指针 pNumbers 来访问，其语法为*(pNumbers+1)。要访问静态数组的第三个元素，可使用 MyNumbers[2]，而要访问动态数组的第三个元素，可使用语法 *(pNumbers+2)。

由于数组变量就是指针，因此也可将用于指针的解除引用运算符（*）用于数组。同样，可将数组运算符（[]）用于指针，如程序清单 8.12 所示。

程序清单 8.12　使用解除引用运算符（*）访问数组中的元素以及将数组运算符（[]）用于指针

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int ARRAY_LEN = 5;
 6:
 7:    // Static array of 5 integers, initialized
 8:    int MyNumbers[ARRAY_LEN] = {24, -1, 365, -999, 2011};
 9:
10:    // Pointer initialized to first element in array
11:    int* pNumbers = MyNumbers;
12:
13:    cout << "Displaying array using pointer syntax, operator*" << endl;
14:    for (int Index = 0; Index < ARRAY_LEN; ++Index)
15:       cout << "Element " << Index << " = " << *(MyNumbers + Index) << endl;
16:
17:    cout << "Displaying array using pointer with array syntax, operator[]" <<
       endl;
18:    for (int Index = 0; Index < ARRAY_LEN; ++Index)
19:       cout << "Element " << Index << " = " << pNumbers[Index] << endl;
20:
21:    return 0;
22: }
```

▼ **输出：**

```
Displaying array using pointer syntax, operator*
Element 0 = 24
Element 1 = -1
Element 2 = 365
Element 3 = -999
Element 4 = 2011
Displaying array using pointer with array syntax, operator[]
Element 0 = 24
Element 1 = -1
Element 2 = 365
Element 3 = -999
Element 4 = 2011
```

▼ **分析：**

在这个应用程序中，第 8 行声明并初始化了一个包含 5 个 int 元素的静态数组。这个应用程序通过两种可相互替换的方式显示该数组的内容，一种是使用数组变量和间接运算符（*），如第 15 行所示；另一种方式是使用指针变量和数组运算符（[]），如第 19 行所示。

该程序表明，数组 MyNumbers 和指针 pNumbers 都具有指针的特点。换句话说，数组类似于在固定内存范围内发挥作用的指针。可将数组赋给指针，如第 11 行所示，但不能将指针赋给数组，因为数组是静态的，不能用作左值。

警告

使用运算符 new 动态分配的指针仍需使用运算符 delete 来释放，虽然其使用语法与静态数组类似。牢记这一点很重要。

如果忘记这样做，应用程序将泄露内存，这很糟糕。

8.3 使用指针时常犯的编程错误

C++让您能够动态地分配内存，以优化应用程序对内存的使用。不同于 C#和 Java 等基于运行时环境的新语言，C++没有自动垃圾收集器对程序已分配但不能使用的内存进行清理。指针使用起来比较棘手，程序员很容易犯错。

8.3.1 内存泄露

这可能是 C++应用程序最常见的问题之一：运行时间越长，占用的内存越多，系统越慢。如果在使用 new 动态分配的内存不再需要后，程序员没有使用配套的 delete 释放，通常就会出现这种情况。

确保应用程序释放其分配的所有内存是程序员的职责。绝不能让下面这样的情形发生：

```
int* pNumbers = new int [5]; // initial allocation
// use pointer pNumbers
...
// forget to release using delete[] pNumbers;
...
// make another allocation and overwrite the pointer
pNumbers = new int[10]; // leaks the previously allocated memory
```

8.3.2 指针指向无效的内存单元

使用运算符*对指针解除引用，以访问指向的值时，务必确保指针指向了有效的内存单元，否则程序要么崩溃，要么行为不端。这看起来合乎逻辑，但一个非常常见的导致应用程序崩溃的原因就是无效指针。

指针无效的原因很多，但都要归结于糟糕的内存管理。程序清单 8.13 演示了一种导致指针无效的典型情形。

程序清单 8.13 使用无效指针

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    // uninitialized pointer (bad)
 6:    int* pTemperature;
 7:
 8:    cout << "Is it sunny (y/n)?" << endl;
 9:    char UserInput = 'y';
10:    cin >> UserInput;
11:
12:    if (UserInput == 'y')
13:    {
14:       pTemperature = new int;
15:       *pTemperature = 30;
16:    }
17:
18:    // pTemperature contains invalid value if user entered 'n'
19:    cout << "Temperature is: " << *pTemperature;
20:
21:    // delete also being invoked for those cases new wasn't done
22:    delete pTemperature;
23:
24:    return 0;
25: }
```

▼ 输出：

```
Is it sunny (y/n)? y
Temperature is: 30
```

再次运行的输出：

```
Is it sunny (y/n)? n
<CRASH!>
```

▼ 分析：

这个程序的问题很多，有些已通过注释指出了。第 14 行分配内存并将其赋给指针，但这行代码仅在用户按 y（表示 yes）时才会执行。用户提供其他输入时，该 if 块都不会执行，因此指针 pTemperature 无效。第二次运行时，用户按 n，导致应用程序崩溃。因为 pTemperature 包含无效的内存地址，而第 19 行对这个无效的指针解除引用，导致应用程序崩溃。

同样，第 22 行对这个指针调用 delete，但并未使用 new 分配这个指针，这也是大错特错。如果有指针的多个拷贝，只需对其中一个调用 delete（应避免指针拷贝满天飞）。

要让程序清单 8.13 所示的程序更好、更安全、更稳定，应对指针进行初始化，确定指针有效后再使用并只释放指针一次（且仅当指针有效时才释放）。

8.3.3 悬浮指针（也叫迷途或失控指针）

使用 delete 释放后，任何有效指针都将无效。因此，在程序清单 8.13 中，即便用户按了 y，在第 22 行前指针 pTemperature 是有效的，但第 22 行调用 delete 后，它也变成无效的了，不应再使用。

为避免这种问题，很多程序员在初始化指针或释放指针后将其设置为 NULL，并在使用运算符*对指针解除引用前检查它是否有效。

8.4 指针编程最佳实践

在应用程序中使用指针时，应遵守一些基本规则，这样您的工作将更轻松。

应该	不应该
务必初始化指针变量，否则它将包含垃圾值。这些垃圾值被解读为地址，但您的应用程序并未获得访问这些地方的授权。如果不能将指针初始化为 new 返回的有效地址或其他有效变量，可将其初始化为 NULL。 使用指针前，务必检查它是否为 NULL。这样，如果指针声明后未赋给有效地址（声明时将其初始化为 NULL 了），就不会像程序清单 8.13 那样错误地使用它们。 务必仅在指针有效时才使用它，否则程序可能崩溃。 对于使用 new 分配的内存，一定要记得使用 delete 进行释放，否则应用程序将泄露内存，进而降低系统的性能。	使用 delete 释放内存块或指针后，不要访问它。 不要对同一个内存地址调用 delete 多次。 使用完动态分配的内存块后，别忘了对其调用 delete，以免泄露内存。

学习一些指针编程最佳实践后，该修改程序清单 8.13 中错误百出的代码了，如程序清单 8.14 所示。

程序清单 8.14 更安全的指针编程——程序清单 8.13 的修正版

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Is it sunny (y/n)? ";
 6:    char UserInput = 'y';
 7:    cin >> UserInput;
 8:
 9:    if (UserInput == 'y')
10:    {
11:       // initialized pointer (good)
12:       int* pTemperature = new int;
13:       *pTemperature = 30;
14:
15:       cout << "Temperature is: " << *pTemperature << endl;
16:
17:       // done using pointer? delete
18:       delete pTemperature;
19:    }
20:
21:    return 0;
22: }
```

▼ 输出：

```
Is it sunny (y/n)? y
Temperature is: 30
```

再次运行的输出：

```
Is it sunny (y/n)? n
```

▼ 分析：

主要差别在于，需要时（即用户按 y 时）才创建指针，并在创建时对其进行初始化，如第 12 行所示。在同一个代码块中释放了内存，避免了在没有赋给指针有效内存地址的情况下使用它（解除引用或调用 delete）。

8.4.1 检查使用 new 发出的分配请求是否得到满足

除非请求分配的内存量特大，或系统处于临界状态，可供使用的内存很少，new 一般都能成功。有些应用程序需要请求分配大块的内存（如数据库应用程序），一般而言，不要假定内存分配能够成功，这很重要。C++提供了两种确保指针有效的方法，默认方法是使用异常，即如果内存分配失败，将引发 std::bad_alloc 异常。这导致应用程序中断执行，除非您提供了异常处理程序，否则应用程序将崩溃，并显示一条类似于“异常未处理”的消息。

第 28 章将详细讨论如何解决这种问题。程序清单 8.15 演示了如何使用异常处理检查分配请求是否失败。

程序清单 8.15 异常处理——在 new 失败时妥善地退出

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    try
 6:    {
 7:       // Request lots of memory space
 8:       int* pAge = new int [536870911];
 9:
10:       // Use the allocated memory
11:
12:       delete[] pAge;
13:    }
14:    catch (bad_alloc)
15:    {
16:       cout << "Memory allocation failed. Ending program" << endl;
17:    }
18:    return 0;
19: }
```

▼ 输出：

```
Memory allocation failed. Ending program
```

▼ 分析：

在您的计算机上，这个程序的执行情况可能不同。在我的计算机上，无法为 536870911 个整型分配内存，如果没有编写异常处理程序（第 14～17 行的 catch 块），程序将以令人非常讨厌的方式结束。使用 Microsoft Visual Studio 以调试模式生成可执行文件，并在 Microsoft Visual Studio 外部执行它时，将出现如图 8.2 所示的输出。

在调试模式下生成的可执行文件包含开发环境插入的异常处理程序，这导致出现图 8.2 所示的消息。在发布模式下生成时，操作系统（这里是 Windows）将非常唐突地终止该应用程序，如

图 8.3 所示。

图 8.2 如果删除异常处理代码，程序清单 8.15 所示的程序将崩溃（使用 MSVC 编译器以调试模式生成）

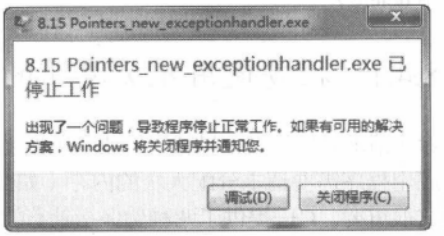


图 8.3 如果删除异常处理代码，程序清单 8.15 所示的程序将崩溃（以发布模式生成）

应用程序像这样崩溃时，将被操作系统终止，如果没有异常处理程序，您都没有机会与用户说“再见”。

通过使用异常处理程序，可让应用程序告诉用户遇到了问题，然后妥善地退出，而不是让操作系统显示一条崩溃消息。

有一个 new 变种——new(nothrow)，它不引发异常，而返回 NULL，让您能够在使用指针前检查其有效性，如程序清单 8.16 所示。

程序清单 8.16 使用 new(nothrow)，它在分配内存失败时返回 NULL

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    // Request lots of memory space, use nothrow version
 6:    int* pAge = new(nothrow) int [0x1fffffff];
 7:
 8:    if (pAge) // check pAge != NULL
 9:    {
10:       // Use the allocated memory
11:       delete[] pAge;
12:    }
13:    else
14:       cout << "Memory allocation failed. Ending program" << endl;
15:
16:    return 0;
17: }
```

▼ 输出：

```
Memory allocation failed. Ending program
```

▼ 分析：

这个程序与程序清单 8.15 相同，但使用的是 new(nothrow)，这样分配内存失败时，将返回 NULL，而不是引发异常 std::bad_alloc。这两种做法都可行，如何选择取决于您。

8.5 引用是什么

引用是变量的别名。声明引用时，需要将其初始化为一个变量，因此引用只是另一种访问相应变量存储的数据的方式。

要声明引用，可使用引用运算符（&），如下面的语句所示：

```
VarType Original = Value;
VarType& ReferenceVariable = Original;
```

要更深入地了解如何声明和使用引用，请参阅程序清单 8.17。

程序清单 8.17 引用是相应变量的别名

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     int Original = 30;
 6:     cout << "Original = " << Original << endl;
 7:     cout << "Original is at address: " << hex << &Original << endl;
 8:
 9:     int& Ref = Original;
10:     cout << "Ref is at address: " << hex << &Ref << endl;
11:
12:     int& Ref2 = Ref;
13:     cout << "Ref2 is at address: " << hex << &Ref2 << endl;
14:     cout << "Ref2 gets value, Ref2 = " << dec << Ref2 << endl;
15:
16:     return 0;
17: }
```

▼ 输出：

```
Original = 30
Original is at address: 0044FB5C
Ref is at address: 0044FB5C
Ref2 is at address: 0044FB5C
Ref2 gets value, Ref2 = 30
```

▼ 分析：

输出表明，无论将引用初始化为变量（如第 9 行所示）还是其他引用（如第 12 行所示），它都指向相应变量所在的内存单元。因此，引用是真正的别名，即相应变量的另一个名字。第 14 行显示了 Ref2 的值，结果与第 6 行显示的 Original 值相同，因为 Ref2 是 Original 的别名，它们位于内存的同一个地方。

8.5.1 是什么让引用很有用

引用让您能够访问相应变量所在的内存单元，这使得编写函数时引用很有用。第 7 章介绍过，典型的函数声明类似于下面这样：

```
ReturnType DoSomething(Type Parameter);
```

调用函数 DoSomething()的代码类似于下面这样：

```
ReturnType Result = DoSomething(argument); // function call
```

上述代码导致将 argument 的值复制给 Parameter，再被函数 DoSomething()使用。如果 argument 占

用了大量内存，这个复制步骤的开销将很大。同样，当 DoSomething()返回值时，这个值被复制给 Result。如果能避免这些复制步骤，让函数直接使用调用者栈中的数据就太好了。为此，可使用引用。

可避免复制步骤的函数版本类似于下面这样：

```
ReturnType DoSomething(Type& Parameter); // note the reference&
```

调用该函数的代码类似于下面这样：

```
ReturnType Result = DoSomething(argument);
```

由于 argument 是按引用传递的，Parameter 不再是 argument 的拷贝，而是它的别名，这类似于程序清单 8.17 中的 Ref。另外，接受引用参数的函数可使用这些参数返回值，如程序清单 8.18 所示。

程序清单 8.18 一个计算平方值并通过引用参数返回结果的函数

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: void ReturnSquare(int& Number)
 4: {
 5:    Number *= Number;
 6: }
 7:
 8:
 9: int main()
10: {
11:    cout << "Enter a number you wish to square: ";
12:    int Number = 0;
13:    cin >> Number;
14:
15:    ReturnSquare(Number);
16:    cout << "Square is: " << Number << endl;
17:
18:    return 0;
19: }
```

▼ 输出：

```
Enter a number you wish to square: 5
Square is: 25
```

▼ 分析：

计算平方的函数位于第 3～6 行。它通过引用参数接受一个要计算其平方的数字，并通过该参数返回结果。如果忘记将参数 Number 声明为引用（&），结果将无法返回到调用函数 main()，因为 ReturnSquare 将使用 Number 的本地拷贝执行运算，而函数结束时该拷贝将被销毁。通过使用引用，可确保 ReturnSquare()对 main()中定义的 Number 所在的内存单元进行操作。这样，函数 ReturenSquare()执行完毕后，也可以在 main()中使用运算结果。

在这个示例中，修改了输入参数。如果要保留这两个值——传入的数字及其平方，可让函数接受两个参数：一个包含输入；另一个提供平方值。

8.5.2 将关键字 const 用于引用

可能需要禁止通过引用修改它指向的变量的值，为此可在声明引用时使用关键字 const：

```
int Original = 30;
const int& ConstRef = Original;
ConstRef = 40; // Not allowed: ConstRef can't change value in Original
int& Ref2 = ConstRef; // Not allowed: Ref2 is not const
const int& ConstRef2 = ConstRef; // OK
```

8.5.3 按引用向函数传递参数

引用的优点之一是，可避免将形参复制给形参，从而极大地提高性能。然而，让被调用的函数直接使用调用函数栈时，确保被调用函数不能修改调用函数中的变量很重要。为此，可将引用声明为 const 的，如程序清单 8.19 所示。const 引用参数不能用作左值，因此试图给它们赋值将无法通过编译。

程序清单 8.19 使用 const 引用确保被调用的函数不能修改按引用传入的值

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: void CalculateSquare(const int& Number, int& Result) // note "const"
 4: {
 5:    Result = Number*Number;
 6: }
 7:
 8: int main()
 9: {
10:    cout << "Enter a number you wish to square: ";
11:    int Number = 0;
12:    cin >> Number;
13:
14:    int Square = 0;
15:    CalculateSquare(Number, Square);
16:    cout << Number << "^2 = " << Square << endl;
17:
18:    return 0;
19: }
```

▼ 输出：

```
Enter a number you wish to square: 27
27^2 = 729
```

▼ 分析：

在前一个程序中，使用同一个参数来接受输入和存储结果，但这里使用了两个参数，一个用于接受输入，另一个用于存储运算结果。为禁止修改传入的值，必须使用关键字 const 将其声明为 const 引用，如第 3 行所示。这让 Number 自动变为输入参数——其值不能修改的参数。

您可以尝试修改第 5 行，使其像程序清单 8.18 那样返回平方值：

```
Number *= Number;
```

这将导致编译错误，指出 const 值不能修改。const 引用是 C++提供的一个功能强大的工具，可用于将参数标识为输入参数，并禁止被调用的函数修改按引用传入的值。乍一看，这可能微不足道，但在多名程序员合作编程时，编写第一个版本的人和改进或修正的人可能不同，通过使用 const 引用可极大地提高编程质量。

8.6 总结

本章介绍了指针和引用。您学习了指针，它可用来访问和操纵内存，还是帮助动态分配内存的工具。您学习了运算符 new 和 delete，它们可用于为单个元素分配和释放内存；还学习了变种 new[...]和 delete[]，它们可用于为数组分配和释放内存。您简要地了解了指针编程和动态内存分配的陷进，知道释放动态分配的内存至关重要，有助于避免内存泄露。引用是别名，将参数传递给函数时，引用可很好地替代指针，因为引用总是有效的。您学习了 const 指针和 const 引用，知道声明函数时应尽可能提高参数的 const 程度。

8.7 问与答

问：既然使用静态数组无需释放内存，为何要动态分配内存？

答：静态数组的长度是固定的，不能根据应用程序的需求增大或缩小，而动态内存分配可满足这样的需求。

问：我声明了两个指针：

```
int* pNumber = new int;
int* pCopy = pNumber;
```

为释放内存，是否需要对它们都调用 delete？

答：这样做是错误的。对 new 返回的地址，只能调用 delete 一次。另外，最好避免让两个指针指向相同的地址，因为对其中一个调用 delete 将导致另一个无效。另外，编写程序时，应避免使用有效性不确定的指针。

问：在什么情况下应使用 new(nothrow)？

答：如果不想处理异常 std::bad_alloc，可使用 new(nothrow)，它在内存分配失败时返回 NULL。

问：下面是我编写的面积计算函数的两个版本：

```
void CalculateArea (const double* const pRadius, double* const pArea);
void CalculateArea (const double& radius, double& area);
```

请问哪个版本更好？

答：使用引用的版本更好，因为引用不可能无效，而指针可能无效。另外，第二个版本也更简单。

问：我编写了如下代码：

```
int Number = 30;
const int* pNumber = &Number;
```

我知道，由于 const 声明，我不能使用指针 pNumber 来修改变量 Number 的值。我可以将 pNumber 赋给一个非 const 指针，再使用该指针来操纵变量 Number 的值吗？

答：不能，您不能修改指针的 const 程度：

```
int* pAnother = pNumber; // cannot assign pointer to const to a non-const
```

问：为何要按引用向函数传递值？

答：可以不这样做，只要影响不大。然而，如果函数接受非常大的对象，则按值传递的开销将非常大，通过使用引用，可极大地提高函数调用的效率。别忘了将 const 用于引用参数，除非函数需要将结果存储在参数中。

问：下面的两个声明有何不同？

```
int MyNumbers[100];
int* MyArrays[100];
```

答：MyNumbers 是一个 int 数组，它指向这样的内存单元的开头，即其中存储了 100 个整数。它是静态的，可替换如下代码：

```
int* MyNumbers = new int [100]; // dynamically allocated array
// use MyNumbers
delete MyNumbers;
```

MyArrays 是一个包含 100 个元素的指针数组，其中的每个指针都可指向一个 int 或 int 数组。

8.8 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

8.8.1 测验

1. 为何不能将 const 引用赋给非 const 引用？
2. new 和 delete 是函数吗？
3. 指针变量包含的值有何特征？
4. 要访问指针指向的数据，应使用哪种运算符？

8.8.2 练习

1. 下面的语句显示什么？

```
0: int Number = 3;
1: int* pNum1 = &Number;
2: *pNum1 = 20;
3: int* pNum2 = pNum1;
4: Number *= 2;
5: cout << *pNum2;
```

2. 下面三个重载函数有何相同和不同之处？

```
int DoSomething(int Num1, int Num2);
int DoSomething(int& Num1, int& Num2);
int DoSomething(int* pNum1, int* pNum2);
```

3. 要让练习 1 中第 3 行的赋值非法，应如何修改第 1 行中 pNum1 的声明（提示：让 pNum1 不能修改它指向的数据）？

4. 查错：下面的代码有何错误？

```
#include <iostream>
using namespace std;
int main()
{
     int *pNumber = new int;
     pNumber = 9;
     cout << "The value at pNumber: " << *pNumber;
     delete pNumber;
     return 0;
}
```

5. 查错：下面的代码有何错误？

```
#include <iostream>
using namespace std;
int main()
{
    int pNumber = new int;
    int* pNumberCopy = pNumber;
    *pNumberCopy = 30;
    cout << *pNumber;
    delete pNumberCopy;
    delete pNumber;
    return 0;
}
```

6. 修正练习 5 的代码后，其输出是什么？

第9章
类和对象

至此，您探索了简单程序的结构。这种程序从 main()开始执行，让您能够声明局部和全局变量和常量、将执行逻辑划分为可接受参数和返回值的函数。这与 C 语言等过程型语言很像，不涉及面向对象。换句话说，您需要学习如何管理数据并将其与方法关联起来。

在本章中，您将学习：

- 什么是类；
- 类如何帮助您整合数据和处理数据的方法（类似于函数）；
- 构造函数、复制构造函数和析构函数；
- C++11 如何通过移动构造函数改进性能；
- 封装和抽象等面向对象的概念；
- this 指针；
- 结构是什么，它与类有何不同。

9.1 类和对象

假设您要编写一个模拟人（如您自己）的程序。人有其特征：姓名、出生日期、出生地和性别，还能做某些事情，如交谈、自我介绍等。前述特征是有关人的数据，而能做的事情是方法，如图 9.1 所示。

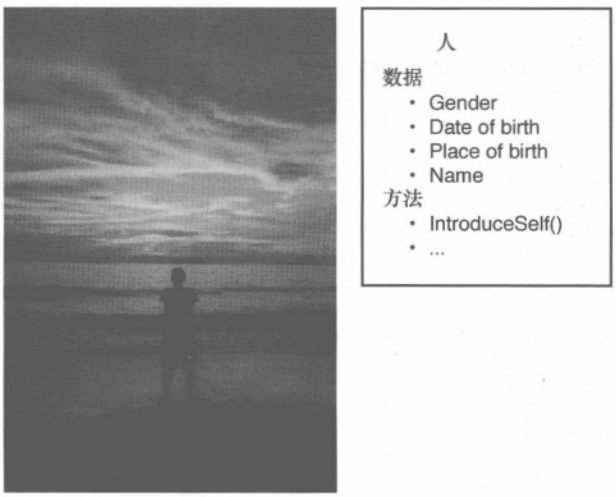


图 9.1　人的简单表示

要模拟人，需要一个结构，将定义人的属性（数据）以及人可使用这些属性执行的操作（类似于函数的方法）整合在一起。这种结构就是类。

9.1.1 声明类

要声明类，可使用关键字 class，并在它后面依次包含类名、一组放在{}内的成员属性和方法以及结尾的分号。

类声明类似于函数声明，将类本身及其属性告诉编译器。类声明本身并不能改变程序的行为，而必须使用它，就像需要调用函数一样。

模拟人类的类类似于下面这样（请暂时不要考虑其中的语法）：

```
class Human
{
   // Data attributes:
   string Name;
   string DateOfBirth;
   string PlaceOfBirth;
   string Gender;

   // Methods:
   void Talk(string TextToTalk);
   void IntroduceSelf();
   …
};
```

不用说，IntroduceSelf()将使用 Talk()以及整个在类 Human 中的一些数据。通过关键字 class，C++提供了一种功能强大的方式，让您能够创建自己的数据类型，并在其中封装属性和使用它们的函数。类的所有属性（这里是 Name、DateOfBirth、PlaceOfBirth 和 Gender）以及在其中声明的函数（Talk()和 IntroduceSelf()）都是类（Human）的成员。

封装指的是将数据以及使用它们的方法进行逻辑编组，这是面向对象编程的重要特征。

注意 方法就是属于类成员的函数。

9.1.2 实例化对象

类相当于蓝图，仅声明类并不会对程序的执行产生影响。在运行阶段，对象是类的化身。要使用类的功能，通常需要根据类实例化一个对象，并通过对象访问成员方法和属性。

实例化 Human 对象与创建其他类型（如 double）的实例类似：

```
double Pi = 3.1415;  // a double declared as a local variable (on stack)
Human Tom;  // An object of class Human declared as a local variable
```

就像为 int 动态分配内存一样，也可使用 new 为 Human 对象动态地分配内存：

```
int* pNumber = new int;  // an integer allocated dynamically on free store
delete pNumber;  // de-allocating the memory
Human* pAnotherHuman = new Human(); // dynamically allocated Human
delete pAnotherHuman;  // de-allocating memory allocated for a Human
```

9.1.3 使用句点运算符访问成员

一个人的例子是 Tom，男性，1970 年出生于阿拉巴马州。Tom 是 Human 类的对象，是这个类存在于现实世界（运行阶段）的化身：

```
Human Tom;  // an instance of Human
```

类声明表明，Tom 有 DateOfBirth 等属性，可使用句点运算符（.）来访问：

```
Tom.DateOfBirth = "1970";
```

这是因为从类声明表示的蓝图可知，属性 DateOfBirth 是类 Human 的一部分。仅当实例化了一个对象后，这个属性在现实世界（运行阶段）才存在。句点运算符（.）用于访问对象的属性。

这也适用于 IntroduceSelf()等方法：

```
Tom.IntroduceSelf();
```

如果有一个指针 pTom，它指向 Human 类的一个实例，则可使用指针运算符（->）来访问成员（这将在下一小节介绍），也可使用间接运算符（*）来获取对象，再使用句点运算符来访问成员：

```
Human* pTom = new Human();
(*pTom).IntroduceSelf();
```

9.1.4 使用指针运算符（->）访问成员

如果对象是使用 new 在自由存储区中实例化的，或者有指向对象的指针，则可使用指针运算符（->）来访问成员属性和方法：

```
Human* pTom = new Human();
pTom->DateOfBirth = "1970";
pTom->IntroduceSelf();
delete pTom;

// Alternatively when you have a pointer:
Human Tom;
Human* pTom = &Tom; // Assign address using reference operator&
pTom->DateOfBirth = "1970"; // is equivalent to Tom.DateOfBirth = "1970";
pTom->IntroduceSelf(); // is equivalent to Tom.IntroduceSelf();
```

程序清单 9.1 是一个值得编译的 Human 类，其中使用了关键字 private 和 public 等。

程序清单 9.1 一个值得编译的 Human 类

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:    string Name;
 8:    int Age;
 9:
10: public:
11:    void SetName (string HumansName)
12:    {
13:       Name = HumansName;
14:    }
15:
16:    void SetAge(int HumansAge)
17:    {
18:       Age = HumansAge;
19:    }
20:
21:    void IntroduceSelf()
22:    {
23:       cout << "I am " + Name << " and am ";
24:       cout << Age << " years old" << endl;
25:    }
26: };
```

```
27:
28: int main()
29: {
30:     // Constructing an object of class Human with attribute Name as "Adam"
31:     Human FirstMan;
32:     FirstMan.SetName("Adam");
33:     FirstMan.SetAge(30);
34:
35:     // Constructing an object of class Human with attribute Name as "Eve"
36:     Human FirstWoman;
37:     FirstWoman.SetName("Eve");
38:     FirstWoman.SetAge (28);
39:
40:     FirstMan.IntroduceSelf();
41:     FirstWoman.IntroduceSelf();
42: }
```

▼ **输出：**

```
I am Adam and am 30 years old
I am Eve and am 28 years old
```

▼ **分析：**

第 4～6 行说明了 C++类的基本构造。请从实用的角度浏览这些代码，忽略还不明白的术语和概念，它们将在本章后面详细介绍。请重点关注 Human 类的结构以及如何在 main()中使用这个类。

这个类包含两个私有（private）变量，其中一个名为 Name，类型为 string（第 7 行），另一个名为 Age，类型为 int（第 8 行）。还有几个使用这些私有变量的公有函数（也叫方法）：SetName()、SetAge()和 IntroduceSelf()，如第 11、16 和 21 行所示。在 main()中，第 31 和 36 行创建了两个 Human 对象。接下来的几行代码使用方法 SetName()和 SetAge()设置对象 FirstMan 和 FirstWoman 的成员变量，这些方法被称为存取器方法。注意到第 40 和 41 行对这两个对象调用了方法 IntroduceSelf()，该方法使用前面设置的成员变量现实两行输出。

您注意到了程序清单 9.1 中的关键字 private 和 public 吗？下面介绍 C++提供的这样的功能，即保护属性，对外部隐藏它们。

9.2 关键字 public 和 private

每个人都有很多个人信息，其中的有些向周围的人公开，如姓名。这样的信息被称为公有的。然而，有些个人信息您可能不想让外人知道，如收入。这种信息是私有的，通常保密。

C++让您能够将类属性和方法声明为公有的，这意味着有了对象后就可获取它们；也可将其声明为私有的，这意味着只能在类的内部（或其友元）中访问。作为类的设计者，您可使用 C++关键字 public 和 private 来指定哪些部分可从外部（如 main()）访问，哪些部分不能。

对程序员来说，能够将属性或方法声明为私有的（private）有何优点呢？请看下述 Human 类的声明（忽略除成员属性 Age 外的其他代码）：

```
class Human
{
private:
   // Private member data:
   int Age;
   string Name;

public:
   int GetAge()
```

```
    {
        return Age;
    }

    void SetAge(int InputAge)
    {
        Age = InputAge;
    }

// ...Other members and declarations
};
```

假设有一个名为 Eve 的 Human 实例：

```
Human Eve;
```

如果试图使用下述代码访问 Eve 的年龄：

```
cout << Eve.Age; // compile error
```

将出现类似于这样的编译错误："错误：Human::Age——不能访问 Human 类声明的私有成员"。要访问 Age，唯一的途径就是通过 Human 类的公有方法 GetAge()，这个方法以编写 Human 类的程序员认为的合适方式暴露 Age：

```
cout << Eve.GetAge(); // OK
```

如果编写 Human 类的程序员愿意，可以让 GetAge()显示的年龄比 Eve 的实际年龄小！换句话说，这意味着 C++允许类决定要暴露哪些属性以及如何暴露。如果 Human 类没有公有成员方法 GetAge()，就可确保用户根本无从查询 Age。在本章后面将介绍的情形下，这种功能很有用。

同样，也不能直接给 Human::Age 赋值：

```
Eve.Age = 22; // compile error
```

要设置年龄，唯一的途径是通过 SetAge()：

```
Eve.SetAge(22); // OK
```

这有很多优点。当前，SetAge()的实现只是直接设置成员变量 Human::Age，但也可在 SetAge()中验证外部输入，避免 Age 被设置为零或负数：

```
class Human
{
private:
    int Age;

public:
    void SetAge(int InputAge)
    {
        if (InputAge > 0)
            Age = InputAge;
    }
};
```

总之，C++让类的设计者能够控制类属性的访问和操纵方式。

9.2.1 使用关键字 private 实现数据抽象

除让您能够将类设计容器，以封装数据以及操作数据的方法外，C++还让您能够使用关键字 private 指定哪些信息不能从外部访问（即在类外不可用）。另外，可将方法声明为公有的（public），以便从外部通过这些方法访问私有信息。因此，类的实现可对外（其他类和 main()等函数）隐藏您认为它们无需知道的东西。

回到 Human 类，其中的 Age 是一个私有成员。您知道，在现实世界中，很多人不想公开自己的真实年龄。要 Human 类向外指出的年龄比实际年龄小两岁很容易，只需在公有方法 GetAge()中将 Age

减 2，再返回结果，如程序清单 9.2 所示。

程序清单 9.2 一个对外隐藏真实年龄并将自己说得更年轻的 Human 类

```
0: #include <iostream>
1: using namespace std;
2:
3: class Human
4: {
5: private:
6:    // Private member data:
7:    int Age;
8:
9: public:
10:   void SetAge(int InputAge)
11:   {
12:      Age = InputAge;
13:   }
14:
15:   // Human lies about his / her Age (if over 30)
16:   int GetAge()
17:   {
18:      if (Age > 30)
19:         return (Age - 2);
20:      else
21:         return Age;
22:   }
23: };
24:
25: int main()
26: {
27:   Human FirstMan;
28:   FirstMan.SetAge(35);
29:
30:   Human FirstWoman;
31:   FirstWoman.SetAge(22);
32:
33:   cout << "Age of FirstMan " << FirstMan.GetAge() << endl;
34:   cout << "Age of FirstWoman " << FirstWoman.GetAge() << endl;
35:
36:   return 0;
37: }
```

▼ 输出：

```
Age of FirstMan 33
Age of FirstWoman 22
```

▼ 分析：

请注意第 16 行的公有方法 Human::GetAge()。由于实际年龄存储在私有成员 Human::Age 中，而该成员不能从外部直接访问，因此外部用户要获悉 Human 对象的 Age 属性，唯一的途径是通过方法 GetAge()。也就是说，对外隐藏了存储在 Human::Age 中的实际年龄。

在面向对象编程语言中，抽象是一个非常重要的概念，让程序员能够决定哪些属性只能让类及其成员知道，类外的任何人都不能访问（友元除外）。

9.3 构造函数

构造函数是一种特殊的函数（方法），在创建对象时被调用。与函数一样，构造函数也可以重载。

9.3.1 声明和实现构造函数

构造函数是一种特殊的函数，它与类同名且不返回任何值。因此，Human 类的构造函数的声明类似于下面这样：

```
class Human
{
public:
   Human(); // declaration of a constructor
};
```

这个构造函数可在类声明中实现，也可在类声明外实现。在类声明中实现（定义）构造函数的代码类似于下面这样：

```
class Human
{
public:
   Human()
   {
       // constructor code here
   }
};
```

在类声明外定义构造函数的代码类似于下面这样：

```
class Human
{
public:
   Human(); // constructor declaration
};

// constructor definition (implementation)
Human::Human()
{
   // constructor code here
}
```

注意 ::被称为作用域解析运算符。例如，Human::DateOfBirth 指的是在 Human 类中声明的变量 DateOfBirth，而::DateOfBirth 表示全局作用域中的变量 DateOfBirth。

9.3.2 何时及如何使用构造函数

构造函数总是在创建对象时被调用，这让构造函数是将类成员变量（int、指针等）初始化为已知值的理想场所。再看一下程序清单 9.2。如果忘记调用 SetAge()，int 变量 Human::Age 将包含未知的垃圾值，因为该变量未初始化（请尝试将第 28 和 30 行注释掉）。程序清单 9.3 是一个更好的 Human 类版本，它使用构造函数初始化 Age。

程序清单 9.3 使用构造函数初始化类成员变量

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
5: {
6: private:
7:    // Private member data:
8:    string Name;
9:    int Age;
```

```
10:
11: public:
12:     // constructor
13:     Human()
14:     {
15:         Age = 0; // initialized to ensure no junk value
16:         cout << "Constructed an instance of class Human" << endl;
17:     }
18:
19:     void SetName (string HumansName)
20:     {
21:         Name = HumansName;
22:     }
23:
24:     void SetAge(int HumansAge)
25:     {
26:         Age = HumansAge;
27:     }
28:
29:     void IntroduceSelf()
30:     {
31:         cout << "I am " + Name << " and am ";
32:         cout << Age << " years old" << endl;
33:     }
34: };
35:
36: int main()
37: {
38:     Human FirstMan;
39:     FirstMan.SetName("Adam");
40:     FirstMan.SetAge(30);
41:
42:     Human FirstWoman;
43:     FirstWoman.SetName("Eve");
44:     FirstWoman.SetAge (28);
45:
46:     FirstMan.IntroduceSelf();
47:     FirstWoman.IntroduceSelf();
48: }
```

▼ 输出：

```
Constructed an instance of class Human
Constructed an instance of class Human
I am Adam and am 30 years old
I am Eve and am 28 years old
```

▼ 分析：

除与程序清单 9.1 相同的输出外，开头还新增了两行。来看看第 36～48 行的 main()，从中可知，新增的两行输出是第 38 和 42 行创建两个对象（FirstMan 和 FirstWoman）间接生成的。由于它们是 Human 类对象，创建它们将自动执行第 13～17 行定义的 Human 类构造函数。这个构造函数包含的 cout 语句生成了这些输出。注意到该构造函数还将 Age 初始化为零。如果您忘记给新创建的对象设置 Age 也没有关系，该构造函数将确保 Age 不是随机值而是零，这表明没有设置属性 Age。

注意

可在不提供参数的情况下调用的构造函数被称为默认构造函数。默认构造函数是可选的。如果您像程序清单 9.1 那样没有提供默认构造函数，编译器将为您创建一个，这种构造函数会创建成员属性，但不会初始化 POD 类型（如 int）的属性。

9.3.3 重载构造函数

与函数一样，构造函数也可重载，因此可创建一个将姓名作为参数的构造函数，如下所示：

```
class Human
{
public:
   Human()
   {
       // default constructor code here
   }

   Human(string HumansName)
   {
       // overloaded constructor code here
   }
};
```

程序清单 9.4 演示了重载构造函数的用途，它在创建 Human 对象时提供了姓名。

程序清单 9.4 包含多个构造函数的 Human 类

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:    // Private member data:
 8:    string Name;
 9:    int Age;
10:
11: public:
12:    // constructor
13:    Human()
14:    {
15:       Age = 0; // initialized to ensure no junk value
16:       cout << "Default constructor creates an instance of Human" << endl;
17:    }
18:
19:    // overloaded constructor that takes Name
20:    Human(string HumansName)
21:    {
22:       Name = HumansName;
23:       Age = 0; // initialized to ensure no junk value
24:       cout << "Overloaded constructor creates " << Name << endl;
25:    }
26:
27:    // overloaded constructor that takes Name and Age
28:    Human(string HumansName, int HumansAge)
29:    {
30:       Name = HumansName;
31:       Age = HumansAge;
32:       cout << "Overloaded constructor creates ";
33:       cout << Name << " of " << Age << " years" << endl;
34:    }
35:
36:    void SetName (string HumansName)
37:    {
38:       Name = HumansName;
```

```
39:    }
40:
41:    void SetAge(int HumansAge)
42:    {
43:       Age = HumansAge;
44:    }
45:
46:    void IntroduceSelf()
47:    {
48:       cout << "I am " + Name << " and am ";
49:      cout << Age << " years old" << endl;
50:    }
51: };
52:
53: int main()
54: {
55:    Human FirstMan; // use default constructor
56:    FirstMan.SetName("Adam");
57:    FirstMan.SetAge(30);
58:
59:    Human FirstWoman ("Eve"); // use overloaded constructor
60:    FirstWoman.SetAge (28);
61:
62:    Human FirstChild ("Rose", 1);
63:
64:    FirstMan.IntroduceSelf();
65:    FirstWoman.IntroduceSelf();
66:    FirstChild.IntroduceSelf();
67: }
```

▼ 输出：

```
Default constructor creates an instance of Human
Overloaded constructor creates Eve
Overloaded constructor creates Rose of 1 years
I am Adam and am 30 years old
I am Eve and am 28 years old
I am Rose and am 1 years old
```

▼ 分析：

Adam 是使用默认构造函数创建的；创建 Eve 时使用的是一个重载的构造函数，该构造函数接受一个 string 参数，并将其赋给 Human::Name；Rose 是使用第三个重载的构造函数创建的，该构造函数接受一个 string 参数和一个 int 参数，并将 int 参数赋给 Human::Age。这个程序表明，重载构造函数很有用，可帮助您初始化变量。

提示

> 您可选择不实现默认构造函数，从而要求实例化对象时必须提供某些参数。

9.3.4 没有默认构造函数的类

在程序清单 9.5 中，Human 类没有默认构造函数，要求创建 Human 对象时必须提供姓名和年龄。

程序清单 9.5 一个有重载的构造函数，但没有默认构造函数的类

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
```

```
 5: {
 6: private:
 7:    // Private member data:
 8:    string Name;
 9:    int Age;
10:
11: public:
12:
13:    // overloaded constructor (no default constructor)
14:    Human(string HumansName, int HumansAge)
15:    {
16:       Name = HumansName;
17:       Age = HumansAge;
18:       cout << "Overloaded constructor creates " << Name;
19:       cout << " of age " << Age << endl;
20:    }
21:
22:    void IntroduceSelf()
23:    {
24:       cout << "I am " + Name << " and am ";
25:      cout << Age << " years old" << endl;
26:    }
27: };
28:
29: int main()
30: {
31:    // Uncomment next line to try creating using a default constructor
32:    // Human FirstMan;
33:
34:    Human FirstMan("Adam", 30);
35:    Human FirstWoman("Eve", 28);
36:
37:    FirstMan.IntroduceSelf();
38:    FirstWoman.IntroduceSelf();
39: }
```

▼ 输出：

```
Overloaded constructor creates Adam of age 30
Overloaded constructor creates Eve of age 28
I am Adam and am 30 years old
I am Eve and am 28 years old
```

▼ 分析：

请注意 main()中的第 32 行，它与程序清单 9.3 中创建 FirstMan 的代码很像，但如果您取消对它的注释，程序将不能通过编译，出现的错误消息为：error:'Human':no appropriate default constructor available。这是因为这个版本的 Human 类只有一个构造函数，该构造函数接受一个 string 参数和一个 int 参数，如第 14 行所示。没有默认构造函数，而在您提供了重载的构造函数时，C++编译器不会为您生成默认构造函数。

这个示例表明，设计类时，可要求创建其对象时必须提供某些参数，就这里而言，是创建 Human 对象时必须提供 Name 和 Age。这个示例还表明，可在创建 Human 对象时提供 Name，且以后不能修改它。这是因为 Human 类的 Name 属性存储在私有 string 变量 Name 中，main()或其他不属于 Human 类成员的实体不能访问或修改它。换句话说，Human 类的用户创建每个对象时，都必须指定姓名（和年龄），且不能修改姓名，这与现实情况相当吻合，您说呢？您的姓名是父母在您出生时取的，别人可以知道，但任何人都无权修改（您自己除外）。

9.3.5 带默认值的构造函数参数

就像函数可以有带默认值的参数一样，构造函数也可以。在下面的代码中，对程序清单 9.5 中第 14 行的构造函数稍做了修改，给参数 Age 指定了默认值 25：

```
class Human
{
private:
   // Private member data:
   string Name;
   int Age;

public:
   // overloaded constructor (no default constructor)
   Human(string HumansName, int HumansAge = 25)
   {
      Name = HumansName;
      Age = HumansAge;
      cout << "Overloaded constructor creates " << Name;
      cout << " of age " << Age << endl;
   }

   // ... other members
};
```

实例化这个类的对象时，可使用下面的语法：

```
Human Adam("Adam"); // Adam.Age is assigned a default value 25
Human Eve("Eve, 18); // Eve.Age is assigned 18 as specified
```

注意

默认构造函数是调用时可不提供参数的构造函数，而并不一定是不接受任何参数的构造函数。因此，下面的构造函数虽然有两个参数，但它们都有默认值，因此也是默认构造函数：

```
class Human
{
private:
   // Private member data:
   string Name;
   int Age;

public:
   // Note default values for the two input parameters
   Human(string HumansName = "Adam", int HumansAge = 25)
   {
      Name = HumansName;
      Age = HumansAge;
      cout << "Overloaded constructor creates " << Name;
      cout << " of age " << Age << endl;
   }
};
```

因为实例化 Human 对象时仍可不提供任何参数：

```
Human Adam; // Human with default Name "Adam", Age 25
```

9.3.6 包含初始化列表的构造函数

您知道，构造函数对初始化成员很有用。另一种初始化成员的方式是使用初始化列表。对于程序

清单 9.5 中接受两个参数的构造函数，其包含初始化列表的变种类似于下面这样：

```
class Human
{
private:
   string Name;
   int Age;

public:
   // constructor takes two parameters to initialize members Age and Name
   Human(string InputName, int InputAge)
        :Name(InputName), Age(InputAge)
   {
      cout << "Constructed a Human called " << Name;
      cout << ", " << Age << " years old" << endl;
   }
// ... other class members
};
```

初始化列表由包含在括号中的参数声明后面的冒号标识，冒号后面列出了各个成员变量及其初始值。初始值可以是参数（如 InputName），也可以是固定的值。使用特定参数调用基类的构造函数时，初始化列表也很有用，这将在第 10 章讨论。

在程序清单 9.6 中，Human 类包含一个带初始化列表的默认构造函数，该默认构造函数的参数都有默认值。

程序清单 9.6 接受带默认值的参数的默认构造函数，并使用初始化列表来设置成员

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:    int Age;
 8:    string Name;
 9:
10: public:
11:    Human(string InputName = "Adam", int InputAge = 25)
12:         :Name(InputName), Age(InputAge)
13:    {
14:       cout << "Constructed a Human called " << Name;
15:       cout << ", " << Age << " years old" << endl;
16:    }
17: };
18:
19: int main()
20: {
21:    Human FirstMan;
22:    Human FirstWoman("Eve", 18);
23:
24:    return 0;
25: }
```

▼ 输出：

```
Constructed a Human called Adam, 25 years old
Constructed a Human called Eve, 18 years old
```

▼ 分析：

第 11～16 行的构造函数包含初始化列表，且用于设置 Name 和 Age 的参数分别包含默认值“Adam”

和 25。因此，第 21 行创建 Human 类的实例 FirstMan 时，自动将默认值赋给了其成员。另一方面，第 22 行创建 FirstWoman 时，给 Name 和 Age 显式地提供了值。

9.4 析构函数

与构造函数一样，析构函数也是一种特殊的函数。与构造函数不同的是，析构函数在对象销毁时自动被调用。

9.4.1 声明和实现析构函数

与构造函数一样，析构函数也看起来像一个与类同名的函数，但前面有一个波浪号（～）。因此，Human 类的析构函数的声明类似于下面这样：

```
class Human
{
   ~Human(); // declaration of a destructor
};
```

这个析构函数可在类声明中实现，也可在类声明外实现。在类声明中实现（定义）析构函数的代码类似于下面这样：

```
class Human
{
public:
   ~Human()
   {
       // destructor code here
   }
};
```

在类声明外定义析构函数的代码类似于下面这样：

```
class Human
{
public:
   ~Human(); // destructor declaration
};

// destructor definition (implementation)
Human::~Human()
{
   // destructor code here
}
```

正如您看到的，析构函数的声明与构造函数稍有不同，那就是包含波浪号（～）。然而，析构函数的作用与构造函数完全相反。

9.4.2 何时及如何使用析构函数

每当对象不再在作用域内或通过 delete 被删除，进而被销毁时，都将调用析构函数。这使得析构函数是重置变量以及释放动态分配的内存和其他资源的理想场所。

使用 C 风格 char 缓冲区时，您必须自己管理内存分配等，因此本书始终建议不要使用它们，而使用 std::string。std::string 等工具都是类，它们充分利用了构造函数和析构函数，还有将在第 12 章介绍的运算符。程序清单 9.7 所示的类 MyString 在构造函数中为一个字符串分配内存，并在析构函数中释放它。

程序清单 9.7 一个简单的类，它封装了一个 C 风格字符串并通过析构函数释放它

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MyString
 4: {
 5: private:
 6:    char* Buffer;
 7:
 8: public:
 9:    // Constructor
10:    MyString(const char* InitialInput)
11:    {
12:       if(InitialInput != NULL)
13:       {
14:          Buffer = new char [strlen(InitialInput) + 1];
15:          strcpy(Buffer, InitialInput);
16:       }
17:       else
18:          Buffer = NULL;
19:    }
20:    // Destructor: clears the buffer allocated in constructor
21:    ~MyString()
22:    {
23:       cout << "Invoking destructor, clearing up" << endl;
24:       if (Buffer != NULL)
25:          delete [] Buffer;
26:    }
27:
28:    int GetLength()
29:    {
30:       return strlen(Buffer);
31:    }
32:
33:    const char* GetString()
34:    {
35:        return Buffer;
36:    }
37: }; // end of class MyString
38:
39: int main()
40: {
41:    MyString SayHello("Hello from String Class");
42:    cout << "String buffer in MyString is " << SayHello.GetLength();
43:    cout << " characters long" << endl;
44:
45:    cout << "Buffer contains: ";
46:    cout << "Buffer contains: " << SayHello.GetString() << endl;
47: }
```

▼ 输出：

```
String buffer in MyString is 23 characters long
Buffer contains: Hello from String Class
Invoking destructor, clearing up
```

▼ 分析：

这个类封装了一个 C 风格字符串（MyString::Buffer），让您使用字符串时无需分配和释放内存。我们最感兴趣的是第 10～19 行的构造函数 MyString()以及第 21～26 行的析构函数～MyString()。这个

构造函数构造 MyString 对象。它通过输入参数获取一个输入字符串；然后使用 strlen 确定输入字符串的长度并为 C 风格字符串 Buffer 分配内存（第 14 行）；再使用 strcpy 将输入字符串复制到新分配的内存中（第 15 行）。如果传递给参数 InitialInput 的值为 NULL，MyString::Buffer 也被初始化为 NULL（以防该指针包含随机值，否则使用它来访问内存单元将非常危险）。析构函数的代码确保构造函数分配的内存自动被归还给系统。它检查 MyString::Buffer 是否为 NULL，如果不是，则对其执行 delete[]，这对应于构造函数中的 new[...]。注意到在 main()中，程序员无需调用 new 和 delete。这个类不仅对程序员隐藏了实现，还正确地释放了分配的内存。main()执行完毕时，将自动调用析构函数~MyString()，输出证明了这一点——其中包含析构函数中 cout 语句的输出。

类更好地处理了字符串，这是析构函数的众多用途之一。在更智能地使用指针方面，析构函数也扮演了重要角色，第 26 章将演示这一点。

注意

> 析构函数不能重载，每个类都只能有一个析构函数。如果您忘记了实现析构函数，编译器将创建一个伪（dummy）析构函数并调用它。伪析构函数为空，即不释放动态分配的内存。

9.5 复制构造函数

第 7 章介绍过，对于程序清单 7.1 中的函数 Area()，传递的实参被复制：

```
double Area(double InputRadius);
```

因此，调用 Area()时，实参被复制给形参 InputRadius。这种规则也适用于对象（类的实例）。

9.5.1 浅复制及其存在的问题

程序清单 9.7 所示的 MyString 类包含一个指针成员，它指向动态分配的内存（这些内存是在构造函数中使用 new 分配的，并在析构函数中使用 delete[]进行释放）。复制这个类的对象时，将复制其指针成员，但不复制指针指向的缓冲区，其结果是，两个对象指向同一块动态分配的内存。这被称为浅复制，会威胁程序的稳定性，如程序清单 9.8 所示。

程序清单 9.8 按值传递类（如 MyString）的对象带来的问题

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MyString
 4: {
 5: private:
 6:    char* Buffer;
 7:
 8: public:
 9:    // Constructor
10:    MyString(const char* InitialInput)
11:    {
12:       if(InitialInput != NULL)
13:       {
14:          Buffer = new char [strlen(InitialInput) + 1];
15:          strcpy(Buffer, InitialInput);
16:       }
17:       else
18:          Buffer = NULL;
19:    }
20:
```

```
21:    // Destructor
22:    ~MyString()
23:    {
24:       cout << "Invoking destructor, clearing up" << endl;
25:       if (Buffer != NULL)
26:          delete [] Buffer;
27:    }
28:
29:    int GetLength()
30:    {
31:       return strlen(Buffer);
32:    }
33:
34:    const char* GetString()
35:    {
36:       return Buffer;
37:    }.
38: };
39:
40: void UseMyString(MyString Input)
41: {
42:    cout << "String buffer in MyString is " << Input.GetLength();
43:    cout << " characters long" << endl;
44:
45:    cout << "Buffer contains: " << Input.GetString() << endl;
46:    return;
47: }
48:
49: int main()
50: {
51:    MyString SayHello("Hello from String Class");
52:
53:    // Pass SayHello as a parameter to the function
54:    UseMyString(SayHello);
55:
56:    return 0;
57: }
```

▼ 输出：

```
String buffer in MyString is 23 characters long
Buffer contains: Hello from String Class
Invoking destructor, clearing up
Invoking destructor, clearing up
<crash as seen in Figure 9.2>
```

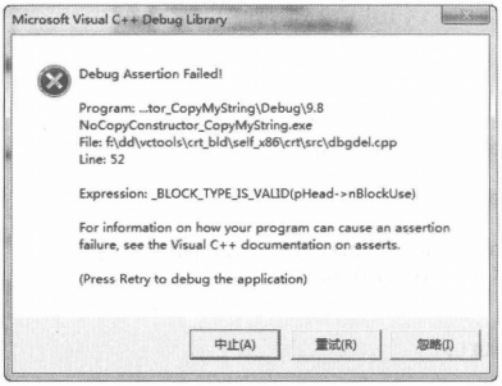


图 9.2 在 MS Visual Studio 调试模式下执行程序清单 9.8 时出现的崩溃屏幕

▼ **分析：**

在程序清单 9.7 中，这个类运行正常，为何会导致程序清单 9.8 崩溃呢？相比于程序清单 9.7，程序清单 9.8 唯一不同的地方在于，在 main()中，将使用 MyString 对象 SayHello 的工作交给了函数 UseMyString，如第 54 行所示。在 main()中将工作交给这个函数的结果是，对象 SayHello 被复制到形参 Input，并在 UseMyString()中使用它。编译器之所以进行复制，是因为函数 SayHello 的参数 Input 被声明为按值（而不是按引用）传递。对于整型、字符和原始指针等 POD 数据，编译器执行二进制复制，因此 SayHello.Buffer 包含的指针值被复制到 Input 中，即 SayHello.Buffer 和 Input.Buffer 指向同一个内存单元，如图 9.3 所示。

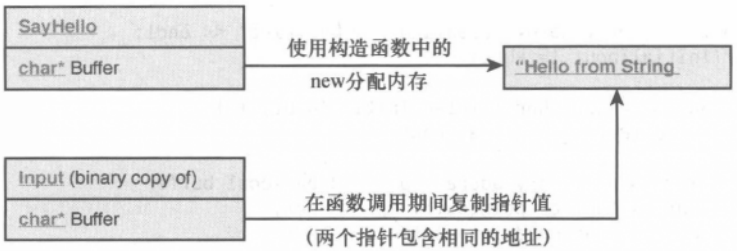


图 9.3 调用 UseMyString()时，SayHello 被浅复制到 Input 中

二进制复制并不深复制指向的内存单元，这导致两个 MyString 对象指向同一个内存单元。函数 UseMyString()返回时，变量 Input 不再在作用域内，因此被销毁。为此，将调用 MyString 类的析构函数，而该析构函数使用 delete 释放分配给 Buffer 的内存（如程序清单 9.8 的第 26 行所示）。这将导致 main()中的对象 SayHello 指向的内存无效，而等 main()执行完毕时，SayHello 将不再在作用域内，进而被销毁。然而，第 26 行对不再有效的内存地址调用 delete（销毁 Input 时释放了该内存，导致它无效）。图 9.2 的调试断言消息指出错误出在第 52 行（本书的行号从零开始，因此是第 51 行），因为未能成功地销毁这里创建的对象 SayHello。

注意

在这里，编译器没有进行深复制，因为编译时它不确定指针成员 MyString::Buffer 指向的是多少字节的内存。

9.5.2 使用复制构造函数确保深复制

复制构造函数是一个特殊的重载构造函数，编写类的程序员必须提供它。每当对象被复制（包括将对象按值传递给函数）时，编译器都将调用复制构造函数。

为 MyString 类声明复制构造函数的语法如下：

```
class MyString
{
   MyString(const MyString& CopySource); // copy constructor
};

MyString::MyString(const MyString& CopySource)
{
   // Copy constructor implementation code
}
```

复制构造函数接受一个以引用方式传入的当前类的对象作为参数。这个参数是源对象的别名，您使用它来编写自定义的复制代码，确保对所有缓冲区进行深复制，如程序清单 9.9 所示。

程序清单 9.9 定义一个复制构造函数，确保对动态分配的缓冲区进行深复制

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

class MyString
{
private:
   char* Buffer;

public:
   // constructor
   MyString(const char* InitialInput)
   {
      cout << "Constructor: creating new MyString" << endl;
      if(InitialInput != NULL)
      {
         Buffer = new char [strlen(InitialInput) + 1];
         strcpy(Buffer, InitialInput);

         // Display memory address pointed by local buffer
         cout << "Buffer points to: 0x" << hex;
         cout << (unsigned int*)Buffer << endl;
      }
      else
         Buffer = NULL;
   }

   // Copy constructor
   MyString(const MyString& CopySource)
   {
      cout << "Copy constructor: copying from MyString" << endl;

      if(CopySource.Buffer != NULL)
      {
         // ensure deep copy by first allocating own buffer
         Buffer = new char [strlen(CopySource.Buffer) + 1];

         // copy from the source into local buffer
         strcpy(Buffer, CopySource.Buffer);

         // Display memory address pointed by local buffer
         cout << "Buffer points to: 0x" << hex;
         cout << (unsigned int*)Buffer << endl;
      }
      else
         Buffer = NULL;
   }

   // Destructor
   ~MyString()
   {
      cout << "Invoking destructor, clearing up" << endl;
      if (Buffer != NULL)
         delete [] Buffer;
   }

   int GetLength()
   {
      return strlen(Buffer);
   }

```

```
60:     const char* GetString()
61:     {
62:        return Buffer;
63:     }
64: };
65:
66: void UseMyString(MyString Input)
67: {
68:     cout << "String buffer in MyString is " << Input.GetLength();
69:     cout << " characters long" << endl;
70:
71:     cout << "Buffer contains: " << Input.GetString() << endl;
72:     return;
73: }
74:
75: int main()
76: {
77:     MyString SayHello("Hello from String Class");
78:
79:     // Pass SayHello by value (will be copied)
80:     UseMyString(SayHello);
81:
82:     return 0;
83: }
```

▼ 输出：

```
Constructor: creating new MyString
Buffer points to: 0x0040DA68
Copy constructor: copying from MyString
Buffer points to: 0x0040DAF8
String buffer in MyString is 17 characters long
Buffer contains: Hello from String Class
Invoking destructor, clearing up
Invoking destructor, clearing up
```

▼ 分析：

大多数代码都与程序清单 9.8 类似，只是构造函数新增了多行 cout 语句，还新增了一个复制构造函数（第 27～45 行）。首先，将重点放在 main()上。与前一个示例一样，它创建了对象 SayHello，如第 77 行所示。创建 SayHello 导致了第 1 行输出，这是由 MyString 的构造函数的第 12 行生成的。出于方便考虑，这个构造函数还显示了 Buffer 指向的内存地址。接下来，main()将 SayHello 按值传递个函数 UseMyString()，如第 80 行所示，这将自动调用复制构造函数，输出指出了这一点。复制构造函数的代码与构造函数很像，基本思想也相同：检查 CopySource.Buffer 包含的 C 风格字符串的长度（第 34 行），分配相应数量的内存并将返回的指针赋给 Buffer，再使用 strcpy 将 CopySource.Buffer 的内容复制到 Buffer（第 37 行）。这里并非浅复制（复制指针的值），而是深复制，即将指向的内容复制到给当前对象新分配的缓冲区中，如图 9.4 所示。

图 9.4 调用函数 UseMyString()时，将实参 SayHello 深复制到形参 Input 中

程序清单 9.9 的输出表明，拷贝中的 Buffer 指向内存地址不同，即两个对象并未指向同一个动态分配的内存地址。因此，函数 UseMyString()返回、形参 Input 被销毁时，析构函数对复制构造函数分配的内存地址调用 delete[]，而没有影响 main()中 SayHello 指向的内存。因此，这两个函数都执行完毕时，成功地销毁了各自的对象，没有导致应用程序崩溃。

注意

复制构造函数确保下面的函数调用进行深复制：

```
MyString SayHello("Hello from String Class");
UseMyString(SayHello);
```

然而，如果您通过赋值进行复制时，结果如何呢？

```
MyString overwrite("who cares?");
overwrite = SayHello;
```

由于您没有指定任何赋值运算符，编译器提供的默认赋值运算符将导致浅复制。为避免赋值时进行浅复制，您需要实现复制赋值运算符（=）。

复制赋值运算符将在第 12 章深入讨论。程序清单 12.9 是改进后的 MyString，它实现了复制赋值运算符：

```
MyString::operator= (const MyString& CopySource)
{
   //... copy assignment operator code
}
```

警告

通过在复制构造函数声明中使用 const，可确保复制构造函数不会修改指向的源对象。另外，复制构造函数的参数必须按引用传递，否则调用它时将复制实参的值，导致对源数据进行浅复制——这正是您要极力避免的。

应该	不应该
类包含原始指针成员（char *等）时，务必编写复制构造函数和复制赋值运算符。 编写复制构造函数时，务必将接受源对象的参数声明为 const 引用。 务必将类成员声明为 std::string 和智能指针类（而不是原始指针），因为它们实现了复制构造函数，可减少您的工作量。	除非万不得已，不要类成员声明为原始指针。

注意

MyString 类包含原始指针成员 char* Buffer，这里使用它旨在阐述为何需要复制构造函数。如果您编写类时需要包含字符串成员，用于存储姓名等，应使用 std::string 而不是 char*。在没有使用原始指针的情况下，您都不需要编写复制构造函数。这是因为编译器添加的默认复制构造函数将调用成员对象（如 std::string）的复制构造函数。

C++11

9.5.3 有助于改善性能的移动构造函数

由于 C++的特征和需求，有些情况下对象会自动被复制。请看下面的代码：

```
class MyString
{
   // pick implementation from Listing 9.9
};
MyString Copy(MyString& Source)
```

```
{
   MyString CopyForReturn(Source.GetString());  // create copy
   return CopyForReturn;  // return by value invokes copy constructor
}
int main()
{
   MyString sayHello ("Hello World of C++");
   MyString sayHelloAgain(Copy(sayHello)); // invokes 2x copy constructor

   return 0;
}
```

正如注释指出的，实例化 sayHelloAgain 时，由于调用了函数 Copy(sayHello)，而它按值返回一个 MyString，因此调用了复制构造函数两次。然而，这个返回的值存在时间很短，且在该表达式外不可用。因此，C++编译器严格地调用复制构造函数反而降低了性能，如果复制的是很大的动态对象数组，对性能的影响将很大。

为避免这种性能瓶颈，除编写复制构造函数，还应编写一个移动构造函数。移动构造函数的语法如下：

```
// move constructor
MyString(MyString&& MoveSource)
{
   if(MoveSource.Buffer != NULL)
   {
      Buffer = MoveSource.Buffer; // take ownership i.e.  'move'
      MoveSource.Buffer = NULL;   // set the move source to NULL i.e. free it
   }
}
```

有移动复制构造函数时，C++11 编译器将自动使用它来“移动”临时资源，从而避免深复制。实现移动构造函数后，应将前面的注释改成下面这样：

```
   MyString sayHelloAgain(Copy(sayHello)); // invokes 1x copy constructor, 1x
move constructor
```

移动构造函数通常是利用移动赋值运算符实现的，这将在第 12 章更详细地讨论。程序清单 12.12 是一个更好的 MyString 版本，实现了移动构造函数和移动赋值运算符。

9.6 构造函数和析构函数的其他用途

本章介绍了几个重要的基本概念，如构造函数、析构函数以及使用关键字 public 和 private 等抽象数据和方法。等您设计类时，这些概念让您能够控制其对象的创建、复制和销毁方式，还可控制其数据的暴露方式。

下面介绍几种有趣的模式，它们可帮助您解决众多重要的设计问题。

9.6.1 不允许复制的类

假设您需要模拟国家的政体。一个国家只能有一位总统，而 President 类面临如下风险：

```
President OurPresident;
DoSomething(OurPresident); // duplicate created in passing by value
President clone;
clone = OurPresident; // duplicate via assignment
```

显然，需要避免这样的情况发生。编写操作系统时，您需要模拟一个局域网、一个处理器等，为此需要避免这样的资源被复制。如果您不声明复制构造函数，C++将为您添加一个公有的默认复制构造函数，这破坏了您的设计，威胁着您的实现。然而，C++提供了实现这种设计范式的解决方案。

要禁止类对象被复制，可声明一个私有的复制构造函数。这确保函数调用 DoSomething(OurPresident) 无法通过编译。为禁止赋值，可声明一个私有的赋值运算符。

因此，解决方案如下：

```
class President
{
private:
   President(const President&); // private copy constructor
   President& operator= (const President&); // private copy assignment operator

   // … other attributes
};
```

无需给私有复制构造函数和私有赋值运算符提供实现，只需将它们声明为私有的就足以实现您的目标：确保 President 的对象是不可复制的。

9.6.2 只能有一个实例的单例类

前面讨论的 President 类很不错，但存在一个缺陷：无法禁止通过实例化多个对象来创建多名总统：

```
President One, Two, Three;
```

由于复制构造函数是私有的，其中每个对象都是不可复制的，但您的目标是确保 President 类有且只有一个化身，即有了一个 President 对象后，就禁止创建其他的 President 对象。要实现这种功能强大的模式，可使用单例的概念，它使用私有构造函数、私有赋值运算符和静态实例成员。

提示

> 将关键字 static 用于类的数据成员时，该数据成员将在所有实例之间共享。
> 将 static 用于函数中声明的局部变量时，该变量的值将在两次调用之间保持不变。
> 将 static 用于成员函数（方法）时，该方法将在所有成员之间共享。

要创建单例类，关键字 static 必不可少，如程序清单 9.10 所示。

程序清单 9.10 单例类 President，它禁止复制、赋值以及创建多个实例

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class President
 5: {
 6: private:
 7:    // private default constructor (prohibits creation from outside)
 8:    President() {};
 9:
10:    //  private copy constructor (prohibits copy creation)
11:    President(const President&);
12:
13:    // private assignment operator (prohibits assignment)
14:    const President& operator=(const President&);
15:
16:    // member data: Presidents name
17:    string Name;
18:
19: public:
20:    // controlled instantiation
21:    static President& GetInstance()
22:    {
23:       // static objects are constructed only once
24:       static President OnlyInstance;
```

```
25:
26:        return OnlyInstance;
27:    }
28:
29:    // public methods
30:    string GetName()
31:    {
32:        return Name;
33:    }
34:
35:    void SetName(string InputName)
36:    {
37:        Name = InputName;
38:    }
39: };
40:
41: int main()
42: {
43:    President& OnlyPresident = President::GetInstance();
44:    OnlyPresident.SetName("Abraham Lincoln");
45:
46:    // uncomment lines to see how compile failures prohibit duplicates
47:    // President Second; // cannot access constructor
48:    // President* Third= new President(); // cannot access constructor
49:    // President Fourth = OnlyPresident; // cannot access copy constructor
50:    // OnlyPresident = President::GetInstance(); // cannot access operator=
51:
52:    cout << "The name of the President is: ";
53:    cout << President::GetInstance().GetName() << endl;
54:
55:    return 0;
56: }
```

▼ 输出：

```
The name of the President is: Abraham Lincoln
```

▼ 分析：

main()包含几行代码和大量注释，演示了各种创建 President 实例和拷贝的方式，这些方式都行不通。下面逐一进行分析。

```
47:    // President Second; // cannot access constructor
48:    // President* Third= new President(); // cannot access constructor
```

第 47 和 48 行分别试图使用默认构造函数在堆和栈上创建对象，但默认构造函数不可用，因为它是私有的，如第 8 行所示。

```
49:    // President Fourth = OnlyPresident; // cannot access copy constructor
```

第 49 行试图使用复制构造函数创建现有对象的拷贝（在创建对象的同时为其赋值，将调用复制构造函数），但在 main()中不能使用复制构造函数，因为第 11 行将其声明成了私有的。

```
50:    // OnlyPresident = President::GetInstance(); // cannot access operator=
```

第 50 行试图通过赋值创建对象的拷贝，但行不通，因为第 14 行将赋值运算符声明成了私有的。因此，在 main()中，不能创建 President 类的实例，唯一的方法是使用静态函数 GetInstance()来获取 President 的实例，如第 43 行所示。GetInstance()是静态成员，类似于全局函数，无需通过对象来调用它。GetInstance()是在第 21～27 行实现的，它使用静态变量 OnlyInstance 确保有且只有一个 President 实例。为更好地理解这一点，可以认为第 24 行只执行一次（静态初始化），因此 GetInstance()返回唯一一个 President 实例，而不管您如何使用该实例，如 main()中的第 43

和 53 行所示。

警告

为方便扩充应用程序的功能，仅在绝对必要时才使用单例模式。单例模式禁止创建多个实例，在需要多个实例时，这将变成架构瓶颈。
例如，如果应用程序从模拟一个国家升级到模拟联合国（当前，联合国有 192 位成员，因此有 192 位总统），单例模式将是个架构问题。

9.6.3 禁止在栈中实例化的类

栈空间通常有限。如果您要编写一个数据库类，其内部结构包含 1GB 数据，可能应该禁止在栈上实例化它，而只允许在堆上创建其实例。为此，关键在于将析构函数声明为私有的：

```
class MonsterDB
{
private:
   // private destructor
   ~MonsterDB();
   //… collections that reserve a huge amount of data
};
```

这样，便不能像下面这样创建 MonsterDB 类的实例：

```
int main()
{
   MonsterDB myDatabase; // compile error
   // … more code
   return 0;
}
```

上述代码试图在栈上创建实例。编译器知道，当 myDatabase 不再在作用域内时，需要将其销毁，因此编译器自动在 main()末尾调用析构函数，但该析构函数是私有的，即不可用，因此上述语句将导致编译错误。

然而，将析构函数声明为私有的并不能禁止在堆中实例化：

```
int main()
{
   MonsterDB* myDatabase = new MonsterDB(); // no error
   // … more code
   return 0;
}
```

上述代码将导致内存泄露。由于在 main 中不能调用析构函数，因此也不能调用 delete。为解决这种问题，需要在 MonsterDB 类中提供一个销毁实例的静态公有函数（作为类成员，它能够调用析构函数），如程序清单 9.11 所示。

程序清单 9.11 数据库类 MonsterDB，只能使用 new 在自由存储区中创建其对象

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MonsterDB
 4: {
 5: private:
 6:    ~MonsterDB() {}; // private destructor
 7:
 8: public:
 9:    static void DestroyInstance(MonsterDB* pInstance)
10:    {
11:       // static member can access private destructor
12:       delete pInstance;
```

```
13:     }
14:
15:     // ... imagine a few other methods
16: };
17:
18: int main()
19: {
20:     MonsterDB* pMyDatabase = new MonsterDB();
21:
22:     // pMyDatabase -> member methods (....);
23:
24:     // uncomment next line to see compile failure
25:     // delete pMyDatabase; // private destructor cannot be invoked
26:
27:     // use static member to deallocate
28:     MonsterDB::DestroyInstance(pMyDatabase);
29:
30:     return 0;
31: }
```

▼ 分析：

这些代码旨在演示如何创建禁止在栈中实例化的类。为此，将构造函数声明成了私有的，如第 6 行所示。另外，它还包含静态函数 DestroyInstance()，如第 9～13 行所示；在 main()中，第 28 行调用了该静态函数。

9.7 this 指针

在 C++中，一个重要的概念是保留的关键字 this。在类中，关键字 this 包含当前对象的地址，换句话说，其值为&object。当您在类成员方法中调用其他成员方法时，编译器将隐式地传递 this 指针——函数调用中不可见的参数：

```
class Human
{
private:
// … private member declarations
    void Talk (string Statement)
   {
      cout << Statement;
   }

public:

   void IntroduceSelf()
   {
      Talk("Bla bla");
   }
};
```

在这里，方法 IntroduceSelf()使用私有成员 Talk()在屏幕上显示一句话。实际上，编译器将在调用 Talk 时嵌入 this 指针，即 Talk(this, “Blab la”)。

从编程的角度看，this 的用途不多，且大多数情况下都是可选的。例如，在程序清单 9.1 中，可将 SetAge()中访问 Age 的代码修改成下面这样：

```
void SetAge(int HumansAge)
{
   this->Age = HumansAge;    // same as Age = HumansAge
}
```

注意

调用静态方法时，不会隐式地传递 this 指针，因为静态函数不与类实例相关联，而由所有实例共享。
如果要在静态函数中使用实例变量，应显式地声明一个形参，让调用者将实参设置为 this 指针。

9.8 将 sizeof()用于类

您知道，通过使用关键字 class 声明自定义类型，可封装数据属性和使用数据的方法。第 3 章介绍过，运算符 sizeof()用于确定指定类型需要多少内存（单位为字节）。这个运算符也可用于类，在这种情况下，它将指出类声明中所有数据属性占用的总内存量（单位为字节）。sizeof()可能对某些属性进行填充，使其与字边界对齐，也可能不这样做，这取决于您使用的编译器。用于类时，sizeof()不考虑成员函数及其定义的局部变量，如程序清单 9.12 所示。

程序清单 9.12 将 sizeof 用于类及其实例的结果

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

class MyString
{
private:
   char* Buffer;

public:
   // Constructor
   MyString(const char* InitialInput)
   {
      if(InitialInput != NULL)
      {
         Buffer = new char [strlen(InitialInput) + 1];
         strcpy(Buffer, InitialInput);
      }
      else
         Buffer = NULL;
   }

   // Copy Constructor
   MyString(const MyString& CopySource)
   {
      if(CopySource.Buffer != NULL)
      {
          Buffer = new char [strlen(CopySource.Buffer) + 1];
          strcpy(Buffer, CopySource.Buffer);
      }
      else
         Buffer = NULL;
   }

   ~MyString()
   {
      if (Buffer != NULL)
         delete [] Buffer;
   }

   int GetLength()
   {
      return strlen(Buffer);
```

```
42:     }
43:
44:     const char* GetString()
45:     {
46:         return Buffer;
47:     }
48: };
49:
50: class Human
51: {
52: private:
53:     int Age;
54:     bool Gender;
55:     MyString Name;
56:
57: public:
58:     Human(const MyString& InputName, int InputAge, bool InputGender)
59:         : Name(InputName), Age (InputAge), Gender(InputGender) {}
60:
61:     int GetAge ()
62:     {
63:         return Age;
64:     }
65: };
66:
67: int main()
68: {
69:     MyString FirstMan("Adam");
70:     MyString FirstWoman("Eve");
71:
72:     cout << "sizeof(MyString) = " << sizeof(MyString) << endl;
73:     cout << "sizeof(FirstMan) = " << sizeof(FirstMan) << endl;
74:     cout << "sizeof(FirstWoman) = " << sizeof(FirstWoman) << endl;
75:
76:     Human FirstMaleHuman(FirstMan, 25, true);
77:     Human FirstFemaleHuman(FirstWoman, 18, false);
78:
79:     cout << "sizeof(Human) = " << sizeof(Human) << endl;
80:     cout << "sizeof(FirstMaleHuman) = " << sizeof(FirstMaleHuman) << endl;
81:     cout << "sizeof(FirstFemaleHuman) = " << sizeof(FirstFemaleHuman) <<
endl;
82:
83:     return 0;
84: }
```

▼ 输出：

```
sizeof(MyString) = 4
sizeof(FirstMan) = 4
sizeof(FirstWoman) = 4
sizeof(Human) = 12
sizeof(FirstMaleHuman) = 12
sizeof(FirstFemaleHuman) = 12
```

▼ 分析：

这个示例很长，它包含程序清单 9.9 所示的 MyString 类（省略了大部分显示文本的语句）。还包含 Human 类，这个类使用 MyString 对象来存储姓名（Name），并新增了 bool 数据成员 Gender。

首先来分析输出。从中可知，将 sizeof()用于类及其对象时，结果相同。sizeof(MyString)和 sizeof(FirstMan)的值相同，因为类占用的字节数在编译阶段就已确定，且在设计时就知道。虽然

FirstMan 包含 Adam，而 FirstWoman 包含 Eve，但它们占用的字节数相同，这没什么可奇怪的，因为存储姓名的 MyString::Buffer 是一个 char *，这是一个大小固定的指针（在我使用的 32 位系统中，为 4 字节），而与指向的数据量无关。

将 sizeof()用于 Human 时，结果为 12。第 53～55 行表明，Human 包含一个 int 成员、一个 bool 成员和一个 MyString 成员。要获悉内置类型占用的字节数，请参阅程序清单 3.4。从该程序清单可知，int 占用 4 字节，bool 占用 1 字节，而 MyString 占用 4 字节。它们的总和与输出中的 12 字节不符，这是因为 sizeof()的结果受字填充（word padding）和其他因素的影响。

9.9 结构不同于类的地方

关键字 struct 来自 C 语言，在 C++编译器看来，它与类及其相似，差别在于程序员未指定时，默认的访问限定符（public 和 private）不同。因此，除非指定了，否则结构中的成员默认为公有的（而类成员默认为私有的）；另外，除非指定了，否则结构以公有方式继承基结构（而类为私有继承）。继承将在第 10 章详细讨论。

对于程序清单 9.12 所示的 Human 类，对应的结构如下：

```
struct Human
{
   // constructor, public by default (as no access specified is mentioned)
   Human(const MyString& InputName, int InputAge, bool InputGender)
      : Name(InputName), Age (InputAge), Gender(InputGender) {}

   int GetAge ()
   {
      return Age;
   }

private:
   int Age;
   bool Gender;
   MyString Name;
};
```

正如您看到的，结构 Human 与类 Human 很像；结构的实例化与类的实例化也很像：

```
Human FirstMan("Adam", 25, true); // is an instance of struct Human
```

9.10 声明友元

不能从外部访问类的私有数据成员和方法，但这条规则不适用于友元类和友元函数。要声明友元类或友元函数，可使用关键字 friend，如程序清单 9.13 所示。

程序清单 9.13 使用关键字 friend 让外部函数 DisplayAge()能够访问私有数据成员

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:    string Name;
 8:    int Age;
 9:
10:    friend void DisplayAge(const Human& Person);
```

```
11:
12: public:
13:     Human(string InputName, int InputAge)
14:     {
15:         Name = InputName;
16:         Age = InputAge;
17:     }
18: };
19:
20: void DisplayAge(const Human& Person)
21: {
22:     cout << Person.Age << endl;
23: }
24:
25: int main()
26: {
27:     Human FirstMan("Adam", 25);
28:     cout << "Accessing private member Age via friend: ";
29:     DisplayAge(FirstMan);
30:
31:     return 0;
32: }
```

▼ 输出：

```
Accessing private member Age via friend: 25
```

▼ 分析：

第 10 行的声明告诉编译器，函数 DisplayAge()是全局函数，可访问 Human 类的私有数据成员。如果将第 10 行注释掉，第 22 行将导致编译错误。

与函数一样，也可将外部类指定为可信任的朋友，如程序清单 9.14 所示。

程序清单 9.14　使用关键字 friend 让外部类 Utility 能够访问私有数据成员

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:     string Name;
 8:     int Age;
 9:
10:     friend class Utility;
11:
12: public:
13:     Human(string InputName, int InputAge)
14:     {
15:         Name = InputName;
16:         Age = InputAge;
17:     }
18: };
19:
20: class Utility
21: {
22: public:
23:     static void DisplayAge(const Human& Person)
24:     {
25:         cout << Person.Age << endl;
```

```
26:     }
27: };
28:
29: int main()
30: {
31:     Human FirstMan("Adam", 25);
32:     cout << "Accessing private member Age via friend class: ";
33:     Utility::DisplayAge(FirstMan);
34:
35:     return 0;
36: }
```

▼ **输出：**

```
Accessing private member Age via friend class: 25
```

▼ **分析：**

第 10 行指出 Utility 类是 Human 类的友元，这让 Utility 类的所有方法都能访问 Human 类的私有数据成员和方法。

9.11 总结

本章介绍了最重要的 C++关键字和概念——class（类）。您了解到，类封装了成员数据以及使用这些数据的成员函数；您知道，诸如 public 和 private 等访问限定符有助于对外部实体隐藏类的数据和功能。

您学习了复制构造函数的概念，知道 C++让您能够使用移动构造函数消除不必要的复制步骤。您还了解到，通过结合使用这些元素，可实现单例等设计模式。

9.12 问与答

问：类实例和类对象有何不同？

答：基本上是一回事。实例化类时，将获得一个实例，它也被称为对象。

问：要访问成员，可使用句点运算符（.），也可使用指针运算符（->）。请问哪种方式更好？

答：如果有一个指向对象的指针，则使用指针运算符最合适；如果栈中实例化了一个对象，并将其存储到了一个局部变量中，则使用句点运算符最合适。

问：总是应该编写一个复制构造函数吗？

答：如果类的数据成员是设计良好的智能指针、字符串类或 STL 容器（如 std::vector），则编译器生成的默认复制构造函数将调用成员的复制构造函数。然而，如果类包含原始指针成员（如使用 int* 而不是 std::vector<int>表示的动态数组），则需要提供设计良好的复制构造函数，确保将类对象按值传递给函数时进行深复制，创建该数组的拷贝。

问：我的类只有一个构造函数，它接受一个有默认值的参数。请问这个构造函数是默认构造函数吗？

答：是的。只要在不提供参数的情况下创建实例，就可认为这个类有默认构造函数。每个类都只能有一个默认构造函数。

问：在本章的有些示例中，使用了函数 SetAge()来设置成员 Human::Age 的值。为何不将其声明为公有的，这样就能在需要时给它赋值了？

答：从技术角度说，将成员 Human::Age 声明为公有的也可行，但从设计的角度看，将数据成员声明为私有的是个不错的主意。通过使用 GetAge()和 SetAge()等存取函数，提供了一种更优雅、可扩展性更强的私有数据访问方式，让您在设置或重置 Human::Age 的值之前，能够执行错误检查。

问：在复制构造函数中，为何将指向源对象的引用作为参数？

答：这是编译器对复制构造函数的要求。其原因是，如果按值接受源对象，复制构造函数将调用自己，导致没完没了的复制循环。

9.13 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

9.13.1 测验

1. 使用 new 创建类实例时，将在什么地方创建它？
2. 我的类包含一个原始指针 int*，它指向一个动态分配的 int 数组。请问将 sizeof 用于这个类的对象时，结果是否取决于该动态数组包含的元素数？
3. 假设有一个类，其所有成员都是私有的，且没有友元类和友元函数。请问谁能访问这些成员？
4. 可以在一个类成员方法中调用另一个成员方法吗？
5. 构造函数适合做什么？
6. 析构函数适合做什么？

9.13.2 练习

1. 查错：下面的类声明有什么错误？

```
Class Human
{
   int Age;
   string Name;

public:
   Human() {}
}
```

2. 练习 1 所示类的用户如何访问成员 Human::Age？
3. 对练习 1 中的类进行修改，在构造函数中使用初始化列表对所有参数进行初始化。
4. 编写一个 Circle 类，它根据实例化时提供的半径计算面积和周长。将 Pi 包含在一个私有成员常量中，该常量不能在类外访问。

第 10 章
实现继承

面向对象编程基于 4 个重要方面：封装、抽象、继承和多态。继承是一种强大的属性重用方式，是通向多态的跳板。

在本章中，您将学习：

- 编程意义的继承；
- C++继承语法；
- 公有继承、私有继承和保护继承；
- 多继承；
- 隐藏基类方法和切除（slicing）导致的问题。

10.1 继承基础

在 Tom Smith 从祖先那里继承的东西中，最重要的是姓，因此他姓 Smith。另外，他还从父母那里继承了某些价值观以及木雕手艺，因为 Smith 家族数代人都从事木雕行业。这些属性一起标识了 Tom 作为 Smith 家族后代的身份。

在编程领域，您经常会遇到具有类似属性，但细节或行为存在细微差异的组件。在这些情形下，一种解决之道是将每个组件声明为一个类，并在每个类中实现所有的属性，这将重复实现相同的属性。另一种解决方案是使用继承，从同一个基类派生出类似的类，在基类中实现所有通用的功能，并在派生类中覆盖基本功能，以实现让每个类都独一无二的行为。第二种方法通常更佳。面向对象编程支持继承，如图 10.1 所示。

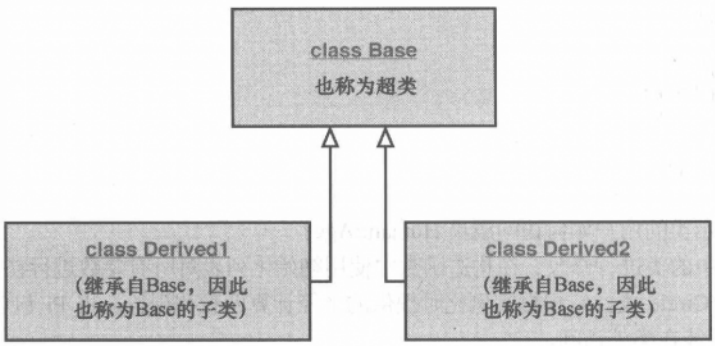


图 10.1 类之间的继承关系

10.1.1 继承和派生

图 10.1 说明了基类与派生类之间的关系。现在要明白基类和派生类是什么可能不容易；派生类继

承了基类，从这种意义上说，它是一个基类，就像 Tom 是 Smith 家族的成员一样。

注意 派生类和基类之间的这种 is-a 关系仅适用于公有继承。本章首先介绍公有继承，让您明白继承的概念以及最常见的继承方式，然后再介绍私有和保护继承。

为方便理解这种概念，先来看基类 Bird（鸟）。Crow（乌鸦）、Parrot（鹦鹉）和 Kiwi（鹬鸵）都是从 Bird 派生而来的。Bird 类将定义鸟的最基本属性，如有羽毛和翅膀、孵蛋、能飞。派生类 Crow、Parrot 和 Kiwi 继承这些属性，并进行定制（例如，Kiwi 不包含 Fly()的实现）。表 10.1 说明了其他几个继承的例子。

表 10.1 日常生活中的公有继承示例

基类	派生类
Fish（鱼）	Goldfish（金鱼）、Carp（鲤鱼）、Tuna（金枪鱼，金枪鱼是一种鱼）
Mammal（哺乳动物）	Human（人）、Elephant（大象）、Lion（狮子）、Platypus（鸭嘴兽，鸭嘴兽是一种哺乳动物）
Bird（鸟）	Crow（乌鸦）、Parrot（鹦鹉）、Ostrich（鸵鸟）、Kiwi（鹬鸵）、Platypus（鸭嘴兽也是一种鸟）
Shape（形状）	Circle 圆（圆是一种形状）、Polygon（多边形）
Polygon（多边形）	Triangle（三角形）、Octagon（八角形，八角形是一种多边形，而多边形是一种形状）

这些示例表明，戴上面向对象编程的眼镜后，就可在周围的很多物体中看到继承的例子。鱼是金枪鱼的基类，因为与鲤鱼一样，金枪鱼也是一种鱼，具备鱼的所有特征，如冷血。然而，金枪鱼在外观、游泳方式上不同于鲤鱼，另外，它是一种海水鱼。因此，金枪鱼和鲤鱼都从基类鱼那里继承了一些共同的特征，同时新增了一些特征，让它们彼此不同。图 10.2 说明了这一点。

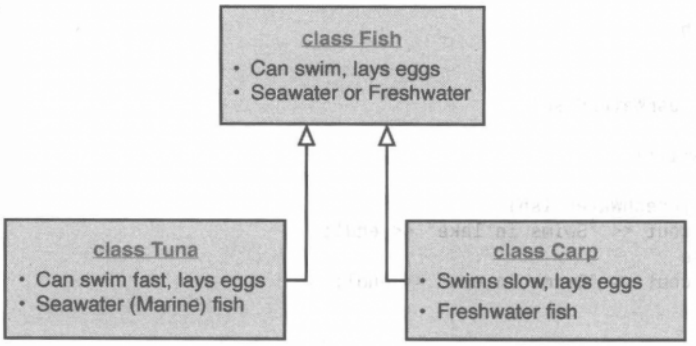


图 10.2 Tuna、Carp 和 Fish 之间的继承关系

鸭嘴兽（Platypus）能游，但是一种特殊动物，具备哺乳动物的特征，如通过哺乳喂养后代，同时具备鸟类的特征（如产卵）和爬行动物的特征（有毒腺）。因此，可将 Platypus 类视为继承了两个基类——Mammal 和 Bird。这种继承被称为多继承，将在本章后面讨论。

10.1.2 C++派生语法

如何从 Fish 类派生出 Carp 类呢？C++派生语法如下：

```
// declaring a super class
class Base
{
   // ... base class members
};
// declaring a sub-class
class Derived: access-specifier Base
```

```
{
   // ... derived class members
};
```

其中 access-specifier 可以是 public（这是最常见的，表示派生类是一个基类）、private 或 protected（表示派生类有一个基类）。

下面的继承层次结构表明，Carp 类是从 Fish 类派生而来的：

```
class Fish
{
 // ... Fish's members
};

class Carp:public Fish
{
// ... Carp's members
};
```

程序清单 10.1 从 Fish 类派生出了 Carp 和 Tuna 类，这些代码能够通过编译。

有关术语的说明

阅读介绍继承的文献时，您将遇到“从...继承而来”和“从...派生而来”等术语，它们的含义相同。同样，基类也被称为超类；从基类派生而来的类称为派生类，也叫子类。

程序清单 10.1 鱼类世界呈现的一种简单的继承层次结构

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    bool FreshWaterFish;
 7:
 8:    void Swim()
 9:    {
10:       if (FreshWaterFish)
11:          cout << "Swims in lake" << endl;
12:       else
13:          cout << "Swims in sea" << endl;
14:    }
15: };
16:
17: class Tuna: public Fish
18: {
19: public:
20:    Tuna()
21:    {
22:       FreshWaterFish = false;
23:    }
24: };
25:
26: class Carp: public Fish
27: {
28: public:
29:    Carp()
30:    {
31:       FreshWaterFish = true;
32:    }
33: };
34:
35: int main()
```

```
36: {
37:     Carp myLunch;
38:     Tuna myDinner;
39:
40:     cout << "Getting my food to swim" << endl;
41:
42:     cout << "Lunch: ";
43:     myLunch.Swim();
44:
45:     cout << "Dinner: ";
46:     myDinner.Swim();
47:
48:     return 0;
49: }
```

▼ 输出：

```
Getting my food to swim
Lunch: Swims in lake
Dinner: Swims in sea
```

▼ 分析：

在 main()中，第 37 和 38 行分别创建了一个 Carp 对象和一个 Tuna 对象：MyLunch 和 MyDinner。第 43 和 46 行调用方法 Swim()让这些对象游动。接下来，看看 Tuna 和 Carp 的类定义，如第 17～24 行和第 26～33 行所示。正如您看到的，这些类非常紧凑，好像都没有定义 main()中调用的 Swim()方法。显然，Swim()来自第 3～15 行定义的基类 Fish。由于 Fish 类声明了公有方法 Swim()，从 Fish 类（通过公有继承，如第 17 和 26 行所示）派生而来的 Tuna 和 Carp 类将自动暴露该基类的公有方法 Swim()。注意到 Carp 和 Tuna 类的构造函数初始化了基类的 FreshWaterFish 标记，该标记决定了 Fish::Swim()显示的输出。

10.1.3 访问限定符 protected

在程序清单 10.1 中，Fish 类包含公有属性 FreshWaterFish。派生类 Tuna 和 Carp 通过设置它来定制 Fish 的行为——在海水还是淡水中游动。然而，程序清单 10.1 存在一个严重的缺陷：如果您愿意，可在 main()中修改这个被声明为公有的标记，这为在 Fish 类外部使用类似于下面的代码操纵该标记打开了方便之门：

```
myDinner.FreshWaterFish = true; // making Tuna a fresh water fish!
```

这种缺陷显然应该避免。您需要让基类的某些属性能在派生类中访问，但不能在继承层次结构外部访问。这意味着您希望 Fish 类的布尔标记 FreshWaterFish 可在派生类 Tuna 和 Carp 中访问，但不能在实例化 Tuna 和 Carp 的 main()中访问。为此，可使用关键字 protected。

注意

> 与 public 和 private 一样，protected 也是一个访问限定符。将属性声明为 protected 时，相当于允许派生类和友元类访问它，但禁止在继承层次结构外部（包括 main()）访问它。

要让派生类能够访问基类的某个属性，可使用访问限定符 protected，如程序清单 10.2 所示。

程序清单 10.2 一种更好的 Fish 类设计，它使用关键字 protected 将成员属性只暴露给派生类

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5: protected:
```

```
 6:    bool FreshWaterFish; // accessible only to derived classes
 7:
 8: public:
 9:   void Swim()
10:    {
11:       if (FreshWaterFish)
12:          cout << "Swims in lake" << endl;
13:       else
14:          cout << "Swims in sea" << endl;
15:    }
16: };
17:
18: class Tuna: public Fish
19: {
20: public:
21:    Tuna()
22:    {
23:       FreshWaterFish = false; // set base class protected member
24:    }
25: };
26:
27: class Carp: public Fish
28: {
29: public:
30:    Carp()
31:    {
32:       FreshWaterFish = false;
33:    }
34: };
35:
36: int main()
37: {
38:    Carp myLunch;
39:    Tuna myDinner;
40:
41:    cout << "Getting my food to swim" << endl;
42:
43:    cout << "Lunch: ";
44:    myLunch.Swim();
45:
46:    cout << "Dinner: ";
47:    myDinner.Swim();
48:
49:    // uncomment line below to see that protected members
50:    // are not accessible from outside the class hierarchy
51:    // myLunch.FreshWaterFish = false;
52:
53:    return 0;
54: }
```

▼ 输出：

```
Getting my food to swim
Lunch: Swims in lake
Dinner: Swims in sea
```

▼ 分析：

虽然程序清单 10.2 的输出与程序清单 10.1 相同，但对 Fish 类做了大量重要的修改，如第 3～19 行所示。第一项也是最显而易见的修改是，布尔成员 Fish::FreshWaterFish 现在是一个 protected 属性，因此不能在 main()中访问，如第 51 行所示（如果取消对这行的注释，将导致编译错误）。但在派生类 Tuna

和 Carp 中，仍可访问这个 protected 属性，如第 23 和 32 行所示。这个小程序表明，通过使用关键字 protected，可对需要继承的基类属性进行保护，禁止在继承层次结构外部访问它。

这是面向对象编程的一个非常重要的方面，它与数据抽象和继承一起确保派生类可安全地继承基类的属性，同时禁止在继承层次结构外部对其进行修改。

10.1.4 基类初始化——向基类传递参数

如果基类包含重载的构造函数，需要在实例化时给它提供实参，该怎么办呢？创建派生对象时将如何实例化这样的基类？方法是使用初始化列表，并通过派生类的构造函数调用合适的基类构造函数，如下面的代码所示：

```
class Base
{
public:
   Base(int SomeNumber) // overloaded constructor
   {
      // Do something with SomeNumber
   }
};
Class Derived: public Base
{
public:
   Derived(): Base(25)  // instantiate class Base with argument 25
   {
      // derived class constructor code
   }
};
```

对 Fish 类来说，这种机制很有用。通过给 Fish 的构造函数提供一个布尔输入参数，以初始化 Fish::FreshWaterFish，可强制每个派生类都指出它是淡水鱼还是海水鱼，如程序清单 10.3 所示。

程序清单 10.3 包含初始化列表的派生类构造函数

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: protected:
 6:    bool FreshWaterFish; // accessible only to derived classes
 7:
 8: public:
 9:    // Fish constructor
10:    Fish(bool IsFreshWater) : FreshWaterFish(IsFreshWater){}
11:
12:    void Swim()
13:    {
14:       if (FreshWaterFish)
15:          cout << "Swims in lake" << endl;
16:       else
17:          cout << "Swims in sea" << endl;
18:    }
19: };
20:
21: class Tuna: public Fish
22: {
23: public:
24:    Tuna(): Fish(false) {}
25: };
```

```
26:
27: class Carp: public Fish
28: {
29: public:
30:    Carp(): Fish(true) {}
31: };
32:
33: int main()
34: {
35:    Carp myLunch;
36:    Tuna myDinner;
37:
38:    cout << "Getting my food to swim" << endl;
39:
40:    cout << "Lunch: ";
41:    myLunch.Swim();
42:
43:    cout << "Dinner: ";
44:    myDinner.Swim();
45:
46:    // uncomment line 48 to see that protected members
47:    // are not accessible from outside the class heirarchy
48:    // myLunch.FreshWaterFish = false;
49:
50:    return 0;
51: }
```

▼ 输出：

```
Getting my food to swim
Lunch: Swims in lake
Dinner: Swims in sea
```

▼ 分析：

现在，Fish 有一个构造函数，它接受一个默认参数，用于初始化 Fish::FreshWaterFish。因此，要创建 Fish 对象，必须提供一个用于初始化该保护成员的参数。这样，Fish 类便避免了保护成员包含随机值，尤其是派生类忘记设置它时。派生类 Tuna 和 Carp 被迫定义一个这样的构造函数，即使用合适的参数（true 或 false，表示是否是淡水鱼）来实例化基类 Fish，如第 24 和 30 行所示。

注意

在程序清单 10.3 中，派生类没有直接访问布尔成员变量 Fish::FreshWaterFish，虽然这是一个 protected 成员，这是因为这个变量是通过 Fish 的构造函数设置的。

为最大限度地提高安全性，对于派生类不需要访问的基类属性，别忘了将其声明为私有的。

10.1.5 在派生类中覆盖基类的方法

如果派生类实现了从基类继承的函数，且返回值和特征标相同，就相当于覆盖了基类的这个方法，如下面的代码所示：

```
class Base
{
public:
   void DoSomething()
   {
      // implementation code… Does something
   }
};
```

```
class Derived:public Base
{
public:
   void DoSomething()
   {
      // implementation code… Does something else
   }
};
```

因此，如果使用 Derived 类的实例调用方法 DoSomething()，调用的将不是 Base 类中的这个方法。

如果 Tuna 和 Carp 类实现了自己的 Swim()方法，则在程序清单 10.3 的 main()中，下述代码将调用 Tuna::Swim()的实现，这相当于覆盖了基类 Fish 的方法 Swim()：

```
36:    Tuna myDinner;
// ...other lines
44:    myDinner.Swim();
```

程序清单 10.4 演示了这一点。

程序清单 10.4　派生类 Tuna 和 Carp 覆盖了基类 Fish 的方法 Swim()

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: private:
 6:    bool FreshWaterFish;
 7:
 8: public:
 9:    // Fish constructor
10:    Fish(bool IsFreshWater) : FreshWaterFish(IsFreshWater){}
11:
12:    void Swim()
13:    {
14:       if (FreshWaterFish)
15:          cout << "Swims in lake" << endl;
16:       else
17:          cout << "Swims in sea" << endl;
18:    }
19: };
20:
21: class Tuna: public Fish
22: {
23: public:
24:    Tuna(): Fish(false) {}
25:
26:    void Swim()
27:    {
28:       cout << "Tuna swims real fast" << endl;
29:    }
30: };
31:
32: class Carp: public Fish
33: {
34: public:
35:    Carp(): Fish(true) {}
36:
37:    void Swim()
38:    {
39:       cout << "Carp swims real slow" << endl;
40:    }
41: };
```

```
42:
43: int main()
44: {
45:    Carp myLunch;
46:    Tuna myDinner;
47:
48:    cout << "Getting my food to swim" << endl;
49:
50:    cout << "Lunch: ";
51:    myLunch.Swim();
52:
53:    cout << "Dinner: ";
54:    myDinner.Swim();
55:
56:    return 0;
57: }
```

▼ **输出：**

```
Getting my food to swim
Lunch: Carp swims real slow
Dinner: Tuna swims real fast
```

▼ **分析：**

输出表明，第51行的myLunch.Swim()调用的是第37～40行定义的Carp::Swim()。同样，第54行的myDinner.Swim()调用的是第26～29行定义的Tuna::Swim()。换句话说，基类Fish中Swim()的实现（第12～18行）被派生类Tuna和Carp类中的Swim()覆盖了。要调用Fish::Swim()，要么让派生类在其成员函数中显式地使用它，要么在main()中使用作用域解析运算符显式地调用它，这将在本章后面演示。

10.1.6 调用基类中被覆盖的方法

在程序清单10.4中，派生类Tuna通过实现Swim()覆盖了Fish类的Swim()函数，其结果如下：

```
Tuna myDinner;
myDinner.Swim(); // will invoke Tuna::Swim()
```

在程序清单10.4中，如果要在main()中调用Fish::Swim()，需要使用作用域解析运算符（::），如下所示：

```
myDinner.Fish::Swim();  // will invoke Fish::Swim() in spite of being a Tuna
```

稍后的程序清单10.5演示了如何通过派生类的实例调用基类的成员。

10.1.7 在派生类中调用基类的方法

通常，Fish::Swim()包含适用于所有鱼类（包括金枪鱼和鲤鱼）的通用实现。如果要在Tuna::Swim()和Carp::Swim()的实现中重用Fish::Swim()的通用实现，可使用作用域解析运算符（::），如下面的代码所示：

```
class Carp: public Fish
{
public:
   Carp(): Fish(true) {}

   void Swim()
   {
      cout << "Carp swims real slow" << endl;
      Fish::Swim();  // use scope resolution operator::
   }
};
```

程序清单10.5演示了这一点。

程序清单 10.5 在基类方法和 main()中，使用作用域解析运算符（::）来调用基类方法

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

class Fish
{
private:
   bool FreshWaterFish;

public:
   // Fish constructor
   Fish(bool IsFreshWater) : FreshWaterFish(IsFreshWater){}

   void Swim()
   {
      if (FreshWaterFish)
         cout << "Swims in lake" << endl;
      else
         cout << "Swims in sea" << endl;
   }
};

class Tuna: public Fish
{
public:
   Tuna(): Fish(false) {}

   void Swim()
   {
      cout << "Tuna swims real fast" << endl;
   }
};

class Carp: public Fish
{
public:
   Carp(): Fish(true) {}

   void Swim()
   {
      cout << "Carp swims real slow" << endl;
      Fish::Swim();
   }
};

int main()
{
   Carp myLunch;
   Tuna myDinner;

   cout << "Getting my food to swim" << endl;

   cout << "Lunch: ";
   myLunch.Swim();

   cout << "Dinner: ";
   myDinner.Fish::Swim();

   return 0;
}
```

▼ 输出：

```
Getting my food to swim
Lunch: Carp swims real slow
Swims in lake
Dinner: Swims in sea
```

▼ 分析：

第 37～41 行的 Carp::Swim()使用作用域解析运算符（::）调用了基类方法 Fish::Swim()，而第 55 行演示了如何在 main 中，使用作用域解析运算符（::）通过 Tuna 实例调用基类方法 Fish::Swim()。

10.1.8 在派生类中隐藏基类的方法

覆盖的一种极端情形是，Tuna::Swim()可能隐藏 Fish::Swim()的所有重载版本，使得调用这些重载版本会导致编译错误（因此称为被隐藏），程序清单 10.6 演示了这一点。

程序清单 10.6 Tuna::Swim()隐藏了重载方法 Fish::Swim(bool)

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    void Swim()
 7:    {
 8:        cout << "Fish swims... !" << endl;
 9:    }
10:
11:    void Swim(bool FreshWaterFish)
12:    {
13:       if (FreshWaterFish)
14:          cout << "Swims in lake" << endl;
15:       else
16:          cout << "Swims in sea" << endl;
17:    }
18: };
19:
20: class Tuna: public Fish
21: {
22: public:
23:    void Swim()
24:    {
25:       cout << "Tuna swims real fast" << endl;
26:    }
27: };
28:
29: int main()
30: {
31:    Tuna myDinner;
32:
33:    cout << "Getting my food to swim" << endl;
34:
35:    // myDinner.Swim(false); // compile failure: Fish::Swim(bool) is hidden
by Tuna::Swim()
36:    myDinner.Swim();
37:
38:    return 0;
39: }
```

▼ 输出：

```
Getting my food to swim
Tuna swims real fast
```

▼ 分析：

这个版本的 Fish 类与前面的 Fish 类有点不同。除采用尽可能简单的版本诠释当前问题外，这个 Fish 版本还包含两个重载的 Swim()方法：一个不接受任何参数，如第 6～9 行所示，另一个接受一个 bool 参数，如第 11～17 行所示。鉴于 Tuna 以公有方式继承了 Fish，您可能认为通过 Tuna 实例可调用这两个版本的 Fish::Swim()方法。然而，由于 Tuna 实现了自己的 Tuna::Swim()，如第 23～26 行所示，这对编译器隐藏了 Fish::Swim(bool)。如果取消对第 35 行的注释，将出现编译错误。

要通过 Tuna 实例调用 Fish::Swim(bool)，可采用如下解决方案。

- 解决方案 1：在 main()中使用作用域解析运算符（::）。

```
myDinner.Fish::Swim();
```

- 解决方案 2：在 Tuna 类中，使用关键字 using 解除对 Fish::Swim()的隐藏。

```
class Tuna: public Fish
{
public:
   using Fish::Swim;   // unhide Swim methods in base class Fish

   void Swim()
   {
      cout << "Tuna swims real fast" << endl;
   }
};
```

- 解决方案 3：在 Tuna 类中，覆盖 Fish::Swim()的所有重载版本（如果需要，可通过 Tuna::Fish(...)调用方法 Fish::Swim()）。

```
class Tuna: public Fish
{
public:
   void Swim(bool FreshWaterFish)
   {
      Fish::Swim(FreshWaterFish);
   }

   void Swim()
   {
      cout << "Tuna swims real fast" << endl;
   }
};
```

10.1.9 构造顺序

如果 Tuna 类是从 Fish 类派生而来的，创建 Tuna 对象时，先调用 Tuna 的构造函数还是 Fish 的构造函数？另外，实例化对象时，成员属性（如 Fish::FreshWaterFish）是调用构造函数之前还是之后实例化？基类对象在派生类对象之前被实例化，因此，首先构造 Tuna 对象的 Fish 部分，这样实例化 Tuna 部分时，成员属性（具体地说是 Fish 的保护和公有属性）已准备就绪，可以使用了。实例化 Fish 部分和 Tuna 部分时，先实例化成员属性（如 Fish::FreshWaterFish），再调用构造函数，确保成员属性准备就绪，可供构造函数使用。这也适用于 Tuna::Tuna()。

10.1.10 析构顺序

Tuna 实例不再在作用域内时，析构顺序与构造顺序相反。程序清单 10.7 所示的简单示例演示了构造顺序和析构顺序。

程序清单 10.7 基类、派生类及其成员的构造顺序和析构顺序

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

class FishDummyMember
{
public:
   FishDummyMember()
   {
      cout << "FishDummyMember constructor" << endl;
   }

   ~FishDummyMember()
   {
      cout << "FishDummyMember destructor" << endl;
   }
};

class Fish
{
protected:
   FishDummyMember dummy;

public:
   // Fish constructor
   Fish()
   {
      cout << "Fish constructor" << endl;
   }

   ~Fish()
   {
      cout << "Fish destructor" << endl;
   }
};

class TunaDummyMember
{
public:
   TunaDummyMember()
   {
      cout << "TunaDummyMember constructor" << endl;
   }

   ~TunaDummyMember()
   {
      cout << "TunaDummyMember destructor" << endl;
   }
};


class Tuna: public Fish
{
```

```
52: private:
53:    TunaDummyMember dummy;
54:
55: public:
56:    Tuna()
57:    {
58:       cout << "Tuna constructor" << endl;
59:    }
60:    ~Tuna()
61:    {
62:       cout << "Tuna destructor" << endl;
63:    }
64:
65: };
66:
67: int main()
68: {
69:    Tuna myDinner;
70: }
```

▼ **输出：**

```
FishDummyMember constructor
Fish constructor
TunaDummyMember constructor
Tuna constructor
Tuna destructor
TunaDummyMember destructor
Fish destructor
FishDummyMember destructor
```

▼ **分析：**

第 67～70 行的 main()很短，但输出量很大。实例化一个 Tuna 对象就生成了这些输出，这是由于构造函数和析构函数包含 cout 语句。为帮助理解成员变量是如何被实例化和销毁的，定义了两个毫无用途的类——FishDummyMember 和 TunaDummyMember，并在其构造函数和析构函数中包含了 cout 语句。Fish 和 Tuna 类分别将这些类的对象作为成员，如第 20 和 53 行所示。输出表明，实例化 Tuna 对象时，将从继承层次结构顶部开始，因此首先实例化 Tuna 对象的 Fish 部分。为此，首先实例化 Fish 的成员属性，即 Fish::dummy。构造好成员属性（如 dummy）后，将调用 Fish 的构造函数。构造好基类部分后，将实例化 Tuna 部分——首先实例化成员 Tuna::dummy，再执行构造函数 Tuna::Tuna()的代码。输出表明，析构顺序正好相反。

10.2 私有继承

前面介绍的都是公有继承，私有继承的不同之处在于，指定派生类的基类时使用关键字 private：

```
class Base
{
   // ... base class members and methods

};

class Derived: private Base     // private inheritance
{
   // ... derived class members and methods
};
```

私有继承意味着在派生类的实例中，基类的所有公有成员和方法都是私有的——不能从外部访问。换句话说，即便是 Base 类的公有成员和方法，也只能被 Derived 类使用，而无法通过 Derived

实例来使用它们。

这与前面的示例截然不同。在程序清单 10.1 中，可在 main()中通过 Tuna 实例调用 Fish::Swim()，因为 Fish::Swim()是个公有方法，且 Tuna 类是以公有方式从 Fish 类派生而来的。如果将第 17 行的 public 改为 private，该程序清单将无法通过编译。

因此，从继承层次结构外部看，私有继承并非 is-a 关系。私有继承使得只有子类才能使用基类的属性和方法，因此也被称为 has-a 关系。在现实世界中，存在一些私有继承的例子，如表 10.2 所示。

表 10.2　　现实世界的私有继承示例

基类	派生类
Motor（发动机）	Car（汽车，汽车有发动机）
Heart（心脏）	Mammal（哺乳动物，哺乳动物有心脏）
Refill（笔芯）	Pen（钢笔，钢笔有笔芯）
Moon（月亮）	Sky（天空，天空有月亮）

下面来看看汽车与发动机之间的私有继承关系，如程序清单 10.8 所示。

程序清单 10.8　Car 类以私有方式继承 Motor 类

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Motor
 4: {
 5: public:
 6:    void SwitchIgnition()
 7:    {
 8:       cout << "Ignition ON" << endl;
 9:    }
10:    void PumpFuel()
11:    {
12:       cout << "Fuel in cylinders" << endl;
13:    }
14:    void FireCylinders()
15:    {
16:       cout << "Vroooom" << endl;
17:    }
18: };
19:
20: class Car:private Motor
21: {
22: public:
23:    void Move()
24:    {
25:       SwitchIgnition();
26:       PumpFuel();
27:       FireCylinders();
28:    }
29: };
30:
31: int main()
32: {
33:    Car myDreamCar;
34:    myDreamCar.Move();
35:
36:    return 0;
37: }
```

▼ 输出：

```
Ignition ON
Fuel in cylinders
Vroooom
```

▼ 分析：

Motor 类是在第 3～18 行定义的，它非常简单，包含 3 个公有的成员函数，它们分别开启点火开关、喷油和点火。Car 类使用关键字 private 继承了 Motor 类，如第 20 行所示。公有函数 Car::Move() 调用了基类 Motor 的成员函数。如果在 main()中插入下述代码：

```
myDreamCar.PumpFuel();
```

将无法通过编译，编译错误类似于下面这样：error C2247: Motor::PumpFuel not accessible because 'Car' uses 'private' to inherit from 'Motor'。

注意

如果有一个 SuperCar 类，它继承了 Car 类，则不管 SuperCar 和 Car 之间的继承关系是什么样的，SuperCar 都不能访问基类 Motor 的公有成员和方法。这是因为 Car 和 Motor 之间是私有继承关系，这意味着除 Car 外，其他所有实体都不能访问 Motor 的公有成员。换句话说，编译器在确定派生类能否访问基类的公有或保护成员时，考虑的是继承层次结构中最严格的访问限定符。

10.3 保护继承

保护继承不同于公有继承之处在于，声明派生类继承基类时使用关键字 protected：

```
class Base
{
   // ... base class members and methods

};

class Derived: protected Base      // protected inheritance
{
   // ... derived class members and methods
};
```

保护继承与私有继承的类似之处如下：

- 它也表示 has-a 关系；
- 它也让派生类能够访问基类的所有公有和保护成员；
- 在继承层次结构外面，也不能通过派生类实例访问基类的公有成员。

随着继承层次结构的加深，保护继承将与私有继承有些不同：

```
class Derived2: protected Derived
{
   // can access members of Base
};
```

在保护继承层次结构中，子类的子类（即 Derived2）能够访问 Base 类的公有成员，如程序清单 10.9 所示。如果 Derived 和 Base 之间的继承关系是私有的，就不能这样做。

程序清单 10.9 SuperCar 类以保护方式继承了 Car 类，而 Car 类以保护方式继承了 Motor 类

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Motor
 4: {
```

```
 5: public:
 6:    void SwitchIgnition()
 7:    {
 8:       cout << "Ignition ON" << endl;
 9:    }
10:    void PumpFuel()
11:    {
12:       cout << "Fuel in cylinders" << endl;
13:    }
14:    void FireCylinders()
15:    {
16:       cout << "Vroooom" << endl;
17:    }
18: };
19:
20: class Car:protected Motor
21: {
22: public:
23:    void Move()
24:    {
25:       SwitchIgnition();
26:       PumpFuel();
27:       FireCylinders();
28:    }
29: };
30:
31: class SuperCar:protected Car
32: {
33: public:
34:    void Move()
35:    {
36:       SwitchIgnition();  // has access to base members
37:       PumpFuel();        // due to "protected" inheritance between Car and
          Motor
38:       FireCylinders();
39:       FireCylinders();
40:       FireCylinders();
41:    }
42: };
43:
44: int main()
45: {
46:    SuperCar myDreamCar;
47:    myDreamCar.Move();
48:
49:    return 0;
50: }
```

▼ 输出：

```
Ignition ON
Fuel in cylinders
Vroooom
Vroooom
Vroooom
```

▼ 分析：

Car 类以保护方式继承了 Motor 类，如第 20 行所示，而 SuperCar 类以保护方式继承了 Car 类，如第 31 行所示。正如您看到的，SuperCar::Move()的实现使用了基类 Motor 中定义的公有方法。能否经由中间基类 Car 访问终极基类 Motor 的公有成员呢？这取决于 Car 和 Motor 之间的继承关系。如果继

承关系是私有的，而不是保护的，SuperCar 将不能访问 Motor 类的公有成员，因为编译器根据最严格的访问限定符来确定访问权。请注意，SuperCar 和 Car 之间的继承关系不会影响它对 Motor 类公有成员的访问权，因此即便将第 31 行的 protected 改为 public 或 private，也改变不了这个程序能否通过编译的命运。

警告

仅当必要时才使用私有或保护继承。

对于大多数使用私有继承的情形（如 Car 和 Motor 之间的私有继承），更好的选择是，将基类对象作为派生类的一个成员属性。通过继承 Motor 类，相当于对 Car 类进行了限制，使其只能有一台发动机，同时，相比于将 Motor 对象作为私有成员，没有任何好处可言。汽车在不断发展，例如，混合动力车除电力发动机外，还有一台汽油发动机。在这种情况下，让 Car 类继承 Motor 类将成为兼容性瓶颈。

注意

将 Motor 对象作为 Car 类的私有成员被称为组合（composition）或聚合（aggergation），这样的 Car 类类似于下面这样：

```
class Car
{
private:
   Motor heartOfCar;

public:
   void Move()
   {
      heartOfCar.SwitchIgnition();
      heartOfCar.PumpFuel();
      heartOfCar.FireCylinders();
   }
};
```

这是一种不错的设计，让您能够轻松地在 Car 类中添加 Motor 成员，而无需改变继承层次机构，也不用修改客户看到的设计。

10.4 切除问题

如果程序员像下面这样做，结果将如何呢？

```
Derived objectDerived;
Base objectBase = objectDerived;
```

如果程序员像下面这样做，结果又将如何呢？

```
void FuncUseBase(Base input);
...
Derived objectDerived;
FuncUseBase(objectDerived);  // objectDerived will be sliced when copied during
  function call
```

它们都将 Derived 对象复制给 Base 对象，一个是通过显式复制，另一个是通过传递参数。在这些情形下，编译器将只复制 objectDerived 的 Base 部分，即不是整个对象，而是 Base 能容纳的部分，这通常不是程序员的本意。这种无意间裁减数据，导致 Derived 变成 Base 的行为称为切除（slicing）。

警告

要避免切除问题，不要按值传递参数，而应以指向基类的指针或 const 引用的方式传递。

10.5 多继承

本章前面说过，在有些情况下，采用多继承是合适的，如对鸭嘴兽来说就很合适，这是因为鸭嘴兽具备哺乳动物、鸟类和爬行动物的特征。为应对这样的情形，C++允许继承多个基类：

```
class Derived: access-specifier Base1, access-specifier Base2
{
   // class members
};
```

图 10.3 是 Platypus（鸭嘴兽）的类图，这与 Tuna 和 Carp 的类图（见图 10.2）完全不同。

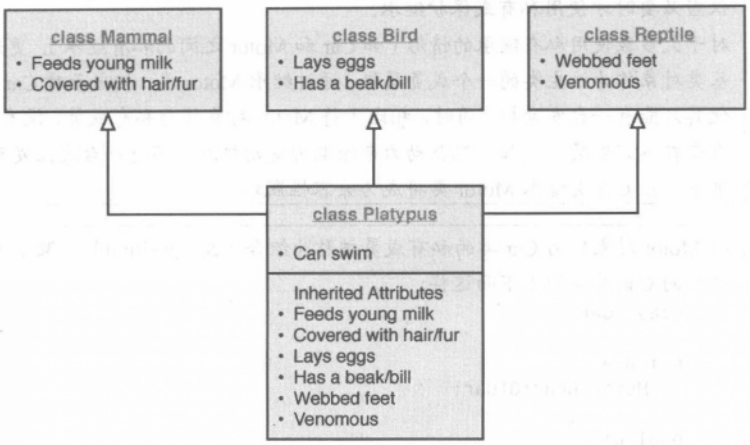


图 10.3 Platypus 类与 Mammal、Reptile 和 Bird 类之间的关系

因此，表示 Platypus 类的 C++代码如下：

```
class Platypus: public Mammal, public Reptile, public Bird
{
   // ... platypus members
};
```

程序清单 10.10 的 Platypus 类演示了多继承。

程序清单 10.10 使用多继承模拟具备哺乳动物、鸟类和爬行动物特征的鸭嘴兽

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Mammal
 4: {
 5: public:
 6:    void FeedBabyMilk()
 7:    {
 8:       cout << "Mammal: Baby says glug!" << endl;
 9:    }
10: };
11:
12: class Reptile
13: {
14: public:
15:    void SpitVenom()
16:    {
17:       cout << "Reptile: Shoo enemy! Spits venom!" << endl;
18:    }
19: };
20:
21: class Bird
22: {
23: public:
24:    void LayEggs()
25:    {
26:       cout << "Bird: Laid my eggs, am lighter now!" << endl;
27:    }
```

```
28: };
29:
30: class Platypus: public Mammal, public Bird, public Reptile
31: {
32: public:
33:    void Swim()
34:    {
35:       cout << "Platypus: Voila, I can swim!" << endl;
36:    }
37: };
38:
39: int main()
40: {
41:    Platypus realFreak;
42:    realFreak.LayEggs();
43:    realFreak.FeedBabyMilk();
44:    realFreak.SpitVenom();
45:    realFreak.Swim();
46:
47:    return 0;
48: }
```

▼ 输出：

```
Bird: Laid my eggs, am lighter now!
Mammal: Baby says glug!
Reptile: Shoo enemy! Spits venom!
Platypus: Voila, I can swim!
```

▼ 分析：

Platypus 类的定义非常简单，如第 30～37 行所示。除继承 Mammal、Reptile 和 Bird 等 3 个类外，它几乎什么都没做。在 main()中的第 41～44 行，可通过派生类 Platypus 的对象调用这 3 个基类的方法，该对象被恰当地命名为 realFreak。除调用从 Mammal、Reptile 和 Bird 类继承而来的方法外，第 45 行还调用了 Platypus::Swim()。这个程序演示了多继承的语法，还表明派生类暴露了从众多基类继承而来的公有属性（这里是公有成员方法）。

注意

鸭嘴兽会游泳，但不属于鱼类。因此，在程序清单 10.10 中，没有仅为方便使用现有的 Fish::Swim()函数，而让 Platypus 也继承 Fish。做设计决策时，别忘了公有继承应表示 is-a 关系，因此不应为实现重用目标而不分青红皂白地使用公有继承。可采取其他方式实现这种目标。

应该	不应该
要建立 is-a 关系，务必创建公有继承层次结构。 要建立 has-a 关系，务必创建私有或保护继承层次结构。 务必牢记，公有继承意味着继承派生类的类能够访问基类的公有和保护成员。 务必牢记，私有继承意味着继承派生类的类也不能访问基类的成员。 务必牢记，保护继承意味着继承派生类的类能够访问基类的公有和保护方法。 务必牢记，无论继承关系是什么，派生类都不能访问基类的私有成员。	不要仅为重用微不足道的方法而创建继承层次结构。 不要不分青红皂白地使用私有或公有继承，因为这可能给应用程序的可扩展性带来架构瓶颈。 在派生类中，不要编写与基类方法同名但参数不同的方法，以免隐藏基类方法。

10.6 总结

在本章中，您学习了C++继承的基本知识。您了解到，公有继承在派生类和基类之间建立is-a关系，而私有和保护继承建立has-a关系。您得知，通过使用访问限定符protected，可将基类的属性暴露给派生类，同时对继承层次结构外部隐藏它们。您了解到，保护继承与私有继承的不同之处在于，派生类的子类可访问基类的公有和保护成员，而使用私有继承时不能。您学习了覆盖和隐藏方法的基本知识，还知道可使用关键字using避免隐藏方法。

现在，您可以回答一些问题，然后选择面向对象编程的下一个主要支柱——多态。

10.7 问与答

问：我需要模拟哺乳动物（Mammal）以及一些具体的哺乳动物，如人（Human）、狮子（Lion）和鲸鱼（Whale）。我该使用继承层次结构吗？如果该使用，应使用哪种继承关系？

答：鉴于人、狮子和鲸鱼都是哺乳动物，这是is-a关系，因此应使用公有继承，并将Mammal作为基类，而Human、Lion和Whale类从它派生而来。

问：术语派生类和子类有何不同？

答：是一回事，它们都表示从基类派生而来的类。

问：以公有方式继承基类的派生类能访问基类的私有成员吗？

答：不能。编译器总是执行最严格的访问限定符。无论继承关系如何，类的私有成员都不能在类外访问，一个例外是类的友元函数和友元类。

10.8 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录D的答案。在继续学习下一章前，请务必弄懂这些答案。

10.8.1 测验

1. 我希望基类的某些成员可在派生类中访问，但不能在继承层次结构外访问，该使用哪种访问限定符？
2. 如果一个函数接受一个基类对象作为参数，而我将一个派生类对象作为实参按值传递给它，结果将如何？
3. 该使用私有继承还是组合？
4. 在继承层次结构中，关键字using有何用途？
5. Derived类以私有方式继承了Base类，而SubDerived类以公有方式继承了Derived类。请问SubDerived类能访问Base类的公有成员吗？

10.8.2 练习

1. 创建程序清单10.10所示的Platypus对象时，将以什么样的顺序调用构造函数？

2. 使用代码说明 Polygon、Triangle 和 Shape 类之间的关系。

3. D2 类继承了 D1 类，而 D1 类继承了 Base 类。要禁止 D2 访问 Base 的公有方法，应使用哪种访问限定符？在什么地方使用？

4. 下面的代码表示哪种继承关系？

```
class Derived: Base
{
   // ... Derived members
};
```

5. 查错：下述代码有何问题？

```
class Derived: public Base
{
   // ... Derived members
};
void SomeFunc (Base value)
{
   // …
}
```

第11章
多　态

学习继承的基本知识、创建继承层次结构并明白公有继承实际上模拟的是 is-a 关系后，该学习面向对象编程的核心——多态，并应用这些知识了。

在本章中，您将学习：

- 多态意味着什么；
- 虚函数的用途和用法；
- 什么是抽象类及如何声明它们；
- 虚继承意味着什么以及在什么情况下使用。

11.1　多态基础

Poly 源自希腊语，意思是“多”，而 morph 是“形态”的意思。多态（Polymorphism）是面向对象语言的一种特征，让您能够以类似的方式处理不同类型的对象。本章重点介绍多态行为，这种行为也被成为子类型多态（subtype polymorphism），在 C++中，可通过继承层次结构来实现。

11.1.1　为何需要多态行为

在第 10 章，您发现 Tuna 和 Carp 从 Fish 那里继承了方法 Swim()，如程序清单 10.1 所示。然而，Tuna 和 Carp 都可提供自己的 Swim()方法，以定制其游泳方式，但鉴于它们都是 Fish，如果将 Tuna 实例作为实参传递给 Fish 参数，并通过该参数调用 Swim()，最终执行的将是 Fish::Swim()，而不是 Tuna::Swim()。程序清单 11.1 演示了这种问题。

注意

为最大限度地减少代码，以提高可读性，本章的所有代码示例都进行了简化，使其足以诠释相关主题即可。在您自己编程时，应根据应用程序的设计和用途正确地编写类，并创建合理的继承层次结构。

程序清单 11.1　将 Tuna 实例传递给 Fish 参数，并通过该参数调用方法

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5: public:
6:     void Swim()
7:     {
```

```
 8:        cout << "Fish swims!" << endl;
 9:    }
10: };
11:
12: class Tuna:public Fish
13: {
14: public:
15:    // override Fish::Swim
16:    void Swim()
17:    {
18:        cout << "Tuna swims!" << endl;
19:    }
20: };
21:
22: void MakeFishSwim(Fish& InputFish)
23: {
24:    // calling Fish::Swim
25:    InputFish.Swim();
26: }
27:
28: int main()
29: {
30:    Tuna myDinner;
31:
32:    // calling Tuna::Swim
33:    myDinner.Swim();
34:
35:    // sending Tuna as Fish
36:    MakeFishSwim(myDinner);
37:
38:    return 0;
39: }
```

▼ 输出：

```
Tuna swims!
Fish swims!
```

▼ 分析：

Tuna 类以公有方式继承了 Fish 类，如第 12 行所示，它还覆盖了方法 Fish::Swim()。在 main()中，第 33 行直接调用了 Tuna::Swim()，并将 myDinner（其类型为 Tuna）作为参数传递给了 MakeFishSwim()，而该函数将其视为 Fish 引用，如第 22 行的声明所示。换句话说，虽然传入的是 Tuna 对象，MakeFishSwim(Fish&)也将其视为 Fish，进而调用 Fish::Swim。第 2 行输出表明，虽然传入的是 Tuna 对象，但得到的却是 Fish 的输出（这也适用于 Carp 对象）。

理想情况下，用户希望 Tuna 对象表现出金枪鱼的行为，即便通过 Fish 参数调用 Swim()时亦如此。换句话说，第 25 行调用 InputFish.Swim()时，用户希望执行的是 Tuna::Swim()。要实现这种多态行为——让 Fish 参数表现出其实际类型（派生类 Tuna）的行为，可将 Fish::Swim()声明为虚函数。

11.1.2 使用虚函数实现多态行为

可通过 Fish 指针或 Fish 引用来访问 Fish 对象，这种指针或引用可指向 Fish、Tuna 或 Carp 对象，但您不知道也不关心它们指向的是哪种对象。要通过这种指针或引用调用方法 Swim()，可以像下面这样做：

```
pFish->Swim();
myFish.Swim();
```

您希望通过这种指针或引用调用Swim()时，如果它们指向的是Tuna对象，则可像Tuna那样游泳，如果指向的是Carp对象，则可像Carp那样游泳，如果指向的是Fish，则可像Fish那样游泳。为此，可在基类Fish中将Swim()声明为虚函数：

```
class Base
{
   virtual ReturnType FunctionName (Parameter List);
};
class Derived
{
   ReturnType FunctionName (Parameter List);
};
```

通过使用关键字virtual，可确保编译器调用覆盖版本。也就是说，如果Swim()被声明为虚函数，则将参数myFish（其类型为Fish&）设置为一个Tuna对象时，myFish.Swim()将执行Tuna::Swim()，如程序清单11.2所示。

程序清单11.2 将Fish::Swim()声明为虚函数带来的影响

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

class Fish
{
public:
   virtual void Swim()
   {
      cout << "Fish swims!" << endl;
   }
};

class Tuna:public Fish
{
public:
   // override Fish::Swim
   void Swim()
   {
      cout << "Tuna swims!" << endl;
   }
};

class Carp:public Fish
{
public:
   // override Fish::Swim
   void Swim()
   {
      cout << "Carp swims!" << endl;
   }
};

void MakeFishSwim(Fish& InputFish)
{
   // calling virtual method Swim()
   InputFish.Swim();
}

int main()
{
   Tuna myDinner;
```

```
41:    Carp myLunch;
42:
43:    // sending Tuna as Fish
44:    MakeFishSwim(myDinner);
45:
46:    // sending Carp as Fish
47:    MakeFishSwim(myLunch);
48:
49:    return 0;
50: }
```

▼ 输出：

```
Tuna swims!
Carp swims!
```

▼ 分析：

函数 MakeFishSwim(Fish&)与程序清单 11.1 中完全相同，但输出截然不同。首先，根本没有调用 Fish::Swim()，因为存在覆盖版本 Tuna::Swim()和 Carp::Swim()，它们优先于被声明为虚函数的 Fish::Swim()。这很重要，它意味着在 MakeFishSwim()中，可通过 Fish&参数调用派生类定义的 Swim()，而无需知道该参数指向的是哪种类型的对象。

这就是多态：将派生类对象视为基类对象，并执行派生类的 Swim()实现。

11.1.3 为何需要虚构造函数

程序清单 11.1 演示了一个问题，那就是将派生类对象传递给基类参数时，并通过该参数调用函数时，将执行基类的函数。然而，还存在一个问题：如果基类指针指向的是派生类对象，通过该指针调用运算符 delete 时，结果将如何呢？

将调用哪个析构函数呢？请看程序清单 11.3。

程序清单 11.3 在函数中通过基类指针调用运算符 delete

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    Fish()
 7:    {
 8:       cout << "Constructed Fish" << endl;
 9:    }
10:    ~Fish()
11:    {
12:       cout << "Destroyed Fish" << endl;
13:    }
14: };
15:
16: class Tuna:public Fish
17: {
18: public:
19:    Tuna()
20:    {
21:       cout << "Constructed Tuna" << endl;
22:    }
23:    ~Tuna()
24:    {
```

```
25:       cout << "Destroyed Tuna" << endl;
26:    }
27: };
28:
29: void DeleteFishMemory(Fish* pFish)
30: {
31:    delete pFish;
32: }
33:
34: int main()
35: {
36:    cout << "Allocating a Tuna on the free store:" << endl;
37:    Tuna* pTuna = new Tuna;
38:    cout << "Deleting the Tuna: " << endl;
39:    DeleteFishMemory(pTuna);
40:
41:    cout << "Instantiating a Tuna on the stack:" << endl;
42:    Tuna myDinner;
43:    cout << "Automatic destruction as it goes out of scope: " << endl;
44:
45:    return 0;
46: }
```

▼ 输出：

```
Allocating a Tuna on the free store:
Constructed Fish
Constructed Tuna
Deleting the Tuna:
Destroyed Fish
Instantiating a Tuna on the stack:
Constructed Fish
Constructed Tuna
Automatic destruction as it goes out of scope:
Destroyed Tuna
Destroyed Fish
```

▼ 分析：

在 main()中，第 37 行使用 new 在自由存储区中创建了一个 Tuna 实例；然后马上使用辅助函数 DeleteFishMemory()释放分配的内存，如第 39 行所示。出于比较的目的，第 42 行在栈上创建了另一个 Tuna 实例——局部变量 myDinner，在 main()结束时，它将不再在作用域内。输出是由 Fish 和 Tuna 类的构造函数和析构函数中的 cout 语句生成的。注意到由于使用了关键字 new，在自由存储区中构造了 Tuna 和 Fish，但 delete 没有调用 Tuna 的析构函数，而只调用了 Fish 的析构函数；而构造和析构局部变量 myDinner 时，调用了基类和派生类的构造函数和析构函数，这形成了鲜明的对比。在第 10 章中，程序清单 10.7 演示了派生类对象的构造和析构过程；它表明，在析构过程中，需要调用所有相关的析构函数，包括~Tuna()；显然是什么地方出了问题。

这个程序清单表明，对于使用 new 在自由存储区中实例化的派生类对象，如果将其赋给基类指针，并通过该指针调用 delete，将不会调用派生类的析构函数。这可能导致资源未释放、内存泄露等问题，必须引起重视。

要避免这种问题，可将析构函数声明为虚函数，如程序清单 11.4 所示。

程序清单 11.4 将析构函数声明为虚函数，确保通过基类指针调用 delete 时，将调用派生类的析构函数

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
```

```
 5: public:
 6:    Fish()
 7:    {
 8:       cout << "Constructed Fish" << endl;
 9:    }
10:    virtual ~Fish()   // virtual destructor!
11:    {
12:       cout << "Destroyed Fish" << endl;
13:    }
14: };
15:
16: class Tuna:public Fish
17: {
18: public:
19:    Tuna()
20:    {
21:       cout << "Constructed Tuna" << endl;
22:    }
23:    ~Tuna()
24:    {
25:       cout << "Destroyed Tuna" << endl;
26:    }
27: };
28:
29: void DeleteFishMemory(Fish* pFish)
30: {
31:    delete pFish;
32: }
33:
34: int main()
35: {
36:    cout << "Allocating a Tuna on the free store:" << endl;
37:    Tuna* pTuna = new Tuna;
38:    cout << "Deleting the Tuna: " << endl;
39:    DeleteFishMemory(pTuna);
40:
41:    cout << "Instantiating a Tuna on the stack:" << endl;
42:    Tuna myDinner;
43:    cout << "Automatic destruction as it goes out of scope: " << endl;
44:
45:    return 0;
46: }
```

▼ 输出：

```
Allocating a Tuna on the free store:
Constructed Fish
Constructed Tuna
Deleting the Tuna:
Destroyed Tuna
Destroyed Fish
Instantiating a Tuna on the stack:
Constructed Fish
Constructed Tuna
Automatic destruction as it goes out of scope:
Destroyed Tuna
Destroyed Fish
```

▼ 分析：

相比于程序清单 11.3，程序清单 11.4 唯一不同的地方是，在第 10 行声明基类 Fish 的析构函数时，添加了关键字 virtual。这种修改导致将运算符 delete 用于 Fish 指针（如第 31 行所示）时，如果该指针

指向的是 Tuna 对象，则编译器不仅会执行 Fish::~Fish()，还会执行 Tuna::~Tuna()。输出还表明，无论 Tuna 对象是使用 new 在自由存储区中实例化的（第 37 行），还是以局部变量的方式在栈中实例化的，构造函数和析构函数的调用顺序都相同。

注意

务必像下面这样将基类的析构函数声明为虚函数：

```
class Base
{
public:
   virtual ~Base() {};  // virtual destructor
};
```

这可避免将 delete 用于 Base 指针时，不会调用派生类的析构函数的情况发生。

11.1.4 虚函数的工作原理——理解虚函数表

注意

对学习使用多态而言，并非必须掌握本节的内容。您可以跳过本节；如果想满足您的好奇心，也可阅读它。

在程序清单 11.2 的函数 MakeFishSwim()中，虽然程序员通过 Fish 引用调用 Swim()，但实际调用的却是方法 Carp::Swim()或 Tuna::Swim()。显然，在编译阶段，编译器并不知道将传递给该函数的是哪种对象，无法确保在不同的情况下执行不同的 Swim()方法。该调用哪个 Swim()方法显然是在运行阶段决定的，这是使用实现多态的不可见逻辑完成的，而这种逻辑是编译器在编译阶段提供的。

请看下面的 Base 类，它声明了 N 个虚函数：

```
class Base
{
public:
   virtual void Func1()
   {
      // Func1 implementation
   }
   virtual void Func2()
   {
      // Func2 implementation
   }

   // .. so on and so forth
   virtual void FuncN()
   {
      // FuncN implementation
   }
};
```

下面的 Derived 类继承了 Base 类，并覆盖了除 Base::Func2()外的其他所有虚函数：

```
class Derived: public Base
{
public:
   virtual void Func1()
   {
      // Func2 overrides Base::Func2()
   }

   // no implementation for Func2()

   virtual void FuncN()
   {
      // FuncN implementation
   }
};
```

编译器见到这种继承层次结构后，知道 Base 定义了一些虚函数，并在 Derived 中覆盖了它们。在这种情况下，编译器将为实现了虚函数的基类和覆盖了虚函数的派生类分别创建一个虚函数表（Virtual Function Table，VFT）。换句话说，Base 和 Derived 类都将有自己的虚函数表。实例化这些类的对象时，将创建一个隐藏的指针（我们称之为 VFT*），它指向相应的 VFT。可将 VFT 视为一个包含函数指针的静态数组，其中每个指针都指向相应的虚函数，如图 11.1 所示。

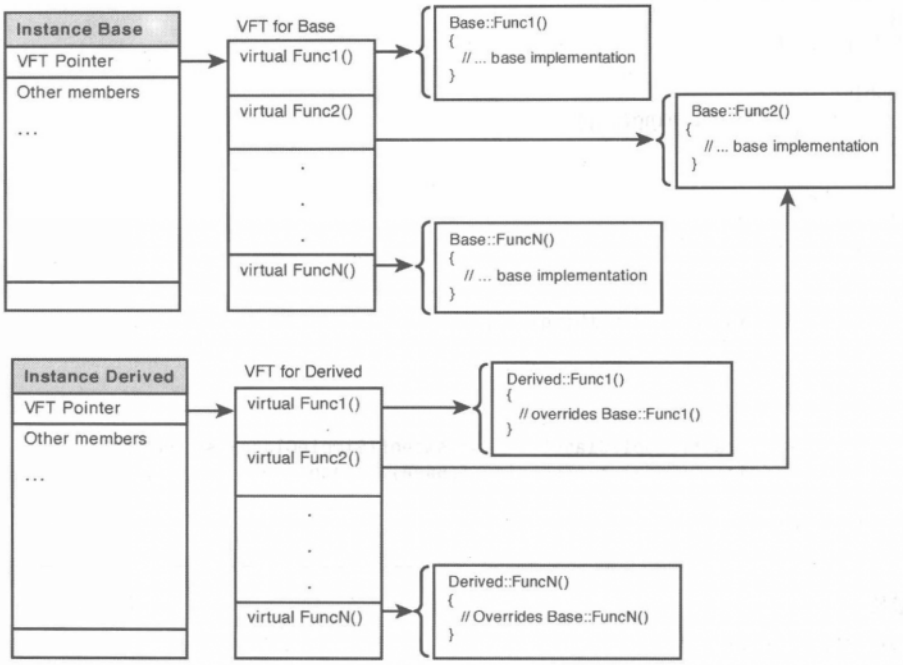


图 11.1 类 Base 和 Derived 的虚函数表

每个虚函数表都由函数指针组成，其中每个指针都指向相应虚函数的实现。在类 Derived 的虚函数表中，除一个函数指针外，其他所有函数指针都指向 Derived 本地的虚函数实现。Derived 没有覆盖 Base::Func2()，因此相应的函数指针指向 Base 类的 Func2()实现。

这意味着遇到下述代码时，编译器将查找 Derived 类的 VFT，确保调用 Base::Func2()的实现：

```
CDerived objDerived;
objDerived.Func2();
```

调用被覆盖的方法时，也将如此：

```
void DoSomething(Base& objBase)
{
   objBase.Func1();   // invoke Derived::Func1
}
int main()
{
   Derived objDerived;
   DoSomething(objDerived);
};
```

在这种情况下，虽然将 objDerived 传递给了 objBase，进而被解读为一个 Base 实例，但该实例的 VFT 指针仍指向 Derived 类的虚函数表，因此通过该 VTF 执行的是 Derived::Func1()。

虚函数表就是这样帮助实现 C++多态的。

程序清单 11.5 将 sizeof 用于两个相同的类（一个包含虚函数，另一个不包含），并对结果进行比较，从而证明了确实存在隐藏的虚函数表指针。

程序清单 11.5 对两个相同的类（一个包含虚函数，另一个不包含）进行比较，证明确实存在隐藏的虚函数表指针

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class SimpleClass
 4: {
 5:    int a, b;
 6:
 7: public:
 8:    void FuncDoSomething() {}
 9: };
10:
11: class Base
12: {
13:    int a, b;
14:
15: public:
16:    virtual void FuncDoSomething() {}
17: };
18:
19: int main()
20: {
21:    cout << "sizeof(SimpleClass) = " << sizeof(SimpleClass) << endl;
22:    cout << "sizeof(Base) = " << sizeof(Base) << endl;
23:
24:    return 0;
25: }
```

▼ 输出：

```
sizeof(SimpleClass) = 8
sizeof(Base) = 12
```

▼ 分析：

最大程度地简化了这个示例。其中有两个类——SimpleClass 和 Base，它们包含的成员数量和类型都相同，但在 Base 中，将 FuncDoSomething()声明成了虚函数，而在 SimpleClass 中没有这样做。添加关键字 virtual 带来的影响是，编译器将为 Base 类生成一个虚函数表，并其虚函数表指针（一个隐藏成员）预留空间。在 32 位系统中，Base 类占用的内存空间多了 4 字节，这证明确实存在这样的指针。

注意

在 C++中，可使用类型转换运算符 dynamic_cast 确定 Base 指针指向的是否是 Derived 对象，再根据结果执行额外的操作。

这被称为运行阶段类型识别（Run Time Type Identification，RTTI）。虽然大多数 C++编译器都支持 RTTI，但应尽可能避免这样做。因为需要知道基类指针指向的是派生类对象通常是一种糟糕的编程实践。

RTTI 和 dynamic_cast 将在第 13 章讨论。

11.1.5 抽象基类和纯虚函数

不能实例化的基类被称为抽象基类，这样的基类只有一个用途，那就是从它派生出其他类。在 C++中，要创建抽象基类，可声明纯虚函数。

以下述方式声明的虚函数被称为纯虚函数：

```
class AbstractBase
{
public:
   virtual void DoSomething() = 0;  // pure virtual method
};
```

该声明告诉编译器，AbstractBase 的派生类必须实现方法 DoSomething()：

```
class Derived: public AbstractBase
{
public:
   void DoSomething()   // pure virtual method
   {
      cout << "Implemented virtual function" << endl;
   }
};
```

AbstractBase 类要求 Derived 类必须提供虚方法 DoSomething()的实现。这让基类可指定派生类中方法的名称和特征（Signature），即指定派生类的接口。再次以 Tuna 类为例，假定它继承了 Fish 类，但没有覆盖 Fish::Swim()，因此不能游得很快。这种实现存在缺陷。通过将 Swim 声明为纯虚函数，让 Fish 变成抽象基类，可确保从 Fish 派生而来的 Tuna 类实现 Tuna::Swim()，从而像金枪鱼那样游动，如程序清单 11.6 所示。

程序清单 11.6　Fish 是 Tuna 和 Carp 的抽象基类

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    // define a pure virtual function Swim
 7:    virtual void Swim() = 0;
 8: };
 9:
10: class Tuna:public Fish
11: {
12: public:
13:    void Swim()
14:    {
15:       cout << "Tuna swims fast in the sea!" << endl;
16:    }
17: };
18:
19: class Carp:public Fish
20: {
21:    void Swim()
22:    {
23:       cout << "Carp swims slow in the lake!" << endl;
24:    }
25: };
26:
27: void MakeFishSwim(Fish& inputFish)
28: {
29:    inputFish.Swim();
30: }
31:
32: int main()
33: {
34:    // Fish myFish;   // Fails, cannot instantiate an ABC
35:    Carp myLunch;
36:    Tuna myDinner;
37:
38:    MakeFishSwim(myLunch);
```

```
39:    MakeFishSwim(myDinner);
40:
41:    return 0;
42: }
```

▼ 输出：

```
Carp swims slow in the lake!
Tuna swims fast in the sea!
```

▼ 分析：

main()的第 1 行（第 34 行）被注释掉了，它意义重大。它表明，编译器不允许您创建 Fish 实例。编译器要求您创建具体类（如 Tuna）的对象，这与现实世界一致。第 7 行声明了纯虚函数 Fish::Swim()，这迫使 Tuna 和 Carp 必须分别实现 Tuna::Swim()和 Carp::Swim()。第 27～30 行实现了 MakeFishSwim (Fish&)，这表明虽然不能实例化抽象基类，但可将指针或引用的类型指定为抽象基类。抽象基类提供了一种非常好的机制，让您能够声明所有派生类都必须实现的函数。如果 Trout 类从 Fish 类派生而来，但没有实现 Trout::Swim()，将无法通过编译。

注意 抽象基类常被简称为 ABC。

ABC 有助于约束程序的设计。

11.2 使用虚继承解决菱形问题

第 10 章介绍过，鸭嘴兽具备哺乳动物、鸟类和爬行动物的特征，这意味着 Platypus 类需要继承 Mammal、Bird 和 Reptile。然而，这些类都从同一个类——Animal 派生而来，如图 11.2 所示。

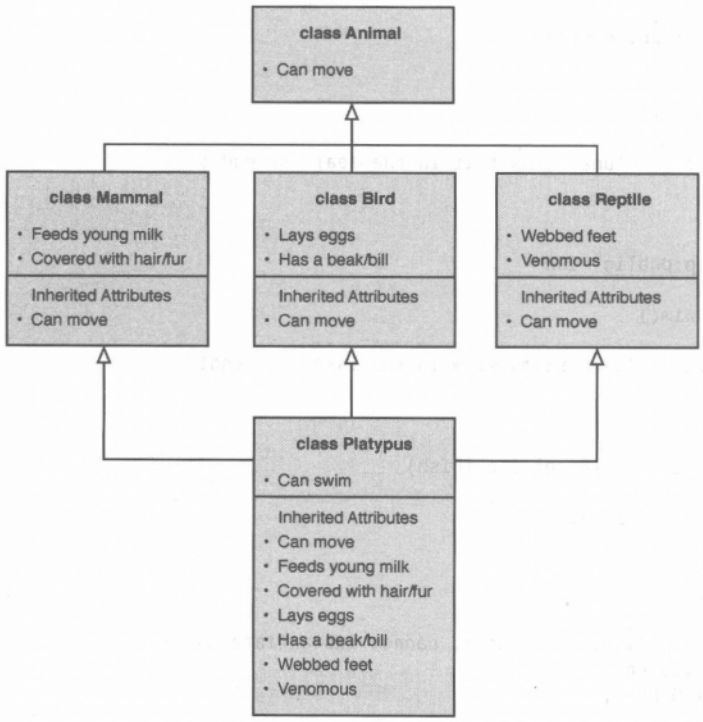


图 11.2 采用多继承的 Platypus 类的类图

实例化 Platypus 时，结果将如何呢？对于每个 Platypus 实例，将实例化多少个 Animal 实例呢？程序清单 11.7 回答了这个问题。

程序清单 11.7 每个 Platypus 实例包含多少个基类 Animal 的实例

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

class Animal
{
public:
   Animal()
   {
      cout << "Animal constructor" << endl;
   }

   // sample method
   int Age;
};

class Mammal:public Animal
{
};

class Bird:public Animal
{
};

class Reptile:public Animal
{
};

class Platypus:public Mammal, public Bird, public Reptile
{
public:
   Platypus()
   {
      cout << "Platypus constructor" << endl;
   }
};

int main()
{
   Platypus duckBilledP;

   // uncomment next line to see compile failure
   // Age is ambiguous as there are three instances of base Animal
   // duckBilledP.Age = 25;

   return 0;
}
```

▼ 输出：

```
Animal constructor
Animal constructor
Animal constructor
Platypus constructor
```

▼ 分析：

输出表明，由于采用了多继承，且 Platypus 的全部三个基类都是从 Animal 类派生而来的，因此第

38 行创建 Platypus 实例时，自动创建了三个 Animal 实例。这太可笑了，因为鸭嘴兽是一种动物，继承了哺乳动物、鸟类和爬行动物的属性。存在多个 Animal 实例带来的问题并非仅限于会占用更多内存。Animal 有一个整型成员——Animal::Age，为方便说明问题，将其声明成了公有的。如果您试图通过 Platypus 实例访问 Animal::Age（如第 42 行所示），将导致编译错误，因为编译器不知道您要设置 Mammal::Animal::Age、Bird::Animal::Age 还是 Reptile::Animal::Age。更可笑的是，如果您愿意，可以分别设置这三个属性：

```
duckBilledP.Mammal::Animal::Age = 25;
duckBilledP.Bird::Animal::Age = 25;
duckBilledP.Reptile::Animal::Age = 25;
```

显然，鸭嘴兽应该只有一个 Age 属性，但您希望 Platypus 类以公有方式继承 Mammal、Bird 和 Reptile。解决方案是使用虚继承。如果派生类可能被用作基类，派生它是最好使用关键字 virtual：

```
class Derived1: public virtual Base
{
   // ... members and functions
};
class Derived2: public virtual Base
{
   // ... members and functions
};
```

程序清单 11.8 列出了更佳的 Platypus 类声明（实际上是更佳的 Mammal、Bird 和 Reptile 类声明）。

程序清单 11.8　在继承层次结构中使用关键字 virtual，将基类 Animal 的实例个数限定为 1

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Animal
 4: {
 5: public:
 6:    Animal()
 7:    {
 8:       cout << "Animal constructor" << endl;
 9:    }
10:
11:    // sample method
12:    int Age;
13: };
14:
15: class Mammal:public virtual Animal
16: {
17: };
18:
19: class Bird:public virtual Animal
20: {
21: };
22:
23: class Reptile:public virtual Animal
24: {
25: };
26:
27: class Platypus:public Mammal, public Bird, public Reptile
28: {
29: public:
30:    Platypus()
31:    {
32:       cout << "Platypus constructor" << endl;
33:    }
34: };
```

```
35:
36: int main()
37: {
38:     Platypus duckBilledP;
39:
40:     // no compile error as there is only one Animal::Age
41:     duckBilledP.Age = 25;
42:
43:     return 0;
44: }
```

▼ **输出：**

```
Animal constructor
Platypus constructor
```

▼ **分析：**

如果将这里的输出与程序清单 11.7 的输出进行比较，将发现构造的 Animal 实例数减少到了 1 个，这表明只构造了一个 Platypus。这是因为从 Animal 类派生 Mammal、Bird 和 Reptile 类时，使用了关键字 virtual，这样 Platypus 继承这些类时，每个 Platypus 实例只包含一个 Animal 实例。这解决了很多问题，其中之一是第 41 行能够通过编译，不再像程序清单 11.7 中那样存在二义性。

注意

在继承层次结构中，继承多个从同一个类派生而来的基类时，如果这些基类没有采用虚继承，将导致二义性。这种二义性被称为菱形问题（Diamond Problem）。
其中的“菱形”可能源自类图的形状（如果使用直线和斜线表示 Platypus 经由 Mammal、Bird 和 Reptile 与 Animal 建立的关系，将形成一个菱形）。

注意

C++关键字 virtual 的含义随上下文而异（我想这样做的目的很可能是为了省事），对其含义总结如下：
在函数声明中，virtual 意味着当基类指针指向派生对象时，通过它可调用派生类的相应函数。
从 Base 类派生出 Derived1 和 Derived2 类时，如果使用了关键字 virtual，则意味着再从 Derived1 和 Derived2 派生出 Derived3 时，每个 Derived3 实例只包含一个 Base 实例。
也就是说，关键字 virtual 被用于实现两个不同的概念。

11.3 可将复制构造函数声明为虚函数吗

这个节标题是个疑问句，这是有道理的。从技术上说，C++不支持虚复制构造函数。但如果能实现虚复制构造函数，则创建一个基类指针集合（如静态数组，其中的每个元素指向不同的派生类对象）：

```
// Tuna, Carp and Trout are classes that inherit public from base class Fish
Fish* pFishes[3];
Fishes[0] = new Tuna();
Fishes[1] = new Carp();
Fishes[2] = new Trout();
```

并将其赋给另一个相同类型的数组时，则虽然是通过 Fish 指针调用的复制构造函数，但将复制指向的派生类对象，并对其进行深复制。

然而，这只是一种美好的梦想。

根本不可能实现虚复制构造函数，因为在基类方法声明中使用关键字 virtual 时，表示它将被派生类的实现覆盖，这种多态行为是在运行阶段实现的。而构造函数只能创建固定类型的对象，不具备多态性，因此 C++不允许使用虚复制构造函数。

虽然如此，存在一种不错的解决方案，就是定义自己的克隆函数来实现上述目的：

```
class Fish
{
public:
   virtual Fish* Clone() const = 0; // pure virtual function
};

class Tuna:public Fish
{
// ... other members
public:
   Tuna * Clone() const  // virtual clone function
   {
      return new Tuna(*this);  // return new Tuna that is a copy of this
   }
};
```

虚函数 Clone 模拟了虚复制构造函数，但需要显式地调用，如程序清单 11.9 所示。

程序清单 11.9 Tuna 和 Carp 包含 Clone 函数，它们模拟了虚复制构造函数

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    virtual Fish* Clone() = 0;
 7:    virtual void Swim() = 0;
 8: };
 9:
10: class Tuna: public Fish
11: {
12: public:
13:    Fish* Clone()
14:    {
15:       return new Tuna (*this);
16:    }
17:
18:    void Swim()
19:    {
20:       cout << "Tuna swims fast in the sea" << endl;
21:    }
22: };
23:
24: class Carp: public Fish
25: {
26:    Fish* Clone()
27:    {
28:       return new Carp(*this);
29:    }
30:    void Swim()
31:    {
32:       cout << "Carp swims slow in the lake" << endl;
33:    }
34: };
35:
36: int main()
37: {
38:    const int ARRAY_SIZE = 4;
39:
40:    Fish* myFishes[ARRAY_SIZE] = {NULL};
41:    myFishes[0] = new Tuna();
42:    myFishes[1] = new Carp();
```

```
43:    myFishes[2] = new Tuna();
44:    myFishes[3] = new Carp();
45:
46:    Fish* myNewFishes[ARRAY_SIZE];
47:    for (int Index = 0; Index < ARRAY_SIZE; ++Index)
48:       myNewFishes[Index] = myFishes[Index]->Clone();
49:
50:    // invoke a virtual method to check
51:    for (int Index = 0; Index < ARRAY_SIZE; ++Index)
52:       myNewFishes[Index]->Swim();
53:
54:    // memory cleanup
55:    for (int Index = 0; Index < ARRAY_SIZE; ++Index)
56:    {
57:       delete myFishes[Index];
58:       delete myNewFishes[Index];
59:    }
60:
61:    return 0;
62: }
```

▼ **输出：**

```
Tuna swims fast in the sea
Carp swims slow in the lake
Tuna swims fast in the sea
Carp swims slow in the lake
```

▼ **分析：**

在 main()中，第 40～44 行声明了一个静态基类指针（Fish *）数组，并各个元素分别设置为新创建的 Tuna、Carp、Tuna 和 Carp 对象。注意到 myFishes 数组能够存储不同类型的对象，这些对象都是从 Fish 派生而来的。这太酷了，因为本书前面的大部分数组包含的都是相同类型的数据，如 int。如果这还不够酷，您还可以在循环中使用虚函数 Fish::Clone 将其复制到另一个 Fish*数组（myNewFishes）中，如第 48 行所示。注意到这里的数组很小，只有 4 个元素，但即便数组长得多，复制逻辑也差别不大，只需调整循环结束条件即可。第 52 行进行了核实，它通过新数组的每个元素调用虚函数 Swim()，以验证 Clone()复制了整个派生类对象，而不仅仅是 Fish 部分。输出表明，确实像预期的那样复制了整个派生类对象。

应该	不应该
对于将被派生类覆盖的基类方法，务必将其声明为虚函数。 纯虚函数导致类变成抽象基类，且在派生类中必须提供纯虚函数的实现。 务必考虑使用虚继承。	别忘了给基类提供一个虚析构函数。 别忘了，编译器不允许您创建抽象基类的实例。 别忘了，在菱形继承层次结构中，虚继承旨在确保只有一个基类实例。 用于创建继承层次结构和声明基类函数时，关键字 virtual 的作用不同，请不要混为一谈。

11.4 总结

在本章中，您学习了如何使用多态，以充分发挥 C++继承的威力。您学习了如何声明和编写虚函数。通过基类指针或引用调用虚方法时，如果它指向的是派生类对象，将调用派生类的方法实现。纯虚函数是一种特殊的虚函数，确保基类不能被实例化，让这种基类非常适合用于

定义派生类必须实现的接口。最后，您学习了多继承导致的菱形问题以及如何使用虚继承解决这种问题。

11.5 问与答

问：为何在基类函数声明中使用关键字 virtual（因为即便不使用该关键字，代码也能通过编译）？

答：如果不使用关键字 virtual，就不能确保 objBase.Function()执行 Derived::Function()。另外，代码够通过编译并不意味着其质量上乘。

问：编译器为何创建虚函数表？

答：用于存储函数指针，确保调用正确的虚函数版本。

问：基类总应包含一个虚析构函数吗？

答：最好如此。如果编写了如下代码：

```
Base* pBase = new Derived();
delete pBase;
```

则仅当析构函数~Base()被声明为虚函数时，delete pBase 才会调用析构函数~Derived()。

问：抽象基类（ABC）都不能被实例化，它有何用途呢？

答：ABC 并非为实例化而创建的，而仅充当基类。它包含纯虚函数，指定了派生类必须实现哪些函数，可充当接口。

问：在继承层次结构中，需要在所有虚函数声明中都使用关键字 virtual，还是只需在基类中这样做？

答：只需在基类的虚函数声明中使用关键字 virtual 即可。

问：在 ABC 中，可定义成员函数和成员属性吗？

答：当然可以。这样的 ABC 也不能被实例化，因为它至少包含一个纯虚函数，派生类必须实现该函数。

11.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

11.6.1 测验

1. 假设您要模拟形状——圆和三角形，并要求每个形成都必须实现函数 Area()和 Print()。您该如何办？

2. 编译器为每个类都创建虚函数表吗？

3. 我编写了一个 Fish 类，它有两个公有方法、一个纯虚函数和几个成员属性。这个类是抽象基类吗？

11.6.2 练习

1. 创建一个继承层次结构，实现问题 1 中的 Circle 和 Triangle 类。
2. 差错：下面的代码有何问题？

```
class Vehicle
{
public:
   Vehicle() {}
   ~Vehicle(){}
};
class Car: public Vehicle
{
public:
   Car() {}
   ~Car() {}
};
```

3. 给定练习 2 所示的（错误）代码，像下面这样创建并销毁 Car 实例时，将按什么样的顺序执行构造函数和析构函数？

```
Vehicle* pMyRacer = new Car;
delete pMyRacer;
```

第 12 章
运算符类型与运算符重载

关键字 class 让您不仅能够封装数据和方法，还能封装运算符，以简化对对象执行的操作。通过使用这些运算符，可以像第 5 章处理整数那样，对对象执行赋值或加法运算。与函数一样，运算符也可以重载。

在本章中，您将学习：

- 使用关键字 operator；
- 单目运算符与双目运算符；
- 转换运算符；
- C++11 新增的移动复制运算符；
- 不能重新定义的运算符。

12.1 C++运算符

从语法层面看，除使用关键字 operator 外，运算符与函数几乎没有差别。运算符声明看起来与函数声明极其相似：

```
return_type operator operator_symbol (...parameter list...);
```

其中 operator_symbol 是程序员可定义的几种运算符类型之一。可以是+（加）、&&（逻辑 AND）等。编译器可根据操作数区分运算符。那么，C++在支持函数的情况下为何还要提供运算符呢？

来看封装了年、月、日的实用类 Date：

```
Date Holiday (25, 12, 2011);  // initialized to 25th Dec 2011
```

如果要将这个 Date 对象指向下一天（2011 年 12 月 26 日），下面两种方法哪种更方便、更直观呢？

方法 1（使用运算符）：

```
++ Holiday;
```

方法 2（使用虚构的函数 Date::Increment()）：

```
Holiday.Increment();  // 26th Dec 2011
```

显然，方法 1 优于方法 2。基于运算符的机制更容易使用，也更直观。通过在 Date 类中实现运算符<，将可以像下面这样对两个日期进行比较：

```
if(Date1 < Date2)
{
   // Do something
}
else
{
   // Do something else
}
```

运算符并非仅能用于管理日期的类。想想程序清单 9.9 所示的实用字符串类 MyString 吧，加法运算符（+）让您能够轻松地拼接字符串：

```
MyString sayHello ("Hello ");
MyString sayWorld ("world");
MyString sumThem (sayHello + sayWorld);  // uses operator+ (unavailable in
Listing 9.9)
```

注意 要实现相关运算符，需要做额外的工作，但类使用起来将更容易，因此值得这样做。

C++运算符分两大类：单目运算符与双目运算符。

12.2 单目运算符

顾名思义，单目运算符只对一个操作数进行操作。实现为全局函数或静态成员函数的单目运算符的典型定义如下：

```
return_type operator operator_type (parameter_type)
{
    // ... implementation
}
```

作为类成员的单目运算符的定义如下：

```
return_type operator operator_type ()
{
    // ... implementation
}
```

12.2.1 单目运算符的类型

可重载（或重新定义）的单目运算符如表 12.1 所示。

表 12.1 单目运算符

运算符	名称	运算符	名称
++	递增	&	取址
--	递减	~	求反
*	解除引用	+	正
->	成员选择	-	负
!	逻辑非	转换运算符	转换运算符

12.2.2 单目递增与单目递减运算符

要在类声明中编写单目前缀递增运算符（++），可采用如下语法：

```
Date& operator ++ ()
{
    // operator implementation code
    return *this;
}
```

而后缀递增运算符（++）的返回值不同，且有一个输入参数（但并非总是使用它）：

```
Date operator ++ (int)
{
   // Store a copy of the current state of the object, before incrementing day
```

```
    Date Copy (*this);

    // operator implementation code (that increments this object)

    // Return the state before increment was performed
    return Copy;
}
```

前缀和后缀递减运算符的声明语法与递增运算符类似，只是将声明中的++替换成了--。程序清单12.1是一个简单的Date类，让您能够使用运算符++对日期进行递增。

程序清单12.1 一个处理日、月、年的日历类，可对日期执行递增和递减操作

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Date
 4: {
 5: private:
 6:    int Day;  // Range: 1 - 30 (lets assume all months have 30 days!
 7:    int Month;
 8:    int Year;
 9:
10: public:
11:    // Constructor that initializes the object to a day, month and year
12:    Date (int InputDay, int InputMonth, int InputYear)
13:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
14:
15:    // Unary increment operator (prefix)
16:    Date& operator ++ ()
17:    {
18:       ++Day;
19:       return *this;
20:    }
21:
22:    // Unary decrement operator (prefix)
23:    Date& operator — ()
24:    {
25:       —Day;
26:       return *this;
27:    }
28:
29:    void DisplayDate ()
30:    {
31:       cout << Day << " / " << Month << " / " << Year << endl;
32:    }
33: };
34:
35: int main ()
36: {
37:    // Instantiate and initialize a date object to 25 Dec 2011
38:    Date Holiday (25, 12, 2011);
39:
40:    cout << "The date object is initialized to: ";
41:    Holiday.DisplayDate ();
42:
43:    // Applying the prefix increment operator
44:    ++ Holiday;
45:
46:    cout << "Date after prefix-increment is: ";
47:
48:    // Display date after incrementing
```

```
49:     Holiday.DisplayDate ();
50:
51:     — Holiday;
52:     — Holiday;
53:
54:     cout << "Date after two prefix-decrements is: ";
55:     Holiday.DisplayDate ();
56:
57:     return 0;
58: }
```

▼ 输出：

```
The date object is initialized to: 25 / 12 / 2011
Date after prefix-increment is: 26 / 12 / 2011
Date after two prefix-decrements is: 24 / 12 / 2011
```

▼ 分析：

我们感兴趣的代码是第 16～27 行，它们递增和递减运算符的实现，这些运算符让您能够将 Date 对象存储的日期向前或向后推一天，如 main()中的第 44、51 和 52 行所示。前缀递增运算符先执行递增操作，再返回指向当前对象的引用。

注意

这个版本的 Date 类做了最大程度的简化，只阐述了如何实现前缀递增运算符（++）和前缀递减运算符（--）。这里假定每个月都包含 30 天，且没有考虑导致月份甚至年份加 1 的情形。

要支持后缀递增和递减运算符，只需在 Date 类中添加如下代码：

```
// postfix differs from prefix operator in return-type and parameters
Date operator ++ (int)
{
   // Store a copy of the current state of the object, before incrementing day
   Date Copy (Day, Month, Year);

   ++Day;

   // Return the state before increment was performed
   return Copy;
}
// postfix decrement operator
Date operator — (int)
{
   Date Copy (Day, Month, Year);

   —Day;

   return Copy;
}
```

这样，就可像下面这样使用 Date 对象了：

```
Date Holiday (25, 12, 2011);  // instantiate
++ Holiday;  // using prefix increment operator++
Holiday ++;  // using postfix increment operator++
— Holiday;  // using prefix decrement operator—
Holiday —;  // using postfix decrement operator—
```

注意

在上述后缀运算符的实现中，首先复制了当前对象，再将对当前对象执行递增或递减运算，最后返回复制的对象。

换句话说，如果只想执行递增运算，可使用++ object，也可使用 object ++，但应选择前者，这样可避免创建一个未被使用的临时拷贝。

12.2.3 转换运算符

在程序清单 12.1 的 main()中，如果添加下述代码行：

```
cout << Holiday;  // error in absence of conversion operator
```

将导致这样的编译错误：error: binary '<<' : no operator found which takes a right-hand operand of type 'Date' (or there is no acceptable conversion)。这种错误表明，cout 不知道如何解读 Date 实例，因为 Date 类不支持相关的运算符。

然而，cout 能够很好地显示 const char *：

```
std::cout << "Hello world"; // const char* works!
```

因此，要让 cout 能够显示 Date 对象，只需添加一个返回 const char*的运算符：

```
operator const char*()
{
   // operator implementation that returns a char*
}
```

程序清单 12.2 提供了该运算符的简单实现。

程序清单 12.2 使用转换运算符将 Date 转换为 const char*

```
 0: #include <iostream>
 1: #include <sstream>
 2: #include <string>
 3: using namespace std;
 4:
 5: class Date
 6: {
 7: private:
 8:    int Day;  // Range: 1 - 30 (lets assume all months have 30 days!
 9:    int Month;
10:    int Year;
11:
12:    string DateInString;
13:
14: public:
15:
16:    // Constructor that initializes the object to a day, month and year
17:    Date (int InputDay, int InputMonth, int InputYear)
18:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
19:
20:    operator const char*()
21:    {
22:       ostringstream formattedDate;
23:       formattedDate << Day << " / " << Month << " / " << Year;
24:
25:       DateInString = formattedDate.str();
26:       return DateInString.c_str();
27:    }
28: };
29:
30: int main ()
31: {
32:    // Instantiate and initialize a date object to 25 Dec 2011
33:    Date Holiday (25, 12, 2011);
34:
35:    cout << "Holiday is on: " << Holiday << endl;
36:
37:    return 0;
38: }
```

▼ 输出：

```
Holiday is on: 25 / 12 / 2011
```

▼ 分析：

第 20～27 行实现了将 Date 转换为 const char*的运算符，main()中的第 35 行演示了这样做的好处。现在，可在 cout 语句中直接使用 Date 对象，因为 cout 能够理解 const char*。编译器自动将合适运算符（这里只有一个）的返回值提供给 cout，从而在屏幕上显示日期。在转换为 const char*的运算符中，使用 std::ostringstream 将整型成员转换成了一个 std::string 对象，如第 23～25 所示。原本也可直接返回 formattedDate.str()，但没有这样做，而将其拷贝存储在私有成员 Date::DateInString 中，如第 25 行所示。这是因为 formattedDate 是一个局部变量，将在运算符返回时被销毁，因此运算符返回时，通过 str()获得的指针将无效。

这个运算符让您能够以新的方式使用 Date 类。现在，您甚至可以将 Date 对象直接赋给 string 对象：

```
string strHoliday (Holiday); // OK! Compiler invokes operator const char*
strHoliday = Date(11, 11, 2011); // also OK!
```

注意

应根据类的可能用法编写尽可能多的运算符。如果应用程序需要 Date 对象的整数表示，可编写如下转换运算符：

```
operator int()
{
   // your conversion code here
}
```

这样便可将 Date 对象当做整数使用：

```
SomeFuncThatTakesInt(Date(25, 12, 2011));
```

12.2.4 解除引用运算符（*）和成员选择运算符（->）

解除引用运算符（*）和成员选择运算符（->）在智能指针类编程中应用最广。智能指针是封装常规指针的类，旨在通过管理所有权和复制问题简化内存管理。在有些情况下，智能指针甚至能够提高应用程序的性能。智能指针将在第 26 章详细讨论，这里只简要地如何重载运算符，以帮助智能指针完成其工作。

请看程序清单 12.3 中 std::unique_prt 的用法，它使用了运算符*和->，让您能够像使用普通指针那样使用智能指针类。

程序清单 12.3 使用智能指针 std:: unique_prt 管理动态分配的 Date 对象

```
 0: #include <iostream>
 1: #include <memory>  // include this to use std::unique_ptr
 2: using namespace std;
 3:
 4: class Date
 5: {
 6: private:
 7:    int Day;
 8:    int Month;
 9:    int Year;
10:
11:    string DateInString;
12:
13: public:
14:    // Constructor that initializes the object to a day, month and year
15:    Date (int InputDay, int InputMonth, int InputYear)
```

```
16:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
17:
18:    void DisplayDate()
19:    {
20:       cout << Day << " / " << Month << " / " << Year << endl;
21:    }
22: };
23:
24: int main()
25: {
26:    unique_ptr<int> pDynamicAllocInteger(new int);
27:    *pDynamicAllocInteger = 42;
28:
29:    // Use smart pointer type like an int*
30:    cout << "Integer value is: " << *pDynamicAllocInteger << endl;
31:
32:    unique_ptr<Date> pHoliday (new Date(25, 11, 2011));
33:    cout << "The new instance of date contains: ";
34:
35:    // use pHoliday just as you would a Date*
36:    pHoliday->DisplayDate();
37:
38:    // no need to do the following when using unique_ptr:
39:    // delete pDynamicAllocInteger;
40:    // delete pHoliday;
41:
42:    return 0;
43: }
```

▼ 输出：

```
Integer value is: 42
The new instance of date contains: 25 / 11 / 2011
```

▼ 分析：

第 26 行声明了一个指向 int 的智能指针，它演示了智能指针类 unique_ptr 的模板初始化语法。同样，第 32 行声明了一个指向 Date 对象的智能指针。这里的重点是模式，请暂时不要考虑细节。

注意 如果这种模板语法看起来很难理解，也不用担心，因为第 14 章将讨论模板。

这个示例表明，可像使用普通指针那样使用智能指针，如第 30 和 36 行所示。第 30 行使用了 *pDynamicAllocInteger 来显示指向的 int 值，而第 36 行使用了 pHoliday->DisplayData()，就像这两个变量的类型分别是 int*和 Date*。其中的秘诀在于，智能指针类 std::unique_ptr 实现了运算符*和->。程序清单 12.4 是一个简单而基本的智能指针类的实现。

程序清单 12.4 在一个简单的智能指针类中实现运算符*和->

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: template <typename T>
 4: class smart_pointer
 5: {
 6: private:
 7:     T* m_pRawPointer;
 8: public:
 9:      smart_pointer (T* pData) : m_pRawPointer (pData) {}     // constructor
10:      ~smart_pointer () {delete m_pRawPointer ;}          // destructor
11:
```

```
12:      T& operator* () const    // dereferencing operator
13:      {
14:         return *(m_pRawPointer);
15:      }
16:
17:      T* operator-> () const    // member selection operator
18:      {
19:          return m_pRawPointer;
20:      }
21:  };
22:
23: class Date
24: {
25: private:
26:    int Day, Month, Year;
27:    string DateInString;
28:
29: public:
30:    // Constructor that initializes the object to a day, month and year
31:    Date (int InputDay, int InputMonth, int InputYear)
32:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
33:
34:    void DisplayDate()
35:    {
36:       cout << Day << " / " << Month << " / " << Year << endl;
37:    }
38: };
39:
40: int main()
41: {
42:    smart_pointer<int> pDynamicInt(new int (42));
43:    cout << "Dynamically allocated integer value = " << *pDynamicInt;
44:
45:    smart_pointer<Date> pDate(new Date(25, 12, 2011));
46:    cout << "Date is = ";
47:    pDate->DisplayDate();
48:
49:    return 0;
50: }
```

▼ 输出：

```
Dynamically allocated integer value = 42
Date is = 25 / 12 / 2011
```

▼ 分析：

这是程序清单 12.3 的翻版，但使用了自己的 smart_pointer 类，它是在第 3～21 行定义的。这里使用了模板声明语法，以便能够对该智能指针进行定制，以指向任何类型，如 int（第 42 行）或 Date（第 45 行）。这个智能指针类包含一个私有成员，其类型与指向的数据的类型相同；该成员是在第 7 行声明的。基本上，该智能指针类旨在自动管理该成员指向的资源，包括在析构函数中自动释放它，如第 10 行所示。这个析构函数确保即使您使用 new 创建了指向的对象，也无需对其调用 delete，且不会导致内存泄露。请将重点放在运算符*的实现上，如第 12～15 行所示。其返回类型为 T&，即一个引用，指向具体化该模板时指定的类型。该实现返回一个引用，指向智能指针指向的实例。同样，运算符->（如第 17～20 行所示）的返回类型为 T*，即一个指针，指向具体化该模板时指定的类型。在运算符->的实现中，第 19 行返回了成员指针。总之，这两个运算符让 smart_pointer 类隐藏了对原始指针进行内存管理的方式，让您能够像使用普通指针一样使用它，因此是智能指针。

注意

与普通指针相比，除能够在指针离开作用域后释放其占用的内存外，智能指针还有很多其他功能，这将在第26章详细讨论。

如果读者对程序清单12.3中unique_prt的用法感到好奇，可参考编译器或IDE提供的头文件<memory>中的unique_prt实现，以了解它在幕后所做的工作。

12.3 双目运算符

对两个操作数进行操作的运算符称为双目运算符。以全局函数或静态成员函数的方式实现的双目运算符的定义如下：

```
return_type operator_type (parameter1, parameter2);
```

以类成员的方式实现的双目运算符的定义如下：

```
return_type operator_type (parameter);
```

以类成员的方式实现的双目运算符只接受一个参数，其原因是第二个参数通常是从类属性获得的。

12.3.1 双目运算符的类型

表12.2列出了可在C++应用程序中重载或重新定义的双目运算符。

表12.2 可重载的双目运算符

运算符	名称	运算符	名称
,	逗号	<	小于
!=	不等于	<<	左移
%	求模	<<=	左移并赋值
%=	求模并赋值	<=	小于或等于
&	按位与	=	赋值、复制赋值和移动赋值
&&	逻辑与	==	等于
&=	按位与并赋值	>	大于
*	乘	>=	大于或等于
*=	乘并赋值	>>	右移
+	加	>>=	右移并赋值
+=	加并赋值	^	异或
-	减	^=	异或并赋值
-=	减并赋值	|	按位或
->*	指向成员的指针	|=	按位或并赋值
/	除	||	逻辑或
/=	除并赋值	[]	下标运算符

12.3.2 双目加法与双目减法运算符

与递增/递减运算符类似，如果类实现了双目加法和双目减法运算符，便可将其对象加上或减去指定类型的值。再来看看日历类Date，虽然前面实现了将Date递增以便前移一天的功能，但它还不支持增加5天的功能。为实现这种功能，需要实现双目加法运算符，如程序清单12.5中的代码所示。

程序清单 12.5 实现了双目加法运算符的日历类

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Date
 4: {
 5: private:
 6:    int Day, Month, Year;
 7:
 8: public:
 9:
10:    // Constructor that initializes the object to a day, month and year
11:    Date (int InputDay, int InputMonth, int InputYear)
12:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:    // Binary addition operator
15:    Date operator + (int DaysToAdd)
16:    {
17:       Date newDate (Day + DaysToAdd, Month, Year);
18:       return newDate;
19:    }
20:
21:    // Binary subtraction operator
22:    Date operator - (int DaysToSub)
23:    {
24:       return Date(Day - DaysToSub, Month, Year);
25:    }
26:
27:    void DisplayDate ()
28:    {
29:       cout << Day << " / " << Month << " / " << Year << endl;
30:    }
31: };
32:
33: int main()
34: {
35:    // Instantiate and initialize a date object to 25 Dec 2011
36:    Date Holiday (25, 12, 2011);
37:
38:    cout << "Holiday on: ";
39:    Holiday.DisplayDate ();
40:
41:    Date PreviousHoliday (Holiday - 19);
42:    cout << "Previous holiday on: ";
43:    PreviousHoliday.DisplayDate();
44:
45:    Date NextHoliday(Holiday + 6);
46:    cout << "Next holiday on: ";
47:    NextHoliday.DisplayDate ();
48:
49:    return 0;
50: }
```

▼ 输出：

```
Holiday on: 25 / 12 / 2011
Previous holiday on: 6 / 12 / 2011
Next holiday on: 31 / 12 / 2011
```

▼ 分析：

第 14～25 行是双目运算符+和-的实现，让您能够使用简单的加法和减法语法，如 main()中的第

41 和 45 行所示。

对字符串类来说，双目加法运算符也很有用。第 9 章分析了简单的字符串包装类 MyString，它封装了一个 C 风格字符串，并提供了内存管理、复制等功能，如程序清单 9.9 所示。但这个类不支持使用如下语法将两个字符串拼接起来：

```
MyString Hello("Hello ");
MyString World(" World");
MyString HelloWorld(Hello + World);  // error: operator+ not defined
```

不用说，实现运算符+后，MyString 使用起来将非常容易，值得去实现它：

```
MyString operator+ (const MyString& AddThis)
{
   MyString NewString;

   if (AddThis.Buffer != NULL)
   {
      NewString.Buffer = new char[GetLength() + strlen(AddThis.Buffer) + 1];
      strcpy(NewString.Buffer, Buffer);
      strcat(NewString.Buffer, AddThis.Buffer);
   }

   return NewString;
}
```

程序清单 9.9 中添加上述代码，并提供实现为空的私有默认构造函数 MyString()后，便可使用加法语法了。本章后面的程序清单 12.12 提供了一个 MyString 类，它实现了+等运算符。

注意

运算符提高了类的可用性，但实现的运算符必须合理。对于 Date 类，您实现了加法和减法运算符，但对于 MyString 类，只实现了加法运算符（+）。这是因为对字符串执行减法运算的可能性极少，实现这样的运算符很可能是在浪费时间。

12.3.3 实现运算符+=与-=

加并赋值运算符支持语法 a+=b;，这让程序员可将对象 a 增加 b。这样，程序员可重载加并赋值运算符，使其接受不同类型的参数 b。程序清单 12.6 让您能够给 Date 对象加上一个整数。

程序清单 12.6 定义运算符+=和-=，以便将日历向前或向后翻整型输入参数指定的天数

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Date
 4: {
 5: private:
 6:    int Day, Month, Year;
 7:
 8: public:
 9:
10:    // Constructor that initializes the object to a day, month and year
11:    Date (int InputDay, int InputMonth, int InputYear)
12:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:    // Binary addition assignment
15:    void operator+= (int DaysToAdd)
16:    {
17:       Day += DaysToAdd;
18:    }
19:
20:    // Binary subtraction assignment
```

```
21:     void operator-= (int DaysToSub)
22:     {
23:        Day -= DaysToSub;
24:     }
25:
26:     void DisplayDate ()
27:     {
28:        cout << Day << " / " << Month << " / " << Year << endl;
29:     }
30: };
31:
32: int main()
33: {
34:     // Instantiate and initialize a date object to 25 Dec 2011
35:     Date Holiday (25, 12, 2011);
36:
37:     cout << "Holiday is on: ";
38:     Holiday.DisplayDate ();
39:
40:     cout << "Holiday -= 19 gives: ";
41:     Holiday -= 19;
42:     Holiday.DisplayDate();
43:
44:     cout << "Holiday += 25 gives: ";
45:     Holiday += 25;
46:     Holiday.DisplayDate ();
47:
48:     return 0;
49: }
```

▼ 输出：

```
Holiday is on: 25 / 12 / 2011
Holiday -= 19 gives: 6 / 12 / 2011
Holiday += 25 gives: 31 / 12 / 2011
```

▼ 分析：

运算符+=和-=是在第 14～24 行定义的。这些运算符让您能够加上或减去指定的天数，如 main() 中的下述代码所示：

```
41:     Holiday -= 19;
45:     Holiday += 25;
```

运算符+=和-=接受一个 int 参数，让您能够给 Date 对象加上或减去指定的天数，就像处理的是整数一样。您还可提供运算符+=的重载版本，让它接受一个虚构的 CDays 对象作为参数：

```
// The addition-assignment operator that add a CDays to an existing Date
void operator += (const CDays& mDaysToAdd)
{
    Day += mDaysToAdd.GetDays ();
}
```

注意

乘并赋值运算符（*=）、除并赋值运算符（/=）、求模并赋值运算符（%=）、减并赋值运算符（-=）、左移并赋值运算符（<<=）、右移并赋值运算符（>>=）、异或并赋值运算符（^=）、按位或并赋值运算符（|=）以及按位与并赋值运算符（&=）的语法都与程序清单 12.6 所示的加并赋值运算符类似。

虽然重载运算符的最终目标是让类更直观，更易于使用，但很多时候实现这些运算符并没有意义。例如，前面的日历类 Date 绝对不会用到按位与并赋值运算符&=。这个类的用户应该不会想通过 greatDay &= 20; 等操作获得有用的结果。

12.3.4 重载等于运算符（==）和不等运算符（!=）

如果像下面这样将两个 Date 对象进行比较，结果将如何呢？

```
if (Date1 == Date2)
{
    // Do something
}
else
{
    // Do something else
}
```

由于还没有定义等于运算符，编译器将对这两个对象进行二进制比较，并仅当它们完全相同时才返回 true。在有些情况下（包括现在的 Date 类），这是可行的。然而，如果类有一个非静态字符串成员，它包含字符串值（char *），如程序清单 9.9 所示的 MyString，则比较结果可能不符合预期。在这种情况下，对成员属性进行二进制比较时，实际上将比较字符串指针，而字符串指针并不相等（即使指向的内容相同），因此总是返回 false。

因此，正确的做法是定义比较运算符。等于运算符的通用实现如下：

```
bool operator== (const ClassType& compareTo)
{
   // comparison code here, return true if equal else false
}
```

实现不等运算符时，可重用等于运算符：

```
bool operator!= (const ClassType& compareTo)
{
   // comparison code here, return true if inequal else false

}
```

不等运算符的结果与等于运算符相反（逻辑非）。程序清单 12.7 列出了日历类 Date 定义的比较运算符。

程序清单 12.7 运算符==和!=

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Date
 4: {
 5: private:
 6:    int Day, Month, Year;
 7:
 8: public:
 9:
10:    // Constructor that initializes the object to a day, month and year
11:    Date (int InputDay, int InputMonth, int InputYear)
12:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:    bool operator== (const Date& compareTo)
15:    {
16:       return ((Day == compareTo.Day)
17:             && (Month == compareTo.Month)
18:            && (Year == compareTo.Year));
19:    }
20:
21:    bool operator!= (const Date& compareTo)
22:    {
23:       return !(this->operator==(compareTo));
24:    }
```

```
25:
26:    void DisplayDate ()
27:    {
28:       cout << Day << " / " << Month << " / " << Year << endl;
29:    }
30: };
31:
32: int main()
33: {
34:    Date Holiday1 (25, 12, 2011);
35:    Date Holiday2 (31, 12, 2011);
36:
37:    cout << "Holiday 1 is: ";
38:    Holiday1.DisplayDate();
39:    cout << "Holiday 2 is: ";
40:    Holiday2.DisplayDate();
41:
42:    if (Holiday1 == Holiday2)
43:       cout << "Equality operator: The two are on the same day" << endl;
44:    else
45:       cout << "Equality operator: The two are on different days" << endl;
46:
47:    if (Holiday1 != Holiday2)
48:       cout << "Inequality operator: The two are on different days" << endl;
49:    else
50:       cout << "Inequality operator: The two are on the same day" << endl;
51:
52:    return 0;
53: }
```

▼ **输出：**

```
Holiday 1 is: 25 / 12 / 2011
Holiday 2 is: 31 / 12 / 2011
Equality operator: The two are on different days
Inequality operator: The two are on different days
```

▼ **分析：**

等于运算符（==）的实现很简单，它在年、月、日都相同时返回 true，如第 14～19 行所示。实现不等运算符时，重用了等于运算符的代码，如第 23 行所示。有了这两个运算符后，就可对两个 Date 对象（Holiday1 和 Holiday2）进行比较了，如 main()中的第 42 和 47 行所示。

12.3.5 重载运算符<、>、<=和>=

程序清单 12.7 所示的代码让 Date 类足够聪明，能够判断两个 Date 对象是否相等。然而，如果要使用该类执行类似下面的条件检查，该如何办呢？

```
if (Date1 < Date2) {// do something}
```

或

```
if (Date1 <= Date2) {// do something}
```

或

```
if (Date1 > Date2) {// do something}
```

或

```
if (greatDay >= Date2) {// do something}
```

如果能够使用这个日历类来比较两个日期，确定哪个在前、哪个在后，将很有用。编写这类的程序员应实现这种比较，让这个类对用户来说尽可能友好和直观，如程序清单 12.8 所示。

程序清单 12.8 实现运算符<、>、<=和>=

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Date
 4: {
 5: private:
 6:    int Day, Month, Year;
 7:
 8: public:
 9:
10:    // Constructor that initializes the object to a day, month and year
11:    Date (int InputDay, int InputMonth, int InputYear)
12:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:    bool operator== (const Date& compareTo)
15:    {
16:       return ((Day == compareTo.Day)
17:             && (Month == compareTo.Month)
18:            && (Year == compareTo.Year));
19:    }
20:
21:    bool operator< (const Date& compareTo)
22:    {
23:       if (Year < compareTo.Year)
24:          return true;
25:       else if (Month < compareTo.Month)
26:          return true;
27:       else if (Day < compareTo.Day)
28:          return true;
29:       else
30:          return false;
31:    }
32:
33:    bool operator<= (const Date& compareTo)
34:    {
35:       if (this->operator== (compareTo))
36:          return true;
37:       else
38:          return this->operator< (compareTo);
39:    }
40:
41:    bool operator > (const Date& compareTo)
42:    {
43:       return !(this->operator<= (compareTo));
44:    }
45:
46:    bool operator>= (const Date& compareTo)
47:    {
48:       if(this->operator== (compareTo))
49:          return true;
50:       else
51:          return this->operator> (compareTo);
52:    }
53:
54:    bool operator!= (const Date& compareTo)
55:    {
56:       return !(this->operator==(compareTo));
57:    }
58:
59:    void DisplayDate ()
```

```
60:    {
61:       cout << Day << " / " << Month << " / " << Year << endl;
62:    }
63: };
64:
65: int main()
66: {
67:    Date Holiday1 (25, 12, 2011);
68:    Date Holiday2 (31, 12, 2011);
69:
70:    cout << "Holiday 1 is: ";
71:    Holiday1.DisplayDate();
72:    cout << "Holiday 2 is: ";
73:    Holiday2.DisplayDate();
74:
75:    if (Holiday1 < Holiday2)
76:       cout << "operator<: Holiday1 happens first" << endl;
77:
78:    if (Holiday2 > Holiday1)
79:       cout << "operator>: Holiday2 happens later" << endl;
80:
81:    if (Holiday1 <= Holiday2)
82:       cout << "operator<=: Holiday1 happens on or before Holiday2" << endl;
83:
84:    if (Holiday2 >= Holiday1)
85:       cout << "operator>=: Holiday2 happens on or after Holiday1" << endl;
86:
87:    return 0;
88: }
```

▼ **输出：**

```
Holiday 1 is: 25 / 12 / 2011
Holiday 2 is: 31 / 12 / 2011
operator<: Holiday1 happens first
operator>: Holiday2 happens later
operator<=: Holiday1 happens on or before Holiday2
operator>=: Holiday2 happens on or after Holiday1
```

▼ **分析：**

这里要讨论的运算符是在第21～52行实现的。注意到实现这些运算符时，重用了其他运算符的代码。在main()函数的第75～84行，使用了这些运算符，以演示这些运算符使得使用Date类简单而直观。

12.3.6 重载复制赋值运算符（=）

有时候，需要将一个类实例的内容赋给另一个类实例，如下所示：

```
Date Holiday(25, 12, 2011);
Date AnotherHoliday(1, 1, 2010);
AnotherHoliday  = Holiday;  // uses copy assignment operator
```

如果您没有提供复制赋值运算符，这将调用编译器自动给类添加的默认复制赋值运算符。根据类的特征，默认复制赋值运算符可能不可行，具体地说是它不复制类管理的资源。与复制构造函数一样，为确保进行深复制，您需要提供复制赋值运算符：

```
ClassType& operator= (const ClassType& CopySource)
{
   if(this != &copySource)  // protection against copy into self
   {
      // Assignment operator implementation
   }
   return *this;
}
```

如果类封装了原始指针，如程序清单 9.9 所示的 MyString 类，则确保进行深复制很重要。如果没有实现赋值运算符，编译器将提供默认的复制赋值运算符，但它只复制 char* Buffer 包含的地址，而不复制指向的内存中的内容。这与没有提供复制构造函数时出现的情况相同。为确保赋值时进行深复制，应定义复制赋值运算符，如程序清单 12.9 所示。

程序清单 12.9 对程序清单 9.9 所示的 MyString 类进行改进，添加了复制赋值运算符

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MyString
 4: {
 5: private:
 6:    char* Buffer;
 7:
 8: public:
 9:    // constructor
10:    MyString(const char* InitialInput)
11:    {
12:       if(InitialInput != NULL)
13:       {
14:          Buffer = new char [strlen(InitialInput) + 1];
15:          strcpy(Buffer, InitialInput);
16:      }
17:       else
18:          Buffer = NULL;
19:    }
20:
21:    // insert copy constructor from Listing 9.9
22:    MyString(const MyString& CopySource);
23:
24:    // Copy assignment operator
25:    MyString& operator= (const MyString& CopySource)
26:    {
27:       if ((this != &CopySource) && (CopySource.Buffer != NULL))
28:       {
29:          if (Buffer != NULL)
30:           delete[] Buffer;
31:
32:          // ensure deep copy by first allocating own buffer
33:          Buffer = new char [strlen(CopySource.Buffer) + 1];
34:
35:          // copy from the source into local buffer
36:          strcpy(Buffer, CopySource.Buffer);
37:       }
38:      return *this;
39:    }
40:
41:    // Destructor
42:    ~MyString()
43:    {
44:       if (Buffer != NULL)
45:          delete [] Buffer;
46:    }
47:
48:    int GetLength()
49:    {
50:       return strlen(Buffer);
51:    }
52:
```

```
53:    operator const char*()
54:    {
55:       return Buffer;
56:    }
57: };
58:
59: int main()
60: {
61:    MyString String1("Hello ");
62:    MyString String2(" World");
63:
64:    cout << "Before assignment: " << endl;
65:    cout << String1 << String2 << endl;
66:    String2 = String1;
67:    cout << "After assignment String2 = String1: " << endl;
68:    cout << String1 << String2 << endl;
69:
70:    return 0;
71: }
```

▼ **输出：**

```
Before assignment:
Hello  World
After assignment String2 = String1:
Hello Hello
```

▼ **分析：**

在这个示例中，笔者故意省略了复制构造函数，旨在减少代码行（但您编写这样的类时，应添加它，详情请参阅程序清单 9.9）。复制赋值运算符是在第 25～39 行实现的，其功能与复制构造函数很像。它首先检查源和目标是否同一个对象。如果不是，则释放成员 Buffer 占用的内存，再重新给它分配足以存储复制源中文本的内存，然后使用 strcpy()进行复制，如第 36 行所示。

注意

相比于程序清单 9.9，程序清单 12.9 的另一个细微差别在于，使用返回 const char*的转换运算符替代了函数 GetString()，如第 53～56 行所示。该运算符让 MyString 类使用起来更容易，如第 68 行所示——使用一条 cout 语句显示了两个 MyString 实例的内容。

警告

如果您编写的类管理着动态分配的资源（如 C 风格字符串 char*）、动态数组等，除构造函数和析构函数外，请务必实现复制构造函数和复制赋值运算符。
如果没有考虑对象被复制时出现的资源所有权问题，您的类就是不完整的，使用时甚至会有危险。

提示

要创建不允许复制的类，可将复制构造函数和复制赋值运算符都声明为私有的。只需这样声明（甚至都不用提供实现）就足以让编译器在遇到试图复制对象（将对象按值传递给函数或将一个对象赋给另一个对象）的代码时引发错误。

12.3.7 下标运算符

下标运算符让您能够像访问数组那样访问类，其典型语法如下：

```
return_type& operator [] (subscript_type& subscript);
```

编写封装了动态数组的类（如封装了 char* Buffer 的 MyString）时，通过实现下标运算符，可轻松地随机访问缓冲区中的各个字符：

```
class MyString
{
    // ... other class members
public:
    /*const*/ char& operator [] (int Index) /*const*/
   {
        // return the char at position Index in Buffer
   }
};
```

程序清单 12.10 是一个简单的示例，演示了下标运算符（[]）让用户能够使用常规数组语法来遍历 MyString 实例包含的字符。

程序清单 12.10 在 MyString 类中实现下标运算符，以便随机访问 MyString::Buffer 包含的字符

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class MyString
 5: {
 6: private:
 7:    char* Buffer;
 8:
 9:    // private default constructor
10:    MyString() {}
11:
12: public:
13:    // constructor
14:    MyString(const char* InitialInput)
15:    {
16:       if(InitialInput != NULL)
17:       {
18:          Buffer = new char [strlen(InitialInput) + 1];
19:          strcpy(Buffer, InitialInput);
20:       }
21:       else
22:          Buffer = NULL;
23:    }
24:
25:    // Copy constructor: insert from Listing 9.9 here
26:    MyString(const MyString& CopySource);
27:
28:    // Copy assignment operator: insert from Listing 12.9 here
29:    MyString& operator= (const MyString& CopySource);
30:
31:    const char& operator[] (int Index) const
32:    {
33:       if (Index < GetLength())
34:          return Buffer[Index];
35:    }
36:
37:    // Destructor
38:    ~MyString()
39:    {
40:       if (Buffer != NULL)
41:          delete [] Buffer;
42:    }
43:
44:    int GetLength() const
45:    {
46:       return strlen(Buffer);
47:    }
```

```
48:
49:    operator const char*()
50:    {
51:       return Buffer;
52:    }
53: };
54:
55: int main()
56: {
57:    cout << "Type a statement: ";
58:    string strInput;
59:    getline(cin, strInput);
60:
61:    MyString youSaid(strInput.c_str());
62:
63:    cout << "Using operator[] for displaying your input: " << endl;
64:    for(int Index = 0; Index < youSaid.GetLength(); ++Index)
65:       cout << youSaid[Index] << " ";
66:    cout << endl;
67:
68:    cout << "Enter index 0 - " << youSaid.GetLength() - 1 << ": ";
69:    int InIndex = 0;
70:    cin >> InIndex;
71:    cout << "Input character at zero-based position: " << InIndex;
72:    cout << " is: "<< youSaid[InIndex] << endl;
73:
74:    return 0;
75: }
```

▼ 输出：

```
Type a statement: Hey subscript operators[] are fabulous
Using operator[] for displaying your input:
H e y   s u b s c r i p t   o p e r a t o r s [ ]   a r e   f a b u l o u s
Enter index 0 - 37: 2
Input character at zero-based position: 2 is: y
```

▼ 分析：

这个程序很有趣，它接受用户输入的句子，并使用它创建一个 MyString 对象，如第 61 行所示；接下来，在一个 for 循环中，使用下标运算符（[]）和数组语法逐字符地打印该字符串，如第 64～65 行所示。下标运算符（[]）是在第 31～35 行实现的，它首先确保指定的位置没有超出 char*Buffer 末尾，然后返回指定位置处的字符。

警告

实现运算符时，应使用关键字 const，这很重要。在程序清单 12.10 中，将下标运算符（[]）的返回类型声明成了 const char&。即便没有关键字 const，该程序也能通过编译。这里使用它旨在禁止使用下面这样的代码：

```
MyString sayHello("Hello World");
sayHello[2] = 'k';error: operator[] is const
```

通过使用 const，可禁止从外部通过运算符[]直接修改成员 MyString::Buffer。除将返回类型声明为 const 外，还将该运算符的函数类型设置成为 const，这将禁止该运算符修改类的成员属性。一般而言，应尽可能使用 const，以免无意间修改数据，并最大限度地保护类的成员属性。

实现下标运算符时，可在程序清单 12.10 所示版本的基础上进行改进。这个版本只实现了一个下标运算符，它可用于读写动态数组的元素。

然而，也可实现两个下标运算符，其中一个为 const 函数，另一个为非 const 函数：

```
char& operator [] (int nIndex);  // use to write / change Buffer at Index
char& operator [] (int nIndex) const; // used only for accessing char at Index
```

编译器很聪明，能够在读取 MyString 对象时调用 const 函数，而在对 MyString 执行写入操作时调用非 const 函数。因此，如果愿意，可在两个下标函数中实现不同的功能。例如，一个运算符记录对容器的写入操作，而另一个记录对容器的读取操作。还有其他双目运算符可被重定义或重载（如表 12.2 所示），但本章不打算介绍它们。这些运算符的实现与已讨论的运算符类似。

如果其他运算符（如逻辑运算符和按位运算符）有助于改善您编写的类，就应实现它们。显然，诸如 Date 等日历类没有必要实现逻辑运算符，但处理字符串和数字的类可能需要实现它们。

应根据类的目标和用途重载运算符或实现新的运算符。

12.4 函数运算符 operator()

operator()让对象像函数，被称为函数运算符。函数运算符用于标准模板库（STL）中，通常是 STL 算法中。其用途包括决策。根据使用的操作数数量，这样的函数对象通常称为单目谓词或双目谓词。下面分析一个非简单的函数对象，如程序清单 12.11 所示，以便理解使用如此有意思的名称的原因！

程序清单 12.11 一个使用 operator()实现的函数对象

```
1: #include <iostream>
2: #include <string>
3:using namespace std;
4:
5: class CDisplay
6: {
7: public:
8:    void operator () (string Input) const
9:    {
10:      cout << Input << endl;
11:   }
12:};
13:
14:int main ()
15:{
16:   CDisplay mDisplayFuncObject;
17:
18:   // equivalent to mDisplayFuncObject.operator () ("Display this string!");
19:   mDisplayFuncObject ("Display this string!");
20:
21:   return 0;
22: }
```

▼ 输出：

```
Display this string!
```

▼ 分析：

第 8～11 行实现了 operator()，然后在 main()函数的第 18 行使用了它。注意，之所以能够在第 18 行将对象 mDisplayFuncObject 用作函数，是因为编译器隐式地将它转换为对函数 operator()的调用。

因此，这个运算符也称为 operator()函数，对象 CDisplay 也称为函数对象或 functor。第 21 章将详尽地讨论这个主题。

C++11

用于高性能编程的移动构造函数和移动赋值运算符

移动构造函数和移动赋值运算符乃性能优化功能，属于 C++11 标准的一部分，旨在避免复制不必

要的临时值（当前语句执行完毕后就不再存在的右值）。对于那些管理动态分配资源的类，如动态数组类或字符串类，这很有用。

1. 不必要的复制带来的问题

请看程序清单 12.5 实现的加法运算符，注意到它创建并返回一个拷贝；减法运算符亦如此。使用下面的语法创建新的 MyString 实例时，情况将如何呢？

```
MyString Hello("Hello ");
MyString World("World");
MyString CPP(" of C++");
MyString sayHello(Hello + World + CPP);  // operator+, copy constructor
MyString sayHelloAgain ("overwrite this");
sayHelloAgain = Hello + World + CPP;  // operator+, copy constructor, copy
assignment operator=
```

这种方式非常直观，它使用双目加法运算符（+）将三个字符串拼接起来。该运算符的实现类似于下面这样：

```
MyString operator+ (const MyString& AddThis)
{
   MyString NewString;

   if (AddThis.Buffer != NULL)
   {
      // copy into NewString
   }
   return NewString;  // return copy by value, invoke copy constructor
}
```

这个加法运算符（+）让您能够使用直观的表达式轻松地拼接字符串，但也可能导致性能问题。创建 sayHello 时，需要执行加法运算符两次，而每次都将创建一个按值返回的临时拷贝，导致执行复制构造函数。复制构造函数执行深复制，而生成的临时拷贝在该表达式执行完毕后就不再存在。总之，该表达式导致生成一些临时拷贝（准确地说是右值），而它们在当前语句执行完毕后就不再需要。这一直是 C++带来的性能瓶颈，直到最近才得以解决。

C++11 解决了这个问题：编译器意识到需要创建临时拷贝时，将转而使用移动构造函数和移动赋值运算符——如果您提供了它们。

2. 声明移动构造函数和移动赋值运算符

移动构造函数的声明语法如下：

```
Class MyClass
{
private:
   Type* PtrResource;

public:
   MyClass(); // default constructor
   MyClass(const MyClass& CopySource); // copy constructor
   MyClass& operator= (const MyClass& CopySource); // copy assignment operator

   MyClass(MyClass&& MoveSource) // Move constructor, note &&
   {
      PtrResource = MoveSource.PtrResource; // take ownership, start move
      MoveSource.PtrResource = NULL;
   }

   MyClass& operator= (MyClass&& MoveSource)//move assignment operator, note &&
   {
      if(this != &MoveSource)
```

```
        {
            delete [] PtrResource;  // free own resource
            PtrResource = MoveSource.PtrResource;  // take ownership, start move
            MoveSource.PtrResource = NULL;  // free move source of ownership
        }
    }
};
```

从上述代码可知，相比于常规赋值构造函数和复制赋值运算符的声明，移动构造函数和移动赋值运算符的不同之处在于，输入参数的类型为 MyClass&&。另外，由于输入参数是要移动的源对象，因此不能使用 const 进行限定，因为它将被修改。返回类型没有变，因为它们分别是构造函数和赋值运算符的重载版本。

在需要创建临时右值时，遵循 C++的编译器将使用移动构造函数（而不是复制构造函数）和移动赋值运算符（而不是复制赋值运算符）。移动构造函数和移动赋值运算符的实现中，只是将资源从源移到目的地，而没有进行复制。程序清单 12.12 演示了如何使用这两项 C++11 新增功能对 MyString 类进行优化。

程序清单 12.12 除复制构造函数和复制赋值运算符外，还包含移动构造函数和移动赋值运算符的 MyString 类

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MyString
 4: {
 5: private:
 6:    char* Buffer;
 7:
 8:    // private default constructor
 9:    MyString(): Buffer(NULL)
10:    {
11:       cout << "Default constructor called" << endl;
12:    }
13:
14: public:
15:    // Destructor
16:    ~MyString()
17:    {
18:       if (Buffer != NULL)
19:          delete [] Buffer;
20:    }
21:
22:    int GetLength()
23:    {
24:       return strlen(Buffer);
25:    }
26:
27:    operator const char*()
28:    {
29:       return Buffer;
30:    }
31:
32:    MyString operator+ (const MyString& AddThis)
33:    {
34:       cout << "operator+ called: " << endl;
35:       MyString NewString;
36:
37:       if (AddThis.Buffer != NULL)
```

```
38:       {
39:          NewString.Buffer = new char[GetLength() + strlen(AddThis.Buffer) +
1];
40:          strcpy(NewString.Buffer, Buffer);
41:          strcat(NewString.Buffer, AddThis.Buffer);
42:       }
43:
44:       return NewString;
45:    }
46:
47:    // constructor
48:    MyString(const char* InitialInput)
49:    {
50:       cout << "Constructor called for: " << InitialInput << endl;
51:       if(InitialInput != NULL)
52:       {
53:          Buffer = new char [strlen(InitialInput) + 1];
54:          strcpy(Buffer, InitialInput);
55:       }
56:       else
57:          Buffer = NULL;
58:    }
59:
60:    // Copy constructor
61:    MyString(const MyString& CopySource)
62:    {
63:       cout<<"Copy constructor to copy from: "<<CopySource.Buffer<<endl;
64:       if(CopySource.Buffer != NULL)
65:       {
66:          // ensure deep copy by first allocating own buffer
67:          Buffer = new char [strlen(CopySource.Buffer) + 1];
68:
69:          // copy from the source into local buffer
70:          strcpy(Buffer, CopySource.Buffer);
71:       }
72:       else
73:          Buffer = NULL;
74:    }
75:
76:    // Copy assignment operator
77:    MyString& operator= (const MyString& CopySource)
78:    {
79:       cout<<"Copy assignment operator to copy from: "<<CopySource.Buffer<< endl;
80:       if ((this != &CopySource) && (CopySource.Buffer != NULL))
81:       {
82:          if (Buffer != NULL)
83:           delete[] Buffer;
84:
85:          // ensure deep copy by first allocating own buffer
86:          Buffer = new char [strlen(CopySource.Buffer) + 1];
87:
88:          // copy from the source into local buffer
89:          strcpy(Buffer, CopySource.Buffer);
90:       }
91:
92:      return *this;
93:    }
94:
95:    // move constructor
96:    MyString(MyString&& MoveSource)
97:    {
98:       cout << "Move constructor to move from: " << MoveSource.Buffer << endl;
```

```
99:       if(MoveSource.Buffer != NULL)
100:       {
101:          Buffer = MoveSource.Buffer; // take ownership i.e.  'move'
102:          MoveSource.Buffer = NULL;   // free move source
103:       }
104:     }
105:
106:    // move assignment operator
107:    MyString& operator= (MyString&& MoveSource)
108:    {
109:      cout<<"Move assignment operator to move from: "<<MoveSource.Buffer<<endl;
110:      if((MoveSource.Buffer != NULL) && (this != &MoveSource))
111:      {
112:         delete Buffer; // release own buffer
113:
114:         Buffer = MoveSource.Buffer; // take ownership i.e.  'move'
115:         MoveSource.Buffer = NULL;   // free move source
116:      }
117:
118:      return *this;
119:   }
120: };
121:
122: int main()
123: {
124:    MyString Hello("Hello ");
125:    MyString World("World");
126:    MyString CPP(" of C++");
127:
128:    MyString sayHelloAgain ("overwrite this");
129:    sayHelloAgain = Hello + World + CPP;
130:
131:    return 0;
132: }
```

▼ 输出：

没有移动构造函数和移动赋值构造函数（将第95～119行注释掉）时的输出：

```
Constructor called for: Hello
Constructor called for: World
Constructor called for:  of C++
Constructor called for: overwrite this
operator+ called:
Default constructor called
Copy constructor to copy from: Hello World
operator+ called:
Default constructor called
Copy constructor to copy from: Hello World of C++
Copy assignment operator to copy from: Hello World of C++
```

添加移动构造函数和移动赋值构造函数后的输出：

```
Constructor called for: Hello
Constructor called for: World
Constructor called for:  of C++
Constructor called for: overwrite this
operator+ called:
Default constructor called
Move constructor to move from: Hello World
operator+ called:
Default constructor called
Move constructor to move from: Hello World of C++
Move assignment operator to move from: Hello World of C++
```

▼ **分析：**

这个代码示例很长，但大部分都在本书前面介绍过。在该程序清单中，最重要的部分是第 95～119 行，其中实现了移动构造函数和移动赋值运算符。这些 C++11 新增功能生成的输出使用粗体表示。注意到相比于没有这两个实体时，输出变化很大。如果您查看移动构造函数和移动赋值运算符的实现，将发现移动语义基本上是通过接管移动源中资源的所有权实现的，如移动构造函数的第 101 行和移动赋值运算符的第 114 行所示。接下来，将移动源指针设置为 NULL，如第 102 和 115 行所示。这样，移动源被销毁时，通过析构函数（第 16～20 行）调用的 delete 什么也不会做，因为所有权已转交给目标对象。注意到在没有移动构造函数时，将调用复制构造函数，它对指向的字符串进行深复制。总之，移动构造函数避免了不必要的内存分配和复制步骤，从而节省了大量的处理时间。

移动构造函数和移动赋值运算符是可选的。不同于复制构造函数和复制赋值运算符，如果您没有提供移动构造函数和移动赋值运算符，编译器并不会添加默认实现。

对于管理动态分配资源的类，可使用 C++11 新增的这项功能对其进行优化，避免在只需临时拷贝的情况下进行深复制。

12.5 不能重载的运算符

虽然 C++提供了很大的灵活性，让程序员能够自定义运算符的行为，让类更易于使用，但 C++也有所保留，不允许程序员改变有些运算符的行为。表 12.3 列出了不能重新定义的运算符。

表 12.3 不能重载或重新定义的运算符

运算符	名称	运算符	名称
.	成员选择	?:	条件三目运算符
.*	指针成员选择	sizeof	获取对象/类类型的大小
::	作用域解析		

应该	不应该
务必实现让类易于使用的运算符，但不要实现无助于实现这个目的的运算符。	别忘了，如果您没有提供复制赋值运算符和复制构造函数，编译器将提供其默认版本，而这些版本不一定会对包含的原始指针进行深复制。
对于包含原始指针成员的类，除给它提供复制构造函数和析构函数外，务必给它提供复制赋值运算符。	别忘了，即便您没有提供移动赋值运算符和移动构造函数，编译器也不会替您创建它们，而是退而求其次，转而使用复制赋值运算符和复制构造函数。
如果使用的是遵循 C++11 的编译器，则对于管理动态分配资源（如数组）的类，务必给它提供移动赋值运算符和移动构造函数。	别忘了，是否实现运算符完全由您决定，但实现它们有助于改善类的可用性。
	别忘了，对于您自己编写的智能指针类，除非实现了运算符*和->，否则它就不是指针。另外，除非您实现析构函数，并对复制赋值和复制构造的情形深思熟虑，否则它就不够智能。

12.6 总结

本章介绍了如何各种运算符，让类更易于使用。编写管理资源（如动态数组或字符串）的类时，

除析构函数外，还需至少提供复制构造函数和复制赋值运算符。对于管理动态数组的实用类，如果包含移动构造函数和移动赋值运算符，就可避免将包含的资源深复制给临时对象。最后，您学习了.、.*、::、?:和 sizeof 等不能重新定义的运算符。

12.7 问与答

问：我编写的类封装了一个动态整型数组，请问我至少应该实现哪些函数和方法？

答：编写这样的类时，必须明确定义下述情形下的行为：通过赋值直接复制对象或通过按值传递给函数间接复制对象。通常，应实现复制构造函数、复制赋值运算符和析构函数。另外，如果想改善这个类在某些情况下的性能，还应实现移动构造函数和移动赋值运算符。

问：假设有一个类对象 object，而我希望支持语法 cout << object;，请问需要实现哪个运算符？

答：您需要实现一个转换运算符，让类能被解读为 std::cout 支持的类型。一种解决方案是，像程序清单 12.2 那样定义运算符 char* ()。

问：自己编写智能指针类时，至少需要实现哪些函数和运算符？

答：智能指针必须能够像常规指针那样使用，如*pSmartPtr 或 pSmartPtr->Func()。为此，需要实现运算符*和->。要确保它足够智能，还需合理地编写析构函数，以自动释放/归还资源；另外，还需实现复制构造函数和复制赋值运算符，以明确定义复制和赋值的方式（也可将复制构造函数和复制赋值运算符声明为私有的，以禁止复制和赋值）。

12.8 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

12.8.1 测验

1. 可以像下面这样，编写两个版本的下标运算符，一个的返回类型为 const，另一个为非 const 吗？

```
const Type& operator[](int Index);
Type& operator[](int Index); // is this OK?
```

2. 可以将复制构造函数或复制赋值运算符声明为私有的吗？
3. 给 Date 类实现移动构造函数和移动赋值运算符有意义吗？

12.8.2 练习

1. 为 Date 类编写一个转换运算符，将其存储的日期转换为整数。
2. DynIntegers 类以 int*私有成员的方式封装了一个动态分配的数组，请给它编写移动构造函数和移动赋值运算符。

第 13 章
类型转换运算符

类型转换是一种机制，让程序员能够暂时或永久性改变编译器对对象的解释。注意，这并不意味着程序员改变了对象本身，而只是改变了对对象的解释。可改变对象解释方式的运算符称为类型转换运算符。

在本章中，您将学习：

- 为何需要类型转换运算符；
- 为什么有些 C++程序员不喜欢传统的 C 风格类型转换；
- 4 个 C++类型转换运算符；
- 向上转换和向下转换；
- 为什么 C++类型转换运算符并非总是最佳选择。

13.1 为何需要类型转换

如果 C++应用程序都编写得很完善，其处于类型是安全的且是强类型的世界，则没有必要进行类型转换，也不需要类型转换运算符。然而，在现实世界中，不同模块往往由使用不同环境的个人和厂商编写，他们需要相互协作。因此，程序员经常需要让编译器按其所需的方式解释数据，让应用程序能够成功编译并正确执行。

来看一个真实的例子：虽然 C++编译器支持 bool，但是很多年前使用 C 语言编写的库仍在使用。这些针对 C 语言编译器编写的库必须依赖于整型来保存布尔值，因此对这些编译器来说，bool 类型类似于下面这样：

```
typedef unsigned short BOOL;
```

而返回布尔值的函数可能这样声明：

```
BOOL IsX ();
```

如果要在新应用程序中使用一个这样的库，而该应用程序将使用最新的 C++编译器进行编译，则程序员必须让其使用的 C++编译器能够理解数据类型 bool，同时让库能够理解数据类型 bool。为此，可使用类型转换：

```
bool bCPPResult = (bool)IsX ();     // C-Style cast
```

在 C++的发展过程中，不断有新的 C++类型转换运算符出现，这导致 C++编程社区分裂成两个阵营：一个阵营继续在其 C++应用程序中使用 C 风格类型转换，另一个阵营转而使用 C++编译器引入的类型转换关键字。前一个阵营认为，C++类型转换难以使用，且有时候功能变化不大，只有理论意义。后一个阵营则显然由 C++语法纯粹论者组成，他们通过指出 C 风格类型转换的缺陷以支持其论点。在现实世界中，这两个观点各行其道，读者最好通过阅读本章了解每种风格的优缺点，然后形成自己的见解。

13.2 为何有些C++程序员不喜欢C风格类型转换

C++程序员在赞颂这门编程语言时提到的优点之一是类型安全。实际上，大多数C++编译器都不会让下面这样的语句通过编译：

```
char* pszString = "Hello World!";
int* pBuf = pszString;    // error: cannot convert char* to int*
```

这是非常正确的！

当前，C++编译器仍需向后兼容，以确保遗留代码能够通过编译，因此支持下面这样的语法：

```
int* pBuf = (int*)pszString; // Cast one problem away, create another
```

然而，C风格类型转换实际上强迫编译器根据程序员的选择来解释目标对象。就上述代码而言，程序员并不认为编译器报告错误是合理的，因此强迫编译器遵从自己的意愿。然而，对不希望类型转换破坏其倡导的类型安全的C++程序员来说，这是无法接受的。

13.3 C++类型转换运算符

虽然类型转换有缺点，但也不能抛弃类型转换的概念。在很多情况下，类型转换是合理的需求，可解决重要的兼容性问题。C++提供了一种新的类型转换运算符，专门用于基于继承的情形，这种情形在C语言编程中并不存在。

4个C++类型转换运算符如下：

- static_cast
- dynamic_cast
- reinterpret_cast
- const_cast

这4个类型转换运算符的使用语法相同：

```
destination_type result = cast_type <destination_type> (object_to_be_casted);
```

13.3.1 使用static_cast

static_cast用于在相关类型的指针之间进行转换，还可显式地执行标准数据类型的类型转换——这种转换原本将自动或隐式地进行。用于指针时，static_cast实现了基本的编译阶段检查，确保指针被转换为相关类型。这改进了C风格类型转换，在C语言中，可将指向一个对象的指针转换为完全不相关的类型，而编译器不会报错。使用static_cast可将指针向上转换为基类类型，也可向下转换为派生类型，如下面的示例代码所示：

```
Base* pBase = new Derived ();    // construct a Derived object
Derived* pDerived = static_cast<Derived*>(pBase);    // ok!

// CUnrelated is not related to Base via any inheritance heirarchy
CUnrelated* pUnrelated = static_cast<CUnrelated*>(pBase); // Error
//The cast above is not permitted as types are unrelated
```

注意

将Derived*转换为Base*被称为向上转换，无需使用任何显式类型转换运算符就能进行这种转换：

```
Derived objDerived;
Base* pBase = &objDerived;  // ok!
```

将Base*转换为Derived*被称为向下转换，如果不使用显式类型转换运算符，就无法进

```
行这种转换:
    Derived objDerived;
    Base* pBase = &objDerived; // Upcast -> ok!
    Derived* pDerived = pBase; // Error: Downcast needs explicit cast
```

然而，static_cast 只验证指针类型是否相关，而不会执行任何运行阶段检查。因此，程序员可使用 static_cast 编写如下代码，而编译器不会报错：

```
Base* pBase = new Base ();
Derived* pDerived = static_cast<Derived*>(pBase); // Still no errors!
```

其中 pDerived 实际上指向一个不完整的 Derived 对象，因为它指向的对象实际上是 Base()类型。由于 static_cast 只在编译阶段检查转换类型是否相关，而不执行运行阶段检查，因此 pDerived->SomeDerivedClassFunction()能够通过编译，但在运行阶段可能导致意外结果。

除用于向上转换和向下转换外，static_cast 还可在很多情况下将隐式类型转换为显式类型，以引起程序员或代码阅读人员的注意：

```
double dPi = 3.14159265;
int Num = static_cast<int>(dPi); // Making an otherwise implicit cast, explicit
```

在上述代码中，使用 Num = dPi 将获得同样的效果，但使用 static_cast 可让代码阅读者注意到这里使用了类型转换，并指出（对知道 static_cast 的人而言）编译器根据编译阶段可用的信息进行了必要的调整，以便执行所需的类型转换。

13.3.2 使用 dynamic_cast 和运行阶段类型识别

顾名思义，与静态类型转换相反，动态类型转换在运行阶段（即应用程序运行时）执行类型转换。可检查 dynamic_cast 操作的结果，以判断类型转换是否成功。使用 dynamic_cast 运算符的典型语法如下：

```
destination_type* pDest = dynamic_cast <class_type*> (pSource);

if (pDest)    // Check for success of the casting operation before using pointer
    pDest->CallFunc ();
```

例如：

```
Base* pBase = new Derived();

// Perform a downcast
Derived* pDerived = dynamic_cast <Derived*> (pBase);

if (pDerived)    // Check for success of the cast
    pDerived->CallDerivedClassFunction ();
```

如上述代码所示，给定一个指向基类对象的指针，程序员可使用 dynamic_cast 进行类型转换，并在使用指针前检查指针指向的目标对象的类型。在上述示例代码中，目标对象的类型显然是 Derived，因此这些代码只有演示价值。然而，情况并非总是如此，例如，将 Derived*传递给接受 Base*参数的函数时。该函数可使用使用 dynamic_cast 判断基类指针指向的对象的类型，再执行该类型特有的操作。总之，可使用 dynamic_cast 在运行阶段判断类型，并在安全时使用转换后的指针。程序清单 13.1 使用了一个您熟悉的继承层次结构 —— Tuna 和 Carp 类从基类 Fish 派生而来，其中的函数 DetectFishtype()动态的检查 Fish 指针指向的对象是否是 Tuna 或 Carp。

注意

这种在运行阶段识别对象类型的机制称为运行阶段类型识别（runtime type identification，RTTI）。

程序清单 13.1 使用动态转换判断 Fish 指针指向的是否是 Tuna 对象或 Carp 对象

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    virtual void Swim()
 7:    {
 8:       cout << "Fish swims in water" << endl;
 9:    }
10:
11:    // base class should always have virtual destructor
12:    virtual ~Fish() {}
13: };
14:
15: class Tuna: public Fish
16: {
17: public:
18:    void Swim()
19:    {
20:       cout << "Tuna swims real fast in the sea" << endl;
21:    }
22:
23:    void BecomeDinner()
24:    {
25:       cout << "Tuna became dinner in Sushi" << endl;
26:    }
27: };
28:
29: class Carp: public Fish
30: {
31: public:
32:    void Swim()
33:    {
34:       cout << "Carp swims real slow in the lake" << endl;
35:    }
36:
37:    void Talk()
38:    {
39:       cout << "Carp talked crap" << endl;
40:    }
41: };
42:
43: void DetectFishType(Fish* InputFish)
44: {
45:    Tuna* pIsTuna = dynamic_cast <Tuna*>(InputFish);
46:    if (pIsTuna)
47:    {
48:       cout << "Detected Tuna. Making Tuna dinner: " << endl;
49:       pIsTuna->BecomeDinner();   // calling Tuna::BecomeDinner
50:    }
51:
52:    Carp* pIsCarp = dynamic_cast <Carp*>(InputFish);
53:    if(pIsCarp)
54:    {
55:       cout << "Detected Carp. Making carp talk: " << endl;
56:       pIsCarp->Talk();  // calling Carp::Talk
57:    }
58:
```

```
59:    cout << "Verifying type using virtual Fish::Swim: " << endl;
60:    InputFish->Swim(); // calling virtual function Swim
61: }
62:
63: int main()
64: {
65:    Carp myLunch;
66:    Tuna myDinner;
67:
68:    DetectFishType(&myDinner);
69:    cout << endl;
70:    DetectFishType(&myLunch);
71:
72:    return 0;
73:}
```

▼ **输出：**

```
Detected Tuna. Making Tuna dinner:
Tuna became dinner in Sushi
Verifying type using virtual Fish::Swim:
Tuna swims real fast in the sea

Detected Carp. Making carp talk:
Carp talked crap
Verifying type using virtual Fish::Swim:
Carp swims real slow in the lake
```

▼ **分析：**

这是一个您在第 10 章熟悉的继承层次结构：Tuna 和 Carp 从基类 Fish 派生而来。为方便解释，这两个派生类不仅实现了虚函数 Swim()，还分别包含一个特有的函数，即 Tuna::BecomeDinner()和 Carp::Talk()。这个示例的独特之处在于，给定一个基类指针（Fish*），您可动态地检测它指向的是否是 Tuna 或 Carp。这种动态检测（运行阶段类型识别）是在第 43～61 行定义的函数 DetectFishType()中进行的。在第 45 行，使用 dynamic_cast 传入的基类指针（Fish*）参数指向的是否是 Tuna 对象。如果该 Fish*指向的是 Tuna 对象，该运算符将返回一个有效的地址，否则将返回 NULL。因此，总是需要检查 dynamic_cast 的结果是否有效。如果通过了第 46 行的检查，您便知道指针 pIsTuna 指向的是一个有效的 Tuna 对象，因此可以使用它来调用函数 Tuna::BecomeDinner()，如第 49 行所示。如果传入的 Fish*参数指向的是 Carp 对象，则使用它来调用函数 Carp::Talk()，如第 56 行所示。返回之前，DetectFishType()调用了 Swim()，以验证对象类型；Swim()是一个虚函数，这行代码将根据指针指向的对象类型，调用相应类（Tuna 或 Carp）中实现的方法 Swim()。

警告 务必检查 dynamic_cast 的返回值，看它是否有效。如果返回值为 NULL，说明转换失败。

13.3.3 使用 reinterpret_cast

reinterpret_cast 是 C++中与 C 风格类型转换最接近的类型转换运算符。它让程序员能够将一种对象类型转换为另一种，不管它们是否相关；也就是说，它使用如下所示的语法强制重新解释类型：

```
Base * pBase = new Base ();
CUnrelated * pUnrelated = reinterpret_cast<CUnrelated*>(pBase);
// The code above was not good programming, even when it compiles!
```

这种类型转换实际上是强制编译器接受 static_cast 通常不允许的类型转换，通常用于低级程序（如驱动程序），在这种程序中，需要将数据转换为 API 能够接受的简单类型（例如，有些 API 只能使用字节流，即 unsigned char*）：

```
SomeClass* pObject = new SomeClass ();
// Need to send the object as a byte-stream...
unsigned char* pBytes = reinterpret_cast <unsigned char*>(pObject);
```

上述代码使用的类型转换并没有改变源对象的二进制表示，但让编译器允许程序员访问 SomeClass 对象包含的各个字节。由于其他 C++类型转换运算符都不允许执行这样的转换，因此使用 reinterpret_cast 时，程序员将收到类型转换不安全（不可移植）的警告。

警告

> 应尽量避免在应用程序中使用 reinterpret_cast，因为它让编译器将类型 X 视为不相关的类型 Y，这看起来不像是优秀的设计或实现。

13.3.4 使用 const_cast

const_cast 让程序员能够关闭对象的访问修饰符 const。您可能会问：为何要进行这种转换？在理想情况下，程序员将经常在正确的地方使用关键字 const。不幸的是，现实世界并非如此，像下面这样的代码随处可见：

```
class SomeClass
{
public:
    // ...
    void DisplayMembers ();    // a display function ought to be const
};
```

在下面的函数中，以 const 引用的方式传递 mData 对象显然是正确的。毕竟，显示函数应该是只读的，不应调用非 const 成员函数，即不应调用能够修改对象状态的函数。然而，DisplayMembers()本应为 const 的，但却没有这样定义。如果 SomeClass 归您所有，且源代码受您控制，则可对 DisplayMembers()进行修改。然而，在很多情况下，它可能属于第三方库，无法对其进行修改。在这种情况下，const_cast 将是您的救星。

```
void DisplayAllData (const SomeClass& mData)
{
    mData.DisplayMembers ();  // Compile failure
    // reason for failure: call to a non-const member using a const reference
}
```

在这种情况下，调用 DisplayMembers()的语法如下：

```
void DisplayAllData (const SomeClass& mData)
{
    SomeClass& refData = const_cast <SomeClass&>(mData);
    refData.DisplayMembers();    // Allowed!
}
```

除非万不得已，否则不要使用 const_cast 来调用非 const 函数。一般而言，使用 const_cast 来修改 const 对象可能导致不可预料的行为。

另外，const_cast 也可用于指针：

```
void DisplayAllData (const SomeClass* pData)
{
    // pData->DisplayMembers(); Error: attempt to invoke a non-const function!
    SomeClass* pCastedData = const_cast <SomeClass*>(pData);
    pCastedData->DisplayMembers();    // Allowed!
}
```

13.4 C++类型转换运算符存在的问题

并非所有人都喜欢使用 C++类型转换，即使那些 C++拥趸也如此。其理由很多，从语法繁琐而不

够直观到显得多余，不一而足。

来比较一下下面的代码：

```
double dPi = 3.14159265;

// C++ style cast: static_cast
int Num = static_cast <int>(dPi);     // result: nNum is 3

// C-style cast
int Num2 = (int)dPi;              // result: Num2 is 3

// leave casting to the compiler
int Num3 = dPi;              // result: Num3 is 3. No errors!
```

在这 3 种方法中，程序员得到的结果都相同。在实际情况下，第 2 种方法可能最常见，其次是第 3 种，但几乎没有人使用第 1 种方法。无论采用哪种方法，编译器都足够聪明，能够正确地进行类型转换。这让人觉得类型转换运算符将降低代码的可读性。

同样，static_cast 的其他用途也可使用 C 风格类型转换进行处理，且更简单：

```
// using static_cast
Derived* pDerived = static_cast <Derived*>(pBase);
// But, this works just as well...
Derived* pDerivedSimple = (Derived*)pBase;
```

因此，使用 static_cast 的优点常常被其拙劣的语法所掩盖。Bjarne Stroustrup 准确地描述了这种境况："由于 static_cast 如此拙劣且难以输入，因此您在使用它之前很可能会三思。这很不错，因为类型转换在现代 C++中是最容易避免的。"

再来看其他运算符。在不能使用 static_cast 时，可使用 reinterpret_cast 强制进行转换；同样，可以使用 const_cast 修改访问修饰符 const。因此，在现代 C++中，除 dynamic_cast 外的类型转换都是可以避免的。仅当需要满足遗留应用程序的需求时，才需要使用其他类型转换运算符。在这种情况下，程序员通常倾向于使用 C 风格类型转换而不是 C++类型转换运算符。重要的是，应尽量避免使用类型转换；而一旦使用类型转换，务必要知道幕后发生的情况。

应该	不应该
请牢记，将 Derived*转换为 Base*被称为向上转换；这种转换是安全的。 请牢记，将 Base*转换为 Derived*被称为向下转换；除非使用 dynamic_cast，否则这种转换不安全。 请牢记，创建继承层次结构时，应尽量将函数声明为虚函数。这样通过基类指针调用这些函数时，如果该指针指向的是派生类对象，将调用相应类的函数版本。	使用 dynamic_cast 时，别忘了对转换得到的指针进行检查，看其是否有效。 设计应用程序时，不要使用 dynamic_cast 来依赖于 RTTI。

13.5 总结

本章介绍了各种 C++类型转换运算符以及支持和反对类型转换运算符的根据。一般而言，应避免使用类型转换。

13.6 问与答

问：是否可使用 const_cast 对指向常量对象的指针或引用进行类型转换，以便修改常量对象的

内容？

答：不要这样做。这样做的结果是不确定的，也绝不是您希望的。

问：我需要一个 Bird*，但只有一个 Dog*。编译器不允许将指向 Dog 对象的指针用作 Bird*。然而，当我使用 reinterpret_cast 将 Dog*转换为 Bird*时，编译器并不报错。看起来可使用这个指针来调用 Bird 的成员函数 Fly()，可以这样做吗？

答：绝对不要这样做。reinterpret_cast 只改变对指针的解释，并不改变指向的对象（它还是 Dog）。对 Dog 对象调用 Fly()函数将得不到所需的结果，还可能导致应用程序出现故障。

问：我有一个 Base 指针 pBase，它指向一个 Derived 对象。我确信 pBase 指向的是一个 Derived 对象，是否还需要使用 dynamic_cast？

答：由于您确定指向的是 Derived 对象，因此可使用 static_cast 提高运行性能。

问：C++提供了类型转换运算符，但却建议尽量不使用它们。这是为什么？

答：您家里备有阿司匹林，却不会把它当饭吃。仅当真正需要时才使用类型转换。

13.7 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

13.7.1 测验

1. 您有一个基类对象指针 pBase，要确定它指向的是否是 Derived1 或 Derived2 对象，应使用哪种类型转换？

2. 假设您有一个指向对象的 const 引用，并试图通过它调用一个您编写的公有成员函数，但编译器不允许您这样做，因为该函数不是 const 成员。您将修改这个函数还是使用 const_cast？

3. 判断对错：仅在不能使用 static_cast 时才应使用 reinterpret_cast，这种类型转换是必须和安全的。

4. 判断对错：优秀的编译器将自动执行很多基于 static_cast 的类型转换，尤其是简单数据类型之间的转换。

13.7.2 练习

1. 查错：下述代码有何问题？

```
void DoSomething(Base* pBase)
{
   Derived* pDerived = dynamic_cast <Derived*>(pBase);
   pDerived->DerivedClassMethod();
}
```

2. 假设有一个 Fish 指针（pFish），它指向一个 Tuna 对象：

```
Fish* pFish = new Tuna;
Tuna* pTuna = <what cast?>pFish;
```

要让一个 Tuna 指针指向该指针指向的 Tuna 对象，应使用哪种类型转换？请使用代码证明您的看法。

第 14 章
宏和模板简介

现在，读者应对基本的 C++语法有深入认识，能够理解使用 C++编写的程序，为学习有助于高效编写程序的语言特性做好了准备。

在本章中，您将学习：

- 预处理器简介；
- 关键字#define 与宏；
- 模板简介；
- 如何编写函数模板和模板类；
- 宏和模板之间的区别；
- 使用 C++11 新增的 static_assert 进行编译阶段检查。

14.1 预处理器与编译器

在第 2 章，您首次接触到了预处理器。顾名思义，预处理器在编译器之前运行，换句话说，预处理器根据程序员的指示，决定实际要编译的内容。预处理器编译指令都以#打头，例如：

```
// tell the preprocessor to insert the contents of header iostream here
#include <iostream>

// define a macro constant
#define ARRAY_LENGTH 25
int MyNumbers[ARRAY_LENGTH]; // array of 25 integers

// define a macro function
#define SQUARE(x) ((x) * (x))
int TwentyFive = SQUARE(5);
```

本章重点介绍上述代码演示的两种预处理器编译指令，一是使用#define 定义常量，二是使用#define 定义宏函数。这两个编译指令都告诉编译器，将每个宏实例（ARRAY_LENGTH 或 SQUARE）替换为其定义的值。

注意 宏也进行文本替换。预处理器只是就地将标识符替换为指定的文本。

14.2 使用#define 定义常量

使用#define 定义常量的语法非常简单：

```
#define identifier value
```

例如，要定义将被替换为25的常量ARRAY_LENGTH，可使用如下代码：

```
#define ARRAY_LENGTH 25
```

这样，每当预处理器遇到标识符ARRAY_LENGTH时，都会将其替换为25。

```
int MyNumbers [ARRAY_LENGTH] = {0};
double Radiuses [ARRAY_LENGTH] = {0.0};
std::string Names [ARRAY_LENGTH];
```

对于上述三行代码，预处理器运行完毕后，编译器看到的代码如下：

```
int MyNumbers [25] = {0};  // an array of 25 integers
double Radiuses [25] = {0.0};  // an array of 25 doubles
std::string Names [25];  // an array of 25 std::strings
```

替换将在所有代码中进行，包括下面这样的for循环：

```
for(int Index = 0; Index < ARRAY_LENGTH; ++Index)
   MyNumbers[Index] = Index;
```

编译器看到的上述循环如下：

```
for(int Index = 0; Index < 25; ++Index)
   MyNumbers[Index] = Index;
```

要准确地了解#define宏的工作原理，请参阅程序清单14.1。

程序清单14.1 声明并使用定义常量的宏

```
 0: #include <iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: #define ARRAY_LENGTH 25
 5: #define PI 3.1416
 6: #define MY_DOUBLE double
 7: #define FAV_WHISKY "Jack Daniels"
 8:
 9: int main()
10: {
11:    int MyNumbers [ARRAY_LENGTH] = {0};
12:    cout << "Array's length: " << sizeof(MyNumbers) / sizeof(int) << endl;
13:
14:    cout << "Enter a radius: ";
15:    MY_DOUBLE Radius = 0;
16:    cin >> Radius;
17:    cout << "Area is: " << PI * Radius * Radius << endl;
18:
19:    string FavoriteWhisky (FAV_WHISKY);
20:    cout << "My favorite drink is: " << FAV_WHISKY << endl;
21:
22:    return 0;
23: }
```

▼ 输出：

```
Array's length: 25
Enter a radius: 2.1569
Area is: 14.7154
My favorite drink is: Jack Daniels
```

▼ 分析：

第3～7行定义了4个宏常量：ARRAY_LENGTH、PI、MY_DOUBLE和FAV_WHISKY。正如您看到的，第11行使用了常量ARRAY_LENGTH来指定数组的长度，而第12行使用了运算符sizeof()间接地核实数组长度。第15行使用MY_DOUBLE声明了类型为double的变量Radius，而

第 17 行使用了 PI 来计算圆的面积。最后，第 19 行使用了 FAV_WHISKY 来初始化一个 std::string 对象，而第 20 行在 cout 语句中直接使用了该常量。所有这些语句都表明，预处理器只是进行文本替换。

程序清单 14.1 大量地使用了“死板的”文本替换，但这种文本替换也有缺点。

提示

预处理器进行死板的文本替换，这减轻了您的负担，但并非总能减轻编译器的负担。在程序清单 14.1 的第 7 行中，如果您这样定义 FAV_WHISKY:

```
#define FAV_WHISKY 42 // "Jack Daniels"
```

则第 19 行实例化 std::string 的代码将导致编译错误。但如果没有这行代码，该程序将通过编译，并打印如下内容:

```
My favorite drink is: 42
```

这样的输出显然不符合逻辑，而最重要的是，编译器却没有检测到这一点。另外，对于使用宏定义的常量 PI，您没有太大的控制权：其类型是 double 还是 float？答案是都不是。在预处理器看来，PI 就是 3.1416，根本不知道其数据类型。

定义常量时，更好的选择是使用关键字 const 和数据类型，因此下面的定义好得多:

```
const int ARRAY_LENGTH = 25;
const double PI = 3.1416;
const char* FAV_WHISKY = "Jack Daniels";
typedef double MY_DOUBLE;  // use typedef to alias a type
```

14.2.1 使用宏避免多次包含

C++程序员通常在.h 文件（头文件）中声明类和函数，并在.cpp 文件中定义函数，因此需要在.cpp 文件中使用预处理器编译指令#include <header>来包含头文件。如果在头文件 class1.h 中声明了一个类，而这个类将 class2.h 中声明的类作为其成员，则需要在 class1.h 中包含 class2.h。如果设计非常复杂，即第二个类需要第一个类，则在 class2.h 中也需要包含 class1.h!

然而，在预处理器看来，两个头文件彼此包含对方会导致递归问题。为避免这种问题，可结合使用宏以及预处理器编译指令#ifndef 和#endif。

包含<header2.h>的 head1.h 类似于下面这样：

```
#ifndef HEADER1_H _//multiple inclusion guard:
#define HEADER1_H_ // preprocessor will read this and following lines once
#include <header2.h>

class Class1
{
   // class members
};
#endif  // end of header1.h
```

header2.h 与此类似，但宏定义不同，且包含的是<header1.h>：

```
#ifndef HEADER2_H_//multiple inclusion guard
#define HEADER2_H_
#include <header1.h>

class Class2
{
   // class members
};
#endif // end of header2.h
```

注意 #ifndef 可读作 if-not-defined。这是一个条件处理命令，让预处理器仅在标识符未定义时才继续。#endif 告诉预处理器，条件处理指令到此结束。

因此，预处理器首次处理 header1.h 并遇到#ifndef 后，发现宏 HEADER1_H_还未定义，因此继续处理。#ifndef 后面的第一行定义了宏 HEADER1_H_，确保预处理器再次处理该文件时，将在遇到包含#ifndef 的第一行时结束，因为其中的条件为 false。header2.h 与此类似。在 C++编程领域，这种简单的机制无疑是最常用的宏功能之一。

14.3 使用#define 编写宏函数

预处理器对宏指定的文本进行简单替换，因此也可以使用宏来编写简单的函数，例如：

```
#define SQUARE(x) ((x) * (x))
```

这个宏计算平方值。同样，计算圆面积的宏类似于下面这样：

```
#define PI 3.1416
#define AREA_CIRCLE(r) (PI*(r)*(r))
```

宏函数通常用于执行非常简单的计算。相比于常规函数调用，宏函数的优点在于，它们将在编译前就地展开，因此在有些情况下有助于改善代码的性能。程序清单 14.2 演示了如何使用这些宏函数。

程序清单 14.2 使用计算平方值、圆面积、最小值和最大值的宏函数

```
 0: #include <iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: #define SQUARE(x) ((x) * (x))
 5: #define PI 3.1416
 6: #define AREA_CIRCLE(r) (PI*(r)*(r))
 7: #define MAX(a, b) (((a) > (b)) ? (a) : (b))
 8: #define MIN(a, b) (((a) < (b)) ? (a) : (b))
 9:
10: int main()
11: {
12:    cout << "Enter an integer: ";
13:    int Input1 = 0;
14:    cin >> Input1;
15:
16:    cout << "SQUARE(" << Input1 << ") = " << SQUARE(Input1) << endl;
17:    cout << "Area of a circle with radius " << Input1 << " is: ";
18:    cout << AREA_CIRCLE(Input1) << endl;
19:
20:    cout << "Enter another integer: ";
21:    int Input2 = 0;
22:    cin >> Input2;
23:
24:    cout << "MIN(" << Input1 << ", " << Input2 << ") = ";
25:    cout << MIN (Input1, Input2) << endl;
26:
27:    cout << "MAX(" << Input1 << ", " << Input2 << ") = ";
28:    cout << MAX (Input1, Input2) << endl;
29:
30:    return 0;
31: }
```

▼ 输出：

```
Enter an integer: 36
SQUARE(36) = 1296
Area of a circle with radius 36 is: 4071.51
Enter another integer: -101
MIN(36, -101) = -101
MAX(36, -101) = 36
```

▼ 分析：

第4～8行包含几个宏函数，它们分别计算平方值、圆面积以及两个数中的最大值和最小值。注意到第6行的AREA_CIRCLE使用了宏常量PI来计算圆面积，这表明一个宏可使用另一个宏。毕竟，宏是向预处理器发出的文本替换命令。下面来分析使用MIN宏的第25行：

```
cout << MIN (Input1, Input2) << endl;
```

编译器进行编译时，这行代码变成了下面这样，即将宏就地展开了：

```
cout << (((Input1) < (Input2)) ? (Input1) : (Input2)) << endl;
```

警告

宏不考虑数据类型，因此使用宏函数很危险。例如，理想情况下，AREA_CIRCLE的返回类型应为double，这样可确保返回的圆面积的精度，使其不依赖于半径的精度。

14.3.1 为什么要使用括号

再来看一下计算圆面积的宏：

```
#define AREA_CIRCLE(r) (PI*(r)*(r))
```

上述代码比较古怪，使用了大量的括号。而在程序清单7.1中，函数Area()的代码如下：

```
// Function definitions (implementations)
double Area(double InputRadius)
{
   return Pi * InputRadius * InputRadius;  // look, no brackets?
}
```

编写宏时使用了大量括号，而在函数中，同样的公式看起来完全不同。这是为什么呢？原因在于宏的计算方式——预处理器支持的文本替换机制。

请看下面的宏，它省略了大部分括号：

```
#define AREA_CIRCLE(r) (PI*r*r)
```

如果使用类似于下面的语句调用这个宏，结果将如何呢？

```
cout << AREA_CIRCLE (4+6);
```

展开后，编译器看到的语句如下：

```
cout << (PI*4+6*4+6);  // not the same as PI*10*10
```

根据运算符优先级，将先执行乘法运算，再执行加法运算，因此编译器将这样计算面积：

```
cout << (PI*4+24+6);   // 42.5664 (which is incorrect)
```

在省略了括号的情况下，简单的文本替换破坏了编程逻辑！使用括号有助于避免这种问题：

```
#define AREA_CIRCLE(r) (PI*(r)*(r))
cout << AREA_CIRCLE (4+6);
```

经过替换后，编译器看到的表达式如下：

```
cout << (PI*(4+6)*(4+6));  // PI*10*10, as expected
```

通过使用括号，让宏代码不受运算符优先级的影响，从而能够正确地计算面积。

14.3.2 使用 assert 宏验证表达式

编写程序后，立即单步执行以测试每种路径很不错，但可能不现实。比较现实的做法是，插入检查语句，对表达式或变量的值进行验证。

assert 宏让您能够完成这项任务。要使用 assert 宏，需要包含<assert.h>，其语法如下：

```
assert (expression that evaluates to true or false);
```

下面是一个示例，它使用 assert()来验证指针的值：

```
#include <assert.h>
int main()
{
   char* sayHello = new char [25];
   assert(sayHello != NULL); // throws up a message if pointer is NULL

   // other code

   delete [] sayHello;
   return 0;
}
```

assert()在指针无效时将指出这一点。为演示这一点，我将 sayHello 初始化为 NULL，并在调试模式下执行，Visual Studio 弹出了如图 14.1 所示的窗口。

图 14.1 使用 assert 检查无效指针的结果

在 Microsoft Visual Studio 中，assert()让您能够单击“重试”按钮返回应用程序，而调用栈将指出哪行代码没有通过断言测试。这让 assert()成为一项方便的调试功能；例如，可使用 assert 对函数的输入参数进行验证。长期而言，assert 有助于改善代码的质量，强烈推荐使用它。

注意

在大多数开发环境中，assert()通常在发布模式下被禁用，因此它仅在调试模式下显示错误消息。
另外，在有些开发环境中，assert()被实现为函数，而不是宏。

警告

由于断言在发布模式下不可用，对于对应用程序正确运行至关重要的检查（如检查 dynamic_cast 的返回值），为确保它们在发布模式下也会执行，应使用 if 语句，这很重要。断言可帮助您找出问题，但不能因此不再代码中对指针做必要的检查。

14.3.3 使用宏函数的优点和缺点

宏函数可用于不同的变量类型。再来看一下程序清单 14.2 中的下述代码行：

```
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
```

可将宏函数 MIN 用于整型：

```
cout << MIN(25, 101) << endl;
```

也可将其用于双精度数：

```
cout << MIN(0.1, 0.2) << endl;
```

如果 MIN()为常规函数，必须编写两个不同的版本：MIN_INT()和 MIN_DOUBLE()，前者接受 int 参数并返回一个 int 值，而后者接受 double 参数并返回一个 double 值。使用宏函数减少了代码行，这是一种细微的优势，诱使某些程序员使用宏来定义简单函数。宏函数将在编译前就地展开，因此简单宏的性能由于常规函数调用。这是因为函数调用要求创建调用栈、传递参数等，这些开销占用的 CPU 时间通常比 MIN 执行的计算还多。

虽然具备这些优点，宏也存在严重的问题，那就是不支持任何形式的类型安全。另外，复杂的宏调试起来也不容易。

如果需要编写独立于类型的泛型函数，又要确保类型安全，可使用模板函数，而不是宏函数。如果要改善性能，可将函数声明为内联的。

第 7 章介绍过，要编写内联函数，可使用关键字 inline，如程序清单 7.10 所示。

应该	不应该
尽可能不要自己编写宏函数。 尽可能使用 const 变量，而不是宏常量。 请牢记，宏并非类型安全的，预处理器不执行类型检查。	在宏函数的定义中，别忘了使用括号将每个变量括起。 为在头文件中避免多次包含，别忘了使用#ifndef、#define 和#endif。 别忘了在代码中大量使用 assert()，它们在发行版本中将被禁用，但对提高代码的质量很有帮助。

现在该学习使用模板进行泛型编程了。

14.4 模板简介

模板可能是 C++语言中最强大却最少被使用（或被理解）的特性之一。

在 C++中，模板让程序员能够定义一种适用于不同类型的对象的行为。这听起来有点像宏（参见前面用于判断两个数中哪个更大的简单宏 MAX），但宏不是类型安全的，而模板是类型安全的。

14.4.1 模板声明语法

模板声明以关键字 template 打头，接下来是类型参数列表。这种声明的格式如下：

```
template <parameter list>
template function / class declaration..
```

关键字 template 标志着模板声明的开始，接下来是模板参数列表。该参数列表包含关键字

typename，它定义了模板参数 objectType，objectType 是一个占位符，针对对象实例化模板时，将使用对象的类型替换它。

```
template <typename T1, typename T2 = T1>
bool TemplateFunction(const T1& param1, const T2& param2);

// A template class
template <typename T1, typename T2 = T1>
class Template
{
private:
   T1 m_Obj1;
   T2 m_Obj2;

public:
   T1 GetObj1() {return m_Obj1; }
   // ... other members
};
```

上述代码演示了一个模板函数和一个模板类，它们都接受两个模板参数：T1 和 T2，其中 T2 的类型默认为 T1。

14.4.2 各种类型的模板声明

模板声明可以是：

- 函数的声明或定义；
- 类的定义或声明；
- 类模板的成员函数或成员类的声明或定义；
- 类模板的静态数据成员的定义；
- 嵌套在类模板中的类的静态数据成员的定义；
- 类或类模板的成员模板的定义。

14.4.3 模板函数

假设要编写一个函数，它适用于不同类型的参数，为此可使用模板语法！下面来分析一个模板声明，它与前面讨论的 MAX 宏等价——返回两个参数中较大的一个：

```
template <typename objectType>
const objectType& GetMax (const objectType& value1, const objectType& value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
}
```

下面是一个使用该模板的示例：

```
int Integer1 = 25;
int Integer2 = 40;
int MaxValue = GetMax <int> (Integer1, Integer2);
double Double1 = 1.1;
double Double2 = 1.001;
double MaxValue = GetMax <double> (Double1, Double2);
```

注意到调用 GetMax 时使用了<int>，这将模板参数 objectType 指定为 int。上述代码将导致编译器

生成模板函数 GetMax 的两个版本，如下所示：

```
const int& GetMax (const int& value1, const int& value2)
{
    //...
}
const double & GetMax (const double& value1, const double& value2)
{
    // ...
}
```

然而，实际上调用模板函数时并非一定要指定类型，因此下面的函数调用没有任何问题：

```
int MaxValue = GetMax(Integer1, Integer2);
```

在这种情况下，编译器很聪明，知道这是针对整型调用模板函数，如程序清单 14.3 所示。然而，对于模板类，必须显式地指定类型。

程序清单 14.3 模板函数 GetMax，它返回两个参数中较大的一个

```
 0: #include<iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: template <typename Type>
 5: const Type& GetMax (const Type& value1, const Type& value2)
 6: {
 7:     if (value1 > value2)
 8:         return value1;
 9:     else
10:         return value2;
11: }
12:
13: template <typename Type>
14: void DisplayComparison(const Type& value1, const Type& value2)
15: {
16:    cout << "GetMax(" << value1 << ", " << value2 << ") = ";
17:    cout << GetMax(value1, value2) << endl;
18: }
19:
20: int main()
21: {
22:    int Int1 = -101, Int2 = 2011;
23:    DisplayComparison(Int1, Int2);
24:
25:    double d1 = 3.14, d2 = 3.1416;
26:    DisplayComparison(d1, d2);
27:
28:    string Name1("Jack"), Name2("John");
29:    DisplayComparison(Name1, Name2);
30:
31:    return 0;
32: }
```

▼ 输出：

```
GetMax(-101, 2011) = 2011
GetMax(3.14, 3.1416) = 3.1416
GetMax(Jack, John) = John
```

▼ 分析：

该程序清单包含两个模板函数：第 4～11 行的 GetMax()；第 13～18 行的 DisplayComparison()，它使用了 GetMax()。在 main()函数中，第 23、26 和 29 行表明，可将同一个模板函数用于不同类型的数

据：int、double 和 std::string。模板函数不仅可以重用（就像宏函数一样），而且更容易编写和维护，还是类型安全的。

请注意，调用 DisplayComparison 时，也可显式地指定类型，如下所示：

```
23:   DisplayComparison<int>(Int1, Int2);
```

然而，调用模板函数时没有必要这样做。您无需指定模板参数的类型，因为编译器能够自动推断出类型；但使用模板类时，需要这样做。

14.4.4 模板与类型安全

程序清单 14.3 中的模板函数 DisplayComparison()和 GetMax()是类型安全的，这意味着不能像下面这样进行无意义的调用：

```
DisplayComparison(Integer, "Some string");
```

这种调用将导致编译错误。

14.4.5 模板类

第 9 章介绍过，类是一种编程单元，封装类属性以及使用这些属性的方法。属性通常是私有成员，如 Human 类中的 int Age。类是设计蓝图，其实际表示为对象。例如，可将 Tom 视为 Human 类的一个对象，其 Age 属性为 15。如果对于某些寿命非常长的人，您想使用 long long 变量来存储其年龄，而对于寿命较短的人，则使用 short 变量来存储其年龄，该如何办呢？此时模板类可派生用场。模板类是模板化的 C++类，是蓝图的蓝图。使用模板类时，可指定要为哪种类型具体化类。这让您能够创建不同的 Human 对象，即有的年龄存储在 long long 成员中，有的存储在 int 成员中，还有的存储在 short 成员中。

下面是一个简单的模板类，它只有一个模板参数 T：

```
template <typename T>
class MyFirstTemplateClass
{
public:
   void SetValue  (const T& newValue) { Value = newValue; }
   const T& GetValue() const {return Value;}

private:
   T Value;
};
```

类 CMyFirstTemplate 用于保存一个类型为 T 的变量，该变量的类型是在使用模板时指定的。下面来看该模板类的一种用法：

```
MyFirstTemplateClass <int> HoldInteger;  // Template instantiation
HoldInteger.SetValue(5);
std::cout << "The value stored is: " << HoldInteger.GetValue() << std::endl;
```

这里使用该模板类来存储和检索类型为 int 的对象，即使用 int 类型的模板参数实例化 Template 类。同样，这个类也可以用于处理字符串，其用法类似：

```
MyFirstTemplateClass <char*> HoldString;
HoldString.SetValue("Sample string");
std::cout << "The value stored is: " << HoldString.GetValue() << std::endl;
```

因此，这个类定义了一种模式，并可针对不同的数据类型实现这种模式。下面是一个可定制的 Human 类，可以通过参数 T 指定 Age 的类型：

```
template <typename T>
class CustomizableHuman
{
public:
    void SetAge (const T& newValue) { Age = newValue; }
    const T& GetAge() const {return Age;}

private:
    T Age;   // T is type you choose to customize this template for!
};
```

使用这个模板时，在模板实例化语法中指定类型：

```
CustomizableHuman<int> NormalLifeSpan;  // instantiate for type int
  NormalLifeSpan.SetAge(80);
CustomizableHuman<long long> LongLifeSpan;  // instantiate for type long long
  LongLifeSpan.SetAge(3147483647);
CustomizableHuman<short> ShortLifeSpan; // instantiate for type short
  ShortLifeSpan.SetAge(40);
```

14.4.6 模板的实例化和具体化

对于模板，术语实例化的含义稍有不同。用于类时，实例化通常指的是根据类创建对象。但用于模板时，实例化指的是根据模板声明以及一个或多个参数创建特定的类型。

因此，对于下面的模板声明：

```
template <typename T>
class TemplateClass
{
    T m_member;
};
```

使用该模板时将编写这样的代码：

```
TemplateClass <int> IntTemplate;
```

这种实例化创建的特定类型称为具体化。

14.4.7 声明包含多个参数的模板

模板参数列表包含多个参数，参数之间用逗号分隔。因此，如果要声明一个泛型类用于存储两个类型可能不同的对象，可以使用如下所示的代码（这个模板类包含两个模板参数）：

```
template <typename T1, typename T2>
class HoldsPair
{
private:
    T1 Value1;
    T2 Value2;
public:
    // Constructor that initializes member variables
    HoldsPair (const T1& value1, const T2& value2)
    {
        Value1 = value1;
        Value2 = value2;
    };
    // ... Other function declarations
};
```

在这里，类 HoldsPair 接受两个模板参数，参数名分别为 T1 和 T2。可使用这个类来存储两个类型相同或不同的对象，如下所示：

```
// A template instantiation that pairs an int with a double
HoldsPair <int, double> pairIntDouble (6, 1.99);

// A template instantiation that pairs an int with an int
HoldsPair <int, int> pairIntDouble (6, 500);
```

14.4.8 声明包含默认参数的模板

可以修改前面的 HoldsPair <…>，将模板参数的默认类型指定为 int：

```
template <typename T1=int, typename T2=int>
class HoldsPair
{
    // ... method declarations
};
```

这与给函数指定默认参数值极其类似，只是这里指定的是默认类型。

这样，前述第二种 HoldsPair 用法可以简写为：

```
// A template instantiation that pairs an int with an int (default type)
HoldsPair <> pairIntDouble (6, 500);
```

14.4.9 一个模板示例

下面使用前面讨论的 HoldsPair 模板来进行开发，如程序清单 14.4 所示。

程序清单 14.4 包含两个成员属性的模板类

```
 0: // Declaring default paramter types, first int, second float
 1: template <typename T1=int, typename T2=double>
 2: class HoldsPair
 3: {
 4: private:
 5:    T1 Value1;
 6:    T2 Value2;
 7: public:
 8:    // Constructor that initializes member variables
 9:    HoldsPair (const T1& value1, const T2& value2)
10:    {
11:       Value1 = value1;
12:       Value2 = value2;
13:    };
14:
15:    // Accessor functions
16:    const T1 & GetFirstValue () const
17:    {
18:       return Value1;
19:    };
20:
21:    const T2& GetSecondValue () const
22:    {
23:       return Value2;
24:    };
25: };
26:
27: #include <iostream>
28: using namespace std;
29:
30: int main ()
31: {
32:    // Two instantiations of template HoldsPair -
```

```
33:    HoldsPair <> mIntFloatPair (300, 10.09);
34:    HoldsPair<short,char*>mShortStringPair(25,"Learn templates,love C++");
35:
36:    // Output values contained in the first object...
37:    cout << "The first object contains -" << endl;
38:    cout << "Value 1: " << mIntFloatPair.GetFirstValue () <<  endl;
39:    cout << "Value 2: " << mIntFloatPair.GetSecondValue () << endl;
40:
41:    // Output values contained in the second object...
42:    cout << "The second object contains -" << endl;
43:    cout << "Value 1: " << mShortStringPair.GetFirstValue () <<  endl;
44:    cout << "Value 2: " << mShortStringPair.GetSecondValue () << endl;
45:
46:    return 0;
47: }
```

▼ 输出：

```
The first object contains -
Value 1: 300
Value 2: 10.09
The second object contains -
Value 1: 25
Value 2: Learn templates, love C++
```

▼ 分析：

这个简单程序演示了如何声明模板类 HoldsPair 来存储两个值，这两个值的类型取决于模板的参数列表。第 1 行有一个模板参数列表，它定义了两个参数（T1 和 T2），这两个参数的默认类型分别为 int 和 double。存取器函数 GetFirstValue ()和 GetSecondValue()用于查询对象的值，它们将根据模板实例化语法返回正确的对象类型。HoldsPair 定义了一种模式，可通过重用该模式针对不同的变量类型实现相同的逻辑。因此，使用模板可提高代码的可复用性。

14.4.10 模板类和静态成员

前面说过，模板是用于创建类的蓝图，而类是用于创建对象的蓝图。在模板类中，静态成员属性的工作原理是什么样的呢？第 9 章介绍过，如果将类成员声明为静态的，该成员将由类的所有实例共享。模板类的静态成员与此类似，由特定具体化的所有实例共享。也就是说，如果模板类 T 包含静态成员 X，该成员将在针对 int 具体化的所有实例之间共享；同样，它还将在针对 double 具体化的所有实例之间共享，且与针对 int 具体化的实例无关。换句话说，可以认为编译器创建了两个版本的 X：X_int 用于针对 int 具体化的实例，而 X_double 针对 double 具体化的实例，程序清单 14.5 演示了这一点。

程序清单 14.5 静态成员对模板类和实例的影响

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: template <typename T>
 4: class TestStatic
 5: {
 6: public:
 7:    static int StaticValue;
 8: };
 9:
10: // static member initialization
```

```
11: template<typename T> int TestStatic<T>::StaticValue;
12:
13: int main()
14: {
15:     TestStatic<int> Int_Year;
16:     cout << "Setting StaticValue for Int_Year to 2011" << endl;
17:     Int_Year.StaticValue = 2011;
18:     TestStatic<int> Int_2;
19:
20:     TestStatic<double> Double_1;
21:     TestStatic<double> Double_2;
22:     cout << "Setting StaticValue for Double_2 to 1011" << endl;
23:     Double_2.StaticValue = 1011;
24:
25:     cout << "Int_2.StaticValue = " << Int_2.StaticValue << endl;
26:     cout << "Double_1.StaticValue = " << Double_1.StaticValue << endl;
27:
28:     return 0;
29: }
```

▼ 输出：

```
Setting StaticValue for Int_Year to 2011
Setting StaticValue for Double_2 to 1011
Int_2.StaticValue = 2011
Double_1.StaticValue = 1011
```

▼ 分析：

在第 17 和 23 行，分别为针对 int 和 double 的模板具体化设置了成员 StaticValue。在 main()中的第 25 和 26 行，通过另一个实例（Int_2 和 Double_1 ）读取了这个静态成员的值。输出表明，得到的 StaticValue 不同：一个是 2011，这是通过另一个针对 int 具体化的实例设置的；另一个为 1011，这是通过另一个针对 double 具体化的实例设置的。

也就是说，对于针对每种类型具体化的类，编译器确保其静态变量不受其他类的影响。模板类的每个具体化都有自己的静态成员。

注意

在程序清单 14.5 中，第 11 行不可或缺，它初始化模板类的静态成员：

```
template<typename T> int TestStatic<T>::StaticValue;
```

对于模板类的静态成员，通用的初始化语法如下：

```
template<template parameters> StaticType ClassName<Template
  Arguments>::StaticVarName;
```

C++11

使用 static_assert 执行编译阶段检查

static_assert 是 C++11 新增的一项功能，让您能够在不满足指定条件时禁止编译。这好像不可思议，但对模板类来说很有用。您可能想禁止针对 int 实例化模板类，为此可使用 static_assert，它是一种编译阶段断言，可用于在开发环境（或控制台中）显示一条自定义消息：

```
static_assert(expression being validated, "Error message when check fails");
```

要禁止针对类型 int 实例化模板类，可使用 static_assert()，并将 sizeof(T)与 sizeof(int)进行比较，如果它们相等，就显示一条错误消息：

```
static_assert(sizeof(T) != sizeof(int), "No int please!");
```

程序清单 14.6 演示了一个模板类，它禁止针对特定类型实例化它。

程序清单 14.6 一个挑剔的模板类，在您针对 int 类型实例化时，它使用 static_assert 发出抗议

```
 0: template <typename T>
 1: class EverythingButInt
 2: {
 3: public:
 4:    EverythingButInt()
 5:    {
 6:       static_assert(sizeof(T) != sizeof(int), "No int please!");
 7:    }
 8: };
 9:
10: int main()
11: {
12:    EverythingButInt<int> test;  // template instantiation with int.
13:    return 0;
14: }
```

▼ 输出：

没有输出，因为这个程序不能通过编译，它显示一条错误消息，指出您指定的类型不正确：

```
error: No int please!
```

▼ 分析：

编译器发出的抗议是在第 6 行指定的。static_assert 是 C++11 新增的一项功能，让您能够禁止不希望的模板实例化。

14.4.11 在实际 C++编程中使用模板

模板最重要也是最强大的应用是在标准模板库（STL）中。STL 由一系列模板类和函数组成，它们分别包含泛型实用类和算法。这些 STL 模板类让您能够实现动态数组、链表以及包含键-值对的容器，而 sort 等算法可用于这些容器，从而对容器包含的数据进行处理。

前面介绍的模板语法有助于读者使用本书后面将详细介绍的 STL 容器和函数；更深入地理解 STL 将有助于使用 STL 中经过测试的可靠实现，从而编写出更高效的 C++程序，还有助于避免在模板细节上浪费时间。

应该	不应该
务必使用模板来实现通用概念。 务必使用模板而不是宏。	编写模板函数和模板类时，别忘了尽可能使用 const。 别忘了，模板类的静态成员由特定具体化的所有实例共享。

14.5 总结

本章更详细地介绍了预处理器。每当您运行编译器时，预处理器都将首先运行，对#define 等指令进行转换。

预处理器执行文本替换，但在使用宏时替换将比较复杂。通过使用宏函数，可根据在编译阶段传递给宏的参数进行复杂的文本替换。将宏中的每个参数放在括号内以确保进行正确的替换，这很重要。

模板有助于编写可重用的代码，它向开发人员提供了一种可用于不同数据类型的模式。模板可以取代宏，且是类型安全的。学习本章介绍的模板知识后，便为学习如何使用 STL 做好了准备！

14.6 问与答

问：在头文件中，为何要防范多次包含？

答：多次包含防范使用#ifndef、#define 和#endif，可避免头文件出现多次包含或递归包含错误，有时还可提高编译速度。

问：如果所需的功能使用宏函数和模板都能实现，在什么情况下应使用宏函数，而不是模板？

答：在任何情况下都应使用模板，而不是宏函数，因为模板不但提供了通用实现，还是类型安全的。宏函数不是类型安全的，最好不要使用。

问：调用模板函数时，需要指定模板参数类型吗？

答：通常不需要，因为编译器能够根据函数调用使用的实参推断出模板参数类型。

问：对于特定模板类，每个静态成员有多少个版本？

答：这完全取决于针对多少种类型实例化了该模板类。如果针对 int、string 和自定义类型 X 实例化了该模板类，则每个静态成员都有三个不同的版本——每种模板具体化一个。

14.7 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

14.7.1 测验

1. 什么是多次包含防范（inclusion guard）？
2. #define debug 0 与#undef debug 之间的区别何在？
3. 如果使用参数 20 调用下面的宏，结果将是多少？

```
#define SPLIT(x) x / 5
```

3. 如果用 10+10 调用问题 2 中的 HALVE 宏，结果将是多少？
5. 如何修改 SPLIT 宏以避得到错误的结果？

14.7.2 练习

1. 编写一个将两个数相乘的宏。
2. 编写一个模板，实现练习 1 中宏的功能。
3. 实现模板函数 swap，它交换两个变量的值。
4. 查错：您将如何改进下面的宏使其计算输入值的 1/4？

```
#define QUARTER(x) (x / 4)
```

5. 编写一个简单的模板类，它存储两个数组，数组的类型是通过模板参数列表指定的。数组包含 10 个元素，模板类应包含存取器函数，可用于操作数组元素。

第 15 章
标准模板库简介

简单地说，标准模板库（STL）是一组模板类和函数，向程序员提供了：

- 用于存储信息的容器；
- 用于访问容器存储的信息的迭代器；
- 用于操作容器内容的算法。

本章概述 STL 的这 3 个重要方面。

15.1 STL 容器

容器是用于存储数据的 STL 类，STL 提供了两种类型的容器类：

- 顺序容器；
- 关联容器。

另外，STL 还提供了被称为容器适配器（Container Adapter）的类，它们是顺序容器和关联容器的变种，包含的功能有限，用于满足特殊的需求。

15.1.1 顺序容器

顾名思义，顺序容器按顺序存储数据，如数组和列表。顺序容器具有插入速度快但查找操作相对较慢的特征。

STL 顺序容器包括：

- **std::vector**——操作与动态数组一样，在最后插入数据；可将 vector 视为书架，您可在一端添加和拿走图书；
- **std::deque**——与 std::vector 类似，但允许在开头插入或删除元素；
- **std::list**——操作与双向链表一样。可将它视为链条，对象被连接在一起，您可在任何位置添加或删除对象；
- **std::forward_list**——类似于 std::list，但是单向链表，只能沿一个方向遍历。

STL vector 类与数组类似，允许随机访问元素，即可使用下标运算符（[ ]）指定元素在 vector 中的位置（索引），从而直接访问或操作元素。另外，STL vector 是动态数组，因此能够根据应用程序在运行阶段的需求自动调整长度。为保留数组能够根据位置随机访问元素的特征，大多数 STL vector 实现都将所有元素存储在连续的存储单元中，因此需要调整长度的 vector 通常会降低应用程序的性能，这取决于它包含的对象类型。第 4 章简要地介绍了 vector（参见程序清单 4.4），而第 17 章将更详细地讨论该容器。

可将 STL list 类视为普通链表的 STL 实现。虽然 list 中的元素不能像 STL vector 中的元素那样随机

访问，但 list 可使用不连续的内存块组织元素，因此它不像 std::vector 那样需要给内部数组重新分配内存，进而导致性能问题。STL list 类将在第 18 章更详细地讨论。

15.1.2 关联容器

关联容器按指定的顺序存储数据，就像词典一样。这将降低插入数据的速度，但在查询方面有很大的优势。

STL 提供的关联容器包括：

- **std::set**——存储各不相同的值，在插入时进行排序；容器的复杂度为对数；
- **std::unordered_set**——存储各不相同的值，在插入时进行排序；容器的复杂度为常数。这种容器是 C++11 新增的；
- **std::map**——存储键-值对，并根据唯一的键排序；容器的复杂度为对数；
- **std::unordered_map**——存储键-值对，并根据唯一的键排序；容器的复杂度为对数。这种容器是 C++11 新增的；
- **std::multiset**——与 set 类似，但允许存储多个值相同的项，即值不需要是唯一的；
- **std::unordered_multiset**——与 unordered_set 类似，但允许存储多个值相同的项，即值不需要是唯一的。这种容器是 C++11 新增的；
- **std::multimap**——与 map 类似，但不要求键是唯一的；
- **std::unordered_multimap**——与 unordered_map 类似，但不要求键是唯一的。这种容器是 C++11 新增的。

可通过谓词函数编程定制 STL 容器的排序标准。

提示

有些 STL 实现也支持关联容器 hash_set、hash_multiset、hash_map 和 hash_multimap，它们与标准支持的 unordered_*容器类似。在有些情况下，hash_*和 unordered_*容器有更好的元素搜索性能，因为其元素访问时间为常量（不依赖于容器包含的元素数）。通常，这些容器提供了与相应的标准容器相同的公有方法，因此使用起来很容易。

使用遵循标准的容器时，代码将更容易在不同平台和编译器之间移植，因此是更好的选择。另外，虽然遵循标准的容器的性能呈对数降低，但这可能并不会严重影响应用程序。

15.1.3 选择正确的容器

显然，可能有多种 STL 容器能够满足应用程序的需求，这时必须做出选择，这种选择很重要，因为错误的选择将导致应用程序性能低下。

因此，在选择容器前，评估各种容器的优缺点很重要，如表 15.1 所示。

表 15.1 STL 容器类的特点

容　器	优　点	缺　点
std::vector（顺序容器）	在末尾插入数据时快（时间固定） 可以像访问数组一样进行访问	调整大小时将影响性能 搜索时间与容器包含的元素个数成正比 只能在末尾插入数据
std::deque（顺序容器）	具备 vector 的所有优点，还可在容器开头插入数据，插入时间也是固定的	有 vector 的所有缺点 与 vector 不同的是，根据规范，deque 不需要支持 reserve()函数，该函数让程序员能够给 vector 预留内存空间，以免频繁地调整大小，从而提高性能

续表

容　器	优　点	缺　点
std::list（顺序容器）	在 list 开头、中间或末尾插入数据，所需时间都是固定的 将元素从 list 中删除所需的时间是固定的，而不管元素的位置如何 插入或删除元素后，指向 list 中其他元素的迭代器仍有效	不能像数组那样根据索引随机访问元素 搜索速度比 vector 慢，因为元素没有存储在连续的内存单元中 搜索时间与容器中的元素个数成正比
std::forward_list（顺序容器）	单向链表类，只能沿一个方向遍历	只能使用 push_front()在链表开头插入元素
std::set（关联容器）	搜索时间不是与容器中的元素个数（而与元素个数的对数）成正比，因此搜索速度通常比顺序容器快得多	元素的插入速度比顺序容器慢，因为在插入时对元素进行排序
std::unordered_set（关联容器）	搜索、插入和删除的速度几乎不受容器包含的元素个数的影响	由于元素未被严格排序，因此不能依赖于元素在容器中的相对位置
std::multiset（关联容器）	需要存储非唯一的值时，应使用这种容器	插入速度可能比顺序容器慢，因为在插入时对元素（键-值对）进行排序
std::unordered_multiset（关联容器）	需要存储非唯一的值时，应使用这种容器，而不是 unorder_set 性能与 unordered_set 类似，即搜索、插入和删除元素的时间是固定的，不受容器长度的影响	由于元素未被严格排序，因此不能依赖于元素在容器中的相对位置
std::map（关联容器）	用于存储键-值对的容器，搜索时间与元素个数的对数成正比，因此搜索速度通常比顺序容器快得多	插入时进行排序，因此插入速度比顺序容器慢
std::unordered_map（关联容器）	搜索、插入和删除元素的时间是固定的，不受容器长度的影响	元素未被严格排序，不适合用于顺序很重要的情形
std::multimap（关联容器）	在需要存储键-值且要求键不唯一时，应选择这种容器，而不是 std::map	插入时进行排序，因此插入速度比顺序容器慢
std::unordered_multimap（关联容器）	在需要存储键-值且要求键不唯一时，应选择这种容器，而不是 multimap 搜索、插入和删除元素的时间是固定的，不受容器长度的影响	元素未被严格排序，在需要依赖于元素的相对顺序时，不能使用它

15.1.4 容器适配器

容器适配器（Container Adapter）是顺序容器和关联容器的变种，其功能有限，用于满足特定的需求。主要的适配器类如下。

- **std::stack**：以 LIFO（后进先出）的方式存储元素，让您能够在栈顶插入（压入）和删除（弹出）元素。
- **std::queue**：以 FIFO（先进先出）的方式存储元素，让您能够删除最先插入的元素。
- **std::priority_queue**：以特定顺序存储元素，因为优先级最高的元素总是位于队列开头。

这些容器将在第 24 章详细讨论。

15.2 STL 迭代器

最简单的迭代器是指针。给定一个指向数组中的第一个元素的指针，可递增该指针使其指向下一个元素，还可直接对当前位置的元素进行操作。

STL 中的迭代器是模板类，从某种程度上说，它们是泛型指针。这些模板类让程序员能够对 STL 容器进行操作。注意，操作也可以是以模板函数的方式提供的 STL 算法，迭代器是一座桥梁，让这些

模板函数能够以一致而无缝的方式处理容器，而容器是模板类。

STL 提供的迭代器分两大类。

- **输入迭代器**：通过对输入迭代器解除引用，它将引用对象，而对象可能位于集合中。最严格的输入迭代器确保只能以只读的方式访问对象。
- **输出迭代器**：输出迭代器让程序员对集合执行写入操作。最严格的输出迭代器确保只能执行写入操作。

上述两种基本迭代器可进一步分为三类。

- **前向迭代器**：这是输入迭代器和输出迭代器的一种细化，它允许输入与输出。前向迭代器可以是 const 的，只能读取它指向的对象；也可以改变对象，即可读写对象。前向迭代器通常用于单向链表。
- **双向迭代器**：这是前向迭代器的一种细化，可对其执行递减操作，从而向后移动。双向迭代器通常用于双向链表。
- **随机访问迭代器**：这是对双向迭代器的一种细化，可将其加减一个偏移量，还可将两个迭代器相减以得到集合中两个元素的相对距离。随机访问迭代器通常用于数组。

注意 从实现层面说，可将“细化”视为继承或具体化。

15.3 STL 算法

查找、排序和反转等都是标准的编程需求，不应让程序员重复实现这样的功能。因此 STL 以 STL 算法的方式提供这些函数，通过结合使用这些函数和迭代器，程序员可对容器执行一些最常见的操作。

最常用的 STL 算法如下所示。

- **std::find**：在集合中查找值。
- **std::find_if**：根据用户指定的谓词在集合中查找值。
- **std::reverse**：反转集合中元素的排列顺序。
- **std::remove_if**：根据用户定义的谓词将元素从集合中删除。
- **std::transform**：使用用户定义的变换函数对容器中的元素进行变换。

这些算法都是 std 命名空间中的模板函数，要使用它们，必须包含标准头文件<algorithm>。

15.4 使用迭代器在容器和算法之间交互

下面通过一个示例阐述迭代器如何无缝地将容器和 STL 算法连接起来。程序清单 15.1 所示的程序使用了 STL 顺序容器 std::vector（它类似于动态数组）来存储一些整数，再使用 std::find 算法在集合中查找一个整数。请注意迭代器是如何在这两者之间搭建桥梁的。请不要担心语法和功能，std::vector 等容器以及 std::find 等算法将分别在第 17 章和第 23 章详细讨论。如果您觉得这部分很复杂，可暂时跳过。

程序清单 15.1 在 vector 中查找元素及其位置

```
1: #include <iostream>
2: #include <vector>
3: #include <algorithm>
4: using namespace std;
5:
6: int main ()
7: {
8:    // A dynamic array of integers
9:    vector <int> vecIntegerArray;
```

```
10:
11:     // Insert sample integers into the array
12:     vecIntegerArray.push_back (50);
13:     vecIntegerArray.push_back (2991);
14:     vecIntegerArray.push_back (23);
15:     vecIntegerArray.push_back (9999);
16:
17:     cout << "The contents of the vector are: " << endl;
18:
19:     // Walk the vector and read values using an iterator
20:     vector <int>::iterator iArrayWalker = vecIntegerArray.begin ();
21:
22:     while (iArrayWalker != vecIntegerArray.end ())
23:     {
24:         // Write the value to the screen
25:         cout << *iArrayWalker << endl;
26:
27:         // Increment the iterator to access the next element
28:         ++ iArrayWalker;
29:     }
30:
31:     // Find an element (say 2991) in the array using the 'find' algorithm...
32:     vector <int>::iterator iElement = find (vecIntegerArray.begin ()
33:                             ,vecIntegerArray.end (), 2991);
34:
35:     // Check if value was found
36:     if (iElement != vecIntegerArray.end ())
37:     {
38:         // Value was found... Determine position in the array:
39:         int Position = distance (vecIntegerArray.begin (), iElement);
40:         cout << "Value "<< *iElement;
41:         cout << " found in the vector at position: " << Position << endl;
42:     }
43:
44:     return 0;
45: }
```

▼ **输出：**

```
The contents of the vector are:
50
2991
23
9999
Value 2991 found in the vector at position: 1
```

▼ **分析：**

程序清单 15.1 演示了如何使用迭代器遍历向量。迭代器是一个接口，将算法（find）连接到其要操作的数据所属的容器（如 vector）。第 20 行声明了迭代器对象 iArrayWalker，并将其初始化为指向容器开头，即 vector 的成员函数 begin()返回的值。第 22～29 行演示了如何在循环中使用该迭代器遍历并显示 vector 包含的元素，这与显示静态数组的内容极其相似。迭代器的用法在所有 STL 容器中都相同。所有容器都提供了 begin()函数和 end()函数，其中前者指向第一个元素，后者指向容器中最后一个元素的后面。这就是第 22 行的 while 循环在 end()前面而不是 end()处结束的原因。第 32 行演示了如何使用 find 在 vector 中查找值。find 操作的结果也是一个迭代器，通过将该迭代器与容器末尾进行比较，可判断 find 是否成功，如第 36 行所示。如果找到了元素，便可对该迭代器解除引用（就像对指针解除引用一样）以显示该元素。算法 distance 计算找到的元素的所处位置的偏移量。

如果将程序清单 15.1 中所有的 vector 都替换为 deque，代码仍能通过编译并完美地运行。这表明

迭代器让您能够轻松地使用算法和容器。

C++11

使用关键字 auto 让编译器确定类型

在程序清单 15.1 中，有多个迭代器声明，这些声明类似于下面这样：

```
20:    vector <int>::iterator iArrayWalker = vecIntegerArray.begin ();
```

上述迭代器类型定义看起来令人恐怖。如果您使用的是遵循 C++11 的编译器，可将这行代码简化成下面这样：

```
20:    auto iArrayWalker = vecIntegerArray.begin (); // compiler detects type
```

注意到将变量类型声明为 auto 时，必须对其进行初始化，这样编译器才能根据初始值推断变量的类型。

15.5 STL 字符串类

STL 提供了一个专门为操纵字符串而设计的模板类：std::basic_string<T>，该模板类的两个常用具体化如下。

- **std::string**：基于 char 的 std::basic_string 具体化，用于操纵简单字符串。
- **std::wstring**：基于 wchar_t 的 std::basic_string 具体化，用于操纵宽字符串。

第 16 章将详细讨论这个实用类，届时您将发现，它使得使用和操纵字符串非常简单。

15.6 总结

本章介绍了 STL 容器、迭代器和算法后面的基本概念，还简要地介绍了 basic_string<T>，这个类将在本书后面详细讨论。容器、迭代器和算法是最重要的 STL 概念，深入理解它们有助于在应用程序中高效地使用 STL。第 17～25 章将更详细地解释这些概念的实现及其应用。

15.7 问与答

问：我需要一个数组，但不知道它应包含多少个元素。请问我应使用哪种 STL 容器？

答：std::vector 或 std::deque 能够很好地满足这种需求。这两种容器都负责管理内存，并可根据应用程序需求动态地调整大小。

问：我的应用程序经常需要执行搜索操作，我应选择哪种容器？

答：诸如 std::map 和 std::set 及其 unordered 变种等关联容器最适合于需要经常进行搜索的应用程序。

问：我要存储键-值对，并希望能够快速完成查找，但键可能不是唯一的。我应选择哪种容器？

答：关联容器 std::multimap 适合这种需求。multimap 可存储非唯一的键-值对，查找速度也快，这是关联容器的一个特点。

问：我要开发一个能够在不同平台和编译器之间移植的应用程序，该程序还需要使用能够根据键快速查询的容器。我应使用 std::map 还是 std::hash_map？

答：移植性是一个重要约束条件，必须使用遵循标准的容器。如果您使用的是遵循 C++11 的编译器，也可使用 std::unordered_map。

15.8 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

测验

1. 要包含一个对象数组，并允许在开头和末尾插入对象，应使用哪种容器？
2. 要存储元素以进行快速查找，应选择哪种容器？
3. 要使用 std:set 存储元素，并根据除元素值外的其他条件进行存储和查找，可能吗？
4. STL 的哪部分将算法和容器连接起来，让算法能够操作容器中的元素？
5. 如果应用程序要移植到不同的平台，并使用不同的 C++编译器进行编译，是否可选择使用容器 hash_set？

第 16 章
STL string 类

标准模板库（STL）向程序员提供了一个用于字符串操作的容器类。string 类不仅能够根据应用程序的需求动态调整大小，还提供了很有用的辅助函数（方法），可帮助操作字符串，这让程序员能够在应用程序中使用标准的、经过测试的可移植功能，并将其主要精力放在开发应用程序的重要功能上。

在本章中，您将学习：

- 为何需要字符串操作类；
- 如何使用 STL string 类；
- STL 如何帮助您轻松地执行拼接、附加、查找以及其他字符串操作；
- 如何使用基于模板的 STL string 实现。

16.1 为何需要字符串操作类

在 C++中，字符串是一个字符数组。第 4 章介绍过，最简单的字符数组可这样定义：

```
char staticName [20];
```

该语句声明了一个名为 staticName 的字符数组（也叫字符串），其长度是固定的（静态的），包含 20 个元素。可以看到，这个缓冲区可存储一个长度有限的字符串，如果试图存储的字符数超出限制将溢出。不能调整静态数组的长度，为避开这种限制，C++支持动态分配内存，因此可以如下定义更动态的字符数组：

```
char* dynamicName = new char [ArrayLength];
```

这定义了一个动态分配的字符数组，其长度由变量 ArrayLength 的值指定，而这种值是在运行阶段确定的，因此该数组的长度是可变的。然而，如果要在运行阶段改变数组的长度，必须首先释放以前分配给它的内存，再重新分配内存来存储数据。

如果将 char*用作类的成员属性，情况将更复杂。将对象赋给另一个对象时，如果编写正确的复制构造函数和赋值运算符，两个对象将包含同一个指针的拷贝，该指针指向相同的缓冲区。其结果是，两个对象的字符串指针存储的地址相同，指向同一个内存单元。其中一个对象被销毁时，另一个对象中的指针将非法，让应用程序面临崩溃的危险。

字符串类帮您解决了这些问题。STL 字符串类 std::string 和 std::wstring 分别模拟了普通字符串和宽字符串，可提供如下帮助：

- 减少了程序员在创建和操作字符串方面需要做的工作；
- 在内部管理内存分配细节，从而提高了应用程序的稳定性；
- 提供了复制构造函数和赋值运算符，可确保成员字符串得以正确复制；
- 提供了帮助执行复制、截短、查找和删除等操作的实用函数；

- 提供了用于比较的运算符；
- 让程序员能够将精力放在应用程序的主要需求而不是字符串操作细节上。

注意

> std::string 和 std::wstring 实际上是同一个模板类（std::basic_string<T>）的具体化，即分别针对类型 char 和 wchar 的具体化。学习使用其中一个后，就能使用这些方法和运算符用于另一个。

稍后将以 std::string 为例，介绍 STL 字符串类提供的一些辅助函数。

16.2 使用 STL string 类

最常用的字符串函数包括：

- 复制；
- 连接；
- 查找字符和子字符串；
- 截短；
- 使用标准模板库提供的算法实现字符串反转和大小写转换。

要使用 STL string 类，必须包含头文件<string>。

16.2.1 实例化和复制 STL string

string 类提供了很多重载的构造函数，因此可以多种方式进行实例化和初始化。例如，可使用常量字符串初始化 STL string 对象或将常量字符串赋给 STL string 对象：

```
const char* constCStyleString = "Hello String!";
std::string strFromConst (constCStyleString);
```

或：

```
std::string strFromConst = constCStyleString;
```

上述代码与下面的代码类似：

```
std::string str2 ("Hello String!");
```

显然，实例化并初始化 string 对象时，无需关心字符串长度和内存分配细节。STL string 类的构造函数将自动完成这些工作。

同样，可使用一个 string 对象来初始化另一个：

```
std::string str2Copy (str2);
```

可让 string 的构造函数只接受输入字符串的前 n 个字符：

```
// Initialize a string to the first 5 characters of another
std::string strPartialCopy (constCStyleString, 5);
```

还可这样初始化 string 对象，即使其包含指定数量的特定字符：

```
// Initialize a string object to contain 10 'a's
std::string strRepeatChars (10, 'a');
```

程序清单 16.1 演示了一些实例化和复制 STL string 的常见方法。

程序清单 16.1 实例化和复制 STL string 的方法

```
0: #include <string>
1: #include <iostream>
2:
3: int main ()
4: {
```

```
 5:    using namespace std;
 6:    const char* constCStyleString = "Hello String!";
 7:    cout << "Constant string is: " << constCStyleString << endl;
 8:
 9:    std::string strFromConst (constCStyleString);  // constructor
10:    cout << "strFromConst is: " << strFromConst << endl;
11:
12:     std::string str2 ("Hello String!");
13:    std::string str2Copy (str2);
14:    cout << "str2Copy is: " << str2Copy << endl;
15:
16:    // Initialize a string to the first 5 characters of another
17:    std::string strPartialCopy (constCStyleString, 5);
18:    cout << "strPartialCopy is: " << strPartialCopy << endl;
19:
20:    // Initialize a string object to contain 10 'a's
21:    std::string strRepeatChars (10, 'a');
22:    cout << "strRepeatChars is: " << strRepeatChars << endl;
23:
24:    return 0;
25: }
```

▼ **输出：**

```
Constant string is: Hello String!
strFromConst is: Hello String!
str2Copy is: Hello String!
strPartialCopy is: Hello
strRepeatChars is: aaaaaaaaaa
```

▼ **分析：**

上述示例代码演示了如何实例化 STL string 对象，还演示了如何将 STL string 对象初始化为另一个字符串、字符串的一部分或多个相同的字符。ConstCStyleString 是一个包含示例值的 C 风格字符串，它是第 6 行初始化的。从第 9 行可知，使用 std::string 的构造函数进行复制非常简单。第 12 行将另一个常量字符串复制给 std::string 对象 str2，第 13 行演示了如何使用 std::string 的另一个重载构造函数来复制 std::string 对象，从而获得 str2Copy。第 17 行演示了如何进行部分复制。第 21 行演示如何实例化一个 std::string 对象，并将其初始化为包含多个相同的字符。这段代码只是一个小型演示程序，演示了通过使用 std::string 及其众多复制构造函数，创建、复制和显示字符串很容易。

注意

如果要将一个 C 风格字符串复制到另一个 C 风格字符串中，程序清单 16.1 的第 9 行代码将为：

```
const char* constCStyleString = "Hello World!";

// To create a copy, first allocate memory for one...
char * pszCopy = new char [strlen (constCStyleString) + 1];
strcpy (pszCopy, constCStyleString);  // The copy step

// deallocate memory after using pszCopy
delete [] pszCopy;
```

可以看到，这需要更多的代码，导致错误的可能性也更大，同时程序员还需要负责管理内存的分配与释放。这些工作 STL string 都能完成，还能完成其他工作！

16.2.2 访问 std::string 的字符内容

要访问 STL string 的字符内容，可使用迭代器，也可采用类似于数组的语法并使用下标运算符（[]）

提供偏移量。要获得 string 对象的 C 风格表示，可使用成员函数 c_str ()，如程序清单 16.2 所示。

程序清单 16.2 两种访问 STL string 字符元素的方式：运算符[ ]和迭代器

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

int main ()
{
    using namespace std;

    // The sample string
    string strSTLString ("Hello String");

     // Access the contents of the string using array syntax
    cout<<"Displaying the elements in the string using array-syntax: "<<endl;
    for ( size_t nCharCounter = 0
        ; nCharCounter < strSTLString.length ()
        ; ++ nCharCounter )
    {
        cout << "Character [" << nCharCounter << "] is: ";
        cout << strSTLString [nCharCounter] << endl;
    }
    cout << endl;

    // Access the contents of a string using iterators
    cout << "Displaying the contents of the string using iterators: " <<
    endl;
    int charOffset = 0;
    string::const_iterator iCharacterLocator;
    for ( iCharacterLocator = strSTLString.begin ()
        ; iCharacterLocator != strSTLString.end ()
        ; ++ iCharacterLocator )
    {
       cout << "Character [" << charOffset ++ << "] is: ";
       cout << *iCharacterLocator << endl;
    }
    cout << endl;

    // Access the contents of a string as a C-style string
    cout << "The char* representation of the string is: ";
    cout << strSTLString.c_str () << endl;

    return 0;
}
```

▼ 输出：

```
Displaying the elements in the string using array-syntax:
Character [0] is: H
Character [1] is: e
Character [2] is: l
Character [3] is: l
Character [4] is: o
Character [5] is:
Character [6] is: S
Character [7] is: t
Character [8] is: r
Character [9] is: i
Character [10] is: n
Character [11] is: g
```

```
Displaying the elements in the string using array-syntax:
Character [0] is: H
Character [1] is: e
Character [2] is: l
Character [3] is: l
Character [4] is: o
Character [5] is:
Character [6] is: S
Character [7] is: t
Character [8] is: r
Character [9] is: i
Character [10] is: n
Character [11] is: g

The char* representation of the string is: Hello String
```

▼ **分析：**

上述代码演示了访问string内容的多种方式。迭代器很重要，因为很多string成员函数都以迭代器的方式返回其结果。第12～18行使用std::string类实现的下标运算符（[]）以类似数组的语法显示string中的字符。注意，这个运算符要求提供偏移量，如第17行所示。因此，确保不超出string的边界很重要，即读取字符时，提供的偏移量不能大于string的长度。第25～31行也逐字符显示string的内容，但使用的是迭代器。

16.2.3 拼接字符串

要拼接字符串，可使用运算符+=，也可使用成员函数append：

```
string strSample1 ("Hello");
string strSample2 (" String!");
strSample1 += strSample2;    // use std::string::operator+=
// alternatively use std::string::append()
strSample1.append (strSample2);  // (overloaded for char* too)
```

程序清单16.3演示了这两种方式。

程序清单16.3 使用加法赋值运算符（+=）或方法append()拼接字符串

```
 0: #include <string>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:    using namespace std;
 6:
 7:    string strSample1 ("Hello");
 8:    string strSample2 (" String!");
 9:
10:    // Concatenate
11:    strSample1 += strSample2;
12:    cout << strSample1 << endl << endl;
13:
14:    string strSample3 (" Fun is not needing to use pointers!");
15:    strSample1.append (strSample3);
16:    cout << strSample1 << endl << endl;
17:
18:    const char* constCStyleString = " You however still can!";
19:    strSample1.append (constCStyleString);
20:    cout << strSample1 << endl;
21:
```

```
22:    return 0;
23: }
```

▼ **输出：**

```
Hello String!

Hello String! Fun is not needing to use pointers!

Hello String! Fun is not needing to use pointers! You however still can!
```

▼ **分析：**

第 11、15 和 19 行演示了各种拼接 STL string 的方法。请注意运算符+=的用法以及 append 函数的功能。append 函数有多个重载版本，能够接受另一个 string 对象（如第 15 行所示），还能接受一个 C 风格字符串。

16.2.4 在 string 中查找字符或子字符串

STL string 类提供了成员函数 find，该函数有多个重载版本，可在给定 string 对象中查找字符或子字符串。

```
// Find substring "day" in a string strSample, starting at position 0
size_t charPos = strSample.find ("day", 0);

// Check if the substring was found, compare against string::npos
if (charPos != string::npos)
   cout << "First instance of \"day\" was found at position " << charPos;
else
   cout << "Substring not found." << endl;
```

程序清单 16.4 演示了 std::string::find 的用法。

程序清单 16.4 使用 string::find()查找子字符串或字符

```
 0: #include <string>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:    using namespace std;
 6:
 7:    string strSample ("Good day String! Today is beautiful!");
 8:    cout << "The sample string is: " << endl;
 9:    cout << strSample << endl << endl;
10:
11:    // Find substring "day" in it...
12:    size_t charPos = strSample.find ("day", 0);
13:
14:    // Check if the substring was found...
15:    if (charPos != string::npos)
16:       cout << "First instance of \"day\" was found at position " << charPos;
17:    else
18:       cout << "Substring not found." << endl;
19:
20:    cout << endl << endl;
21:
22:    cout << "Locating all instances of substring \"day\"" << endl;
```

```
23:    size_t SubstringPos = strSample.find ("day", 0);
24:
25:    while (SubstringPos != string::npos)
26:    {
27:       cout << "\"day\" found at position " << SubstringPos << endl;
28:
29:       // Make find search forward from the next character
30:       size_t nSearchPosition = SubstringPos + 1;
31:
32:       SubstringPos = strSample.find ("day", nSearchPosition);
33:    }
34:
35:    cout << endl;
36:
37:    cout << "Locating all instances of character 'a'" << endl;
38:    const char charToSearch = 'a';
39:    charPos = strSample.find (charToSearch, 0);
40:
41:    while (charPos != string::npos)
42:    {
43:       cout << "'" << charToSearch << "' found";
44:       cout << " at position: " << charPos << endl;
45:
46:       // Make find search forward from the next character
47:       size_t charSearchPos = charPos + 1;
48:
49:       charPos = strSample.find (charToSearch, charSearchPos);
50:    }
51:
52:    return 0;
53: }
```

▼ 输出：

```
The sample string is:
Good day String! Today is beautiful!

First instance of "day" was found at position 5

Locating all instances of substring "day"
"day" found at position 5
"day" found at position 19

Locating all instances of character 'a'
'a' found at position: 6
'a' found at position: 20
'a' found at position: 28
```

▼ 分析：

第12～18行演示了find函数的最简单用法，它判断在string中是否找到了特定子字符串。这是通过将find操作的结果与std::string::npos（实际值为–1）进行比较实现的，std::string::npos表明没有找到要搜索的元素。如果find函数没有返回npos，它将返回一个偏移量，指出子字符串或字符在string中的位置。

这段代码演示了如何在while循环中使用find函数在STL string查找指定字符或子字符串的所有实例。这里使用的find函数的重载版本接受两个参数：要搜索的子字符串或字符，以及命令find从哪里开始搜索的偏移量。

注意 STL string 还有一些与 find()类似的函数，如 find_first_of()、find_first_not_of()、find_last_of()和 find_last_not_of()，这些函数可满足程序员的其他编程需求。

16.2.5 截短 STL string

STL string 类提供了 erase 函数，可用于：

- 在给定偏移位置和字符数时删除指定数目的字符；

```
string strSample ("Hello String! Wake up to a beautiful day!");
strSample.erase (13, 28);  // Hello String!
```

- 在给定指向字符的迭代器时删除该字符；

```
strSample.erase (iCharS); // iterator points to a specific character
```

- 在给定由两个迭代器指定的范围时删除该范围内的字符。

```
   strSample.erase (strSample.begin (), strSample.end ()); // erase from begin
to end
```

程序清单 16.5 的示例演示了 string::erase()函数的各种重载版本的用途。

程序清单 16.5 使用 string::erase 从指定偏移位置或迭代器指定的位置开始截短字符串

```
 0: #include <string>
 1: #include <algorithm>
 2: #include <iostream>
 3:
 4: int main ()
 5: {
 6:     using namespace std;
 7:
 8:     string strSample ("Hello String! Wake up to a beautiful day!");
 9:     cout << "The original sample string is: " << endl;
10:     cout << strSample << endl << endl;
11:
12:     // Delete characters from the string given position and count
13:     cout << "Truncating the second sentence: " << endl;
14:     strSample.erase (13, 28);
15:     cout << strSample << endl << endl;
16:
17:     // Find a character 'S' in the string using STL find algorithm
18:     string::iterator iCharS = find ( strSample.begin ()
19:                                      , strSample.end (), 'S');
20:
21:     // If character found, 'erase' to deletes a character
22:     cout << "Erasing character 'S' from the sample string:" << endl;
23:     if (iCharS != strSample.end ())
24:         strSample.erase (iCharS);
25:
26:     cout << strSample << endl << endl;
27:
28:     // Erase a range of characters using an overloaded version of erase()
29:     cout << "Erasing a range between begin() and end(): " << endl;
30:     strSample.erase (strSample.begin (), strSample.end ());
31:
32:    // Verify the length after the erase() operation above
33:     if (strSample.length () == 0)
34:         cout << "The string is empty" << endl;
35:
36:     return 0;
37: }
```

▼ 输出：

```
The original sample string is:
Hello String! Wake up to a beautiful day!

Truncating the second sentence:
Hello String!
Erasing character 'S' from the sample string:
Hello tring!

Erasing a range between begin() and end():
The string is empty
```

▼ 分析：

该程序清单演示 erase 函数的 3 个版本。其中一个版本在给定偏移位置和字符数的情况下删除指定数目的字符，如第 14 行所示；另一个版本在给定指向字符的迭代器的情况下删除指定的字符，如第 24 行所示；最后一个版本在给定由两个迭代器指定的范围的情况下删除该范围内的字符，如第 30 行所示。在这里，范围是由 string 的成员函数 begin()和 end()指定的，它包含字符串的所有内容，因此对该范围调用 erase()将清除 string 对象的全部内容。注意，string 类还提供了 clear()函数，该函数清除全部内容并重置 string 对象。

C++11

使用 auto 简化冗长的迭代器声明

对于程序清单 16.5 中冗长的迭代器声明，C++11 可帮助简化：

```
18:     string::iterator iCharS = find ( strSample.begin ()
19:                                    , strSample.end (), 'S');
```

为此，可使用第 3 章介绍的关键字 auto：

```
auto iCharS = find ( strSample.begin ()
                   , strSample.end (), 'S');
```

编译器将根据 std::find 的返回类型自动推断变量 iCharS 的类型。

16.2.6 字符串反转

有时需要反转字符串的内容。假设要判断用户输入的字符串是否为回文，方法之一是将其反转，再与原来的字符串进行比较。反转 STL string 很容易，只需使用泛型算法 std::reverse：

```
string strSample ("Hello String! We will reverse you!");
reverse (strSample.begin (), strSample.end ());
```

程序清单 16.6 演示了如何将算法 std::reverse 用于 std::string。

程序清单 16.6 使用 std::reverse 反转 STL string

```
 0: #include <string>
 1: #include <iostream>
 2: #include <algorithm>
 3:
 4: int main ()
 5: {
 6:    using namespace std;
 7:
 8:    string strSample ("Hello String! We will reverse you!");
 9:    cout << "The original sample string is: " << endl;
10:    cout << strSample << endl << endl;
```

```
11:
12:    reverse (strSample.begin (), strSample.end ());
13:
14:    cout << "After applying the std::reverse algorithm: " << endl;
15:    cout << strSample << endl;
16:
17:    return 0;
18: }
```

▼ **输出：**

```
The original sample string is:
Hello String! We will reverse you!

After applying the std::reverse algorithm:
!uoy esrever lliw eW !gnirtS olleH
```

▼ **分析：**

第 12 行的 std::reverse 算法根据两个输入参数指定的边界反转边界内的内容。在这里，两个边界分别是 string 对象的开头和末尾，因此整个字符串都被反转。只要提供合适的输入参数，也可将字符串的一部分反转。注意，边界不能超过 end()。

16.2.7 字符串的大小写转换

要对字符串进行大小写转换，可使用算法 std::transform，它对集合中的每个元素执行一个用户指定的函数。在这里，集合是 string 对象本身。程序清单 16.7 演示了如何对 string 中的字符进行大小写转换。

程序清单 16.7 使用 std::transform 将 STL string 转换为大写

```
 0: #include <string>
 1: #include <iostream>
 2: #include <algorithm>
 3:
 4: int main ()
 5: {
 6:    using namespace std;
 7:
 8:    cout << "Please enter a string for case-convertion:" << endl;
 9:    cout << "> ";
10:
11:    string strInput;
12:    getline (cin, strInput);
13:    cout << endl;
14:
15:    transform(strInput.begin(),strInput.end(),strInput.begin(),toupper);
16:    cout << "The string converted to upper case is: " << endl;
17:    cout << strInput << endl << endl;
18:
19:    transform(strInput.begin(),strInput.end(),strInput.begin(),tolower);
20:    cout << "The string converted to lower case is: " << endl;
21:    cout << strInput << endl << endl;
22:
23:    return 0;
24: }
```

▼ 输出：

```
Please enter a string for case-convertion:
> ConverT thIS StrINg!

The string converted to upper case is:
CONVERT THIS STRING!

The string converted to lower case is:
convert this string!
```

▼ 分析：

第15行和第19行演示了如何使用std::transform来改变STL string的大小写。

16.3 基于模板的STL string实现

前面说过，std::string类实际上是STL模板类std::basic_string <T>的具体化。容器类basic_string的模板声明如下：

```
template<class _Elem,
    class _Traits,
    class _Ax>
    class basic_string
```

在该模板定义中，最重要的参数是第一个：_Elem，它指定了basic_string对象将存储的数据类型。因此，std::string使用_Elem=char具体化模板basic_string的结果，而wstring使用_Elem= wchar具体化模板basic_string的结果。

换句话说，STL string类的定义如下：

```
typedef basic_string<char, char_traits<char>, allocator<char> >
    string;
```

而STL wstring类的定义如下：

```
typedef basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t> >
    string;
```

因此，前面介绍的所有string功能和函数实际上都是basic_string提供的，它们也适用于STL wstring类。

提示 如果编写的应用程序需要更好地支持非拉丁字符，如中文和日文，应使用std::wstring。

16.4 总结

本章介绍了STL string类，它是标准模板库提供的一个容器，可满足程序员众多的字符串操作需求。使用这个类的优点是，原本由程序员负责的内存管理、字符串比较和字符串操作都将由STL框架提供的一个容器类完成。

16.5 问与答

问：我要使用std::reverse来反转一个字符串，要使用这个函数，需要包含哪个头文件？

答：要使用std::reverse ，需要包含头文件<algorithm>。

问：使用tolower()函数将字符串转换为小写时，std::transform的作用是什么？

答： std::transform 对 string 对象中指定边界内的每个字符调用 tolower()函数。

问：为什么 std::wstring 和 std::string 的行为和成员函数完全相同？
答： 因为它们都是具体化模板类 std::basic_string 的结果。

问：STL string 类的比较运算符<在执行比较时是否区分大小写？
答： 区分大小写。

16.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

16.6.1 测验

1. std::string 具体化了哪个 STL 模板类？
2. 如果要对两个字符串进行区分大小写的比较，该如何做？
3. STL string 与 C 风格字符串是否类似？

16.6.2 练习

1. 编写一个程序检查用户输入的单词是否为回文。例如，ATOYOTA 是回文，因为该单词反转后与原来相同。
2. 编写一个程序，告诉用户输入的句子包含多少个元音字母。
3. 将字符串的字符交替地转换为大写。
4. 编写一个程序，将 4 个 string 对象分别初始化为 I、Love、STL 和 String，然后在这些字符串之间添加空格，再显示整个句子。

第 17 章
STL 动态数组类

动态数组让程序员能够灵活地存储数据，无需在编写应用程时就知道数组的长度。显然，这是一种常见的需求，标准模板库（STL）通过 std::vector 类提供了现成的解决方案。

在本章中，您将学习：

- std::vector 的特点；
- 典型的 vector 操作；
- vector 的大小与容量；
- STL deque 类。

17.1 std::vector 的特点

vector 是一个模板类，提供了动态数组的通用功能，具有如下特点：

- 在数组末尾添加元素所需的时间是固定的，即在末尾插入元素的所需时间不随数组大小而异，在末尾删除元素也如此；
- 在数组中间添加或删除元素所需的时间与该元素后面的元素个数成正比；
- 存储的元素数是动态的，而 vector 类负责管理内存。

vector 是一种动态数组，其结构如图 17.1 所示。

图 17.1 vector 的内部构造

提示

要使用 std::vector 类，需要包含下面的头文件：

```
#include <vector>
```

17.2 典型的 vector 操作

std::vector 类的行为规范和公有成员是由 C++标准定义的，因此，遵循该标准的所有 C++编程平台都支持本章将介绍的 vector 操作。

17.2.1 实例化 vector

vector 是一个模板类，需要使用第 14 章介绍的方法进行实例化。要实例化 vector，需要指定要在该动态数组中存储的对象类型：

```
std::vector<int> vecDynamicIntegerArray;  // vector containing integers
std::vector<float> vecDynamicFloatArray;  // vector containing floats
std::vector<Tuna> vecDynamicTunaArray;  // vector containing Tunas
```

要声明指向 list 中元素的迭代器，可以这样做：

```
std::list<int>::const_iterator iElementInSet;
```

如果需要可用于修改值或调用非 const 函数的迭代器，可使用 iterator 代替 const_iterator。

鉴于 std::vector 有多个重载的构造函数，您可在实例化 vector 时指定它开始应包含的元素数以及这些元素的初始值，还可使用 vector 的一部分来实例化另一个 vector。

程序清单 17.1 演示了几种实例化 vector 的方式。

程序清单 17.1 各种实例化 std::vector 的方式：指定长度和初始值以及复制另一个 vector 中的值

```
 0: #include <vector>
 1:
 2: int main ()
 3: {
 4:   std::vector <int> vecIntegers;
 5:
 6:    // Instantiate a vector with 10 elements (it can grow larger)
 7:    std::vector <int> vecWithTenElements (10);
 8:
 9:    // Instantiate a vector with 10 elements, each initialized to 90
10:    std::vector <int> vecWithTenInitializedElements (10, 90);
11:
12:    // Instantiate one vector and initialize it to the contents of another
13:    std::vector <int> vecArrayCopy (vecWithTenInitializedElements);
14:
15:    // Using iterators instantiate vector to 5 elements from another
16:    std::vector <int> vecSomeElementsCopied ( vecWithTenElements.cbegin ()
17:                            , vecWithTenElements.cbegin () + 5 );
18:
19:    return 0;
20: }
```

▼ 分析：

上述代码演示了如何为整型具体化 vector 类，即实例化一个存储整型数据的 vector。该 vector 名为 vecIntegers，它使用了默认构造函数。在不知道容器最小需要多大，即不知道要存储多少个整数时，默认构造函数很有用。实例化 vector 的第 2 种和第 3 种方式如第 10 行和第 13 行所示，在这里，程序员知道 vector 至少应包含 10 个元素。注意，这并没有限制容器最终的大小，而只是设置了初始大小。第 4 种形式如第 16 行和第 18 行所示，它使用一个 vector 实例化另一个 vector 的内容，即复制 vector 对象或其一部分。这是所有 STL 容器都支持的构造函数。最后一种形式是使用迭代器。vecSomeElementCopied 包含 vecWithTenElements 的前 5 个元素。

注意

第 4 个构造函数只能用于类型类似的对象，因此可使用一个包含整型对象的 vector 来实例化 vecArrayCopy——另一个整型 vector，但如果其中一个 vector 包含的对象类型为 float，代码将不能通过编译。

提示

cbegin()和 cend()是否导致编译错误？

如果您使用的编译器没有遵循 C++11 标准，请使用 begin()和 end()分别代替该程序中的 cbegin()和 cend()。

cbegin()和 cend()的不同之处（优点）在于，它们返回一个迭代器，但较老的编译器不支持它们。

17.2.2 使用 push_back()在末尾插入元素

实例化一个整型 vector 后，接下来需要在 vector 中插入元素（整数）。在 vector 中插入元素时，元素将插入到数组末尾，这是使用成员方法 push_back 完成的：

```
vector <int> vecIntegers;  // declare a vector of type int

// Insert sample integers into the vector:
vecIntegers.push_back (50);
vecIntegers.push_back (1);
```

程序清单 17.2 演示了如何使用 push_back()在 std::vector 中动态地添加元素。

程序清单 17.2 使用 push_back 在 vector 中插入元素

```
 0: #include <iostream>
 1: #include <vector>
 2: using namespace std;
 3:
 4: int main ()
 5: {
 6:    vector <int> vecIntegers;
 7:
 8:    // Insert sample integers into the vector:
 9:    vecIntegers.push_back (50);
10:    vecIntegers.push_back (1);
11:    vecIntegers.push_back (987);
12:    vecIntegers.push_back (1001);
13:
14:    cout << "The vector contains ";
15:    cout << vecIntegers.size () << " Elements" << endl;
16:
17:    return 0;
18: }
```

▼ 输出：

```
The vector contains 4 Elements
```

▼ 分析：

第 9～12 行的 push_back 是 vector 类的一个公有成员方法，用于在动态数组末尾插入对象。请注意函数 size ()的用法，它返回 vector 中存储的元素数。

C++11

初始化列表

C++11 通过 std::initialize_list<>支持初始化列表，让您能够像处理静态数组那样，在实例化 vector 的同时初始化其元素：

```
vector<int> vecIntegers = {50, 1, 987, 1001};
// alternatively:
vector<int> vecMoreIntegers {50, 1, 987, 1001};
```

如果在程序清单 17.2 中使用这种语法，可节省 3 行代码，但这里没有这样做，因为编写本书时，Microsoft Visual C++ 2010 编译器的 std::vector 实现不支持初始化列表。

17.2.3 使用 insert()在指定位置插入元素

push_back 在 vector 末尾插入元素。如果要在中间插入元素，该如何办呢？很多 STL 容器（包括 std::vector）都包含 insert()函数，且有多个重载版本。

其中一个版本让您能够指定插入位置：

```
// insert an element at the beginning
vecIntegers.insert (vecIntegers.begin (), 25);
```

另一个版本让您能够指定插入位置、要插入的元素数以及这些元素的值（都相同）：

```
// Insert 2 numbers of value 45 at the end
vecIntegers.insert (vecIntegers.end (), 2, 45);
```

还可将另一个 vector 的内容插入到指定位置：

```
// Another vector containing 2 elements of value 30
vector <int> vecAnother (2, 30);

// Insert two elements from another container in position [1]
vecIntegers.insert (vecIntegers.begin () + 1,
                    vecAnother.begin (), vecAnother.end ());
```

可使用迭代器（通常是由 begin()或 end()返回的）告诉 insert()您想将新元素插入到什么位置。

提示

> 也可将该迭代器设置为 STL 算法（如 std::find()函数）的返回值。std::find()可用于查找元素，然后在这个位置插入另一个元素（这将导致查找的元素向后移）。

程序清单 17.3 演示了 vector::insert()的各种重载版本。

程序清单 17.3 使用函数 vector::insert 在指定位置插入元素

```
 0: #include <vector>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: void DisplayVector(const vector<int>& vecInput)
 5: {
 6:    for (auto iElement = vecInput.cbegin() // auto and cbegin(): C++11
 7:         ; iElement != vecInput.cend()  // cend() is new in C++11
 8:         ; ++ iElement )
 9:       cout << *iElement << ' ';
10:
11:    cout << endl;
12: }
13:
14: int main ()
15: {
16:    // Instantiate a vector with 4 elements, each initialized to 90
17:    vector <int> vecIntegers (4, 90);
18:
19:    cout << "The initial contents of the vector: ";
20:    DisplayVector(vecIntegers);
21:
22:    // Insert 25 at the beginning
23:    vecIntegers.insert (vecIntegers.begin (), 25);
24:
25:    // Insert 2 numbers of value 45 at the end
26:    vecIntegers.insert (vecIntegers.end (), 2, 45);
27:
28:    cout << "Vector after inserting elements at beginning and end: ";
29:    DisplayVector(vecIntegers);
```

```
30:
31:    // Another vector containing 2 elements of value 30
32:    vector <int> vecAnother (2, 30);
33:
34:    // Insert two elements from another container in position [1]
35:    vecIntegers.insert (vecIntegers.begin () + 1,
36:       vecAnother.begin (), vecAnother.end ());
37:
38:    cout << "Vector after inserting contents from another vector: ";
39:    cout << "in the middle:" << endl;
40:    DisplayVector(vecIntegers);
41:
42:    return 0;
43: }
```

▼ 输出：

```
The initial contents of the vector: 90 90 90 90
Vector after inserting elements at beginning and end: 25 90 90 90 90 45 45
Vector after inserting contents from another vector: in the middle:
25 30 30 90 90 90 90 45 45
```

▼ 分析：

上述代码演示了 insert 函数的强大功能，它让您能够将值插入到容器中间。第 17 行的 vector 包含 4 个元素，每个元素都被初始化为 90；然后，使用了成员函数 vector::insert 的各种重载版本。第 23 行在开头添加了一个元素；第 26 行在末尾添加了两个元素，它们的值都是 45。第 35 行演示了如何将一个 vector 的元素插入到另一个 vector 中间（这里是第一个元素后面，偏移量为 1）。

虽然函数 vector::insert 功能众多，但给 vector 添加元素时，应首选 push_back()。

请注意，将元素插入 vector 时，insert()可能是效率最低的（插入位置不是末尾时），因为在开头或中间插入元素时，将导致 vector 类将后面的所有元素后移（为要插入的元素腾出空间）。根据容器中包含的对象类型，这种移动操作可能需要调用复制构造函数或赋值运算符，因此开销可能很大。在上述例子中，vector 包含的是 int 对象，移动开销不是很大。但在其他情况下，情况可能并非如此。

提示

如果需要频繁地在容器中间插入元素，应选择使用第 18 章将介绍的 std::list。

警告

您使用的 C++编译器是否较老

在程序清单 17.3 中，函数 DisplayVector()使用了 C++11 关键字 auto 来声明迭代器的类型，如第 6 行所示。对于这个示例以及后面的示例，如果要使用非 C++11 编译器编译它们，需要将 auto 替换为显式类型，这里为 vector<int>::const_iterator。

因此，如果您使用的是较老的编译器，需要将 DisplayVector()修改成下面这样：

```
// for older C++ compilers
void DisplayVector(const vector<int>& vecInput)
{
   for (vector<int>::const_iterator iElement = vecInput.begin
  ()
        ; iElement != vecInput.end ()
        ; ++ iElement )
       cout << *iElement << ' ';

   cout << endl;
}
```

17.2.4 使用数组语法访问 vector 中的元素

可使用下列方法访问 vector 的元素：使用下标运算符（[]）以数组语法方式访问；使用成员函数 at()；使用迭代器。

程序清单 17.1 演示了如何创建一个包含 10 个元素的 vector 实例：

```
std::vector <int> vecArrayWithTenElements (10);
```

可使用类似于数组的语法访问并设置各个元素：

```
vecArrayWithTenElements[3] = 2011; // set 4th element
```

程序清单 17.4 演示了如何使用下标运算符（[]）访问元素。

程序清单 17.4 使用数组语法访问 vector 中的元素

```
 0: #include <iostream>
 1: #include <vector>
 2:
 3: int main ()
 4: {
 5:    using namespace std;
 6:    vector <int> vecIntegerArray;
 7:
 8:    // Insert sample integers into the vector:
 9:    vecIntegerArray.push_back (50);
10:    vecIntegerArray.push_back (1);
11:    vecIntegerArray.push_back (987);
12:    vecIntegerArray.push_back (1001);
13:
14:    for (size_t Index = 0; Index < vecIntegerArray.size (); ++Index)
15:    {
16:       cout << "Element[" << Index << "] = " ;
17:       cout << vecIntegerArray[Index] << endl;
18:    }
19:
20:    // changing 3rd integer from 987 to 2011
21:    vecIntegerArray[2] = 2011;
22:    cout << "After replacement: " << endl;
23:    cout << "Element[2] = " << vecIntegerArray[2] << endl;
24:
25:    return 0;
26: }
```

▼ 输出：

```
Element[0] = 50
Element[1] = 1
Element[2] = 987
Element[3] = 1001
After replacement:
Element[2] = 2011
```

▼ 分析：

在第 17、21 和 23 行，像使用静态数组那样，使用下标运算符（[]）访问并设置了 vector 的元素。下标运算符接受一个从零开始的元素索引，与静态数组一样。注意到第 15 行的 for 循环将索引与 vector::size()进行比较，确保它跨越 vector 的边界。

警告

使用[]访问 vector 的元素时，面临的风险与访问数组元素相同，即不能超出容器的边界。使用下标运算符（[ ]）访问 vector 的元素时，如果指定的位置超出了边界，结果将是不确定的（什么情况都可能发生，很可能是访问违规）。

更安全的方法是使用成员函数 at():

```
// gets element at position 2
cout << vecIntegerArray.at (2);
// the vector::at() version of the code above in Listing
  17.4, line 17:
cout << vecIntegerArray.at(Index);
```

at()函数在运行阶段检查容器的大小，如果索引超出边界（无论如何都不能这样做），将引发异常。

下标运算符（[ ]）只有在保证边界完整性的情况下才是安全的，如前一个例子所示

17.2.5 使用指针语法访问 vector 中的元素

也可使用迭代器以类似于指针的语法访问 vector 中的元素，如程序清单 17.5 所示。

程序清单 17.5 使用指针语法（迭代器）访问 vector 中的元素

```
#include <iostream>
#include <vector>

int main ()
{
   using namespace std;
   vector <int> vecIntegers;

   // Insert sample integers into the vector:
   vecIntegers.push_back (50);
   vecIntegers.push_back (1);
   vecIntegers.push_back (987);
   vecIntegers.push_back (1001);

   // Access objects in a vector using iterators:
   vector <int>::iterator iElementLocator = vecIntegers.begin ();
   // iterator declared using C++11 keyword auto (uncomment next line)
   // auto iElementLocator = vecIntegers.begin ();

   while (iElementLocator != vecIntegers.end ())
   {
      size_t Index = distance (vecIntegers.begin (),
                               iElementLocator);

      cout << "Element at position ";
      cout << Index << " is: " << *iElementLocator << endl;

      // move to the next element
      ++ iElementLocator;
   }

   return 0;
}
```

▼ 输出：

```
Element at position 0 is: 50
Element at position 1 is: 1
```

```
Element at position 2 is: 987
Element at position 3 is: 1001
```

▼ 分析：

在这个例子中，迭代器有点像指针，迭代器的用法很像指针算术运算，如第 25 和 29 行所示。在第 25 行，使用了解除引用运算符（*）来访问存储在 vector 中的值，而第 29 行使用了运算符++递增迭代器，使其指向下一个元素。第 21 行使用了 std::distance 来计算元素的偏移量（相对于开头的位置），这是根据 begin()和指向元素的迭代器计算得到的。

17.2.6 删除 vector 中的元素

除支持使用 push_back 方法在末尾插入元素外，vector 还支持使用 pop_back 函数将末尾的元素删除。使用 pop_back 将元素从 vector 中删除所需的时间是固定的，即不随 vector 存储的元素个数而异。程序清单 17.6 演示了如何使用函数 pop_back 删除 vector 末尾的元素。

程序清单 17.6 使用 pop_back 删除最后一个元素

```
 0: #include <iostream>
 1: #include <vector>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayVector(const vector<T>& vecInput)
 6: {
 7:    for (auto iElement = vecInput.cbegin() // auto and cbegin(): C++11
 8:        ; iElement != Input.cend()  // cend() is new in C++11
 9:        ; ++ iElement )
10:      cout << *iElement << ' ';
11:
12:    cout << endl;
13: }
14:
15: int main ()
16: {
17:    vector <int> vecIntegers;
18:
19:    // Insert sample integers into the vector:
20:    vecIntegers.push_back (50);
21:    vecIntegers.push_back (1);
22:    vecIntegers.push_back (987);
23:    vecIntegers.push_back (1001);
24:
25:    cout << "Vector contains " << vecIntegers.size () << " elements: ";
26:    DisplayVector(vecIntegers);
27:
28:    // Erase one element at the end
29:    vecIntegers.pop_back ();
30:
31:    cout << "After a call to pop_back()" << endl;
32:    cout << "Vector contains " << vecIntegers.size () << " elements: ";
33:    DisplayVector(vecIntegers);
34:
35:    return 0;
36: }
```

▼ 输出：

```
Vector contains 4 elements: 50 1 987 1001
After a call to pop_back()
Vector contains 3 elements: 50 1 987
```

▼ 分析：

上述输出表明，第29行的pop_back函数将vector的最后一个元素删除，从而减少了vector包含的元素数。第32行再次调用size()，以证明vector包含的元素少了一个，如输出所示。

注意

在程序清单17.3中，DisplayVector()只能接受整型vector作为参数，而在程序清单17.6中，它是个模板函数（第4-13行）。这有助于将该模板函数重用于浮点型vector:

```
vector <float> vecFloats;
DisplayVector(vecFloats); // works, as this is a generic
  function
```

该模板函数接受任何类型的vector作为参数，只要该类型支持运算符*，且其返回值可被cout理解。

17.3 理解大小和容量

vector的大小指的是实际存储的元素数，而vector的容量指的是在重新分配内存以存储更多元素前vector能够存储的元素数。因此，vector的大小小于或等于容量。

要查询vector当前存储的元素数，可调用size()：

```
cout << "Size: " << vecIntegers.size ();
```

要查询vector的容量，可调用capacity()：

```
cout << "Capacity: " <<  vecIntegers.capacity () << endl;
```

如果vector需要频繁地给其内部动态数组重新分配内存，将对性能造成一定的影响。在很大程度上说，这种问题可以通过使用成员函数reserve (number)来解决。reserve函数的功能基本上是增加分配给内部数组的内存，以免频繁地重新分配内存。通过减少重新分配内存的次数，还可减少复制对象的时间，从而提高性能，这取决于存储在vector中的对象类型。程序清单17.7说明了size()和capacity()之间的区别。

程序清单17.7 演示size()和capacity()

```
 0: #include <iostream>
 1: #include <vector>
 2:
 3: int main ()
 4: {
 5:    using namespace std;
 6:
 7:    // Instantiate a vector object that holds 5 integers of default value
 8:    vector <int> vecIntegers (5);
 9:
10:    cout << "Vector of integers was instantiated with " << endl;
11:    cout << "Size: " << vecIntegers.size ();
12:    cout << ", Capacity: " <<  vecIntegers.capacity () << endl;
13:
14:    // Inserting a 6th element in to the vector
15:    vecIntegers.push_back (666);
16:
17:    cout << "After inserting an additional element... " << endl;
18:    cout << "Size: " << vecIntegers.size ();
19:    cout << ", Capacity: " <<  vecIntegers.capacity () << endl;
```

```
20:
21:     // Inserting another element
22:     vecIntegers.push_back (777);
23:
24:     cout << "After inserting yet another element... " << endl;
25:     cout << "Size: " << vecIntegers.size ();
26:     cout << ", Capacity: " <<  vecIntegers.capacity () << endl;
27:
28:     return 0;
29: }
```

▼ **输出：**

```
Vector of integers was instantiated with
Size: 5, Capacity: 5
After inserting an additional element...
Size: 6, Capacity: 7
After inserting yet another element...
Size: 7, Capacity: 7
```

▼ **分析：**

第 8 行实例化了一个包含 5 个整型对象的 vector，这些整型对象使用默认值 0。第 11 行和第 12 行分别显示 vector 的大小和容量，它们在实例化后相等。第 15 行在 vector 中插入了第 6 个元素。鉴于在插入前 vector 的容量为 5，因此 vector 的内部缓冲区没有足够的内存来存储第 6 个元素。换句话说，vector 为扩大其容量以存储 6 个元素，需要重新分配内部缓冲区。重新分配的逻辑实现是智能的：为避免插入下一个元素时再次重新分配，提前分配了比当前需求更大的容量。

从输出可知，在容量为 5 的 vector 中插入第 6 个元素时，将容量增大到了 7。size()总是指出 vector 存储的元素数，当前其值为 6。第 22 行插入了第 7 个元素，这次没有扩大容量，因为已分配的内存足以满足需求。这时大小和容量相等，这表明 vector 的容量已经用完，再次插入元素将导致 vector 重新分配其内部缓冲区、复制现有的元素再插入新值。

注意

在重新分配 vector 内部缓冲区时提前增加容量方面，C++标准没有做任何规定，这取决于使用的 STL 实现。

17.4 STL deque 类

deque 是一个 STL 动态数组类，与 vector 非常类似，但支持在数组开头和末尾插入或删除元素。要实例化一个整型 deque，可以像下面这样做：

```
// Define a deque of integers
deque <int> dqIntegers;
```

提示

要使用 std::deque，需要包含头文件<deque>：

```
#include<deque>
```

deque 的内部结构如图 17.2 所示。

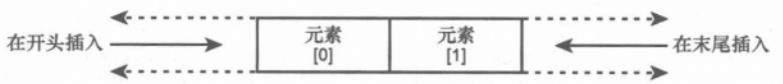


图 17.2 deque 的内部结构

deque 与 vector 极其相似，也支持使用方法 push_back()和 pop_back()在末尾插入和删除元素。与 vector 一样，deque 也使用运算符[]以数组语法访问其元素。deque 与 vector 的不同之处在于，它还允许

您使用 push_front 和 pop_front 在开头插入和删除元素，如程序清单 17.8 所示。

程序清单 17.8 实例化一个 STL deque，并使用 push_front 和 pop_front 在开头插入和删除元素

```
 0: #include <deque>
 1: #include <iostream>
 2: #include <algorithm>
 3:
 4: int main ()
 5: {
 6:    using namespace std;
 7:
 8:    // Define a deque of integers
 9:    deque <int> dqIntegers;
10:
11:    // Insert integers at the bottom of the array
12:    dqIntegers.push_back (3);
13:    dqIntegers.push_back (4);
14:    dqIntegers.push_back (5);
15:
16:    // Insert integers at the top of the array
17:    dqIntegers.push_front (2);
18:    dqIntegers.push_front (1);
19:    dqIntegers.push_front (0);
20:
21:    cout << "The contents of the deque after inserting elements ";
22:    cout << "at the top and bottom are:" << endl;
23:
24:    // Display contents on the screen
25:    for ( size_t nCount = 0
26:       ; nCount < dqIntegers.size ()
27:       ; ++ nCount )
28:    {
29:       cout << "Element [" << nCount << "] = ";
30:       cout << dqIntegers [nCount] << endl;
31:    }
32:
33:    cout << endl;
34:
35:    // Erase an element at the top
36:    dqIntegers.pop_front ();
37:
38:    // Erase an element at the bottom
39:    dqIntegers.pop_back ();
40:
41:    cout << "The contents of the deque after erasing an element ";
42:    cout << "from the top and bottom are:" << endl;
43:
44:    // Display contents again: this time using iterators
45:    // if on older compilers, remove auto and uncomment next line
46:    // deque <int>::iterator iElementLocator;
47:    for (auto iElementLocator = dqIntegers.begin ()
48:       ; iElementLocator != dqIntegers.end ()
49:       ; ++ iElementLocator )
50:    {
51:       size_t Offset = distance (dqIntegers.begin (), iElementLocator);
52:       cout << "Element [" << Offset << "] = " << *iElementLocator << endl;
53:    }
54:
55:    return 0;
56: }
```

▼ 输出：

```
The contents of the deque after inserting elements at the top and bottom are:
Element [0] = 0
Element [1] = 1
Element [2] = 2
Element [3] = 3
Element [4] = 4
Element [5] = 5

The contents of the deque after erasing an element from the top and bottom are:
Element [0] = 1
Element [1] = 2
Element [2] = 3
Element [3] = 4
```

▼ 分析：

第 9 行实例化了一个整型 deque，其语法与实例化整型 vector 极其相似。第 12～14 行演示了 deque 的成员函数 push_back 的用法，然后第 17～19 行演示了 push_front 的用法，push_front 是 deque 唯一不同于 vector 的地方。pop_front 的用法类似，如第 36 行所示。要显示 deque 的内容，第一种方法是使用数组语法访问其元素，第二种方法是使用迭代器。使用迭代器时（如第 47～53 行所示），使用算法 std::distance 计算元素的偏移位置，这与程序清单 17.5 中处理 vector 时相同。

应该	不应该
在不知道需要存储多少个元素时，务必使用动态数组 vector 或 deque。 请牢记，vector 只能在一端扩容，为此可使用方法 push_back()。 请牢记，deque 可在两端扩容，为此可使用方法 push_back()和 push_front()。	别忘了，方法 pop_back()删除集合中的最后一个元素。 别忘了，方法 pop_front()删除 deque 的第一个元素。 访问动态数组时，不要跨越其边界。

17.5 总结

本章介绍了将 vector 和 deque 用作动态数组的基本知识，还解释了大小与容量的概念。通过对 vector 进行优化，减少了重新分配内部缓冲区的次数。重新分配缓冲区时，需要复制容器包含的对象，这可能降低性能。vector 是最简单的 STL 容器，也是最常用和最高效的。

17.6 问与答

问：vector 会改变其存储的元素的顺序吗？
答：vector 是一种顺序容器，元素的存储顺序与插入顺序相同。

问：要将元素插入到 vector 中，应使用哪个函数？元素将插入到 vector 的什么位置？
答：成员函数 push_back 将元素插入到 vector 末尾。

问：哪个函数用于获悉存储在 vector 中的元素个数？
答：成员函数 size ()返回存储在 vector 中的元素个数。对于所有 STL 容器，该函数都如此。

问：随着vector包含的元素增多，在vector末尾插入或删除元素所需的时间是否更长？

答：否。在vector末尾插入或删除元素所需的时间是固定的。

问：使用成员函数reserve的优点是什么？

答：reserve (...)为vector的内部缓冲区分配内存空间，这样在插入元素时vector就不需要重新分配缓冲区并复制现有内容。根据vector存储的对象类型，为vector预留内存空间可能改善性能。

问：在插入元素方面，deque与vector是否不同？

答：没有。在插入元素方面，deque的特点与vector类似。将元素插入到末尾时，两者所需的时间都是固定的，而将元素插入到中间时，所需的时间与容器包含的元素数成正比。然而，vector只允许在末尾插入，而deque允许在开头和末尾插入。

17.7 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录D的答案。在继续学习下一章前，请务必弄懂这些答案。

17.7.1 测验

1. 在vector的开头或中间插入元素时，所需的时间是否是固定的？
2. 有一个vector，对其调用函数size()和capacity()时分别返回10和20。还可再插入多少个元素而不会导致vector重新分配其缓冲区？
3. pop_back函数有何功能？
4. 如果vector <int>是一个整型动态数组，那vector <CMammal>是什么类型的动态数组？
5. 能否随机访问vector中的元素？如果是，如何访问？
6. 哪种迭代器可用于随机访问vector中的元素？

17.7.2 练习

1. 编写一个交互式程序，它接受用户输入的整数并将其存储到vector中。用户应能够随时使用索引查询vector中存储的值。
2. 对练习1中的程序进行扩展，使其能够告诉用户他查询的值是否在vector中。
3. Jack在eBay销售广口瓶。为帮助他打包和发货，请编写一个程序，让他能够输入每件商品的尺寸，将其存储在vector中再显示到屏幕上。

第 18 章
STL list 和 forward_list

标准模板库（STL）以模板类 std::list 的方式向程序员提供了一个双向链表。双向链表的主要优点是，插入和删除元素的速度快，且时间是固定的。从 C++11 起，您还可使用单向链表 std::forward_list，这种链表只能沿一个方向遍历。

在本章中，您将学习：

- 如何实例化 list 和 forward_list；
- 使用 STL 链表类，包括插入和删除元素；
- 如何对元素进行反转和排序。

18.1 std::list 的特点

链表是一系列节点，其中每个节点除包含对象或值外还指向下一个节点，即每个节点都链接到下一个节点和前一个节点，如图 18.1 所示。

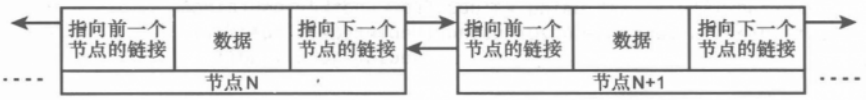


图 18.1 双向链表的可视化表示

list 类的 STL 实现允许在开头、末尾和中间插入元素，且所需的时间固定。

提示

要使用 std::list 类，需要包含头文件<list>：

```
#include <list>
```

18.2 基本的 list 操作

要使用遵循标准的 STL list 类，必须包含头文件<list>。std 命名空间中的模板类 list 是一种泛型实现，要使用其成员函数，必须实例化该模板。

18.2.1 实例化 std::list 对象

要实例化模板 list，需要指定要在其中存储的对象类型，因此实例化 list 的语法类似于下面这样：

```
std::list<int> listIntegers;  // list containing integers
std::list<float> listFloats;  // list containing floats
std::list<Tuna> listTunas;  // list containing objects of type Tuna
```

要声明一个指向list中元素的迭代器，可以像下面这样做：

```
std::list<int>::const_iterator iElementInSet;
```

如果需要一个这样的迭代器，即可以使用它来修改值或调用非const函数，可将const_iterator替换为iterator。

鉴于std::list的实现提供了一组重载的构造函数，您可以创建包含指定元素数的list，并初始化每个元素，如程序清单18.1所示。

程序清单18.1 各种实例化std::list的方式：指定元素数和初始值

```
 0: #include <list>
 1: #include <vector>
 2:
 3: int main ()
 4: {
 5:    using namespace std;
 6:
 7:    // instantiate an empty list
 8:    list <int> listIntegers;
 9:
10:    // instantiate a list with 10 integers
11:    list<int> listWith10Integers(10);
12:
13:    // instantiate a list with 4 integers, each initialized to 99
14:    list<int> listWith4IntegerEach99 (10, 99);
15:
16:    // create an exact copy of an existing list
17:    list<int> listCopyAnother(listWith4IntegerEach99);
18:
19:    // a vector with 10 integers, each 2011
20:    vector<int> vecIntegers(10, 2011);
21:
22:    // instantiate a list using values from another container
23:    list<int> listContainsCopyOfAnother(vecIntegers.cbegin(),
24:                                        vecIntegers.cend());
25:
26:    return 0;
27: }
```

▼ 分析：

这个程序没有输出，它演示了如何使用各种重载的构造函数来创建整型list。第8行创建了一个空list；第11行创建了一个包含10个整型元素的list；第14行创建了一个名为listWith4IntegersEach99的list，它包含10个整型元素，且每个元素都被初始化为99；第17行演示了如何创建一个这样的list，即其内容与另一个list完全相同。第20～24行令人惊讶！您首先实例化了一个vector，它包含10个整型元素，每个元素都被初始化为2011；接下来，第23行实例化了一个list，它包含从vector复制而来的元素，这是使用C++11新增的vector::cbegin()和vector::cend()返回的const迭代器复制的。该程序清单表明，迭代器让容器的实现彼此独立，其通用功能让您能够使用vector中的值实例化list，如第23和24行所示。

提示

> cbegin()和cend()是否导致编译错误？
>
> 如果您使用的编译器没有遵循C++11标准，请使用begin()和end()分别代替该程序中的cbegin()和cend()。cbegin()和cend()是C++11新增的，它们返回一个const迭代器，不能用于修改元素。

注意 如果将程序清单 18.1 与程序清单 17.1 进行比较，将发现实例化不同容器的方式类似。随着您越来越多地使用 STL 容器进行编程，这种模式将越来越明显，越来越容易理解。

18.2.2 在 list 开头或末尾插入元素

与 deque 类似，要在 list 开头插入元素，可使用其成员方法 push_front。要在末尾插入，可使用成员方法 push_back。这两个方法都接受一个参数，即要插入的值：

```
listIntegers.push_back (-1);
listIntegers.push_front (2001);
```

程序清单 18.2 演示了这两个方法对整型 list 的影响。

程序清单 18.2 使用 push_front 和 push_back 在 list 中插入元素

```
 0: #include <list>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents (const T& Input)
 6: {
 7:    for (auto iElement = Input.cbegin() // auto and cbegin: C++11
 8:        ; iElement != Input.cend()
 9:        ; ++ iElement )
10:        cout << *iElement << ' ';
11:
12:    cout << endl;
13: }
14:
15: int main ()
16: {
17:    std::list <int> listIntegers;
18:
19:    listIntegers.push_front (10);
20:    listIntegers.push_front (2011);
21:    listIntegers.push_back (-1);
22:    listIntegers.push_back (9999);
23:
24:    DisplayContents(listIntegers);
25:
26:    return 0;
27: }
```

▼ 输出：

```
2011 10 -1 9999
```

▼ 分析：

第 19～22 行演示了如何使用 push_front 和 push_back。以实参方式提供给 push_front 的值被插入到 list 开头，而传递给 push_back 的值被插入到 list 末尾。使用模板函数 DisplayContents()显示了 list 的内容，以显示插入的元素的排列顺序（它们并非以插入顺序存储）。

警告

关键字 auto 是否导致编译错误？

在程序清单 18.2 中，函数 DisplayContents()使用了 C++11 关键字 auto 来声明迭代器的类型，如第 7 行所示。另外，它还使用了 cbegin()和 cend()，这些函数是 C++11 新增的，返回一个 const_iterator。对于这个示例以及以后的示例，如果要使用不遵循 C++11 的编译器编译，需要将 auto 替换为显式类型。

因此，如果您使用的是较老的编译器，需要将 DisplayContents()修改成下面这样：

```
template <typename T>
void DisplayContents (const T& Input)
{
   for (T::const_iterator iElement = Input.begin ()
       ; iElement != Input.end ()
       ; ++ iElement )
       cout << *iElement << ' ';

   cout << endl;
}
```

注意

程序清单 18.2 中的 DisplayContents()（第 4～13 行）比程序清单 17.6 中的 DisplayVector()更通用（注意到参数列表不同）。DisplayVector()可用于任何 vector，而不管其存储的元素类型如何，而 DisplayContents()可用于任何容器。

调用程序 18.2 中的 DisplayContents()时，如果实参设置为 vector 或 list，该函数也将正确运行。

18.2.3 在 list 中间插入元素

std::list 的特点之一是，在其中间插入元素所需的时间是固定的，这项工作是由成员函数 insert 完成的。成员函数 list::insert 有 3 种版本。

- **第 1 种版本：**

```
iterator insert(iterator pos, const T& x)
```

在这里，insert 函数接受的第 1 个参数是插入位置，第 2 个参数是要插入的值。该函数返回一个迭代器，它指向刚插入到 list 中的元素。

- **第 2 种版本：**

```
void insert(iterator pos, size_type n, const T& x)
```

该函数的第 1 个参数是插入位置，最后一个参数是要插入的值，而第 2 个参数是要插入的元素个数。

- **第 3 种版本：**

```
template <class InputIterator>
void insert(iterator pos, InputIterator f, InputIterator l)
```

该重载版本是一个模板函数，除一个位置参数外，它还接受两个输入迭代器，指定要将集合中相应范围内的元素插入到 list 中。注意，输入类型 InputIterator 是一种模板参数化类型，因此可指定任何集合（数组、vector 或另一个 list）的边界。

程序清单 18.3 演示了如何使用函数 list::insert 的这些重载版本。

程序清单 18.3 在 list 中插入元素的各种方法

```
0: #include <list>
1: #include <iostream>
2: using namespace std;
3:
```

```
 4: template <typename T>
 5: void DisplayContents (const T& Input)
 6: {
 7:    for (auto iElement = Input.cbegin() // auto and cbegin: C++11
 8:         ; iElement != Input.cend()
 9:         ; ++ iElement )
10:       cout << *iElement << ' ';
11:
12:    cout << endl;
13: }
14:
15: int main ()
16: {
17:    list <int> listIntegers1;
18:
19:    // Inserting elements at the beginning...
20:    listIntegers1.insert (listIntegers1.begin (), 2);
21:    listIntegers1.insert (listIntegers1.begin (), 1);
22:
23:    // Inserting an element at the end...
24:    listIntegers1.insert (listIntegers1.end (), 3);
25:
26:    cout << "The contents of list 1 after inserting elements:" << endl;
27:    DisplayContents (listIntegers1);
28:
29:    list <int> listIntegers2;
30:
31:    // Inserting 4 elements of the same value 0...
32:    listIntegers2.insert (listIntegers2.begin (), 4, 0);
33:
34:    cout << "The contents of list 2 after inserting '";
35:    cout << listIntegers2.size () << "' elements of a value:" << endl;
36:    DisplayContents (listIntegers2);
37:
38:    list <int> listIntegers3;
39:
40:    // Inserting elements from another list at the beginning...
41:    listIntegers3.insert (listIntegers3.begin (),
42:                          listIntegers1.begin (), listIntegers1.end ());
43:
44:    cout << "The contents of list 3 after inserting the contents of ";
45:    cout << "list 1 at the beginning:" << endl;
46:    DisplayContents (listIntegers3);
47:
48:    // Inserting elements from another list at the end...
49:    listIntegers3.insert (listIntegers3.end (),
50:                          listIntegers2.begin (), listIntegers2.end ());
51:
52:    cout << "The contents of list 3 after inserting ";
53:    cout << "the contents of list 2 at the beginning:" << endl;
54:    DisplayContents (listIntegers3);
55:
56:    return 0;
57: }
```

▼ 输出：

```
The contents of list 1 after inserting elements:
1 2 3
The contents of list 2 after inserting '4' elements of a value:
0 0 0 0
```

```
The contents of list 3 after inserting the contents of list 1 at the beginning:
1 2 3
The contents of list 3 after inserting the contents of list 2 at the beginning:
1 2 3 0 0 0 0
```

▼ 分析：

在程序清单 18.3 中，begin()和 end()是 list 的成员函数，分别返回指向 list 开头和末尾的迭代器；几乎对所有 STL 容器来说都如此。函数 list::insert 接受一个迭代器参数，元素将插入到该参数指定的位置前面。end()函数返回的迭代器（如第 24 行所示）指向 list 中最后一个元素的后面，因此这行代码将 3 插入到末尾。第 32 行在一个 list 开头插入了 4 个元素，这些元素的值都为 0。第 41 行和第 42 行演示了如何将一个 list 的内容插入到另一个 list 开头。虽然这里演示的是将一个整型 list 插入到另一个 list 中，但也可将插入范围指定为 vector 的边界（像程序清单 18.1 那样使用 begin()和 end()）或普通静态数组的边界。

18.2.4 删除 list 中的元素

list 的成员函数 erase 有两种重载版本：一个接受一个迭代器参数并删除迭代器指向的元素，另一个接受两个迭代器参数并删除指定范围内的所有元素。程序清单 18.4 演示了如何使用 list::eras 函数删除一个元素或指定范围内的所有元素。

程序清单 18.4 删除 list 中的元素

```
 0: #include <list>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents(const T& Input)
 6: {
 7:    for(auto iElement = Input.cbegin() // auto and cbegin: C++11
 8:       ; iElement != Input.cend()
 9:       ; ++ iElement )
10:      cout << *iElement << ' ';
11:
12:    cout << endl;
13: }
14:
15: int main()
16: {
17:    std::list <int> listIntegers;
18:
19:    // Insert elements at the beginning and end...
20:    listIntegers.push_back(4);
21:    listIntegers.push_front(3);
22:    listIntegers.push_back(5);
23:
24:    // Store an iterator obtained in using the 'insert' function
25:    auto iValue2 = listIntegers.insert(listIntegers.begin(), 2);
26:
27:    cout << "Initial contents of the list:" << endl;
28:    DisplayContents(listIntegers);
29:
30:    listIntegers.erase(listIntegers.begin(), iValue2);
31:    cout << "Contents after erasing a range of elements:" << endl;
32:    DisplayContents(listIntegers);
33:
```

```
34:    cout << "After erasing element '"<< *iValue2 << "':" << endl;
35:    listIntegers.erase(iValue2);
36:    DisplayContents(listIntegers);
37:
38:    listIntegers.erase(listIntegers.begin(), listIntegers.end());
39:    cout << "Number of elements after erasing range: ";
40:    cout << listIntegers.size() << endl;
41:
42:    return 0;
43: }
```

▼ **输出：**

```
Initial contents of the list:
2 3 4 5
Contents after erasing a range of elements:
2 3 4 5
After erasing element '2':
3 4 5
Number of elements after erasing range: 0
```

▼ **分析：**

在 main()中，第 20～25 行使用了各种方式在 list 中插入元素。用于插入一个元素时，insert()返回一个迭代器，该迭代器指向新插入的元素。在第 25 行，将指向值为 2 的元素的迭代器存储到了变量 iValue2 中，以便第 35 行使用它来调用 erease()，从而将该元素从 list 中删除。第 30 和 38 行演示了如何使用 erease()来删除指定范围内的元素。在第 30 行，删除了从 begin()到值为 2 的元素之间的所有元素（不包括值为 2 的元素）；而在第 38 行，删除了从 begin()到 end()之间的所有元素，这相当于清空整个 list。

注意

程序清单 18.4 的第 40 行表明，可使用方法 size()确定 std::list 的大小，就像 vector 一样。这种模式适用于所有 STL 容器类。

18.3 对 list 中的元素进行反转和排序

list 的一个独特之处是，指向元素的迭代器在 list 的元素重新排列或插入元素后仍有效。为实现这种特点，list 提供了成员方法 sort 和 reverse，虽然 STL 也提供了这两种算法，且这些算法也可用于 list 类。这些算法的成员函数版本确保元素的相对位置发生变化后指向元素的迭代器仍有效。

18.3.1 使用 list::reverse()反转元素的排列顺序

list 提供了成员函数 reverse()，该函数没有参数，它反转 list 中元素的排列顺序：

```
listIntegers.reverse();  // reverse order of elements
```

程序清单 18.5 演示了如何使用 reverse()。

程序清单 18.5 反转 list 中元素的排列顺序

```
0: #include <list>
1: #include <iostream>
2: using namespace std;
3:
4: template <typename T>
```

```
 5: void DisplayContents(const T& Input)
 6: {
 7:    for(auto iElement = Input.cbegin() // auto and cbegin: C++11
 8:      ; iElement != Input.cend()
 9:      ; ++ iElement )
10:      cout << *iElement << ' ';
11:
12:    cout << endl;
13: }
14:
15: int main()
16: {
17:    std::list <int> listIntegers;
18:
19:    // Insert elements at the beginning and end
20:    listIntegers.push_front(4);
21:    listIntegers.push_front(3);
22:    listIntegers.push_front(2);
23:    listIntegers.push_front(1);
24:    listIntegers.push_front(0);
25:    listIntegers.push_back(5);
26:
27:    cout << "Initial contents of the list:" << endl;
28:    DisplayContents(listIntegers);
29:
30:    listIntegers.reverse();
31:
32:    cout << "Contents of the list after using reverse():" << endl;
33:    DisplayContents(listIntegers);
34:
35:    return 0;
36: }
```

▼ 输出：

```
Initial contents of the list:
0 1 2 3 4 5
Contents of the list after using reverse():
5 4 3 2 1 0
```

▼ 分析：

如第 30 行所示，reverse()只是反转 list 中元素的排列顺序。它是一个没有参数的简单函数，确保指向元素的迭代器在反转后仍有效——如果程序员保存了该迭代器。

18.3.2 对元素进行排序

list 的成员函数 sort()有两个版本，其中一个没有参数：

```
listIntegers.sort();  // sort in ascending order
```

另一个接受一个二元谓词函数作为参数，让您能够指定排序标准：

```
bool SortPredicate_Descending (const int& lsh, const int& rsh)
{
   // define criteria for list::sort: return true for desired order
   return (lsh > rsh);
}
// Use predicate to sort a list:
listIntegers.sort (SortPredicate_Descending);
```

程序清单 18.6 演示了这两个版本。

程序清单 18.6 使用 list::sort()将整型 list 按升序和降序排列

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

bool SortPredicate_Descending (const int& lsh, const int& rsh)
{
   // define criteria for list::sort: return true for desired order
   return (lsh > rsh);
}

template <typename T>
void DisplayContents (const T& Input)
{
   for (auto iElement = Input.cbegin() // auto and cbegin: C++11
      ; iElement != Input.cend()
      ; ++ iElement )
      cout << *iElement << ' ';

   cout << endl;
}

int main ()
{
   list <int> listIntegers;

   // Insert elements at the beginning and end
   listIntegers.push_front (444);
   listIntegers.push_front (2011);
   listIntegers.push_front (-1);
   listIntegers.push_front (0);
   listIntegers.push_back (-5);

   cout << "Initial contents of the list are - " << endl;
   DisplayContents (listIntegers);

   listIntegers.sort ();

   cout << "Order of elements after sort():" << endl;
   DisplayContents (listIntegers);

   listIntegers.sort (SortPredicate_Descending);
   cout << "Order of elements after sort() with a predicate:" << endl;
   DisplayContents (listIntegers);

   return 0;
}
```

▼ 输出：

```
Initial contents of the list are -
0 -1 2011 444 -5
Order of elements after sort():
-5 -1 0 444 2011
Order of elements after sort() with a predicate:
2011 444 0 -1 -5
```

▼ 分析：

该示例演示了如何对整型 list 进行排序。开头几行代码创建了一个 list 对象，并在其中插入一些值。

第35行演示了不带参数的sort()函数的用法，它使用运算符<比较整数（就整型而言，该运算符是由编译器实现的），并将元素按默认的升序排列。然而，如果程序员要覆盖这种默认行为，必须向sort函数提供一个二元谓词。第4～8行定义了函数SortPredicate_Descending，它是一个二元谓词，帮助list的sort函数判断一个元素是否比另一个元素小。如果不是，则交换这两个元素的位置。换句话说，您告诉了list如何解释小于，就这里而言，小于的含义是第一个参数大于第二个参数。将这个谓词作为参数传递给了sort()函数，如第40行所示。这个谓词仅在第一个值比第二个值大时返回true。也就是说，使用该谓词时，仅当第一个元素（lsh）的数字值比第二个元素（rsh）大时，sort才认为第一个元素比第二个元素小。基于这种解释，sort交换元素的位置，以满足谓词指定的标准。

18.3.3 对包含对象的list进行排序以及删除其中的元素

如果list的元素类型为类，而不是int等简单内置类型，如何对其进行排序呢？假设有一个包含地址簿条目的list，其中每个元素都是一个对象，包含姓名、地址等内容，如何确保按姓名对其进行排序呢？

答案是采取下面两种方式之一：

- 在list包含的对象所属的类中，实现运算符<。
- 提供一个排序二元谓词——一个这样的函数，即接受两个输入值，并返回一个布尔值，指出第一个值是否比第二个值小。

在实际的应用程序中，很少使用STL容器来存储整数，而是存储用户定义的类型，如类或结构。程序清单18.7是一个使用联系人list的示例。乍一看，这个示例很长，但大部分代码都很简单。

程序清单18.7 存储对象的list：创建一个联系人列表

```
 0: #include <list>
 1: #include <string>
 2: #include <iostream>
 3: using namespace std;
 4:
 5: template <typename T>
 6: void DisplayContents (const T& Input)
 7: {
 8:    for(auto iElement = Input.cbegin() // auto, cbegin and cend: c++11
 9:      ; iElement != Input.cend()
10:      ; ++ iElement )
11:      cout << *iElement << endl;
12:
13:    cout << endl;
14: }
15:
16: struct ContactItem
17: {
18:    string strContactsName;
19:    string strPhoneNumber;
20:    string strDisplayRepresentation;
21:
22:    // Constructor and destructor
23:    ContactItem (const string& strName, const string & strNumber)
24:    {
25:       strContactsName = strName;
26:       strPhoneNumber = strNumber;
27:       strDisplayRepresentation = (strContactsName + ": " + strPhoneNumber);
28:    }
29:
```

```
30:    // used by list::remove() given contact list item
31:    bool operator == (const ContactItem& itemToCompare) const
32:    {
33:       return (itemToCompare.strContactsName == this->strContactsName);
34:    }
35:
36:    // used by list::sort() without parameters
37:    bool operator < (const ContactItem& itemToCompare) const
38:    {
39:       return (this->strContactsName < itemToCompare.strContactsName);
40:    }
41:
42:    // Used in DisplayContents via cout
43:    operator const char*() const
44:    {
45:      return strDisplayRepresentation.c_str();
46:    }
47: };
48:
49: bool SortOnPhoneNumber (const ContactItem& item1,
50:                         const ContactItem& item2)
51: {
52:    return (item1.strPhoneNumber < item2.strPhoneNumber);
53: }
54:
55: int main ()
56: {
57:    list <ContactItem> Contacts;
58:    Contacts.push_back(ContactItem("Jack Welsch", "+1 7889 879 879"));
59:    Contacts.push_back(ContactItem("Bill Gates", "+1 97 7897 8799 8"));
60:    Contacts.push_back(ContactItem("Angela Merkel", "+49 23456 5466"));
61:    Contacts.push_back(ContactItem("Vladimir Putin", "+7 6645 4564 797"));
62:    Contacts.push_back(ContactItem("Manmohan Singh", "+91 234 4564 789"));
63:    Contacts.push_back(ContactItem("Barack Obama", "+1 745 641 314"));
64:
65:    cout << "List in initial order: " << endl;
66:    DisplayContents(Contacts);
67:
68:    Contacts.sort();
69:    cout << "After sorting in alphabetical order via operator<:" << endl;
70:    DisplayContents(Contacts);
71:
72:    Contacts.sort(SortOnPhoneNumber);
73:    cout << "After sorting in order of phone numbers via predicate:" << endl;
74:    DisplayContents(Contacts);
75:
76:    cout << "After erasing Putin from the list: ";
77:    Contacts.remove(ContactItem("Vladimir Putin", ""));
78:    DisplayContents(Contacts);
79:
80:    return 0;
81: }
```

▼ **输出：**

```
List in initial order:
Jack Welsch: +1 7889 879 879
Bill Gates: +1 97 7897 8799 8
Angela Merkel: +49 23456 5466
Vladimir Putin: +7 6645 4564 797
Manmohan Singh: +91 234 4564 789
Barack Obama: +1 745 641 314
```

```
After sorting in alphabetical order via operator<:
Angela Merkel: +49 23456 5466
Barack Obama: +1 745 641 314
Bill Gates: +1 97 7897 8799 8
Jack Welsch: +1 7889 879 879
Manmohan Singh: +91 234 4564 789
Vladimir Putin: +7 6645 4564 797

After sorting in order of phone numbers via predicate:
Barack Obama: +1 745 641 314
Jack Welsch: +1 7889 879 879
Bill Gates: +1 97 7897 8799 8
Angela Merkel: +49 23456 5466
Vladimir Putin: +7 6645 4564 797
Manmohan Singh: +91 234 4564 789

After erasing Putin from the list:
Barack Obama: +1 745 641 314
Jack Welsch: +1 7889 879 879
Bill Gates: +1 97 7897 8799 8
Angela Merkel: +49 23456 5466
Manmohan Singh: +91 234 4564 789
```

▼ 分析：

首先，将重点放在第 57～81 行的 main()函数上。第 57 行实例化了一个 list，它包含类型为 ContactItem 的地址簿条目。第 58～63 行使用了一些著名技术专家和政治家的姓名和电话号码（虚构的）填充了该 list，而第 66 行显示了该 list 的内容。第 68 行调用了 list::sort，但没有提供谓词函数。在没有提供谓词的情况下，函数 sort 检查 ContactItem 是否定义了运算符<，发现第 37～40 行定义了该运算符。ContactItem::operator<让 list::sort 按姓名的字母顺序排列元素（而不是根据电话号码或随机逻辑进行排序）。要根据电话号码进行排序，可在调用 list::sort()时提供二元谓词函数 SortOnPhoneNumber()，如第 72 行所示。这个函数是在第 49～53 行实现的，它根据电话号码（而不是姓名）对两个类型为 ContactItem 的输入参数进行比较，从而让 list::sort 根据电话号码对这个名人列表进行排序，如输出所示。最后，第 77 行使用 list::remove()将一个名人的联系信息从 list 中删除。您将参数设置成了包含该名人姓名的 ContactItem 对象，list::remove()使用第 30～34 行实现的 ContactItem::operator==将该对象与 list 中的元素进行比较。该运算符在姓名相同时返回 true，向 list::remove()指出了匹配标准。

这个例子表明 STL list 是一个模板类，可用于创建任何对象类型的列表，它还说明了运算符与谓词的重要性。

C++11

std::forward_list

从 C++11 起，您可以使用 forward_list，而不是双向链表 std::list。std::forward_list 是一种单向链表，即只允许沿一个方向遍历，如图 18.2 所示。

图 18.2 单向链表的可视化表示

提示

要使用 std::forward_list，需要包含头文件<forward_list>：

```
#include<forward_list>
```

forward_list 的用法与 list 很像，但只能沿一个方向移动迭代器，且插入元素时只能使用函数 push_front()，而不能使用 push_back()。当然，总是可以使用 insert()及其重载版本在指定位置插入元素。

程序清单 18.8 演示了 forward_list 类的一些函数。

程序清单 18.8 forward_list 的基本插入和删除操作

```
 0: #include<forward_list>
 1: #include<iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents (const T& Input)
 6: {
 7:    for (auto iElement = Input.cbegin() // auto and cbegin: C++11
 8:       ; iElement != Input.cend ()
 9:       ; ++ iElement )
10:       cout << *iElement << ' ';
11:
12:    cout << endl;
13: }
14:
15: int main()
16: {
17:    forward_list<int> flistIntegers;
18:    flistIntegers.push_front(0);
19:    flistIntegers.push_front(2);
20:    flistIntegers.push_front(2);
21:    flistIntegers.push_front(4);
22:    flistIntegers.push_front(3);
23:    flistIntegers.push_front(1);
24:
25:    cout << "Contents of forward_list: " << endl;
26:    DisplayContents(flistIntegers);
27:
28:    flistIntegers.remove(2);
29:    flistIntegers.sort();
30:    cout << "Contents after removing 2 and sorting: " << endl;
31:    DisplayContents(flistIntegers);
32:
33:    return 0;
34: }
```

▼ 输出：

```
Contents of forward_list:
1 3 4 2 2 0
Contents after removing 2 and sorting:
0 1 3 4
```

▼ 分析：

该示例表明，forward_list 与 list 很像。鉴于 forward_list 不支持双向迭代，因此只能对迭代器使用运算符++，而不能使用--。在这个示例中，第 28 行使用函数 remove(2)删除了值为 2 的所有元素；第 29 行调用了 sort()，这将使用默认的排序谓词，即 std::less<T>。

forward_list 的优点在于，它是一种单向链表，占用的内存比 list 稍少，因为只需指向下一个元素，而无需指向前一个元素。

应该	不应该
如果需要频繁地插入或删除元素（尤其是在中间插入或删除时），应使用 std::list，而不是 std::vetor。因为在这种情况下，vector 需要调整其内部缓冲区的大小，以支持数组语法，还需执行开销高昂的复制操作，而 list 只需建立或断开链接。 请记住，可使用成员方法 push_front()和 push_back()分别在 list 开头和末尾插入元素。 对于要使用 list 等 STL 容器存储其对象的类，别忘了在其中实现运算符<和==，以提供默认的排序和删除谓词。 请记住，像其他 STL 容器类一样，总是可以使用 list::size()来确定 list 包含多少个元素。 请记住，像其他 STL 容器类一样，可使用方法 list::clear()清空 list。	无需频繁在两端插入或删除元素，且不用在中间插入或删除元素时，请不要使用 list；在这些情况下，vector 和 deque 的速度要快得多。 如果不想根据默认标准进行删除或排序，别忘了给 sort()和 remove()提供一个谓词函数。

18.4 总结

本章介绍 list 的特征以及各种 list 操作。现在读者知道了 list 的最常用函数，能够创建用于存储任何类型的对象的 list。

18.5 问与答

问：list 为何提供诸如 sort 和 remove 等成员函数？

答：STL list 类需要确保指向 list 中元素的迭代器始终有效，而不管如何在 list 中移动该元素。虽然 STL 算法也可用于 list，但 list 的成员函数可确保 list 的上述特征，即将 list 排序后，指向 list 中元素的迭代器仍指向原来的元素。

问：使用存储 CAnimal 对象的 list 时，为让 list 的成员函数能够正确处理 CAnimal 对象，应为 CAnimal 类实现哪些运算符？

答：对于其对象将存储在 STL 容器中的类，必须为它实现默认比较运算符==和运算符<。

问：对于下述代码行，该如何将关键字 auto 替换为显式类型？

```
list<int> listIntegers(10);  // list of 10 integers
auto iFirstElement = listIntegers.begin();
```

答：如果您使用的是不遵循 C++11 的老式编译器，应将关键字 auto 替换为显式类型，如下所示：

```
list<int> listIntegers(10);  // list of 10 integers
list<int>::iterator iFirstElement = listIntegers.begin();
```

18.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

18.6.1 测验

1. 与在开头或末尾插入元素相比，在 STL list 中间插入元素是否会降低性能？
2. 假设有两个迭代器分别指向 STL list 对象中的两个元素，然后在这两个元素之间插入了一个元素。请问这种插入是否会导致这两个迭代器无效？
3. 如何清空 std::list 的内容？
4. 能否在 list 中插入多个元素？

18.6.2 练习

1. 编写一个程序，它接受用户输入的数字并将它们插入到 list 开头。
2. 使用一个简短的程序来演示这样一点：在 list 中插入一个新元素，导致迭代器指向的元素的相对位置发生变化后，该迭代器仍有效。
3. 编写一个程序，使用 list 的 insert 函数将一个 vector 的内容插入到一个 STL list 中。
4. 编写一个程序，对字符串 list 进行排序以及反转排列顺序。

第 19 章
STL 集合类

标准模板库(STL)向程序员提供了一些容器类，以便在应用程序中进行频繁而快速的搜索。std::set 和 std::multiset 用于存储一组经过排序的元素，其查找元素的复杂度为对数，而 unordered 集合的插入和查找时间是固定的。

在本章中，您将学习：

- 如何使用 STL 容器 set、multiset、unordered_set 和 unordered_multiset；
- 插入、删除和查找元素；
- 使用 STL 容器 set、multiset、unordered_set 和 unordered_multiset 的优缺点；

19.1 简介

容器 set 和 multiset 让程序员能够在容器中快速查找键，键是存储在一维容器中的值。set 和 multiset 之间的区别在于，后者可存储重复的值，而前者只能存储唯一的值。

图 19.1 表明，set 只能包含唯一的人名，而 multiset 可存储重复的人名。STL 容器是泛型模板类，因此是通用的，可用于存储字符串、整型、结构或类对象。

图 19.1 包含人名的 set 和 multiset 的可视化表示

为实现快速搜索，STL set 和 multiset 的内部结构像二叉树，这意味着将元素插入到 set 或 multiset 时将对其进行排序，以提高查找速度。这还意味着不像 vector 那样可以使用其他元素替换给定位置的元素，位于 set 中特定位置的元素不能替换为值不同的新元素，这是因为 set 将把新元素同二叉树中的其他元素进行比较，进而将其放在其他位置。

提示

要使用 std::set 或 set::multiset 类，需要包含头文件<set>：

```
#include <set>
```

19.2 STL set 和 multiset 的基本操作

STL set 和 multiset 都是模板类，要使用其成员函数，必须先实例化。

19.2.1 实例化 std::set 对象

要实例化一个特定类型的 set 或 multiset，必须针对该类型具体化模板类 std::set 或 std::multiset：

```
std::set <int> setIntegers;
std::multiset <int> msetIntegers;
```

要声明一个包含 Tuna 对象的 set 或 multiset，应这样编写代码：

```
std::set <Tuna> setIntegers;
std::multiset <Tuna> msetIntegers;
```

要声明一个指向 set 或 multiset 中元素的迭代器，应这样做：

```
std::set<int>::const_iterator iElementInSet;
std::multiset<int>::const_iterator iElementInMultiset;
```

如果需要一个可用于修改值或调用非 const 函数的迭代器，应将 const_iterator 替换为 iterator。

鉴于 set 和 multiset 都是在插入时对元素进行排序的容器，如果您没有指定排序标准，它们将使用默认谓词 std::less，确保包含的元素按升序排列。

要创建二元排序谓词，可在类中定义一个 operator()，让它接受两个参数（其类型与集合存储的数据类型相同），并根据排序标准返回 true。下面是一个这样的排序谓词，它按降序排列元素：

```
// used as a template parameter in set / multiset instantiation
template <typename T>
struct SortDescending
{
   bool operator()(const T& lhs, const T& rhs) const
   {
      return (lhs > rhs);
   }
};
```

然后，在实例化 set 或 multiset 时指定该谓词，如下所示：

```
// a set and multiset of integers (using sort predicate)
set <int, SortDescending<int> > setIntegers;
multiset <int, SortDescending<int> > msetIntegers;
```

除上述实例化方式外，还可这样创建 set 或 multiset，即包含另一个 set 或 multiset 的部分或全部元素，如程序清单 19.1 所示。

程序清单 19.1 各种实例化 set 和 multiset 的方式

```
 0: #include <set>
 1:
 2: // used as a template parameter in set / multiset instantiation
 3: template <typename T>
 4: struct SortDescending
 5: {
 6:    bool operator()(const T& lhs, const T& rhs) const
 7:    {
 8:       return (lhs > rhs);
 9:    }
10: };
11:
12: int main ()
13: {
```

```
14:    using namespace std;
15:
16:    // a simple set or multiset of integers (using default sort predicate)
17:    set <int> setIntegers1;
18:    multiset <int> msetIntegers1;
19:
20:    // set and multiset instantiated given a user-defined sort predicate
21:    set<int, SortDescending<int> > setIntegers2;
22:    multiset<int, SortDescending<int> > msetIntegers2;
23:
24:    // creating one set from another, or part of another container
25:    set<int> setIntegers3(setIntegers1);
26:    multiset<int> msetIntegers3(setIntegers1.cbegin(), setIntegers1.cend());
27:
28:    return 0;
29: }
```

▼ 分析：

这个程序没有输出，但演示了各种实例化整型 set 和 multiset 的方式。第 17 和 18 行演示了最简单的实例化方式：省略除类型外的其他所有模板参数，这导致使用默认排序谓词，即 std::less<T>。如果要覆盖这种默认行为，需要像第 3～10 行那样定义一个谓词，并像第 21 和 22 行那样使用它。该谓词将元素按降序排列（默认为升序）。最后，第 25 和 26 行演示了这样两种实例化方式：使用一个 set 来实例化另一个 set；使用 set 的特定范围内的元素来实例化 multiset，这里也可使用 vector、list 或其他任何 STL 容器类，只要它能够通过 cbegin()和 cend()返回描述边界的迭代器。

提示

> cbegin()和 cend()导致了编译错误吗？
> 如果要使用不遵循 C++11 的编译器编译该程序，请将 cbegin()和 cend()分别替换为 begin()和 end()。cbegin()和 cend()是 C++11 新增的，它们返回一个 const 迭代器，不能用于修改元素。

19.2.2 在 set 或 multiset 中插入元素

set 和 multiset 的大多数函数的用法类似，它们接受类似的参数，返回类型也类似。例如，要在这两种容器中插入元素，都可使用成员函数 insert，这个函数接受要插入的值：

```
setIntegers.insert (-1);
msetIntegers.insert (setIntegers.begin (), setIntegers.end ());
```

程序清单 19.2 演示了如何在这些容器中插入元素。

程序清单 19.2 在 STL set 或 multiset 中插入元素

```
 0: #include <set>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents (const T& Input)
 6: {
 7:    for(auto iElement = Input.cbegin () // auto, cebgin(): C++11
 8:       ; iElement != Input.cend () // cend() is new in C++11
 9:       ; ++ iElement )
10:       cout << *iElement << ' ';
11:
12:    cout << endl;
13: }
```

```
14:
15: int main ()
16: {
17:     set <int> setIntegers;
18:     multiset <int> msetIntegers;
19:
20:     setIntegers.insert (60);
21:     setIntegers.insert (-1);
22:     setIntegers.insert (3000);
23:     cout << "Writing the contents of the set to the screen" << endl;
24:     DisplayContents (setIntegers);
25:
26:     msetIntegers.insert (setIntegers.begin (), setIntegers.end ());
27:     msetIntegers.insert (3000);
28:
29:     cout << "Writing the contents of the multiset to the screen" << endl;
30:     DisplayContents (msetIntegers);
31:
32:     cout << "Number of instances of '3000' in the multiset are: '";
33:     cout << msetIntegers.count (3000) << "'" << endl;
34:
35:     return 0;
36: }
```

▼ 输出：

```
Writing the contents of the set to the screen
-1 60 3000
Writing the contents of the multiset to the screen
-1 60 3000 3000
Number of instances of '3000' in the multiset are: '2'
```

▼ 分析：

第 4～13 行是通用的模板函数 DisplayContents()，您在第 17 和 18 章见过，它将 STL 容器的内容显示到控制台（屏幕）。第 17 行和第 18 行定义了一个 set 对象和一个 multiset 对象。第 20～22 行调用成员函数 insert()将值插入到 set 中。第 19 行演示了如何使用 insert()将一个 set 的内容插入到一个 multiset 中，这里是将 setIntegers 的内容插入到 multiset msetIntegers 中。第 27 行插入一个值为 3000 的元素，该元素已存储在 multiset 中。从输出可知，multiset 能够存储多个相同的值。第 32 行和第 33 行演示了成员函数 multiset::count()的用法，它返回 multiset 中有多少个元素存储了指定的值。

提示

mulitset::count()确定 multiset 包含多少个这样的元素，即其值与通过实参传递给这个函数的值相同。

提示

关键字 auto 导致了编译错误吗？

在程序清单 19.2 中，函数 DisplayContents()使用了 C++11 关键字 auto 来指定迭代器的类型，如第 7 行所示。另外，它还使用了 C++11 新增的 cbegin()和 cend()，它们返回一个 const_iterator。

对于这个示例以及后面的示例，如果要使用不遵循 C++11 的编译器进行编译，需要将 auto 替换为显式类型。

因此，如果使用的是较旧的编译器，需要将 DisplayContents()修改成下面这样：

```
template <typename T>
void DisplayContents (const T& Input)
{
   for (T::const_iterator iElement = Input.begin () //
  explicit type
```

```
            ; iElement != Input.end ()
            ; ++ iElement )
            cout << *iElement << ' ';

        cout << endl;
    }
```

19.2.3 在 STL set 或 multiset 中查找元素

诸如 set、multiset、map 和 multimap 等关联容器都提供了成员函数 find()，它让您能够根据给定的键来查找值：

```
auto iElementFound = setIntegers.find (-1);

// Check if found...
if (iElementFound != setIntegers.end ())
   cout << "Element " << *iElementFound << " found!" << endl;
else
   cout << "Element not found in set!" << endl;
```

程序清单 19.3 演示了 find()的用法。multiset 可包含多个值相同的元素，因此对于 multiset，这个函数查找第一个与给定键匹配的元素。

程序清单 19.3 使用成员函数 find

```
 0: #include <set>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: int main ()
 5: {
 6:    set<int> setIntegers;
 7:
 8:    // Insert some random values
 9:    setIntegers.insert (43);
10:    setIntegers.insert (78);
11:    setIntegers.insert (-1);
12:    setIntegers.insert (124);
13:
14:    // Write contents of the set to the screen
15:    for (auto iElement = setIntegers.cbegin ()
16:        ; iElement != setIntegers.cend ()
17:        ; ++ iElement )
18:       cout << *iElement << endl;
19:
20:    // Try finding an element
21:    auto iElementFound = setIntegers.find (-1);
22:
23:    // Check if found...
24:    if (iElementFound != setIntegers.end ())
25:       cout << "Element " << *iElementFound << " found!" << endl;
26:    else
27:       cout << "Element not found in set!" << endl;
28:
29:    // Try finding another element
30:    auto iAnotherFind = setIntegers.find (12345);
31:
32:    // Check if found...
33:    if (iAnotherFind != setIntegers.end ())
34:       cout << "Element " << *iAnotherFind << " found!" << endl;
```

```
35:    else
36:       cout << "Element 12345 not found in set!" << endl;
37:
38:    return 0;
39: }
```

▼ **输出：**

```
-1
43
78
124
Element -1 found!
Element 12345 not found in set!
```

▼ **分析：**

第 21～27 行演示了成员函数 find()的用法。第 24 行将 find()返回的迭代器与 end()进行比较，以核实是否找到了指定的元素。如果该迭代器有效，便可使用*iElementFound 访问它指向的值。

注意 程序清单 19.3 所示的示例也适用于 multiset，只需在将第 6 行的 set 替换为 multiset 即可，这不会影响应用程序。

19.2.4 删除 STL set 或 multiset 中的元素

诸如 set、multiset、map 和 multimap 等关联容器都提供了成员函数 erase()，它让您能够根据键删除值：

```
setObject.erase (key);
```

erase 函数的另一个版本接受一个迭代器作为参数，并删除该迭代器指向的元素：

```
setObject.erase (iElement);
```

通过使用迭代器指定的边界，可将指定范围内的所有元素都从 set 或 multiset 中删除：

```
setObject.erase (iLowerBound, iUpperBound);
```

程序清单 19.4 演示了如何使用 erase()来删除 mset 或 multiset 中的元素。

程序清单 19.4 使用 multiset 的成员函数 erase

```
 0: #include <set>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents (const T& Input)
 6: {
 7:    for(auto iElement = Input.cbegin () // auto, cebgin(): C++11
 8:       ; iElement != Input.cend () // cend() is new in C++11
 9:       ; ++ iElement )
10:       cout << *iElement << ' ';
11:
12:    cout << endl;
13: }
14:
15: typedef multiset <int> MSETINT;
16:
17: int main ()
18: {
19:    MSETINT msetIntegers;
```

```
20:
21:    // Insert some random values
22:    msetIntegers.insert (43);
23:    msetIntegers.insert (78);
24:    msetIntegers.insert (78);    // Duplicate
25:    msetIntegers.insert (-1);
26:    msetIntegers.insert (124);
27:
28:    cout << "multiset contains " << msetIntegers.size () << " elements.";
29:    cout << " These are: " << endl;
30:
31:    // Write contents of the multiset to the screen
32:    DisplayContents(msetIntegers);
33:
34:    cout << "Please enter a number to be erased from the set" << endl;
35:    int nNumberToErase = 0;
36:    cin >> nNumberToErase;
37:
38:    cout << "Erasing " << msetIntegers.count (nNumberToErase);
39:    cout << " instances of value " << nNumberToErase << endl;
40:
41:    // Try finding an element
42:    msetIntegers.erase (nNumberToErase);
43:
44:    cout << "multiset contains " << msetIntegers.size () << " elements.";
45:    cout << " These are: " << endl;
46:    DisplayContents(msetIntegers);
47:
48:    return 0;
49: }
```

▼ 输出：

```
multiset contains 5 elements. These are:
-1 43 78 78 124
Please enter a number to be erased from the set
78
Erasing 2 instances of value 78
multiset contains 3 elements. These are:
-1 43 124
```

▼ 分析：

注意到第 15 行使用了 typedef。第 38 行使用了 count()来确定有多少个元素包含特定的值。实际的删除操作是在第 42 行执行的，它删除与特定值匹配的所有元素。

函数 erase()被重载了。可将迭代器（如 find 返回的迭代器）传递给 erase()，以删除找到的元素，如下所示：

```
MSETINT::iterator iElementFound = msetIntegers.find (nNumberToErase);
if (iElementFound != msetIntegers.end ())
   msetIntegers.erase (iElementFound);
else
   cout << "Element not found!" << endl;
```

同样，erase()还可用于从 multiset 中删除指定范围内的元素：

```
MSETINT::iterator iElementFound = msetIntegers.find (nValue);

if (iElementFound != msetIntegers.end ())
    msetIntegers.erase (msetIntegers.begin (), iElementFound);
```

上述代码删除从开头到值为 nValue 的所有元素（不包含 nValue）。要清空 set 和 multiset 的内容，可使用其成员函数 clear()。

学习 set 和 multiset 的基本函数后，接下来看一个使用 set 容器的实际应用程序。程序清单 19.5 是基于菜单的电话簿的简单实现，它让用户能够插入、查找、删除和显示人名和电话号码。

程序清单 19.5 一个使用 STL set 及其成员函数 find 和 erase 的电话簿

```
 0: #include <set>
 1: #include <iostream>
 2: #include <string>
 3: using namespace std;
 4:
 5: template <typename T>
 6: void DisplayContents (const T& Input)
 7: {
 8:    for(auto iElement = Input.cbegin () // auto, cebgin(): C++11
 9:       ; iElement != Input.cend () // cend() is new in C++11
10:       ; ++ iElement )
11:       cout << *iElement << endl;
12:
13:    cout << endl;
14: }
15:
16: struct ContactItem
17: {
18:    string strContactsName;
19:    string strPhoneNumber;
20:    string strDisplayRepresentation;
21:
22:    // Constructor and destructor
23:    ContactItem (const string& strName, const string & strNumber)
24:    {
25:       strContactsName = strName;
26:       strPhoneNumber = strNumber;
27:       strDisplayRepresentation = (strContactsName + ": " + strPhoneNumber);
28:    }
29:
30:    // used by set::find()
31:    bool operator == (const ContactItem& itemToCompare) const
32:    {
33:       return (itemToCompare.strContactsName == this->strContactsName);
34:    }
35:
36:    // used as a sort predicate
37:    bool operator < (const ContactItem& itemToCompare) const
38:    {
39:       return (this->strContactsName < itemToCompare.strContactsName);
40:    }
41:
42:    // Used in DisplayContents via cout
43:    operator const char*() const
44:    {
45:       return strDisplayRepresentation.c_str();
46:    }
47: };
48:
49: int main ()
50: {
51:    set<ContactItem> setContacts;
52:    setContacts.insert(ContactItem("Jack Welsch", "+1 7889 879 879"));
53:    setContacts.insert(ContactItem("Bill Gates", "+1 97 7897 8799 8"));
54:    setContacts.insert(ContactItem("Angela Merkel", "+49 23456 5466"));
55:    setContacts.insert(ContactItem("Vladimir Putin", "+7 6645 4564 797"));
```

```
56:    setContacts.insert(ContactItem("Manmohan Singh", "+91 234 4564 789"));
57:    setContacts.insert(ContactItem("Barack Obama", "+1 745 641 314"));
58:    DisplayContents(setContacts);
59:
60:    cout << "Enter a person whose number you wish to delete: ";
61:    string NameInput;
62:    getline(cin, NameInput);
63:
64:    auto iContactFound = setContacts.find(ContactItem(NameInput, ""));
65:    if(iContactFound != setContacts.end())
66:    {
67:       // Erase the contact found in set
68:       setContacts.erase(iContactFound);
69:       cout << "Displaying contents after erasing: " << NameInput << endl;
70:       DisplayContents(setContacts);
71:    }
72:    else
73:       cout << "Contact not found" << endl;
74:
75:    return 0;
76: }
```

▼ **输出：**

```
Angela Merkel: +49 23456 5466
Barack Obama: +1 745 641 314
Bill Gates: +1 97 7897 8799 8
Jack Welsch: +1 7889 879 879
Manmohan Singh: +91 234 4564 789
Vladimir Putin: +7 6645 4564 797

Enter a person whose number you wish to delete: Jack Welsch
Displaying contents after erasing: Jack Welsch
Angela Merkel: +49 23456 5466
Barack Obama: +1 745 641 314
Bill Gates: +1 97 7897 8799 8
Manmohan Singh: +91 234 4564 789
Vladimir Putin: +7 6645 4564 797
```

▼ **分析：**

这个程序清单与按字母顺序对 std::list 进行排序的程序清单 18.7 很像，差别在于 std::set 排序是在插入元素时进行的。输出表明，您不需要调用任何函数来对 set 中的元素进行排序，因为排序是在插入元素时进行的。您让用户指定要删除的条目，然后第 64 行调用 find()找到该条目，而第 68 行使用 erase()删除该条目。

提示

这个电话簿实现是基于 STL set 的，因此不允许多个元素包含相同的值。如果要让电话簿能够存储两个相同的人名（如 Tom），则应使用 STL multiset。如果 setContacts 为 multiset，上述代码仍可正确运行。要使用 multiset 存储多个值相同的元素，应使用 count()成员函数来获悉有多少个元素包含特定的值，这在前面的代码示例中演示过。在 multiset 中，类似的元素是相邻的，find 函数返回一个迭代器，它指向找到的第一个元素。可将该迭代器递增以获取下一个元素。

19.3 使用 STL set 和 multiset 的优缺点

对需要频繁查找的应用程序来说，STL set 和 multiset 很有优势，因为其内容是经过排序的，

因此查找速度更快。然而，为提供这种优势，容器在插入元素时进行排序。因此，插入元素时有额外开销，因为需要对元素进行排序——如果应用程序将频繁使用 find()等函数，则这种开销是值得的。

find()利用了内部的二叉树结构。这种有序的二叉树结构使得 set 和 multiset 与顺序容器（如 vector）相比有一个缺点：在 vector 中，可以使用新值替换迭代器（如 std::find()返回的迭代器）指向的元素；但 set 根据元素的值对其进行了排序，因此不能使用迭代器覆盖元素的值，虽然通过编程可实现这种功能。

C++11

STL 散列集合实现 std::unordered_set 和 std::unordered_multiset

STL std::set 和 std::multiset 使用 std::less<T>或提供的谓词对元素（同时也是键）进行排序。相对于 vector 等未经排序的容器，在经过排序的容器中查找的速度更快，其 sort 的复杂度为对数。这意味着在 set 中查找元素时，所需的时间不是与元素数成正比，而是与元素数的对数成正比。因此，相比于包含 100 个元素的 set，在包含 10000 个元素的 set 中查找时，需要的平均时间将翻一倍（因为 100^2 = 10000，即 log(10000) = 2 × log(100)）。

相比于未经排序的容器（查找时间与元素数成正比），这极大地改善了性能，但有时候这还不够。程序员和数学家都喜欢探索插入和排序时间固定的方式，一种这样的方式是使用基于散列的实现，即使用散列函数来计算排序索引。将元素插入散列集合时，首先使用散列函数计算出一个唯一的索引，再根据该索引决定将元素放到哪个桶（bucket）中。

STL 提供的容器类 std::unordered_set 就是基于散列的 set。

提示

要使用 STL 容器 std::unordered_set 或 std::unordered_multiset，需要包含头文件<unordered_set>：

```
#include<unordered_set>
```

相比于 std::set，这个类的用法差别不大：

```
// instantiation:
unordered_set<int> usetInt;

// insertion of an element
usetInt.insert(1000);

// find():
auto iPairThousand = usetInt.find(1000);

if (iPairThousand != usetInt.end())
   cout << *iPairThousand << endl;
```

然而，unordered_set 的一个重要特征是，有一个负责确定排列顺序的散列函数：

```
unordered_set<int>::hasher HFn = usetInt.hash_function();
```

程序清单 19.6 演示了 std::unordered_set 提供的一些常见方法的用法。

程序清单 19.6 使用 std::unordered_set 及其方法 insert()、find()、size()、max_bucket_count()、load_factor()和 max_load_factor()

```
0: #include<unordered_set>
1: #include <iostream>
2: using namespace std;
3:
4: template <typename T>
5: void DisplayContents(const T& Input)
6: {
```

```
 7:    cout << "Number of elements, size() = " << Input.size() << endl;
 8:    cout << "Max bucket count = " << Input.max_bucket_count() << endl;
 9:    cout << "Load factor: " << Input.load_factor() << endl;
10:    cout << "Max load factor = " << Input.max_load_factor() << endl;
11:    cout << "Unordered set contains: " << endl;
12:
13:    for(auto iElement = Input.cbegin() // auto, cbegin: c++11
14:       ; iElement != Input.cend()  // cend() is new in C++11
15:       ; ++ iElement )
16:       cout<< *iElement << ' ';
17:
18:    cout<< endl;
19: }
20:
21: int main()
22: {
23:    // instantiate unordered_set of int to string:
24:    unordered_set<int> usetInt;
25:
26:    usetInt.insert(1000);
27:    usetInt.insert(-3);
28:    usetInt.insert(2011);
29:    usetInt.insert(300);
30:    usetInt.insert(-1000);
31:    usetInt.insert(989);
32:    usetInt.insert(-300);
33:    usetInt.insert(111);
34:    DisplayContents(usetInt);
35:    usetInt.insert(999);
36:    DisplayContents(usetInt);
37:
38:    // find():
39:    cout << "Enter int you want to check for existence in set: ";
40:    int Key = 0;
41:    cin >> Key;
42:    auto iPairThousand = usetInt.find(Key);
43:
44:    if (iPairThousand != usetInt.end())
45:       cout << *iPairThousand << " found in set" << endl;
46:    else
47:       cout << Key << " not available in set" << endl;
48:
49:    return 0;
50: }
```

▼ 输出：

```
Number of elements, size() = 8
Max bucket count = 8
Load factor: 1
Max load factor = 1
Unordered set contains:
1000 -3 2011 300 -1000 -300 989 111
Number of elements, size() = 9
Max bucket count = 64
Load factor: 0.140625
Max load factor = 1
Unordered set contains:
1000 -3 2011 300 -1000 -300 989 999 111
Enter int you want to check for existence in set: -1000
-1000 found in set
```

▼ **分析：**

这个示例创建了一个整型 unordered_set，在其中插入 8 个值，再显示其内容，包括 max_bucket_count()、load_factor()和 max_load_factor()提供的统计信息，如第 8～10 行所示。输出表明，最初的桶数为 8 个，而该容器包含 8 个元素，因此负载系数为 1，与最大负载系数相同。插入第 9 个元素时，unordered_set 重新组织：创建 64 个桶并重新创建散列表，而负载系数降低了。main()中的其他代码表明，在 unordered_set 中查找元素的语法与 set 类似。find()返回一个迭代器，在使用该迭代器之前，需要核实 find()是否成功，如第 44 行所示。

注意

散列函数通常用于根据键在散列表中查找值，详情请参阅第 20 章中介绍 std::unordered_map 的一节。

std::unordered_map 是 C++11 新增的一种散列表实现。

应该	不应该
请牢记，STL set 和 multiset 容器针对频繁查找的情形进行了优化。 请牢记，std::multiset 可存储多个值相同的元素（键），而 std::set 只能存储不同的值。 务必使用 multiset::count(value)确定有多少个元素包含特定的值。 请牢记，set::size()和 multiset::size()指出容器包含多少个元素。	对于其对象将存储在 set 或 multiset 等容器中的类，别忘了在其中实现运算符<和==。前者将成为排序谓词，而后者将用于 set::find()等函数。 在需要频繁插入而很少查找的情形下，不要使用 std::set 或 std::multiset；在这种情形下，std::vector 和 std::list 通常更适合。

19.4 总结

本章介绍了 STL set 和 multiset 及其重要的成员函数和特征，还通过一个基于菜单的简单电话簿演示了如何使用 set 和 multiset，该电话簿提供了搜索和删除功能。

19.5 问与答

问：如何声明一个其元素按降序排列的整型 set？

答：set<int>定义一个整型 set，这种 set 使用默认排序谓词 std::less<T>将元素按升序排列，也可将其定义为 set<int, less <int>>。要按降序排列，应将 set 定义为 set<int, greater <int>>。

问：如果在一个字符串 set 中插入字符串 Jack 两次，将发生什么情况？

答：set 不能存储多个相同的值，因此 std::set 类的实现不允许插入第二个 Jack。

问：在前一个例子中，如果需要两个 Jack，该如何办？

答：set 只能存储唯一的值。应选择使用 multiset。

问：multiset 的哪个成员函数指出容器有多少个元素包含特定的值？

答：函数 count (value)。

问：我使用函数 find 在 set 中找到了一个元素，并有一个指向该元素的迭代器。能否使用这个迭代器来修改它指向的元素的值？

答：不能。有些 STL 实现可能允许用户通过迭代器（如 find 函数返回的迭代器）修改元素的值，但不应这样做。应将指向 set 中元素的迭代器视为 const 迭代器，即使 STL 实现没有强制这样做。

19.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

19.6.1 测验

1. 使用 set <int>声明整型 set 时，排序标准将由哪个函数提供？
2. 在 multiset 中，重复的值以什么方式出现？
3. set 和 multiset 的哪个成员函数指出容器包含多少个元素？

19.6.2 练习

1. 在不修改 ContactItem 的情况下，扩展本章的电话簿应用程序，使其能够根据电话号码查找人名（提示：定义 set 时指定一个二元谓词，以便根据电话号码对元素进行排序，从而覆盖根据运算符< 进行排序的默认方式）。

2. 定义一个 multiset 来存储单词及其含义，即将 multiset 用作词典（提示：multiset 存储的对象应是一个包含两个字符串的结构，其中一个字符串为单词，另一个字符串是单词的含义）。

3. 通过一个简单程序演示 set 不接受重复的元素，而 multiset 接受。

第 20 章
STL 映射类

标准模板库（STL）向程序员提供了一些容器类，供需要进行频繁而快速搜索的应用程序使用。在本章中，您将学习：

- 如何使用 STL map、multimap、unordered_map 和 unordered_multimap；
- 插入、删除和查找元素；
- 提供自定义的排序谓词；
- 散列表工作原理简介。

20.1 STL 映射类简介

map 和 multimap 是键–值对容器，支持根据键进行查找，如图 20.1 所示。

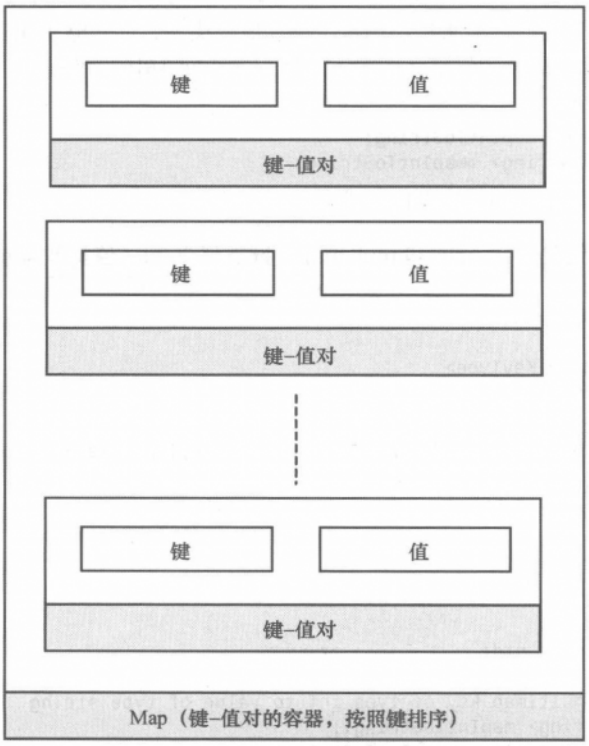


图 20.1 键–值对容器

map 和 multimap 之间的区别在于，后者能够存储重复的键，而前者只能存储唯一的键。

为实现快速查找，STL map 和 multimap 的内部结构看起来像棵二叉树。这意味着在 map 或 multimap 中插入元素时将进行排序；还意味着不像 vector 那样可以使用其他元素替换给定位置的元素，位于 map 中特定位置的元素不能替换为值不同的新元素，这是因为 map 将把新元素同二叉树中的其他元素进行比较，进而将它放在其他位置。

提示

要使用 STL map 或 multimap 类，需要包含头文件<map>：

```
#include<map>
```

20.2 std::map 和 std::multimap 的基本操作

STL map 和 multimap 都是模板类，要使用其成员函数，必须先实例化。

20.2.1 实例化 std::map 和 std::multimap

要实例化将整数用作键、将字符串用作值的 map 或 multimap，必须具体化模板类 std::map 或 std::multimap。实例化模板类 map 时，需要指定键和值的类型以及可选的谓词（它帮助 map 类对插入的元素进行排序）。因此，典型的 map 实例化语法如下：

```
#include <map>
using namespace std;
...
map <keyType, valueType, Predicate=std::less <keyType> > mapObject;
multimap <keyType, valueType, Predicate=std::less <keyType> > mmapObject;
```

第三个模板参数是可选的。如果您值指定了键和值的类型，而省略了第三个模板参数，std::map 和 std::multimap 将把 std::less<>用作排序标准。因此，将整数映射到字符串的 map 或 multimap 类似于下面这样：

```
std::map<int, string> mapIntToString;
std::multimap<int, string> mmapIntToString;
```

程序清单 20.1 更详细地说明了实例化方式。

程序清单 20.1 实例化 STL map 和 multimap（键类型为 int，值类型为 string）

```
 0: #include<map>
 1: #include<string>
 2:
 3: template<typename KeyType>
 4: struct ReverseSort
 5: {
 6:    bool operator()(const KeyType& key1, const KeyType& key2)
 7:    {
 8:       return (key1 > key2);
 9:    }
10: };
11:
12: int main ()
13: {
14:    using namespace std;
15:
16:    // map and multimap key of type int to value of type string
17:    map<int, string> mapIntToString1;
18:    multimap<int, string> mmapIntToString1;
19:
```

```
20:    // map and multimap constructed as a copy of another
21:    map<int, string> mapIntToString2(mapIntToString1);
22:    multimap<int, string> mmapIntToString2(mmapIntToString1);
23:
24:    // map and multimap constructed given a part of another map or multimap
25:    map<int, string> mapIntToString3(mapIntToString1.cbegin(),
26:                                     mapIntToString1.cend());
27:
28:    multimap<int, string> mmapIntToString3(mmapIntToString1.cbegin(),
29:                                           mmapIntToString1.cend());
30:
31:    // map and multimap with a predicate that inverses sort order
32:    map<int, string, ReverseSort<int> > mapIntToString4
33:       (mapIntToString1.cbegin(), mapIntToString1.cend());
34:
35:    multimap<int, string, ReverseSort<int> > mmapIntToString4
36:       (mapIntToString1.cbegin(), mapIntToString1.cend());
37:
38:    return 0;
39: }
```

▼ 分析：

首先，将重点放在第 12～39 行的 main()函数上。第 21 和 22 行演示了实例化键类型为 int、值类型为 string 的 map 和 multimap 的最简单方式；第 25～29 行演示了如何创建一个 map 或 multimap，并使用另一个 map 或 multimap 中指定范围内的值对其进行初始化；第 31～36 行演示了在实例化 map 或 multimap 时如何自定义排序标准。请注意，默认使用排序标准 std::less<T>，这将元素按升序排列。如果要改变这种行为，可提供一个谓词——一个实现了 operator()的类或结构。第 4～10 行定义了一个这样的谓词——结构 ReverseSort，第 32 和 35 行实例化 map 和 multimap 时使用了该谓词。

提示

> cbegin()和 cend()是否导致编译错误？
> 如果要使用不遵循 C++11 的编译器编译该程序，请将 cbegin()和 cend()分别替换为 begin()和 end()。cbegin()和 cend()是 C++11 新增的，它们返回一个 const 迭代器，不能用于修改元素。

20.2.2 在 STL map 或 multimap 中插入元素

map 和 multimap 的大多数函数的用法类似，它们接受类似的参数，返回类型也类似。例如，要在这两种容器中插入元素，都可使用成员函数 insert：

```
std::map<int, std::string> mapIntToString1;
// insert pair of key and value using make_pair function
mapIntToString.insert (make_pair (-1, "Minus One"));
```

鉴于这两种容器包含的元素都是键-值对，因此也可直接使用 std::pair 来指定要插入的键和值：

```
mapIntToString.insert (pair <int, string>(1000, "One Thousand"));
```

另外，还可使用类似于数组的语法进行插入。这种方式对用户不太友好，是由下标运算符（[]）支持的：

```
mapIntToString [1000000] = "One Million";
```

还可使用 map 来实例化 multimap：

```
mapIntToString [1000000] = "One Million";
```

You can also instantiate a `multimap` as a copy of a `map`:

```
std::multimap<int, std::string> mmapIntToString(mapIntToString.cbegin(),
                                                mapIntToString.cend());
```

程序清单 20.2 演示了各种插入元素的方式。

程序清单 20.2 使用 insert()以及数组语法（运算符[]）在 STL map 或 multimap 中插入元素

```
 0: #include <map>
 1: #include <iostream>
 2: #include<string>
 3:
 4: using namespace std;
 5:
 6: // Type-define the map and multimap definition for easy readability
 7: typedef map <int, string> MAP_INT_STRING;
 8: typedef multimap <int, string> MMAP_INT_STRING;
 9:
10: template <typename T>
11: void DisplayContents (const T& Input)
12: {
13:    for(auto iElement = Input.cbegin() // auto and cbegin(): C++11
14:       ; iElement != Input.cend()  // cend() is new in C++11
15:       ; ++ iElement )
16:       cout << iElement->first << " -> " << iElement->second << endl;
17:
18:    cout << endl;
19: }
20:
21: int main ()
22: {
23:    MAP_INT_STRING mapIntToString;
24:
25:    // Insert key-value pairs into the map using value_type
26:    mapIntToString.insert (MAP_INT_STRING::value_type (3, "Three"));
27:
28:    // Insert a pair using function make_pair
29:    mapIntToString.insert (make_pair (-1, "Minus One"));
30:
31:    // Insert a pair object directly
32:    mapIntToString.insert (pair <int, string>(1000, "One Thousand"));
33:
34:    // Insert using an array-like syntax for inserting key-value pairs
35:    mapIntToString [1000000] = "One Million";
36:
37:    cout << "The map contains " << mapIntToString.size ();
38:    cout << " key-value pairs. They are: " << endl;
39:    DisplayContents(mapIntToString);
40:
41:    // instantiate a multimap that is a copy of a map
42:    MMAP_INT_STRING mmapIntToString(mapIntToString.cbegin(),
43:                            mapIntToString.cend());
44:
45:    // The insert function works the same way for multimap too
46:    // A multimap can store duplicates - insert a duplicate
47:    mmapIntToString.insert (make_pair (1000, "Thousand"));
48:
49:    cout << endl << "The multimap contains " << mmapIntToString.size ();
50:    cout << " key-value pairs. They are: " << endl;
51:    cout << "The elements in the multimap are: " << endl;
52:    DisplayContents(mmapIntToString);
53:
54:    // The multimap can also return the number of pairs with the same key
55:    cout << "The number of pairs in the multimap with 1000 as their key: "
56:         << mmapIntToString.count (1000) << endl;
57:
58:    return 0;
59: }
```

▼ 输出：

```
The map contains 4 key-value pairs. They are:
-1 -> Minus One
3 -> Three
1000 -> One Thousand
1000000 -> One Million


The multimap contains 5 key-value pairs. They are:
The elements in the multimap are:
-1 -> Minus One
3 -> Three
1000 -> One Thousand
1000 -> Thousand
1000000 -> One Million

The number of pairs in the multimap with 1000 as their key: 2
```

▼ 分析：

第 7 行和第 8 行给模板类 map 和 multimap 的实例指定了别名，这样可让代码看起来更简单（避免使用模板带来的混乱）。第 10～19 行是针对 map 和 multimap 改写后的 DisplayContents()，它使用迭代器来访问表示键的 first 以及表示值的 second。第 26～32 行演示了使用重载方法 insert()将键-值对插入到 map 中的各种方式。第 35 行表明，可使用数组语法（运算符[]）在 map 中插入元素。请注意，这些插入方式也适用于 multimap，如第 47 行所示，这行代码在 multimap 中插入了一个重复的元素。有趣的是，第 42 和 43 行使用 map 的内容初始化 multimap。输出表明，这两种容器都自动根据键升序排列输入的键-值对；输出还表明，multimap 可存储两个键相同（这里是 1000）的键-值对。第 56 行使用了 multimap::count()，这个函数指出有多少个元素包含指定的键。

提示

关键字 auto 是否导致编译错误？

在程序清单 20.2 中，函数 DisplayContents()使用了 C++11 关键字 auto 来指定迭代器类型，如第 13 所示。对于这个示例及后面的示例，要使用不遵循 C++11 的编译器对其进行编译，需要将 auto 替换为显式类型。

因此，如果您使用的是较旧的编译器，需要将 DisplayContents()修改成下面这样：

```
template <typename T>
void DisplayContents (const T& Input)
{
   for (T::const_iterator iElement = Input.begin ()
      ; iElement != Input.end ()
      ; ++ iElement )
      cout << iElement->first << " -> " << iElement->second
<< endl;

   cout << endl;
}
```

20.2.3 在 STL map 或 multimap 中查找元素

诸如 map 和 multimap 等关联容器都提供了成员函数 find()，它让您能够根据给定的键查找值。find()总是返回一个迭代器：

```
multimap <int, string>::const_iterator iPairFound = mapIntToString.find(Key);
```

您应首先检查该迭代器，确保 find()已成功，再使用它来访问找到的值：

```
if (iPairFound != mapIntToString.end())
{
   cout << "Key " << iPairFound->first << " points to Value: ";
   cout << iPairFound->second << endl;
}
else
   cout << "Sorry, pair with key " << Key << " not in map" << endl;
```

提示

如果您使用的编译器遵循 C++11 标准，可使用关键字 auto 来简化迭代器声明：

```
auto iPairFound = mapIntToString.find(Key);
```

编译器将根据 map::find()的返回类型自动推断出迭代器的类型。

程序清单 20.3 演示了 multimap::find 的用法。

程序清单 20.3 使用成员函数 find 在 map 中查找键-值对

```
 0: #include <map>
 1: #include <iostream>
 2: #include <string>
 3: using namespace std;
 4:
 5: template <typename T>
 6: void DisplayContents (const T& Input)
 7: {
 8:    for(auto iElement = Input.cbegin () // auto and cbegin(): C++11
 9:       ; iElement != Input.cend()  // cend() is new in C++11
10:       ; ++ iElement )
11:       cout << iElement->first << " -> " << iElement->second << endl;
12:
13:    cout << endl;
14: }
15:
16: int main()
17: {
18:     map<int, string> mapIntToString;
19:
20:     mapIntToString.insert(make_pair(3, "Three"));
21:     mapIntToString.insert(make_pair(45, "Forty Five"));
22:     mapIntToString.insert(make_pair(-1, "Minus One"));
23:     mapIntToString.insert(make_pair(1000, "Thousand"));
24:
25:     cout << "The multimap contains " << mapIntToString.size();
26:     cout << " key-value pairs. They are: " << endl;
27:
28:     // Print the contents of the map to the screen
29:     DisplayContents(mapIntToString);
30:
31:    cout << "Enter the key you wish to find: ";
32:    int Key = 0;
33:    cin >> Key;
34:
35:    auto iPairFound = mapIntToString.find(Key);
36:    if (iPairFound != mapIntToString.end())
37:    {
38:       cout << "Key " << iPairFound->first << " points to Value: ";
39:       cout << iPairFound->second << endl;
40:    }
41:    else
42:       cout << "Sorry, pair with key " << Key << " not in map" << endl;
43:
44:     return 0;
45: }
```

▼ 输出：

```
The multimap contains 4 key-value pairs. They are:
-1 -> Minus One
3 -> Three
45 -> Forty Five
1000 -> Thousand

Enter the key you wish to find: 45
Key 45 points to Value: Forty Five
```

再次运行的输出（这次 find()没有找到匹配的元素）：

```
The multimap contains 4 key-value pairs. They are:
-1 -> Minus One
3 -> Three
45 -> Forty Five
1000 -> Thousand

Enter the key you wish to find: 2011
Sorry, pair with key 2011 not in map
```

▼ 分析：

在 main()函数中，第 20～23 行使用几个键-值对填充一个 map，每个键-值对都将整型键映射到字符串值。用户指定要用于在 map 中查找元素的键后，第 35 行使用函数 find()根据指定的键在 map 中进行查找。map::find()总是返回一个迭代器，核实 find()操作成功总是明智的，为此可将返回的迭代器与 end()进行比较，如第 36 行所示。如果该迭代器有效，则通过成员 second 访问值，如第 39 行所示。第二次运行时，您指定的键 2011 不包含在 map 中，因此显示了一条错误消息。

警告 使用 find()返回的迭代器之前，务必通过检查迭代器核实 find()操作成功了。

20.2.4 在 STL multimap 中查找元素

如果程序清单 20.3 使用的是 multimap，容器可能包含多个键相同的键-值对，因此需要找到与指定键对应的所有值。为此，可使用 multimap::count()确定有多少个值与指定的键对应，再对迭代器递增，以访问这些相邻的值：

```
auto iPairFound = mmapIntToString.find(Key);

// Check if "find" succeeded
if(iPairFound != mmapIntToString.end())
{
    // Find the number of pairs that have the same supplied key
    size_t nNumPairsInMap = mmapIntToString.count(1000);

    for( size_t nValuesCounter = 0
        ; nValuesCounter < nNumPairsInMap  // stay within bounds
        ; ++ nValuesCounter )
    {
        cout << "Key: " << iPairFound->first;  // key
        cout << ", Value [" << nValuesCounter << "] = ";
        cout << iPairFound->second << endl;    // value

        ++ iPairFound;
    }
}
else
    cout << "Element not found in the multimap";
```

20.2.5 删除 STL map 或 multimap 中的元素

map 和 multimap 提供了成员函数 erase()，该函数删除容器中的元素。调用 erase 函数时将键作为参数，这将删除包含指定键的所有键-值对：

```
mapObject.erase (key);
```

erase 函数的另一种版本接受迭代器作为参数，并删除迭代器指向的元素：

```
mapObject.erase (iElement);
```

还可使用迭代器指定边界，从而将指定范围内的所有元素都从 map 或 multimap 中删除：

```
mapObject.erase (iLowerBound, iUpperBound);
```

程序清单 20.4 演示了函数 erase()的用法。

程序清单 20.4 删除 multimap 中的元素

```
 0: #include<map>
 1: #include<iostream>
 2: #include<string>
 3: using namespace std;
 4:
 5: template<typename T>
 6: void DisplayContents(const T& Input)
 7: {
 8:    for(auto iElement = Input.cbegin () // auto and cbegin(): C++11
 9:       ; iElement != Input.cend()  // cend() is new in C++11
10:       ; ++ iElement )
11:       cout<< iElement->first<< " -> "<< iElement->second<< endl;
12:
13:    cout<< endl;
14: }
15:
16: int main()
17: {
18:    multimap<int, string>  mmapIntToString;
19:
20:    // Insert key-value pairs into the multimap
21:    mmapIntToString.insert(make_pair(3, "Three"));
22:    mmapIntToString.insert(make_pair(45, "Forty Five"));
23:    mmapIntToString.insert(make_pair(-1, "Minus One"));
24:    mmapIntToString.insert(make_pair(1000, "Thousand"));
25:
26:    // Insert duplicates into the multimap
27:    mmapIntToString.insert(make_pair(-1, "Minus One"));
28:    mmapIntToString.insert(make_pair(1000, "Thousand"));
29:
30:    cout<< "The multimap contains "<< mmapIntToString.size();
31:    cout<< " key-value pairs. "<< "They are: "<< endl;
32:    DisplayContents(mmapIntToString);
33:
34:    // Erasing an element with key as -1 from the multimap
35:    auto NumPairsErased = mmapIntToString.erase(-1);
36:    cout<< "Erased " << NumPairsErased << " pairs with -1 as key."<< endl;
37:
38:    // Erase an element given an iterator from the multimap
39:    auto iPairLocator = mmapIntToString.find(45);
40:    if(iPairLocator != mmapIntToString.end())
41:    {
42:       mmapIntToString.erase(iPairLocator);
43:       cout<< "Erased a pair with 45 as key using an iterator"<< endl;
```

```
44:     }
45:
46:     // Erase a range from the multimap...
47:     cout<< "Erasing the range of pairs with 1000 as key."<< endl;
48:     mmapIntToString.erase( mmapIntToString.lower_bound(1000)
49:                            , mmapIntToString.upper_bound(1000) );
50:
51:     cout<< "The multimap now contains "<< mmapIntToString.size();
52:     cout<< " key-value pair(s)."<< "They are: "<< endl;
53:     DisplayContents(mmapIntToString);
54:
55:     return 0;
56: }
```

▼ **输出：**

```
The multimap contains 6 key-value pairs. They are:
-1 -> Minus One
-1 -> Minus One
3 -> Three
45 -> Forty Five
1000 -> Thousand
1000 -> Thousand

Erased 2 pairs with -1 as key.
Erased a pair with 45 as key using an iterator
Erasing the range of pairs with 1000 as key.
The multimap now contains 1 key-value pair(s).They are:
3 -> Three
```

▼ **分析：**

第 21～28 行将一些插入到 multimap 中，有些是重复的（因为 multimap 不同于 map，它允许插入重复的元素）。将键-值对插入到 multimap 后，第 35 行调用接受一个键作为参数的 erase 函数，将所有包含该键（-1）的键-值对都删除。multimap::erase(Key)的返回值为删除的元素数，这显示到了屏幕上。第 39 行使用 find(45)返回的迭代器，将键为 45 的键-值对从 multimap 中删除。第 48 和 49 行表明，可使用 lower_bound()和 upper_bound()来指定范围，从而将包含特定键的所有键-值对都删除。

20.3 提供自定义的排序谓词

map 和 multimap 的模板定义包含第 3 个参数，该参数是确保 map 能够正常工作的排序谓词。如果没有指定这个参数（如前面的示例所示），将使用 std::less <>提供的默认排序标准，该谓词使用<运算符来比较两个对象。

要提供不同的排序标准，可编写一个二元谓词——实现了 operator()的类或结构：

```
template<typename KeyType>
struct Predicate
{
   bool operator()(const KeyType& key1, const KeyType& key2)
   {
      // your sort priority logic here
   }
};
```

对于键类型为 std::string 的 map，默认排序谓词 std::less<T>导致根据 std::string 类定义的<运算符进行排序，因此区分大小写。很多应用程序（如电话簿）要求执行插入和搜索操作时不区分大小写，

为满足这种需求，一种解决方案是在实例化 map 时提供一个排序谓词，它根据不区分大小写的比较结果返回 true 或 false，如下所示：

```
map <keyType, valueType, Predicate> mapObject;
```

程序清单 20.5 对此做了更详细的解释。

程序清单 20.5 提供自定义排序谓词——电话簿应用程序

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

template <typename T>
void DisplayContents (const T& Input)
{
   for(auto iElement = Input.cbegin () // auto and cbegin(): C++11
      ; iElement != Input.cend()  // cend() is new in C++11
      ; ++ iElement )
      cout << iElement->first << " -> " << iElement->second << endl;

   cout << endl;
}

struct PredIgnoreCase
{
    bool operator()(const string& str1, const string& str2) const
    {
        string str1NoCase(str1), str2NoCase(str2);
        std::transform(str1.begin(), str1.end(), str1NoCase.begin(),
        tolower);
        std::transform(str2.begin(), str2.end(), str2NoCase.begin(),
        tolower);

        return(str1NoCase< str2NoCase);
    };
};

typedef map<string, string> DIRECTORY_WITHCASE;
typedef map<string, string, PredIgnoreCase> DIRECTORY_NOCASE;

int main()
{
    // Case-insensitive directory: case of string-key plays no role
    DIRECTORY_NOCASE dirCaseInsensitive;

    dirCaseInsensitive.insert(make_pair("John", "2345764"));
    dirCaseInsensitive.insert(make_pair("JOHN", "2345764"));
    dirCaseInsensitive.insert(make_pair("Sara", "42367236"));
    dirCaseInsensitive.insert(make_pair("Jack", "32435348"));

    cout << "Displaying contents of the case-insensitive map:"<< endl;
   DisplayContents(dirCaseInsensitive);

    // Case-sensitive map: case of string-key affects insertion & search
    DIRECTORY_WITHCASE dirCaseSensitive(dirCaseInsensitive.begin()
      , dirCaseInsensitive.end());

    cout << "Displaying contents of the case-sensitive map:"<< endl;
   DisplayContents(dirCaseSensitive);
```

```
52:     // Search for a name in the two maps and display result
53:     cout << "Please enter a name to search: "<< endl<< "> ";
54:     string strNameInput;
55:     cin >> strNameInput;
56:
57:     // find in the map...
58:     auto iPairInNoCaseDir = dirCaseInsensitive.find(strNameInput);
59:     if(iPairInNoCaseDir != dirCaseInsensitive.end())
60:     {
61:         cout << iPairInNoCaseDir->first<< "'s number in the case-insensi-
            tive";
62:         cout << " directory is: "<< iPairInNoCaseDir->second<< endl;
63:     }
64:     else
65:     {
66:         cout << strNameInput<< "'s number not found ";
67:         cout << "in the case-insensitive directory"<< endl;
68:     }
69:
70:     // find in the case-sensitive map...
71:     auto iPairInCaseSensDir = dirCaseSensitive.find(strNameInput);
72:     if(iPairInCaseSensDir != dirCaseSensitive.end())
73:     {
74:         cout<<iPairInCaseSensDir->first<<"'s number in the case-sensitive";
75:         cout << " directory is: "<< iPairInCaseSensDir->second<< endl;
76:     }
77:     else
78:     {
79:         cout << strNameInput<< "'s number was not found ";
80:         cout << "in the case-sensitive directory"<< endl;
81:     }
82:
83:     return 0;
84:}
```

▼ 输出：

```
Displaying contents of the case-insensitive map:
Jack -> 32435348
John -> 2345764
Sara -> 42367236

Displaying contents of the case-sensitive map:
Jack -> 32435348
John -> 2345764
Sara -> 42367236

Please enter a name to search:
> sara
Sara's number in the case-insensitive directory is: 42367236
sara's number was not found in the case-sensitive directory
```

▼ 分析：

程序清单 20.5 实现了两个电话簿。其中一个在实例化时没有提供谓词，因此使用默认谓词 std::less <T>，导致根据区分大小写的 std::string::operator<进行排序和查找。实例化另一个电话簿时指定了谓词 PreIgnoreCase，该谓词是在第 17～27 行定义的，它将两个字符串转换为小写后再进行比较。从输出可知，在这两个 map 中查找 sara 时，在不区分大小写的 map 中能够找到 Sara，但在区分大小写的 map 中找不到该条目。

注意

在程序清单 20.5 中，也可将结构 PredIgnoreCase 声明为类。在这种情况下，需要给 operator()加上关键字 public。在 C++编译器看来，结构类似于类，但成员默认为公有的，继承方式也默认为公有的。

这个示例演示了如何使用谓词来定制 map 的行为，它还表明键可以是任何类型的，而程序员可提供一个谓词以定义这种类型的 map 的行为。注意到这里使用的谓词是一个实现了运算符()的结构，但也可以是类。这种也可作为函数的对象被称为函数对象（或 Functor），这将在第 21 章更详细地介绍。

注意

std::map 非常适合用于存储键-值对，让您能够根据指定的键查找值。在查找方面，map 的性能确实比 STL vector 和 list 高，但查找速度会随元素的增加而降低。map 的复杂度为对数，即所需的时间与 map 包含的元素数的对数成正比。

简单地说，对数复杂度意味着 std::map 和 std::set 等容器包含的元素数从 10000 减少到 100 时，速度将提高一倍。

vector 的元素未经排序，其查找复杂度为线性，这意味着当元素数从 10000 减少到 100 时，速度将提高 100 倍。

虽然对数复杂度已相当不错，但别忘了，map、multimap、set 和 multiset 等容器在插入时对元素进行排序，因此其插入速度更慢。因此，大家一直在寻找速度更快的容器，数学家和程序员找到了最优秀的容器，其插入和查找时间固定。散列表就是一种这样的容器，其插入时间是固定的，根据键查找元素的时间也几乎是固定的（大多数情况下如此），而不受容器大小的影响。

C++11

基于散列表的 STL 键-值对容器 std::unordered_map

从 C++11 起，STL 支持散列映射——std::unordered_map 类。要使用这个模板类，需要包含头文件<unordered_map>：

```
#include<unordered_map>
```

unordered_map 的平均插入和删除时间是固定的，查找元素的时间也是固定的。

20.3.1 散列表的工作原理

这一直是博士论文探讨的主题，本书不打算详细讨论它，只简要地讨论散列表的工作原理。

可将散列表视为一个键-值对集合，根据给定的键，可找到相应的值。散列表与简单映射的区别在于，散列表将键-值对存储在桶中，每个桶都有索引，指出了它在散列表中的相对位置（类似于数组）。这种索引是使用散列函数根据键计算得到的：

```
Index = HashFunction(Key, TableSize);
```

使用 find()根据键查找元素时，将使用 HashFunction()计算元素的位置，并返回该位置的值，就像数组返回其存储的元素那样。如果 HashFunction()不佳，将导致多个元素的索引相同，进而存储在同一个桶中，即桶变成了元素列表。这种情形被称为冲突（collision），它会降低查找速度，使查找时间不再是固定的。

20.3.2 使用 C++11 散列表 unordered_map 和 unordered_multimap

从使用的角度看，这两种容器与 std::map 和 std::multimap 差别不大，可以类似的方式执行实例化、插入和查找：

```
// instantiate unordered_map of int to string:
unordered_map<int, string> umapIntToString;

// insert()
umapIntToString.insert(make_pair(1000, "Thousand"));

// find():
auto iPairThousand = umapIntToString.find(1000);
cout << iPairThousand->first << " —> " << iPairThousand->second << endl;

// find value using array semantics:
cout << "umapIntToString[1000] = " << umapIntToString[1000] << endl;
```

然而，一个重要的特点是，unordered_map 包含一个散列函数，用于计算排列顺序：

```
unordered_map<int, string>::hasher HFn =
        umapIntToString.hash_function();
```

要获悉键对应的索引，可调用该散列函数，并将键传递给它：

```
size_t HashingValue1000 = HFn(1000);
```

鉴于 unordered_map 将键-值对存储在桶中，在元素数达到或接近桶数时，它将自动执行负载均衡：

```
cout << "Load factor: " << umapIntToString.load_factor() << endl;
cout << "Max load factor = " << umapIntToString.max_load_factor() << endl;
cout << "Max bucket count = " << umapIntToString.max_bucket_count() << endl;
```

load_factor()指出了 unordered_map 桶的填满程度。因插入元素导致 load_factor()超过 max_load_factor()时，unordered_map 将重新组织以增加桶数，并重建散列表，如程序清单 20.6 所示。

提示

std::unordered_map 与 unordered_map 类似，只是可存储多个键相同的键-值对。

std::unordered_multimap 的用法与 std::multimap 很像，但包含一些散列表特有的函数，如程序清单 20.6 所示。

程序清单 20.6 实例化 STL 散列表实现 unordered_map，并使用 insert()、find()、size()、max_bucket_count()、load_factor()和 max_load_factor()

```
 0: #include<iostream>
 1: #include<string>
 2: #include<unordered_map>
 3: using namespace std;
 4:
 5: template <typename T1, typename T2>
 6: void DisplayUnorderedMap(unordered_map<T1, T2>& Input)
 7: {
 8:    cout << "Number of pairs, size(): " << Input.size() << endl;
 9:    cout << "Max bucket count = " << Input.max_bucket_count() << endl;
10:    cout << "Load factor: " << Input.load_factor() << endl;
11:    cout << "Max load factor = " << Input.max_load_factor() << endl;
12:    cout << "Unordered Map contains: " << endl;
13:
14:    for(auto iElement = Input.cbegin() // auto, cbegin: c++11
15:       ; iElement != Input.cend()  // cend() is new in C++11
16:       ; ++ iElement )
17:       cout<< iElement->first<< " -> "<< iElement->second<< endl;
18: }
19:
20: int main()
21: {
22:    unordered_map<int, string> umapIntToString;
23:    umapIntToString.insert(make_pair(1, "One"));
24:    umapIntToString.insert(make_pair(45, "Forty Five"));
25:    umapIntToString.insert(make_pair(1001, "Thousand One"));
```

```
26:    umapIntToString.insert(make_pair(-2, "Minus Two"));
27:    umapIntToString.insert(make_pair(-1000, "Minus One Thousand"));
28:    umapIntToString.insert(make_pair(100, "One Hundred"));
29:    umapIntToString.insert(make_pair(12, "Twelve"));
30:    umapIntToString.insert(make_pair(-100, "Minus One Hundred"));
31:
32:    DisplayUnorderedMap<int, string>(umapIntToString);
33:
34:    cout << "Inserting one more element" << endl;
35:    umapIntToString.insert(make_pair(300, "Three Hundred"));
36:    DisplayUnorderedMap<int, string>(umapIntToString);
37:
38:    cout << "Enter key to find for: ";
39:    int Key = 0;
40:    cin >> Key;
41:
42:    auto iElementFound = umapIntToString.find(Key);
43:    if (iElementFound != umapIntToString.end())
44:    {
45:       cout << "Found! Key " << iElementFound->first << " points to value ";
46:       cout << iElementFound->second << endl;
47:    }
48:    else
49:       cout << "Key has no corresponding value in unordered map!" << endl;
50:
51:    return 0;
52: }
```

▼ 输出：

```
Number of pairs, size(): 8
Max bucket count = 8
Load factor: 1
Max load factor = 1
Unordered Map contains:
-1000 -> Minus One Thousand
1001 -> Thousand One
1 -> One
-100 -> Minus One Hundred
45 -> Forty Five
-2 -> Minus Two
12 -> Twelve
100 -> One Hundred
Inserting one more element
Number of pairs, size(): 9
Max bucket count = 64
Load factor: 0.140625
Max load factor = 1
Unordered Map contains:
1 -> One
-1000 -> Minus One Thousand
1001 -> Thousand One
-100 -> Minus One Hundred
45 -> Forty Five
-2 -> Minus Two
300 -> Three Hundred
12 -> Twelve
100 -> One Hundred
100 -> One Hundred
Enter key to find for: 300
Found! Key 300 points to value Three Hundred
```

▼ **分析：**

从输出可知，该程序中的 unordered_map 最初 8 个键-值对和 8 个桶，但插入第 9 个元素时自动调整了大小，而桶数增加到 64 个。注意到第 9～11 行使用了方法 max_bucket_count()、load_factor()和 max_load_factor()。除此之外，其他代码与使用 std::map 时差别不大，这包括使用 find()的第 42 行。与用于 std::map 时一样，该函数返回一个迭代器，需要将其与 end()进行比较，以确定操作成功。

警告

无论使用的键是什么，都不要编写依赖于 unordered_map 中元素排列顺序的代码。在 unordered_map，元素相对顺序取决于众多因素，其中包括键、插入顺序、桶数等。
这些容器为提高查找性能进行了优化，遍历其中的元素时，不要依赖于元素的排列顺序。

注意

在不发生冲突的情况下，std::unordered_map 的插入和查找时间几乎是固定的，不受包含的元素数的影响。然而，这并不意味着它优于在各种情形下复杂度都为对数的 std::map。在包含的元素不太多的情况下，固定时间可能长得多，导致 std::unordered_map 的速度比 std::map 慢。
选择容器类型时，务必执行模拟实际情况的基准测试。

应该	不应该
需要存储键-值对且键是唯一的时，务必使用 map。 需要存储键-值对且键可能重复时（如电话簿），务必使用 multimap。 请牢记，与其他 STL 容器一样，map 和 multimap 都有成员方法 size()，它指出容器包含多少个键-值对。 必须确保插入和查找时间固定时（通常是包含的元素非常多时），务必使用 unordered_map 或 unordered_multimap。	别忘了，multimap::count(Key)指出容器中有多少个元素的键为 Key。 别忘了检查 find()的返回值——将其与容器的 end()进行比较。

20.4 总结

本章介绍了 STL map 和 multimap 的用法及其重要的成员函数和特征，这些容器的复杂度为对数。STL 还提供了散列表容器 unordered_map 和 unordered_multimap，这些容器的 insert()和 find()性能不受容器大小的影响。您还学习了使用谓词定制排序标准的重要性，程序清单 20.5 的电话簿应用程序演示了这一点。

20.5 问与答

问：如何声明一个其元素按降序排列的整型 set？

答：map <int>定义一个整型 map，这种 map 使用默认排序谓词 std::less <T>将元素按升序排列，也可将其定义为 map <int, less <int> >。要按降序排列，应将 map 定义为 map <int, greater <int> >。

问：如果在一个字符串 map 中插入字符串 Jack 两次，将发生什么情况？

答：map 不能存储重复的值，因此 std::map 类的实现不允许再次插入 Jack。

问：在前一个例子中，如果需要两个 Jack，该如何办？

答：map 不能存储重复的值，应选择使用 multimap。

问：multimap 的哪一个成员函数返回容器中有多少个元素包含特定的值？

答：count (value)函数。

问：我使用函数 find()在 map 中找到了一个元素，并有一个指向该元素的迭代器。能否使用这个迭代器来修改它指向的元素的值？

答：不能。有些 STL 实现可能允许用户通过迭代器（如函数 find()返回的迭代器）修改元素的值，但不应这样做。应将指向 map 中元素的迭代器视为 const 迭代器，即使 STL 实现没有强制这样做。

问：我使用的编译器较老，不支持关键字 auto。该如何声明一个变量，用于存储 map::find()的返回值？

答：迭代器总是使用下述语法定义的：

```
container<Type>::iterator variableName;
```

对于指向整型 map 中元素的迭代器，其声明如下：

```
std::map<int>::iterator iPairFound = mapIntegers.find(1000);
if (iPairFound != mapIntegers.end())
   ;// Do Something
```

20.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

20.6.1 测验

1. 使用 map <int>声明整型 map 时，排序标准将由哪个函数提供？
2. 在 multimap 中，重复的值以什么方式出现？
3. map 和 multimap 的哪个成员函数指出容器包含多少个元素？
4. 在 map 中的什么地方可以找到重复的值？

20.6.2 练习

1. 编写一个应用程序来实现电话簿，它不要求人名是唯一的。应选择哪种容器？写出容器的定义。
2. 下面是电话簿应用程序中一个 map 的定义：

```
map <wordProperty, string, fPredicate> mapWordDefinition;
```

其中 wordProperty 是一个结构：

```
struct wordProperty
{
    string strWord;
    bool bIsFromLatin;
};
```

请定义二元谓词 fPredicate，用于帮助该 map 根据 wordProperty 键包含的 string 属性对元素进行排序。

3. map 不接受重复元素，而 multimap 接受；请编写一个简单程序来演示这一点。

第 21 章
理解函数对象

函数对象（也叫 functor）听起来陌生或难以理解，但它们是 C++实体，即使您没有用过，也很可能见过，只是您没有意识到而已。

在本章中，您将学习：

- 函数对象的概念；
- 将函数对象用作谓词；
- 如何使用函数对象实现一元和二元谓词。

21.1 函数对象与谓词的概念

从概念上说，函数对象是用作函数的对象；但从实现上说，函数对象是实现了 operator()的类的对象。虽然函数和函数指针也可归为函数对象，但实现了 operator()的类的对象才能保存状态（即类的成员属性的值），才能用于标准模板库（STL）算法。

C++程序员常用于 STL 算法的函数对象可分为下列两种类型。

- **一元函数**：接受一个参数的函数，如 f(x)。如果一元函数返回一个布尔值，则该函数称为谓词。
- **二元函数**：接受两个参数的函数，如 f(x, y)。如果二元函数返回一个布尔值，则该函数称为二元谓词。

返回布尔类型的函数对象通常用于需要进行判断的算法。组合两个函数对象的函数对象称为自适应函数对象。

21.2 函数对象的典型用途

可以通过长篇大论从理论上解释函数对象，也可通过小型应用程序看看函数对象是什么样的及其工作原理。下面将采取后一种实用方法，直接看看如何在 C++编程中使用函数对象。

21.2.1 一元函数

只对一个参数进行操作的函数称为一元函数。一元函数的功能可能很简单，如在屏幕上显示元素，如下所示：

```
// A unary function
template <typename elementType>
void FuncDisplayElement (const elementType & element)
{
    cout << element << ' ';
};
```

函数 FuncDisplayElement 接受一个类型为模板化类型 elementType 的参数，并使用控制台输出语句 std::cout 将该参数显示出来。该函数也可采用另一种表现形式，即其实现包含在类或结构的 operator()中：

```
// Struct that can behave as a unary function
template <typename elementType>
struct DisplayElement
{
    void operator () (const elementType& element) const
    {
        cout << element << ' ';
    }
};
```

提示

DisplayElement 是一个结构，如果它是类，则必须给 operator()指定访问限定符 public。结构相当于成员默认为公有的类。

这两种实现都可用于 STL 算法 for_each，将集合中的内容显示在屏幕上，每次显示一个元素，如程序清单 21.1 所示。

程序清单 21.1 使用一元函数将集合的内容显示在屏幕上

```
 0: #include <algorithm>
 1: #include <iostream>
 2: #include <vector>
 3: #include <list>
 4:
 5: using namespace std;
 6:
 7: // struct that behaves as a unary function
 8: template <typename elementType>
 9: struct DisplayElement
10: {
11:     void operator () (const elementType& element) const
12:     {
13:         cout << element << ' ';
14:     }
15: };
16:
17: int main ()
18: {
19:     vector <int> vecIntegers;
20:
21:     for (int nCount = 0; nCount < 10; ++ nCount)
22:         vecIntegers.push_back (nCount);
23:
24:     list <char> listChars;
25:
26:     for (char nChar = 'a'; nChar < 'k'; ++nChar)
27:         listChars.push_back (nChar);
28:
29:     cout << "Displaying the vector of integers: " << endl;
30:
31:     // Display the array of integers
32:     for_each ( vecIntegers.begin ()    // Start of range
33:           , vecIntegers.end ()         // End of range
34:           , DisplayElement <int> () ); // Unary function object
35:
36:     cout << endl << endl;
37:     cout << "Displaying the list of characters: " << endl;
38:
```

```
39:     // Display the list of characters
40:     for_each ( listChars.begin ()          // Start of range
41:           , listChars.end ()          // End of range
42:           , DisplayElement <char> () );// Unary function object
43:
44:     return 0;
45: }
```

▼ 输出：

```
Displaying the vector of integers:
0 1 2 3 4 5 6 7 8 9

Displaying the list of characters:
a b c d e f g h i j
```

▼ 分析：

第 8～15 行包含函数对象 DisplayElement，它实现了 operator()。在第 32～34 行，将这个函数对象用于了 STL 算法 std::for_each。for_each 接受 3 个参数：第 1 个参数指定范围的起点，第 2 个参数指定范围的终点，第 3 个参数是要对指定范围内的每个元素调用的函数。换句话说，这些代码将对 vector verIntergers 中的每个元素调用 DisplayElement::operator()。注意，在这里可不使用结构 DisplayElement，而使用 FuncDisplayElement，其效果相同。第 40～42 行显示了字符 list 的内容。

提示

C++11 引入了 lambda 表达式，即匿名函数对象。

在程序清单 21.1 中，如果不使用结构 struct DisplayElement<T>，而使用 lambda 表达式，可极大地简化代码。为此，删除定义该结构的代码，并使用如下代码替换 main()函数中使用该结构的 3 行代码（第 32-34 行）：

```
// Display the array of integers using lambda expression
for_each ( vecIntegers.begin ()     // Start of range
      , vecIntegers.end ()          // End of range
      , [](int& Element) {cout << element << ' '; } ); //
      Lambda expression
```

引入 lambda 表达式是 C++的一项重大改进，请务必阅读第 22 章，更深入地学习 lambda 表达式。在程序清单 22.1 中，在 for_each 中使用了 lambda 表达式来显示容器的内容，而不像程序清单 21.1 那样使用函数对象。

如果能够使用结构的对象来存储信息，则使用在结构中实现的函数对象的优点将显现出来。这是 FuncDisplayElement 不像结构那么强大的地方，因为结构除 operator()外还可以有成员属性。下面是一个稍做修改的版本，它使用了成员属性：

```
template <typename elementType>
struct DisplayElementKeepCount
{
    int Count;

    DisplayElementKeepCount ()  // constructor
    {
        Count = 0;
    }

    void operator () (const elementType& element)
    {
        ++ Count;
        cout << element << ' ';
    }
};
```

在上述代码中，DisplayElementKeepCount 对前一个版本稍做了修改。operator()不再是 const 成员函数，因为它对成员 Count 进行递增（修改），以记录自己被调用用于显示数据的次数。该计数是通过公有成员属性 Count 暴露的。程序清单 21.2 演示了使用可保存状态的函数对象的优点。

程序清单 21.2 使用函数对象存储状态

```
 0: #include<algorithm>
 1: #include<iostream>
 2: #include<vector>
 3: using namespace std;
 4:
 5: template<typename elementType>
 6: struct DisplayElementKeepCount
 7: {
 8:    int Count;
 9:
10:    // Constructor
11:    DisplayElementKeepCount() : Count(0) {}
12:
13:    // Display the element, hold count!
14:    void operator()(const elementType& element)
15:    {
16:       ++ Count;
17:       cout << element<< ' ';
18:    }
19: };
20:
21: int main()
22: {
23:    vector<int> vecIntegers;
24:    for(int nCount = 0; nCount< 10; ++ nCount)
25:       vecIntegers.push_back(nCount);
26:
27:    cout << "Displaying the vector of integers: "<< endl;
28:
29:    // Display the array of integers
30:    DisplayElementKeepCount<int> Result;
31:    Result = for_each( vecIntegers.begin()   // Start of range
32:                     , vecIntegers.end()     // End of range
33:                     , DisplayElementKeepCount<int>() );// function object
34:
35:    cout << endl<< endl;
36:
37:    // Use the state stores in the return value of for_each!
38:    cout << "'" << Result.Count << "' elements were displayed!"<< endl;
39:
40:    return 0;
41: }
```

▼ 输出：

```
Displaying the vector of integers:
0 1 2 3 4 5 6 7 8 9

'10' elements were displayed!
```

▼ 分析：

这个例子与程序清单 21.1 所示示例的最大区别在于，将 DisplayElementKeepCount 用作 for_each 的返回值。算法 for_each()对容器中的每个元素调用结构 DisplayElementKeepCount 实现的 operator()

时，operator()显示该元素，并递增存储在 Count 中的内部计数。在 for_each 执行完毕后，第 38 行使用这个对象指出显示了多少个元素。注意，在这种情况下，如果使用普通函数而不是在结构中实现的函数，将无法以如此直接的方式提供这种功能。

21.2.2 一元谓词

返回布尔值的一元函数是谓词。这种函数可供 STL 算法用于判断。程序清单 21.3 所示的谓词判断输入元素是否为初始值的整数倍。

程序清单 21.3 一个一元谓词，它判断一个数字是否为另一个数字的整数倍

```
 0: // A structure as a unary predicate
 1: template <typename numberType>
 2: struct IsMultiple
 3: {
 4:    numberType Divisor;
 5:
 6:    IsMultiple (const numberType& divisor)
 7:    {
 8:       Divisor = divisor;
 9:    }
10:
11:    bool operator () (const numberType& element) const
12:    {
13:       // Check if the divisor is a multiple of the divisor
14:       return ((element % Divisor) == 0);
15:    }
16: };
```

▼ 分析：

这里的 operator()返回布尔值，可用作一元谓词。该结构有一个构造函数，它初始化除数的值。然后用保存在对象中的这个值来判断要比较的元素是否可以被它整除，如 operator()的实现所示，它使用数学运算取模%来返回除法运算的余数。然后将余数与零进行比较，以判断被除数是否为除数的整数倍。

在程序清单 21.4 中，使用了程序清单 21.3 所示的谓词，来判断集合中的数是否为用户输入的除数的整数倍。

程序清单 21.4 在 std::find_if()中使用一元谓词 IsMutilple，在 vector 中查找一个能被用户提供的除数整除的元素

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3: using namespace std;
 4: // Insert struct IsMultiple from Listing 21.3 here
 5: int main ()
 6: {
 7:    vector <int> vecIntegers;
 8:    cout << "The vector contains the following sample values: ";
 9:
10:    // Insert sample values: 25 - 31
11:    for (int nCount = 25; nCount < 32; ++ nCount)
12:    {
13:       vecIntegers.push_back (nCount);
14:       cout << nCount << ' ';
```

```
15:    }
16:    cout << endl << "Enter divisor (> 0): ";
17:    int Divisor = 2;
18:    cin >> Divisor;
19:
20:    // Find the first element that is a multiple of 4 in the collection
21:    auto iElement = find_if ( vecIntegers.begin ()
22:                            , vecIntegers.end ()
23:                            , IsMultiple<int>(Divisor) );
24:
25:    if (iElement != vecIntegers.end ())
26:    {
27:       cout << "First element in vector divisible by " << Divisor;
28:       cout << ": " << *iElement << endl;
29:     }
30:
31:    return 0;
32: }
```

▼ 输出：

```
The vector contains the following sample values: 25 26 27 28 29 30 31
The first element in the vector that is divisible by 4 is: 28
```

▼ 分析：

这个例子首先声明了一个整型 vector，第 11～15 行将一些值插入到该容器中。第 21～23 行的 find_if 使用了一元谓词。这里将函数对象 IsMutilple 初始化为用户提供的除数，find_if 对指定范围内的每个元素调用一元谓词 IsMutilple::operator()。当 operator()返回 true（即元素可被用户提供的除数整除）时，find_if 返回一个指向该元素的迭代器。然后，将 find_if()操作的结果与容器的 end()进行比较，以核实是否找到了满足条件的元素，如第 25 行所示。接下来使用迭代器 iElement 显示该元素的值，如第 28 行所示。

提示 要了解如何使用 lambda 表达式简化程序清单 21.4 所示的程序，请参阅程序清单 22.3。

一元谓词被大量用于 STL 算法中。例如，std::partition 算法使用一元谓词来划分范围，stable_partition 算法也使用一元谓词来划分范围，但保持元素的相对顺序不变。诸如 std::find_if()等查找函数以及 std::remove_if()等删除元素的函数也使用一元谓词，其中 std::remove_if()删除指定范围内满足谓词条件的元素。

21.2.3 二元函数

如果函数 f(x, y)根据输入参数返回一个值，它将很有用。这种二元函数可用于对两个操作数执行运算，如加、减、乘、除等。下面的二元函数返回输入参数的积：

```
template <typename elementType>
class Multiply
{
public:
    elementType operator () (const elementType& elem1,
                             const elementType& elem2)
    {
        return (elem1 * elem2);
    }
};
```

同样，在上述实现中最重要的是 operator()，它接受两个参数并返回它们的积。在 std::transform 等算法中，可使用该二元函数计算两个容器内容的乘积。程序清单 21.5 演示了如何在 std::transform 中使

用该二元函数。

程序清单 21.5 使用二元函数将两个范围相乘

```
 0: #include <vector>
 1: #include <iostream>
 2: #include <algorithm>
 3:
 4: template <typename elementType>
 5: class Multiply
 6: {
 7: public:
 8:     elementType operator () (const elementType& elem1,
 9:         const elementType& elem2)
10:     {
11:         return (elem1 * elem2);
12:     }
13: };
14:
15: int main ()
16: {
17:     using namespace std;
18:
19:     // Create two sample vector of integers with 10 elements each
20:     vector <int> vecMultiplicand, vecMultiplier;
21:
22:     // Insert sample values 0 to 9
23:     for (int nCount1 = 0; nCount1 < 10; ++ nCount1)
24:         vecMultiplicand.push_back (nCount1);
25:
26:     // Insert sample values 100 to 109
27:     for (int nCount2 = 100; nCount2 < 110; ++ nCount2)
28:         vecMultiplier.push_back (nCount2);
29:
30:     // A third container that holds the result of multiplication
31:     vector <int> vecResult;
32:
33:     // Make space for the result of the multiplication
34:     vecResult.resize (10);
35:     transform ( vecMultiplicand.begin (), // range of multiplicands
36:                 vecMultiplicand.end (), // end of range
37:                 vecMultiplier.begin (),  // multiplier values
38:                 vecResult.begin (), // range that holds result
39:                 Multiply <int> () );    // the function that multiplies
40:
41:     cout << "The contents of the first vector are: " << endl;
42:     for (size_t nIndex1 = 0; nIndex1 < vecMultiplicand.size (); ++ nIndex1)
43:         cout << vecMultiplicand [nIndex1] << ' ';
44:     cout << endl;
45:
46:     cout << "The contents of the second vector are: " << endl;
47:     for (size_t nIndex2 = 0; nIndex2 < vecMultiplier.size (); ++nIndex2)
48:         cout << vecMultiplier [nIndex2] << ' ';
49:     cout << endl << endl;
50:
51:     cout << "The result of the multiplication is: " << endl;
52:     for (size_t nIndex = 0; nIndex < vecResult.size (); ++ nIndex)
53:         cout << vecResult [nIndex] << ' ';
54:
55:     return 0;
56: }
```

▼ 输出：

```
The contents of the first vector are:
0 1 2 3 4 5 6 7 8 9
The contents of the second vector are:
100 101 102 103 104 105 106 107 108 109

The result of the multiplication held in the third vector is:
0 101 204 309 416 525 636 749 864 981
```

▼ 分析：

第5～13行包含类Multiply，这与前一个代码示例相同。在这个示例中，使用算法std::transform将两个范围的内容相乘，并将结果存储在第三个范围中。在这里，这三个范围分别存储在类型为std::vector的vecMultiplicand、vecMultiplier和vecResult中。在第35～39行，使用std::transform将vecMultiplicand中的每个元素与vecMultiplier中对应的元素相乘，并将结果存储在vecResult中。乘法运算是通过调用二元函数Multiply::operator()执行的，对源范围和目标范围内的每个元素都调用了该函数。operator()的返回值保存在vecResult中。

这个示例演示了如何使用二元函数对STL容器中的元素执行算术运算。

21.2.4 二元谓词

接受两个参数并返回一个布尔值的函数是二元谓词。这种函数用于诸如std::sort()等STL函数中，程序清单21.6使用了一个二元谓词，它将两个字符串都转换为小写，再对其进行比较。这个谓词可用于对字符串vector进行不区分大小写的排序。

程序清单21.6 对字符串进行不区分大小写排序的二元谓词

```
 0: #include <algorithm>
 1: #include <string>
 2: using namespace std;
 3:
 4: class CompareStringNoCase
 5: {
 6: public:
 7:    bool operator () (const string& str1, const string& str2) const
 8:    {
 9:       string str1LowerCase;
10:
11:       // Assign space
12:       str1LowerCase.resize (str1.size ());
13:
14:       // Convert every character to the lower case
15:       transform (str1.begin (), str1.end (), str1LowerCase.begin (),
16:                  tolower);
17:
18:       string str2LowerCase;
19:       str2LowerCase.resize (str2.size ());
20:       transform (str2.begin (), str2.end (), str2LowerCase.begin (),
21:                  tolower);
22:
23:       return (str1LowerCase < str2LowerCase);
24:    }
25: };
```

▼ 分析：

在operator()中实现的二元谓词中，首先使用std::transform()将输入字符串转换为小写，如第15行

和第 20 行所示；然后使用字符串的比较运算符<进行比较，并返回结果。该二元谓词可用于算法 std::sort()，对包含在字符串 vector 中的动态数组进行排序，如程序清单 21.7 所示。

程序清单 21.7 使用函数对象 CompareStringNoCase 对字符串 vector 进行不区分大小写的排序

```
// Insert class CompareStringNoCase from Listing 21.6 here
#include <vector>
#include <iostream>

template <typename T>
void DisplayContents (const T& Input)
{
   for(auto iElement = Input.cbegin() // auto, cbegin and cend: c++11
      ; iElement != Input.cend ()
      ; ++ iElement )
      cout << *iElement << endl;
}

int main ()
{
   // Define a vector of string to hold names
   vector <string> vecNames;

   // Insert some sample names in to the vector
   vecNames.push_back ("jim");
   vecNames.push_back ("Jack");
   vecNames.push_back ("Sam");
   vecNames.push_back ("Anna");

   cout << "The names in vector in order of insertion: " << endl;
   DisplayContents(vecNames);

   cout << "Names after sorting using default std::less<>: " << endl;
   sort(vecNames.begin(), vecNames.end());
   DisplayContents(vecNames);

   cout << "Names after sorting using predicate that ignores case:" << endl;
   sort(vecNames.begin(), vecNames.end(), CompareStringNoCase());
   DisplayContents(vecNames);

   return 0;
}
```

▼ 输出：

```
The names in vector in order of insertion:
jim
Jack
Sam
Anna
Names after sorting using default std::less<>:
Anna
Jack
Sam
jim
Names after sorting using predicate that ignores case:
Anna
Jack
jim
Sam
```

▼ 分析：

输出显示了 vector 在三个阶段的内容。第一次按插入顺序显示内容。第二次是在使用默认排序谓词 less<T>重新排序（如第 28 行所示）后进行的；输出表明，jim 没有紧跟在 Jack 后面，这是因为使用 string::operator<排序时区分大小写。为确保 jim 紧跟在 Jack 后面（虽然大小写不同），最后一次显示内容前，使用了程序清单 21.6 实现的排序谓词 CompareStringNoCase<>，如第 32 行所示。

很多 STL 算法都使用二元谓词。例如，删除相邻重复元素的 std::unique()、排序算法 std::sort()、排序并保持相对顺序的 std::stable_sort()以及对两个范围进行操作的 std::transform()，这些 STL 算法都需要使用二元谓词。

21.3 总结

本章介绍了函数对象（也叫 functor）。在结构或类中实现函数对象时，它将比简单函数有用得多，因为它也可用于存储与状态相关的信息。本章还介绍了谓词，它是一类特殊的函数对象。另外，还通过一些实际示例说明了谓词的用途。

21.4 问与答

问：谓词是一种特殊的函数对象，其特殊之处何在？
答：谓词总是返回布尔值。

问：调用诸如 remove_if 等函数时，应使用哪种函数对象？
答：应使用通过构造函数将值作为初始状态的一元谓词。

问：对于 map 应使用哪种函数对象？
答：应使用二元谓词。

问：没有返回值的简单函数是否可用作谓词？
答：可以。没有返回值的函数也很有用，例如，可用于显示输入的数据。

21.5 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

21.5.1 测验

1. 返回布尔值的一元函数称为什么？
2. 不修改数据也不返回布尔值的函数对象有什么用？请通过示例阐述您的观点。
3. 函数对象这一术语的定义是什么？

21.5.2 练习

1. 编写一个一元函数，它可供 std::for_each 用来显示输入参数的两倍。
2. 进一步扩展上述谓词，使其能够记录它被调用的次数。
3. 编写一个用于降序排序的二元谓词。

第 22 章
C++ lambda 表达式

lambda 是一种定义匿名函数对象的简洁方式，这是 C++11 新增的。

在本章中，您将学习：

- 如何编写 lambda 表达式；
- 如何将 lambda 表达式用作谓词；
- 如何编写可存储和可操作状态的 lambda 表达式。

22.1 lambda 表达式是什么

可将 lambda 表达式视为包含公有 operator()的匿名结构（或类），从这种意义上说，lambda 表达式属于第 21 章介绍的函数对象。深入分析如何编写 lambda 表达式前，先来看看程序清单 21.1 中的函数对象：

```
// struct that behaves as a unary function
template <typename elementType>
struct DisplayElement
{
    void operator () (const elementType& element) const
    {
        cout << element << ' ';
    }
};
```

这个函数对象将一个元素显示到屏幕上，通常用于 std::for_each 等算法中：

```
// Display the array of integers
for_each ( vecIntegers.begin ()    // Start of range
      , vecIntegers.end ()         // End of range
      , DisplayElement <int> () ); // Unary function object
```

如果使用 lambda 表达式，可将上述代码（包括函数对象的定义）简化为下述 3 行：

```
// Display the array of integers using lambda expression
for_each ( vecIntegers.begin ()    // Start of range
      , vecIntegers.end ()         // End of range
      , [](int& Element) {cout << element << ' '; } ); // Lambda expression
```

编译器见到下述 lambda 表达式时：

```
[](int Element) {cout << element << ' '; }
```

自动将其展开为类似于结构 DisplayElement<int>的表示：

```
struct NoName
{
    void operator () (const int& element) const
    {
        cout << element << ' ';
    }
};
```

22.2 如何定义 lambda 表达式

lambda 表达式的定义必须以方括号（[]）打头。这些括号告诉编译器，接下来是一个 lambda 表达式。方括号的后面是一个参数列表，该参数列表与不使用 lambda 表达式时提供给 operator()的参数列表相同。

22.3 一元函数对应的 lambda 表达式

与一元 operator(Type)对应的 lambda 表达式接受一个参数，其定义如下：

```
[](Type paramName) {  // lambda expression code here;  }
```

请注意，如果您愿意，也可按引用传递参数：

```
[](Type& paramName) {  // lambda expression code here;  }
```

程序清单 22.1 演示了如何在算法 for_each 中使用 lambda 表达式来显示标准模板库（STL）容器的内容。

程序清单 22.1 在算法 for_each()中使用 lambda 表达式而不是函数对象来显示容器中的元素

```
 0: #include <algorithm>
 1: #include <iostream>
 2: #include <vector>
 3: #include <list>
 4:
 5: using namespace std;
 6:
 7: int main ()
 8: {
 9:    vector <int> vecIntegers;
10:
11:    for (int nCount = 0; nCount < 10; ++ nCount)
12:       vecIntegers.push_back (nCount);
13:
14:    list <char> listChars;
15:    for (char nChar = 'a'; nChar < 'k'; ++nChar)
16:       listChars.push_back (nChar);
17:
18:    cout << "Displaying vector of integers using a lambda: " << endl;
19:
20:    // Display the array of integers
21:    for_each ( vecIntegers.begin ()    // Start of range
22:             , vecIntegers.end ()        // End of range
23:             , [](int& element) {cout << element << ' '; } ); // lambda
24:
25:    cout << endl << endl;
26:    cout << "Displaying list of characters using a lambda: " << endl;
27:
28:    // Display the list of characters
29:    for_each ( listChars.begin ()        // Start of range
30:             , listChars.end ()        // End of range
31:             , [](char& element) {cout << element << ' '; } ); // lambda
32:
33:    cout << endl;
34:
35:    return 0;
36: }
```

▼ **输出：**

```
Displaying vector of integers using a lambda:
0 1 2 3 4 5 6 7 8 9

Displaying list of characters using a lambda:
a b c d e f g h i j
```

▼ **分析：**

这里使用了两个 lambda 表达式，如第 23 和 31 行所示。这两个 lambda 表达式很像，只是输入参数不同，因为根据两个容器包含的元素类型对它们进行了定制。第一个 lambda 表达式接受一个 int 参数，并使用它来显示整型 vector 中的元素，每次一个；第二个 lambda 表达式接受一个 char 参数，并使用它来显示 std::list 中的 char 元素。

注意

程序清单 22.1 的输出与程序清单 21.1 相同，这并非巧合。事实上，这个程序是程序清单 21.1 所示程序的 lambda 表达式版，而程序清单 21.1 使用的是函数对象 DisplayElement<T>。

如果将这两个程序清单进行比较，您将发现，通过使用 lambda 表达式，可让 C++代码更简单、更简洁。

22.4 一元谓词对应的 lambda 表达式

谓词可帮助您做出决策。一元谓词是返回 bool 类型（true 或 false）的一元表达式。lambda 表达式也可返回值，例如，下面的 lambda 表达式在 Num 为偶数时返回 true：

```
[](int& Num) {return ((Num % 2) == 0); }
```

在这里，返回值的性质让编译器知道该 lambda 表达式的返回类型为 bool。

在算法中，可将 lambda 表达式用作一元谓词。例如，可在 std::find_if()中使用上述 lambda 表达式找出集合中的偶数，如程序清单 22.2 所示。

程序清单 22.2 在算法 std::find_if()中，将 lambda 表达式用作一元谓词，以查找集合中的偶数

```
 0: #include<algorithm>
 1: #include<vector>
 2: #include<iostream>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    vector<int> vecNums;
 8:    vecNums.push_back(25);
 9:    vecNums.push_back(101);
10:    vecNums.push_back(2011);
11:    vecNums.push_back(-50);
12:
13:    auto iEvenNum = find_if( vecNums.cbegin()
14:                           , vecNums.cend()  // range to find in
15:                           , [](const int& Num){return ((Num % 2) == 0); } );
16:
17:    if (iEvenNum != vecNums.cend())
18:       cout << "Even number in collection is: " << *iEvenNum << endl;
19:
20:    return 0;
21: }
```

▼ **输出：**

```
Even number in collection is: -50
```

▼ **分析：**

用作一元谓词的lambda表达式如第15行所示。算法find_if()对指定范围内的每个元素调用该一元谓词；如果该谓词返回true，find_if()将返回一个指向相应元素的迭代器，指出找到了一个满足条件的元素。这里的谓词是一个lambda表达式，当find_if()使用一个偶数调用它（即对2求模的结果为零）时，它将返回true。

注意

程序清单22.2不但演示了如何将lambda表达式用作一元谓词，还在该lambda表达式中使用了const。
务必使用const来限定输入参数，在输入参数为引用时尤其如此。

22.5 通过捕获列表接受状态变量的lambda表达式

在程序清单22.2中，您创建了一个一元谓词，它在整数能被2整除（即为偶数）时返回true。如果要让它更通用，在数字能被用户指定的除数整除时返回true，该如何办呢？为此，需要让lambda表达式接受该“状态”——除数：

```
int Divisor = 2; // initial value
…
auto iElement = find_if ( begin of a range
                        , end of a range
        , [Divisor](int dividend){return (dividend % Divisor) == 0; } );
```

一系列以状态变量的方式传递的参数（[...]）也被称为lambda表达式的捕获列表（capture list）。

注意

上述lambda表达式只有一行代码，但与程序清单21.3所示的16行代码等价，该程序清单以结构IsMultiple<>的方式定义了一个一元谓词。
这表明，C++11新增的lambda表达式大幅提高了编程效率。

程序清单22.3演示了如何使用lambda表达式，根据状态变量在集合中查找可被用户提供的除数整除的元素。

程序清单22.3 使用存储状态的lambda表达式来判断一个数字能否被另一个数字整除

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3: using namespace std;
 4:
 5: int main ()
 6: {
 7:    vector <int> vecIntegers;
 8:    cout << "The vector contains the following sample values: ";
 9:
10:    // Insert sample values: 25 - 31
11:    for (int nCount = 25; nCount < 32; ++ nCount)
12:    {
13:       vecIntegers.push_back (nCount);
14:       cout << nCount << ' ';
15:    }
16:    cout << endl << "Enter divisor (> 0): ";
17:    int Divisor = 2;
```

```
18:    cin >> Divisor;
19:
20:    // Find the first element that is a multiple of divisor
21:    vector <int>::iterator iElement;
22:    iElement = find_if ( vecIntegers.begin ()
23:                       , vecIntegers.end ()
24:          , [Divisor](int dividend){return (dividend % Divisor) == 0; } );
25:
26:    if (iElement != vecIntegers.end ())
27:    {
28:       cout << "First element in vector divisible by " << Divisor;
29:       cout << ": " << *iElement << endl;
30:    }
31:
32:    return 0;
33: }
```

▼ **输出：**

```
The vector contains the following sample values: 25 26 27 28 29 30 31
Enter divisor (> 0): 4
First element in vector divisible by 4: 28
```

▼ **分析：**

包含状态并用作为此的 lambda 表达式如第 24 行所示。除数是一个状态变量，相当于程序清单 21.3 中的 IsMultiple::Divisor，因此状态变量类似于 C++11 之前的函数对象类中的成员。您可以将状态传递给 lambda 表达式，并根据状态的性质相应地使用它。

注意

> 程序清单 22.3 与程序清单 21.4 等价，但使用的是 lambda 表达式，而不是函数对象。这里通过使用 C++11 新增的 lambda 表达式，减少了 16 行代码。

22.6 lambda 表达式的通用语法

lambda 表达式总是以方括号打头，并可接受多个状态变量，为此可在捕获列表（[...]）中指定这些状态变量，并用逗号分隔：

```
[StateVar1, StateVar2](Type& param) { // lambda code here; }
```

如果要在 lambda 表达式中修改这些状态变量，可添加关键字 multable：

```
[StateVar1, StateVar2](Type& param) mutable { // lambda code here; }
```

这样，便可在 lambda 表达式中修改捕获列表（[]）中指定的变量，但离开 lambda 表达式后，这些修改将无效。要确保在 lambda 表达式内部对状态变量的修改在其外部也有效，应按引用传递它们：

```
[&StateVar1, &StateVar2](Type& param) { // lambda code here; }
```

lambda 表达式还可接受多个输入参数，为此可用逗号分隔它们：

```
[StateVar1, StateVar2](Type1& var1, Type2& var2) { // lambda code here; }
```

如果要向编译器明确地指定返回类型，可使用->，如下所示：

```
[State1, State2](Type1 var1, Type2 var2) -> ReturnType
{ return (value or expression ); }
```

最后，复合语句（{}）可包含多条用分号分隔的语句，如下所示：

```
[State1, State2](Type1 var1, Type2 var2) -> ReturnType
{ Statement 1; Statement 2; return (value or expression); }
```

注意

如果 lambda 表达式包含多行代码，您必须显式地指定返回类型。
本章后面的程序清单 22.5 演示了一个跨越多行并指定了返回类型的 lambda 表达式。

总之，lambda 表达式更简洁，从功能上说，完全可替代下面这样的函数对象：

```
template<typename Type1, typename Type2>
struct IsNowTooLong
{
   // State variables
   Type1 var1;
   Type2 var2;

   // Constructor
   IsNowTooLong(const Type1& in1, Type2& in2): var1(in1), var2(in2) {};

   // the actual purpose
   ReturnType operator()
   {
      Statement 1;
      Statement 2;
      return (value or expression);
   }
};
```

22.7 二元函数对应的 lambda 表达式

二元函数接受两个参数，还可返回一个值。与之等价的 lambda 表达式如下：

```
[...](Type1& param1Name, Type2& param2Name) {  // lambda code here;  }
```

程序清单 22.4 演示了一个 lambda 表达式，并在 std::transform 中使用它将两个等长的 vector 中对应的元素相乘，再将结果存储到第三个 vector 中。

程序清单 22.4 将 lambda 表达式用作二元函数，以便将两个容器中的元素相乘，并将结果存储到第三个容器中

```
 0: #include <vector>
 1: #include <iostream>
 2: #include <algorithm>
 3:
 4: int main ()
 5: {
 6:    using namespace std;
 7:
 8:    // Create two sample vector of integers with 10 elements each
 9:    vector <int> vecMultiplicand, vecMultiplier;
10:
11:    // Insert sample values 0 to 9
12:    for (int nCount1 = 0; nCount1 < 10; ++ nCount1)
13:       vecMultiplicand.push_back (nCount1);
14:
15:    // Insert sample values 100 to 109
16:    for (int nCount2 = 100; nCount2 < 110; ++ nCount2)
17:       vecMultiplier.push_back (nCount2);
18:
19:    // A third container that holds the result of multiplication
20:    vector <int> vecResult;
21:
22:    // Make space for the result of the multiplication
23:    vecResult.resize (10);
```

```
24:
25:    transform ( vecMultiplicand.begin (), // range of multiplicands
26:                vecMultiplicand.end (), // end of range
27:                vecMultiplier.begin (),  // multiplier values
28:                vecResult.begin (), // range that holds result
29:                [](int a, int b) {return a * b; } );  // lambda
30:
31:    cout << "The contents of the first vector are: " << endl;
32:    for (size_t nIndex1 = 0; nIndex1 < vecMultiplicand.size (); ++ nIndex1)
33:       cout << vecMultiplicand [nIndex1] << ' ';
34:    cout << endl;
35:
36:    cout << "The contents of the second vector are: " << endl;
37:    for (size_t nIndex2 = 0; nIndex2 < vecMultiplier.size (); ++nIndex2)
38:       cout << vecMultiplier [nIndex2] << ' ';
39:    cout << endl << endl;
40:
41:    cout << "The result of the multiplication is: " << endl;
42:    for (size_t nIndex = 0; nIndex < vecResult.size (); ++ nIndex)
43:       cout << vecResult [nIndex] << ' ';
44:    cout << endl;
45:
46:    return 0;
47: }
```

▼ **输出：**

```
The contents of the first vector are:
0 1 2 3 4 5 6 7 8 9
The contents of the second vector are:
100 101 102 103 104 105 106 107 108 109

The result of the multiplication is:
0 101 204 309 416 525 636 749 864 981
```

▼ **分析：**

lambda 表达式如第 29 行所示，它被用作 std::transform 的一个参数。算法 std::transform 接受两个范围作为输入，执行二元函数指定的变换算法，并将结果存储在目标容器中。这里的二元函数是一个 lambda 表达式，它接受两个整数作为输入，并返回相乘得到的结果。返回值被 std::transform 存储到 vecResult 中。输出指出了两个容器的内容以及将对应的元素相乘得到的结果。

注意 程序清单 22.4 演示了与程序清单 21.5 中的函数对象类 Multiply<>等价的 lambda 表达式。

22.8 二元谓词对应的 lambda 表达式

返回 true 或 false，可帮助决策的二元函数被称为二元谓词。这种谓词可用于 std::sort()等排序算法中，这些算法对容器中的两个值调用二元谓词，以确定将哪个放在前面。与二元谓词等价的 lambda 表达式的通用语法如下：

```
[...](Type1& param1Name, Type2& param2Name) {  // return bool expression; }
```

程序清单 22.5 演示了如何将 lambda 表达式用于排序。

程序清单 22.5 在 std::sort()中，将 lambda 表达式用作二元谓词，以便进行区分大小写的排序

```
0: #include <algorithm>
1: #include <string>
2: #include <vector>
```

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

template <typename T>
void DisplayContents (const T& Input)
{
   for(auto iElement = Input.cbegin() // auto, cbegin and cend: c++11
      ; iElement != Input.cend ()
      ; ++ iElement )
      cout << *iElement << endl;
}

int main ()
{
   // Define a vector of string to hold names
   vector <string> vecNames;

   // Insert some sample names in to the vector
   vecNames.push_back ("jim");
   vecNames.push_back ("Jack");
   vecNames.push_back ("Sam");
   vecNames.push_back ("Anna");

   cout << "The names in vector in order of insertion: " << endl;
   DisplayContents(vecNames);

   cout << "Names after sorting using default std::less<>: " << endl;
   sort(vecNames.begin(), vecNames.end());
   DisplayContents(vecNames);

   cout << "Names after sorting using predicate that ignores case:" << endl;
   sort(vecNames.begin(), vecNames.end(),
       [](const string& str1, const string& str2) -> bool  // lambda
         {
          string str1LowerCase;

          // Assign space
          str1LowerCase.resize (str1.size ());

          // Convert every character to the lower case
          transform(str1.begin(), str1.end(),
          str1LowerCase.begin(),tolower);

          string str2LowerCase;
          str2LowerCase.resize (str2.size ());
          transform (str2.begin (), str2.end (), str2LowerCase.begin (),
                     tolower);

          return (str1LowerCase < str2LowerCase);
         }  // end of lambda
      );  // end of sort
   DisplayContents(vecNames);

   return 0;
}
```

▼ 输出：

```
The names in vector in order of insertion:
jim
Jack
```

```
Sam
Anna
Names after sorting using default std::less<>:
Anna
Jack
Sam
jim
Names after sorting using predicate that ignores case:
Anna
Jack
jim
Sam
```

▼ 分析：

这里演示的 lambda 表达式是第 34～52 行的 std::sort()的第三个参数，该 lambda 表达式很长，跨越了第 35～51 行。它表明，lambda 表达式可跨越多行，但必须显式地指定返回类型，如第 35 行所示。输出表明，插入元素后，jim 位于 Jack 前面；第 30 行以默认方式排序（即未提供 lambda 表达式或谓词）后，jim 位于 Sam 后面，因为这将根据 string::operator<以区分大小写的方式进行排序。最后，使用不区分大小写的 lambda 表达式排序（如第 34～52 行所示）后，jim 紧跟在 Jack 后面，这符合用户的预期。另外，这个 lambda 表达式跨越多行。

注意

程序清单 22.5 中的超大型 lambda 表达式与程序清单 21.6 所示的 CompareStringNoCase 类等效，程序清单 21.7 使用了 CompareStringNoCase 类。

显然，在这种情况下，使用 lambda 表达式并非最佳选择，因为如果使用函数对象，将能在多条 std::sort()语句中重用它，还可将其用于其他需要二元谓词的算法。

因此，仅当 lambda 表达式简短、高效时，才应使用它。

应该	不应该
请牢记，lambda 表达式总是以[]或[state1, state2,...]打头。 请牢记，除非使用关键字 mutable 进行指定，否则不能修改捕获列表中指定的状态变量。	别忘了，lambda 表达式是实现了 operator()的匿名类（或结构）。 编写 lambda 表达式时，别忘了使用 const 对参数进行限定。 lambda 表达式的语句块（{}）包含多条语句时，别忘了显式地指定返回类型。 不要使用很长（包含多条语句）的 lambda 表达式，而应转而使用函数对象，因为每次使用 lambda 表达式时，都需要重新定义它，这无助于提高代码的重用性。

22.9 总结

本章介绍了 C++11 新增的一项非常重要的功能：lambda 表达式。lambda 是匿名的函数对象，可接受参数、存储状态、返回值以及跨越多行。您学习了如何在 find()、sort()、transform()等 STL 算法中使用 lambda 表达式，而不是函数对象。lambda 表达式可提高 C++编程速度和效率，应尽可能使用它们。

22.10 问与答

问：是否在任何情况下都应使用 lambda 表达式，而不是函数对象？

答：使用跨越多行的 lambda 表达式（如程序清单 22.5 所示）可能无助于提高编程效率，此时应使用易于重用的函数对象。

问：lambda 表达式的状态参数是如何传递的？按值传递还是按引用传递？

答：编写下面这样包含捕获列表的 lambda 表达式时：

```
[Var1, Var2, ... N](Type& Param1, ... ) { …expression ;}
```

将复制状态参数 Var1 和 Var2，而不是按引用传递它们。如果要按引用传递它们，应使用下面的语法：

```
[&Var1, &Var2, ... &N](Type& Param1, ... ) { ...expression ;}
```

在这种情况下，对状态变量所做的修改在 lambda 表达式外部仍将有效，因此请务必小心。

问：在 lambda 表达式中，可使用函数中的局部变量吗？

答：可使用捕获列表来传递局部变量：

```
[Var1, Var2, ... N](Type& Param1, ... ) { ...expression ;}
```

要传递所有的局部变量，可使用如下语法：

```
[=](Type& Param1, ... ) { ...expression ;}
```

22.11 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

22.11.1 测验

1. 编译器如何确定 lambda 表达式的起始位置？
2. 如何将状态变量传递给 lambda 表达式？
3. 如何指定 lambda 表达式的返回类型？

22.11.2 练习

1. 编写一个可用作二元谓词的 lambda 表达式，帮助将元素按降序排列。
2. 编写一个这样的 lambda 表达式，即用于 for_each 时，给 vector 等容器中的元素加上用户指定的值。

第 23 章
STL 算法

标准模板库（STL）中很重要的一部分是通用函数，这些函数位于头文件<algorithm>中，可帮助操作容器的内容。本章介绍如何使用各种 STL 算法以及如何根据应用程序的需求定制这些算法。

在本章中，您将学习：

- 如何使用 STL 算法简化模板代码；
- 帮助在 STL 容器中执行计数、搜索、查找、删除等操作的通用函数。

23.1 什么是 STL 算法

查找、搜索、删除和计数是一些通用算法，其应用范围很广。STL 通过通用的模板函数提供了这些算法以及其他的很多算法，可通过迭代器对容器进行操作。要使用 STL 算法，程序员必须包含头文件<algorithm>。

注意

> 虽然大多数算法都通过迭代器对容器进行操作，但并非所有算法都对容器进行操作，因此并非所有算法都需要迭代器。有些算法接受一对值，例如，swap()将这对值交换。同样，min()和 max()也对值进行操作。

23.2 STL 算法的分类

STL 算法分两大类：非变序算法与变序算法。

23.2.1 非变序算法

不改变容器中元素的顺序和内容的算法称为非变序算法。表 23.1 列出了一些主要的非变序算法。

表 23.1　　非变序算法

算　法	描　述
计数算法	
count	在指定范围内查找值与指定值匹配的所有元素
count_if	在指定范围内查找值满足指定条件的所有元素
搜索算法	
search	在目标范围内，根据元素相等性（即运算符==）或指定二元谓词搜索第一个满足条件的元素

续表

算　法	描　述
search_n	在目标范围内搜索与指定值相等或满足指定谓词的 n 个元素
find	在给定范围内搜索与指定值匹配的第一个元素
find_if	在给定范围内搜索满足指定条件的第一个元素
find_end	在指定范围内搜索最后一个满足特定条件的元素
find_first_of	在目标范围内搜索指定序列中的任何一个元素第一次出现的位置；在另一个重载版本中，它搜索满足指定条件的第一个元素
adjacent_find	在集合中搜索两个相等或满足指定条件的元素
比较算法	
equal	比较两个元素是否相等或使用指定的二元谓词判断两者是否相等
mismatch	使用指定的二元谓词找出两个元素范围的第一个不同的地方
lexicographical_compare	比较两个序列中的元素，以判断哪个序列更小

23.2.2 变序算法

变序算法改变其操作的序列的元素顺序或内容，表 23.2 列出了 STL 提供的一些最有用的变序算法。

表 23.2　　变序算法

算　法	描　述
初始化算法	
fill	将指定值分配给指定范围中的每个元素
fill_n	将指定值分配给指定范围中的前 n 个元素
generate	将指定函数对象的返回值分配给指定范围中的每个元素
generate_n	将指定函数的返回值分配给指定范围中的前 n 个元素
修改算法	
for_each	对指定范围内的每个元素执行指定的操作。当指定的参数修改了范围时，for_each 将是变序算法
transform	对指定范围中的每个元素执行指定的一元函数
复制算法	
copy	将一个范围复制到另一个范围
copy_backward	将一个范围复制到另一个范围，但在目标范围中将元素的排列顺序反转
删除算法	
remove	将指定范围中包含指定值的元素删除
remove_if	将指定范围中满足指定一元谓词的元素删除
remove_copy	将源范围中除包含指定值外的所有元素复制到目标范围
remove_copy_if	将源范围中除满足指定一元谓词外的所有元素复制到目标范围
unique	比较指定范围内的相邻元素，并删除重复的元素。该算法还有一个重载版本，它使用二元谓词来判断要删除哪些元素
unique_copy	将源范围内的所有元素复制到目标范围，但相邻的重复元素除外

续表

算　法	描　述
替换算法	
replace	用一个值来替换指定范围中与指定值匹配的所有元素
replace_if	用一个值来替换指定范围中满足指定条件的所有元素
排序算法	
sort	使用指定的排序标准对指定范围内的元素进行排序，排序标准由二元谓词提供。排序可能改变相等元素的相对顺序
stable_sort	类似于 sort，但在排序时保持相对顺序不变
partial_sort	将源范围内指定数量的元素排序
partial_sort_copy	将源范围内的元素复制到目标范围，同时对它们排序
分区算法	
partition	在指定范围中，将元素分为两组：满足指定一元谓词的元素放在第一个组中，其他元素放在第二组中。不一定会保持集合中元素的相对顺序
stable_partition	与 partition 一样将指定范围分为两组，但保持元素的相对顺序不变
可用于排序容器的算法	
binary_search	用于判断一个元素是否存在于一个排序集合中
lower_bound	根据元素的值或二元谓词判断元素可能插入到排序集合中的第一个位置，并返回一个指向该位置的迭代器
upper_bound	根据元素的值或二元谓词判断元素可能插入到排序集合中的最后一个位置，并返回一个指向该位置的迭代器

23.3 使用 STL 算法

要学习表 23.1 和表 23.2 所示 STL 算法的用法，最佳的方法是亲自动手实践。为此，读者可通过下面的示例代码学习如何使用算法，并在自己的应用程序使用它们。

23.3.1 根据值或条件查找元素

STL 算法 find()和 find_if()用于在 vector 等容器中查找与值匹配或满足条件的元素。find()的用法如下：

```
auto iElementFound = find ( vecIntegers.cbegin () // Start of range
                          , vecIntegers.cend ()   // End of range
                          , NumToFind );          // Element to find

// Check if find succeeded
if ( iElementFound != vecIntegers.cend ())
   cout << "Result: Value found!" << endl;
```

find_if()的用法与此类似，但需要通过第三个参数提供一个一元谓词（返回 true 或 false 的一元函数）：

```
auto iEvenNumber = find_if ( vecIntegers.cbegin () // Start of range
                           , vecIntegers.cend ()  // End of range
             , [](int element) { return (element % 2) == 0; } );
```

```
if (iEvenNumber != vecIntegers.cend ())
   cout << "Result: Value found!" << endl;
```

这两个函数都返回一个迭代器，您需要将其同容器的 end()或 cend()进行比较，检查查找操作是否成功。如果成功，便可进一步使用该迭代器。程序清单 23.1 使用 find()在 vector 中查找一个值，并使用 find_if()找到第一个偶数。

程序清单 23.1 使用 find()在 vector 中查找一个整数，并使用 find_if 以及一个用 lambda 表达式表示的一元谓词查找第一个偶数

```
 0: #include <iostream>
 1: #include <algorithm>
 2: #include <vector>
 3:
 4: int main()
 5: {
 6:    using namespace std;
 7:    vector<int> vecIntegers;
 8:
 9:    // Inserting sample values -9 to 9
10:    for (int SampleValue = -9; SampleValue < 10; ++ SampleValue)
11:       vecIntegers.push_back (SampleValue);
12:
13:    cout << "Enter number to find in collection: ";
14:    int NumToFind = 0;
15:    cin >> NumToFind;
16:
17:    auto iElementFound = find ( vecIntegers.cbegin () // Start of range
18:                              , vecIntegers.cend ()   // End of range
19:                              , NumToFind );          // Element to find
20:
21:    // Check if find succeeded
22:    if ( iElementFound != vecIntegers.cend ())
23:       cout << "Result: Value " << *iElementFound << " found!" << endl;
24:    else
25:       cout << "Result: No element contains value " << NumToFind << endl;
26:
27:    cout << "Finding the first even number using find_if: " << endl;
28:
29:    auto iEvenNumber = find_if ( vecIntegers.cbegin () // Start of range
30:                               , vecIntegers.cend ()  // End of range
31:                  , [](int element) { return (element % 2) == 0; } );
32:
33:    if (iEvenNumber != vecIntegers.cend ())
34:    {
35:       cout << "Number '" << *iEvenNumber << "' found at position [";
36:       cout << distance (vecIntegers.cbegin (), iEvenNumber);
37:       cout << "]" << endl;
38:    }
39:
40:    return 0;
41: }
```

▼ 输出：

```
Enter number to find in collection: 7
Result: Value 7 found!
Finding the first even number using find_if:
Number '-8' found at position [1]
```

再次运行的输出：

```
Enter number to find in collection: 2011
Result: No element contains value 2011
Finding the first even number using find_if:
Number '-8' found at position [1]
```

▼ 分析：

在 main()函数中，首先创建了一个整型 vector，并将其初始化为-9～9。第 17～19 行使用 find()查找用户输入的数字；第 29～31 行使用 find_if()在指定范围内查找第一个偶数。第 31 行的一元谓词是以 lambda 表达式的方式提供的，该 lambda 表达式在 element 能被 2 整除时返回 true。

警告

在程序清单 23.1 中，注意到总是将 find()或 find_if()返回的迭代器同 cend()进行比较，以核实其有效性。这种检查绝不能省略，因为它表明 find()操作成功。不能想当然地认为 find()操作成功。

提示

在程序清单 17.5 的第 21 行，也演示了如何使用 std::distance 来确定找到的元素相对于 vector 开头的偏移量。

23.3.2 计算包含给定值或满足给定条件的元素数

算法 std::count()和 count_if()计算给定范围内的元素数。std::find()计算包含给定值（使用相当运算符==进行测试）的元素数：

```
size_t nNumZeroes = count (vecIntegers.begin (),vecIntegers.end (),0);
cout << "Number of instances of '0': " << nNumZeroes << endl << endl;
```

std::count_if()计算这样的元素数，即满足通过参数传递的一元谓词（可以是函数对象，也可以是 lambda 表达式）：

```
// Unary predicate:
template <typename elementType>
bool IsEven (const elementType& number)
{
   return ((number % 2) == 0); // true, if even
}
...
// Use the count_if algorithm with the unary predicate IsEven:
size_t nNumEvenElements = count_if (vecIntegers.begin (),
                                    vecIntegers.end (), IsEven <int> );
cout << "Number of even elements: " << nNumEvenElements << endl;
```

程序清单 23.2 演示了这些函数的用法。

程序清单 23.2 使用 std::count()和 count_if()分别计算有多少个元素包含指定值和满足指定条件

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3:
 4: // A unary predicate for the *_if functions
 5: template <typename elementType>
 6: bool IsEven (const elementType& number)
 7: {
 8:    return ((number % 2) == 0); // true, if even
 9: }
10:
11: int main ()
12: {
13:    using namespace std;
14:    vector <int> vecIntegers;
```

```
15:
16:    cout << "Populating a vector<int> with values from -9 to 9" << endl;
17:    for (int nNum = -9; nNum < 10; ++ nNum)
18:       vecIntegers.push_back (nNum);
19:
20:    // Use count to determine the number of '0's in the vector
21:    size_t nNumZeroes = count (vecIntegers.begin (),vecIntegers.end (),0);
22:    cout << "Number of instances of '0': " << nNumZeroes << endl << endl;
23:
24:    // Use the count_if algorithm with the unary predicate IsEven:
25:    size_t nNumEvenElements = count_if (vecIntegers.begin (),
26:                              vecIntegers.end (), IsEven <int> );
27:
28:    cout << "Number of even elements: " << nNumEvenElements << endl;
29:    cout << "Number of odd elements: ";
30:    cout << vecIntegers.size () - nNumEvenElements << endl;
31:
32:    return 0;
33: }
```

▼ **输出：**

```
Populating a vector<int> with values from -9 to 9
Number of instances of '0': 1

Number of even elements: 9
Number of odd elements: 10
```

▼ **分析：**

第 21 行使用了 count()来计算 vector<int>包含多少个值为零的元素；同样，第 25 行使用了 count_if()来计算 vector 包含多少个偶数元素。注意到第 3 个参数是第 6～9 行定义的一元谓词 IsEven()。为计算 vector 包含多少个奇数元素，将 vector 包含的元素总数（由 size()返回）减去 count_if()返回的值。

注意

程序清单 23.2 在 count_if()中使用了谓词 IsEven()，而在程序清单 23.1 中，在 find_if()中使用了一个 lambda 表达式来完成 IsEven()的工作。

使用 lambda 表达式可节省多行代码，但别忘了，如果将这两个示例合并成一个，则可将 IsEven()用于 find_if()和 count_if()中，从而提高了代码的可重用性。

23.3.3 在集合中搜索元素或序列

程序清单 23.1 演示了如何在容器中查找元素，但有时需要查找序列或模式。在这种情况下，应使用 search()或 search_n()。search()用于在一个序列中查找另一个序列：

```
auto iRange = search ( vecIntegers.begin () // Start of range
                     , vecIntegers.end ()     // End of range to search in
                     , listIntegers.begin ()  // Start of range to search for
                     , listIntegers.end () ); // End of range to search for
```

search_n()用于在容器中查找 n 个相邻的指定值：

```
auto iPartialRange = search_n ( vecIntegers.begin () // Start range
                              , vecIntegers.end ()   // End range
                              , 3      // Count of item to be searched for
                              , 9 );   // Item to search for
```

这两个函数都返回一个迭代器，它指向找到的第一个模式；使用该迭代器之前，务必将其与 end()进行比较。程序清单 23.3 演示了 search()和 search_n()的用法。

程序清单 23.3 使用 search 和 search_n 在集合中查找序列

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <list>
 3: #include <iostream>
 4: using namespace std;
 5:
 6: template <typename T>
 7: void DisplayContents (const T& Input)
 8: {
 9:    for(auto iElement = Input.cbegin() // auto, cbegin and cend: c++11
10:      ; iElement != Input.cend()
11:      ; ++ iElement )
12:      cout << *iElement << ' ';
13:
14:    cout << endl;
15: }
16:
17: int main ()
18: {
19:    // A sample container - vector of integers - containing -9 to 9
20:    vector <int> vecIntegers;
21:    for (int nNum = -9; nNum < 10; ++ nNum)
22:       vecIntegers.push_back (nNum);
23:
24:    // Insert some more sample values into the vector
25:    vecIntegers.push_back (9);
26:    vecIntegers.push_back (9);
27:
28:    // Another sample container - a list of integers from -4 to 4
29:    list <int> listIntegers;
30:    for (int nNum = -4; nNum < 5; ++ nNum)
31:       listIntegers.push_back (nNum);
32:
33:    cout << "The contents of the sample vector are: " << endl;
34:    DisplayContents (vecIntegers);
35:
36:    cout << "The contents of the sample list are: " << endl;
37:    DisplayContents (listIntegers);
38:
39:    cout << "search() for the contents of list in vector:" << endl;
40:    auto iRange = search ( vecIntegers.begin () // Start of range
41:                         , vecIntegers.end () // End range to search in
42:                         , listIntegers.begin () // Start range to search for
43:                         , listIntegers.end () ); // End range to search for
44:
45:    // Check if search found a match
46:    if (iRange != vecIntegers.end ())
47:    {
48:       cout << "Sequence in list found in vector at position: ";
49:       cout << distance (vecIntegers.begin (), iRange) << endl;
50:    }
51:
52:    cout << "Searching {9, 9, 9} in vector using search_n(): " << endl;
53:    auto iPartialRange = search_n ( vecIntegers.begin () // Start range
54:                                  , vecIntegers.end ()   // End range
55:                                  , 3 // Count of item to be searched for
56:                                  , 9 );   // Item to search for
57:
58:    if (iPartialRange != vecIntegers.end ())
```

```
59:    {
60:       cout << "Sequence {9, 9, 9} found in vector at position: ";
61:       cout << distance (vecIntegers.begin (), iPartialRange) << endl;
62:    }
63:
64:    return 0;
65: }
```

▼ **输出：**

```
The contents of the sample vector are:
-9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 9 9
The contents of the sample list are:
-4 -3 -2 -1 0 1 2 3 4
search() for the contents of list in vector:
Sequence in list found in vector at position: 5
Searching {9, 9, 9} in vector using search_n():
```

▼ **分析：**

在这个例子中，首先定义了两个容器：一个 vector 和一个 list，并使用一些整型值填充它们。第 40 行使用 search()在 vector 中查找 list。由于要在整个 vector 中查找 list 的全部内容，因此使用了这两个容器类的成员方法 begin()和 end()返回的迭代器来指定范围。这表明迭代器在算法和容器之间搭建了桥梁；在算法看来，提供迭代器的容器的特征无关紧要，因为它只使用迭代器，因此能够在 vector 中无缝地查找 list 的全部内容。第 53 行使用 search_n()搜索序列{9,9,9}在 vector 中首次出现的位置。

23.3.4 将容器中的元素初始化为指定值

STL 算法 fill()和 fill_n()用于将指定范围的内容设置为指定值。fill()将指定范围内的元素设置为指定值：

```
vector <int> vecIntegers (3);

// fill all elements in the container with value 9
fill (vecIntegers.begin (), vecIntegers.end (), 9);
```

顾名思义，fill_n()将 n 个元素设置为指定的值，接受的参数包括起始位置、元素数以及要设置的值：

```
fill_n (vecIntegers.begin () + 3, /*count*/ 3, /*fill value*/ -9);
```

程序清单 23.4 表明，使用这些算法可轻松设置 vector<int>中元素的值。

程序清单 23.4 使用 fill()和 fill_n()设置容器中元素的初始值

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3:
 4: int main ()
 5: {
 6:     using namespace std;
 7:
 8:     // Initialize a sample vector with 3 elements
 9:     vector <int> vecIntegers (3);
10:
11:     // fill all elements in the container with value 9
12:     fill (vecIntegers.begin (), vecIntegers.end (), 9);
13:
14:     // Increase the size of the vector to hold 6 elements
```

```
15:     vecIntegers.resize (6);
16:
17:     // Fill the three elements starting at offset position 3 with value -9
18:     fill_n (vecIntegers.begin () + 3, 3, -9);
19:
20:     cout << "Contents of the vector are: " << endl;
21:     for (size_t nIndex = 0; nIndex < vecIntegers.size (); ++ nIndex)
22:     {
23:         cout << "Element [" << nIndex << "] = ";
24:         cout << vecIntegers [nIndex] << endl;
25:     }
26:
27:     return 0;
28: }
```

▼ 输出：

```
Contents of the vector are:
Element [0] = 9
Element [1] = 9
Element [2] = 9
Element [3] = -9
Element [4] = -9
Element [5] = -9
```

▼ 分析：

程序清单 23.4 使用函数 fill()和 fill_n()将容器的内容初始化为两组不同的值，如第 12 行和第 18 行所示。请注意，在使用值填充范围前调用了函数 resize()，它实际上创建了随后要填充的元素。fill()算法对整个 vector 进行操作，而 fill_n()可对 vector 的一部分进行操作。

23.3.5 使用 std::generate()将元素设置为运行阶段生成的值

函数 fill()和 fill_n()将集合的元素设置为指定的值，而 generate()和 generate_n()等 STL 算法用于将集合的内容设置为一元函数返回的值。

可使用 generate()将指定范围内的元素设置为生成器函数返回的值：

```
generate ( vecIntegers.begin (), vecIntegers.end ()    // range
         , rand );    // generator function to be called
```

generate_n()与 generate()类似，但您指定的是要设置的元素数，而不是闭区间：

```
generate_n (listIntegers.begin (), 5, rand);
```

因此，可使用这两种算法将容器设置为文件的内容或随机值，如程序清单 23.5 所示。

程序清单 23.5 使用 generate()和 generate_n()将集合设置为随机值

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <list>
 3: #include <iostream>
 4:
 5: int main ()
 6: {
 7:     using namespace std;
 8:
 9:     vector <int> vecIntegers (10);
10:     generate ( vecIntegers.begin (), vecIntegers.end ()    // range
11:              , rand );    // generator function to be called
12:
```

```
13:     cout << "Elements in the vector of size " << vecIntegers.size ();
14:     cout << " assigned by 'generate' are: " << endl << "{";
15:     for (size_t nCount = 0; nCount < vecIntegers.size (); ++ nCount)
16:         cout << vecIntegers [nCount] << " ";
17:
18:     cout << "}" << endl << endl;
19:
20:     list <int> listIntegers (10);
21:     generate_n (listIntegers.begin (), 5, rand);
22:
23:     cout << "Elements in the list of size: " << listIntegers.size ();
24:     cout << " assigned by 'generate_n' are: " << endl << "{";
25:     list <int>::const_iterator iElementLocator;
26:     for ( iElementLocator = listIntegers.begin ()
27:         ; iElementLocator != listIntegers.end ()
28:         ; ++ iElementLocator )
29:         cout << *iElementLocator << ' ';
30:
31:     cout << "}" << endl;
32:
33:     return 0;
34: }
```

▼ 输出：

```
Elements in the vector of size 10 assigned by 'generate' are:
{41 18467 6334 26500 19169 15724 11478 29358 26962 24464 }

Elements in the list of size: 10 assigned by 'generate_n' are:
{5705 28145 23281 16827 9961 0 0 0 0 0 }
```

▼ 分析：

程序清单 23.5 使用函数 generate()将 vector 的所有元素都设置为函数 rand 提供的随机值。注意到函数 generate()接受一个范围作为输入，并为该范围内的每个元素调用指定的函数对象 rand；而 generate_n()接受起始位置，调用指定的函数对象 rand count 次，以设置容器中 count 个元素的值。容器中不在指定范围内的元素不受影响。

23.3.6 使用 for_each()处理指定范围内的元素

算法 for_each()对指定范围内的每个元素应用了指定的一元函数对象，其用法如下：

```
unaryFunctionObjectType mReturn = for_each ( start_of_range
                                           , end_of_range
                                           , unaryFunctionObject );
```

也可使用接受一个参数的 lambda 表达式代替一元函数对象。

返回值表明，for_each()返回用于对指定范围内的每个元素进行处理的函数对象（functor）。这意味着使用结构或类作为函数对象可存储状态信息，并在 for_each()执行完毕后查询这些信息，如程序清单 23.6 所示。该程序清单使用函数对象显示指定范围内的元素，并使用它了确定显示了多少个元素。

程序清单 23.6 使用 for_each()显示集合的内容

```
0: #include <algorithm>
1: #include <iostream>
2: #include <vector>
3: #include <string>
4: using namespace std;
5:
```

```
// Unary function object type invoked by for_each
template <typename elementType>
struct DisplayElementKeepCount
{
   int Count;

   // Constructor
   DisplayElementKeepCount (): Count (0) {}

   void operator () (const elementType& element)
   {
      ++ Count;
      cout << element << ' ';
   }
};

int main ()
{
   vector <int> vecIntegers;
   for (int nCount = 0; nCount < 10; ++ nCount)
      vecIntegers.push_back (nCount);

   cout << "Displaying the vector of integers: " << endl;

   // Display the array of integers
   DisplayElementKeepCount<int> Functor =
      for_each ( vecIntegers.begin ()   // Start of range
             , vecIntegers.end ()       // End of range
             , DisplayElementKeepCount<int> () );// Functor

   cout << endl;

   // Use the state stored in the return value of for_each!
   cout << "'" << Functor.Count << "' elements were displayed" << endl;

   string Sample ("for_each and strings!");
   cout << "String: " << Sample << ", length: " << Sample.length() << endl;

   cout << "String displayed using lambda:" << endl;
   int NumChars = 0;
   for_each ( Sample.begin()
            , Sample.end ()
            , [&NumChars](char c) { cout << c << ' '; ++NumChars; } );

   cout << endl;
   cout << "'" << NumChars << "' characters were displayed" << endl;

   return 0;
}
```

▼ **输出：**

```
Displaying the vector of integers:
0 1 2 3 4 5 6 7 8 9
'10' elements were displayed
String: for_each and strings!, length: 21
String displayed using lambda:
f o r _ e a c h   a n d   s t r i n g s !
'21' characters were displayed
```

▼ **分析：**

上述代码不仅演示了如何使用 for_each()，还说明了它返回函数对象这一特点。返回的函数对象可存储信息，如被调用的次数。上述代码定义了两个范围，一个包含在整型 vector vecIntergers 中，另一个是 std::string 对象 Sample。第 32 行和第 45 行分别对这两个范围调用了 for_each()，前者将一元谓词 DisplayElementKeepCount 用作函数对象，而后者使用了一个 lambda 表达式。for_each()对指定范围内的每个元素调用 operator()，而 operator()将元素显示在屏幕上，并将内部计数加 1。for_each()执行完毕后返回该函数对象，其成员 Count 指出了函数对象被调用的次数。在实际编程中，将信息（或状态）存储在算法返回的对象中很有帮助。第 45 行的 for_each()执行的操作与第 32 行的 for_each()相同，但操作的是一个 std::string，且使用的是 lambda 表达式，而不是函数对象。

23.3.7 使用 std::transform()对范围进行变换

std::for_each()和 std::transform()很像，都对源范围内的每个元素调用指定的函数对象。然而，std::transform()有两个版本，第一个版本一个接受一元函数，常用于将字符串转换为大写或小写（使用的一元函数分别是 toupper()和 tolower()）：

```
string Sample ("THIS is a TEst string!");
transform ( Sample.begin ()          // start of source range
          , Sample.end ()          // end of source range
          , strLowerCaseCopy.begin ()// start of destination range
          , tolower );        // unary function
```

第二个版本接受一个二元函数，让 transform()能够处理一对来自两个不同范围的元素：

```
// add elements in two ranges and store in a third
transform ( vecIntegers1.begin ()   // start of source range 1
          , vecIntegers1.end ()     // end of source range 1
          , vecIntegers2.begin ()   // start of source range 2
          , dqResultAddition.begin ()// store result in a deque
          , plus <int> () );        // binary function plus
```

不像 for_each()那样只处理一个范围，这两个版本的 transform()都将指定变换函数的结果赋给指定的目标范围。程序清单 23.7 演示了 std::transform()的用法。

程序清单 23.7 使用一元函数和二元函数的 std::transform()

```
 0: #include <algorithm>
 1: #include <string>
 2: #include <vector>
 3: #include <deque>
 4: #include <iostream>
 5: #include <functional>
 6:
 7: int main ()
 8: {
 9:     using namespace std;
10:
11:     string Sample ("THIS is a TEst string!");
12:     cout << "The sample string is: " << Sample << endl;
13:
14:     string strLowerCaseCopy;
15:     strLowerCaseCopy.resize (Sample.size ());
16:
17:     transform ( Sample.begin ()          // start of source range
18:               , Sample.end ()          // end of source range
19:               , strLowerCaseCopy.begin ()// start of destination range
20:               , tolower );        // unary function
```

```
21:
22:     cout << "Result of 'transform' on the string with 'tolower':" << endl;
23:     cout << "\"" << strLowerCaseCopy << "\"" << endl << endl;
24:
25:     // Two sample vectors of integers...
26:     vector <int> vecIntegers1, vecIntegers2;
27:     for (int nNum = 0; nNum < 10; ++ nNum)
28:     {
29:         vecIntegers1.push_back (nNum);
30:         vecIntegers2.push_back (10 - nNum);
31:     }
32:
33:     // A destination range for holding the result of addition
34:     deque <int> dqResultAddition (vecIntegers1.size ());
35:
36:     transform ( vecIntegers1.begin ()     // start of source range 1
37:               , vecIntegers1.end ()       // end of source range 1
38:               , vecIntegers2.begin ()     // start of source range 2
39:               , dqResultAddition.begin ()// start of destination range
40:               , plus <int> () );          // binary function
41:
42:     cout << "Result of 'transform' using binary function 'plus': " << endl;
43:     cout <<endl << "Index   Vector1 + Vector2 = Result (in Deque)" << endl;
44:     for (size_t nIndex = 0; nIndex < vecIntegers1.size (); ++ nIndex)
45:     {
46:         cout << nIndex << "     \t " << vecIntegers1 [nIndex]  << "\t+    ";
47:         cout << vecIntegers2 [nIndex]  << " \t  =    ";
48:
49:         cout << dqResultAddition [nIndex] << endl;
50:     }
51:
52:     return 0;
53: }
```

▼ 输出：

```
The sample string is: THIS is a TEst string!
Result of using 'transform' with unary function 'tolower' on the string:
"this is a test string!"

Result of 'transform' using binary function 'plus':
Index   Vector1 + Vector2 = Result (in Deque)
0        0      +   10    =    10
1        1      +   9     =    10
2        2      +   8     =    10
3        3      +   7     =    10
4        4      +   6     =    10
5        5      +   5     =    10
6        6      +   4     =    10
7        7      +   3     =    10
8        8      +   2     =    10
9        9      +   1     =    10
```

▼ 分析：

该示例演示了 std::transform()的两种版本的用法：一种版本使用一元函数 tolower 处理一个范围，如第 20 行所示；另一个版本使用二元函数 plus 处理两个范围，如第 40 行所示。第一个版本修改字符串的大小写，将每个字符都改为小写。如果使用 touppe 而不是 tolower，将把字符变为大写。std::transform()的另一个版本如第 36～40 行所示，它对来自两个输入范围（这里是两个 vector）内的元素进行操作：使用一个二元谓词——STL 函数 plus()（由头文件<functional>提供）将两个元

素相加。std::transform()每次处理一对元素，它将这对元素提供给二元函数 plus，然后将结果赋给目标范围中的元素——这里是 std::deque 中的元素。注意，这里使用另一种容器来存储结果只是出于演示目的。这个例子表明，通过使用迭代器，可将容器及其实现同 STL 算法分离：transform()是一种处理范围的算法，它无需知道实现这些范围的容器的细节。因此，虽然这里的输入范围为 vector，而输出范围为 deque，但该算法仍管用——只要指定范围的边界（提供给 transform 的输入参数）有效。

23.3.8 复制和删除操作

STL 提供了三个重要的复制函数：copy()、copy_if()和 copy_backward()。copy 沿向前的方向将源范围的内容赋给目标范围：

```
auto iLastPos = copy ( listIntegers.begin ()  // start source range
                     , listIntegers.end ()     // end source range
                     , vecIntegers.begin () );// start dest range
```

copy_if()仅在指定的一元谓词返回 true 时才复制元素：

```
// copy odd numbers from list into vector
copy_if ( listIntegers.begin(), listIntegers.end()
        , iLastPos
        , [](int element){return ((element % 2) == 1);});
```

注意

> copy_if()是 C++11 新增的算法，包含在命名空间 std 中；如果您使用的编译器较旧（为遵循 C++11），使用该算法可能导致问题。

copy_backward()沿向后的方向将源范围的内容赋给目标范围：

```
copy_backward ( listIntegers.begin ()
              , listIntegers.end ()
              , vecIntegers.end () );
```

remove()将容器中与指定值匹配的元素删除：

```
// Remove all instances of '0', resize vector using erase()
auto iNewEnd = remove (vecIntegers.begin (), vecIntegers.end (), 0);
vecIntegers.erase (iNewEnd, vecIntegers.end ());
```

remove_if()使用一个一元谓词，并将容器中满足该谓词的元素删除：

```
// Remove all odd numbers from the vector using remove_if
iNewEnd = remove_if (vecIntegers.begin (), vecIntegers.end (),
          [](int element) {return ((element % 2) == 1);} ); //predicate

vecIntegers.erase (iNewEnd , vecIntegers.end ());  // resizing
```

程序清单 23.8 演示了这些复制和删除函数的用法。

程序清单 23.8 一个演示 copy()、copy_if()、copy_backward()、remove()和 remove_if()的示例，它将 list 的内容复制到 vector 中，并删除包含零或偶数的元素

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <list>
 3: #include <iostream>
 4: using namespace std;
 5:
 6: template <typename T>
 7: void DisplayContents(const T& Input)
 8: {
 9:    for ( auto iElement = Input.cbegin() // auto, cbegin: c++11
10:        ; iElement != Input.cend()  // cend() is new in C++11
```

```
11:         ; ++ iElement)
12:       cout << *iElement << ' ';
13:
14:    cout << "¦ Number of elements: " << Input.size() << endl;
15: }
16: int main ()
17: {
18:    list <int> listIntegers;
19:    for (int nCount = 0; nCount < 10; ++ nCount)
20:       listIntegers.push_back (nCount);
21:
22:    cout << "Source (list) contains:" << endl;
23:    DisplayContents(listIntegers);
24:
25:    // Initialize the vector to hold twice as many elements as the list
26:    vector <int> vecIntegers (listIntegers.size () * 2);
27:
28:    auto iLastPos = copy ( listIntegers.begin ()  // start source range
29:                         , listIntegers.end ()     // end source range
30:                         , vecIntegers.begin () );// start dest range
31:
32:    // copy odd numbers from list into vector
33:    copy_if ( listIntegers.begin(), listIntegers.end()
34:            , iLastPos
35:            , [](int element){return ((element % 2) == 1);});
36:
37:    cout << "Destination (vector) after copy and copy_if:" << endl;
38:    DisplayContents(vecIntegers);
39:
40:    // Remove all instances of '0', resize vector using erase()
41:    auto iNewEnd = remove (vecIntegers.begin (), vecIntegers.end (), 0);
42:    vecIntegers.erase (iNewEnd, vecIntegers.end ());
43:
44:    // Remove all odd numbers from the vector using remove_if
45:    iNewEnd = remove_if (vecIntegers.begin (), vecIntegers.end (),
46:              [](int element) {return ((element % 2) == 1);} ); //predicate
47:
48:    vecIntegers.erase (iNewEnd , vecIntegers.end ());  // resizing
49:
50:    cout << "Destination (vector) after remove, remove_if, erase:" << endl;
51:    DisplayContents(vecIntegers);
52:
53:    return 0;
54: }
```

▼ 输出：

```
Source (list) contains:
0 1 2 3 4 5 6 7 8 9 ¦ Number of elements: 10
Destination (vector) after copy and copy_if:
0 1 2 3 4 5 6 7 8 9 1 3 5 7 9 0 0 0 0 0 ¦ Number of elements: 20
Destination (vector) after remove, remove_if, erase:
2 4 6 8 ¦ Number of elements: 4
```

▼ 分析：

第 28 行使用了 copy()，它将 list 的内容复制到 vector 中。第 33 行使用了 copy_if()，它将 listIntegers 中的所有奇数都复制到 vecIntegers 中，并存储到从 copy()返回的迭代器 iLastPos 开始的位置。第 41 行使用了 remove()，它删除 vecIntegers 中所有值为零的元素。第 45 行使用 remove_if()删除了所有奇数。

警告

程序清单 23.8 表明，remove()和 remove_if()都返回一个指向容器末尾的迭代器，但容器 vecIntegers 一直未调整大小。删除算法删除元素时，其他元素将向前移，但 size()返回的值没变，这意味着 vector 末尾还有其他值。要调整容器的大小（这很重要，否则末尾将包含不需要的值），需要调用 erase()，并将 remove()或 remove_if()返回的迭代器传递给它，如第 42 和 48 行所示。

23.3.9 替换值以及替换满足给定条件的元素

STL 算法 replace()与 replace_if()分别用于替换集合中等于指定值和满足给定条件的元素。replace()根据比较运算符==的返回值来替换元素：

```
cout << "Using 'std::replace' to replace value 5 by 8" << endl;
replace (vecIntegers.begin (), vecIntegers.end (), 5, 8);
```

replace_if()需要一个用户指定的一元谓词，对于要替换的每个值，该谓词都返回 true：

```
cout << "Using 'std::replace_if' to replace even values by -1" << endl;
replace_if (vecIntegers.begin (), vecIntegers.end ()
   , [](int element) {return ((element % 2) == 0); }, -1);
```

程序清单 23.9 演示了这两个函数的用法。

程序清单 23.9 使用 replace()和 replace_if()在指定范围内替换值

```
 0: #include <iostream>
 1: #include <algorithm>
 2: #include <vector>
 3: using namespace std;
 4:
 5: template <typename T>
 6: void DisplayContents(const T& Input)
 7: {
 8:    for ( auto iElement = Input.cbegin() // auto, cbegin: c++11
 9:         ; iElement != Input.cend()  // cend() is new in C++11
10:         ; ++ iElement)
11:       cout << *iElement << ' ';
12:
13:    cout << "¦ Number of elements: " << Input.size() << endl;
14: }
15: int main ()
16: {
17:    vector <int> vecIntegers (6);
18:
19:    // fill first 3 elements with value 8, last 3 with 5
20:    fill (vecIntegers.begin (), vecIntegers.begin () + 3, 8);
21:    fill_n (vecIntegers.begin () + 3, 3, 5);
22:
23:    // shuffle the container
24:    random_shuffle (vecIntegers.begin (), vecIntegers.end ());
25:
26:    cout << "The initial contents of the vector are: " << endl;
27:    DisplayContents(vecIntegers);
28:
29:    cout << endl << "Using 'std::replace' to replace value 5 by 8" << endl;
30:    replace (vecIntegers.begin (), vecIntegers.end (), 5, 8);
31:
32:    cout << "Using 'std::replace_if' to replace even values by -1" << endl;
33:    replace_if (vecIntegers.begin (), vecIntegers.end ()
34:       , [](int element) {return ((element % 2) == 0); }, -1);
```

```
35:
36:     cout << endl << "Contents of the vector after replacements:" << endl;
37:     DisplayContents(vecIntegers);
38:
39:     return 0;
40: }
```

▼ **输出：**

```
The initial contents of the vector are:
5 8 5 8 8 5 ¦ Number of elements: 6

Using 'std::replace' to replace value 5 by 8
Using 'std::replace_if' to replace even values by -1

Contents of the vector after replacements:
-1 -1 -1 -1 -1 -1 ¦ Number of elements: 6
```

▼ **分析：**

该示例给整型 vector vecIntegers 填充值，然后使用 STL 算法 std::random_shuffle 将这些值打乱，如第 24 行所示。第 30 行使用 replace()将所有的 5 都替换为 8。因此，第 33 行使用 replace_if()将所有偶数都替换为-1 后，集合包含 6 个元素，每个元素的值都为-1，如输出所示。

23.3.10 排序、在有序集合中搜索以及删除重复元素

在实际的应用程序中，经常需要排序以及在有序范围内（出于性能考虑）进行搜索。经常需要对一组信息进行排序，为此可使用 STL 算法 sort()：

```
sort (vecIntegers.begin (), vecIntegers.end ()); // ascending order
```

这个版本的 sort()将 std::less<>用作二元谓词，而该谓词使用 vector 存储的数据类型实现的运算符<。您可使用另一个重载版本，以指定谓词，从而修改排列顺序：

```
sort (vecIntegers.begin (), vecIntegers.end (),
     [](int lhs, int rsh) {return (lhs > rhs);} ); // descending order
```

同样，在显示集合的内容前，需要删除重复的元素。要删除相邻的重复值，可使用 unique()：

```
auto iNewEnd = unique (vecIntegers.begin (), vecIntegers.end ());
vecIntegers.erase (iNewEnd, vecIntegers.end ());  // to resize
```

要进行快速查找，可使用 STL 算法 binary_search()，这种算法只能用于有序容器：

```
bool bElementFound = binary_search (vecIntegers.begin (), vecIntegers.end (),
2011);

if (bElementFound)
   cout << "Element found in the vector!" << endl;
```

程序清单 23.10 使用 STL 算法 std::sort()将一个范围排序，使用 std::binary_search()在有序的范围内进行搜索，然后使用 std::unique()删除相邻的重复元素（执行 sort()后，重复的元素将彼此相邻）。

程序清单 23.10　使用 sort()、binary_search()和 unique()

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <string>
 3: #include <iostream>
 4: using namespace std;
 5:
 6: template <typename T>
 7: void DisplayContents(const T& Input)
```

```
 8: {
 9:    for ( auto iElement = Input.cbegin() // auto, cbegin: c++11
10:         ; iElement != Input.cend()  // cend() is new in C++11
11:         ; ++ iElement)
12:       cout << *iElement << endl;
13: }
14: int main ()
15: {
16:    vector<string> vecNames;
17:    vecNames.push_back ("John Doe");
18:    vecNames.push_back ("Jack Nicholson");
19:    vecNames.push_back ("Sean Penn");
20:    vecNames.push_back ("Anna Hoover");
21:
22:    // insert a duplicate into the vector
23:    vecNames.push_back ("Jack Nicholson");
24:
25:    cout << "The initial contents of the vector are:" << endl;
26:    DisplayContents(vecNames);
27:
28:    cout << "The sorted vector contains names in the order:" << endl;
29:    sort (vecNames.begin (), vecNames.end ());
30:    DisplayContents(vecNames);
31:
32:    cout << "Searching for \"John Doe\" using 'binary_search':" << endl;
33:    bool bElementFound = binary_search (vecNames.begin (), vecNames.end (),
34:                                        "John Doe");
35:
36:    if (bElementFound)
37:       cout << "Result: \"John Doe\" was found in the vector!" << endl;
38:    else
39:       cout << "Element not found " << endl;
40:
41:    // Erase adjacent duplicates
42:    auto iNewEnd = unique (vecNames.begin (), vecNames.end ());
43:    vecNames.erase (iNewEnd, vecNames.end ());
44:
45:    cout << "The contents of the vector after using 'unique':" << endl;
46:    DisplayContents(vecNames);
47:
48:    return 0;
49: }
```

▼ 输出：

```
The initial contents of the vector are:
John Doe
Jack Nicholson
Sean Penn
Anna Hoover
Jack Nicholson
The sorted vector contains names in the order:
Anna Hoover
Jack Nicholson
Jack Nicholson
John Doe
Sean Penn
Searching for "John Doe" using 'binary_search':
Result: "John Doe" was found in the vector!
The contents of the vector after using 'unique':
Anna Hoover
```

```
Jack Nicholson
John Doe
Sean Penn
```

▼ 分析：

上述代码首先对 vector vecNames 进行排序，如第 29 行所示，再使用 binary_search 在其中查找 John Doe，如第 33 行所示。同样，第 42 行使用 std::unique()删除第二个相邻的重复元素。请注意，与 remove()一样，unique()也不调整容器的大小。它将元素前移，但不会减少元素总数。为避免容器末尾包含不想要或未知的值，务必在调用 unique()后调用 vector::erase()，并将 unique()返回的迭代器传递给它，如第 42 和 43 行所示。

警告

binary_search()算法只能用于经过排序的容器，如果将其用于未经排序的 vector，结果可能出乎意料。

注意

stable_sort()的用法与 sort()类似，这在前面介绍过。stable_sort()确保排序后元素的相对顺序保持不变。为确保相对顺序保持不变，将降低性能，这是一个需要考虑的因素，尤其在元素的相对顺序不重要时。

23.3.11 将范围分区

std::partition()将输入范围分为两部分：一部分满足一元谓词；另一部分不满足。

```
bool IsEven (const int& nNumber)  // unary predicate
{
   return ((nNumber % 2) == 0);
}
...
partition (vecIntegers.begin(), vecIntegers.end(), IsEven);
```

然而，std::partition()不保证每个分区中元素的相对顺序不变。在相对顺序很重要，需要保持不变时，应使用 std::stable_partition()：

```
stable_partition (vecIntegers.begin(), vecIntegers.end(), IsEven);
```

程序清单 23.11 演示了这些算法的用法。

程序清单 23.11　使用 partition()和 stable_partition()将整型范围分为偶数值和奇数值

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3: using namespace std;
 4:
 5: bool IsEven (const int& nNumber)
 6: {
 7:    return ((nNumber % 2) == 0);
 8: }
 9:
10: template <typename T>
11: void DisplayContents(const T& Input)
12: {
13:    for ( auto iElement = Input.cbegin() // auto, cbegin: c++11
14:          ; iElement != Input.cend()  // cend() is new in C++11
15:          ; ++ iElement)
16:       cout << *iElement << ' ';
17:
18:    cout << "¦ Number of elements: " << Input.size() << endl;
19: }
```

```
20: int main ()
21: {
22:    vector <int> vecIntegers;
23:
24:    for (int nNum = 0; nNum < 10; ++ nNum)
25:       vecIntegers.push_back (nNum);
26:
27:    cout << "The initial contents: " << endl;
28:    DisplayContents(vecIntegers);
29:
30:    vector <int> vecCopy (vecIntegers);
31:
32:    cout << "The effect of using partition():" << endl;
33:    partition (vecIntegers.begin (), vecIntegers.end (), IsEven);
34:    DisplayContents(vecIntegers);
35:
36:    cout << "The effect of using stable_partition():" << endl;
37:    stable_partition (vecCopy.begin (), vecCopy.end (), IsEven);
38:    DisplayContents(vecCopy);
39:
40:    return 0;
41: }
```

▼ 输出：

```
The initial contents:
0 1 2 3 4 5 6 7 8 9 ¦ Number of elements: 10
The effect of using partition():
0 8 2 6 4 5 3 7 1 9 ¦ Number of elements: 10
The effect of using stable_partition():
0 2 4 6 8 1 3 5 7 9 ¦ Number of elements: 10
```

▼ 分析：

上述代码将包含在 vector vecIntegers 中的整型范围分为偶数和奇数。第一次分区是在第 33 行使用 std::partition()完成的，第二次是在第 37 行使用 stable_partition()完成的。为便于比较，这里将范围 vecIntegers 复制到 vecCopy 中，并对前者使用 std::partition()，对后者使用 std::stable_partition()。与使用 partition()相比，使用 stable_partition()的效果很明显，如输出所示。stable_partition()保持每个分区中元素的相对顺序不变。注意，保持相对顺序不变是有代价的，这种代价可能很小（如在这个例子中），也可能很大，这取决于包含在范围中的对象类型。

注意 stable_partition()的速度比 partition()慢，因此应只在容器中元素的相对顺序很重要时使用它。

23.3.12 在有序集合中插入元素

将元素插入到有序集合中时，将其插入到正确位置很重要。为满足这种需求，STL 提供了 lower_bound()和 upper_bound()等函数：

```
auto iMinInsertPos = lower_bound ( listNames.begin(), listNames.end()
                                 , "Brad Pitt" );
// alternatively:
auto iMaxInsertPos = upper_bound ( listNames.begin(), listNames.end()
                                 , "Brad Pitt" );
```

lower_bound()和 upper_bound()都返回一个迭代器，分别指向在不破坏现有顺序的情况下，元素可插入到有序范围内的最前位置和最后位置。

程序清单 23.12 演示了如何使用 lower_bound()将元素插入到在有序的人名 list 中的最前位置。

程序清单 23.12 使用 lower_bound()和 upper_bound()在有序范围中插入元素

```
 0: #include <algorithm>
 1: #include <list>
 2: #include <string>
 3: #include <iostream>
 4: using namespace std;
 5:
 6: template <typename T>
 7: void DisplayContents(const T& Input)
 8: {
 9:    for ( auto iElement = Input.cbegin() // auto, cbegin: c++11
10:         ; iElement != Input.cend()  // cend() is new in C++11
11:         ; ++ iElement)
12:       cout << *iElement << endl;
13: }
14: int main ()
15: {
16:    list<string> listNames;
17:
18:    // Insert sample values
19:    listNames.push_back ("John Doe");
20:    listNames.push_back ("Brad Pitt");
21:    listNames.push_back ("Jack Nicholson");
22:    listNames.push_back ("Sean Penn");
23:    listNames.push_back ("Anna Hoover");
24:
25:    cout << "The sorted contents of the list are: " << endl;
26:    listNames.sort ();
27:    DisplayContents(listNames);
28:
29:    cout << "The lowest index where \"Brad Pitt\" can be inserted is: ";
30:    auto iMinInsertPos = lower_bound ( listNames.begin (), listNames.end ()
31:                                     , "Brad Pitt" );
32:    cout << distance (listNames.begin (), iMinInsertPos) << endl;
33:
34:    cout << "The highest index where \"Brad Pitt\" can be inserted is: ";
35:    auto iMaxInsertPos = upper_bound ( listNames.begin (), listNames.end ()
36:                                     , "Brad Pitt" );
37:    cout << distance (listNames.begin (), iMaxInsertPos) << endl;
38:
39:    cout << endl;
40:
41:    cout << "List after inserting Brad Pitt in sorted order: " << endl;
42:    listNames.insert (iMinInsertPos, "Brad Pitt");
43:
44:    DisplayContents(listNames);
45:    return 0;
46: }
```

▼ 输出:

```
The sorted contents of the list are:
Anna Hoover
Brad Pitt
Jack Nicholson
John Doe
Sean Penn
The lowest index where "Brad Pitt" can be inserted is: 1
The highest index where "Brad Pitt" can be inserted is: 2
```

```
List after inserting Brad Pitt in sorted order:
Anna Hoover
Brad Pitt
Brad Pitt
Jack Nicholson
John Doe
Sean Penn
```

▼ **分析：**

可将元素插入到有序集合中的两个位置：一个是 lower_bound()返回的最前位置（离集合开头最近），另一个是 upper_bound()返回的迭代器，它是最后位置（离集合开头最远）。在程序清单 23.12 中，要插入到有序集合的字符串 Brad Pitt 已包含在集合中（这是在第 20 行插入的），因此最前位置和最后位置不同（否则，两者将相同）。在第 30 行和第 35 行分别调用了这两个函数。如输出所示，第 42 行使用 lower_bound()返回的迭代器将字符串插入 list 后，list 仍处于有序状态。因此，这些算法可帮助确定不破坏集合有序状态的元素插入位置。使用 upper_bound()返回的迭代器也如此。

应该	不应该
使用算法 remove()、remove_if()或 unique()后，务必使用容器的成员方法 erase()调整容器的大小。 使用 find()、find_if()、search()或 search_n()返回的迭代器之前，务必将其与容器的 end()进行比较，以确定它有效。 仅当元素的相对顺序很重要时，才应使用 stable_partition()（而不是 partition()）和 stable_sort()（而不是 sort()），因为 stable_*可能降低应用程序的性能。	调用 unique()删除重复的相邻值之前，别忘了使用 sort()对容器进行排序。sort()确保包含相同值的元素彼此相邻，这样 unique()才能发挥作用。 别忘了，binary_search()只能用于排序后的容器。

23.4 总结

本章介绍了 STL 中最重要、功能最强大的方面：算法。在本章中，读者了解了各种类型的算法，并通过示例对其用法有更清晰的认识。

23.5 问与答

问：诸如 std::transform()等变序算法能否用于关联容器（如 std::set）？

答：即使可以，也不应这样做。应将关联容器的内容视为常量，这是因为关联容器在插入元素时进行排序，因此元素的相对位置不仅对 find()等函数来说很重要，对容器的效率也很重要。因此，不应将诸如 std::transform()等变序算法用于 STL set。

问：要将顺序容器的每个元素都设置为特定的值，可使用 std::transform()吗？

答：虽然可以使用 std::transform()，但使用 fill()或 fill_n()更合适。

问：copy_backward()是否会反转目标容器中元素的排列顺序？

答：不会。STL 算法 copy_backward()按相反的顺序复制元素，但不改变元素的排列顺序，即它从

范围末尾开始复制到开头。如果要反转集合中元素的排列顺序，应使用 std::reverse()。

问：是否应对 list 使用 std::sort()?

答： std::sort()可用于 list，用法与用于其他顺序容器一样。然而，list 需要保持一个特殊特征：对 list 的操作不会导致现有迭代器失效，而 std::sort()不能保证该特征得以保持。因此，STL list 通过成员函数 list::sort()提供了 sort 算法。应使用该函数，因为它确保指向 list 中元素的迭代器不会失效，即使元素的相对位置发生了变化。

问：为什么在将元素插入到有序范围中时，使用 lower_bound()或 upper_bound()等函数很重要?

答： 这两个函数分别提供有序集合中的第一个位置和最后一个位置，将元素插入这些位置时不会破坏集合的有序状态。

23.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

23.6.1 测验

1. 要将 list 中满足特定条件的元素删除，应使用 std::remove_if()还是 list::remove_if()?
2. 假设有一个包含 ContactItem 对象的 list，在没有显式指定二元谓词时，函数 list::sort()将如何对这些元素进行排序?
3. STL 算法 generate()将调用函数 generator()多少次?
4. std::transform()与 std::for_each()之间的区别何在?

23.6.2 练习

1. 编写一个二元谓词，它接受字符串作为输入参数，并根据不区分大小写的比较结果返回一个值。
2. 演示 STL 算法（如 copy）如何使用迭代器实现其功能——复制两个类型不同的容器存储的序列，而无需知道目标集合的特征。
3. 您正在编写一个应用程序，它按星星在地平线上升起的顺序记录它们的特点。在天文学中，星球的大小很重要，其升起和落下的相对顺序亦如此。如果要根据星星的大小对这个集合进行排序，应使用 std::sort()还是 std::stable_sort()?

第 24 章
自适应容器：栈和队列

标准模板库（STL）提供了一些这样的容器，即使用其他容器模拟栈和队列的行为。这种内部使用一种容器但呈现另一种容器的行为特征的容器称为自适应容器（adaptive container）。

在本章中，您将学习：

- 栈和队列的行为特征；
- 使用 STL stack；
- 使用 STL queue；
- 使用 STL priority_queue。

24.1 栈和队列的行为特征

栈和队列与数组或 list 极其相似，但对插入、访问和删除元素的方式有一定的限制。可将元素插入到什么位置以及可从什么位置删除元素决定了容器的行为特征。

24.1.1 栈

栈是 LIFO（后进先出）系统，只能从栈顶插入或删除元素。可将栈视为一叠盘子，最后叠上去的盘子首先被取下来，而不能取下中间或底部的盘子。图 24.1 说明了这种“在顶部添加和删除”的元素组织方式。

泛型 STL 容器 std::stack 模拟了栈的这种行为。

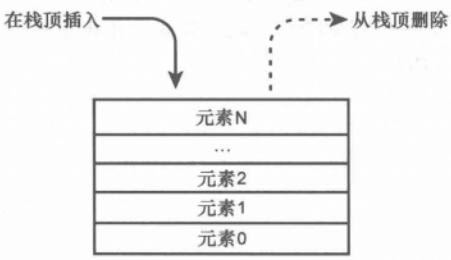


图 24.1 对栈的操作

提示

要使用 std::stack，必须包含头文件<stack>：

```
#include <stack>
```

24.1.2 队列

队列是 FIFO（先进先出）系统，元素被插入到队尾，最先插入的元素最先删除。可将队列视为一系列在邮局排队购买邮票的人：先加入队列的人先离开。图 24.2 说明了这种“在末尾加入、从开头删除”的元素组织方式。

泛型 STL 容器 std::queue 模拟了队列的这种行为。

提示

要使用 std::queue，必须包含头文件<queue>：

```
#include <queue>
```

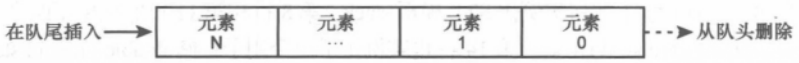


图 24.2 队列的操作

24.2 使用 STL stack 类

STL stack 是一个模板类，要使用它，必须包含头文件<stack>。它是一个泛型类，允许在顶部插入和删除元素，而不允许访问中间的元素。从这种角度看，std::stack 的行为很像一叠盘子。

24.2.1 实例化 stack

在有些 STL 实现中，std::stack 的定义如下：

```
template <
   class elementType,
   class Container=deque<Type>
> class stack;
```

参数 elementType 是 stack 存储的对象类型。第二个模板参数 Container 是 stack 使用的默认底层容器实现类。stack 默认在内部使用 std::deque 来存储数据，但可指定使用 vector 或 list 来存储数据。因此，实例化整型栈的代码类似于下面这样：

```
std::stack <int> stackInts;
```

要创建存储类（如 Tuna）对象的栈，可使用下述代码：

```
std::stack <Tuna> stackTunas;
```

要创建使用不同底层容器的栈，可使用如下代码：

```
std::stack <double, vector <double> > stackDoublesInVector;
```

程序清单 24.1 演示了各种实例化方式。

程序清单 24.1 实例化 STL stack

```
0: #include <stack>
1: #include <vector>
2:
3: int main ()
4: {
5:    using namespace std;
6:
7:    // A stack of integers
8:    stack <int> stackInts;
9:
```

```
10:     // A stack of doubles
11:     stack <double> stackDoubles;
12:
13:     // A stack of doubles contained in a vector
14:     stack <double, vector <double> > stackDoublesInVector;
15:
16:     // initializing one stack to be a copy of another
17:     stack <int> stackIntsCopy(stackInts);
18:
19:     return 0;
20: }
```

▼ **分析：**

该示例没有输出，但演示了如何实例化 STL 模板 stack。第 8 行和第 11 行实例化了两个 stack 对象，分别用于存储类型为 int 和 double 的元素。第 14 行也实例化了一个用于存储 double 元素的 stack，但将第二个模板参数（stack 在内部使用的集合类）指定为 vector。如果没有指定第二个模板参数，stack 将自动使用默认的 std::deque。最后，第 17 行表明，可使用一个 stack 对象的拷贝来创建另一个 stack 对象。

24.2.2 stack 的成员函数

stack 改变了另一种容器（如 deque、list 或 vector）的行为，通过限制元素插入或删除的方式实现其功能，从而提供严格遵守栈机制的行为特征。表 24.1 解释了 stack 类的公有成员函数并演示了如何将这些函数用于整型栈。

表 24.1　　std::stack 的成员函数

函　数	描　述
push	在栈顶插入元素 stackInts.push (25);
pop	删除栈顶的元素 stackInts.pop ();
empty	检查栈是否为空并返回一个布尔值 if (stackInts.empty ()) DoSomething ();
size	返回栈中的元素数 size_t nNumElements = stackInts.size ();
top	获得指向栈顶元素的引用 cout << "Element at the top = " << stackInts.top ();

如表所示，stack 的公有成员函数只提供了这样的方法，即插入或删除元素的位置符合栈的行为特征。也就是说，虽然底层容器可能是 deque、vector 或 list，但禁用了这些容器的有些功能，以实现栈的行为特征。

24.2.3 使用 push()和 pop()在栈顶插入和删除元素

要插入元素，可使用成员方法 stack<T>::push()：

```
stackInts.push (25); // 25 is atop the stack
```

根据定义，通常只能访问栈顶元素，为此可使用成员方法 top()：

```
cout << stackInts.top() << endl;
```

要删除栈顶元素，可使用成员方法 pop()：

```
stackInts.pop (); // pop: removes topmost element
```

程序清单 24.2 演示了如何使用 push()和 pop()在栈中插入和删除元素。

程序清单 24.2　使用整型 stack

```
 0: #include <stack>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:    using namespace std;
 6:    stack <int> stackInts;
 7:
 8:    // push: insert values at top of the stack
 9:    cout << "Pushing {25, 10, -1, 5} on stack in that order:" << endl;
10:    stackInts.push (25);
11:    stackInts.push (10);
12:    stackInts.push (-1);
13:    stackInts.push (5);
14:
15:    cout << "Stack contains " << stackInts.size () << " elements" << endl;
16:    while (stackInts.size () != 0)
17:    {
18:       cout << "Popping topmost element: " << stackInts.top() << endl;
19:       stackInts.pop (); // pop: removes topmost element
20:    }
21:
22:    if (stackInts.empty ())  // true: due to previous pop()s
23:       cout << "Popping all elements empties stack!" << endl;
24:
25:    return 0;
26: }
```

▼ 输出：

```
Pushing {25, 10, -1, 5} on stack in that order:
Stack contains 4 elements
Popping topmost element: 5
Popping topmost element: -1
Popping topmost element: 10
Popping topmost element: 25
Popping all elements empties stack!
```

▼ 分析：

该示例首先使用 stack::push()将一些值插入到整型 stack stackInts 中，如第 10～13 行所示；然后使用 stack::pop()从 stack 中删除元素。stack 只允许访问栈顶元素，可使用成员 stack::top()访问栈顶元素，如第 18 行所示。使用 stack::pop()可每次从 stack 中删除一个元素，如第 19 行所示。第 19 行所属的 while 循环确保不断执行 pop()操作，直到 stack 为空。从元素弹出的顺序可知，最后插入的元素最先弹出，这说明了 stack 的典型 LIFO（后进先出）特征。

程序清单 24.2 演示了 stack 的所有 5 个成员函数。注意，被 stack 类用作底层容器的所有 STL 顺序容器都提供了 push_back 和 insert，但它们不是 stack 的公有成员函数；用于访问非容器顶部元素的迭代器也如此。stack 只暴露了栈顶元素，而没有暴露其他任何元素。

24.3　使用 STL queue 类

STL queue 是一个模板类，要使用它，必须包含头文件<queue>。queue 是一个泛型类，只允许在末尾插入元素以及从开头删除元素。queue 不允许访问中间的元素，但可以访问开头和末尾的元素。从这种意义上说，std::queue 的行为与超市收银台前的队列极其相似。

24.3.1 实例化 queue

std::queue 的定义如下：

```
template <
   class elementType,
   class Container = deque<Type>
> class queue;
```

其中 elementType 是 queue 对象包含的元素的类型。Container 是 std::queue 用于存储其数据的集合类型，可将该模板参数设置为 std::list、vector 或 deque，默认为 deque。

实例化整型 queue 的最简单方式如下：

```
std::queue <int> qIntegers;
```

如果要创建这样的 queue，即其元素类型为 double，并使用 std::list（而不是默认的 queue）存储这些元素，可以像下面这样做：

```
std::queue <double, list <double> > qDoublesInList;
```

与 stack 一样，也可使用一个 queue 来实例化另一个 queue：

```
std::queue<int> qCopy(qIntegers);
```

程序清单 24.3 演示了各种实例化 std::queue 的方式。

程序清单 24.3 实例化 STL queue

```
0: #include <queue>
1: #include <list>
2:
3: int main ()
4: {
5:    using namespace std;
6:
7:    // A queue of integers
8:    queue <int> qIntegers;
9:
10:   // A queue of doubles
11:   queue <double> qDoubles;
12:
13:   // A queue of doubles stored internally in a list
14:   queue <double, list <double> > qDoublesInList;
15:
16:   // one queue created as a copy of another
17:   queue<int> qCopy(qIntegers);
18:
19:   return 0;
20: }
```

▼ 分析：

上述示例演示了如何实例化 STL 泛型类 queue，第 8 行创建了一个整型 queue，而第 11 行创建了一个双精度型 queue。第 14 行实例化 queue qDoublesInList 时，显式地指定 queue 使用底层容器 std::list 来管理内部数据，这是通过第二个模板参数指定的。如果没有指定第二个模板参数（就像实例化前两个 queue 那样），默认将使用底层容器 std::deque 来管理 queue 的内容。

24.3.2 queue 的成员函数

与 std::stack 一样，std::queue 的实现也是基于 STL 容器 vector、list 或 deque 的。queue 提供了几个

成员函数来实现队列的行为特征。表 24.2 通过程序清单 24.3 所示的整型 queue qIntegers 解释了 queue 的成员函数。

表 24.2 std::queue 的成员函数

函　数	描　述
push	在队尾（即最后一个位置）插入一个元素 qIntegers.push (25);
pop	将队首（即最开始位置）的元素删除 qIntegers.pop ();
front	返回指向队首元素的引用 cout << "Element at front: " << qIntegers.front ();
back	返回指向队尾元素（即最后插入的元素）的引用 cout << "Element at back: " << qIntegers. back ();
empty	检查队列是否为空并返回一个布尔值 if (qIntegers.empty ()) cout << "The queue is empty!";
size	返回队列中的元素数 size_t nNumElements = qIntegers.size ();

STL queue 没有提供 begin()和 end()等函数，而大多数 STL 容器都提供了这些函数，包括 queue 类在底层使用的 deque、vector 或 list。这是有意为之的，旨在只允许对 queue 执行符合队列行为特征的操作。

24.3.3 使用 push()在队尾插入以及使用 pop()从队首删除

对于 queue，元素在末尾插入，这是使用成员方法 push()完成的：

```
qIntegers.push (5);  // elements pushed are inserted at the end
```

删除是在开头进行的，这是使用成员方法 pop()完成的：

```
qIntegers.pop ();  // removes element at front
```

与 stack 不同，queue 允许查看其两端的元素，即容器的开头和末尾：

```
cout << "Element at front: " << qIntegers.front() << endl;
cout << "Element at back: " << qIntegers.back () << endl;
```

程序清单 24.4 演示了如何插入、删除和查看元素。

程序清单 24.4　在整型 queue 中插入、删除和查看元素

```
 0: #include <queue>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:    using namespace std;
 6:    queue <int> qIntegers;
 7:
 8:    cout << "Inserting {10, 5, -1, 20} into queue" << endl;
 9:    qIntegers.push (10);
10:    qIntegers.push (5);  // elements pushed are inserted at the end
11:    qIntegers.push (-1);
12:    qIntegers.push (20);
13:
14:    cout << "Queue contains " << qIntegers.size () << " elements" << endl;
15:    cout << "Element at front: " << qIntegers.front() << endl;
16:    cout << "Element at back: " << qIntegers.back () << endl;
```

```
17:
18:    while (qIntegers.size () != 0)
19:    {
20:       cout << "Deleting element: " << qIntegers.front () << endl;
21:       qIntegers.pop ();  // removes element at front
22:    }
23:
24:    if (qIntegers.empty ())  // true as all elements have been pop()-ed
25:       cout << "The queue is now empty!" << endl;
26:
27:    return 0;
28: }
```

▼ **输出：**

```
Inserting {10, 5, -1, 20} into queue
Queue contains 4 elements
Element at front: 10
Element at back: 20
Deleting element: 10
Deleting element: 5
Deleting element: -1
Deleting element: 20
```

▼ **分析：**

在第 9～12 行，使用 push()在队列 qIntegers 的末尾插入元素。第 15 行和第 16 行分别使用函数 front()和 back()引用了队首和队尾的元素。第 18～22 行的 while 循环显示队首的元素，然后使用 pop()删除它（第 21 行），直到队列为空。从输出可知，元素被删除的顺序与插入顺序相同，因为元素在队尾插入，从队首删除。

24.4 使用 STL 优先级队列

STL priority_queue 是一个模板类，要使用它，也必须包含头文件<queue>。priority_queue 与 queue 的不同之处在于，包含最大值（或二元谓词认为是最大值）的元素位于队首，且只能在队首执行操作。

24.4.1 实例化 priority_queue 类

std::priority_queue 类的定义如下：

```
template <
   class elementType,
   class Container=vector<Type>,
      class Compare=less<typename Container::value_type>
>
class priority_queue
```

其中 elementType 是一个模板参数，指定了优先级队列将包含的元素的类型。第二个模板参数指定 priority_queue 在内部将使用哪个集合类来存储数据，第三个参数让程序员能够指定一个二元谓词，以帮助队列判断哪个元素应位于队首。如果没有指定二元谓词，priority_queue 类将默认使用 std::less，它使用运算符<比较对象。

要实例化整型 priority_queue，最简单的方式如下：

```
std::priority_queue <int> pqIntegers;
```

如果要创建一个这样的 priority_queue，即其元素类型为 double，且按小到大的顺序存储在 std::deque 中，则可这样做：

```
priority_queue <int, deque <int>, greater <int> > pqIntegers_Inverse;
```

与 stack 一样，也可使用一个 priority_queue 来实例化另一个 priority_queue：

```
std::priority_queue <int>  pqCopy(pqIntegers);
```

程序清单 24.5 演示了如何实例化 priority_queue 对象。

程序清单 24.5 实例化 STL priority_queue

```
 0: #include <queue>
 1:
 2: int main ()
 3: {
 4:    using namespace std;
 5:
 6:    // A priority queue of integers sorted using std::less <> (default)
 7:    priority_queue <int> pqIntegers;
 8:
 9:    // A priority queue of doubles
10:    priority_queue <double> pqDoubles;
11:
12:    // A priority queue of integers sorted using std::greater <>
13:    priority_queue <int, deque <int>, greater <int> > pqIntegers_Inverse;
14:
15:    // a priority queue created as a copy of another
16:    priority_queue <int>  pqCopy(pqIntegers);
17:
18:    return 0;
19: }
```

▼ 分析：

第 7 行和第 10 行实例化了两个 priority_queue，其元素类型分别为 int 和 double。由于没有指定其他模板参数，因此将默认使用 std::vector 作为内部数据的容器，并默认使用 std::less 提供的比较标准。因此，这两个队列将包含的值最大的元素放在队首。然而，实例化 pqIntegers_Inverse 时，通过第二个参数指定使用 deque 作为内部容器，并将谓词指定为 std::greater，该谓词导致最小的元素位于队首。

本章后面的程序清单 24.7 说明了使用谓词 std::greater<T>带来的影响。

24.4.2 priority_queue 的成员函数

queue 提供了成员函数 front()和 back()，但 priority_queue 没有。表 24.3 简要地介绍了 priority_queue 的成员函数。

表 24.3 std::priority_queue 的成员函数

函　数	描　述
push	在优先级队列中插入一个元素 pqIntegers.push (10);
pop	删除队首元素，即最大的元素 pqIntegers.pop ();
top	返回指向队列中最大元素（即队首元素）的引用 pqIntegers. cout << "The largest element inpriority queue is: " << pqIntegers.top ();
empty	检查优先级队列是否为空并返回一个布尔值 if (pqIntegers.empty ()) cout << "The queue is empty!";
size	返回优先级队列中的元素个数 size_t nNumElements = pqIntegers.size ();

从该表可知，只能使用top()来访问队列的成员，该函数返回值最大的元素，最大的元素是根据用户指定的谓词或默认的std::less确定的。

24.4.3 使用push()在priority_queue末尾插入以及使用pop()在priority_queue开头删除

要在priority_queue中插入元素，可使用成员方法push()：

```
pqIntegers.push (5);  // elements are placed in sorted order
```

要在priority_queue开头删除元素，可使用pop()：

```
pqIntegers.pop ();  // removes element at front
```

程序清单24.6演示了如何使用priority_queue的成员函数。

程序清单24.6 使用priority_queue的成员函数push()、top()和pop()

```
 0: #include <queue>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:    using namespace std;
 6:
 7:    priority_queue <int> pqIntegers;
 8:    cout << "Inserting {10, 5, -1, 20} into the priority_queue" << endl;
 9:    pqIntegers.push (10);
10:    pqIntegers.push (5);  // placed in sorted order
11:    pqIntegers.push (-1);
12:    pqIntegers.push (20);
13:
14:    cout << "Deleting the " << pqIntegers.size () << " elements" << endl;
15:    while (!pqIntegers.empty ())
16:    {
17:       cout << "Deleting topmost element: " << pqIntegers.top () << endl;
18:       pqIntegers.pop ();
19:    }
20:
21:    return 0;
22: }
```

▼ 输出：

```
Inserting {10, 5, -1, 20} into the priority_queue
Deleting the 4 elements
Deleting topmost element: 20
Deleting topmost element: 10
Deleting topmost element: 5
Deleting topmost element: -1
```

▼ 分析：

这个示例首先将一些整数插入到priority_queue中（如第9～12行所示），然后使用pop()删除队首元素，如第18行所示。从输出可知，值最大的元素位于队首，因此调用priority_queue::pop()将删除容器中值最大的元素；可通过方法top()访问该元素，如第17行所示。由于这里没有提供优先级谓词，优先级队列自动将元素按降序排列（最大的值位于队首）。

下一个示例（程序清单24.7）使用谓词std::greater <int>实例化一个priority_queue。该谓词导致优先级队列认为包含的数字最小的元素为最大的元素，并将其放在队首。

程序清单 24.7 通过使用谓词将值最小的元素放在 priority_queue 开头

```
 0: #include <queue>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:    using namespace std;
 6:
 7:    // Define a priority_queue object with greater <int> as predicate
 8:    priority_queue <int, vector <int>, greater <int> > pqIntegers;
 9:
10:    cout << "Inserting {10, 5, -1, 20} into the priority queue" << endl;
11:    pqIntegers.push (10);
12:    pqIntegers.push (5);
13:    pqIntegers.push (-1);
14:    pqIntegers.push (20);
15:
16:    cout << "Deleting " << pqIntegers.size () << " elements" << endl;
17:    while (!pqIntegers.empty ())
18:    {
19:       cout << "Deleting topmost element " << pqIntegers.top () << endl;
20:       pqIntegers.pop ();
21:    }
22:
23:    return 0;
24: }
```

▼ 输出：

```
Inserting {10, 5, -1, 20} into the priority queue
Deleting 4 elements
Deleting topmost element -1
Deleting topmost element 5
Deleting topmost element 10
Deleting topmost element 20
```

▼ 分析：

在这个示例中，大多数代码以及提供给 priority_queue 的所有值都与前一个示例（程序清单 24.6）相同，但输出表明这两个队列的行为不同。这个 priority_queue 使用谓词 greater <int>比较其元素，如第 8 行所示。该谓词导致包含的整数最小的元素被认为是最大的，因此放在队首。这样，第 19 行使用的函数 top()总是显示 priority_queue 中最小的整数，然后第 20 行使用 pop()将其删除。

因此，弹出元素时，该 priority_queue 按升序弹出整数。

24.5 总结

本章阐述了 3 个重要的自适应容器——STL stack、queue 和 priority_queue。这些容器使用顺序容器并对其进行改造，以满足其内部数据存储需求，再通过成员函数呈现出栈与队列独特的行为特征。

24.6 问与答

问：能否修改栈中间的元素？

答：不能，这不符合栈的行为特征。

问：能否对队列中的所有元素进行迭代？

答：队列不支持迭代器，只能访问队尾的元素。

问：STL 算法能否用于自适应容器？

答：STL 算法使用迭代器。由于 stack 和 queue 类都没有提供标记范围两端的迭代器，因此无法将 STL 算法用于这些容器。

24.7 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

24.7.1 测验

1. 能否修改 priority_queue 的行为，使得值最大的元素最后弹出？
2. 假设有一个包含 Coins 对象的 priority_queue，要让 priority_queue 将币值最大的硬币放在队首，需要为 Coins 定义哪种成员运算符？
3. 假设有一个包含 6 个 Coins 对象的 stack，能否访问或删除第一个插入的 Coins 对象？

24.7.2 练习

1. 邮局有一个包含人（Person 类）的队列。Person 包含两个成员属性，分别用于存储年龄和性别，其定义如下：

```
class Person
{
    public:
        int Age;
        bool IsFemale;
};
```

请编写一个二元谓词，帮助 priority_queue 优先向老人和妇女提供服务。

2. 编写一个程序，使用 stack 类反转用户输入的字符串的排列顺序。

第 25 章
使用 STL 位标志

位是存储设置与标志的高效方法。标准模板库（STL）提供了可帮助组织与操作位信息的类。
在本章中，您将学习：

- bitset 类；
- vector<bool>。

25.1 bitset 类

std::bitset 是一个 STL 类，用于处理以位和位标志表示的信息。std::bitset 不是 STL 容器类，因为它不能调整长度。这是一个实用类，针对处理长度在编译阶段已知的位序列进行了优化。

提示

要使用 std::bitset 类，必须包含头文件<bitset>：

```
#include <bitset>
```

实例化 std::bitset

实例化这个模板类时，必须通过一个模板参数指定实例需要管理的位数：

```
bitset <4> fourBits;  // 4 bits initialized to 0000
```

还可将 bitset 初始化为一个用字符串字面量（char*）表示的位序列：

```
bitset <5> fiveBits ("10101"); // 5 bits 10101
```

使用一个 bitset 来实例化另一个 bitset 非常简单：

```
bitset <8> eightBitsCopy(eightbits);
```

程序清单 25.1 演示了一些实例化 bitset 类的方式。

程序清单 25.1　实例化 std::bitset

```
 0: #include <bitset>
 1: #include <iostream>
 2: #include <string>
 3:
 4: int main ()
 5: {
 6:    using namespace std;
 7:
 8:    // instantiate a bitset object
 9:    bitset <4> fourBits;  // 4 bits initialized to 0000
10:    cout << "Initial contents of fourBits: " << fourBits << endl;
11:
```

```
12:    bitset <5> fiveBits ("10101"); // 5 bits 10101
13:    cout << "Initial contents of fiveBits: " << fiveBits << endl;
14:
15:    bitset <8> eightbits (255); // 8 bits initialized to long int 255
16:    cout << "Initial contents of eightBits: " << eightbits << endl;
17:
18:    // instantiate one bitset as a copy of another
19:    bitset <8> eightBitsCopy(eightbits);
20:
21:    return 0;
22: }
```

▼ **输出：**

```
Initial contents of fourBits: 0000
Initial contents of fiveBits: 10101
Initial contents of eightBits: 11111111
```

▼ **分析：**

该示例演示了4种创建bitset对象的方式：通过默认构造函数将位序列初始化为0（如第9行所示）；通过C风格字符串指定位序列的字符串表示（如第12行所示）；通过unsigned long指定二进制序列对应的十进制值（如第15行所示）；使用复制构造函数（如第19行所示）。注意，在每个实例中，都需要通过一个模板参数指定位序列包含的位数。位数在编译阶段指定，而不是动态的。指定bitset的位数后，便不能插入更多的位，而不像vector那样调整在编译阶段指定的长度。

25.2 使用std::bitset及其成员

bitset类提供了很多成员函数，可用于在bitset中插入位、设置（重置）内容、读取内容（将内容写入到流中）。它还提供了一些运算符，用于显示位序列、执行按位逻辑运算等。

25.2.1 std:bitset的运算符

第12章介绍了运算符，还介绍了运算符最重要的作用是提高类的可用性。std::bitset提供一些很有用的运算符，如表25.1所示，这些运算符让bitset使用起来非常容易。表25.1通过程序清单25.1所示的bitset对象fourBits演示这些运算符的用法。

表25.1　　std::bitset提供的运算符

运　算　符	描　　述
运算符<<	将位序列的文本表示插入到输出流中 cout << fourBits;
运算符>>	将一个字符串插入到bitset对象中 "0101" >> fourBits;
运算符&	执行按位与操作 bitset <4> result (fourBits1 & fourBits2);
运算符|	执行按位或操作 bitwise <4> result (fourBits1 | fourBits2);
运算符^	执行按位异或操作 bitwise <4> result (fourBits1 ^ fourBits2);
运算符~	执行按位取反操作 bitwise <4> result (~fourBits1);

续表

运算符 >>=	执行按位右移操作 fourBits >>= (2); //右移两位
运算符<<=	执行按位左移操作 fourBits <<= (2); // 左移两位

运算符[N]	返回指向位序列中第（N+1）位的引用
	fourBits [2] = 0; // 将第 3 位设置为 0
	bool bNum = fourBits [2]; //读取第 3 位

除这些运算符外，std::bitset 还提供了|=、&=、^=和~=等运算符，用于对 bitset 对象执行按位操作。

25.2.2 std::bitset 的成员方法

位可以存储两种状态：要么是已设置（1），要么是重置（0）。要对 bitset 的内容进行操作，可使用表 25.2 列出的成员函数对 bitset 中的一位或所有位进行操作。

表 25.2 std::bitset

函　数	描　述
set	将序列中的所有位都设置为 1 fourBits.set (); //现在序列包含 1111
set (N, val=1)	将第 N+1 位设置为 val 指定的值（默认为 1） fourBits.set (2, 0); // 将第 3 位设置为 0
reset	将序列中的所有位都重置为 0 fourBits.reset (); // 现在序列包含 0000
reset (N)	将偏移位置为（N+1）的位清除 fourBits.reset (2); //现在第 3 位的值为 0
flip	将位序列中的所有位取反 fourBits.flip (); // 0101 将变为 1010
size	返回序列中的位数 size_t NumBits = fourBits.size (); // 返回 4
count	返回序列中值为 1 的位数 size_t NumBitsSet = fourBits.count (); size_t NumBitsReset = fourBits.size () – fourBits.count ();

程序清单 25.2 演示了这些成员方法和运算符的用法。

程序清单 25.2 使用 bitset 执行逻辑运算

```
 0: #include <bitset>
 1: #include <string>
 2: #include <iostream>
 3:
 4: int main ()
 5: {
 6:    using namespace std;
 7:    bitset <8> inputBits;
 8:    cout << "Enter a 8-bit sequence: ";
 9:
10:    cin >> inputBits;  // store user input in bitset
11:
12:    cout << "Number of 1s you supplied: " << inputBits.count () << endl;
13:    cout << "Number of 0s you supplied: ";
14:    cout << inputBits.size () - inputBits.count () << endl;
```

```
15:
16:    bitset <8> inputFlipped (inputBits);  // copy
17:    inputFlipped.flip ();  // toggle the bits
18:
19:    cout << "Flipped version is: " << inputFlipped << endl;
20:
21:    cout << "Result of AND, OR and XOR between the two:" << endl;
22:    cout << inputBits << " & " << inputFlipped << " = ";
23:    cout << (inputBits & inputFlipped) << endl;  // bitwise AND
24:
25:    cout << inputBits << " ¦ " << inputFlipped << " = ";
26:    cout << (inputBits ¦ inputFlipped) << endl;  // bitwise OR
27:
28:    cout << inputBits << " ^ " << inputFlipped << " = ";
29:    cout << (inputBits ^ inputFlipped) << endl; // bitwise XOR
30:
31:    return 0;
32: }
```

▼ 输出：

```
Enter a 8-bit sequence: 10110101
Number of 1s you supplied: 5
Number of 0s you supplied: 3
Flipped version is: 01001010
Result of AND, OR and XOR between the two:
10110101 & 01001010 = 00000000
10110101 ¦ 01001010 = 11111111
10110101 ^ 01001010 = 11111111
```

▼ 分析：

该示例是一个交互式程序，它不仅演示了使用 std::bitset 在两个位序列之间执行按位运算很简单，还演示了如何使用 std::bitset 的流运算符。std::bitset 实现了移位运算符（>>和<<），让您能够轻松地将位序列打印到屏幕上以及读取用户以字符串形式输入的位序列。第 10 行将用户提供的序列填充到 inputBits 中。第 12 行使用 count()获得序列中值为 1 的位数。为计算序列中值为零的位数，将返回 bitset 中位数的 size()与 count()相减，如第 14 行所示。inputFlipped 位于 inputBits 的一个副本的开头，然后使用 flip()将序列中所有位取反，如第 17 行所示。现在它包含的序列中每位的值都与原来相反（即 0 变成 1，1 变成 0）。其他代码演示了对两个 bitset 执行 AND、OR 和 XOR 等操作的结果。

注意

STL bitset 的缺点之一是不能动态地调整长度。仅当在编辑阶段知道序列将存储多少位时才能使用 bitset。

为克服这种缺点，STL 向程序员提供了 vector<bool>类（在有些 STL 实现中为 bit_vector）。

25.3 vector<bool>

vector<bool>是对 std::vector 的部分具体化，用于存储布尔数据。这个类可动态地调整长度，因此程序员无需在编译阶段知道要存储的布尔标志数。

提示

要使用 std::vector<bool>类，必须包含头文件<vector>：

```
#include <vector>
```

25.3.1 实例化 vector<bool>

实例化 vector<bool>的方式与实例化 vector 类似，有一些方便的重载构造函数可供使用：

```
vector <bool> vecBool1;
```

例如，可创建一个这样的 vector，即它最初包含 10 个布尔元素，且每个元素都被初始化为 1（即 true）：

```
vector <bool> vecBool2 (10, true);
```

还可使用一个 vector<bool>创建另一个 vector<bool>：

```
vector <bool> vecBool2Copy (vecBool2);
```

程序清单 25.3 演示了一些实例化 vector<bool>的方式。

程序清单 25.3 实例化 vector<bool>

```
 0: #include <vector>
 1:
 2: int main ()
 3: {
 4:     using namespace std;
 5:
 6:     // Instantiate an object using the default constructor
 7:     vector <bool> vecBool1;
 8:
 9:     // Initialize a vector with 10 elements with value true
10:     vector <bool> vecBool2 (10, true);
11:
12:     // Instantiate one object as a copy of another
13:     vector <bool> vecBool2Copy (vecBool2);
14:
15:     return 0;
16: }
```

▼ 分析：

该示例演示了一些创建 vector<bool>对象的方式：第 7 行使用默认构造函数；第 10 行创建了一个包含 10 个布尔标志的对象，其中每个标志的值都为 true；第 13 行演示了如何通过复制 vector<bool>对象来创建另一个 vector<bool>对象。

25.3.2 vector<bool>的成员函数和运算符

vector<bool>提供了函数 flip()，用于将序列中的布尔值取反，这与函数 bitset<>::flip()很像。

除了这个方法外，vector<bool>与 std::vector 极其相似，例如，可使用 push_back 将标志位插入到序列中。程序清单 25.4 更详细地演示了这个类的用法。

程序清单 25.4 使用 vector<bool>

```
 0: #include <vector>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: int main ()
 5: {
 6:    vector <bool> vecBoolFlags (3);  // instantiated to hold 3 bool flags
 7:    vecBoolFlags [0] = true;
 8:    vecBoolFlags [1] = true;
 9:    vecBoolFlags [2] = false;
10:
11:    vecBoolFlags.push_back (true); // insert a fourth bool at the end
12:
13:    cout << "The contents of the vector are: " << endl;
```

```
14:    for (size_t nIndex = 0; nIndex < vecBoolFlags.size (); ++ nIndex)
15:       cout << vecBoolFlags [nIndex] << ' ';
16:
17:    cout << endl;
18:    vecBoolFlags.flip ();
19:
20:    cout << "The contents of the vector are: " << endl;
21:    for (size_t nIndex = 0; nIndex < vecBoolFlags.size (); ++ nIndex)
22:       cout << vecBoolFlags [nIndex] << ' ';
23:
24:    cout << endl;
25:
26:    return 0;
27: }
```

▼ **输出：**

```
The contents of the vector are:
1 1 0 1
The contents of the vector are:
0 0 1 0
```

▼ **分析：**

在这个示例中，通过运算符[]访问了 vector 中的布尔标志（如第 7～9 行所示），这与访问普通 vector 很像。第 18 行使用函数 flip()将位标志取反，即将所有 0 都转换为 1，将所有 1 都转换为 0。注意到第 11 行使用了 push_back。虽然第 6 行将 vecBoolFlags 初始化为包含 3 个标志，但可动态地添加标志，如第 11 行所示；而使用 std::bitset 时，标志数是在编译阶段指定的，不能增加。

25.4 总结

本章介绍了用于处理位序列和位标志最有效的工具：std::bitset 类。另外还介绍 vector<bool>类，它也可以存储布尔标志——其位数不需要在编译时就确定。

25.5 问与答

问：在可以使用 std::bitset 和 vector<bool>的情况下，您将选择使用哪个类来存储二进制标志？

答：bitset，因为它更适合这种需求。

问：假设有一个名为 myBitSeq 的 std::bitset 对象，它包含一定数量的位。如何确定它包含多少个值为 0（或 false）的位？

答：bitset::count()返回值为 1 的位数，bitset::size()返回总位数，将后者减去前者将得到序列中值为 0 的位数。

问：能否使用迭代器来访问 vector<bool>中的元素？

答：可以。vector<bool>是 std::vector 的部分具体化，它支持迭代器。

问：能否在编译阶段指定要存储在 vector<bool>中的元素个数？

答：可以。为此，可调用重载构造函数并指定元素数，也可在实例化后调用函数 vector<bool>::resize。

25.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

25.6.1 测验

1. bitset 能否扩展其内部缓冲区以存储可变的元素数？
2. 为什么 bitset 不属于 STL 容器类？
3. 您会使用 std::vector 来存储位数在编译阶段就知道的固定位数吗？

25.6.2 练习

1. 创建一个长 4 位的 bitset 对象，并使用一个数字来初始化它，然后显示结果并将其与另一个 bitset 对象相加（注意：bitsets 不支持语法 bitsetA = bitsetX + bitsetY）。
2. 请演示如何将 bitset 对象中的位取反。

第 26 章
理解智能指针

管理堆（或自由存储区）中的内存时，C++程序员并非一定要使用常规指针，而可使用智能指针。

在本章中，您将学习：

- 什么是智能指针以及为什么需要智能指针；
- 智能指针是如何实现的；
- 各种智能指针；
- 为何不应使用已摒弃的 std::auto_ptr；
- C++标准库提供的智能指针类 std::unique_ptr；
- 深受欢迎的智能指针库。

26.1 什么是智能指针

简单地说，C++智能指针是包含重载运算符的类，其行为像常规指针，但智能指针能够及时妥善地销毁动态分配的数据，并实现了明确的对象生命周期，因此更有价值。

26.1.1 常规（原始）指针存在的问题

与其他现代编程语言不同，C++在内存分配、释放和管理方面向程序员提供了全面的灵活性。不幸的是，这种灵活性是把双刃剑，一方面，它使 C++成为一种功能强大的语言，另一方面，它让程序员能够制造与内存相关的问题，如动态分配的对象没有正确地释放时将导致内存泄漏。

例如：

```
CData *pData = mObject.GetData ();
/*
   Questions: Is object pointed by pData dynamically allocated using new?
   Who will perform delete: caller or the called?
   Answer: No idea!
*/
pData->Display ();
```

在上述代码中，没有显而易见的方法获悉 pData 指向的内存：

- 是否是从堆中分配的，因此最终需要释放；
- 是否由调用者负责释放；
- 对象的析构函数是否会自动销毁该对象。

虽然这种不明确性可通过添加注释以及遵循编码实践来部分缓解，但这些机制太松散，无法有效地避免因滥用动态分配的数据和指针而导致的错误。

26.1.2 智能指针有何帮助

鉴于使用常规指针以及常规的内存管理方法存在的问题，当C++程序员需要管理堆（自由存储区）中的数据时，并非一定要使用它们，而可在程序中使用智能指针，以更智能的方式分配和管理内存：

```
smart_pointer<CData> spData = mObject.GetData ();

// Use a smart pointer like a conventional pointer!
spData->Display ();
(*spData).Display ();

// Don't have to worry about de-allocation
// (the smart pointer's destructor does it for you)
```

智能指针的行为类似常规指针（这里将其称为原始指针），但通过重载的运算符和析构函数确保动态分配的数据能够及时地销毁，从而提供了更多有用的功能。

26.2 智能指针是如何实现的

这个问题暂时可以简化为："智能指针 spData 是如何做到像常规指针的？"答案如下：智能指针类重载了解除引用运算符（*）和成员选择运算符（->），让程序员可以像使用常规指针那样使用它们。运算符重载在第12章讨论过。

另外，为让您能够在堆中管理各种类型，几乎所有良好的智能指针类都是模板类，包含其功能的泛型实现。由于是模板，它们是通用的，可以根据要管理的对象类型进行具体化。

程序清单26.1是一个简单智能指针类的实现。

程序清单26.1 智能指针类最基本的组成部分

```
 0: template <typename T>
 1: class smart_pointer
 2: {
 3: private:
 4:     T* m_pRawPointer;
 5: public:
 6:     smart_pointer (T* pData) : m_pRawPointer (pData) {}     // constructor
 7:     ~smart_pointer () {delete pData;};                      // destructor
 8:
 9:     // copy constructor
10:     smart_pointer (const smart_pointer & anotherSP);
11:     // copy assignment operator
12:     smart_pointer& operator= (const smart_pointer& anotherSP);
13:
14:     T& operator* () const        // dereferencing operator
15:     {
16:         return *(m_pRawPointer);
17:     }
18:
19:     T* operator-> () const        // member selection operator
20:     {
21:         return m_pRawPointer;
22:     }
23: };
```

▼ 分析：

该智能指针类实现了两个运算符：*和->，如第14～17行及第19～22行所示，它们让这个类

能够用作常规意义上的“指针”。例如，如果有一个 Tuna 类，则可这样对该类型的对象使用智能指针：

```
smart_pointer <Tuna> pSmartTuna (new Tuna);
pSmartTuna->Swim();
// Alternatively:
(*pSmartDog).Swim ();
```

这个 smart_pointer 类还没有实现使其非常智能，从而胜于常规指针的功能。构造函数（如第 6 行所示）接受一个指针，并将其保存到该智能指针类内部的一个指针对象中。析构函数释放该指针，从而实现了自动内存释放。

注意 使智能指针真正“智能”的是复制构造函数、赋值运算符和析构函数的实现，它们决定了智能指针对象被传递给函数、赋值或离开作用域（即像其他类对象一样被销毁）时的行为。介绍完整的智能指针实现前，需要了解一些智能指针类型。

26.3 智能指针类型

内存资源管理（即实现的内存所有权模型）是智能指针类与众不同的地方。智能指针决定在复制和赋值时如何处理内存资源。最简单的实现通常会导致性能问题，而最快的实现可能并非适合所有应用程序。因此，在应用程序中使用智能指针前，程序员应理解其工作原理。

智能指针的分类实际上就是内存资源管理策略的分类，可分为如下几类：

- 深复制；
- 写时复制（Copy on Write，COW）；
- 引用计数；
- 引用链接；
- 破坏性复制。

下面首先简要地介绍一下这些策略，再探索 C++标准库提供的智能指针 std::unique_ptr。

26.3.1 深复制

在实现深复制的智能指针中，每个智能指针实例都保存一个它管理的对象的完整副本。每当智能指针被复制时，将复制它指向的对象（因此称为深复制）。每当智能指针离开作用域时，将（通过析构函数）释放它指向的内存。

虽然基于深复制的智能指针看起来并不比按值传递对象优越，但在处理多态对象时，其优点将显现出来。如下所示，使用智能指针可避免切除（slicing）问题：

```
// Example of Slicing When Passing Polymorphic Objects by Value
// Fish is a base class for Tuna and Carp, Fish::Swim() is virtual
void MakeFishSwim (Fish aFish)     // note parameter type
{
    aFish.Swim(); // virtual function
}

// ... Some function
Carp freshWaterFish;
MakeFishSwim (freshWaterFish);  // Carp will be 'sliced' to Fish
// Slicing: only the Fish part of Carp is sent to MakeFishSwim()

Tuna marineFish;
MakeFishSwim(marineFish); // Slicing again
```

如果程序员选择使用深复制智能指针，便可解决切除问题，如程序清单 26.2 所示。

程序清单 26.2 使用基于深复制的智能指针将多态对象作为基类对象进行传递

```
 0: template <typename T>
 1: class deepcopy_smart_pointer
 2: {
 3: private:
 4:     T* m_pObject;
 5: public:
 6:     //... other functions
 7:
 8:     // copy constructor of the deepcopy pointer
 9:     deepcopy_smart_pointer (const deepcopy_smart_pointer& source)
10:     {
11:         // Clone() is virtual: ensures deep copy of Derived class object
12:         m_pObject = source->Clone ();
13:     }
14:
15:    // copy assignment operator
16:     deepcopy_smart_pointer& operator= (const deepcopy_smart_pointer& source)
17:    {
18:       if (m_pObject)
19:          delete m_pObject;
20:
21:       m_pObject = source->Clone ();
22:    }
23:
24: };
```

▼ 分析：

可以看到，deepcopy_smart_pointer 在第 9～13 行实现了一个复制构造函数，使得能够通过函数 Clone()函数对多态对象进行深复制——对象必须实现函数 Clone()。另外，它还实现了复制赋值运算符，如第 16～22 行所示。为简单起见，这里假设基类 Fish 实现的虚函数为 Clone。通常，实现深复制模型的智能指针通过模板参数或函数对象提供该函数。

下面是 deepcopy_smart_pointer 的一种用法：

```
deepcopy_smart_ptr<Carp> freshWaterFish(new Carp);
MakeFishSwim (freshWaterFish);  // Carp will not be 'sliced'
```

构造函数实现的深复制将发挥作用，确保传递的对象不会出现切除问题——虽然从语法上说，目标函数 MakeFishSwim()只要求基类部分。

基于深复制的机制的不足之处在于性能。对有些应用程序来说，这可能不是问题，但对于其他很多应用程序来说，这可能导致程序员不使用智能指针，而将指向基类的指针（常规指针 Fish*）传递给函数，如 MakeFishSwim()。其他指针类型以各种方式试图解决这种性能问题。

26.3.2 写时复制机制

写时复制机制（Copy on Write，COW）试图对深复制智能指针的性能进行优化，它共享指针，直到首次写入对象。首次调用非 const 函数时，COW 指针通常为该非 const 函数操作的对象创建一个副本，而其他指针实例仍共享源对象。

COW 深受很多程序员的喜欢。实现 const 和非 const 版本的运算符*和->，是实现 COW 指针功能的关键。非 const 版本用于创建副本。

重要的是，选择 COW 指针时，在使用这样的实现前务必理解其实现细节。否则，复制时将出现

复制得太少或太多的情况。

26.3.3 引用计数智能指针

引用计数是一种记录对象的用户数量的机制。当计数降低到零后，便将对象释放。因此，引用计数提供了一种优良的机制，使得可共享对象而无法对其进行复制。如果读者使用过微软的 COM 技术，肯定知道引用计数的概念。

这种智能指针被复制时，需要将对象的引用计数加 1。至少有两种常用的方法来跟踪计数：

- 在对象中维护引用计数；
- 引用计数由共享对象中的指针类维护。

前者称为入侵式引用计数，因为需要修改对象以维护和递增引用计数，并将其提供给管理对象的智能指针。COM 采取的就是这种方法。后者是智能指针类将计数保存在自由存储区（如动态分配的整型），复制时复制构造函数将这个值加 1。

因此，使用引用计数机制，程序员只应通过智能指针来处理对象。在使用智能指针管理对象的同时让原始指针指向它是一种糟糕的做法，因为智能指针将在它维护的引用计数减为零时释放对象，而原始指针将继续指向已不属于当前应用程序的内存。引用计数还有一个独特的问题：如果两个对象分别存储指向对方的指针，这两个对象将永远不会被释放，因为它们的生命周期依赖性导致其引用计数最少为 1。

26.3.4 引用链接智能指针

引用链接智能指针不主动维护对象的引用计数，而只需知道计数什么时候变为零，以便能够释放对象。

之所以称为引用链接，是因为其实现是基于双向链表的。通过复制智能指针来创建新智能指针时，新指针将被插入到链表中。当智能指针离开作用域进而被销毁时，析构函数将把它从链表中删除。与引用计数的指针一样，引用链接指针也存在生命周期依赖性导致的问题。

26.3.5 破坏性复制

破坏性复制是这样一种机制，即在智能指针被复制时，将对象的所有权转交给目标指针并重置原来的指针。

```
destructive_copy_smartptr <SampleClass> pSmartPtr (new SampleClass ());

SomeFunc (pSmartPtr);     // Ownership transferred to SomeFunc
// Don't use pSmartPtr in the caller any more!
```

虽然破坏性复制机制使用起来并不直观，但它有一个优点，即可确保任何时刻只有一个活动指针指向对象。因此，它非常适合从函数返回指针以及需要利用其“破坏性”的情形。

程序清单 26.3 是一种破坏性复制指针的实现，它没有采用推荐的标准 C++编程方法。

警告

std::auto_ptr 是最流行（也可以说是最臭名昭著，取决于您如何看）的破坏性复制指针。被传递给函数或复制给另一个指针后，这种智能指针就没有用了。C++11 摒弃了 std::auto_ptr，您应使用 std::unque_ptr，这种指针不能按值传递，而只能按引用传递，因为其复制构造函数和复制赋值运算符都是私有的。

程序清单 26.3 一个破坏性复制智能指针

```
 0: template <typename T>
 1: class destructivecopy_pointer
 2: {
 3: private:
 4:    T* pObject;
 5: public:
 6:    destructivecopy_pointer(T* pInput):pObject(pInput) {}
 7:    ~destructivecopy_pointer() { delete pObject; }
 8:
 9:    // copy constructor
10:    destructivecopy_pointer(destructivecopy_pointer& source)
11:    {
12:       // Take ownership on copy
13:       pObject = source.pObject;
14:
15:       // destroy source
16:       source.pObject = 0;
17:    }
18:
19:    // copy assignment operator
20:    destructivecopy_pointer& operator= (destructivecopy_pointer& rhs)
21:    {
22:       if (pObject != source.pObject)
23:       {
24:          delete pObject;
25:          pObject = source.pObject;
26:          source.pObject = 0;
27:       }
28:    }
29: };
30:
31: int main()
32: {
33:    destructivecopy_pointer<int> pNumber (new int);
34:    destructivecopy_pointer<int> pCopy = pNumber;
35:
36:    // pNumber is now invalid
37:    return 0;
38: }
```

▼ 分析：

程序清单 26.3 演示了基于破坏性复制的智能指针实现的最重要部分。第 10～17 行和第 20～28 行分别是复制构造函数和赋值运算符。这些函数实际上使源指针在复制后失效，即复制构造函数在复制后将源指针设置为 NULL，这就是“破坏性复制”的由来。赋值运算符亦如此。因此在第 34 行被赋给另一个指针后，pNumber 就不再有效，这种行为不符合赋值操作的目的。

警告

对破坏性复制智能指针的实现来说，程序清单 26.3 所示的复制构造函数和复制赋值运算符至关重要，但也深受诟病。不同于大多数 C++类，该智能指针类的复制构造函数和赋值运算符不能接受 const 引用，因为它在复制源引用后使其无效。这不仅不符合传统复制构造函数和赋值运算符的语义，还让智能指针类的用法不直观。复制或赋值后销毁源引用不符合预期。鉴于这种智能指针销毁源引用，这也使得它不适合用于 STL 容器，如 std::vector 或其他任何动态集合类。这些容器需要在内部复制内容，这将导致指针失效。由于种种原因，很多程序员像躲避瘟疫一样避免使用破坏性复制智能指针。

提示

C++标准一直支持 auto_ptr，它是一种基于破坏性复制的智能指针。C++11 终于摒弃了该智能指针，现在您应使用 std::unique_ptr。

C++11

使用 std::unique_ptr

std::unique_ptr 是 C++11 新增的，与 auto_ptr 稍有不同，因为它不允许复制和赋值。

提示

要使用 std:unique_ptr，必须包含头文件<memory>：

```
#include <memory>
```

unique_ptr 是一种简单的智能指针，类似于程序清单 26.1 所示的智能指针，但其复制构造函数和赋值运算符被声明为私有的，因此不能复制它，即不能将其按值传递给函数，也不能将其赋给其他指针。程序清单 26.4 演示了其用法。

程序清单 26.4 使用 std::unique_ptr

```
#include <iostream>
#include <memory>  // include this to use std::unique_ptr
using namespace std;

class Fish
{
public:
    Fish() {cout << "Fish: Constructed!" << endl;}
    ~Fish() {cout << "Fish: Destructed!" << endl;}

    void Swim() const {cout << "Fish swims in water" << endl;}
};

void MakeFishSwim(const unique_ptr<Fish>& inFish)
{
   inFish->Swim();
}

int main()
{
   unique_ptr<Fish> smartFish (new Fish);

   smartFish->Swim();
   MakeFishSwim(smartFish); // OK, as MakeFishSwim accepts reference

   unique_ptr<Fish> copySmartFish;
   // copySmartFish = smartFish; // error: operator= is private

   return 0;
}
```

▼ 输出：

```
Fish: Constructed!
Fish swims in water
Fish swims in water
Fish: Destructed!
```

▼ 分析：

从输出可知，虽然 smartFish 指向的对象是在 main()中创建的，但它被自动销毁，您无需调用 delete

运算符。这是 unique_ptr 的行为：当指针离开作用域时，将通过析构函数释放它拥有的对象。注意到第 23 行将 smartFish 作为参数传递给了 MakeFishSwim()，这样做不会导致复制，因为 MakeFishSwim() 的参数为引用，如第 13 行所示。如果删除第 13 行的引用符号&，将出现编译错误，因为复制构造函数是私有的。同样，不能像第 26 行那样将一个 unique_ptr 对象赋给另一个 unique_ptr 对象，因为复制赋值运算符是私有的。

总之，unique_ptr 比 C++11 已摒弃的 auto_ptr 更安全，因为复制和赋值不会导致源智能指针对象无效。它在销毁时释放对象，可帮助您进行简单的内存管理。

26.4 深受欢迎的智能指针库

显然，C++标准库提供的智能指针并不能满足所有程序员的需求，这就是还有很多其他智能指针库的原因。

Boost（www.boost.org）提供了一些经过测试且文档完善的智能指针类，还有很多其他的实用类。有关 Boost 智能指针的更详细信息，请访问 http://www.boost.org/libs/smart_ptr/ smart_ptr.htm；在这里还可下载相关的库。

同样，在 Windows 平台上编写 COM 应用程序的程序员应使用 ATL 框架中的智能指针类（如 CComPtr 和 CComQIPtr）来管理 COM 对象，而不要使用原始指针。

26.5 总结

本章介绍了使用正确的智能指针有助于编写使用指针的代码，并有助于减少与内存分配和对象拥有权相关的问题。本章还介绍了各种智能指针类型，并指出在应用程序中使用智能指针类前务必要了解其行为。现在您知道，不应使用 std::auto_ptr，因为它在复制和赋值时导致源指针无效。您还学习了最新的智能指针类 std::unique_ptr，这是 C++11 新增的。

26.6 问与答

问：在需要指针 vector 时，是否应将 auto_ptr 作为 vector 存储的对象类型？

答：通常，不应使用 std::auto_prt，因为它已被摒弃。另外，复制或赋值操作将导致源对象不可用。

问：要成为智能指针类，需要实现哪两个运算符？

答：运算符*和->，这两个运算符使得可像使用常规指针那样使用类的对象。

问：假设有一个应用程序，其中的 Class1 和 Class2 类分别包含一个指向 Class2 对象和 Class1 对象的成员属性。在这种情况下，是否应使用引用计数指针？

答：不应该。由于生命周期依赖性，引用计数将不会减少到零，导致两个类的对象永久性地留在堆中。

问：智能指针有多少种？

答：成千上万，甚至数百万。程序员应只使用文档完善且来源可靠（如来自 Boost）的智能指针。

问：string 类在自由存储区中动态地管理字符数组，它也是智能指针吗？

答：不是。string 类通常没有实现运算符*和->，因此不属于智能指针。

26.7 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

26.7.1 测试

1. 为应用程序编写自己的智能指针前应查看什么地方？
2. 智能指针是否会严重降低应用程序的性能？
3. 引用计数智能指针在什么地方存储引用计数？
4. 引用链接指针使用的链表机制是单向链表还是双向链表？

26.7.2 练习

1. 查错：指出下述代码中的错误：

```
std::auto_ptr<SampleClass> pObject (new SampleClass ());
std::auto_ptr<SampleClass> pAnotherObject (pObject);
pObject->DoSomething ();
pAnotherObject->DoSomething();
```

2. 使用 **unique_ptr** 类实例化一个 **Carp** 对象，而 **Carp** 类继承了 **Fish** 类。将该对象作为 **Fish** 指针传递时是否会出现切除问题。

3. 查错：指出下述代码中的错误：

```
std::unique_ptr<Tuna> myTuna (new Tuna);
unique_ptr<Tuna> copyTuna;
copyTuna = myTuna;
```

第 27 章
使用流进行输入和输出

在本书第 1 章，您使用了 std::cout 在屏幕上显示 Hello World；实际上，从该章起您就一直在使用流。现在该给予 C++的这部分应有的关注，从实用的角度介绍流。

在本章中，您将学习：

- 什么是流，如何使用它们；
- 如何使用流来读写文件；
- 有用的 C++流操作。

27.1 流的概述

假设您要开发一个程序，它从磁盘读取数据、将数据显示到屏幕上、从键盘读取用户输入以及将数据存储到磁盘中。在这种情况下，倘若不管数据来自或前往什么设备或位置，都能以相同的方式处理读写操作，那该有多好！这正是 C++流提供的功能。

C++流是读写（输入和输出）逻辑的通用实现，让您能够统一的模式读写数据。不管是磁盘或键盘读取数据，还是将输入写入显示器或磁盘，这些模式都相同。您只需使用合适的流类，类的实现将负责处理与设备和操作系统相关的细节。

再来看一下您编写的第一个程序（程序清单 1.1）中相关的代码行：

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

std:cout 是 ostream 类的一个对象，用于输出到控制台。要使用 std::cout，需要包含提供它的头文件<iostream>，这个头文件还提供了 std::cin，让您能够从流中读取数据。

那么，我说流让您能够以一致的方式访问不同的设备时，是什么意思呢？如果要将 Hello World 写入文本文件，可将同样的语法用于文件流对象 fsHW：

```
fsHW << "Hello World!" << endl;  // "Hello World!" into a file stream
```

正如您看到的，选择正确的流类后，将 Hello World 写入文件与将其显示到屏幕上并没有太大的不同。

提示

> 用于写入流时，运算符<<被称为流插入运算符，可将其用于写入屏幕、文件等。用于将流中的数据写入变量时，运算符>>被称为流提取运算符，可将其用于从键盘、文件等读取输入。

接下来，本章将从实用的角度探讨流。

27.2 重要的 C++流类和流对象

C++提供了一组标准类和头文件，可帮助您执行重要而常见的输入/输出操作。表 27.1 列出了您将

经常使用的类。

表 27.1 std 命名空间中常用的 C++流类

类/对象	用 途
cout	标准输出流，通常被重定向到控制台
cin	标准输入流，通常用于将数据读入变量
cerr	用于显示错误信息的标准输出流
fstream	用于操作文件的输入和输出流，继承了 ofstream 和 ifstream
ofstream	用于操作文件的输出流类，即用于创建文件
ifstream	用于操作文件的输入流类，即用于读取文件
stringstream	用于操作字符串的输入和输出流类，继承了 istringstream 和 ostringstream，通常用于在字符串和其他类型之间进行转换

注意

cout、cin 和 cerr 分别是流类 ostream、istream 和 ostream 的全局对象。由于是全局对象，它们在 main()开始之前就已初始化。

使用流类时，可指定为您执行特定操作的控制符（manipulator）。std::endl 就是一个这样的控制符，您一直在使用它来插入换行符：

```
std::cout  << "This lines ends here" << std::endl;
```

表 27.2 列出了其他几个控制符和标志。

表 27.2 std 命名空间中常用于流的控制符

类/对象	用 途
输出控制符	
endl	插入一个换行符
ends	插入一个空字符
基数控制符	
dec	让流以十进制方式解释输入或显示输出
hex	让流以十六进制方式解释输入或显示输出
oct	让流以八进制方式解释输入或显示输出
浮点数表示操作符	
fixed	让流以定点表示法显示数据
scientific	让流以科学表示法显示数据
<iomanip>控制符	
setprecision	设置小数精度
setw	设置字段宽度
setfill	设置填充字符
setbase	设置基数，与使用 dec、hex 或 oct 等效
setiosflag	通过类型为 std::ios_base::fmtflags 的掩码输入参数设置标志
resetiosflag	将 std::ios_base::fmtflags 参数指定的标志重置为默认值

27.3 使用 std::cout 将指定格式的数据写入控制台

std::cout 用于写入到标准输出流，可能是本书前面使用得最多的流。下面更详细地介绍它，并使用

一些控制符来改变数据的对齐和显示方式。

27.3.1 使用 std::cout 修改数字的显示格式

可以让 cout 以十六进制或八进制方式显示整数。程序清单 27.1 演示了如何使用 cout 以各种格式显示输入的数字。

程序清单 27.1 使用 cout 和<iomanip>控制符以十进制、十六进制和八进制格式显示整数

```
 0: #include <iostream>
 1: #include <iomanip>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Enter an integer: ";
 7:    int Input = 0;
 8:    cin >> Input;
 9:
10:    cout << "Integer in octal: " << oct << Input << endl;
11:    cout << "Integer in hexadecimal: " << hex << Input << endl;
12:
13:    cout << "Integer in hex using base notation: ";
14:    cout<<setiosflags(ios_base::hex|ios_base::showbase|ios_base::uppercase);
15:    cout << Input << endl;
16:
17:    cout << "Integer after resetting I/O flags: ";
18:    cout<<resetiosflags(ios_base::hex|ios_base::showbase|ios_base::uppercase);
19:    cout << Input << endl;
20:
21:    return 0;
22: }
```

▼ 输出：

```
Enter an integer: 253
Integer in octal: 375
Integer in hexadecimal: fd
Integer in hex using base notation: 0XFD
Integer after resetting I/O flags: 253
```

▼ 分析：

这个代码示例使用了表 27.2 所示的控制符，以修改 cout 显示用户输入的整数 Input 的方式。注意到第 10 和 11 行使用了控制符 oct 和 hex。第 14 行使用了 setiosflags()让 cout 以十六进制方式（并使用大写字母）显示该数字，其结果是 cout 将 253 显示为 OXFD。第 18 行使用了 resetiosflags()，其效果是再次使用 cout 显示该整数时，将显示为十进制。要将显示整数时使用的基数改为十进制，也可使用下面这种方式：

```
cout << dec << Input << endl;  // displays in decimal
```

对于诸如 Pi 等数字，可指定 cout 显示它们时使用的精度（小数点后面的位数），还可指定以定点表示法或科学表示法显示它们。程序清单 27.2 演示了如何设置这些格式。

程序清单 27.2 使用 cout 以定点表示法和科学表示法显示 Pi 和圆面积

```
 0: #include <iostream>
 1: #include <iomanip>
 2: using namespace std;
```

```
3:
4: int main()
5: {
6:     const double Pi = (double)22.0 / 7;
7:     cout << "Pi = " << Pi << endl;
8:
9:     cout << endl << "Setting precision to 7: " << endl;
10:    cout << setprecision(7);
11:    cout << "Pi = " << Pi << endl;
12:    cout << fixed << "Fixed Pi = " << Pi << endl;
13:    cout << scientific << "Scientific Pi = " << Pi << endl;
14:
15:    cout << endl << "Setting precision to 10: " << endl;
16:    cout << setprecision(10);
17:    cout << "Pi = " << Pi << endl;
18:    cout << fixed << "Fixed Pi = " << Pi << endl;
19:    cout << scientific << "Scientific Pi = " << Pi << endl;
20:
21:    cout << endl << "Enter a radius: ";
22:    double Radius = 0.0;
23:    cin >> Radius;
24:    cout << "Area of circle: " << 2*Pi*Radius*Radius << endl;
25:
26:    return 0;
27: }
```

▼ 输出：

```
Pi = 3.14286

Setting precision to 7:
Pi = 3.142857
Fixed Pi = 3.1428571
Scientific Pi = 3.1428571e+000

Setting precision to 10:
Pi = 3.1428571429e+000
Fixed Pi = 3.1428571429
Scientific Pi = 3.1428571429e+000

Enter a radius: 9.99
Area of circle: 6.2731491429e+002
```

▼ 分析：

输出表明，第 7 行和第 10 行分别将精度设置为 7 和 10 后，显示的 Pi 值不同。另外，控制符 scientific 导致计算得到的圆面积被显示为 6.2731491429e + 002。

27.3.2 使用 std::cout 对齐文本和设置字段宽度

可使用 setw()控制符来设置字段宽度，插入到流中的内容将在指定宽度内右对齐。在这种情况下，还可使用 setfill()指定使用什么字符来填充空白区域，如程序清单 27.3 所示。

程序清单 27.3 使用控制符 setw()设置字段宽度，并使用 setfill()指定填充字符

```
0: #include <iostream>
1: #include <iomanip>
2: using namespace std;
3:
4: int main()
```

```
 5: {
 6:    cout << "Hey - default!" << endl;
 7:
 8:    cout << setw(35);  // set field width to 25 columns
 9:    cout << "Hey - right aligned!" << endl;
10:
11:    cout << setw(35) << setfill('*');
12:    cout << "Hey - right aligned!" << endl;
13:
14:    cout << "Hey - back to default!" << endl;
15:
16:    return 0;
17: }
```

▼ 输出：

```
Hey - default!
                Hey - right aligned!
***************Hey - right aligned!
Hey - back to default!
```

▼ 分析：

第 8 行使用了 setw(35)，而第 11 行使用了 setw(35)和 setfill('*')，输出说明了这样做的效果。从输出可知，第 11 行导致使用 setfill()指定的星号来填充文本前的空白区域。

27.4 使用 std::cin 进行输入

std::cin 多才多艺，让您能够将输入读取到基本类型（如 int、double 以及 C 风格字符串 char*）变量中。您还可使用 getline()从键盘读取一行输入。

27.4.1 使用 std::cin 将输入读取到基本类型变量中

使用 cin 可将标准输入读取到 int、double 和 char 变量中，程序清单 27.4 演示了如何读取用户输入的简单数据类型。

程序清单 27.4 使用 cin 将输入读取到 int 变量中，将使用科学表示法的浮点数读取到 double 变量中，将三个字符分别读取到 char 变量中

```
 0: #include<iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Enter an integer: ";
 6:    int InputInt = 0;
 7:    cin >> InputInt;
 8:
 9:    cout << "Enter the value of Pi: ";
10:    double Pi = 0.0;
11:    cin >> Pi;
12:
13:    cout << "Enter three characters separated by space: " << endl;
14:    char Char1 = '\0', Char2 = '\0', Char3 = '\0';
15:    cin >> Char1 >> Char2 >> Char3;
16:
```

```
17:    cout << "The recorded variable values are: " << endl;
18:    cout << "InputInt: " << InputInt << endl;
19:    cout << "Pi: " << Pi << endl;
20:    cout << "The three characters: " << Char1 << Char2 << Char3 << endl;
21:
22:    return 0;
23: }
```

▼ 输出：

```
Enter an integer: 32
Enter the value of Pi: 0.314159265e1
Enter three characters separated by space:
c + +
The recorded variable values are:
InputInt: 32
Pi: 3.14159
The three characters: c++
```

▼ 分析：

在程序清单27.4中，最有趣的部分是，您使用指数表示法输入Pi的值时，cin也将其读取到了double变量Pi中。注意到可以使用一行代码将输入读取到三个字符变量中，如第15行所示。

27.4.2 使用std::cin:get将输入读取到char数组中

cin让您能够将输入直接写入int变量，也可将输入直接写入char数组（C风格字符串）：

```
cout << "Enter a line: " << endl;
char CStyleStr [10] = {0};  // can contain max 10 chars
cin >> CStyleStr;  // Danger: user may enter more than 10 chars
```

写入C风格字符串缓冲区时，务必不要超越缓冲区的边界，以免导致程序崩溃或带来安全隐患，这至关重要。因此，将输入读取到char数组（C风格字符串）时，下面是一种更好的方法：

```
cout << "Enter a line: " << endl;
char CStyleStr[10] = {0};
cin.get(CStyleStr, 9);  // stop inserting at the 9th character
```

这种将文本插入到char数组（C风格字符串）的方式更安全，程序清单27.5演示了这一点。

程序清单27.5 插入到C风格字符串中时不超越其边界

```
 0: #include<iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Enter a line: " << endl;
 7:    char CStyleStr[10] = {0};
 8:    cin.get(CStyleStr, 9);
 9:    cout << "CStyleStr: " << CStyleStr << endl;
10:
11:    return 0;
12: }
```

▼ 输出：

```
Enter a line:
Testing if I can cross the bounds of the buffer
CStyleStr: Testing i
```

▼ 分析：

从输出可知，只将用户输入的前 9 个字符读取到了 C 风格字符串中，这是因为第 8 行使用的是 cin:get。处理 C 风格字符串时，这是最安全的方式。

提示 尽可能不要使用 C 风格字符串和 char 数组；只要可能，就应使用 std::string 而不是 char*。

27.4.3 使用 std::cin 将输入读取到 std::string 中

cin 多才多艺，甚至可使用它将用户输入的字符串直接读取到 std::string 中：

```
std::string Input;
cin >> Input;  // stops insertion at the first space
```

程序清单 27.6 演示了如何使用 cin 将输入读取到 std::string。

程序清单 27.6 使用 cin 将文本插入到 std::string 中

```
 0: #include<iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Enter your name: ";
 7:    string Name;
 8:    cin >> Name;
 9:    cout << "Hi " << Name << endl;
10:
11:    return 0;
12: }
```

▼ 输出：

```
Enter your name: Siddhartha Rao
Hi Siddhartha
```

▼ 分析：

输出表明，并未按程序设计的那样显示整个姓名。我在第 8 行使用了 cin，希望 Name 存储整个姓名，而不仅仅是名字。为什么会这样呢？显然是由于 cin 遇到空白后停止插入。

要读取整行输入（包括空白），需要使用 getline()：

```
string Name;
getline(cin, Name);
```

程序清单 27.7 演示了如何结合使用 getline()和 cin。

程序清单 27.7 使用 getline()和 cin 读取整行用户输入

```
 0: #include<iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Enter your name: ";
 7:    string Name;
 8:    getline(cin, Name);
 9:    cout << "Hi " << Name << endl;
10:
11:    return 0;
12: }
```

▼ 输出：

```
Enter your name: Siddhartha Rao
Hi Siddhartha Rao
```

▼ 分析：

第8行的getline()确保不跳过空白字符，现在输出包含整行用户输入。

27.5 使用 std::fstream 处理文件

C++提供了 std::fstream，旨在以独立于平台的方式访问文件。std::fstream 从 std::ofstream 那里继承了写入文件的功能，并从 std::ifstream 那里继承了读取文件的功能。

换句话说，std::fstream 提供了读写文件的功能。

提示

要使用 std::fstream 类或其基类，需要包含头文件<fstream>：

```
#include <fstream>
```

27.5.1 使用 open()和 close()打开和关闭文件

要使用 fstream、ofstream 或 ifstream 类，需要使用方法 open()打开文件：

```
fstream myFile;
myFile.open("HelloFile.txt",ios_base::in¦ios_base::out¦ios_base::trunc);

if (myFile.is_open())
{
   // do reading or writing here

   myFile.close();
}
```

open()接受两个参数：第一个是要打开的文件的路径和名称（如果没有提供路径，将假定为应用程序的当前目录设置），第二个是文件的打开模式。在上述代码中，指定了模式 ios_base::trunc（即便指定的文件存在，也重新创建它）、ios_base::in（可读取文件）和 ios_base::out（可写入文件）。

注意到在上述代码中使用了 is_open()，它检测 open()是否成功。

警告

别忘了关闭文件流以保存其内容，这至关重要。

还有另一种打开文件流的方式，那就是使用构造函数：

```
fstream myFile("HelloFile.txt",ios_base::in¦ios_base::out¦ios_base::trunc);
```

如果只想打开文件进行写入，可使用如下代码：

```
ofstream myFile("HelloFile.txt", ios_base::out);
```

如果只想打开文件进行读取，可使用如下代码：

```
ifstream myFile("HelloFile.txt", ios_base::in);
```

提示

无论是使用构造函数还是成员方法 open()来打开文件流，都建议您在使用文件流对象前，使用 open()检查文件打开操作是否成功。

可在下述各种模式下打开文件流。

- _ios_base::app：附加到现有文件末尾，而不是覆盖它。
- ios_base::ate：切换到文件末尾，但可在文件的任何地方写入数据。
- ios_base::trunc：导致现有文件被覆盖，这是默认设置。

- ios_base::binary：创建二进制文件（默认为文本文件）。
- ios_base::in：以只读方式打开文件。
- ios_base::out：以只写方式打开文件。

27.5.2 使用 open()创建文本文件并使用运算符<<写入文本

有打开的文件流后，便可使用运算符<<向其中写入文本，如程序清单 27.8 所示。

程序清单 27.8 使用 ofstream 新建一个文本文件并向其中写入文本

```
 0: #include<fstream>
 1: #include<iostream>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    ofstream myFile;
 7:    myFile.open("HelloFile.txt", ios_base::out);
 8:
 9:    if (myFile.is_open())
10:    {
11:       cout << "File open successful" << endl;
12:
13:       myFile << "My first text file!" << endl;
14:       myFile << "Hello file!";
15:
16:       cout << "Finished writing to file, will close now" << endl;
17:       myFile.close();
18:    }
19:
20:    return 0;
21: }
```

▼ 输出：

```
File open successful
Finished writing to file, will close now
```

Content of file HelloFile.txt:

```
My first text file!
Hello file!
```

▼ 分析：

第 7 行以 ios_base::out 模式（即只写模式）打开文件。第 9 行检查 open()是否成功，然后使用插入运算符<<写入该文件流，如第 13 和 14 行所示。最后，第 17 行关闭文件流。

注意

程序清单 27.8 表明，写入文件的方式与使用 cout 写入到标准输出（控制台）的方式相同。这表明，C++流让您能够以类似的方式处理不同的设备：使用 cout 将文本显示到屏幕的方式与使用 ofstream 写入文件的方式相同。

27.5.3 使用 open()和运算符>>读取文本文件

要读取文件，可使用 fstream 或 ifstream，并使用标志 ios_base::in 打开它。程序清单 27.9 演示了如何读取程序清单 27.8 创建的文件 HelloFile.txt。

程序清单 27.9 从程序清单 27.8 创建的文件 HelloFile.txt 中读取文本

```
 0: #include<fstream>
 1: #include<iostream>
 2: #include<string>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    ifstream myFile;
 8:    myFile.open("HelloFile.txt", ios_base::in);
 9:
10:    if (myFile.is_open())
11:    {
12:       cout << "File open successful. It contains: " << endl;
13:       string fileContents;
14:
15:       while (myFile.good())
16:       {
17:          getline (myFile, fileContents);
18:         cout << fileContents << endl;
19:       }
20:
21:       cout << "Finished reading file, will close now" << endl;
22:       myFile.close();
23:    }
24:    else
25:       cout << "open() failed: check if file is in right folder" << endl;
26:
27:    return 0;
28: }
```

▼ 输出：

```
File open successful. It contains:
My first text file!
Hello file!
Finished reading file, will close now
```

注意 鉴于程序清单 27.9 读取程序清单 27.9 创建的文本文件 HelloFile.txt，因此您需要将该文件移到该项目的工作目录，或者将该程序清单合并到前一个程序清单中。

▼ 分析：

与往常一样，您使用 is_open()检查第 8 行调用 open()是否成功。请注意，这里没有使用提取运算符>>将文件内容直接读取到第 18 行使用 cout 显示的 string，而是使用 getline()从文件流中读取输入，这与程序清单 27.7 使用它来读取用户输入的方式完全相同：每次读取一行。

27.5.4 读写二进制文件

写入二进制文件的流程与前面介绍的流程差别不大，重要的是在打开文件时使用 ios_base::binary 标志。通常使用 ofstream::write 和 ifstream::read 来读写二进制文件，如程序清单 27.10 所示。

程序清单 27.10 将一个结构写入二进制文件并使用该文件的内容创建一个结构

```
 0: #include<fstream>
 1: #include<iomanip>
 2: #include<string>
```

```
 3: #include<iostream>
 4: using namespace std;
 5:
 6: struct Human
 7: {
 8:     Human() {};
 9:    Human(const char* inName, int inAge, const char* inDOB) : Age(inAge)
10:    {
11:       strcpy(Name, inName);
12:       strcpy(DOB, inDOB);
13:    }
14:
15:    char Name[30];
16:    int Age;
17:    char DOB[20];
18: };
19:
20: int main()
21: {
22:    Human Input("Siddhartha Rao", 101, "May 1910");
23:
24:    ofstream fsOut ("MyBinary.bin", ios_base::out ¦ ios_base::binary);
25:
26:    if (fsOut.is_open())
27:    {
28:      cout << "Writing one object of Human to a binary file" << endl;
29:       fsOut.write(reinterpret_cast<const char*>(&Input), sizeof(Input));
30:       fsOut.close();
31:    }
32:
33:    ifstream fsIn ("MyBinary.bin", ios_base::in ¦ ios_base::binary);
34:
35:    if(fsIn.is_open())
36:    {
37:       Human somePerson;
38:       fsIn.read((char*)&somePerson, sizeof(somePerson));
39:
40:       cout << "Reading information from binary file: " << endl;
41:       cout << "Name = " << somePerson.Name << endl;
42:       cout << "Age = " << somePerson.Age << endl;
43:       cout << "Date of Birth = " << somePerson.DOB << endl;
44:    }
45:
46:    return 0;
47: }
```

▼ 输出：

```
Writing one object of Human to a binary file
Reading information from binary file:
Name = Siddhartha Rao
Age = 101
Date of Birth = May 1910
```

▼ 分析：

第 22～31 行创建了结构 Human 的一个实例（该结构包含属性 Name、Age 和 BOD），并使用 ofstream 将其持久化到磁盘中的二进制文件 MyBinary.bin 中。接下来，第 33～34 使用另一个类型为 ifstream 的流对象读取这些信息。输出的 Name 等属性是从二进制文件中读取的。该示例还演示了如何使用 ifstream::read 和 ofstream::write 来读写文件。注意到第 29 行使用了 reinterpret_cast，它让编译器将

结构解释为 char*。第 38 行使用 C 风格类型转换方式，这与第 29 行的类型转换方式等价。

注意

如果不是处于解释的目的，我将把结构 Human 及其属性持久化到一个 XML 文件中。XML 是一种基于文本和标记的存储格式，在持久化信息方面提供了灵活性和可扩展性。发布这个程序后，如果您对其进行升级，给结构 Human 添加了新属性（如 NumChildren），则需要考虑新版本使用的 ifstream::read，确保它能够正确地读取旧版本创建的二进制数据。

27.6 使用 std::stringstream 对字符串进行转换

假设您有一个字符串，它包含字符串值 45，如何将其转换为整型值 45 呢？如何将整型值 45 转换为字符串 45 呢？C++提供的 stringstream 类是最有用的工具之一，让您能够执行众多的转换操作。

提示

要使用 std::stringstream 类，需要包含头文件<sstream>：

```
#include <sstream>
```

程序清单 27.11 演示了一些简单的 stringstream 操作。

程序清单 27.11 使用 std::stringstream 在整型和字符串之间进行转换

```
 0: #include<fstream>
 1: #include<sstream>
 2: #include<iostream>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    cout << "Enter an integer: ";
 8:    int Input = 0;
 9:    cin >> Input;
10:
11:    stringstream converterStream;
12:    converterStream << Input;
13:    string strInput;
14:    converterStream >> strInput;
15:
16:    cout << "Integer Input = " << Input << endl;
17:    cout << "String gained from integer, strInput = " << strInput << endl;
18:
19:    stringstream anotherStream;
20:    anotherStream << strInput;
21:    int Copy = 0;
22:    anotherStream >> Copy;
23:
24:    cout << "Integer gained from string, Copy = " << Copy << endl;
25:
26:    return 0;
27: }
```

▼ 输出：

```
Enter an integer: 45
Integer Input = 45
String gained from integer, strInput = 45
Integer gained from string, Copy = 45
```

▼ 分析：

该程序让用户输入一个整型值，并使用运算符<<将其插入到一个 stringstream 对象中，如第 12 行

所示；然后，您使用提取运算符将这个整数转换为 string，如第 14 行所示。接下来，您将存储在 strInput 中的字符串转换为整数，并将其存储到 Copy 中。

应该	不应该
只想读取文件时，务必使用 ifstream。 只想写入文件时，务必使用 ofstream。 插入文件流或从文件流中提取之前，务必使用 is_open()核实是否成功地打开了它。	使用完文件流后，别忘了使用方法 close()将其关闭。 别忘了，使用代码 cin>>strData;从 cin 提取内容到 string 中时，通常导致 strData 只包含空白前的文本，而不是整行。 别忘了，函数 getline(cin, strData);从输入流中获取整行，其中包括空白。

27.7 总结

本章从实用的角度介绍了流。您了解到，从本书开头起，您就一直在使用输入/输出流，如 cout 和 cin。现在，您知道了如何创建简单的文本文件以及如何读写这种文件。您了解到，stringstream 可帮助您在简单类型（如整型）和字符串之间进行转换。

27.8 问与答

问：我发现，可使用 fstream 来读取和写入文件，那么什么情况下该使用 ofstream 和 ifstream 呢？

答：如果您的代码或模块只需读取文件，就应使用 ifstream；同样，如果您的代码或模块只需写入文件，就应使用 ofstream。在这两种情形下，都可使用 fstream，但为确保数据和代码的完整性，最好像使用 const 那样采取更严厉的策略，不过并非必须这样做。

问：什么情况下应使用 cin.get()？什么情况下应使用 cin.getline()？

答：cin.getline()确保您捕获用户输入的整行，包括空白在内；cin.get()帮助您以每次一个字符的方式捕获用户输入。

问：什么情况下应使用 stringstream？

答：stringstream 提供了一种方便的途径，让您能够在整型及其他简单类型和字符串之间进行转换，程序清单 27.11 演示了这一点。

27.9 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

27.9.1 测验

1. 在只需写入文件的情况下，应使用哪种流？
2. 如何使用 cin 从输入流中获取一整行？

3. 在需要将 std::string 对象写入文件时，应使用 ios_base::binary 模式吗？
4. 使用 open()打开流后，为何还要使用 is_open()进行检查？

27.9.2 练习

1. 查错：找出下述代码中的错误：

```
fstream myFile;
myFile.open("HelloFile.txt", ios_base::out);
myFile << "Hello file!";
myFile.close();
```

2. 查错：找出下述代码中的错误：

```
ifstream MyFile("SomeFile.txt");
if(MyFile.is_open())
{
   MyFile << "This is some text" << endl;
   MyFile.close();
}
```

第 28 章
异常处理

本章标题说明了一切：处理打断程序流程的特殊情形。本书前面一直采取最乐观的态度，假定内存分配将成功、文件能找到等，但现实往往并非如此。

在本章中，您将学习：

- 什么是异常；
- 如何处理异常；
- 异常处理对提供稳定的 C++应用程序有何帮助。

28.1 什么是异常

假设您的程序分配内存、读写数据、保存到文件，一切都在开发环境中完美地执行；您的应用程序使用了数 GB 内存，却没有泄露一字节，对此您很是自豪！您发布该应用程序，用户将其部署到各种工作站。有些工作站已购买 10 年，还有些工作站的硬盘都不转了。不久后您就收到了抱怨邮件，有些用户抱怨说“访问违规”，有些说出现“未处理的异常”。

“未处理”和“异常”，就是这样。显然，程序在开发环境中表现不错，为何麻烦不断呢？

现实世界千差万别，没有两台计算机是相同的，即便硬件配置一样。这是因为在特定时间，可用的资源量取决于计算机运行的软件及其状态，因此即便在开发环境中内存分配完美无缺，在其他环境中也可能出问题。

这样的情形很少见，但确实会发生。这些问题导致“异常”。

异常会打断应用程序的正常流程。毕竟，如果没有内存可用，应用程序就无法完成分配给它的任务。然而，应用程序可处理这种异常：向用户显示一条友好的错误消息、采取必要的挽救措施并妥善地退出。

通过对异常进行处理，有助于避免出现“访问违规”和“未处理的异常”等屏幕，还可避免收到相关的抱怨邮件。下面来看看 C++都向您提供了哪些应对意外的工具。

28.2 导致异常的原因

异常可能是外部因素导致的，如系统没有足够的内存；也可能是应用程序内部因素导致的，如使用的指针包含无效值或除数为零。为向调用者指出错误，有些模块引发异常。

注意 为防止代码引发异常，可对异常进行处理，让它们“不会出现异常”。

28.3 使用 try 和 catch 捕获异常

在捕获异常方面，try 和 catch 是最重要的 C++关键字。要捕获语句可能引发的异常，可将它们放在 try 块中，并使用 catch 块对 try 块可能引发的异常进行处理：

```
void SomeFunc()
{
   try
   {
      int* pNumber = new int;
      *pNumber = 999;
      delete pNumber;
   }
   catch(...)  // ... catches all exceptions
   {
      cout << "Exception in SomeFunc(), quitting" << endl;
   }
}
```

28.3.1 使用 catch(...)处理所有异常

第 8 章说过，成功分配内存时，默认形式的 new 返回一个指向该内存单元的有效指针，但失败时引发异常。程序清单 28.1 演示了如何捕获使用 new 分配内存时可能引发的异常，并在计算机不能分配请求的内存时进行处理。

程序清单 28.1 使用 try 和 catch 捕获并处理内存分配异常

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Enter number of integers you wish to reserve: ";
 6:    try
 7:    {
 8:       int Input = 0;
 9:       cin >> Input;
10:
11:       // Request memory space and then return it
12:       int* pReservedInts = new int [Input];
13:       delete[] pReservedInts;
14:    }
15:    catch (...)
16:    {
17:       cout << "Exception encountered. Got to end, sorry!" << endl;
18:    }
19:    return 0;
20: }
```

▼ 输出：

```
Enter number of integers you wish to reserve: -1
Exception encountered. Got to end, sorry!
```

▼ 分析：

在这个示例中，我请求为-1 个整数预留内存。这很荒谬，但用户经常做荒谬的事。如果没有异常处理程序，该程序将以讨厌的方式终止。但由于有异常处理程序，程序显示了一条得体的消息：Got to

end, sorry!。

注意

如果在 Visual Studio 中运行该程序，可能出现一条调试模式消息，如图 28.1 所示。

图 28.1 请求分配的内存量无效导致的异常

单击“忽略”将执行异常处理程序。这是一条调试模式消息，但即便是在发行模式下，异常处理也将让程序妥善地退出。

程序清单 28.1 演示了 try 块和 catch 块的用法。catch()像函数一样接受参数，参数...意味着 catch 块将捕获所有的异常。然而，在这个示例中，您可能想指定特定的异常类型 std::bad_alloc，因为这是 new 失败时引发的异常。通过捕获特定类型的异常，有助于处理这种类型的异常，如显示一条消息，准确地指出出了什么问题。

28.3.2 捕获特定类型的异常

程序清单 28.1 所示的异常是由 C++标准库引发的。这种异常的类型是已知的，在这种情况下，更好的选择是只捕获这种类型的异常，因为您能查明导致异常的原因，执行更有针对性的清理工作，或至少是向用户显示一条准确的消息，如程序清单 28.2 所示。

程序清单 28.2 捕获 std::bad_alloc 类型的异常

```
 0: #include <iostream>
 1: #include<exception>  // include this to catch exception bad_alloc
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Enter number of integers you wish to reserve: ";
 7:    try
 8:    {
 9:       int Input = 0;
10:       cin >> Input;
11:
12:       // Request memory space and then return it
13:       int* pReservedInts = new int [Input];
14:       delete[] pReservedInts;
15:    }
16:    catch (std::bad_alloc& exp)
17:    {
```

```
18:       cout << "Exception encountered: " << exp.what() << endl;
19:       cout << "Got to end, sorry!" << endl;
20:    }
21:    catch(...)
22:    {
23:       cout << "Exception encountered. Got to end, sorry!" << endl;
24:    }
25:    return 0;
26: }
```

▼ 输出：

```
Enter number of integers you wish to reserve: -1
Exception encountered: bad allocation
Got to end, sorry!
```

▼ 分析：

如果将程序清单 28.2 的输出与程序清单 28.1 的输出进行比较，您将发现现在能够提供应用程序中断的准确原因，即 bad allocation。这是因为新增了一个 catch（是的，有两个 catch 块），其中一个捕获类型为 bad_alloc&的异常，如第 16～20 行所示，这种异常是由 new 引发的。

提示

一般而言，可根据可能出现的异常添加多个 catch()块，这将很有帮助。
如程序清单 28.2 所示，catch(...)捕获未被其他 catch 块显式捕获的所有异常。

28.3.3 使用 throw 引发特定类型的异常

程序清单 28.2 捕获 std::bad_alloc 时，实际上是捕获 new 引发的 std::bad_alloc 类对象。您可以引发自己选择的异常，为此只需使用关键字 throw：

```
void DoSomething()
{
   if(something_unwanted)
      throw Value;
}
```

程序清单 28.3 将两个数相除，演示了如何使用 throw 引发自定义异常。

程序清单 28.3 在试图除以零时引发一种自定义异常

```
 0: #include<iostream>
 1: using namespace std;
 2:
 3: double Divide(double Dividend, double Divisor)
 4: {
 5:    if(Divisor == 0)
 6:       throw "Dividing by 0 is a crime";
 7:
 8:    return (Dividend / Divisor);
 9: }
10:
11: int main()
12: {
13:    cout << "Enter dividend: ";
14:    double Dividend = 0;
15:    cin >> Dividend;
16:    cout << "Enter divisor: ";
17:    double Divisor = 0;
18:    cin >> Divisor;
19:
```

```
20:    try
21:    {
22:       cout << "Result of division is: " << Divide(Dividend, Divisor);
23:    }
24:    catch(char* exp)
25:    {
26:       cout << "Exception: " << exp << endl;
27:       cout << "Sorry, can't continue!" << endl;
28:    }
29:
30:    return 0;
31: }
```

▼ **输出：**

```
Enter dividend: 2011
Enter divisor: 0
Exception: Dividing by 0 is a crime
Sorry, can't continue!
```

▼ **分析：**

上述代码表明，通过捕获类型为 char*的异常（第 24 行），可捕获调用函数 Divide()可能引发的异常（第 6 行）。另外，这里没有将整个 main()都放在 try{ };中，而只在其中包含可能引发异常的代码。这通常是一种不错的做法，因为异常处理也可能降低代码的执行性能。

28.4 异常处理的工作原理

在程序清单 28.3 中，您在函数 Divide()中引发了一个类型为 char*的异常，并在函数 main()中使用处理程序 catch(char*)捕获它。

每当您使用 throw 引发异常时，编译器都将查找能够处理该异常的 catch(Type)。异常处理逻辑首先检查引发异常的代码是否包含在 try 块中，如果是，则查找可处理这种异常的 catch(Type)。如果 throw 语句不在 try 块内，或者没有与引发的异常兼容的 catch()，异常处理逻辑将继续在调用函数中寻找。因此，异常处理逻辑沿调用栈向上逐个地在调用函数中寻找，直到找到可处理异常的 catch(Type)。在退栈过程的每一步中，都将销毁当前函数的局部变量，因此这些局部变量的销毁顺序与创建顺序相反。程序清单 28.4 演示了这一点。

程序清单 28.4 出现异常时销毁局部对象的顺序

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: struct StructA
 4: {
 5:    StructA() {cout << "Constructed a struct A" << endl; }
 6:    ~StructA() {cout << "Destroyed a struct A" << endl;  }
 7: };
 8:
 9: struct StructB
10: {
11:    StructB() {cout << "Constructed a struct B" << endl; }
12:    ~StructB() {cout << "Destroyed a struct B" << endl; }
13: };
14:
15: void FuncB()  // throws
16: {
17:       cout << "In Func B" << endl;
18:       StructA objA;
```

```
19:       StructB objB;
20:       cout << "About to throw up!" << endl;
21:       throw "Throwing for the heck of it";
22: }
23:
24: void FuncA()
25: {
26:    try
27:    {
28:       cout << "In Func A" << endl;
29:       StructA objA;
30:       StructB objB;
31:       FuncB();
32:      cout << "FuncA: returning to caller" << endl;
33:    }
34:    catch(const char* exp)
35:    {
36:       cout << "FuncA: Caught exception, it says: " << exp << endl;
37:       cout << "FuncA: Handled it here, will not throw to caller" << endl;
38:       // throw;  // uncomment this line to throw to main()
39:    }
40: }
41:
42: int main()
43: {
44:    cout << "main(): Started execution" << endl;
45:    try
46:    {
47:       FuncA();
48:    }
49:    catch(const char* exp)
50:    {
51:       cout << "Exception: " << exp << endl;
52:    }
53:    cout << "main(): exiting gracefully" << endl;
54:    return 0;
55: }
```

▼ 输出：

```
main(): Started execution
In Func A
Constructed a struct A
Constructed a struct B
In Func B
Constructed a struct A
Constructed a struct B
About to throw up!
Destroyed a struct B
Destroyed a struct A
Destroyed a struct B
Destroyed a struct A
FuncA: Caught exception, it says: Throwing for the heck of it
FuncA: Handled it here, will not throw to caller
main(): exiting gracefully
```

▼ 分析：

在程序清单 28.4 中，main()调用了 FuncA()，FuncA()调用了 FuncB()，而 FuncB()引发异常，如第 21 行所示。函数 FuncA()和 main()都能处理这种异常，因为它们都包含 catch(const char*)。引发异常的 FuncB()没有 catch()块，因此 FuncB()引发的异常将首先由 FuncA()中的 catch 块（第 34～39 行）处理，

因为是 FuncA()调用了 FuncB()。注意到 FuncA()认为这种异常不严重，没有继续将其传播给 main()。因此，在 main()看来，就像没有问题发生一样。如果解除对第 38 行的注释，异常将传播给 FuncB 的调用者，即 main()也将收到这种异常。

输出指出了对象的创建顺序（与实例化它们的代码的排列顺序相同），还指出了引发异常后对象被销毁的顺序（与实例化顺序相反）。不仅在引发异常的 FuncB()中创建的对象被销毁，在调用 FuncB()并处理异常的 FuncA()中创建的对象也被销毁。

警告

> 程序清单 28.4 表明，引发异常时将对局部对象调用析构函数。
> 如果因出现异常而被调用的析构函数也引发异常，将导致应用程序异常终止。

28.4.1 std::exception 类

程序清单 28.2 捕获 std::bad_alloc 时，实际上是捕获 new 引发的 std::bad_alloc 对象。std::bad_alloc 继承了 C++标准类 std::exception，而 std::exception 是在头文件<exception>中声明的。

下述重要异常类都是从 std::exception 派生而来的。

- bad_alloc：使用 new 请求内存失败时引发。
- bad_cast：试图使用 dynamic_cast 转换错误类型（没有继承关系的类型）时引发。
- ios_base::failure：由 iostream 库中的函数和方法引发。

std::exception 类是异常基类，它定义了虚方法 what()；这个方法很有用且非常重要，详细地描述了导致异常的原因。在程序清单 28.2 中，第 18 行的 exp.what()提供了信息 bad allocation，让用户知道什么地方出了问题。由于 std::exception 是众多异常类型的基类，因此可使用 catch(const exception&)所有将 std::exception 作为基类的异常：

```
void SomeFunc()
{
   try
   {
      // code made exception safe
   }
   catch (const std::exception& exp) // catch bad_alloc, bad_cast, etc
   {
      cout << "Exception encountered: " << exp.what() << endl;
   }
}
```

28.4.2 从 std::exception 派生出自定义异常类

可以引发所需的任何异常。然而，让自定义异常继承 std::exception 的好处在于，现有的异常处理程序 catch(const std::exception&)不但能捕获 bad_alloc、bad_cast 等异常，还能捕获自定义异常，因为它们的基类都是 exception。程序清单 28.5 演示了这一点。

程序清单 28.5 继承 std::exception 的 CustomException 类

```
0: #include <exception>
1: #include <iostream>
2: #include <string>
3: using namespace std;
4:
5: class CustomException: public std::exception
6: {
7:    string Reason;
```

```
 8: public:
 9:    // constructor, needs reason
10:    CustomException(const char* why):Reason(why) {}
11:
12:    // redefining virtual function to return 'Reason'
13:    virtual const char* what() const throw()
14:    {
15:       return Reason.c_str();
16:    }
17: };
18:
19: double Divide(double Dividend, double Divisor)
20: {
21:    if(Divisor == 0)
22:       throw CustomException("CustomException: Dividing by 0 is a crime");
23:
24:    return (Dividend / Divisor);
25: }
26:
27: int main()
28: {
29:    cout << "Enter dividend: ";
30:    double Dividend = 0;
31:    cin >> Dividend;
32:    cout << "Enter divisor: ";
33:    double Divisor = 0;
34:    cin >> Divisor;
35:    try
36:    {
37:       cout << "Result of division is: " << Divide(Dividend, Divisor);
38:    }
39:    catch(exception& exp)// catch CustomException, bad_alloc, etc
40:    {
41:       cout << exp.what() << endl;
42:       cout << "Sorry, can't continue!" << endl;
43:    }
44:
45:    return 0;
46: }
```

▼ 输出：

```
Enter dividend: 2011
Enter divisor: 0
CustomException: Dividing by 0 is a crime
Sorry, can't continue!
```

▼ 分析：

程序清单 28.3 在除以零时引发简单的 char*异常，这里对其进行了修改，实例化了 CustomException 类的一个对象，这个类是在第 5～17 行定义的，它继承了 std::exception。注意到这个自定义异常类实现了虚函数 what()，如第 13～16 所示；该函数返回引发异常的原因。在 main()中，第 39～43 行的 catch(exception&)不但处理异常 CustomException，还处理 bad_alloc 等其他将 exception 作为基类的异常。

注意

请注意程序清单 28.5 中虚方法 CustomException::what()的声明（如第 13 行所示）：

```
virtual const char* what() const throw()
```

它以 throw()结尾，这意味着这个函数本身不会引发异常。这是对异常类的一个重要约束，如果您在该函数中包含一条 throw 语句，编译器将发出警告。如果函数以 throw(int)结尾，意味着该函数可能引发类型为 int 的异常。

应该	不应该
务必捕获类型为 std::exception 的异常。 务必从 std::exception 派生出自定义异常类。 务必谨慎地引发异常。异常不能替代返回值(如 true 或 false)。	不要在析构函数中引发异常。 不要认为内存分配总能成功，务必将使用 new 的代码放在 try 块中，并使用 catch(std::exception&) 捕获可能发生的异常。 不要在 catch()块中包含实现逻辑或分配资源的代码，以免在处理异常的同时导致异常。

28.5 总结

本章介绍了 C++编程的一个重要部分。确保应用程序离开开发环境后依然稳定很重要，这有助于提高用户满意度，并提供直观的用户体验，而这正是异常的用武之地。您发现，分配资源或内存的代码可能失败，因此需要处理它们可能引发的异常。您学习了 C++异常类 std::exception，如果需要编写自定义异常类，最好继承 std::exception。

28.6 问与答

问：为何引发异常，而不是返回错误？

答：不是什么时候都可以返回错误。如果调用 new 失败，需要处理 new 引发的异常，以免应用程序崩溃。另外，如果错误非常严重，导致应用程序无法正常运行，应考虑引发异常。

问：为何自定义异常类应继承 std::exception？

答：当然，并非必须这样做，但这让您能够重用捕获 std::exception 异常的所有 catch()块。编写自己的异常类时，可以不继承任何类，但必须在所有相关的地方插入新的 catch(MyNewExceptionType&)语句。

问：我编写的函数引发异常，必须在该函数中捕获它吗？

答：完全不必，只需确保调用栈中有一个函数捕获这类异常即可。

问：构造函数可引发异常吗？

答：构造函数实际上没有选择余地！它们没有返回值，指出问题的唯一途径是引发异常。

问：析构函数可引发异常吗？

答：从技术上说可以，但这是一种糟糕的做法，因为异常导致退栈时也将调用析构函数。如果因异常而调用的析构函数引发异常，将给原本就稳定并试图妥善退出的应用程序雪上加霜。

28.7 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

28.7.1 测验

1. std::exception 是什么?
2. 使用 new 分配内存失败时，将引发哪种异常?
3. 在异常处理程序（catch 块）中，为大量 int 变量分配内存以便备份数据合适吗?
4. 假设有一个异常类 MyException，它继承了 std::exceptiion，您将如何捕获这种异常对象?

28.7.2 练习

1. 查错：下述代码有何错误?

```
class SomeIntelligentStuff
{
   bool StuffGoneBad;
public:
   ~ SomeIntelligentStuff()
   {
      if(StuffGoneBad)
         throw "Big problem in this class, just FYI";
   }
};
```

2. 查错：下述代码有何错误?

```
int main()
{
   int* pMillionIntegers = new int [1000000];
   // do something with the million integers

   delete []pMillionIntegers;
}
```

3. 查错：下述代码有何错误?

```
int main()
{
   try
   {
      int* pMillionIntegers = new int [1000000];
      // do something with the million integers

      delete []pMillionIntegers;
   }
   catch(exception& exp)
   {
      int* pAnotherMillionIntegers = new int [1000000];
      // take back up of pMillionIntegers and save it to disk
   }
}
```

第 29 章
继续前行

您学习了 C++编程的基本知识。事实上，您已经知道使用标准模板库（STL）、模板和标准库有助于编写高效而紧凑的代码了。现在该考虑考虑性能，并了解最佳编程实践了。

在本章中，您将学习：

- 当今的处理器有何不同；
- C++应用程序如何充分利用处理器的功能；
- 线程和多线程技术；
- C++编程最佳实践；
- 利用其他资源提高 C++技能。

29.1 当今的处理器有何不同

就在不久前，计算机还通过利用处理速度更快的处理器来提高速度；处理速度的度量单位为赫兹（Hz）、兆赫（MHz）或吉赫（GHz）。例如，Intel 8086（如图 29.1 所示）是 1978 年发布的一款 16 位微处理器，时钟频率大约为 10MHz。

那时候，处理器的速度得到了极大的提高，C++应用程序的速度也得到了极大的提高。当时大家都采取伺机而动的策略，利用改进的硬件性能来提高软件的响应速度。虽然当今的处理器越来越快，但真正的创新在于处理器包含的内核数。编写本书期间，Intel 正销售一款 64 位微处理器，它内嵌 6 个 3.2GHz 内核，如图 29.2 所示；当前的发展趋势时，内核数量将不断增多。事实上，智能手机都装备了多核处理器。

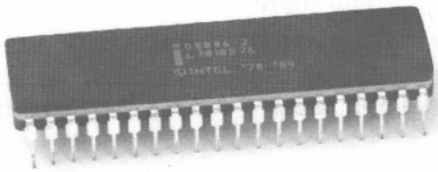


图 29.1 Intel 8086 微处理器

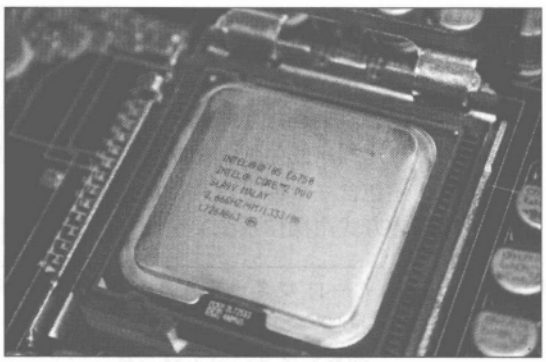


图 29.2 Intel 多核处理器

可将多核处理器视为一块包含多个处理器的芯片。这些处理器并行地运行，每个处理器都有独立

的一级缓存，能够彼此独立地工作。

处理器速度越快，应用程序的性能越高，这合乎逻辑。多核处理器对应用程序性能有何帮助呢？显然，每个内核都能并行地运行应用程序，但这并不能提高应用程序的速度。本书前面介绍的 C++应用程序都是单线程的，不能充分利用多核处理能力。这些应用程序运行在一个线程中，因此只能利用一个内核，如图 29.3 所示。

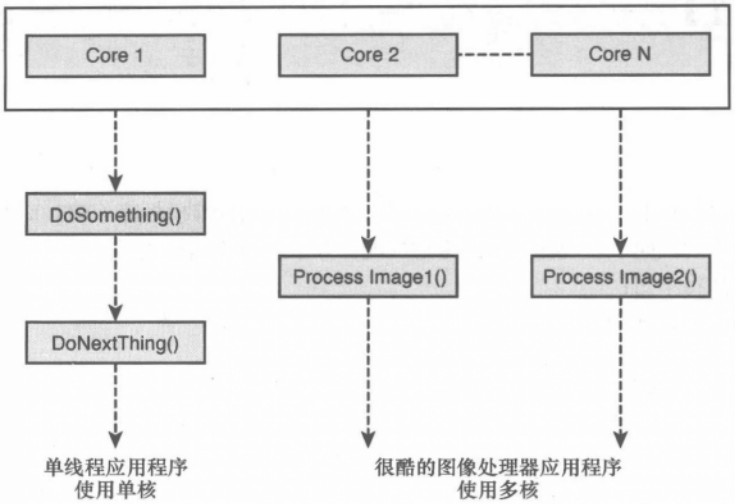


图 29.3 多核处理器中的单线程应用程序

如果应用程序依次执行所有的任务，操作系统（OS）分配给它的时间可能与队列中的其他应用程序一样多，而且它只占用处理器的一个内核。换句话说，在多核处理器中，这种应用程序的运行方式与多年前没什么不同。

29.2 如何更好地利用多个内核

关键在于创建多线程应用程序。所有的线程都并行地运行，操作系统可让它们在多个内核中运行。详细讨论线程和多线程技术超出了本书的范围，这里只简要地介绍这个主题，让您对高性能编程有大致认识。

29.2.1 线程是什么

应用程序代码总是运行在线程中。线程是一个同步执行实体，其中的语句依次执行。可将 main() 的代码视为在应用程序的主线程中执行。在这个主线程中，可以创建并行运行的线程。如果应用程序除主线程外，还包含一个或多个并行运行的线程，则被称为多线程应用程序。

线程的创建方式由操作系统决定，您可直接调用操作系统提供的 API 来创建线程。

提示

C++11 规定由线程函数负责为您调用操作系统 API，这提高了多线程应用程序的可移植性。

如果您编写的应用程序将在特定操作系统上运行，请了解该操作系统提供的用于编写多线程应用程序的 API。

注意

创建线程的方式随操作系统而异，C++11 在头文件<thread>中提供了 std::thread，它隐藏了与平台相关的细节。

编写本书时，主要编译器对这种功能的支持都不太完美。如果您针对特定平台编写应用程序，最好只使用针对该操作系统的线程函数。

编写 C++应用程序时，如果您希望其中的线程是可移植的，请务必了解 Boost 线程库，其网址为 www.boost.org。

29.2.2 为何要编写多线程应用程序

使用多线程技术的应用程序并行地执行特定任务的多个会话（session）。假设有 10000 名用户在 Amazon 购物，您是其中的一员。Amazon 的 Web 服务器当然不会其他 9999 位用户都等待，而是创建多个同时为用户服务的线程。如果该 Web 服务器运行在多核处理器上（我敢打赌，肯定是这样），这些线程将能够充分利用内核，向用户提供最佳的性能。

另一个常见的多线程示例是，与用户交互（例如，通过进度条）的同时做其他工作的应用程序。这样的应用程序通常包含用户界面线程和工作线程，其中前者负责显示和更新用户界面以及接受用户输入，而后者在后台完成其任务。磁盘碎片整理工具就是一个这样的应用程序。用户单击“开始”按钮后，将创建一个工作线程，负责扫描和整理磁盘碎片；与此同时，用户界面线程将显示进度，并提供取消碎片整理的选项。为让用户界面线程显示进度，整理碎片的工作线程需要定期地提供进度；同样，为让工作线程在用户撤销时停止工作，用户界面线程需要提供这种信息。

注意

多线程应用程序常常要求线程彼此通信，这样应用程序才能成为一个整体，而不是一系列互不关心、各自为政的线程。

另外，顺序也很重要，您不希望用户界面线程在负责整理碎片的工作线程之前结束。在有些情况下，一个线程需要等待另一个线程。例如，读取数据库的线程应等待写入数据库的线程结束。让一个线程等待另一个线程被称为线程同步。

29.2.3 线程如何交换数据

线程可共享变量，可访问全局数据。创建线程时，可给它提供一个指向共享对象（结构或类）的指针，如图 29.4 所示。

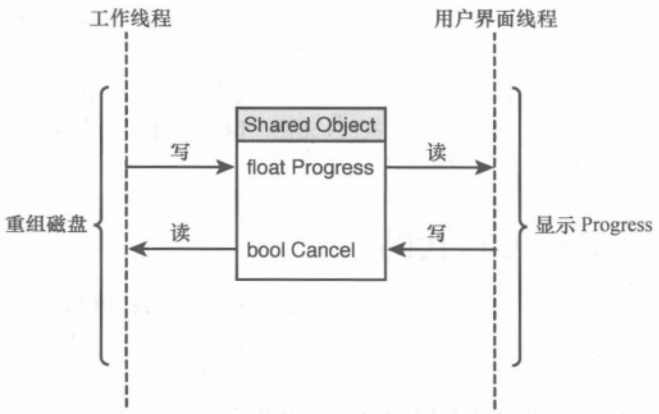


图 29.4 工作线程和用户界面线程共享数据

线程将数据写入其他线程能够存取的内存单元，这让线程能够共享数据，从而彼此进行通信。在磁盘碎片整理工具中，工作线程知道进度，而用户界面线程需要获悉这种信息；工作线程定期地存储进度（用整数表示的百分比），而用户界面线程可使用它来显示进度。

这种情形非常简单：一个线程创建信息，另一个线程使用它。如果多个线程读写相同的内存单元，结果将如何呢？有些线程开始读取数据时，其他线程可能还未结束写入操作，这将给数据的完整性带来威胁。

这就是需要同步线程的原因所在。

29.2.4 使用互斥量和信号量同步线程

线程是操作系统级实体，而用来同步线程的对象也是操作系统提供的。大多数操作系统都提供了信号量（semaphore）和互斥量（mutex），供您用来同步线程。

互斥量通常用于避免多个线程同时访问同一段代码。换句话说，互斥量指定了一段代码，其他线程要执行它，必须等待当前执行它的线程结束并释放该互斥量。接下来，下一个线程获取该互斥量，完成其工作，并释放该互斥量。

通过使用信号量，可指定多少个线程可同时执行某个代码段。只允许一个线程访问的信号量被称为二值信号量（binary semaphore）。

注意 除这些同步对象外，可能还有其他同步对象可供使用，这取决于您使用的操作系统。例如，Windows 支持临界区，临界区指定了不允许多个线程同时执行的代码。

29.2.5 多线程技术带来的问题

要使用多线程技术，必须妥善地同步线程，否则，您将有大量的无眠之夜。多线程应用程序面临的问题很多，下面是最常见的两个。

- **竞争状态**：多个线程试图写入同一项数据。哪个线程获胜？该对象处于什么状态？
- **死锁**：两个线程彼此等待对方结束，导致它们都处于“等待”状态，而应用程序被挂起。

妥善地同步可避免竞争状态。一般而言，线程被允许写入共享对象时，您必须格外小心，确保：

- 每次只能有一个线程写入；
- 在当前执行写入的线程结束前，不允许其他线程读取该对象。

通过确保任何情况下都不会有两个线程彼此等待，可避免死锁。为此，可使用主线程同步工作线程，也可在线程之间分配任务时，确保工作负荷分配明确。可以让一个线程等待另一个线程，但绝不要同时让后者也等待前者。

编写多线程应用程序本身是个专题，详细介绍这个引人注目且激动人心的主题超出了本书的范围。要学习多线程编程，可参阅大量有关该主题的在线文档，也可亲自动手实践。一旦掌握了这个主题，就能让 C++应用程序充分利用未来将发布的多核处理器。

29.3 编写杰出的 C++代码

相比于面世之日，C++发生了巨大变化，主要的编译器厂商在标准化方面做出了巨大努力，还有大量工具和函数，这些都有助于编写简洁的 C++代码。编写可靠且易于理解的 C++应用程序真的很容易。

下面的一些最佳实践可帮助您创建优质的 C++应用程序。

- 给变量指定（无论是对您还是其他人来说都）有意义的名称。值得多花点时间给变量取个好名。
- 对于 int、float 等变量，务必进行初始化。
- 务必将指针初始化为 NULL 或有效的地址——如运算符 new 返回的地址。
- 使用数组时，绝不要跨越其边界。跨越数组边界被称为缓冲区溢出，可导致安全漏洞。
- 不要使用 C 风格字符串（char*），也不要使用 strelen()和 strcopy()等函数。std::string 更安全，还提供了很多有用的方法，如获取长度、进行复制和附加的方法。
- 仅当确定要包含的元素数时才使用静态数组。如果不确定，应使用 std::vector 等动态数组。
- 声明和定义接受非 POD 类型作为输入的函数时，应考虑将参数声明为引用，以免调用函数时执行不必要的复制步骤。
- 如果类包含原始指针成员，务必考虑如何在复制或赋值时管理内存资源所有权，即应考虑编写复制构造函数和赋值运算符。
- 编写管理动态数组的实用类时，务必实现移动构造函数和移动赋值运算符，以改善性能。
- 务必正确地使用 const。理想情况下，get()函数不应修改类成员，因此应将其声明为 const 函数。同样，除非要修改函数参数包含的值，否则应将其声明为 const 引用。
- 不要使用原始指针，而应尽可能使用合适的智能指针。
- 编写实用类时，务必花精力实现让它使用起来更容易的运算符。
- 在有选择余地的情况下，务必使用模板而不是宏。模板不但是通用的，还是类型安全的。
- 编写类时，如果其对象将存储在诸如 vector 和 list 等容器中，或者被用作映射中的键，务必实现运算符<，它将用作默认排序标准。
- 如果您编写的 lambda 表达式很长，应考虑转而使用函数对象，即实现了 operator()的类，因为函数对象可重用，且只有一个地方需要维护。
- 绝不要认为运算符 new 肯定会成功。对于分配资源的代码，务必处理其可能引发的异常，即将其放在 try 块中，并编写相应的 catch()块。
- 绝不要在析构函数中引发异常。

这个清单并非包罗万象，但涵盖了一些最重要的要点，有助于编写杰出且易于维护的 C++代码。

29.4 更深入地学习 C++

祝贺您在学习 C++方面取得了巨大进步，要沿这条道路继续前行，最佳的方式是动手编写大量代码！

C++是一种复杂的语言，越多动手编程，您对幕后的情况就了解得越深入。诸如 Visual Studio 等开发环境提供了智能感知功能，可助您一臂之力，还可满足您的好奇心，如显示 string 类中您从未见过的成员。现在该开始在实践中学习了！

29.4.1 在线文档

要更详细地了解 STL 容器及其方法、算法和功能，请访问 MSDN（http://msdn.microsoft.com/），它非常深入地介绍了标准模板库。

提示

> 阅读 MSDN 中的 STL 文档时，别忘了选择正确的 Visual Studio 版本，因为从 Visual Studio 2010 开始才支持 C++11。

编写本书时，主流 C++编译器都未全面支持所有的 C++11 功能。例如，Visual Studio 2010 不支持可变参数模板。在 GNU GCC 编译器 4.6 版中，std::thread 的实现存在问题。一般而言，了解编译器接下来将支持哪些 C++11 功能是个不错的主意，为此可参阅其在线文档。Visual Studio 开发小组维护着

一个名为 C++11 Core Language Feature Support 的博客，其网址为 http://blogs.msdn.com/b/vcblog/archive/2010/04/06/c-0x-core-language-features-in-vc10-the-table.aspx；而 GCC 提供了一个支持页面，其网址为 http://gcc.gnu.org/projects/cxx0x.html。

注意

编写本书时，这两个编译器都支持 C++11 推荐的大部分功能。另外，本书的代码都使用了这两个编译器进行了测试。

29.4.2 提供指南和帮助的社区

有很多活跃的 C++社区。在 CodeGuru（www.CodeGuru.com）或 CodeProject（www.CodeProject.com）网站注册后，您就可询问自己遇到的技术问题，并获得社区的帮助。

如果您对自己的水平有自信，可为这些社区的其他成员提供帮助。您将发现，在解答难题的过程中，您将学到很多。

29.5 总结

这是本书的最后一章，实际上是您探索 C++的开篇！在此之前，您学习了 C++基本知识和高级概念；而在本章中，您学习了多线程编程的理论基础。您了解到，要充分利用多核处理器，唯一的途径就是将逻辑放在多个线程中，并支持并行处理。您知道了如何避开多线程编程面临的陷阱。最后，您学习了一些基本的 C++编程最佳实践。您知道，要编写优质的 C++代码，不仅要使用高级概念，还应给变量指定别人能够明白的名称，处理异常以应付意外情况，并使用智能指针等实用类（而不是原始指针）。您已为进入专业 C++编程领域做好了充分准备。

29.6 问与答

问：我编写了一个应用程序，对其性能很满意，还应考虑在其中实现多线程功能吗？

答：根本不用考虑。并非所有应用程序都需要是多线程的，仅当应用程序需要并行地执行任务或同时为众多用户提供服务时，才需要使用多线程技术。

问：既然主流编译器还未全面支持 C++11，我为何不采用老式编程风格呢？

答：首先，两款主流编译器（Microsoft Visual C++和 GNU GCC）都支持大部分 C++11 功能，只有少数功能除外。另外，C++11 让编程更容易。使用关键字 auto 可简化迭代器声明，而 lambda 表达式让 for_each()结构非常紧凑，无需编写函数对象。因此，使用 C++11 编程的好处已非常明显。

29.7 作业

作业包括测验，帮助读者加深对所学知识的理解。请尽量先完成测验，再对照附录 D 的答案。

测验

1. 我编写了一个图像处理应用程序，它在校正对比度时没有响应，我该如何办？

2. 我编写了一个多线程应用程序，能够以极快的速度访问数据库，但有时取回的数据不完整，请问是哪里出了问题？

附录 A
二进制和十六进制

对使用 C++编写更好的应用程序来说，理解二进制和十六进制的工作原理并非至关重要，但这有助于更深入地了解幕后发生的情况。

A.1 十进制

我们日常使用的数字用 0～9 表示，这种数字被称为十进制数。十进制使用 10 个不同的数字，其基数为 10。

基数为 10 时，如果从零开始对各位进行编号，则每位表示的值为该位的数字乘以 10 编号，因此：

$957 = 9 \times 10^2 + 5 \times 10^1 + 7 \times 10^0 = 9 \times 100 + 5 \times 10 + 7$

在数字 957 中，7 对应的位编号为 0，5 对应的位编号为 1，9 对应的位编号为 2。这些位编号将用作以 10 为底的指数，如上述示例所示。别忘了，任何数的零次方都为 1，因此 10^0 和 1000^0 的值相同，都是 1。

注意

对十进制来说，10 的幂很重要。在十进制数中，各位的量级分别是 10、100、1000 等。

A.2 二进制

二进制的基数为 2。在二进制中，每位只有两种可能的状态，用数字 0 和 1 表示。在 C++中，0 和 1 分别对应于 false 和 true（true 为非零值）。

就像十进制数是根据 10 的幂计算其表示的值一样，二进制数根据 2 的幂计算其表示的值：

$101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = 5_{10}$

因此，二进制数 101 对应的十进制数为 5。

注意

在二进制数中，各位的量级 2 的幂，即分别是 4、8、16、32 等。其中，指数为当前位的编号，而编号从零开始。

为更深入地了解二进制，请看表 A.1，其中列出了 2 的幂。

表 A.1 2 的幂

次方	值	二进制表示
0	$2^0 = 1$	1
1	$2^1 = 2$	10
2	$2^2 = 4$	100
3	$2^3 = 4$	1000

续表

次方	值	二进制表示
4	$2^4=4$	10000
5	$2^5=4$	100000
6	$2^6=4$	1000000
7	$2^7=4$	10000000

A.2.1 计算机为何使用二进制

二进制广泛使用的历史较短，电子学和计算机的发展使其应用得以普及。电子学和电子元件的发展导致了一个新系统，它将元件状态视为 ON（高电平）或 OFF（低电平）。

ON 和 OFF 状态非常适合使用 1 和 0 来表示，而这正是二进制使用的数字，因此二进制可用于算术计算。通过开发电子门，很容易支持第 5 章介绍的逻辑运算，如 NOT、AND、OR 和 XOR，这使得使用二进制进行条件处理很容易。

A.2.2 位和字节

位是计算系统中基本单位，包含一个二值状态。因此，如果位包含状态 1，则称为被“设置”，如果包含状态 0，则称为被“重置”。一系列位称为字节；从理论上说，一个字节包含的位数并非固定的，它随硬件而异。

然而，大多数计算系统都假定一个字节包含 8 位，这是处于简单和方便的考虑，因为 8 为 2^3。由于一个字节包含 8 位，因此能存储 2^8（256）个不同的值，这足以表示 ASCII 字符集中的所有字符。

A.2.3 1KB 相当于多少字节

1KB 为 1024（2^{10}）字节。同样，1MB 为 1024KB，1GB 为 1024MB，1TB 为 1024GB。

A.3 十六进制

十六进制的基数为 16。在十六进制中，各位的值用 0-9 和 A-F 表示，因此十进制数 10 对应的十六进制值为 A，十进制数 15 对应的十六进制值为 F：

十进制数	十六进制数	十进制数	十六进制数
0	0	8	8
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F

在十进制中，各位的量级为 10 的幂，而在二进制中，各位的量级为 2 的幂；同样，在十六进制中，各位的量级为 16 的幂：

$0x31F = 3\times16^2 + 1\times16^1 + F\times16^0 = 3\times256 + 16 + 15 = 799_{10}$

注意 根据约定，使用前缀 0x 表示十六进制数。

A.3.1 为何需要十六进制

计算机使用二进制。在计算机中，每个内存单元的状态为 0 或 1。然而，如果人类在计算机中使用 0 和 1 来表示编程信息，则表示少量信息就需大量空间。因此，如果不使用二进制值 1111，而使用十六进制值 F，效率将高得多。

每个十六进制位可表示 4 个二进制位，因此使用两个十六进制位就能表示一个字节的状态，其效率非常高。

注意 另一种进制是八进制，用得较少。八进制的基数为 8，每位用数字 0～7 表示。

A.4 不同进制之间的转换

处理数字时，您可能需要用不同的进制表示同一个数字，如二进制值的十进制表示或十进制数的十六进制表示。

前面的示例演示了如何将二进制数或十六进制数转换为十进制数，下面来看看如何将十进制数转换为二进制和十六进制。

A.4.1 通用转换步骤

在不同进制之间进行转换时，将从要转换的数字开始，不断地除以目标基数，并从最右边开始，不断将余数填入目标数中。下一次执行除法运算时，将前一次除法运算的商作为被除数，并将目标基数作为除数。

这个过程将不断持续下去，直到余数可用目标进制的一位表示，且商为零。

这种方法也被称为分解法（breakdown method）。

A.4.2 从十进制转换为二进制

要将十进制数 33 转换为二进制，步骤如下：

第 1 位：将 33 除以 2，商为 16，余数为 1

第 2 位：将 16 除以 2，商为 8，余数为 0

第 3 位：将 8 除以 2，商为 4，余数为 0

第 4 位：将 4 除以 2，商为 2，余数为 0

第 5 位：将 2 除以 2，商为 1，余数为 0

第 6 位：将 1 除以 2，商为 0，余数为 1

因此十进制数 33 的二进制表示为：100001。

同样，要将十进制数 156 转换为二进制，步骤如下：

第 1 位：将 156 除以 2，商为 78，余数为 0

第 2 位：将 78 除以 2，商为 39，余数为 0

第 3 位：将 39 除以 2，商为 19，余数为 1

第 4 位：将 19 除以 2，商为 9，余数为 1

第 5 位：将 9 除以 2，商为 4，余数为 1
第 6 位：将 4 除以 2，商为 2，余数为 0
第 7 位：将 2 除以 2，商为 1，余数为 0
第 8 位：将 1 除以 2，商为 0，余数为 1
因此十进制数 156 的二进制表示为：10011100。

A.4.3 从十进制转换为十六进制

步骤与转换为二进制相同，但除以基数 16，而不是 2。
因此，要将十进制数 5211 转换为十六进制，步骤如下：
第 1 位：将 5211 除以 16，商为 325，余数为 11（对应的十六进制数为 B）
第 2 位：将 325 除以 16，商为 20，余数为 5
第 3 位：将 20 除以 16，商为 1，余数为 4
第 4 位：将 1 除以 16，商为 0，余数为 1
因此十进制数 5211 的二进制表示为：145B。

提示

要更深入地了解各种进制的工作原理，可编写一个类似于程序清单 27.1 的简单 C++程序，在其中使用 std::cout 和控制符显示一个整数的十六进制、十进制和八进制表示。
要显示整数的二进制表示，可使用第 25 章介绍的 std::bitset，并参阅程序清单 25.1。

附录 B
C++关键字

关键字是语言保留给编译器使用的。不能将关键字用作类、变量或函数的名称。

```
asm                 false               signed
auto                final               sizeof
bool                float               static
break               for                 static_assert
case                friend              static_cast
catch               goto                struct
char                if                  switch
class               inline              template
const               int                 this
constexpr           long                throw
const_cast          long long int       true
continue            mutable             try
decltype            namespace           typedef
default             new                 typeid
delete              operator            typename
do                  override            union
double              private             unsigned
dynamic_cast        protected           using
else                public              virtual
enum                register            void
explicit            reinterpret_cast    volatile
export              return              wchar_t
extern              short               while
```

另外，下述关键字也被保留：

```
and                 compl               or_eq
and_eq              not                 xor
bitand              not_eq              xor_eq
bitor               or
```

附录 C
运算符优先级

书写表达式时，您最好使用括号明确地指出运算的计算顺序；如果没有括号，编译器将根据预定义的优先级确定运算的计算顺序。表 C.1 列出了运算符优先级，程序员未明确指定计算顺序时，C++编译器将根据优先级进行确定。

表 C.1　　运算符优先级

等级	名称	运算符
1	作用域解析运算符	::
2	成员选择、下标、函数调用、后缀递增和后缀递减	.、->、()、++、--
3	sizeof、前缀递增和递减、求补、逻辑 NOT、单目加和减、取址和解除引用、new、new[]、delete、delete[]、类型转换、sizeof( )	++、--、^、!、+、-、&、*、()
4	用于指针的成员选择	.*、->*
5	乘、除、求模	*、/、%
6	加、减	+、-
7	移位（左移和右移）	<<、>>
8	不等关系	<、<=、>、>=
9	相等关系	==、!=
10	按位 AND	&
11	按位 XOR	^
12	按位 OR	|
13	逻辑 AND	&&
14	逻辑 OR	||
15	条件运算符	?:
16	赋值运算符	=、*=、/=、%=、+=、-=、<<=、>>=、&=、|=、^=
17	逗号运算符	,

附录D
答　案

第1章

测验

1. 解释器是一种对代码（或字节码）进行解释并执行相应操作的工具；编译器将代码作为输入，并生成目标文件。就C++而言，编译和链接后，将得到一个可执行文件，处理器可直接执行它，而无需做进一步解释。

2. 编译器将C++代码文件作为输入，并生成一个使用机器语言的目标文件。通常，您的代码依赖于库和其他代码文件中的函数。链接器负责建立这些链接，并生成一个可执行文件，它集成了您指定的所有直接或间接依存关系。

3. 编写代码；通过编译创建目标文件；通过链接创建可执行文件；执行应用程序以便进行测试；调试；修复代码中的错误并重复这些步骤。

4. C++支持可移植的线程模型，让程序员能够使用标准C++线程函数创建多线程应用程序。通过在多个CPU核心中同时执行应用程序的不同线程，最大限度地发挥了多核处理器的潜力。

练习

1. 显示x减y、x乘以y和x加y的结果。

2. 输出应如下：

```
2 48 14
```

3. 在第1行，包含iostream的预编译器指令应以#打头。

4. 它显示如下内容：

```
Hello Buggy World
```

第2章

测验

1. C++代码区分大小写，在编译器看来，Int与表示整型的int不是一回事。

2. 可以。

```
/*  if you comment using this C-style syntax
then you can span your comment over multiple lines */
```

练习

1. 因为C++编译器区分大小写，不知道std::Cout是什么以及它后面的字符串为何不以左引号打

头。另外，声明 main 时，总是应该将其返回类型指定为 int。

2. 下面是修正后的程序：

```
#include <iostream>
int main()
{
     std::cout << "Is there a bug here?"; // no bug anymore
     return 0;
}
```

3. 下面的程序修改了程序清单 2.4，以演示减法和乘法：

```
##include <iostream>
#using namespace std;
u
// Function declaration
iint DemoConsoleOutput();
{
 int main()
 {
   // Call i.e. invoke the function
    DemoConsoleOutput();

    return 0;
 }

 // Function definition
 int DemoConsoleOutput()
 {
    cout << "Performing subtraction 10 - 5 = " << 10 - 5 << endl;
   cout << "Performing multiplication 10 * 5 = " << 10 * 5 << endl;

   return 0;
}
```

▼ 输出：

```
Performing subtraction 10 - 5 = 5
Performing multiplication 10 * 5 = 50
```

第 3 章

测验

1. 有符号整型变量的最高有效位（MSB）用作符号位，指出了整数值是正还是负，而无符号整型变量只能存储正整数。

2. #define 是一个预处理器编译指令，让编译器对定义的值进行文本替换。然而，它不是类型安全的，是一种原始的常量定义方式，应避免使用。

3. 确保变量包含非随机的确定值。

4. 2。

5. 这个变量名不具描述性，并重复地指出了变量的类型。这种代码虽然能够通过编译，但难以理解和维护，应避免。声明变量时，应使用能揭示其用途的名称，如：

```
int Age = 0;
```

练习

1. 方式有很多，下面是其中的两种：

```
enum YOURCARDS {ACE = 43, JACK, QUEEN, KING};
// ACE is 43, JACK is 44, QUEEN is 45, KING is 46
// Alternatively..
enum YOURCARDS {ACE, JACK, QUEEN = 45, KING};
// ACE is 0, JACK is 1, QUEEN is 45 and KING is 46
```

2. 参考程序清单 3.4，并对其进行修改以获得这个问题的答案。

3. 下面的程序要求用户输入圆的半径，并计算其面积和周长：

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

int main()
{
   const double Pi = 3.1416;

   cout << "Enter circle's radius: ";
    double Radius = 0;
   cin >> Radius;

   cout << "Area = " << Pi * Radius * Radius << endl;
   cout << "Circumference = " << 2 * Pi * Radius << endl;

   return 0;
}
```

▼ 输出：

```
Enter circle's radius: 4
Area = 50.2656
Circumference = 25.1328
```

4. 如果将计算得到的面积和周长存储到 int 变量中，将出现编译警告（而不是编译错误），且输出类似于下面这样：

```
Enter circle's radius: 4
Area = 50
Circumference = 25
```

5. auto 让编译器根据赋给变量的初始值自动确定其类型。这里的代码未初始化变量，无法通过编译。

第 4 章

测验

1. 对于包含 5 个元素的数组，其第一个元素和最后一个元素的索引分别是 0 和 4。

2. 不应该，因为使用 C 风格字符串来存储用户输入很不安全，用户可能输入比数组长度更长的字符串。

3. 一个终止空字符。

4. 这取决于您如何使用它。如果将其用于 cout 语句，将不断读取字符，直到遇到终止空字符。这将跨越数组边界，可能导致应用程序崩溃。

5. 只需将矢量声明中的 int 替换为 char 即可：

```
vector<char> DynArrChars (3);
```

练习

1. 下面是一种可能的解决方案，这里只初始化了包含“车”的棋盘方格，但足以让您明白其中的要点：

```
int main()
{
  enum SQUARE
  {
      NOTHING = 0,
     PAWN,
     ROOK,
     KNIGHT,
     BISHOP,
     KING,
     QUEEN
   };

  SQUARE ChessBoard[8][8];

  // Initialize the squares containing rooks
  ChessBoard[0][0] = ChessBoard[0][7] = ROOK;
  ChessBoard[7][0] = ChessBoard[7][7] = ROOK;

  return 0;
}
```

2. 要设置第 5 个元素，应使用 MyNumbers[4]，因为索引从零开始。

3. 这里使用了数组的第 4 个元素，但之前没有初始化，也没有赋值，因此输出是不确定的。务必初始化变量和数组，否则它将包含最后一次存储在相应内存单元中的值。

第 5 章

测验

1. 用户可能想将两个浮点数相除，而 int 变量不能包含小数，因此应使用 float。

2. 编译器将操作数视为整数，因此结果为 4。

3. 由于分子为 32.0 而不是 32，因此编译器将此视为为浮点数运算，结果为浮点数，大约为 4.571。

4. sizeof 为运算符，不能重载。

5. 与预期不符，因为加法运算符的优先级高于移位运算符，因此将对 number 移 6（1+5）位，而不是 1 位。

6. XOR 的结果为 false，如表 5.5 所示。

练习

1. 下面是一种解决方案：

```
int Result = ((number << 1) + 5) << 1; // unambiguous even to reader
```

2. result 包含将 number 向左移 7 位的结果，因为运算符+的优先级高于运算符<<。

3. 下面的程序让用户输入两个布尔值，并显示对其执行各种按位运算的结果：

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

int main()
{
   cout << "Enter a boolean value true(1) or false(0): ";
   bool Value1 = false;
   cin >> Value1;

   cout << "Enter another boolean value true(1) or false(0): ";
  bool Value2 = false;
```

```
    cin >> Value2;

    cout << "Result of bitwise operators on these operands: " << endl;
    cout << "Bitwise AND: " << (Value1 & Value2) << endl;
    cout << "Bitwise OR: " << (Value1 | Value2) << endl;
    cout << "Bitwise XOR: " << (Value1 ^ Value2) << endl;

    return 0;
}
```

▼ **输出：**

```
Enter a boolean value true(1) or false(0): 1
Enter another boolean value true(1) or false(0): 0
Result of bitwise operators on these operands:
Bitwise AND: 0
Bitwise OR: 1
Bitwise XOR: 1
```

第 6 章

测验

1. 缩进并非为方便编译器，而是为了方便其他需要阅读或理解代码的程序员。
2. 通过避免使用 goto，可防止代码不直观且难以维护。
3. 参见练习 1 的解决方案，其中使用了递减运算符。
4. 由于 for 语句中的条件不满足，该循环一次也不会执行，其中的 cout 语句也不会执行。

练习

1. 您需要知道，数组的索引从零开始，而最后一个元素的索引为 Length -1：

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

int main()
{
    const int ARRAY_LEN = 5;
    int MyNumbers[ARRAY_LEN]= {-55, 45, 9889, 0, 45};

    for (int nIndex = ARRAY_LEN - 1; nIndex >= 0; --nIndex)
        cout<<"MyNumbers[" << nIndex << "] = "<<MyNumbers[nIndex]<<endl;

    return 0;
}
```

▼ **输出：**

```
MyNumbers[4] = 45
MyNumbers[3] = 0
MyNumbers[2] = 9889
MyNumbers[1] = 45
MyNumbers[0] = -55
```

2. 下面的嵌套循环类似于程序清单 6.13，但以倒序方式将一个数组的每个元素都与另一个数组的每个元素相加：

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

int main()
```

```
{
   const int ARRAY1_LEN = 3;
   const int ARRAY2_LEN = 2;

   int MyInts1[ARRAY1_LEN] = {35, -3, 0};
   int MyInts2[ARRAY2_LEN] = {20, -1};

   cout << "Adding each int in MyInts1 by each in MyInts2:" << endl;

   for(int Array1Index=ARRAY1_LEN-1;Array1Index>=0;--Array1Index)
      for(int Array2Index=ARRAY2_LEN-1;Array2Index>=0;--Array2Index)
         cout<<MyInts1[Array1Index]<<" + "<<MyInts2[Array2Index] \
         << " = " << MyInts1[Array1Index] + MyInts2[Array2Index] << endl;

   return 0;
}
```

▼ **输出：**

```
0 + -1 = -1
0 + 20 = 20
-3 + -1 = -4
-3 + 20 = 17
35 + -1 = 34
35 + 20 = 55
```

3. 需要将值为 5 的 int 常量替换为下述让用户输入的代码：

```
cout << "How many Fibonacci numbers you wish to calculate: ";
int NumsToCal = 0;
cin >> NumsToCal;
```

4. 下面的 switch-case 结构使用枚举常量指出用户选择的颜色是否包含在彩虹中：

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

int main()
{
   enum COLORS
   {
      VIOLET = 0,
      INDIGO,
      BLUE,
      GREEN,
      YELLOW,
      ORANGE,
      RED,
      CRIMSON,
      BEIGE,
      BROWN,
      PEACH,
      PINK,
      WHITE,
   };

   cout << "Here are the available colors: " << endl;
   cout << "Violet: " << VIOLET << endl;
   cout << "Indigo: " << INDIGO << endl;
   cout << "Blue: " << BLUE << endl;
   cout << "Green: " << GREEN << endl;
   cout << "Yellow: " << YELLOW << endl;
   cout << "Orange: " << ORANGE << endl;
   cout << "RED: " << RED << endl;
   cout << "Crimson: " << CRIMSON << endl;
```

```
    cout << "Beige: " << BEIGE << endl;
    cout << "Brown: " << BROWN << endl;
    cout << "Peach: " << PEACH << endl;
    cout << "Pink: " << PINK << endl;
    cout << "White: " << WHITE << endl;

    cout << "Choose one by entering code: ";
    int YourChoice = BLUE; // initial
    cin >> YourChoice;

    switch (YourChoice)
    {
    case VIOLET:
    case INDIGO:
    case BLUE:
    case GREEN:
    case YELLOW:
    case ORANGE:
    case RED:
       cout << "Bingo, your choice is a Rainbow color!" << endl;
       break;

    default:
       cout << "The color you chose is not in the rainbow" << endl;
       break;
    }

    return 0;
}
```

▼ 输出：

```
Here are the available colors:
Violet: 0
Indigo: 1
Blue: 2
Green: 3
Yellow: 4
Orange: 5
RED: 6
Crimson: 7
Beige: 8
Brown: 9
Peach: 10
Pink: 11
White: 12
Choose one by entering code: 4
Bingo, your choice is a Rainbow color!
```

5. 在 for 循环条件中，程序员不小心将 10 赋给了一个变量。

6. 在 while 语句后面是一条空语句（;），因此无法实现预期的循环。另外，由于控制 while 的 LoopCounter 永远不会递增，因此 while 循环永远不会结束，它后面的语句不会执行。

7. 遗漏了 break。由于之前所有的 case 没有退出 switch-case 结构 break，因此 default 部分总是会执行。

第 7 章

测验

1. 这些变量的作用域为当前函数。

2. SomeNumber 是指向调用函数中相应变量的引用，而不是其拷贝。
3. 递归函数。
4. 重载的函数。
5. 栈顶！可将栈视为一叠盘子，可取出最上面的盘子，栈指针指向的就是这个地方。

练习

1. 函数原型将类似于下面这样：

```
double Area (double Radius); // circle
double Area (double Radius, double Height); // cylinder
```

函数实现（定义）使用提供的公式计算体积，并将其作为返回值返回给调用者。

2. 请参阅程序清单 7.8。函数原型类似于下面这样：

```
void ProcessArray(double Numbers[], int Length);
```

3. 要让函数 Area 发挥作用，参数 Result 应为引用：

```
void Area(double Radius, double &Result)
```

4. 要么将有默认值的参数放在列表末尾，要么给所有参数都指定默认值。
5. 该函数应通过引用将其输出数据返回给调用者。

```
void Area (double Radius, double &Area, double &Circumference)
{
   Area = 3.14 * Radius * Radius;
   Circumference = 2 * 3.14 * Radius;
}
```

第 8 章

测验

1. 如果编译器允许这样做，将能轻松地突破 const 引用的限制：不能修改它指向的数据。
2. 它们是运算符。
3. 内存地址。
4. 运算符*。

练习

1. 40。
2. 在第一个重载的函数中，实参将被复制给形参；在第二个函数中，不会复制，相反，形参是指向实参的引用，且函数可以修改它们；第三个使用的是指针，指针不同于引用，可能为 NULL 或无效，因此使用前必须核实它们是有效的。
3. 使用关键字 const：

```
1: const int* pNum1 = &Number;
```

4. 将整数直接赋给了指针，这将把指针包含的内存地址改为相应的整数值：

```
*pNumber = 9; // previously: pNumber = 9;
```

5. new 返回给 pNumber 的内存地址被复制给了 pNumberCopy，不能对该内存地址调用 delete 两次。删除其中一条 delete 语句。
6. 30。

第 9 章

测验

1. 在自由存储区中创建，与使用 new 给 int 变量分配内存时一样。

2. sizeof( )根据声明的数据成员计算类的大小。将 sizeof( )用于指针时，结果与指向的数据量无关，因此类包含指针成员时，将 sizeof 用于该类的结果也是固定的。

3. 除该类的成员方法外，在其他地方都不能访问。

4. 可以。

5. 构造函数通常用于初始化数据成员和资源。

6. 析构函数通常用于释放资源和内存。

练习

1. C++区分大小写。类声明应以 class（而不是 Class）打头，且以分号（;）结尾，如下所示：

```
class Human
{
   int Age;
   string Name;

public:
   Human() {}
};
```

2. 由于 Human::Age 是私有成员（别忘了，不同于结构，类的成员默认为私有），且没有公有的存取函数，因此这个类的用户无法访问 Age。

3. 在下面的 Human 类中，构造函数包含一个初始化列表：

```
class Human
{
   int Age;
   string Name;

public:
   Human(string InputName, int InputAge)
        : Name(InputName), Age(InputAge) {}
};
```

4. 注意到按要求未将 Pi 向外暴露：

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

class Circle
{
   const double Pi;
   double Radius;

public:
   Circle(double InputRadius) : Radius(InputRadius), Pi(3.1416) {}
   double GetCircumference()
   {
      return 2*Pi*Radius;
   }

   double GetArea()
   {
```

```
        return Pi*Radius*Radius;
    }
};

int main()
{
    cout << "Enter a radius: ";
    double Radius = 0;
    cin >> Radius;

    Circle MyCircle(Radius);

    cout << "Circumference = " << MyCircle.GetCircumference() << endl;
    cout << "Area = " << MyCircle.GetArea() << endl;

    return 0;
}
```

第 10 章

测验

1. 通过使用访问限定符 protected，可确保派生类能够访问基类的成员，但不能通过派生类实例进行访问。

2. 派生类对象被切除，只按值传递对应于基类的部分。切除导致的行为无法预测。

3. 使用组合，这样可提高设计的灵活性。

4. 用于避免隐藏基类方法。

5. 不能，因为 Derived 类与 Base 类是私有继承关系，这导致 Base 类对 SubDerived 类隐藏了其公有成员，即 SubDerived 不能访问它们。

练习

1. 构造顺序与类声明中指定的顺序相同，即依次为 Mammal、Bird、Reptile 和 Platypus；析构顺序则相反。下面的程序演示了这一点：

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

class Mammal
{
public:
    void FeedBabyMilk()
    {
        cout << "Mammal: Baby says glug!" << endl;
    }

    Mammal()
    {
        cout << "Mammal: Contructor" << endl;
    }
    ~Mammal()
    {
        cout << "Mammal: Destructor" << endl;
    }
};

class Reptile
```

```
{
public:
   void SpitVenom()
   {
      cout << "Reptile: Shoo enemy! Spits venom!" << endl;
   }

   Reptile()
   {
      cout << "Reptile: Contructor" << endl;
   }
   ~Reptile()
   {
      cout << "Reptile: Destructor" << endl;
   }
};

class Bird
{
public:
   void LayEggs()
   {
      cout << "Bird: Laid my eggs, am lighter now!" << endl;
   }

   Bird()
   {
      cout << "Bird: Contructor" << endl;
   }
   ~Bird()
   {
      cout << "Bird: Destructor" << endl;
   }
};

class Platypus: public Mammal, public Bird, public Reptile
{
public:
   Platypus()
   {
      cout << "Platypus: Contructor" << endl;
   }
   ~Platypus()
   {
      cout << "Platypus: Destructor" << endl;
   }
};

int main()
{
   Platypus realFreak;
   //realFreak.LayEggs();
   //realFreak.FeedBabyMilk();
   //realFreak.SpitVenom();

   return 0;
}
```

2. 类似于下面这样：

```
class Shape
{
   // ... Shape members
```

```
};

class Polygon: public Shape
{
   // ... Polygon members
}

class Triangle: public Polygon
{
   // ... Triangle members
}
```

3. 要禁止 D2 类访问 Base 类的公有成员，D1 类和 Base 类之间的继承关系应为私有的。

4. 类的继承关系默认为私有。如果 Derived 是结构，继承关系将为公有。

5. SomeFunc 按值接受一个类型为 Base 的参数。这意味着下面的函数调用将发生切除，导致不稳定和不可预测的结果：

```
Derived objectDerived;
SomeFunc(objectDerived); // will slice
```

第 11 章

测验

1. 声明抽象基类 Shape，并在其中将 Area( )和 Print( )声明为纯虚函数，从而要求 Circle 和 Triangle 必须实现这些函数。

2. 编译器只为包含虚函数的类（包括派生类）创建 VFT。

3. 是抽象基类，因为它不能被实例化。只要类至少包含一个纯虚函数，它就是抽象基类，而不管它是否包含其他定义完整的函数和属性。

练习

1. 继承层次结构如下，其中 Shape 是抽象基类，Circle 和 Triangle 从 Shape 派生而来：

Triangle is as below:

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

class Shape
{
public:
   virtual double Area() = 0;
   virtual void Print() = 0;
};

class Circle
{
   double Radius;
public:
   Circle(double inputRadius) : Radius(inputRadius) {}

   double Area()
   {
      return 3.1415 * Radius * Radius;
   }

   void Print()
   {
```

```
        cout << "Circle says hello!" << endl;
    }
};

class Triangle
{
    double Base, Height;
public:
    Triangle(double inputBase, double inputHeight) : Base(inputBase),
Height(inputHeight) {}

    double Area()
    {
        return 0.5 * Base * Height;
    }

    void Print()
    {
        cout << "Triangle says hello!" << endl;
    }
};

int main()
{
    Circle myRing(5);
    Triangle myWarningTriangle(6.6, 2);

    cout << "Area of circle: " << myRing.Area() << endl;
    cout << "Area of triangle: " << myWarningTriangle.Area() << endl;

    myRing.Print();
    myWarningTriangle.Print();

    return 0;
}
```

2. 缺少虚析构函数。

3. 实例化时，依次调用构造函数 Car( )和 Vehicle；由于没有虚析构函数，销毁时只调用～Car( )。

第 12 章

测验

1. 不可以。C++不允许两个函数的名称相同，但返回类型不同。您可编写运算符[ ]的这样两种实现：它们的返回类型相同，但一个为 const 函数，另一个不是。在这种情况下，如果执行的是与赋值相关的操作，编译器将使用非 const 版本，否则使用 const 版本：

```
Type& operator[](int Index) const;
Type& operator[](int Index);
```

2. 可以，但仅当您希望类不允许复制或赋值时才应这样做。

3. 只有动态分配的资源才会导致复制构造函数和复制赋值运算符进行不必要的内存分配和释放，而 Date 类没有包含动态分配的资源，因此给它提供移动构造函数或移动赋值运算符没有意义。

练习

1. 转换运算符 int( )如下所示：

```
class Date
{
   int Day, Month, Year;
public:
   operator int()
   {
      return ((Year * 10000) + (Month * 100) + Day);
   }

   // constructor etc
};
```

2. 移动构造函数和移动赋值运算符如下所示：

```
class DynIntegers
{
private:
   int* pIntegers;

public:
   // move constructor
   DynIntegers(DynIntegers&& MoveSource)
   {
      pIntegers = MoveSource.pIntegers;  // take ownership
      MoveSource.pIntegers = NULL;  // release ownership from source
   }

   // move assignment operator
   DynIntegers& operator= (DynIntegers&& MoveSource)
   {
      if(this != &MoveSource)
      {
         delete [] pIntegers;  // release own resources
         pIntegers = MoveSource.pIntegers;
         MoveSource.pIntegers = NULL;
      }
      return *this;
   }

   ~DynIntegers() {delete[] pIntegers;}  // destructor

   // implement default constructor, copy constructor, assignment operator
};
```

第 13 章

测验

1. dynamic_cast。
2. 当然是修改函数。一般而言，除非万不得已，否则不要使用 const_cast 和类型转换运算符。
3. 对。
4. 对。

练习

1. 总是应该检查动态转换的结果，看其是否有效：

```
void DoSomething(Base* pBase)
{
   Derived* pDerived = dynamic_cast <Derived*>(pBase);
```

```
    if(pDerived) // check for validity
       pDerived->DerivedClassMethod();
}
```

2. 由于知道指向的是 Tuna 对象，应使用 static_cast。为证明这一点，对程序清单 13.1 进行修改，将 main( )改成下面这样：

```
int main()
{
   Fish* pFish = new Tuna;
   Tuna* pTuna = static_cast<Tuna*>(pFish);
   // Tuna::BecomeDinner will work only using valid Tuna*
   pTuna->BecomeDinner();

   // virtual destructor in Fish ensures invocation of ~Tuna()
   delete pFish;

   return 0;
}
```

第 14 章

测验

1. 这是一个预编译器结构，用于避免多次或递归包含头文件。
2. 4。
4. 结果为 10 + 10 / 5 = 10 + 2 =12。
5. 加上括号：

```
#define SPLIT(x) ((x) / 5)
```

练习

1. 如下所示：

```
#define MULTIPLY(a,b) ((a)*(b))
```

2. 模板函数如下：

```
template<typename T> T Split(const T& input)
{
   return (input / 5);
}
```

3. 模板函数 swap 的定义如下：

```
template <typename T>
void Swap (T& x, T& y)
{
     T temp = x;
     x = y;
     y = temp;
}
```

4. `#define QUARTER(x) ((x)/ 4)`
5. 该模板类的定义类似于下面这样：

```
template <typename Array1Type, typename Array2Type>
class TwoArrays
{
private:
        Array1Type Array1 [10];
        Array2Type Array2 [10];
```

```
public:
        Array1Type& GetArray1Element(int Index){return Array1[Index];}
        Array2Type& GetArray2Element(int Index){return Array2[Index];}
};
```

第 15 章

测验

1. deque。只有 deque 允许在容器开头和末尾插入元素，且时间是固定的。

2. 如果要存储的是键-值对，应选择 std::set 或 std::map；如果有重复的元素，应选择 std::multiset 或 std::multimap。

3. 可能。实例化 std::set 模板时，可提供第二个模板参数，它是一个二元谓词，set 类将它用作排序标准。可根据应用程序的需求定义该二元谓词，它必须能够对元素进行排序。

4. 迭代器在算法和容器之间架设了桥梁，让算法能够在不知道容器类型的情况下对容器进行操作。

5. hash_set 并非 C++标准容器，因此不应在有移植性需求的应用程序中使用它。在这种情况下，应使用 std::map。

第 16 章

测验

1. std::basic_string <T>。

2. 将两个字符串复制到两个副本对象中，再将每个复制的字符串都转换为小写或大写。然后对转换后的字符串进行比较，并返回比较结果。

3. 否。C 风格字符串实际上是类似于字符数组的原始指针，而 STL string 是一个类，实现了各种运算符和成员函数，使字符串操作和处理尽可能简单。

练习

1. 该程序需要使用 std::reverse：

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

int main ()
{
    using namespace std;

    cout << "Please enter a word for palindrome-check:" << endl;
    string strInput;
    cin >> strInput;

    string strCopy (strInput);
    reverse (strCopy.begin (), strCopy.end ());

    if (strCopy == strInput)
        cout << strInput << " is a palindrome!" << endl;
    else
        cout << strInput << " is not a palindrome." << endl;

    return 0;
}
```

2. 使用 std::find：

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

using namespace std;

// Find the number of character 'chToFind' in string "strInput"
int GetNumCharacters (string& strInput, char chToFind)
{
    int nNumCharactersFound = 0;

    size_t nCharOffset = strInput.find (chToFind);
    while (nCharOffset != string::npos)
    {
        ++ nNumCharactersFound;

        nCharOffset = strInput.find (chToFind, nCharOffset + 1);
    }
    return nNumCharactersFound;
}

int main ()
{

    cout << "Please enter a string:" << endl << "> ";
    string strInput;
    getline (cin, strInput);

    int nNumVowels = GetNumCharacters (strInput, 'a');
    nNumVowels += GetNumCharacters (strInput, 'e');
    nNumVowels += GetNumCharacters (strInput, 'i');
    nNumVowels += GetNumCharacters (strInput, 'o');
    nNumVowels += GetNumCharacters (strInput, 'u');

    // DIY: handle capitals too..

    cout << "The number of vowels in that sentence is: " << nNumVowels;

    return 0;
}
```

3. 使用函数 toupper：

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

int main ()
{
    using namespace std;

    cout << "Please enter a string for case-conversion:" << endl;
    cout << "> ";

    string strInput;
    getline (cin, strInput);
    cout << endl;

    for ( size_t nCharIndex = 0
        ; nCharIndex < strInput.length ()
        ; nCharIndex += 2)
        strInput [nCharIndex] = toupper (strInput [nCharIndex]);
```

```
    cout << "The string converted to upper case is: " << endl;
    cout << strInput << endl << endl;

    return 0;
}
```

4. 可以这样编写程序：

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

int main ()
{
    using namespace std;

    const string str1 = "I";
    const string str2 = "Love";
    const string str3 = "STL";
    const string str4 = "String.";

    string strResult = str1 + " " + str2 + " " + str3 + " " + str4;

    cout << "The sentence reads:" << endl;
    cout << strResult;

    return 0;
}
```

第 17 章

测验

1. 否。仅当在 vector 末尾插入元素时所需的时间才是固定的。
2. 10 个。插入第 11 个元素，将导致重新分配内存。
3. 删除最后一个元素，即删除末尾的元素。
4. 类型 CMammal。
5. 通过下标运算符（[]）或函数 at()。
6. 随机访问迭代器。

练习

1. 下面是一种解决方案：

```
#include <vector>
#include <iostream>

using namespace std;

char DisplayOptions ()
{
    cout << "What would you like to do?" << endl;
    cout << "Select 1: To enter an integer" << endl;
    cout << "Select 2: Query a value given an index" << endl;
    cout << "Select 3: To display the vector" << endl << "> ";
    cout << "Select 4: To quit!" << endl << "> ";

    char ch;
    cin >> ch;

    return ch;
```

```
}

int main ()
{
    vector <int> vecData;

    char chUserChoice = '\0';
    while ((chUserChoice = DisplayOptions ()) != '4')
    {
    if (chUserChoice == '1')
        {
            cout << "Please enter an integer to be inserted: ";
            int nDataInput = 0;
            cin >> nDataInput;

            vecData.push_back (nDataInput);
        }
        else if (chUserChoice == '2')
        {
            cout << "Please enter an index between 0 and ";
            cout << (vecData.size () - 1) << ": ";
            int nIndex = 0;
            cin >> nIndex;

            if (nIndex < (vecData.size ()))
            {
                cout<<"Element ["<<nIndex<<"] = "<<vecData[nIndex];
                cout << endl;
            }
        }
        else if (chUserChoice == '3')
        {
            cout << "The contents of the vector are: ";
            for (size_t nIndex = 0; nIndex < vecData.size (); ++ nIndex)
                cout << vecData [nIndex] << ' ';
            cout << endl;
        }
    }
    return 0;
}
```

2. 使用 std::find 算法：

```
vector <int>::iterator iElementFound = std::find (vecData.begin (),
                                        vecData.end (), nDataInput);
```

3. 修改练习 1 的解决方案，以接受用户输入并显示 vector 的内容。

第 18 章

测验

1. 可将元素插入到 list 中间，可也将其插入到两端，插入位置不会影响性能。
2. list 的独特之处在于，这些操作不会导致现有迭代器失效。
3. mList.clear ();或 mList.erase (mList.begin(), mList.end());
4. 可以。insert 函数的一个重载版本可用于插入集合中特定范围内的元素。

练习

1. 这类似于第 17 章中练习 1 的解决方案，唯一需要修改的地方是使用 list 的 insert 函数，如下

所示：

```
mList.insert (mList.begin(),nDataInput);
```

2. 存储两个指向 list 元素的迭代器，使用 list 的 insert 函数在中间插入一个元素，然后使用这两个迭代器来演示在插入元素后它们仍指向以前的元素。

3. 下面是一种可能的解决方案：

```
#include <vector>
#include <list>
#include <iostream>

using namespace std;

int main ()
{
    vector <int> vecData (4);
    vecData [0] = 0;
    vecData [1] = 10;
    vecData [2] = 20;
    vecData [3] = 30;

    list <int> listIntegers;

    // Insert the contents of the vector into the beginning of the list
    listIntegers.insert (listIntegers.begin (),
                         vecData.begin (), vecData.end());

    cout << "The contents of the list are: ";

    list <int>::const_iterator iElement;
    for ( iElement = listIntegers.begin ()
        ; iElement != listIntegers.end ()
        ; ++ iElement)
        cout << *iElement << " ";

    return 0;
};
```

4. 下面是一种可能的解决方案：

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

using namespace std;

int main ()
{
    list <string> listNames;
    listNames.push_back ("Jack");
    listNames.push_back ("John");
    listNames.push_back ("Anna");
    listNames.push_back ("Skate");

    cout << "The contents of the list are: ";

    list <string>::const_iterator iElement;
    for (iElement = listNames.begin(); iElement!=listNames.end();
++iElement)
        cout << *iElement << " ";
    cout << endl;

    cout << "The contents after reversing are: ";
```

```
    listNames.reverse ();
    for (iElement = listNames.begin(); iElement!=listNames.end();
++iElement)
        cout << *iElement << " ";
    cout << endl;

    cout << "The contents after sorting are: ";
    listNames.sort ();
    for (iElement = listNames.begin(); iElement!=listNames.end();
++iElement)
        cout << *iElement << " ";
    cout << endl;

    return 0;
}
```

第 19 章

测验

1. 默认排序标准由 std::less<>指定，它使用运算符<来比较两个整数，并在第一个小于第二个时返回 true。

2. 重复的元素在一起，它们彼此相邻。

3. size()，所有 STL 容器都是。

练习

1. 该二元谓词可以是这样的：

```
struct FindContactGivenNumber
{
    bool operator () (const CContactItem& lsh, const CContactItem& rsh)
const
    {
        return (lsh.strPhoneNumber < rsh.strPhoneNumber);
    }
};
```

2. 该结构和 multiset 的定义如下：

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

using namespace std;

struct PAIR_WORD_MEANING
{
    string strWord;
    string strMeaning;

    PAIR_WORD_MEANING (const string& sWord, const string& sMeaning)
        : strWord (sWord), strMeaning (sMeaning) {}

    bool operator< (const PAIR_WORD_MEANING& pairAnotherWord) const
    {
        return (strWord < pairAnotherWord.strWord);
    }
};

int main ()
```

```
{
    multiset <PAIR_WORD_MEANING> msetDictionary;
    PAIR_WORD_MEANING word1 ("C++", "A programming language");
    PAIR_WORD_MEANING word2 ("Programmer", "A geek!");

    msetDictionary.insert (word1);
    msetDictionary.insert (word2);

    return 0;
}
```

3. 下面是一种解决方案：

```
#include <set>
#include <iostream>

using namespace std;

template <typename T>
void DisplayContent (const T& sequence)
{
    T::const_iterator iElement;

    for (iElement = sequence.begin(); iElement!=sequence.end(); ++iElement)
        cout << *iElement << " ";
}

int main ()
{
    multiset <int> msetIntegers;

    msetIntegers.insert (5);
    msetIntegers.insert (5);
    msetIntegers.insert (5);

    set <int> setIntegers;
    setIntegers.insert (5);
    setIntegers.insert (5);
    setIntegers.insert (5);

    cout << "Displaying the contents of the multiset: ";
    DisplayContent (msetIntegers);
    cout << endl;

    cout << "Displaying the contents of the set: ";
    DisplayContent (setIntegers);
    cout << endl;

    return 0;
}
```

第 20 章

测验

1. 默认排序标准由 std::less<>指定。
2. 彼此相邻。
3. size ( )。
4. 在 map 中找不到重复的元素！

练习

1. 可包含重复元素的关联容器，如std::multimap：

```
std::multimap <string, string> multimapPeopleNamesToNumbers;
```

2. 下面是一种解决方案：

```
struct fPredicate
{
    bool operator< (const wordProperty& lsh, const wordProperty& rsh) const
    {
        return (lsh.strWord < rsh. strWord);
}
};
```

3. 参考第19章中练习3的解决方案。

第21章

测验

1. 一元谓词。
2. 它可以显示数据或计算元素个数。
3. 在C++中，在应用程序运行阶段存在的所有实体都是对象，因此结构和类也可用作函数，这称为函数对象。注意，函数也可通过函数指针来调用，它们也是函数对象。

练习

1. 下面是一种解决方案：

```
template <typename elementType=int>
struct Double
{
    void operator () (const elementType element) const
    {
            cout << element * 2 << ' ';
    }
};
```

可以这样使用该一元谓词：

```
int main ()
{
        vector <int> vecIntegers;

        for (int nCount = 0; nCount < 10; ++ nCount)
                vecIntegers.push_back (nCount);

        cout << "Displaying the vector of integers: " << endl;

        // Display the array of integers
        for_each ( vecIntegers.begin ()        // Start of range
                 , vecIntegers.end ()              // End of range
                 , Double <> () ); // Unary function object

        return 0;
}
```

2. 添加一个整型成员，每次调用operator()时都递增该成员：

```
template <typename elementType=int>
struct Double
```

```
{
    int m_nUsageCount;

    // Constructor
    Double () : m_nUsageCount (0) {};

    void operator () (const elementType element) const
    {
        ++ m_nUsageCount;
            cout << element * 2 << ' ';
    }
};
```

3. 该二元谓词的定义如下：

```
template <typename elementType>
class CSortAscending
{
public:
    bool operator () (const elementType& num1,
                      const elementType& num2) const
    {
        return (num1 < num2);
    }
};
```

可以这样使用该谓词：

```
int main ()
{
    std::vector <int> vecIntegers;

    // Insert sample numbers: 100, 90... 20, 10
    for (int nSample = 10; nSample > 0; — nSample)
        vecIntegers.push_back (nSample * 10);

    std::sort ( vecIntegers.begin (), vecIntegers.end (),
                CSortAscending<int> () );

    for ( size_t nElementIndex = 0;
          nElementIndex < vecIntegers.size ();
          ++ nElementIndex )
          cout << vecIntegers [nElementIndex] << ' ';

    return 0;
}
```

第 22 章

测验

1. lambda 表达式总是以[]打头。
2. 通过捕获列表：[Var1, Var2, ...] (Type& param){...;}。
3. 像下面这样做：

```
[Var1, Var2, ...](Type& param) -> ReturnType { ...; }
```

练习

1. 该 lambda 表达式类似于下面这样：

```
sort(vecNumbers.begin(), vecNumbers.end(),
    [](int Num1, int Num2) {return (Num1 > Num2); } );
```

2. 该 lambda 表达式类似于下面这样：

```
cout << "Number do you wish to add to all elements: ";
int NumInput = 0;
cin >> NumInput;

for_each(vecNumbers.begin(), vecNumbers.end(),
        [=](int& element) {element += NumInput;} );
```

下面的示例演示了练习 1 和练习 2 的解决方案：

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

template <typename T>
void DisplayContents (const T& Input)
{
   for(auto iElement = Input.cbegin() // auto, cbegin and cend: c++11
      ; iElement != Input.cend ()
      ; ++ iElement )
      cout << *iElement << ' ';
   cout << endl;
}

int main()
{
   vector<int> vecNumbers;
   vecNumbers.push_back(25);
   vecNumbers.push_back(-5);
   vecNumbers.push_back(122);
   vecNumbers.push_back(2011);
   vecNumbers.push_back(-10001);
   DisplayContents(vecNumbers);

   sort(vecNumbers.begin(), vecNumbers.end());
   DisplayContents(vecNumbers);

   sort(vecNumbers.begin(), vecNumbers.end(),
      [](int Num1, int Num2) {return (Num1 > Num2); } );
   DisplayContents(vecNumbers);

   cout << "Number do you wish to add to all elements: ";
   int NumInput = 0;
   cin >> NumInput;

   for_each(vecNumbers.begin(), vecNumbers.end(),
      [=](int& element) {element += NumInput;} );

   DisplayContents(vecNumbers);

   return 0;
}
```

▼ 输出：

```
25 -5 122 2011 -10001
-10001 -5 25 122 2011
2011 122 25 -5 -10001
Number do you wish to add to all elements: 5
2016 127 30 0 -9996
```

第 23 章

测验

1. 使用函数 std::list::remove_if( )，因为它确保指向 list 中（未被删除的）元素的现有迭代器仍有效。
2. 如果没有显式指定谓词，list::sort( )（和 std::sort( )）将使用 std::less<>，这将使用运算符<对集合中的对象进行排序。
3. 对指定范围内的每个元素调用一次。
4. for_each( )返回函数对象。

练习

1. 下面是一种解决方案：

```
struct CaseInsensitiveCompare
{
    bool operator() (const string& str1, const string& str2) const
    {
        string str1Copy (str1), str2Copy (str2);

        transform (str1Copy.begin (),
                   str1Copy.end(), str1Copy.begin (), tolower);
        transform (str2Copy.begin (),
                   str2Copy.end(), str2Copy.begin (), tolower);

        return (str1Copy < str2Copy);
    }
};
```

2. 下面是一个演示程序。注意到 std::copy( )复制时无需知道集合的特征。它只使用迭代器类：

```
#include <vector>
#include <algorithm>
#include <list>
#include <string>
#include <iostream>

using namespace std;

int main ()
{
    list <string> listNames;
    listNames.push_back ("Jack");
    listNames.push_back ("John");
    listNames.push_back ("Anna");
    listNames.push_back ("Skate");

    vector <string> vecNames (4);
    copy (listNames.begin (), listNames.end (), vecNames.begin ());

    vector <string> ::const_iterator iNames;
    for (iNames = vecNames.begin (); iNames != vecNames.end (); ++ iNames)
        cout << *iNames << ' ';

    return 0;
}
```

3. std::sort( )与 std::stable_sort( )之间的区别在于，后者在排序时保持对象的相对位置不变。由于该应用程序需要按生成顺序存储数据，因此应使用 stable_sort( )，以保持天体事件的相对顺序不变。

第 24 章

测验

1. 可以，通过提供一个谓词。
2. 运算符<。
3. 不能，只能操作栈顶元素。因此，不能访问栈底的 Coins 对象。

练习

1. 该二元谓词可以是运算符<：

```
class Person
{
public:
    int Age;
    bool IsFemale;

    bool operator<  (const Person& anotherPerson) const
    {
        bool bRet = false;
        if (Age > anotherPerson.Age)
            bRet = true;
        else if (IsFemale && anotherPerson.IsFemale)
            bRet = true;

        return bRet;
    }
};
```

2. 只需将字符串依次插入到栈中。弹出数据时，字符串的排列顺序便反转了，因为栈是一种 LIFO 容器。

第 25 章

测验

1. 不能。bitset 可存储的位数在编译阶段就已确定。
2. 因为它不能像其他容器那样动态地调整长度，它也不像容器那样支持迭代器。
3. 不会。在这种情况下使用 std::bitset 最合适。

练习

1. std::bitset 支持实例化、初始化、显示和相加，如下所示：

```
#include <bitset>
#include <iostream>

int main()
{
    // Initialize the bitset to 1001
    std::bitset <4> fourBits (9);

    std::cout << "fourBits: " << fourBits << std::endl;

    // Initialize another bitset to 0010
    std::bitset <4> fourMoreBits (2);

    std::cout << "fourMoreBits: " << fourMoreBits << std::endl;
```

```
    std::bitset<4> addResult(fourBits.to_ulong() +
fourMoreBits.to_ulong());
    std::cout << "The result of the addition is: " << addResult;

    return 0;
}
```

2. 对前一个示例中的 bitset 对象调用函数 flip( ):

```
addResult.flip ();
std::cout << "The result of the flip is: " << addResult << std::endl;
```

第 26 章

测验

1. 我会先看看 www.boost.org，希望您也如此！

2. 不会。一般而言，如果智能指针编写得好（且选择正确）的话是不会的。

3. 如果是侵入式的，将由指针拥有的对象保存引用计数；否则，指针可将这种信息保存在自由存储区中的共享对象中。

4. 需要双向遍历链表，因此必须是双向链表。

练习

1. 语句 pObject->DoSomething ();有问题，因为指针在复制时失去了对对象的拥有权。这将导致程序崩溃（或发生令人非常不愉快的事情）。

2. 代码类似于下面这样：

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

class Fish
{
public:
   Fish() {cout << "Fish: Constructed!" << endl;}
   ~Fish() {cout << "Fish: Destructed!" << endl;}

   void Swim() const {cout << "Fish swims in water" << endl;}
};

class Carp: public Fish
{
};

void MakeFishSwim(const unique_ptr<Fish>& inFish)
{
   inFish->Swim();
}

int main ()
{
   unique_ptr<Fish> myCarp (new Carp); // note this
   MakeFishSwim(myCarp);

   return 0;
}
```

鉴于 MakeFishSwim( )接受的参数为引用，不会导致复制，因此不会出现切除问题。另外，请注意

变量 myCarp 的实例化语法。

3. unique_ptr 的复制构造函数和复制赋值运算符都是私有的，因此不允许复制和赋值。

第 27 章

测验

1. 在只需写入文件时，应使用 ofstream。
2. 使用 cin.getline( )。参见程序清单 27.7。
3. 不应该，因为 std::string 包含的是文本信息。您可使用默认模式，即文本模式。
4. 检查 open( )是否成功。

练习

1. 您打开了一个文件，但在使用流并关闭它之前，没有使用 is_open( )检查 open( )是否成功。
2. 您不能插入到 ifstream。ifstream 设计用于输入，而不是输出，因此不支持流插入运算符<<。

第 28 章

测验

1. 这是一个类，类似于其他类，但创建它旨在用作其他异常类（如 bad_alloc）的基类。
2. try 块是一组可能导致异常的语句。
3. catch 语句是一个例程，其参数指出了它能处理的异常类型。它位于 try 块后面，用于捕获 try 块内可能引发的异常。
4. 异常是一个对象，可包含在用户创建的类中能定义的任何信息。

练习

1. 绝不要在析构函数中引发异常。
2. 没有处理代码可能引发的异常，即缺少 try...catch 块：
3. 绝不要在 catch 块中分配内存，更不用说为一百万个整数分配内存了。如果 try 块内的代码分配内存失败，将导致恶性循环。

第 29 章

测验

1. 您的应用程序好像是在一个线程中执行所有的任务。因此，如果图像处理本身（调整对比度）是处理器密集型的，UI 将没有响应。应该将这两种操作放在两个线程中，这样操作系统将在这两个线程之间切换，向 UI 线程和执行对比度调整的工作线程提供处理器时间。
2. 可能没有妥善地同步线程。您的应用程序同时读写同一个对象，导致数据不一致。请使用一个二值信号量，在数据库表被访问时禁止对其进行修改。

附录 E
ASCII 码

计算机使用位和字节进行工作，位和字节表示的基本上是数字。为表示字符，制定了美国信息互换标准编码（ASCII）标准，这种标准已被广泛采用。ASCII 给拉丁字母 A～Z 和 a～z、数字 0～9 以及一些特殊键击（如 DEL）和特殊字符（如空格）指定了 7 位编码。

7 位可表示 128 种不同的值，其中前 32 个（0～31）被保留，用于表示与打印机等外设交互的控制字符。

E.1 可打印字符 ASCII 表

ASCII 编码 32～127 用于表示可打印字符，如 0～9、A～Z、a～z 以及其他一些字符（如空格）。下表列出了这些符号的 ASCII 编码对应的十进制值和十六进制值。

符号	十进制值	十六进制值	描述
	32	20	空格
!	33	21	惊叹号
“	34	22	双引号
#	35	23	井号
$	36	24	美元符号
%	37	25	百分比符号
&	38	26	和号
‘	39	27	单引号
(	40	28	左括号
)	41	29	右括号
*	42	2A	星号
+	43	2B	加号
,	44	2C	逗号
-	45	2D	连字符
.	46	2E	句点
/	47	2F	斜杠
0	48	30	零
1	49	31	一
2	50	32	二

续表

符号	十进制值	十六进制值	描述
3	51	33	三
4	52	34	四
5	53	35	五
6	54	36	六
7	55	37	七
8	56	38	八
9	57	39	九
:	58	3A	冒号
;	59	3B	分号
<	60	3C	小于号（或左尖括号）
=	61	3D	等于号
>	62	3E	大于号（或右尖括号）
?	63	3F	问号
@	64	40	At 符号
A	65	41	大写字母 A
B	66	42	大写字母 B
C	67	43	大写字母 C
D	68	44	大写字母 D
E	69	45	大写字母 E
F	70	46	大写字母 F
G	71	47	大写字母 G
H	72	48	大写字母 H
I	73	49	大写字母 I
J	74	4A	大写字母 J
K	75	4B	大写字母 K
L	76	4C	大写字母 L
M	77	4D	大写字母 M
N	78	4E	大写字母 N
O	79	4F	大写字母 O
P	80	50	大写字母 P
Q	81	51	大写字母 Q
R	82	52	大写字母 R
S	83	53	大写字母 S
T	84	54	大写字母 T
U	85	55	大写字母 U
V	86	56	大写字母 V
W	87	57	大写字母 W
X	88	58	大写字母 X
Y	89	59	大写字母 Y

续表

符号	十进制值	十六进制值	描述
Z	90	5A	大写字母Z
[	91	5B	左中括号
\	92	5C	反斜杠
]	93	5D	右中括号
^	94	5E	脱字符号
_	95	5F	下划线
`	96	60	重音符号
a	97	61	小写字母a
b	98	62	小写字母b
c	99	63	小写字母c
d	100	64	小写字母d
e	101	65	小写字母e
f	102	66	小写字母f
g	103	67	小写字母g
h	104	68	小写字母h
i	105	69	小写字母i
j	106	6A	小写字母j
k	107	6B	小写字母k
l	108	6C	小写字母l
m	109	6D	小写字母m
n	110	6E	小写字母n
o	111	6F	小写字母o
p	112	70	小写字母p
q	113	71	小写字母q
r	114	72	小写字母r
s	115	73	小写字母s
t	116	74	小写字母t
u	117	75	小写字母u
v	118	76	小写字母v
w	119	77	小写字母w
x	120	78	小写字母x
y	121	79	小写字母y
z	122	7A	小写字母z
{	123	7B	左大括号
|	124	7C	竖线
}	125	7D	右大括号
~	126	7E	波浪符号
	127	7F	DEL键