宝典丛书 200万

C++ 宝典

- 内容全面系统，涉及使用C++进行程序开发的大部分知识
- 讲解清晰易懂，结合示意图和示例让初学者快速理解概念
- 实例典型实用，以大量实例和开发技巧让读者体验实际编程，加深理解
- 注重综合应用，通过综合案例让读者了解C++的实际应用方法，提高开发水平

李鹏程 等编著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

C++宝典

李鹏程　等编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

C++是近年来广泛使用的程序设计语言，它在 C 语言的基础上发展而来，实现了由面向过程到面向对象的转变，全面支持面向对象的程序设计方法。

本书分为 5 部分，分别是 C++基础、面向对象编程、标准模板库、底层开发和综合案例。前 4 部分循序渐进地讲解了 C++程序的组成及开发过程、程序中的数据、语句和表达式、程序流程控制、数组与字符串、指针与引用、函数、函数模板、错误与异常处理、宏与预编译、面向对象基础、类的封装、重载操作符和自定义转换、类的继承、多继承和虚拟继承、多态、类模板、文件流、使用标准模板库 STL、序列式容器、关联式容器、函数对象和算法、名称空间、位操作和在 C++中嵌入汇编等知识。最后一部分介绍了两个应用 C++的典型综合案例——图书管理系统和学生管理系统。

本书涉及面广，从基本知识到高级内容和核心概念，再到综合案例，几乎涉及了 C++开发的所有重要知识。本书适合所有想全面学习 C++开发技术的人员阅读，也适合各种使用 C++进行开发的工程技术人员使用。对于经常使用 C++进行项目开发的人员来说，本书是一本不可多得的案头必备参考书。

图书在版编目（CIP）数据

C++宝典 / 李鹏程等编著. -- 北京 ： 电子工业出版社, 2010.5
（宝典丛书）
ISBN 978-7-121-10693-4

I. ①C… II. ①李… III. ①C 语言 - 程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2010)第 065288 号

责任编辑：董　英
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：39.75　　字数：1119 千字
印　　次：2010 年 5 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei.com.cn。
服务热线：(010) 88258888。

前　言

C++是近年来最流行、最广泛使用的程序设计语言之一。C++是在 C 语言的基础上发展而来的，并实现了由面向过程到面向对象的转变，全面支持面向对象的程序设计方法。C++语言在软件行业一直处于领先地位，在其他领域中也有着广泛的应用。同时，C++自身也在不断完善，其未来发展方向也很明确，那就是作为高性能软件开发的基础，在平台软件开发中发挥主要作用。

笔者长期从事 C++的开发工作，十分了解初学者在学习过程中可能遇到的一些问题和困惑。本书的目的是让初学者对 C++语言有一个全面系统的认识。为了让读者能够理解 C++开发的核心思想，本书在讲解的时候尽量结合笔者的独特理解和感受，使读者能够举一反三。此外，通过学习本书中的实例，读者还能为学习其他高级语言打下基础。

本书特色

1．内容全面系统，具有参考价值

作为 C++的宝典书，本书介绍了 C++语言的基础知识、面向对象编程、标准模块库、底层开发等多方面的内容，内容涉及面广，从基本知识到高级内容和核心概念，再到综合案例，几乎涉及了 C++开发的所有重要知识。

2．概念讲解形象贴切，适合初学者学习

本书针对 C++语言的特点，在讲解各种语言概念的时候，都给出了实际的开发例子，并尽量使

用图形化讲解，让初学者在第一次接触概念时就能够迅速掌握。

3．实例贴近实际，加深理解程度

本书在讲解知识点时，贯穿了大量有针对性的典型实例，并给出了对应的开发技巧，以便让读者更好地理解各种概念和方法，体验实际编程方法，加深理解程度。

4．综合实例讲解，提高应用水平

本书每章的结尾都讲解了针对本章内容的综合实例，介绍如何综合运用多种 C++知识。最后一部分还详细介绍了使用 C++开发数据库应用系统的全过程。通过这些综合实例，可以快速提高读者的 C++应用水平。

本书所用示例代码可到 www.broadview.com.cn 下载。

本书内容及体系结构

第 1 部分　C++基础（第 1 章～第 11 章）

本部分主要包括 C++概述、C++程序的组成及开发过程、程序中的数据、语句和表达式、程序流程控制、数组与字符串、指针与引用、函数、函数模板、错误与异常处理、宏与预编译等内容。通过本部分的学习，读者可以掌握 C++开发的基础流程和 C++编程的基本语法知识。

第 2 部分　面向对象编程（第 12 章～第 19 章）

本部分主要包括面向对象基础、类的封装、重载操作符和自定义转换、类的继承、多继承和虚拟继承、多态、类模板和文件流等内容。通过本部分的学习，读者可以掌握 C++面向对象编程的核心概念和常用方法。

第 3 部分　标准模板库（第 20 章～第 24 章）

本部分主要包括使用标准模板库 STL、序列式容器、关联式容器、函数对象和算法、名称空间

等内容。通过本部分的学习，读者可以掌握 C++中标准模板库 STL 的使用方法，以及与之相关的知识和方法。

第 4 部分 底层开发（第 25 章和第 26 章）

本部分主要包括位操作和在 C++中嵌入汇编等内容。通过本部分的学习，读者可以掌握 C++中关于底层开发的核心内容。

第 5 部分 综合案例（第 27 章和第 28 章）

本部分主要介绍两个完整的 C++应用系统的实现过程，即图书管理系统和学生管理系统。通过本部分的学习，读者可以学会如何全面应用前面章节所学的开发技术进行软件项目开发，达到可以独立开发项目的水平。

本书读者对象

- 没有任何编程语言学习经验的 C++语言初学者。
- 有志于成为 C++语言程序员的读者。
- 非计算机专业需要学习 C++语言的读者。
- 有程序语言基础或正在学习数据结构需要参考 C++语言的读者。

本书作者

本书主要由李鹏程编写，其他参与编写的人员有张金霞、于锋、张伟、曾广平、刘海峰、刘涛、赵宝永、郑莲华、张涛、杨强、陈涛、罗渊文、李居英、郭永胜。在此对所有参与编写的人员表示感谢！由于笔者水平有限，书中可能还存在疏漏和错误，还望广大读者批评指正。

目　　录

Part

第 1 部分　C++基础

第 1 章　走进 C++

本章包括

- ◆ 从 C 到 C++的发展历程
- ◆ 面向过程和面向对象软件开发方法
- ◆ 认识类与对象
- ◆ 面向对象编程的特点
- ◆ C++的应用现状及发展方向
- ◆ 学习 C++的一般方法

C++是近年来最流行、最广泛使用的程序设计语言之一。C++是在 C 语言的基础上发展而来的，并实现了由面向过程到面向对象的转变，全面支持面向对象的程序设计方法。Java 和 C#等程序设计语言的出现与流行，并没有取代 C++在软件行业中的领先地位。同时，C++自身也在不断完善，其未来发展方向也很明确，那就是作为高性能软件开发的基础，在平台软件开发中发挥主要作用。本章主要介绍 C++语言的发展历程、类与对象的基本概念、面向对象编程的特点、C++的应用现状及发展方向，以及学习 C++程序设计的一般方法。

1.1　初识 C++

C++是由 C 语言发展而来的，与 C 语言完全兼容。C++在 C 语言语法的基础上进行了扩充，最主要的就是引入了类，增加了面向对象机制，如继承、派生、多态等，从而实现了由面向过程向面向对象的转变。

1.1.1　从 C 到 C++

在 C 语言诞生以前，编写高效的系统软件主要采用汇编语言。用汇编语言编写的程序可直接编译为二进制码，不会像高级语言那样在编译过程中插入多余代码，其目标代码非常简洁、高效，对硬件的操控能力也很强。但是汇编语言过分依赖于计算机硬件，而不同 CPU 的汇编指令也不同，从而导致了汇编语言的可读性和可移植性都很差。

普通高级语言的可读性和可移植性虽然比汇编语言要好很多，但它又缺乏汇编语言那样的高效性和对底层硬件的强大操控能力。于是人们迫切需要一种中间语言，即一种介于汇编语言和高级语言之间的语言。这种语言既要有非常高的指令效率和强大的底层控制能力，又要具有高级语言的简洁性、易懂性和较好的可移植性。C 语言就是这样一种语言。

C 语言语法简洁灵活，程序书写自由，能够像汇编语言一样进行位、字节、地址操作，具有高级语言的简洁性，以及低级语言的高效性和对底层强大的控制能力；C 语言具有丰富的运算符和数据结构，能实现各种复杂数据结构的操作运算；C 语言引入了指针，使得程序能够直接访问物理内存地址，提高了程序的效率和灵活性。

20 世纪 80 年代，科研人员发明了 C++。一开始 C++是作为 C 语言的增强版出现的，从给 C 语言增加类开始，不断地增加新特性。虚函数（virtual function）、操作符重载（operator overloading）、多重继承（multiple inheritance）、模板（template）、异常（exception）、RTTI

（Run-Time Type Identification）、名称空间（namespace）逐渐被加入标准。

刚诞生的 C++和现在的版本有很大区别。首先，当时还没有真正的 C++编译器，C++代码都是先转化为 C 代码，然后用 C 编译器编译的；其次，那时的 C++没有继承，没有现在这么多关键字，也没有虚函数，虚函数是最后才被加入 C++的主要特性。

C++经过了多次演变和修订，每次逐步增加一些新的关键字和新特性，最后才变成现在的样子。从 C 到 C++语言的演化过程如图 1.1 所示。

图 1.1　从 C 到 C++语言的演化历程

虽然 C++与 C 语言有着很深的渊源，但 C++并不依赖于 C 语言。读者在学习时可以完全不学 C 语言，而直接学习 C++，在大多数场合下 C++完全可以取代 C 语言。

1.1.2　从面向过程到面向对象

面向过程是一种以事件为中心的编程思想，面向对象是一种以事物为中心的编程思想。传统的面向过程软件开发方法存在着以下几个问题：

- 重用性差。
- 难以维护。
- 难以适应需求变化较大的情形。

开发大型软件系统经常面临的一个问题是需求模糊或需求变化较大。传统的面向过程软件开发方法很难适应这种情况，从而导致了软件开发周期长、成本高。

之所以面向过程的开发方法容易导致上述问题，主要是因为其主要手段是功能分解。首先从系统高层入手，自顶向下，不断地把复杂的处理分解为子过程，这样一层一层地分解下去，直到仅剩下若干个容易实现的子过程为止。然后用相应的工具来描述底层的各个过程。因此，面向过程的开发方法实际上建立在底层的各个过程上，如图 1.2 所示。

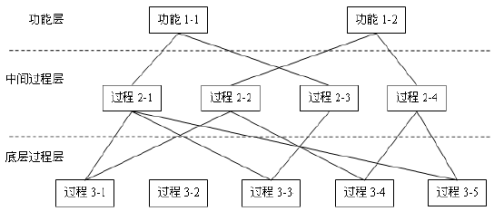


图 1.2　面向过程的软件结构

用户需求的变化大部分是针对功能的，这种变化对于基于过程的设计是灾难性的，一旦中间某个过程出现变更，后续过程都必须做变更，这种变更的结果是软件质量得不到保障，软件也无法按时交付使用。所以用面向过程的开发方法设计出来的系统结构常常是不稳定的，用户需求的变化往往会造成系统结构的较大变化，从而需要花费很大代价才能实现这种变化。

使用面向过程的方法开发的软件，当其底层的过程发生变化时，将影响到上层，原因就是其结构的层次性。这样带来的后果就是修改非常复杂。

面向对象（Object Oriented，OO）是当前计算机界关心的重点，它是20世纪90年代至今软件开发的主流方法。面向对象的概念和应用已超越了程序设计和软件开发，扩展到了一个很宽的范围，如数据库系统、交互式界面、应用结构、应用平台、分布式系统、网络管理结构、CAD技术、人工智能等领域。

面向对象是现实世界模型的自然延伸。现实世界中的任何实体都可以看做对象，对象之间通过消息相互作用。另外，现实世界中的任何实体都可归属于某类事物，任何对象都是某类事物的实例。如果说传统的面向过程的编程语言是以过程为中心、以算法为驱动的，面向对象的编程语言则是以对象为中心、以消息为驱动的。用公式表示，面向过程的编程语言为“程序=算法+数据”，面向对象的编程语言为“程序=对象+事件”。

1.1.3 认识类与对象

第一个面向对象的程序设计语言是Simula-67，它的出现是为了解决模拟问题。典型的模拟问题是银行出纳业务，包括出纳部门、顾客、业务、货币的单位等大量的“对象”。把那些在程序执行期间除了状态之外其他方面都一样的对象归纳在一起，就构成了对象的“类”，这就是“类”一词的来源。

类与对象的关系，就是数据类型与变量的关系。实际上，类就是开发者自定义的数据类型，而对象就是该类型的变量。

类描述了一组有相同特性（数据元素）和相同行为（函数）的对象。之所以要用类，是因为基本的数据类型难以描述现实领域中的各种事物。使用类就可以将基本的数据类型组合成复杂的数据对象。面向对象技术能很容易地将大量问题归纳成为一个简单的解，这一发现导致了大量面向对象程序设计（Object Oriented Programming，简称OOP）语言的产生，其中最著名的是Smalltalk——C++之前最成功的OOP语言。

抽象数据类型的创建是面向对象程序设计中的一个基本概念。抽象数据类型几乎能像内部类型一样准确地工作。程序员可以创建该类型的变量并操纵这些变量。对象可以是人们要进行研究的任何事物，从最简单的整数到复杂的飞机等均可以看做对象。对象不仅可以表示具体的事物，还可以表示抽象的规则、计划或事件。

例如，可以用一个对象来表示汽车。所有汽车都有这样一些属性，如颜色、车轮、引擎等。同时也有启动、停止、加速等一些动作。这样对象汽车就把它自己的一些共同属性（如颜色、轮子等）和方法（如启动、加速等）都组织在一起了，如图1.3所示。

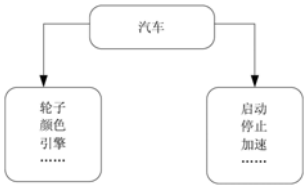


图 1.3 对象的属性和方法

面向对象编程用对象模拟实际事物，利用类把事物的属性和方法组织到一起，其中某一类的对象有一些共同特性和行为，这就是我们通常讲的封装。我们还可以利用继承关系从现有对象派生出新对象，并在新对象中增加独特的特性，实现对原有对象的扩充，如卡车类对象有汽车类的属性，但卡车的轮子数多、载重量大。

1.1.4 面向对象编程的特点

面向对象编程将对象的数据和方法封装成对象，对象的数据和方法是密切联系的。对象实现了信息隐藏，即在对象外部不能直接看到对象的数据和对象的方法是如何实现的。对象通过定义接口或者公有方法实现与外部通信，对象通常不知道其他对象的实现方法，因为实现细节隐藏在对象内部。就像我们可以很好地驾驶汽车而不需要知道发动机、传送系统和燃料系统内部如何工作一样，对象的使用者只需调用对象的接口，而不必关心具体的实现。

C++程序员需要重点考虑的是如何生成自己的用户自定义类型。每个类包含一组数据和一组操作数据的函数。类的数据组件称为数据成员，而类的函数组件称为成员函数。内部类型的实例称为变量，而用户自定义类型的实例称为对象。程序员用内部类型作为构造用户自定义类型的基本组件。C++中关注的重点是类而不是函数。系统中的名词帮助 C++语言程序员确定组类，由这些类生成系统的对象。对象的类就像房子的蓝图，我们可以用一张蓝图建造多个房子，同样也可以用一个类生成多个对象。

面向对象的思想已经涉及到软件开发的各个方面，如面向对象的分析（OOA，Object Oriented Analysis）、面向对象的设计（OOD，Object Oriented Design），以及我们经常说的面向对象的编程（OOP，Object Oriented Programming）。

1.2 C++的应用现状

现阶段，计算机程序设计语言种类繁多，比较流行的有 C++，C#和 Java。C#和 Java 由于晚于 C++出现，对 C++的不足之处进行了大量的改进，都很容易学习。采用 C#和 Java 做项目开发的速度要比采用 C++快很多，因此得到了大力的推广。但是，C++作为具有 20 多年历史的程序设计语言，有着大量的技术沉淀和大量的专业人才，这使得 C++在现代软件领域中仍占据着举足轻重的地位。C++的应用领域如图 1.4 所示。

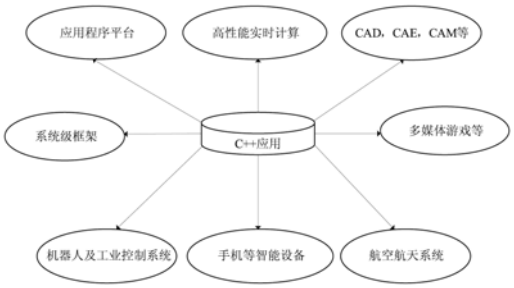


图 1.4　C++的应用领域

C++对于现在 OO（Object Oriented）思想的成熟以及在企业开发中的大量应用是功不可没的。但随着 1995 年网络时代的到来和 Java 语言的诞生，C++逐步进入了一个尴尬的境地。由于没有跨平台以及网络应用的优势，在企业级应用开发中 C++逐步被 Java 所替代。

C#与 Java 由于在库的支持下大大简化了网络应用程序的开发，使得越来越多的人开始学习和使用 C#与 Java。但是 C++在灵活性和效率方面仍然占有很大优势。在大型应用软件的开发过程中，一般底层开发优先选择 C++，这样可以使得底层平台有很高的效率，同时具有很大的灵活性。另外，在软件拓展、移植和维护上 C++也有很好的表现。

1.3　C++未来的发展

C++发源于 C，自 C++出现以来，在越来越多的场合逐渐取代了 C 语言的地位，取得了行业基础的地位。业界大量使用 C++写成的产品代码以及大量的 C++职业软件工程师就是最直接的证明。

C++在面向企业的软件开发中，在开发便捷性等方面的确要比 C#和 Java 差很多，其中一个主要问题是 C++语言没有强大的库的支持，而且由于 C++很灵活，要用好 C++需要长时间的积累，学习曲线比较陡峭。但是，C++在系统级复杂应用程序以及高性能、实时、并行和嵌入式领域占据着主导地位，同时在对灵活性和底层操作要求较高的软件开发中占据着绝对的优势。未来在一些由 C 或者 C++开发的传统软件中，C++依然是开发者的首选。

1.4　如何学习 C++

学习任何东西都必须从基础开始，但掌握 C++语法只是迈出了学习 C++的第一步。学习程序设计的一条最根本原则是“多练、多动手编写程序、多读源代码”。只有多编写代码才能达到熟练使用语言的目标，才能更深刻地理解语言的精髓。那么该怎样学习 C++呢？

- 全面了解 C++语言。
- 养成良好的编程习惯。
- 多写代码，多读代码。
- 坚持读 C++方面的著作。
- 掌握 C++基础后，尽量多参加 C++项目。
- 开阔视野，广泛涉猎。

希望读者在自己的学习实践过程中注意上面几点。

1.5 小结

C++作为当今最流行的程序设计语言之一，在软件开发领域有着广泛的应用，从复杂的操作系统到大型应用程序，从数据库系统到网络应用，都有着C++活跃的身影，体现着C++不可替代的地位与作用。

C++语言灵活，对底层控制能力强，程序运行效率高，在保证程序的执行速度同时，也尽可能地节省了内存空间，能充分发挥计算机硬件的性能。C++全面支持面向对象程序设计方法，符合现代软件工程思想，从语言层面为面向对象的软件设计思想提供了强有力的保障。

学习C++不是一件容易的事情，希望读者能够循序渐进、脚踏实地地打好基础。学好程序设计的根本在于多加练习，同时勤动手、多动脑也能够激发我们的学习热情。C++的学习资源非常丰富，读者一定要好好利用这些资源。只要坚持，就一定能够收获成功，一定能够在软件设计领域大展身手，让C++为我们各自的专业领域服务。

◆ ◆ ◆

第 2 章　C++程序的组成及开发过程

本章包括

- C++程序的开发过程
- 编译器和编译过程
- C++程序的调试方法
- C++程序的组成部分
- C++程序的开发环境
- 开发多文件程序的过程

C++程序由数据（常量、变量，包括对象）和函数组成。本书将分别对各个组成部分进行介绍。但是，如何将各个部分组织成一个完整的程序，才是本书的重点。本章从一个简单的程序开始，讲述一个完整程序的组成以及 C++程序的开发过程。

2.1　一般开发过程

C 和 C++程序的开发一般要经历 4 个过程：编辑、编译、链接、运行。

- 编辑：将程序的设计思想转换成代码。
- 编译：将代码转换成机器码。
- 链接：将程序编译后的各部分机器码和 C++库函数等链接成一个可执行程序。
- 运行：在计算机上执行上述可执行程序。

如果源代码中没有任何错误，那么开发可以一次完成。在实际开发中，任何程序多多少少总会存在一些问题。这些问题包括语法、逻辑方面的错误，也包括效率方面的损失。如果是语法方面的错误，则在程序编译过程中就会出错，编译器会给出错误提示。开发者可以根据这个错误提示修改源代码，然后重新编译。

链接的过程中也可能出错，其原因可能是程序中引用的某个函数或者某个变量不存在，或者程序中使用的静态链接库或动态链接库不存在。关于链接错误的讨论超出了本书的范畴，因此在这里不做更多的说明。

如果是逻辑方面的错误（比如程序的输出不是预期的结果），或者运行效率很低，则没有明显的错误提示。开发者要么根据自己的经验去纠正错误，要么利用调试器对程序进行调试，找到原因所在。因此，在大多数情况下，开发程序还包括一个调试的过程。

所谓调试，就是在程序运行过程中，监视相关源代码的执行情况。一般来讲，程序在编译之后会变成机器码，而机器码对于人来讲是非常难以阅读的。根据机器码找到运行过程中的错误非常困难，所以应当利用一个调试器，在程序运行过程中，监视可读性好的源代码的执行情况。

一旦错误被发现，就必须修正。这包括重新编辑源代码、重新编译、重新链接、重新运行程序以及重新调试。很多时候，这个过程会反复多次，直到最终程序运行无误，如图 2.1 所示。

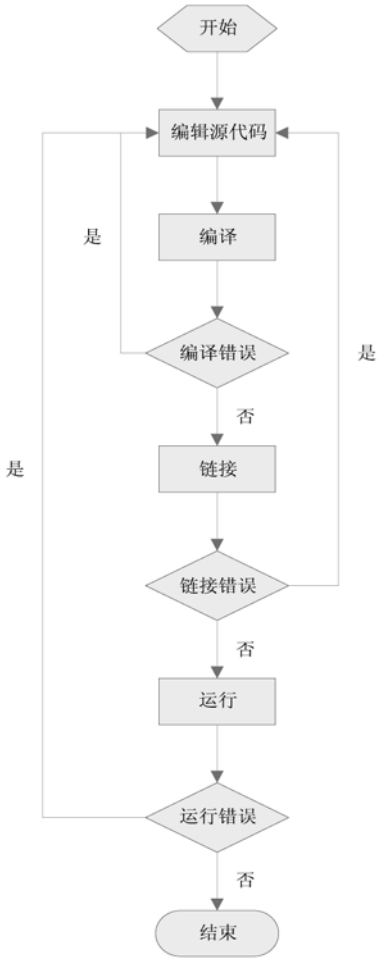


图 2.1 C++程序的一般开发过程

2.2 从简单程序开始

学习编程最好从简单的程序开始，虽然简单程序不能包含所有的开发技巧，但仍然可以从中学习到程序的基本组成和一般开发过程。这其中包括书写源代码、编译和链接。完成链接后，一个可执行程序就生成了，用户就可以在操作系统中运行它了(双击文件，或者在命令行窗口中输入命令)。下面就以一个简单的程序为例，来讲解开发程序的各个步骤。

2.2.1 书写源代码

C++程序是用类似自然语言的方式写成的文本，这个文本就是所谓的源代码。显然，要书写 C++

程序必须要有一个文本编辑器。读者可以根据自己的喜好选择编辑器，如 Windows 的记事本、DOS 的 Edit 命令、EMACS 和 Vim 等。当然如果读者已经熟悉了某种集成开发环境，如微软的 Visual Studio 系列，或者开源的 Dev-C++，那就更加方便了。

无论用什么编辑器书写源代码，最终保存的文件必须是简单的纯文本文件。而且文本中除了符合 C++语法的部分外，不能有任何其他的成分，否则会导致编译错误，以致不能生成可执行文件。

源代码文件简称为源文件，其扩展名一般是.cpp。但是 C++标准并没有对扩展名做出规定，不同的编译器可能会有不同的要求，甚至任何扩展名的文件都可以当做 C++源文件。本书中如没有特别说明，则源文件的扩展名都是.cpp。

编写求两个整数和的程序，如示例代码 2.1 所示。

示例代码 2.1

```
#include <cstdlib>                  // 包含头文件 cstdlib
#include <iostream>                 // 包含头文件 iostream

using namespace std;                // 使用名称空间 std
/* add：求两个数之和的函数
  输入：两个整数
  返回：上述两个整数的和
*/
int add( int x, int y ){            // 求两个整数之和的函数
    int z = x + y;                  // 计算参数 x 与 y 的和，并保存到变量 z 中
    return z;                       // 返回 z 的值
}

int main(int argc, char *argv[])    // 主函数
{
    int a = add( 1, 2 );            // 调用求和函数求（1+2），并将结果保存到变量 a 中
    cout<<"1 + 2 = "<<a<<endl;      // 输出计算结果
    system("PAUSE");                // 调用 system 函数，等待用户输入
    return EXIT_SUCCESS;            // 主函数返回
}
```

2.2.2 编译成目标文件

源代码书写完成，不代表一个程序开发结束，还需要对源代码进行编译和链接。编译的目的是将源代码转换成计算机可以辨认的二进制指令和数据，并保存到一个二进制文件中，即目标文件。目标文件通常以.obj 作为扩展名。编译成的目标文件仍然不是可执行程序，还需要链接器的处理。

2.2.3 链接成可执行程序

链接的作用是将目标文件与一个或多个库处理成一个可执行程序。库是可链接文件的集合，一般编译器都会提供一些库，包括 C++标准库，还有一些库可以购买得到，程序员也可以自己开发。

综上所述，创建一个可执行程序的步骤如下：

step 1 书写源代码，保存到源代码文件中，扩展名是.cpp。

step 2 将源代码文件编译成目标文件，扩展名是.obj。

step 3 将目标文件和各种必需的库链接成可执行程序。

2.2.4 运行程序

链接完成后，可执行文件就生成了，如何运行就是操作系统的事情了。但此时程序员的任务并没有结束，还需要检验程序是否运行正常，并且符合设计的要求。通常来讲，很少有程序能够一次运行成功，总会有一些意想不到的错误出现。此时就需要程序员返回去检查源代码，重新编译、链接、试运行，而这个过程也经常会反复多次才能结束。

编译示例代码 2.1，然后进行链接，完成后生成可执行程序。在操作系统中用命令行或者其他方法运行这个可执行程序，其运行结果如图 2.2 所示。

图 2.2 程序的运行结果

2.3 C++程序的组成

上一节中讲的是一个非常简单的程序，但其中已经包含了完整程序的各个组成部分，下面将一一进行简单介绍。

1. 预处理指令

程序前两行行首的字符“#”是一个预处理标志。程序在编译之前，先运行预处理器。预处理器遍历整个源代码，找到由“#”开始的行，并执行其后的预处理指令。include 是一条预处理指令，其含义是将后面的文件复制到当前源代码文件中。include 指令后面的文件，在 C 和 C++中习惯上称为头文件。文件名周围的尖括号“<>”表明这个文件是一个工程或标准头文件。预处理器首先从预定义的目录中开始查找，预定义的目录可通过环境变量和预处理器的命令行选项指定。如果文件名被双引号（""）包围，则从当前目录开始查找。

C++标准的头文件不带有扩展名。当然，C++同 C 一样，仍然可以指定头文件的扩展名为.h。有些编译器也允许使用别的扩展名。

2. 注释

在两个 include 指令之后，都有一行由双斜杠“//”引领的文本，这行文本称为“注释”。注释的内容不参与编译，因此可以用任何语言书写。注释从双斜杠开始，到本行结束。还有一种注释是由两组字符“/*”和“*/”包围的文本，这种注释同 C 语言中的注释一样，可以跨越多行。

3. 使用名称空间

程序的第 4 行（其上面为一空行）表明在当前文件中使用名称空间 std。所谓名称空间就是名称的搜索范围，如第 17 行的 cout，在示例代码中没有定义，编译器会到 std 名称空间中继续搜索。

4. 一般函数

在示例代码中，定义了 add 函数。程序中的函数同数学中的函数含义差不多，输入一些参数，函数就会返回一个值。不过在程序中，函数更加灵活，既可以不接受任何参数，也可以不返回任何值。而且在程序中定义函数的主要目的是实现某个功能，而不仅仅是返回一个值，这将在以后的章节中讨论。

add 函数由 4 部分组成：返回类型（最左面的 int）、函数名（add）、参数列表（int x, int y）和函数体。返回类型指的是函数返回值的类型。函数名和参数列表唯一标识了程序中的一个函数。函数体由一对花括号“{”和“}”包围起来，其中是实现函数功能的语句，即首先计算两个参数的和，然后返回这个和。return 是函数返回指令，其后跟的值就是函数的返回值。

5. 语句

在 C++程序中，最基本的组成单元是语句。一个语句由一个分号结束。在代码中，每个由分号结束的句子都是一条语句。如程序中的 using 语句，变量定义和初始化语句，函数返回语句，函数调用、定义变量并用函数调用的结果初始化变量的语句等。

6. 变量

示例程序中的“z”是一个变量。所谓变量就是由一个名称标识的一段内存，这段内存用来保存数据，并且数据的值是可变的。不同的变量，其内存大小也不一样，从一个字节到多个字节不等，取决于变量的类型。示例中 “z”的类型是 int，即整型，代表类似-1，0，123 这样的整数。

7. 主函数

每个程序只有一个主函数，即 main 函数。main 函数是整个程序的入口函数，所有的 C 和 C++程序都是从 main 函数开始运行的。同一般函数一样，main 函数也由 4 部分组成：返回类型、函数名、参数列表和函数体。标准 C++中对前 3 部分都有明确的规定，正如示例程序中第 14 行一样。值得注意的是，有的编译器要求 main 函数返回 void 型，这个已经不符合 C++标准了。

8. 调用函数

在主函数的程序代码中，调用了前面定义的函数 add，求 1+2，并将结果保存到变量 a 中。调用函数时只要在函数名后面加上参数即可。参数用小括号“(”和“)”包围起来，紧接在函数名之后即可。“int a =”的作用是声明一个变量（保存数据的实体），并将函数调用的返回值赋给这个变量。int 是变量的类型，这里应同函数的返回类型保持一致。

9. 使用 cout 对象

在示例代码中，使用 cout 对象输出了一条信息到命令行窗口中。所谓对象，简而言之，就是自定义数据类型的变量。而 cout 对象是 C++标准库中提供的一个对象，用于输出数据。与之相对应，还有一个用于输入的对象 cin。这些都将在后面的章节中详细介绍。为了使用这两个对象，必须包含头文件 iostream。

紧接在 cout 对象后面的是输出重定向运算符“<<”, 其后的所有内容都将输出到命令行窗口中，如“1 + 2 = ”。cout 对象可以连续使用多个“<<”运算符，从而一次性输出多组数据，如后面的变量 a 和最后的 endl，endl 表示换行，即在输出变量 a 后另起一行。

10. 使用宏

在示例代码中，主函数的返回值“EXIT_SUCCESS”实际上是个宏。这个宏是在头文件 cstdlib 中定义的：

```
#define EXIT_SUCCESS    0
```

这是一个预处理指令，其含义是：在书写源代码时，可以用“EXIT_SUCCESS”代替“0”。当预处理器处理源文件时，会将文件中的所有“EXIT_SUCCESS”都替换成“0”。这样当开始编译时，第 19 行的实际代码是“return 0;”。

2.4 注释

注释是这样一种文本，其本身不参与编译，而只是帮助别人理解程序。如果程序比较简单，那么理解起来就很简单。如果程序比较复杂，别人在看程序时就会觉得难以理解。即便是程序员自己，在过了一段时间后，也有可能忘记当初为什么要写这些。

2.4.1 注释的类型

C++的注释有两种类型：双斜杠型（从“//”开始的一行）和斜杠星型（包含在“/*”和“*/”之间）。双斜杠型注释是 C++引入的新的注释类型。这种注释从“//”开始，直到本行结束，双斜杠型注释不能跨行。斜杠星型注释则从“/*”开始，直到“*/”结束，可以跨越多行，这种注释是从 C 语言继承来的。

一般来讲，程序中使用双斜杠型注释就足够了。只有当注释的内容比较多时（10 行以上），才有必要使用斜杠星型的注释。

斜杠星型的注释不能嵌套，例如下述的注释：

```
/*这种注释不能嵌套，
    /*注释在这里结束
    */
*/
```

注释在第 3 行就会结束。因为由“/*”开始的注释，在遇到的第一个“*/”处结束。第 4 行的“*/”由于没有对应的“/*”，所以会被当做错误的语法，将导致编译错误。

2.4.2 使用注释的注意事项

用户在程序中使用注释的时候，需要注意下面的内容。

- 简单的代码不必注释：如果代码足够简单，那么代码就可以说明问题，不必再加上多余的注释。
- 注释要说明设计的原因，而不是描述代码：阅读代码的人更加关心的是程序为什么要这么设计，而不是代码的功能。
- 修改代码时，记得修改注释：很多人在修改代码时，往往会忘记修改注释，这样注释就成了过时的，甚至是错误的注释，当别人阅读这样的代码时，很容易被误导。

2.5 标准 IO 对象

C++标准库提供了两个对象，分别用于数据的输入（Input）和输出（Output）。输出对象 cout 在前面已经介绍过了，现在来看一下输入对象 cin。同 cout 一样，要使用 cin 对象，必须先包含头

文件 iostream：

```
#include <iostream>
```

使用 cin 对象的语法如下：

```
cin>>var;
```

其含义是从键盘输入一个数据，并保存到变量 var 中。

C++的标准 IO 对象 cin 和 cout 同 C 语言的 scanf 函数和 printf 函数功能类似，但其功能更加强大。在使用 C++中的 IO 对象时不需要指定数据的类型，而使用 C 语言的 IO 函数必须指定。

下面给出一个程序提示用户输入整数，并将其保存到一个变量中，然后再输出，如示例代码 2.2 所示。

示例代码 2.2

```
#include <cstdlib>                      // 包含头文件 cstdlib
#include <iostream>                     // 包含头文件 iostream

using namespace std;                    // 使用名称空间 std

int main(int argc, char *argv[])        // 主函数
{
    int var = 0;                        // 保存输入数据的变量
    cout<<"请输入一个整数：";             // 提示用户输入
    cin>>var;                           // 将用户输入的数据保存到变量 var 中
    cout<<"刚才输入的数是："<<var<<endl;  // 输出用户刚才输入的数据
    cout<<endl;                         // 输出一个空行
    system("PAUSE");                    // 调用函数 system，等待用户反应
    return EXIT_SUCCESS;                // 主函数返回
}
```

编译并链接上述源代码，运行生成的可执行程序，其结果如图 2.3 所示。

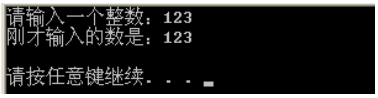


图 2.3　输入和输出数据结果

在上述源代码中，保存输入数据的变量是 int 型的，即整型的。如果用户输入的数据不是整型的，而是别的类型的，如 abc，则不会改变变量的值。

2.6 使用名称空间

名称空间（namespace）是包含各种名称（变量、函数、类等的标识符）的域。使用名称空间的主要目的是避免名称冲突。随着应用程序规模越来越大，各种变量、函数、类等的标识符也越来越多，添加一个新的标识符很容易与已经存在的标识符冲突。如果使用名称空间，则某个标识符就被限制在特定的名称空间中，从而减少了发生冲突的可能性。

如果要使用名称空间中的变量、函数、类等，需要指明其所在的名称空间。指明名称空间的方法有两种：一种是在标识符前面加上名称空间名和域运算符“::”，例如 std::cin（std 是标准 C++名称空间）；另外一种是在使用标识符之前，使用 using namespace 加上名称空间名，这样以后用该名称空间中的标识符时就不用再指明了。

标准 C++有一个 std 名称空间，所有 C++标准库中的标识符都在这个名称空间中。例如标准 IO 对象 cin 和 cout 就定义在这个名称空间中：

```
namespace std{
    ……
    ostream cin;
    ostream cout;
    ……
}
```

前面介绍的简单程序中就使用了标准名称空间 std：

```
using namespace std;
```

如果不加入上述语句，在使用 cout 对象时，必须这样书写：

```
std::cout<<……
```

其中“std::”的含义是在名称空间 std 中查找名称 cout。如果程序比较短小，像这样使用 cout 对象也没有什么不可以的。但是一旦程序比较长，而且多次用到 cout 对象，如果每次都要在前面加上“std::”，显然比较烦琐。

2.7 编译器和编译过程

C++编译器是一种系统软件，用来将源代码转换成机器码。现在比较著名的编译器有 Microsoft 的 CL、Borland 的 BCC、Intel 的 ICC，以及开源的、跨平台的 GCC。虽然 C++标准是唯一的，但各个编译器的实现未必一致。尽管如此，各种编译器的组成以及编译过程还是基本一致的。

因为不同种类计算机的硬件平台未必相同，所以一种编译器一般是针对一种或者几种平台设计的。为某种平台编译的程序，一般不能在另外一种平台上运行，例如针对 32 位计算机编译的程序，一般不能在 64 位计算机上运行。

一般来讲，C++编译器由预处理器、词法分析器、语法分析器、语义分析器、代码优化器以及机器码生成器组成，其组成结构如图 2.4 所示。

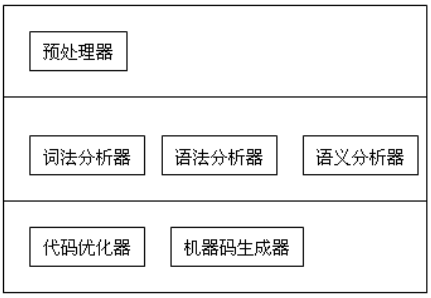


图 2.4　编译器的组成结构

预处理器的作用是根据源代码中的预处理指令，对程序文本进行处理。词法分析器、语法分析器和语义分析器合称编译器前端。其中词法分析器的作用是从左至右逐个字符地对源程序进行扫描，用其中由字符组成的单词产生一个个的单词符号，把作为字符串的源程序改造成为单词符号串的中间程序。例如对于“aVar = bVar + cVar ;”语句，词法分析的目标就是要找出 aVar，bVar，cVar，以及“=”和“+”这些符号，并结合符号表对其进行管理，上面语句进行词法分析之后的符号表如图 2.5 所示。

符号名	符号值
v1	aVar
o1	=
v2	bVar
o2	+
v3	cVar

图 2.5　词法分析之后的符号表

语法分析器以上述中间程序作为输入，分析其中的单词符号串是否符合语法规则。语法分析器首先判断各种单词符号所属的类型，如表达式、赋值、循环等。然后判断这些符号是否符合语法规则，再按照语法规则将每条语句构造成语法树或其他结构。上面例句的语法树如图 2.6 所示。

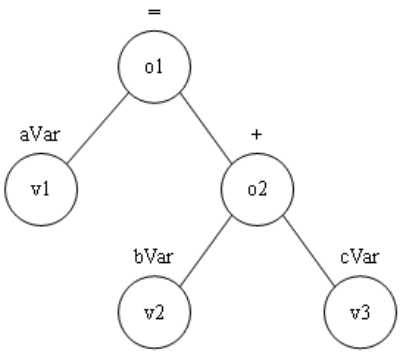


图 2.6　语法树

语义分析器根据语义规则对语法树中的语法单元进行静态语义检查，如类型检查和转换等，其目的在于保证语法正确的结构在语义上也是合法的。例如，对于上图的语法树，右侧的子树表现出来的语法结构是两个操作数相加，语义分析器的任务就是判断这两个操作数能否相加，需不需要进

行类型转换等。

语法分析和语义分析通常是交互进行的。每完成一步语法分析，就进行语义分析。如果需要进行类型转换等操作，就在上述语法树中插入一些操作，构造新的语法树，然后再进行语义分析。在这个过程中，如果有不符合语法、语义的代码，编译器就会报错。

代码优化器和机器码生成器合称编译器后端。优化是编译器的一个重要组成部分。由于编译器将源程序翻译成中间代码的工作是机械的、按固定模式进行的，因此，生成的中间代码往往在时间和空间上有很大浪费。当需要生成高效目标代码时，就必须进行优化。目标代码的优化主要包括以下三个方面：

- 生成较短的目标代码。
- 充分利用计算机中的寄存器，减少目标代码访问存储单元的次数。
- 充分利用计算机指令系统的特点，以提高目标代码的质量。

目标代码生成器把语法分析后或优化后的中间代码变换成目标代码。目标代码有以下三种形式：

- 可以立即执行的机器语言代码。
- 待装配的机器语言模块，由链接程序将其和其他模块链接起来，转换成能执行的机器语言代码。
- 汇编语言代码，必须经过汇编程序汇编后，才能成为可执行的机器语言代码。

2.8 选择集成开发环境

在程序开发过程的各个阶段中要用不同的工具，例如编写源代码时要用文本编辑器、编译时要用编译器、链接时要用链接器。早期的程序员在开发时经常在上述几种工具之间来回切换，非常烦琐，现在开发程序则不必这么麻烦，利用集成开发环境即可完成所有工作，极大地提高了编程的效率。

集成开发环境（Integrated Developing Environment，简称 IDE）是一种应用程序。该类程序提供图形化的用户界面工具，并在其中集成文本编辑器、编译器、链接器以及调试器，方便开发者查找程序中的错误。通过集成开发环境，开发者可以进行代码编写、软件分析、编译链接以及调试等工作，不必在各种工具之间进行切换，使用起来非常方便。在 C++程序开发中，有很多 IDE 可以使用，例如微软的 Visual Studio 系列、Borland 的 C++ Builder 以及开源的 Dev-C++等。

2.9 Dev-C++简介

Dev-C++是免费的开源集成开发环境。在本书成稿时，Dev-C++的最新版本是 4.9.9.2，本书将以该版本为例讲解 Dev-C++的安装和使用。相比其他 IDE 软件，Dev-C++的一个突出优点就是免费，这对于初学编程的人来讲非常实用。另外，Dev-C++使用 GCC 编译器，比较符合 C++标准，有利于初学者掌握正确的概念。

Dev-C++包括多页面窗口、工程编辑器以及调试器等，提供高亮度语法显示，方便开发者书写和阅读程序。Dev-C++有完善的调试功能，方便查找程序中的错误。利用 Dev-C++来学习 C 或者 C++是个不错的选择。

2.9.1 安装

下面将详细讲解 Dev-C++的安装过程。

step 1 双击 Dev-C++的安装程序，选择安装语言，如图 2.7 所示。

step 2 选择要安装的组件，如图 2.8 所示。如果选择了“Associate C and C++ files to Dev-C++”组件，可以将 Dev-C++与 C/C++程序源文件关联起来，包括*.h，*.c 和*.cpp 文件。安装完成后，通过双击上述类型文件的图标，就可以打开 Dev-C++编辑文件。

图 2.7 选择安装语言

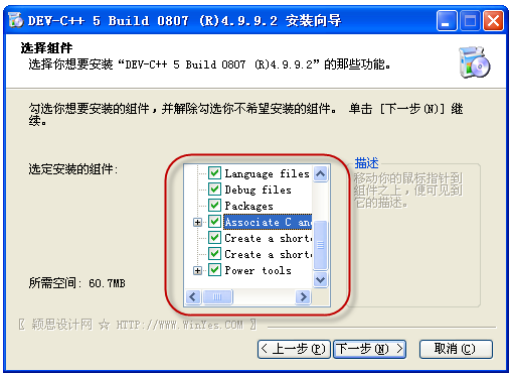


图 2.8 选择要安装的组件

这里省略了一些安装步骤，只显示关键部分。另外，Dev-C++的各个版本在安装时具体过程也可能略有不同，以实际过程为准。

step 3 在安装向导对话框中选择程序的安装路径，然后单击“下一步”按钮，开始安装，如图 2.9 所示。

step 4 安装完成后，在“环境选项”对话框中可以设置工作环境，如选择界面的语言，如图 2.10 所示。

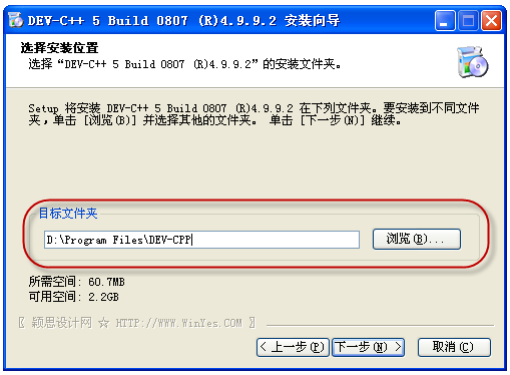


图 2.9 选择安装路径

图 2.10 选择界面语言

2.9.2 建立工程

在大多数集成开发环境里开发程序的第一步就是要建立工程，Dev-C++也是如此。所谓工程就是所有软件相关文件的集合。一个工程还可以保存开发者的一些设置，如编译选项、调试信息等。下面详细讲解建立工程的步骤。

一个工程可以用来维护多个源文件之间的关系，并可以编译成一个可执行程序或其他类型的软件，如动态链接库等。

step 1 创建工程。选择“文件”→“新建”→“工程”命令，创建新的工程，如图 2.11 所示。

step 2 选择工程类型。在“新工程”对话框中，选择工程类型，如图 2.12 所示。

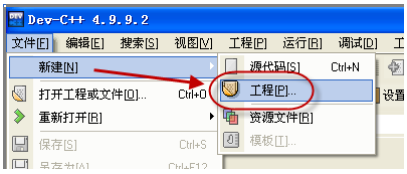


图 2.11 建立工程

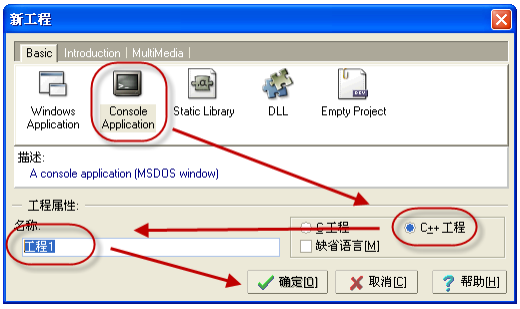


图 2.12 选择工程类型

step 3 单击对话框中的“确定”按钮，在弹出的对话框中选择工程的保存路径，如图 2.13 所示。

step 4 工程建立之后，Dev-C++会自动为工程加入源文件 main.cpp，并且该源文件中已经写好主函数，不过该文件还没有保存。开发者单击工具栏中的“保存”按钮，或开始编译，Dev-C++会弹出一个保存对话框，如图 2.14 所示。

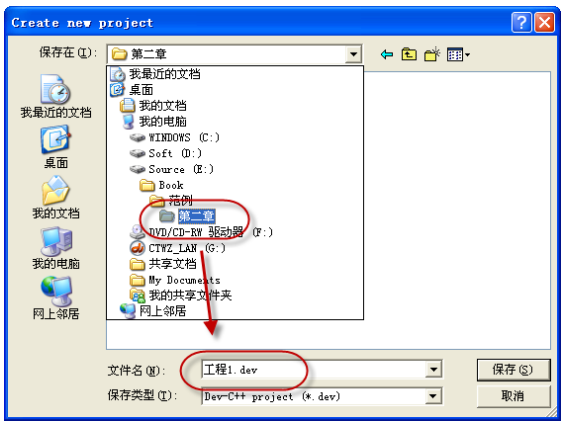


图 2.13 选择保存工程的位置

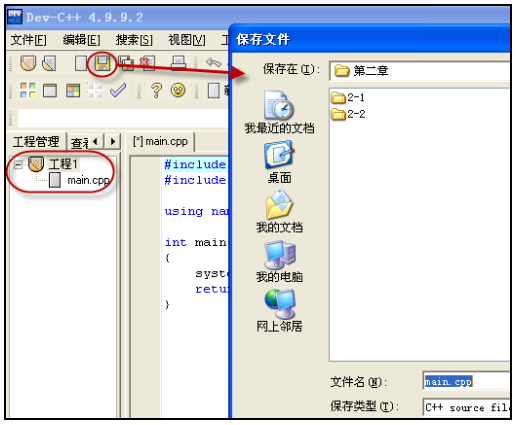


图 2.14 保存文件

2.9.3 编译和运行

在工程中加入文件之后就可以开始编程了。如果要进行多文件编程，可以继续向工程中添加文件，可以通过单击新建源代码窗口工具栏中的按钮实现，如图 2.15 所示。

加入的新文件也需要保存，其过程同保存工程默认的文件一样。添加新文件并书写代码之后，就可以开始编译了。其过程很简单，只需选择相应的菜单命令或者单击工具栏中的按钮即可，如图 2.16 所示。

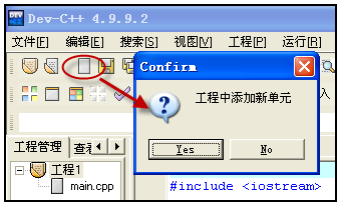


图 2.15　加入新文件

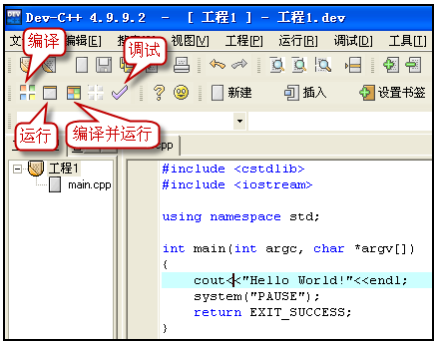


图 2.16　编译、运行以及调试按钮

工具栏中的“编译并运行”按钮将“编译”和“运行”两个动作合为一体了，开发者也可以通过直接按快捷键“F9”来激活这个动作。如图 2.16 所示例程的运行结果如图 2.17 所示。

图 2.17　运行结果

2.10　程序的调试

如果代码能够通过编译和链接，并且生成了可执行程序，那么它就可以在计算机上运行了。但是程序的运行并不一定是一帆风顺的，很有可能存在逻辑错误或者效率方面的问题。此时就应当通过调试来查找导致错误的原因。

调试的英文为 Debug。Bug 是虫子、害虫的意思，这里是指计算机程序中的错误。Debug 是消灭害虫的意思，而在计算机中就是消灭错误的意思。当然，消灭错误首先要找到错误的原因，然后才能进行修正。如果修正后还有错误，则继续调试。

2.10.1　调试的基本过程

在程序的使用过程中，一旦有错误报告，开发人员就要对程序进行调试，并最终消灭错误。一个完整的调试过程一般要经历 4 个步骤。

step 1　错误（Bug）重现。

step 2 查找原因。

step 3 修正错误。

step 4 验证。

如果最后一步验证不成功，则从第二步重新开始，直至最后消灭错误。调试过程如图 2.18 所示。

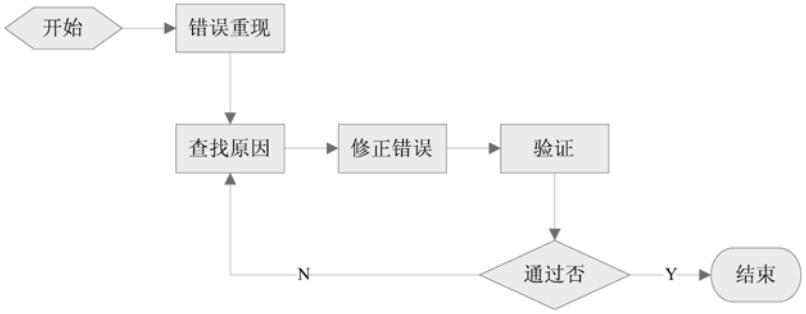


图 2.18 调试过程

错误重现，指的是根据用户或者测试人员的描述，设定条件，重复导致错误的步骤，使得错误出现在被调试的系统中，以方便开发人员进行调试。查找原因，即综合利用各种调试工具，使用各种调试手段，寻找导致错误的根源。通常测试人员报告的是软件错误所表现出的外在症状，如界面或执行结果中所表现出的异常，或者与软件需求和功能规约不符的地方。而这些外在症状总是由一个或多个内在因素所导致的，要么是由于代码的缺陷造成的，要么是某些设置造成的。

查找原因是调试过程中最关键的步骤，也是最耗时的步骤。因此，平常所说的调试，如不加以说明，一般指的就是查找原因的过程。

修正错误，即针对上述步骤查找到的错误原因，修改代码或者设置。代码修改之后，对于编译型语言或者混合型语言，还要对代码进行编译，并进行必要的配置。验证，即按照用户或者测试人员的描述，重复导致错误的步骤，检验一下修正之后错误是否还在。如果通过验证，则开发人员应当报告问题已解决，并说明解决方案。如果没有通过验证，则应当继续查找原因，修正错误，并进行验证。这个过程有时要重复进行，直至问题解决。

2.10.2 调试手段

这里所说的调试手段指的是查找原因的各种方法，包括设置断点、运行到断点、单步执行、查看变量等。通过 IDE 工具中的调试器可以在程序运行过程中，在源代码中要考察的位置设置断点，并通过

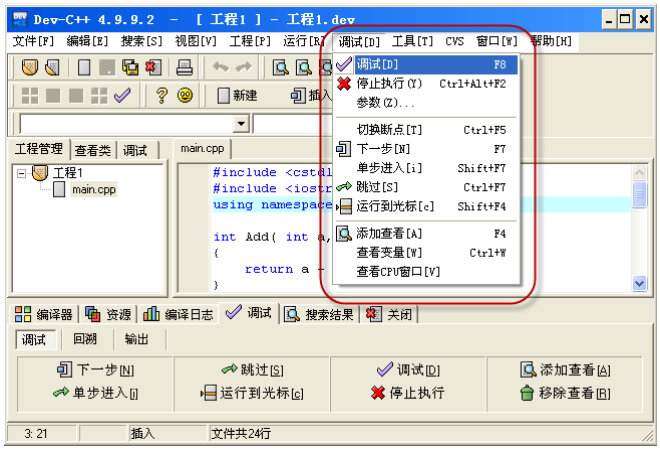


图 2.19 Dev-C++的“调试”菜单

调试命令直接运行到该处，然后通过单步执行和查看变量的方式监视每条语句的执行情况。

下面将通过 Dev-C++这个 IDE 软件讲解上述各种调试方法。在 Dev-C++的菜单栏中有一个“调试”菜单，其中集成了各种调试命令，如图 2.19 所示。同时，也可以利用窗口下面的调试工具栏进行操作。

各种调试命令的功能如表 2.1 所示。

表 2.1　各种调试命令的功能

调试命令	快捷方式	功能
调试	F8	以调试方式启动程序，运行到第一个断点处或需要用户输入的地方
停止执行	Ctrl+Alt+F2	停止调试
参数		传递给程序主函数的参数
切换断点	Ctrl+F5	在当前光标所在行设置或取消断点
下一步	F7	仅执行下一条语句
单步进入	Shift+F7	进入当前函数，并停止在函数的第一条语句之前
跳过	Ctrl+F7	跳出当前函数，并执行后面的语句，直至下一个断点或程序结束
运行到光标	Shift+F4	运行程序，直至当前光标所在位置的语句
添加查看	F4	查看表达式的值，该表达式不一定是程序中的代码
查看变量	Ctrl+W	查看当前所选变量的值
查看 CPU 窗口		查看 CPU 中各个寄存器的值以及当前程序的汇编代码

在程序以调试模式开始运行之前，首先应当设置断点。断点的功能是停止继续执行其所在处的语句以及后面的语句，但并不是结束程序，而是等待调试者的进一步操作。调试者可以通过“下一步”命令逐条运行语句，或者查看程序的当前状态。

在开始调试之前，在哪里设置断点是调试人员首先应当考虑的问题。通常，根据用户或测试人员对错误的描述，可以估计出大概的出错地方，调试的断点就应当设置在相应的地方。

当程序暂停执行之后，就可以通过“添加查看”、“查看变量”等命令检查程序的当前状态。如果检查的结果与期望的值不同，那么在此之前运行过的语句很可能就是导致错误的原因。通过进一步分析，或者设置新的断点重新调试，不断尝试，最终可以找到原因所在。

2.10.3 调试实例

下面的程序是一个有问题的程序。其原本的目的是求两个数之和，并输出结果值。但从实际运行的情况来看，其结果值存在误差。本节就以此程序为例，讲解程序的调试过程。

```
#include <cstdlib>
#include <iostream>
```

```
using namespace std;                    // 使用名称空间 std

int Add(int a, int b)                   // 求和函数
{
   return a - b;
}

int main(int argc, char *argv[])        // 主函数
{
   int x = Add( 1, 2 );                 // 调用函数 Add，求 1 和 2 的和

   cout<< x <<endl;                     // 输出结果值

   system("PAUSE");                     // 暂停程序的执行
   return EXIT_SUCCESS;
}
```

下面详细讲解调试的步骤。

step 1 既然是求和的结果不对，那么一定是求和函数 Add 的问题。所以，应当在调用 Add 函数的地方设置断点，如图 2.20 所示。

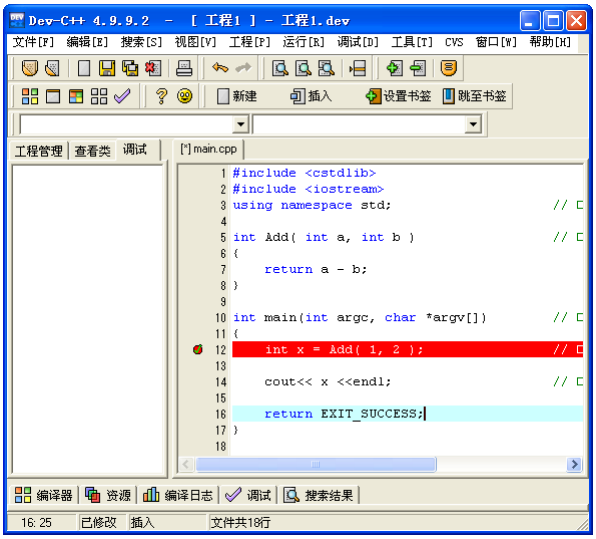


图 2.20 设置断点

step 2 断点设置好之后，就应当以调试的方式运行程序。程序从主函数开始执行，直至运行到第一个断点处停止执行，如图 2.21 所示。

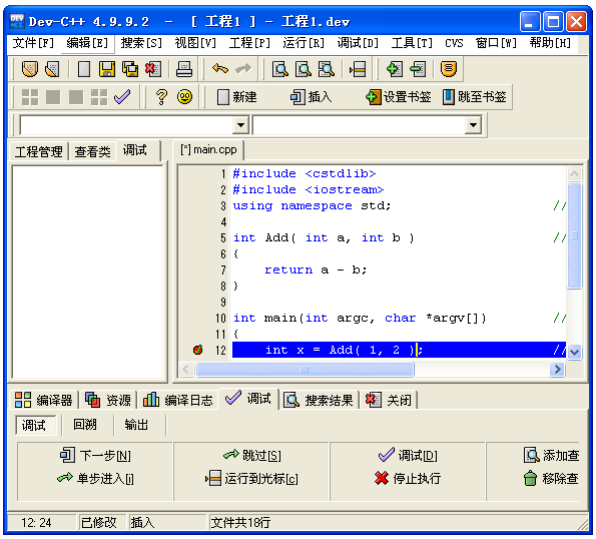


图 2.21　执行到断点处

step 3　断点由红色变成蓝色，表示程序暂停在该断点处。该处的语句是调用函数 Add，通过“单步进入”命令进入到该函数内部，以监视函数 Add 的运行情况，如图 2.22 所示。

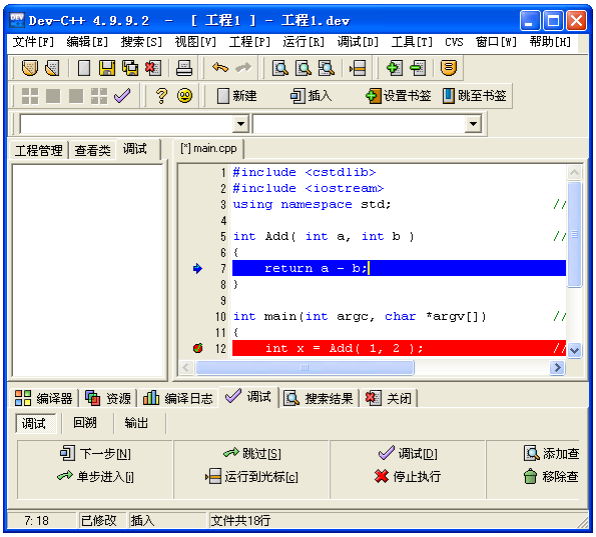


图 2.22　单步进入函数

此时如果使用“下一步”命令，则会执行第 13 行语句，而不是进入函数内部。

step 4　程序进入到函数 Add 内部之后，暂停在函数的第一条语句处。此时可以通过“添加查看”命令来查看程序的运行状态，如图 2.23 所示。

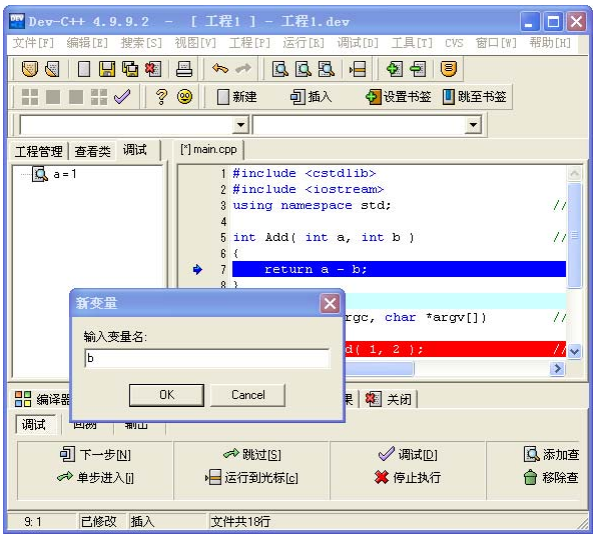


图 2.23 查看参数的值

step 5 有经验的调试人员可以很容易地察觉程序的错误，即本来应当是“return a + b;”的语句错写成了“return a - b;”。因此，只要将其中的减号改成加号即可。不过，从调试的角度来看，还可以通过查看表达式来找到错误的原因。在第 7 行，函数 Add 返回表达式“a - b”。为了查看该表达式的结果，可以通过“添加查看”命令，来查看该表达式的值，如图 2.24 所示。

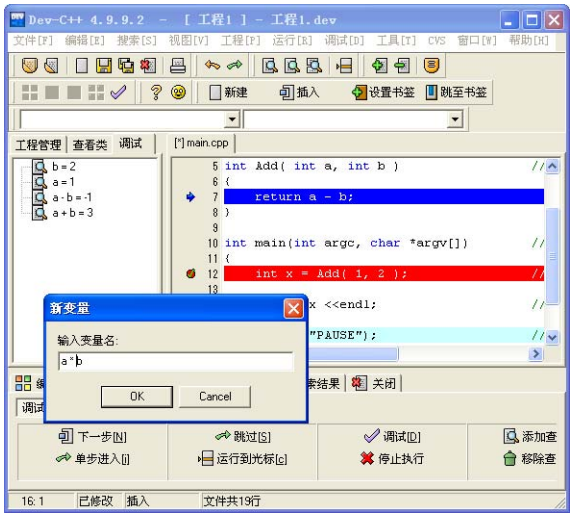


图 2.24 查看表达式的值

step 6 通过执行几次“下一步”命令，一直运行到第 14 行。此时可以查看变量 x 的值。在查看值的窗口中，可以通过右键快捷菜单选择“修改数据”命令，手工修改变量的值，如图 2.25 所示。

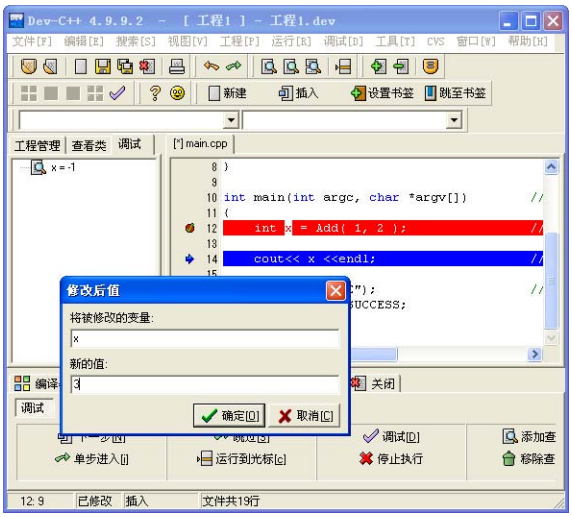


图 2.25　修改变量的值

step 7　将上述 x 变量的值修改为 3，然后通过“跳过”命令运行完当前的主函数，其输出结果如图 2.26 所示。

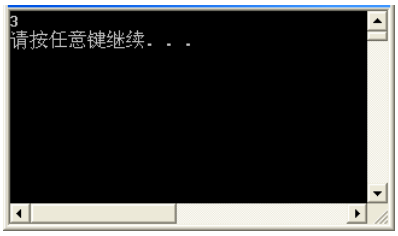


图 2.26　输出结果

至此，调试结束。开发人员应当根据上述查看结果，修改程序，并重新运行程序进行验证。

2.11　综合实例

通常在一个实际的工程中都会有多个源文件，本节就来练习建立一个多文件的程序。本实例建立一个具有两个源文件的程序，并在主程序中调用另外一个源文件中的函数，该函数计算两个参数之和，并返回结果值。建立 C++工程如图 2.27 所示。

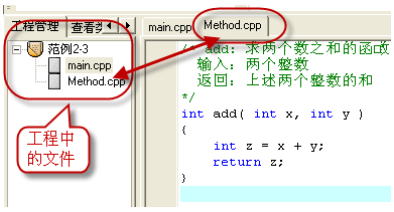


图 2.27　多文件工程

程序如示例代码 2.3 所示。

示例代码 2.3

```
//////////////////////////////////////////////////////////////////////
// main.cpp
#include <cstdlib>
#include <iostream>

using namespace std;                          // 使用名称空间

int add(int x, int y);                        // 函数声明

int main(int argc, char *argv[])              // 主函数
{
    cout<<"——多文件程序——"<<endl;
    cout<<"3 + 4 = ";                         // 输出提示信息
    cout<< add( 3,4 ) << endl;                // 输出函数调用结果，并换行

   system("PAUSE");                           // 等待用户反应
   return EXIT_SUCCESS;                       // 主函数返回
}
//////////////////////////////////////////////////////////////////////
// Method.cpp

/* add：求两个数之和的函数
  输入：两个整数
  返回：上述两个整数的和
*/
int add(int x, int y)                         // 求两个整数之和的函数
{
    int z = x + y;                            // 计算参数 x 与 y 的和，并保存到变量 z 中
    return z;                                 // 返回 z 的值
}
```

读者可建立一个控制台工程以及相应的源文件，并将上述代码复制到文件中，然后编译并运行，结果如图 2.28 所示。

图 2.28 多文件程序的运行结果

2.12 小结

本章主要介绍了 C++程序的必要组成部分，以及开发一个程序的主要步骤，并简单介绍了一下开发 C++程序的集成开发环境。另外，本章最后按照实际的开发过程，列举了一个具有多个源文件的程序实例。在后面的章节中，还将进一步详细介绍 C++程序的各个组成部分。

◆ ◆ ◆

第 3 章　程序中的数据

本章包括

- 常量和变量的概念
- 变量的定义、初始化和赋值
- const 常量的使用
- C++中的数据类型
- 结构体、共用体、枚举体
- 使用宏替换字面常量

处理数据是程序的主要目的，包括输入原始数据、对数据进行各种计算、输出处理的结果。一个什么都不做的程序是没有意义的。当初人们发明计算机，开始编写第一个程序的目的，就是为了计算炮弹的轨迹。当然，现在程序中数据处理的范围已经非常广泛了，从财务统计到航天飞机轨道计算，从计算机图像处理到卫星遥测，无处不见数据处理的身影。本章的重点是理解 C++中的各种数据类型，以及用来表示数据的常量和变量的用法。

3.1　常量和变量

现实世界中的事物有很多信息，比如声音、图像、个数、名称等。这些信息在计算机中的表达就是数据。不过，由于计算机本身的限制，不可能完整、清楚地表达整个信息，而且实际上这么做也是不必要的。在程序中只需将代表信息特征的、需要程序处理的部分抽取出来即可。程序中的数据有常量和变量之分，下面将一一介绍。

实际上计算机只能处理二进制数，即由 0 和 1 组成的数字，就连计算机本身的指令也是由这样的二进制数表示的。所以，所有计算机能够处理的数据在计算机内部都是二进制数。为了方便人们理解，并提高程序开发的效率，像 C++这样的高级语言都会按照自己的方式去定义和组织数据，并使之以符合人类思维的形式出现。

3.1.1　什么是常量

顾名思义，常量的值是不能改变的。C++中的常量有三种：字面常量、符号常量和枚举型常量。这里只介绍字面常量。源代码中出现的数值，如 1，2.3，a，Hello World 等，称为字面常量。之所以称为“字面”，是因为只能用其值来指代，称之为“常量”，是因为其值不能被改变。例如 123 这个常量，其值只能是 123，而不能被改成 456。

字面常量没有名字，要使用一个字面常量就只能通过常量的值。在程序中字面常量的功能是用来初始化变量、给变量赋值、参与表达式计算等。

字面常量是有类型的，这是由其值决定的。例如字面常量 1 和 2.3 就是两个不同类型的常量，一个是整数，另一个是小数。另外还有表示单个字符的常量，如 a。以及表示一串字符的常量，如 Hello World。不同类型的常量，其能够接受的处理方法也不相同，例如 1 和 2.3 可以进行四则运算，而 Hello World 就不行。

为了区分不同类型的常量，在 C++中有数据类型的概念，这将在 3.2 节中进行讲述。

3.1.2 什么是变量

变量，顾名思义，其值是可以改变的。在 C++程序中，变量就是用名称标明的一块儿内存，其中可以存储数据。在程序的运行过程中，变量的值会发生改变。假设 a 是变量，则其典型的使用过程如下：

```
a = 1;          // 用一个字面常量给变量 a 赋值
a = 1 + 2;      // 计算表达式（1+2 这个算式是一个表达式），并将结果赋给 a
a – 1;          // 变量 a 参与表达式计算
2 * a;          // 同上
```

变量有名称，正如上述例子所示，在程序中操作一个变量是通过变量名实现的。变量名不是任意的，必须符合一定的命名规则：变量名只能由字母、数字以及下画线组成，并且必须以字母或下画线开头，字母包括小写字母 a~z 以及大写字母 A~Z，数字是 0~9。下面给出几个变量名：

```
abc             // 正确的标识符
_def            // 正确的标识符
9gh             // 错误的标识符
```

在 C++中，变量名是一种标识符。标识符是对程序中实体名称的统称。这些实体包括符号常量、枚举常量、变量、函数、类、结构体、模板等。标识符的命名规则同变量的命名规则是一样的，即只能由字母、数字和下画线组成，并且不能以数字开头。

变量名区分大小写，如 Abc 和 abc 就是两个不同的变量名，代表两个不同的变量。C++语言本身对变量名的长度没有限制，但从阅读和理解程序的角度来讲，变量名不应该过长。命名变量时应该以易记、意思明确为原则，即容易记住，并且能够清晰地表达编程者的意图。如 birthDate，sum 和 idNum 就是很好的变量名，而 dfds34 和 really_long_variable_name_not_ok 就不是好的变量名。

变量的标识符在其作用域内是唯一的。关于变量的作用域在后文中将有详细的介绍，这里只要保证标识符不重复就可以了。变量除了有名称、值这两个特性之外，还有一个类型属性。同常量的类型一样，变量的类型也是为了区别不同类型的数据，用不同的方法进行处理。

3.1.3 定义变量

要在程序中使用变量，必须先定义变量。在程序中定义变量的目的是让程序分配一块儿内存，并为其命名。这个名字就是变量的标识符。与 C 语言不同，C++可以随时定义所需的变量，而不必放在函数的开始处。定义变量时，先指定变量的类型，再给出变量名，并以分号";"结束，格式如下：

```
类型 变量名;
```

以分号结束变量的定义，表示变量的定义也是一个语句。其中的"类型"就是这个变量所能处理的数据的类型。为了表示各种数据类型，C++语言设立了多个关键字，或者说是类型名，例如：

- short，int 和 long 表示不同长度的整型数。这里的长度指的是变量所占内存的大小，不同的编译器有不同的实现。一般在 32 位计算机中，short 表示 2 个字节的整数，int 表示 4 个字节的整数，long 表示 4 个字节（或 8 个字节，一般是 4 个字节）的整数。
- char 表示字符型数据。在 32 位计算机中一般是一个字节。除了 char，表示字符型数据的还有 wchar_t，不过 wchar_t 表示的是宽字符。一个宽字符在 32 位计算机中一般占 2 个字节。
- float，double 和 long double 分别表示单精度、双精度和长双精度的浮点型数据，其长度分别是 4 字节、8 字节和 16 字节。

由于数据类型众多，而上述关键字远远满足不了程序对数据类型的需求，所以 C++也允许用户自定义数据类型。C++中的类就是这样一种用户自定义类型，另外还有结构体、共用体、枚举体等。典型的变量定义如下：

```
int     a;      // 定义一个 int 型变量 a
char    b;      // 定义一个 char 型变量 b
float   c;      // 定义一个 float 型变量 c
double  d;      // 定义一个 double 型变量 d
Point   pt;     // 定义一个 Point 型变量 pt，Point 是用户自定义的变量类型，不是 C++关键字
```

C++允许在一个语句中定义多个变量，但必须是同一类型的，例如：

```
int a, b, c = 2;
double x = 0.0, y = 1.2;
char m, n;
```

所谓关键字就是 C++语言的保留字，用户不能更改其含义，只能按其规定的用途使用。例如上述的各种类型名就是关键字，主函数名 main 也是关键字，表示函数返回的 return 也是关键字。关键字不能用做标识符，否则会引起编译错误。

变量也是有生命周期的，每个变量的生命周期都是从其定义时开始。也就是说，一个变量只有被定义了，才能开始使用。至于变量的生命周期什么时候结束，则在以后的章节中进行讲述。不过，可以肯定的一点是，当程序结束时，变量的生命周期也就结束了，不能保留到下次程序运行。

3.1.4 初始化变量

无论开发者是否指定，变量在定义后都会有一个初始值。如果不指定，则这个值就是一个未定

义的值。所谓未定义，就是C++标准并没有规定具体的数值，而是由编译器根据需要自行指定。但这个值是没有意义的，开发者在编写程序时不能依赖这个值。

变量的初始化可以在定义变量时进行，只需在变量名后加上等号“=”和一个初始化值即可。这个初始化值可以是一个常量，也可以是一个变量。初始化变量的语法如下：

```
类型 变量名 = 初始化值;
```

其含义是定义一个变量，并将该变量初始化为等号右边的值。

例如，定义一个整型变量，并将其初始化为123，语句如下：

```
int  a = 123;
```

其他类型变量的初始化也是一样的，例如：

```
char        c = 'c';
wchar_t     w = L'w';
double      d = 1.23;
```

变量也可以用已经存在的变量进行初始化，例如：

```
int a = 123;
int b = a;
```

如果变量定义时没有初始化，则其值取决于编译器的实现。这样的值是没有意义的，所以使用未经初始化的变量比较危险，例如：

```
int  sum;                // 定义一个变量，用来保存1到100的和。警告：sum没有初始化！
for( int i=1; i<=100; i++ )     // 将后面花括号“{”和“}”中的语句重复100遍
{
    sum += i;            // 将当前i的值加到变量sum上
}
cout<< sum <<endl
```

上述代码的本意是求1到100的和，但打印出来的值并不是期望的5050，而是一个只有编译器才能确定的值。

C++的语法允许在一个语句里定义多个变量，也允许在一个语句里初始化多个变量，语法如下：

```
类型 变量名1 = 值1, 变量名2 = 值2, ……, 变量名n = 值n;
```

其含义是依次定义n个变量，并分别初始化每个变量，例如：

```
int a = 1, b = 2, c = 3;         // 依次定义并初始化变量a，b，c
```

另外，C++也支持链式初始化，即用等号“=”连接多个变量名，并在最后一个变量名后面加上一个等号和一个初始化值，这样就可以将全部变量都初始化成一个值，其语法如下：

```
类型 变量名1 = 变量名2 =……= 变量名n = 初始化值;
```

例如：

```
int a = b = c = 123;
double d = e = f = 1.23;
```

注意，链式初始化的过程是从右往左的，即先定义最右面的变量，并将其初始化为等号右边的值，然后依次向左定义并初始化各个变量。

3.1.5 为变量赋值

既然是变量，则在定义之后，其值就是可以修改的。修改变量值的过程就是赋值，其语法同初始化语法类似，只是没有类型关键字，即：

```
变量 = 值;
```

其中等号左边的变量是用变量名标识的，等号右边的值是一个常量或者已经存在的变量。这里的等号称为赋值运算符，其作用就是将等号右边的值赋给等号左边的变量，例如：

```
a = 345;        // 将 345 赋给整型变量 a
c = 'd';        // 将'd'赋给字符型变量 c
w = L'x';       // 将宽字符'x'赋给宽字符型变量 w
d = 4.56;       // 将 4.56 赋给双精度浮点型变量 d
```

为变量赋值时不仅可以用字面常量，也可以用已经定义好的变量，例如：

```
int a = 123;
int b = a;      // 将变量 b 初始化为 a 的值，运算后，b 的值为 123
a = b;          // 将变量 b 的值赋给 a
```

虽然都是使用等号“=”，但初始化和赋值的含义是不一样的。初始化是给未曾使用的变量设定一个值，而赋值则是修改已经在使用的变量的值；初始化只发生一次，即在变量定义时，而赋值则可以发生多次。

另外，为变量赋值也支持链式赋值，即为一系列的变量赋以同样的值。其语法同变量的链式初始化类似，只是没有开始的类型关键字，即：

```
变量名 1 = 值 1, 变量名 2 = 值 2, ……, 变量名 n = 值 n;
```

例如，将整型变量 a，b，c 和 d 都赋值为 123：

```
a = b = c = d = 123;
```

链式赋值的顺序同链式初始化的顺序相同，都是从右向左的，即先给最右面的变量赋值，然后依次向左给每个变量赋值。

变量的值，无论是初始化的值，还是后来赋的值，其类型都应当与变量的类型相同，或者符合转换规则。我们可以将整数转换为浮点型数，浮点型数也可以转换为整数（存在精度损失），后文中将会详细介绍这个转换规则，这里只要保持类型相同即可。

3.2 数据类型

C++是一种强类型的语言，也就是说程序中用到的数据都是某种类型的数据，不存在某个数据没有类型的情况。C++标准定义了一些常用的数据类型，下面将一一进行讲述。另外，C++也支持

用户自定义类型。

3.2.1 整型

表示整数的基本类型有 int（整型）、short int（短整型，int 可省略）、long int（长整型，int 可省略）以及 long long int（长长整型，int 可省略），分别表示不同长度的整数值。除非特别说明，上述各种整数的类型都可以称为整型。

各种整型字面常量如下：

```
123          // 整型或短整形
456L         // 长整型数，带有后缀“L”
789l         // 长整型数，带有后缀“l”（小写的 L）
901LL        // 长长整型数，带有后缀“LL”
234ll        // 长长整型数，带有后缀“ll”
```

在默认情况下，整型字面常量被当做一个 int 型值。如果在字面常量后面加一个“L”或“l”字母，即 L 的大写形式或者小写形式，则将其指定为 long 型。一般情况下应该避免使用小写字母，因为它很容易被误当做数字“1”。带有两个 L 后缀（“LL”或者“ll”）的是 long long 型数。注意，长长整型数的后缀，大小写不可混用，如 123Ll 就是一个错误的写法。

short 型的字面常量没有后缀，其形式同 int 型的字面常量一致，编译器会根据需要将这个字面常量解释为 short 型或者 int 型。

各种整型的字面常量可以被写成十进制、八进制或者十六进制的形式，例如十进制的 24 可以写成下面几种形式中的任意一种：

```
24      // 十进制
030     // 八进制，带有前缀“0”
0xF8    // 十六进制，带有前缀“0x”
0XF8    // 十六进制，带有前缀“0X”
```

如果一个字面常量在程序中写成上述形式，则编译器会自动将其当做整型数对待。在整型字面常量前面加一个“0”，该值将被解释成一个八进制数，而在前面加一个“0x”或“0X”，则会使一个整型字面常量被解释成十六进制数。定义并初始化各种整型变量的例子如下：

```
short   a = 123;        // 短整型变量 a，与“short int a;”等价
int     b = 456;        // 整型变量 b
long    c = 789L;       // 长整型变量 c，与“long int c;”等价
int     e = 024;        // 整型变量 e，初始化为八进制的 024，即十进制的 20
long    f = 0x1A;       // 长整型变量 f，初始化为十六进制的 0x1A，即十进制的 26
```

3.2.2 特殊整型

实际上在 C++标准中，并没有具体规定某种类型数据所占内存的大小，而只是规定了其大小与机器字的关系。所谓机器字的大小就是计算机的位数，例如 16 位计算机的机器字是 16 位，2 个字

节；32 位计算机的机器字是 32 位，4 个字节。

C++标准规定 short 的大小是半个机器字，int 的大小是一个机器字，而 long 的大小是一到两个机器字。这就存在一个程序移植方面的问题：如果两个计算机的机器字大小不一样（比如一个是 32 位，而另一个是 64 位），则在定义、初始化、赋值时对整型变量的操作就不一样，从而导致这两台计算机的程序不可以互相移植。

为了解决这个问题，C++标准引入了一些特殊类型关键字用来表示确定大小的整型数。如__int8，__int16，__int32 和__int64 分别表示 8 位、16 位、32 位和 64 位的整数，使用这些类型定义变量的语法同 int 型一样，如：

```
__int8  a = 12;
__int16 b = 35;
__int32 c = 215;
__int64 d = 4234;
```

3.2.3 无符号整型

整数分为无符号整数和有符号整数。之所以要区分有符号和无符号整数，是因为有时需要使用负数，有时则不需要。例如表示温度时应该用有符号数，而计数时则只用无符号数就够了。

如果要明确指明某个类型是无符号类型，则需要在该类型符前面加上 unsigned 关键字，例如无符号 int 型就是 unsigned int，无符号 long 型就是 unsigned long。有符号整型可以在类型符前面加上 signed 关键字，但一般可以省略，即 signed int 和 int 表示的都是有符号整型。无符号整型字面常量可以在一般字面常量后面加上后缀“u”或“U”来表示，如：

```
123u，345LU，57uL
```

定义、初始化和赋值无符号变量的方法如下：

```
unsigned int x = 123u;
unsigned long y = 546LU;
```

3.2.4 浮点型

小数在计算机中称为浮点型数，其数据类型有 float（单精度浮点数）、double（双精度浮点数）和 long double（长双精度浮点数，或称扩展精度浮点数）。各种浮点数的字面常量如下：

```
1.23f       // 单精度浮点数，带有后缀“f”
1.23F       // 单精度浮点数，带有后缀“F”
4.56        // 双精度浮点数，不带有任何后缀
7.          // 双精度浮点数，省略小数点后面的 0
8.9L        // 长双精度浮点数，带有后缀“L”
```

没有后缀的浮点型字面常量是 double 型的。单精度字面常量带有后缀“f”或“F”。类似地，长双

精度字面常量带有后缀“l”或“L”。

浮点型字面常量的后缀“f”、“F”、“l”、“L”只能用在十进制形式中。

浮点型字面常量可以被写成普通的十进制形式，或者用科学计数法表示。科学计数法是指数值中带有指数的表示方式，如：

```
3e2     // 表示 3*10^2
1.2E-2  // 表示 1.2*10^-2
```

其中指数的底数是 10，用 e 或 E 表示，后面的数值是 10 的次数。使用浮点型定义变量的语法如下：

```
float       a = 1.23f;
double      b = 3.45;
long double c = 5.72L;
```

3.2.5 字符型

字符的数据类型是 char，通常用来表示单个字符或小整数。字符型字面常量用两个单引号('')包围的字符表示，如：

```
'a'     // 字符常量 a
'1'     // 字符常量 1
'+'     // 字符常量+
' '     // 字符常量空格
```

字符型数据的定义、初始化和赋值方法如下：

```
char  c;            // 定义一个字符型变量
c = 'A';            // 为字符型变量赋值
char  d = 'D';      // 定义并初始化一个字符型变量
```

字符型数据和整型数有一定的转换关系。计算机不能直接存储字符，所以所有字符都是用数字编码来表示和处理的。最早的编码是 ASCII 码，例如'a'的 ASCII 码是 97，'b'的 ASCII 码是 98，'+'的 ASCII 码是 43。如果一个字符被当做整数使用，则其值就是对应的 ASCII 码。如果一个整数被当做字符使用，则该字符就是这个整数在 ASCII 码表中对应的字符。

一个整型变量可以用一个字符初始化，也可以用一个字符赋值；反之，一个字符变量也可以用一个整数进行初始化和赋值，如：

```
int  x = 'a';           // 定义一个整型变量，并用字符'a'初始化
cout<< x << endl;       // 输出的结果是 97，因为'a'的 ASCII 码是 97
```

```
x = 'A';                // 将变量 x 赋值为'A'
cout<< x << endl;       // 输出的结果是 65，因为'A'的 ASCII 码是 65

char y = 66;            // 定义一个字符型变量，并用整数 66 初始化
cout<< y << endl;       // 输出的结果是'B'，因为 66 对应的字符是'B'
y = 98;                 // 将变量 y 赋值为 98
cout<< y << endl;       // 输出的结果是'b'，因为 98 对应的字符是'b'
```

在用整数表示一个字符时，注意不要超过其取值范围。ASCII 码表总共有 128 个字符，所以与字符数据对应的整数也就有 128 个，其取值范围是 0~127。

下面编写程序依次输出大小写字母对应的 ASCII 码值，如示例代码 3.1 所示。

示例代码 3.1

```
#include <cstdlib>
#include <iostream>

using namespace std;                // 使用名称空间 std

int main(int argc, char *argv[])    // 主函数
{
    int a = 'a';                    // 定义整型变量 a，并初始化为字符'a'的 ASCII 码
    for( int i=0; i<26; i++ )       // 遍历 26 个小写字母
    {
        cout<< char(a+i) <<'='<<a + i<<'\t';   // 输出字母及对应的 ASCII 码
        if( 0 == (i+1) % 5 )        // 每输出 5 个，换一次行
        {
            cout<<endl;
        }
    }
    cout<<endl;                     // 分隔小写和大写字母

    a = 'A';                        // 将变量 a 赋值为大写字母'A'的 ASCII 码
    for( int i=0; i<26; i++ )       // 遍历 26 个大写字母
    {
        cout<< char(a+i) <<'='<<a + i<<'\t';   // 输出字母及对应的 ASCII 码
        if( 0 == (i+1) % 5 )        // 每输出 5 个，换一次行
        {
            cout<<endl;
        }
    }
    cout<<endl;                     // 为整个列表换行

   system("PAUSE");                 // 等待用户反应
   return EXIT_SUCCESS;             // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源代码文件中，编译并运行，结果如图 3.1 所示。

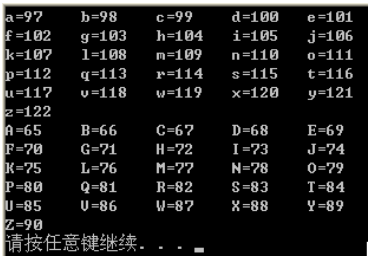


图 3.1 字母与对应的 ASCII 码

上面代码中“a+i”的作用是求得第 i 个字符的 ASCII 码值（一个整数），前面的 char(a+i)的作用是将求得的 ASCII 码值再转换成对应的字符。

3.2.6 无符号字符型

前面的 char 型是有符号字符型，只不过没有编号为负数的 ASCII 码字符而已。除了有符号字符型之外，在 C++中还有无符号字符型，其类型关键字为 unsigned char。无符号字符型的取值，除了包括 ASCII 码表上的所有字符外，还包括一个扩展 ASCII 码表上的所有字符。使用目前常见的键盘，无法输入扩展 ASCII 码表上的字符，但可以通过字符与整数的关系，来初始化或赋值无符号字符型变量，例如：

```
unsigned char c = 128; // 定义并初始化无符号字符型变量
cout<< c << endl;      // 输出无符号型字符
```

3.2.7 转义字符

在 C++中有些字符不能直接表示，只能通过特殊的方法来表示，比如换行符、制表符、响铃符等。对于这类字符可以用在某个字符前加一个反斜杠来表示。反斜杠的含义是：改变其后面字符的意义，用来表示另外的字符。这个表示方法也称为转义。一般的转义字符如表 3.1 所示。

表 3.1 转义字符表 1

字符	含义
\n	换行符
\r	回车符
\t	制表符
\v	垂直制表符
\b	退格符，当前输出位置向后退一格
\a	响铃符，当输出这个字符时，计算机的蜂鸣器会响一声

使用转义字符时，虽然在单引号内有两个字符，但实际上反斜杠表示一个转义的开始。如'\n'就是将原来的字符 n 转义成一个换行符，'\a'就是将原来的字符 a 转义成响铃符。

换行和回车是不一样的。换行只是将输出位置换到下一行，不改变输出的横坐标；回车是回到行首，不改变输出的纵坐标。

有些有歧义的字符，也需要用反斜杠开始的转义序列表示。比如，一个反斜杠字符就无法单用一个“\”表示，因为一个反斜杠已经用来表示转义的开始；单引号也无法单独表示，必须借助一个反斜杠进行转义。像这样的字符如表 3.2 所示。

表 3.2　转义字符表 2

字符	含义
\\	表示反斜杠，否则会同转义序列的开始标志相混淆
\’	单引号，否则会同表示字符的单引号相混淆
\”	双引号，C 和 C++中的字符串类型由双引号开始和结束，如不转义，则会引起混淆
\ddd	表示三位八进制数
\xdd	表示两位十六进制数

例如，为了输出上述字符，在程序中应当如下书写代码：

```
cout<< '\\' <<endl;     // 输出单个反斜杠
cout<< '\'' <<endl;     // 输出单个单引号
cout<< '\123' << endl; // 输出大写字母“S”, 八进制数 123 在 ASCII 表中对应“S”
cout<< '\x42' <<endl;   // 输出大写字母“B”, 十六进制数 42 在 ASCII 表中对应“B”
```

3.2.8 宽字符型

在 C++中，字符型除了 char 类型，还有一种 wchar_t 类型，表示宽字符。其字面常量是在 char 型字符前加上前缀“L”，例如：

```
L'a'        // 宽字符型常量 a
```

所谓宽字符，指的是用两个字节表示的字符。C++中使用宽字符是为了支持 Unicode。随着计算机技术的飞速发展，各种语言的数据都需要在计算机中存储和处理。但 ASCII 码仅能处理英文字符，远远满足不了人们的需要。于是人们发明了一种多字节编码，即用多个变长字节表示字符。但这种方式并没有在各种语言中统一，一个编码在一种语言中代表一个字符，而在另外一种语言中又表示其他的字符。后来人们对各种语言编码的方式做了一次统一，其目的是用同一种编码方式处理世界上所有语言中的字符，这就是 Unicode 编码。Unicode 编码规范非常复杂，读者可以参考相关的资料和书籍，限于篇幅，这里不做深入介绍，这里读者只需知道 C++的 wchar_t 类型是用来表示和处理 Unicode 字符的即可。宽字符变量的定义、初始化和赋值如下：

```
wchar_t  w;             // 定义一个宽字符型变量
w = L'W';               // 为变量赋值
wchar_t  v = L'V';      // 定义并初始化一个宽字符型变量
```

3.2.9 布尔型

布尔型数据的类型用 bool 表示，其字面常量只有两个：true 和 false，分别表示逻辑真和逻辑假。布尔型变量的定义、初始化和赋值如下：

```
bool  b;                // 定义一个 bool 类型的变量
b = true;               // 为变量赋值
bool  c = false;        // 定义并初始化一个变量
```

在程序中，一个布尔型变量常用来表示某个条件是否成立，例如：

```
int a = 0, b = 0;       // 定义两个整型变量
cin>>a;                 // 由用户输入变量的值
cin>>b;
bool flag = false;      // 定义一个布尔型变量
if( a > b )             // 如果变量 a 大于变量 b
{
    flag = true;        // 将布尔型变量设成 true
}
```

在程序中，布尔型变量可以看做整型变量。字面常量 true 可以当做整数 1，false 可以当做整数 0，例如：

```
int a = true;           // 定义一个 int 型变量，并初始化为 true，其实是被初始化为 1
cout<< a <<endl;        // 实际输出 1
a = false;              // 将 false 赋值给 int 型变量，其实是将 0 赋值给变量
cout<< a <<endl;        // 实际输出 0
```

当其他类型的数据转换为布尔型数据时，只要是非 0 的数据都将转换为 true，而 0 则转换为 false，例如：

```
bool b = -123;          // true
b = 0;                  // false
b = '\0';               // false
b = 'a';                // true
b = 0.0;                // false
b = 1.2;                // true
```

3.3 变量与内存的关系

通过前面的学习，读者已经了解到变量就是一段内存。既然是内存，就有地址、大小等方面的属性，下面将一一进行讲述。

3.3.1 变量的地址

计算机的内存是按照字节进行编码的。内存中一个字节的编码就是这个字节的地址。操作系统可以按照内存地址对内存进行存取操作。在 C++程序中，定义一个变量就是声明占用一块儿内存，变量名就指向这块儿内存的首地址。变量名与内存地址的关系如图 3.2 所示。

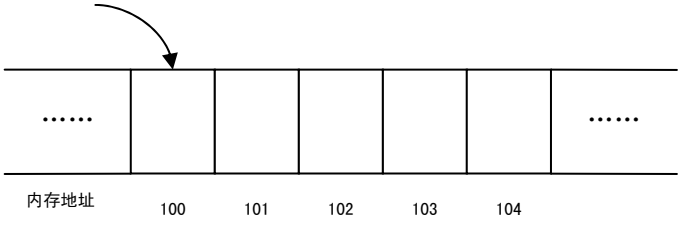


图 3.2 变量名与内存地址的关系

3.3.2 变量的字节长度

变量是用来存储数据的，而不同类型的数据对内存大小的要求是不一样的。数据占用的内存数量称为数据的大小，通常用机器字来衡量。一个机器字即计算机在一个时钟周期内，能够从内存中读取的数据大小，在 32 位机器中是 2 个字节（32 位），在 64 位机器中就是 4 个字节（64 位）。

C++标准规定了内建的数据类型的大小。例如，int 是一个机器字，short 是半个机器字，long 是一个或两个机器字；float 是一个机器字，double 是两个机器字，long double 是 3 或 4 个机器字。对于某个确定版本的编译器和某种计算机处理器，这些尺寸都是固定的。

虽然数据的尺寸是用机器字来衡量的，但通常在表述数据大小时却是以字节为单位的。在 32 位机器中，一个机器字是 4 个字节，所以，在 32 位平台中常用的 int 型是 4 个字节，short 型是 2 个字节，long 型一般也是 4 个字节。

在 C++程序中定义一个变量，即表示占用一块儿内存。变量名代表这块儿内存的首地址，变量的类型则决定了这块儿内存的长度。例如，在程序中定义两个变量：

```
char c = 'c';
int a = 12;
```

其占用内存情况如图 3.3 所示。

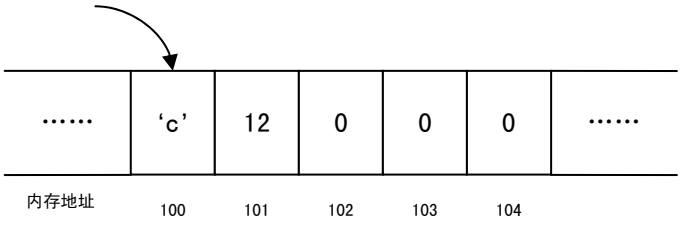


图 3.3　变量占用内存情况

3.3.3　计算数据的字节长度

sizeof 运算符的作用是返回一个类型或数据的字节长度。sizeof 的使用方法比较灵活，在 sizeof 关键字后面可带括号，也可不带括号；括号中可以是类型名，也可以是变量名。其格式如下：

```
sizeof( <类型名> );
sizeof( <变量名或者常量> );
sizeof <变量名或者常量>;
```

sizeof 运算的结果是一个 size_t 型的数据。size_t 是一种内建数据类型的别名，关于类型别名将在以后的章节中说明，目前可以将其作为整型使用。下面使用 sizeof 运算符计算类型和数据的尺寸，如示例代码 3.2 所示。

示例代码 3.2

```
#include <cstdlib>
#include <iostream>

using namespace std;                    // 使用名称空间 std

int main(int argc, char *argv[])        // 主函数
{
    // 定义各种类型的变量，并初始化
    short a0 = 0;
    int a1 = 1;
    long a2 = 2L;
    char c = 'c';
    float d0 = 1.23F;
    double d1 = 4.56;
    bool b = true;

    // 计算并输出各种数据类型和数据的字节长度
    cout<<"shot: "<<sizeof(short)<<'\t'<<"0: "<<sizeof(0)<<endl;
    cout<<"int: "<<sizeof(int)<<'\t'<<"1: "<<sizeof(1)<<endl;
    cout<<"long: "<<sizeof(long)<<'\t'<<"2L: "<<sizeof 2L<<endl;
                                                    // 注意：sizeof 没有用括号
    cout<<"char: "<<sizeof(char)<<'\t'<<"\'c\': "<<sizeof('c')<<endl;
    cout<<"float: "<<sizeof(float)<<'\t'<<"1.23F: "<<sizeof(1.23F)<<endl;
    cout<<"double: "<<sizeof(double)<<'\t'<<"4.56: "<<sizeof(4.56)<<endl;
    cout<<"bool: "<<sizeof(bool)<<'\t'<<"true: "<<sizeof(true)<<endl;
    // cout<<sizeof int<<endl;      // 不使用括号的形式，不能用于类型

   system("PAUSE");                 // 等待用户反应
   return EXIT_SUCCESS;             // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源代码文件中，编译并运行，结果如图 3.4 所示。

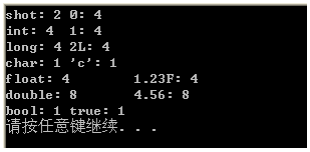


图 3.4 用 sizeof 运算符计算字节长度

不同的计算机平台，计算的结果可能不一样，这取决于所用计算机的处理器和编译器。

本书中如果没有特别指出，使用的计算机平台都是 32 位的。

3.3.4 变量的取值范围

变量的类型决定了可取值的类型，而类型的字节长度则决定了变量可取值的范围。如字符型变量只占一个字节，它的值可以是 ASCII 和扩展 ASCII 字符集中的 256 个字符和符号。短整型（short）占两个字节，即 16 位，可以取值的范围是-2^{16}~2^{16}-1，即-32 768 ~ 32 767。其他各种数据类型的字节长度及其取值范围如表 3.3 所示。

表 3.3 变量字节长度及其取值范围

类型	长度（字节）	取值范围
bool	1	true 或 false

unsigned short int	2	0 ~ 65 535

（续表）

类型	长度（字节）	取值范围
short int	2	-32 768 ~ 32 767
unsigned long int	4	0 ~ 4 294 967 295
long int	4	-2 147 483 648 ~ 2 147 483 647
int	4	-2 147 483 648 ~ 2 147 483 647
unsigned int	4	0 ~ 4 294 967 295
char	1	256 个字符
float	4	1.2e-38 ~ 3.4e38
double	8	2.2e-308 ~ 1.8e308

3.4 自定义数据类型

C++内置的数据类型远远满足不了人们对数据类型的需要，为此，C++标准也支持用户自定义数据类型。自定义数据类型的种类有结构体、共用体、枚举体和类。在 C++语言中，结构体、共用体与类的定义基本相同，只有很小的差别。

3.4.1 结构体

结构体是一个包含多个数据成员的集合，到底包含哪些数据成员，开发者可以自己定义。定义结构体的关键字是 struct，其后是结构体名称，然后是由两个花括号包围的结构体。花括号中是组成结构体的各种成员，可以是数据，也可以是函数。每个成员用分号“;”作为结尾。定义结构体格式如下：

```
struct 标识符
{
成员列表
};
```

例如定义一个拥有整型数和浮点数的结构体：

```
struct SomeStruct       // SomeStruct 是结构体名称
{
    float a;            // 成员
    int b;              // 成员
};                      // 分号表示结构体定义结束
```

结构体定义最后的花括号后面必须带有一个分号，表示定义的结束。

由于结构体也是一种数据类型，所以也可以用来定义变量，其方法和定义普通变量的方法一样。例如定义一个 SomeStruct 结构体的变量：

```
SomeStruct  s;
```

为了访问结构体变量中的成员，可以使用运算符“.”，例如：

```
s.a = 10;
```

上面是对结构体变量 s 的成员 a 进行赋值的操作。结构体类型本身不占用任何内存空间，只有结构体变量才占用内存空间，其占用的内存空间大小是各个数据成员类型大小之和。不过各种编译器出于效率的考虑，往往会对结构体变量进行内存对齐操作，从而使得结构体变量所占内存大小往往大于各个数据成员类型大小的和。

3.4.2 共用体

与结构体不同，共用体的数据成员存储时共享存储空间。共用体在有的书中也被称为联合体类。定义共用体类型用关键字 union，形式为：

```
union 标识符
{
成员表
};
```

例如，定义一个共用体类型，要求包含一个整数、一个字符和一个单精度浮点数：

```
union SomeUnion
{
    int  a;
    char b;
    float c;
};
```

定义共用体变量同定义普通类型的变量类似，如：

```
SomeUnion a;
```

对于共用体类型数据，其各个数据成员占用的内存是共享的，修改一个成员，就等于修改其他成员，例如：

```
union  INT_UN
{
    int  a;
    int  b;
};
……
INT_UN  x;
x.a = 123;
cout<< x.b << endl;
```

上述程序代码的输出结果是 123。虽然程序没有直接给 x 的成员 b 赋值，但由于是共用体，其成员变量共享内存，所以修改 a 就会影响到 b，所以 b 的值也是 123。

3.4.3 枚举体

如果要使变量只能使用有限的几个值，则应当使用枚举体。之所以叫枚举体，就是因为在定义枚举体类型时，需要将所有可能的值一一列举出来。定义枚举体的关键字是 enum。例如定义一个代表星期几的变量，这样的变量的取值范围是固定的，就是星期一到星期天这 7 个值，所以应当使用枚举体。

```
enum Day
{Mon, Tue, Wed, Thu, Fri, Sat, Sun };
```

枚举体类型的变量可以如下定义：

```
Day day = Tue;
```

其中 day 是一个枚举体 Day 类型的变量，其可能的取值就是定义 Day 时列举的 7 个值，在这里就是 Tue。

虽然枚举体的值是由一些字符组成的，但这些值却可以当成整型数使用。如没有特别指明，则枚举体中的值就是从 0 开始的整数。例如在 Day 中，Mon 对应的整数值是 0，Tue 对应的整数值是 1，依此类推。

定义枚举体时所列举的各个标识符代表的是一系列常量，在程序中只能当做常量使用，而不能像变量一样为其赋值。

在定义枚举体时，也可以指定每个分量的值。一旦某个分量的值被指定，则其后分量的值就会在这个值的基础上开始递增。例如，可以根据使用习惯将 Day 中 Mon 的值改成 1：

```
enum Day
{Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun };
```

如上定义后 Tue 的值就是 2，Wed 值是 3，依此类推。

3.5 用宏替换字面常量

字面常量，如 123，'a'等，虽然简单、直接，但使用起来并不方便。由于没有标识符，所以后来的人在阅读程序时就难以知道其由来。而且，如果一个字面常量被多次用到，还容易写错。例如，圆周率就是这样一个常量：

```
double  radius = 1.2;                          // 圆半径
double  perimeter = radius * 2 * 3.14159265;   // 求圆的周长
```

```
double  area = 3.14159265 * radius * radius;   // 求圆的面积
double  perigonRadian = 2 * 3.14159265;        // 求周角的弧度
double  rightRadian = 0.25 * 3.1415927;        // 求直角的弧度，这里的圆周率写错了！
```

在上述代码中有三个主要问题：

- 直接使用字面常量不容易理解。例如在程序中直接使用 3.14159265，由于这只是个数值，其本身不能自我说明，除程序员自己，别人难以知道这就是圆周率。
- 字面常量容易写错。在上面的代码中就是如此，第 5 行的圆周率跟前面使用的圆周率都不同。虽然都是真实圆周率的近似值，单独使用都没错，但在一个程序中应当保持一致，而直接使用字面常量难以保证这一点。
- 如果程序中有多处使用同一个字面常量，那么当程序逻辑发生改变需要修改这个字面常量时就很麻烦，需要手动修改所有用到的地方。例如上述程序，如果实际计算时发现圆周率的精度不够，要改成 3.141592653589 才合乎要求，则需要查找所有用到原有圆周率的地方，并逐个进行修改，非常烦琐。

要解决这个问题，可以使用宏。宏是一个预编译指令，其语法如下：

```
#define <宏名>   <宏的内容>    // 注意这里没有用“;”结尾
```

上述语法称为宏定义。有了这个宏定义之后，源代码中的“宏的内容”就可以用“宏名”替换。例如，在上述那个圆周率的例子中，源代码可以这么改写：

```
#define PI  3.14159265                  // 用宏替代常量 3.14159265
double  radius = 1.2;                   // 圆半径
double  perimeter = radius * 2 * PI;    // 求圆的周长
double  area = PI * radius * radius;    // 求圆的面积
double  perigonRadian = 2 * PI;         // 求周角的弧度
double  rightRadian = 0.25 * PI;        // 求直角的弧度
// double   radian = PL;                // 写错了！程序中没有定义 PL
```

宏的名字一般写成大写，用于与一般的标识符区别开来。

与原来不使用宏的代码相比，使用宏有如下好处：

- 宏比字面常量好理解。字面常量只有结合其使用情况才能明确其含义。而定义良好的宏，其本身的名字就可以表明其含义。
- 字面常量如果写错，则代表的是另外的常量，编译器不会报错。而宏名如果写错，编译就不会通过，程序的开发者就可以按照编译器的提示进行修改。

- 使用宏的程序比仅使用字面常量的程序容易修改。例如要提高圆周率的精度，只要在定义宏的语句中进行修改即可，以后代码中使用的 PI 都会是修改后的值。

源代码中的宏，在编译之前由预处理器进行宏展开。所谓宏展开，就是将源代码中所有的“宏名”用“宏的内容”进行替换，其效果就如同直接书写“宏的内容”一样。宏展开时，预编译器不会对宏的内容进行任何检查，而是忠实地执行文本替换工作。例如，即便将宏 PI 定义成 3.14abc15926 也是合法的，但在编译时会报错。

3.6 用 const 定义常量

尽管宏（#define）的使用非常方便，但宏只是进行文本替换，而替换的内容没有任何限定。这样做虽然灵活，也非常危险。为此，C++引入了另外一种更好的定义常量的方法——const 常量，其语法如下：

```
const  <类型>  <常量名>  =  <初始化值>;
```

上面的语法类似于变量的定义，只是在前面加上了关键字 const。const 是限定符，表明其后定义的数据是一个常量。const 常量在定义时必须进行初始化，因为一个常量定义后，其值是不能修改的。const 常量最大的优点是其定义中规定了常量的类型，因此，编译器可以根据其类型来使用。现在用 const 常量来修改前面那个圆周率的例子：

```
const   double  pi = 3.14159265;        // 定义const常量pi并初始化为3.14159265
double  radius = 1.2;                   // 圆半径
double  perimeter = radius * 2 * pi;    // 求圆的周长
double  area = pi * radius * radius;    // 求圆的面积
double  perigonRadian = 2 * pi;         // 求周角的弧度
double  rightRadian = 0.25 * pi;        // 求直角的弧度
```

其中常量 pi 的定义已经指定了其类型是 double，这样在初始化时 pi 能够接受的值就只能是 double 型的数据。

3.7 综合实例

在本节中将通过综合实例，详细讲解本章所讲语法知识的用法。希望读者通过本节的内容，灵活掌握本章的内容。

3.7.1 计算圆的周长和面积

圆的直径可以使用整数表示，但由于圆周率是一个浮点数，所以圆的周长和面积应当使用浮

点数表示。下面设计一个程序，要求用户输入圆的直径，由程序输出圆的周长和面积，如示例代码 3.3 所示。

示例代码 3.3

```
#include <cstdlib>
#include <iostream>

using namespace std;                                    // 使用标准名称空间

int main(int argc, char *argv[])                        // 主函数
{
    cout<<"——计算圆的周长和面积——"<<endl;                 // 提示信息
    cout<<endl;

    const double PI = 3.1415926;                        // 定义常量 PI
    double radius = 0;                                  // 圆半径变量

    cout<<"请输入圆的半径"<<endl;                          // 提示信息
    cin>>radius;                                        // 输入半径
    cout<<"周长："<< 2* PI * radius <<endl;               // 计算并输出圆的周长
    cout<<"面积："<< PI*radius*radius <<endl;             // 计算并输出圆的面积

    cout<<endl;
    system("PAUSE");                                    // 等待用户输入
    return EXIT_SUCCESS;                                // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源代码文件中，编译并运行，结果如图 3.5 所示。

图 3.5 计算圆的周长和面积结果

3.7.2 三角形的类型判断和面积计算

三角形是比较简单的二维几何图形，具有边长、角度等属性。在程序中可以用一个结构体来表示三角形。在一些有关图形的计算中，常常需要对三角形进行计算和处理，这些处理可以转变成针对这个三角形结构体的处理。下面设计一个程序，要求用户输入三角形的三条边的边长，判断三角形的类型，并求出三角形的面积，如示例代码 3.4 所示。

示例代码 3.4

```
#include <cstdlib>
```

```
#include <iostream>

using namespace std;                        // 使用标准名称空间

struct Triangle                             // 三角形结构体
{
   double a;                                // 边长
   double b;                                // 边长
   double c;                                // 边长
};

int main(int argc, char *argv[])            // 主函数
{
   cout<<"——三角形的计算和表示——"<<endl;   // 提示信息
cout<<endl;

   Triangle t;                              // 三角形变量

   cout<<"依次输入三角形的边长："<<endl;          // 提示信息
   cin>>t.a;                                // 输入
   cin>>t.b;
   cin>>t.c;

   double max = t.a > t.b ? t.a : t.b;      // 求两边中的最大值
   double x = t.a + t.b - max;              // 求两边中的较小值
   double z = max > t.c ? max : t.c;        // 求三边中的最大值
   double y = t.a + t.b + t.c - x - z;      // 求第三边

   if( z <= 0 || z >= x + y )               // 判断三角形的合法性
   {
      cout<<"输入的边长不能组成一个三角形。"<<endl;
      cout<<endl;
      system("PAUSE");                      // 等待用户输入
      return EXIT_SUCCESS;
   }

   char *pType = NULL;
   if(z*z == x*x + y*y)                     // 是否是直角三角形
   {
      pType = "直角";
   }
   else if( z*z < x*x + y*y )               // 是否是锐角三角形
   {
      pType = "锐角";
   }
   else
   {
      pType = "钝角";                       // 否则是钝角三角形
```

```
    }
    cout<<"三角形类型："<<pType<<endl;              // 输出三角形类型
    cout<<endl;
    system("PAUSE");                                // 等待用户输入
    return EXIT_SUCCESS;                            // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源代码文件中，编译并运行，结果如图 3.6 所示。

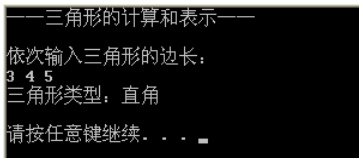


图 3.6　三角形的类型判断和面积计算结果

3.8 小结

本章的重点是 C++程序中的各种数据类型以及变量与内存的关系。一般来讲，程序就是用来处理数据的，所以如何表示和存储各种数据就是程序的基础。当然，程序不仅要能表示和存储数据，还要能够对其进行有效的处理。在程序中，处理数据的基本手段就是语句和表达式，这将是下一章要讲解的内容。另外，除了本章所讲的几种基本数据类型外，在 C++中还有指针、引用、对象等类型，这些都将在后面的章节中一一进行讲述。

◆　　◆　　◆

第 4 章　语句和表达式

本章包括

- 语句和语句块的概念
- 表达式的概念
- 表达式中的各种运算符
- 运算符的优先级和结合性
- 类型转换

通过上一章的学习，读者已经了解了在程序中如何存取数据，这一章我们来学习如何处理数据。在程序中，数据处理是通过语句和表达式完成的。一条语句或一个表达式就是数据处理的一个步骤，多个表达式和语句按照一定顺序组合在一起，就构成了一次完整的数据处理过程。

4.1 语句和语句块

语句是数据处理过程中的最小步骤，其标志是结尾处的分号“;”，例如：

```
int a = 0;
int b = 1;
int x = 0;
;                          // 空语句
x = a + b;
x = add( a, b );           // add 是一个函数，接受两个整型参数
```

上述这些代码都是语句。请注意第 4 行并不是一个书写错误，而是表示什么都不做的空语句。虽然空语句没有什么意义，但在 C++程序中是允许的。

4.1.1 空格的作用

语句中的空格有的是必需的，而有的则完全是出于美观的目的，例如：

```
int a = 0;
```

这是一个变量声明及初始化的语句。其中关键字“int”和变量“a”之间的空格是必需的，否则就构成了一个新的标识符“inta”。而“a”,“=”和“0”之间的空格则仅仅是美观的需要，如果不加空格，写成：

```
int a=0;
```

虽然表达的意思是一样的，但看上去不如带有空格那么清楚。下面的语句更加明显地突出了空格的重要性：

```
x=add(a,b);
```

4.1.2 语句块的组织

在 C++程序中，多个连续的语句可以组成语句块（也称为复合语句）。语句块必须用花括号“{”和“}”包围起来，例如：

```
double  area = 0;
{   // 语句块开始
```

```
    double  width = 3;
    double  height = 4;
    area = width * height;
}   // 语句块结束
cout<<"The area is "<<area<<endl;
```

语句块可以嵌套，例如：

```
{       // 块 1 开始

    ……
    {   // 块 2 开始

        ……
    }   // 块 2 结束

    ……
}       // 块 1 结束
```

4.1.3 语句块中的变量

虽然看起来，语句块好像没什么作用，加不加花括号并没什么影响，但实际上语句块限定了变量的作用域（作用范围）。在语句块中声明的变量，其作用范围从声明语句开始，直到语句块结束。一旦超出作用范围，则该变量就不再有效了。

例如，在 4.1.2 节的例子中，变量 width 和 height 的作用范围分别从第 3 行和第 4 行开始，直到第 6 行结束。从第 7 行开始，就不能再使用这两个变量了，否则会导致编译错误。变量 area 是在语句块之外定义的，所以 area 可以在第 7 行以后使用。

4.2 什么是表达式

所谓表达式就是一个对各种数值（包括对象）进行计算的过程。既然是计算，就会有结果。从这个意义上来说，表达式表达的就是一个数值，而这个数值是经过各种计算得到的。例如“3+2”就是一个表达式，其结果是“5”。

参与计算过程的数值称为操作数，而表示操作的符号称为运算符。一个表达式由一个或多个操作数以及零或多个运算符构成。最简单的表达式由一个字面常量或一个变量构成，例如：

```
123
'a'
Variable
```

表达式的结尾没有分号“;”。如果带上分号，则构成了一条语句。例如“123”是一个表达式，而“123;”则是由一个字面常量表达式构成的语句。

字面常量表达式的结果是其字面值，而变量表达式的结果就是这个变量的值。通常情况下，表达式中有一个或多个运算符。例如在“3+2”这个表达式中，“+”是一个运算符，表示对操作数“3”和“2”进行相加运算。常见的表达式还有：

```
PI * radius * radius                        // 计算圆面积的表达式
1.2 * (baisicSalary + bonus - penalty) // 计算薪水的表达式
```

如果一个表达式的操作数也是一个表达式，则前者称为复合表达式，而后者称为子表达式。例如，在计算薪水的例子中，表示相乘操作（*）的第二个操作数“(baisicSalary + bonus - penalty)”就是一个子表达式，而整个计算薪水的表达式就是一个复合表达式。

子表达式的计算顺序是由运算符决定的，准确地说是由运算符的优先级和结合性决定的，这将在后文中进行说明。

4.3 运算符分类

C++为了实现各种操作，引入了几十种运算符。如算术运算符（+，-，*，/），求变量内存地址的取址运算符（&），以及比较大小的关系运算符（<，>，==，<=，>=）等。按照运算符作用的操作数个数不同，可以将运算符分为一元运算符、二元运算符和三元运算符。

- 一元运算符 :如取址运算符(&),其操作数只有一个变量。常见的取址表达式如“&variable”，其结果是变量 variable 在内存中的地址。
- 二元运算符：如算术运算符，每个都需要两个操作数，如“1+2”,“3*4”等。
- 三元运算符：在 C++中只有一个接受三个操作数的运算符“?:”，常见的表达式如“a>b?a:b”，其含义是如果 a 大于 b，则表达式的结果是 a 的值，否则是 b 的值。

按照运算符的操作效果，可以分为算术运算符、赋值运算符、关系运算符、条件运算符、sizeof 运算符、位操作运算符等。

4.3.1 算术运算符

算术运算符包括加、减、乘、除（四则运算）运算符（+、-、*、/）和求模运算符（%）。算术运算符是二元运算符，其操作数一般是整数或浮点数（或者是结果为整数或浮点数的表达式）。如果两个操作数都是整数，则运算的结果也是整数，例如：

```
1 + 2          // 结果是整数 3
3 - 4          // 结果是整数-1
5 * 6          // 结果是整数 30
6 / 2          // 结果是整数 3
5 / 2          // 结果是整数 2
```

在除法操作中，如果商中含有小数部分，则小数将被截掉（舍尾法），如 1 / 3 的结果是 0，5 / 2 的结果是 2。C++中的除法运算同算术中的除法运算一样，除数都不可为 0，否则会导致程序崩溃。在编程时要注意避免除数为 0 的情况，例如：

```
int a, b;                    // a 是被除数，b 是除数
```

```
cin>>a;                     // 输入被除数
cin>>b;                     // 输入除数
if ( 0 != b ){              // 如果 b 不是 0，才执行“{}”中的语句
    cout<< a / b <<endl;    // 输出 a/b 的值
}
```

求模运算符（或称取余运算符）%用于计算两个数相除的余数，左边的操作数被右边的操作数除。该运算符只能用来处理整数，包括 char，short，int 和 long 型的数。当两个操作数都是正数时，结果为正。但是，如果有一个或两个操作数是负数，则余数的符号取决于编译器的实现，因此可移植性无法保证。求模运算举例如下：

```
3.14 % 3;       // 编译错误，%不能用于浮点数
21 % 6;         // 结果是 3
21 % 7;         // 结果是 0
21 % -5;        // 编译器相关：结果为 -1 或 1
```

4.3.2 算术运算的溢出

在计算机中，数据的取值范围是有限制的，取决于其类型的尺寸，例如 short 型数据只能取 -32 768 ~ 32 767 之间的数。如果某些算术运算的结果超出了范围，则会导致数据的“上溢出”或“下溢出”。一旦发生溢出，则其结果是未定义的，例如：

```
short a = 32767;            // 声明短整型变量 a，并赋初值为上限
a = a + 1;                  // 加 1，试图超过上限
cout<<a<<endl;              // 输出变量的值
unsigned short b = 65535;   // 声明无符号短整型变量，并赋初值为上限
b = b + 1;                  // 加 1，试图超过上限
cout<<b<<endl;              // 输出变量的值
```

其输出结果并不是 32 768 和 65 536。不管实际输出的是什么结果，都是没有意义的，因为C++标准并没有对此做出规定。编程时，依据一种没有定义的行为是很危险的，而且其可移植性无法保证。

4.3.3 赋值运算符

C++的赋值运算符是“=”，这是一个二元运算符，接受两个操作数，其使用方法如下：

```
a = b;
```

其含义是将右操作数 b 的值赋给左操作数 a，即用 b 的值覆盖 a 的值。右操作数可以是常量，也可以是变量；而左操作数必须是一个左值，即该操作数有一个相关联的、可写的地址值。下面是一个明显的非左值赋值的例子：

```
1024 = ival;                    // 错误
```

一种可能的解决方案为：

```
int value = 1024;
```

```
value = ival;                    // 正确
```

给 const 常量赋值也是错误的，例如：

```
const int value = 1024;          // const 常量的声明和初始化
value = ival;                    // 错误，不能给 const 常量赋值
```

语句“a=b;”和“int a=b;”是不同的。前者是赋值，而后者是声明一个变量并初始化。初始化与赋值不同，初始化只能发生在变量声明时，而在程序中，一个变量可以多次赋值。

由“=”连接两个操作数，构成一个赋值表达式，其结果值是赋值完成后左操作数的值。例如，在计算表达式“a = 3”时，先将 3 赋给左操作数 a，然后取 a 的值作为这个表达式的结果值。

4.3.4 自增和自减运算符

自增运算符（++）的目的是简化变量加 1 的操作，如操作 a = a + 1 可以简化为：

```
a++;        // 即 a = a + 1;
```

同样，自减运算符（--）的目的是简化变量减 1 的操作，如 a = a－1 可以简化为:

```
a--;        // 即 a = a – 1;
```

在 C++程序中这种加 1 和减 1 的操作是非常普遍的，在以后的编程中将经常看到，所以对这两种操作的简化很有必要。C++也支持这两个运算符的前置版本，即运算符在前、变量在后，例如：

```
++a;
--a;
```

自增和自减运算符“++”和“--”后置和前置所产生的结果是不同的。后置版本是先使用变量的值，然后再对变量施行自加和自减操作，例如：

```
int a = 6;
int b = a++;        // 该条语句运行过后，b 的值是 6，a 的值是 7
int c = a--;        // 该条语句运行过后，c 的值是 7，a 的值是 6
```

上面语句中 b 的值是 6，而不是 7。其中第 2 行中 b 的初始化值即“a++”，是一个表达式。由于是后置版本，所以表达式的值是 a 自加之前的值，即 6，b 也初始化为 6。基于同样的原理，读者可以自行分析一下第 3 行的运行结果。前置版本是先对变量施行自加和自减操作，然后再使用变量的值，例如：

```
int a = 6;
int b = ++a;        // 该条语句运行过后，b 的值是 7，a 的值是 7
int c = --a;        // 该条语句运行过后，c 的值是 6，a 的值是 6
```

上面语句中 b 的值是 7，而不是 6。其中第 2 行中表达式“++a”由于是前置版本，所以表达式的值是 a 自加之后的值，即 7，b 也初始化为 7。基于同样的原理，读者可以自行分析一下第 3 行的运行结果。

4.3.5 关系运算符

在 C++中，常见的关系运算符如表 4.1 所示。

表 4.1 常见的关系运算符

运算符	功能	用法
<	小于	a < b
<=	小于等于	a <= b
>	大于	a > b
>=	大于等于	a >= b
==	等于	a == b
!=	不等于	a != b

关系运算符是二元运算符，用于比较两个操作数的大小关系。关系运算符的操作数是两个表达式，比较的结果是一个 bool 型的值，即 true 或 false，例如：

```
bool res = false;           // 声明并初始化 bool 型变量
res = 3 < 4;                // res 的值是 true
res = 4 < 3;                // false
int a = 5;
int b = 3;
es = a >= b;                // true
res = a + 1 == 2 * b - 1;   // true
res = a != b;               // true
```

4.3.6 逻辑运算符

在 C++中，常见的逻辑运算符如表 4.2 所示。

表 4.2 常见的逻辑运算符

运算符	功能	用法
!	逻辑非	!a
&&	逻辑与	a && b
||	逻辑或	a || b

逻辑非是一元运算符，其含义是求表达式的相反值，例如：

```
bool a = false;
bool res = !a;      // res 的值是 true
res = !res;         // res 的值是 false
```

逻辑与运算符（&&）和逻辑或运算符（||）都是二元运算符，其结果值的计算方法如下：

- 只有当逻辑与（&&）运算符的两个操作数都为 true 时，结果值才为 true。
- 对于逻辑或（||）运算符，只要两个操作数之一为 true，结果值就为 true。

例如：

```
bool res = false;
res = true && false;        // res 为 false，因为两个操作数中有一个是 false
```

```
res = true || false;        // res 为 true，因为两个操作数中有一个是 true
```

这些操作数按从左至右的顺序计算。只要能够得到表达式的值 true 或 false，运算就会结束。例如给定以下表达式：

```
expr1 && expr2
expr1 || expr2
```

如果下列条件有一个满足：则保证不会计算 expr2。

- 在逻辑与表达式中 expr1 的计算结果为 false。
- 在逻辑或表达式中 expr1 的计算结果为 true。

4.3.7 条件运算符

条件运算符（?:）是唯一的一个三元运算符，其语法格式如下：

```
expr1 ? expr2 : expr3;
```

其含义是：如果子表达式 expr1 的结果是 true，则整个表达式的结果是子表达式 expr2 的值；否则，整个表达式的结果是子表达式 expr2 的值。

条件表达式的典型用法如下：

```
int a = 20, b = 30;
…
int larger = ( a > b ) ?a : b;
int smaller = ( a < b) ? a : b;
```

上面代码用于求两个数中的较大者和较小者。

4.3.8 逗号运算符

逗号表达式是一系列由逗号分开的表达式，这些表达式从左向右计算，逗号表达式的结果是最右边表达式的值。例如，在“a += 3, c * 5;”中，程序依然会先执行 a 的自增，再执行 c * 5。

4.3.9 位运算符

在 C++中，常见位运算符的功能与用法如表 4.3 所示。

表 4.3 常见位运算符的功能与用法

运算符	功能	用法
~	按位非	~expr
<<	左移	expr1 << expr2
>>	右移	expr1 >> expr2
&	按位与	expr1 & expr2
^	按位异或	expr1 ^ expr2
|	按位或	expr1 | expr2
&=	按位与赋值	expr1 &= expr2
^=	按位异或赋值	expr1 ^= expr2

|=	按位或赋值	expr1 |= expr2

位运算符把操作数解释成有序的位集合，每个位的值是 0 或 1。位运算符允许程序员设置或测试独立的位。

位运算符一般用于处理整数。虽然整数可以是有符号的，也可以是无符号的，但一般是无符号的。因为在大多数的位操作中符号位的处理是未定义的，因此不同的编译器对符号位的处理可能会不同。

1. 按位非运算符~

该操作符的作用是翻转操作数的每一位。如果某位是 1，则改为 0；如果某位是 0，则改为 1。例如：

```
short  a = 5;
short  b = ~a;      // b 的值是-6
```

短整型 5 的二进制是 0000 0101，每位取反，则结果值的二进制是 1111 1010，即-6。

2. 左移运算符<<和右移运算符>>

该操作符的作用是将其左边操作数的位向左或向右移动，移动的位数是右边的操作数。例如：

```
short  a = 5 << 1;  // a 的值是 10，5 左移 1 位的二进制结果是 0000 1010，即十进制的 10
short  b = 5 >> 2;  // b 的值是 1，5 右移 2 位的二进制结果是 0000 0001，即十进制的 1
```

在移位过程中，某些位被移到存储区外面，则该位将被丢弃。左移运算符（<<）从右边开始用 0 补空位。对于右移运算符（>>），如果操作数是无符号数，则从左边开始插入 0；如果操作数是有符号数，则插入符号位的复制值或者插入 0，这由具体的编译器决定。

3. 按位与运算符&

对于进行按位与运算的两个整数操作数的每个位，如果两个操作数在该位处都是 1，则结果值的该位为 1，否则为 0。例如：

```
short  a = 5 & 4;// 即 0000 0101 & 0000 0100，得二进制结果 0000 0100，即十进制的 4
```

请不要把该运算符与逻辑与运算符&&相混淆。

4. 按位异或运算符

对于进行按位异或运算的两个整数操作数的每个位，如果两个操作数在该位处只有一个是 1（注意不是两个同时为 1），则结果值的该位为 1，否则为 0。例如：

```
short  a = 5 ^ 4;// 即 0000 0101 & 0000 0100，得二进制结果 0000 0001，即十进制的 1
```

5. 按位或运算符|

对于按位或运算的两个整数操作数的每个位，如果两个操作数在该位处只有一个是 1，则结果

值的该位为 1，如果都是 0，则结果值的该位为 0。例如：

```
short  a = 5 ^ 4;// 即 0000 0101 & 0000 0100，得二进制结果 0000 0101，即十进制的 5
```

4.3.10 复合赋值运算符

除了普通的赋值运算符“=”之外，在 C++程序中还可以将其他运算符与“=”相结合，构造一种复合赋值运算符，如“+=”、“*=”、“&=”等。这种运算符表示的是先用其他运算符计算左操作数和右操作数，然后将结果赋给左操作数。例如：

```
int a = 5;
a += 3;     // 实际是 a = a + 12;
a *= 4;     // 实际是 a = a * 4;
a &= 1;     // 实际是 a = a & 1;
```

采用复合赋值运算符，可以使程序更加简洁明了。

4.4 运算符的优先级和结合性

程序中的优先级同算术中的优先级概念一样，都指的是哪一个运算符先计算。例如在下面的语句中：

```
int  x = 6 + 3 * 4 / 2 - 2;
```

表示乘除的运算符（*和/）先于加减运算符（+和-）计算。运算符的结合性指的是同样优先级的两个运算符相邻时，先计算哪一个。如果左面的先计算，则该级运算符具有左结合性，否则具有右结合性。

算术运算符具有左结合性。例如在表达式“a + b - c”中，加减运算符的优先级相同，并具有左结合性，所以首先计算左面的加号，然后将计算的结果减去 c，就是表达式的结果，即语句的计算过程是：

step 1 乘除的优先级高，所以先计算“3*4/2”这个表达式。

step 2 乘除具有左结合性，所以先计算“3*4”，得 12，再除以 2 得 6。

step 3 再计算加减，即“6+6-2”。

step 4 加减具有左结合性，所以先计算左边的“6+6”，得 12，再算“12-2”，得 10。

C++中各种运算符的优先级和结合性如表 4.4 所示。

表 4.4 运算符的优先级和结合性

优先级	符号	名称	结合性
1	()	括号操作符或函数调用操作符	自左向右
	[]	数组操作符	
	->	指向成员操作符	
	.	成员操作符	
2	!	逻辑非操作符	相反，自右向左

	~	取反操作符	
	++	自增操作符	
	--	自减操作符	
	+	正号操作符	
	-	负号操作符	
	*	指针操作符	
	(type)	类型转换操作符	
	sizeof	字节操作符	
3	*	乘法操作符	自左向右
	/	除法操作符	
	%	求余操作符	
4	+	加法操作符	自左向右
	-	减法操作符	
5	>>	右移操作符	自左向右
	<<	左移操作符	
6	<	小于操作符	自左向右
	>	大于操作符	
	<=	小于等于操作符	
	>=	大于等于操作符	
7	==	等于操作符	自左向右
	!=	不等于操作符	
8	&	按位与操作符	自左向右
9	^	按位异或操作符	自左向右
10	|	按位或操作符	自左向右
11	&&	逻辑与操作符	自左向右
12	||	逻辑或操作符	自左向右
13	?:	条件操作符	相反，自右向左
14	=	赋值操作符	相反，自右向左
	+= ,-= ,*= ,/= ,%=	运算赋值操作符	
	&=，^=，|=	位操作赋值操作符	
	<<=，>>=	位移赋值操作符	
15	,	逗号操作符	自左向右

4.5 类型转换

在表达式的计算过程中，经常需要计算两个不同类型的操作数，或者将一种类型的值赋给另一种类型的变量。例如下列表达式：

```
1.2 + 3;        // double 型数据与 int 型数据相加
```

```
int a = 4.56;   // 将 double 型数据赋给 int 型变量
```

不同类型的数据在计算机中的处理方式不一样，例如 double 型和 int 型数据在内存组织和计算方法的实现上都不相同，所以为了完成上述操作必须进行类型转换。

4.5.1 隐式类型转换

对于内置类型 ,如 char ,int ,double 等 ,C++定义了一个标准转换规则。如 int 型转换为 double 型时，数值不变；double 型转换为 int 型，用舍尾法；char 型转换为 int 型时，采用对应的 ASCII 码。必要时，编译器隐式地按照这个规则对数据类型进行转换。转换的原则是：小类型总是被提升成大类型，以防止精度损失；大类型转换为小类型时，给出警告（不是编译错误）。所谓的小类型、大类型是从类型尺寸的角度来讲的。也就是说，占用内存小的类型总是转换成占用内存大的类型，如 char 型到 int 型、int 型到 double 型。

隐式类型转换发生在下列这些典型的情况下：

- 混合类型的算术表达式中。
- 用一种类型的表达式给另一种类型的变量赋值。
- 把一个表达式传递给一个函数调用，其类型与形参的类型不相同。
- 从一个函数返回一个表达式，其类型与函数的返回类型不相同。

如果一个算术表达式中，参与运算的各个数据的类型不相同，则其中尺寸最大的数据类型成为目标类型。运算之前，编译器先将各个操作数转换成目标类型，这种转换也被称为算术转换。例如在表达式“1.2 + 3”中，两个操作数分别为 double 型（两个机器字）和 int 型（一个机器字），double 比 int 的尺寸大，所以在计算前先将 3 转换为 double 型，其值为 3.0。这样表达式的结果也是 double 型。

当用一种类型的表达式给另一种类型的变量赋值时也会发生类型转换 ,此时目标类型是被赋值变量的类型。例如下列表达式：

```
double  a = 123;    // 标准的隐式转换
int  b = 3.14159;   // 精度损失，会导致一个编译警告
```

为一个函数传递参数时，如果实际传递的参数与函数声明中所用参数的类型不相同，那么就需要将所传递参数转换成函数所需参数的类型。例如 C++中求平方根的库函数 sqrt 接受一个 double 型参数，如果传入一个 int 型数据，则该数据会被提升为 double 型。

在函数的 return(返回)语句中 ,如果 return 表达式的类型与函数声明的返回类型不相同，就需要将实际返回值转换成函数的返回类型。在这种情况下目标转换类型是函数的返回类型。例如：

```
double difference( int ival1, int ival2 )
{
    // 返回值被提升为 double 类型
    return ival1 - ival2;
}
```

大类型向小类型转换在算术表达式中不会出现，但对于其他几种情形，却可能存在。虽然不会导致编译错误，但最好还是尽量避免，因为这会导致数据的精度损失。

4.5.2 特殊的隐式转换

在 C++中有一些比较特殊的类型转换，包括：

- char 型数据可以当做整型数使用。
- bool 型数据与其他数据类型的转换。
- 在算术表达式中，尺寸小于整型的类型总是先被提升为整型。

由于在 C++程序中字符是用 ASCII 码表示的，所以字符可以当做整型数使用，其值是对应的 ASCII 码值。例如'A'可以当做 65 使用，'a'可以当做 97 使用。

```
int iVal = 'A';     // iVal的值是65
iVal = 'a';         // 97
```

同样，在 ASCII 码表范围内的整数（0~256），也可以当做对应的字符使用。例如：

```
char cVal = 66;     // cVal的值是'B'
cVal = 98;          // 'b'
```

当程序中需要将其他数据类型转换为 bool 型时，如果传入的数据为零，则转换为 false；如果不为 0，则转换为 true。将 bool 型数据转换为其他类型数据时，如果传入的数据为 true，则转换为 1；否则，转换为 0。例如：

```
bool res = 0;           // res为false
res = 1;                // true
res = -1;               // true
res = 1.234;            // true
res = 0.0;              // false
res = 'a';              // true
int iVal = true;        // iVal为1
iVal = false;           // 0
double dVal = true;     // dVal为1.0
dValue = false;         // 0.0
```

算术表达式中小于整型的类型，总是先被提升为整型。在算术表达式中，如果只含有 char 型、short 型等数据，则这些数据先会被转换成整型。例如：

```
short  sVal = 1;
char  cVal = 'a';
cout<< sVal + cVal <<endl; // sVal转换成int型的1，cVal转换成int型的97
```

程序的输出结果是 98。

4.5.3 显式类型转换

如果要将某个数据显式地转换为其他数据类型，则可以使用下面的表达式：

```
类型(数据)
(类型)数据
```

上述表达式是等价的。例如把一个浮点型数据转换成整型，可以写为：

```
int a = (int) 3.1416;
```

也可以写为：

```
int a = int (3.1416 );
```

上述类型转换的方法继承自 C 语言，C++也提供了自己的类型转换运算符 static_cast。例如，为了将 double 型数据转换成 int 型数据，应当如下写：

```
int a = static_cast<int>( 3.1416 );
```

虽然 C++的运算符写起来比较复杂，但是这样写可以明确地告诉程序的阅读者："这里需要一个类型转换"，因此也方便了程序的维护。

4.6 综合实例

在本节中，将选择几个典型的综合实例讲解如何在 C++中使用表达式和语句。希望读者在学习这些例子后，能对 C++的基本语法有更详细的了解。

4.6.1 找出某个范围内的素数

所谓素数就是只能被 1 和自身整除的正整数。比如，1 就是一个素数，2 和 3 也是，但 4 不是，4 除了可以被 1 和自身整除外，还可以被 2 整除。从这个定义出发，要判断一个数是不是素数可以采用下面的算法：先判断这个数是不是 1 或 2，如果是则返回 true。如果不是，则求出该数的平方根，然后遍历大于等于 2 并小于这个平方根的所有整数，如果该数能够被这样的整数整除，则不是素数，否则就是一个素数。程序如示例代码 4.1 所示。

示例代码 4.1

```
#include <cstdlib>
#include <iostream>
#include <cmath>

using namespace std;                          // 使用名称空间 std

int main(int argc, char *argv[])              // 主函数
{
    cout<<"——求素数——"<<endl;                 // 输出提示信息

    int min = 0, max = 0;                     // 素数的范围

    cout<<"请输入一个范围："<<endl;            // 输出提示信息
    cout<<"大于等于：";
    cin>>min;                                 // 输入最小值
    cout<<endl;                               // 换行

    if( min < 1 )                             // 求取范围只能是正整数
```

```
    {
        min = 1;
    }

    cout<<"小于：";                          // 输出提示信息
    cin>>max;                                // 输入最大值
    cout<<endl;                              // 换行

    while( max <= min )                  // 如果输入有误，直到用户输入正确的最大值为止
    {
        cout<<"范围的上限必须大于下限。"       // 输出出错信息
            <<"请重新输入："<<endl;
        cin>>max;                            // 重新输入最大值
    }

    cout<<"上述范围内的素数是: "<<endl;       // 输出提示信息

    for( int i=min; i<max; i++ )             // 遍历范围内的所有整数
    {
        bool flag = true;                    // 标明当前整数是否是素数的标志
        if( 1 == i || 2 == i )               // 如果 1 或 2 在范围内直接输出
        {
            flag = true;
        }
        else                                 // 除 1 和 2 之外的其他数
        {
           int s = static_cast<int>(sqrt(i));  // 求平方根
           for( int j=2; j<=s; j++ )           // 从 2 开始遍历，直到平方根
           {
                if( (i % j) == 0 )             // 如果可以被整除，则不是素数
                {
                    flag = false;
                }
            }
        }

        if( flag )                           // 如果是素数
        {
            cout<<i<<endl;                   // 输出
        }
    }

   system("PAUSE");                          // 等待用户反应
   return EXIT_SUCCESS;                      // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 4.1 所示。

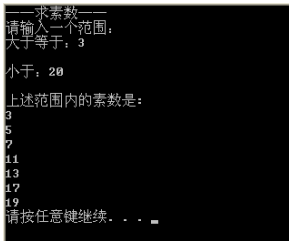


图 4.1 求素数输出结果

4.6.2 求最大值

假设允许用户输入 5 个整数，求其中的最大值，程序如示例代码 4.2 所示。

示例代码 4.2

```
#include <cstdlib>
#include <iostream>

using namespace std;                          // 使用名称空间 std

int main(int argc, char *argv[])              // 主函数
{
    cout<<"——求最大值——"<<endl;               // 输出提示信息
    int temp = 0;                             // 保存输入值的变量
    int max = 0;                              // 保存最大值的变量
    cout<<"请输入 5 个整数："<<endl;           // 输出提示信息

    for( int i=0; i<5; i++ )                  // 循环 5 次
    {
        cin>>temp;                            // 输入数据
        if( temp > max )                      // 如果当前输入的数据比最大值还大
        {
            max = temp;                       // 修改最大值
        }
    }

    cout<<"最大值是："<<max<<endl;            // 输出最大值

   system("PAUSE");                           // 等待用户反应
   return EXIT_SUCCESS;                       // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 4.2 所示。

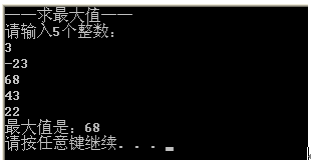


图 4.2 求最大值输出结果

4.7 小结

本章主要讲述了 C++中的语句、表达式和运算符。在 C++程序中，数据通过运算符组合成表达式。一个表达式后面加上分号，则该表达式就变成了一条语句。本章的重点是各种运算符的使用方法，以及运算符的优先级和结合性。另外，类型转换也是读者需要掌握的。

◆ ◆ ◆

第 5 章　程序流程控制

本章包括

- 描述程序流程的基本方法
- 用于循环控制的 while 语句、do…while 语句和 for 语句
- 用于流程跳转的 goto 语句
- 用于分支控制的 if 语句、if…else 语句和 switch 语句
- 中止循环的 break 语句和 continue 语句

程序流程指的是程序指令的执行次序。根据逻辑学家的研究，一般用顺序、分支和循环就可以描述所有的程序流程。顺序是最自然的流程，如果不加以控制，程序中的语句就是以其编写时的次序执行的，所以本章主要介绍分支和循环。

5.1　程序流程的描述

描述程序流程有很多种方式，比如伪码、流程图、UML（统一建模语言）的顺序图和活动图等。伪码和流程图比较适合描述短小的程序，特别是解决某个具体问题的算法，它们清晰、精确，使阅读者可以很容易地了解算法的细节。而 UML 图则适合描述大型程序，特别是用面向对象方法开发的程序，它便于阅读者从系统、整体的高度了解整个软件系统。本节只介绍伪码和流程图方法，关于 UML 图，请读者参考专门的书籍。

5.1.1　伪码

伪码即伪代码，就是用一种非常自然的、口语的方式来描述程序。伪码不受计算机语言语法的限制，行文非常自由，可以用任何熟悉的语言进行描述。相比代码，伪码可以更好地贴近人类的思维，更容易为阅读者所理解。例如，描述一个求三个数 a，b，c 中最大值的伪码如下：

step 1　比较 a 和 b，将其中较大的值赋给 x。

step 2　比较 x 和 c，将其中较大的值赋给 y。

step 3　输出 y。

这样就完成了一个简单程序的流程描述。这段流程描述也可以当做程序的设计文本，在后续的编码过程中可以根据这段伪码进行编程。

5.1.2　流程图

流程图以图的形式来描述程序的流程。程序中的每一个或几个步骤用一个图形符号表示，并结合图形中的文字说明该步骤的功用。每个图形符号之间还可以用带箭头的线连接，用来表示步骤间的执行次序。上节中伪码的例子就可以用如图 5.1 所示的流程图进行描述。

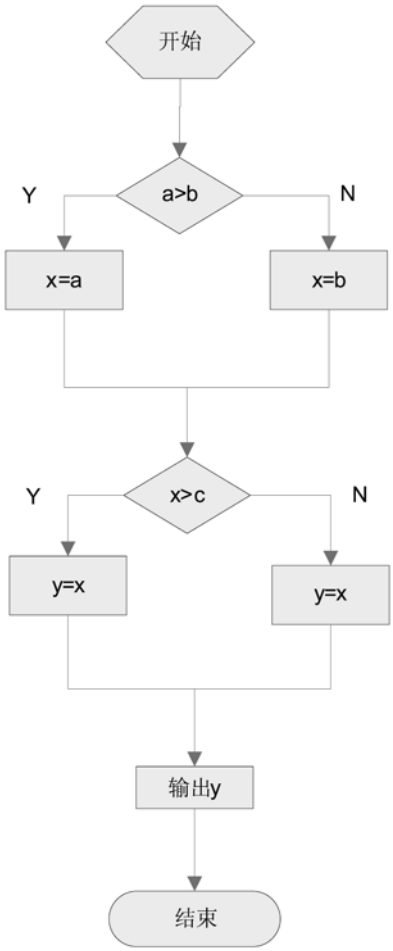


图 5.1　求三个数中的最大值

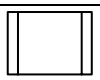
常用的流程图符号如表 5.1 所示。

表 5.1　常用的流程图符号

符号	名称	说明
	开始	程序开始
	结束	程序结束
	过程	数据处理的过程
	选择	程序中的逻辑判断

（续表）

符号	名称	说明

	预定义过程	已经存在的程序
	连接线	各步骤的连接线

5.2 分支

所谓分支，指的是根据预设的条件，选择不同的语句序列进行执行。在 C++程序中，“条件”是用一个表达式表示的，该表达式的结果是一个 bool 值或者某个整数值。在程序执行过程中，如果某个条件得到了满足，或者不满足，那么程序就会沿着相应的语句序列顺序执行下去。至于如何选择，则要借助于特殊的控制语句 if…else 或 switch…case。

5.2.1 if 语句

使用 if 语句的作用是：只有当满足一定条件时，才执行后续的语句或语句块。

执行一条语句的 if 语句如下：

```
if ( <条件表达式> )
    <语句>;
```

执行语句块的 if 语句如下：

```
if ( <条件表达式> )
{
    <语句 1>;
    <语句 2>;
    ……
}
```

If 语句的执行流程如图 5.2 所示。

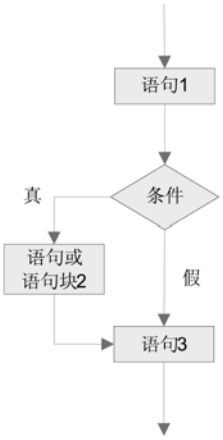


图 5.2 if 语句的执行流程

由图可见，只有当满足“条件”时（即条件表达式的结果为 true 时），才会执行语句或语句块 2。

不满足“条件”时（即条件表达式的结果为 false 时），执行 If 语句后面的语句 3。在 C++语言中，条件是用条件表达式表示的。所谓条件表达式指的是一个结果为 bool 值的表达式，例如：

```
a < b
a >= b
a != b
a == b
a > b && a < 0
```

由于其他数据类型和 bool 型数据间存在隐式数据类型转换的关系，所以当用其他表达式作为条件表达式时，虽然其结果不是 bool 型，但仍然可以使用。例如，对于表达式“a + b”，如果其结果值是 0，则转换成条件表达式的结果就是 false；如果其结果值是非 0，则转换成条件表达式的结果就是 true。

在实际编程中，经常会出现的一个错误是将“a = b”当成“a == b”。前者是一个赋值表达式，其结果是 b 的值，而后者是一个判断 a 和 b 是否相等的条件表达式，其结果是一个 bool 值。由于存在隐式类型转换关系，所以在编译时并不会报告错误，只有等到运行时才可能出现意料之外的结果。

5.2.2 if…else 语句

if 语句的语义是“如果…… ，则……”，C++还提供了一个 if…else 语句，表示“如果…… ，则…… ；否则……”语义。简单的 if…else 语句语法如下：

```
if ( <条件表达式> )
    <语句 1>;
else
    <语句 2>;
```

执行语句块的 if…else 语句语法如下：

```
if ( <条件表达式> )
{
    <语句 1>;
    <语句 2>;
    ……
}
else
{
    <语句 3>;
    <语句 4>;
    ……
}
```

其流程图如图 5.3 所示。

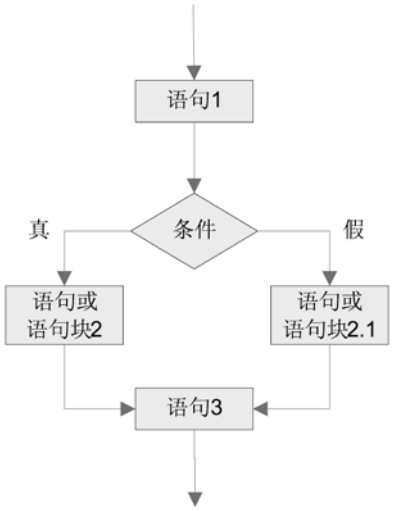


图 5.3 if…else 语句的执行流程

与 If 语句的执行流程比较，可以得出 if 语句同 if…else 语句的区别：if 语句没有处理条件不满足的情况，而 if…else 语句有。

下面用 if…else 语句求两个整数中的较大值，程序如示例代码 5.1 所示。

示例代码 5.1

```
#include <cstdlib>
#include <iostream>

using namespace std;                // 使用名称空间 std

int main(int argc, char *argv[])    // 主函数
{
    int a = 0, b = 0;               // 用于比较大小的两个整型变量
    cout<<"请输入两个整数："<<endl;   // 提示用户输入
    cin>>a;                         // 输入第一个数
    cin>>b;                         // 输入第二个数
    cout<<"较大的整数是："<<endl;     // 输出提示
    if( a <= b ){                   // 如果 a 小于等于 b
        cout<<b<<endl;              // 输出较大值 b
    }
    else{                           // a 不小于等于 b，即 a 大于 b
        cout<<a<<endl;              // 输出较大值 a
    }
   system("PAUSE");                 // 等待用户输入
   return EXIT_SUCCESS;             // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 5.4

所示。

图 5.4　求两个整数中的较大值结果

在上面的程序中，利用一个 if…else 语句判断输入的第一个整数是否小于等于第二个整数。如果是，则输出第二个整数；如果不是，则输出第一个整数。

5.2.3 if…else 语句的嵌套

if…else 语句可以嵌套，即其中的语句块可以是另外一个 if…else 语句，例如下面的形式：

```
if ( <条件表达式 1> )
    if ( <条件表达式 2> )
        <语句 1>;
    else
        <语句 2>;
else
    if ( <条件表达式 3> )
        <语句 3>;
    else
        <语句 4>;
```

其中的每个语句又可以是一个语句块。在嵌套的 if…else 语句中，相配的 if 和 else 不是靠缩进决定的，而是遵循一定的规则。这个规则是：else 和最后出现、并且没有匹配的 if 相匹配。例如在上述例子中，相互匹配的 if 和 else 分别是：第 4 行的 else 和第 2 行的 if，第 6 行的 else 和第 1 行的 if，第 9 行的 else 和第 7 行的 if。

即便是程序员知道这样的匹配规则，在阅读上述代码时仍然会感到困难，比较好的方法是尽量使用复合语句。例如，上述例子可以改写如下：

```
if ( <条件表达式 1> )
{
    if ( <条件表达式 2> )
        <语句 1>;
    else
        <语句 2>;
    <语句 3>
}
else
{
    if ( <条件表达式 3> )
        <语句 4>;
    else
        <语句 5>;
```

```
    <语句 6>
}
```

如此一改，if 和 else 的匹配关系非常明确，程序阅读起来也十分方便，其流程图如图 5.5 所示。

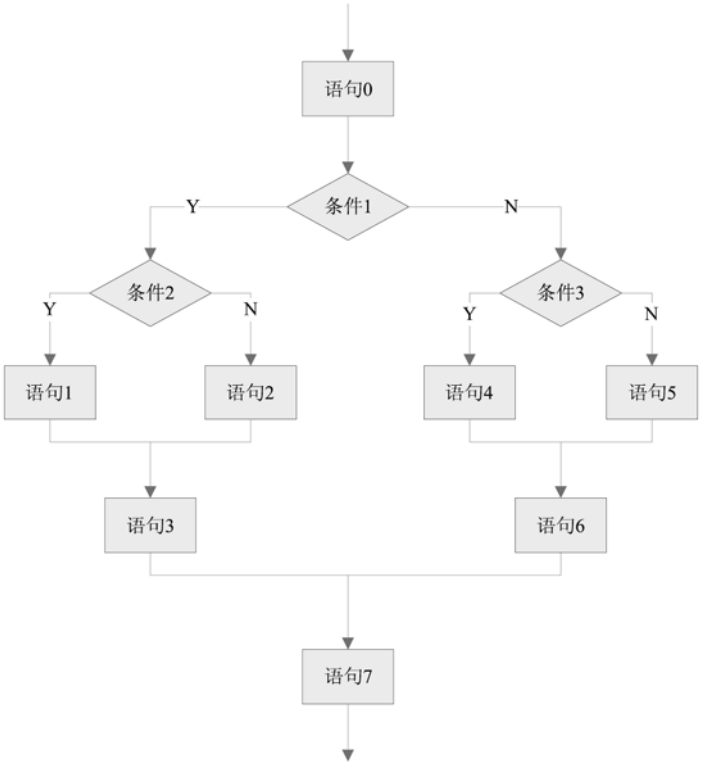


图 5.5 if…else 语句嵌套的流程图

if…else 语句的嵌套还有一种比较常见的形式，即 if…else 级联：

```
if ( <条件表达式 1> )
    语句 1;
else if ( <条件表达式 2> )
    语句 2;
else if( <条件表达式 3> )
    语句 3;
.....
else
    语句 n;
```

在执行上述语句时，程序依次计算每个条件表达式。一旦某个条件满足，则执行对应的语句或语句块，并且不再判断后面的条件。因此，每个条件表达式应当是互斥的。

下面利用 if…else 嵌套语句，评定学生成绩的档次。成绩的满分为 100，分为 5 档：0~59 为 E，60~69 为 D，70~79 为 C，80~89 为 B，90~100 为 A。根据输入的成绩，输出学生成绩的档次。程

序如示例代码 5.2 所示。

示例代码 5.2

```
#include <cstdlib>
#include <iostream>

using namespace std;                          // 使用名称空间 std

int main(int argc, char *argv[])              // 主函数
{
    int degree = 0;                           // 保存成绩的变量
    cout<<"请输入学生成绩："<<endl;           // 提示用户输入
    cin>>degree;                              // 用户输入成绩
    if( degree < 0 || degree > 100 ){         // 判断输入是否合法
        cout<<"成绩只能在 0~100 之间"<<endl;  // 输出不合法信息
        return EXIT_SUCCESS;                  // 主函数返回，程序结束
    }

    if( 0 <= degree && degree < 60 ){         // 如果成绩不及格
        cout<<'E'<<endl;                      // E 档
    }
    else if( 60 <= degree && degree < 70 ){       // 60~69
        cout<<'D'<<endl;                      // D 档
    }
    else if( 70 <= degree && degree < 80 ){       // 70~79
        cout<<'D'<<endl;                      // C 档
    }
    else if( 80 <= degree && degree < 90 ){       // 80~89
        cout<<'D'<<endl;                      // B 档
    }
    else if( 90 <= degree && degree <= 100 ){     // 90~100
        cout<<'D'<<endl;                      // A 档
    }

    system("PAUSE");                          // 等待用户反应
    return EXIT_SUCCESS;                      // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 5.6 所示。

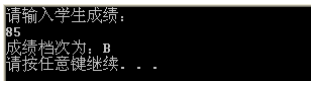


图 5.6　判断成绩档次结果

在上面的代码中，通过一系列的 if...else 判断，决定了成绩所属的档次。各个 if 中的条件是互斥的，所以可以保证一个成绩只能对应一个档次。

其实在上述例子中，每个条件表达式还可以改进。例如对于表达式：

```
0<=degree && degree<60
```

可以省略前面的表达式，而写成：

```
degree < 60
```

这是因为第一个 if 语句，已经处理了 degree<0 以及 100 < degree 的情况。此时 degree 的值只能在 0 和 100 之间，所以不必再去判断 degree 是否大于等于 0，只要判断 degree 是否小于 60 即可。同样对于第一个 else if 条件表达式也可以写成 degree < 70，依此类推。

5.2.4 switch 语句

if…else 语句虽然可以根据表达式的结果，执行不同的语句，但是如果表达式存在多个可能的结果，则多个 if…else 语句未免显得过于烦琐。而且在编程过程中，很可能会由于疏忽而漏掉一个或多个 if…else 判断。为此 C++提供了 switch 语句，用来在这种情况下替代 if…else 语句。其常见的使用方法如下：

```
switch ( expr )
{
case <常量 1>:
    <语句 1>;
    break;
case <常量 2>:
    <语句 2>;
    break;
……
case <常量 n>:
    <语句 n>;
    break;
default:
    <default 语句>;
}
```

switch 语句由以下 4 部分构成：

- switch 表达式。
- 一组 case 标签。
- 与一个或一组 case 标签相关联的语句或语句块。
- default 标签。

switch 表达式由 switch 关键字和一个表达式组成（被括号包围），其语法如下：

```
switch ( expr )
```

表达式 expr 的结果值是一个整数或者枚举值，或者其他可以转换成整数或枚举值的值。

switch 表达式后面是一组由一对花括号“{”和“}”包围的 case 标签。所谓 case 标签是由关键字 case 后接一个常量表达式及冒号构成的。这个常量表达式将被用来与 switch 表达式的结果做比较。例如：

```
case 'a':
case 123:
```

```
case eSaturday:     // eSaturday 是表示周六的枚举值
case iVal:          // iVal 是一个符号常量，即 const 常量
```

在一个或一组 case 标签后面有跟这个（组）标签相关联的语句或语句块。每个语句块后面都要跟一个 break 语句，表示语句块的结束。例如：

```
case eSaturday:
    <语句 1>;
    <语句 2>;
    ……
    break;
case eSunday:
    <语句 4>;
    ……
```

如果 switch 表达式的值是 eSaturday，则执行第 2 行至第 5 行的语句。当执行到第 5 行时，发现有 break 语句，则停止执行 switch 语句，转而执行 switch 语句后面的语句，即 switch 结束花括号"}"后面的语句。如果没有第 5 行的 break 语句，则继续执行"case Sunday:"后面的语句，直至遇到一个 break 语句。

有的时候，程序员可能就是需要多个 case 标签对应同一个语句或语句块，则连续的多个 case 标签可以只加一个 break 语句。例如：

```
switch ( day )
{
case eMonday:
case eTuesday:
case eWednesday:
case eThursday:
case eFriday:
    <语句 1>;
    break;
case eSaturday:
case eSunday:
    <语句 2>;
    break;
}
```

在上述的 switch 语句中，对于工作日（周一到周五），执行语句 1；对于周末，则执行语句 2。

何时不加 break 语句，取决于程序员的设计。虽然这样做没什么不妥，但对于后来维护程序的人则未免有些麻烦。因为后来者并不知道没有加上的 break 语句是故意为之，还是编程者的疏忽。所以，为了方便程序的维护，最好在不加 break 语句的地方加上注释，说明为什么不加。

switch 语句通常还会有一个 default 标签，放在所有 case 标签的后面。default 标签后直接跟一个冒号，中间只允许存在空格或制表符，而不是表达式。default 标签的冒号后面是跟 default 相关联的语句或语句块。如果 switch 表达式与任意一个 case 标签都不匹配，则 default 标签后

面的语句将被执行。

假设一个 switch 语句带有三个常量标签和 default 标签，则其执行过程如图 5.7 所示。

与嵌套的 if…else 语句不同，switch 语句在计算完“条件表达式”之后，并不依次判断其结果值是否与某个常量相等，而是直接跳到对应的标签处开始执行。例如，如果条件表达式计算的结果是常量 3，则程序会直接跳到语句 3 开始执行，并不会依次判断结果是不是常量 1 和常量 2。

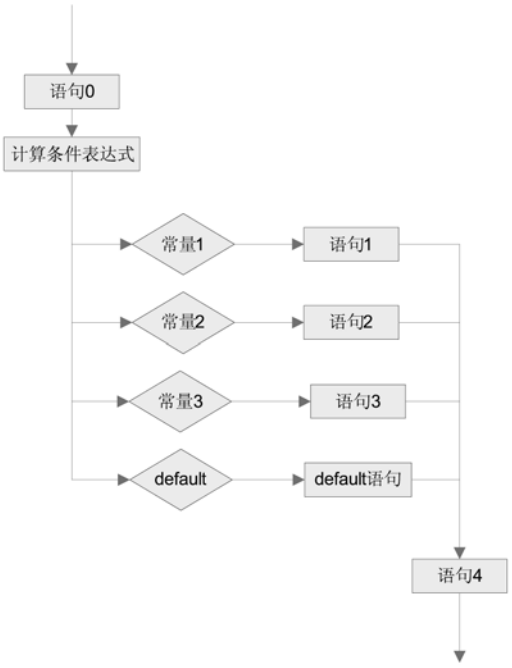


图 5.7 switch 语句的执行过程

前面的例子用多个嵌套的 if…else 语句来判断成绩的档次，其过程未免显得有些烦琐，下面用 switch 语句对其进行改进，如示例代码 5.3 所示。

示例代码 5.3

```
#include <cstdlib>
#include <iostream>

using namespace std;                          // 使用名称空间 std

int main(int argc, char *argv[])              // 主函数
{
    int degree = 0;                           // 保存成绩的变量
    cout<<"请输入学生成绩："<<endl;            // 提示用户输入
    cin>>degree;                              // 输入成绩
    cout<<"成绩档次为：";                      // 提示输出信息
```

```
    switch( degree / 10 ){                  // 根据成绩被 10 除的商，判断成绩档次
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
            cout<<'E'<<endl;                // E 档，60 分以下不及格，全为 E 档
            break;
        case 6:
            cout<<'D'<<endl;                    // D 档
            break;
        case 7:
            cout<<'C'<<endl;                    // C 档
            break;
        case 8:
            cout<<'B'<<endl;                    // B 档
            break;
        case 9:
            cout<<'A'<<endl;                    // A 档
            break;
        default:
            cout<<"成绩只能在 0~100 之间"<<endl;   // 输出不合法信息
    }

    system("PAUSE");                            // 等待用户反应
    return EXIT_SUCCESS;                        // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 5.8 所示。

图 5.8　用 switch 语句判断成绩档次结果

在上面的代码中，并没有直接根据成绩的范围求档次，而是根据成绩被 10 除的商来判断的。由于成绩是整数，所以对于每个档次，其所得商是固定的，如档次 B，即成绩在 80~89 范围内，其中每个数除以 10 都得 8。

5.3 循环

在本节中，将详细讲解程序控制中的循环结构。和前面类似，本节将详细讲解各种循环结构的功能和语法结构，并结合具体例子进行分析。

5.3.1 while 语句

while 语句用于控制一个语句或语句块循环执行。while 语句包含一个条件判断，只有条件为真时，其对应的语句或语句块才会循环执行。执行一条语句的 while 语句语法如下：

```
while ( expr )
    <语句>;
```

执行语句块的 while 语句语法如下：

```
while ( expr )
```

```
{
    <语句 1>;
    <语句 2>;
    ……
}
```

在上述语法示例中 ,expr 是一个表达式 ,也是 while 语句的循环条件。expr 的结果应当是 bool 值，如果不是，则需要进行类型转换。

while (expr)之后的语句或语句块称为循环体。当程序执行到 while (expr)时，先计算表达式 expr 的值，若其值为真，则执行循环体中的语句。循环体执行完一次后，再次判断 expr 的值，若为真 ,则再次执行循环体。如此反复 ,直到 expr 的值为 false ,然后执行 while 语句之后的语句。整个执行过程如图 5.9 所示。

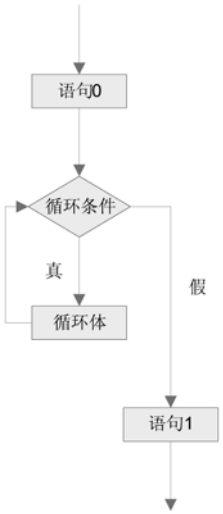


图 5.9 while 语句的执行过程

前面计算成绩档次的程序有一个显著的缺点 ,那就是在程序的一次运行中 ,只能判断一个成绩的档次。如果要判断别的成绩的档次，需要重新运行程序，非常烦琐。显然，如果程序能够循环重复执行，就会非常方便。下面用 while 语句改进判断成绩档次的程序，如示例代码 5.4 所示。

示例代码 5.4

```
#include <cstdlib>
#include <iostream>

using namespace std;                          // 使用名称空间 std

int main(int argc, char *argv[])              // 主函数
{
    int degree = 0;                           // 保存成绩的变量
    cout<<"请输入学生成绩："<<endl;           // 提示用户输入
    cin>>degree;                              // 输入成绩
```

```
    while( 0 <= degree && degree <=100 )   // 只要成绩合法，就可以判断档次
    {
        cout<<"成绩档次为：";                // 提示输出信息
        switch( degree / 10 ){
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
            case 5:
                cout<<'E'<<endl;        // E 档，60 分以下不及格，全为 E 档
                break;
            case 6:
                cout<<'D'<<endl;        // D 档
                break;
            case 7:
                cout<<'C'<<endl;        // C 档
                break;
            case 8:
                cout<<'B'<<endl;        // B 档
                break;
            case 9:
                cout<<'A'<<endl;        // A 档
                break;
        }
        cin>>degree;                    // 输入另外一个成绩
    }
    system("PAUSE");                    // 等待用户反应
    return EXIT_SUCCESS;                // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 5.10 所示。

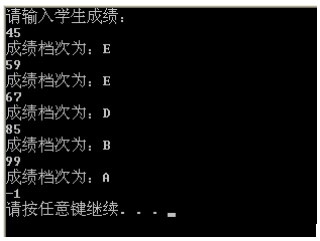


图 5.10　循环判断成绩档次结果

在示例代码中，首先使用一个 while 语句进行判断，只要输入的成绩合法，则后面的循环体就可以重复执行，然后判断成绩的档次。用户输入之后则重新判断表达式的值是否为 true。

5.3.2 do…while 语句

do…while 语句的目的也是控制代码的循环执行，执行一条语句的 do…while 语句如下：

```
do
```

```
    <语句>;
while ( expr );
```

执行一个语句块的 do…while 语句如下：

```
do
{
    <循环体语句>;
} while ( expr );
```

do…while 语句最后的分号表示语句的结束，不可省略，否则会导致编译错误。

与 while 语句不同，do…while 语句先执行一次循环体，再判断循环条件是否满足。其执行过程如图 5.11 所示。

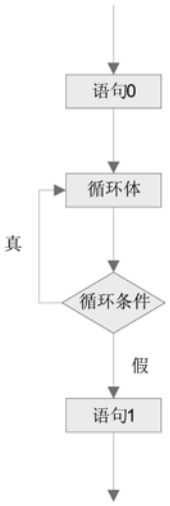


图 5.11 do…while 语句的执行过程

例如下面的 do…while 循环语句：

```
do{
    cout<<"Hello World."<<endl;
} while ( false );
```

虽然循环条件为 false，但上述代码仍然输出“Hello World”。而对于 while 循环：

```
while (false ){
    cout<<"Hello World"<<endl;
}
```

当循环条件为 false 时，循环体不能执行，所以也没有任何输出。

5.3.3 for 语句

for 语句也用来控制代码的循环执行。for 语句更加灵活，功能也更加强大。执行一条语句的 for 语句语法结构如下：

```
for ( <初始化>; <条件表达式>; <动作> )
```

```
    <语句>;
```

执行一个语句块的 for 语句语法结构如下：

```
for ( <初始化>; <条件表达式>; <动作> )
{
    <语句 1>;
    <语句 2>;
    ……
}
```

for 后面的括号中是三条独立的语句，分别是初始化、条件计算和动作。for 循环的执行过程是：

step 1 执行初始化语句，完成循环的初始化。

step 2 计算条件表达式，并检测其结果值是否为真。

step 3 如果条件为真，则执行循环体。

step 4 执行动作语句。

每次执行完循环体之后，则重复执行第 2 步至第 4 步。例如：

```
for( int i=0; i<10; i++ )
{
    cout<<i<<endl;
}
```

其执行过程是：

step 1 声明并初始化变量 i。

step 2 计算表达式 i<10 的值，其值是真，执行循环体，输出 i 的值。

step 3 执行动作 i++。

step 4 重复执行第 2 步和第 3 步，不断输出 i 值，并增加 i 的值，直至 i 值为 10，不满足条件，循环结束。

for 语句的执行过程如图 5.12 所示。

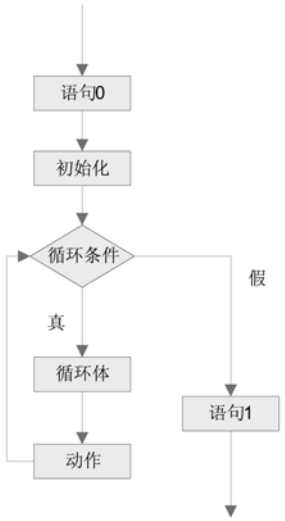


图 5.12 for 语句的执行过程

for 语句中的三条语句都可以是空白语句，但分号不可以省略。例如：

```
for ( ; <条件表达式>; <动作> )
for( ; ; <动作> )
for(<初始化>; ; )
for( ; ; )
```

以上都是合法的 for 语句。下面利用 for 循环，获取 100 以内 2 的幂，程序如示例代码 5.5 所示。

示例代码 5.5

```
#include <cstdlib>
#include <iostream>
03
using namespace std;                          // 使用名称空间 std
05
int main(int argc, char *argv[])              // 主函数
{
    for( int power=1; power<=100; power*=2 )// 利用 for 循环求 2 的幂，限定 100 以内
    {
        cout<<power<<endl;                    // 输出 2 的幂
    }
   system("PAUSE");                           // 等待用户反应
   return EXIT_SUCCESS;                       // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 5.13 所示。

图 5.13 利用 for 循环求 100 以内 2 的幂结果

在 for 循环语句中，初始化语句将变量 power 初始化为 1，也是 2 的 0 次幂。条件表达式 power<=100，限定这个幂是 100 以内的值。而动作语句 power*=2 则依次求 2 的幂，即后一个幂是前一个幂的 2 倍。

在 for 循环的初始化语句中可以声明并初始化变量。根据 C++标准规定，这个变量的作用域限于 for 语句，即从声明开始，直到最后一条循环体语句结束。但是，并不是所有的编译器都支持这一点。不管编译器是否支持，程序员在编程时都应当遵循标准，否则程序的可移植性就很难保证。

5.4 循环控制语句

在循环过程中，有时循环条件过于复杂，难以用一个表达式说明，需要借助特殊的手段控制循环过程。在 C++程序中，这种特殊手段包括 break 语句和 continue 语句。

5.4.1 break 语句

break 语句的作用是终止整个循环，执行循环语句后面的语句。在循环中 break 语句通常与一个 if 语句配合使用。例如在下面的程序中使用 break 语句：

```
for ( int i=0; i<5; i++ ){      // for 循环控制
    if( i == 2 ){               // 如果 i 的值是 2
        break;                  // 结束循环
    }
    cout<< i << ' ';            // 输出 i 的值和一个空格
}
```

输出结果为“0 1 ”。当 i 的值为 2 时运行 break 语句，结束循环，1 后面的值不再输出，虽然 for 语句中的条件表达式表明可以一直输出到 4。

break 语句只能用在 switch 语句和循环语句中，否则会引起编译错误。

5.4.2 continue 语句

continue 语句的作用是终止本次循环，开始下一次循环。上面 break 语句实例中如果将第 3 行的语句改为 continue，则输出结果是“0 1 3 4 ”。

continue 语句只能用在循环语句中。

5.5 流程跳转语句 goto

使用 goto 语句的目的是实现无条件分支，而这个分支只能是函数内部的分支。运行 goto 语句将跳转到同一函数内部的某个位置，该位置由一个标号表明。goto 语句的语法如下：

```
goto label;
```

这里的 label 是用户定义的标识符标号。label 只能用做 goto 的目标，并且必须由冒号结束。label 处必须有程序语句，如果什么都不执行，则应当加一个空语句。

```
end: ;  // 空语句
}       // goto 的目标不能是一个语句块的结束花括号，所以应当加一个空语句
```

goto 语句应当与一个 if 语句配合使用，否则 goto 语句与目标标号之间的语句将被无条件跳过。例如：

```
for( int i=0; i<10; i++ ){
    if( i == 5 )
        goto END;
    cout<< i << endl;
}
```

```
END:
    cout<<"Jump here"<<endl;
```

如果没有 if 语句，程序不会输出任何整数；加上这个 if 语句，程序就会输出 0~4 的整数。

编程时尽量避免使用 goto 语句，否则会使程序难以理解和维护。

5.6 小结

本章详细讲解了 C++中的程序控制语句。在实际开发过程中 ,用户需要经常使用各种控制语句，根据实际的需要来控制程序的走向。灵活地使用各种控制语句，可以给开发带来各种便利。希望读者能够在实际开发中，综合使用各种语句。

◆ ◆ ◆

第 6 章　数组与字符串

本章包括

- 数组的概念和定义方法
- 数组的初始化
- 数组元素的访问方法
- 一维数组的应用
- 二维数组的应用
- 字符串的定义及基本操作
- 字符串处理函数

前面章节中所涉及到的变量和常量都是一个数据，而且数据与数据之间的关系是松散的。本章将介绍一种数据集合类型——数组。与结构体不同，数组是同一类型数据的集合。而且在内存中，数组中的元素依次排列，一个紧邻一个。字符串也是数组的一种，不过字符串比较特殊，而且 C++标准库也提供了很多专门用于操纵字符串的函数。本章的重点是介绍数组和字符串在程序中的应用。

6.1　什么是数组

在程序设计中，往往需要把多个类型相同的数据放在一块儿处理。例如一组表示学生成绩的整数，或者一组表示坐标的二维点对象。这些数据的共同特点是类型相同。为了方便存储和处理这些数据，可以使用数组。

所谓数组，就是一组按照顺序排列在一起且类型相同的多个数据。数组中的各个元素，也就是每个数据，都没有单独的名字。要访问其中的某一个数据，需要使用数组名和下标。数组名就是这个数组的标识符，而下标就是某个数据在数组中的序号，该序号从 0 开始。访问数组元素时，数组名后面跟一对中括号，下标写在中括号里面，其语法如下：

```
数组名[ 下标 ];
```

假设某个数组 array 有 10 个元素，则访问第一个、第三个和第十个元素可分别写为 array[0]，array[2]和 array[9]。

数组有一维数组和多维数组之分。只有一个下标的数组即一维数组，有两个下标的数组称为二维数组，依此类推。本章首先讲述一维数组，如果没有特别指出，本书中的数组指的就是一维数组。

数组有两个属性，即类型和长度，数组的类型就是元素的类型，数组的长度就是这个数组中元素的个数。例如一个存储 10 个整型数的数组，其类型就是整型，而长度就是 10。由于元素的下标是从 0 开始的，所以该数组的第一个元素的下标是 0，而最后一个元素的下标是 9，如图 6.1 所示。

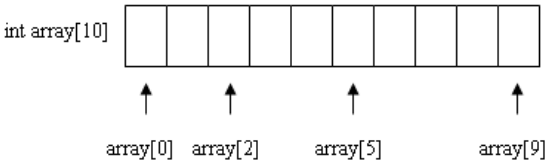


图 6.1 数组的内存布局

除了类型相同外，数组中的数据还有一个特点，即各元素在内存中按照顺序依次排列。也就是说，在内存中数组是一块儿连续的内存，其大小等于各元素所占内存大小之和。由于各元素类型相同，所以数组所占内存大小也等于元素个数与一个元素所占内存大小的乘积。

在如图 6.1 所示的数组中，设其类型为整型，数组第一个元素的内存地址是 1000，则后续元素的地址依次递增一个整数所占字节数(因平台不同而不同，一般来讲 32 位平台中整数占 4 字节)，依次为 1004，1008……直到 1036，如图 6.2 所示。

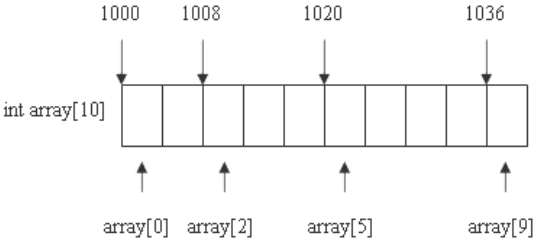


图 6.2 数组在内存中的地址

不要把数组长度和数组所占内存大小混为一谈。数组长度是一个习惯性的说法，其含义就是数组中所含元素的个数，而数组所占的内存则是其中各元素所占内存大小的总和。

6.2 定义数组

要使用数组就要先定义数组。数组定义包括数组名、数组类型和数组长度。数组类型也就是数组中各元素的数据类型，数组长度由一个常量表达式指定。定义数组的语法如下：

```
类型说明符 数组名[ 常量表达式 ];
```

其中，类型说明符可以是任何 C++内置数据类型，如整型、浮点型、布尔型等；也可以是自定义类型，如结构体、枚举体、类等。数组名是该数组的标识符，用来唯一标识数组，其命名必须符合标识符的命名规范。

数组的长度用一个常量表达式表示。该常量表达式必须是一个在编译的时候可以计算出的值，这是因为编译器要知道为该数组分配多大的内存空间。常量表达式只能是整型字面常量、枚举常量或者用常量表达式初始化的整型符号常量 (用 const 修饰的常量)。例如：

```
int arr1[10];            // 定义一个数组，元素类型为 int，名字为 arr1，长度为 10
```

```
bool arr_b[3];          // 定义一个数组，元素类型为 bool，名字为 arr_b，长度为 3
```

下面几个数组的定义包含了两个初学者容易犯的错误：

```
int length1 = 20;               // 定义变量
const int length2 = 128;        // 定义常量
const int length3 = get_size();// 常量的值需要通过函数计算
int arr1[length1];              // 错误，length1 是变量
int arr2[length2];              // 正确，length2 是符号常量
int arr3[length2+1];            // 正确，常量表达式
int arr4[length3];              // 错误，虽然是常量，但是只有在运行的时候才知道值
```

在上述例子中，length1 为变量，值只有在运行的时候才可以确定，不可以作为数组长度，因此 arr1 的定义是错误的。length2 是一个用常量表达式初始化的整型常量，可以作为数组长度，因此 arr2 的定义是正确的。表达式 length2+1 是常量表达式，在编译的时候可以计算得其值为 129，因此 arr3 的定义是正确的。length3 虽然为常量，但是其值需要等运行的时候才可以得到，也不符合数组长度的要求，因此 arr4 的定义是错误的。

同变量的定义一样，数组也可以连续定义。例如：

```
int a[30], b[20];       // 定义了两个整型数组 a 和 b，长度分别为 30 和 20
```

6.3 初始化数组

在定义数组的时候，可以为每个数组元素提供初值，这就是所谓的初始化数组。元素的这些初值包含在两个花括号（“{”和“}”）中，并用逗号分隔，称为初始化列表。初始化数组的语法如下：

```
类型说明符 数组名[ 常量表达式 ] = {值 1, 值 2, ……, 值 n};
```

其中，各个元素的初始值同数组长度一样，也必须是在编译时就可以确定的值，不能为变量。例如：

```
int a1[3] = { 0, 1, 2 };// 定义一个长度为 3 的整型数组，并将各元素依次初始化为 1，2，3
const  int  x = 0;      // 符号常量
int a2[3] = { x, x+1, x+2 };
                        // 定义一个长度为 3 的整型数组，并将各元素依次初始化为 0，1，2
```

数组初始值的个数可以比数组的长度少。在这种情况下，数组将按照次序初始化前面的元素，后面的元素则初始化为 0。例如：

```
int a3[10] = { 1, 2, 3, 4 };
```

上面的语句把数组的前 4 个元素分别初始化为 1，2，3，4，后面 6 个元素则初始化为 0。

对于数组如果不提供初始化列表，则其值全部为随机数（不一定为 0）；如果提供初始化列表，但初始化列表中元素的个数少于数组的长度，则不足部分的数组元素用 0 来初始化。

但是初始化列表中元素的个数不能多于数组的长度。例如，下面的定义就是错误的：

```
int a4[2] = { 1, 2, 3 };            // 初始化值太多，错误
```

如果在定义数组的时候为所有元素都提供了初始化值，那么代表数组长度的常量表达式可以省略。此时数组的长度就是初始值的个数。例如：

```
int a5[5] = { 1, 2, 3, 4, 5 };
```

也可以写为：

```
int a5[ ] = { 1, 2, 3, 4, 5 };      // 数组长度为 5
```

对于一个普通的变量，可以在定义之后为变量赋值。例如：

```
int a;
a = 10;
```

但是对于一个数组，定义之后就不能像初始化那样为数组赋值。例如下面的写法就是错误的：

```
int a[10];
a = { 1, 2, 3 };                    // 错误的写法，不可以这样赋值
```

要想在定义之后修改某个元素的值，只能单独对该元素进行赋值。如何访问和操作数组中的各个元素，将在后面进行说明。

6.4 操作数组

操作数组的首要条件是能够访问数组中的元素。访问数组元素的基本方法是用“数组名[下标]”的形式。其中下标需要是一个整型常量或者结果为整型数的表达式。与数组定义时的长度表达式不同，数组的下标可以是一个变量或者含有变量的表达式。例如：

```
int arr[10];            // 整型的数组
arr[5];                 // 常量下标
arr[i];                 // 整型变量下标
arr[i+1];               // 整型变量表达式下标
```

在 C++中数组的下标是从 0 开始的。对于长度为 *N* 的数组，其元素下标的范围为 0 ~ *N*-1。因此，arr[0]表示数组 arr 的第一个元素，arr[1]表示数组 arr 的第二个元素，依此类推。对于上面例子中长度为 10 的数组，其最后一个元素就是 arr[9]。

使用下标访问数组元素时，注意不可超出下标的取值范围。在 C++语言中，编译器并不会对数组下标的有效性做出检测，超出数组下标范围的操作会导致不可预料的后果。因此，程序员必须自己保证数组下标的有效性。

在定义数组之后，不可以直接对数组进行赋值。要想修改其中元素的值，只能通过下标的方法先找到要操作的元素，然后进行处理。例如：

```
int const len = 10;     // 定义常量，作为数组长度
nt arr[len];
arr[0] = 0;
arr[1] = 1;
```

```
arr[2] = 2;
```

实际中经常用循环来改变数组元素的值，例如：

```
int const len = 10;     // 定义常量，作为数组长度
nt arr[len];
for( int i = 0; i < len; i ++ )
{
    arr[i] = i;         // 用循环访问数组元素
}
```

上面的代码是给数组元素赋值。初学者在学习 C++的时候，经常犯的一个错误就是直接对数组进行赋值等操作，例如想要让两个数组的元素有同样的值，初学者经常试图这么写：

```
a = b;                  // a 和 b 是长度相同的数组，错误的方法
```

上面的赋值语句是错误的，正确的做法就是用循环：

```
for( int i = 0; i < len; i ++ )
{
    a[i] = b[i];        // 赋值
}
```

同样，在使用 cin 和 cout 输入与输出数组元素的时候，也只能依次对每个元素进行操作，而不能直接对数组进行操作。下面是一个简单的使用数组的例子，输入 10 个数，计算其总和。如果定义 10 个整型变量，则程序结构非常烦琐，因此这里使用数组，代码非常简单，如示例代码 6.1 所示。

示例代码 6.1

```
#include <iostream>
using namespace std;                    // 使用名称空间 std
#define MAX_SIZE 10                     // 定义一个表示数组长度的宏
int main(int argc, char *argv[])        // 主函数
{
    int arr[MAX_SIZE];                  // 定义数组
    cout<<"请输入 10 个整数"<<endl;        // 输出提示信息
    for( int i = 0; I < MAX_SIZE; i ++ )        // 遍历整个数组
    {
        cin>>arr[i];                    // 输入数据
    }
    int sum = 0;                        // 表示数组元素之和的变量
    for( int i = 0; i < MAX_SIZE; i ++ )        // 遍历整个数组
    {
        sum += arr[i];                  // 求和
    }
    cout<<"10 个元素的和为："<<sum<<endl; // 输出
    system("PAUSE");                    // 等待用户反应
    return EXIT_SUCCESS;                // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 6.3 所示。

```
请输入10个整数
1 2 3 4 5 6 7 8 9 10
10个元素的和为：55
Press any key to continue . . .
```

图 6.3 数组求和结果

在上面代码中，数组长度为 10，其有效下标范围为 0~9。

6.5 数组的缺点

C++中的数组，就是一堆连续变量的集合，其在使用上有如下一些缺点：

- 数组的长度在定义的时候必须确定，不能为变量。
- 数组定义了之后，其长度不能再改变。
- 不能直接用一个数组给另外一个数组赋值。
- 不能向数组中进行插入操作。
- 在使用数组的时候不能得到数组的长度。

因为上面的缺点，数组在使用过程中有很大的限制。因此，C++标准库提供了一个功能强大的数组模板类 vector。这个数组的使用非常灵活，可以克服上述缺点。关于 vector 的使用，将在第 21 章进行详细讲述。

6.6 二维数组

数组是一堆连续变量的集合。如果把一个数组看成一个大的变量，把一堆这样的变量集合在一块，那就成了一个新的数组，这就是二维数组。本节重点介绍二维数组的定义及操作，多维数组的定义和操作可由二维数组类推而得到。

6.6.1 什么是二维数组

前面介绍的数组只有一个下标，称为一维数组，其数组元素也称为单下标变量。一维数组常用来表示线性的数据。而在实际的应用中有的数据用线性结构不好表示，例如下面的矩阵：

```
1 0 0
0 1 0
0 0 1
```

这是一个 3 行 3 列的矩阵。当然可以用一个长度为 9 的数组来存储这些数据，但是却不能表现出数据的逻辑结构。因此 C++语言允许构造多维数组。多维数组元素有多个下标，以标识在数组中的位置，所以也称为多下标变量。二维数组是常见的多维数组。虽然可以定义三维、四维甚至更高维数的数组，但其现实意义不如二维数组明显，所以用得也比较少。

二维数组具有两个下标，可以分别表示行号和列号，因此用二维数组来表示一个矩阵就方便了很多。

矩阵在计算机编程中应用得非常广泛，特别是在计算机图形图像处理中，矩阵更是不可或缺的。很多算法都建立在矩阵的基础上。当然，表示矩阵的方法有很多，可以用一维数组或二维数组，也可以使用向量 vector 以及用户自定义的类。

6.6.2 定义二维数组

同定义一维数组一样，定义二维数组也需要指定数组元素的类型。不同的是，二维数组需要两个常量表达式来表明两个维的长度。二维数组的定义格式为：

```
类型说明符 数组名[常量表达式 1][常量表达式 2];
```

其中“常量表达式 1”表示第一维的长度，“常量表达式 2”表示第二维的长度。例如：

```
01  int arr[3][2];      // 定义了一个二维的 3 行 2 列的数组
```

通常情况下，可以把上面定义的二维数组看做如图 6.4 所示的形式，其中每个单元格代表一个元素。

图 6.4　二维数组

同一维数组一样，二维数组的两个常量表达式也必须是编译时可以计算出的值。

6.6.3 二维数组的初始化

二维数组在定义的时候也可以直接初始化。二维数组在概念上可以看做用一维数组作为数组元素的一维数组，因此其初始化可以看做数个一维数组的初始化。其初始化格式如下：

```
类型说明符 数组名[常量表达式 1][常量表达式 2] =
{
    {值，值，……},
    {值，值，……},
    ……
```

```
};
```

里面的花括号中的值分别用来初始化二维数组的每一行。第一个花括号中的值初始化第一行，第二个花括号中的值初始化第二行，不足的部分用 0 来补齐。因此，对于下面的初始化：

```
int arr[4][4] =
{
    { 1, 2, 3, 4 },
    { 5, 6, 7 },
    { 8, 9 },
};
```

其结果为：

```
1 2 3 4
5 6 7 0
8 9 0 0
0 0 0 0
```

对于二维数组的初始化，如果前面的每一段都提供了足够的初始化值，则花括号可以省略，即可以如下定义：

```
类型说明符 数组名[常量表达式 1][常量表达式 2]={值,值, ……};
```

因此，对于下面的初始化：

```
int arr[3][2] =
{
    {1, 2},
    {3, 4},
    {5, 6}
};
```

也可以这么初始化：

```
int arr[3][2] = {1,2,3,4,5,6};
```

这两种方法相比，分段初始化更清晰，也更灵活，因为每一段都可以只提供部分初始化值。在二维数组初始化的时候，如果为所有的元素都提供了初始值，则第一维的长度可以省略，例如：

```
int arr[3][2] = { 1,2,3,4,5,6};
```

也可以这么写：

```
int arr[][2] = {1,2,3,4,5,6};
```

对于二维数组，只能省略第一维的长度，第二维的长度不能省略。编译器可以根据初始值的个数以及第二维的长度，算出第一维的长度。对于其他更高维数的数

组，同样第一维的长度也是可以省略的，前提是能够根据初始值的个数和其他维数的长度算出第一维的长度。

6.6.4 操作二维数组

二维数组同一维数组一样，数组元素共用同一个名字，使用下标对每个元素进行访问。不过对于二维数组元素来说，要使用两个下标来访问。访问二维数组的格式如下：

```
数组名[下标][下标];
```

同一维数组的访问一样，这两个下标都需要是整型常量或者变量。例如：

```
int arr[3][4];                  // 定义数组
arr[1][2] = 1;                  // 给数组赋值
```

前面已经提到，二维数组可以看做是行、列的排列，对于 arr[1][2]的操作如图 6.5 所示。

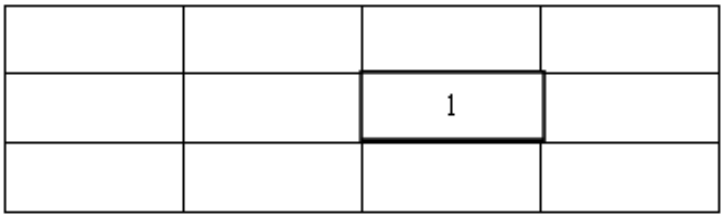


图 6.5　二维数组的操作

第一维的下标代表行的位置，第二维的下标代表列的位置。每一维的下标都从 0 开始。因此 arr[1][2]表示的是第 2 行、第 3 列的元素。下面演示二维数组的输入与输出，如示例代码 6.2 所示。

示例代码 6.2

```
#include <iostream>
using namespace std;                            // 使用名称空间 std
int main(int argc, char *argv[])                // 主函数
{
    int matrix[4][3];                           // 定义二维数组
    cout<<"请依次输入 12 个整数："<<endl;          // 输出提示信息
    for( int i = 0; i < 4; i ++ )               // 遍历第一维
        for( int j = 0; j < 3; j ++ )           // 遍历第二维
            cin>>matrix[i][j];                  // 输入数据
    cout<<"输出数组元素："<<endl;                  // 输出提示信息
    for( int i = 0; i < 4; i ++ )               // 遍历第一维
    {
        for( int j = 0; j < 3; j ++ )           // 遍历第二维
        {
            if( j > 0 ) cout<<" ";              // 数组前面的空格
```

```
            cout<<matrix[i][j];                    // 输出数组元素
        }
        cout<<endl;                                // 换行
    }
    system("PAUSE");                               // 等待用户反应
    return EXIT_SUCCESS;                           // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到工程中，编译并运行，其结果如图 6.6 所示。

图 6.6 二维数组的输入输出结果

上面定义了一个 4 行 3 列的二维数组，并通过循环改变下标来访问数组。

6.6.5 二维数组的存储

虽然可以用二维数组来表示一个矩阵，但系统并不允许程序按照这种方式存储数据。二维数组的行、列表现形式只是便于对二维数组的理解，事实上二维数组的存储和一维数组一样，都是线性的结构，如图 6.7 所示。

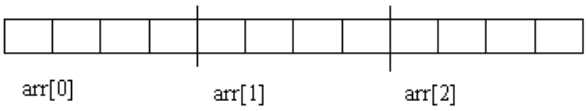


图 6.7 二维数组的线性结构

上图是二维数组的存储结构，可以看出，所有的数组元素都是连续存储的，并没有分段。理解这样的存储结构对于后面学习二维数组的指针有很大的帮助。下面分别定义一个长度为 9 的一维数组和一个 3×3 的二维数组，然后输出其各元素的地址，对比两种方式的区别，程序如示例代码 6.3 所示。

在 C++语言中，使用&符号取得变量的存储地址。

示例代码 6.3

```
#include <iostream>
using namespace std;
int main()
{
    int vec[9] = {1,2,3,4,5,6,7,8,9};
    int matrix[3][3] = {
        {1,2,3},
        {4,5,6},
        {7,8,9}
```

```
    };
    cout<<"一维数组各元素的地址和值分别为："<<endl;
    for( int i = 0; i < 9; i ++ )
        cout<<&vec[i]<<"\t"<<vec[i]<<endl;

    cout<<"二维数组各元素的地址和值分别为："<<endl;
    for( int i = 0; i < 3; i ++ )
    {
        for( int j = 0; j < 3; j ++ )
        {
            cout<<&matrix[i][j]<<"\t"<<matrix[i][j]<<endl;
        }
    }
    return 0;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 6.8 所示。

```
一维数组各元素的地址和值分别为：
0012FF2C        1
0012FF30        2
0012FF34        3
0012FF38        4
0012FF3C        5
0012FF40        6
0012FF44        7
0012FF48        8
0012FF4C        9
二维数组各元素的地址和值分别为：
0012FF00        1
0012FF04        2
0012FF08        3
0012FF0C        4
0012FF10        5
0012FF14        6
0012FF18        7
0012FF1C        8
0012FF20        9
Press any key to continue . . .
```

图 6.8　求数组元素地址结果

从结果中可以看到，一维数组和二维数组在表现形式上不同。而实际上，在内部的存储方式上，两者是相同的。对于程序里定义的二维数组来说，元素 matrix[0][2]和元素 matrix[1][0]是相邻的。在如图 6.7 所示的数组中，编译器在碰到 arr[1][2]的时候，需要计算其存储的真实地址，计算方式为：先得到 arr[1]的偏移量为 4，然后再计算出 arr[1][2]的偏移量为 7，也就是说相对首地址的第 7 个存储位置。

在 C++语言中，对于多维数组，最右边的下标是最先变化的。因此，在使用的时候左边的下标要尽量少变化，以加快访问速度。

6.7 字符串

字符串是个特殊的字符数组。与普通字符数组不同的是，字符串在结尾处有一个字符\0，表示字符串的结束。在 C++中字符串有特殊的初始化方式和专门的处理函数。本节将介绍字符串的特性，在下一节中将介绍专门用于处理字符串的函数。

6.7.1 什么是字符串

在 C++语言中，没有专门的字符串类型，一个字符串，其实就是一个字符数组。不过并不是数组中的所有字符都是字符串的一部分，字符串是以字符\0 表示字符串的结束的。所以在字符数组中，所有在字符\0 之前的字符才是字符串中的有效字符。

字符串也有字面常量，其形式是用双引号包围起来一串字符。例如：

```
"Hello World";
```

前面讲到的字符，用的是单引号，字符串用的是双引号，这点不要弄反了。"a"是一个字符串，而不是一个字符，虽然这个字符串只有一个有效字符。

上面定义了一个字符串，其存储形式如图 6.9 所示。

H	e	l	l	o		W	o	r	l	d	\0

图 6.9 字符串的存储形式

每一个字符串，后面都有一个不可见的字符\0，这个字符是字符串的结束标志。因此上面的字符串有 12 个字符。双引号表示其中的内容是一个字符串，字符串中并不包括双引号。

前面讲字符类型的时候讲过转义字符，比如字符\n 表示换行。字符串中可以包含转义字符，例如：

```
"Hello\nWorld";
```

如果输出上面的字符串，则会分两行输出。" "表示只有空格字符的字符串，而""也是一个字符串，这个字符串只有一个结束标志。字符串还可以分多行定义，例如：

```
"Hello"
"World";
```

上面的定义相当于"HelloWorld"，编译器会自动把分行定义的字符串连接起来。需要注意的是，只有在结束的时候才可以有语句结束的分号“;”，前面都不能有。

字符串的结束标志不需要显式地提供，编译器会自动在后面追加。

6.7.2 定义字符串

用双引号包围的字符串是字符串类型的字面常量，不可以修改。而作为变量的字符串，其实就是字符数组。定义一个字符串其实就是定义一个字符数组，例如：

```
char name[10];          // 定义一个名字是 name 的字符串
```

也可以定义二维的或者多维的字符数组，例如：

```
char names[10][10];     // 定义一个二维字符数组，通常称这样的数组为字符串数组
```

6.7.3 字符串的初始化

同普通数组一样，字符数组也允许在定义的时候直接初始化，例如：

```
char name[10] = {'J', 'a', 's', 'o', 'n'};
```

同样，如果为数组的所有元素提供了初始值，则字符数组的长度可以省略，例如：

```
char name[] = {'J', 'a', 's', 'o', 'n'};
```

这时数组长度被自动定义为 5。

还可以用字符串来作为字符数组的初始值，因此还可以这么写：

```
char name[] = "Jason";
```

上面的语句定义了一个长度为 6 的字符数组，其最后的一个字符为\0。在这种初始化方式下，不需要花括号“{”和“}”。

通常用上面的方法对字符数组进行初始化，而不用单个字符的方式。

6.7.4 操作字符串

字符串作为一个字符数组，同样可以使用下标的方法来操作每一个字符，不过字符串有一些普通数组没有的特性。

前面在介绍数组的时候提到过，cin 和 cout 不可以直接操作数组，但是字符数组是个特例。可以直接用 C++的标准输入 cin 输入整个数组的元素，用标准输出 cout 输出整个字符数组。下面对一个输入的字符串进行反转，其算法是：第一个字符和最后一个字符交换，第二个字符和倒数第二个字符交换，依此类推。假定字符串为 abcdefg，那么其交换过程示意如下：

```
a   < ------>  g
b   < ------>  f
c   < ------>  e
d   保持不变
```

交换之后就成了 gfedcba。

交换到字符 d 时，也就是字符串一半的时候，这时的结果已经是 gfedcba，交换已经完成了，如果继续再交换下去，则又会恢复成交换之前的样子。

该实例的具体代码如示例代码 6.4 所示。

示例代码 6.4

```
#include <iostream>
using namespace std;                              // 使用名称空间 std
int main(int argc, char *argv[])                  // 主函数
{
    char buffer[128];                             // 字符数组
    cin>>buffer;                                  // 输入字符串
    int len = (int)strlen( buffer );              // 取得字符串的长度
```

```
    for( int i = 0; i < len / 2; i ++ )          // 遍历字符串
    {
        char temp = buffer[i];                   // 交换两个字符
        buffer[i] = buffer[len - i - 1];
        buffer[len -i - 1] = temp;
    }
    cout <<"反转之后的字符串为："                  // 输出提示信息
         <<buffer<<endl;
    system("PAUSE");                             // 等待用户反应
    return EXIT_SUCCESS;                         // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 6.10 所示。

图 6.10 字符串反转结果

对于 buffer[0]，其对应的要交换的为 buffer[len-1]，对于第 i 个字符 buffer[i]，其需要交换的为 buffer[len –i –1]。最后来看循环结束的条件：如果字符串的长度 len 为偶数，最后一对需要交换的是 buffer[len/2-1]和 buffer[len/2]；如果字符串的长度 len 为奇数，buffer[len/2]是中间字符，不需要交换，最后一对需要交换的字符是 buffer[len/2-1]和 buffer[len/2+1]。因此用条件 i<len/2 就可以保证交换完全并且没有多余的交换。

6.7.5 字符串的数组

也可以定义二维的字符数组，也就是字符串的数组。例如下面定义可以存储多个人名的数组：

```
char names[10][40];
```

同普通的二维数组一样，字符串数组在定义的时候也可以初始化：

```
char names[10][40] =
{
    "Tom",
    "Mary",
    "Jacky",
    "Jason"
};
```

上面定义了一个长度为 10 的字符串数组，并给前 4 个进行了初始化，剩余部分用空字符串来填充。

不能遗忘两个字符串中间的”,”。

6.8 字符串处理函数

虽然可以对字符数组用字符串进行初始化，可以直接输入、输出字符串，但是在使用字符数组的时候还是有很多不便，比如不能直接对字符数组进行赋值、不能比较两个字符数组的大小等。因此 C++提供了一些函数来处理字符串，下面介绍几个常用的函数。

6.8.1 字符串复制函数 strcpy

字符串是一个字符数组，因此它就有数组的局限性，不能用一个字符串给另外一个字符串赋值，例如：

```
char name[10];
name = "Jason";                    // 错误，不可以这样直接赋值
```

为了方便实现上面的功能，C++标准库中提供了 strcpy 函数，其定义格式为：

```
strcpy( 字符数组名 1, 字符数组名 2 )
```

函数把字符数组 2 中的字符串复制到字符数组 1 中，串结束标志\0 也一同复制。使用例子如下：

```
char str1[128],str2[] = "Hello World";
strcpy( str1, str2 );              // 把 str2 复制到 str1
```

程序员需要自己保证 str1 的长度足够容纳 str2 的内容，如果 str2 的长度超过了 str1 的长度，可能会导致系统崩溃。

其中第二个参数可以直接是一个字符串，因此也可以这样使用：

```
char str[128];
strcpy( str, "Hello World" );      // 直接给 str 赋值
```

在下面的例子中需要用户输入两个字符串，然后把这两个字符串的内容互换，程序如示例代码 6.5 所示。

示例代码 6.5

```
#include <iostream>
using namespace std;                              // 使用名称空间 std
int main(int argc, char *argv[])                  // 主函数
{
    cout<<"请依次输入两个字符串："<<endl;
    char str1[128],str2[128];                     // 字符串变量
    cin>>str1;                                    // 输入
    cin>>str2;
    cout<<"第一个字符串为："<<str1<<endl;
```

```
    cout<<"第二个字符串为："<<str2<<endl;
    cout<<endl<<"交换两个字符串"<<endl;
    char temp[128];                                  // 临时字符串变量
    strcpy( temp, str1 );                            // 交换两个字符串
    strcpy( str1, str2 );
    strcpy( str2, temp );

    cout<<"交换之后的字符串为："<<endl<<endl;
    cout<<"第一个字符串为："<<str1<<endl;
    cout<<"第二个字符串为："<<str2<<endl;
    system("PAUSE");                                 // 等待用户反应
    return EXIT_SUCCESS;                             // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 6.11 所示。

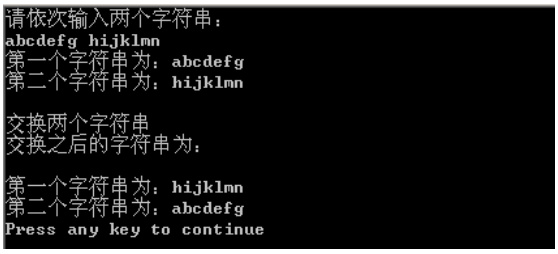


图 6.11 交换字符串结果

strcpy 函数复制的是字符\0 之前的内容（包括这个结束标志），不是整个字符数组的内容。

6.8.2 计算字符串长度函数 strlen

对于字符数组，可以通过 sizeof 来得到其定义的长度，但是这个长度对于字符串是没有意义的。因为通常想要得到的是其有效内容的长度，也就是在字符\0 之前的字符的个数。在 C++标准库中提供了函数 strlen，可以实现这个功能。strlen 函数的格式如下：

```
strlen( 字符数组名 )
```

strlen 将字符串的实际长度作为函数的返回值。使用例子如下：

```
char str[128] = "Hello World";
int len = strlen( str );                    // 取得字符串的长度
```

上面的 len 值为 11，包括里面的空格，但是不包括里面的字符串结束标志。下例中需要用户输入一个字符串，然后用 strlen 函数计算其长度，程序如示例代码 6.6 所示。

示例代码 6.6

```
#include <iostream>
using namespace std;
int main()
{
```

```
    cout<<"请输入一个字符串："<<endl;
    char str[128];
    cin>>str;
    cout<<"字符串的长度为:"<<strlen(str)<<endl;
    return 0;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 6.12 所示。

图 6.12　使用 strlen 函数示例结果

6.8.3 字符串连接函数 strcat

对于两个整数 a 和 b，下面的代码结果是把两个整数相加：

```
int a = 3;
int b = 5;
int c = a + b;
```

对于字符串，其加法的意义应该是把两个字符串连接起来，对于下面两个定义：

```
char str1[128] = "Hello World";
char str2[128] = "C++ Programming";
```

如果想把两个字符串连接起来，直接用+是不可以的，在 C++标准库中提供了一个可以实现这个功能的函数，这个函数就是 strcat。函数格式如下：

```
strcat ( 字符数组名 1, 字符数组名 2 )
```

函数把字符数组 2 中的字符串连接到字符数组 1 中字符串的后面，并删去字符串 1 后的串标志\0。本函数返回值是字符数组 1 的首地址。使用例子如下：

```
char str1[128] = "Hello World";
char str2[128] = "C++ Programming";
strcat( str1, str2 );                    // 连接字符串
```

上面的代码把 str2 连接到 str1 的后面，现在 str1 为“Hello WorldC++ Programming”。下例需要用户输入两个字符串，然后把它们连接起来，程序如示例代码 6.7 所示。

示例代码 6.7

```
#include <iostream>
using namespace std;                     // 使用名称空间 std
int main(int argc, char *argv[])         // 主函数
{
    cout<<"请输入两个字符串:"<<endl;              // 输出提示信息
    char str1[128];                              // 字符串变量
    char str2[128];
    char buffer[128];
    cin>>str1>>str2;                             // 输入字符串
    strcpy( buffer, str1 );                      // 复制
    strcat( buffer, str2 );                      // 连接
```

```
    cout<<"新生成的字符串为:"<<buffer<<endl;      // 输出结果字符串
    system("PAUSE");                              // 等待用户反应
    return EXIT_SUCCESS;                          // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 6.13 所示。

图 6.13 strcat 函数使用示例结果

6.8.4 字符串比较函数 strcmp

对于两个字符串，其比较是有意义的，但是字符数组不能直接进行比较，因此在 C++标准库中提供了对字符串进行比较的函数，函数格式如下：

```
strcmp( 字符数组名 1，字符数组名 2 )
```

函数按照 ASCII 码顺序比较两个数组中的字符串，并由函数返回值返回比较结果：

- 字符串 1 = 字符串 2，返回值 = 0。
- 字符串 2>字符串 2，返回值>0。
- 字符串 1<字符串 2，返回值<0。

当两个字符串相等的时候，返回值为 0，因此，在比较两个字符串是否相等的时候，应当像下面的代码这么写：

```
if( strcmp( str1, str2 ) == 0 )    // 判断两个字符串是否相同
```

而不能这么写：

```
if( strcmp( str1, str2 ) )         // 错误的写法，结果跟希望正好相反
```

下例需要用户输入两个字符串，然后对它们进行比较，程序如示例代码 6.8 所示。

示例代码 6.8

```
#include <iostream>
using namespace std;                              // 使用名称空间 std
int main(int argc, char *argv[])                  // 主函数
{
    cout<<"请输入两个字符串:"<<endl;              // 输出提示信息
    char str1[128];                               // 字符串变量
    char str2[128];
    cin>>str1>>str2;                              // 输入字符串
    int res = strcmp( str1, str2 );               // 比较
    cout<<"比较的结果:"<<endl;                    // 输出比较结果
    cout<<str1;
    if( res == 0 ) cout<<"==";
    if( res > 0 ) cout<<">";
    if( res < 0 ) cout<<"<";
    cout<<str2<<endl;
```

```
    system("PAUSE");                                    // 等待用户反应
    return EXIT_SUCCESS;                                // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 6.14 所示。

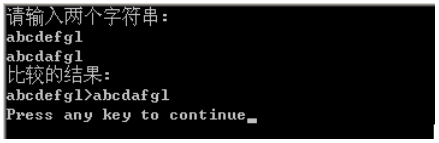


图 6.14　使用 strcmp 函数示例结果

在 C++的标准库中提供了一个 string 类。使用这个类，可以直接进行赋值、比较、计算长度和连接等操作。

6.9 综合实例

本节将详细讲解使用数组和字符串的综合实例。对于每个实例，都将详细讲解实例的背景和具体的代码。

6.9.1 数组元素排序

在这个实例程序中，需要输入 10 个数，按照从大到小的顺序重新排列，然后输出到屏幕。这里重点是展示怎么使用数组，因此排序使用了最简单的“冒泡排序”。

“冒泡排序”是一种最简单的排序方法，冒泡排序的基本概念是：依次比较相邻的两个数，将大数放在前面，小数放在后面。即首先比较第 1 个和第 2 个数，将大数放在前面，小数放面后面，然后比较第 2 个和第 3 个数，将大数放在前面，小数放在后面，如此继续，直至比较最后两个数，将大数放在前面，小数放在后面，此时第一趟排序结束，在最后面的数必定是所有数中的最小数。重复以上过程，第二趟结束，在倒数第二个位置得到一个新的最小数。如此下去，直至最终完成排序。

由于在排序过程中总是大数往前放，小数往后放，相当于气泡往上升，所以称为“冒泡排序”。下面用一个简单的例子演示该算法的工作过程，用一个简单的数组：

```
1 5 9 6 7
```

第一趟排序的过程如下：

step 1　比较第 1 个数和第 2 个数，1 小于 5，交换 1 和 5，得到结果如下：

```
5 1 9 6 7
```

step 2　比较第 2 个数和第 3 个数，1 小于 9，交换 1 和 9，得到结果如下：

```
5 9 1 6 7
```

step 3　比较第 3 个数和第 4 个数，1 小于 6，交换 1 和 6，得到结果如下：

```
5 9 6 1 7
```

step 4　比较第 4 个数和第 5 个数，1 小于 7，交换 1 和 7，得到结果如下：

```
5 9 6 7 1
```

到此为止，已经到了最后一个，因此第一趟结束，最小的 1 浮到了最后。同样，第二趟结束之后，结果如下：

```
9 6 7 5 1
```

第三趟结束之后，结果如下：

```
9 7 6 5 1
```

此时，已经达到了从大到小排列的目的，以后的比较将不再交换。

可以用一个变量来记录在比较的过程中是否发生了交换，如果某一趟一直没有交换，则该排序已经完成。

上面就是“冒泡排序”的工作过程。本实例对应的程序如示例代码 6.9 所示。

示例代码 6.9

```
#include <iostream>
using namespace std;                                    // 使用名称空间 std
int main(int argc, char *argv[])                        // 主函数
{
    int const len = 10;                                 // 数组长度
    int arr[len];                                       // 数组变量
    cout<<"请输入 10 个元素："<<endl;                    // 输出提示信息
    for( int i = 0; i < len; i ++ )                     // 依次输入数组元素
    {
        cin>>arr[i];
    }
13
    for( int i = 0; i < len; i ++ )                         // 遍历数组
    {
        for( int j  = 0; j < len – i – 1; j ++ )            // 遍历
        {
            if( arr[j] < arr[j+1] )
            {
                int temp = arr[j];                          // 交换两个数
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    cout<<"输入的数据，从大到小排序的结果为："<<endl;
    for( int i = 0; i < len; i ++ )                         // 遍历数组
    {
        cout<<arr[i]<<" ";                                  // 输出数组元素
    }
    cout<<endl;
    system("PAUSE");                                        // 等待用户反应
    return EXIT_SUCCESS;                                    // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 6.15 所示。

图 6.15　数组排序实例结果

在上面的代码中，数组长度为 10，其有效下标范围为 0~9。从第 13 行开始到第 24 行的双层循环是本程序的重点，外层循环控制排序的趟数，内层循环保证最小的数浮到后面。

6.9.2 输出杨辉三角

本实例的功能是输入一个整数，输出其杨辉三角。杨辉三角是如下所示的特征数，它常被用在二项式的展开计算上。

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

其每一行的数据，都可以由前一行前后两个数相加得到。

假如已经得到第 4 行的数据：

```
1 3 3 1
```

下面由第 4 行计算第 5 行的值，其第 5 行的值分别为：

```
1 1+3 3+3 3+1 1
```

也就是：

```
1 4 6 4 1
```

输出杨辉三角程序如示例代码 6.10 所示。

示例代码 6.10

```
#include <iostream>
using namespace std;                          // 使用名称空间 std
int main(int argc, char *argv[])              // 主函数
{
    cout<<"请输入杨辉三角的行数（1~10）:"<<endl;
    int n;                                    // 行数
    cin>>n;
    int a[20] = {0}, b[20] = {0};
    for( int i = 0; i < n; i ++ )             // 处理 n 行
    {
        // 第一个和最后一个都为 1
        b[0] = 1;
        b[i] = 1;
        for( int j = 1; j < i; j ++ )
        {
            b[j] = a[j] + a[j-1];
        }
        // 输出前导的空格
        for( int j = 0; j < n - i - 1; j ++ )
```

```
            cout<<' ';
        for( int j = 0; j <= i; j ++ )
        {
            if( j > 0 ) cout<<' ';
            cout<<b[j];
        }
        cout<<endl;
        // 完成本行的计算，把结果存入 a 中，重新计算下一行
        for( int j = 0; j <= i; j++ )
            a[j] = b[j];
    }
    system("PAUSE");                        // 等待用户反应
    return EXIT_SUCCESS;                    // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 6.16 所示。

图 6.16 输出杨辉三角实例结果

6.9.3 字符串处理函数的使用

下面程序中，用一个二维的字符数组来存储输入的名字字符串，排除掉重复的字符串，并保证剩余的字符串从小到大排列，每一次输入都要调整现有的数组。调整的方法如下：从头到尾和所有的字符串进行比较，如果输入的字符串大，则继续向后比较；如果有相同的字符串，则忽略当前输入；如果输入的字符串比当前字符串小，则输入的字符串应该插入到当前字符串的位置，其余的字符串都需要向后移。程序如示例代码 6.11 所示。

示例代码 6.11

```
#include <iostream>
using namespace std;
int main()
{
    char names[20][50];
    cout<<"请输入名字，以#结束："<<endl;
    int count = 0;
    for(;;)
    {
        char temp[50];
        cin>>temp;
        if( strcmp( temp, "#" ) == 0 ) break;
        int i;
        for( i = 0; i < count; i ++ )
        {
            int res = strcmp( temp, names[i] );
            if( res < 0 )
            {
                // 找到要插入的位置，把后面的名字向后移动，空出插入位置
```

```
                for( int j = count; j > i; j -- )
                {
                    strcpy( names[j], names[j-1] );
                }
                strcpy( names[i], temp );
                count++;
                break;
            } else if( res == 0 )
            {
                // 发现相同字符串，不做处理
                break;
            }
        }
        if( i == count )
        {
            strcpy( names[count], temp );
            count++;
        }
    }
    cout<<"按照字符顺序，从小到大为（除去相同名字）: "<<endl;
    for( int i = 0; i < count; i ++ )
    {
        cout<<names[i]<<endl;
    }
    return 0;
}
```

建立一个控制台工程，将上述代码复制到源文件中，编译并运行，结果如图 6.17 所示。

```
请输入名字，以#结束：
Tom
Jason
Jack
Andy
Bart
Howard
House
Jane
Jason
#
按照字符顺序，从小到大为（除去相同名字）：
Andy
Bart
House
Howard
Jack
Jane
Jason
Tom
Press any key to continue . . .
```

图 6.17　使用字符串处理函数实例结果

6.10 小结

C++中的数组是一个连续的具有相同类型的数据元素的集合，这些元素连续存储，没有单独的名字，只能根据在数组中的位置来访问。当数组中的元素是一个数组的时候，这就是个二维数组。二维数组虽然在存储上仍然是线性的，但是可以表现为一个矩阵。

当数组的类型为字符类型的时候，数组就是字符数组，这个字符数组可以用来存储一个字符串。字符串是一个字符的集合，其最后暗含了一个\0 结束标志。在 C++中提供了 strcpy，strlen，strcat 和 strcmp 等处理函数对字符串进行处理。

◆ ◆ ◆

第 7 章　指针与引用

本章包括

- 指针的概念
- 指针与数组的关系
- 通过指针管理内存的方法
- 常量指针与指针常量
- 引用的定义和使用
- 指针与引用的区别

指针与引用也是程序中的数据类型。与前面章节中讲过的整型、浮点型不同，指针和引用是复合数据类型。该类型的数据在定义时不仅要说明是指针还是引用，还要说明是什么类型数据的指针或引用。指针和引用的用法比较特殊，而且也非常灵活。如果使用得当，可以提高软件的品质；如果使用不当，也会带来很多问题。本章就详细讲述指针与引用的概念和使用。

7.1　什么是指针

计算机中的数据都是存储在内存中的，内存中的这些数据都有地址，指针就是记录这些地址的变量，指针的类型表示指针指向的地址存储的数据的类型。

7.1.1　指针与内存的关系

在计算机中所有的数据都存放在内存中，存放的最小单位是位（bit）。一个 bit 就是一个二进制的最小单元，其值只能为 0 或者 1。8 个 bit 是一个字节（byte），字节是计算机中常用的基本单位，一般的计量都是以字节为单位的，比如 char 类型数据占 1 个字节，int 类型数据占 4 个字节，double 类型数据占 8 个字节等。

int 类型数据在内存中所占的字节数，跟使用的硬件系统、编译器有关。在程序中可以使用 sizeof 运算符检测其所占内存的大小。一般来讲，在 32 位系统下一个 int 类型数据占 4 个字节。

在操作系统中，这些内存单元都被编上了号。通过这些编号，可以快速地找到该内存单元，这些编号就是所谓的内存地址。在 C++中可以用一个指针类型的变量来存放内存地址，这个变量被称为指针变量，或简称指针。指针与变量的关系如图 7.1 所示。

在图中，x 是一个整型变量，其值为 3，存储在起始地址为 0x011A 的 4 个存储单元中。p 为一个整型指针变量，其值为 0x011A，也就是变量 x 的内存地址。在这种情况下，可以说 p 是一个指向变量 x 的指针。而一个变量所占内存的第一个字节的地址，就是该变量的地址。

指针的值是一个内存地址。这个地址可以是变量的地址，也可以是数组的首地址，甚至是函数的地址。在本章中主要讲述指向变量的指针和指向数组的指针，而指向函数的指针，还有指向类成

员变量的指针等复杂的情况将在后面的章节中讲述。

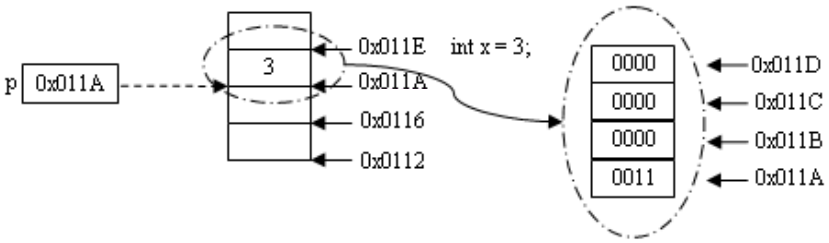


图 7.1 指针与变量的关系

7.1.2 定义指针变量

指针是复合类型的数据。定义指针变量时不仅要指明该变量是个指针，还要说明这是一个什么类型的指针，即指针所指数据的类型。因此，定义指针变量包括三个内容，从前到后依次为：

- 指针所指向数据的类型。
- 指针类型修饰符“*”。
- 指针标识符，即指针名。

其一般形式为：

```
类型标识符 * 指针名;
```

其中，“*”表示这是一个指针变量，变量名为定义的指针变量名，类型标识符表示本指针变量指向数据的类型。例如：

```
int * pt;
```

表示 pt 是一个指针变量，其值为某个整型变量的地址，或者说 pt 指向一个整型变量。至于到底指向哪一个整型变量，可以通过将某个变量的地址值赋给 pt 来决定。下面还有几个例子：

```
int * pt;       // 定义一个指向整数的指针变量
char* pc;       // 定义一个指向字符的指针变量
float *pf;      // 定义一个指向浮点数的指针变量
```

同定义普通变量一样，指针变量也可以连续定义。如一次定义两个整型指针变量：

```
int  *p1, *p2;
```

但是需要注意的是，每一个指针变量前面的指针修饰符“*”都是必不可少的，否则该变量就不是一个指针，而是一个普通类型的变量。例如下面的变量定义语句：

```
int  * p1, p2;
```

这不是定义两个指针变量，而是定义一个指针变量 p1 以及一个整型变量 p2，因为 p2 的前面没有指针修饰符“*”。

7.1.3 使用指针指向数据

指针变量作为一个变量，跟普通的变量一样，在使用之前不仅要定义说明，还必须赋予具体的值，即让指针指向某个具体的数据。如果使用没有赋值的指针，可能会造成系统混乱，甚至程序崩溃。指针的值必须是一个地址，且必须保证其有效性。在 C++语言中，使用取地址运算符“&”来取得一个变量的地址，其语法如下：

```
&变量名;
```

如“&x”表示变量 x 的地址，“&y”表示变量 y 的地址。取地址的变量必须是已经存在的变量。取地址运算符“&”只能应用于内存中存在的数据，如变量、数组元素等，不能用于表达式、常数或者寄存器（register）变量。

对于一个整型的指针变量 p，如果要让其指向整型变量 a，可以在定义 p 时将其初始化为 a 的地址，或者在定义 p 之后，将 p 赋值为 a 的地址。这两种方法是等价的，不过推荐使用第一种方法，这样可以保证一开始指针就是有效的，否则，未经初始化的指针将指向一个不确定的内存地址，操作这样的指针是非常危险的。指针的定义、初始化和赋值如下：

```
int x;              // 定义一个整型变量
int * p1 = &x;      // 定义一个整型指针，并用变量的地址初始化指针
……
int * pt;           // 定义一个整型指针，未初始化
pt = &x;            // 将指针赋值为某个变量的地址
```

在上面指针的定义中，指针变量是 pt 而不是*pt，因此要给指针赋值，等号左边是 pt 而不是*pt。另外，不允许把一个无效的地址，比如数字，赋给指针。指针变量和一般的变量是一样的，存放的值是可以改变的，也就是说可以改变其指向。

```
int a;
int b;
int * p = &a;       // 指针 p 指向变量 a
p = &b;             // 修改 p，使 p 指向变量 b
```

一个指针变量，只能指向同类型的变量。比如一个整型指针只能指向整型变量，而不能指向其他类型的变量。因此不能用其他类型变量的地址初始化指针变量，或者为其赋值。

7.1.4 获取被指数据

在程序中，有时需要知道某个指针所指的数据，此时就应当使用间接引用运算符“*”。与取地址运算符“&”相反，间接引用运算符“*”作用于指针，得到指针所指向的变量，其语法如下：

```
*指针名;
```

例如，在下面的程序中：

```
int  x = 123;
```

```
int * p = &x;
```

用变量 x 的地址初始化指针 p，即让指针 p 指向变量 x。除了直接访问变量 x 之外，也可以使用*p 来间接地访问 x。因此，

```
int a = *p;     // p = &x;
```

也就相当于：

```
int a = x;
```

下面的两个赋值语句也是等效的，都可以将变量 x 赋值为 0。

```
*p = 0;
x = 0;
```

通过指针来访问变量，虽然其效果和直接访问变量一样，但指针毕竟是一种间接的方式，其速度略慢于直接访问。不过指针也给程序的开发带来了很大的灵活性，其原因在于指针也是一个变量，可以在运行时修改其指向，从而达到“使用一个指针，访问多个变量”的目的。

在对指针使用间接运算符“*”之前，必须让指针指向一个合法的地址，否则程序的行为将是难以预料的，甚至会导致系统崩溃。原因是一个非法的地址可能是某个重要的数据地址，或者是某一条程序指令的地址，它们如果被修改，则程序的行为无法预料。例如下面的指针用法就是不正确的：

```
int  * p = NULL;            // 空指针，不指向任何空间
*p = 0;                     // 会导致程序崩溃
int  * q;                   // 定义一个指针变量，但没有初始化，指针指向一个无效地址
*q = 123;                   // 使用未初始化的指针变量，程序行为不可预料
```

值得指出的是，间接引用运算符“*”和取地址运算符“&”的作用正好相反，对一个变量取地址，并对结果使用间接运算符“*”，将得到变量。例如：

```
int a;
*(&a) = 0;                  // 就相当于 a = 0;
```

下面是一个完整的例子，演示了如何在指针和变量之间建立联系，以及如何使用指针，具体代码如示例代码 7.1 所示。

示例代码 7.1

```
#include <iostream>
using namespace std;
int main()
{
    int a = 0;
    int * p = &a;                           // 指针 p 指向了变量 a
    cout<<"p 指向的地址为:"<< p <<endl;
    *p = 1;                                 // 修改指向变量的值
    cout<<"修改了*p 为 1 之后，a 的值为:"<<a<<endl;
    int b = *p;                             // 取得指针指向的变量
    cout<<"b 的值为:"<<b<<endl;
    int * q = p;                            // 定义另外一个指针
    *q = 2;                                 // 通过指针 q 修改指向变量的值
```

```
    cout<<"修改了*q 为 2 之后，a 的值为："<<a<<endl;
    return 0;
  }
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 7.2 所示。

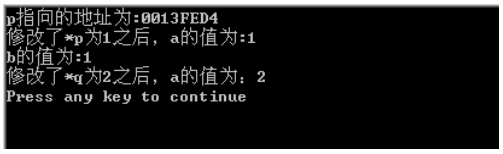


图 7.2　指针与变量示例结果

在程序的第 7 行定义了一个整型的指针 p，并同时使其指向变量 a。因此在第 9 行修改*p 的时候，其实是间接地修改了 a 的值。在第 11 行中把*p 赋给 b，其实也就是把 a 初始化为 b。在第 13 行中定义另外的整型指针变量 q，并使其和 p 有同样的值，因此 q 也指向了变量 a，等同于指针 p。

7.1.5　指针的运算

指针变量也有加减运算。指针可以加减某个整型数，指针与指针可以相减，但指针与指针相加是没有意义的。另外，指针的乘除也没有意义。指针的值是一个内存地址，而一个内存地址可以用一个整型数表示，因此指针的运算可以看做整型数间的运算。不过，指针运算有其特殊性。如果某个数据占据多个字节，那么该数据的指针只能指向其起始地址，即第一个字节的地址，指向中间某个字节是没有意义的。

因此 C++标准规定：指针加减某个整数，其效果等同于将指针移动整数个变量大小。假设一个整型指针 p 指向整型变量 x（占用 4 个字节的内存），而 x 的地址是 1000，则 p+1 指向的地址就是 1004，p-3 指向的地址就是 988。一般来讲，指针加减 n，就相当于指针的地址值加减 n*sizeof（数据类型）。例如，一个 short 型（2 字节大小）指针加减整型数的运算如图 7.3 所示。

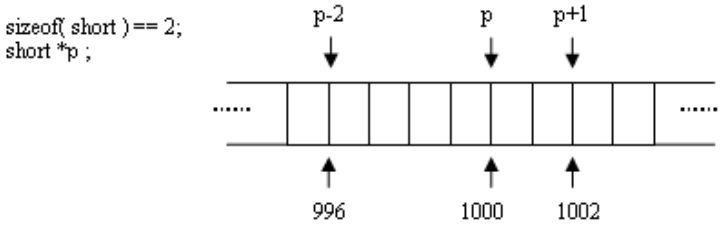


图 7.3　short 型指针加减整数的运算

取地址运算符“&”只能用于已经定义的变量（已分配内存），而不能用于表达式。但间接引用运算符没有这个限制，当然表达式的结果必须是某个内存地址。例如可以使用“*(p + 1)”、“*(p－2)”等求得指针移动后指向的数据。

指针与指针相减表示两个指针间可以存储的变量的个数。假设两个 short 型指针 p 和 q 所指的地址分别是 1000 和 996，则 p-q 的值就是 2，即两个 short 型变量。下面的程序可以检验 int 型指针和 double 型指针加减整数以及指针相减的结果，程序如示例代码 7.2 所示。

示例代码 7.2

```
#include <iostream>
using namespace std;
int main()
{
    cout<<"int 类型的指针"<<endl;
    int a;
    int * pi1 = &a;
    int * pi2 = pi1 + 1;
    cout<<"pi1 = "<<pi1<<endl;
    cout<<"pi2 = pi1 + 1, pi2 = "<<pi2<<endl;
    cout<<"pi2 - pi1 = "<<pi2 - pi1<<endl;
    cout<<"double 类型的指针"<<endl;
    double f;
    double * pf1 = &f;
    double * pf2 = pf1 + 1;
    cout<<"pf1 = "<<pf1<<endl;
    cout<<"p22 = pf1 + 1, pf2 = "<<pf2<<endl;
    cout<<"pf2 - pf1 = "<<pf2 - pf1<<endl;
    return 0;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 7.4 所示。

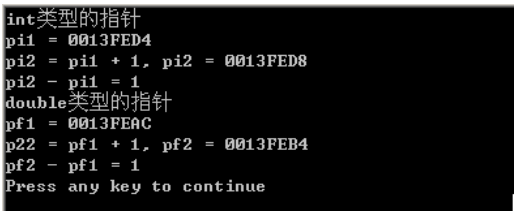


图 7.4 指针的运算结果

在上面的例子中，指针的类型是 int，因此对指针加 1，移动的是 4 个字节，而 double 类型的指针加 1 移动的是 8 个字节。两个指针的减法，结果是两个地址之间可存放的变量的个数。

7.2 指针与数组

数组中的元素，除了用数组名加下标的方法进行访问外，也可以用指针访问，而且用指针访问数组形式更加简单，使用也更加灵活。

7.2.1 指向数组的指针

数组是同类型数据的集合，集合中的元素按照顺序在内存中连续排列。用指针指向数组其实就是让指针指向这段连续内存的首地址，也就是数组中第一个元素（下标为 0）的地址。定义一个指向数组的指针变量同定义普通指针没有什么不同，即定义一个跟数组元素类型相同的指针即可，语法如下：

```
类型标识符 * 指针名;
```

例如，在下面的代码中就分别定义了两个指针，并指向同一个数组：

```
int arr[10];                // 定义一个整型数组
int * p ;                   // 定义一个与数组元素类型相同的指针
p = &arr[0];                // 让指针指向数组
int * q = &arr[0];          // 定义一个指针，并指向数组（用数组首地址初始化指针）
```

除了将第一个元素的地址赋给指针外，用指针指向数组也可以直接使用数组名。在 C++程序中，数组名代表的就是数组的首地址（这也是为什么只能给数组元素赋值，而不能直接对数组赋值的原因）。因此，定义一个指针并指向一个数组就可以使用如下的代码：

```
int arr[10];                // 定义一个数组
int *p;                     // 定义一个跟数组元素类型相同的指针
p = arr;                    // 用指针指向数组
int *q = arr;               // 定义一个指针，并指向一个数组
```

既然数组名代表的是数组的首地址，那么数组名也可以当做一个指针使用。对数组名使用间接引用运算符“*”就可以得到数组中的第一个元素。也就是说对“*数组名”的操作，等同于对“数组名[0]”的操作，对两者进行赋值、计算等效果是一样的。

在这里，虽然 p，&arr[0]，arr 都代表了同一个地址，但是它们之间是有区别的，p 是一个变量，而&arr[0]和 arr 则是两个表示地址的常量，可以修改 p 的值，但是却不可以修改&arr[0]和 arr 这两个常量的值。

7.2.2 使用指针访问数组

数组是一段连续的内存，指针可以指向数组，而且可以通过加减整数来移动指针。所以，可以通过指针来访问数组，即数组中的元素。指针指向数组的首地址，进而可以移动指针到指定的元素。也可以一次只增加 1，从而达到遍历整个数组的目的。

使用指针访问数组，同用下标访问数组的效果是一样的。例如一个指向数组 arr 首地址的指针 p，访问第 i+1 个元素（下标为 i），可以用*(p + i)，也可以用 arr[i]，这两种方法是等价的。由于数组名代表的是数组的首地址，所以也可以用*(arr + i)来访问第 i+1 个元素，如图 7.5 所示。

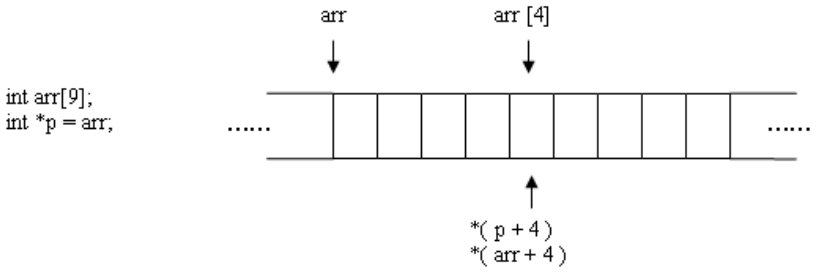


图 7.5 用指针访问数组

使用指针访问数组时也要注意不要越界，即保证指针指向数组第一个到最后一个元素，超出这个范围将导致不可预料的后果。下例将一个数组的元素进行反转，即将第一个元素放到最后，第二个元素放到倒数第二个位置，依此类推，程序如示例代码 7.3 所示。

示例代码 7.3

```
#include <cstdlib>
#include <iostream>
using namespace std;                                  // 使用名称空间 std
int main(int argc, char *argv[])                      // 主函数
{
    cout<<"——反转数组——"<<endl;                        // 输出提示信息
    cout<<endl;

    int ary[10] = { 0, 1, 2, 3, 4,                    // 定义数组
                    5, 6, 7, 8, 9 };

    cout<<"反转前："<<endl;
    for( int i=0; i<10; i++ )                         // 输出反转前的数组
    {
        cout<<ary[i]<<" ";
    }
    cout<<endl;

    int *p = ary;                                     // 指向数组头的指针
    int *q = p + sizeof( ary ) / sizeof( ary[0] ) - 1; // 指向数组尾的指针

    while( p < q )                                    // 只要两个指针不相遇
    {
        int t = *p;                                   // 交换两个指针所指元素的值
        *p = *q;
        *q = t;

        p++;                                          // 前面的指针递增
        q--;                                          // 后面的指针递减
    }

    cout<<"反转后："<<endl;
```

```
    for( int i=0; i<10; i++ )                    // 输出反转后的数组
    {
        cout<<ary[i]<<" ";
    }
    cout<<endl;

    cout<<endl;
    system("PAUSE");                             // 等待用户反应
    return EXIT_SUCCESS;                         // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 7.6 所示。

图 7.6　反转数组结果

在上面的程序中，分别对两个指针施行自加和自减操作，从而实现指针的移动。并且前指针往后移，后指针往前移，两个指针指向的数据正好是需要交换的两个元素。

7.2.3 指向字符串的指针

在 C++程序中字符串是用字符数组表示和存储的。既然用指针可以访问数组，自然用指针也可以访问字符串。不过字符串有一个特殊的结尾字符\0，所以在用指针访问字符串时需要特别注意。字符串的指针就是一个 char 类型的指针（宽字符串用 wchar_t 类型的指针）。用指针指向字符串，同指向数组没有什么区别，要么在定义指针后给指针赋值，要么在定义时就初始化。例如：

```
char str[] ="Hello World";      // 定义一个字符串
char * p = str;                 // 定义一个字符指针，并初始化为字符串的首地址
char * q;                       // 定义一个字符指针
q = str;                        // 赋值为字符串首地址
```

上面的定义方法，还可以简单地这么写：

```
char * pstr = "Hello World";    // 定义一个字符串指针
```

这种定义方法从表面上看好像违背了指针的概念：指针就是一个指向地址的变量，在这个地址还没有确定的时候，不能给这个地址里面赋值。但其实这种写法是正确的，原因是在 C++中这是一种特殊的情况，这个字符串“Hello World!”被定义在 C++的常量存储区中，指针 pstr 是一个指向常量存储区中一块内存的指针。

常量存储区的内容是不可以修改的，因此对于上面定义的字符串 pstr，不可以修改其内容。

在对字符串进行赋值的时候，初学者很容易犯一个错误——直接给字符串的指针赋值。但正像不能直接给数组赋值一样，也不能直接给字符串的指针赋值。

```
char buffer[16];            // 定义一个字符数组
char * p = buffer;          // 定义一个字符指针，并指向字符数组
p = "Hello World";          // 字符指针指向字符串常量。注意，这并不能为字符数组赋值
```

在上述程序中，初学者的目的或许是要把字符串“Hello World”存储在数组 buffer 中，但实际的运行效果却不是这样的。在赋值前，字符指针和字符串的关系如图 7.7 所示。

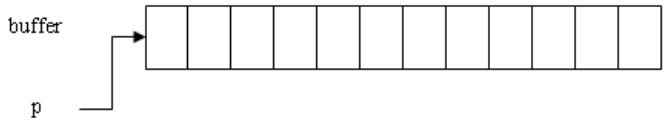


图 7.7 字符指针指向字符数组

在赋值后，字符指针和字符串的关系如图 7.8 所示。

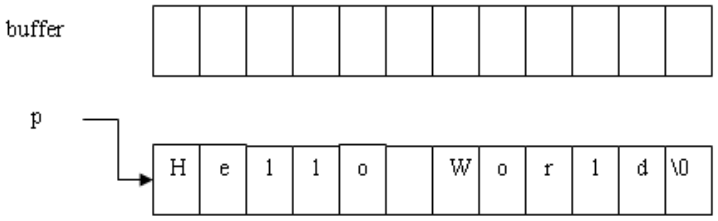


图 7.8 赋值后字符指针和字符串的关系

其实赋值语句只是让 p 指向了一个常量字符串，而没有达到修改 buffer 的目的。正确的做法是使用 C++的库函数 strcpy，进行字符串复制：

```
strcpy( p, "Hello World" );
```

调用该函数之后，结果如图 7.9 所示。

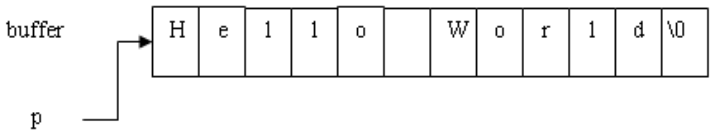


图 7.9 使用字符串复制函数的结果

这个函数改变的是指针所指向内存的值，因此 p 仍旧指向 buffer，而 buffer 里面的内容已经修改。当使用数组名来访问字符串的时候，因为数组名是一个地址常量，不能被修改，因此只能采用改变下标的方法来依次访问字符串的元素。可是当使用指针的时候，因为指针本身可以通过自增、自减等操作来移动。因此使用指针访问字符串非常灵活方便。看下面的代码：

```
while( *pSrc++ = *pDst++ );
```

这句简单的代码，就是前面提到的字符串复制函数 strcpy 的关键部分。pSrc 和 pDst 是两个字符指针。因为字符串的结束标志是\0，其值也就是 0，因此可以作为结束 while 循环的条件。++操作保证了每次赋值之后同时移动到字符串的下一位。

下面的代码将数组中小写的字母转换为大写字母，这需要从头到尾处理整个字符数组，函数 strlen 可以得到需要处理的字符数组的长度。对于一个字符 c，如果其值在 a 和 z 之间，那么就是小写字母。要把一个小写字母变为大写字母，需要在原来的值上增加 A-a（相减），也就是小写字母到大写字母的距离。这里使用了 4 种不同的方法，其结果都是把一个字符数组中的小写字母变为大写字母并输出。程序如示例代码 7.4 所示。

示例代码 7.4

```
#include <cstdlib>
#include <iostream>
using namespace std;                                  // 使用名称空间 std
int main(int argc, char *argv[])                      // 主函数
{
   cout<<"——字符串大小写转换——"<<endl;
   cout<<endl;

    cout<<"=======使用下标法=================="<<endl<<endl;
    char str1[]="hello world";
    cout<<"原来的字符串是：" <<str1<<endl;
    int len = (int)strlen(str1);                      // 获取字符串长度
    for( int i = 0; i < len; i ++ )                   // 遍历字符串
    {
        if( str1[i] >= 'a' && str1[i] <= 'z' )        // 如果是小写字符
        {
            str1[i] += ('A' - 'a');                   // 转换
        }
    }
    cout<<"大写的字符串是：" <<str1<<endl<<endl;       // 输出转换后的结果
    cout<<"=======通过数组名得到数组元素======"<<endl<<endl;
    char str2[]="hello world";
    cout<<"原来的字符串是：" <<str2<<endl;
    for(int i = 0; *(str2+i); i++ )                   // 遍历字符串
    {
        if( *(str2+i) >= 'a' && *(str2+i) <= 'z' ) // 如果是小写字符
        {
            *(str2+i) += ('A' - 'a');                 // 转换
        }
    }
    cout<<"大写的字符串是：" <<str2<<endl<<endl;
    cout<<"=======通过指针得到元素地址========"<<endl<<endl;
    char str3[]="hello world";
    cout<<"原来的字符串是：" <<str3<<endl;
    char * p = str3;                                  // 用指针指向字符串
    for(int i = 0; *(p+i); i++ )                      // 遍历字符串
    {
        if( *(p+i) >= 'a' && *(p+i) <= 'z' )          // 如果是小写字符
```

```
        {
            *(p+i)+= ('A' - 'a');                    // 转换
        }
    }
    cout<<"大写的字符串是："<<str3<<endl<<endl;
    cout<<"=======移动指针====================="<<endl<<endl;
    char str4[] = "hello world";
    cout<<"原来的字符串是："<<str4<<endl;
    p = str4;                                        // 用指针指向字符串
    for( ; *p ;  )                                   // 遍历字符串
    {
        if( *p >= 'a' && *p <= 'z' )                 // 如果是小写字符
        {
            *p += ('A' - 'a');                       // 转换
        }
        p++;
    }
    cout<<"大写的字符串是："<<str4<<endl<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

建立一个控制台工程，把上面 4 种方法的代码复制到源文件中，编译并运行，其结果如图 7.10 所示。

图 7.10　字符串转换结果

在第一个程序中，用的是下标法，arr[i]就是第 i 个元素；在第二个程序中，利用数组名计算数组元素的地址；第三个程序和第二个类似，不同的是用了另外的指针来指向数组的首地址，然后利用这个地址计算数组元素的地址；第四个程序使用 p++来改变当前的指针 p，使 p 指向下一个元素。

当操作符*和操作符++同时作用于一个指针的时候，要注意操作符的优先级，++操作符的优先级高于*操作符，因此*p++等价于*(p++)。

7.3 指针与动态内存分配

指针的值是个地址，这个地址指向内存中的某一个区域。在使用指针的时候，需要保证指向地址的有效性。因此，了解变量在内存中的存储方式，对于理解指针的生存周期有着非常重要的作用。在 C++中，内存分为 5 个不同的区，不同区上的内存有不同的特点。

7.3.1 程序中内存的分配方式

在 C++程序中内存分为 5 个区，分别是栈、堆、自由存储区、全局/静态存储区和常量存储区。程序中的各种数据都存储在这些内存区域中。

栈区由编译器自动分配和释放，存放函数的参数以及局部变量。其分配运算内置于处理器的指令集中，效率很高。但是可使用的总量有限，一般不会超过 1M 字节。

堆区中内存的分配和释放由开发者负责。一般用运算符 new 分配内存，并用运算符 delete 释放内存。一个 new 要对应一个 delete，否则会导致内存泄露。如果开发者没有释放，在程序结束的时候操作系统会自动回收。在堆上可分配的内存比栈上大了很多，且使用非常灵活。

自由存储区和堆类似，但是其内存管理是通过库函数 malloc 和 free 等进行的。在 C 程序中经常使用，虽然在 C++程序中仍然可以使用，但不如用堆方便，这里不多做介绍。

全局/静态存储区中存放的是全局变量和静态变量。该存储区分配的内存在整个程序运行期间一直有效，直到程序结束由系统回收。

常量存储区中存储的是常量，通常不允许修改。在程序中定义的常量以及指针字符串都存储在这里。

对于初学者来说，经常区分不清楚上面几种内存分配方式。下面有个简单的例子，使用了其中的几种内存分配方式：

```
#include <iostream>
using namespace std;
int a;                      // 全局变量，存储在全局/静态存储区
int main()
{
    int b;                  // 局部变量，存储在栈上
    int * p = new int();    // 由运算符 new 分配的，存储在堆上
    static int d;           // 静态变量，存储在全局/静态存储区
    const int e = 0;        // 常量，存储在常量存储区
    delete p;               // 释放堆中的内存
    return 0;
}
```

C++程序中的内存都要从上面的 5 个区中分配。不过栈、全局/静态存储区以及常量存储区中的分配是由编译器来进行的，并且在程序运行之前已经分配，因此称之为静态分配；堆以及自由存储区上内存的分配，是在程序运行的过程中进行的，称之为动态内存分配。下节中主要介绍通过

new 进行的动态内存分配。

只有在堆上和自由存储区中分配的内存需要开发者管理。其他存储区中的变量只要定义即可，其内存的分配和释放由编译器负责。

7.3.2 在堆上分配内存

在堆上分配内存，要使用 new 关键字，后面跟一个数据类型。如果需要对分配出的内存进行初始化，则在类型后面加上一个括号，并带有初始值。为了保存分配出内存的地址，应当使用一个指针指向 new 的结果。其语法如下：

```
类型标识符 *指针名  =  new  类型标识符(初始值);
```

例如：

```
int * p = new int(3);
```

上面的语句在堆上分配一块整型变量的内存，并使指针 p 指向这块内存。括号中的 3 为这块内存提供一个初始值，该内存分配成功之后，会用 3 去初始化，因此*p 的值为 3。在堆上分配内存的时候也可以不提供初始值，例如：

```
int * p = new int;          // 不提供初始值
```

对于这种不提供显式初始化的情况，如果 new 的类型是类，则会调用该类的默认构造函数；如果是内部数据类型，则不会被初始化。

用 new 不但可以为一个变量分配内存，还可以为一个数组分配内存，其方法是在 new 的类型标识符后面跟一个中括号，其中是数组的长度。其语法如下：

```
类型标识符 *指针名 =  new 类型[长度];
```

例如：

```
int * pArr = new int[10];       // 定义一个长度为 10 的数组
```

同数组的定义不同，用运算符 new 为数组分配内存空间时，其长度可以是变量，而不必是常量。因为动态内存分配在程序运行的时候才分配空间，所以当用 new 在堆上定义数组的时候，其长度可以是一个变量。例如：

```
int n = 10;
int * parr = new int[n];        // 可以用变量作为动态生成的数组的长度
```

因此当数组的长度不确定的时候，一般都要使用 new 的方法定义数组。new 数组也有其缺点，那就是不能提供显式初始化值。

7.3.3 释放堆上的内存

通过 new 分配的内存必须由开发者自己去释放，一块内存如果没有被释放，则会一直存在到该应用程序结束。在 C++中使用 delete 来释放内存。对于单个内存，释放语法为：

```
delete 指针名;
```

对于数组，释放语法为：

```
delete []指针名;
```

其中，中括号“[]”表示释放的是一个数组。用 delete 运算符释放为数组分配的内存时，不需要指定数组的长度。例如：

```
int * p = new int;
delete p;                    // 释放单个变量的内存
int * pArr = new int[10];
delete []pArr;               // 释放数组
```

在下面的程序中，需要用户输入数组的长度以及数组中每个元素的值，采用动态创建的方法。程序如示例代码 7.5 所示。

示例代码 7.5

```
#include <iostream>
using namespace std;
int main()
{
    int * pCount = new int;                          // 动态创建一个变量
    cout<<"请输入数组的长度："<<endl;
    cin>>*pCount;
    cout<<"请输入数组的元素："<<endl;
    int * pArray = new int[*pCount];                 // 动态创建一个数组
    for( int i = 0; i < *pCount; i ++ )
    {
        cin>>pArray[i];                              // 输入数组元素
    }
    cout<<"数组元素为："<<endl;
    for( int i = 0; i < *pCount; i ++ )
    {
        cout<<pArray[i]<<endl;                       // 输出数组
    }
    delete pCount;                                   // 释放变量
    delete pArray;                                   // 释放数组
    return 0;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 7.11 所示。

图 7.11 动态内存分配结果

因为数组的长度依赖于输入，是个变量，因此需要用 new 的方法。因为 int 是 C++的内部数据类型，因此[]可以省略。

new 分配的内存，都需要显式地 delete。如果缺少了 delete，则会造成内存泄漏。含有这种错误的代码每被调用一次就丢失一块内存。也许刚开始时系统的内存充足，其影响还不明显，等到程序运行足够长的时间后，系统内存就会被耗尽，程序也就会崩溃。

但是对于不是用 new 分配的内存地址，则在其上使用 delete 是不合法的。下面有几个错误地使用 delete 的例子：

```
int i;                          // 局部变量
int * pi = &i;                  // 指针指向变量
string str = "abcdef";          // 字符串指针
double * pd = new double(33);   // 通过 new 分配的内存
delete str;                     // 错误，str 不是一个动态创建的对象
delete pi;                      // 错误，pi 指向一个实例变量
delete pd;                      // 正确
```

对于上面的两个错误，编译器能检查出对 str 的删除是错误的，因为编译器知道 str 并不是一个指针类型。但是对于第二个错误，编译器并不能断定一个指针指向什么类型的对象，不能在编译时检查出来。因此在程序中 new 和 delete 出现的次数必须相同，而且对于一块使用 new 分配的内存，只能 delete 一次，否则也会造成系统错误。

7.4 const 与指针

const 是一个常量修饰符。使用 const 修饰一个变量时，将使其值不可改变，从而变成一个常量。const 也可以在定义指针时使用，不过此时 const 不仅可以修饰指针变量，也可以修饰指针所指向的数据。

7.4.1 指向 const 的指针

指向 const 的指针，指的是一个指针指向的数据是常量，不可以被修改，但指针变量本身可以被修改。其定义格式是在普通指针定义前加上 const 关键字：

```
const 类型名 * 指针名;
```

例如：

```
const int * p;
```

在上面的定义中标识符 p 前面的“*”表明 p 是一个指针，int 表明 p 是一个指向整型数据的指针，而 const 则表明 p 所指的数据是一个常量，不能被修改。严格来讲是不能通过指针 p 间接修改这个数据的。例如：

```
int a = 0;              // 定义变量
const int * pa;         // 定义一个 const 指针
pa = &a;                // 可以改变指针指向的地址
*pa = 1;                // 错误，不能通过该 const 指针修改其值
a = 2;                  // a 是变量，可以修改
```

在上面的代码中，试图对*pa 的修改是错误的，因为 pa 是一个指向 const 的指针。但是 a 本身还是一个变量，因此可以直接对 a 进行修改。因此，指向 const 的指针，这个 const 限制的是通过指针来修改变量的权限，而不是修改变量的权限，该变量还是可以被修改的。对于符号常量，即用 const 修饰的变量，只能用指向 const 的指针来指向，而不能用普通的指针。例如：

```
const int a = 10;       // const 变量
int * p1 = &a;          // 错误
const int * p2 = &a;    // 正确
```

指向 const 的指针，只限制了指向变量不可以被修改，但是指针本身的指向是可以修改的：

```
int a ;
int b;
const int * p = &a;
p = &b;                 // 修改指针的指向
```

在函数定义中，经常用指针来代替数组，对于只读的数组，经常用指向 const 的指针来限制。

7.4.2 const 指针

const 指针，指的是指针变量本身是一个常量，只能指向定义时所给的那个数据，而不能指向别处，而对被指向的数据是没有影响的。其定义语法是在指针名前加上 const 关键字，而不是在最前面：

```
类型名 * const 指针名;
```

例如：

```
int a;                  // 定义一个变量
int * const p = &a;     // 定义一个 const 指针指向这个变量
```

当 const 指针定义之后，可以修改其指向变量的值。

```
*p = 1;              // 正确，修改其指向的值
int b;
p = &b;              // 错误，不可以修改其指向
```

因为不能修改 const 指针的指向，所以在定义的时候必须给一个初始值。下面的定义是错误的：

```
int * const p;       // 错误，没有初始化值
```

7.4.3 指向 const 的 const 指针

上面讲述了 const 在两个位置的不同效果，const 还可以同时出现在这两个位置，其定义格式为：

```
const 类型名 * const 指针名;
```

这是前面两种情况的组合，其指向的变量和指向都不可以被修改，例如：

```
int a = 0;                    // 定义一个变量
const int * const p = &a;     // 指向 const 的 const 指针
```

对于这个指针 p，下面两种情况的修改都是错误的：

```
*p = 1;              // 错误
int b;
p = &b;              // 错误
```

p 表示一个指向 const 的指针，因此既不能修改指针的指向，也不能修改指针指向地址的变量。

三种 const 指针的定义很相似，很难以区分，这里有一个简单的技巧：从标识符的开始处读它，并从里向外读，const 指定那个“最靠近”的。因此根据这个理解：第一个定义表示 int 不可以被改变，第二个表示 p 不可以被改变，第三个表示 int 和 p 都不可以被改变。

7.5 引用

虽然指针的使用非常灵活和高效，但使用起来却不是非常方便。如果使用不当，很容易导致某些不易察觉的错误。为此，C++引入了引用。

7.5.1 定义引用

引用也是一种数据类型。不过，引用不能独立存在，而只能依附于一个变量。所以定义一个引用，必须要指明是哪个变量的引用。定义一个引用包括目标变量的数据类型、引用修饰符“&”、引用的标识符以及目标变量的标识符，其语法如下：

```
类型标识符 &引用名 = 目标变量名;
```

其中类型标识符是目标变量的类型。“&”是引用修饰符，表示定义的是一个引用。而被引用的变量则通过赋值运算符指定。例如：

```
int a;                  // 定义一个变量
int & b = a;            // 定义一个上述变量的引用
```

“&”在这里不是取地址运算符，而是一个引用修饰符。

一旦定义，则始终跟其目标变量绑定，而不能改变为其他变量的引用。假设 b 是变量 a 的引用，则在 b 的生命周期内，b 始终都是 a 的引用，而不能再改变为其他变量的引用。另外，引用在其生命周期内完全可以替代其目标变量。也就是说，所有施加于引用上的操作，其效果都等同于直接对引用的目标变量进行操作。而且一旦目标变量的值发生了改变，引用的值也会发生同样的改变。

鉴于引用的不可变更性，以及引用与目标变量的等价性，一个变量的引用也可以看做该变量的别名。定义一个引用只不过是给变量另外起了一个名字。这样两个名字拥有一个实体，对一个名字的操作自然也会影响到另外一个名字。

引用的上述行为和特性类似于 const 指针。首先，const 指针也是定义后不能再指向别的变量的；其次，通过 const 指针可以间接地修改其目标变量，而且对目标变量的修改也会影响 const 指针。

7.5.2 常引用

定义引用时也可以用 const 进行修饰，其结果为一个常引用。其语法如下：

```
const 类型标识符 &引用名 = 目标变量名;
```

例如：

```
int a;                  // 定义一个变量
const int & b = a;      // 定义 a 的引用
a = 1;                  // 正确，可以修改变量
b = 2;                  // 错误，不可以通过常引用修改变量
```

上面语句为 a 定义了一个 const 的引用 b，这样，就可以限制通过引用 b 来修改目标变量 a，相当于给了这个别名只读不写的权限。

一般的引用在定义时必须有一个已经存在的变量，而常引用则没有这样的限制，可以定义一个字面常量的常引用。例如：

```
const int &a = 12;      // 定义一个字面常量的常引用
```

对于符号常量，即被 const 修饰的变量，其对应的引用必为常引用。否则对于一个变量实体，其一个名字显示不可修改，而另一个名字显示可以修改，这样显然是矛盾的。

7.6 引用与指针的区别

引用是 C++特有的新类型（与 C 相比）。在很多情况下，引用提供了与指针操作同等的功能。但是引用和指针还是有一些区别的，在使用时应当根据实际情况进行选择。例如：

- 引用必须与一个变量绑定，不存在没有任何目标的引用。因此如果在使用的过程中有可能出现什么都不指向的情况，则应该使用指针，可以把一个空值给指针。而若一个变量肯定指向某个对象，不允许为空，则可以使用引用。
- 引用仅仅是一个变量实体的别名。因此在使用引用之前，不需要检测其合法性。但指针在使用之前必须检测其指向的对象是否为空，不能对空指针进行取值操作。
- 指针可以重新赋值以重新指向新的对象，引用在初始化之后就不可以改变指向对象了。
- 指针可以指向数组的地址来替代数组使用，而引用不可以替代数组，引用只能指向数组中的某一个元素。
- 指针可以用于在堆上生成的对象，delete 的对象只能是一个指针，不能是引用。

7.7 综合实例

指针是 C++中的重要知识，在本节中，将列举典型的例子讲解如何综合应用指针解决实际问题。读者可以根据这些例子，举一反三，掌握用指针如何解决现实问题。

7.7.1 数组元素排序

在前面的章节中，曾经用“冒泡排序”的方法对一个数组进行排序。本例仍使用这个算法，不过在本例中将动态创建数组，并且使用指针来访问数组元素。程序如示例代码 7.6 所示。

示例代码 7.6

```
#include <cstdlib>
#include <iostream>
using namespace std;                                    // 使用名称空间 std
int main(int argc, char *argv[])                        // 主函数
{
    cout<<"——冒泡排序——"<<endl;
    cout<<endl;

    int length = 0;                                     // 记录数组的长度
    cout<<"请输入数组的长度：<<endl;
    cin>>length;                                        // 输入数组长度
    cout<<"请依次输入这"<<length
        <<"个数据，用空格或者回车隔开："<<endl;
```

```
    int * pArr = new int[length];              // 为数组分配内存空间
    for( int i = 0; i < length; i ++ )
    {
        cin>> pArr[i];              // 输入数组数据
    }

    int * p = pArr + length - 1;    // p 指向最后一个需要交换的元素
    while( p != pArr )              // 只要 p（后指针）不是指向第一个元素
    {
        int * q = pArr;             // q 指向前面的元素
        while( q < p )              // 只要 q（前指针）仍然是在 p（后指针）的前面
        {
            if( *q < *(q+1) )       // 如果两个元素需要交换
            {
                int temp = *q;      // 交换这两个数
                *q = *(q+1);
                *(q+1) = temp;
            }
            q++;                    // 前指针往后移
        }
        p--;                        // 后指针往前移
    }

    cout<<"从大到小依次为："<<endl;
    for( int i =0; i < length; i ++ )  // 遍历数组
    {
        cout<<pArr[i]<<" ";             // 输出元素
    }
    cout<<endl;
    delete pArr;                        // 释放数组

    cout<<endl;
    system("PAUSE");                    // 等待用户反应
    return EXIT_SUCCESS;                // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 7.12 所示。

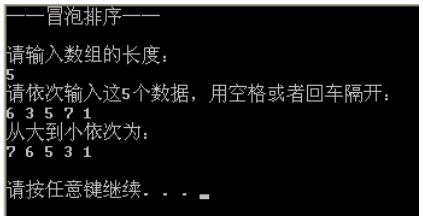


图 7.12　数组排序结果

7.7.2 输出杨辉三角

在前一章中曾经讲过有关杨辉三角的例子。在那个例子中用两个数组来计算，每计算一次，都要进行一次数组的复制，比较烦琐。本例使用指针来代替数组进行计算，可以展现使用指针的方便之处。程序如示例代码 7.7 所示。

示例代码 7.7

```
#include <cstdlib>
#include <iostream>
using namespace std;                                    // 使用名称空间 std
int main(int argc, char *argv[])                        // 主函数
{
    cout<<"——杨辉三角形——"<<endl;                        // 输出提示信息
    cout<<endl;

    cout<<"请输入杨辉三角的行数（1~10）:"<<endl;
    int n = 0;
    cin>>n;                                             // 输入行数

    int a[20] = {0},b[20] = {0};                        // 定义两个辅助数组
    int * last = a;                                     // 定义指向数组的指针
    int * current = b;
    for( int i = 0; i < n; i ++ )                       // 对于三角形中的每一行
    {
        current[0] = 1;                                 // 第一个元素为 1
        current[i] = 1;                                 // 最后一个元素为 1

        for( int j = 1; j < i; j ++ )                   // 对于每行中的其他元素
        {
            current[j] = last[j] + last[j-1];           // 等于上一行中两个相邻元素的和
        }
        for( int j = 0; j < n - i - 1; j ++ )           // 对于每一行前面的空格
        {
            cout<<' ';                                  // 输出前导的空格
        }

        for( int j = 0; j <= i; j ++ )                  // 对于当前行中的每一个元素
        {
            cout<<current[j]<<' ';                      // 输出
        }
        cout<<endl;                                     // 换行

        int * temp = current;                           // 完成本行的计算，改变指针的指向
        current = last;
        last = temp;
    }

    cout<<endl;
    system("PAUSE");                                    // 等待用户反应
```

```
    return EXIT_SUCCESS;                          // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 7.13 所示。

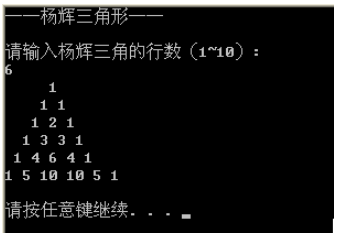


图 7.13　杨辉三角的输出结果

7.8 小结

本章的主要内容是指针与引用的使用。指针其实是一个存储地址的变量。指针可以指向各种数据，而且也可以通过指针间接地访问其指向的数据。另外，在 C++程序中利用指针管理动态分配的内存也是非常重要的内容。

指针毕竟是一个特殊的变量，其使用与普通变量很不相同。所以 C++引入了引用的概念。引用一方面类似 const 指针的行为，另一方面其使用方法同普通变量非常相似。

到本章为止，C++程序中常用的几种数据类型以及数据计算的基本方法都已经讲过了。如何将这些元素组织成有效的程序，并能够重复利用已有的代码将是下一章的重点内容。

◆　　◆　　◆

第8章 函 数

本章包括

- 什么是函数，为什么要使用函数
- 函数的声明和定义
- 参数传递
- 函数中的变量
- 什么是递归函数
- 什么是内联函数
- 函数的重载

前面已经学习了变量、常量、表达式和语句，这些都是组成程序的基本要素。本章将学习程序中另外一个重要的概念——函数。C++程序通过函数将上述基本要素结合在一起，完成一定的功能。函数也是程序中的最小模块，可以重复使用。本章将主要讲述函数的声明和定义，以及如何使用函数，递归函数、内联函数以及函数重载也是本章的重要内容。

8.1 什么是函数

数学中的函数可以根据输入的数值求得一个确定的值。与此类似，C++中的函数也可以根据输入的数据，返回一个结果值。不同的是，C++中的函数涵盖的意义更加广泛，定义也更加灵活，不仅可以求数值，还可以执行一组相关的操作。在后文中，如不加特别说明，所谓的函数都是指 C++中的函数。

8.1.1 函数的组成部分

函数由 4 部分构成：返回类型、函数名、参数列表和函数体。函数名就是函数的名字，即函数的标识符。既然是标识符，就必须遵循标识符的命名规则。同变量的标识符相同，函数的标识符也只能由字母、数字以及下画线组成，并且不能以数字开头。

函数名要求能够描述这个函数的功能或者定义该函数的目的。不准确或者没有意义的函数名往往会误导开发者，并导致程序难以阅读；而准确、有意义的函数名则可以提高程序的可读性，也是对开发者的一种指导。

函数的返回类型指的是函数会返回数据的类型。例如一个函数返回一个 int 型数据，则该函数的返回类型就是 int。如果某个函数不返回任何值，则其返回类型是 void，定义这样的函数的目的不是求一个值，而只是为了执行一组操作。

参数是函数的“输入”数据（与此相对应，返回值可以看做函数的“输出”数据）。函数从其使用形式来看就是接受“输入”的参数，并返回一个结果值。函数的参数位于一个括号中，并且用逗号分隔。括号中的部分就称做函数的参数列表。

例如对于求最小值的函数 min()，其功能是求两个整数中的最小值。根据其功能描述，min() 函数接受两个整型参数，其参数列表是：

```
(int x, int y)
```

函数体是一个语句块，即由花括号“{”和“}”包含的一组语句。这些语句用来实现函数的功能或目的。如果函数要返回一个数据，则函数体中至少需要一个 return 语句。其语法如下：

```
return 结果表达式;
return ( 结果表达式 );
```

上述两种形式是等价的，即结果表达式带不带括号都可以。

如果函数不需要返回数据，即函数的返回类型是 void，则在函数体中不需要 return 语句。即便是加了 return 语句，也不能带任何表达式。

函数的 4 个部分组合在一起，就构成了一个函数的完整定义，即：

```
返回类型  函数名( 参数 1,  参数 2, ……)
{
    语句 1;
    语句 2;
    ……
    语句 n;
    return 结果表达式;
}
```

下面是函数定义的一些例子：

```
int min( int p1, int p2 )              // 求最小值的函数 ( 有返回值 )
{
    return( p1 < p2 ? p1 : p2 );       // 返回较小值
}
void outputSum( int a, int b )         // 输出两数之和的函数 ( 无返回值 )
{
    int c = a + b;                     // 求两个参数的和
    cout<<"The sum is "<< c <<endl;    // 输出求和的结果
}
```

8.1.2 调用函数

所谓调用函数，就是使用函数的功能返回一个值或者执行一组语句。调用函数时，在函数名后跟一个括号，其中是各个参数，用逗号分隔，其语法如下：

```
返回类型  变量 = 函数名 ( 参数列表 );    // 调用有返回值的函数
函数名 ( 参数列表 );           // 调用 void 类型的函数，或者虽然有返回值，但不需要保存
```

例如，定义求绝对值的函数 abs：

```
int abs( int iobj )
{
    return( iobj < 0 ? -iobj : iobj ); // 返回 iobj 的绝对值
}
```

其调用方法如下：

```
int a = 0;              // 声明并初始化一个整型变量
cin>>a;                 // 输入任意一个整型数
int b = abs ( a );      // 求输入整数的绝对值
```

上述程序的最后一行，调用 abs 函数，并用其返回值初始化变量 b。

8.1.3 为什么要使用函数

函数是语句的集合，但不是任何语句的集合都是函数（语句块就不是函数）。将多个语句组成函数是有目的的。或者是为了执行一组固定的操作，或者是为了求得某个数值，也可能是上述两个目的的组合。使用函数的最大好处就是可重复使用。虽然为了实现同一目的，可以用同样的语句复制粘贴，但这样一来，将导致程序代码急剧膨胀。而且当需要改变程序时，所有粘贴的代码都需要一一改变，不仅效率极低，而且也很容易因疏忽而遗漏。

8.2 函数的声明和定义

C++标准规定：函数在调用之前，必须先声明。函数的声明包括返回类型、函数名和参数列表。函数声明就是为了告诉编译器存在这样一个函数，并且可以在后面的程序中使用。与函数定义不同，函数的声明不包含函数体。函数的声明也称做函数原型，而函数的定义也称做函数实现。

8.2.1 函数的参数列表

在函数的声明和定义中，不能没有参数列表。如果函数不需要参数，则可以用空参数列表或只带一个 void 关键字的参数列表，例如：

```
int function();
int function( void );
```

这两种声明方式是等价的，都表示不接受任何参数。声明函数时，参数列表中的每个参数可以有名字，也可以没名字，即只带一个类型关键字，例如：

```
int max ( int, int );
int max ( int a, int b );
```

这两种声明方式也是等价的，都表示接受两个整型参数。

尽管声明函数时不需要给出参数的名字，但是如果参数有名字，则可以提示该参数的含义，开发者可以根据参数的名字传入合适的参数。

尽管声明时不需要给出参数的名字，但在定义时必须有，否则会引起编译错误。而且参数名在函数定义时也有着特殊的意义，这将在后面的小节中说明。

8.2.2 调用函数前先声明

函数声明的作用就是告诉调用者如何使用该函数，即函数接受什么类型的参数、参数的个数以及函数的返回类型。函数声明只是函数定义的一部分，它缺少函数体。如果在调用前函数已经定义，则不必另外声明，因为编译器已经知道了该函数的全部信息。例如：

```
int max ( int a, int b ){          // 求最大值函数的定义
    return a>b ? a : b;            // 返回两个数中的最大值
}
int main(int argc, char *argv[]){  // 主函数
    int x = 0, y = 0;              // 声明并初始化两个整型变量
    cin>>x;                        // 输入整数
    cin>>y;                        // 输入整数
    int z = max ( x, y );          // 求最大值
    return EXIT_SUCCESS;           // 主函数返回
}
```

当编译器顺序编译下来，直到第 9 行调用函数 max 时，max 函数接受的参数及其返回值类型都已经明确，所以不需要另外声明。如果 max 函数在 main 函数之后定义，则在调用前必须声明。

```
int max ( int a, int b );          // 求最大值函数的声明
int main(int argc, char *argv[]){  // 主函数
    int x = 0, y = 0;              // 声明并初始化两个整型变量
    cin>>x;                        // 输入整数
    cin>>y;                        // 输入整数
    int z = max ( x, y );          // 求最大值
    return EXIT_SUCCESS;           // 主函数返回
}
int max ( int a, int b ){          // 求最大值的函数定义
    return a>b ? a : b;            // 返回两个数中的最大值
}
```

函数声明也是一条语句，后面必须带有分号“;”，表示语句的结束。

上述代码中，第 1 行就是函数的声明，即告诉编译器有一个名为 max 的函数，接受两个整型参数，并返回一个整数。在后面的代码中，调用 max 函数时，编译器就会在整个源文件中查找这个 max 函数的定义。

8.2.3 在头文件中声明函数

对于只有一个源文件的简单程序，不需要函数声明，只要保证函数在调用前定义即可。而对于复杂的大型程序，往往有很多源文件，而且一个源文件中定义的函数，往往会被其他源文件使用，

因此就需要在其他源文件中声明该函数。在有多个源文件的程序中，往往把函数的声明放在头文件中。当别的源文件要声明函数时，只要包含头文件即可（用#include 指令）。

使用各种库函数时也是如此。使用库中的某个函数时，首先要声明，即包含相应的头文件。调用时，编译器会根据函数的声明，到库中查找函数的定义。

函数不能只有声明，还必须有定义。函数的定义一般在源文件中实现。例如，求最大值的 max 函数，这是一个基本函数，经常会用到，有必要将其声明放在一个头文件中，然后在一个源文件中实现 max 函数，在别的源文件中调用 max 函数之前，需要包含上述头文件。其开发过程如下。

step 1 写一个头文件 method.h，用来声明 max 函数，其内容如下：

```
// 声明一个求最大值的函数，接受两个整型参数，返回一个整型数
int max ( int a, int b );
```

step 2 写一个源文件 method.cpp，用来定义 max 函数，其内容如下：

```
int max( int a, int b ){            // 求最大值函数的定义
    return a > b ? a : b;           // 返回两个值中的最大值
}
```

step 3 在别的源文件中调用 max 函数时，先要包含头文件 method.h，例如在主函数中调用时：

```
#include "method.h"                 // 包含头文件 method.h，即声明函数 max
int main(int argc, char *argv[])    // 主函数
{
    cout<< max ( 1, 2 ) <<endl;     // 调用 max 函数，并输出结果
    return EXIT_SUCCESS;            // 主函数返回
}
```

8.2.4 定义函数

一个函数的定义由返回类型、函数名、参数列表和函数体组成。前面的 3 个部分称为函数的声明或函数原型。相对函数体来讲，也称为函数头。函数体是一个语句块，由花括号“{”和“}”包围起来。函数中的 return 语句除了返回数据之外，还有一个重要的功能，就是结束函数的运行，在 return 语句之后的语句不会被执行。例如在下列函数体中：

```
{
    ……
    return 123;                     // 函数返回
    cout<<"函数结束！"<<endl;        // 这行语句不会执行
}
```

第 4 行的语句永远不会执行，因为在第 3 行函数已经返回，并结束了运行。定义函数时不允许嵌套，即在函数体中不能定义另外一个函数。下面的情形是不允许出现的：

```
void f1()
{
    int f2()    // 错误！函数定义不允许嵌套
    {
```

```
        return 0;
    }
}
```

尽管函数定义不允许嵌套，但可以在函数体中声明另外一个函数。

8.2.5 函数实例——判断闰年

在公历纪年中，为了弥补公历与地球公转周期的误差，有的年份要比其他年份多一天。多出来的一天称做闰日，有闰日的年份称做闰年，没有闰日的年份称做平年。闰日通常放在二月份，所以闰年的二月份有 29 天（通常有 28 天）。

闰年的计算方法是：如果年份数不能被 4 整除，则是平年；如果能被 4 整除，但不是 100 的倍数，也是闰年；如果既能被 100 整除，又能被 400 整除则为闰年，否则为平年。例如，1996 年可以被 4 整除，又不是 100 的倍数，所以是闰年；2100 年能被 100 整除，但不能被 400 整除，所以不是闰年；而 2000 年即可以被 100 整除，又能被 400 整除，所以是闰年。这个计算流程如图 8.1 所示。

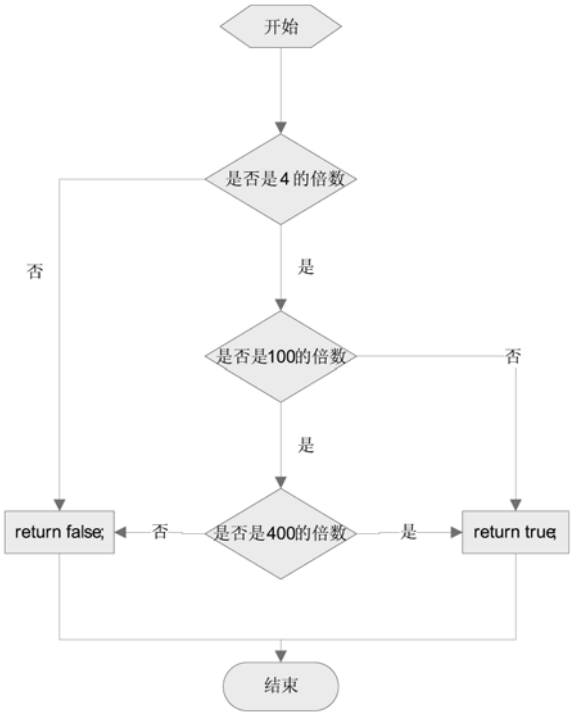


图 8.1　判断闰年的流程图

开发一个程序，用来判断用户输入的年份（如 1997，2008 等）是不是闰年，程序如示例代码 8.1 所示。

示例代码 8.1

```
#include <cstdlib>
```

```
#include <iostream>
using namespace std;                                    // 使用名称空间 std
bool isLeapYear( int year )                             // 判断是否是闰年的函数
{
    if( ( 0 == year % 4 ) && ( 0 != year %100 ) )
                                                        // 是否是被 4 整除，并且不是 100 的倍数
    {
        return true;
    }
    if( 0 == year % 400 )                               // 是否是 400 的倍数
    {
        return true;
    }
    return false;
}
int main(int argc, char *argv[])                        // 主函数
{
    cout<<"———判断闰年———"<<endl;                        // 输出提示信息
    cout<<"请输入一个年份【输入负数退出程序】："<<endl;
    int x = 0;                                          // 保存年份的变量
    do{                                                 // do...while 循环
        cin>> x;                                        // 输入年份数
        if( x < 0 )                                     // 如果年份是负数，退出循环
        {
            break;
        }
        if( isLeapYear( x ) )                           // 调用 isLeapYear 函数
        {
            cout<< x <<"年是闰年"<<endl<<endl;          // 输出结果信息
        }
        else
        {
            cout<< x <<"年不是闰年"<<endl<<endl;        // 输出结果信息
        }
    }while(true);                                       // do...while 循环

    system("PAUSE");                                    // 等待用户反应
    return EXIT_SUCCESS;                                // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 8.2 所示。

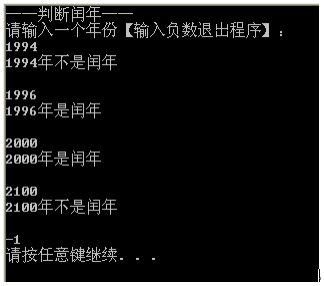


图 8.2　判断是否是闰年结果

在函数 isLeapYear()的定义中，函数过程可以更加简略，例如可以写成：

```
if( !(year%4) && year%100 || !(year%400) )
{
    return true;
}
return false;
```

不过，虽然这样代码的行数减少了，程序看上去更简洁，但相比原来的版本，这个版本不太利于别人理解，增加了程序维护的成本。

书写代码时，应当尽量用清楚、明白的算法流程，而不是采用复杂的表达式。

8.3 参数传递

调用函数时，开发者应当按照参数列表为函数传入所需的参数。例如 isLeapYear(int year)函数的参数列表是一个整数，调用时应当给 isLeapYear()函数传递一个整数，例如 isLeapYear(x)，isLeapYear(2000)。

8.3.1 函数的形参和实参

形参是函数定义时的参数。之所以称为形参，是因为这些参数实际并不存在，只是在形式上代表运行时实际出现的参数。实参是函数调用时传入的参数，也是程序运行时实际存在的参数。例如在下面的程序中：

```
int min ( int a, int b )                // 求最小值的函数定义，a 和 b 是形参
{
    return a<b ? a : b;                 // 返回两个参数中的最小值
}
05
int main(int argc, char *argv[])        // 主函数
{
    cout<< min ( 1, 2 ) <<endl;         // 调用 min 函数，并输出结果。1 和 2 是实参
    int  x = 0, y = 0;                  // 声明并初始化两个整型变量
    cin>>x;                             // 输入一个整型数
```

```
    cin>>y;                                  // 输入另一个整型数
    int  z = min( x, y ); // 调用 min 函数，x 和 y 作为实参传入，并用返回值初始化变量 z
    return EXIT_SUCCESS;                     // 主函数返回
}
```

在第一行中，参数 a 和 b 就是形参。在程序的后面，调用函数 min 时传入的常量 1 和 2，以及传入的变量 x 和 y 都是实参。形参和实参是相对的概念，有时会相互转化。当函数 A()调用函数 B()时，由于是在 A()的函数体中，所以 A()的参数对于 A()来讲是形参。但是在调用 B()函数时，可以使用 A()的参数，此时这个参数对于 B()来讲就是实参。例如：

```
void outputMin ( int x, int y )          // output 函数，输出两个参数中的最小值
{
    int z = min( x, y );                 // 调用 min 函数，求参数中的最小值
    cout<<"最小值是："<< z <<endl;       // 输出最小值
}
```

对于 outputMin 函数，x 和 y 是形参；对于 min 函数（第 3 行的调用），x 和 y 是实参。

8.3.2 值传递

实参传递给函数后，系统将构建一份实参的副本，其值与实参的值相同。此后函数将针对这份副本进行操作，对原始的实参没有任何影响。C++函数的这种约定称为“值传递”。

函数的形参就是实参的副本。

例如下列函数：

```
void swap( int a, int b )
{
    int temp = a;
    a = b;
    b = a;
}
……
int x = 0, y = 1;
swap ( x, y );
cout<<"x == "<< x <<endl;
cout<<"y == "<< y <<endl;
```

或许 swap 函数作者的意图是交换两个参数的值，即期待程序输出“x == 1”和“y == 0”，但实际输出结果却如下：

```
x == 0
y == 1
```

这说明 swap 函数的调用并没有改变两个实参变量的值。

8.3.3 参数类型检查

调用函数时，编译器会检查实参与形参的类型是否匹配。如果不匹配，则检查是否存在实参类

型到形参类型的隐式类型转换。如果存在，则按照隐式类型转换规则进行转换；否则，编译器就会报告一个编译错误。例如：

```
bool equal( int a, int b )      // 判断两个整数是否相等的函数
{
    return a == b;              // 用“==”运算符进行比较
}
……
bool res = false;               // 表示是否相等的变量
res = equal( 1, 1 );        // 传入整型，类型匹配
res = equal( 'a', 'a' );    // char 型转换为 int 型，ASCII 码相等，返回 true
res = equal( 'a', 'b' );    // char 型转换为 int 型，ASCII 码不等，返回 false
res = equal( 3.2, 3.8 );    // double 型转换为 int 型，舍尾后都是 3，返回 true
res = equal( 3.2, 4.0 );    // double 型转换为 int 型，舍尾后分别是 3 和 4，返回 false
```

8.3.4 使用默认实参

默认实参是声明函数时给参数指定的默认值。指定默认实参的方法类似于初始化一个变量。例如：

```
void function( int a, int b = 123 );
```

在上述 function 函数的声明中，参数 a 是一个普通的参数，而参数 b 则是一个具有默认实参的参数，这个默认实参就是 123。

对于声明中带有默认实参的函数，调用时可以不给该参数传实参。虽然调用时没有指定实参，但在函数运行时，该参数仍然有值，这个值就是声明时的默认实参。例如，下面定义一个具有两个默认实参的函数，虽然调用时一个参数都没有指定，但两个参数都有实参。

```
void outputVal( int iVal = 123, double dVal = 3.14159 )
                    // 指定参数 iVal 和 dVal 的默认实参
{
    cout<< iVal <<endl;     // 输出参数的值
    cout<< dVal <<endl;
}
……
outputVal();        // 调用函数，没有指定实参，函数以默认值 123 和 3.14159 作为实参
```

上述代码的输出结果是：

```
123
3.14159
```

虽然省略了传递实参的麻烦，但是默认值未必符合特定情形下的要求。因此在某些情况下，不能使用默认值，而需要调用者明确指定。调用者指定的实参会覆盖参数的默认值，也就是说函数运行时会按照调用者指定的值运行。例如调用 outpuVal 函数：

```
outputVal( 456 );           // 覆盖 iVal 的默认实参，使用 dVal 的默认实参
```

一个参数只能在一个文件中被指定一次默认实参，但可以在多次声明中依次向前指定其他参数的默认实参。在头文件 method.h 中声明一个函数：

```
int abc( int a, int b, int c = 3);
```

在源文件 method.cpp 中定义这个函数：

```
#include "method.h"         // 包含头文件 method.h
int abc( int a, int b, int c = 3 )
            // 错误，#include 头文件之后，abc 的参数 c 的默认实参重复，应当去掉“=3”
{
    ……
}
```

在源文件 main.cpp 中使用函数 abc 时：

```
#include "method.h"                 // 包含头文件 method.h
int abc ( int a, int b=2, int c ); // 正确，重新声明函数 abc，指定 b 的默认实参为 2
```

在 method.h 中，b 是没有默认实参的最后一个参数，所以在 main.cpp 中可以指定其默认实参。

8.4 指针和引用参数

到目前为止，前面讲述的函数都是以传值的方式传递参数的。传值方式传递的只是实参的值，而不是实参本身。这样做一方面会有效率的问题，因为对于大的数据类型有一次赋值过程；另一方面在函数中也并不能改变实参的值，因为函数中只是构建了实参的一个副本。用指针和引用作为参数，可以弥补上述不足，下面将一一讲述。

8.4.1 指针参数

在函数的参数列表中，可以使用指针类型的参数。传递给指针参数的实参可以是一个指针变量，也可以是一个变量的地址。例如，下面函数的参数就是一个 int 类型的指针：

```
void function ( int * );
```

调用函数 function 时也必须传入一个 int 型的指针变量。例如：

```
int *p;             // 定义一个 int 型指针变量
……                  // 某些操作改变了 p 的值
function( p );      // 以 p 作为实参，调用函数 function
int iVal = 0;       // 定义整型变量
function( &iVal );  // 获取整型变量的地址，并以指针的形式传递给函数
```

使用指针作为参数可以弥补值传递的不足：

- 提高了传递参数的效率。
- 函数可以修改实参指针所指变量的值。

对于比较大的数据类型，例如定义了很多数据成员的结构体，假设是 100 个整型数成员。如果将这样的结构体变量作为参数来传递，则在函数内部也要构建同样的一个结构体变量，并将其数据成员依次赋值。这样在内存中就会有两份同样的结构体变量，占据了 800（200×4）个字节的内存。这无论在空间和时间上都是巨大的浪费。类也是如此。其实在 C++中，结构体与类的差别很小，

在后面的章节中将会有详细的说明。

如果用指针传递，则可以消除值传递带来的空间和时间上的浪费。这主要是因为指针的值是一个地址值（在 32 位系统中是一个 4 字节的整型数），参数传递时只需传递这个地址值即可。

用指针作为参数，遵从的也是值传递的原则。实际传递的值是指针的值，而不是指针所指变量的值。

例如，对于 SomeStruct 这样一个大的结构体，用指针作为参数可以写为：

```
void function( SomeStruct *p );
```

而用 SomeStruct 变量作为参数调用 function 函数时可以写为：

```
SomeStruct a;       // 定义一个 SomeStruct 的变量
……                  // 使用变量 a 的操作
SomeStruct *p = &a; // 定义一个 SomeStruct 类型的指针，并指向上述变量（获取 a 的地址值）
functioin( p );     // 用 SomeStruct 类型的指针作为实参，调用 function 函数
```

这样在调用函数时，函数就可以只构造一个指针变量的副本，而不用构造整个结构体的副本，大大节省了内存，而且省去了给每个数据成员赋值的时间，也大大提高了时间效率。用指针作为参数还有一个好处，就是函数可以修改指针实参所指变量的值。这是因为通过指针参数传递的是变量的地址，在函数内可以通过这个地址获取变量，从而也就可以修改变量的地址。例如：

```
void function( int *p )// 定义函数 function
{
    *p = 123;               // 获取指针形参所指变量，并给该变量赋值
}
……
int a = 0;                  // 定义整型变量 a，并初始化为 0
function( &a );             // 将变量 a 的地址传递给 function 函数
cout<< a;                   // 输出变量 a 的值
```

上述程序将输出 123，而不是 0。这就是因为在函数内部将变量 a 的值修改了。而如果参数是 int 类型的，则达不到这种效果。下面的例子定义一个函数，交换两个变量的值。由于使用普通类型的参数不能修改实参的值，所以可以用指针作为参数。程序如示例代码 8.2 所示。

示例代码 8.2

```
#include <cstdlib>
#include <iostream>
using namespace std;                // 使用名称空间 std
void swap( int *a, int *b )         // 交换函数的定义
{
    int temp = *a;                  // 用一个临时变量保存 a 指针所指变量的值
    *a = *b;                        // 将 b 指针所指变量的值赋给 a 指针所指的变量
    *b = temp;                      // 将临时变量的值赋给 b 指针所指的变量
}
```

```
int main(int argc, char *argv[])   // 主函数
{
    cout<<"——指针参数——"<<endl;   // 输出提示信息
    cout<<"请输入两个整型数："<<endl;
    int a = 0, b = 0;              // 保存用户输入的变量
    cin>> a;                       // 输入
    cin>> b;

    swap(&a, &b);                      // 调用交换函数

    cout<<"交换后: "<< a <<"\t"<<endl;  // 输出交换后的值

    system("PAUSE");                   // 等待用户反应
    return EXIT_SUCCESS;               // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 8.3 所示。

图 8.3 交换两个变量值的结果

在 swap 函数的定义中，用指针作为参数，避免了使用普通类型的参数，从而达到了交换变量值的目的。

8.4.2 数组参数

用数组作为函数参数，即函数的形参是数组。例如：

```
void function( int a[10] );
```

上述的函数声明表示 function 函数接受一个整型数组作为参数。但是，用数组作为函数参数时，数组的长度是没有意义的，也就是说上述函数的声明同以下的声明是等效的：

```
void function( int a[] );
void function( int a[100] );   // 数组长度没有意义，所以 a[10]同 a[100]是等效的
```

所以在定义函数时，不能依赖于数组参数中数组的长度。实际上，编译器会自动将数组参数作为指针进行解释，这个指针指向一块儿连续的内存。而这样的一个指针中不会保存长度信息，所以函数声明时，数组参数的长度是没有意义的。为了弥补这个缺点，可以在参数列表中再附加一个参数，用以传递数组的长度，例如：

```
void function( int a[], int n );
```

其中第二个参数 n 就是数组的长度。

定义函数时数组参数作为指针使用，而这个指针就指向数组实参的首地址，也就是数组中第一个元素的地址。

正因为在定义函数时，数组参数被当做指针，所以数组参数也可以用指针参数表示：

```
void function( int *p );
```

这个声明同 function(int a[])的声明是等价的。一般来讲，如果用数组作为函数的参数，则函数调用时应当将一个数组作为实参。因为数组参数在函数定义时被当做指针，所以也可以将指针作为实参。至于这个指针到底指向什么样的内存，则是由调用者确定的。用数组作为实参，则传递给函数的是数组名，调用过程如下：

```
int a[] = {1, 2, 3 };   // 定义数组
function( a );          // 以数组名作为实参，调用函数 function
```

或者也可以先取得数组的首地址，作为指针传递给函数：

```
int a[] = {1, 2, 3 };   // 定义数组
int *p = a;             // 获取数组首地址，也可以写为 int p = &a[0]
function( p );          // 以指向数组首地址的指针作为实参，调用函数 function
```

因为数组参数在定义函数时被当做指针使用，传递的值是数组的首地址。所以在函数内如果修改了数组参数中某个元素的值，也就修改了对应的数组实参中元素的值。C++程序中常用的排序函数就利用了数组参数的这个特性，它在函数中对数组参数进行排序，函数运行完后，数组中的元素就是排序后的结果。

下面的例子对一个整型数组按照递增的顺序进行排序。这里采用简单的选择排序算法，其算法思想主要是：依次确定第 i (i=0，1，…，n-1) 个位置上的元素，这个元素的值也就是第 i 个元素到第 n-1 个元素中的最小值。其流程如图 8.4 所示。

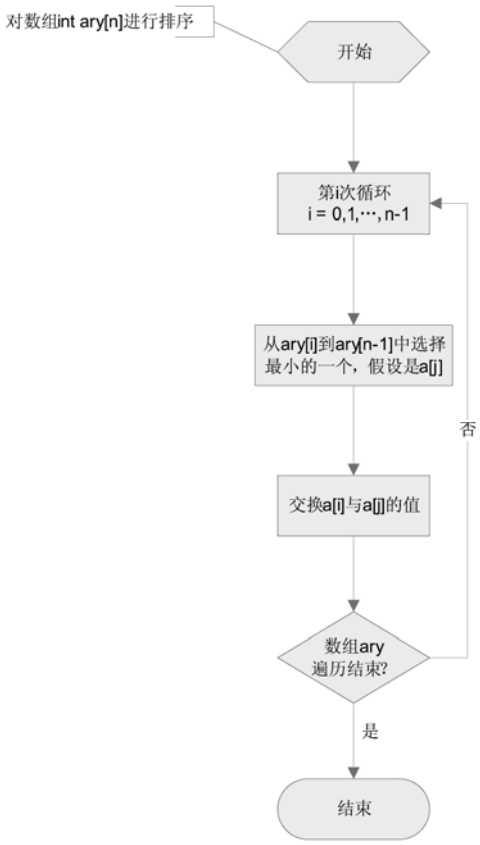


图 8.4　选择排序算法的流程图

程序如示例代码 8.3 所示。

示例代码 8.3

```
#include <cstdlib>
#include <iostream>
using namespace std;                        // 使用名称空间 std
void sort( int a[], int n )                 // 定义函数 sort
{
    for( int i=0; i<n; i++ )                // 遍历数组中的每一个元素
    {
        int j = i;                          // 记录当前元素的索引值
        for( int k=i; k<n; k++ )            // 遍历第 i 个到第 n-1 个元素
        {
            if( a[k] < a[j] )               // 找到其中最小元素的下标，保存到变量 j 中
            {
                j = k;
            }
        }
        if( j != i )                        // 如果最小值不是当前元素
        {                                   // 则交换两个元素的值
            int temp = a[i];
```

```
            a[i] = a[j];
            a[j] = temp;
        }
    }
}
int main(int argc, char *argv[])        // 主函数
{
    cout<<"——数组参数——"<<endl;          // 输出提示信息
    cout<<"请输入五个整数：" <<endl;
    int a[5];                           // 定义数组
    for( int i=0; i<5; i++ )            // 依次输入 5 个元素
    {
        cin>> a[i];
    }

    sort( a, 5 );                       // 对数组进行排序
    cout<<"排序后的结果：" <<endl;        // 输出提示信息
    for( int i=0; i<5; i++ )            // 输出排序后的结果
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;                         // 输出一个换行符

    system("PAUSE");                    // 等待用户反应
    return EXIT_SUCCESS;                // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 8.5 所示。

图 8.5　数组排序的结果

在函数 sort 的定义中直接修改了形参数组中的元素，由于数组参数被当做指针使用，所以实际修改的就是实参数组中的元素。

8.4.3 引用参数

使用引用参数的好处同使用指针参数的好处是相同的，也是提高了传递参数的效率，而且也可以在函数内部修改实参变量的值。例如：

```
void swap( int &a, int &b )
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
……
int a = 1;
int b = 2;
swap( a, b );
cout<< a<<" "<<b;
```

上述程序运行后将输出下面的结果：

```
2 1
```

8.5 函数中的变量

本节将详细讲解函数中变量的使用方法。变量是函数的主要内容，变量的类型和作用范围将对函数的功能起到很大的影响，下面将结合具体的例子详细讲解。

8.5.1 局部变量

定义函数时，在函数体中声明的变量称为局部变量。之所以称为局部变量，是因为这些变量的作用范围仅限于函数体内，在函数外部不能访问。例如：

```
void function()
{
    int a = 0;                          // 局部变量
}
int main(int argc, char *argv[])        // 主函数
{
    cout<< a << endl;                  // 编译错误，标识符“a”没有定义
    return EXIT_SUCCESS;               // 主函数返回
}
```

在上述程序中，函数 function 中定义了一个变量 a，其作用域仅仅限于 function 的函数体内。由于变量 a 并不存在，所以“a”这个标识符也就没有定义。函数体其实就是一个语句块。C++规定在语句块内声明的变量，其作用域仅限于这个语句块内，所以函数体中声明的变量，其作用域不会超出函数体的范围。

形参的作用域也仅限于函数体内，所以形参也可以当做局部变量使用。

8.5.2 全局变量

函数之外声明的变量称为全局变量。全局变量的作用域是整个程序，即在任何一个函数中都可以访问。全局变量一旦被修改，在其他函数中访问到的值也会随之而变。例如：

```
int gVar = 0;                       // 全局变量
void function()
{
    cout<< gVar <<",";              // 输出全局变量的值
    gVar++;                         // 全局变量自增
```

```
}
int main(int argc, char *argv[])   // 主函数
{
    gVar = 1;                      // 修改全局变量
    for( int i=0; i<3; i++ )       // 循环 3 次
    {
        function();                // 调用函数 function
    }
    return EXIT_SUCCESS;           // 主函数返回
}
```

运行上述程序将输出“1,2,3,”，而不是“0,1,2,”。在上述程序中，变量 gVar 声明（第 1 行）在所有函数之外，是全局变量。程序中任何地方都可以访问 gVar，如 function 函数和主函数。主函数（第 9 行）修改了 gVar 的值，其后再调用 function 函数，此时 function 访问 gVar 得到的值就是修改后的值。

鉴于全局变量的特性，可以用全局变量来保存函数的结果，即原来用返回值当做函数的结果值，现在可以用全局变量当做函数的结果值。例如：

```
int gVar = 0;                  // 用于保存计算结果的全局变量
void add ( int a, int b )      // 求和函数
{
    gVar = a + b;              // 将求和的结果保存到全局变量中
}
……
add( 1, 2 );                   // 求 1+2
int result = gVar;             // 从全局变量中获取结果
```

上面的求和函数将结果保存到全局变量中，在调用求和函数之后，可以从全局变量中获取结果。

尽量减少全局变量的使用。虽然在程序中使用全局变量非常方便，可以使数据在函数间共享。但是这样做也使得一个函数的逻辑依赖于另外一个函数的逻辑，函数间的耦合性非常高，导致程序复杂化。另外，如果一个函数赋给全局变量一个错误的值，则会引起其他函数出错，而且这样的错误根源也难以查找。

8.5.3 全局变量的初始化

同局部变量不同，全局变量的初始化是强制性的。如果在定义全局变量时没有给出初始化的值，则全局变量自动初始化为 0。例如：

```
int gVal;
int main(int argc, char *argv[])
{
    cout<<gVal<<endl;
    return EXIT_SUCCESS;
}
```

虽然在定义全局变量 gVar 时没有给出初始化值，但是运行时程序将输出 0。如果程序中定义了多个全局变量，而且都位于同一个源文件中，则这些全局变量会按照其定义时的顺序进行初始化。如果程序中的全局变量分散在多个源文件中，则其初始化的顺序是不一定的，C++标准没有对此做出规定。

8.5.4 多个源文件共享全局变量

全局变量的作用域虽然是整个程序，但在使用时仍然有特殊的要求。假设有两个源文件 file1.cpp 和 file2.cpp，其中 file1.cpp 中定义了一些全局变量，如果 file2.cpp 中的函数要使用这些全局变量，则必须在使用前声明。声明的方法是：

```
extern 类型 全局变量名;
```

其中 extern 关键字表明这个全局变量是在别的文件中定义的，需要在本文件中使用。例如，假设在 file1.cpp 中定义了如下的全局变量：

```
//////////////////////////////////////////////////////////////////////
// file1.cpp   源文件 1
int  gVar1 = 0;
double  gVar2 = 0.0;
```

则在 file2.cpp 中要使用上述两个变量时，必须做如下的声明：

```
//////////////////////////////////////////////////////////////////////
// file2.cpp   源文件 2
extern  int  gVar1 = 0;
extern  double  gVar2 = 0.0;
```

在全局变量前加上 extern 关键字只是用来声明一个全局变量，而不是定义全局变量。如果只有 extern 声明，没有定义，则全局变量仍然不存在。编译器会报告“某某全局变量没有定义”的错误。

8.5.5 静态变量

静态变量分两种，一种是函数内的静态变量，一种是全局的静态变量，其特点是变量定义时带有 static 关键字。例如：

```
static int gVar;            // 在函数外，定义全局的静态变量
void function()
{
    static int iVar;        // 在函数内，定义函数内的静态变量
    ……
}
```

函数内的静态变量也称为局部静态变量，其作用域只限于函数内部，别的函数不能访问。局部静态变量的生命周期同全局变量一样，在整个程序运行期间都有效。例如：

```
void function()
{
    static int iVal = 0;
    cout<< iVal++ <<',';
}
……
for( int i=0; i<3; i++ )
{
    function();
}
```

上述程序运行时将输出“0,1,2,”，而不是“0,0,0,”。这是因为 iVal 是一个局部静态变量，其生命周期在程序运行期间一直有效，所以函数 function 每一次运行结束后，并没有销毁 iVal。而且，函数 function 每一次访问到的 iVal 的值都是上一次函数运行的结果。例如，function 第一次运行后，iVal 的值是 1，第二次运行访问 iVal 的值就是 1，同样第三次访问到的值就是 2。

基于这个特性，可以利用局部静态变量保存每一次函数运行时的状态，以供函数再次被调用时使用。虽然全局变量也可以做到这一点，但是任何函数都可以访问，不利于控制。而局部静态变量只有本函数能够访问，可以有效地限制其作用域。例如，某些严格的安全系统对用户试图登录的次数有限制，可以用静态变量记录这个次数，超过限定次数后则阻止用户继续尝试，用全局变量则给了其他函数修改变量的机会，不符合安全性的要求。

下面的例子用静态变量记录函数被调用的次数（这里是用户试图登录系统的次数），程序如示例代码 8.4 所示。

示例代码 8.4

```
#include <cstdlib>
#include <iostream>
using namespace std;                          // 使用名称空间 std
void login()                                  // 定义登录函数 login
{
    static int sLogNum = 0;                   // 记录登录次数的静态变量
    sLogNum++;                                // 递增登录次数
    if( sLogNum > 3 )                         // 如果超过允许的登录次数
    {
        cout<<"登录次数已超过 3 次，不允许登录！"<<endl; // 输出禁止信息
    }
    else                                      // 如果登录次数在允许范围内
    {
        cout<<"登录成功。"<<endl;             // 输出登录成功的信息
    }
}
int main(int argc, char *argv[])              // 主函数
{
    for( int i=0; i<4; i++ )                  // 尝试 4 次登录系统
    {
        login();                              // 调用登录函数
    }

    system("PAUSE");                          // 等待用户反应
    return EXIT_SUCCESS;                      // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 8.6 所示。

图 8.6　静态变量的使用结果

login 函数中用静态变量 sLogNum 记录被调用的次数，每调用一次，sLogNum 就递增一次。因为是静态变量，所以 sLogNum 只有第一次被初始化为 0，其后每一次访问时都是上一次修改后的值。

8.5.6 全局静态变量

同一般全局变量类似，全局静态变量也是在函数外部定义的变量，只是定义之前带有 static

关键字。例如：

```
static int gVar;
```

跟一般全局变量不同的是：全局静态变量的作用域仅限于定义这个变量的源文件，而不是整个程序。例如，假设程序中有 file1.cpp 和 file2.cpp 两个源文件，其中 file1.cpp 中定义了全局静态变量 gVar，则在 file2.cpp 中试图访问 gVar 时，会遇到一个编译错误："变量 gVar 未定义"，这就是因为 gVar 是定义在 file1.cpp 中的全局静态变量，只能在 file1.cpp 中访问。

8.6 递归函数

如果一个函数在其定义中又调用自身，则称为递归函数，调用自身的过程叫做递归。递归分为直接递归和间接递归。直接递归是指函数直接调用自身，间接递归则指 A 函数调用了 B 函数，而 B 函数又调用 A 函数。函数递归应当有个终止条件，即当某个条件满足时，递归就应当停止。否则递归没完没了地继续下去，程序就会陷入死循环。

有些问题本身就是用递归定义的，适合用递归函数来解决。典型的问题如 Fibonacci 数列：

```
1，1，2，3，5，8，13，21，…
```

在这个数列中，第 1 和第 2 个数都是 1。其后的每一个数，都是这个数前面两个数的和，例如，第 3 个数 2 = 1 + 1，第 4 个数 3 = 1 + 2，第 8 个数 21 = 8 + 13。

如果要求第 n 个数，则首先应当求出第 n-2 和第 n-1 个数。为了求第 n-2 和第 n-1 个数，又需要求第 n-4 和第 n-3 个数……如此递推下去，直至第 1 和第 2 个数，而这两个数是已知的，递归到此终止。递归终止以后，就开始"反递归"，即根据递归终止时求得的值，一步步返回去，求得目标值。

程序员编程时只要编写递归过程，反递归过程则由编译器负责完成。

用递归函数求 Fibonacci 数列中的第 n 个数，n = 1，2，3，…，其程序如示例代码 8.5 所示。

示例代码 8.5

```
#include <cstdlib>
#include <iostream>
using namespace std;                    // 使用名称空间 std
int Fibonacci( int n )                  // 求 Fibonacci 数列中的第 n 个数
{
    if( 1==n || 2==n ){                 // 第 1 和第 2 个数是已知的，都是 1
        return 1;
    }
    return Fibonacci(n-1) +             // F(n) = F(n-1) + F(n-2)
         Fibonacci(n-2);
}
int main(int argc, char *argv[])        // 主函数
{
```

```
    cout<<"Fibonacci 数列中的第 n 个数：<<endl;    // 提示本程序的功能
    int n = 0;                              // 变量 n
    cout<<"请输入 n：";                      // 提示用户输入
    cin>>n;                                 // 将输入保存到变量 n 中
    while( 0 < n ){                         // 当输入小于等于 0 时退出
        cout<<"F("<<n<<") == "              // 调用递归函数，求得数列中的第 n 个数
            <<Fibonacci(n)<<endl<<endl;
        cout<<"请输入 n：";                  // 提示用户输入
        cin>>n;                             // 将输入保存到变量 n 中
    };

    system("PAUSE");                        // 等待用户反应
    return EXIT_SUCCESS;                    // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 8.7 所示。

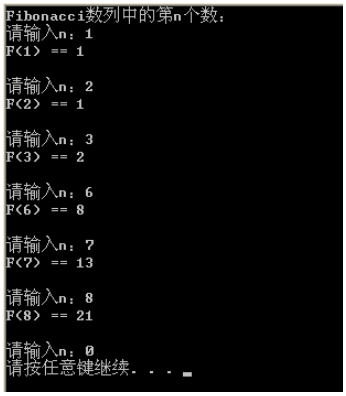


图 8.7　递归求 Fibonacci 数列中的第 n 个数结果

在 Fibonacci 函数的定义中，“1==n || 2==n”就是递归终止条件。如果这个条件不满足，则按照 Fibonacci 的定义 F(n) = F(n-1) + F(n-2)进行递归求解。

虽然递归在解决某些问题方面非常方便，但也是有代价的。每次递归调用都要重新创建一份函数的副本。如果函数中有大量的变量，并且递归层次很多，则内存很快就会被消耗殆尽。因此，应用递归时，函数应当尽量简单，而且要尽量减少递归的层次。

8.7 内联函数

源代码编译完之后，函数就变成了一个指令的集合。调用函数时，系统将跳转到这些指令集的首地址开始运行。当函数返回时，系统就跳回到函数调用处的下一条指令继续执行。不管调用多少

次，每次系统都跳转到同一地址，程序中也只有一个函数的复制。

虽然函数节省了空间，但也不是没有代价。在调用函数的两次跳转过程中，存在一些影响性能的系统开销。如果函数本身非常短小，只有一两条指令，则跳转花费的时间就会占到较大的比重。如果可以避免跳转，则程序的执行效率就会大大提高。例如求和函数：

```
int add( int a, int b )
{
    return a + b;
}
```

求和在计算机中是最基本的运算，只要一条指令就可以完成。如果写成函数，则需要附加两次跳转、参数传递、函数返回等操作，这将极大地影响效率。所以，对于如此简单的功能，最好不要使用函数，而是直接计算。

在 C++中，如果在函数的声明前加上 inline 关键字，则称为内联函数。对于内联函数，编译器不创建真实的函数，而只是在函数调用处展开（即将函数的代码直接复制到调用处）。这样，在“调用”函数时就不用跳转了，避免了使用真实函数的代价。例如，对于 add 函数，如果其声明为：

```
inline int add( int a, int b );
```

函数调用如下：

```
int x = add( 1, 2 );
```

编译后，实际的代码是：

```
int x = 1 + 2;
```

尽管在调用处展开内联函数，同复制函数代码是一样的，但还是使用内联函数方便，否则当需要修改代码时，就要在所有用到的地方进行修改。

如果函数是在头文件中定义的，则编译器会自动将该函数视为内联函数。不过，inline 关键字和在头文件中定义函数，只是对编译器的一种建议。到底要不要将函数作为内联函数，取决于编译器的判断。有的函数是不适合作为内联函数的，如递归函数，或者有很多语句的函数。这样的函数即便是加了 inline 关键字，或者定义在头文件中，编译器依然会将其当做非内联的一般函数对待。

在头文件中定义一个内联函数，求两个整数中的最小值，程序如示例代码 8.6 所示。

示例代码 8.6

```
//////////////////////////////////////////////////////////////////////////
// main.cpp 主程序文件
#include <cstdlib>
#include <iostream>
#include "method.h"                    // 包含方法头文件，使用内联函数 min
using namespace std;                   // 使用名称空间 std
int main(int argc, char *argv[])       // 主函数
{
    int x = 0, y = 0;                  // 用于比较的变量
```

```
    cout<<"请输入两个整数：<<endl;       // 提示用户输入
    cin>>x;                              // 将输入保存到变量中
    cin>>y;
    cout<<"最小值：<<min(x, y)<<endl;   // 输出最小值
    system("PAUSE");                     // 等待用户反应
    return EXIT_SUCCESS;                 // 主函数返回
}
////////////////////////////////////////////////////////////////////////
// method.h
int min( int a, int b )            // 求最小值的函数
{
    return a < b ? a : b;          // 返回两个参数中的最小值
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 8.8 所示。

图 8.8　使用内联函数求最小值的结果

min 函数在头文件 method.h 中定义，编译器自动将其识别为内联函数。并且函数非常短小，所以会在调用处展开。

8.8　函数重载

在 C++程序中不允许有相同的函数出现，否则调用时无法区分到底使用哪一个。区分两个函数靠的不仅是函数名，还有函数的参数列表。如果多个函数拥有相同的函数名，但参数列表不同，则称为函数重载，例如：

```
int  function();
int  function( int );
int  function( double );
int  function( int, double );
```

虽然上述 4 个函数的名字都是“function”，但参数列表不同，所以可以共存于一个程序中。只是在调用时，需要传入不同的实参，以便调用所需的目标函数。

函数的返回类型不能用来区别函数。如果两个函数仅返回类型不同，则第二个出现的函数会被认为是对第一个函数的错误重复。

8.8.1　为什么需要函数重载

其实重载早在 C 语言中就已经存在了，例如各种算术运算符。整数和浮点数在内存中的表示方式是不一样的，但在进行各种算术运算时用的运算符却都是一样的，例如：

```
1 + 2;              // 应用整数的加法
```

```
1.1 + 2.2;          // 应用浮点数的加法
```

上述表达式中就应用了重载的加号运算符。如果不使用重载，那么对于整数的加法和浮点数的加法就要采取两种完全不同的运算符。整数仍然用“+”，而浮点数或许可以用“？”表示加法运算。这样做不仅使程序变得很复杂，也无助于解决问题。所以，只有重载才是解决问题的办法。

C++语言将重载进行了进一步扩展，不仅运算符可以重载，而且函数也可以重载。这样，一个函数名就可以应用到不同的参数列表上，从而对不同的参数列表采取不同的操作。

8.8.2 什么时侯需要重载函数

定义一组函数，其目的相似，但是参数（类型或数量）不同，此时就应当使用函数重载。因为目的相似，所以可以用同一个函数名来标识。或许读者会想到给函数名加上不同的前缀或后缀来区别函数，从而避免重载，但那样做无疑会使程序变得复杂。

“函数重载”重载的其实是函数名，因此准确的说法是“函数名重载”。不过“函数重载”已经受到广泛的认同，在后文中将不加区别地使用这两种说法。

下面的例子用于判断某个数据是不是 0。对于整数，显然只要判断是否与 0 相等即可；对于浮点数，则不能直接比较，通常的做法是判断其绝对值是否小于某个很小的数，例如 1e-6；对于字符，则应当比较其是否等于字符 0，而不是 ASCII 码 0。显然，这些判断目的类似，但目标数据不同，所以应当使用函数重载。程序如示例代码 8.7 所示。

示例代码 8.7

```
#include <cstdlib>
#include <iostream>
using namespace std;                                    // 使用名称空间 std
bool isZero( int x ){                                   // 是否为 0 的整数
    return 0 == x;
}
bool isZero( double x ){                                // 是否为 0 的浮点数
    return -1e-6 < x && x < 1e-6;
}
bool isZero( char x ){                                  // 是否为'0'的字符
    return '0' == x;
}
int main(int argc, char *argv[])                        // 主函数
{
    cout<< "Is Zero?" <<endl;                           // 输出提示
    cout<< 1 <<'\t' << isZero(1) <<endl;                // false
    cout<< 0 <<'\t' << isZero(0) <<endl;                // true
    cout<< 1.0 <<'\t' << isZero(1.0) <<endl;            // false
    cout<< 1e-5 << '\t' << isZero(1e-5) <<endl;         // false
    cout<< 1e-7 << '\t' << isZero(1e-7) <<endl;         // true
    cout<< 'a' << '\t' << isZero('a') <<endl;               // false
    cout<< '0' << '\t' << isZero('0') <<endl;               // true
```

```
    system("PAUSE");                                    // 等待用户反应
    return EXIT_SUCCESS;                                // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 8.9 所示。

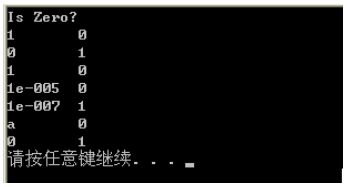


图 8.9　用重载函数判断是不是 0 的结果

从判断是否为 0 的角度来讲，对于任何数据的判断都只用 isZero 这个函数名就足够了。但其内部操作不同，而且处理的参数也不同，所以应当使用重载。

8.8.3 函数重载解析

如果一个函数名被重载，那么在调用时选择函数的过程就是重载解析。例如下面的重载函数：

```
void  f( int, int );
void  f( float, float );
int  main() {
    ……
    f( t1,t2 );
    return 0;
}
```

函数重载解析过程将决定：用实参 t1 和 t2 能否调用函数 f（两个重载函数中的一个），用 t1 和 t2 调用是否存在二义性，到底调用的是 f(int，int)还是 f(float，float)。函数重载解析的过程有如下 3 个步骤：

step 1　确定实参列表的属性，确定候选函数的集合。

step 2　根据实参的个数和类型确定合适的函数。

step 3　选择精确匹配的函数。

下面将按照上述步骤依次来解析下述代码：

```
void  f();
void  f( int );
void  f( double, double = 3.4 );
void  f( char*, char* );
int main() {
    f( 7.6 );
    return 0;
}
```

函数重载解析的第一步是确定候选函数。候选函数是与被调用函数同名的函数，并且在调用点上其声明可见。在本例中有 4 个候选函数：f()，f(int)，f(double, double)以及 f(char*, char*)。

函数重载解析的第一步还要确定实参的属性，即其数目和类型。在本例中实参表由一个 double

型的实参构成。

第二步根据实参属性从候选函数中选择合适的函数。合适的函数的参数个数应与实参列表中的参数数目相同 ,并且实参与该函数形参的类型之间必须存在转换。在这个例子中有两个合适的函数：f(int)和 f(double, double)。f(int)只有一个参数，而且存在从实参类型 double 到形参类型 int 之间的转换。f(double, double)也是一个合适的函数，因为其第二个参数给出了默认值，而第一个形参类型是 double，与实参类型精确匹配。

如果函数重载解析过程的第二步没有找到任何合适的函数，则函数调用就是错误的。

函数重载解析的第三步是选择精确匹配的函数。为了选择这个函数 ,从实参类型到相应可行函数参数所用的转换都要划分等级。最精确匹配的函数应符合以下的标准：

- 应用在实参上的转换，不比调用其他可行函数所需的转换差。
- 在某些实参上的转换，要比其他可行函数对该参数的转换好。

当考虑可行函数 f(int)时，应用的转换是个标准转换：将 double 型的实参转换成 int 型。当考虑可行函数 f(double)时，实参的类型 double 与相应的参数精确匹配，因此该调用的最佳可行函数是 f(double, double)。

如果函数重载解析的第三步没有找到最精确匹配的函数，则该函数调用是有二义的，即没有找到一个比其他可行函数都好的函数。

8.9 函数指针

所谓函数指针就是指向函数的指针。函数指针也是一个变量，可以指向不同的函数。同时通过函数指针可以调用其指向的函数，从而使得函数调用更加灵活。

8.9.1 函数地址

函数也是有地址的。编译之后的函数，其实是一组指令的集合。这样一组指令在程序运行时存在于内存中，其起始地址就是该函数的地址，也称做函数的入口地址。在编写程序时，可以用函数名来表示函数的地址，也可以在函数名之前加上取地址符号“&”表示函数的地址。这两种方式是等价的。例如对于下列求和函数：

```
int Add( int a, int b ){
    return a + b;
}
```

函数名“Add”以及“&Add”都表示 Add 函数的地址，如图 8.10 所示。

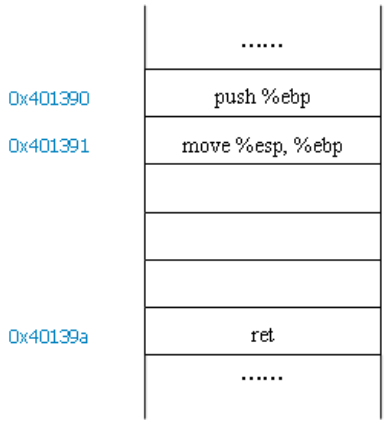


图 8.10　内存中的 Add 函数

如图 8.10 所示，内存地址 0x401390 即函数 Add 在内存中的地址。可以使用强制类型转换，将该地址保存在一个长整型变量中：

```
long a = (long) Add;                    // 使用 C 语言方式的类型转换
long b = (long)&Add;
long c = static_cast<long>( Add );      // 使用 C++语言方式的 static_cast 操作符
long d = static_cast<long>( &Add );
```

另外，也可以通过库函数或者 cout 对象将 Add 函数的地址输出到显示器上：

```
printf("%p", Add); // 调用 printf 函数，输出 Add 函数的地址
printf("%p", &Add);
cout<< hex << Add; // 使用 cout 对象输出 Add 函数的地址，其中 hex 表示以 16 进制方式输出
cout<< hex << &Add;
```

8.9.2 定义函数指针

定义函数指针包括两个部分，即声明和初始化。在函数声明的基础上做一个小小的改变，就可以将其变成一个函数指针的声明。例如对于 Add 函数，只要将函数名 Add 替换成一个函数指针名，例如“pf”，并在其前面加上“(*”、在后面加上“)”即可：

```
int (*pf)( int a, int b);           // 定义函数指针变量 pf
```

函数指针名“*pf”两侧的括号不能省略，否则就成了一个返回“int *”类型的函数声明。正是这个括号使得星号“*”和标识符“pf”组成一个整体，表示 pf 是一个指针。

对于函数指针的声明，参数列表和返回类型是其关键部分。只要这两项确定，就确定了该函数指针的类型。这个类型也决定了该函数指针所能指向的函数类型，即具有同样参数列表和返回类型的函数。

有的时候函数的参数列表和返回值比较复杂，每次定义这样的函数指针都要重写一遍比较烦琐。因此可以用类型定义运算符“typedef”为该函数定义一个简单的类型名。有了这样一个类型名

之后，就可以用来定义函数指针变量，而不用重写函数参数列表和返回类型。例如：

```
typedef int ( * FUN_PTR ) ( int a, int b );    // 定义函数指针类型 FUN_PTR
FUN_PTR pf1;
FUN_PTR pf2;
```

虽然相比定义函数指针变量，定义函数指针类型只是多了一个 typedef 关键字。但也正因为如此，FUN_PTR 也不再是一个变量，而变成了一个类型。同普通指针一样，如果没有明确初始化，则函数指针的值是一个随机数，使用这样的指针非常危险。因此在使用函数指针之前必须对其进行初始化或者赋一个初值，即将一个函数名赋给该函数指针变量。例如：

```
int Add( int a, int b ){       // 定义函数 Add
    return a + b;
}
// 定义一个接受两个 int 型参数并返回 int 型值的函数指针变量 pf
int ( * pf ) ( int a, int b );
pf = Add;                      // 用 Add 函数给 pf 赋初值
// pf = &Add;                  // 同上一行等价
int ( * pf1 ) ( int a, int b ) = Add;
typedef int ( * FUN_PTR ) ( int a, int b );
FUN_PTR pf2 = Add;
FUN_PTR pf3 = &Add;
```

8.9.3 使用函数指针调用函数

使用函数指针调用函数同函数调用一样，只要在函数指针后面加上实参列表即可。例如：

```
int Add( int a, int b ){           // 定义函数 Add
    return a + b;
}
// 定义一个接受两个 int 型参数，并返回 int 型值的函数指针变量 pf
int ( * pf ) ( int a, int b );
pf = Add;                          // 将 pf 指向 Add
int x = pf( 3, 4 );                // 通过函数指针 pf 调用函数 Add
```

正如上例第 7 行所示，使用函数指针时不必像使用一般指针那样解引用。不过有时为了明确起见，也可以解引用。例如：

```
int x = ( * pf ) ( 3, 4 );         // 函数指针解引用
```

这样做的好处是可以明确指明 pf 是一个函数指针，否则只有看到定义，才能分辨出 pf 到底是一个函数指针，还是一个函数。

8.9.4 函数指针的用途

如果仅仅是像上节那样使用函数指针，那就有点儿画蛇添足了，还不如直接使用函数来得方便。实际上，函数指针通常用做回调函数。所谓回调函数是指将一个函数用函数指针保存下来，然后直到需要的时候再进行调用。至于函数指针到底保存的是什么函数，则由设置者决定，而调用者是无从知晓的。这个调用过程称做回调，这个函数指针称做回调函数指针，简称回调函数。

比如现在有一个排序函数 sort，用来排序一个数组。但是按照什么标准排序（从大到小，还是从小到大），则需要由 sort 函数的调用者确定。调用者可以通过给函数传递一个函数指针来确定排序的标准。下面使用回调函数确定排序标准，程序如示例代码 8.8 所示。

示例代码 8.8

```
#include <cstdlib>
#include <iostream>
using namespace std;                                  // 使用名称空间 std
typedef bool (*FUN_PTR)( int a, int b );              // 定义函数指针类型
void sort(int ary[], int n, FUN_PTR pf )              // 排序函数
{
    for( int i=0; i<n-1; i++ )                        // 遍历第 1 个到第 n-1 个元素
    {
        int index = i;
        int val = ary[i];
        for( int j=i+1; j<n; j++ )                    // 遍历当前元素之后的所有元素
        {
            if( pf(val, ary[j]) )                     // 调用回调函数
            {
                val = ary[j];                         // 保存目标值
                index = j;                            // 保存目标值所在位置
            }
        }
        if( val != ary[i] )                           // 如果目标值与当前元素不同
        {
            ary[index] = ary[i];                      // 调换当前元素与目标元素的值
            ary[i] = val;
        }
    }
}
int main(int argc, char *argv[])
{
    cout<<"——使用函数指针确定排序标准——"<<endl;
    int ary[3] = {5, 3, 6};

    bool less(int a, int b);                      // 声明函数 less
    sort(ary, 3, &less);                          // 使用 less 回调函数，从大到小排列数组

    for( int i=0; i<3; i++ )                      // 输出排序后的结果
    {
```

```
        cout<<ary[i]<<' ';
    }
    cout<<endl;

    bool big(int a, int b);            // 声明函数 big
    sort(ary, 3, &big);                // 使用 big 回调函数，从小到大排列数组
    for( int i=0; i<3; i++ )           // 输出排序后的结果
    {
        cout<<ary[i]<<' ';
    }
    cout<<endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
bool less(int a, int b)                // 比较两个参数的大小
{
    return a < b;
}
bool big(int a, int b)                 // 比较两个参数的大小
{
    return a > b;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 8.11 所示。

图 8.11 使用回调函数排序结果

8.10 综合实例

本节将结合具体的实例详细讲解如何使用 C++中的函数。希望读者能自行运行本节的实例，以便对 C++中的函数功能有深入的了解。

8.10.1 判断素数

在前面的章节中曾经做过有关素数的练习，但没有用函数，现在用一个函数来判断某个数是否是素数，程序如示例代码 8.9 所示。

示例代码 8.9

```
#include <cstdlib>
#include <iostream>
#include <cmath>
using namespace std;                        // 使用名称空间 std
bool IsPrimeNumber( int num )
{
```

```
    if( 1 == num || 2 == num )              // 如果是1或者2，直接返回true
    {
        return true;
    }
    int s = static_cast<int>(sqrt(num));    // 求平方根
    for( int i=2; i<=s; i++ )               // 从2开始遍历，直到平方根
    {
        if( (num % i) == 0 )                // 如果可以被整除，则不是素数
        {
            return false;
        }
    }

    return true;
}
int main(int argc, char *argv[])            // 主函数
{
    cout<<"——判断是否素数——"<<endl;         // 输出提示信息

    int num = 0;                            // 保存输入值的变量
    cout<<"请输入一个正整数："<<endl;       // 输出提示信息
    cin>>num;                               // 输入
    while( num > 0 )                        // 只有正整数才能判断是否是素数
    {
        if( IsPrimeNumber( num ) )          // 调用函数判断num是否是素数
        {
            cout<<num<<"是一个素数"<<endl;  // 输出提示信息
        }
        else
        {
            cout<<num<<"不是一个素数"<<endl; // 输出提示信息
        }
        cout<<endl<<"请输入一个正整数："<<endl;   // 输入下一个数
        cin>>num;
    }

    system("PAUSE");                        // 等待用户反应
    return EXIT_SUCCESS;                    // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 8.12 所示。

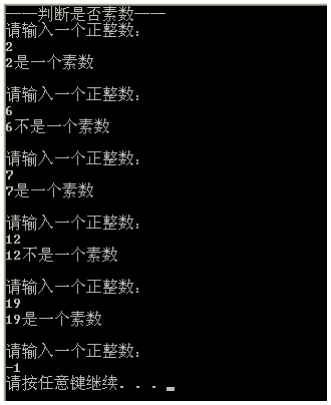


图 8.12 使用函数判断素数结果

8.10.2 分割字符串

对于一个字符串，有时可能需要按照某个标识字符分为两部分，即标识字符之前作为一个字符串，之后作为另一个字符串。下例就是用来实现上述功能的，它根据某个字符，将目标字符串分成两部分，并返回该分界字符的位置，程序如示例代码 8.10 所示。

示例代码 8.10

```
#include <cstdlib>
#include <iostream>
using namespace std;                    // 使用名称空间 std
int SplitString( char *dest,            // 目标字符串
                 char token,            // 分界字符
                 char *str1,            // 分界字符前的字符串
                 char *str2 )           // 分界字符后的字符串
{
    assert( dest );                     // 检验参数的有效性
    assert( str1 );
    assert( str2 );

    int index = 0;                      // 保存分界位置的变量
    bool befToken = true;               // 是否在标志前
    while( *dest != '\0' )              // 遍历整个目标字符串
    {
        if( *dest != token )            // 如果当前字符不是分界字符
        {
            if( befToken )              // 如果在分界字符前
            {
                index++;                // 分界位置加 1
                *(str1++) = *(dest++); // 保存分界字符前的字符串
            }
            else
            {
```

```
                *(str2++) = *(dest++);  // 保存分界字符后的字符串
            }
        }
        else
        {
            befToken = false;           // 跨越了分界字符
            dest++;                     // 处理下一个字符
        }
    }
    *str1 = '\0';                       // 标识字符串的结尾
    *str2 = '\0';
    return index;
}
int main(int argc, char *argv[])
{
    cout<<"——分割字符串——"<<endl;      // 输出提示信息

    const int LEN = 16;                 // 字符串长度变量

    cout<<"请输入字符串：<<endl;        // 输出提示信息
    char dest[LEN]={'\0'};              // 初始化目标字符串
    cin>>dest;                          // 输入目标字符串

    cout<<"请输入一个分界字符："<<endl;  // 输出提示信息
    char token = ' ';                   // 分界字符变量
    cin>>token;                         // 输入分界字符
    cout<<endl;

    char str1[LEN] = {'\0'};            // 初始化字符串
    char str2[LEN] = {'\0'};

    int index = SplitString( dest, token,  // 调用分割字符串的函数
                str1, str2 );
    cout<<"字符串在位置"<<index          // 输出分割结果
        <<"处分割成两个字符串："<<endl;
    cout<<str1<<'\t'<<str2<<endl;
    system("PAUSE");                    // 等待用户反应
    return EXIT_SUCCESS;                // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 8.13 所示。

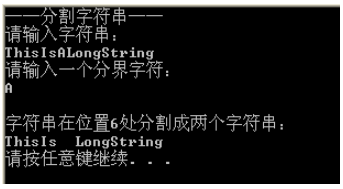


图 8.13 分割字符串结果

8.11 小结

本章重点讲述了函数的定义和使用，以及参数传递、函数重载等。通过函数，可以将实现某种功能的一系列语句组织成一个整体，以后使用时只要使用函数名并传入相应的参数即可，而不必重复实现该功能。但是 C++是一种强类型的语言，对于参数传递有着严格的类型限制。为了突破这个限制，使得一个函数可以处理多种数据类型，C++引入了函数模板的概念，这将在下一章进行讲述。

◆ ◆ ◆

第 9 章　函 数 模 板

本章包括

- 函数模板的定义
- 函数模板的实参推演
- 函数模板的编译方式
- 函数模板定义中的标识符解析
- 函数模板的特化和重载
- 函数的匹配规则

函数模板是 C++的新特性 ,这个概念在 C 语言中是没有的。函数模板不是一个实实在在的函数，而是对逻辑功能相同、但数据类型不同的一组函数的统一描述。利用函数模板，可以对函数的类型（包括返回类型和参数类型）进行参数化处理，也就是函数的类型也可以像变量一样改变。利用函数模板可以用一种逻辑过程，处理不同类型的数据，从而极大地提高编程的效率。本章的重点就是学习函数模板的定义和使用。

9.1　为什么要使用函数模板

模板是对类型的参数化。对函数的类型（包括返回类型和参数类型）进行参数化，就可以使用函数处理某个类型范围内的若干种类型的对象，而函数体则不需要改变。C++是一种强类型语言，也就是说，C++中数据的类型一旦确定，就不能修改了，这种特性也体现在函数上。C++函数在声明和定义时，必须指定函数的返回类型以及各个参数的类型。调用函数时也必须按照函数的声明，传入适当类型的实参，并用相应类型的变量保存函数的返回值。如果要用这个函数处理其他类型的参数，则必须定义新的函数。

虽然调用函数时实参和形参的类型可以不一致，但由于存在类型转换，数据精度可能会受到影响，而这种影响通常是不能被接受的。所以必须针对不同的数据类型，重新定义函数的各个版本。

例如定义一个简单的求数组中最小值的函数，要求可以处理各种数据类型。如果不使用模板，那么开发者不得不针对每种类型写一个定义：

```
int min ( int ary[], int n )           // 整型数组求最小值
{
    int  minVal = 0;                   // 保存最小值的变量
    for( int i=0; i<n; i++ )           // 遍历数组
    {
        minVal = minVal < ary[i] ? minVal : ary[i];        // 记录最小值
    }
    return minVal;                     // 返回最小值
}
double min( double ary[], int n )      // 求 double 型数组中的最小值
```

```
{
    double  minVal = 0.0;   // 以下的逻辑过程同上面的求整型数组中最小值的逻辑一致
    for( int i=0; i<n; i++ )
    {
    minVal = minVal < ary[i] ? minVal : ary[i]; // 记录最小值
    }
    return minVal;                    // 返回最小值
}
……
```

这里只写出了针对 int 型和 double 型数组的函数。为了适应各种情形，还应当编写针对 float，char，short，long，unsigned int 等所有数据类型的函数。除此之外，还要支持自定义数据类型（如结构体、类）。如此一来，代码量就会大大增加。而且，一旦算法逻辑发生改变，或者要纠正某个错误，所有的重载函数都需要修改。显然这种面面俱到的方式并不是一种好的做法。

如果简单分析一下上述代码就会发现：所有这些函数在算法逻辑上都是一样的，不同的只是所处理数据的类型。如果能将函数的返回类型以及参数类型当做变量对待，当需要处理不同类型数据的函数时，只要将这些“类型变量”赋值为所需的类型就好了。这样，只编写一次通用算法逻辑就可以处理所有的数据类型，这就是函数模板的含义。函数模板示意图如图 9.1 所示。

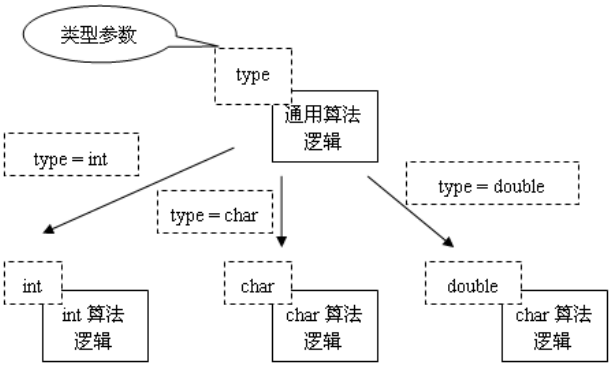


图 9.1 函数模板示意图

下面的代码使用函数模板改进了上述求最小值程序，如何定义并使用函数模板将在后面的章节中讲解。

```
template< typename T >          // 定义函数模板，T 是类型“变量”，代表某种类型
T min( T ary[ ], int n )        // 使用类型 T 定义函数的返回值，参数类型
{
    T minVal = 0;               // 使用类型 T 声明临时变量
    for( int i=0; i<n; i++ )
    {
        minVal = minVal < ary[i] ? minVal : ary[i];        // 记录最小值
    }
    return minVal;              // 返回最小值
}
……
int a1[3] = { 1, 5, 3 };                // 以下是使用函数模板 min 的例子
```

```
int m1 = min( a1, 3 );                  // 求整型数组的最小值
double a2[3] = { 1.2, 1.0, 3.4 };
double m2 = min( a2, 3 );               // 求浮点型数组的最小值
```

显然，通过使用函数模板，避免了定义多余的重载函数，提高了开发效率。

9.2 定义函数模板

定义函数模板其实同定义函数的差别不大，其核心都是定义一个算法逻辑。唯一不同的地方就是函数模板将其使用的类型进行了参数化，从而达到了这样目的：对不同类型的数据，应用同一算法逻辑。

9.2.1 抽取通用算法逻辑

定义函数模板，常常是出于这样的动机：定义一个对各种数据类型都通用的算法逻辑。譬如定义一个表示加法的函数，要求传入两个参数，并返回这两个参数的和。事实上，参数的类型以及返回类型都不是问题的核心，关键的问题是加法。而且对于常见的各种数据类型，如整型、浮点型等，都只要使用加法运算符“+”即可。所以这里的通用算法逻辑就是两个参数相加，并返回其结果。抽取通用算法逻辑如图 9.2 所示。

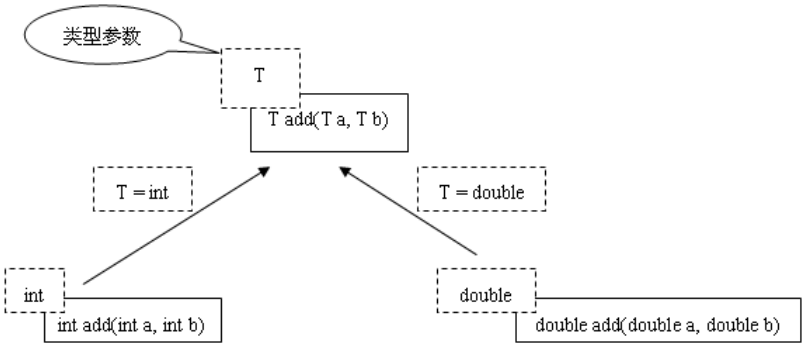


图 9.2 抽取通用算法逻辑

实际上，不仅仅类型可以作为模板参数，常量也可以作为模板参数。比如在上节中定义的求数组最小值的函数模板中，数组大小也可以作为模板参数。

所谓通用算法逻辑只是一个相对的概念，对于某些特殊的数据类型，或许并不适合该算法。比如字符串的相加（表示字符串连接），直接使用加法运算符“+”就没有意义。对于这种特殊的情况，应当对函数模板进行特化处理，具体细节请读者参考有关模板特化的小节。

9.2.2 语法

在 C++中定义函数模板使用 template 关键字，其后是用逗号分隔的模板参数列表，用尖括号“<”和“>”括起来，该列表不能为空。模板参数可以是一个类型参数，代表一种类型；也可以是一个非类型参数，代表一个常量。其语法如下：

```
template< 模板参数列表 >
返回类型 函数名 ( 函数参数列表 )
{
    函数体
}
```

模板参数列表中的每个参数用逗号分隔，并且不可为空。可以是一个或多个类型参数，也可以是一个或多个非类型参数，甚至可以是两种参数的混合。对于类型参数，前面要带上 typename 关键字；对于非类型参数，可以像声明普通参数一样声明。例如：

```
<typename T1, typename T2, int num, double value>
```

上述模板参数列表中声明了两个类型参数 T1 和 T2，代表两种数据类型，可以是内置的类型，也可以是自定义类型。同时，该模板参数列表还声明了两个非类型参数 num 和 value，表示该模板在定义过程中可以将 num 和 value 当做常量使用。

类型参数也是一个标识符，符合标识符的要求，即只能由数字、字母和下画线组成，并且不能以数字开头。

typename 关键字也可以用 class 关键字代替，两者意义相同，都可以用来声明类型参数。但是用 typename 更好一些，可以明白地表示后面的参数是一个“类型名”。而且 typename 是 C++标准化的产物，而 class 关键字则是为了支持 C++标准化之前的程序而保留下来的。

在函数参数列表以及函数体的定义过程中，可以使用模板参数列表中的参数。如果是类型参数，可以用该类型声明函数的返回值、形参、临时变量；如果是非类型的参数，可以在函数体的定义中当做常量使用，也可以当做函数的默认实参。

模板参数列表之后就是一个参数化的函数定义。这里所谓的参数化，主要指的是对函数返回类型、参数类型的参数化，以及对函数中常量类型的参数化。也就是说，在使用函数模板时，可以根据需要设定上述数据的具体类型。

例如定义一个求最小值的函数模板，对于整数、浮点数其计算过程都是一样的，不一样的只是参数和返回值的类型，所以可以将这两部分参数化，其定义如下：

```
template<typename  Type>
Type  min ( Type ary[], int n )     // Type 型数组求最小值
{
    Type  minVal = 0;               // 保存最小值的变量
    for( int i=0; i<n; i++ )        // 遍历数组
    {
        if( ary[i] < minVal )       // 如果当前元素比当前最小值还小
        {
            minVal = ary[i];        // 修改保存最小值变量
```

```
        }
    }
    return minVal;                  // 返回最小值
}
```

在上述函数模板 min 的定义中，第一个 Type 是函数返回类型的参数，第二个 Type 和后面的 ary[]表示 Type 类型的数组形参。第四行的 Type 用来在函数体中定义一个变量，该变量的类型是 Type 类型。Type 是 min 模板的模板参数，Type 可以用来表示多种数据类型，只是此时还未确定，要到模板 min 被使用时才能确定。

模板的类型参数 Type 与具体类型的关系有点儿类似变量和变量值的关系。Type 是一个类型变量，其类型就是“类型”，而其值就是某种具体类型，如 int，double 等。

9.2.3 使用非类型参数

函数模板中使用非类型参数，其目的就是为函数引入一个常量，以供定义函数时使用。当然这个常量也可以作为函数参数的默认值使用。例如，对于一个数组求最小值，可以用一个非类型的参数表示数组长度。下面的例子使用非类型参数的常量来定义函数模板。

```
template <class Type, int size>// 定义函数模板，类型参数 Type，非类型参数 size
Type min( Type ary [ ], int n=size )
                                // 接受类型为 Type 的数组，以及默认为 size 的数组长度
{
    Type minVal = 0;            // 定义 Type 类型的变量 minVal
    for( int i=0; i<n; i++ )    // 遍历数组
    {
        if( ary[i] < minVal )   // 如果当前元素比最小值还小
        {
            minVal = ary[i];    // 修改最小值
        }
    }
    return minVal;              // 返回最小值
}
```

C++标准规定模板参数必须在程序编译时确定。因此非类型的模板参数的值必须是一个常量，而且是编译时就能确认的常量，包括字面常量、符号常量(const 常量)。

9.3 使用函数模板

由于函数模板不是函数，所以使用函数模板也不是函数调用。在使用函数模板之前，必须先确定模板的参数。在 C++中，函数模板参数的确定可以由用户明确指定，也可以由编译器自行推导。本节主要讲述如何使用函数模板以及模板参数的推演。

9.3.1 实例化函数模板

函数模板不是真正的函数，不能直接使用，必须先实例化。实例化的过程就是根据一组或更多具体类型或值构造出具体的函数。这个过程是隐式的，编译器会根据使用模板时的情况自动构造。例如对于函数模板：

```
template <typename Type, int size>
Type min( Type (&r_array)[size] )
{
    Type min_val = r_array[0];
    for ( int i = 1; i < size; ++i )
    if ( r_array[i] < min_val )
        min_val = r_array[i];
    return min_val;
}
```

如果在程序中有下列语句：

```
int ia[] = { 10, 7, 14, 3, 25 };
double da[7] = { 10.2, 9.1, 14.5, 3.2, 25.0, 19.8 };
int i = min( ia );
double d = min( da );
```

则函数模板 min 被实例化两次。一次是针对含 5 个 int 类型数的数组类型，另一次是针对含 7 个 double 类型数的数组类型。类型参数 Type 和非类型参数 size 都被用做函数参数。为了判断用做模板实参的实际类型和值，编译器需要检查函数调用中提供的函数实参的类型。在上述例子中 ia 是含 5 个 int 类型数的数组，da 是含 7 个 double 类型数的数组，编译器根据这两个实参决定模板实参的类型和值。

函数模板 min 被实例化后，形成了两个具体的函数，就像是程序员直接写成的那样。只不过这两个函数作为编译器的中间结果，只对编译器是可见的，而程序员看不到。函数模板实例化过程如图 9.3 所示。

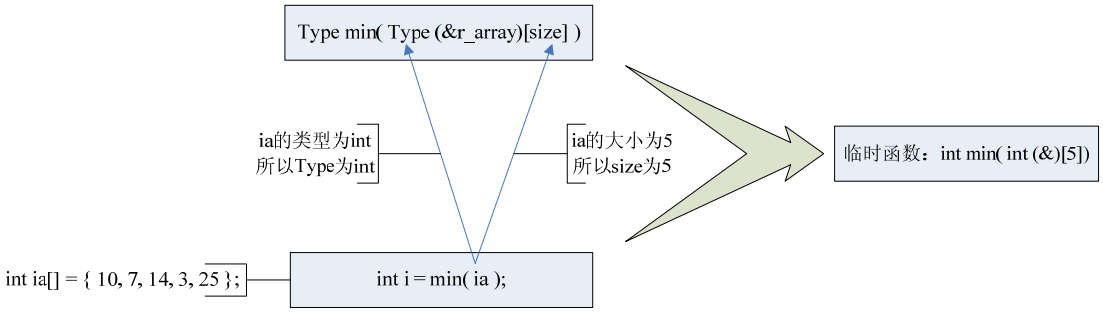


图 9.3 函数模板实例化过程

下面定义一个函数模板，用于在一个数组中查找某个值。如果找到了，则返回该值在数组中的位置；如果找不到，就返回-1。程序如示例代码 9.1 所示。

示例代码 9.1

```
#include <cstdlib>
#include <iostream>
using namespace std;                    // 使用名称空间 std
```

```
template <typename T, int size>    // 函数模板参数，包括类型参数 T 和数值参数 size
int find( T (&ary)[ size ], T var ) // 定义查找函数：在 T 类型数组 ary 中查找元素 var
{
    for( int i = 0; i < size; i++ )    // 遍历整个数组
    {
        if( var == ary[i] )            // 如果找到
        {
            return i;                  // 返回当前位置
        }
    }
    return -1;                         // 如果没找到就返回-1
}
int main(int argc, char *argv[])       // 主函数
{
    cout<<"——实例化模板——"<<endl;
    int ia[5] = {1, 2, 3, 4, 5};       // 整型数组，5 个元素
    double da[6] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};     // 浮点型数组，6 个元素
    cout<<"整数 3 的位置: "<< find( ia, 3 )<<endl;        // 在 ia 中查找 3
    cout<<"整数 6 的位置: "<< find( ia, 6 )<<endl;        // 在 ia 中查找 6
    cout<<"浮点数 3.3 的位置: "<< find( da, 3.3 )<<endl;  // 在 da 中查找 3.3
    cout<<"浮点数 7.7 的位置: "<< find( da, 7.7 )<<endl;  // 在 da 中查找 7.7

    system("PAUSE");                   // 等待用户反应
    return EXIT_SUCCESS;               // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 9.4 所示。

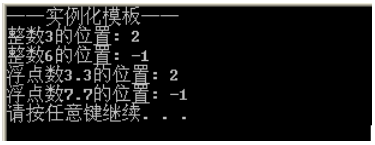


图 9.4　使用查找函数模板结果

因为目标数组的类型多种多样，所以在定义查找函数时不能假定参数的类型，应当使用函数模板。在主函数中，使用函数模板 find，编译器根据参数的类型和值决定模板参数，从而创建函数模板的实例。

9.3.2　取函数模板的地址

取函数模板的地址时也会导致函数模板的实例化。因为函数模板并不是真正的函数，所以其本身也没有地址。但是 C++并不把“&函数模板名”(取地址符号‘&’也可以省略) 这样的用法当做一个错误，而是先对该函数模板进行实例化，然后取实例化所生成函数的地址。

取函数模板的地址实际上是取函数模板实例的地址。函数模板在程序运行时并不存在，存在的只是其实例。函数模板实例化发生在编译期，而取地址则发生在程序运行期。

同取普通函数的地址一样，在取之前需要先定义一个指向该函数模板实例的函数指针。该函数指针的类型就是用来实例化函数模板的参数。例如：

```
template <typename Type, int size>
Type min( Type (&r_array)[size] )          // 定义一个求数组中最小值的函数模板
{
    Type min_val = r_array[0];
    for ( int i = 1; i < size; ++i )
    if ( r_array[i] < min_val )
        min_val = r_array[i];
    return min_val;
}
int main(int argc, char *argv[])           // 主函数
{
    typedef int(*PF)(int (&)[3]);
                        // 定义一个函数指针类型，接受含 3 个整数的数组，返回 int 类型值
    PF pf = min;                           // 定义并初始化数组
    int ary[3] = {5, 3, 9};                // 取函数模板地址
    cout<<"数组中的最小值："<< pf( ary );  // 输出结果值
}
```

在上述例子中，函数指针 pf 用模板 min 的地址进行初始化。pf 的类型是一个接受拥有 3 个整型元素的数组，并返回一个整型数的函数。当用模板 min 的地址初始化函数指针 pf 的时候，编译器就会通过 pf 的类型定义来获取模板实例化参数，并进行实例化。实例化的结果是一个实实在在的函数，运行时就可以获取其地址。

在本例中，实例化得到的结果就是 int min(int (&)[3])。

在取函数模板实例的地址时，模板实参必须是确定的、唯一的类型或值。如果不唯一，则会导致编译错误。

例如，在下面的例子中，函数模板的实例是一个函数实参，但该函数又存在重载，这就使得编译器无法确定具体的模板实例化参数。

```
template <typename Type, int size>
Type min( Type (&r_array)[size] )          // 定义一个求数组中最小值的函数模板
{
    Type min_val = r_array[0];
    for ( int i = 1; i < size; ++i )
    if ( r_array[i] < min_val )
        min_val = r_array[i];
    return min_val;
}
typedef int(*INT_FUNC)(int (&)[ 3 ]);
typedef double(*DOU_FUNC)(double (&)[4]);
```

```
void func( INT_FUNC ) {……}            // 重载函数 1
void func( DOU_FUNC ) {……}            // 重载函数 2
int main () {
func( & min );                         // 错误！无法决定使用哪一个重载函数版本
}
```

在这个例子中，func 函数有两个重载版本，其接受的参数分别为两个不同类型的函数指针：INT_FUNC 和 DOU_FUNC。如果将函数模板 min 作为参数传递给 func 函数，则用这两种类型的函数指针去实例化 min 模板就会导致冲突。因为无法判断到底应该用哪种类型的函数指针去实例化模板。要解决这个问题，可以用以下两种方法：

- 一种是明确指明模板实例化参数。
- 一种是使用强制类型转换。

```
int main() {
    func( min<int, 3> );                        // 明确指明模板实例化参数
    func( static_cast< INT_FUNC >( min ) );     // 强制转换指定实参类型
}
```

明确指明模板实例化参数的方法也称为显式模板实参，具体说明请参考后面的章节。

9.3.3 函数模板实参的推演

在前述的各个例子中，读者可以看到函数模板的调用方法同普通函数的调用方法没有什么两样。可是函数模板并不是真正的函数，必须实例化后才能使用。因此，在调用函数模板（实际上是调用实例化得到的函数）的过程中必然存在一个实例化的过程，而函数模板的实例化必须先求得模板实参。

当函数模板被调用时，编译器根据函数的实参决定模板实参，这个过程称为模板实参推演。在推演过程中，函数的返回类型并不参与其中，这主要是因为函数调用可以不取返回值。既然程序中没有返回值信息，那么也无法根据返回类型来进行实参推演。函数模板实参推演的具体步骤如下：

step 1 依次检查每个函数实参，以确定在每个函数参数的类型中出现的模板参数。

step 2 如果找到模板参数，则通过检查函数实参的类型，推演出相应的模板实参。

step 3 函数的形参和实参的类型不必完全匹配，只要能够将实参转换为形参的类型即可。

函数模板实参推演的前两个步骤比较简单，只要根据函数实参和模板参数出现的顺序，一一对应即可找到相应的参数。例如对于下面的程序：

```
template <typename T, int size>
int find( T (&ary)[ size ], T var );
double da[6] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
int x = find( da, 3.3 );
```

程序中存在一个函数模板 find，接受两个参数，分别是 T 类型的数组，大小为 size，以及一个要查找的数，类型也是 T。第 4 行是这个函数模板的调用，传入的实参分别是数组 da 和 3.3。针对这个函数模板调用进行实参推演，过程如下：

step 1 找到实参 da 和 3.3，其类型都是 double。

step 2 根据函数模板的声明，通过实参数组 da 得到第一个模板类型参数 T 的值是 double，通过实参 3.3 得到第二个模板类型参数 T 的值也是 double，同时，数组 da 的大小是 6，所以模板实参 size 的值也是 6。

多个函数实参可以参加同一个模板实参的推演过程。如果模板参数在函数参数表中出现多次，则每个推演出来的类型都必须相同。例如：

```
template <class T> T min5( T, T ) { /* ..... */ }
unsigned int ui;
min( ui, 1024 );
```

在调用函数模板 min 时，根据第一个参数推演出来的模板实参是 unsigned int，而根据第二个参数推演出来的是 int。这两个类型不同，因此这个函数模板不能这么使用，而应当如下调用：

```
min( ui, 1024u );
```

这样，根据两个函数实参推演出来的模板实参都是 unsigned int，符合要求。

9.3.4 显式指定函数模板的实参

显式指定函数模板的实参也称为显式实例化，其作用主要是解决模板实参推演时的二义性问题。既然指定了模板实参，那么在使用函数模板时就不必进行实参推演了，也就避免了实参推演的二义性问题。例如上节的 min 模板，可以如下显式指定其模板实参：

```
min<unsigned int>( ui, 1024 );
```

在上述例子中，通过模板实参表<unsigned int>强制将函数模板 min 的实参指定为 unsigned int。虽然其第二个函数参数 1024 并不是 unsigned int 类型的，但是因为模板实参已经指定，所以在编译器中 1024 也会被当成 unsigned int 使用，二义性的问题也就不存在了。

上述调用的第二个实参是字面常量 1024，其类型是 int。因为通过显式模板实参已经推定函数的参数类型为 unsigned int，所以参数 1024 会被转换为类型为 unsigned int 的参数。

9.4 实参推演中的类型转换

在函数模板实参的推演过程中，函数实参的类型不一定要严格匹配相应模板实参的类型。只要能够将函数实参转换成相应的模板实参即可。下列三种类型转换是允许的：

- 左值转换。
- 限定符修饰转换。
- 到基类的转换（该基类根据一个类模板实例化而来）。

9.4.1 左值转换

所谓左值，就是可以放在赋值表达式左面的值，也就是可以改变值的程序实体，变量、引用就是左值。所谓右值，就是放在赋值表达式右面的值，也就是可以从中读取数据的程序实体，例如各种常量、表达式等，变量也是右值。左值和右值可如下理解：

```
左值 = 右值
```

左值转换包括从左值到右值的转换、从数组名到指针的转换 ,以及从函数名到函数指针的转换。这里所说的转换，并不是将左值变成右值，而是说将左值当做右值使用。事实上，在 C++中所有的左值都可以从中读取数据，而这个读取的过程就是所谓的左值到右值的转换。

数组名和函数名是右值，而指针和函数指针是左值。数组名和函数名都可以转换为相应的指针。被关键字 const 修饰的符号常量是右值，而普通的变量则是左值。普通变量可以转换为相应的符号常量。

假设有如下的函数模板：

```
template <typename T, typename T2>
void Fun( const T v1, T2 *v2 )
{
    ……
}
```

在下面的程序中，虽然传递给模板 Fun 的并不是符号常量和指针，但是编译器依然可以推导出正确的模板实参。

```
int main(int argc, char *argv[])                // 主函数
{
    int a = 12;                                 // 普通整型变量
    double ary[] = {1, 2, 3};                   // 双精度浮点型数组

    Fun( a, ary );                              // 使用函数模板 Fun
    system("PAUSE");                            // 暂停
    return EXIT_SUCCESS;                        // 返回
}
```

变量 a 是一个普通的整型变量，是一个左值。a 作为实参，对应模板 Fun 的第一个参数。该参数是一个被 const 修饰的右值。由于左值可以转换为右值，所以可以推定 T1 的类型为 int。同样，由于数组可以转换为指针，所以可以推定 T2 的类型是 double。

9.4.2 限定符修饰转换

限定修饰符包括 const 和 volatile。限定符转换指的是普通的指针变量可以转换为 const 或者 volatile 型指针。如果模板实参和形参之间存在这样的转换，那么编译器也可以推导出正确的模板实参类型。

volatile 的含义是“易变的；变化多端的”，该关键字通常用在多线程程序中。如果一个变量没有被 volatile 修饰，则编译程序通常会对其取值过程进行优化。即不必每次都去内存中取值，而是尽可能在 CPU 的寄存器中取值。但是在多线程程序中，该变量在内存中的值可能会被其他线程修改，如果只是在寄存器中取值，有可能取到的不是最新的值。如果变量用 volatile 修饰，则编译程序不会优化，而是每次都到内存中取值。

限定符修饰转换指的是可以将限定修饰符 const 和 volatile 施加到指针变量上。在进行模板实参推演时，如果实际给出的参数是指针，但模板形参是被 const 或 volatile 修饰的指针，则编译器仍然可以推导出正确的模板实参。例如：

```
#include <cstdlib>
#include <iostream>
#include <string>                          // 包含头文件 string
using namespace std;                       // 使用名称空间 std
template<typename T>
void Foo( const T *ptr )                   // 定义函数模板 Foo
{
    cout<< *ptr << endl;
}
struct Student                             // 定义学生类
{
    string id;                             // 学号
    string name;                           // 名字
};
ostream & operator<<(ostream &os,          // 重载输出运算符
                 const Student &s )
{
    os<<s.id<<' '<<s.name;                 // 输出学生的学号和名字
    return os;
}
int main(int argc, char *argv[])           // 主函数
{
    int a = 123;                           // 定义整型变量
    char b = 'x';                          // 定义字符型变量
    Student s;                             // 定义学生对象
    s.id = "001";                          // 学号
    s.name = "Mike";                       // 名字

    Foo( &a );                             // 调用模板 Foo
    Foo( &b );                             // 调用模板 Foo
    Foo( &s );                             // 调用模板 Foo
   system("PAUSE");                        // 暂停
   return EXIT_SUCCESS;                    // 返回
```

```
}
```

在上述程序中，函数模板 Foo 接受的参数是一个 const 指针。但是在实际使用该模板时传入的函数实参却是普通的指针。不过，由于编译器在进行函数模板的实参推演时，允许 const 修饰转换。所以，依然可以精确地推定模板的实参，依次为 int，char 和 Student。

9.4.3 到基类的转换

有的时候，模板的函数参数是一个类模板，而该类模板的某个实例有派生类。在使用该函数模板时，传递的参数可能是其派生类的对象，而不是基类的对象。尽管不能跟函数模板声明中的形参精确匹配，但是由于允许将派生类转换为基类，所以编译器仍然能够推演出正确的模板实参。

所谓类模板，简单来说就是定义一个类时，类中的某些数据类型进行了参数化。有关类模板的详细说明，请参见后面的章节。

例如，存在如下的类模板：

```
template< typename T >                    // 类模板参数 T
class Base                                // 类模板 Base
{
    ……
};
```

该类模板存在如下的派生类模板：

```
template< typename T >                    // 类模板参数 T
class Derived : public Base<T>            // 类模板 Derived 派生自 Base
{
    ……
};
```

如果存在如下的函数模板：

```
template< typename T >                    // 函数模板参数
void Foo( Base<T> obj )                   // 函数模板 Foo
{
    ……
}
```

函数模板 Foo 的函数形参是类模板 Base 某个实例的对象。

在使用模板 Foo 时，传递的函数实参可以是 Derived 的某个实例的对象，例如 Derived<int> iObj 或者 Derived<double> dObj。这个实参不必一定是 Base 的某个实例的对象。这是因为在进行模板实参推演时，允许派生类到基类的转换。例如：

```
int main(int argc, char *argv[])          // 主函数
{
```

```
    Derived<double> d;                          // 定义 Derived<double>对象
    Foo( d );                                   // 调用函数模板 Foo
    system("PAUSE");                            // 暂停
    return EXIT_SUCCESS;                        // 返回
}
```

在上述程序中，虽然 Derived<double>并不是 Base<double>，但是编译程序依然可以推演出正确的模板实参 double。

9.5 函数模板的编译

函数模板明显不同于普通的 C++语言特征。其他语言特征，例如数据类型、运算符、类等，可以跟机器语言一一对应。但模板定义的仅仅是一个逻辑过程，还必须指定具体的数据类型才能发挥作用。因此模板的编译比较特殊，不像普通的语言特征那样即刻就可以编译出相应的机器码。

9.5.1 函数模板的两种编译方式

对于函数模板的编译，C++标准规定了方式，即包含式和分离式，下面详细讲解这两种编译方式。

1. 包含编译方式

在包含编译方式下，模板在头文件中定义。如果某个源文件需要实例化模板，就包含模板所在的头文件，例如：

```
//////////////////////////////////////////////////////////////////////////
// template.h
template <typename Type>        // 包含方式：模板定义在头文件中
Type min( Type t1, Type t2 ) {
    return t1 < t2 ? t1 : t2;
}
```

在每个使用 min 实例的文件中都要包含该头文件，例如：

```
//////////////////////////////////////////////////////////////////////////
// some.cpp
#include "template.h"           // 在使用模板实例之前包含模板定义

......
int i, j;
double dobj = min( i, j );
```

虽然定义模板的头文件可以被包含在许多源程序文件中，但是这并不意味着每次调用前都要进行实例化。对于某种类型，编译器只实例化模板一次。但是，真正的实例化动作发生在何时何地，要取决于编译器的具体实现。

2. 分离编译方式

在分离编译方式下，头文件中是函数模板的声明，而模板定义则放在另外的源文件中，例如：

```
//////////////////////////////////////////////////////////////////////////
```

```
// template.h
template <typename Type>           // 分离方式：头文件中只提供模板声明
Type min( Type t1, Type t2 );
```

在源文件中定义函数模板：

```
//////////////////////////////////////////////////////////////////////
// template.cpp
export  template <typename Type>   // 模板定义
Type min( Type t1, Type t2 ) {
    return t1 < t2 ? t1 : t2;
}
```

使用函数模板 min 实例的程序只需在使用该实例之前包含这个头文件：

```
//////////////////////////////////////////////////////////////////////
// some.cpp
#include "model2.h"                // 使用模板的源文件包含模板头文件

……
int  i, j;
double d = min( i, j );
```

在定义模板的源文件中，在关键字 template 前加上关键字 export，表示声明一个可导出的函数模板。

虽然分离方式能够很好地分离模板的声明和定义，但是并不是所有的编译器都支持这种编译方式，即使支持也未必总能支持得很好。所以，在定义模板时最好总是放在头文件中，否则程序的可移植性就难以保证。

9.5.2 函数模板实例的编译时机

函数模板本身并不能直接使用，必须实例化后才能使用。但是由于数据类型众多，不能为每一种数据类型都实例化一个模板实例，所以编译器采取的策略是直到使用时才编译相应的实例。例如在程序中有一个函数模板 Add 的定义：

```
template<typename T>                    // 模板参数列表
T Add( T t1, T t2 )                     // 加法模板
{
    return t1 + t2;                     // 返回两个参数的和
}
```

当遇到该模板定义时，编译器只是保存了该模板的内部形式，而不直接进行编译，直到在程序中遇到类似如下的使用才编译：

```
int main(int argc, char *argv[])        // 主函数
{
    cout<< Add( 1, 2 ) <<endl;          // 调用模板
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

在上述程序中，当编译器遇到第 3 行的语句时，由于使用了 Add(1, 2)，所以编译器才真正实例化模板 Add，并编译相应的实例函数“int Add(int, int)”。

但是，模板的某个实例可能会被多次使用，而且是在不同的地方（可能在不同的函数中）。而且由于程序的运行是动态的，编译器无法确知模板的某个实例何时被第一次使用。对于早期的 C++编译器，这是一个难以解决的问题，所以通常的策略是编译每一个函数模板实例。当然这么做也有问题，会导致模板的同一实例被编译多次。尽管编译器会从多个编译结果中选择一个作为最终结果，但是编译的时间有可能会大大延长。

为此，C++标准允许使用显式实例化声明，用来确定模板实例化的时间，从而解决一个模板实例被多次编译的问题。其语法如下：

```
template 返回值 模板名< 模板参数列表 >( 函数形参列表 );
```

注意，在上述语法中关键字 template 是必不可少的。另外在函数模板名称之后要显式地指定模板参数。例如，模板 Add 的 int 型实例的显式声明如下：

```
template int Add<int>( int, int );
```

尽管 C++标准没有规定模板实例显式声明的位置，但是模板某个类型实例的显式声明在程序中只能出现一次。如果出现两次或两次以上，则会导致编译错误。另外，在显式声明出现之前，模板的定义必须是可见的，否则也会导致编译错误。

模板实例的显式声明只对一个源文件有效。如果在其他源文件中也使用到了该实例，那么为了消除多余的模板实例，就需要借助一个编译选项，该选项在不同的 C++编译器中有不同的名字。例如在 IBM 编译器 Visual Age for C++ for indows 3.5 版本中，该选项为/ft-，在其他编译器中的名字请参看相应的编译器手册。

9.6 函数模板定义中的标识符解析

在函数模板的定义中，会用到很多标识符，其中有一些可能是用户定义的函数或者函数模板。编译器在处理函数模板的定义时，一个很重要的任务就是要找到被调用的函数或者使用哪个函数模板实例，这就是所谓的标识符解析。

例如，在下面的例子中定义了一个模板 Foo，该模板又调用了函数 printData 来打印参数。但是该函数已被重载，因此编译器在处理模板 Foo 时，应当进行必要的标识符解析。

```
#include <cstdlib>
#include <iostream>
using namespace std;            // 使用名称空间 std
void printData(int n)           // 打印整型数的函数 printData。该函数定义必须放在
{                               // 模板 Foo 的定义之前，否则会导致编译错误
    cout<< n;
}
template<typename T>
void Foo( T t1 )                // 模板 Foo
{
    printData( 123 );           // 打印整型数
```

```
    cout<<endl;
    printData( t1 );            // 打印参数 t1
    cout<< endl;
}
void printData(double v)        // 打印浮点数的函数 printData。该函数定义可以放在
{                               // 模板 Foo 的定义之前，也可以放在之后。但必须放在
    cout<< v;                   // Foo( 3.14 )之前
}
int main(int argc, char *argv[])   // 主函数
{
    Foo( 3.14 );                // 以浮点数为参数，调用模板 Foo
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

在上述程序中，模板 Foo 的定义中两次调用了函数 printData。不过，第一次调用的参数是整形数，第二次调用的参数是模板 Foo 的函数形参 t1，而 t1 的类型未知，依赖于 Foo 的模板实参。

由于第一次调用所引用的 printData 重载版本是确定的，所以其相关定义必须出现在模板 Foo 的定义之前。第二次调用所引用的 printData 重载版本并不确定，依赖于模板实参，所以相关定义并不要求出现在 Foo 之前，但是，这个相关定义必须出现在 Foo 的相关实例之前。例如，在主函数中调用 Foo(3.14)，这意味着 Foo 的实例中要调用 printData(3.14)，所以 printData 的 double 型重载版本必须在此之前定义。

C++标准规定，编译器对模板定义中的标识符解析应当分为两个步骤：首先，不依赖于模板参数的标识符在模板定义时被解析；其次，依赖于模板参数的标识符在模板实例化时被解析。

9.7 函数模板的特化

很多时候，定义一个适合所有类型的函数模板非常困难，这主要是因为对于某种逻辑操作，各种数据类型的实现并不相同。例如，同样是比较大小，整型和浮点型数的比较就与字符串类型的比较不同。整型和浮点型数据只要调用各种比较运算符（<，>，==等）即可，但是字符串的比较就需要调用专门的比较函数，例如 strcmp 和 wstrcmp。而自定义数据类型，如结构体、类，比较大小就更加特殊。在这种情况下就需要用到函数模板的特化。假设函数模板 max 的定义如下：

```
// 通用的模板定义
template <class T>
T max( T t1, T t2 ) {
    return (t1 > t2 ? t1 : t2);
}
```

如果用 const char*型实参实例化 max 模板，程序员的本意是比较字符串的大小，但实例化的结果却是比较两个指针的大小。为了获得正确的语义，必须针对 const char*类型，为函数模板 max 提供特化的版本。在函数模板特化定义中先是关键字 template 和一对尖括号“<>”，然后是函数模

板特化的定义。该定义包括被特化模板的返回类型、模板名、模板实参列表以及函数参数表和函数体。其语法如下：

```
// 模板特化定义
template <> 特化返回类型 模板名<模板实参列表> ( 函数参数列表 )
{
    // 模板函数体
}
```

例如，针对 const char *类型，模板 max 显式特化为：

```
typedef const char *PCC;                          // 定义 const char *的类型别名
template<> PCC max< PCC >( PCC s1, PCC s2 ) {  // 用 const char *特化模板 max
    return ( strcmp( s1, s2 ) > 0 ? s1 : s2 );
                                                  // 调用库函数 strcmp 比较字符串的大小
}
```

由于有了这个特化版本，当在程序中调用函数 max(const char*,const char*)时，真正被调用的是特化的版本，而不是用类型 const char*实例化的模板。对所有用两个 const char*型实参进行调用的 max 都会调用这个特化的定义，而对于其他的调用则都是先实例化模板，然后再调用。

对于模板实参列表，如果函数模板的返回类型与函数参数的类型相同，则该列表也可以省略。例如在上例中，参数的类型和返回类型都是 PCC，所以上述特化模板也可以写成 template<> PCC max(PCC s1, PCC s2) {……}。

下面定义一个函数模板，表示两个数据相加，并用 char *特化该模板，即对于字符串，该模板的实际定义是两个字符串相连。程序如示例代码 9.2 所示。

示例代码 9.2

```
//////////////////////////////////////////////////////////////////////
// main.cpp
#include <cstdlib>
#include <iostream>
#include "template.h"                          // 包含模板 add 的定义头文件
using namespace std;                           // 使用名称空间 std
int main(int argc, char *argv[])               // 主函数
{
cout<<"——模板特化——"<<endl;                    // 输出提示信息
    cout<<"1 + 2 = "<< add ( 1, 2 ) <<endl;            // 用 int 实例化 add
    cout<<"1.1 + 2.2 = "<< add ( 1.1, 2.2 ) <<endl;    // 用 double 实例化 add
    char s1[16] = "Hello ";
    char s2[6] = "World";
    cout<<"Hello + World = "<< add ( s1, s2 ) <<endl;  // 用 char *特化 add

    system("PAUSE");                                   // 等待用户反应
    return EXIT_SUCCESS;                               // 主函数返回
}
```

```
////////////////////////////////////////////////////////////////////////
// template.h
#include <string.h>
template <typename T>                          // 通用模板定义
T add ( T a, T b ){
    return a + b;                              // 两数相加
}
template <>                                    // 针对 char *类型，特化模板 add
char* add ( char *a, char *b ){
    return strcat( a, b );                     // 连接字符串
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 9.5 所示。

图 9.5　模板特化结果

在模板的定义文件 template.h 中，定义了模板 add 的普通版本和特化版本。事实上，特化版本也可以放在别的地方，只要在引用前可见即可，例如在 main 函数定义之前。

9.8 函数模板的重载

和普通函数一样，函数模板也可以重载。重载的函数模板，模板名称相同，但函数形参列表不同。例如：

```
template <typename Type>
Type min( const Type*, int );          // 注意第二个参数
template <typename Type>
Type min( Type, Type );                // 两个参数类型相同
```

在上述定义中，两个模板的模板形参相同，都是一个类型（typename Type），模板名称也相同，都是 min。但函数形参列表不一样，一个是(const Type *, int)，另一个是（Type, Type）。所以这两个函数模板是重载的。同函数重载一样，函数模板的返回类型并不是重载的判断条件。因为在某些情况下，调用函数模板并不需要考虑返回值。下面的主函数的定义说明了前面定义的两个模板如何使用。

```
#include <cmath>
int main()
{
    int ia[1024];
    ……
    int ival = min( ia, 1024 );    // Type == int; min( const int*, int )
    double dval = min( 1.1, 2.2 ); // Type == double; min( double, double )
    return 0;
}
```

在上述主函数的定义中，第 6 行的第二个参数是一个整型数 1024。从定义上来讲，此处更适

合第一个函数模板，所以，该处的调用得到的模板实例就是第一个模板的实例。

如果存在多个候选的模板，那么在实参推演过程中，编译器倾向于选择那些函数实参跟形参类型相近的模板。例如在上面的例子中，实参 1024 是一个整型，而候选函数模板中，其中一个的形参也是整型，那么该模板就会被选来进行实例化。

尽管定义重载函数模板同定义重载函数一样简单，但二者也有不同。重载函数没有二义性的问题，而重载函数模板则可能导致二义性。例如重载模板 min，使其既可以支持类型参数相同的情况，也可以支持类型参数不同的情况：

```
template <typename T>
int min( T, T ) { /* ..... */ }                    // 两个参数类型相同的函数模板
template <typename T, typename U>
int min( T, U );                                   // 两个参数类型不相同的函数模板
```

如果以如下的形式使用函数模板，将会导致二义性：

```
min ( 1024, 512 );
```

因为在这个例子中，编译器无法决定到底该实例化哪个模板。第二个函数模板，虽然其模板参数声明为两个类型（T 和 U），但并没有限制这两个类型必须不同，所以语句“min(1024, 512);”也可以看做是调用第二个模板的实例。显然这样做会引起与第一个模板的冲突，从而导致编译错误。

但是，在这种情况下其实可以不必重载函数模板。因为所有能够匹配 min(T, T)的调用，也完全可以匹配 min5(T, U)。所以应该只提供 min(T, U)的声明，而 min(T, T)应该被删除。

在某些情况下，尽管可以采用重载函数模板，但在进行程序设计时，仍然应当尽量少地使用，以避免不小心带来的二义性。

9.9 函数匹配规则

函数模板可以与普通函数同名。例如在一个程序中可以同时具有一个 min 函数，也可以有一个或多个 min 函数模板。在这种情况下编译器需要决定是用非模板函数，还是函数模板实例。C++ 标准规定：非模板函数具有最高的优先权，如果不存在匹配的非模板函数，则最匹配的和最特化的函数具有高优先权。

如果在多个重载的函数模板中，有一个模板在进行模板实参推演时不需要类型转换，则该模板就是最匹配的。如果对于调用时的实参类型，存在特化的版本，并且类型也不需要类型转换，则该模板称为最特化的。

看下面的例子：

```
template <class T> void f(T);                  // (1)
template <class T> void f(int, T, double);     // (2)
template <class T> void f(T*);                 // (3)
```

```
template < > void f<int> (int);              // (4)
void f(double);                              // (5)
bool b;                                      // bool变量
int i;                                       // int变量
double d;                                    // double变量
f(b);                                        // 以 T = bool 调用 (1)
f(i,42,d);                                   // 以 T = int 调用(2)
f(&i);                                       // 以 T = int* 调用(3)
f(d);                                        // 调用(5)
```

9.10 综合实例

本节将给出几个使用函数模板的完整实例，来说明如何定义和使用函数模板。标准C++的标准模板库（STL）是一个比较完备的模板库，读者在学习模板的时候可以参考该库的实现。有关STL的具体内容，可参见后续的章节。

9.10.1 数组求和函数模板

数组可以支持的数据类型多种多样，如整型、浮点型、字符串等。本实例要求定义一个函数模板，其参数是数组名和元素个数，返回值是数组中各元素的和。由于字符串比较特殊，所以应当专门针对字符串数组进行特化。

在后续的章节中读者可以看到一个用户自定义的类也可以进行相加操作，只要该类支持相关的运算符（如“+”和“+=”）即可。对于这些自定义类的数组，也可以应用本实例的数组求和模板。

程序如示例代码9.3所示。

示例代码 9.3

```
#include <cstdlib>
#include <iostream>
using namespace std;                         // 使用名称空间 std
template<typename T, int size>
T SumArray( T (&ary)[size] )                 // 数组求和模板
{
    T sum = 0;
    for( int i=0; i<size; i++ )              // 遍历数组
    {
        sum += ary[i];                       // 求和
    }
    return sum;                              // 返回和
}
```

```
template<int size>
char * SumArray( char *(&ary)[size] )      // 数组求和模板针对字符串数组的特化
{
    int n = 0;
    for( int i=0; i<size; i++ )            // 遍历数组
    {
        n += strlen( ary[i] );             // 求结果字符串的长度
    }
    char *sum = new char[n+1];              // 为结果字符串分配空间
    sum[0] = '\0';                         // 初始化结果字符串
    for( int i=0; i<size; i++ )            // 遍历字符串数组
    {
        strcat(sum, ary[i]);               // 连接各个字符串
    }
    return sum;                            // 返回结果字符串
}
int main(int argc, char *argv[])           // 主函数
{
    // 普通数组求和
    int nAry[5] = {1, 2, 3, 4, 5};         // 整型数组
    cout<< SumArray( nAry ) <<endl;        // 数组求和，并输出结果
    double  dAry[5] = {1.1, 2.2, 3.3, 4.4, 5.5}; // 浮点型数组
    cout<< SumArray( dAry ) <<endl;        // 数组求和，并输出结果
    // 字符串数组求和
    char* chAry[2];                        // 定义字符串数组
    chAry[0] = new char[6];                // 为数组中的字符串分配空间
    chAry[0][0] = '\0';                    // 初始化
    chAry[1] = new char[6];
    chAry[1][0] = '\0';
    strcpy( chAry[0], "Hello" );           // 为数组中的字符串赋值
    strcpy( chAry[1], "World" );
    char *pRes = SumArray( chAry );        // 连接字符串数组
    cout<< pRes <<endl;                    // 输出字符串数组连接后的结果
    delete []pRes;                         // 回收内存
    delete chAry[0];                       // 回收内存
    delete chAry[1];                       // 回收内存
    system("PAUSE");                       // 暂停
    return EXIT_SUCCESS;                   // 返回
}
```

在上述实例中，模板 SumArray 的函数形参是一个数组的引用。注意，这里必须是数组的引用。因为只有这样，数组长度才可以被当做类型的一部分。否则一个

数组实参只能当做一个指针传递给模板 SumArray 的实例，从而导致实参推演不能得到模板实参 size。

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 9.6 所示。

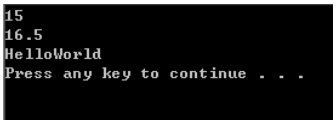


图 9.6　数组求和结果

9.10.2 数组排序函数模板

数组排序也是编程中常用的算法。同数组求和一样，为了支持不同数据类型的数组，有必要将排序算法设置成函数模板。

本实例中用到的排序算法是选择排序算法。该算法的思想是：依次遍历数组中未排序的部分，从中选择一个最小值，并将该值插入到已排序部分的后一位，重复执行上述步骤，直到数组中的所有元素都已排好序。

定义一个函数模板，对数组进行排列。由于字符串之间的比较操作不能使用关系运算符（<，<=，>，>=），所以应当针对字符串数组进行特化。程序如示例代码 9.4 所示。

示例代码 9.4

```
#include <cstdlib>
#include <iostream>
using namespace std;
template<typename T, int size>
void SortArray( T (&ary)[size] )           // 数组排序模板
{
   for( int i=0; i<size-1; i++ )           // 遍历数组
   {
      T minVal = ary[i];                   // 暂将当前元素当做最小值
      int k = i;                           // 记录当前位置
      for( int j=i+1; j<size; j++ )        // 遍历未排序元素
      {
         if( ary[j] < minVal )             // 如果未排序的元素比当前最小值还小
         {
            minVal = ary[j];               // 更新最小值
            k = j;                         // 记录新的最小元素的位置
         }
      }
      if( k != i )                         // 如果最小元素不是当前元素
      {
```

```
            ary[k] = ary[i];                    // 交换当前元素与最小元素的值
            ary[i] = minVal;
        }
    }
}
template<int size>
void SortArray( char *(&ary)[size] )        // 数组排序模板针对字符串数组的特化
{
    for( int i=0; i<size-1; i++ )           // 遍历数组
   {
        char *pMinVal = ary[i];             // 暂将当前字符串当做最小字符串
        for( int j=i+1; j<size; j++ )       // 遍历未排序字符串
        {
            if( strcmp(ary[j], pMinVal) < 0 )// 如果未排序字符串小于当前最小字符串
            {
                pMinVal = ary[j];           // 记录新的最小字符串的位置
            }
        }
        if( pMinVal != ary[i] )             // 如果最小字符串不是当前字符串
        {
            char *temp = new char[ strlen(ary[i])+1 ];
            strcpy( temp, ary[i] );         // 保存当前字符串的内容
            strcpy( ary[i], pMinVal );      // 交换当前字符串与最小字符串的内容
            strcpy( pMinVal, temp );
           delete []temp;
        }
   }
}
int main(int argc, char *argv[])
{
    // 普通数组排序
    int nAry[5] = {4, 2, 5, 3, 1};          // 整型数组
    SortArray( nAry );                      // 排序
    for( int i=0; i<5; i++ )                // 遍历数组
    {
        cout<< nAry[i] << ' ';              // 输出数组内容
    }
    cout<<endl;
    double  dAry[5] = {1.1, 2.2, 3.3, 4.4, 5.5};   // 浮点型数组
    SortArray( dAry );                      // 排序
    for( int i=0; i<5; i++ )                // 遍历数组
    {
        cout<< dAry[i] << ' ';              // 输出数组内容
    }
    cout<<endl;
    // 字符串数组排序
    char* chAry[3];                         // 定义字符串数组
    chAry[0] = new char[4];                 // 为数组中的字符串分配空间
```

```
    chAry[0][0] = '\0';                    // 初始化
    chAry[1] = new char[4];
    chAry[1][0] = '\0';
    chAry[2] = new char[4];
    chAry[2][0] = '\0';
    strcpy( chAry[0], "bbb" );             // 为数组中的字符串赋值
    strcpy( chAry[1], "aaa" );
    strcpy( chAry[2], "ccc" );
    SortArray( chAry );                    // 排序
    cout<< chAry[0] <<endl;                // 输出排序结果
    cout<< chAry[1] <<endl;
    cout<< chAry[2] <<endl;
    delete chAry[0];                       // 回收内存
    delete chAry[1];
    delete chAry[2];
    system("PAUSE");                       // 暂停
    return EXIT_SUCCESS;                   // 返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 9.7 所示。

图 9.7　数组排序结果

9.11 小结

使用模板可以将处理数据的算法进行泛化。泛化的好处是可以将一种算法应用到多种不同的数据上，从而避免相同逻辑的无意义重复，减少代码量。但是对于某些数据类型应用某个函数模板可能并不合适，此时就可以针对该数据类型对函数模板进行特化处理。

在程序中使用函数模板，实际上是使用该模板的实例。在程序编译过程中，编译器根据实参类型推演出具体的模板参数，然后用这些模板参数实例化模板，并将产生的实例编译成具体的机器码。模板实参推演并不要求参数类型完全匹配，如果在实参和形参之间存在左值转换、限定修饰符转换或到基类的转换，那么编译器也可以推演出正确的数据类型。

模板的编译有两种方式：一种是包含式，即模板的定义在头文件中；另一种是分离式，即模板的声明和定义分别放在头文件和源文件中。目前大多数编译器只支持前者，即包含式，但包含式有可能导致重复实例化。此时可以使用模板显式实例化声明，并结合抑制隐式实例化的编译选项，来强制实例化模板。函数模板也可以重载，即定义名称相同但函数形参列表不同的模板。注意，在重载函数模板的实例化过程中，有可能导致二义性的问题。

◆　　◆　　◆

第 10 章　错误与异常处理

本章包括

- 利用函数返回值识别错误
- 如何处理错误
- 什么是异常
- 如何抛出异常
- 如何捕获异常
- 怎么自定义异常类

在 C++中，函数的返回值可以被用来标志函数处理成功或者失败。异常处理是 C++语言的一个主要特征，它提供了完美的出错处理方法。使用异常处理，可以提高程序的健壮性。本章就重点讲述怎么使用异常来提高程序的健壮性。

10.1　识别和处理错误

当程序得到的数据不在期望的范围之内时，就是个错误。错误的数据通常都不能被处理，必须筛选出来。错误通常指程序可以产生预期的结果，但是这个结果不是程序员期望的。

10.1.1　利用函数返回值识别错误

在设计程序的时候，不但要考虑程序的功能，还要考虑程序的可维护性。把所有的代码都放在一起，是开发人员的恶梦。好的做法是，根据不同的功能，把代码放在不同的文件中，使用函数把代码分成小功能块。一般情况下，一个函数的代码不应该超过 200 行。如果代码超过 200 行，就要考虑把其中一部分代码定义为函数。

因此，对于一个大的系统，函数的调用关系会很复杂，函数的调用链会很深。如图 10.1 所示为笔者参与开发的某系统的一个调用栈，可以看到，这个调用栈有 38 层。

调用栈展示的是在运行过程中函数调用的关系。在 Visual Studio 开发环境下，当调试程序的时候，可以通过 Alt+F7 快捷键调出函数调用栈。

函数调用栈越深，系统越脆弱。当调用中某一个函数出现错误的时候，会影响上层函数的处理，很容易导致系统崩溃。出问题的函数越靠近调用链的尾部，出问题造成的破坏越大。

为了使系统更健壮，必须要考虑错误处理。没有错误处理的系统，总是假设一切情况都是正常的，而实际上这样的系统是很脆弱的。因为错误是无处不在的，造成这个错误的原因，可能是程序存在 bug、逻辑错误等。因此对错误的处理是非常必要的。

错误处理的一种方法是用函数的返回值来表示处理的结果。函数在处理数据的时候，如果发现了错误的数据，可以用一个特殊的结果来表示处理不成功。比如下面是一个常用的输出函数的定义：

```
int printf(const char *,……);
```

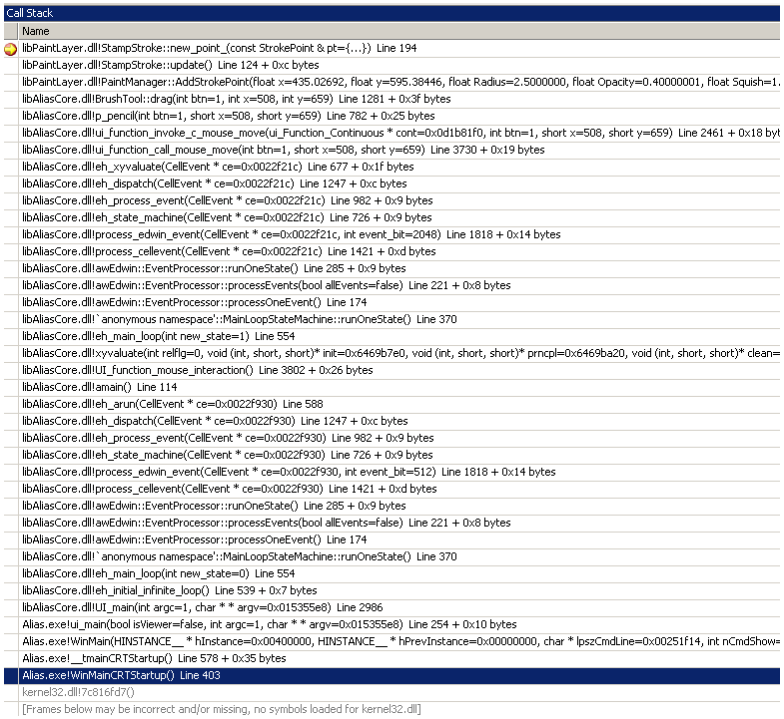


图 10.1　函数调用栈

上述输出函数有一个返回值，代表了该输出语句输出的字符的个数。如果该函数执行失败，则返回一个负数。

错误一般是指该结果不是期望的，但是可以预料到，可以想办法避免进一步带来的不正常。

10.1.2 对错误结果做出相应处理

要设计健壮的系统，必须要考虑到错误的情况，对函数的返回结果进行处理。在大型的系统设计中，对错误处理的代码要占到所有代码的一半以上。继续使用错误的结果，会导致不可预料的结果。因此错误剔除得越早，造成的危害就越小。

下面是一个关于文件复制操作的例子，使用了简单的错误处理。例子提供了简单的文件复制功能，从源文件复制到目标文件。系统要求源文件必须存在，而目标文件则不做要求。因此当源文件不存在时，复制不能进行。在例子中对文件操作函数的返回结果进行了判断，当出现错误的时候终止操作。详细的程序如示例代码 10.1 所示。

示例代码 10.1

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main()
{
```

```
FILE * fin;
FILE * fout;
char src_name[128];
char dst_name[128];
cout<<"请输入源文件的文件名："<<endl;
cin>>src_name;
cout<<"请输入目标文件的文件名："<<endl;
cin>>dst_name;
fin = fopen( src_name,"r" );
if( NULL == fin )
{
    cout<<"打开源文件失败!"<<endl;
    return 0;
}
fout = fopen( dst_name, "w" );
if( NULL == fout )
{
    cout<<"打开目标文件失败！"<<endl;
    fclose( fin );
    return 0;
}
//文件复制
while( true )
{
    char buffer[128];
    int count = fread( buffer, sizeof(char), 128 , fin );
    if( count == 0 )
    {
        cout<<"读文件错误！"<<endl;
        break;
    }
    int write_count = fwrite( buffer, sizeof(char), count, fout );
    if( write_count < count )
    {
        cout<<"写文件错误!"<<endl;
    }
    if( feof( fin ) )
            break;
}

//关闭文件
if( fclose( fin ) )
{
    cout<<"关闭源文件失败!"<<endl;
}
if( fclose( fout ) )
{
    cout<<"关闭目标文件失败!"<<endl;
}

cout<<"复制完成!"<<endl;
return 0;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 10.2 所示。

图 10.2　文件复制错误结果

出现这样的结果是因为源文件 src.txt 不存在，而源文件是以读模式打开的，必须要存在。在工程的目录下新建一个 src.txt 文件，写一些内容进去，重新运行程序，结果如图 10.3 所示。

图 10.3　文件复制正确结果

fopen 函数打开一个文件，打开成功，返回文件指针，若失败则返回 NULL，因此需要对 fopen 的返回结果进行判断。同理也要对 fread 和 fwrite 的返回结果进行判断，看是否成功。对函数返回结果进行判断，可以大幅提高程序的健壮性，但是也存在很大的局限性，例如：

- 将用户函数和出错处理程序放在一起，出错程序的编写比较烦琐。也就是说在哪里发现错误，哪里就要处理错误。
- 使用返回值表现错误，可以传递的信息有限。
- 错误处理对别人的错误无能为力。比如调用了别人的函数，而该函数中对错误的处理不够完善。
- 调用者经常会忽略函数的返回值，也就是说对于函数的返回值，如果不加处理，编译器是不会提出警告的。

鉴于这种情况，C++中提供了更加完美的处理方法，也就是异常处理。后面将详细介绍异常的使用方法。

10.2　什么是异常

异常就是运行时出现的不正常，例如运行时耗尽了内存或者遇到意外的非法输入。异常存在于程序的正常功能之外，并要求程序立即处理。

10.2.1　什么时候出现异常

前面介绍了错误处理，错误都是可以预料的。但是当发生没有预料到的错误的时候，就会造成破坏。比如下面的代码：

```
double a = 0;
double b = 10;
double delta = b/a;              // 除 0 错误
```

在做除法之前，如果没有对 a 的值进行判断，就会出现类似上面的错误。还有程序员有时也

会犯一些错误，如下面对数组访问的代码：

```
int a[10];
for( int i = 0; i < =10; i ++ ) a[i] = i;
```

对于这样的不可预料的错误，都可能造成系统的崩溃。异常就是不确定的，会产生破坏的错误，上面的代码都会造成异常。上面的异常是由系统产生的，在 C++中也提供了产生异常的机制。在很多时候，代码有错误判断，但是不知道该如何处理。这个时候就可以人为地产生一个异常，把这个错误反馈到上层调用中。

10.2.2 异常与错误的区别

上一节简单地介绍了异常，使用错误和异常都是提高系统健壮性的方法，不过二者也有不同的地方。下面简单介绍一下错误和异常的区别。错误是可以预见到并且知道如何处理的情况，而异常是指不可预见的情况或者出错但不知道怎么处理的情况。因此，如果出错后知道怎么处理就用错误，不知道怎么处理就用异常。

要使用函数返回值来传递错误，必须逐层地传递。每一层函数调用都要处理，否则错误的传递就停止了。也就是说每一层函数都要有对错误的处理，这样的处理相当烦琐。而异常处理则没有这样的情况。使用异常，可以把出现错误的代码和处理错误的代码分离，使结构更加清晰。使用异常，有一个异常抛出的地方，有一个异常处理的地方，其他函数都不需要考虑这个异常。

使用异常，可以把内层错误直接转移到适当的外层来处理，简化了处理流程。且异常传递的信息可以非常丰富，不受函数返回类型的限制。

在前面介绍函数返回值的时候提到过返回值经常会被忽略掉，也就是说程序员处在一种函数都可以正确地结束的思路中。在这种情况下错误会被复制、放大，而异常则没有这种情况。一旦抛出异常，正常地执行代码就会终止，并且必须在函数的调用链上提供对这个异常的处理。因此对于这种不应该被忽略的错误，应该使用异常。

10.3 抛出异常

如果程序发生异常情况，而在当前环境中获取不到异常处理的足够信息，可以将异常抛出。抛出异常就是创建一个包含出错信息的对象，把该对象送到更大的环境中去。

10.3.1 主动抛出异常

当一段程序中发现错误数据，但是该程序不知道如何处理的时候，可以抛出异常。在 C++中，使用 throw 关键字来抛出异常，抛出异常的语法如下：

```
throw 错误信息;
```

在上面的语句中，throw 是关键字，表明要抛出一个异常。错误信息是一个变量，可以是任何类型，它存储了异常的必要信息。

异常的类型和函数需要的返回类型没有任何关系。

```
throw 1;
```

抛出一个整数的异常。而下面的代码：

```
throw "abcde";
```

则抛出一个字符串的异常。需要注意的是，throw 后面应该是一个变量，而不仅仅是一个类型。例如：

```
throw int;                    // 错误的写法
```

上面的写法是错误的，因为 int 是一个类型而不是一个值。如果只想抛出一个整型的异常而并不关心其值是多少，可以这么写：

```
throw int();
```

上面的语句会新创建一个 int 型变量，并且默认用 0 来初始化，然后抛出。一旦用 throw 抛出异常，程序的执行将会被终止，从本函数开始，查找可以处理该异常的地方。如果本函数中找不到匹配的处理，则退出本函数，到上一层查找，一直找到整个函数调用链的头部。如果找到 main 函数还没有找到合适的处理，则程序会失败。

下面是一个抛出异常的例子。在本例中，只简单地在函数中抛出异常，不进行异常处理，因此程序会中途退出。程序如示例代码 10.2 所示。

示例代码 10.2

```
#include <iostream>
using namespace std;
void g( int n )
{
    cout<<"进入函数 g"<<endl;
    if( n != 1 ) throw int();  // 若参数不为 1，则抛出异常
    cout<<"退出函数 g"<<endl;
}
void f( int n )
{
    cout<<"进入函数 f"<<endl;
    g( n );
    cout<<"退出函数 f"<<endl;
}
int main()
{
    f( 1 );
    cout<<endl;
    f( 2 );
    return 0;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 10.4 所示。

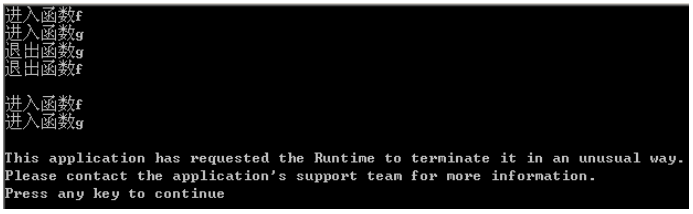


图 10.4 抛出异常结果

在主函数中调用 f，在 f 中调用 g。在第一次调用的时候，没有抛出异常，因此进入函数和退出函数的输出都可以执行；在第二次调用的时候，在 g 里抛出异常，退出函数，因此只有进入函数的输出，没有退出函数的输出。

10.3.2 自定义异常类

上面抛出的异常值，可以是 C++的内部数据类型，也可以是自定义的类型，也就是自定义类。任何自定义的类对象都可以作为异常值。在 C++的标准中，没有明确规定异常的类型。不过在 C++的标准库中定义了几个异常类型。在<stdexcept>头文件中定义了这些异常类，下面是其部分代码摘录：

```
namespace std
{
    class logic_error;          // : public exception
    class domain_error;         // : public logic_error
    class invalid_argument;     // : public logic_error
    class length_error;         // : public logic_error
    class out_of_range;         // : public logic_error
    class runtime_error;        // : public exception
    class range_error;          // : public runtime_error
    class overflow_error;       // : public runtime_error
    class underflow_error;      // : public runtime_error
}
```

在 C++的标准中，推荐在运行库时使用这几个异常类，或者从这几个异常类派生。也就是说 C++的标准库的实现中，可能会抛出上面列出的异常类。因此，自定义的异常类可以从上面列出的标准 C++异常类继承，也可以不从任何类继承。下面是几个自定义异常类的定义：

```
class myError                          // 不从任何异常类继承
{
public:
……
};
class myExcept : public exception      // 从标准异常类继承
{
};
```

上面定义了两个异常类，一个从标准异常类继承，而另一个则不从任何异常类继承。要抛出这样的异常类，跟前面介绍的抛出异常方法一样，例如：

```
throw myError();
throw myExcept;
```

10.4 捕获异常

如果一个函数抛出一个异常，它必须假定该异常能被捕获和处理。正如前面提到的，允许对一个问题集中在一处解决，这也正是C++语言异常处理的一个优点。本节介绍如何捕获异常。

10.4.1 try 块

如果在函数内抛出一个异常 ,系统将在异常抛出时退出函数。如果不想在异常抛出时退出函数，可在函数内创建一个特殊块用于捕获这个异常并处理错误。这个块就是 try 块。try 块由关键字 try 引导，其通用语法形式是：

```
try{
   // 可能会产生异常的代码
}
```

try 块可以嵌套出现。

可能产生异常的代码必须放在 try 块的里面 ,放在 try 块外面的代码产生的异常不能被该 try 块所捕获。

10.4.2 异常处理器

异常处理器由多个 catch 函数块组成，catch 函数块中的参数列表只能有一个参数，用于匹配由 throw 抛出的异常的类型。其语法格式为：

```
catch( 异常类型 异常值 )
{
    // 处理异常
}
```

catch 后面的定义和函数的定义类似，不过 catch 里面只能有一个参数。可以设置多个 catch 块，用于捕获各种不同类型的异常。catch 块紧挨着 try 块，其语法形式为：

```
try
{
    // 可能产生异常的代码
}
catch( type1  id1 )
{
    // 错误处理
}
catch( type2  id2 )
{
    // 错误处理
}
```

异常处理器都必须紧跟在 try 块之后，try 块后面至少有一个 catch 块。一旦某个异常被抛出，

异常处理机制将会按照我们书写的 catch 块的代码顺序依次寻找匹配的异常处理器。一旦找到一个相匹配的异常处理器，则像调用函数一样进入到该处理器里进行处理。

在查找匹配 catch 块期间，找到的 catch 块不必是与异常最匹配的那个 catch 块，相反，将选中第一个找到的可以处理该异常的 catch 块。因此，在 catch 子句列表中，最特殊的 catch 必须最先出现。catch 与 switch 不同，不需要在每个 case 块的后面加 break，以中断后面程序的执行。

在测试块中不同的函数调用可能会产生相同的异常情况，这时候只需要一个异常处理器。

下面是关于异常抛出和捕获的简单例子，在函数中抛出一个整型异常，在主函数中捕获这个异常。程序如示例代码 10.3 所示。

示例代码 10.3

```
#include <iostream>
using namespace std;
void func1( int n )
{
    if( n == 1 )
    {
        cout<<"func1 抛出异常"<<endl;
        throw 1;
    }
}
void func2( int n )
{
    if( n == 2 )
    {
        cout<<"func2 抛出异常"<<endl;
        throw 2;
    }
}
int main()
{
    for( int i = 1; i < 3; i ++ )
    {
        try{
            func1( i );
            func2( i );
        }
        catch( int )
        {
            cout<<"捕捉到 int 类型的异常"<<endl;
        }
    }
    return 0;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 10.5 所示。

图 10.5　捕获异常结果

函数 func1 和 func2 都可以抛出 int 类型的异常，因此只有一个 int 类型的异常处理器。整个 try 块在 for 循环之内，因此异常的抛出不影响 for 循环的运行。如果只关心异常的类型，则异常处理器中的标识符可以省略。

10.4.3　异常对象

在抛出异常的时候，被抛出的变量称为异常对象，本节介绍异常对象的生存周期。因为在处理异常的时候会释放局部存储，所以被抛出的对象就不能再局部存储，而是用 throw 表达式初始化一个特殊对象。异常对象由编译器管理，而且保证驻留在可能被激活的任意 catch 块都可以访问的空间。

异常对象被传给对应的 catch 块，传递的过程类似于函数参数的传递。如果使用值传递的方式，那么 catch 块中得到的将是异常对象的一个副本，要想在 catch 块中得到异常对象，那就需要用引用传递。因此上节的例子中的 catch 块也可以这么写：

```
catch (int & n)
{
}
```

在这种情况下，catch 块中得到的就是异常对象本身，而不是异常对象的副本。在处理完异常之后，异常对象会被销毁。

10.4.4　栈展开

抛出异常的时候，将暂停当前函数的执行，开始查找匹配的 catch 子句。首先检查 throw 本身是否在 try 块内部，如果是，检查与该 try 块相关的 catch 子句，看其中是否有一个与被抛出对象相匹配。如果找到匹配的 catch 子句，就处理异常；如果找不到，就退出当前函数（释放当前函数的内存并撤销局部对象），并且继续在调用函数中查找。

如果对抛出异常的函数调用是在 try 块中，则检查与该 try 块相关的 catch 子句。如果找到匹配的 catch 子句，就处理异常；如果找不到匹配的 catch 子句，调用函数也退出，并且继续在调用这个函数的函数中查找。

这个过程，称之为栈展开（stack unwinding），沿嵌套函数调用链继续向上，直至为异常找到一个 catch 子句。只要找到能够处理异常的 catch 子句，就进入该 catch 子句，并在该处理代码中继续执行。当 catch 结束的时候，在紧接着与该 try 块相关的最后一个 catch 子句之后的点继续执行。

下面是一个在函数调用中抛出异常的例子，不同类型的异常在不同的函数中被处理，因此每次抛出异常所导致的栈展开也不同。程序如示例代码 10.4 所示。

示例代码 10.4

```
#include <iostream>
using namespace std;
void f( int n )
{
    cout<<"进入函数 f，n = "<<n<<endl;
    try{
        if( n == 1 )
        {
            throw 1;            // 抛出 int 类型的异常
        }
        if( n == 2 )
        {
            throw 2.0;          // 抛出 double 类型的异常
        }

        if( n == 3 )
        {
            throw "Hello";      // 抛出字符串类型的异常
        }
    }
    catch( char * )
    {
        cout<<"捕获字符串类型的异常！"<<endl;
    }
    cout<<"退出函数 f，n = "<<n<<endl;
}
void g( int n )
{
    cout<<"进入函数 g，n = "<<n<<endl;
    try{
        f( n );
    }
    catch( double  )
    {
        cout<<"捕获 double 类型的异常！"<<endl;
    }
    cout<<"退出函数 g，n = "<<n<<endl;
}
int main()
{
    for( int i = 3; i > 0 ; i -- )
    {
        try{
            g( i );
        }
        catch( int  )
        {
            cout<<"捕获 int 类型的异常！"<<endl;
        }
        cout<<endl;
    }
    return 0;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 10.6 所示。

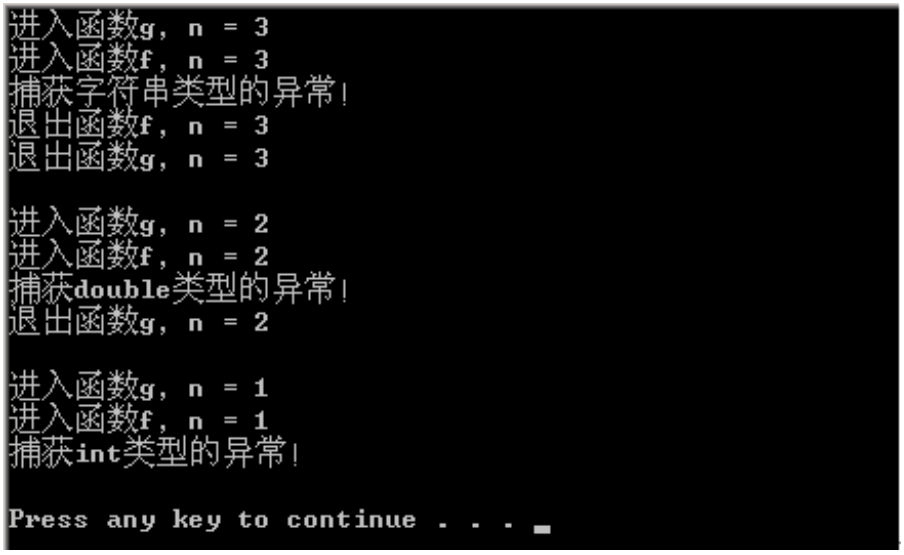


图 10.6　捕获异常结果

当函数参数为 3 时，抛出字符串类型的异常，这个异常可以被函数 f 的异常处理器所捕获，因此会继续执行 try 块后面的输出语句。当函数参数为 2 时，抛出 double 类型的异常，函数 f 中不能处理该异常，因此直接退出该函数，到函数 g 中继续查找，可以找到对应的异常处理器，因此 g 中 try 块后面的输出语句继续执行。当函数参数为 1 时，抛出 int 类型的异常，栈展开一直到 main 函数中。

10.4.5　重新抛出

在异常的处理过程中，有可能单个 catch 不能完全处理一个异常，catch 可能确定该异常必须由函数调用链中更上层的函数来处理。这个时候 catch 可以重新抛出（rethrow）这个异常，重新抛出可以将异常传递给函数调用链中更上层的函数。重新抛出是后面不跟类型或表达式的一个 throw：

```
throw;
```

空 throw 将重新抛出异常对象，它只能出现在 catch 或者从 catch 调用的函数中。重新抛出的过程，其实就是中断当前的异常处理器的处理，把异常对象沿函数调用链向上继续传递。重新抛出所抛出的异常对象，还是原来的那个异常对象，而不是形参的对象。比如，在一个函数中抛出了下面的异常：

```
throw 1;
```

在函数的调用链上有下面的异常处理：

```
catch( int n)
{
    n = 2;
    throw;          // 重新抛出
}
```

上面代码重新抛出异常时，异常对象的值还是原来的 1，而不是修改之后的 2。要修改形参的值，可以用引用的方法，例如：

```
catch( int  & n)
{
    n = 2;
    throw;          // 重新抛出
}
```

上面修改的才是原来的异常对象。

重新抛出抛出的还是原来的异常对象，不会新产生异常对象。要想在重新抛出中修改异常对象，需要使用引用传递。

10.4.6 捕获全部异常

所有的异常都必须被处理，否则就会使程序终止。因此当不能确定所有异常的类型的时候，可以捕获全部异常。捕获全部异常的代码如下：

```
catch(…)
{
    // 处理异常
}
```

在这里用“...”来表示所有的异常，可以跟所有的异常类型匹配。在这种情况下捕获的异常，不能知道异常的类型，也不能得到异常对象。前面提到过，在 catch 子句列表中，最特殊的 catch 必须最先出现。而 catch(...)可以捕获所有的异常，是应用范围最广的异常处理方法，因此它必须在列表的最后出现，否则后面的 catch 肯定不能匹配上。

10.5 函数与异常

当一个函数有可能会产生异常的时候，其调用者很希望知道该函数所有可能抛出的异常类型，否则就不知道如何编写程序来捕获所有可能的异常情况。C++语言提供了异常规格说明语法，可以清晰地告诉使用者函数抛出的异常的类型。

10.5.1 异常规格说明

C++语言提供了异常规格说明语法，可以清晰地告诉使用者函数抛出的异常的类型，这样使用者就可以方便地进行异常处理。异常规格说明存在于函数说明中，位于参数列表之后。异常规格说明再次使用了关键字 throw，其语法格式如下：

```
void f() throw ( 类型 1, 类型 2, 类型 3);
```

前面的函数定义可以是任何类型的函数定义，例如：

```
void f() throw( int, char);
```

上面的语句规定只能抛出 int 和 char 类型的异常，而不能抛出其他类型的异常。

在抛出异常的时候，throw 后面必须是异常值，而在进行异常规格说明的时候，throw 后面需要有括号，括号里面是异常的类型。

而传统函数声明：

```
void f();
```

则意味着可能抛出任何一种异常。因此，不提供任何异常规格说明，并不是不抛出异常，而是可以抛出任何类型的异常。要不抛出任何异常，则使用下面的定义：

```
void f() throw ();
```

也就是说其异常类型列表为空，在这种情况下则不会产生任何异常。当函数中试图抛出不在异常规格说明里面的异常的时候，编译器会调用 unexpected()函数。可以使用 set_unexpected()函数来把自定义的函数设置为 unexpected()函数。

visual studio 编译器不支持异常规格说明，因此下面代码应该在非 visual studio 的环境下运行才能得到预期的结果。

下面通过一个例子来展示怎么使用异常规格说明，程序如示例代码 10.5 所示。

示例代码 10.5

```
#include <iostream>
using namespace std;
class CMyExcept1{};
class CMyExcept2{};
void g()
{
    throw 1;
}
```

```
void f( int type ) throw( CMyExcept1, CMyExcept2 )
{
    if( type == 1 ) throw CMyExcept1();
    if( type == 2 ) throw CMyExcept2();
    g();
}
void myUnexcepted()
{
    cout<<"产生其它异常"<<endl;
}
int main()
{
    set_unexpected( myUnexcepted );
    for( int i = 1; i <=3; i ++ )
    {
        try{
            f(i);
        }
        catch( CMyExcept1 &  )
        {
            cout<<"捕获到异常 CExcept1"<<endl;
        }
        catch( CMyExcept2 & )
        {
            cout<<"捕获到异常 CExcept2"<<endl;
        }
    }
    return 0;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，其结果如图 10.7 所示。

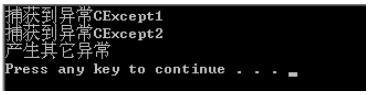


图 10.7　使用异常规格说明示例结果

10.5.2 异常安全的函数

当一个函数抛出异常的时候，会结束当前的执行，在调用链上查找可以处理该异常的 catch 块，这个过程可能导致某些数据的不完善。因此使用异常的一个原则就是：尽量设计异常安全的函数。异常安全的函数是指即使在这个函数的执行过程中出现异常，也不会影响重要的数据或状态。要设计异常安全的函数，要求在调用可能会产生异常的语句之前，函数必须提供下面几个保证：

- 函数提供基本保证，允诺如果一个异常被抛出，程序中剩下的每一个东西都处于合法的状态，没有对象或数据结构被破坏。
- 函数提供强力保证，允诺如果一个异常被抛出，程序的状态不会发生变化。调用这样的函数，如果它们成功了，它们就完全成功；如果它们失败了，程序的状态就跟它们从来没有被调用过一样。

其实，简单地说，设计异常安全函数的原则就是：尽量把可能抛出异常的代码写在函数的前面，让异常尽早抛出，即在改变重要数据或状态之前抛出异常。

10.6 使用异常的注意事项

在使用异常的时候，有以下几个需要注意的地方：

- 在 catch 块中，只能重新抛出原来的异常，而不能抛出新的异常。
- 在抛出异常之后，在该函数的调用链上，一直到可以正确地处理该异常为止，中间的函数的栈空间都会被释放。也就是说对于栈上的类对象，会调用其析构函数，因此在析构函数中不要抛出异常。
- 在定义 catch 块的时候，只要碰上可以处理该异常的 catch 语句，该异常的处理就结束了。因此对于捕捉全部异常 catch(...)这样的定义，应该放在 catch 块的末尾。
- 要设计异常安全的函数。

10.7 综合实例

本节中，将讲解一个综合实例。在本实例中，不同的函数抛出不同的异常，而主程序则分别处理这些异常。程序如示例代码 10.6 所示。

示例代码 10.6

```
#include <iostream>
using namespace std;
void fun1( bool flag )
{
    if( flag )
        throw 1;
}
void fun2( bool flag )
{
    if( flag )
        throw 'c';
}
void fun3( bool flag )
{
    if( flag )
        throw "abcde";
}
class CMyExcept{
public:
    void display()
    {
        cout<<"自定义异常类"<<endl;
    }
};
void fun4( bool flag )
{
    if( flag )
        throw CMyExcept();
}
void usage()
{
```

```
    const char * str="1 抛出整数类型异常\n"
        "2 抛出字符类型异常\n"
        "3 抛出字符串类型异常\n"
        "4 抛出自定义类型异常\n"
        "0 退出循环";
    cout<<str<<endl;
}
int main()
{
    usage();
    bool flag = false;
    while( !flag )
    {
        int select;
        cin>>select;
        try{
            if( 0 == select ) flag = true;
            fun1( select == 1 );
            fun2( select == 2 );
            fun3( select == 3 );
            fun4( select == 4 );
        }
        catch( int i )
        {
            cout<<"捕捉到整数类型异常，数据为：";
            cout<<i<<endl;
        }
        catch( char c )
        {
            cout<<"捕捉到字符类型异常，数据为：";
            cout<<c<<endl;
        }
        catch( char * str )
        {
            cout<<"捕捉到字符串类型异常，数据为：";
            cout<<str<<endl;
        }
        catch( CMyExcept & e )
        {
            cout<<"捕捉到自定义类型异常，数据为：";
            e.display();
        }
    }
    return 0;
}
```

建立一个控制台工程，并将上述代码复制到源代码中，编译并运行，其结果如图 10.8 所示。

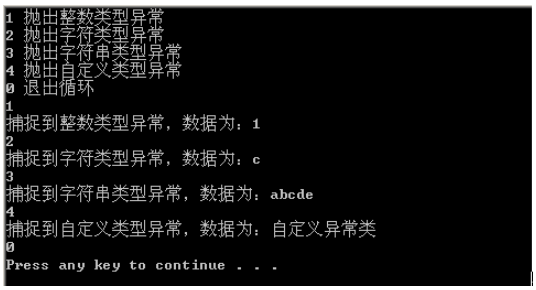


图 10.8　使用异常实例结果

10.8 小结

异常是 C++语言的一个重要特征。使用异常可以保证程序更加健壮。在 C++中使用 throw 关键字抛出一个异常，这个异常可以是任何数据类型。使用 try 和 catch 块来捕获异常。任何异常都需要被捕获，没有被捕获的异常会导致程序的终止。在 catch 块中可以重新把异常抛出。当函数中可能会抛出异常的时候，应当在函数的定义中设置异常规格说明。没有异常规格说明的函数可以抛出任何类型的异常。异常安全的函数是指即使在这个函数的执行过程中出现异常，也不会影响重要的数据或状态，要尽量设计异常安全的函数。

◆　　◆　　◆

第 11 章　宏与预编译

本章包括

- 预处理器和编译器
- 预处理器的任务
- 宏的作用
- 带参数的宏
- 宏指令和预定义的宏

程序在编译之前先要经过预编译阶段。预编译的任务是根据程序中的宏指令补充和完善源代码。有了宏指令，可以使得某些源代码编辑任务变得轻松，同时也可以控制哪些源代码需要编译，哪些不需要编译。

11.1 预处理器和编译器

预处理器是专门用来处理宏指令的程序。在编译器运行之前，会先运行预处理器，查找所有的预处理指令。预处理指令以“#”开头，而且不以“;”结束，以区别于一般的语句。预处理器根据预处理指令生成新的源代码文件（临时文件，可以通过编译器的选项输出到指定目录中）。

编译器的作用是把源代码转化成汇编语言或机器指令。但是，编译器并不是直接编译程序员写成的源文件，而是编译经预处理器处理后所产生的新的源文件。这些新的源文件经过编译器生成目标文件，再经过链接器链接生成最终的可执行程序，如图 11.1 所示。

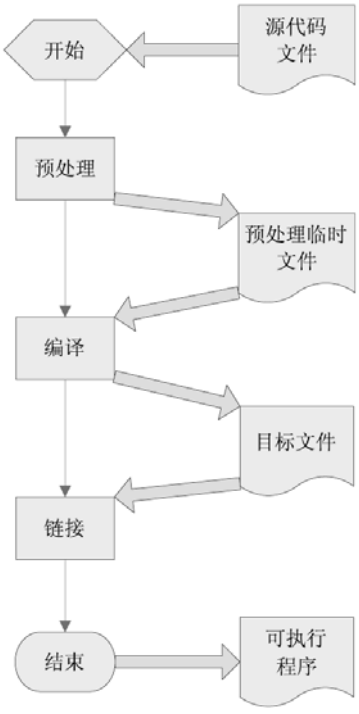


图 11.1　预处理、编译、链接过程

11.2 预处理器的任务

简单来讲，预处理器的任务就是执行源代码中的预处理指令，并对源代码进行相应的处理。因此，从预处理指令的类型来讲，预处理器的任务包括如下的几个部分：

- 将其他文件包含到当前文件中。
- 定义宏，用来取代某些复杂文本。
- 定义类似函数的宏，以更加灵活的方式控制源代码。
- 实施条件编译，即有选择地编译源代码的某些部分。

本节先介绍一下最常用的包含文件指令以及使用宏替换文本的指令。

11.2.1 包含文件

包含文件的预处理指令是“#include”，相信读者应该已经很熟悉了，其作用就是将别的文件包含到当前文件中。在实际使用中，一般被包含的文件是头文件。头文件中一般是函数、类等的声明，包含到当前文件中后，就可以在当前文件中引用头文件中的函数、类等。例如对于下面的源代码文件：

```
// 源代码文件 Some.cpp
#include <header.h>              // 使用预处理指令“#include”包含头文件“header.h”
void Function()                  // 本文件中定义的函数 Function
{
    SomeFunction();              // 调用头文件“header.h”中声明的函数 SomeFunction
}
```

假设头文件的内容如下：

```
// 头文件 header.h
void SomeFunction();
```

则经过预处理器处理后产生的临时源代码文件如下：

```
void SomeFunction();
void Function()
{
    SomeFunction();
}
```

按照 C/C++语言的语法要求，要使用某个标识符（变量、函数、类等），必须在使用之前先声明。虽然在 Some.cpp 文件中没有函数 SomeFunction 的声明，但是由于包含了头文件 header.cpp，所以仍然可以使用该函数。

11.2.2 搜索头文件

使用“#include”指令包含头文件时，其后的头文件有两种方式，一种是使用双引号，一种是使用尖括号。例如：

```
#include <stdheader.h>
#include "header.h"
```

如果文件名用尖括号<和>括起来，表明这个文件是一个工程或 C++标准库头文件。预编译器会首先搜索在工程中预定义的目录，然后搜索 C++编译器的安装目录。可以通过设置工程搜索路径环境变量或命令行选项来修改。例如，对于 Dev-C++，可以通过“工具”→“编译器选项”→“目录”来修改头文件的搜索路径，如图 11.2 所示。

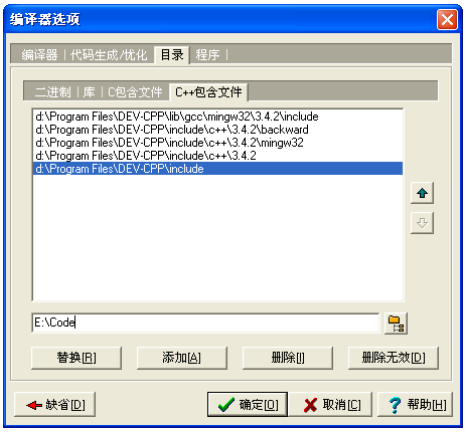


图 11.2　设置头文件搜索路径

如果文件名用一对引号括起来，则表明该文件是用户提供的头文件。预编译器首先从当前文件目录开始搜索，如果找不到，就从工程中定义的目录和编译器的安装目录查找。另外，也可以明确指定头文件的路径。例如包含 C 盘下的头文件 Header.h：

```
#include "C:\Header.h"
```

由于#include 指令不是 C++语句，所以在头文件的字符串中，不必使用双斜杠来间隔每一级路径。

11.2.3 展开宏

所谓宏，是程序中定义的用于替换复杂文本的简短文本。在程序的预编译期，预编译程序会解析源代码文本，执行一个替换源程序的动作，把宏引用的地方替换成定义处的文本。这个动作叫做宏的展开。在 C/C++语言中，用预处理指令#define 定义宏。下面是宏定义的一般格式：

```
#define 宏名 可替换文本
```

宏名，即宏的名称，在源代码中替换其后的文本。可替换文本，即宏名所指代的文本内容。在#define、宏名和可替换文本之间用空格（或制表符）分隔。预编译程序将#define 之后的第一个和第二个空格之间的文本作为宏名，其后所有的文本作为“可替换文本”，而不管中间有多少个空格。例如：

```
#define MACRO 1 + 2 + 3
```

其中只有 MACRO 是宏名，而文本“1 + 2 + 3”则是要替换的源代码文本。

由于预处理指令仅在预编译时处理，不参与编译，所以不受 C++语法限制。所以宏只要在使用前定义即可，而不必计较是否在某个语句块中。但从源代码清晰、可读的原则来讲，宏一般定义在文件的开头。

下面这段程序是宏的一个简单例子：

```
#define NUMBER 100
int main()
{
    cout << NUMBER << endl;         // 预编译时，NUMBER 被替换成 100
    return 0;
}
```

cout 后面的 NUMBER 在预编译时会被替换成 100。不论宏被定义成什么样的文本，都要根据其替换之后的结果进行判断。假设将上例的 NUMBER 宏定义如下：

```
#define NUMBER 1<<2
```

单从这个宏定义看，好像是将数字 1 左移 2 位，结果为整数 4。但是经预编译之后，第 5 行的代码被转换成如下的代码行：

```
cout<<1<<2<<endl;
```

所以当宏 NUMBER 被用在这个地方时，符号“<<”并不是左移，而是变成了输出对象 cout 的输出运算符。

在源代码文本中，并不是所有宏名出现的地方都会进行展开。如果这个宏名出现在一个双引号中，则该宏名就成了字符串的一部分，也就不会进行展开了。

11.3 宏的作用

使用预处理指令#define 定义的宏主要有三个方面的作用：

- 替代字面常量。
- 替代运算符。
- 声明某个符号已经被定义，通常用于条件编译。

下面将详细讲解这三个方面的作用。

11.3.1 替代字面常量

用宏替代字面常量是宏非常重要的一个功能，其好处是直观、简洁、修改方便。譬如对于圆周率π，其值是一个无理数常量 3.14159……。如果在程序中每一处要使用圆周率的地方都直接书写这个常量，那么源代码修改起来就不太方便。一旦要求改变数字的精度，减少或增加π的小数位数，则所有使用到的地方都要修改。无疑修改量会比较大，而且也不能保证每一处都修改对。

如果使用宏替换字面常量 3.14159，而程序中使用圆周率的地方都使用宏而不是圆周率本身，那么当修改圆周率的数值时，只要修改这个宏定义即可，大大减少了编程人员的工作量，提高了工

作效率。例如：

```
#define PI 3.1415926
int main()
{
    int r = 3;
    double area = PI * r * r;
    double len = 2 * PI * r;
}
```

当然，相对于常量，使用宏替换字面常量也有缺点。字面常量在编译时处理，有类型信息。而宏则是在预编译时展开，只是进行单纯的文本替换，没有类型信息。因此对于一些类型要求比较严格的地方，使用宏有一定的风险。针对这种情况，更好的方法是使用符号常量，即用 const 修饰的常量。例如，对于上述的圆周率的宏定义，可以改成：

```
const double PI 3.1415926
```

定义宏可以使用前面已经定义的宏。假设已经存在一个宏 NUMBER，其替换的文本是 100，当定义新的宏时，可以在宏的可替换文本中使用 NUMBER。例如：

```
#define 2N 2*NUMBER
#define N1 NUMBER+1
```

那么在预编译时，宏 2N 展开的结果就是“2*100”，N1 展开的结果就是“100+1”。

11.3.2 替代运算符

除了可以用宏替代字面常量，还可以用宏替代某些运算符，包括加、减、乘、除、逻辑与和逻辑非等，甚至函数和语句块的花括号。例如：

```
#define ADD     +
#define SUB     -
#define MUL     *
#define DIV     /
#define AND     &&
#define OR  ||
#define NOT     !
#define BEGIN {
#define END     }
#define OUT cout<<
```

利用上述宏定义，可以编写出貌似违反 C++语法、但实际上合法的源代码。例如：

```
void Function( )
BEGIN
    int x = 1 ADD 2;
    int y = 3 MUL 4;
    if(NOT x OR Y)
        BEGIN
        OUT Y;
    END
END
```

除了简单地替换某些操作符，还可以定义一些类似函数的宏，即可以接受参数的宏。其定义和使用都类似于函数，但又与函数不同，详细内容请参考后面的小节。

11.3.3 声明已定义符号

利用预处理指令#define 可以声明某个符号（宏名）已经定义过了，其格式如下：

```
#define SOME_NAME
```

如果在源代码中存在上述语句，则对预编译器来说，一个名称为 SOME_NAME 的宏已经被定义了，尽管没有指明要替换的文本。当然，也可以为该宏名指定要替换的文本，尽管没有必要。宏的这种特性通常应用在条件编译中。有关条件编译的内容将在后面介绍。

11.3.4 预定义的宏

在 C++标准中，规定了一些预定义的宏。也就是说这些宏不需要由开发者定义，而是由预编译器提供，开发者只要使用即可。如表 11.1 所示是一些常用的预定义宏。

表 11.1 常用的预定义宏

预定义宏	含义
__FILE__	代表当前源代码文件名的字符串文字
__LINE__	代表当前源代码中的行号的整数常量
__DATE__	进行预编译的日期 (“Mmm dd yyyy”形式的字符串文字)
__TIME__	源文件预编译时间，格式“hh：mm：ss”
__func__	当前所在函数名

上述预定义宏两侧都是两道下画线，而不是一道。

其中宏__LINE__表示该宏所在行的行号。这个行号可以用预处理指令#line 进行修改。预处理指令#line 的使用格式如下：

```
#line 新的起始行号
```

如果在源代码文件中存在上述预处理指令，则该指令的下一行将以新的起始行号为编号，以后各行以此为基准进行递增。譬如#line 指令定义了新的起始行号为 100，则在该指令的下一行使用宏__LINE__得到的值就是 100。#line 指令也可以修改宏__FILE__所表示的文件名，其格式如下：

```
#line 新的起始行号 "新文件名"
```

下例使用预定义的宏输出某些源代码的信息，程序如示例代码 11.1 所示。

示例代码 11.1

```
#include <cstdlib>
#include <iostream>
using namespace std;                          // 使用名称空间 std
void fun()
{
   cout<<"函数名：    "<<__func__<<endl;
}
int main(int argc, char *argv[])              // 主函数
{
```

```
    cout<<"——输出预编译信息——"<<endl;
    cout<<"文件名：     "<<__FILE__<<endl;
    cout<<"行号：       "<<__LINE__<<endl;
    cout<<"预编译日期： "<<__DATE__<<endl;
    cout<<"预编译时间： "<<__TIME__<<endl;

    fun();

    #line 100 "测试文件"
    cout<<"——修改行号和文件名——"<<endl;
    cout<<"文件名：     "<<__FILE__<<endl;
    cout<<"行号：       "<<__LINE__<<endl;

    system("PAUSE");                          // 暂停程序
    return EXIT_SUCCESS;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 11.3 所示。

图 11.3　输出预编译信息结果

11.4 带参数的宏

简单的宏定义，如上节例子所示，只能进行简单的文字替换，扩展能力有限。如果宏能够像函数那样带有参数，并根据参数的不同自动展开成不同的文本，则宏的扩展能力将大大提高。实际上，在 C++标准中，是可以利用预处理命令#define 定义带参数的宏的。

11.4.1 定义带参数的宏

定义带参数宏的语法同定义函数类似，在紧跟宏名之后有用括号包围起来的参数列表，但没有返回值类型、参数类型等的声明。其语法如下：

```
#define 宏名(参数 1, 参数 2, ……, 参数 n) 可替换文本
```

如果可替换文本一行写不完，可以分成多行，并在每一行的结尾处加上续行符号“\”(除了最后一行)。其语法如下：

```
#define 宏名(参数 1, 参数 2, ……, 参数 n) 可替换文本行 1 \
                                  可替换文本行 2 \
                                  ……
```

可替换文本行 m

要定义带参数的宏，则在宏名和左括号之间不能有空格（或制表符），否则就成了普通的宏定义，括号及其里面的内容也将会成为可替换文本的一部分。但是，在括号中的各参数之间可以任意添加空格。

在可替换文本中可以引用括号中的参数，从而根据参数的不同，展开成不同的源代码文本。例如定义一个比较两数大小的宏：

```
#define LESS(x, y)  x < y
```

假设在程序中有如下的引用：

```
bool result = LESS(3, 4);
```

则经过预编译后，上述语句将展开成如下的语句：

```
bool result = 3 < 4;
```

在上例中，LESS 是带参数宏的名称，x 和 y 则是该宏的形式参数（类似函数的形参）。在引用 LESS 宏时，可以用实参（类似函数的实参）3 和 4 分别替换形参 x 和 y。当进行宏展开时，可替换文本中用到形参的地方也将由相应实参一一替代。

下例定义一个宏，接受两个参数，然后比较这两个参数的大小，并输出其中的较小值，程序如示例代码 11.2 所示。

示例代码 11.2

```
#include <cstdlib>
#include <iostream>
using namespace std;                        // 使用名称空间 std
// 定义宏，输出两个数中的较小值
// 注意：以双斜杠引导的注释不能写到宏定义中，否则将成为宏的一部分
// 但是可以加/**/包围的注释，而且必须加在续行符之前
#define LESS(x, y) {  /*注释*/ \
    if( x < y ) \
    { \
        cout<<"较小值："<<x<<endl; \
    } \
    else \
    { \
        cout<<"较小值："<<y<<endl; \
    } \
}
int main(int argc, char *argv[])        // 主函数
{
    cout<<"——求较小值的宏——"<<endl;     // 输出提示信息

    LESS( 1, 2 )                        // 求 1 和 2 中的较小值，注意语句结尾没有分号
    LESS( 5, 4 )                        // 求 5 和 4 中的较小值，注意语句结尾没有分号
    cout<<endl;
```

```
    system("PAUSE");                    // 程序暂停执行
    return EXIT_SUCCESS;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 11.4 所示。

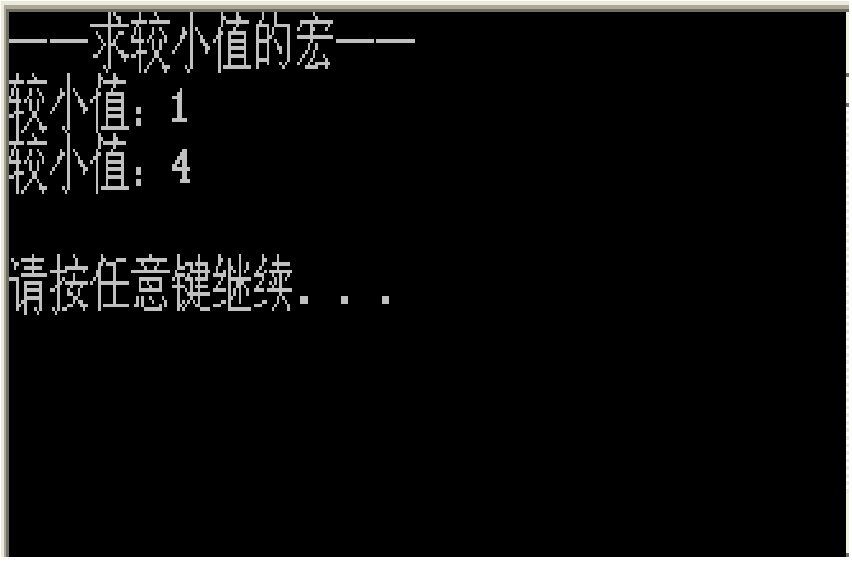


图 11.4　用宏求较小值并输出结果

在上述例子中，使用了带参数的宏 LESS，但是在语句的结尾处并没有分号。乍一看好像不符合 C++的语法规范，但是实际上并没有错。这是因为宏是在预编译时处理的，而不是编译期。而且，宏展开后的结果符合 C++语法规范，所以这些语句的结尾处没有分号并不会出错。

11.4.2 注意宏展开的结果

宏的行为有时候和所认知的行为会有些不同，主要原因是通常认为宏作为语句块，会优先执行，但这是不对的，宏并没有那么“聪明”，或者可以说预编译器并没有那么“聪明”，预编译器所做的只是忠实地将宏展开。来看下面的例子：

```
#define INT(x) x+x
int main()
{
    cout << 5 * INT(5) << endl;     // 输出 30
    return 0;
}
```

如果凭借第一印象，很可能会做这样的计算：5×（5 + 5）= 50。但是，预编译器并不这么认为，而会做这样的解释：5 × 5 + 5 = 30。忠实地将原来的宏展开。所以读者写宏的时候一定要小心，预防宏的行为不符合预期。如果想让结果输出 50 也很简单，只要给宏加上括号即可：

```
#define INT(x) (x + x)
```

再来看一个例子，以加深理解：

```
#define Multiplies(x, y) x*y
```

```
#define Multiplies1(x, y) (x)*(y)
int main()
{
    int iResult = Multiplies(4, 5 + 6);
    cout << iResult << endl;           // 4 * 5 + 6 = 26
    iResult = Multiplies1(4, 5 + 6);
    cout << iResult << endl;           // (4) * (5 + 6) = 44
    return 0;
}
```

11.4.3 带参数的宏与函数的异同

带参数的宏在定义和使用方面同函数非常相似，都可以接受参数，并且可以根据不同的参数产生不同的结果。但是这两者毕竟是不同的东西，存在很大的差异，例如：

- 函数在程序运行时依然存在，但宏只存在于预编译期，程序运行时并不存在。
- 函数所占内存空间只有一份，但如果宏被调用多次，则其语句也将重复多次。
- 函数调用时有时间和空间方面的额外开销，但宏没有。
- 函数接受参数的传递，但宏只是对参数的简单替换。
- 函数的参数有类型信息，但宏的参数没有类型信息。
- 函数可以调试，宏不能调试。

宏在源代码预编译时被展开，在程序运行时就只有展开之后的代码了。但是函数在程序运行时却实实在在地存在于内存中，占据一定的内存空间。

函数无论被调用多少次，其所占内存空间是一定的。但是宏不一样，宏在被引用处展开，引用了多少次，就展开多少次。所以在程序运行时，宏所对应的语句在内存中也将占有多份空间。函数在调用时需要保存当前 CPU 的现场，比如各个寄存器、指令计数器等的内容。然后从函数的起始地址开始调用函数中的各条指令。函数调用完毕后，又要恢复 CPU 现场。相比宏，这些都是函数调用的额外开销。对于宏，程序运行时只要依次执行其相应的语句即可。

函数的参数是通过栈（内存中的一段）进行传递的，调用时先把参数依次压入栈中，函数可以从栈中取出这些参数。但宏的参数在预编译时已经完成替换，程序运行时不需要传递。

函数的参数有类型信息，在调用时可以进行类型检查。如果类型不相符，或者不能通过合理的类型转换使类型相符，则函数调用就不能进行下去。但是宏就不同，宏在定义时其参数就没有类型信息，在使用宏时也不会进行类型检查。

11.4.4 特殊的宏符号

在宏的替换文本中，可以出现一些特殊的符号，起到特定的解释作用，请看下面的例子：

```
#define VARIABLE_A(x) iTemp##x      // 展开后相当于 iTempx
#define VARIABLE_B(x) #@x           // 展开后相当于'x', x 只允许是单独的字符
#define VARIABLE_C(x) #x            // 展开后相当于"x", x 允许是多个字符
int main()
{
    int VARIABLE_A(1) = 100;        // 起连接作用，相当于 iTemp1
    cout << VARIABLE_A(1) << endl;
    cout << VARIABLE_B(a) << endl;      // 字符化，相当于'a'
    cout << VARIABLE_C(1+a) << endl;    // 字符串化，相当于"1+a"
    return 0;
}
```

第 1 行的##符号是字符串连接符号，连接前后两部分。第 2 行的#@符号是字符化符号，后面的内容是一个单独的字符。第 3 行的#符号是字符串化符号，后面的内容是字符串。如果一个宏定义很长，超出一行，可以在每行的后面使用续行符'\'。

11.5 宏指令和预定义的宏

在本节中，将详细讲解 C++中的宏指令和预定义的宏内容，并讲解如何利用预定义的宏进行编译等内容。

11.5.1 宏指令

C/C++中的宏指令都是在 ANSI 标准中的，表 11.2 列出了常见的宏指令。

表 11.2 常见的宏指令

#define	#error	#include	#if
#else	#elif	#endif	#ifdef
#ifndef	#undef	#line	#pragma

#define 已经在前面介绍过了，这里就不再讨论。#error 可以强迫编译程序停止编译，用来在编译期检查环境是否符合要求或者与约束的条件发生了冲突。其使用格式是：

```
#error token-string
```

当程序在编译过程当中遇到这个关键字，就会停止编译，产生一个错误信息，并且输出后面的 token-string。例如：

```
#if !defined(__cplusplus)
#error C++ compiler required.
#endif
```

上面这段代码的意思是在预编译期检查当前的程序是否是 C++编程环境，如果不是，就定义

#error ,让编译器停止编译。#if 和#endif 会在后面介绍 ,这里只需要将#if 视为普通的 if 判断语句，将#endif 看成是 if 判断的结束。

#include 使编译程序将#include 所指向的源文件导入进当前的源文件，被包含的文件必须被尖括号或者引号包围起来。#if，#else，#elif，#endif，#ifdef 和#ifndef 属于条件编译命令，可以对程序的各个部分有选择地进行编译。#undef 命令用来取消前面定义过的宏名。例如：

```
#define NUMBER 10
int _tmain(int argc, _TCHAR* argv[])
{
#ifdef NUMBER
    cout << NUMBER << endl;                    // 输出 10
#endif
#undef NUMBER

#ifdef NUMBER
    cout << NUMBER << endl;
#else
    cout << "找不到 NUMBER 定义" << endl;         // 输出“找不到 NUMBER 定义”
#endif
    return 0;
}
```

在程序的第 1 行定义了 NUMBER 宏，在第 6 行输出 10。在第 8 行取消 NUBMER 宏的定义，所以后面会输出“找不到 NUMBER 定义”。

11.5.2 利用预定义的宏指令进行有条件编译

预定义宏#if，#else，#elif，#endif，#ifdef 和#ifndef 都是条件编译命令。所谓条件编译，就是可以将源文件中的代码分成几个部分 ,有选择地编译各个部分。对于前三个宏#if ,#else 和#elif，可以理解为 if，else 和 else if，#endif 表示这个条件编译选择的结束。看下面的程序：

```
#define NUMBER 5
int main(int argc, _TCHAR* argv[])
{
#if NUMBER == 10
    cout << "Number is equal to 10." << endl;
#elif NUMBER == 5
    cout << "Number is equal to 5." << endl;
#else
    cout << "Others." << endl;
#endif
    return 0;
}
```

程序的第 1 行定义 NUMBER 宏，令其等于 5。后面的代码判断是否等于相应的值，选择性地编译后面的语句，由于定义 NUMBER 等于 5，编译器输出“Number is equal to 5”。

另一种条件编译的方式就是使用#ifdef 和#ifndef。同样，这两个命令也用#endif 作为作用域的结束。#ifdef 判断后面的标识符是否被定义，通常都是指预定义的宏，#ifndef 就是#ifdef 的取反。看下面的程序：

```
#define DEBUG
int main(int argc, _TCHAR* argv[])
```

```
{
#ifdef DEBUG
    cout << "Debug version." << endl;
#endif
    return 0;
}
```

第 4 行判断是否定义 DEBUG 标识符（在第 1 行定义），然后编译后面的语句。

11.5.3 文件包含命令和包含警卫

文件包含命令，就是指宏指令#include，这个指令的作用就是包含当前文件所需类型的定义。通常后面有两种形式的包含：第一种是尖括号包含的文件，表示要到系统目录下去寻找；第二种是引号包含的文件，表示在当前工程目录中去寻找。当工程中文件重多时，很有可能出现一个头文件被多次包含的情况。但是，C++中同一个类型被声明两次是非法的，来看一个简单的例子。

CAnimal 类定义包含在文件 Animal.h 中：

```
#include "stdafx.h"
class CAnimal
{
public:
    CAnimal();
    virtual void Output();
};
```

CPig 类定义包含在头文件 pig.h 中：

```
#include "stdafx.h"
#include "Animal.h"
class CPig : public CAnimal
{
public:
    CPig();
    virtual void Output();
};
```

由于 CPig 是从类 CAnimal 上继承下来的，所以在 pig.h 中需要包含 Animal.h 的头文件。CHorse 类定义包含在头文件 horse.h 中：

```
#include "stdafx.h"
#include "Animal.h"
class CHorse : public CAnimal
{
public:
    CHorse();
    virtual void Output();
};
```

由于 CHorse 是从类 CAnimal 上继承下来的，所以在 horse.h 中需要包含 Animal.h 的头文件。

接下来看一下 main 函数：

```
#include "stdafx.h"
#include "pig.h"
#include "horse.h"
int main()
{
    CHorse horse;
    CPig pig;
    return 0;
```

```
}
```

在 main 函数中使用了 CHorse 和 CPig 两个类，所以需要包含这两个类的头文件，但是，这两个类都包含 CAnimal 的定义，编译器不知道该选择哪个，会报告重复定义的错误。这个时候就需要包含警卫。

所谓包含警卫就是用一组宏命令将头文件包起来，使其不会被重复包含，看 Animal.h 的例子：

```
#ifndef ANIMAL_H
#define ANIMAL_H
……
#endif
```

#ifndef 是定义在头文件所有内容之前的，#endif 是定义在所有内容之后的，用预编译命令#ifndef 和#endif 将整个 Animal.h 头文件中的内容包起来。这样便不会有编译错误了，下面来分析一下。

当 main 函数所在的文件第一次包含 pig.h 文件时，同时会导入 Animal.h 中的内容，这时预编译器分析当前文件没有定义 ANIMAL_H，就会在当前文件中定义。当再次包含 horse.h 文件时，导入 Animal.h，发现文件中已经定义了 ANIMAL_H，就会查找#else 或者#endif，这时会直接跳转到#endif，不会包含当前文件的任何内容。

同理，如果有人想继续从 Horse 上继承，例如 WhiteHorse 或 BlackHorse 之类的，在一个地方同时使用，这时就需要在 horse.h 中加上包含警卫。通常的习惯是在所有的头文件中都加入包含警卫。

关键字#pragma once 可以起到相同的作用（仍然有差别）。

11.6 综合实例

本节将讲解模拟标准库中的一个预定义的宏，它用来实现断言的检查，检查程序是否在执行当中按照预想的行为执行，当与预期不符时，输出出错的断言内容、所在的文件和行数，同时终止程序。程序如下：

```
#include "stdafx.h"
using namespace std;
#define DEBUG
#ifndef DEBUG
#include <assert.h>
#else
#define assert(test)\
if(!(test))\
{\
    cout << "Assertion Failed : " << #test << endl;\
    cout << "file " << __FILE__ << endl;\
    cout << "line " << __LINE__ << endl;\
    abort();\
}
#endif

int main(int argc, _TCHAR* argv[])
{
    char* pchArray = NULL;
    assert(pchArray == NULL);
    pchArray = new char[10];
    assert(pchArray != NULL);
    delete [] pchArray;
    assert(pchArray == NULL);
    return 0;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 11.5 所示。

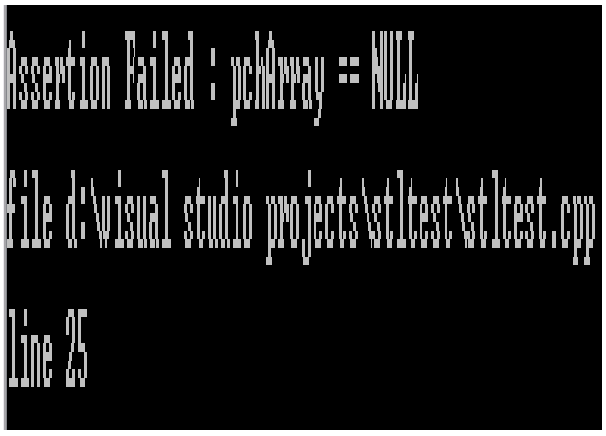


图 11.5 输出结果

11.7 小结

本章主要讨论了预编译器，在程序开始编译之前要先来解释程序，生成处理后的文本交给编译器。通过本章的学习，读者应该学会了宏的使用，了解了宏中的特殊操作符，如（#）字符串化、（@#）字符化和（\）续行符。同时应认识到了使用宏时的陷阱，即注意括号的使用。使用类似函数的宏的好处是可以提高程序运行的速度，弊端是不能完全模拟函数的调用。另外，还应该知道了常见的宏指令和预定义宏的用处以及包含警卫的使用方法。

◆ ◆ ◆

Part

第 2 部分　面向对象编程

第 12 章　面向对象基础

本章包括

- 对象和类的概念
- 对象的三个重要特征
- 面向过程和面向对象的区别
- 面向对象的分析
- 面向对象的设计

随着计算机硬件的迅猛发展，人们对软件的需求越来越多，对软件可重用性、可扩展性的要求也越来越高。而传统的基于数据和函数的开发方法，已经越来越难以达到上述要求。所以从 90 年代以来，面向对象已经逐渐成为软件业的主流开发思想。本章的重点是讲解面向对象的开发思想，并在最后给出一个实际的综合实例。

12.1 对象与类详解

本节将详细讲解 C++编程中的对象和类的概念。这些概念是后面章节内容的基础，读者应该深刻理解。

12.1.1 什么是对象

程序中的对象是对现实对象的抽象。现实中的对象包括可感知的物体以及思维中的概念。例如，天鹅、闹钟、飞机等各种物体可以看做一个个的对象，学校、公司、家庭等概念也可以看做一个个的对象。现实对象有一个特点，即它是具有属性和行为的整体。例如一个闹钟具有当前时间刻度、预定时间等“属性”，并具有指示时间、响铃等“行为”；而学校则具有学生、教师等“属性”，并具有招生、教育等“行为”。在程序中，属性可以抽象成数据，而行为可以抽象成函数，一个数据和函数的集合就构成了一个对象。

程序中的对象是一个整体，其属性和行为不可分离。要访问对象的属性（数据）和行为（函数）只能通过对象进行。例如要查看闹钟的“当前时间”或“预定时间”，只能通过闹钟这个对象进行，指示时间和响铃也只能通过闹钟进行。由此带来的一个好处是可以对相关的数据和函数按照需要分类管理，从而提高程序代码的可读性，以及软件的可维护性。

例如，在程序中表示一个闹钟对象，可以先定义一个表示时间的结构体 Time（其成员可以是三个整数，分别表示时、分、秒），然后用 Time 的两个变量分别表示“当前时间”和“预定时间”属性。对于“指示时间”和“到时响铃”两种行为，可以分别用两个函数表示，如图 12.1 所示，左图是现实中的闹钟对象，右图是程序中的闹钟对象。

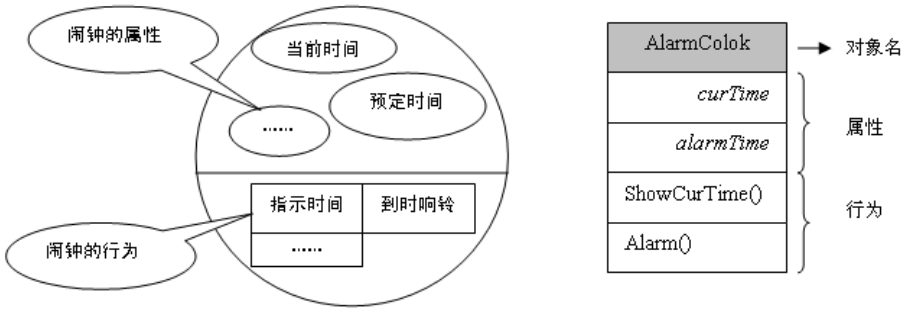


图 12.1 闹钟对象

12.1.2 什么是类

无论是在现实世界中，还是在程序中，对象都是一个完整的个体，而这些个体都可以按照某种规则进行分类。例如一个闹钟就属于闹钟类，一个三角形就属于三角形类。分类的规则是对象的属性相同（数量相同，类型也相同），行为也相同。

例如所有闹钟都具有形状、当前时间、预定时间等属性。这些属性的个数相同，不同闹钟的相同属性的类型也相同。另外，所有闹钟都具有指示当前时间、到时响铃等行为。因此，所有的闹钟都可以归为闹钟类，如图 12.2 所示。

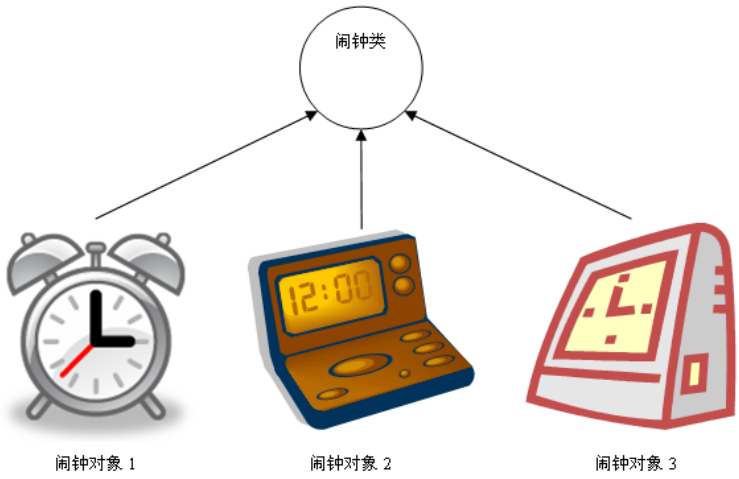


图 12.2 闹钟类与闹钟对象

同一类的不同对象是靠属性值区别的。例如在图 12.2 中，各个闹钟的“形状”属性互不相同，当前时间和预定时间属性值也都不尽相同，由此可以区分不同的闹钟对象。如果同一类的两个对象的属性值完全相同，则两个对象就是相同（相等）的。

为了在程序中表示一个对象，应当采用一种特殊的数据结构来集成数据和函数。在 C 语言中结构体虽然可以用来集成数据，但不能用来集成函数。所以 C++语言采用了特殊的数据结构类

（class）来集成数据和函数，而类的变量就是对象。同结构体一样，类也是可以自定义的，即开发者可以设置类中包含的数据以及函数。例如，一个闹钟类的定义如下：

```
class AlarmClock
{
public:
    void  showCurTime();                    // 显示当前时间
    void  setAlarmTime( Time alarmTime );   // 设定响铃时间
    void  alarm();                          // 响铃
private:
    int  m_shape;                           // 闹钟的形状编号
    Time  m_curTime;                        // 当前时间
    Time  m_alarmTime;                      // 响铃时间
};
```

类与对象的关系就是类型与变量的关系。在程序中要使用一个对象，就用这个类声明一个变量，这个变量就是所需的对象。例如声明并使用一个闹钟对象：

```
AlarmClock clock;                       // 创建一个闹钟对象
……
clock.showCurTime();                    // 显示闹钟的当前时间
……
clock.alarm();                          // 响铃
```

本节只给出了类的简单定义，有关类的详细说明，请参考下一章的内容。

12.2 对象的特征

对象有三个重要的特征：封装性、继承性和多态性。由于程序中的对象是对现实对象的模拟，所以在学习对象的时候，读者也可以将这两种对象对比起来理解。

12.2.1 封装性

对象从组成上来讲是数据和函数的集合。但是对于对象的使用者而言，并不能访问这个集合中的所有部分。使用者只能访问对象暴露出来的、可以被外部使用的部分，这些部分也可以看做对象的接口。而其他未暴露出来的部分则是对象的私有属性或者私有函数。私有属性和私有函数是为了实现对象的接口而存在的。

在前面的插图中，闹钟的时间、闹铃属性以及指示时间、到时响铃行为就是闹钟对象的对外接口，即外部使用者能够访问到的部分。而为了实现这些属性和行为，在对象的内部往往还有一些隐藏的部分，比如指针转动速度、预定响铃时间等私有属性，以及驱动指针转动、计算是否到预定时间等私有函数，如图 12.3 所示。

程序中的对象具有封装性，这一点同现实中的对象是一致的。人们在使用闹钟时，根本不需要知道其内部是怎么实现的，而设计闹钟的人也无意向用户暴露其中的细节。

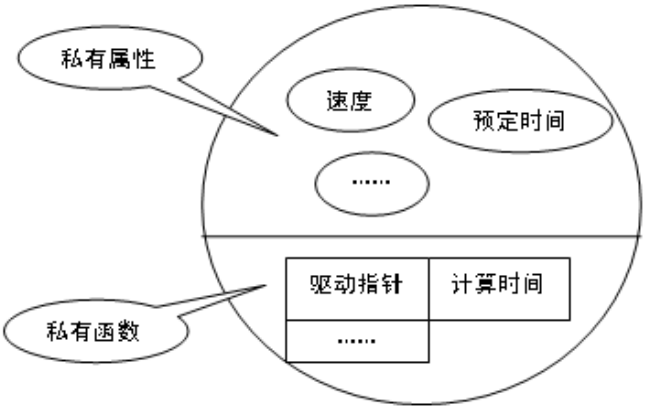


图 12.3 闹钟对象的私有属性和私有函数

12.2.2 继承性

继承是一种逻辑上的层次关系。在人类的思维中，有这样一种层次性的分类方法，即将一些事物按照一般到特殊的关系，先分成大类，再分成小类，各个小类也可以继续细分。大类与小类就构成了一种逻辑上的层次关系。大类（高层的类）称做基类（或者父类），小类（低层的类）称做派生类（或者子类）。基类与派生类的关系就是继承关系，派生类继承自基类。

例如，一个学校类，其下还应当有小学、中学、大学、职校等各种学校类，而大学类又可以分为本科学校类和专科学校类，其继承关系如图 12.4 所示。

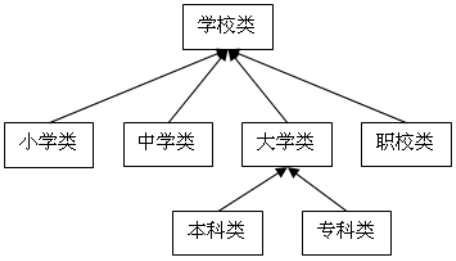


图 12.4 学校类的继承关系

学校类具有所有学校的共同属性，如校名、校址等，以及共同的行为，如招生、教育、考试等。但这些并不符合所有学校的情况，比如在招生的范围、教育的方式、考试的内容等方面都不相同。所以针对不同级别的学校，还应当设计不同的类。

继承还有另外一层重要的含义，即派生类中具有基类的所有组成部分。因为从基类到派生类是从一般到特殊，“特殊”只是比“一般”多了一些属性和行为，而不会缺少。从这个意义上讲，派生类“是”一个基类，例如大学是学校，中学是学校，小学也是学校。

通过继承，可以提高软件模块的重用性。一方面，派生类可以使用基类的方法和属性；另一方面，对基类的操作也可以施加到派生类上。

12.2.3 多态性

多态性指的是同一行为在基类和派生类间，以及各个派生类间的表现并不相同。这里所说的同一行为，指的是在基类里定义并由各个派生类继承的函数。由于基类在定义这个函数时考虑的是一般的情况，并没有针对各种特殊情形，所以派生类往往需要重新定义这个函数。当系统运行时，虽然使用的是同一个函数名，但具体调用的函数却是被派生类重新定义的版本。由于不同的派生类有不同的定义，从而一个行为就会表现出不同的形式，也就是多态。

例如对于各种学校类，作为学校的基本功能，定义基类“学校类”时就应当定义招生、教育、考试等各种行为。但对于各种具体的学校类，如小学、中学、大学等，这些行为又是有差别的，必须根据各自的具体情况进行重新定义。当说一个学校招生时，不同的学校有不同的招生对象、方法等，所以招生这个行为就在不同种类的学校间表现出了多态性。

12.3 面向过程与面向对象详解

面向对象其实不是什么新的概念。早在上个世纪 60 年代，面向对象的开发思想就已经被提了出来。但在那个年代，面向过程似乎更加符合当时人们对计算机科学的认知。虽然面向对象的开发语言也有一些，例如 Simula67，但毕竟没有面向过程的语言那么繁荣。缺少了语言层面的支持，面向对象的开发思想也就没有在软件业内得到大规模推行。这种状况一直持续到上世纪 90 年代，由于 C++语言的出现，才使得面向对象的开发思想得到重视，并日益在软件开发中担当起重任。为了使读者充分理解面向过程编程与面向对象编程的区别，本节将简要介绍一下二者。

12.3.1 面向过程

在面向过程的开发中，开发者关注的是解决问题过程中每一个步骤的实现以及步骤的次序。只有实现好每个步骤，并且按照正确的次序进行调用，问题才会得到解决。这就是所谓的过程。如图 12.5 所示就是一个典型的面向过程的程序结构。

图 12.5 面向过程的程序结构

在 C 和 C++语言中，一个步骤就是一个函数或者一个语句块（复合语句）。

例如，为了求解一个三角形的面积，开发者必须知道这个三角形的各种尺寸。而这些尺寸却是

多种多样的，尺寸种类的不同导致了求面积的方法不同。如果这些尺寸是三条边的长度，则根据对应的公式求面积。

```
double a, b, c;                                          // 代表三角形三条边长的变量
……                                                       // 获取边长的值
double p = ( a + b + c ) / 2;                            // 求面积的辅助变量
double s = sqrt( p * ( p - a ) * ( p - b ) * ( p - c ) ); // 求三角型面积
```

如果提供的尺寸包括一条边以及三角形在这条边上的高，则用边长乘高度再除以 2 求面积。

```
double a, h;                                  // 代表三角形边和高的变量
……                                            // 获取边长和高的值
double s = a * h / 2;                         // 求三角形的面积
```

如果提供的尺寸是两条边以及这两条边所夹的角度，则用两条边长的乘积乘以角度的正弦值再除以 2 求面积。

```
double a, b, alpha;                           // 代表三角形边长和角度的变量
……                                            // 获取边长和角度的值
double s = a * b * sin( alpha );              // 求三角形面积
```

很显然，为了求得三角形的面积，开发者必须了解已知数据的种类。这样就会使得程序依赖于底层的实现细节（三角形的数据种类就是这种细节）。一旦底层的细节发生改变，则上层的程序就要随之改变。在一个设计不合理的程序中，这种细节的改变甚至会导致整个软件项目的失败。因此，采用面向过程方法开发的程序，难以控制其维护成本。

面向过程还有一个难以克服的困难——缺少封装性。这表现在两个方面，一个是无法封装数据，另外一个是无法封装函数。

当一组数据需要几个函数联合起来进行处理的时候，要么采用全局变量，要么将数据作为参数。全局变量的缺点是显而易见的，即无法控制访问数据的行为，任何函数都有可能修改全局变量，甚至不相关的函数。如果将数据当做函数参数，则会导致函数的参数列表过于复杂，同时也会因参数传递导致效率方面的损失。

一般来讲，如果函数参数过多，则应当使用结构体来封装相关的参数，但这样会增加一些不必要的数据类型。

在面向过程的语言中，函数的使用是没有限制的。而在实际开发中，一个函数的目标数据是固定的。如果不限制函数的作用范围，函数与数据之间就不会有清晰的逻辑关系，从而导致程序的可读性降低。

如果能够将函数与目标数据绑定在一起，并限制外界的访问，则可以在一定程度上克服面向过程的缺点。一方面，在形式上容易表达数据与函数间的逻辑关系，另一方面，从语言层面上减少了非法访问的机会。

12.3.2 面向对象

在面向对象的开发中，解决问题的各个步骤已经不再是开发的核心。如何实现对象以及如何使用对象才是最重要的。实现对象首先要确定程序中需要哪些对象以及这些对象的属性和行为。确定后，开发者就要在程序中实现这个对象，即定义对象所属的类。完成类的定义之后，就可以在程序中用类创建对象，并通过对象间的协作来解决问题了。如图 12.6 所示就是一个典型的面向对象的程序结构。

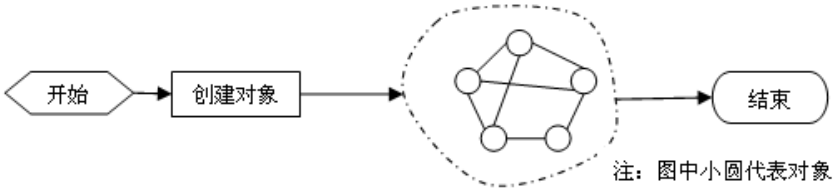


图 12.6　面向对象的程序结构

例如，为了求解三角形的面积，应当先建立一个三角形对象。该对象存储了各种属性数据，并实现了求面积的接口方法，求三角形面积时，只要调用这个三角形对象的求面积接口即可。

```
Triangle tri;                 // Triangle 是三角形类，tri 是一个三角形对象
……                            // 设置三角形的属性
double s = tri.getArea();     // 调用三角形对象的 getArea()方法求面积
```

在上面的例子中，三角形对象求面积的接口方法很简单，是一个没有任何参数的函数。使用者并不能从该接口获知任何实现的细节。这些细节都封装在对象的内部，外界无法得知。所以，这是一个与实现无关的接口。在程序开发以及以后的维护中，只要这个接口不变，无论其内部实现如何改变，都不会影响使用者的调用，在很大程度上降低了软件的维护成本。

12.4 面向对象的分析（OOA）

在开始系统设计之前，软件开发应当经历系统分析阶段。其目的是在需求分析的基础上，描述系统的整体功能，并建立问题领域的模型。在面向对象的软件开发中，系统分析的方法是面向对象的分析方法，简称 OOA（Object Oriented Analysis）。在 OOA 的建模过程中，采用了很多面向对象的思想。

在传统的面向过程的软件开发中，系统分析的方法是结构化的分析方法。其特征是对系统的功能进行模块化、层次化的划分，并用数据在一系列模块间的流动表示系统行为。

12.4.1 对问题领域进行建模

所谓问题领域就是软件系统所要处理的领域。在 OOA 过程中，对问题领域进行建模就是从对系统的分析中识别出所需的对象，并用对象和对象之间的关系来描述整个软件系统。

例如对于简单的求三角形面积的问题。经过简单的分析，可以很容易地得出结论，即求解这个问题只需一个三角形对象即可。如果是复杂的系统，则需要更多的类，类之间的协作关系也变得更加复杂。例如在一个窗口中绘制复杂图形，而图形可能是由多个基本图形（三角形、矩形、圆形等）构成的，在这样的系统中需要画布类、图形容器类、图形基类、各个图形类等，类对象之间的协作关系如图 12.7 所示。

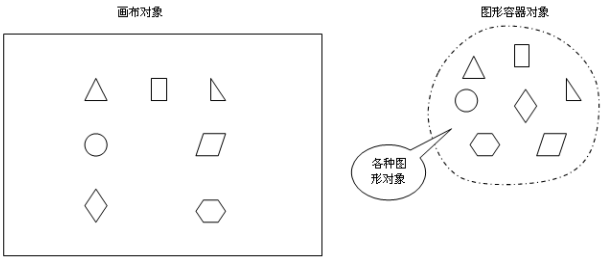


图 12.7　类之间的协作关系

12.4.2 OOA 的基本原则

在面向对象的分析过程中，最基本的原则就是从对象的角度出发，思考和分析整个软件系统，即程序就是由对象和对象间的相互作用构成的。为了方便程序开发实践中的操作，在这个原则的基础上又衍生出了一系列的原则，包括：

- 抽象——抽取数据和过程特征部分。
- 封装——将数据和处理函数组织在一起。
- 分类——描述对象的类型。
- 继承——描述对象的概念层次结构。
- 聚合——用组装的方法建立对象。
- 关联——建立对象间的联系。
- 消息通信——对象间协作的方法。
- 粒度控制——简化信息。
- 行为分析——分析对象的行为。

抽象、封装、分类和继承在前面的章节中都有讲到，这里不再赘述。

聚合原则用在描述对象的组成上。一个复杂的对象往往是多种简单对象的复合体。因此，在程序中应当将简单对象作为复杂对象的数据成员（属性），从而以自然的方式描述一个复杂事物。例如，一所学校就是由校长、教师、学生等组成的复杂对象，如图 12.8 所示。

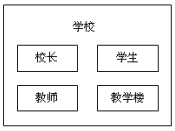


图 12.8　学校对象的组成

聚合和继承描述了对象（类）间的不同关系。继承描述的是 is-a 关系，即派生类的对象是一个基类对象，例如小学是学校，中学也是学校。聚合描述的是 has-a 关系，即一个对象中拥有另外一个对象，例如学校中有校长、教师、学生等。

关联表示的是对象之间的联系。与继承、聚合不同，关联关系比较松散，一个对象存在与否并不影响另外一个对象。例如，教师对象与学生对象之间的关系就是关联关系，无论有没有学生，都不会影响教师对象的存在，反之亦然；学生对象与课程对象的关系也是这样，如图 12.9 所示。

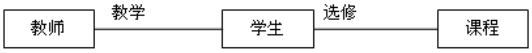


图 12.9　对象间的关联关系

消息通信原则指的是对象之间只能通过消息进行通信，而不允许直接存取对象的属性。消息通信原则是由封装引起的。封装的对象只能通过公开的接口与外界通信，而这个接口，即接口的调用行为就是所谓的消息。在 OOA 中要求用消息连接表示出对象之间的动态联系。例如银行的客户同账户之间的消息通信包括存钱、取钱、改密码等，账户同银行的消息通信有冻结、解冻、统计等，如图 12.10 所示。

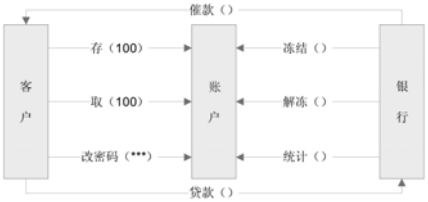


图 12.10　消息通信

粒度控制指的是在 OOA 中，应当将分析的主题限制在适当的范围内，不要过度深入细节。OOA 的目标是要得到一个系统整体性的描述，如果过度深入细节，会使分析陷入复杂化，反而达不到预期的效果。例如，一个对象在 OOA 中只要给出该对象的对外接口即可，至于内部怎么实现则不考虑。

细节方面的问题是在设计和编程阶段解决的。

行为分析原则是 OOA 的指导性原则，其含义是要求开发者从分析问题领域中各种事物的行为入手。事物的行为是其内部属性的反映，也直接代表其对外接口。所以对事物行为的分析可以得出一个对象的定义。另外，事物之间的行为存在依赖和顺序关系，而这些关系也反映了对象之间的关系。因此，经过行为分析就可以从静态和动态两个方面对系统进行建模。

12.4.3　OOA 的基本步骤

面向对象分析的基本步骤如下：

step 1 确定系统功能。在分析开始前，首先要确定软件系统要解决哪些问题，系统要提供什么功能。

step 2 找出对象并分类。通过对系统中各种事物的抽象，得出所需要的对象，并对这些对象进行分类，建立“类”类型。

step 3 确定类之间的关系。这些关系包括继承关系、聚合关系、关联关系等。

step 4 确定对象的属性。这些属性是对象的标识性、关键性的属性，并可以被外界访问。

step 5 确定对象的函数。这些函数表示的是对象的行为。

上述过程如图 12.11 所示。

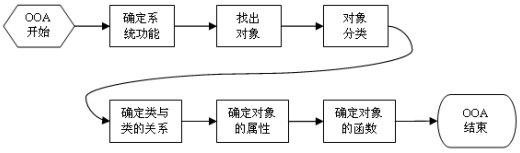


图 12.11 OOA 过程

在面向对象的分析以及后面的面向对象的设计中，通常会采用统一建模语言作为描述工具。统一建模语言简称 UML，是一种通用的、图形化的建模方法。可以用来描述系统的静态和动态模型，包括类图、对象图、时序图、协作图、用例图等。有关 UML 语言的详细说明请参看相关书籍，本书中也大量应用了 UML 语言来对软件系统进行分析和设计。

12.5 面向对象的设计（OOD）

在对软件系统进行分析之后，开发进入设计阶段。设计的目的是对分析的结果进一步规范和整理，以便进行后面的代码设计。在面向对象的开发中，设计的方法是面向对象的设计（Object Oriented Design），简称 OOD。在 OOD 过程中同样也采取了很多面向对象的思想。

12.5.1 OOD 要解决的问题

OOA 过程的结果是一个概念性的结果，并不能直接用于 OOP，即面向对象的编程。在开始编写代码之前，需要对 OOA 所抽象出来的对象、类以及其他相关文档进行整理和求精，使之更符合 OOP 的需要。这个过程就是所谓的 OOD。

OOD 的目标是将对象的属性和方法进一步精化，改正错误的内容、删去不必要和重复的内容等。有时根据需要，还应当添加一些辅助的属性和方法，这些方法仅用于辅助对象接口的实现，原则上应当不对外公开，而是作为对象的私有成员。

12.5.2 OOD 的一些基本原则

同 OOA 一样，OOD 也有一些基本原则需要遵从，包括：

- 单一职责原则。
- 开放封闭原则。
- 替换原则。
- 依赖倒置原则。
- 接口隔离原则（ISP）。

对于上述的 OOD 原则，如果想深入理解，则需要较多语言方面的知识，特别是有关类的一些知识。而这些知识安排在后面的章节中，所以在讲述这些原则时，本节只做简要的解释。

单一职责指的是一个类应当只负责一方面的功能。单一职责的目标是简化类的定义，提高程序的可维护性。如果一个类牵涉多方面的功能，则导致修改类的原因就会很多。而一旦类被修改，其他依赖于该类的代码就需要修改，从而带来很多潜在的问题；即便不需要修改，其他依赖该类的源文件也要重新编译，影响开发效率。

开放封闭原则指的是软件系统应当对扩展开放，对修改封闭。该原则主要应用在系统需要增加功能的情况下。增加功能最好的方法就是增加新的类，而不是去修改已有的代码。后者往往会对已有的系统造成较大的冲击，从而降低软件的可维护性。

替换原则指的是在程序中所有用到基类对象的地方，都可以用派生类对象代替。从逻辑上来讲，派生类对象和基类的关系是 is-a 的关系，即派生类对象也是一个基类的对象。既然如此，派生类的对象当然可以替代基类的对象。例如，三角形类派生自图形类，在程序中有设置图形对象原点的代码，显然，这个操作也可以应用于三角形对象。

依赖倒置原则指的是程序应当依赖于高层的抽象，而不是底层的实现细节。在面向对象的开发中，类（对象）以及类（对象）间的协作关系是对现实的抽象，而基类更是对派生类的抽象。抽象的东西一般很少发生改变，因此依赖于抽象的程序其维护的成本就比较低。

接口隔离原则指的是在设计类时使用多个专门的接口，而不是一个大而全的接口。这里的接口指的是一个特殊的基类，其中仅有成员函数。应用接口分离原则，一个接口发生了改变，不至于影响到其他接口，这样依赖于其他接口的程序也不用改变，从而提高了程序的可维护性。

在面向对象的设计过程中，有一些模式可以遵循。《设计模式：可复用面向对象软件的基础》（作者：（美）Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides，机械工业出版社，ISBN：7111075757）一书针对设计过程中的普遍问题，提出了很多有效的设计模式，包括工厂模式、单件模式、策略模式等，开发者在设计过程中，应当尽量采用。

12.5.3 OOD 的基本步骤

面向对象设计的基本步骤如下：

step 1 细化类。从实现的角度出发，按照功能细化类，并增加必要的辅助类。

step 2 明确类之间的关系。用继承、聚合、关联来描述类之间的关系。

step 3 定义属性。确定属性的类型，并在必要时增加或删减属性。

step 4 描述对象之间的协作关系。描述系统动态运行的情况。

step 5 利用设计模式优化设计。

上述步骤如图 12.12 所示。

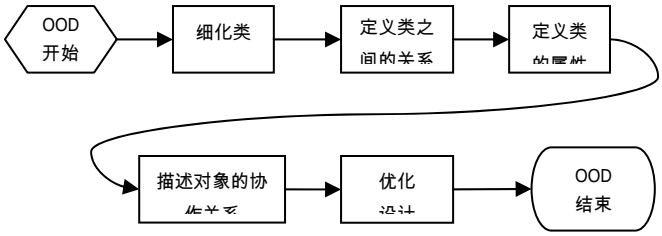


图 12.12 OOD 过程

12.6 综合实例

本节的综合实例将分析和设计一个自动售货机的控制系统，其中将用到一些 UML 图。一般来讲，UML 图比较通俗易懂，对于不太明显的图，书中也会有适当的注释，如果读者想深入了解 UML，请参考相应的书籍。

所谓用例就是系统的功能。从用户的角度出发，将系统看做一个整体，该整体所体现出来的功能就是系统的用例。一个自动售货机的用例如图 12.13 所示。

下面各图中的说明使用中文，在实际的分析和设计中，应当使用英文，以便与程序中的各种标识符相对应。

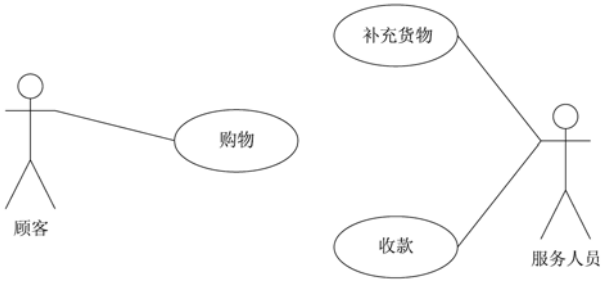


图 12.13 自动售货机的用例图

从用例图出发，可以分析出系统应当具有的各种类及其之间的关系。自动售货机的类图如图 12.14 所示。

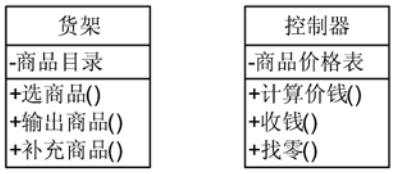


图 12.14　自动售货机的类图

类图描述的是系统的静态属性，而时序图描述的则是系统的动态行为。自动售货机的时序图如图 12.15 所示。

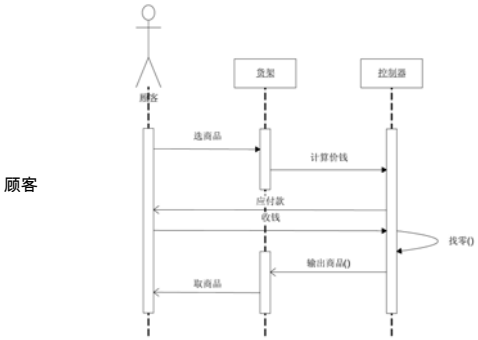


图 12.15　自动售货机的时序图

12.7 小结

本章的重点是讲述现代软件开发思想，即面向对象。面向对象包括两部分内容，即面向对象的分析和面向对象的设计。当然，这两部分只是软件开发的前期工作，要完成一个真正能够运行的软件，还应当运用面向对象的思想，借助面向对象的语言进行编程。从下一章开始，将讲述如何运用C++语言进行面向对象的编程。

◆　　◆　　◆

第13章 类的封装

本章包括

- 定义类
- 类对象
- 类成员的访问限制
- 类的构造函数和析构函数
- 类的 static 成员

面向对象编程的一个关键原则就是封装，把暴露的数据封装起来，尽可能地让对象管理自己的状态，这个封装起来的结构就是类。类是 C++中最基本的特征，C++语言的早期版本被命名为“带类的 C”，以强调类机制的中心作用。在 C++中，用类来定义自己的抽象数据类型。类是 C++语言中面向对象编程的基石，面向对象编程中的封装、继承、多态等特性，都需要通过类来实现。本章主要讲述类的定义、类的成员组成，以及简单的使用方法。

13.1 定义类

封装和抽象是面向对象的重要特征。封装是人们在现实世界中解决问题时，为了简化问题，对研究对象所采用的一种方法，一种信息屏蔽技术。例如，在看电视的时候，只需要简单地按几个按钮就可以了，没有必要去了解电视内部的结构、信号的接收。这样一来通过对实现细节的封装，就使得本身很复杂的问题变得非常简单、容易操作。封装的目的就是达到接口和实现的分离。

抽象是对象的最简化的接口，向客户提供了所期望的服务。理解抽象的关键在于接口，抽象就是一个明确的接口。对于电视，遥控器就是接口。

用好抽象的关键在于对研究问题的深刻理解。设计得比较好的抽象可以把使用从实现中分离出来，通过提供必备的信息，隐藏起实现细节，让客户以较为安全和可预测的方式使用对象。

在 C++编程中，用类来表示对数据的封装和抽象。类不但提供了对数据的封装，还提供了对数据的处理函数。因此类就是一个包含有若干数据和函数的集合，此数据和函数被称为类的成员。在 C++中类表示一个自定义的数据类型，本节简单地介绍类的定义。

13.1.1 声明一个类

类是对现实事务的抽象，要声明一个类，使用关键字 class，其定义格式为：

```
class 类名字 // 类的定义
{
    // 类成员
};
```

在 C++中也可以用 struct 来定义类，这两种定义方式的默认访问限制是不同的。为了明确起见，本章中始终用 class 来定义类。

上面的格式是C++中最简单的类的定义，其中class是关键字，表示定义的是一个类；类名字表示该类的名字，这个名字需要符合C++中的命名规则；花括号“{”和“}”是该类的定义域，所有类的内部成员（包括成员数据和成员函数）都要在其中定义，类一旦定义完成之后，没有任何方法可以增加类成员；定义结束处的分号是必需的，这点和函数的定义不同。下面的代码定义了一个Employee类：

```
class Employee      // 定义一个 Employee 类
{

};
```

上面简单地定义了一个类，这个类用来表示一个雇员的信息和行为，在后面的章节中会陆续地增加这个类的定义，最后可以完整地记录一个人的信息。

13.1.2 类的数据成员

类是对事物的封装，因此类要封装必要的描述事物的数据。这些数据被包含在类中，称为类的数据成员或者成员变量，类的数据成员的定义和普通的变量定义一样。看下面增加了数据成员的Employee类的定义：

```
class Employee          // 定义 Employee 类
{
public:                 // 定义变量的访问限制
    char name[32];      // 记录名字
    int grade;          // 记录等级
    double salary;      // 记录薪水
};
```

上面的定义中为Employee类增加了三个数据成员，分别用来表示Employee的名字、职位等级和薪水。这些数据成员被组织起来，用来表示同一个雇员的信息。

类的数据成员，可以是任何的数据类型。上面的例子中定义了int类型的数据成员和char数组的数据成员，其他类型的数据成员的定义方法和上面的例子相同。所以可以定义变量的数据类型都可以定义类的数据成员。类的数据成员的定义顺序并没有特别的要求，这些数据成员可以以任何顺序出现，因此交换上面的数据成员定义的顺序对该类的使用并没有什么影响。

数据成员的定义顺序不同，可能会影响该类的size，这是因为C++的编译器中有一个对齐的概念，这里不考虑对齐。

在定义普通变量的时候，可以在定义时直接赋一个初始值，例如：

```
int a = 0;
```

但是在定义类的时候，是不可以给类的数据成员赋初始值的。这是因为类的定义仅仅是定义一个自定义类型，并没有声明对应的对象。因此下面的定义是错误的：

```
class Employee                      // 定义 Employee 类
{
```

```
public:                              // 定义变量的访问限制
    char name[32] = "Jason";         // 记录名字，定义错误
    int grade = 8;                   // 记录等级，定义错误
    double salary;                   // 记录薪水
};
```

上面的定义是错误的，切记不可以在定义类的时候给数据成员赋初始值。要想给类的数据成员赋初始值，只能在构造函数中或者构造函数的初始化表中进行。

13.1.3 类的成员函数

在C++中，类不但封装了数据，还抽象出了对数据的操作，通过这些操作，可以方便地修改类的数据成员。这些操作可以完成类的数据成员之间的协调，可以保证数据的一致性。这些抽象出来的操作就是函数，还可以把函数封装到类中，成为类的成员函数，这些函数主要处理该类的数据成员，成为该类的专用函数。

在前面定义的类 Employee 中，定义了数据成员 grade 和 salary，表示雇员的等级和薪水，这两个数据跟该职员的绩效考评有关，并且不能随便改动。在这里可以定义一个绩效考评的函数 performance，该函数可以根据该雇员的表现修改 grade 以及 salary。因为 performance 函数主要用于改变 Employee 类的 grade 和 salary，所以 performance 函数可以作为该类的成员函数。下面是添加了成员函数的 Employee 类的定义：

```
class Employee                   // 定义 Employee 类
{
public:                          // 定义变量的访问限制
    void performance()
    {
        // 根据年度工作量评定新的 grade
        grade = grade + 1;       // 为了简化程序，直接提高等级
        // 根据表现修改 salary
        salary = salary + 500;   // 简化程序
    }
    char name[32];               // 记录名字
    int grade;                   // 记录等级
    double salary;               // 记录薪水
};
```

类的成员函数的定义和普通的函数定义一样，包括返回类型、函数名、参数列表和函数体，类的成员函数必须在类的声明中定义。在类的成员函数中，可以直接使用类的成员变量。类的成员变量和成员函数的定义，没有严格的顺序要求，可以以任何的顺序出现。可以先定义成员变量，再定义成员函数；也可以先定义成员函数，再定义成员变量；也可以变量和函数交替定义。

在上面的定义中，函数在前面，变量在后面，这是可以的。而在定义全局变量和全局函数的时候，这样的情况是不允许的。

13.1.4 类的组织结构

类封装了数据成员和成员函数，下面介绍类是怎么组织这些成员的，类的大小如何，类的数据成员是如何排列的，以及编译器如何处理成员函数。下面从一个简单的例子入手，先定义一个最简单的类：

```
class simple
{
    int n;
};
```

前面我们学习了使用 sizeof 可以得到一个类型的长度，比如说 sizeof(int)得到的就是 4。在程序中可以用下面的语句来输出类 simple 的长度：

```
cout<<sizeof(simple)<<endl;
```

输出的结果也是 4，那就是说类 simple 的长度也是 4。修改类 simple 的定义，为其再增加一个整型的数据成员，这时其长度为 8。可以看出，类的长度与其数据成员的长度之和有关。下面为 simple 添加成员函数：

```
class simple
{
public:
    int n;              // 定义数据成员
    int m;              // 定义数据成员
    int fun();          // 定义成员函数
};
```

为类 simple 添加了函数定义之后，用上面的方法输出其长度，其长度还是 8。也就是说类的长度只跟数据成员有关，而和成员函数无关。图 13.1 是类 simple 的结构示意图，类中的数据成员，按照定义的先后顺序排列，类的大小完全由数据成员决定。

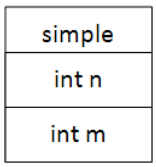


图 13.1　simple 类的结构图

13.1.5 分离成员函数的定义与实现

在前面的章节中讲到，可以把函数的定义和实现分开。在类的成员函数的定义中，也可以把函数的定义和实现分开，即在类中只有函数的定义，而在类外面实现函数。比如上面的函数可以这么定义：

```
class Employee              // 定义 Employee 类
{
public:                     // 定义变量的访问限制
    void performance();     // 只有函数的定义
    char name[32];          // 记录名字
```

```
    int grade;              // 记录等级
    double salary;          // 记录薪水
};
```

在类外面实现函数的时候，要在函数名字前面加上类名字做限制，使用双冒号::来表示函数是属于哪个类的。下面的代码是 performance 函数的实现：

```
void Employee::performance()
{
    // 根据年度工作量评定新的 grade
    grade = grade + 1;      // 为了简化程序，直接提高等级
    // 根据表现修改 salary
    salary = salary + 500;  // 简化程序
}
```

通常情况下，在程序设计中，为了使类的结构清晰，在头文件中只放置类的定义，而类成员函数的实现通常放在.cpp 文件中。

下面的例子是对上面代码的总结，其中需要两个文件，类的定义放在头文件中，而类的实现则放在.cpp 文件中，详细代码如示例代码 13.1 所示。

示例代码 13.1

```
//////////////////////////////////////////////////////////////////////////
// 范例 13-1.h   项目头文件
#pragma once                // 防止头文件被包含多次
class Employee
{
public:
    void performance();
    char name[32];
    int grade;
    double salary;
};
//////////////////////////////////////////////////////////////////////////
// 范例 13-1.cpp   主程序文件
#include <iostream>
using namespace std;
#include "范例 13-1.h"
void Employee::performance()
{
    // 根据年度工作量评定新的 grade
    grade = grade + 1;      // 为了简化程序，直接提高等级
    // 根据表现修改 salary
    salary = salary + 500;  // 简化程序
}
int main()
{
    cout<<"类的定义"<<endl;
    return 0;
}
```

建立一个控制台工程以及相应的.h 和.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 13.2 所示。

```
类的定义
Press any key to continue . . .
```

图 13.2　类的定义示例结果

上面的例子中只定义了职员类 Employee，并没有展示如何使用这个类，在后面的小节中将展示如何使用这个类。

13.2 类对象

前面介绍了如何定义一个类，以及如何定义类的成员函数和成员数据。定义的类是一个用户自定义类型，可以像 C++内部数据类型和其他自定义类型一样使用。本节将介绍如何使用类创建类对象，以及如何访问类的成员函数和成员数据。

13.2.1 定义类对象

类里封装了成员变量和成员函数，这些成员都依附于类，在类外不可以直接操作成员变量和成员函数，必须通过类来调用。但是类是一个类型，不占用存储空间，不可以直接对类进行操作，就像不可以直接给 int 赋值一样，所以下面的代码都是错误的：

```
int = 5;                // 直接给整型赋值，错误
Employee.grade = 8;     // 给类赋值，错误
```

因此要使用类，就要先定义一个类对象，定义类对象和定义普通的变量一样，例如：

```
int a;                  // 定义一个整型变量
Employee emp;           // 定义一个 Employee 类的对象
```

上面的代码定义了一个整型的变量 a，还定义了一个 Employee 类的对象 emp。也可以定义一个类的指针，例如：

```
Employee * pEmp = NULL;     // 定义一个类指针
pEmp = new Employee();      // 指针指向一个新生成的对象
```

类是个类型，因此类只需定义一次，一个类可以定义很多对象。类本身不占用内存，只有定义了类对象之后，对象才占用内存空间。如图 13.3 所示是类与对象的关系。

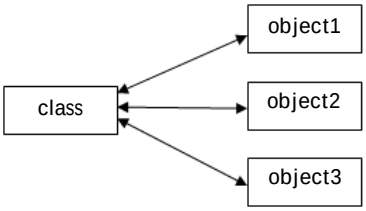


图 13.3　类与对象的关系

13.2.2 访问类对象成员

一旦定义了一个实际的 Employee 类的对象，就可以使用点运算符“.”来访问类对象的成员。比如对于上面定义的 emp 对象，要把该对象的 grade 成员数据设置为 8，应该这么写：

```
emp.grade = 8;
```

要调用类的函数，也要用同样的方法，例如：

```
emp.performance();      // 调用类对象的函数
```

前面讲过通过“*”操作符可以得到指针指向的对象，因此也可以像上面那样操作类指针的成员。对于前面定义的类指针 pEmp，可以这么操作：

```
(*pEmp).grade = 8;      // 调用数据成员
(*pEmp).performance(); // 调用函数成员
```

在通过类指针调用类成员之前，同样要保证指针指向正确的地址。

对于上面对类指针的调用，在 C++中通常用另外一个组合操作符“->”来代替，这个操作符可以直接操作类指针的成员，因此上面的代码还可以用下面的代码替代：

```
pEmp->grade = 8;        // 调用类指针的数据成员
pEmp->performance();    // 调用类指针的成员函数
```

上面两种调用类指针成员的方法是等效的，不过通常用后面的方法。本书后面使用到类指针的时候都使用后一种方法，也就是使用操作符“->”来调用。下面继续完善前面的范例，增加对 Employee 类的使用，详细代码如示例代码 13.2 所示。

示例代码 13.2

```
////////////////////////////////////////////////////////////////////////
// 范例 13-2.h       项目头文件
#pragma once            // 防止头文件被包含多次
class Employee
{
public:
    void performance();
    char name[32];
    int grade;
    double salary;
};
////////////////////////////////////////////////////////////////////////
// 范例 13-2.cpp     主程序文件
#include <iostream>
using namespace std;
#include "范例 13-2.h"
void Employee::performance()
{
    cout<<"调用 performance 函数"<<endl;
    // 根据年度工作量评定新的 grade
```

```
    grade = grade + 1;      // 为了简化程序，直接提高等级
    // 根据表现修改 salary
    salary = salary + 500;  // 简化程序
}
int main()
{
    Employee emp;                   // 定义类对象
    strcpy(emp.name,"Jason");       // 通过类对象给数据成员赋值
    emp.grade = 8;
    emp.salary = 7000;
    emp.performance();              // 调用成员函数
    Employee * p = &emp;            // 用类指针指向类对象
    cout<<p->name<<endl;            // 输出类数据成员
    cout<<p->grade<<endl;
    cout<<p->salary<<endl;
    return 0;
}
```

建立一个控制台工程，以及相应的.h 和.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 13.4 所示。

图 13.4　类的使用示例结果

上面的例子中定义了职员类 Employee 的对象，给这个类对象的数据成员赋值并调用成员函数，然后使用指针来指向这个对象，并且通过指针来访问数据成员。

13.2.3　隐含的 this 指针

在类的成员函数内部，暗含着一个名字是 this 的指针，这个指针指向了调用该函数的类对象。比如在前面的例子中有如下调用：

```
emp.performance();                  // 调用成员函数
```

那么在函数 performance 的内部，这个 this 指针指向的就是 emp 对象。这个指针由编译器去维护，不需要显式地定义，可以直接使用，并且不需要担心这个指针的指向是否正确。下面用一个简单的例子来表明编译器提供这个指针的必要性。

下面定义了一个类 Ball，用这个类来描述一个小球的移动，因此这个 Ball 类需要提供几个移动的函数 moveLeft，moveRight，moveDown 和 moveUp。

```
class Ball                          // 定义 Ball 类
{
public:
    void moveLeft(int dept);        // 向左移动函数
    void moveRight(int dept);       // 向右移动函数
    void moveDown(int dept);        // 向下移动函数
```

```
    void moveUp(int dept);          // 向上移动函数
};
```

在一次移动中，可能会要小球先向左移动 1 个单位，再向下移动 2 个单位，那么对于小球对象 ba，理想情况下希望对用户的调用连成一个连续的表达式：

```
ba.moveLeft(1).moveDown(2);
```

这个语句等价于：

```
ba.moveLeft(1);
ba.moveDown(2);
```

那么，在定义函数的时候，就必须有一个返回值，返回一个该类对象的引用，因此上面的类定义应当修改为：

```
class Ball                              // 定义 Ball 类
{
public:
    Ball & moveLeft(int dept);          // 向左移动函数
    Ball & moveRight(int dept);         // 向右移动函数
    Ball & moveDown(int dept);          // 向下移动函数
    Ball & moveUp(int dept);            // 向上移动函数
};
```

这些函数的返回类型是 Ball&，指明该成员函数返回对其自身类类型的对象的引用，每个函数都返回调用自己的那个对象。在这里，要想返回自身对象，只能使用 this 指针，下面是函数的实现：

```
Ball & Ball::moveLeft( int dept )       // 函数实现
{
    // 移动当前的位置，省略代码
    return *this;                       // 返回当前对象
}
```

使用 this 指针是比较便捷的得到当前对象的方法，在类的操作符重载中还有应用。

13.3 类成员的访问限制

类的特性是封装和抽象，也就是只向外部提供有限的功能接口，其他部分只有内部可见，外部不能访问。在定义类的时候，可以在成员的前面加上访问限制，来限制该成员是否可以被外部访问。本节介绍怎么对类成员的访问进行限制。

13.3.1 一般访问限制

在定义类的时候，可以在成员的前面添加访问限制关键字。一般访问限制的关键字有 public，private 和 protected，它们的访问限制如下。

- public：公有的，这个限制后面所定义的成员（包括数据和函数），可以被外部访问。一般抽象出来供外部使用的接口，都要定义为 public 限制的。

- private：私有的，表示定义的成员只供本类内部使用，外部不可以使用。一般情况下受保护的数据成员和内部函数要定义为 private 限制的。
- protected：受保护的，其特性和 private 一致，外部不可用，只能内部用。不过这个关键字主要用在类继承的时候基类的定义中，本节不做过多的讨论。

友元不受 public，private 和 protected 的限制。

访问限制的使用格式是关键字加冒号":"，例如：

```
class myClass
{
public:             // 公有限制
int a;
private:            // 私有限制
int b;
};
```

上面在类 myClass 中定义了一个公有的变量 a 和私有的变量 b，因此在外部只可以访问 a，而不可以访问 b，看下面的访问代码：

```
myClass my;         // 定义类对象
my.a = 1;           // 正确，可以访问 public 的变量
my.b = 1;           // 错误，不可以访问 private 的变量
```

当定义类的时候，其成员的访问限制默认都是 private，也就是说如果不加任何访问限制，那么所有的成员都是 private 限制的。另外不需要在每一个成员前面都加上访问限制，如果一个成员前面没有访问限制，则其访问限制和前面定义的成员相同。看下面的定义：

```
class myClass
{
    int a;              // private
public:
    int b;              // public
    int c;              // public
private:
    int d;              // private
public:
    int e;              // public
};
```

在上面的定义中，a 是默认的 private，b 因为前面的访问限制所以为 public，而 c 和 b 相同，所以也为 public，d 为 private，e 为 public。

13.3.2 私有与安全性

在前面设计的 Employee 类中，所有的数据都设计为 public，这是非常不可取的。对于 grade 和 salary，是不能随便修改的，只能通过 performance 的评定来改动，因此有必要修改为 private 限制。作为设计的一般准则，应当保持类的数据成员为 private，限制外部随便访问。外部可以随

便修改类的数据成员是非常不安全的，那就像一个外壳打开了的电视机，可以随便去更换元器件，这样谁也不能保证电视机能正常工作。

把数据成员设置为 private，在外部需要得到这个数据的时候，可以提供一个公有的函数来获取或者设置私有成员变量。这也是面向对象中很常用的方法，即使用公有函数来存取私有变量，这样有以下好处：

- 可以很容易地修改实现细节，把细节封装在函数中，对于外部的接口不变，修改了内部细节造成的影响很小。
- 可以统一对数据的有效性、完整性进行检查，而不用担心有什么遗漏。
- 可以提高对数据的保护，防止外部随便修改数据。

因此，前面定义的 Employee 类，应该修改为如下：

```
class Employee              // 定义 Employee 类
{
public:                     // 定义变量的访问限制
    void performance();     // 接口函数限制定义为 public
    int getGrade();         // 返回等级
    double getSalary();     // 返回薪水
private:                    // 数据成员限制定义为 private
    char name[32];          // 记录名字
    int grade;              // 记录等级
    double salary;          // 记录薪水
};
```

13.3.3 友元

前面讲到，一般情况下应当把数据成员限制设置为 private 以限制外部的访问。不过这样也有局限性，必须要为所有外部可能用到的变量添加存取函数，即使只有一个外部函数会用到这个成员变量。因此可以适当地让某些外部函数可以访问私有变量。

在某些情况下，允许特定的非成员函数访问一个类的私有成员，同时仍然阻止一般的访问，这就是友元（friend）。友元机制允许一个类将其非公有成员的访问权限授予指定的函数或类。友元的声明以关键字 friend 开始，它只能出现在类定义的内部。友元声明可以出现在类中的任何地方，并且友元不受其声明出现部分的访问控制的影响。

通常情况下把友元的声明放在类的开始部分或者结尾部分，这样比较直观。

要声明友元，使用 friend 关键字。比如 Employee_Mgr 类专门用来管理 Employee 类，要想让 Employee_Mgr 的对象可以直接访问 Employee 的私有数据，可以为 Employee 的定义添加友元声明：

```
class Employee
{
friend class Employee_Mgr;          // 添加友元声明
……
};
```

类 Employee_Mgr 必须在上面的声明之前已经定义。

添加了上面的友元声明之后，Employee_Mgr 类的所有函数都可以访问类 Employee 对象的所有成员变量和成员函数，无论这些成员是私有的还是公有的。如果不希望 Employee_Mgr 类的所有函数都具有这个特权，可以只为某几个函数设置特权。假如只希望 Employee_Mgr 类的 feedback 函数有这样的权限，那么可以在 friend 后面跟上该函数的声明。也可以设置全局函数为友元，这时候这个全局函数就具备访问私有成员的权限。

下面的例子中定义了一个简单的类 A，A 有一个私有的成员 id，Manager 类和运算符<<通过友元声明来访问 A 的私有成员。详细的代码如示例代码 13.3 所示。

示例代码 13.3

```
#include <iostream>
using namespace std;
class A;                              // 预定义类 A，因为在 Manager 类中要使用 A 类
class Manager
{
public:
    void adjustID( A& a );
};
class A
{
    friend ostream & operator<<( ostream & out, A & a );       // 声明友元
    friend void Manager::adjustID( A & a );
private:
    int id;
};
void Manager::adjustID( A & a )
{
    static int index = 0;       // 声明为静态变量
    a.id = index;
    index ++;                   // 增加 index，以便下次使用
}
ostream & operator<<( ostream & out, A & a )
{
    out<<"id="<<a.id<<endl;     // 输出 id
    return out;                 // 返回输出流，以便继续输出
}
int main()
{
    // 定义 4 个类对象
    A a1;
```

```
    A a2;
    A a3;
    A a4;
    Manager mgr;
    // 调整对象的 ID
    mgr.adjustID( a1 );
    mgr.adjustID( a2 );
    mgr.adjustID( a3 );
    mgr.adjustID( a4 );
    cout<<a1<<a2<<a3<<a4;       // 输出对象
    return 0;
}
```

建立一个控制台工程以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 13.5 所示。

图 13.5 访问限制与友元示例结果

上面的例子中 A 类的成员 id 是 private 限制的，因此正常情况下外部是不能访问这个成员的。为了让 Manager 和操作符<<具有访问权限，在定义 A 类的时候增加了友元声明。

13.4 类的构造函数

构造函数是特殊的成员函数，只要创建了类类型的对象，都要执行构造函数。构造函数的工作是保证每个对象的数据成员具有合适的初始值。本节重点介绍怎么定义和使用构造函数。

13.4.1 构造函数的定义

对于一个整型的变量，在定义的时候可以同时赋一个初值，例如：

```
int a = 1;
```

但是对于类对象，比如前面定义的类 Employee，如果想要在定义类对象的时候赋初值，比如：

```
Employee emp = {"Jason", 8, 7000};
```

上面例子是想在定义的时候，让 emp 对象的数据成员分别为 Jason，8，7000，其实这是不可以的，类对象不可以直接这么赋初值。要想让类对象在生成的时候有一个初值，就要使用构造函数。

构造函数是特殊的成员函数，与其他成员函数不同，构造函数和类同名，而且没有返回类型。与其他成员函数相同的是，构造函数也有形参表和函数体。一个类可以有多个构造函数，每个构造函数必须有与其他构造函数不同数目或类型的形参。下面就为 Employee 类添加构造函数：

```
class Employee                              // 定义 Employee 类
{
public:                                     // 定义变量的访问限制
    Employee( char * n, int g, double s )   // 定义构造函数
```

```
    {
        strcpy( name, n );
        grade = g;
        salary = s;
    }
// 其他部分跟前面的定义相同，省略
};
```

构造函数也受访问限制的约束。

在上面的定义中增加了构造函数，本构造函数接收 3 个参数。有了这个定义之后，就可以在定义类对象的时候赋初值了，例如：

```
Employee emp( "Jason", 8, 7500 );
```

在创建对象的时候，编译器会自动调用构造函数，不需要用户来调用。

13.4.2 构造函数的重载

可以为一个类声明的构造函数的数量没有限制，因此可以根据需要声明多个构造函数，不同的构造函数允许用户指定不同的方法来初始化类。不过同函数的重载一样，这些构造函数的形参必须不同。当一个类定义了多个构造函数的时候，编译器会根据提供的实参的类型和数目来选择构造函数。下面为 Employee 类增加不同的构造函数：

```
class Employee          // 定义 Employee 类
{
public:                         // 定义变量的访问限制
    Employee( char * n, int g, double s ); // 定义带 3 个参数的构造函数
    Employee( char * n );       // 定义带 1 个参数的构造函数
    Employee();                 // 定义无参数的构造函数
// 其他部分跟前面的定义相同，省略
};
Employee emp1("Jason",8,7000); // 用 3 个参数创建，调用带 3 个参数的构造函数
Employee emp2("Jack");         // 用 1 个参数创建，调用带 1 个参数的构造函数
Employee emp3;                 // 不带参数的创建，调用无参数的构造函数
```

用于初始化一个对象的实参类型决定了调用哪个构造函数。在 emp1 的定义中，提供了 3 个参数，所以用带 3 个参数的构造函数初始化 emp1；emp2 提供了 1 个参数，因此调用带 1 个参数的构造函数初始化 emp3；在 emp3 的定义中，没有提供初始化参数，所以调用无参数的构造函数初始化 emp3。

13.4.3 默认构造函数

如果没有为一个类定义任何构造函数，那么在初始化的时候会调用默认构造函数。默认构造函数是由编译器生成的，为所有的数据成员提供初始化。默认构造函数没有任何参数。在最初定义的Employee类中没有构造函数，这个时候使用的就是由编译器提供的默认构造函数。

如果一个类没有定义构造函数，那么编译器会为其添加一个构造函数；但是一个类一旦定义了构造函数，那么编译器就不再提供默认构造函数。看下面的类定义：

```
class myClass
{
    // 定义成员，这里省略
}
```

在没有提供构造函数的时候，可以用下面的方法生成类对象：

```
myClass my;
```

编译器会为该类添加默认构造函数，这里就调用默认构造函数。但是当为该类定义了构造函数的时候，比如：

```
class myClass
{
public:
    myClass( int n );
    // 定义成员，这里省略
}
```

因为该类定义了构造函数，所以编译器不再为其添加默认构造函数，那么这个类就没有无参数的构造函数。因此构造这个类的对象的时候必须提供参数，也就是说下面的语句是错误的：

```
myClass my;         // 错误
```

上面的语句是错误的，因为编译器找不到无参数的构造函数定义，正确的方法就是提供一个初始参数：

```
myClass my(1);
```

如果为类提供了构造函数，同时也应该为其提供一个无参数的构造函数。

13.4.4 复制构造函数

前面介绍了类的构造函数，可以方便地为类指定初始值，下面考虑用一个现有的类对象来创建新的对象的过程。对于 C++内部数据类型，这个过程很容易理解，例如：

```
int a = 0;
int b = a;                  // 用 a 来初始 b
```

对于一个类对象，也可以像上面那样创建，例如：

```
Employee emp1( "Jason", 7, 7000 );
Employee emp2 = emp1;
```

编译器会用位复制来执行上面的过程，新生成的 emp2 所有的数据都和 emp1 一样。对于前面定义的 Employee 类，这个过程是可以接受的。但是对于一些包含了指针数据成员的类，位复制的时候只会复制这个指针，两个指针指向同样的地址，后面的对象依赖于前面的对象，因此仅有位复制是不行的。

在 C++中，可以用自己定义的函数来替代位复制，这个函数就是复制构造函数。复制构造函数是一个特殊的构造函数，该构造函数接收一个本类类型的引用作为参数，参数数目只能是一个。通过使用复制构造函数，可以用一个现有的类对象来创建一个新的类对象。下面为 Employee 类添加一个复制构造函数：

```
class Employee
{
public:
    Employee( const Employee & e );     // 定义复制构造函数
    ......// 省略了其他的构造函数以及成员的定义
};
```

复制构造函数的参数必须使用引用。

有了上面的定义之后，用 emp1 来创建 emp2 的时候，就会调用这个复制构造函数，并且把 emp1 作为参数传入。复制构造函数中要实现对象的复制，因此一般情况下要在复制构造函数中复制成员变量。下面是 Employee 类的复制构造函数的实现：

```
Employee::Employee( const Employee & e )   // 复制构造函数的实现
{
    strcpy( name, e.name );
    grade = e.grade;
    salary = e.salary;
}
```

同构造函数一样，如果一个类没有定义复制构造函数，编译器会自动添加一个复制构造函数。不过一个类只能有一个复制构造函数。前面介绍了复制构造函数的定义，下面介绍在什么情况下编译器会调用复制构造函数。简单地说就是在由类对象创建新的类对象的时候，有下面 3 种表现形式：

- 在定义类对象的时候直接用其他的类对象初始化，比如前面的 Employee emp2 = emp1。
- 调用函数的时候。如果函数的参数为类类型，且没有定义为引用，这时函数是值传递的，

编译器会自动为传入的参数创建一个复制，这个时候会调用复制构造函数。

- 在函数返回的时候。如果函数的返回类型为类类型，且没有定义为引用。那么在函数返回的时候编译器也会用一个复制返回，这个时候会调用复制构造函数。

13.4.5 构造函数初始化列表

与任何其他函数一样，构造函数具有名字、形参表和函数体。与其他函数不同的是，构造函数也可以包含一个构造函数初始化列表，构造函数初始化列表紧跟在构造函数的后面，以一个冒号开始，接着是一个以逗号分隔的数据成员列表，每个数据成员后面跟一个放在圆括号中的初始化式。下面为 Employee 的构造函数提供初始化列表：

```
Employee::Employee():grade(0),salary(0)        // 构造函数初始化列表
{
}
```

在 C++中构造函数的初始化列表是非常有用的，这点让初学者非常疑惑，因为下面的语句也能达到同样的目的，且更加直观：

```
Employee::Employee()
{
    grade = 0;
    salary = 0;
}
```

上面两种方法是有区别的：第一种方法对每个变量只有一次初始化，而后一种方法则先用默认值初始化，然后又赋一个值，也就是说在进入构造函数之前已经完成了初始化。所以说使用初始化列表的效率更高。使用初始化列表时，有一些需要注意的地方：

- 不是所有的数据成员都必须出现在初始化列表中。
- 初始化列表中每个成员只能出现一次，不可以重复初始化。
- 数据成员在初始化列表中的出现顺序与类中定义的顺序无关。

下面的例子中定义了一个 Point 类，用来表示在三维坐标系统下的一个点，类的构造函数可以接收 3 个参数，或者忽略 z 坐标，或者不提供初始化参数。详细的代码如示例代码 13.4 所示。

示例代码 13.4

```
#include <iostream>
using namespace std;
class Point
{
    friend ostream & operator<<( ostream & out, const Point & p );
                                                        // 输出函数
public:
    Point( int x, int y, int z ):                       // 3 个参数的构造函数
      m_x(x),m_y(y),m_z(z)
    {
        cout<<"3 参数构造函数被调用"<<*this<<endl;        // 输出数据
    }
    Point( int x, int y ):                              // 2 个参数的构造函数
      m_x(x),m_y(y),m_z(0)
```

```
    {
        cout<<"2 参数构造函数被调用"<<*this<<endl;          // 输出数据
    }
    Point( ):                                              // 无参数的构造函数
      m_x(0),m_y(0),m_z(0)
    {
        cout<<"无参数构造函数被调用"<<*this<<endl;
    }
    Point( const Point & p ):                              // 复制构造函数
    m_x(p.m_x),m_y(p.m_y),m_z(p.m_z)
    {
        cout<<"复制构造函数被调用"<<*this<<endl;           // 输出数据
    }
private:
    int m_x,m_y,m_z;
};
// 重载输出操作符
ostream & operator<<( ostream & out, const Point & p )
{
    out<<" x = "<<p.m_x<<" y = "<<p.m_y<<" z = "<<p.m_z;
    return out;
}
Point fun( Point p )
{
    return p;
}
int main()
{
    Point p1;
    Point p2( 10,10 );
    Point p3( 10,10,10 );
    Point p4 = p2;
    fun( p3 );
}
```

建立一个控制台工程以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 13.6 所示。

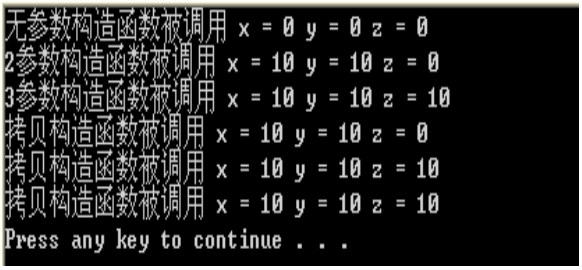


图 13.6　构造函数示例结果

在本例中为 Point 类定义了一个友元输出函数，所以程序可以像输出普通变量那样输出类的数据。在程序代码中，根据提供参数的不同，编译器会选择不同的构造函数。在 fun 的调用过程中，会调用两次复制构造函数，一次是在参数传递过程中，一次是在函数返回时。

13.5 类的析构函数

在创建对象的时候会调用构造函数，相应地，在释放对象的时候会调用析构函数。析构函数也是类中一个特殊的成员函数，同构造函数一样，析构函数也是由编译器自动调用的。本节重点介绍析构函数的定义和使用。

13.5.1 析构函数的定义

析构函数与构造函数相反，当对象脱离其作用域时（例如对象所在的函数已调用完毕），系统自动执行析构函数。析构函数往往用来做“清理善后”的工作，例如在建立对象时用 new 开辟了一片内存空间，应在退出程序前在析构函数中用 delete 释放。

析构函数的定义跟构造函数类似，使用波浪线“~”加类名作为函数名，没有返回类型，没有参数。下面为前面定义的 Employee 类添加一个析构函数的定义：

```
class Employee
{
public:
    ~Employee();                // 析构函数的定义
}
```

一般情况下，构造函数和析构函数都要定义为 public 限制的。

一个类可以定义多个构造函数，但是只能定义一个析构函数。这是因为构造函数可以定义参数，可以通过初始化形参列表的不同来区分，而析构函数不需要提供参数，并且是在释放时由编译器自动调用的，因此析构函数只能定义一个。

13.5.2 默认析构函数

如果没有为类定义析构函数，那么编译器也会自动添加一个析构函数，不过这个由编译器添加的析构函数不进行任何操作。因此，如果有一些需要自己去释放的东西，比如指针或者打开的文件之类的，就一定需要定义析构函数；如果没有这些需要自己释放的东西，可以不定义析构函数，由编译器来提供。下面为前面示例中的类添加一个析构函数，详细的代码如示例代码 13.5 所示。

示例代码 13.5

```
#include <iostream>
using namespace std;
class Point
{
    friend ostream & operator<<( ostream & out, const Point & p );
                                        // 输出函数
public:
```

```
    Point( int x, int y, int z ):      // 3个参数的构造函数
      m_x(x),m_y(y),m_z(z)
    {
        cout<<"3参数构造函数被调用"<<*this<<endl;
    }
    Point( int x, int y ):             // 2个参数的构造函数
      m_x(x),m_y(y),m_z(0)
    {
        cout<<"2参数构造函数被调用"<<*this<<endl;
    }
    Point( ):                          // 无参数的构造函数
      m_x(0),m_y(0),m_z(0)
    {
        cout<<"无参数构造函数被调用"<<*this<<endl;
    }
    Point( const Point & p ):          // 复制构造函数
    m_x(p.m_x),m_y(p.m_y),m_z(p.m_z)
    {
        cout<<"复制构造函数被调用"<<*this<<endl;
    }
    ~Point()                           // 析构函数
    {
        cout<<"析构函数被调用"<<endl;
    }
private:
    int m_x,m_y,m_z;
};
ostream & operator<<( ostream & out, const Point & p )
{
    out<<" x = "<<p.m_x<<" y = "<<p.m_y<<" z = "<<p.m_z;
    return out;
}
Point fun( Point p )
{
    return p;
}
int main()
{
    Point p1;
    Point p2( 10,10 );
    Point p3( 10,10,10 );
    Point p4 = p2;
    fun( p3 );
}
```

建立一个控制台工程以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 13.7 所示。

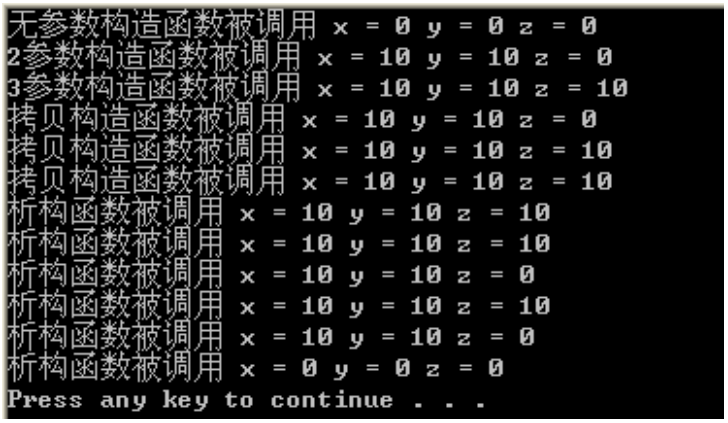


图 13.7 析构函数示例结果

本例中共创建了 6 个对象，所以会调用 6 次析构函数。前两次析构函数是在函数返回时被调用的，后面的 4 次析构函数是在 main 函数返回时被调用的。

13.6 类的 static 成员

对于特定类类型的全体对象而言，访问一个全局对象有时是有必要的，例如，在程序的任意点需要统计已创建的特定类类型对象的数量。但是全局对象会破坏封装，并且没有安全保护，一般的用户代码就可以修改这个值。类可以定义静态成员，达到所有对象公用的目的，但是又不破坏封装。

通常，非 static 数据成员存在于类类型的每个对象中。不像普通的数据成员，static 数据成员独立于该类的任意对象而存在，每个 static 数据成员是与类关联的对象，并不与该类的对象关联。正如类可以定义共享的 static 数据成员一样，类也可以定义 static 成员函数。static 成员函数没有 this 形参，可以直接访问所属类的 static 成员，但不能直接使用非 static 成员。使用类的 static 成员有以下的好处：

- 有利于类的封装，可以把 static 成员定义为私有成员，防止外部访问。
- static 成员是与特定的类相关联的，在外部使用时必须要用类名字作为前缀，可使程序更加清晰。
- static 成员的名字在类的作用域中，可以避免命名冲突。

13.6.1 定义 static 成员

在成员声明前加上关键字 static 可将成员定义为 static 成员。static 成员遵循正常的公有、私有访问限制。每一个要定义为 static 的成员前面都需要有 static 关键字，这点和访问限制不同。对于前面定义的 Point 类，加一个静态的数据和静态的函数，用来表示程序中有几个类处于活动状态：

```
class Point                          // 定义类 Point
{
public:
    static int getCount();           // 定义静态函数
    static int count;                // 定义静态数据成员
};
```

在上面的定义中，先定义了一个 static 函数。static 函数的定义和普通的类成员函数的定义一样，可以直接在定义的后面实现，也可以在类的外面提供函数实现。当在类外面实现的时候，不需要 static 关键字。

在静态函数中，不可以使用 this 指针，因为静态函数是所有对象共用的。同理，在静态函数中也不可以使用非静态的成员变量，只可以使用类的静态变量。

后面是一个 static 数据成员，跟普通的数据成员不同，static 数据成员必须在类定义体的外部定义，定义的时候必须要有类名作为前缀。因此要使用前面定义的 count，还必须有下面的定义：

```
int Point::count = 0;                // 在类外面定义 static 变量并且初始化
```

13.6.2 使用 static 成员

可以使用域操作符::从类直接调用 static 成员，或者通过对象、引用或指向类类型对象的指针间接调用。例如，可以用下面的方法来访问前面定义的静态成员：

```
Point::count = 0;                    // 给静态变量赋值
int n = Point::getCount();           // 调用静态函数
```

在任何地方都可以用上面的方法通过域操作符直接访问 static 成员。不过这个访问也受访问限制的约束。外部不可以访问 private 的 static 成员。也可以通过类对象对静态成员进行访问，下面的访问是可以的：

```
Point center;
center.count = 0;                    // 通过类对象访问静态数据成员
Point * p = &center;
int n = p->getCount();               // 通过指针访问静态函数
```

当在类的内部使用静态成员时，可以直接使用，不需要域操作符。

可以用使用非 static 成员的方法来使用 static 成员。

13.7 综合实例

在本节中，将详细讲解如何使用类的封装来解决实际问题。由于类的封装比较复杂，所以这些例子的代码都比较长，希望读者能耐心阅读。

13.7.1 人的活动

在本例中，定义一个人的类，这个人有名字，他一天的活动包括起床、吃饭、上班和睡觉，分别调用这些方法来模拟人一天的活动。程序如示例代码 13.6 所示。

示例代码 13.6

```
#include <iostream>
#include <string>
using namespace std;
class person{               // 定义 person 类
public:
    person( char * name );  // 定义构造函数
    void GetUp();           // 定义起床函数
    void Eat();             // 定义吃饭函数
    void Work();            // 定义工作函数
    void Sleep();           // 定义睡觉函数
private:
    string m_name;
};
person::person( char * name ): // 实现构造函数
m_name(name)
{
}
void person::GetUp()           // 实现起床函数
{
    cout<<m_name<<" 起床了！"<<endl;
}
void person::Eat()             // 实现吃饭函数
{
    cout<<m_name<<" 去吃饭！"<<endl;
}
void person::Work()            // 实现工作函数
{
    cout<<m_name<<" 上班去了！"<<endl;
}
void person::Sleep()           // 实现睡觉函数
{
    cout<<m_name<<" 睡觉了了！"<<endl;
}
int main()
{
```

```
    person jason("Jason");      // 定义类对象
    jason.GetUp();
    jason.Eat();
    jason.Work();
    jason.Sleep();
    return 0;
}
```

建立一个控制台工程以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 13.8 所示。

图 13.8 人的活动示例结果

13.7.2 自定义字符串类

在前面的章节中，介绍过用字符数组表示的字符串，不过受数组的局限性，在使用上非常不方便，不可以直接赋值，还要考虑数组的生存周期。在本例中自定义一个类 String 来表示字符串，用来克服前面提到的缺点。在 String 类的内部，还需要用 char 类型的数组来存储数据，因为字符串的长度不确定，因此用一个 char 类型的指针，使用动态申请数组的方法。为了防止 String 类的数据被随意修改，把字符数组定义为 private 的。下面是类的初步定义：

```
class String            // 定义字符串类
{
private:                // 定义为 private 的，防止外部直接修改数据
    char * m_data;
};
```

在使用字符串的时候，经常在定义的时候直接初始化，也就是可以用字符数组来构造一个字符串类，因此需要定义构造函数。构造函数可以接收字符数组。为了使用方便，还需要定义一个无参数的构造函数。另外，字符串在函数参数传递以及函数返回的过程中，应该保证数据能够复制，因此要定义一个复制构造函数来复制数据。下面是其构造函数的定义：

```
class String                    // 定义字符串类
{
public:
    String( const char * ) ;    // 字符数组作为参数的构造函数
    String();                   // 无参数构造函数
    String( const String & );   // 复制构造函数
private:                        // 定义为 private 的，防止外部直接修改数据
    char * m_data;
```

```
};
```

对于在构造函数中用 new 申请的空间，需要程序员来释放，释放的工作可以放到析构函数中去做，因此还需要为 String 类定义一个析构函数。对于字符串，用户希望能像普通的字符串那样，可以直接输出，而输出需要访问私有数据，所以还需要为其定义一个友元函数。

到这里为止，已经定义了一个简单的字符串类，可以初始化，也可以通过函数传递这个字符串。详细的代码如示例代码 13.7 所示。

示例代码 13.7

```
#include <iostream>
using namespace std;
class String{
    friend ostream & operator<<( ostream & out, String & str );
public:
    String();                       // 定义无参数构造函数
    String( const char * str );     // 定义构造函数
    String( const String & str );   // 定义复制构造函数
    ~String();
private:
    char * m_data;                  // 字符指针，用来存储数据
};
String::String()                    // 无参数构造函数的实现
{
    m_data = new char[1];
    m_data[0] = '\0';
}
String::String( const char * str ) // 构造函数
{
    int len = strlen( str );
    m_data = new char[len + 1];     // 申请数组
    strcpy( m_data, str );          // 复制数据
}
String::String( const String & str )   // 复制构造函数
{
    int len = strlen( str.m_data );
    m_data = new char[len + 1];     // 申请数组
    strcpy( m_data, str.m_data );   // 复制数据
}
String::~String()                   // 析构函数
{
    delete m_data;                  // 释放申请的内存
}
ostream & operator<<( ostream & out, String & str )        // 重载输出
{
    out<<str.m_data;
    return out;
}
void Display( String str )          // 值传递
{
```

```
    cout<<str<<endl;
}
int main()
{
    String s1( "Hello" );          // 用字符串初始化
    String s2;                     // 无初始化参数
    Display( s1 );
    return 0;
}
```

建立一个控制台工程以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 13.9 所示。

图 13.9　自定义字符串类结果

13.8 小结

类是 C++中最基本的特征，C++允许定义新的类型（即类）以适应程序的需要，同时使程序更短且更易于修改。本章学习了如何定义类以及如何使用类。封装是指从常规访问中保护类成员的能力。可以把类的数据成员和成员函数定义为 private 的，使外部不可以访问；对于类所抽象出的接口，应该定义为 public 的，以便外部访问。通过友元可以让外部拥有访问 private 成员的能力。

类可以定义构造函数，这是特殊的成员函数，控制如何初始化类的对象。可以重载构造函数，为类定义多个不同的构造函数。复制构造函数是用原有对象生成新对象时调用的。在释放类的时候会调用析构函数，一个类只能定义一个析构函数。当没有定义构造函数、析构函数的时候，编译器会自动添加一个。另外，还可以在类中定义 static 成员，供该类所生成的对象共用。

◆　　◆　　◆

第 14 章　重载操作符和自定义转换

本章包括

- ◆如何定义和使用重载操作符
- ◆类成员操作符和全局操作符的区别
- ◆输出和输入操作符的重载
- ◆使用赋值操作符
- ◆算术操作符和关系操作符的重载
- ◆自定义转换

上一章学习了类和对象，知道了类是对事物的封装和抽象，通过对类提供的接口的调用可以操作类对象。为了使自定义的类类型能像内置类型那样使用，可以重载操作符。操作符重载和普通的函数重载类似，不同的是使用操作符作为函数名。利用操作符重载，可以为操作符赋予其他的意义。

14.1　重载操作符的定义和使用

在 C++中定义了很多操作符：有数学运算符，比如++，--，+=等；有关系操作符，比如>，<，==等，可以使用这些操作符操作内部数据类型。在 C++中允许把这些操作符和类结合起来，重新定义这些操作符对类的操作，使程序看起来简洁明了。本节重点介绍操作符重载的定义和使用。

14.1.1　重载操作符的定义格式

定义一个重载操作符就像定义一个函数，不过函数的名字是关键字 operator 后接要定义的操作符符号。像其他任何的函数定义一样，重载操作符具有返回类型和参数列表。重载操作符的定义格式为：

```
返回类型 operator 操作符(参数列表);
```

重载不同的操作符，返回类型和参数列表可能不同，但是相同的是所有的重载都需要有 operator 关键字。例如，对于前面第 13 章的综合实例中定义的 String 类，下面为其定义判断相等操作符==：

```
bool operator==( String & str1, String & str2 )
{
    return strcmp( str1.m_data, str2.m_data ) == 0;
}
```

在上面的语句中，把操作符==重载为一个全局函数，这个函数用两个 String 类的引用作为参数，用来判断两个 String 中存储的字符串是否相同。

在重载操作符的定义中，参数列表中参数的个数取决于两个因素：

- ◆ 操作符是一元操作符还是二元操作符。
- ◆ 把操作符定义为类成员函数还是全局函数。

至于上面两个因素是如何影响函数参数个数的，将在后面详细讨论。

重载的操作符也受类成员访问限制的约束。

14.1.2 可重载的操作符

虽然在C++中几乎所有的操作符都可以重载，不过还是有几个不可以重载的。在C++中可以重载的操作符包括：

- 算术运算符+，-，*，/，%，++，--。
- 位操作运算符&，|，~，^，<<，>>。
- 逻辑运算符!，&&，||。
- 比较运算符<，>，>=，<=，==，!=。
- 赋值运算符=，+=　　，=，*=，/=，%=，&=，|=，^=，<<=，>>=。
- 其他运算符[]，()，->，，，new，delete，new[]，delete[]，->，*。

对于上面列出来的可重载的操作符，后面会陆续介绍其意义以及使用方法。

不可以重载的操作符有::，.，.*和?:。

14.1.3 使用重载操作符

当为某个类重载了操作符之后，就可以像内部类型那样使用操作符操作类对象了。虽然前面说过重载操作符就像定义函数，不过操作符的使用和函数不一样。使用操作符时，不需要用对象加点“.”操作，也不需要像函数那样把函数名放在前面。使用操作符时，可以把操作符放在两个操作对象的中间。前面为类String类定义了判断相等的操作符==，下面的代码是对这个操作符的使用：

```
String str1("Hello");          // 定义字符串类对象
String str2("World");          // 定义字符串类对象
if( str1 == str 2 )            // 调用操作符==，判断两个字符串是否相等
{
    cout<<"两个字符串相等！"<<endl;
} else
{
    cout<<"两个字符串不相等!"<<endl;
}
```

当对对象str1和str2使用==操作符的时候，编译器会查找这个操作符的重载定义，然后像调用函数那样调用操作符，并分别把str1和str2作为第一个和第二个参数。如果没有定义这个操作符重载，则编译器会报告编译错误，不允许这样的调用发生。

由上面的使用例子可以看出，使用重载操作符可以使程序代码看起来更加清晰。不过在使用重载操作符的时候，也有几点应该注意的地方：

- 重载操作符必须有一个类类型或枚举类型参数，也就是说不能所有的参数都为内部类型，至少有一个是自定义的类类型或者枚举类型，这是因为操作符对于内部类型的意义都是固定的。

- 不能改变操作符原来的优先级和结合性，重载之后的操作符还具有操作内部类型的时候的优先级和结合性。
- 重载操作符不保证操作数的求值顺序，并且对于所有的操作数都要求值。

14.1.4 类成员和全局操作符

在前面给 String 类增加操作符的例子中，定义的是一个全局的操作符，还可以把操作符定义为类的成员函数。对于大多数操作符这两种定义方法都是适用的，下面来讨论这两种定义方法的区别。

前面为 String 类定义的判断相等操作符==是一个全局函数，这个函数不属于任何类，因此这个函数需要两个参数，分别表示==两边的操作对象。也可以把这个操作符定义为 String 类的成员函数，这个时候对于==的调用相当于下面的语句：

```
str1.operator == ( str2 );          // ==操作符的意义
```

==操作符就像 str1 的成员函数一样可以被调用，在这种情况下，这个函数只需要一个参数，因此要把==操作符定义为 String 类的成员时，可以像下面这样定义：

```
class String
{
public:
    bool operator==( String & str );           // 重载操作符==
};
bool String::operator ==( String & str )       // 实现重载函数
{
    return strcmp( m_data, str.m_data ) == 0;
}
```

可以看到，把重载操作符定义为类的成员和定义普通的类成员函数一样，可以在类里定义，而在类外面实现。把操作符定义为成员函数和全局函数，本质上区别并不大，定义为成员函数其实就是省略了第一个参数，然后在实现中用 this 来代替。把操作符定义为类成员，其形参的数目比定义为全局函数的形参数目少 1。

前面提到过，重载操作符的参数列表中参数的个数受两个因素影响：一个是操作符是一元的还是二元的，另一个是把操作符定义为成员函数还是全局函数。对于一元操作符，定义为全局函数时需要一个参数，而定义为成员函数时则没有参数；对于二元操作符，定义为全局函数时需要两个参数，而定义为成员函数时需要一个参数。大部分操作符既可以定义为成员函数，也可以定义为全局函数，不过有部分操作符只能定义为一种格式。

对于++操作符，有两个存在形式，可以为++i，也可以为 i++，就是说既可以是左值操作符，也可以是右值操作符。为了区别这两种情形，一般把++i 定义为全局函数，把 i++定义为成员函数。

对于[]操作符，在内部类型中是用来取数组中元素的，必须定义为成员函数，而不能定义为全局函数。

要把操作符定义为类成员函数，必须定义在该操作符左边的对象的类里。对于<<和>>操作符，因为左边的操作对象为 cout 和 cin 等 stream 对象，而这些类的定义是不能改动的，所以这类操作

符都需要定义为全局函数。

14.1.5 操作符重载和友元的关系

前面介绍了可以把重载操作符定义为全局函数,在这种情况下对类来说该操作符就是一个外部函数。因此要想在操作符中访问类的 private 成员，就必须把其定义为所操作类的友元。在前面为 String 类定义全局的操作符==时，使用了私有成员 m_data，因此要在 String 类中添加这个友元声明，下面是其修改了之后的定义：

```
class String
{
    friend bool operator( String & str1, String & str2 );
    // 这里省略了类的其他部分定义
};
bool operator==( String & str1, String & str2 )
{
    return strcmp( str1.m_data, str2.m_data ) == 0;
}
```

下例将完成对复数的定义。在数学中有复数的概念，复数是由实部和虚部组成的。在 C++中没有复数数据类型，本例中定义一个表示复数的类，类中用两个浮点数来分别表示实部和虚部。在本例中为这个复数类重载操作符==，用来判断两个复数是否相同，其他的操作符重载，会在后续的章节中陆续介绍。详细的代码如示例代码 14.1 所示。

示例代码 14.1

```
#include <iostream>
#include <math.h>
using namespace std;
class ComplexNumber             // 定义复数类
{
    friend bool operator==( const ComplexNumber & c1, const ComplexNumber &
c2 );                           // 重载操作符==
public:                         // 定义公用接口
    ComplexNumber( double r, double I )
    :real(r),imaginary(i)
    {
    }
private:                        // 定义私有数据
    double real;                // 定义复数的实部
    double imaginary;           // 定义复数的虚部
};
bool doubleEqual( double d1, double d2 )   // 判断两个 double 类型的数是否相等
{
    // 对于 double 类型，不可以直接判断相等
    static double verysmall = 0.00001;
    return fabs( d1 - d2 ) < verysmall;
}
bool operator==( const ComplexNumber & c1, const ComplexNumber & c2 )
                                    // 重载操作符==
```

```
{
    return doubleEqual( c1.real, c2.real ) && doubleEqual( c2.imaginary,
c2.imaginary );
}
int main()
{
    cout<<"请分别输入两个复数的实部和虚部:"<<endl;
    double r, i;
    cin>> r >> i;
    ComplexNumber c1 ( r, i );      // 构造复数对象
    cin >> r >> i;
    ComplexNumber c2 ( r, i );      // 构造复数对象
    if( c1 == c2 )
    {
        cout<<"两个复数相等！"<<endl;
    } else
    {
        cout<<"两个复数不相等！"<<endl;
    }
    return 0;
}
```

建立一个控制台工程以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 14.1 所示。

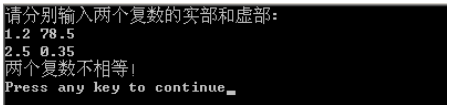


图 14.1 操作符重载结果

对于浮点类型的数，因为存在着精度的问题，不可以直接进行比较。判断两个浮点数是否相等的方法就是看两个数的差是否足够小，因此程序专门定义了一个函数来判断两个浮点数是否相等。这里把判断等==定义为全局函数，所以程序要把操作符定义为类的友元。

14.2 输出和输入操作符的重载

在 C++中为内置类型提供了标准的 I/O 操作，通过标准库 iostream 中定义的 cout 和 cin 对象，可以方便地完成输出和输入操作。如果自定义的类也支持 I/O 操作，那么这些操作的接口应该和 C++中为内置类型提供的接口相同。为了达到这个目的，就需要重载输出和输入操作符，在本节中就主要介绍如何重载输出和输入操作符。

14.2.1 输出操作符<<的重载

当类对象有输出操作的时候，可以重载输出操作符<<。重载输出操作符定义的语法格式为：

```
ostream & operator<<( ostream & out, const classType & obj )
                                                    // 重载输出操作符
{
```

```
    out<< //…输出内容
    return out;
}
```

前面已经提到，重载操作符的时候，应当保留其在内部类型上的意义和使用方法。因此对于输出操作符的重载，第一个参数应当为输出流 ostream 的引用，在该对象上产生输出，输出流为非 const，因为写入到流会改变流的状态，该形参是个引用，因此不能复制 ostream 对象。第二个参数为要输出对象的引用，在这里定义为引用是为了避免复制对象，这个参数可以为 const，因为一般的输出不改变类对象的状态，定义为 const 可以同时输出 const 对象和非 const 对象。因为要保证 cout 的连续输出特性，因此需要返回一个 ostream 对象的引用。

在重载操作符中，经常使用引用而不是指针。

下面是在上一节中已经定义过的 ComplexNumber 类对<<的重载：

```
class ComplexNumber
{
    friend ostream & operator<<( ostream & out, const ComplexNumber & n );
                                            // 定义输出操作符为友元
                                            // 省略其他部分的定义
};
ostream & operator<<( ostream & out, const ComplexNumber & n )
                                            // 操作符<<的实现
{
    out<<n.real<<"+"<<n. imaginary<<"i";    // 输出复数的实部和虚部
    return out;
}
```

常用的复数的表示是 a+bi，上面的输出也是按照这个格式进行的。

在重载输出操作符中，最好不要修改输出格式，最好也不要换行。

前面介绍过，必须要把<<定义为全局函数，而不能定义为类的成员函数。这是因为根据使用习惯，cout 要放在操作符的前面，这样要想把操作符定义为类的成员函数，就必须定义在 ostream 类中，而该类是标准库的组成部分，是不可以修改的，因此只能把该操作符定义为全局对象。因为 I/O 操作通常对非共有数据进行读写，因此类一般要把该操作符定义为友元。

14.2.2 输入操作符>>的重载

输入操作符的重载与输出操作符类似，输入操作符的第一个形参是输入流的引用，第二个形参是要输入的对象。最后还要返回一个输入流，以连续输入。下面是重载输入操作符的定义：

```
istream & operator>>( istream & in, classType & obj )      // 重载输入操作符
{
```

```
    in>> //…输入内容
    return in;
}
```

跟输出操作符对应，输入操作符用的是>>符号，正好表示了数据的流向。在输入的时候，要改变输入类的数据和状态，因此不能像输出那样定义为 const 的引用。下面为前面定义的 ComplexNumber 类添加输入的定义：

```
class ComplexNumber
{
    friend istream & operator>>( istream & in, ComplexNumber & n );
                                            // 定义输入操作符为友元
                                            // 省略其他部分的定义
};
istream & operator<<( istream & out, ComplexNumber & n )  // 操作符>>的实现
{
    In>>n.real>>n.imaginary;                // 输入复数的实部和虚部
    return in;
}
```

这个操作从 istream 形参中读取两个浮点数，分别代表复数的实部和虚部。同输出操作符一样，输入操作符也一定要定义为全局函数，然后在 ComplexNumber 类中定义为友元。

在从 istream 中读取数据的时候，可能会碰上输入错误，这个错误包括：

- 输入流中提供了错误的数据类型。这里需要两个浮点数，如果输入流提供了数字之外的其他字符，可能有类型错误。
- 如果输入流是一个文件，可能会碰上文件结束等错误。

因为在读取数据的时候可能会出现错误，因此要考虑对错误的处理。下面修改上面的代码，添加对错误情况的考虑：

```
istream & operator<<( istream & out, ComplexNumber & n )  // 操作符>>的实现
{
    In>>n.real>>n.imaginary;                // 输入复数的实部和虚部
    If( ! in )
    {
        // 读取数据错误，重置默认值
        n.real = 0;
        n.imaginary = 0;
    }
    return in;
}
```

可以像使用 bool 变量那样对输入流的状态进行判断。

在本节中为前面定义的 ComplexNumber 类添加了输出和输入操作符，可以像内部数据类型那样对 ComplexNumber 对象进行输出和输入。下例是这些代码的汇总，详细的代码如示例代码 14.2 所示。

示例代码 14.2

```
#include <iostream>
#include <math.h>
using namespace std;
class ComplexNumber            // 定义复数类
{
    friend bool operator==( const ComplexNumber & c1, const ComplexNumber &
c2 );                          // 定义==为友元
    friend ostream &operator<<( ostream & out, const ComplexNumber & n );
                                // 定义<<为友元
    friend istream &operator>>( istream & in, ComplexNumber & n );
                                // 定义>>为友元
public:                         // 定义公用接口
    ComplexNumber():real(0),imaginary(0){}
    ComplexNumber( double r, double i )
    :real(r),imaginary(i)
    {
    }
private:                        // 定义私有数据
    double real;                // 定义复数的实部
    double imaginary;           // 定义复数的虚部
};
bool doubleEqual( double d1, double d2 )       // 判断两个 double 类型数是否相等
{
    // 对于 double 类型数，不可以直接判断相等
    static double verysmall = 0.00001;
    return fabs( d1 - d2 ) < verysmall;
}
// 重载操作符==
bool operator==( const ComplexNumber & c1, const ComplexNumber & c2 )
{
    return doubleEqual( c1.real, c2.real ) && doubleEqual( c2.imaginary,
c2.imaginary );
}
ostream & operator<<( ostream & out, const ComplexNumber &n )
{
    out<< n.real<<"+"<<n.imaginary<<+"i";  // 按照 a+bi 的格式输出
    return out;
}
istream & operator>>( istream & in, ComplexNumber & n )
{
    in>>n.real>>n.imaginary;           // 分别输入实部和虚部
    if( !in )
    {
        // 数据输入错误
        n.real = 0;
        n.imaginary = 0;
    }
    return in;
}
int main()
```

```
{
    cout<<"请输入复数的实部和虚部:"<<endl;
    ComplexNumber c;                    // 用默认构造函数创建对象
    cin>>c;                             // 输入对象
    cout<<"你输入的复数为："<<c<<endl;    // 输出对象
    return 0;
}
```

建立一个控制台工程以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 14.2 所示。

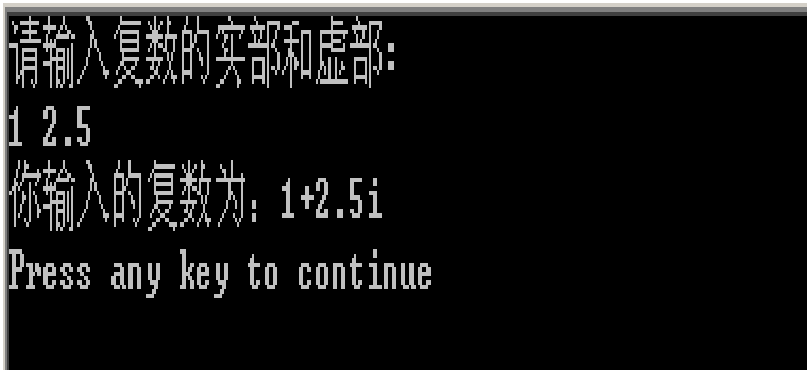


图 14.2 输出和输入操作符重载结果

因为 ComplexNumber 重载了输出和输入操作符，所以可以像内部类型那样处理类对象。在本例中为了方便定义，为 ComplexNumber 类增加了无参数构造函数。

14.3 赋值操作符

在 C++中赋值操作符可以被用于类对象之间的赋值，如果不为一个类提供赋值操作符，编译器会自动为其添加一个赋值操作符。不过编译器添加的赋值操作符只能完成数据成员值的赋值，在有指针数据成员的时候并不能很好地工作。

14.3.1 类的赋值操作

在 C++中类对象可以像普通的内部类型变量那样，用赋值符号=来完成数据的复制。赋值和复制构造函数是不同的，赋值发生在两个已有的对象之间，是用一个存在的对象的数据来设置另外一个对象，复制构造函数是用一个已经存在的对象来生成一个新对象。这两个概念对于初学者来说是不容易区分清楚的，下面用几个例子来帮助读者区分这两个概念：

```
class myClass                   // 定义类 myClass
{
    // 省略了类内部成员的定义
};
myClass c1("abc" );             // 定义对象 c1，调用构造函数
```

```
myClass c2( c1 );                    // 用 c1 创建 c2，调用复制构造函数
myClass c3 = c1;                     // 用 c1 创建 c3，调用复制构造函数
myClass c4;                          // 定义对象 c4，调用构造函数
c4 = c3;                             // 用 c1 给 c4 赋值，调用赋值函数
```

上面定义了一个简单的类 myClass，然后生成了 4 个对象，这 4 个对象的构造过程为：c1 和 c4 的比较明显，是调用构造函数；c2 调用复制构造函数；比较容易出错的是 c3，虽然使用了赋值符号=，但是这里也是用一个对象生成另外一个对象，所以也是调用了复制构造函数；最后用赋值符号给 c4 赋值，调用赋值函数。

只有在两个已经生成的对象之间才可进行赋值操作。

上面在两个已经存在的对象之间赋值，调用了赋值函数。在 C++中，赋值函数、构造函数、析构函数、复制构造函数被称为类的四大基本函数。这四个函数比较特殊，当用户没有为一个类定义这些函数的时候，编译器会自动为类添加这些函数，所以在某些情况下这些函数可以省略。

前面介绍过，类对象占用存储空间，这个空间的大小由类的数据成员来决定。也就是说类对象占用的空间是一段连续的空间，类数据成员按照定义的先后顺序在这个空间中排列。类对象的赋值过程，其实就是两块内存的复制。在前面介绍复制构造函数的时候已经提到，复制过程分为浅复制和深复制，下一节以复制构造函数为例介绍这两种复制过程的差异。

14.3.2 浅复制与深复制

浅复制指的是完全按照内存中的数据内容进行的复制行为，即无论原数据类型是什么，都忠实地将其复制到目标内存区域中。C++编译器默认给出的复制构造函数以及赋值函数执行的都是浅复制操作。例如：

```
class A
{
public:
   A(int i):m(i){}          // 定义构造函数
private:
int  m;
};
A a( 2 );                   // 定义对象
A b(a);                     // 用对象 a 生成对象 b
```

此时将把 a 内存中的数据 2 原样复制到 b 中，也就是说对象 b 的数据成员 m 的值也是 2。完成赋值之后，b 和 a 就没有任何关系了，此时一切正常。这种行为是正确的结果，但是这种赋值操

作对于指针则存在问题。

指针类型数据的内容是一个内存地址，如果原样复制意味着两个不同的指针成员指向了同一块内存区域，例如：

```
p2 = p1;
```

上面的代码使得指针 p2 只是指向了指针 p1 指向的空间，如果这个空间不存在了，那么 p2 的指向就没有任何意义了。这个时候就需要用深复制来处理。深复制指的是如果要复制的内容是指针，那么重新开辟一块内存空间，将原指针指向的内存空间的内容复制到目标指针指向的新的内存空间中。这样一来，两个指针指向的内存空间的值是一致的，但它们分别是独立的内存空间，互不干扰。

在前面定义的 String 类的复制构造函数中，就使用了深复制，下面是其代码：

```
String::String( const String & str )       // 复制构造函数
{
    int len = strlen( str.m_data );
    m_data = new char[len + 1];             // 申请数组
    strcpy( m_data, str.m_data );           // 复制数据
}
```

新生成对象的 m_data 指针指向了一个 new 出来的空间，因此新生成对象不受 str 对象的限制。由编译器为类添加的赋值操作中，默认使用的是浅复制，也就是完全按照内存中的数据进行复制。而要想执行深复制，则需要自己重载赋值操作符，下一节将介绍怎么重载赋值操作符。

14.3.3 重载赋值操作符

前面介绍了重载操作符的必要，重载操作符的语法格式如下：

```
类类型 & operator=( 参数 );
```

类类型就是要定义赋值操作符的类的类型，也就是说要返回一个本类型的引用。函数的参数列表中只能有一个参数。

赋值操作符要返回一个本类型的引用，是为了满足“a = b = c”这样的语句。

下面为前面定义的 String 类重载赋值操作符：

```
class String                                // 定义类 String
{
public:
    String & operator=( const & str );     // 重载赋值操作符
    // 省略其他成员的定义
}
```

重载赋值操作符必须定义为类的成员函数。

在这个函数的实现中，可以像复制构造函数那样为本对象重新申请一块空间来存储字符串。最

后还要返回*this 作为返回值。在前面重载的赋值操作符的参数类型还是一个类类型，这种定义可以满足下面的使用：

```
String str1;
String str2;
str1 = str2;
```

而对于一个字符串的赋值，有的时候有一些其他的赋值方法，例如：

```
String str;
str = "Hello World";           // 用一个字符串赋值
str = 'T';                     // 用一个字符赋值
```

因此要为类 String 定义另外类型参数的赋值。重载赋值操作符参数的类型没有限制，可以为任何类型。可以为一个类定义多个赋值操作符，以满足不同类型的赋值。因此应该为类 String 定义下面的赋值操作符：

```
class String                                    // 定义类 String
{
public:
    String & operator=( const & str );          // 重载赋值操作符
    String & operator=( const char * str );     // 用字符串赋值
    String & operator=( char c );                // 用字符赋值
    // 省略其他成员的定义
}
```

下面是其中一个函数的实现：

```
String & String::operator=( const String & str )
{
    if( &str == this ) return *this;   // 防止有 a = a 这样的赋值，把数据删除
    delete m_data;                     // 删除现在的指针
    int len = strlen( str.m_data );
    m_data = new char[len+1];          // 重新申请指针
    strcpy( m_data, str.m_data );
    return *this;
}
```

在上面的实现代码中，第 3 行是非常必要的。否则，如果有 a = a 这样的赋值，两个操作对象是同一个对象，就不能进行下面的操作了。因为删除了 this 对象的 m_data，str 的 m_data 也就被删除了，不能再使用了。

14.4 算术和关系操作符的重载

算术操作符和关系操作符在 C++中应用得非常广泛 ,本节介绍重载算术操作符和关系操作符的方法。

14.4.1 算术操作符的重载

在 C++中有很多算术操作符，本节以加法为例，讲述重载算术操作符的方法以及需要注意的事项。对于前面定义的 ComplexNumber 类，下面为其添加+操作符的重载：

```
ComplexNumber operator+( const ComplexNumber & c1, const ComplexNumber & c2 )
{
    ComplexNumber tmp(c1);              // 用 c1 构造临时的对象
    tmp.real += c2.real;                // 实部相加
    tmp.imaginary += c2.imaginary;      // 虚部相加
    return tmp;
}
```

加法操作符可以定义为全局函数和成员函数，不过一般情况下定义为全局函数。

上面的函数定义中将两个参数定义为 const 的引用，是因为一般情况下加法不应该修改两个数的值。函数最后返回一个局部对象，因此不能返回引用类型。如果希望不产生一个新的对象，返回已有对象的引用，可以使用复合操作符+=来实现，下面是+=的定义：

```
ComplexNumber &operator+=( ComplexNumber & c1, const ComplexNumber & c2 )
                                    // 定义+=操作符
{
    c1.real += c2.real;             // 实部相加
    c1.imaginary += c2.imaginary;   // 虚部相加
    return c1;
}
```

14.4.2 相等操作符的重载

在前面介绍操作符重载的时候，已经为 ComplexNumber 类定义了==操作符，用来判断两个类对象的数据是否相同，还可以为其添加一个!=判断操作符：

```
bool operator!=( const ComplexNumber&c1, const ComplexNumber&c2 )
                                        // 定义!=操作符
{
    return !( c1 == c2 );               // 可以直接使用==操作符
}
```

这些函数的定义并不重要，重要的是该函数所包含的设计原则：

- 如果类定义了==操作符，该操作符的含义是两个对象包含同样的数据。
- 如果类具有一个操作，能确定该类型的两个对象是否相等，通常将该函数定义为 operator==，而不是定义命名函数，用户习惯于用==来比较对象。
- 如果定义了 operator==，那么也应该定义 operator!=。
- 相等和不等操作符一般应该相互联系起来定义，让一个操作符完成比较对象的实际工作，而另一个操作符只是调用前者。

14.4.3 关系操作符的重载

定义了相等操作符的类一般也具有关系操作符。尤其是一些容器和算法经常使用小于操作符，所以定义 operator<可能相当有用。<操作符的定义和前面定义的==和!=类似，例如：

```
bool operator<( const ComplexNumber&c1, const ComplexNumber&c2 )
                // 定义<操作符
{
    return (c1.real < c2.real && c1.imaginary < c2.imaginary);
}
```

在使用<操作符的时候要慎重，因为当 a 不小于 b，b 也不小于 a 的时候，系统会判断 a==b。

14.5 其他操作符的重载

在 C++中还有一些其他的操作符，也经常使用，本节就介绍如何在 C++中重载这些操作符。在讲解每个重载运算时，都会结合具体的例子进行分析。

14.5.1 下标操作符的重载

对于数组，可以使用下标操作符[]来访问数组中的元素。同样，可以为一个类重载下标操作符来得到某个成员。考虑前面的字符串类，因为可以用下标操作符得到字符串中的字符，所以这里为其添加一个下标操作符，用来像普通的字符串那样操作。下面为 String 类添加下标操作符：

```
class String                                          // 定义 String 类
{
public:
char & operator[](int index)
{
    assert( index >=0 && index < strlen( m_data ) );   // 确保下标有效
    return m_data[index];
}
};
```

在定义下标操作符的时候，其复杂的地方在于，它在用做赋值的左、右操作数时都应该能表现正常。因此函数的返回类型必须为引用。还应该为类定义一个 const 的下标操作，用来针对 const 类对象的操作，不同的是 const 函数应该返回一个 const 的引用。

下标操作符必须定义为类的成员函数，并且只能有一个参数。

14.5.2 成员访问操作符的重载

成员访问操作符指的是解引用操作符*和箭头操作符->，这两个操作符常用在对类的封装中，比

如在定义智能指针的时候。下面用一个简单的例子来展示如何重载这两个操作符，有一个类的定义：

```
class A              // 简单地定义一个类
{
public:
void fun();          // 定义一个函数
};
```

在某些时候，为了安全起见，需要用 B 类对 A 类进行封装，不直接访问 A 类，那么想调用 A 类的 fun 函数时，有一个方法是在 B 类中也定义一个同样的函数，在这个函数中调用 A 类的方法。不过这样灵活性不大，B 类也没有通用性。这里就用到了重载箭头操作符。下面看 B 类的定义：

```
class B
{
public:
A * operator->()             // 重载箭头操作符
{
    return &a;
}
A & operator*()              // 重载解引用操作符
{
    return a;
}
A a;
};
```

定义了这两个操作符之后，看下面的应用：

```
B b;
b->fun();                    // 调用箭头操作符
```

在上面的调用中，b 是个对象，对 b 调用箭头操作符，最后返回一个 A 类的指针，所以虽然 B 类中没有定义 fun 函数，也可以对 B 类调用 fun 函数。在 C++系统设计中，经常用上面的方法来设计封装类，这个封装类被称为智能指针，这里就不再详细讨论了。

14.5.3 括号操作符的重载

可以为类类型的对象重载括号操作符，这个类可以表现为一个函数。下面看一个简单的类的定义：

```
class myFunc()                       // 定义类 myFunc，这个类可以表现为一个函数
{
public:
    bool operator()( int m, int n )// 重载括号操作符
    {
        return a < b;
    }
};
myFunc() min;
if( min( a, b ) )                    // 调用括号操作符
{
}
```

如果不看前面的定义，肯定会以为 min 是个函数而不是个类对象，这正是操作符重载的魅力。

14.6 自定义转换

在 C++中，如果编译器看到一个表达式或者函数使用了一个不合适的类型，它经常会执行一个自动类型转换。内置类型的自动转换，在前面介绍变量类型的时候已经讲过，本节将介绍关于类类型的自动类型转换。在 C++中，可以通过定义自动类型转换函数来为用户定义类型进行转换。这些函数有两种类型：特殊类型的构造函数和重载的运算符。

14.6.1 构造函数转换

如果定义一个构造函数，这个构造函数能把另一类型的对象作为其单个参数，那么这个构造函数允许编译器执行自动类型转换。如下面的例子：

```
class A              // 定义类 A
{
public:
A(int n);            // 整数类型的构造函数
};
void fun( A a );     // 该函数需要一个 A 类型的对象
fun( 1 );            // 构造函数转换
```

在这个例子中，当编译器处理 fun 以整数 1 为参数的调用时，编译器检查 fun 的定义并注意到该函数需要一个 A 类的对象。编译器发现 A 类的构造函数 A::A(int)，于是这个构造函数被调用，产生一个对象，这个对象被传递给函数 fun。

14.6.2 操作符转换

第二种自动类型转换的方法是通过运算符重载。可以为类创建一个成员函数，这个函数通过在关键字 operator 后跟随想要转换到的类型的方法，将当前类型转换为系统的类型。这种形式的操作符重载是独特的，没有指定一个返回类型，这是因为返回类型就是正在重载的运算符的名字。

下面还看前面定义的 String 类，它表示一个字符串，那么应该在很多地方可以替代一个标准的字符串来使用。比如在使用 strcpy 函数复制字符串的时候，也希望 String 类可以作为该函数的参数。下面就为 String 类添加操作符转换：

```
class String
{
public:
    operator char*()           // 定义到 char*类型的转换
    {
        return m_data;
    }
};
```

```
String s("Hello World");
char buffer[128];
strcpy( buffer, s );        // 调用字符串复制函数
```

在上面的例子中，strcpy 函数的第二个参数需要一个 char *类型的参数，但是传递了一个 String 类型的参数，因为 String 定义了到 char *的转换，所以就调用转换函数完成转换。

构造函数转换和操作符转换都可以完成类型转换。不过二者是有区别的，构造函数定义在目标类型的类中，而操作符转换定义在源类型的类中。

14.7 综合实例

在上一章的综合实例中定义了 String 类，本章陆续向这个类中添加了一些重载操作符，在本例中将把这些操作符整理一下，可覆盖大多数讲述到的操作符。在本例中为 String 类定义的函数包括：

- 构造函数。
- 重载赋值操作符。
- 重载下标操作符。
- 重载关系操作符。
- 重载转换操作符。
- 析构函数。

详细的代码如示例代码 14.3 所示。

示例代码 14.3

```
#include <iostream>
#include <assert.h>
using namespace std;
class String{
    // 重载输出操作符
    friend ostream & operator<<( ostream & out, String & str );
    // 重载关系操作符
    friend bool operator==( const String &left, const String &right );
    friend bool operator!=( const String &left, const String &right );
    friend bool operator<( const String &left, const String &right );
public:
    // 定义构造函数
    String();                           // 定义无参数构造函数
    String( const char * str );         // 定义构造函数
    String( const String & str );       // 定义复制构造函数

    // 重载赋值操作符
    String &operator=( const String & str );   // 重载赋值操作符，字符串类
    String &operator=( const char * str );     // 重载赋值操作符，常量字符串
```

```
    String &operator=( char c );                // 重载赋值操作符，字符

    // 重载[]操作符
    char & operator[]( int index );     // 重载下标操作符
    // 重载转换操作符
    operator char*();
    ~String();
private:
    char * m_data;                      // 字符指针，用来存储数据
};

String::String()                        // 无参数构造函数的实现
{
    m_data = new char[1];
    m_data[0] = '\0';
}
String::String( const char * str )      // 构造函数
{
    int len = strlen( str );
    m_data = new char[len + 1];         // 申请数组
    strcpy( m_data, str );              // 复制数据
}
String::String( const String & str )    // 复制构造函数
{
    int len = strlen( str.m_data );
    m_data = new char[len + 1];         // 申请数组
    strcpy( m_data, str.m_data );       // 复制数据
}
String & String::operator=( const String & str )
{
    if( &str == this ) return *this;    // 防止有 a = a 这样的赋值，把数据删除
    delete m_data;
    int len = strlen( str.m_data );
    m_data = new char[len+1];
    strcpy( m_data, str.m_data );
    return *this;
}
String & String::operator=( const char * str )
{
    delete m_data;
    int len = strlen( str );
    m_data = new char[len+1];
    strcpy( m_data, str );
    return *this;
}
String & String::operator=( char c )
{
    delete m_data;
    m_data = new char[2];       // 字符串中只有一个字符，长度为 2
    m_data[0] = c;
    m_data[1] = '\0';
    return *this;
```

```
}
bool operator==( const String &left, const String &right )
{
    return strcmp( left.m_data, right.m_data ) == 0;
}
bool operator!=( const String &left, const String &right )
{
    return !( left == right );
}
bool operator<( const String &left, const String &right )
{
    return strcmp( left.m_data, right.m_data ) < 0;
}
char & String::operator []( int index )
{
    assert( index >= 0 && index < strlen( m_data ) );  // 确保下标有效
    return m_data[index];
}
String::operator char*()
{
    return m_data;
}
String::~String()                  // 析构函数
{
    delete m_data;                 // 释放申请的内存
}
ostream & operator<<( ostream & out, String & str )    // 重载输出
{
    out<<str.m_data;
    return out;
}

int main()
{
    cout<<"赋值操作符的使用!"<<endl;
    String s1("Hello");            // 用字符串初始化
    String s2;                     // 无初始化参数
    s2 = s1;                       // 用类对象赋值
    cout<<s2<<endl;
    s2 = "Hello World";
    cout<<s2<<endl;                // 用常量字符串赋值
    s2 = 'a';                      // 用字符赋值
    cout<<s2<<endl;
    cout<<"小标操作符的使用"<<endl;
    s2 = "Hello World";
    cout<<"变化之前为:"<<s2<<endl;
    s2[1] = 'E';
    cout<<"变化之后为:"<<s2<<endl;
    cout<<"操作符转换:"<<endl;
    char buffer[128];
    strcpy( buffer, s2 );
    cout<<"buffer="<<buffer<<endl;
```

```
    return 0;
}
```

建立一个控制台工程以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 14.3 所示。

图 14.3　String 类的完整定义结果

14.8 小结

操作符重载的目的是使编程更加容易。通过定义重载操作符，可以为自定义类型的对象定义丰富的表达式集合。操作符重载就像定义名字有趣的函数，当操作符出现时，编译器会调用对应的函数。操作符重载中形参的个数由两个因素决定：操作符是一元的还是二元的，操作符被定义为全局函数还是成员函数。当操作符被定义为全局函数的时候，通常被声明为类的友元。

当类有 I/O 操作的时候，通常会为 I/O 操作定义输出和输入操作符。还可以重载算术操作符和关系操作符。可以通过重载下标操作符来得到对象的子元素。通过重载括号操作符，可以像调用函数那样调用类。通过重载箭头操作符可以实现对类的封装。

类可以自定义转换，当一个类型的对象用在需要另外不同类型对象的时候，可以自动应用这些转换。

◆　　◆　　◆

第 15 章　类 的 继 承

本章包括

- 如何确定基类和派生类
- 各种继承类型
- 选择继承方式
- 派生类对象的内存布局
- 派生类的构造和析构
- 使用基类成员

类的继承是面向对象的重要特征。继承一方面使得基类和派生类之间建立起逻辑上的层次关系；另一方面也可以使一个派生类获得其基类的属性和行为，开发者只需根据需求在派生类中增加特殊的属性和行为即可。本章的主要内容就是讲述如何在 C++中实现类的继承。

15.1　确定类的层次

确定类的层次应当遵从一般到特殊的关系。例如对于一系列有关运输工具的类，可以建立如图 15.1 所示的层次结构。

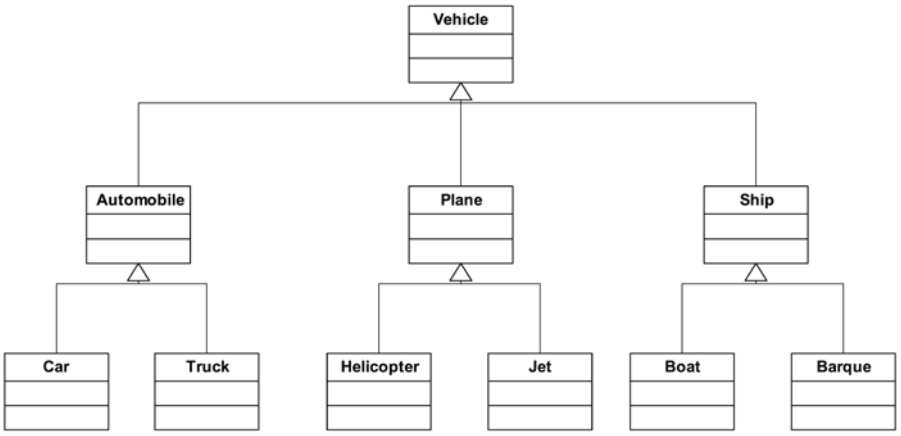


图 15.1　运输工具的类层次结构

Vehicle（运输工具）是一个基类（也称为父类），其中定义所有交通工具都具有的属性和行为，包括速度、载重量、运行等。按照应用的领域不同，交通工具还可以细分为 Automobile（汽车）类、Plane（飞机）类和 Ship（船）类等。细分出来的这些类称为派生类（也称为子类）。派生类同样具有速度、载重量、运行等属性和行为，而这些属性和行为都是从 Vehicle 类继承来的，不必再重新定义。所以，在定义 Automobile，Plane 和 Ship 等类时，只需定义那些它们各自专有的属性和行为即可。

比如，Automobile 类相比 Vehicle 类有轮子、安全气囊等特殊属性，以及转动车轮等特殊行为。Plane 类相比 Vehicle 类有雷达、黑盒子等特殊属性，以及飞行等特殊行为。依此类推，层层递增，就可以建立起完整的运输工具类的层次结构。

确定类的层次结构在分析和设计阶段完成，编程时需要通过程序体现上述层次结构。实现的方法是继承。通过继承，明确了类与类之间的逻辑关系，也使得派生类具有了基类的属性和行为，而不必重复定义。

15.2 继承的类型

在 C++中定义类之间的继承比较简单，只要在普通类定义的类名后面加一个冒号，并加上继承方式和基类名即可。其语法如下：

```
class  派生类名 ： 继承方式  基类名
{
    派生类成员
};
```

其中的继承方式有三种，分别是 public（公有）、protected（保护）和 private（私有）。这三种继承方式也决定了派生类的成员函数对基类成员的可访问性，以及派生类对象对基类成员的可访问性。下面将分别讲述这三种继承方式。

如果不写继承方式，编译器自动将其当做私有继承。

15.2.1 公有继承

公有继承的关键字是 public。例如 Automobile 类继承 Vehicle 类是公有继承，其定义如下：

```
class  Automobile  :  public  Vehicle
{
    Automobile 成员定义
};
```

公有继承时，派生类的成员函数可以访问基类中的公有成员和保护成员，不可以访问其私有成员。通过派生类对象只可访问基类中的公有成员，不能访问其他成员。也就是说，通过公有继承，基类中各个成员的原有属性，在派生类中依然保持，公有的依然是公有的，保护的依然是保护的，私有的依然是私有的。基类和派生类的关系如图 15.2 所示。

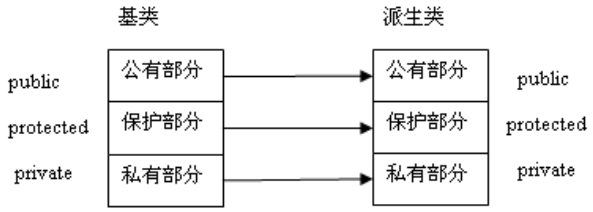


图 15.2　基类与派生类的关系

在下面的例子中，Derived 类公有派生于 Base 类。该例展示了在派生类中访问基类的成员以及通过派生类对象访问基类成员的情况。

```
class  Base                  // 基类
{
public:                      // 公有成员
    int  m_a;
    void  f1();
protected:                   // 保护成员
    int  m_b;
    void  f2();
private:                     // 私有成员
    int  m_c;
    void  f3();
};
class  Derived  :  public  Base  // 派生类
{
public:
    void  f4()
    {
        m_a;                 // 正确，公有继承，m_a 是基类公有的，可以被派生类访问
        f1();                // 正确，公有继承，f1()是基类公有的，可以被派生类访问
        m_b;                 // 正确，公有继承，m_b 是基类保护的，可以被派生类访问
        f2();                // 正确，公有继承，f2()是基类保护的，可以被派生类访问
        m_c;                 // 正确，公有继承，m_c 是基类私有的，可以被派生类访问
        f3();                // 正确，公有继承，f3()是基类私有的，可以被派生类访问
    }
};
int main(int argc, char *argv[])
{
    Derive d;                // 派生类对象
    d.ma;                    // 正确，公有继承，m_a 是基类公有的，可以被外界访问
    d.f1();                  // 正确，公有继承，f1()是基类公有的，可以被外界访问
    /*
    d.mb;                    // 错误，公有继承，m_b 是基类保护的，不能被外界访问
    d.f2();                  // 错误，公有继承，f1()是基类保护的，不能被外界访问
    d.mc;                    // 错误，公有继承，m_c 是基类私有的，不能被外界访问
    d.f3();                  // 错误，公有继承，f3()是基类私有的，不能被外界访问
    */
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

公有派生使得派生类完全支持基类的方法，即通过派生类对象，可以访问基类所有的公共属性和方法。所以，公有派生使得类可以满足面向对象的替换原则，即所有使用基类对象的地方，都可以用派生类对象替换。

15.2.2 私有继承

私有继承的关键字是 private。例如 Car 类从 Automobile 类私有继承，其定义如下：

```
class  Car  :  private  Automobile
{
    Car 类成员定义
};
```

私有继承时，基类的成员在直接派生类中都变成了私有的，只能由直接派生类的方法访问，而无法被外界访问。因此，经过私有派生后，基类成员无法再继续被继承，如图 15.3 所示。

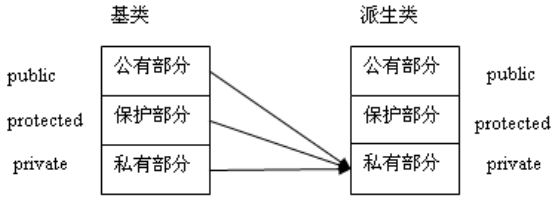


图 15.3　私有派生时基类与派生类的关系

对于同样定义的 Base 类，私有继承时的情况如下：

```
class  Derived  :  private  Base // 派生类
{
public:
    void  f4()
    {
        m_a;                // 正确，私有继承，m_a 是基类公有的，可以被派生类访问
        f1();               // 正确，私有继承，f1()是基类公有的，可以被派生类访问
        m_b;                // 正确，私有继承，m_b 是基类保护的，可以被派生类访问
        f2();               // 正确，私有继承，f2()是基类保护的，可以被派生类访问
        /*
        m_c;                // 错误，私有继承，m_c 是基类私有的，不可以被派生类访问
        f3();               // 错误，私有继承，f3()是基类私有的，不可以被派生类访问
        */
    }
};
int main(int argc, char *argv[])
{
    Derive d;               // 派生类对象
    /*
    d.ma;                   // 错误，私有继承，基类的成员变成私有的，不可以被外界访问
    d.f1();                 // 错误，……

    d.mb;                   // 错误，……
    d.f2();                 // 错误，……
    d.mc;                   // 错误，……
    d.f3();                 // 错误，……
    */
    system("PAUSE");
```

```
    return EXIT_SUCCESS;
}
```

私有继承时，基类所有的公共属性和方法都不再被派生类支持，即通过派生类对象已不能访问基类的成员。所以，通过私有派生，程序已不再满足面向对象的替换原则，即所有用到基类对象的地方，不能被私有继承的派生类对象替换。

15.2.3 保护继承

保护继承的关键字是 protected。例如 Truck 类继承 Automobile 类是保护继承，其定义如下：

```
class  Truck  :  protected  Automobile
{
    Automobile 成员定义
};
```

保护继承时，基类的所有公有成员和保护成员都作为派生类的保护成员，并且只能被派生类的成员函数或友元访问。派生类也只能访问基类的公有和保护成员，而不能访问基类的私有成员。保护派生时基类与派生类的关系如图 15.4 所示。

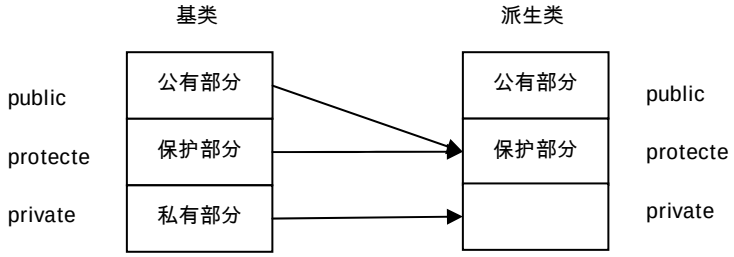


图 15.4 保护派生时基类与派生类的关系

对于同样定义的 Base 类，保护继承时的情况如下：

```
class  Derived  :  protected  Base   // 派生类
{
public:
    void  f4()
    {
        m_a;                // 正确，保护继承，m_a 是基类公有的，可以被派生类访问
        f1();               // 正确，保护继承，f1()是基类公有的，可以被派生类访问
        m_b;                // 正确，保护继承，m_b 是基类保护的，可以被派生类访问
        f2();               // 正确，保护继承，f2()是基类保护的，可以被派生类访问
        /*
        m_c;                // 错误，保护继承，m_c 是基类私有的，不可以被派生类访问
        f3();               // 错误，保护继承，f3()是基类私有的，不可以被派生类访问
        */
    }
};
int main(int argc, char *argv[])
```

```
{
    Derive d;                // 派生类对象
    /*
    d.ma;                    // 错误，保护继承，基类的成员变成保护的，不可以被外界访问
    d.f1();                  // 错误，……

    d.mb;                    // 错误，……
    d.f2();                  // 错误，……
    d.mc;                    // 错误，……
    d.f3();                  // 错误，……
    */
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

同私有继承类似，保护继承的派生类也不再支持基类的公共方法，从而也不再满足面向对象的替换原则。

15.3 选择继承方式

从上一节的讲解中，读者可以看出三种继承方式的主要区别在于派生类对基类各种成员的可访问性，以及通过派生类对象是否可以访问基类的成员。继承方式与基类成员的属性如表 15.1 所示。

表 15.1 继承方式与基类成员的属性

继承方式	基类成员性质	派生类成员性质	派生类访问基类成员	通过派生类对象访问基类成员	可否继续派生
public	public	public	○	○	○
	protected	protected	○	×	○
	private	private	×	×	×
protected	public	protected	○	×	○
	protected	protected	○	×	○
	private	private	×	×	×
private	public	private	○	×	×
	protected	private	○	×	×
	private	private	×	×	×

从对上表的分析可以看出各种派生类具有如下特点：

- 只有公有继承才能通过派生类对象访问基类的公有成员。
- 私有继承可以中断派生链，使得基类成员只为直接派生类所用。
- 保护继承既可以使得基类成员不为外界所见，又可以使得基类成员为后续的派生类所用。

开发者在设计一个类的层次结构时，应当根据以上的特点选择合适的继承方式。一般来讲，应当遵循如下的原则：

- 为了满足面向对象的替换原则，应当尽量使用公有继承。
- 如果派生类只是重用基类的属性和行为，则可以使用私有继承。
- 如果不仅仅是直接派生类重用基类的属性和行为，而是以后的派生类也可以用，则应当使用保护继承。

例如，对于 Automobile 类和 Car 类，Automobile 是对所有汽车类的一种抽象，其所具有的所有公共属性和方法都应该被轿车（即 Car）类支持。而且，所有用到 Automobile 对象的地方，都应该可以用 Car 对象代替。所以 Car 类应当公有继承 Automobile 类。其定义如下：

```
class  Automobile                    // 汽车基类
{
public:
    char  *m_id;                     // 车牌号
    void  drive();                   // 行驶
};
class  Car  :  public  Automobile    // 轿车类
{
    ……
};
```

私有继承适用于需要隐藏基类公有成员的情况，例如企鹅类（Penguin）继承鸟类（Bird）时的情况，Bird 类定义了 Fly()这个公有方法，企鹅也是鸟类的一种，所以 Penguin 类应当从 Bird 类继承，但企鹅不会飞，而会游泳。可以改变一些相关条件，使得企鹅游泳就像是在水中“飞”，从而可以利用 Fly()方法。但又不能对外公开这个 Fly()方法，所以应当使用私有继承。其定义如下：

```
class  Bird                      // 鸟类
{
public:
    void  Eat();                 // 进食
    void  drive();               // 飞翔
};
class  Penguin  :  private  Bird // 企鹅类
{
public:
    void  Swim()                 // 改变条件，使得企鹅可以在水中“飞”
    {
        ……
        Fly();                   // 使用 Bird 类的方法
        ……
    }
};
```

显然，私有继承有些绝对，将基类所有的方法都置为私有，以至于除了直接派生类，以后的派生类都不能访问基类的成员。保护继承可以改变这种极端的情况，使得基类的公有和保护成员可以

继续为以后的派生类所用，但又不至于暴露给外界。例如，对于上述的企鹅类，如果再继续细分，比如由企鹅类再派生一个帝企鹅类，而帝企鹅也要重用 Bird 类的 Fly()方法，则企鹅类就应当使用保护继承，定义如下：

```
class  Penguin  :  protected  Bird       // 企鹅类
{
public:
    void  Swim()                         // 改变条件，使得企鹅可以在水中“飞”
    {
        ……
        Fly();                           // 使用 Bird 类的方法
        ……
    }
};
class  EmperorPenguin  :  public  Penguin    // 帝企鹅类
{
public:
    void Swim2()                         // 帝企鹅类的 Swim2()方法
    {
        ……
        Fly();                           // 使用 Bird 类的方法
        ……
    }
};
```

对于公有继承，因为派生类支持基类的所有公有属性和方法，所以公有继承也被称为接口继承；而私有继承和保护继承只是利用属性和方法，所以这两种继承又被称为实现继承。

15.4 派生类对象的内存布局

派生类继承了基类的属性和行为，在派生类对象中，基类的属性数据也将作为其中的一部分而存在。在对象的内存布局上，首先是基类的数据，然后才是派生类的数据。在派生类对象中的基类部分也被称做基类子对象，如图 15.5 所示。

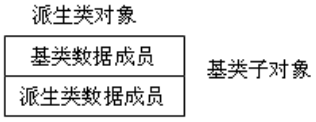


图 15.5　派生类对象的内存布局

例如，在下面的例子中，Derived 类继承 Base 类，其定义如下：

```
class  Base
{
public:
    int  m_a;
```

```
    int  m_b;
};
class  Derived  :  public Base
{
public:
    int  m_c;
    int  m_d;
};
```

Base 类和 Derived 类对象的内存布局如图 15.6 所示。

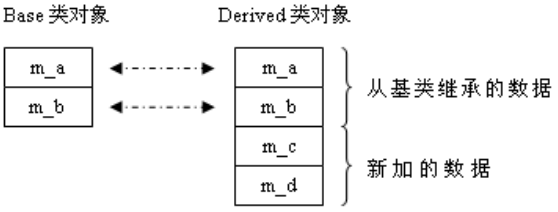


图 15.6 Base 类和 Derived 类对象的内存布局

在对象的内存中，只有非静态的数据成员。如果基类中有静态的数据成员，那么在派生类中这个成员的静态属性将被保留，即成为派生类的一个静态成员。

15.5 将派生类对象转换为基类对象

按照面向对象的替换原则，程序中使用基类对象的地方也能够使用派生类对象。在前面小节的学习中，读者可以看到公有继承方式可以满足这个原则。之所以可以，是因为在派生类对象和基类对象之间存在一个隐式转换关系。也就是说，如果程序需要一个基类对象，而实际提供的是一个派生类对象，则编译器会将派生类对象自动转换成基类对象。例如，声明了接受基类对象作为参数的函数，但传递过来的实参却是派生类对象：

```
void function( Base  obj )      // 接受基类对象作为参数的函数
{
    ……
}
int main(int argc, char *argv[])
{
    Derived  d;                 // 派生类对象
    function( d );              // 将派生类对象作为实参传递给需要基类对象实参的函数
}
```

在参数传递过程中，编译器会将作为实参的派生类对象转换成一个基类对象。这相当于在函数调用时将 Derived 的对象 d 强制转换成 Base 对象，所以 function 函数调用时也可以写成如下的形式：

```
function( (Base) d );
```

派生类对象到基类对象的转换，是按照基类对象的内存布局剪裁派生类对象的内存布局的。因为派生类添加的新成员排列在基类数据成员的后面，所以在转换时，只要从派生类对象的起始地址开始，划一块大小等于基类的内存，并标记为一个基类对象即可，如图 15.7 所示。

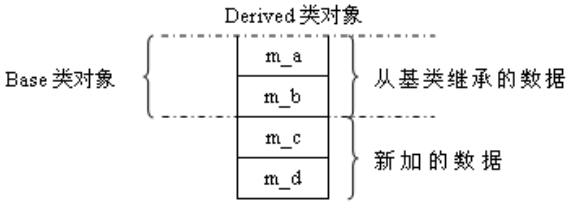


图 15.7　派生类对象到基类对象的转换

15.6　派生类的构造和析构

通过继承，基类变成了派生类的一部分。也就是说，在派生类对象中也有基类对象的成分（基类子对象）。因此派生类也应当负责其基类部分的构造和析构。

15.6.1　构造派生类

在 C++中，构造函数不能被继承，派生类必须重新定义自己的构造函数。在派生类的构造函数中，必须完成两件事情，一是基类子对象的构造，二是派生类新添加成员的初始化。不过这两个步骤并不是在派生类构造函数内部完成的，而是在构造函数的初始化列表里完成的。

定义一个派生类的构造函数时，应当在初始化列表中依次写上基类的构造函数和成员的初始化参数。特别是在基类和数据成员的构造函数带有参数的情况下，其参数只能由开发者指定，所以必须显式地初始化。其语法如下：

```
派生类构造函数名( 参数列表 )
    :  基类构造函数名( 参数列表 ),
       派生类成员初始化
{
    ……
}
```

假如基类具有如下的构造函数：

```
Base( int a );
```

派生类具有如下的数据成员：

```
int  m_e;
```

则派生类的构造函数应当如下定义：

```
Derived( int  x )
    :  Base( x ),
       m_e( x )
{
}
```

如果基类具有默认构造函数，即不带参数的构造函数，那么在定义派生类的构造函数时也可以不显式地调用基类的构造函数。如果派生类新增数据成员的类型是

内置的数据类型，或者是具有默认构造函数的类，那么编译器也会自动添加数据成员的初始化代码。

值得指出的是，在派生类构造函数的初始化列表中，程序总是首先调用基类的构造函数，然后是各个成员的初始化。并且，成员的初始化是按照其定义的顺序进行的。派生类的构造函数在初始化列表完成后调用。所以，派生类的构造总是按照下面的顺序进行：

step 1 基类的构造函数。

step 2 派生类成员的初始化。

step 3 派生类的构造函数。

例如，下面派生类构造函数的初始化列表虽然被打乱了，但是程序仍然会按照上述顺序进行初始化，而不是按照开发者所期待的那样执行。

```
class Data                                  // 某个数据类
{
public:
    Data( int x )                           // 数据类的构造函数
    {
        cout<< "构造 Data: "<<x<<endl;      // 输出数据类的构造信息
    }
};
class Base                                  // 基类
{
public:
    Base(int a)                             // 基类构造函数
    {
        cout<<"构造 Base: "<<a<<endl;       // 输出基类构造信息
    }

};
class Derived : private Base    // 派生类
{
public:
    Derived( int x )            // 派生类构造函数
        :  m_b(x++),            // 故意打乱的初始化列表，开发者期待 m_b 先被初始化
          Base(x++)
    {
        cout<<"构造 Derived: "<<x<<endl;     // 输出派生类构造信息
    }
private:
    Data m_b;                               // 数据成员
};
32
int main(int argc, char *argv[])            // 主函数
{
    Derive d( 1 );                          // 构造派生类对象
    system("PAUSE");
```

```
    return EXIT_SUCCESS;
}
```

上述程序将输出：

```
构造 Base: 1
构造 Data: 2
构造 Derived: 3
```

在 Derived 构造函数的初始化列表里，尽管数据成员 m_b 的初始化写在了基类构造函数 Base(int x)的前面，但在实际运行过程中，仍然是基类构造函数先被调用，然后才进行数据成员的初始化。

尽管有的时候不需要显式地初始化基类子对象和数据成员，但其默认的构造函数仍然会在编译时加到派生类构造函数的初始化列表中。

15.6.2 析构派生类

类的析构函数也不能被继承，所以派生类也需要有自己的析构函数。而且在派生类的析构函数中也需要析构基类子对象以及新增的数据成员。与构造函数不同的是，由于析构函数没有参数，所以基类和新增数据成员的析构可以由编译器代劳，即在编译时由编译器插入相关的析构代码。派生类的析构与构造顺序刚好相反，其顺序是：

step 1 调用派生类的析构函数。

step 2 析构新增数据成员。

step 3 析构基类子对象。

开发者不能影响这个过程，例如在上一节的例子的各个类中分别加入下面的析构函数：

```
Data::~Data()
{
    cout<<"析构 Data"<<endl;
}
Base::~Base()
{
    cout<<"析构 Base"<<endl;
}
Derived::~Derived()
{
    cout<<"析构 Derived"<<endl;
}
```

则程序会输出：

```
构造 Base: 1
构造 Data: 2
构造 Derived: 3
析构 Derived
析构 Data
```

析构 Base

除了构造函数和析构函数不能被继承外，类中的相等运算符（==）也不能被继承，友元也不能。如果相等运算符可以被继承，那么当比较两个派生类对象时，被比较的只是其基类子对象的成员，显然不可行。友元不可以被继承，是因为友元只是类之间的“私人”关系，不能要求两个家族的祖先是朋友，则其子孙也是朋友，显然那样是不符合逻辑的。

下例定义一个二维图形基类，并从中派生出一个矩形类和一个三角形类，并分别计算出各自的中心点坐标，类之间的关系如图 15.8 所示。

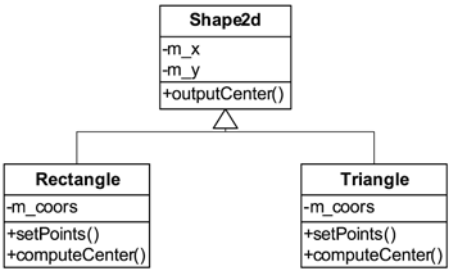


图 15.8 二维图形类的层次关系

其中，中心点定义在基类 Shape2d 中，是每个二维图形都有的属性。Shape2d 类还负责输出中心点坐标。由于各个派生类计算中心点的方法不一样，所以将该算法定义在派生类中。下面的代码中只给出了 Rectangle 类的定义，Triangle 类的定义与其类似，读者可以作为练习，自己定义。

```
#include <cstdlib>
#include <iostream>
using namespace std;                                    // 使用名称空间 std
class Shape2d                                           // 图形基类
{
public:
    Shape2d(double x, double y)                         // 基类的构造函数
        :m_x( x ), m_y( y )                             // 初始化列表
    {
    }

    void outputCenter()                                 // 输出中心点的坐标
    {
        cout<<m_x<<'\t'<<m_y;
    }

protected:
    double m_x, m_y;                                    // 中心点坐标
};
class Rectangle : public Shape2d                        // 矩形类，继承 Shape2d 类
{
public:
```

```
    Rectangle()                                          // 矩形类的构造函数
        : Shape2d(0, 0)                                  // 初始化 Shape2d 类子对象
    {
        memset( m_coors, 0, 8 * sizeof(m_coors[0]) );// 调用库函数，设置点坐标为 0
    }

    void setPoints( double coors[] )                     // 设置点坐标的函数
    {
        memcpy( m_coors, coors, 8 * sizeof(m_coors[0]) );// 调用库函数，设置点坐标
    }

    void computeCenter()                                 // 计算中心点的函数
    {
        m_x = 0.25 * ( m_coors[0] + m_coors[2] +         // 中心点的 x 坐标
                       m_coors[4] + m_coors[6] );
        m_y = 0.25 * ( m_coors[1] + m_coors[3] +         // 中心点的 y 坐标
                       m_coors[5] + m_coors[7] );
    }

private:
    double m_coors[8];                                   // 矩形 4 个顶点的坐标
};
int main(int argc, char *argv[])                         // 主函数
{
    cout<<"——中心点坐标——"<<endl;
    Rectangle rect;                                      // 矩形类的对象
    double coors[8] = { 10, 10, 20, 10,                  // 坐标
                  20, 20, 10, 20, };
    rect.setPoints( coors );                             // 设置坐标
    rect.computeCenter();                                // 计算中心点
    rect.outputCenter();                                 // 输出中心点
    cout<<endl;
   system("PAUSE");                                      // 等待用户反应
   return EXIT_SUCCESS;                                  // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 15.9 所示。

图 15.9　计算图形中心点结果

从图形类的语义角度出发，setPoints()函数和 computeCenter()函数也应当定义在基类 Shape2d 中，因为这两个函数代表的行为是所有二维图形都具有的。但是，不同的图形类又有不同的实现，所以又不得不在派生类中实现。不过，这个矛盾可以通过对象的多态解决，这将在以后的章节中讲述。

15.7 使用基类成员

有的时候由于基类的成员过多，开发者难以了解其全部细节，从而在构建派生类时新增的成员有可能跟基类的某个成员同名。或者出于某种设计的目的，开发者故意增加一个同名成员。此时应当注意如何使用基类成员。

如果派生类的某个成员同基类中的某个成员同名，实际将导致在派生类中不能直接使用基类中的那个成员。也就是说派生类中的成员将基类成员隐藏了，或者说覆盖了。

例如，在下面的 Base 类和 Derived 类中都定义了数据成员 m_a 以及函数成员 f1()，但在定义派生类的成员函数 f2()时，只能使用派生类中定义的版本。

```
class Base
{
public:
    int  f1( int  x );      // 基类中定义 f1()函数
private:
    int  m_a;               // 基类中定义的数据成员 m_a
};
class Derived  :  public  Base
{
public:
    void  f1();             // 派生类中的 f1()函数，与基类中的函数同名，但返回类型不同
    void  f2()
    {
        f1();               // 正确，调用 Derived 类中定义的版本
        // int  x  =  f1( 123 );// 错误，试图调用基类中的版本，但其不在派生类的作用域内
        m_a = Data();       // 正确，调用 Derived 类中定义的版本
        // m_a = 123;       // 错误，试图使用基类中的版本，但其不在派生类的作用域内
    }
private:
    Data  m_a;              // 派生类中的 m_a 数据，与基类中的数据成员同名，但类型不同
};
```

为了改变这种状况，可以用域运算符（::）明确指明某个成员所属的类，其语法是在类名后面加上域运算符，再加上成员的名字。比如为了在定义 Derived 类的成员函数 f2()时，使用基类中的成员，可以如下定义：

```
void  Derived:: f2()        // 定义 Derived 类的 f2()函数
{
    f1();                   // 调用 Derived 类的 f1()函数，也可以写为 Derived::f1()
    int  x  =  Base::f1( 123 ); // 使用 Base 类中的 f1()
    m_a  =  Data();         // 使用 Derived 类的 m_a，也可以写为 Derived::m_a
    Base::m_a  =  123;      // 使用 Base 类中的 m_a
}
```

初学者经常会误以为派生类和基类中的同名函数是重载的关系，实际上这是错误的。因为重载的两个函数首先要处在同一作用域内。基类和派生类是两个不同的作用域，所以派生类的成员函数只能覆盖基类的成员函数，而不能重载。

如果要重载基类中的函数，则在定义派生类时，应当将该函数引入到派生类中，其语法是使用 using 关键字，然后用域运算符指明要重载的函数。比如，为了重载 Base 类中的 f1()函数，Derived 类应当如下定义：

```
class  Derived  :  public  Base      // 定义派生类
{
    using  Base::f1;                // 将基类的成员函数引入到派生类的作用域中
public:
    void  f1();                     // 与基类的 f1()函数重载
    void  f2()                      // 定义函数 f2()
    {
        int  x = f1( 123 );         // 经重载解析，实际调用 Base 类中的 f1()函数
    }
};
```

基类和派生类中定义同名函数有一种特殊情况是虚函数。定义虚函数是为了实现多态，即同一行为不同的类有不同的表现，这将在后面的章节中进行讲述。

15.8 基类类型的指针和引用

通常程序中会有这样的需求：用数组存储一组对象，但对象的类型并不相同，而是属于某个类的不同层次。实际上一个数组只能存储一种类型的数据，所以数组不能直接用来存储上述对象。但可以用一种间接的方法解决这个问题，即利用基类类型的指针和引用。

基类类型的指针和引用可以指向一个基类对象，也可以指向其派生类对象，甚至是多级派生后的对象。并且，基类指针中的地址就是对象的首地址，如图 15.10 所示。

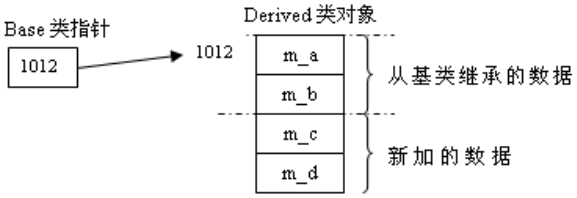


图 15.10　基类指针指向派生类对象

在程序中让一个基类指针指向派生类对象，与让一个派生类指针指向派生类对象没有什么不同，都是定义一个指针，并用取地址运算符（&）获得对象的地址后初始化或赋值给指针即可。例如，用 Base 类指针指向一个 Derived 类对象，可以如下操作：

```
Derived  obj;                   // 派生类对象
Base  *pObj  =  &obj;           // 用基类指针指向派生类对象
```

```
Base *pObj2 = new Derived();
```

用基类类型的引用指向一个派生类对象，则可以如下写：

```
Derived obj;                    // 派生类对象
Base &rObj = obj;               // 用基类引用指向派生类对象
```

基类类型的指针和引用也可以转换成派生类对象的指针或引用。当然前提是该指针或引用本来就指向一个派生类对象，否则虽然转换可以成功进行，但实际上却是非法的。通过这样的派生类指针访问新增数据成员时，实际访问的却是未曾分配给这个对象的内存，从而会导致非法操作。

只有公有继承才可以用基类类型的指针或引用指向派生类对象。私有继承和保护继承，由于其基类子对象部分实际上是不对外公开的，所以通过基类类型的指针指向这样的对象将导致基类子对象暴露给外界，而这是不允许的。

下例用数组保存职员类对象的指针，并针对不同类型职员分别调用不同的计算薪水函数。各种职员的类如图 15.11 所示。

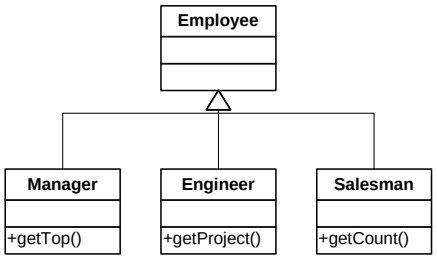


图 15.11 各种职员的类图

程序代码如下：

```
#include <cstdlib>
#include <iostream>
using namespace std;
class Employee                                  // 职员基类
{
};
class Manager : public Employee                 // 管理者类
{
public:
    void getTop()                               // 管理者的函数
    {
        cout<<"Manager Top"<<endl;
    }
};
class Engineer : public Employee                // 工程师类
{
public:
    void getProject()                           // 工程师的函数
    {
        cout<<"Engineer Project"<<endl;
    }
};
```

```
class Salesman : public Employee                // 销售人员类
{
public:
    void getCount()                             // 销售人员的函数
    {
        cout<<"Salesman Count"<<endl;
    }
};
int main(int argc, char *argv[])                // 主函数
{
    cout<<"——基类类型的指针——"<<endl;
    Manager  a;                                 // 各种派生类对象
    Engineer  b;
    Salesman  c;
    Employee *pObj[3] = { &a, &b, &c };         // 基类指针数组

    Manager *pA = static_cast<Manager*>( pObj[0] );    // 转换成派生类指针
    pA->getTop();                               // 调用派生类函数

    Engineer *pB = static_cast<Engineer*>( pObj[1] );
    pB->getProject();

    Salesman *pC = static_cast<Salesman*>( pObj[2] );
    pC->getCount();

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 15.12 所示。

图 15.12　使用基类指针数组结果

在上面的程序中，通过数组管理不同派生类的对象。在使用过程中，可以先将基类类型的指针转换成派生类类型的指针，再调用派生类的成员。

15.9 综合实例

在自然界中有各种各样的动物，按照动物的属性可以对其进行分类。例如基本的分类有哺乳动物、爬行动物、鸟类、昆虫等。对于哺乳动物按照其食性可分为食草动物和食肉动物。食肉动物下又有各种分类，如猫科动物、犬科动物等。假设要做一个动物园的管理系统，那么就有必要按照上述的分类方法建立动物的类层次结构。建立动物的类层次结构关系如图 15.13 所示。

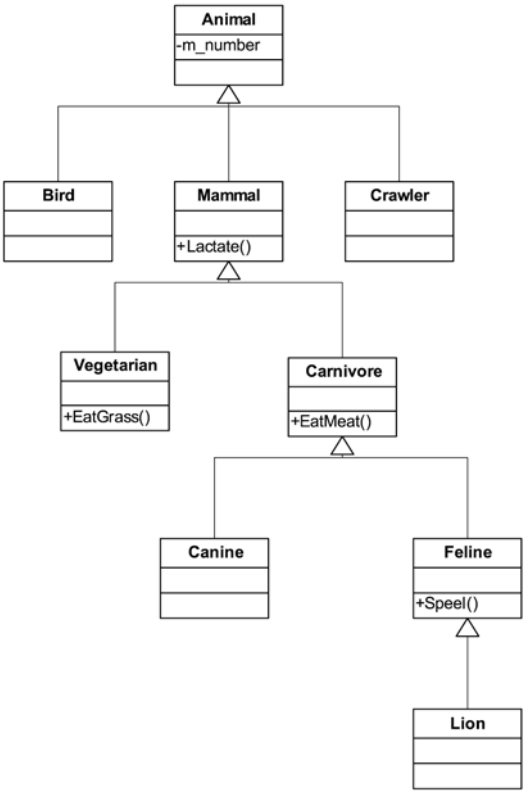


图 15.13 动物的类层次结构

程序代码如下：

限于篇幅，这里只列出了部分类的定义。

```
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;                    // 使用名称空间 std
class Animal                            // 动物类
{
public:
    string m_name;                      // 动物的名称属性
};
class Mammal : public Animal            // 哺乳动物类
{
public:
    void Lactate()                      // 哺乳行为
    {
        cout<<"哺乳"<<endl;
    }
};
```

```
class Carnivore : public Mammal          // 食肉动物类
{
public:
    void EatMeat()                       // 食肉行为
    {
        cout<<"吃肉"<<endl;
    }
};
class Feline : public Carnivore          // 猫科动物类
{
public:
    void Speel()                         // 爬树行为
    {
        cout<<"爬树"<<endl;
    }
};
class Lion : public Feline               // 狮子类
{
};
int main(int argc, char *argv[])         // 主函数
{
    cout<<"——动物的类层次——"<<endl;          // 输出提示信息
    cout<<endl;

    Lion lion;                           // 定义狮子对象
    lion.m_name = "狮子";                 // 设置狮子对象的名字
    cout<<"\""<<lion.m_name<<"\"可以"<<endl;// 输出提示信息

    lion.Lactate();                      // 调用狮子类从基类继承的方法
    lion.EatMeat();
    lion.Speel();

    cout<<endl;
    system("PAUSE");                     // 等待用户反应
    return EXIT_SUCCESS;                 // 主函数退出
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 15.14 所示。

图 15.14　输出狮子的行为结果

15.10 小结

本章主要讲述了在C++中如何定义类之间的继承关系,以及派生类在构造和析构时要注意的问题。另外还讲了如果派生类和基类存在重名的成员,如何使用基类的成员。需要注意的是,本章讲的继承是单继承,即一个派生类只有一个基类,实际上C++也支持为一个派生类定义多个基类,即多继承,这将在下一章讲述。

◆ ◆ ◆

第 16 章　多继承和虚拟继承

本章包括

- 多继承的定义及用法
- 多继承存在的问题
- 虚拟继承的语法及用法
- 虚拟继承的缺点

C++也支持多继承，即一个派生类有多个基类。目前，在是否使用多继承上存在很多争论。有的人认为多继承的功能很强大，可以使用一个派生类支持多个基类；也有人认为多继承使得派生类存在二义性，而且使程序变得复杂。虚拟继承是多继承的一种，使用它目的是为了避免多继承带来的二义性。本章的主要内容就是来探讨一下多继承和虚拟继承。

16.1　为什么要用多继承

在大多数情况下，单继承已经能够满足开发者的需要。但如果事物的逻辑层次结构稍显复杂，单继承就未必能够胜任。比如在某个类层次结构中有两个类 A 和 B，现在要创建一个新类 C，而 C 同时具有 A 和 B 的属性和行为。显然，此时单继承难以满足要求。在程序中描述动物的分类时，就会遇到这种情况，特别是在描述鸭嘴兽时。鸭嘴兽是一种很奇怪的动物，如图 16.1 所示。

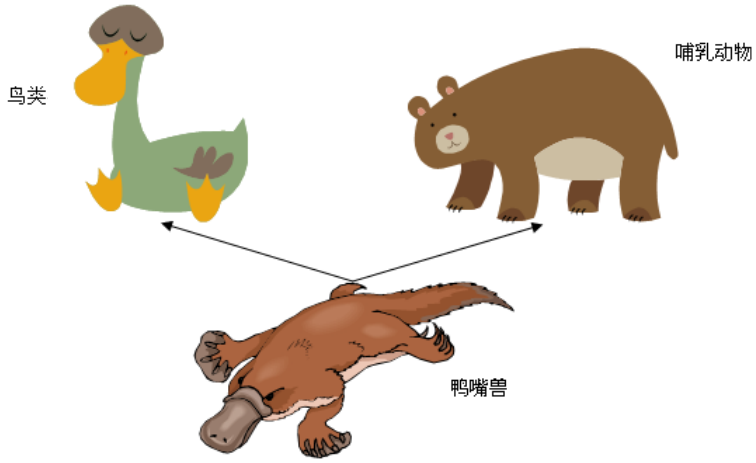


图 16.1　奇怪的鸭嘴兽

从形态学上来讲，鸭嘴兽应该属于鸟类，原因是鸭嘴兽具有扁平的、像鸭子一样的嘴巴，而且是角质的，不像哺乳动物那种肉质的口唇，关键是鸭嘴兽通过下蛋来繁殖后代，这明显是鸟类的特征。然而，鸭嘴兽也靠乳汁来哺育幼仔，浑身密布着浓褐色的短兽毛，这又是哺乳动物的重要特征。所以鸭嘴兽既是鸟类，又是哺乳动物。

在实际的动物分类学中，科学家以兽毛和哺乳作为分类的主要依据，将鸭嘴兽列入哺乳动物类，称其为卵生的哺乳动物。本书只是用鸭嘴兽来说明多继承的问题，在这方面不做细致的考究。

如果要在程序中定义一个鸭嘴兽类，采用单继承肯定是不行的。否则，鸭嘴兽要么是鸟类，要么是哺乳动物，显然不符合实际情况。所以此时应当采用多继承，让鸭嘴兽同时继承鸟类和哺乳动物类的属性和行为。这样，一个鸭嘴兽就既是鸟类，又是哺乳动物，符合实际情况。

16.2 定义多继承

多继承的语法同单继承类似，只需要在定义类时在类名后面依次罗列继承方式和基类即可。继承方式同单继承一样，也有 public，protected 和 private。在多继承中，针对不同的基类可以使用不同的继承方法。其语法如下：

```
class  派生类名 : 继承方式1  基类名1,
                 继承方式2  基类名2,
                 ……
{
    派生类新增成员
};
```

具有两个基类的多继承类图如图 16.2 所示。

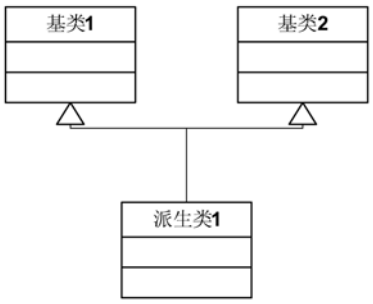


图 16.2 多继承的类图

例如，定义一个鸭嘴兽类，应当继承鸟类和哺乳动物类：

```
class  Bird                          // 鸟类
{
    ……
};
class  Mammal                        // 哺乳动物类
{
    ……
};
class  Duckbill  :  public  Bird,    // 鸭嘴兽类，从鸟类和哺乳动物类派生
                public  Mammal
```

```
{
    ……
};
```

同定义单继承派生类的构造函数一样，定义多继承派生类时也要注意基类的初始化。如果基类没有默认的构造函数，那么在派生类构造函数的初始化列表里就要依次调用各个基类的构造函数。无论开发者如何安排，基类构造函数的调用次序总是按照其定义时的次序。

例如，在 Duckbill 类的构造函数里，由于先继承的是 Bird 类，然后是 Mammal 类，所以在 Duckbill 类对象初始化时，先调用 Bird 类的构造函数，再调用 Mammal 类的构造函数，其后是各数据成员的初始化，最后才是 Duckbill 类的构造函数。

```
Duckbill::Duckbill()    // Duckbill类的构造函数
    : Bird(),           // 第一个基类
     Mammal(),          // 第二个基类
     ……                 // 成员初始化列表
{
    ……
}
```

多继承派生类对象在析构时按照与构造相反的顺序进行，即先调用派生类自己的析构函数，再析构各个数据成员，然后按照相反的顺序，依次调用各个基类的析构函数。

16.3 多继承派生类对象的内存布局

同单继承一样，通过多继承派生类将拥有基类所有的属性和行为。在多继承派生类的对象中，将依次排列各个基类的非静态数据成员以及派生类新增的数据成员。派生类对象内存中的数据是按照定义时的顺序排列的。也就是说，在定义派生类时，排在前面的基类，其数据在派生类对象中也排在前面。

一个多继承的类图如图 16.3 所示。

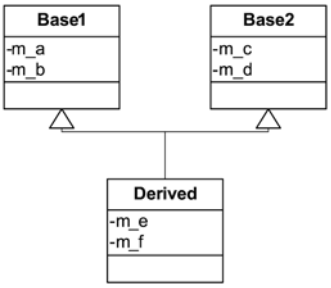


图 16.3　多继承类图

其派生类对象的内存布局如图 16.4 所示。

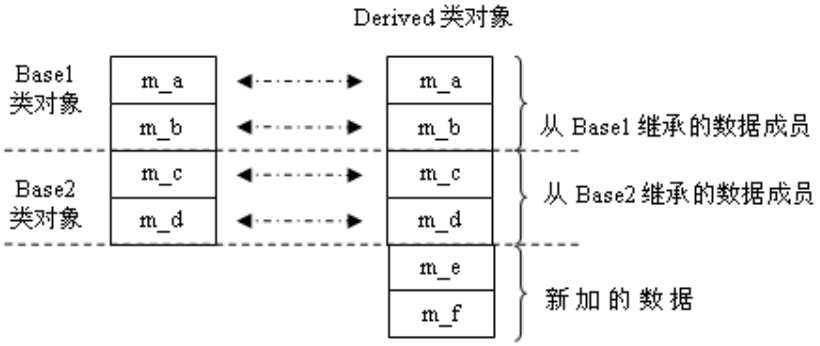


图 16.4 派生类对象的内存布局

派生类对象也可以转换为其基类类型的对象。对于多继承的情况，在转换时编译器可以根据要转换的类型进行适当的转换。例如，对于上面的多继承类，如果要将 Derived 类对象转换成 Base2 类的对象，编译器会从 Derived 对象中按照内存排列的顺序，从中截取出从 Base2 类继承来的部分构成新对象。

```
Derived  d;
Base2  b2 = static_cast<Base2>( d );
```

派生类对象转换为基类对象如图 16.5 所示。

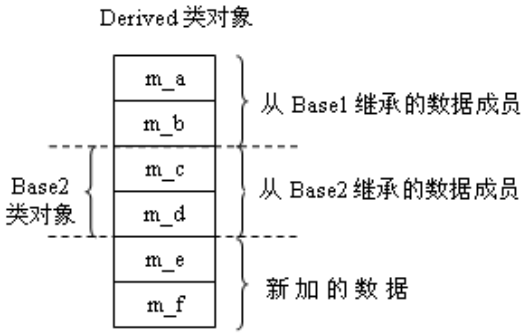


图 16.5 派生类对象转换为基类对象

16.4 访问基类成员

在多继承中，如果多个基类拥有同名成员，那么在访问基类成员时，仅通过成员名并不能区分是哪个基类的成员。解决的方法是在成员名前用域运算符::指明成员所属的基类。通过这种方法访问数据成员和函数成员的语法如下：

```
基类名 :: 数据成员名;                                    // 在派生类成员函数中访问基类成员数据
基类名 :: 函数成员名( 参数列表 );                        // 在派生类成员函数中访问基类成员函数
派生类对象 . 基类名 :: 数据成员名;
派生类对象 . 基类名 :: 函数成员名( 参数列表 );
派生类指针 -> 基类名 :: 数据成员名;
派生类指针 -> 基类名 :: 函数成员名( 参数列表 );
```

对于没有重名的成员，也可以通过域运算符访问指定类的成员。但是既然名字已经是唯一的，这样做就没有必要了。

例如，派生类 Derived 有两个基类 Base1 和 Base2，这两个基类都具有成员数据 m_a 以及成员函数 f2( )，则可以如下访问基类的成员：

```
void  Derived::f1( )                  // 派生类新增的成员函数 f1()
{
    ……
    Base1::m_a  =  123;               // 访问基类的成员 m_a
    Base2::f2( );                     // 调用基类 Base2 的成员函数
    ……
}
Derived  obj;                         // 派生类对象
obj.Base1::m_a = 456;                 // 通过派生类对象访问基类成员
obj.Base2::f2();
Derived  *pObj = new Derived();       // 派生类指针
pObj->Base1::m_a = 456;               // 通过派生类指针访问基类成员
pObj->Base2::f2();
```

下例中定义一个鸭嘴兽类 ,既具有鸟类卵生的特性 ,又具有哺乳动物类用乳汁喂养后代的特性，程序如示例代码 16.1 所示。

示例代码 16.1

```
#include <cstdlib>
#include <iostream>
using namespace std;                        // 使用名称空间 std
class Bird                                  // 鸟类
{
public:
    void LayEggs()                          // 鸟类的“下蛋”方法
    {
        cout<<"下蛋"<<endl;
    }
};
class Mammal                                // 哺乳动物类
{
public:
    void Suckle()                           // 哺乳动物的“哺乳”方法
    {
        cout<<"哺乳"<<endl;
    }
};
class DuckBill : public Bird,               // 鸭嘴兽类多继承自鸟类和哺乳动物类
            public Mammal
{
};
```

```
int main(int argc, char *argv[])        // 主函数
{
    cout<<"——多继承——"<<endl;          // 输出提示信息

    DuckBill  duckbill;                 // 定义一个鸭嘴兽对象

    cout<<"鸭嘴兽的行为："<<endl;       // 输出提示信息
    duckbill.LayEggs();                 // 调用鸭嘴兽的“下蛋”方法
    duckbill.Suckle();                  // 调用鸭嘴兽的“哺乳”方法

    system("PAUSE");                    // 等待用户反应
    return EXIT_SUCCESS;                // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 16.6 所示。

图 16.6 模拟鸭嘴兽的行为结果

在上述程序中定义了一个鸭嘴兽类，该类从鸟类和哺乳动物类多继承而来，从而使得鸭嘴兽同时具有鸟类和哺乳动物类的属性和行为，然后通过鸭嘴兽对象分别调用了鸟类和哺乳动物类的成员函数。

16.5 多继承存在的问题

多继承虽然功能强大，可以让派生类同时具有多个基类的属性和行为，但是多继承同时也会带来一些严重的问题。其中比较常见的问题就是多继承会导致数据重复，并由此带来数据不一致的问题。

比较典型的情况是一个派生类 D 从两个基类 B 和 C 中派生，而这两个基类又有一个共同的基类 A，这就会导致 A 的数据在 D 中被重复两次，如图 16.7 所示。D 多继承 B 和 C，将 B 和 C 的数据复制到 D 中。由于 A 的数据已经分别被 B 和 C 继承，所以 A 的数据在 D 中将重复两次。而且在定义 D 类的成员函数时，或者通过 D 类对象和指针访问成员数据 a 时，必须用域运算符::指明 a 所在的类，即：

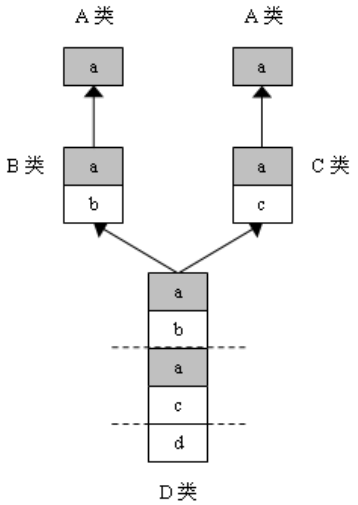


图 16.7 多继承导致数据重复

```
B::a = 1;                   // 在 D 的成员函数中访问 A 类的数据成员
```

```
C::a = 2;
D  dObj;                  // D 类对象
dObj.B::a = 3;            // 通过 D 类对象访问 A 类的数据成员
dObj.C::a = 4;
D  *pObj = new D();       // D 类指针
pObj->B::a = 5;           // 通过 D 类指针访问 A 类的数据成员
pObj->C::a = 6;
```

从编译器的设计角度来讲，当 D 从 B 和 C 继承时并不知道基类 A 的存在。D 只能全盘接受来自 B 和 C 的数据，而无法区分其中的数据 a 到底是从 B 继承而来的，还是从 C 继承而来的。所以要访问数据 a，只能由用户来指明。

从逻辑的角度来讲，在 D 类的对象中 A 的数据应当只有一份。比如有一个动物基类 Animal，它具有重量属性。鸟类(Bird)和哺乳动物类(Mammal)都从 Animal 派生，然后鸭嘴兽类(DuckBill)又从鸟类和哺乳动物类派生。从继承的语义来讲，一个 DuckBill 对象也是一个 Animal，所以鸭嘴兽应当具有重量属性。但是，由于多继承导致数据冗余，所以基类的一份数据，在其间接派生类中产生了多份副本。所以在上述的鸭嘴兽对象中将具有“两”个重量属性。这显然是不符合逻辑的。而且由于数据冗余，也容易导致数据的不一致。例如上例的 D 类，其中继承自 B 的数据 a 和继承自 C 的数据 a 可以分别访问，如果开发者不能始终保证每次修改两个数据使其完全一样，那么就很容易导致数据不一致。

```
D  dObj;                  // 定义 D 类对象
……
dObj.B::a = 1;            // 修改继承自 B 的数据 a
……
dObj.C::a = 2;            // 修改继承自 C 的数据 a
```

显然，在上述代码中很容易导致一个数据 a 有两个不同值，而这种情况是多继承无法克服的一个缺点。另外，如果 A 类的构造函数带有参数（而且没有默认构造函数），那么在 B 类和 C 类构造时就必须调用这个构造函数。假设由于开发者的疏忽，导致 B 类和 C 类在调用 A 类的构造函数时不一致，那么 D 类中的两个数据 a 也就会不一致。

为了解决多继承导致的数据冗余和数据不一致的问题，可以采用下面小节中的虚拟继承机制，也可以禁止最初的基类带有数据。一个不带有任何数据（仅有函数成员）的基类也称做接口。

16.6 虚拟继承

虚拟继承是解决多继承带来的问题的一个重要机制。通过虚拟继承，基类的数据在派生类中将只有一份副本，从而避免了多继承导致的数据冗余和数据不一致问题。

16.6.1 虚拟继承的语法

虚拟继承是在定义派生类时将基类指明为虚基类，或者说派生类以虚拟的方式从基类派生。虚拟继承的方法是在普通继承的基类名前加上 virtual 关键字，如下所示：

```
class 派生类名 : 继承方式  virtual  基类名
{
    派生类的定义
};
```

例如 B 类从 A 类虚拟继承，则 B 类定义如下：

```
class  B : public  virtual  A        // B 类从 A 类虚拟继承
{
    ……
private:
    int b;                           // B 类新增的成员数据
};
```

虚拟继承中的基类也称做“虚基类”。同多继承一样，虚基类也可以有多个，中间用逗号分隔。而且在定义派生类的过程中，虚基类和非虚基类可以一起使用。

一般来讲，虚拟继承用在具有三层以及三层以上的类层次结构中。一旦被声明为虚基类，则在其后的派生类中，该类将一直作为虚基类。

16.6.2 虚拟继承对象的内存布局

如果仅仅是一级派生，并不能体现出虚拟继承的优点。只有在两级和两级以上的多继承中，虚拟继承的优点才能体现出来。当一个新的类多继承几个基类，而这几个基类又“虚拟”继承自同一个基类，则这个新类对象的内存将因虚拟继承而改变，即在新类的对象中，最初那个基类的数据将只有一份副本。

例如，下面的代码在 A 类中定义一个数据 a，B 类和 C 类分别“虚拟”继承 A 类，而 D 类又多继承 B 类和 C 类，则在 D 类的对象中，数据 a 将只有一份，而不像普通多继承那样有两份数据 a，如图 16.8 所示。

```
class A                              // 基类 A
{
private:
    int a;
};
class B : public virtual A           // B 虚拟继承 A
{
    private:
    int b;
};
class C : public virtual A      // C 虚拟继承 A
{
private:
```

```
    int c;
};
class  D :  public B,           // D 多继承 B 和 C
         public C
{
private:
    int d;
};
```

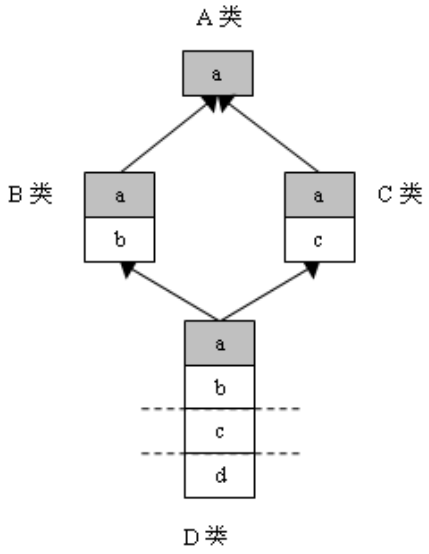


图 16.8　虚拟继承的内存布局

与普通继承不同，在虚拟继承中，派生类对象并不是在其内存中保留一份虚基类数据的副本，而是通过一种间接的引用方式，即将虚基类子对象的数据单独存放，在派生类对象中设置一个指针指向基类子对象。这样，当一个派生类通过多个继承路径继承同一个虚基类时，并不需要产生多个数据副本，而只要维护这个虚基类指针即可。

由于是虚拟继承，无论虚基类在一个派生类中出现多少次，其数据只有一份。因此，在派生类的成员函数定义中，或者通过派生类对象和指针访问虚基类的成员时，可以不使用域运算符::。例如在上例 D 类的定义中，虚基类 A 其实出现了两次，但在定义 D 类的成员函数或者通过 D 类对象（或指针）访问数据 a 时，可以直接访问。

```
a = 1;                  // 在 D 类成员函数中访问数据 a
D  dObj;                    // D 类对象
dObj.a = 2;                 // 通过 D 类对象访问数据 a
D  *pObj = new D;           // D 类指针
pObj->a = 3;                // 通过 D 类指针访问数据 a
```

下例用程序模拟一个水陆两栖坦克。显然，水陆两栖坦克应当多继承坦克和船。因为水陆两栖

坦克既具有普通坦克的属性和行为，也具有船的属性和行为。而且坦克和船都是运输工具。显然“编号”是所有运输工具都应该有的属性，所以应当放在运输工具类中。为了避免多继承导致的数据重复，在由运输工具类派生坦克类和船类时应当使用虚拟继承。类的派生层次如图 16.9 所示。

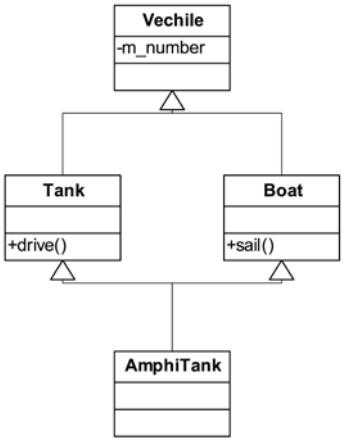


图 16.9 水陆两栖坦克的类层次结构

程序如示例代码 16.2 所示。

示例代码 16.2

```
#include <cstdlib>
#include <iostream>
using namespace std;                    // 使用名称空间 std
class Vehicle                           // 运输工具类
{
public:
    int m_number;                       // 运输工具编号
};
class Tank : public virtual Vehicle     // 坦克类
{
public:
    void drive()                        // 行驶方法
    {
        cout<<"行驶"<<endl;
    }
    void fire()                         // 开火方法
    {
        cout<<"开火"<<endl;
    }
};
class Boat : public virtual Vehicle     // 船类
{
public:
    void sail()                         // 航行方法
    {
        cout<<"航行"<<endl;
```

```
    }
};
class AmphiTank : public Tank,          // 水陆两栖坦克类
                  public Boat
{
};
int main(int argc, char *argv[])        // 主函数
{
    cout<<"——虚拟继承——"<<endl;          // 输出提示信息

    AmphiTank  amphiTank;               // 水陆两栖坦克对象
    amphiTank.m_number = 1;             // 编号

    cout<<"在河水中 ";
    amphiTank.sail();                   // 航行
    cout<<"上岸后 ";
    amphiTank.drive();                  // 行驶
    cout<<"发现目标后 ";
    amphiTank.fire();                   // 开火
    system("PAUSE");                    // 等待用户反应
    return EXIT_SUCCESS;                // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 16.10 所示。

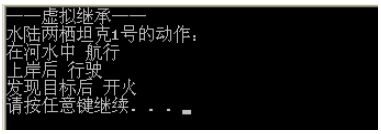


图 16.10　水陆两栖坦克的行为

水陆两栖坦克类（AmphiTank）通过多继承坦克类（Tank）和船类（Boat）具有了水陆两栖的功能，而且 Tank 类和 Boat 类又分别从 Vehicle 类虚拟继承，从而具有了运输工具的属性。

16.6.3 虚拟继承中的构造

由于在虚拟继承中，虚基类的数据只有一份，所以在间接派生类构造时需要特殊处理，即只能初始化虚基类一次。

假设 Vehicle 类有一个带有参数的构造函数（而且没有默认构造函数）Vehicle :: Vehicle(int number)，那么在中间派生类（虚拟继承）Tank 和 Boat 的构造函数中都要显式调用 Vehicle(int number)。但是在 AmphiTank 类多继承自 Tank 类和 Boat 类之后，如果仍然通过两个基类来初始化 Vehicle，那么 Vehicle 将被初始化两次，从而可能导致数据不一致。

所以在 C++中，对于虚基类的初始化进行了特殊处理。如果是在一级派生中，比如 Tank 类虚拟继承 Vehicle 类，那么其初始化同一般继承一样。如果是在多级派生中，那么虚基类的初始化将由最终一级的派生类负责。所以，在水陆两栖坦克的类层次结构中，虚基类 Vehicle 的初始

化应当由最终一级派生类 AmphiTank 负责，即 Vehicle 的构造函数应当放在 AmphiTank 的初始化列表中。

```
AmphiTank::AmphiTank( int number )          // AmphiTank 类的构造函数
    : Vehicle( int number )                 // 初始化虚基类 Vehicle
     Boat( int number ),                    // 初始化基类 Boat
     Tank( int number ),                    // 初始化基类 Tank
{
}
```

为了达到仅初始化一次的效果，虽然在最终派生类中需要初始化中间派生类，但是编译器会抑制中间派生类再去初始化虚基类。也就是说，虽然在定义中间派生类 Tank 和 Boat 的初始化列表时，需要初始化虚基类 Vehicle，但是在 AmphiTank 类的初始化中会取消这个调用。如此，Vehicle 的初始化将只能由 AmphiTank 类显式进行。

如果一个派生类既有虚基类（不一定是直接基类），又有非虚基类，那么无论初始化列表如何排列，虚基类总是先初始化。如果有多个虚基类，那么排在前面的先初始化。

派生类的析构顺序总是与构造顺序相反，所以如果一个派生类有虚基类，则虚基类总是在最后析构。

16.7 虚拟继承的缺点

虚拟继承虽然可以解决多继承带来的数据冗余和数据不一致的缺点，但虚拟继承本身也存在一些问题，具体问题如下：

- 损失效率。为了保证虚基类的数据在派生类中只出现一次，必须采用一种间接的方式访问虚基类，从而在一定程度上损失了效率。
- 派生类要显式初始化其虚拟基类。通常从开发者的角度来讲，设计一个派生类只要初始化其直接基类即可。但是如果在类的派生层次中存在虚拟基类，那么派生类始终要负责这些虚拟基类的初始化，这在一定程度上导致了设计的复杂化。

多继承容易导致数据冗余和数据不一致，而虚拟继承在解决了这个问题的同时又引入了新的问题。对于类层次结构的设计者来讲，可以采取另外一种方法来解决多继承的问题，即只允许一个基类有数据，其他基类只有方法，这样就消除了数据冗余和数据不一致的问题。只有方法没有数据的类也称做接口。

16.8 综合实例

在本节中，将结合综合实例来讲解多继承的应用。在选择实例的时候，尽量保持了实例的实际背景，以使读者了解使用 C++解决实际问题的方法。

16.8.1 改进水陆两栖坦克类

在前面的范例中，虽然采用了虚拟继承，使得水陆两栖坦克类中只有一份基类 Vehicle 的数据。但正如前面分析的那样，虚拟继承也存在效率和初始化较复杂等问题，有必要对其进行改进。

运输工具编号仍然最好放在基类 Vehicle 中。水陆两栖坦克类从坦克类和船类派生，实际上主要是想让其拥有船在水中航行以及坦克在陆地上行驶和开火的行为。而对于这些行为，完全可以建立两个只有方法而没有数据的类，这两个类也不必从 Vehicle 类派生。改进水陆两栖坦克类程序如示例代码 16.3 所示。

示例代码 16.3

```
#include <cstdlib>
#include <iostream>
using namespace std;                          // 使用名称空间 std
class Vehicle                                 // 运输工具类
{
public:
    Vehicle( int num )                        // 运输工具类的构造函数
        : m_number( num )                     // 初始化编号
    {
    }
private:
    int m_number;                             // 编号
};
class IBoat                                   // 船类接口
{
public:
    void sail()                               // 航行方法
    {
        cout<<"航行"<<endl;
    }
};
class ITank                                   // 坦克类接口
{
public:
    void drive()                              // 行驶方法
    {
        cout<<"行驶"<<endl;
    }
    void fire()                               // 开火方法
    {
```

```
        cout<<"开火"<<endl;
    }
};
class AmphiTank : public Vehicle,          // 水陆两栖坦克类
                  public IBoat,            // 继承船类接口
                  public ITank             // 继承坦克类接口
{
public:
    AmphiTank( int num )                   // 水陆两栖坦克类的构造函数
       : Vehicle( num )                    // 初始化基类
    {
    }
};
int main(int argc, char *argv[])
{
    cout<<"——改进水陆两栖坦克——"<<endl;    // 输出提示信息
    AmphiTank  amphiTank(2);               // 水陆两栖坦克对象
    cout<<"水陆两栖坦克 2 号的动作：<<endl;    // 提示信息
    cout<<"在河水中 ";
    amphiTank.sail();                      // 航行
    cout<<"上岸后 ";
    amphiTank.drive();                     // 行驶
    cout<<"发现目标后 ";
    amphiTank.fire();                      // 开火
    system("PAUSE");                       // 等待用户反应
    return EXIT_SUCCESS;                   // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 16.11 所示。

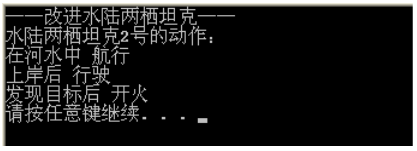


图 16.11 改进的水陆两栖坦克运行结果

16.8.2 改进鸭嘴兽类

对于鸭嘴兽类也可以利用上述方法进行改进。定义一个动物基类 Animal，表示所有动物都有的属性和行为。另外定义两个接口类 IBird 和 IMammal，分别表示鸟类和哺乳动物类的行为。然后定义鸭嘴兽类 Duckbill，多继承上述类。改进鸭嘴兽类程序如示例代码 16.4 所示。

示例代码 16.4

```
#include <cstdlib>
#include <iostream>
using namespace std;                   // 使用名称空间 std
```

```
class Animal                          // 动物类
{
public:
    Animal( int num )                 // 动物类的构造函数
        : m_number( num )             // 初始化动物编号
    {
    }
private:
    int m_number;                     // 动物编号
};
class IBird                           // 鸟类接口
{
public:
    void LayEggs()                    // 鸟类的“下蛋”方法
    {
        cout<<"下蛋"<<endl;
    }
};
class IMammal                         // 哺乳动物类接口
{
public:
    void Suckle()                     // 哺乳动物的“哺乳”方法
    {
        cout<<"哺乳"<<endl;
    }
};
class DuckBill : public Animal,       // 鸭嘴兽类多继承自动物类，以及鸟类和哺乳动物类
                 public Bird,
                 public Mammal
{
};
int main(int argc, char *argv[])      // 主函数
{
    cout<<"——多继承——"<<endl;        // 输出提示信息

    DuckBill  duckbill;               // 定义一个鸭嘴兽对象

    cout<<"鸭嘴兽的行为："<<endl;     // 输出提示信息
    duckbill.LayEggs();               // 调用鸭嘴兽的“下蛋”方法
    duckbill.Suckle();                // 调用鸭嘴兽的“哺乳”方法

   system("PAUSE");                   // 等待用户反应
   return EXIT_SUCCESS;               // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 16.12 所示。

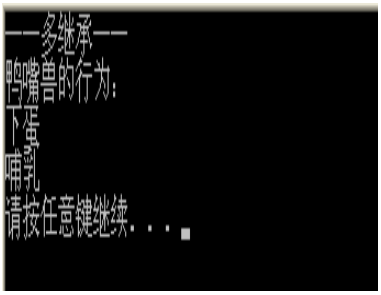


图 16.12 改进的鸭嘴兽类运行结果

16.9 小结

本章重点讲述了多继承及其缺点，以及利用虚拟继承来改进多继承的方法。但虚拟继承并不是最佳的解决方案，在实际开发中，比较理想的策略是只允许一个基类中有数据，而其他基类都是仅有方法的接口类。到目前为止，读者已经学习了面向对象三个基本特征中的两个，即封装和继承，下一章将讲述面向对象的第三个重要特征——多态。

◆ ◆ ◆

第17章 多 态

本章包括

- 多态的概念
- 函数、模板和宏的多态性
- 动态多态
- 虚函数与模板方法
- 纯虚函数与抽象类
- 虚函数与动态绑定

多态是面向对象开发过程中一个非常重要的概念，可以说是面向对象的“核心”。只有体现出多态的特征，一个程序才能称得上是面向对象的程序。本章主要讲述多态的概念以及多态在C++中的实现，重点是如何利用C++类来实现动态多态。

17.1 什么是多态

多态一词来源于希腊语“Polymorphism”，意思是“多形性、拥有多种形态”。面向对象中的多态指的是实体在不同的情况下具有不同的属性或者具有不同的行为。在开发中应用多态可以更好地贴近人的思维，提高开发效率。

多态有狭义和广义之分。从狭义来讲，程序中的实体指的是类对象。在程序运行过程中，情况会随时变化，对象的类型也会随之改变。一旦对象的类型发生改变，对象的行为也会随之改变。从广义来讲，实体指的是程序中的标识符，包括对象、函数、宏、模板等。与对象不同，函数、宏和模板的多态性只能体现在预处理和编译期。

多态毕竟是面向对象的特征，而函数、宏和模板并不是对象，所以对它们的讨论其实已经超出了面向对象的范畴。之所以要在这里列出，主要是因为相关的争论比较多，没有定论，而初学者又难以分辨其界限，所以有必要在这里讲清楚。

17.2 函数、模板和宏的多态性

函数多态实际上就是函数重载。函数被重载之后，各个版本的函数名相同，但参数列表和定义不同。在调用函数时，一个函数名可以接受不同的参数列表，执行不同的代码，所以具有多态性。例如下面的重载函数：

```
void function(int a, int b){
    cout<<"接收整型参数"<<endl;
}
void function(double a, double b){
    cout<<"接收双精度浮点数"<<endl;
}
```

在函数调用时，可以这么用：

```
function(1, 2);              // 调用形参是整型的 function 函数
function(1.0, 2.0);          // 调用形参是双精度浮点数的 function 函数
```

调用者面对的是一个函数名“function”，但可以传入不同类型的参数，调用的结果也不相同。这便是函数重载的多态性。

模板的多态性在于模板可以特化和重载。一个模板名对应着多个实现，当传入不同的参数时，根据匹配原则，编译器会从中选择一个合适的版本。参数不同，匹配的模板版本就会不同，所以具有多态性。例如：

```
template<typename T>
void function(T arg){        // 普通的 function 函数模板
    cout<<"普通版本的函数模板。"<<endl;
}
template< >
void function(int arg){      // 用整型特化的 function 函数模板
    cout<<"特化版本的函数模板。"<<endl;
}
```

编程时，上述的 function 模板可以这么用：

```
function(1.0);               // 使用普通的未经特化的 function 函数模板
function("Hello, World."); // 同上
function(1);                 // 使用用整型特化的 function 函数模板
```

调用者面对的是一个函数模板名“function”，但可以使用不同的参数，从而得到不同的模板实例，当然其运行结果也就不一样了。这便是函数模板的多态性。

宏的多态性指的是在不同的情况下，宏替换的结果会有不同的语义。例如有这样一个宏：

```
#define ADD(x, y)  (x)+(y)
```

而宏 ADD(x, y)可能用在下面的语句中：

```
int x = ADD(1, 2);             // 宏替换结果为 int x = (1)+(2);
int y = 2*ADD(1, 2);           // 宏替换结果为 int y = 2*(1)+(2);
int z = 3*(ADD(1, 2));         // 宏替换结果为 int z = 3*((1)+(2));
```

上述代码中因为宏的使用情况不同，而有不同的计算效果。另外，带有参数的宏虽然类似函数，但是不具有参数类型检查机制，所以可以使用不同类型的参数。例如，上述的宏 ADD(x, y)除了可以对两个整数进行相加操作外，还可以对浮点数甚至字符串进行相加操作。例如：

```
cout<<"1.1 + 2.2 = "<<ADD(1.1, 2.2)<<endl;     // 浮点数相加
string a = "ab", b = "cd";                     // 两个字符串对象
cout<< "\"ab\" + \"cd\" = "<<ADD(a,b)<<endl;   // 字符串相连
```

函数、模板和宏的多态性是静态多态。这是因为它们的多态性仅能体现在预处理期（宏）和编译期（函数、模板）。宏在预处理时完成替换，重载函数在编译期间完成重载解析，模板在使用前必须实例化。一旦编译完成，它们的行为都变成固定的指令，不再改变，所以称为静态多态。

到底静态多态是不是多态，目前还没有定论。读者了解即可，不必拘泥于概念。

17.3 动态多态

与静态多态不同，动态多态是程序运行时的多态。具有动态多态特征的实体是对象。程序运行时，对象的类型会改变，其行为也会随之改变。只是这种改变是有条件的，取决于当时的具体情况，并且受程序的逻辑控制。

C++是一种强类型语言，变量一旦声明，其类型是不会改变的。所以在 C++中实现多态不能直接用对象，而是用指向对象的指针或者引用，并且对象的类型在定义时也有一定要求。

17.3.1 为什么要用动态多态

在讲述为什么要用动态多态之前，先来看一个例子。设计一个图形管理器，负责保存并绘制多个图形对象，如三角形、圆形、矩形等。如果不采用动态多态，程序可能会如图 17.1 那样设计。

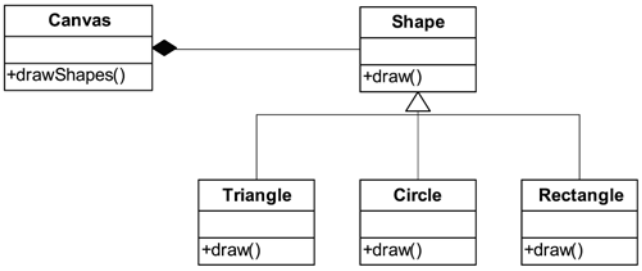


图 17.1 图形管理示意图

其中，Shape 类是所有图形类的基类。既然是代表图形的类，那就得有一个 draw 方法用来绘制自身。Triangle，Circle，Rectangle 等是各种具体的图形类，从 Shape 类派生。每个派生类也都有一个绘制自身的 draw 方法。Canvas 是一个图形管理器。为了保存所有图形，在 Canvas 中定义一个向量：

```
vector<Shape *>  m_shapes;              // 基类指针数组
```

基类类型的指针可以指向派生类的对象。

那么 Canvas 的绘图函数（drawShapes）就要这么写：

```
for( int i=0; i<n; i++ ){               // 遍历所有图形对象
    if( m_shapes[i] 是三角形 ){
        // 将基类指针转换为派生类指针
        Triangle *pObj = static_cast<Triangle *>( m_shapes[i] );
        pObj->draw();           // 调用 Triangle 类的 draw 方法
    }
```

```
    else if( m_shapes[i] 是圆 ){
            Circle *pObj = static_cast<Circle *>( m_shapes[i] );
        pObj->draw();           // 调用 Circle 类的 draw 方法
    }
    else if( shapes[i] 是矩形 ){
        Rectangle *pObj = static_cast<Rectangle *>( m_shapes[i] );
        pObj->draw();           // 调用 Rectangle 类的 draw 方法
    }
    ……
}
```

这段代码有两个显著的缺点：

- 需要向下转换指针类型。因为需要绘制的是派生类的对象，所以应该调用派生类的 draw 方法。但得到的指针却是一个基类类型的指针，不能访问派生类的成员函数。所以为了调用 draw 方法，必须先将基类指针转换为派生类的指针，也就是所谓的向下转换。
- 不方便维护。如果再增加一种图形，则必须修改这部分图形绘制程序，需要添加一个 else…if 判断，以及调用新图形类的 draw 方法，而不是仅仅添加一个新的图形类就可以。如果派生类较少，那么这种设计或许还能应付，一旦派生类很多，有几十、几百的话，这种设计就难以胜任了。

显然，如果不采用某种特殊方法，上述缺点是无法克服的。而 C++的动态多态就是这种特殊方法。先不管动态多态怎么实现，先来看看用动态多态能做什么。下面的代码就是用动态多态改写的 Canvas 的绘图函数：

```
for( int i=0; i<n; i++ ){       // 遍历所有图形对象
    m_shapes[i]->draw();        // 调用派生类的 draw 方法，绘制图形
}
```

很显然，与上一个版本比起来，这段代码改进非常大。不仅写的代码少了，而且也不再需要 if…else 判断和指针类型的转换，程序维护起来方便多了。这就是应用动态多态的好处。

一般来讲，当基类中的函数不能满足派生类的特殊性时，可以考虑用动态多态。而这些与派生类特性相关的函数，也就是所谓的多态行为。在本例中，基类 Shape 的 draw 函数不能代替各个派生类的 draw 函数，而且 draw 函数怎么实现，也是根据派生类不同而不同的。所以 draw 函数是一种多态行为，图形绘制也适用动态多态。

17.3.2 如何实现动态多态

要实现动态多态，必须遵循以下的设计规则：

- 与多态行为相关的函数，在基类中声明为虚函数。
- 派生类改写基类中的虚函数。
- 通过基类类型的指针（或者引用）调用虚函数。

由此可见，实现动态多态依靠的主要是虚函数。虚函数也是类的成员函数，只是在声明时比普通成员函数多了一个 virtual 关键字，语法如下：

```
class ClassName
{
……
virtual <返回类型> Function( 参数列表 );
……
};
```

如果在基类中某个成员函数被声明为虚函数，则其意图就是希望派生类改写。注意，这只是“希望”，不是强迫。派生类也可以选择不改写基类中的虚函数，而是接受基类对该虚函数的默认实现。如果这种默认实现不能满足派生类的特殊需要，则派生类必须将其改写。在派生类的声明中，要将需要改写的虚函数重新声明。例如，Function 是基类里定义的虚函数，在派生类中重新声明如下：

```
virtual <返回类型> Function( 参数列表 );
```

派生类重新声明虚函数时，应当遵循以下原则：

- 函数名相同。
- 参数列表相同。
- 与基类中的虚函数同为 const 或非 const 成员函数。
- 返回值类型相同或者是存在继承关系类型的指针（或引用），即派生类虚函数返回指针的类型继承自基类返回指针的类型。

如果不遵循上述原则，那么派生类就是在声明一个新的成员函数，而不是重新声明，也就达不到多态的效果了。下面的代码是一个合法的重新声明：

```
class Base{
public:
    // 基类中虚函数的声明，返回一个指向基类 A 对象的指针
    virtual A* Function() /*const*/;
};
class Derive : public Base{
public:
    // 重新声明，返回一个指向派生类 B 对象的指针，B 从 A 派生
    virtual B* Function() /*const*/;
};
```

Base 类和 Derive 类中的 Function 函数或者同为 const 成员函数，或者都不是，否则两者就是重载关系。

派生类重新声明虚函数时，virtual 关键字可加可不加。一个成员函数只要在基类里声明为虚函数，则在其后所有层次的派生类中都是虚函数，无论重新声明时是否带有 virtual 关键字。

正如前面提到过的一样，C++中的对象一旦声明，其类型就不再改变，行为也就不会改变。所以在 C++中不能通过对象实现动态多态，而是通过基类的指针或者引用。指针和引用的类型虽然也不会改变，但其指向的对象可以改变，因为 C++允许将一个基类指针或引用指向派生类对象。根据 C++语法规则，通过指针或引用调用的虚函数，实际是所指对象类型的虚函数。

```
Base *pBase = NULL;     // 声明并初始化一个基类指针
```

```
pBase = new Derive();   // 基类指针指向一个派生类对象
pBase->Function();  // 调用虚函数，如果该虚函数被派生类改写，则调用的是派生类的虚函数
```

在程序运行时，基类指针或引用指向的对象随时会变。这种变化在编程时无法预知，而取决于运行时的具体情况以及程序的逻辑，例如用户的操作，或者程序某一步的计算结果。一旦所指对象发生变化，那么通过基类指针或引用调用的虚函数就会变换（调用实际所指对象的虚函数）。这便是动态多态。

17.3.3 用动态多态改进图形绘制程序

在前面小节的那个图形绘制的例子中，基类 Shape 的 draw 函数不能替代派生类的 draw 函数，而且各个派生类的 draw 函数也都具有自己独特的实现，所以适合用动态多态，具体步骤如下：

step 1 改写 Shape 类，将 draw 函数声明为虚函数，即在声明中加上 virtual 关键字。

step 2 保证各个派生类中的 draw 函数与基类中的版本相匹配，本例中只要保证其声明完全相同即可。

step 3 通过 Shape 类指针调用虚函数 draw。

相关代码如下：

```
// 带有虚函数的类
class Shape{                                        // 图形基类
public:
    virtual void draw(){                            // 基类的虚拟 draw 方法
        cout<<"Basic Shape Drawing."<<endl;         // 图形基类的绘制
    }
};
class Triangle: public Shape{                       // 三角形类
public:
    /*virtual*/ void draw(){                        // 重写基类的 draw 方法
        ……                                          // 画三角形
    }
};
class Circle: public Shape{                         // 圆类
public:
    /*virtual*/ void draw(){                        // 重写基类的 draw 方法
        ……                                          // 画圆
    }
};
……
// 图形绘制部分的代码
for( int i=0; i<n; i++ ){                       // 遍历所有图形对象
    m_shapes[i]->draw();                        // 调用派生类的 draw 方法，绘制图形
}
```

通过基类指针或引用访问派生类的成员函数，仅限于虚拟成员函数，而且是已在基类中声明并被派生类改写的虚拟成员函数。

17.3.4 动态多态实例——计算不同职员的薪水

前面已经讲了为什么要用动态多态以及如何实现动态多态，现在再来看一个完整的实例。一般来讲，一个公司在计算职员的薪水时，会根据其职务采取不同的计算方法；经理的薪水可能是固定的，工程师的薪水则除了基本工资外，还可能会有项目奖金；而销售人员一般还会有提成。由此可见，职员的薪水计算方法是一种多态行为，适合用动态多态。下面利用动态多态机制（虚函数），计算不同职务职员的薪水。

各种职员的类图如图 17.2 所示。

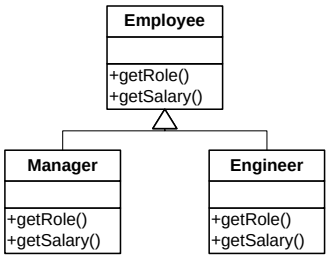


图 17.2　各种职员的类图

主函数部分的流程图如图 17.3 所示。

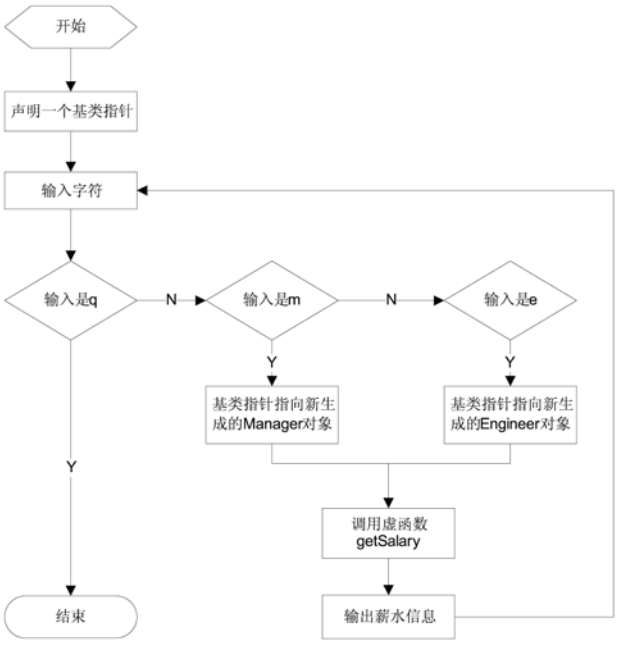


图 17.3　输出各种职员的薪水流程图

程序如示例代码 17.1 所示。

示例代码 17.1

```
//////////////////////////////////////////////////////////////////////
// Employee.h   主程序文件
#include <cstdlib>
#include <iostream>
#include "Employee.h"                // 包含各种职员类的头文件
using namespace std;                 // 使用名称空间 std
int main(int argc, char *argv[])
{
    cout<<"请输入[m:经理，e:工程师, q:结束]: ";        // 提示用户输入
    Employee *pObj = NULL;           // 声明一个职员类指针，并初始化
    string role;                     // 保存用户输入的字符串
    cin>>role;                       // 获取用户的第一次输入
    while( "q" != role ){            // 如果用户输入“q”, 则退出程序
        if( "m" == role ){           // 如果用户输入“m”, 则生成经理类对象
            pObj = new Manager();    // 用职员类指针指向新生成的经理类对象
        }
        else if( "e" == role ){      // 如果用户输入“e”, 则生成工程师对象
            pObj = new Engineer();   // 用职员类指针指向新生成的工程师对象
        }
        cout<<"薪水: "<<pObj->getSalary()<<endl;    // 输出职员的薪水
        delete pObj;                 // 释放内存，销毁新生成的对象
        pObj = NULL;                 // 将指针归零
        cout<<"[m, e, q]: ";         // 重新提示用户输入
        cin>>role;                   // 获取用户输入
    }
   system("PAUSE");                  // 等待用户反应
   return EXIT_SUCCESS;              // 程序结束退出
}
//////////////////////////////////////////////////////////////////////
// Employee.h   各种职员类的头文件
#ifndef _EMPLOYEE_H_
#define _EMPLOYEE_H_
// 职员基类
class Employee{
public:
    virtual double getSalary();     // 获取职员的薪水
};
// 经理类
class Manager : public Employee{
public:
    virtual double getSalary();     // 获取经理的薪水
};
// 工程师类
class Engineer : public Employee{
```

```
public:
    virtual double getSalary();     // 获取工程师的薪水
};
\#endif  // _EMPLOYEE_H_
////////////////////////////////////////////////////////////////////////
// Employee.cpp   各种职员类的实现文件
#include "Employee.h"
// virtual
double  Employee::getSalary(){
    return 0.0;                     // 因为是职员基类，薪水未确定
}
// virtual
double  Manager::getSalary(){
    return 10000;                   // 返回经理的薪水，此处 10000 只是示意
}
// virtual
double  Engineer::getSalary(){
    return 6000;                    // 返回工程师的薪水，此处 6000 只是示意
}
```

建立一个控制台工程，以及相应的.h 和.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 17.4 所示。

图 17.4　计算不同职员的薪水结果

Employee 类是表示职员的基类，拥有所有职员的共同属性和共同行为。Manager 类和 Engineer 类从 Employee 类派生，分别表示具体的经理和工程师。这里限于篇幅，只列出了 getSalary 函数，代表获取薪水的行为。

因为不同的职员其薪水的计算方法也不同，所以 getSalary 属于“多态”行为，应当在 Employee 类中将其声明为虚函数（virtual function）。Manager 类和 Engineer 类是具体的职员类，其获取角色名称和薪水的方法各有不同，因此应当改写基类中的 getSalary 函数。在主程序中，根据用户的输入生成不同的成员对象，并用基类指针指向该对象。因为用户的输入无法预测，所以到底生成什么对象只有到程序运行时才能确定，这便是“动态”多态。

请读者注意，获取用户角色和薪水时，只是通过基类指针调用相应的函数，而没有通过 if…else 判断具体指向的对象。如果增加一种具体职员的类型，比如销售人员（Salesman），那么主程序中只要增加一个 else…if 判断，并用基类指针指向新生成的 Salesman 对象即可，主程序的其他部分并不需要修改。

这样做既节省了很多代码，也符合人的思维。因为基类 Employee 已经定义了职员的共同行为，所以不管是什么职员都可以获取角色名称和薪水。这就是应用动态多态的好处。

17.4 虚函数与模板方法

虚函数的调用除了通过指针和引用，还有一种方式值得注意，即在非虚函数中调用虚函数。如果基类指针调用一个非虚函数，而该函数又调用了一个虚函数，则实际被调用的也是基类指针所指对象的虚函数。例如：

```
class Base{                 // 基类
public:
    void normal(){          // 基类中的非虚函数
        virtualFunction();  // 调用一个虚函数
    }
};
Base *pObj;
…
pObj->normal();             // 通过基类指针 pObj 调用非虚函数 normal
```

如果基类指针 pObj 指向一个派生类对象，而该派生类又改写了 virtualFunction，则通过 normal 函数调用的虚函数将是派生类中的版本。

虚函数的这个调用规则也是设计模式“模板方法”的基础。所谓模板方法指的是基类定义一个算法的步骤，而某些步骤则在派生类中实现。比如在绘制图形时，一般要经过绘图准备、绘图、绘图结束三个步骤。这个过程可以在图形基类中用一个非虚方法（paint）表示。绘图的准备和结束对所有图形都是一样的，可以在图形基类中用非虚方法 begin 和 end 表示。而不同的具体图形有不同的绘图方法，所以应当将绘图方法定义为虚函数 draw。然后派生类改写这个虚函数，实现具体的绘图方法。

“模板方法”中的模板不是“模板（template）”。

下例用模板方法实现图形的绘制过程，程序如示例代码 17.2 所示。

示例代码 17.2

```
#include <cstdlib>
#include <iostream>
using namespace std;
class Shape{                        // 图形基类
public:
    void begin(){}                  // 绘图开始函数，非虚函数
    void end(){}                    // 绘图结束函数，非虚函数
```

```
    virtual void draw(){}           // 绘图方法，虚函数
    void paint(){                   // 绘图的模板方法，表示绘图的过程
        begin();                    // 开始绘图
        draw();                     // 绘图
        end();                      // 结束绘图
    }
};
class Circle : public Shape{        // 圆类，派生自图形基类
public:
    /*virtual*/void draw(){         // 画圆的函数，改写基类的 draw 方法
        cout<<"Draw a circle"<<endl;    // 画圆的示意代码
    }
};
int main(int argc, char *argv[])    // 主函数
{
    Shape *pObj = new Circle();     // 用基类指针指向派生类对象
    pObj->paint();                  // 调用基类的非虚方法，绘图的“模板方法”
    delete pObj;                    // 删除指针
    system("PAUSE");                // 等待用户反应
    return EXIT_SUCCESS;            // 返回
}
```

建立一个控制台工程，以及相应.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 17.5 所示。

图 17.5　用“模板方法”画图结果

基类 Shape 的 paint 函数定义了绘图的一般过程，其中 begin 和 end 函数已经在 Shape 类中定义过了，而 draw 函数是一个虚函数，可以由派生类改写，所以 Shape 的 paint 函数是一个模板方法。

17.5 纯虚函数与抽象类

从类层次的设计角度来讲，基类是其所有派生类的抽象表示，例如 Shape 类对于 Circle，Triangle 等类，以及 Employee 类对于 Manager，Engineer 等类。在现实世界中，这种基类的对象是没有意义的。因为无论什么图形都应该有其具体的类型，无论哪个职员都应该有其具体的职务。所以应当禁止这种基类的实例化。这种不能实例化的基类也称做抽象类。

一个设计良好的类层次结构，应该只有最终的派生类（即叶子一级的类）可以实例化，任何层次的基类都不应该被实例化。但在实际开发中，有时难以一步设计到位，所以一个类到底是不是抽象类，需要根据具体情况决定。

17.5.1 纯虚函数

在C++中是通过纯虚函数来实现一个抽象类的。拥有纯虚函数的抽象类不允许实例化。一个纯虚函数的声明语法如下：

```
virtual <返回类型> function(<参数列表>) = 0;
```

纯虚函数也是虚函数，但与一般虚函数有两个不同点：

- 纯虚函数的声明必须加上“=0”，而一般虚函数没有。
- 类可以不实现纯虚函数，但必须实现一般虚函数。

一般来讲，抽象类不需要实现纯虚函数，因为该虚函数终究要被派生类改写。如果抽象类要实现虚函数，则其实现跟一般虚函数没有什么不同，其含义也是为派生类提供一个默认的实现。派生类也可以静态调用基类的纯虚函数。

如果强行实例化抽象类，则会导致编译出错。例如Shape类，如果成员函数draw为纯虚函数，则Shape类就不能如下实例化：

```
class Shape{
public:
    virtual void draw() = 0;    // 纯虚函数draw，不必实现
};
…
Shape obj;              // 编译错误，Shape类带有纯虚函数，是一个抽象类，不能实例化
```

17.5.2 什么时候用纯虚函数

纯虚函数首先是虚函数。如果某个虚函数不需要在基类中实现，或者实现也没有实际意义，则应该声明为纯虚函数。例如Shape类的draw函数，Shape类是对所有图形类的抽象，到底要画成什么样子在Shape类中不能确定，因此不能给出draw函数的具体实现，draw函数应当声明为纯虚函数。

Employee类的getSalary函数也是如此。因为对于基类Employee来讲，职员的具体职务还没确定，所以其薪水的计算方法也不能确定，在getSalary中返回任何数值都没有意义。因此，getSalary也应当声明为纯虚函数，如下所示：

```
class Employee{
public:
    virtual double getSalary() = 0;
};
```

17.5.3 从抽象类派生具体类

拥有纯虚函数的抽象类不能实例化，要实例化必须首先从抽象类派生出一个具体类，然后实例化该具体类。派生出的具体类必须改写基类的所有纯虚函数，否则还是一个抽象类。例如在Shape类中定义两个纯虚函数draw和getSize，如果Circle类只改写draw函数，则Circle类仍然不能实例化。强行实例化一个抽象类会导致编译错误，如下所示：

```
class Shape{                                        // 抽象类 Shape
public:
    virtual void draw() = 0;                        // 纯虚函数 draw
    virtual void getSize(int &x, int &y) = 0;  // 纯虚函数 getSize
};
class Circle : public Shape{       // 派生类 Circle
public:
    /*virtual*/ void draw(){       // 仅改写纯虚函数 draw
        cout<<"Draw a circle"<<endl;
}
};
。。。
Circle obj;                 // 编译错误，Circle 类未改写纯虚函数 getSize，不能实例化
```

所谓具体类就是可以实例化的类。从类层次的设计来讲，具体类应当是最终的叶子级别的类。从最初的根类到最终的叶子类，中间可以有多级抽象类。

17.5.4 仅有纯虚函数的类——接口

如果一个类的所有成员函数都是纯虚函数，而且没有数据成员，那么该类就是一个接口。例如：

```
class Interface{                        // 接口类，仅有纯虚成员函数，没有数据成员
public:
    virtual void f1() = 0;                  // 接口函数 f1
    virtual int f2( char *p ) = 0;          // 接口函数 f2
    virtual float f3( int x, int y) = 0;    // 接口函数 f3
};
```

所谓接口就是函数的集合。如果一个类继承自某个接口，则说明该类支持这个接口，即该类实现了这个接口中的全部或部分函数，那么编程时就可以通过接口来使用这个类。比如在计算职员薪水的例子中，可以设立一个计算薪水的接口，并让各个职员类支持该接口，如下所示：

```
class ISalaryCount{                         // 计算薪水的接口
public:
    virtual double getBasicSalary() = 0;    // 获取基本薪水的函数
    virtual double getBonus() = 0;          // 获取奖金的函数
};
class Employee : public ISalaryCount{  // 支持 ISalaryCount 接口的 Employee 类
    /*virtual*/ double getBonus(){          // 实现接口的 getBonus 函数
        return 500;                         // 此处返回 500 只是示意
    }
};
// Manager 类从 Employee 类派生，从而也支持 ISalaryCount 接口
class Manager : public Employee{
public:
    /**/ double getBasicSalary(){// 实现 ISalaryCount 接口的 getBasicSalary 函数
```

```
        return 10000;                          // 此处返回 10000 只是示意
    }
};
```

在上述例子中，Employee 类从 ISalaryCount 接口派生，说明 Employee 类支持 ISalaryCount 接口。但 Employee 类并没有实现接口中的全部函数，仍然是一个抽象类。Manager 类从 Employee 类派生，间接派生自 ISalaryCount 接口，所以 Manager 类也支持该接口。由于 Manager 类实现了剩下的纯虚函数 getBasicSalary，不再是一个抽象类，所以 Manager 类可以实例化。

在声明一个接口时，经常用 struct 而不是 class。因为在 C++中这两个关键字的语义基本相同，唯一的区别是 struct 中成员的默认访问权限是 public，而 class 的是 private。这样在用 struct 声明一个接口时就可以不写 public 访问限定符。

17.5.5 图形类的接口

对于一个图形，通常的操作有绘制、获取尺寸、变换位置和形状等，而这些操作都可以看做图形类能够支持的接口。下例通过接口访问和操作具体的图形类，相关的类图如图 17.6 所示。

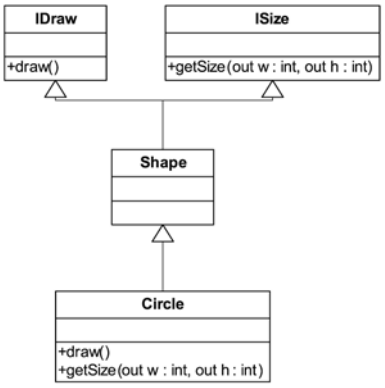


图 17.6 图形类的相关类图

程序如示例代码 17.3 所示。

示例代码 17.3

```
//////////////////////////////////////////////////////////////////////////
// main.cpp 主程序文件
#include <cstdlib>
#include <iostream>
#include "Circle.h"
using namespace std;
int main(int argc, char *argv[]){      // 主函数
    Circle circle(0, 0, 123);          // 声明一个圆对象
    IDraw *pDraw = &circle;            // 获取 IDraw 接口指针，该指针指向圆对象
    pDraw->draw();                     // 通过 IDraw 接口画圆
    ISize *pSize = &circle;            // 获取 ISize 接口指针，该指针指向圆对象
```

```
    int w = 0, h = 0;                   // 获取宽高的变量
    pSize->getSize(w, h);               // 通过 ISize 接口获取圆的宽高
    cout<<"Width: "<<w<<endl;           // 输出宽
    cout<<"Height: "<<h<<endl;          // 输出高
    system("PAUSE");                    // 等待用户反应
    return EXIT_SUCCESS;                // 返回
}
////////////////////////////////////////////////////////////////////////
// Interface.h  接口类头文件
#ifndef __INTERFACE_H__
#define __INTERFACE_H__
struct IDraw{                       // 绘图接口
    virtual void draw() = 0;        // 接口函数 draw
};
struct ISize{                       // 尺寸接口
    virtual void getSize(int &w, int &h) = 0;  // 接口函数 getSize
};
#endif
////////////////////////////////////////////////////////////////////////
// Shape.h  图形类文件
#ifndef __SHAPE_H__
#define __SHAPE_H__
#include "Interface.h"
class Shape : public ISize,public IDraw {  // Shape 类支持 ISize 和 IDraw 接口
};
#endif
////////////////////////////////////////////////////////////////////////
// Circle.h  圆类头文件
#ifndef __CIRCLE_H__
#define __CIRCLE_H__
#include "Shape.h"
class Circle : public Shape{            // 圆类
public:
    Circle(int x, int y, int r);        // 构造函数，输入圆心和半径
    /*virtual*/ void draw();            // 实现 IDraw 接口的 draw 函数
    /*virtual*/ void getSize(int &w, int &h);  // 实现 ISize 接口的 getSize 函数
private:
    int m_x, m_y, m_r;                  // 圆的数据成员
};
#endif
////////////////////////////////////////////////////////////////////////
// Circle.cpp  圆类实现文件
#include <iostream>
#include "Circle.h"
using namespace std;
Circle::Circle(int x, int y, int r)     // 构造函数
    : m_x(x), m_y(y), m_r(r)            // 初始化圆心和半径
{}
```

```
// virtual
void Circle::draw(){                    // 实现 IDraw 接口的 draw 函数
    cout<<"Draw a circle"<<endl;        // 画圆
}
// virtual
void Circle::getSize(int &w, int &h){   // 实现 ISize 接口的 getSize 函数
    w = 2 * m_r;                        // 圆的宽度是半径的两倍
    h = 2 * m_r;                        // 圆的高度是半径的两倍
}
```

建立一个控制台工程，以及相应的.h 和.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 17.7 所示。

图 17.7 画圆并输出圆的宽和高结果

Shape 类从接口 IDraw 和 ISize 继承，表明 Shape 及其派生类都支持这两个接口。但 Shape 类并没有实现接口中的函数。因为从类层次的设计来讲，Shape 类仍然是抽象类（对所有具体图形类的抽象）。而 Circle 类才是最终一级的类，代表“圆”这种具体图形。所以 IDraw 和 ISize 接口中的函数是在 Circle 类中实现的。

要通过接口访问和操作对象，首先要获得指向对象的接口指针。在上述代码中，利用向上类型转换（up-cast）获取了两个指向 Circle 对象的接口指针 pDraw 和 pSize。然后通过这两个指针调用接口中的成员函数。由于是虚函数，所以实际调用的是 Circle 类的函数。

17.6 虚函数与动态绑定

所谓动态绑定指的就是虚函数的调用机制。在程序运行时，指针或引用实际所指对象会发生变化，到底哪个虚函数被调用取决于实际所指的对象。虚函数与指针的这种关联关系就称做动态绑定。

17.6.1 如何实现动态绑定

尽管 C++标准并没有规定如何实现动态绑定，然而现在大多数的 C++编译器都是通过虚函数表（v-table）和虚函数表指针（vptr）来实现的。如果一个类定义了虚函数，则编译器就会为该类创建一个虚函数表。虚函数表是一个数组，其中的每一个元素都是函数指针，指向该类的虚函数。

虚函数表指针，顾名思义是一个指向虚函数表的指针。如果一个类定义了虚函数，则编译器就会为该类的对象嵌入一个虚函数表指针，从而建立对象和虚函数表的索引关系。例如 Shape 类，其声明如下：

```
class Shape{
public:
    virtual void draw();
    virtual void getSize(int &, int &);
    …
```

```
};
```

其虚函数表和虚函数表指针如图 17.8 所示。

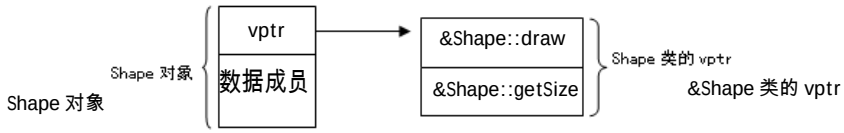


图 17.8　Shape 类的 v-table 和 Shape 对象的 vptr

编译器也会为派生类创建虚函数表。如果派生类没有改写基类中的虚函数，则其虚函数表的内容只是基类虚函数表的复制。如果派生类改写了基类中的虚函数，则其虚函数表中的元素就是改写后的虚函数的地址。例如，Circle 类从 Shape 类派生，并改写 draw 函数：

```
class Circle : public Shape{
public:
    /*virtual*/ void draw();
};
```

则其虚函数表和虚函数表指针如图 17.9 所示。

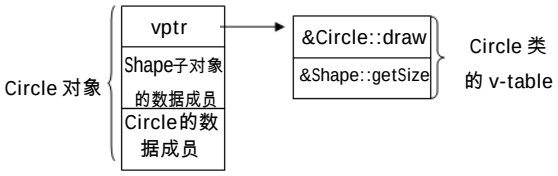


图 17.9　Circle 类的 v-table 和 Circle 对象的 vptr

在为对象嵌入 vptr 时，不同的编译器可能采用不同的策略。有的将 vptr 放在对象的开始地址中，也有的将 vptr 放在基类子对象之后。无论怎样，vptr 与对象的开始地址之差是一个编译期常量。对于基类和其后所有层次的派生类，这个地址之差是一样的。

编译时的动态绑定过程如下：

step 1　编译器通过对象地址计算得到虚函数表指针（vptr）。

step 2　通过虚函数表指针（vptr）得到虚函数表（v-table）的地址。

step 3　根据被调虚函数在类中的次序，得到该虚函数的地址在虚函数表中的索引。

step 4　根据这个索引获取虚函数的入口地址，也就是实际所调虚函数的地址。

显然如何获取正确的 vptr 是非常关键的一步。在通过基类指针或引用调用虚函数时，编译器可以通过指针或引用计算出实际所指对象的开始地址，从而也就可以很容易地得到 vptr。

17.6.2　虚函数的静态调用

有的时候，虽然派生类改写了基类的虚函数，但调用者仍然希望调用基类中的虚函数，这便是虚函数的静态调用。静态调用虚函数时，需要加上域标识符，表明虚函数所属的类，其语法如下：

```
Base *pObj = new Dervied();
pObj-> Base::function();        // 静态调用 Base 类中定义的虚函数 function
```

17.6.3　虚函数的代价

C++的虚函数机制虽然非常有用，但也不是没有代价的，这个代价就是浪费内存。正如前面所说的，编译器会为每个拥有虚函数的类生成一个虚函数表，并为每个类对象嵌入一个虚函数表指针。

假设一个类对象只有一个整型数据，占 4 个字节，再嵌入一个 vptr，那这个类对象占用的内存空间就要翻倍。当类对象的数量比较少的时候，这种内存的浪费尚不明显。一旦类对象的数量变得很多，那内存的浪费就相当惊人了。

还有一种情况，假设基类里声明了很多虚函数，编译器会为基类维护一个比较大的虚函数表。当派生类继承这个基类时，编译器也要为派生类维护一个同样大的虚函数表。如果派生类改写了大部分虚函数，这么做显然是必要的。但是如果派生类只改写了其中的一小部分，或者什么都不改写，编译器仍然要为派生类维护一个很大的虚函数表。这样做显然很浪费内存，尤其是当这种派生类比较多的时候。

17.7 虚拟析构函数

如果基类的析构函数是虚函数，则删除（delete）基类指针时，实际调用的是所指对象的析构函数，即派生类的析构函数。否则派生类的析构函数得不到调用，派生类分配的资源也得不到释放，从而会导致内存泄露。例如：

```
Base *pObj = new Derived();     // 用基类指针指向派生类对象
delete pObj;                    // 删除基类指针
```

删除 Base 类型的指针 pObj 时会调用其析构函数。如果 Base 的析构函数是虚函数，则会按照虚函数的调用规则，调用 pObj 所指对象类型（Derived）的析构函数。否则只能调用 Base 的析构函数。

将基类的析构函数声明为虚函数，只要在其声明前加上 virtual 关键字即可，其语法如下：

```
virtual ~Base();
```

只要基类的析构函数是虚函数，则其所有层次派生类的析构函数都是虚函数。

根据析构函数的调用规则，派生类对象析构时，首先调用其本身的析构函数，然后依次调用其基类的析构函数（从直接基类开始，直至最原始的基类）。这样类层次中的所有析构函数都会被调用，以保证资源被释放。

下例通过虚拟析构函数释放派生类分配的资源，程序如示例代码 17.4 所示。

示例代码 17.4

```
#include <cstdlib>
#include <iostream>
using namespace std;                // 使用标准名称空间
class Base{                         // 基类
public:
    virtual ~Base(){                // 虚拟析构函数
```

```
        cout<<"基类析构"<<endl;      // 基类析构时输出信息
    }
};
class Derived : public Base{        // 派生类
public:
    Derived(){                      // 派生类的构造函数
        m_data = new int[12];       // 派生类分配资源
    }
    /*virtual*/~Derived(){          // 派生类的虚拟析构函数
        delete []m_data;            // 类释放资源
        cout<<"派生类析构"<<endl;    // 派生类析构时输出信息
    }
private:
    int *m_data;                    // 派生类的资源指针
};
int main(int argc, char *argv[])    // 主函数
{
    Base *pObj = new Derived();     // 用基类指针指向派生类对象
    delete pObj;                    // 删除基类指针，调用派生类的虚拟析构函数
    system("PAUSE");                // 等待用户反应
    return EXIT_SUCCESS;            // 退出
}
```

建立一个控制台工程，以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 17.10 所示。

图 17.10　虚拟析构函数示例结果

在上述程序中，如果不将 Base 类的析构函数声明为虚函数，即在第 8 行删除 virtual 关键字，则程序运行后只能输出“基类析构”，也就是说 Derived 类的析构函数没有得到调用，其分配的内存没有得到释放。

在设计一个类层次时，请谨记将基类的析构函数声明为虚函数。

17.8 虚函数的默认实参

虚函数的默认实参是在编译时确定的，而不是在运行时。如果基类中的虚函数有一个默认实参，派生类改写这个虚函数，并声明使用不同的默认实参，那么在调用这个虚函数时，真正使用的默认实参是由调用该函数的指针或引用的类型决定的。如果是一个基类类型的指针或引用，则使用基类中的默认实参；如果是一个派生类类型的指针或引用，则使用派生类的默认实参。例如下面的代码：

```
class Base{
public:
```

```
    virtual void function( int val = 1 ){  // 基类中的虚函数，默认实参为 1
        cout<<val<<endl;                   // 输出实参的值
    }
};
class Derived{
public:
    /*virtual*/void function( int val = 2 ){   // 派生类中的虚函数，默认实参为 2
        cout<<val<<endl;           // 输出实参的值
    }
};
int main(int argc, char *argv[])   // 主函数
{
    Base *pObj = new Derived();    // 用基类指针指向派生类对象
    pObj->function();              // 调用派生类的虚函数
    delete pObj;                   // 删除指针
    return EXIT_SUCCESS;
}
```

在上述代码中，虽然 pObj 实际指向 Derived 类对象，调用的虚函数 function 也是 Derived 类中的版本，但是程序最终的输出结果却是 1，而不是 2。

在编译时，编译器发现 function 函数的调用需要一个默认实参，于是编译器就查找 pObj 类型中的声明。而在编译时，pObj 的类型就是 Base 类，所以这个默认实参只能在 Base 类的声明中查找。查找的结果是 1，所以输出的结果是 1 而不是 2。

当程序运行的时候，程序中并不存在默认实参的信息。不是说编译器做不到，而是如果要这么做的话，势必要花费一定的内存空间和时间，而这对追求效率的 C++ 程序来讲是难以承受的。

17.9 综合实例

本实例模拟各种运输工具的运行。运输工具有很多种，所有运输工具都有运行的功能，但每种运输工具的具体运行行为又各不相同。例如，卡车在陆上行驶，轮船在海中航行，飞机在空中飞行。为了便于在程序中管理各种运输工具，可以设计一个抽象的运输工具基类，然后让各种具体运输工具从中派生。程序如示例代码 17.5 所示。

示例代码 17.5

```
#include <cstdlib>
#include <iostream>
#include <vector>                      // 使用容器 vector
using namespace std;                   // 使用名称空间 std
struct  IVehicle                       // 运输工具抽象类
{
virtual  void move() = 0;              // 运输工具的运行方法
```

```
virtual ~IVehicle(){}                    // 虚拟析构函数
};
class  Truck : public IVehicle           // 卡车类
{
public:
    virtual void move()                  // 改写基类的 move()方法
    {
        cout<<"陆上行驶"<<endl;
    };
};
class Boat : public IVehicle             // 轮船类
{
public:
    virtual void move()                  // 改写基类的 move()方法
    {
        cout<<"海中航行"<<endl;
    }
};
class Plane : public IVehicle            // 飞机类
{
public:
    virtual void move()                  // 改写基类的 move()方法
    {
        cout<<"空中飞行"<<endl;
    }
};
int main(int argc, char *argv[])         // 主函数
{
    cout<<"——模拟运输工具——"<<endl;     // 输出提示信息
    cout<<endl;

    cout<<"选择要生成的运输工具。"
        <<"输入 e 结束。"<<endl;
    cout<<"t:卡车，b:轮船，p:飞机"<<endl;

    char kind;                           // 表示运输工具类型的变量
    cin>>kind;                           // 输入要创建的工具类型

    vector<IVehicle *> vehicles;         // 保存运输工具的 vector 对象
    while( 'e' != kind )                 // 输入字符 e 表示结束输入
    {
        if( 't'!=kind && 'b'!=kind && 'p'!=kind )
                                         // 如果输入的不是类型符，提示输入错误
        {
            cout<<"输入错误！请重新输入："<<endl;
        }
        else
        {
            if( 't' == kind )
            {
                vehicles.push_back(new Truck());   // 创建卡车对象
```

```
            }
            else if( 'b' == kind )
            {
                vehicles.push_back(new Boat());     // 创建轮船对象
            }
            else
            {
              vehicles.push_back(new Plane());      // 创建飞机对象
            }
        }

        cin>>kind;                                  // 输入新的类型
    }
    vector<IVehicle *>::iterator iter = vehicles.begin();  // 容器遍历器

    while( iter != vehicles.end())                  // 遍历所有的运输工具
    {
        (*iter)->move();                            // 调用运输工具的运行方法
        iter++;
    }
    iter = vehicles.begin();
    while( iter != vehicles.end() )
    {
        delete (*iter);                             // 删除运输工具
        iter++;
    }

    cout<<endl;
    system("PAUSE");                                // 等待用户输入
    return EXIT_SUCCESS;                            // 主函数退出
}
```

建立一个控制台工程，以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 17.11 所示。

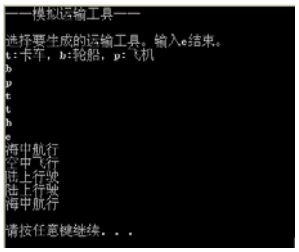


图 17.11　模拟运输工具结果

17.10 小结

本章主要学习了如何利用虚函数机制实现动态多态，以及虚函数机制的实现方法。另外，纯虚函数和抽象类也是本章的重点内容。有了 C++的动态多态和抽象机制，开发程序时就可以在更高的逻辑层面上进行设计了，也更加符合人的逻辑思维了。

◆　　◆　　◆

第18章 类 模 板

本章包括

- 类模板的概念
- 类模板的定义过程
- 类模板的实例化
- 类模板的静态成员和友元
- 类模板的特化和偏特化
- 类模板的匹配规则

本章的主要内容是类模板的定义和使用。同函数模板一样，类模板也是一种对类型进行参数化的技术。读者在学习过程中，可以参照学习函数模板时的经验来理解和使用类模板。不过，这两者之间也有一些差别，书中会有所提示，学习时请注意。

18.1 什么是类模板

顾名思义，类模板就是类的模板。同函数模板一样，类模板也对数据的类型进行了参数化处理。将类模板实例化，也就是指定数据的具体类型，将得到一个具体的、可以定义对象的类。

类模板同函数模板还有一个相似的地方就是模板所能接受的具体类型参数都是有限制的。具体哪些类型可以用来实例化模板，要看模板的具体定义。

在程序中采用函数模板的目的是用相同的程序逻辑处理不同的数据类型，这样就可以复用已有的算法，减少代码冗余。采用类模板也是基于这样的考虑。例如设计一个表示栈的类，如果不采用类模板，则在设计之初就要确定栈中数据的类型，比如设计成处理整型数据的栈：

```
class Stack
{
public:
    void push( int I );            // 压入元素
    int pop();                     // 弹出元素
private:
    enum { size = 100 };           // 栈的初始容量
    int m_array[ size ];           // 保存元素的数组
    int m_count;                   // 元素个数
};
```

但这个类只能用于处理 int 型数据，或者其他与 int 型数据存在转换关系的数据。如果开发者想要用这个 Stack 类来处理图形类（Shape）对象，则不得不另写一个专门用于处理 Shape 对象的栈。为了区别两个类，可以给新类起名为 ShapeStack，或者其他的名字。显然，这不是一个好的解决方法，因为还有很多类型需要用栈来处理。

另外，上述的解决方法也会带来大量的代码冗余。原因也是显而易见的，对于大多数类型来讲，栈的处理逻辑是一样的。如果对不同的类型都要实现一个栈类，就要将相同的压栈、出栈代码重复

多次，而其中不同的仅仅是元素的类型。

如果将 Stack 类改成类模板，则会省去很多麻烦，其定义如下：

```
template< typename  T >
class Stack
{
public:
    void push( T  e );              // 压入元素
    T pop( );                       // 弹出元素
private:
    enum { size = 100 };            // 栈的初始容量
    T m_array[ size ];              // 保存元素的数组
    int m_count;                    // 元素个数
};
```

当需要一个处理指定类型元素的 Stack 类时，只要用该类型实例化 Stack 模板即可，如下所示：

```
Stack<int>  s1;             // 声明一个可以处理 int 型数据的栈对象
Stack<double>  s2;          // 声明一个可以处理 double 型数据的栈对象
Stack<Shape>  s3;           // 声明一个可以处理 Shape 对象的栈对象
```

在上述代码中，依次用特定的数据类型实例化类模板 Stack，得到了三个用以处理特定类型数据的栈类，并分别用这个栈类声明了一个栈对象。简而言之，使用类模板的主要优点是：

- 一个类模板可以处理不同类型的数据。
- 提高程序员的开发效率，不需要为不同的类型去写相同的逻辑。

类模板是 C++语言的一个非常重要的特征，也代表着某种编程趋势——泛型编程。在 C++的标准模板库（Standard Template Library，简称 STL）中就提供了大量的类模板，如 list，deque，map 等。有关 STL 的详细内容将在后面的相关章节中详细说明。

18.2 定义类模板

类模板的定义语法同函数模板的定义语法非常相似。在学习定义类模板时，读者可以参考定义函数模板的经验。学习时要多注意比较，以便加深印象。

18.2.1 语法

一个类模板的定义以关键字 template 作为开始，后面跟一对尖括号，尖括号的里面是模板参数列表，模板参数之间用逗号分隔。如果是类型参数，则以 typename 或 class 声明；如果是非类型参数，则其声明语法同普通函数参数的声明语法类似。在模板参数列表之后，可以像定义类一样定义类模板。其语法如下：

```
template <typename 参数名 1，typename 参数名 2，……>
```

```
class 类模板名
{
    // 类体
};
```

类模板定义最后这个分号不可省。同类的定义一样，类模板的定义也是一条语句，需要一个分号作为语句的结束。

定义类模板的成员函数时，可以直接在类模板的内部定义，也可以在类模板的外部定义。在类模板内部定义时，其语法同定义普通类的成员函数一致。在类的外部定义时，需要声明这个类模板的参数。其语法如下：

```
template< 模板参数列表 >
返回类型  类模板名< 模板参数列表 >::函数名( 函数参数列表 )
{
    函数体
}
```

其中，在类模板名后面的“模板参数列表”可以省略类型参数的 typename（或 class）关键字，以及非类型参数的类型关键字。例如定义 Stack 模板类的 push 和 pop 函数，可以分别在类的内部和外部定义，代码如下：

```
template< typename  T >
class Stack
{
public:
    Stack()                          // 构造函数
        : m_count(0)                 // 初始化元素个数
        { }
    Void push( T  e )                // 压入元素
    {
        m_array[ m_count++ ] = e;    // 保存元素
    }
    T pop( );                        // 弹出元素
private:
    enum { size = 100 };             // 栈的容量
    T m_array[ size ];               // 保存元素的数组
    int m_count;                     // 元素个数
};
template<typename T>                 // 在类模板的外部定义类模板的成员函数
T  stack<T>::pop()
{
    return m_array[ --m_count ]
}
```

在定义类模板的数据成员和函数成员时，可以使用模板参数。其中的类型参数可以用来声明成员数据的类型，可以作为成员函数的返回类型、参数类型以及成员函数定义时的局部变量类型。非类型参数则可以作为常量使用。例如，定义一个可以存放各种数据的数组类模板：

```
// 定义类模板，模板参数包括类型参数 T 和非类型参数 size
template<typename T, int size>
class Array                 // 类模板名
{
public:
    T & operator[](int i)   // 模拟数组操作的取元素运算符“[ ]”
    {
        return m_ary[i]; // 返回第 i 个元素的引用。代码简化了，未处理 i 超出范围的情况
    }

    void set( int i, const T &e )  // 设置第 i 个元素的值为 e
    {
        m_ary[i] = e;              // 这里简化了代码，未处理 i 超出范围的情况
    }

    T sum( )                       // 求元素之和
    {
        T sum;                     // 用类型参数定义临时变量
        for( int i=0; i<size; i++ )// 遍历数组求和
        {
            sum += m_ary[i];
        }
        return sum;                // 返回元素的和
    }
private:
    T m_ary[size];                 // 内置数组，用类型参数和非类型参数定义
};
```

如果在定义类模板时，需要该模板的一个实例，并且该实例的参数就是本模板的参数，则可以直接使用类模板名。例如，在类模板 Stack 中定义指向另一个 Stack 的成员指针，以及在复制构造函数中用另一个 Stack 作为参数，可以写成如下的形式：

```
Stack ( Stack<T>&  s );  // 可以写成 Stack (Stack& s);
Stack<T>*  m_ps;         // 可以写成 Stack* m_ps;
```

18.2.2 非类型参数

同函数模板一样，类模板也可以使用非类型参数。非类型参数通常在模板定义过程中作为常量使用。例如上一节类模板 Array 的第二个模板参数 size 即一个非类型的参数，并且该参数用来设定数组的长度。

众所周知，定义数组时不能用变量指定其长度，而只能用常量。而模板的非类型参数就是一个常量，所以可以用来表示数组的长度。

声明一个类模板的非类型参数同声明一个函数模板的非类型参数的语法一致，即：

```
<……, 类型  参数名, ……>
```

对于带有非类型参数的类模板，实例化时需要用一个常量来指定这个参数，例如：

```
Array< int, 100 >       aryA;           // 用字面常量设定非类型参数
Array< double, 200 >    aryB;
const  int  size = 50;
Array< char, size >     aryC;           // 用符号常量设定非类型参数
enum  ESize
{
    s1 = 100,
    s2 = 200,
};
Array< float,  s1 >     aryD;           // 用枚举常量设定非类型参数
```

18.2.3 模板参数的默认实参

与函数模板不同的是，定义类模板时，可以设定其参数的默认值，即参数的默认实参，这包括默认类型参数和默认非类型参数。其语法同设定函数的默认实参相同。例如，对于 Array 模板，设定其默认保存的数据类型为 int 型，并且其数组尺寸的默认值为 100：

```
template< typename T = int, int size = 100 >  Array
{
    ……
};
```

在实例化一个 Array 模板时，就可以不指定其模板参数的实参，如下所示：

```
Array<>  aryInt;
```

类模板实例化时，包围实参列表的尖括号**<>**不可省略。

18.3 生成类模板的实例

类模板只是一个模板，不是实际的类。使用类模板时，必须先实例化，即为模板参数赋值，包括类型参数和非类型参数。本节将详细讨论如何实例化类模板。

有的文献将类模板实例化的结果称做模板类，读者在阅读的时候注意区分这两个概念。类模板是模板，而模板类则是模板实例化的结果——是真正可以用来声明对象的类。

18.3.1 类型参数的模板实例化

实例化类模板时，对于类型参数，可以用 C++内建的类型实例化，也可以用自定义的类型。与函数模板不同的是，类模板在实例化时没有参数推导机制，所有的模板参数必须由开发者手工指定，

除非模板参数带有默认实参。例如实例化类模板 Array：

```
const  int  size = 100;          // 常量
Array< int , size >      aryA;   // 用 int 和 size 实例化类模板
Array< double,  200 >    aryB;   // 用 double 和字面常量实例化模板
Array< char,  300 >      aryC;   // 用 char 和字面常量实例化模板
Array<>                  aryD;   // 用类模板的默认值实例化模板
class SomeClass                  // 自定义类型
{
    ……
};
Array< SomeClass,  400 >  aryD; // 用自定义类型实例化模板
```

实例化类模板时，编译器用开发者指定的模板实参代替模板定义中的参数，创建出一种新的“类”实例，即一个类。开发者可以用这个新的类去定义或声明一个对象。

如果用自定义的类型来实例化一个类模板，则对自定义类型有一定的要求。因为定义类模板时，实际上也对其类型参数做出了某些假设，例如定义 Array 类模板时，假设其元素的类型支持赋值操作（如设置元素的 set()函数）以及相加操作（如求和的 sum()函数）。因此，实例化时自定义的类型也需要满足这些假设。

类模板实例可以当做普通的类使用，用其定义的对象同用一般类定义的对象在使用上没有什么差别。需要注意的是，类模板只在需要的时候才实例化，如果程序中只是使用类模板的某个指针或者引用，而没有通过这个指针或引用访问对象的数据成员或函数成员，则该类模板不会实例化。例如，在下面的函数中，类模板并不会被实例化：

```
void Print(Stack<int>& vi)
{
    Stack<int>* pvi = &vi;      // Stack<int>并不会被实例化
}
```

但是如果调用它们的成员函数或访问成员变量，则模板会被实例化：

```
void Print(Stack<int>& vi)
{
    Stack<int>* pvi = &vi;
    cout << pvi->Pop() << endl;// 模板会被实例化
}
```

18.3.2 非类型参数的模板实例化

对于模板的非类型参数，在实例化时只要指定其实参即可。但需要注意的是，这个实参必须是一个常量，例如：

```
template<int* ptrInt> CRect{};
template<int size> CRectangle{};
const int c_size = 10;
int size = 10;
CRect<new int[10]> rect;           // 错误，new int[10]只有在运行时才能被计算
```

```
CRectangle<10> rectangleA;          // 正确，10 是常量
CRectangle<c_size> rectangleB;      // 正确，c_size 是常量表达式
CRectangle<size> rectangleC;        // 错误，size 的值在编译期不能被计算
CRect<&size> rectA;                 // 正确，地址是在编译期确定下来的
```

18.3.3 类模板范例

在实际开发中，类模板经常用做一个可以处理各种类型数据的容器，比如一个先入后出的栈。如果将栈定义成一个类模板，则可以用来处理各种数据，方便使用。下例定义一个 Stack 类模板，其模板参数包括类型参数和非类型参数，并用这个类模板来处理不同类型的数据，程序如示例代码 18.1 所示。

示例代码 18.1

```
#include <cstdlib>
#include <iostream>
using namespace std;              // 使用名称空间 std
template<class T, int size = 100>// 带有非类型参数的类模板，设置 size 的默认值为 100
class Stack
{
public:
    Stack()                       // 构造函数
        : m_iCount(0)             // 初始化元素个数
        { }
    bool push(const T &e);        // 放入元素
    bool pop(T &e);               // 弹出元素
    int get_size () const         // 返回元素个数
    {
        return m_iCount;
    }
private:
    T m_iArray[size];             // 栈中的元素用数组来存储
    int m_iCount;                 // 当前栈中元素的个数
};
template<class T, int size>
bool Stack<T, size>::push(const T &e)
{
    if (m_iCount < size)          // 如果栈还没满
    {
        m_iArray[m_iCount++] = e;          // 加入元素到数组中
        return true;
    }
    else
    {
        return false;
    }
}
template<class T, int size>
bool Stack<T, size>::pop(T &e)
```

```
{
    if (m_iCount >= 0)                  // 如果栈中有元素
    {
        e = m_iArray[--m_iCount];       // 弹出元素
        return true;
    }
    else
    {
        return false;
    }
}
int main(int argc, char *argv[])
{
    cout<<"——类模板实例化——"<<endl;    // 输出提示信息
   Stack<int> iv;      // 用 int 型和默认的非类型参数值实例化类模板 Stack
    iv.push(1); cout<<"压入 1"<<endl;   // 依次压入三个整数
    iv.push(2); cout<<"压入 2"<<endl;
    iv.push(3); cout<<"压入 3"<<endl;
    cout << "元素个数: ";               // 输出当前栈中的元素个数
    cout << iv.get_size() <<endl<<endl;

    cout << "弹出元素：";
    int e = 0;
    iv.pop( e );                        // 弹出元素
    cout << e << endl;
    cout << "弹出元素：";
    iv.pop( e );
    cout << e << endl;
    cout << "元素个数: ";
    cout << iv.get_size() << endl;      // 输出当前栈中的元素个数
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译并运行，结果如图 18.1 所示。

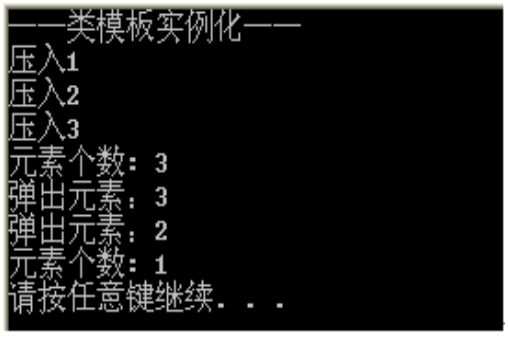


图 18.1 类模板实例化结果

代码中声明了一个 vector 类，push 函数和 pop 函数分别用来向类中添加元素和弹出元素。

18.4 类模板的静态成员

和普通类一样，模板也可以有自己的静态成员。所不同的是，每种类型的类模板的实例都有自己的一组静态成员。和普通类的静态初始化一样，类模板的静态成员也需要在类内声明，在类外定义。其在类外定义的格式如下：

```
template <类型名 参数名1，类型名 参数名2，……>
类型名 类名<类型名 参数名1，类型名 参数名2，……>::静态成员变量 = 初始化值;
```

下面来修改一下 vector 类：

```
template<class T, int size>
class vector
{
public:
    vector() : m_iCount(0)
    {
        sNumber ++;                    // sNumber 加 1
        cout << vector::sNumber << endl;
    }
    ~vector(){}
……
private:
        static int sNumber;            // 被实例化的 vector 对象个数
……
};
template <class T, int size>
int vector<T, size>::sNumber;
                        // 编译器会自动赋初值为 0，等价于 vector<T>::sNumber = 0
```

同样，也可以声明常静态成员：

```
template<class T >
class vector
{
public:
    vector() : m_iCount(0){}
    ~vector(){}
……
private:
    const static int size = 100;          // 第一种实例化 size 的方式
……
};
//template<class T>
//const int vector<T>::size = 100;        // 第二种实例化 size 的方式
```

采用第一种实例化 size 的方式可以替换掉模板中的 size 参数，可以把其视为内建的宏，第二种方式则不行。这是因为编译器对两种定义采用不同编译方式造成的。下面来看一个实际的例子，该例子是坦克计数程序，CTankManager 用不同类型的 Tank 来实例化，并统计这种类型坦克的数量，程序如示例代码 18.2 所示。

示例代码 18.2

```
#include <iostream>
#include <tchar.h>
using namespace std;
template<class T>
class CTankManger
{
public:
    CTankManger(){};
    ~CTankManger();
    bool PushTank(T& tank);
    void OutputTankNumber();
private:
    enum{size = 10};
    static int siCount;
    T mTankArray[size];
};
template<class T>
CTankManger<T>::~CTankManger()
{
}
emplate<class T>
int CTankManger<T>::siCount;
template<class T>
bool CTankManger<T>::PushTank(T& tank)
{
    if(siCount < size)
    {
        cout << "Push tank type :";
        tank.Print();
        mTankArray[siCount] = tank;
        ++siCount;
        return true;
    }
    return false;
}
template<class T>
void CTankManger<T>::OutputTankNumber()
{
    cout << siCount << endl;
}
class CT91
{
public:
    void Print(){
        cout << "T91 tank!" << endl;
    }
};
class CT20
{
public:
    void Print(){
        cout << "T20 tank!" << endl;
    }
};
class CT21
{
public:
    void Print(){
```

```
        cout << "T20 tank!" << endl;
    }
};
int main()
{
    CT91 tank910, tank911, tank912;
    CTankManger<CT91> tm91;
    tm91.PushTank(tank910);
    tm91.PushTank(tank911);
    tm91.PushTank(tank912);
    tm91.OutputTankNumber();
    CT20 tank200, tank201;
    CTankManger<CT20> tm20;
    tm20.PushTank(tank200);
    tm20.PushTank(tank201);
    tm20.OutputTankNumber();
    CT21 tank210, tank211;
    CTankManger<CT21> tm21;
    tm21.PushTank(tank210);
    tm21.PushTank(tank211);
    tm20.OutputTankNumber();
    return 0;
}
```

建立一个控制台工程以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 18.2 所示。

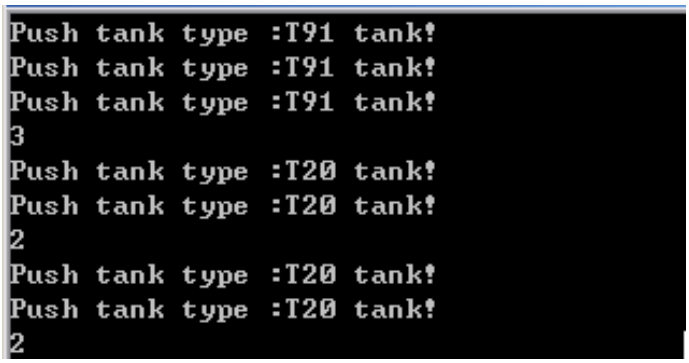


图 18.2　输出每种类型的坦克数量结果

程序中的 CTankManager 是一个管理类，用来管理放在其中的坦克，但是每次它只能管理一种类型的坦克，其中的 siCount 是坦克的计数器，每种类型的坦克都有其自己的计数器。CT91，CT20，CT21 是具体的坦克类型。

18.5 类模板的友元

类模板的友元和普通类的友元一样，使其友元类或者友元函数拥有存取的特权，即可以访问到类中私有或保护的成员和方法。有三种友元可以出现在类模板中：

- 非模板的友元类和友元函数。
- 与模板类型不绑定的友元类和友元函数模板。
- 与模板参数绑定的友元类模板和友元函数模板。

类模板可以接受一个非模板的类或者函数作为友元。对于这种友元类和友元函数，不需要在声明友元前提前声明。由于类模板只在需要的时候才实例化，因此在友元类或者友元函数访问类模板的成员时，类模板已实例化，而此时友元类或友元函数已知，所以非模板的类或者函数不需要提前声明。例如，在如下类模板中声明一个友元类和友元函数：

```
template <class T>
class vector
{
    friend class CDisplay; // 声明 CDisplay 为友元类之前不需要有实现或提前的声明

    friend void Display(); // 声明 Display 为友元函数之前不需要有实现或提前的声明

    ……
};
class CDisplay
{
    ……
};

void Display()
{
    ……
}
```

类模板的友元也可以是类模板或函数模板的实例。如果后者的模板实参与前者的模板参数没有任何关系，也就是说友元模板与类模板的参数不绑定，则在声明友元前也不需要提前声明，其原因同使用非模板的友元一样。例如：

```
template <class Type>
```

```
class vector {

    template<class U>

        friend class CDisplay();           // 之前不需要有实现或提前的声明

    template<class V>
        friend void Display(vector<T>);    // 之前不需要有实现或提前的声明
//……
};
```

如果类模板的友元是类模板或者函数模板的实例，而且其模板实参就是类模板的参数，则友元模板必须提前声明。例如：

```
template <class T>

   class CDisplay{};

template <class T>

   void Display(vector<T>);

template <class T>

class vector

{

    friend class CDisplay<T>;              // 之前需要有实现或提前的声明


    friend void Display<T>(vector<T>);     // 之前需要有实现或提前的声明


//……


};
```

在一个模板被用在一个模板类中作为友元时，其声明或实现必须提前给出。这里的 Display 函数有些特别，记住 Display 后面一定要跟<T>，否则写成如下形式：

```
friend void Display(vector<T>);
```

编译器会把它当成非模板函数。

18.6 类模板的特化

有的时候某些类型不能直接用来实例化类模板，或者说直接实例化不能满足需要，此时就要针对这种类型进行特化，包括特化（即全特化）和偏特化。

18.6.1 类模板的全特化

类模板的特化是为了针对特殊的类型进行特殊的处理。现在为 vector 类添加新的功能，即返回最大值的 max 函数：

```
template<class T>
class vector
{
public:
//……
    T max();
};

template<class T>
T vector<T>::max()
{
    T max = m_iArray[0];
    for (int i = 0; i < m_iCount; ++i)
    {
        if (max < m_iArray[i])
        max = m_iArray[i];
    }
    return max;
}
```

这时发现，max 函数并不具备通用性，因为假定 T 类型是内建类型，或是重载了 operator< 的自定义类型，但是这样是不对的。如果比较的是字符类型，那又该如何呢？答案就是模板的特化。

```
template < >
class vector<const char*>                    // 注意，这时候 size 要用具体的常量值替换掉
{
public:
    vector<const char*>();                   // 特化版的构造函数
    const char* max();                       // 返回 ACSII 值最大的字符串
//……
};
template<>                                   // 注意在 Visual Studio 中需要去掉这一行
vector<const char*>::vector() : m_iCount(0)
{}
template<>                                   // 在 Visual studio 中，不需要加
const char* vector<const char*>::max()
{
    const char* max = m_iArray[0];
    for (int i = 0; i < m_iCount; ++i)
    {
        If(std::strcmp(max, m_iArray[i]) < 0)  // 调用标准函数库 API 进行比较
            max = m_iArray[i];
   }
    return max;
}
int main()
{
    vector<const char*> vc;                  // 实例化特化版的 vector
    return 0;
```

```
}
```

特化时 template 后面的尖括号中是不需要任何参数的。同时，要在类声明的后面写下具体的特化类型。

一定要先声明非特化模板，否则会有编译错误。在 Visual Studio 中，要去掉 template<>声明，否则会报错误“error C2910: 'vector<const char*>::{ctor}' : cannot be explicitly specialized”。

18.6.2 类模板的偏特化

类模板的偏特化是指根据模板的某些参数但不是全部参数进行特化。回想一下 CPoint 类，模板参数列表里面有两个参数 x 和 y。假如有一天有一个需求，要求对在 y 坐标等于 80 的点进行一些特别处理，这个时候就需要对模板参数 y 进行常量化：

```
template<int x>                          // 偏特化声明
class CPoint<x, 80>                      // 将一个参数用常量代替
{
public:
        CPoint() : m_X(x), m_Y(80){}     // 初始化成员变量
// 对 y 坐标在 80 的点进行特别处理
private:
        int m_iX;
        int m_iY;
};
```

部分特化的模板实参表只列出那些模板实参中仍然未知的参数。来看一个更复杂一点例子，回到 vector 类，来看一下如何偏特化 vector 类：

```
template<int size>                           // 偏特化声明
class vector<const char*, size>              // 第一个模板参数是具体的类型
{
public:
    vector<const char*, size>() : m_iCount(0)  // 注意 vector 后面的声明方式
    {}
    ~vector<const char*, size>(){}
//.....
private:
    const char* m_iArray[size];              // 数组类型也要换成具体的类型
    int m_iCount;
};
```

用户可以像下面这样使用偏特化的 vector 类：

```
vector<const char*, 10> vc;
```

这个时候 size 不许有默认的参数。

18.6.3 类模板的匹配规则

类模板的匹配遵循使用“特化程度最高”的模板规则。具体点说，模板参数最准确匹配的拥有最高的实例化优先权（即被编译器优先选择来实例化）。先看下面三种：

```
template <class T> class vector{};              // 普通型
template <class T> class vector<T*>{};          // 对指针类型特化
template<class T> class vector<const T*>{};     // 对 const 型指针特化
template <>   class vector <void*>{};           // 对 void*进行特化
```

根据 vector 不同的模板参数，会寻找最匹配的版本进行实例化，如果是指针类型，会选择第二行定义的模板来实例化，如果一个类型的指针是空类型，就会选择第四行的模板进行实例化，常量类型的指针则会选择第三行的模板。看下面更完整一点的例子：

```
template<class T, class U>
class CTest                                 // 带有两个模板参数的类模板声明
{
public:
    void f() {
    cout << "主模板" << endl;
    }
};
// 强调第一个模板参数是 int 的偏特化类型
template<class U> class CTest<int, U>{
public:
    void f() {
    cout << "T 等于 int" << endl;
    }
};
// 强调第二个模板参数是 double 的偏特化类型
template<class T> class CTest<T, double>{
public:
    void f(){
    cout << "U 等于 double" << endl;
    }
};
// 强调第一个模板参数是指针类型的偏特化类型
template<class T, class U> class CTest<T*, U>{
public:
    void f() {
    cout << "使用了 T*" << endl;
    }
};
// 强调第二个模板参数是指针类型的偏特化类型
template<class T, class U> class CTest<T, U*>{
public:
    void f(){
    cout << "使用了 U*" << endl;
    }
};
```

```
// 强调两个模板参数都是指针类型的偏特化类型
template <class T, class U> class CTest<T*, U*> {
public:
    void f(){
    cout << "使用了 T* 和 U*" << endl;
    }
};
// 强调两个模板参数类型相同的偏特化类型
template<class T> class CTest<T, T>{
public:
    void f(){
    cout << "T 等于 U" << endl;
};
int main()
{
    CTest<float, int> ct1;
    ct1.f();                            // 主模板
    CTest<int, float> ct2;
    ct2.f();                            // T 等于 int
    CTest<float, double> ct3;
    ct3.f();                            // U 等于 double
    CTest<float, float> ct4;
    ct4.f();                            // T 等于 U
    CTest<float*, float> ct5;
    ct5.f();                            // 使用了 T*
    CTest<float, float*> ct6;
    ct6.f();                            // 使用了 U*
    CTest<float*, int*> ct7;
    ct7.f();                            // 使用了 T* 和 U*
    return 0;
}
```

除了 Main Template 以外，其余模板或是特化的参数顺序不对，或是拥有指针类型，或是需要类型相同的参数；ct2 第一个模板参数是 int 类型，第二个为 float 类型，即两个模板参数类型不一样，同时并不拥有指针，所以只有 template<class T, class U>和 template<class U> class CTest<int, U>可以参考，但是由于后者“特化程度最高”，所以被编译器选中。同理可知 ct3 会调用 template<class T> class CTest<T, double>。ct4 由于两个模板参数相同，所以 template<class T> class CTest<T, T>最为匹配。ct5 由于第一个参数是指针类型，第二个参数是普通类型，所以 template<class T, class U> class CTest<T*, U>最匹配。同理可知 ct6 和 ct7 最匹配的模板。

但是，如果模板相近，编译器不能区分应该使用哪个模板，则会报编译错误，例如：

```
CTest<int, int> ct8;
t8.f();     // 错误， 无法区分 template<class U> classCTest<int, U>和
            // template<class T> class CTest<T, T>
CTest<double, double> ct9;
ct9.f();    // 错误，无法区分 template<class T> class CTest<T, double>和
            // template<class T> class CTest<T, T>
CTest<float*, float*> ct10;
ct10.f();   // 错误，无法区分 template <class T, class U> class CTest<T*, U*>和
```

```
            // template<class T> class CTest<T, T>
CTest<int, int*> ct11;
ct11.f()    // 错误，无法区分 template<class U> class CTest<int, U>和
            // template<class T, class U> class CTest<T, U*>
CTest<int*, int*> ct12;
ct12.f();   // 错误，无法区分 template <class T, class U> class CTest<T*, U*>和
            // template<class T> class CTest<T, T>
```

程序分析 :ct8 由于 template<class U> classCTest<int, U>和 template<class T> class CTest<T, T>特化程度相近，编译器无法分辨哪个更好；对于 ct9，ct10，ct11，ct12 的错误原因，请参看代码旁边的注释。在写程序的时候一定要避免出现上面模棱两可的错误。

18.7 综合实例

本例将实现一个智能指针，它可以被视为代理类，专门用来管理指针，它需要满足下面的要求：

- 构造和析构，即指针初始化时自动赋为空，智能指针负责删除内部指针所指的对象。
- 在指针赋值时，避免被赋值的指针所指向的内存丢失，造成内存泄漏。
- 使用上要像使用实际的指针一样。

本例的主要步骤如下：

step 1 在构造函数中默认将指针 p 赋值为空，同时初始化 pointee。默认将指针赋为空是怕没有正确初始化智能指针，导致 pointee 是一个野指针。当智能指针之间相互赋值时，内部指针的所有权要转移。

step 2 重载*和->操作符，使在使用智能指针时就像在使用 pointee 指针一样。

step 3 析构函数自动释放内部指针，也就是说，当智能指针超出自己的作用域时，内部指针也会被自动释放。

step 4 赋值运算符中，首先判断是不是自赋值，然后重置内部指针，并且断开原来智能指针的指向。

如果不判断是否是自赋值，直接重置内部指针时，会因为在调用 release 的时候断开内部指向，使得内部指针 pointee 为空，内存发生泄漏。在 main 函数中添加一对大括号，是为了让 CSmartPointer 作为大括号内部的局部变量，在离开大括号后自动被销毁，更清楚地看到结果。

程序如示例代码 18.3 所示。

示例代码 18.3

```
template<class T>
class CSmartPointer
{
public:
    CSmartPointer(T *p = 0);
    CSmartPointer(CSmartPointer& rhs);
```

```
    ~CSmartPointer();
    T& operator*() const;
    T* operator->() const;
    CSmartPointer& operator=(const CSmartPointer& rhs);
    T* get() const;
    T* release();
    void reset(T *p = 0);
private:
    T* pointee;
};
template<class T>
// 自动初始化指针，防止未赋初值
CSmartPointer<T>::CSmartPointer(T *p = 0) : pointee(p)
{}
template<class T>
// 智能指针相互赋值时，内部指针的所有权转移
CSmartPointer<T>::CSmartPointer(CSmartPointer<T> &rhs)
{
    pointee = rhs.pointee;
    rhs.pointee = 0;
}
template<class T>
CSmartPointer<T>::~CSmartPointer() // 管理指针自动析构
{
    if(NULL != pointee)
        delete pointee;
}
template<class T>
T& CSmartPointer<T>::operator *() const
{
    return *pointee;           // 模拟指针的行为，通过*操作符直接取到指针的实际内容
}
template<class T>
// 模拟指针行为，通过->操作符直接取到内部指针的成员
T* CSmartPointer<T>::operator ->() const
{
    return pointee;
}
template<class T>
CSmartPointer<T>&
CSmartPointer<T>::operator =(const CSmartPointer<T>& rhs)
{
    if(this != &rhs)           // 赋值操作时，首先清空传入智能指针所指向的对象，防止
        reset(rhs.release());  // 两个智能指针同时指向同一块内存，造成两次析构错误
    return *this;
}
emplate<class T>
T* CSmartPointer<T>::get() const // 返回内部指针
{
    return pointee;
}
template<class T>
T* CSmartPointer<T>::release() // 返回内部指针，并且断开与原来指针的联系
{
    T *oldPointee = pointee;
```

```
    pointee = 0;
    return oldPointee;
}
template<class T>
void CSmartPointer<T>::reset(T *p /* = 0 */)
                                // 若传入指针和本身指针的指向不相同，则首先释放自己
{                               // 拥有的指针（否则会造成内存泄漏），然后进行赋值
    if(pointee != p)
    {
        delete pointee;
        pointee = p;
    }
}
```

下面是使用智能指针的一个例子：

```
class Node
{
public:
    Node();
    ~Node();
    void printNode();
};
Node::Node()
{
    cout << "Node constructor!" << endl;
}
Node::~Node()
{
    cout << "Node destructor!" << endl;
}
void Node::printNode()
{
    cout << "this is a node." << endl;
}
int main()
{
    {
        CSmartPointer<Node> spNode(new Node());    // 类型 T 的复制构造函数
        // 调用以智能指针为参数的复制构造函数，同时智能指针的内部指针所有权转移
        CSmartPointer<Node> spNode1 = spNode;
        spNode1->printNode();
    }                                   // 作为局部对象的智能指针将在此处被销毁
                                        // 同时自动释放内部的 Node 指针
    cout << "Program will exit." << endl;
    return 0;
}
```

建立一个控制台工程，以及相应的.cpp 文件，并将上述代码复制到文件中，编译并运行，结果如图 18.3 所示。

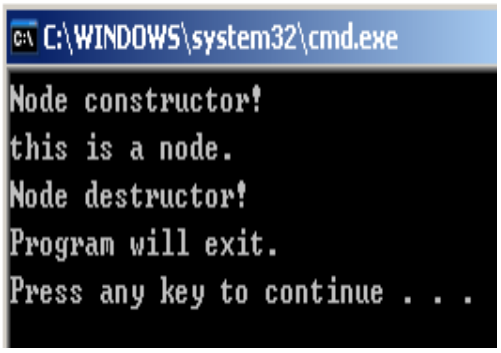


图 18.3　智能指针输出结果

18.8 小结

类模板定义就是指定了怎样根据一个或多个实际类型或值的集合来构造单独的类。C++中的类模板与普通类相似，可以定义自己的成员函数、友元函数和静态成员。模板类的特化有全特化和偏特化两种，要注意在此之前需要普通类模板的定义。

◆　　◆　　◆

第19章 文 件 流

本章包括

- 文件处理的整个过程
- 处理文件流的类
- 打开文件
- 操作文件
- 判断文件流状态
- 重定位读写位置
- 关闭文件

在对文件进行输入输出操作时，除了可以利用操作系统的重定向功能之外，C++也提供了专门的处理文件输入输出的类。利用这些类进行文件操作更加灵活，功能也更加强大。这些类包括用于输出的 ofstream 类，用于输入的 ifstream 类，以及既可用于输出也可用于输入的 fstream 类。上述的文件流类都用于处理编码格式为 ASCII 的文件，至于 Unicode 编码格式的文件则用 wofstream，wifstream 和 wfstream 进行操作。

19.1 文件处理的整个过程

所谓的文件，其实是一个独立的存储数据的单位，通常用一个名字（文件名）来表示。不同于内存中的存储单位，文件是可以永久保存的。即便是计算机断电，文件仍然可以存在。通常来讲，文件可以存储于磁盘、磁带、光盘、闪存盘等各种介质上，只要有相关的设备支持存取即可。

一般来讲，操作文件就是通过程序将文件读入到内存，根据计算序列再将其中的数据转入寄存器，然后由 CPU 进行处理，最终根据需要由程序将数据写回到文件中。其过程如图 19.1 所示。

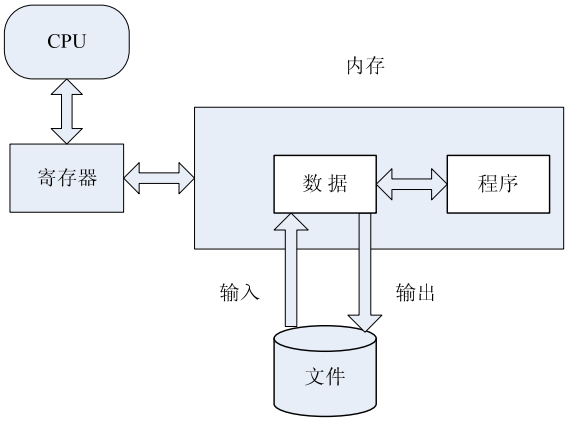


图 19.1　操作文件过程

由于文件存储设备多种多样，各种设备的操作过程差别也很大。如果让用户的程序直接操作这些设备，不仅代码量非常大，而且也不利于程序的移植。假设原有的程序用来处理磁盘上的文件，一旦文件被转移到光盘上，那么程序就要改写，这对用户来讲是非常不利的。因此，当代计算机的各种操作系统都对文件以及存取文件的各种外围设备进行了抽象。不管是存储在什么设备上的文件，用户只要将其看成是一个文件实体即可，具体的输入、输出过程则由操作系统负责。

面对这样一个抽象的文件概念，用户程序在处理文件时，只需遵从如下的步骤即可：

step 1　打开文件。
step 2　读取数据。
step 3　处理数据。
step 4　写回（根据需要）。
step 5　关闭文件。

在文件操作的过程中，还可以根据需要，直接跳转到目标位置进行读写操作。

“打开文件”是一个比较形象的说法。其实际作用是在内存中建立起一个数据结构，该结构与文件相对应。其内容包括文件的名称、起始位置、大小、当前读写位置等信息。通过对该数据结构的处理，就可以达到处理文件的目的。

“关闭文件”与“打开文件”相对应，其作用就是销毁那个与文件相对应的数据结构，回收相关内存。文件被关闭之后，该文件就不能继续读写了，若要读写数据，则必须重新打开文件。

关闭文件与打开文件应当一一对应，一个文件被打开，在处理完毕后应当立即关闭，否则会导致内存泄露，更严重的可能会导致文件的完整性和正确性被破坏。

读写操作是文件处理过程中的关键步骤。读写之前必须要满足一个前提条件，即用户应当了解文件的逻辑结构，包括文件中存有哪些数据、各种数据的类型是什么、数据之间的顺序结构和关联关系是怎样的等。如果不清楚这些内容，那么就无法完成文件的读写。即便完成了读写，读入的内容也不会是用户预期的，如果写入，就会很容易破坏文件的完整性和正确性。文件处理的大体过程如图 19.2 所示。

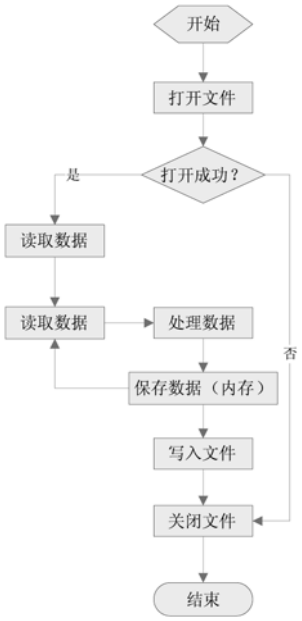


图 19.2 处理文件的大体过程

由于用户指定的文件可能不存在、磁盘空间可能不足等很多问题会导致出错，所以在文件操作的过程中需要注意错误处理。

19.2 处理文件流的类

在 C 语言中，处理文件必须通过一个结构体 FILE 进行，这个结构体的主要内容就是文件名、文件起始位置、大小、当前读写位置等信息。在处理文件的过程中，用户程序必须要通过这个结构体进行，例如打开文件的 fopen 函数，读数据的 fscanf 函数，写数据的 fprintf 函数，关闭文件的 fclose 函数等。至于这些函数的具体使用方法，这里不多做说明，有兴趣的读者可以参考相关文档。下面用一个例子来说明上述函数的使用。

假设在 C 盘中有一个文件，其文件名为 a.txt，其中的内容为：

```
Hello
World
处理文件
```

要求读取该文件的内容，并将其内容输出到屏幕上，程序如下：

```
#include <cstdlib>
#include <cstdio>                                   // 包含头文件
#include <stdio.h>                                  // 也可以使用传统形式的头文件
#include <iostream>
using namespace std;                                // 使用名称空间

int main(int argc, char* argv[])                    // 主函数
{
    FILE *pFile = fopen("C:\\a.txt", "r+");         // 以读写方式打开文件
    if ( NULL == pFile )                            // 判断是否成功
    {
        cout<<"打开文件失败"<<endl;                  // 输出失败信息
        system("PAUSE");
        return EXIT_SUCCESS;                        // 返回
    }
    char str[256];
    while( EOF != fscanf(pFile, "%s", str ) )       // 读取文件内容
    {
        cout<< str <<endl;                          // 输出
    }
    fprintf( pFile, "END" );                        // 在文件尾部加上字符串
    fclose( pFile );                                // 关闭文件

    system("PAUSE");                                // 暂停
    return EXIT_SUCCESS;                            // 返回
}
```

在上述程序中，首先以读写的方式打开一个文件，并对打开文件是否成功进行了判断和处理。然后读取文件中的内容，并将其输出。输出完毕之后，上述程序还在文件末尾附加了一个字符串“END”。最后，关闭文件。程序的运行结果如图 19.3 所示。

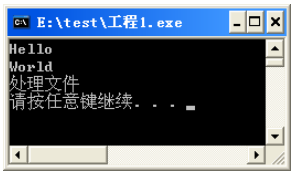


图 19.3　读写文件结果

打开文件有多种方式，包括只读方式、只写方式、读写方式等，在打开文件时应当根据文件处理的目的进行选择。另外，文件的打开方式也限制了文件的一些操作，例如以只读方式打开的文件就不可写，而以只写方式打开的文件则不可读。

事实上，通过 FILE 结构体对文件进行操作并不是很理想，主要是因为这么做并不符合面向对

象的编程思想。因此，C++提供了一系列专门用于处理文件的文件流类，这些类包括专门用于从文件中输入数据的 ifstream 类，专门用于向文件输出数据的 ofstream 类，以及既可输入也可输出的 fstream 类，如图 19.4 所示。

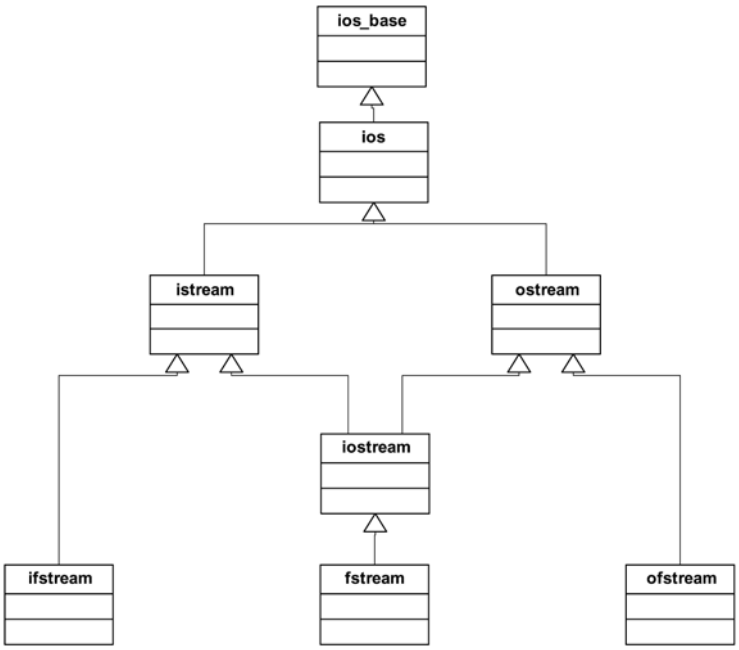


图 19.4 文件流类

文件编码有 ASCII 码和 Unicode 编码之分，相应的处理文件的类也有这样的分别。上述的 ifstream，ofstream 和 fstream 类都是用于处理 ASCII 码编码文件的。如果要处理 Unicode 编码的文件，则应当使用对应的 wifstream，wofstream 和 wfstream 类。为简单起见，本书中的文件流类都使用 ASCII 码格式的版本。

在本章后面的小节中将详细介绍这些文件流类的使用，这里先通过改造前面文件读写例程，简单展示一下使用文件流类编程的过程。

```
#include <cstdlib>
#include <fstream>                              // 包含头文件
#include <iostream>
using namespace std;                            // 使用名称空间

int main(int argc, char* argv[])                // 主函数
{
    fstream file;                               // 构造文件流对象
    file.open("C:\\a.txt");                     // 以读写方式打开文件
```

```
    if ( file.fail() )                          // 判断是否成功
    {
        cout<<"打开文件失败"<<endl;              // 输出失败信息
        system("PAUSE");
        return EXIT_SUCCESS;                    // 返回
    }
    char str[256];
    while( file >> str )                        // 读取文件内容
    {
        cout<< str <<endl;                      // 输出
    }
    file<<"END";                                // 在文件尾部加上字符串
    file.close();                               // 关闭文件

    system("PAUSE");                            // 暂停
    return EXIT_SUCCESS;                        // 返回
}
```

上述程序的基本步骤如下：

step 1 使用 fstream 文件流类构造对象 file，然后调用该对象的 open 方法打开一个文件。这里没有指定文件打开的方式，这是因为对于 fstream 类（既支持输入又支持输出），默认的文件打开方式就是读写方式。调用 open 方法之后，还可以调用 fail 函数，检查打开文件是否成功。

step 2 调用 file 的输入运算符“>>”进行数据的输入。该运算符返回的是文件流对象的引用。在 while 语句中，可以通过调用该对象的类型转换运算符“operator bool()”来判断是否已经到达文件末尾。

step 3 输入完毕之后，还可以调用 file 对象的输出运算符“>>”向文件中写入数据。因此，文件流的使用同一般数据流的使用基本一致。

step 4 调用 file 的 close 函数将文件流关闭。不过，这一步并不是必需的，因为文件流对象在析构的时候会自动调用 close 函数关闭文件流。

上述程序的运行结果如图 19.5 所示。

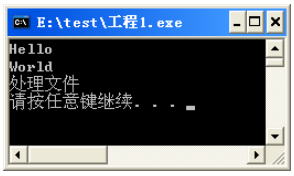


图 19.5 使用文件流对象读写文件结果

19.3 打开文件

处理文件的第一步是打开文件。打开文件涉及了几个方面的问题，一是打开文件的方式，二是

打开文件的函数，三是需要处理文件不存在、磁盘空间不够等问题。

19.3.1 打开文件的函数

处理文件的第一步是打开文件，用文件流类打开文件有两种方式，一种是利用类的构造函数，另一种是调用成员函数 open。这两种打开文件的方法在效果上是相同的。所有的文件流类都具有两个版本的构造函数，即一个不带参数的默认构造函数和一个带参数的构造函数。构造函数中的参数指定了目标文件以及文件打开的模式，如下所示：

```
// fstream 的构造函数
fstream ( );
explicit fstream ( const char * filename,                  // 目标文件
            ios_base::openmode mode = ios_base::in | ios_base::out );
                                                           // 打开模式

// ifstream 的构造函数
ifstream ( );
explicit ifstream ( const char * filename,                 // 目标文件
             ios_base::openmode mode = ios_base::in );     // 打开模式

// ofstream 的构造函数
fstream ( );
explicit fstream ( const char * filename,                  // 目标文件
            ios_base::openmode mode =ios_base::out );      // 打开模式
```

其中带参数的构造函数可以直接打开目标文件。注意其第二个参数是打开文件的模式，用数据流基类 ios_base 的数据成员指定。一般来讲每种文件流都有其固定的打开模式，例如输入文件流 ifstream 就是 ios_base::in，输出文件流 ofstream 就是 ios_base::out。即便是编程时给错了打开模式也不会出问题。例如，构造输入文件流对象时，用的打开模式却是 ios_base::out，这种情况下并不会影响文件的读入，同时也不要企图向文件中写入数据，因为输入文件流 ifstream 没有输出函数。

使用构造函数打开文件的程序如下：

```
#include <cstdlib>
#include <fstream>                                  // 包含头文件
#include <iostream>
using namespace std;                                // 使用名称空间

int main(int argc, char* argv[])                    // 主函数
{
    ofstream outfile( "target.txt", ios_base::out );   // 以写模式打开文件，
    ifstream infile( "source.txt", ios_base::in );     // 以读模式打开文件
    fstream file( "file.txt", ios_base::in | ios_base::out );
                                                        // 以读写模式打开文件

    outfile.close();                                // 关闭文件
    infile.close();
    file.close();

    system("PAUSE");                                // 暂停
    return EXIT_SUCCESS;                            // 返回
}
```

使用不带参数的构造函数，意味着该文件流对象的目标文件还没有确定。在后面的程序中，可以使用 open 函数打开指定的文件。各个文件流的 open 函数如下：

```
// fstream 的 open 函数
void open ( const char * filename,
          ios_base::openmode mode = ios_base::in | ios_base::out );

// ifstream 的 open 函数
void open ( const char * filename,
          ios_base::openmode mode = ios_base::in );

// ofstream 的 open 函数
void open ( const char * filename,
          ios_base::openmode mode = ios_base::out );
```

比较构造函数，可以看出 open 函数跟带参数的构造函数非常相似，其目的也很相似，都是以指定的模式打开指定的文件。使用 open 函数打开文件的程序如下：

```
#include <cstdlib>
#include <fstream>                                  // 包含头文件
#include <iostream>
using namespace std;                                // 使用名称空间

int main(int argc, char* argv[])                    // 主函数
{
```

```
    ofstream outfile;
    outfile.open( "target.txt", ios_base::out );   // 打开文件

    ifstream infile;
    infile.open( "source.txt", ios_base::in );     // 同上

    fstream file;
    file.open( "file.txt", ios_base::in | ios_base::out ); // 同上

    outfile.close();                               // 关闭文件
    infile.close();
    file.close();

    system("PAUSE");                               // 暂停
    return EXIT_SUCCESS;                           // 返回
}
```

一般来讲，用构造函数打开文件同用 open 函数打开文件在功能上是一致的。但是从程序的逻辑角度来讲，如果一个文件流对象用构造函数打开一个文件，则该对象只处理该文件；如果用 open 函数打开，则该对象还可以用来处理其他文件。

文件操作完毕之后，应当调用文件流对象 close 函数将文件流关闭。如果没有调用，那么当文件流对象析构的时候，文件流也会被关闭。

使用文件流的 open 函数打开文件时有一点需要注意，那就是如果该文件流已经打开了一个文件，那么在打开新文件之前，必须关闭当前文件，否则 open 函数调用会返回失败。例如：

```
#include <cstdlib>
#include <fstream>                                 // 包含头文件
#include <iostream>
using namespace std;                               // 使用名称空间

int main(int argc, char* argv[])                   // 主函数
{
    ifstream file;
    file.open("C:\\a.txt", ios_base::out);         // 以读写方式打开文件
    if ( file.fail() )                             // 判断是否成功
    {
        cout<<"打开文件失败"<<endl;                 // 输出失败信息
        system("PAUSE");
        return EXIT_SUCCESS;                       // 返回
    }
    // file.close();                               // 未调用 close 函数
    file.open("C:\\b.txt");
```

```
    if ( file.fail() )                          // 判断是否成功
    {
        cout<<"打开文件失败"<<endl;              // 输出失败信息
        system("PAUSE");
        return EXIT_SUCCESS;                    // 返回
    }

    file.close();                               // 关闭文件

    system("PAUSE");                            // 暂停
    return EXIT_SUCCESS;                        // 返回
}
```

由于在程序第二次调用 open 函数之前没有调用 close 函数，所以文件流对象 file 仍处于打开文件的状态。因此，该处的 open 调用并不成功，fail 函数将返回 true，表示文件操作出错。运行结果如图 19.6 所示。

图 19.6　错误的 open 调用结果

19.3.2 打开文件的方式

在上述例子中，构造函数和 open 函数的第二个参数用以指明文件的打开模式。其类型是定义在 ios_base 类中的 openmode 类型，其可能的取值如表 19.1 所示。

表 19.1　openmode 的值及含义

Openmode 的值	含义
app	在文件流的最后添加数据
ate	当文件流对象第一次打开时，定位到文件流的最后
binary	以二进制的方式操作文件

in	从文件中输入数据
out	向文件输出数据
trunc	当文件流对象创建时，清除文件中的内容，该文件必须是已经存在的

其中，out 是输出模式，表示只能向文件流中输出数据。在这种模式下，如果目标文件已存在，则文件中已有的内容将会被删除，取而代之的是输出到文件中的内容。如果目标文件不存在，则会创建一个新的文件。

用 ofstream 对象打开文件还有一种附加模式 ios_base::app。在这种模式下，如果目标文件已经存在，则其中的内容不会被删除，而只是在这个文件的后面增加新的内容。在默认情况下，用 ofstream 对象操纵文件，其打开文件的模式是输出模式（ios_base::out）。下面程序中的 4 个文件操作语句是等效的：

```
#include <cstdlib>
#include <fstream>                              // 包含头文件
#include <iostream>
using namespace std;                            // 使用名称空间

int main(int argc, char* argv[])                // 主函数
{
    ofstream outfile ( "target.txt", ios_base::out );
    ofstream outfile ( "target.txt" );
    outfile.open( "target.txt", ios_base::out );
    outfile.open( "target.txt" );

    system("PAUSE");                            // 暂停
    return EXIT_SUCCESS;                        // 返回
}
```

对于 ifstream 对象，打开文件的默认模式是 ios_base::in，所以下面程序中的 4 个文件操作语句也是等效的：

```
#include <cstdlib>
```

```
#include <fstream>                                  // 包含头文件
#include <iostream>
using namespace std;                               // 使用名称空间

int main(int argc, char* argv[])                   // 主函数
{
    ifstream  outfile ( "source.txt", ios_base::in );
    ifstream  infile ( "source.txt" );
    infile.open( "source.txt" , ios_base::in );
    infile.open( "source.txt" );

    system("PAUSE");                               // 暂停
    return EXIT_SUCCESS;                           // 返回
}
```

除了表 19.1 中列出的打开模式外，还可以对其中的模式进行组合，从而创造出新的文件打开模式。组合的方法是将几种模式进行按位或运算，即调用按位或运算符“|”。例如，文件流类 fstream 既可以对文件进行读操作，也可以对文件进行写操作，但是没有“读写”这种模式，因此必须组合出新的打开模式，如下所示：

```
ios_base::in | ios_base::out
```

另外，还可以根据程序的需要，组合出如下的几种文件打开模式：

```
ios_base::in | ios_base::binary                // 输入，并以二进制方式操作文件
ios_base::in | ios_base::out | ios_base::ate   // 在文件末尾进行输入、输出操作
ios_base::app | ios_base::binary               // 只在文件末尾添加二进制数据
```

使用文件流打开文件的方式同传统的 C 语言打开文件的方式基本相同，但表示方法并不相同。在 C++中是用 ios_base 的成员变量表示的，而在 C 语言中，则是用一个字符串表示的，如表 19.2 所示。

表 19.2　C 语言打开文件的方式及含义

打开方式	含义

“r”	从文件中输入数据
“w”	向文件输出数据，如果该文件已经存在，则清空原有的内容
“a”	在文件流的最后添加数据
“r+”	以读写的方式打开一个文件，该文件必须已经存在
“w+”	打开一个空文件，并进行读写操作。如果文件已经存在，则清空文件内容
“a+”	当文件流对象第一次打开时，定位到文件流的最后

同使用文件流一样，C 语言也可以以二进制的方式处理文件。此时就应当在上述的各个字符串中加入一个字符“b”，即“rb”、“wb”、“ab”、 “r+b”、 “w+b”、 “a+b”。对于读写混合的方式，字符“b”也可以放在“+”号之前，但必须在基本字符之后，即“rb+”、“wb+”、“ab+”。

文本方式与二进制方式的主要区别在于对换行符的处理上。当以文本方式读取换行符时，换行符会被转换成“\r\n”、“\n\r”或者“\n”(不同的平台会有不同的结果)；而在二进制方式下，换行符一律被转换成“\n”。

19.4 操作文件

在本节中，将详细讲解如何使用文件流对文件进行操作。常见的操作包括读和写，下面分别详细讲解。

19.4.1 格式化读写

由于 ofstream 从 ostrearn 类派生， ifstream 从 istream 类派生，所有 ostream 和 istream 的操作 ofstream 和 ifstream 类对象也都可以使用。例如：

```
outfile<< "Hello World";    // 用输出文件流对象 outfile 输出数据
infile>> var;               // 用输入文件流对象 infile 输入数据到变量 var 中
```

与 ostream 对象不同，ofstream 对象输出数据的目标是文件而不是屏幕。ifstream 对象也与 istream 对象不同，其数据的来源是文件而不是键盘。

输入运算符“>>”和输出运算符“<<”是有格式的输入输出，也就是说这两个运算符对数据的解析是按照一定的格式进行的。这里所谓的格式指的是文本格式，因为这两个运算符对数据的读写是按照文本格式进行的。当输出一个数据时，先将该数据转换成对应的编码（ASCII 码、Unicode 码等），再将编码后的二进制数据写到文件中。例如，在一个 ASCII 编码的系统中，利用运算符“<<”输出整数 24 的过程如图 19.7 所示。

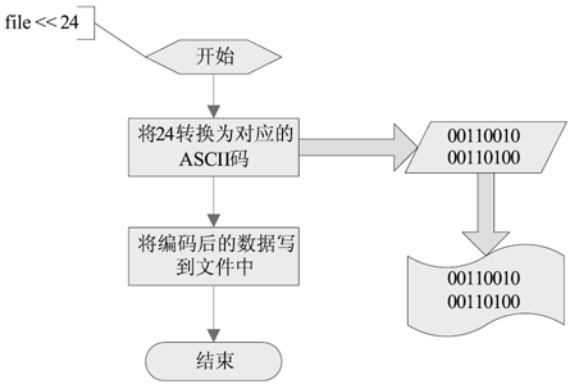


图 19.7　格式化输出整数 24 的过程

当输入一个数据时，先从文件中读入二进制数据，再按照一定的编码格式转换成对应的文本，然后再转换成对应的数据类型。例如，在一个 ASCII 编码的系统中，利用运算符“>>”输入整数 24 的过程如图 19.8 所示。

下例打开一个已经存在的文件，这个文件中保存了 10 个整数；然后读出这些整数，并找出最大值，将结果输出到另外一个文件中；再打开这个结果文件，并将结果输出到屏幕上。程序如示例代码 19.1 所示。

示例代码 19.1

```
#include <cstdlib>
```

```
#include <fstream>                    // 包含文件流头文件
#include <iostream>
using namespace std;                  // 使用名称空间 std

int main(int argc, char *argv[])      // 主函数
{
    ifstream infile( argv[1] );       // 用文件输入流打开第一个参数对应的文件
    if ( !infile )                    // 如果打开不成功
    {
        cout<<"文件"<<argv[1]         // 输出出错信息
            <<"不存在！"<<endl;
        return EXIT_FAILURE;          // 主函数返回
    }

    int x = 0, max = 0;
    for( int i=0; i<10; i++ )         // 读入 10 个数
    {
        infile>> x;
        if( x > max )                 // 找最大值
        {
            max = x;
        }
    }

    ofstream outfile( argv[2] );      // 打开第二个参数对应的文件
    outfile<< max;                    // 将最大值输出到文件中

    cout<<"最大值是："<< max <<endl; // 输出最大值到屏幕

    system("PAUSE");                  // 等待用户反应
    return EXIT_SUCCESS;              // 主函数返回
}
```

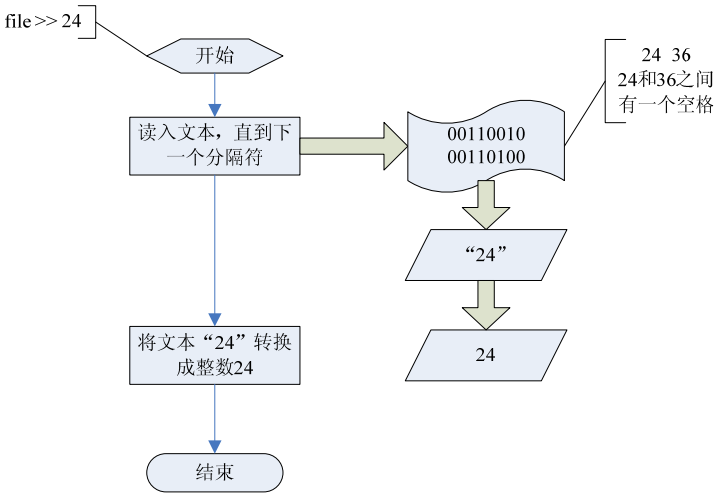


图 19.8　格式化输入整数 24 的过程

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 19.9 所示。

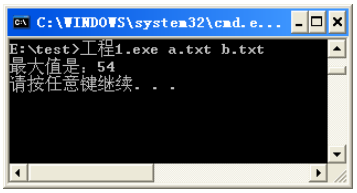


图 19.9　文件的输入输出结果

本例从命令行参数中获取目标文件。第一个参数 a.txt 文件是保存数据的文件，第二个参数 b.txt 文件是保存处理结果的文件。

19.4.2 无格式读数据

区别于“<<”和“>>”的有格式读写，其他文件流的读写函数则是无格式的。在读写过程中，文件流一律将数据当成单个的或连续的字符序列。例如，一个整数就被当成 4 个字符序列，一个双精度浮点数就被当成 8 个字符序列。下面以 get 和 write 函数为例进行说明。

连续的字符序列并不是以特殊字符“\0”结尾的字符串，而纯粹就是一系列字符。为了避免读写越界，通常要指定这个序列的长度。

函数 get 有如下 6 种重载形式：

```
int get();                                    // 读取一个字符，并返回其值
istream& get ( char& c );                     // 读取一个字符
istream& get ( char* s, streamsize n );       // 读取指定长度字符序列，直到分行符
istream& get ( char* s, streamsize n, char delim );
                                              // 读取指定长度字符序列，直到指定的分隔符
istream& get ( streambuf& sb );               // 读取字符序列到流缓冲器，直到分行符
istream& get ( streambuf& sb, char delim );
                                              // 读取字符序列到流缓冲器，直到指定的分隔符
```

其中第三种重载形式比较常用，下面用一个例子进行说明。本例使用文件流的 get 函数从文件中读取一个整数，并存放到一个变量中。为了达到这个目的，可以将变量的地址作为 get 函数的第

一个参数，虽然其类型是 char*，不是 int *，但是可以通过类型转换实现。程序如示例代码 19.2 所示。

示例代码 19.2

```
#include <cstdlib>
#include <iostream>
#include <fstream>                          // 包含头文件
using namespace std;                        // 使用名称空间 std

int main(int argc, char *argv[])            // 主函数
{
    cout<<"————从文件中输入一个整数————"<<endl;
    int x = 0;                              // 保存数据的变量
    fstream file;                           // 文件流对象
    file.open( "C:\\x.txt" );               // 打开文件
    file.get( (char *)&x, sizeof(int) );    // 从文件流中读入整数到变量中
    cout<< "获取整数："<< x <<endl;          // 输出结果
    file.close();                           // 关闭文件

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

建立一个控制台工程，并将上述代码复制到主程序文件中，编译成可执行程序。假设目标文件“C:\\x.txt”中存有文本 24，则运行上述程序的结果如图 19.10 所示。

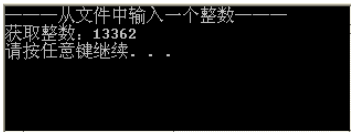


图 19.10 从文件中读入整数结果

目标文件中的文本是 24，其十六进制 ASCII 码为 3234。当将这些数据读入到一个整数变量的地址空间中之后，“32”就成为整数的低字节部分，而“34”则成为整数的高字节部分，转换成十进制就是 13362。

19.4.3 无格式写数据

无格式写数据的函数是 write，该函数没有重载，只有如下一种形式：

```
ostream & write( const char *, streamsize n );
```

由于 write 的第一个参数是 const char *类型，所以在写数据之前，应当先获取数据的地址，并转换成目标类型。在上一节的例子中，文件中保存的是文本 24，而不是整数 24。如果要直接在文件中存储整数 24，则程序应如示例代码 19.3 所示。

示例代码 19.3

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

int main(int argc, char *argv[])
{
    cout<< "————输出整数到文件————"<<endl;

    int x = 24;
    fstream file;
    file.open( "C:\\y.txt" );
    file.write( (char *)&x, sizeof(int) );
    file.close();

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

用记事本打开文件“C:\\y.txt”，如图 19.11 所示。

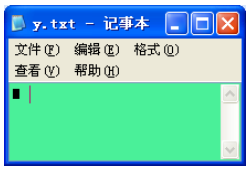


图 19.11　输出数据结果

由于 ASCII 码 24 是一个不可打印的字符，所以显示的结果如图 19.11 所示。如果以十六进制的方式查看该文件，则结果就是：

```
18 00 00 00
```

十进制 24 转换成十六进制就是 18。由于在文件中低位先存，所以 18 出现在开头的位置，其

他的 00 是整数 24 的高位。

19.5 判断文件流状态

在文件流的操作过程中，处理各种可能会发生的错误非常重要，否则会导致程序崩溃。这些错误包括文件未打开、已到文件末尾、读写出错等。

19.5.1 检查文件是否打开

如果有一个文件已经同文件流关联起来，则说明该文件已经被打开，可以进行操作。如果没有，那么就不能进行操作。检查的办法是调用 is_open 函数，打开，则返回 true，否则返回 false。例如：

```
#include <cstdlib>
#include <iostream>
#include <fstream>                          // 包含头文件
using namespace std;                        // 使用名称空间 std

int main(int argc, char *argv[])            // 主函数
{
    fstream file;                           // 文件流对象
    cout<< boolalpha<<file.is_open() <<endl;       // 文件未打开，输出 false
    file.open( "C:\\a.txt" );               // 打开文件
    cout<< file.is_open() <<endl;           // 文件已打开，输出 true
    file.close();                           // 关闭文件
    cout<< file.is_open() <<endl;           // 文件已关闭，输出 false

    system("PAUSE");
    return EXIT_SUCCESS;                    // 返回
}
```

19.5.2 文件流的状态

同一般的数据流一样，文件流也具有 4 种基本状态。每种状态都由一个 ios_base 中的静态数据成员表示，如表 19.3 所示。

表 19.3 流对象状态（iostate）的值及含义

iostate 值	含义
ios_base::eofbit	已经到达文件末尾
ios_base::badbit	流的完整性被破坏
ios_base::failbit	不能读写期望的数据
ios_base::goodbit	流处于正常的状态

在数据流基类 ios 中有一个数据成员用来保存当前的状态，其值可能是上述 4 个值中的一个，也可能是其中几个值按位或的结果。例如，在读写操作过程中，如果失败，数据流的状态可能被设置成 ios_base::badbit | ios_base::failbit。

同一般数据流的操作一样 ,文件流也可以通过调用相应的函数来判断是否处于某种状态。例如，判断是否已到文件结尾的 eof 函数，判断完整性的 bad 函数，判断操作是否失败的 fail 函数，判断是否处于正常状态的 good 函数。

fail 函数检测 baidbit 或者 failbit 是否设置，而 bad 函数则只检测 badbit 是否设置，good 函数则检测 eofbit，badbit 和 failbit 其中之一或者其组合是否设置。

值得注意的是，当用同一个文件流对象处理不同的文件时，需要清除前一个文件操作留下的状态值。否则，在后面文件处理时进行状态判断会导致意外的结果。清除状态值的方法是调用 clear 函数。

例如，在下面的程序中，由于前一个文件读写已到末尾，所以设置了 eofbit 状态值，如果不清除该值，那么在后面读写文件的条件判断时就会出错。

```
#include <cstdlib>
#include <fstream>                                  // 包含头文件
#include <iostream>
```

```
using namespace std;                                    // 使用名称空间

int main(int argc, char* argv[])                        // 主函数
{
    ifstream file;
    file.open("C:\\a.txt", ios_base::out);              // 以读写方式打开文件
    if ( file.fail() )                                  // 判断是否成功
    {
        cout<<"打开文件失败"<<endl;                      // 输出失败信息
        system("PAUSE");
        return EXIT_SUCCESS;                            // 返回
    }
    char str[256];
    while(file>>str)                                    // 输入数据，直到文件结尾
    {
        cout<<str<<endl;
    }
    file.close();                                       // 关闭文件

    // file.clear();                                    // 未调用 clear 函数

    file.open("C:\\b.txt");                             // 打开另外一个文件
    if ( file.fail() )                                  // 判断是否成功
    {
        cout<<"打开文件失败"<<endl;                      // 输出失败信息
        system("PAUSE");
        return EXIT_SUCCESS;                            // 返回
    }

    file.close();                                       // 关闭文件

    system("PAUSE");                                    // 暂停
    return EXIT_SUCCESS;                                // 返回
}
```

运行结果如图 19.12 所示。

图 19.12　未清除状态值导致的错误

19.6 重定位读写位置

操纵文件时，有时需要从特定的位置开始读写。在 C++中，可以通过调用文件流对象的 seekp

和 seekg 成员函数来改变这个位置，文件输出流对象用 seekp 函数，文件输入流对象用 seekg 函数。这两个函数各有两个重载版本，如下所示：

```
basic_ostream& seekp( pos_type _Pos );
basic_ostream& seekp( off_type _Off, ios_base::seekdir _Way );

basic_istream& seekg( pos_type _Pos );
basic_istream& seekg( off_type _Off, ios_base::seekdir _Way );
```

其中_Pos 参数是文件中的绝对位置（其实是从文件起始处开始的）。使用 seekp 和 seekg 函数的第一个版本，表示移动到参数指定的绝对位置。_Way 参数的值是一个枚举值，表示文件中的一些特殊位置，如表 19.4 所示。

表 19.4　文件流中的特殊位置值及含义

位置值	含义
ios_base::beg	文件起始位置
ios_base::cur	文件当前位置
ios_base::end	文件结尾

_Off 参数的值也是一个以字节为单位的整数，表示的是从_Way 参数表示的位置处开始的偏移量，正数表示向前移动，负数表示向后移动。例如：

```
#include <cstdlib>
#include <iostream>
#include <fstream>                      // 包含头文件
using namespace std;                    // 使用名称空间 std

int main(int argc, char *argv[])        // 主函数
{
    fstream file ( "somefile.txt" );    // 以可读写的形式打开文件 somefile.txt
    ……                                  // 某些文件操作改变了文件流对象的当前读写位置
    file.seekp( 16, ios_base::cur );    // 从当前读写位置开始，向前移动 16 个字节
    ……
    file.seekp( -10, ios_base::cur );   // 从当前读写位置开始，向后移动 10 个字节
    system("PAUSE");
    return EXIT_SUCCESS;                // 返回
}
```

假设程序操作完成后，文件流对象 file 的当前读写位置是 100，则运行完第 10 行之后，file 的当前读写位置是 116。如果第 11 行的文件操作完成后，file 的读写位置是 200，则运行完第 12 行之后，file 的当前读写位置是 190。

为了获取当前文件流对象的读写位置，可以调用成员函数 tellp 和 tellg，前者是输出流的成员函数，后者是输入流的成员函数。这两个函数的返回值是绝对位置。例如：

```
file.seekp( 100 );
cout<< file.tellp();
```

上述程序输出 100。

19.7 关闭文件

文件使用完之后，应当将文件关闭（实际关闭的是操纵文件的文件流对象）。关闭的方法是调用文件流对象的 close 成员函数。例如：

```
#include <cstdlib>
#include <iostream>
#include <fstream>                          // 包含头文件
using namespace std;                        // 使用名称空间 std

int main(int argc, char *argv[])            // 主函数
{
    ofstream outfile;                       // 声明一个输出文件流对象
    ……                                      // 使用文件流对象处理文件
    oufile.close();                         // 关闭文件流对象

    ifstream infile;                        // 声明一个输入文件流对象
    ……                                      // 使用文件流对象处理文件
    infile.close();                         // 关闭文件流对象
    system("PAUSE");
    return EXIT_SUCCESS;                    // 返回
}
```

用户也可以不显式地调用 close 函数，当文件流对象析构时，close 函数会被自动调用。

正如在前面的章节中曾经提到的：如果在文件流对象关闭后，继续使用该文件流对象处理其他文件，则需要调用 clear 函数清除当前文件流的状态值。

19.8 综合实例

本节的综合实例将模拟读写电子邮件的过程。假设电子邮件由如下几部分组成：

- 收件人数量，一个二进制格式的整数，记录收件人信箱的数量。
- 收件人信箱。
- 发件人信箱。
- 主题。
- 内容。

19.8.1 模拟生成电子邮件

在程序中，首先以写文件的方式生成一个新的文件，该文件用来保存电子邮件中的信息，文件名来自命令行参数。然后提示用户输入各种必要的信息，用户输入的内容立即写入到文件中，输入 -1 表示该部分内容结束。程序如示例代码 19.4 所示。

示例代码 19.4

```
#include <cstdlib>
#include <iostream>
#include <fstream>                  // 包含头文件
#include <string>
using namespace std;                // 使用名称空间 std

int main(int argc, char *argv[])    // 主函数
{
    assert( argc > 1 );
    ofstream file;                  // 输出文件流对象
    file.open( argv[1] );           // 打开文件
    if( !file )                     // 判断打开文件是否成功
    {
        cout<<"打开文件失败"<<endl;
        system("PAUSE");
```

```
        return EXIT_SUCCESS;
    }
    int count = 0;                  // 记录收件人数量
    file.write( (char *)&count, sizeof(int) );     // 写入
    string str;                                    // 保存键盘输入的内容
        cout<<"——请输入收件人邮箱——"<<endl;
    while(cin>>str && str != "-1")                 // 输入收件人信箱，直到用户输入-1
    {
        file<<str<<' ';                            // 输入收件人信箱到文件中
       count++;                                    // 记录收件人信箱的数目
    }

    long pos = file.tellp();                       // 获取当前的输出位置
    file.seekp(ios_base::beg);                     // 重定位到文件开头
    file.write( (char *)&count, sizeof(int) );     // 写入收件人信箱数目
    file.seekp( pos );                             // 重定位

    cout<<"——请输入发件人邮箱——"<<endl;
    if(cin>>str)                                   // 输入发件人信箱
    {
        file<<str;                                 // 将发件人信箱输出到文件
    }
    cout<<"——请输入主题——"<<endl;
    if(cin>>str)                                   // 输入主题
    {
        file<<str;                                 // 将主题保存到文件中
    }
    cout<<"——请输入邮件内容——"<<endl;
    while(cin>>str && str != "-1")                 // 输入邮件内容，直到用户输入-1
    {
        file<<str<<' ';                            // 将邮件内容保存到文件中
    }

    file.close();                                  // 关闭文件

    system("PAUSE");                               // 暂停
    return EXIT_SUCCESS;                           // 返回
}
```

建立一个控制台工程，将上述程序代码复制到主程序文件中，并进行编译，然后在命令行窗口中运行上述程序，并给定参数 mail，结果如图 19.13 所示。

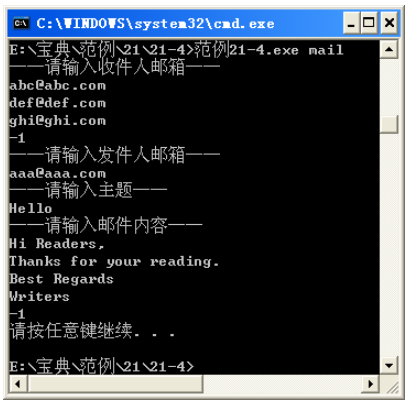


图 19.13　生成邮件结果

生成的文件如图 19.14 所示。

图 19.14　生成的文件

19.8.2　模拟读电子邮件

读取邮件内容并显示到屏幕上的程序如示例代码 19.5 所示。

示例代码 19.5

```
#include <cstdlib>
#include <iostream>
#include <fstream>                          // 包含头文件
#include <string>
using namespace std;                        // 使用名称空间 std

int main(int argc, char *argv[])            // 主函数
{
    assert( argc > 1 );
    ifstream file;                          // 输入文件流对象
    file.open( argv[1] );                   // 打开文件
    if( !file )                             // 判断打开文件是否成功
    {
       cout<<"打开文件失败"<<endl;
       system("PAUSE");
       return EXIT_SUCCESS;
    }
    string str;                             // 保存文件内容的字符串

    int count = 0;                          // 记录收件人信箱个数的整数
```

```
    file.read((char*)&count, sizeof(int)); // 读取收件人信箱个数
    cout<<"——收件人信箱——"<<endl;
    for( int i=0; i<count; i++ )           // 依次读取每个收件人信箱
    {
        file>>str;                         // 输入收件人信箱
        cout<<str<<endl;                   // 输出收件人信箱
    }
    cout<<"——收件人信箱——"<<endl;
    file>>str;                             // 输入发件人信箱
    cout<<str<<endl;                       // 输出发件人信箱

    cout<<"——主题——"<<endl;
    file>>str;                             // 输入主题
    cout<<str<<endl;                       // 输出主题

    cout<<"——内容——"<<endl;
    while(file>>str)                       // 输入内容，直到文件末尾
    {
        cout<<str<<endl;                   // 输出邮件内容
    }
    file.close();                          // 关闭文件

    system("PAUSE");                       // 暂停
    return EXIT_SUCCESS;                   // 返回
}
```

建立一个控制台工程，将上述程序代码复制到主程序文件中，并进行编译，然后在命令行窗口中运行上述程序，并给定参数 mail，结果如图 19.15 所示。

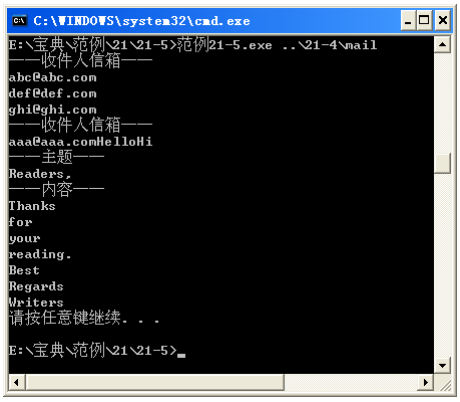


图 19.15　读取邮件内容结果

19.9 小结

本章的主要内容是介绍 C++中文件流的操作。文件流的操作经历打开文件、读写文件、关闭文件等几个步骤。打开文件之后，要注意判断文件是否打开成功。读写文件过程中有可能会出错，所以也需要对文件流的状态进行判断。文件的读写分为格式化读写和无格式化读写。格式化读写指的是按照字符、数字的编码格式进行读写；而无格式化读写，则将文件中的数据一律视为二进制格式，不加以区分。在文件的读写过程中，还可以根据需要重定位读写位置。操作文件之后，需要关闭文件，可以直接调用文件流对象的 close 函数，也可以通过文件流对象的析构函数自动关闭文件。

◆ ◆ ◆

Part

第 3 部分　标准模板库

第 20 章　使用标准模板库 STL

本章包括

- STL 的形成过程
- 各种常见的 STL 版本
- STL 的组成部分
- 容器的分类及常用方法
- 了解配置器、适配器
- 迭代器的用法

在 C++程序中，模板是非常有用的东西。通过模板可以支持泛型编程，即函数模板和类模板可以支持多种类型的数据，而不用重复相同的逻辑过程。既然模板如此有用，所以就有必要开发一套基于模板的库。这就是本章所要讲述的标准模板库，英文是 Standard Template Library，简称 STL，本章的主要内容是 STL 的历史以及 STL 中的容器和迭代器。

20.1 STL 的形成

STL 是 Standard Template Library 的简称，即标准模板库的简称。STL 是以模板为基础的一套标准库，是 C++标准的一个组成部分，可以将其看做一套支持泛型编程、兼顾效率和易用性、设计精巧的工具集，其形成经过了二三十年的发展，并拥有各种实现版本。

20.1.1 STL 的历史

在上世纪 70 年代后半期，Stepanov 考虑能否将应用程序中的算法抽象出来，使得一个算法可以支持多种数据类型，这便是最早的泛型思维。在后来的几年中，Stepanov 又先后尝试用 Schema 语言和 Ada 语言设计了一些大型程序库。

所谓泛型，是指具有在多种数据类型上皆可操作的含意，与模板有些相似，但不等同于模板。泛型编程和面向对象编程不同，并不需要通过虚函数表、虚函数指针这样额外的间接层来调用函数。通过泛型编程可以编写完全一般化并可重复使用的算法，效率与针对某特定数据类型而设计的算法相同。

1992 年，Stepanov 参加了有关算法研究的项目，回到了泛型算法研究的工作上来。经过长期的努力，开发了包含大量数据结构和算法的模板库，这便是现在 STL 的雏形，即 HP STL。在包括 Bjarne Stroustrup 等人的帮助下，Stepanov 又对 STL 进行了改进，加入了内存分配器的模块，也就是现在 STL 的 allocator，使其可以独立于具体的内存模式，从而实现了跨平台。此后，C++不断改进，STL 也不断地演化。至 1998 年，ANSI/ISO C++标准正式定案，STL 一直是 C++标准中不可或缺的一部分。STL 的历史如图 20.1 所示。

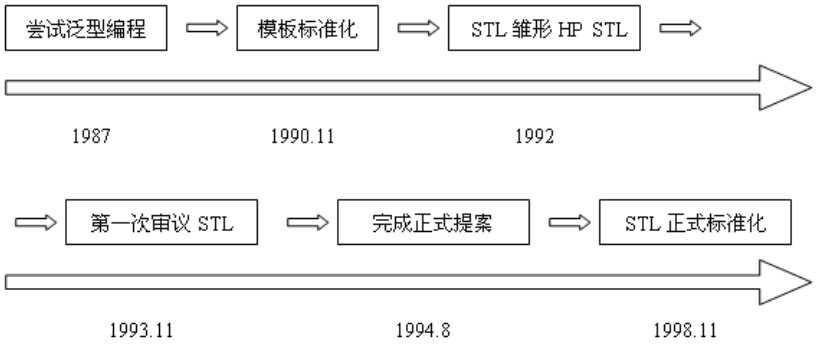


图 20.1 STL 的历史

20.1.2 STL 的各种版本

ANSI/ISO C++中的 STL 仅仅是一个规范，要在开发中应用 STL 还必须将其付诸实现。事实上，自从 STL 正式成为 C++标准后，经过多年的发展已经形成了很多实现版本。如表 20.1 所示就是一些常见的、比较有影响的 STL 的实现版本。

表 20.1 常见的 STL 的实现版本

版本	作者	特点
HP STL	Alexander Stepanov Meng Lee	最早的实现版本 开放源码
Plauger STL	P.J.Plauger	继承 HP STL 可读性较差 由 Visual C++使用
Rouge Wave STL	Rouge Wave 公司	继承 HP STL 非开放源码 由 C++ Builder 使用
SGI STL	Silicon Graphics Computer System, Inc. 公司	继承 HP STL 开放源码 主要用于 Linux 平台
STL port	Boris Fomitchev	以 SGI STL 为蓝本 可以跨平台 开放源码

下面简单介绍一下几个常见的 STL 版本。

- HP STL：HP STL 是最早的 STL 实现版本，即 Alexander Stepanov 在惠普的 Palo Alto 实验室时和 Meng Lee 共同完成的。每个 HP STL 头文件都有一份声明，允许任何人免费使用、复制、修改、传播、贩卖这份软件及其说明文件。需要注意的是，需要在所有文件中加上 HP 的版本声明和运用权限声明，属于开放源码的范畴。

- Plauger STL：P.J.Plauger（后面简称其为 PJ）STL 是个人作品，由 PJ 本人实现，PJ 版本继承 HP 版本，所以每个头文件中都有 HP 的版本声明，此外还有 PJ 的个人版权声明（HP 版本并没有强迫其继承产品必须开放源代码）。PJ 是标准 C 中 stdio 库的早期实现者，C/C++ User's Journal 的主编，与 Microsoft 保持着良好的关系。PJ 版本被 Visual C++使用，但是由于 Visual C++对 C++语言的支持不是很理想，导致 PJ 版本的表现也受到影响。
- Rouge Wave STL：Rouge Wave 版本是由 Rouge Wave 公司开发的，也是由 HP 版本继承而来的，除了 HP 的版本声明，还加上了 Rouge Wave 的公司版权声明。Rouge Wave 也不是开放源码的，所以不能修改和销售。Rouge Wave 版本被 C++ Builder 采用。这个版本有较好的可读性。
- SGI STL：SGI 版本是由 Silicon Graphics Computer System, Inc. 公司开发的，也是 HP STL 的一个继承版本，设计者和编写者包括 Alexander Stepanov 和 Matt Austern。属于开放源码，可以修改、复制和销售。SGI STL 主要被 Linux 下的 C++编译器 GCC 使用。GCC 对 C++标准支持得很好，给予了这个版本的实现积极正面的影响。
- STL port：STL port 起源于俄国人 Boris Fomitchev 的一个开发项目，即以 SGI STL 为蓝本，将其移植到一些主流编译器上，像 C++ Builder 和 Visual C++。SGI STL 属于开放源码，所以 STLport 有权这样做。读者可以到 http://www.stlport.org 去查看并下载 STL port。

20.2 STL 的组成部分

STL 是以模板形式提供的编程组件，解决了很多基础性的编程问题。例如数据的组织、查找、计算等。通过使用 STL，可以让开发人员将主要精力集中在程序的高层逻辑上，而不是底层的操作上，如此可以较大地提高开发效率。本节将带领读者来浏览一下 STL 的主要内容。STL 由六大部分组成，包括容器、算法、迭代器、函数对象、适配器和配置器，各部分的内容如表 20.2 所示。

表 20.2 STL 主要组成部分的内容

组成部分	内容
容器	存储和组织数据的类模板
迭代器	访问容器中数据的一种编程组件
算法	对容器中数据进行查找、计算的模板方法
函数对象	重载了括号运算符()的模板类，作为某种计算策略
适配器	用来修饰容器、迭代器或函数对象的编程组件
配置器	负责内存空间的配置与管理

各个部分的关系如图 20.2 所示。从图中可以看出，容器是 STL 的核心，用来存储和组织数据。配置器是为容器服务的，负责其内存空间的分配和管理。迭代器是为算法服务的，算法通过迭代器访问容器中的数据。函数对象也是为算法服务的，通过配置不同的函数对象，可以改变算法的计算

策略。适配器通过对已有的容器、迭代器和函数对象的改造，可以创造出新的编程组件。

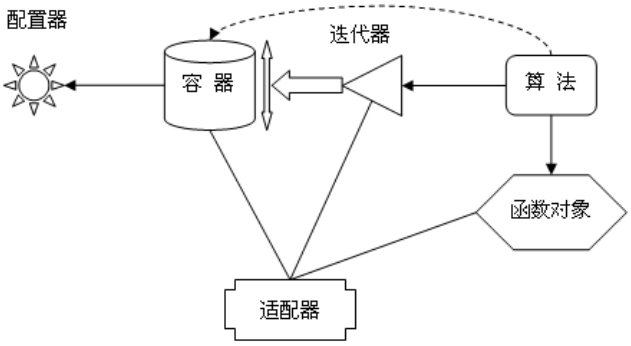


图 20.2 STL 各组成部分之间的关系

下例使用了 STL 的各种主要元素，包括容器、迭代器、算法、函数对象。通过这个例子，读者可以对 STL 的使用有一个整体的印象。程序如示例代码 20.1 所示。

示例代码 20.1

```
#include <cstdlib>
#include <iostream>
#include <vector>                          // 包含容器 vector 的头文件
#include <algorithm>                       // 包含算法头文件
using namespace std;                       // 使用名称空间 std
template<class T>
struct Print                               // 定义一个 Print 结构，作为函数对象来使用
{
   void operator()(T& x) const             // 重载括号()运算符
   {
      if( x % 2 == 0 )                     // 如果参数是偶数
      {
           cout<<x<<' ';                   // 输出参数
       }
   }
};
int main(int argc, char *argv[])           // 主函数
{
    cout<<"——使用 STL——"<<endl;
    cout<<endl;

   vector<int> vcInts;                     // 实例化一个 vector 容器
   for(int i = 0; i < 10; ++i)             // 将整数 0 到 9 插入到容器中
   {
     vcInts.push_back(i);
   }

   vector<int>::iterator iterBegin, iterEnd;   // 定义两个迭代器
   iterBegin = vcInts.begin();             // 指向容器的开始
```

```
    iterEnd = vcInts.end();              // 指向容器的结尾
    cout<<"输出所有元素："<<endl;
    for(; iterBegin != iterEnd; ++iterBegin)    // 用迭代器遍历整个容器
    {
         cout << *iterBegin << " ";             // 输出容器中的各个元素
    }
    cout << endl;

    cout<<"输出偶数元素：<<endl;
    iterBegin = vcInts.begin();                 // 重新指向容器的开始
    for_each(iterBegin, iterEnd, Print<int>());
                            // 通过算法 for_each 和函数对象 Print 输出容器中的元素

    cout<<endl<<endl;
    system("PAUSE");                            // 等待用户反应
    return EXIT_SUCCESS;                        // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 20.3 所示。

图 20.3　使用 STL 示例结果

由于代码重载了括号运算符() ,所以可以当做一个函数对象使用。程序中定义了一个只存放 int 型数据的 vector 容器对象。vector 容器的行为类似于一个数组，但可以随着数据的增加，自动增加存储空间。随后定义了两个迭代器，分别指向容器的开始和结尾，可以利用这两个迭代器遍历整个容器。程序最后调用了 for_each 算法，并结合使用了函数对象 Print。for_each 的行为类似于 for 循环。

20.3 容器的分类

容器（containers），顾名思义，就是用来装东西的器皿。程序中的容器指的是用来存储和组织数据的数据结构。为了支持泛型编程，这种数据结构应当写成模板类，所以在 STL 中所谓的容器就是各种类模板。

STL 容器基本上都支持容量的自动增长。添加数据时，如果容量不能满足要求，容器就会自动分配新的内存，以用来容纳添加的数据。

按照存储方式分类，容器可以分为序列式容器和关联式容器。所谓序列式容器，是指元素依照

加入的顺序进行排放，加入元素时没有按照其大小进行排序，但加完后可以排序。所谓关联式容器，是指元素都是由两部分组成的，分为键值（key）和实值（value）。键值和实值之间存在对应关系。当数据加入到容器中时，按照键值进行排序，因此关联式容器是自动有序的。如表 20.3 所示就是常用的各种 STL 容器。

表 20.3 常用的 STL 容器

分类	容器	功能	头文件
序列式容器	vector	向量，线性存储空间，类似数组	#include <vector>
	list	双向链表，非线性存储空间	#include <list>
	slit	单向链表，非线性存储空间	#include <slist>
	deque	双端队列	#include <deque>
	stack	栈，数据先进后出	#include <stack>
	queue	队列，数据先进先出	#include <queue>
	priority_queue	优先级队列	#include <queue>
关联式容器	set	集合	#include <set>
	map	映射	#include <map>
	multiset	可重复集合	#include <set>
	multimap	可重复映射	#include <map>

STL 的头文件使用 C++标准样式的头文件，即没有扩展名“.h”。但是这样的头文件里面往往包含了一个同名的但带有“.h”扩展名的头文件。例如容器 map 的头文件是“map”，但在“map”这个文件里又包含了头文件“map.h”。这主要是因为各个版本的 STL 往往是以前某个版本 STL（大多数是 HP STL）的改进，所以要包含其相应的头文件，而这个版本的头文件又是在 C++标准确定前建立的。

序列式容器有两种存储方式。一种是连续存储，即容器中相邻元素的存储单元是连续的，新加入的元素总是放在紧邻最后一个元素的下一个存储单元中，数组就是这样一种连续存储的序列式容器。另外一种方式是链式存储，即相邻元素的存储单元不是连续的，而是用某种方式将其连接在一起，可以在元素中用一个（或多个）指针，指向相邻元素存储单元的地址，或者用整数表示相邻元素的存储索引。两种存储方式如图 20.4 所示。

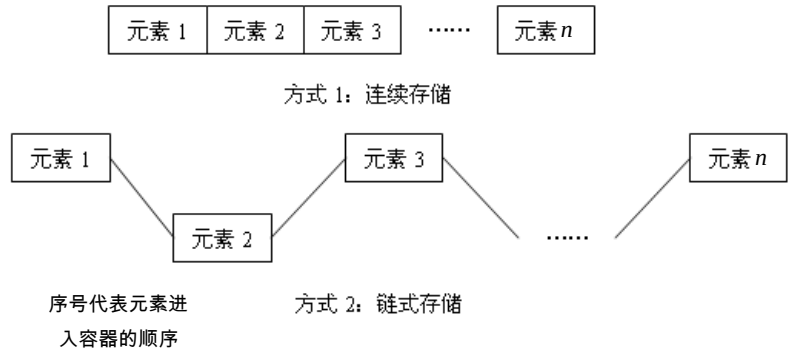


图 20.4　序列式容器的两种存储方式

关联式容器也有两种存储方式。一种是使用搜索二叉树，容器中的元素依照其键值进行排序。另外一种是利用哈希表，容器中的元素按照其键值，根据哈希算法安置在不同的存储单元中。其两种存储方式如图 20.5 所示。

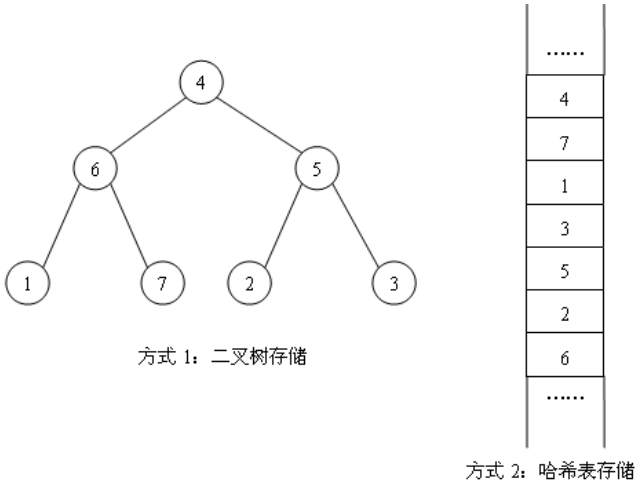


图 20.5　关联式容器的两种存储方式

图中的序号代表元素进入容器的顺序。

标准 STL 是用红黑树来实现关联式容器的。红黑树是一种查找效率更高的平衡搜索二叉树，相关内容将在后面的章节中进行介绍。某些版本的 STL，比如 SGI STL 中有一些非标准的关联式容器，如 hashtable，hashset，hashmap 等是用哈希表实现的，本书不进行讲解，请读者参考相关书籍。

20.4　容器的常用方法

容器是用来存储数据的，因此容器一般具有如下的方法：

◆ 初始化（容量、初始值、元素的比较方法等）。

◆ 数据的增、删、改、查。

◆ 容器数据统计（容量大小、元素数目）。

本节将详细讲解如何使用容器。

20.4.1 初始化容器

容器的初始化是在容器构造时完成的。初始化的内容包括容量、初始值、元素的比较方法等。一般来讲，序列式容器要求初始化容量，而关联式容器（基于红黑树的）要求初始化元素的比较方法。对于基于哈希表实现的关联式容器也要求初始化容量。例如：

```
vector<int> vec( 12 );                  // 构造整型向量，初始容量 12
list<int> lst( 34 );                    // 构造整型链表，初始容量 34
struct ltstr                            // 重载了括号运算符的函数对象
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) < 0;
  }
};
map<const char *, int > strIntMap( ltstr() );
                            // 构造字符串到整型的映射，初始化键值比较方法为 ltstr
set<const char *> strSet( ltstr() );
                            // 构造字符串集合，初始化键值比较方法为 ltstr
```

两种类型的容器都可以在构造时设置初始值，初始值是用两个迭代器对象所标识的容器区间，新容器中的元素被初始化为这个容器区间内的值。例如：

```
int ary1[5] = {1, 2, 3, 4, 5};                  // 整型数组
vector<int> vec( ary1, ary1+5 );                // 用数组初始化向量
ist<int> intList( vec.begin(), vec.end() );     // 用向量初始化链表
set<int> intSet( list.begin(), list.end() );    // 用链表初始化集合
```

数组也是一种容器，由 C++编译器预定义。vector 和 list 的成员函数 begin 和 end 返回的是容器的迭代器，分别指示的是容器的头和尾。

20.4.2 增加元素

为了增加元素，两种容器都提供了 insert 方法，该方法支持在指定的位置加入新的元素。在 STL 中位置一般用迭代器指定，如下所示：

```
int ary1[5] = {1, 2, 3, 4, 5};          // 整型数组
vector<int> vec(ary1, ary1+5);          // 用数组初始化向量
vector<int>::iterator iter = vec.begin() + 2;
                                        // 定义迭代器，指向容器 vec 的第三个元素
vec.insert( iter, 6 );                  // 在 vec 的第三个元素处插入一个新元素 6
set<int> intSet( vec.begin(), vec.end() );      // 构造一个整型数集合
```

```
set<int>::iterator iter = intSet.begin() + 2;  // 集合的迭代器，指向第三个元素
intSet.insert( iter, 7 );                      // 在集合的第三个元素处插入一个新元素 7
```

由于在序列式容器中，相邻两个元素的存储单元存在先后关系，插入元素时要维护这样的关系。所以，对于连续存储的容器，要移动插入位置后面的元素；对于链式存储的容器，要建立新的链接关系。连续存储的容器，如果其容量不能满足数据插入的需求，则应当重新分配足够大的内存，复制原来的数据，并插入新值。在上述例子中，在向量中插入数据的效果如图 20.6 所示。

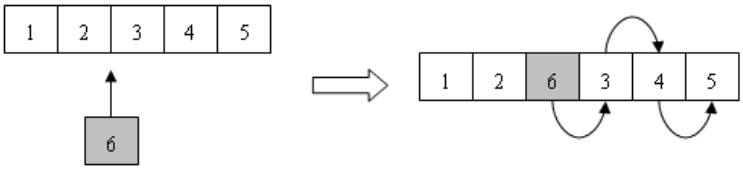


图 20.6　在 vector 中插入元素

关联式容器插入数据的效果与序列式容器插入数据的效果是不同的。由于关联式容器是自动排序的，插入的元素会按照其键值的大小放置到合适的地方，所以关联式容器并不会在期望的地方插入元素，其传入的迭代器只能作为参考，如图 20.7 所示。

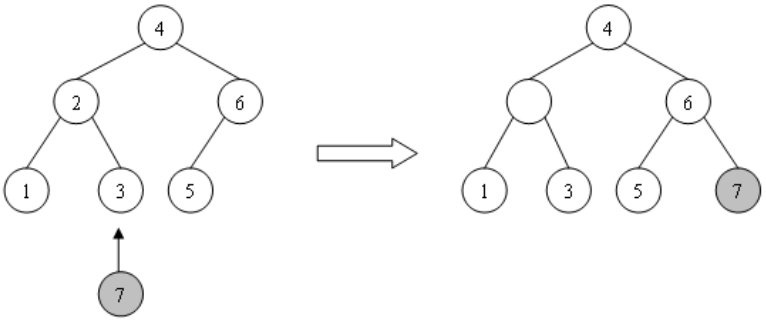


图 20.7　在关联式容器中插入数据

除了 insert 方法，序列式容器还提供了压入（push）元素的方法，根据容器的不同，分别使用 push_back，push_front，push 方法。压入元素没有 insert 操作那么灵活，而只能在容器头或容器尾操作，例如：

```
vector<int> vec();              // 声明一个向量
vec.push_back( 1 );             // 压入数据
vec.push_back( 2 );
list<int> ls();                 // 声明一个链表
ls.push_back( 1 );              // 压入数据
ls.push_front( 2 );
stack<int> st();                // 声明一个栈
st.push( 1 );                   // 压入数据
st.push( 2 );
```

上面操作的流程图如图 20.8 所示。

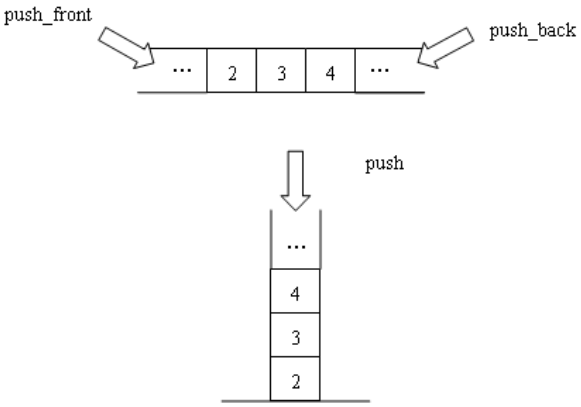


图 20.8 在序列式容器中压入数据流程图

20.4.3 删除元素

为了删除元素，两种容器都提供了 erase 方法，该方法支持删除指定位置、指定区间的元素。另外，也可以利用 clear 方法，删除容器中的所有元素。对于关联式容器，还可以删除指定键值的元素。当然，删除数据之后，应当对容器的存储结构进行重新组织，以保证结构的完整性。例如：

```
int ary1[6] = {1, 2, 3, 4, 5, 6};  // 整型数组
vector<int> vec(ary1, ary1+6);     // 用数组初始化向量
vector<int>::iterator iter = vec.begin() + 2;
                                   // 定义迭代器，指向容器 vec 的第三个元素
vec.erase( iter );                 // 删除第三个元素
ter = vec.begin() + 2;             // 定义迭代器，指向容器 vec 的第三个元素
vec.erase( iter, vec.end() );      // 删除第二个元素之后的所有元素
set<int> intSet( ary1, ary1+5 );   // 构造一个集合
set.erase( 4 );                    // 删除键值为 4 的元素( 集合的键值与实值是一致的 )
```

上面操作的流程图如图 20.9 所示。

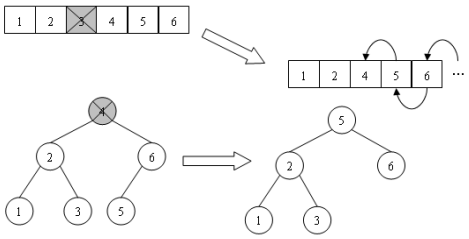


图 20.9 从容器中删除元素流程图

20.4.4 查找元素

并不是所有的容器都提供了专门的查找方法。序列式容器由于其存储结构的限制（未排序），查找效率较低，所以没有提供查找方法。而关联式容器在创建时就排好了序，插入元素时也会自动排序，所以非常适合查找。因此，关联式容器提供了专门的查找方法 find，其参数是元素的键值，返回值是指向目标元素的迭代器。例如：

```
int ary[5] = { 3, 1, 5, 2, 4};              // 整数型数组
set<int> intSet( ary, ary+5 );              // 构造集合
set<int>::iterator iter = intSet.find( 4 ); // 查找集合中键值为 4 的元素
```

对于序列式容器，可以使用某些特殊的方法来查找特定的元素。例如有的容器重载了下标运算符“[]”，给出索引就可以找到对应的元素，这一点类似数组。也可以通过成员函数 at 实现相同的功能。另外，序列式容器普遍都有 front 和 back 方法，用来返回第一个和最后一个元素，栈是个例外，只有函数 top 用来返回栈顶元素。例如：

```
int ary1[6] = {1, 2, 3, 4, 5, 6};       // 整型数组
vector<int> vec(ary1, ary1+6);          // 用数组初始化向量
int x = vec[ 3 ];                       // 查找向量中的第三个元素
list<int> intList( vec.begin(), vec.end() );   // 构造链表
x = intList.front();                    // 查找链表中的第一个元素
x = intList.back();                     // 查找链表中的最后一个元素
stack<int，vector<int> > st;            // 构造栈
……
x = st.top();                           // 查找栈顶元素
```

除了利用容器提供的专门查找方法外，还可以利用 STL 算法中相应的函数模板进行查询，例如 find，find_if，find_end 和 find_first_of。但是，出于效率方面的考虑，请尽量使用容器自身提供的方法。

20.4.5 修改元素

若要修改元素，则必须先查找到元素。如果查找到的结果是容器中元素的引用，则可以直接对该引用赋值。如果查找到的结果是指向容器中元素的迭代器，则可以通过迭代器的方法修改元素的值。例如：

```
int ary1[6] = {1, 2, 3, 4, 5, 6};       // 整型数组
vector<int> vec(ary1, ary1+6);          // 用数组初始化向量
vec[3] = 7;                             // []返回的是向量元素的引用
int &x = vec[3];
x = 7;                                  // 效果与第 3 行相同
stack<int，vector<int> > st;            // 构造栈
……
st.top() = x;                           // 修改栈顶元素
int &y = st.top();                      // 查找栈顶元素
y = x;                                  // 效果等同于第 8 行
map<int, int> intMap();                 // 构造映射
……
map<int, int>::iterator iter = intMap.find( 5 );   // 查找键值为 5 的元素
iter->second = 3;                       // 将该元素的实值修改为 3
map[5] = 3;                             // 效果与第 12 行和第 13 行相同
```

20.4.6 统计容器数据

统计容器数据主要包括两个方面，即容量和元素数。容量指的是该容器当前所能容纳的元素的

总数，元素数指的是当前容器中元素的个数。实际上只有向量的容量跟元素数可能不一致，对于其他容器两者的数量是一样的。各个容器求元素数的方法是一致的，都是调用 size 方法。例如：

```
int ary1[6] = {1, 2, 3, 4, 5, 6};       // 整型数组
vector<int> vec(ary1, ary1+6);          // 用数组初始化向量
cout<< vec.size() <<endl;               // 输出 5
list<int> ls( vec.begin(), vec.end() );
cout<< ls.size() <<endl;                // 输出 5
map<int, int> m;
m[1] = 1;
m[2] = 2;
cout<< m.size() << endl;                // 输出 2
```

对于向量，有一个成员函数是 reserve(size_type size)，该函数用来为当前向量对象预留至少可以存储 size 个元素的空间。注意是“至少”，向量出于自身空间管理的需要，有可能在 size 的基础上增加必要的空间。例如：

```
vector<int> v;                      // 构造向量
v.reserve(20);                      // 保留 20 个元素的空间
v.push_back(0);                     // 压入元素 0
v.push_back(1);                     // 压入元素 1
cout<<v.capacity()<<endl;           // 输出容量，20
cout<<v.size()<<endl;               // 输出元素数，2
```

上面的操作如图 20.10 所示。

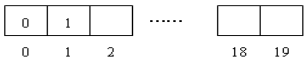


图 20.10 向量：20 个存储空间，2 个元素

与 reserve()函数相似，序列式容器还有一个 resize()成员函数。该函数除了接受一个 size 参数外，还接受一个 val 参数。其目的是将容器中的元素数目设置为 size，并将新创建的元素值设置为 val。

如果调用向量的 resize()函数，而且设置的新元素数低于当前向量的容量，则不会修改容量。只有当新设的元素数大于容量时，才会修改容量。

```
vector<int> vec(2, 1);                  // 构造向量，2 个元素，其值为 1
vec.resize(20，5);                      // 重设元素数，全部置为 5
cout<<vec..size()<<endl;                // 输出元素数，20
list<int> ls( vec.begin(), vec.end()); // 构造链表
cout<<ls.size()<<endl;                  // 输出链表元素数，20
```

上面的操作如图 20.11 所示。

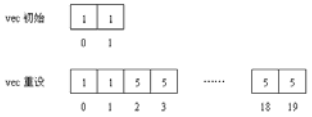


图 20.11 重设向量元素数

为了判断容器是否为空，即容器中有没有元素，除了可以判断元素数是否为 0，还可以直接调用容器的 empty()方法。该方法返回一个 bool 值，如果为 true，则表示容器中已经没有元素；如果为 false，则表示容器中还有元素。

20.4.7 其他方法

还有一些方法是在使用容器时经常用到的，例如：

- begin：求指向容器开始元素的迭代器。
- end：求容器结尾元素后一个存储单元的迭代器。
- rbegin：求指向容器反向开始元素的迭代器。
- rend：求容器反向结尾元素后一个存储单元的迭代器。
- swap：交换两个容器的内容。
- ==，!=，<，<=，>，>=：比较运算符，判断两个容器之间的关系。

有关迭代器的操作，将在后面的章节中介绍，这里着重介绍一下容器的 swap 操作，以及容器间的比较。

STL 中的容器一般都会提供 swap 方法，其目的是交换两个容器的内容。交换内容的两个容器，其类型必须是相同的，比如同为向量，或者同为链表。另外，两个容器的大小最好也相同，否则会出现意想不到的问题。例如：

```
int a1[] = {1, 2, 3};
int a2[] = {4, 5, 6};

vector<int> v1(a1, a1+3), v2(a2, a2+3);        // 构造两个向量
v1.swap( v2 );                                 // 交换两个向量
vector<int>::iterator iter = v2.begin();
for(; iter != v2.end(); iter++)                // 输出向量 v2 的内容
{
    cout<<*iter<<endl;
}
```

上面的操作如图 20.12 所示。

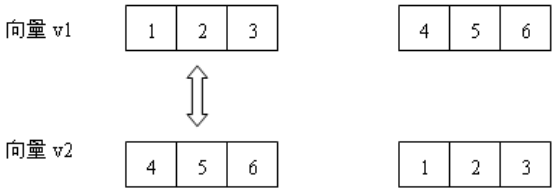


图 20.12　交换两个容器的内容

为了方便容器间的大小比较，STL 还提供了一系列的关系运算符。这些运算符都是全局运算符，定义在容器之外，并声明为容器类模板的友元。当然，不一定全部关系运算符都是容器的友元。比如，有的 STL 版本就只把“==”和“<”指定为容器的友元，而其他的各种运算符在实现时都要调用这两个运算符。容器比较的基本原则是“逐个遍历，一一比较”，即

- 容器比较的结果，即第一对不相等元素间的比较结果。
- 如果两个容器元素数目不相等，则容器不相等。

例如：

```
vector <int> v1, v2;                    // 定义两个向量
```

```
v1.push_back( 1 );                      // 在 v1 中加入数据
v1.push_back( 2 );
v1.push_back( 4 );
v2.push_back( 1 );                      // 在 v2 中加入数据
v2.push_back( 3 );
cout<< "v1 < v2" << v1 < v2 <<endl;     // 输出 1，true
```

20.5 配置器

STL 配置器是用来分配和管理内存空间的。开发人员在使用容器时，大多数时候不需要考虑容器以及容器中元素内存空间的分配和管理，这些任务都由 STL 的配置器处理了。STL 的配置器也是一个类模板，当在容器中使用时，需要指定模板实参，也就是指定该配置器是为什么类型的元素分配和管理内存空间的。

尽管配置器对于 STL 如此重要，但 C++标准并没有给出一个 STL 的标准实现，而只是给出了一个标准“接口”。STL 的各个实现版本可以根据自身的特点以及不同平台的差异，给出不同的实现，只要这个实现符合“接口”定义即可。

这里所谓的接口，并不是纯虚基类，而是一个规定，即规定了符合该接口定义的类所必须具有的一些要素，如内置类型定义、成员函数以及数据成员等。

STL 中的配置器，无论其类名如何，必须符合如下的“接口”定义，如表 20.4 所示。

表 20.4 STL 配置器的接口定义

类别	内容	含义
必要的类型定义	value_type	值类型
	pointer	指针类型
	const_pointer	常指针类型
	reference	引用类型
	const_reference	常引用类型
	size_type	数量类型
	difference_type	指针偏移量类型
	rebind	相关类型的分配器
必要的函数定义	allocator()	默认构造函数
	allocator(const allocator &)	复制构造函数
	~allocator()	析构函数
	pointer address(reference x)	返回某个对象的地址
	const_pointer address(const_reference x)	返回某个对象的地址
	pointer allocate(size_type n, const void* =0)	分配 n 个元素的空间
	void deallocate(pointer p, size_type n)	释放已分配的空间

	size_type max_size()	返回可成功分配的最大值
	void construct(pointer p, const T& x)	在指定地址构建对象
	void destroy(pointer p)	析构指针

其中 difference_type 表示的是两个指针间的偏移量类型，在配置器中一般定义为：

```
typedef ptrdiff_t difference_type;
```

rebind 也是一个类模板的类型定义，其作用是为相关类型的数据分配内存空间。一个容器的配置器，其原始的目的是为其元素分配内存。但是，这个元素可能是一个类，其中还有别的数据。为了使用同样的策略给这些内部数据分配内存，还应当针对这些内部数据定义一个配置器，这就是 rebind，其定义如下：

```
template< typename T>
class allocator
{
public:
    ……
    template <class U>
    struct rebind
    {
        typedef allocator<U> other;
    };
};
```

作为一个配置器，内存的分配与释放是最关键的。这两个功能分别由函数 allocate()和 deallocate()实现。为了尽可能地提供高效率的内存管理，配置器的开发者应当非常细致地实现这两个函数，比如利用内存池技术等，读者可以参看相关的资料。

至于 construct()和 destruct()方法，则是在指定内存上构造和析构对象。实现 construct()方法可以使用所谓的 placement new，而 destruct()则可以直接调用对象的析构方法。

对配置器实现的深入讨论已经超出了本书的范围。如果仅仅是使用 STL，则了解配置器的接口即可。而各种版本的 STL 都提供了自己的实现，各个容器也都将此实现作为默认的模板实参，作为该容器的配置器。

20.6 迭代器

迭代器的设计基于这样的理念：提供一种对象，使之能够按照某种顺序遍历容器中的元素，并且不必暴露容器的内部实现方法。这里所指的对象是广义的，只要能够实现该理念的程序实体，都可以称之为迭代器。迭代器使得容器与算法可独立设计，更加有效地支持了泛型编程。

20.6.1 迭代器思想

正如前面所讲，容器有很多种，因为容器的内部实现各不相同，所以其遍历方式也必然各不相同。例如向量可以像数组那样通过指针访问元素，链表则必须逐个元素地遍历，而关联式容器则必

须按照树的结构进行遍历，如图 20.13 所示。

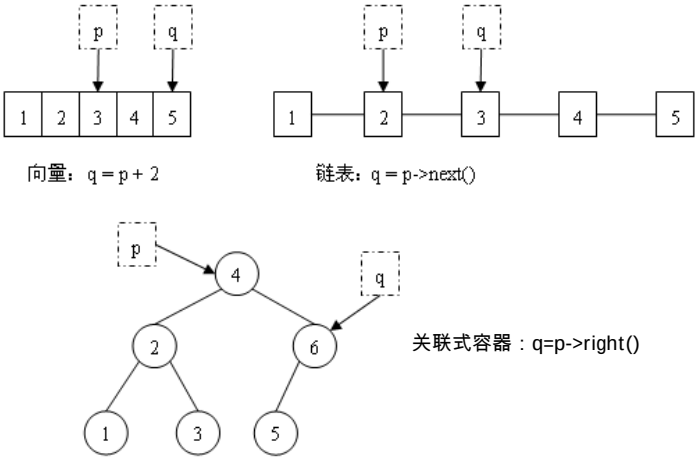


图 20.13 各种容器的遍历方法

如果能够提供一种统一的界面，使得各种容器都能按照一种方法访问和修改容器，那将会使程序设计变得简洁，而且易于理解。这种统一的界面就是迭代器。

众所周知，使用指针访问数组是非常方便的。例如通过指针可以取值、赋值，也可以移动指针指向不同的数组元素。同时，指针之间也可以互相比较，以判断两者是否相同。因此，STL 迭代器在设计时主要模拟了指针的行为。另外，对于数组这个容器来讲，指针就是其迭代器。所以为了统一起见，迭代器也必须与指针的行为相同，这样才可以将指针当做迭代器。

实际上，在 STL 中迭代器被设计成了智能指针。所谓智能指针就是一种模拟指针行为的对象，该对象支持指针的所有操作，并且可以自动析构。

20.6.2 迭代器分类

迭代器按照其功能，可以划分为以下五类：

- 输入迭代器。
- 输出迭代器。
- 前向迭代器。
- 双向迭代器。
- 随机迭代器。

各种迭代器如图 20.14 所示。

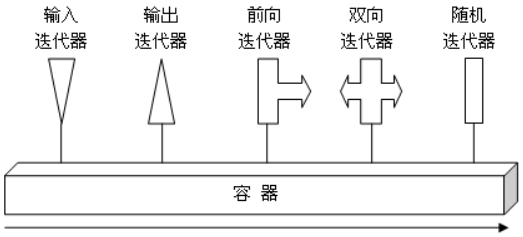


图 20.14　各种迭代器

输入迭代器只能向前移动，且每次只能移动一步，只能读输入迭代器指向的东西，而且只能读一次，输入流类（istream）的迭代器是这一种类的典型代表。输出迭代器与输入迭代器类似，只不过用于输出，输出迭代器只能向前移动，每次只能移动一步，只能写输出迭代器指向的东西，而且只能写一次，输出流类（ostream）的迭代器是这一种类的典型代表。

输入流是数据的来源，可能是文件、数组等，其中容纳了各种各样的数据，因此可以将输入流当做容器。同样输出流也可以当做容器。

输入和输出迭代器比较受限，即只能用于输入和输出。而前向迭代器则比这两种迭代器功能更加强大。前向迭代器像输入和输出迭代器一样仅向前移动，但可以同时读写其指向的元素，而且可以读写多次。例如单向链表（slist）的迭代器就属于这一种。

双向迭代器则既可以向前移动也可以向后移动，并且也可以像前向链表那样多次读写元素。STL 中链表（list）的迭代器属于这一种类。其他容器如 set，multiset，map 和 multimap 的迭代器也都是双向迭代器。

功能最全面的迭代器是随机迭代器。这种迭代器在双向迭代器的基础上加上了“偏移”能力，即迭代器可以向前或向后跳转一个偏移量。vector，deque 和 string 的迭代器就是随机迭代器。

上述各种类型的迭代器，其功能是依次增强的。其演化关系如图 20.15 所示。

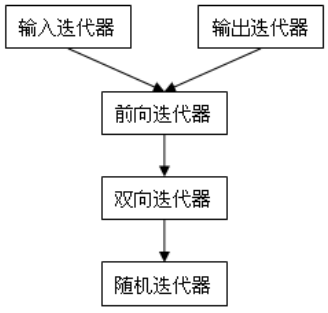


图 20.15　各种迭代器的演化关系

尽管随机迭代器的功能是最强的，但这并不是说随机迭代器可以替代其他类型的迭代器。一种容器能够提供什么类型的迭代器取决于容器的类型。比如有这样一个只读容器，其数据只能一个一个地读取，并且只能读取当前数据的下一个数据，那么这样的容器就只能支持输入迭代器，如输入流。对于单向链表容器，其数据可读可写，但由于只能沿着链表方向逐个元素地前进，所以只能支

持前向迭代器。各种容器支持的迭代器分类如表 20.5 所示。

表 20.5 各种容器支持的迭代器分类

容器	迭代器分类
vector	随机迭代器
deque	随机迭代器
list	双向迭代器
slist	前向迭代器
set	双向迭代器

（续表）

容器	迭代器分类
multiset	双向迭代器
map	双向迭代器
multimap	双向迭代器
stack	不支持
queue	不支持
priority_queue	不支持

数组是 C/C++语言编译器内置的一种容器，可以将指针当做容器的迭代器。由于指针可以随意跳转到指定的数组元素上，并且通过指针可以读、也可以写数组元素。所以指针可以作为数组的随机迭代器。

20.6.3 定义迭代器变量

在 STL 中，大多数容器（除了 queue，stack，priority_queue）都定义了自己的迭代器类型。根据其使用目的，分别定义为 iterator，const_iterator，reverse_iteraotr 以及 const_reverse_iterator。这里所说的迭代器类型与上一节的迭代器分类并不矛盾。迭代器分类其实对应的是迭代器的属性，而迭代器类型则真真正正是一种数据类型。例如向量的迭代器定义如下：

```
template<typename _Tp, typename _Alloc = allocator<_Tp> >
class vector : protected _Vector_base<_Tp, _Alloc>
{
public:

    ……
    typedef __gnu_cxx::__normal_iterator<pointer, vector_type> iterator;
    typedef __gnu_cxx::__normal_iterator<const_pointer, vector_type> const_iterator;
    typedef std::reverse_iterator <const_iterator> const_reverse_iterator;
    typedef std::reverse_iterator<iterator> reverse_iterator;

    ……
}
```

其中__gnu_cxx::__normal_iterator 和 std::reverse_iterator 都是类模板，可以根据不同的模板参数实例化迭代器，各类型迭代器的目的如表 20.6 所示。

表 20.6　各类型迭代器的目的

类型	名称	目的
iterator	正向迭代器	从容器头向容器尾迭代
reverse_iterator	反向迭代器	从容器尾向容器头迭代
const_iterator	常量正向迭代器	正向迭代，不能修改所指元素
const_reverse_iterator	常量反向迭代器	反向迭代，不能修改所指元素

容器有头有尾，其头表示容器的开始处，尾表示容器的结尾。对于序列式容器，容器的头尾就是其线性存储空间的起始和结束；而对于关联式容器，其存储空间不是线性的，所以其头尾分别表示最小元素和最大元素（之后）的位置。容器一般都会提供 begin()和 end()方法，用来取得指向容器头和尾的正向迭代器。对于反向迭代器，可以使用 rbegin()和 rend()方法，分别取得容器头和尾的反向迭代器。rend()指向容器第一个元素之前的存储位置。两个方法如图 20.16 所示。

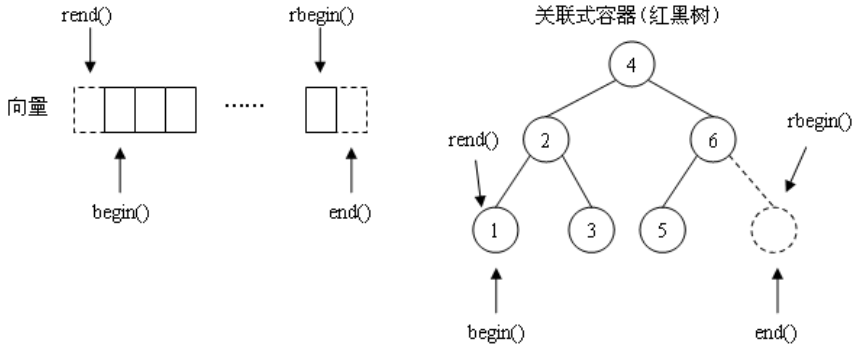


图 20.16　容器的 begin()和 end()方法

可以用这两个方法来初始化迭代器变量，或者给迭代器变量赋值。例如：

```
vector<int> a;                                  // 定义一个整型容器
……
vector<int>::iterator iter1 = a.begin();   // 定义一个迭代器，指向容器的头
vector<int>::iterator iter2 = a.end();     // 定义一个迭代器，指向容器的尾
```

20.6.4　迭代器的基本用法

在程序中要使用迭代器，必须先定义迭代器变量。由于迭代器的类型都是迭代器的内嵌类型，所以在使用时要加上容器的域。例如：

```
vector<int>::iterator  iter1;                   // 定义一个整型向量的迭代器
vector<string>::iterator  iter2;                // 定义一个字符串向量的迭代器
```

迭代器变量在使用之前必须初始化或赋初值。此时可以调用 begin()，end()，rbegin()，rend()

等方法来获得指向容器头尾或反向头尾的迭代器，并用来初始化或者给迭代器变量赋初值。

```
int ary<int> = {1, 2, 3, 4, 5};              // 定义数组
vector<int> vec( ary, ary+5 );               // 构造向量
vector<int>::iterator iter = vec.begin();  // 定义正向迭代器，并用向量的头初始化
for( ; iter != vec.end(); iter++ )           // 遍历向量
{
    cout<< *iter <<endl;                     // 输出迭代器所指元素
}
vector<int>::reverse_iterator iter = vec.rbegin();
                                          // 定义反向迭代器，并用向量的反向头初始化
for( ; iter != vec.rend(); iter++ )          // 反向遍历向量
{
    cout<< *iter <<endl;                     // 输出迭代器所指元素
}
```

除了上述定义、初始化和赋初值操作外，迭代器一般还支持如表 20.7 所示的基本操作。

表 20.7 迭代器的基本操作

操作	重载运算符	含义	支持迭代器
比较	==，!=	比较两个迭代器是否指向同一元素	全部
取值	*	取迭代器所指元素	不支持输出迭代器
赋值	*	给迭代器所指元素赋值	不支持输入迭代器
访问成员	->	引用迭代器所指元素的成员	全部
步进	++	移动迭代器，指向下一个元素	前向、双向、随机迭代器
步退	--	移动迭代器，指向前一个元素	双向、随机迭代器
跳跃	+，-，+=，-=	移动迭代器，跨越参数指定数目的元素	随机迭代器

由于数组也是容器，并且可以将指向数组元素的指针当做迭代器使用，所以读者可以结合数组和指针来理解上述表中的各种操作。实际上，迭代器的原始设计思想就是为了模仿指针的行为。

比较迭代器有两个重要的意义，一是避免重复操作同一元素，二是检查迭代器是否已经到了容器的尾部。正如前面讲到的一样，容器的尾部并不属于容器，而是在容器尾部之后的位置。这个位置通常用来检查迭代器是否有效。迭代器的比较方法是：

```
vector<int> vec;                    // 定义向量
......                              // 某些向量操作
vector<int>::iterator iter = vec.begin()   // 定义向量迭代器，指向向量的头
......                              // 向量操作，修改迭代器
if( !(iter == vec.end() )           // 判断当前迭代器是否指向容器尾
{
    ......
}
```

或者

```
if( iter != vec.end() )            // 判断当前迭代器是否指向容器尾
{
    ……
}
```

除了输入迭代器，其他各种类型的迭代器都支持取值操作。取得的值就是该迭代器所指向的元素。取值的方法是利用运算符“*”解引用迭代器。另外也可以利用访问成员运算符“->”，访问迭代器所指元素的成员。

使用关联式容器的迭代器时，注意要访问的是键值还是实值。如果是键值，则其值为(*iter).first；如果是实值，则其值为(*iter).second。当然也可以使用指针访问成员的运算符“->”，如 iter->first 和 iter->second。

下面看一个例子：

```
int ary[5] = {1, 2, 3, 4, 5};              // 定义数组
vector<int> vec( ary, ary+5 );             // 用数组初始化向量
vector<int>::iterator iter = vec.begin();  // 定义迭代器，指向向量头
for( ; iter != vec.end(); iter++ )         // 遍历向量
{
    cout<< *iter <<endl;                   // 输出迭代器所指元素
}
map<int, char> m;                          // 定义映射
m[0] = 'a';                                // 加入元素
m[1] = 'b';
m[2] = 'c';
map<int, char>::iterator iterMap = map.begin();    // 定义迭代器，指向映射头
for( ; iterMap != m.end(); iter++ )        // 遍历映射
{
    cout<< iter->first <<endl; // 输出键值，等同于 cout<< (*iter).first<<endl;
    cout<< iter->second <<endl;// 输出实值，等同于 cout<< (*iter).second<<endl;
}
```

由于解引用运算符“*”实际返回的是迭代器所指元素的引用，所以也可以利用“*”给容器中的元素赋值。当然，如果容器中存储的是类对象，那么也可以利用访问成员运算符“->”访问元素的成员，并给其赋值。例如：

```
int ary[5] = {1, 2, 3, 4, 5};          // 定义数组
vector<int> vec( ary, ary+5 );         // 用数组初始化向量
vector<int>::iterator iter = vec.begin();  // 定义迭代器，指向向量头
for( ; iter != vec.end(); iter++ )     // 遍历向量
{
    *iter *= 2;                        // 将迭代器所指元素乘以 2
}
```

```
map<int, char> m;                          // 定义映射
m[0] = 'a';                                // 加入元素
m[1] = 'b';
m[2] = 'c';
map<int, char>::iterator iterMap = map.begin();    // 定义迭代器，指向映射头
for( ; iterMap != m.end(); iter++ )        // 遍历映射
{
    // iter->first += 1;                   // 错误！关联式容器不能改键值
    iter->second += 1;                     // 修改映射的实值
}
```

上面的例子中为了遍历容器，在每个 for 循环中使用了迭代器的后自加运算符“++”。该运算符的含义是步进迭代器，使得迭代器指向当前元素的下一个元素。除了使用自加运算符之外，还可以使用自减运算符“--”，即步退迭代器，指向当前元素的前一个元素。

步进迭代器也可以使用前自加元素符“++”，如++iter，而且使用前自加比使用后自加的效率要高。这是因为在大多数迭代器的实现中，都是调用前自加运算符来重载后自加运算符的。

例如：

```
int ary[5] = {1, 2, 3, 4, 5};              // 定义数组
vector<int> vec( ary, ary+5 );             // 用数组初始化向量
vector<int>::iterator iter = vec.rend();   // 定义迭代器，指向向量头
for( ; iter != vec.rbegin(); )             // 遍历向量
{
    *(--iter) = 2;                         // 将迭代器所指元素乘以 2
}
map<int, char> m;                          // 定义映射
m[0] = 'a';                                // 加入元素
m[1] = 'b';
m[2] = 'c';
map<int, char>::iterator iterMap = map.rend(); // 定义迭代器，指向映射头
for( ; iterMap != m.rbegin();)             // 遍历映射
{
    // (--iter)->first += 1;               // 错误！关联式容器不能改键值
    (--iter)->second += 1;                 // 修改映射的实值
}
```

除了步进和步退迭代器之外，对于随机迭代器还可以使用加减运算符“+”、“-”，跨越指定数量的元素。迭代器跨跃如图 20.17 所示。

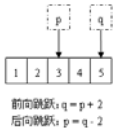


图 20.17 迭代器跨跃

20.6.5 使用迭代器区间

使用迭代器不仅可以访问容器的某一个指定元素，还可以用两个迭代器指定一个范围，从而访问该范围内的所有元素。不过需要注意的是，两个迭代器指定的范围是一个前闭后开的区间。也就是说前一个迭代器所指的元素包含在区间内，而后一个迭代器所指的元素则不在区间内，而是区间尾部的后一个元素，即[iter1, iter2)。

容器的很多函数都支持迭代器的区间操作。例如，删除元素的成员函数 erase()，该函数不仅支持删除单个迭代器所指元素，即：

```
iterator erase( iterator _Where );
```

也支持删除某个迭代器区间内的所有元素：

```
iterator erase( iterator _First, iterator _Last );
```

调用上述删除区间的版本时，容器中所有从迭代器_First 开始到_Last 之前（不包括_Last）的元素都将被删除。例如，删除向量 vInts 中的元素：

```
vector<int> vInts(4);                    // 定义一个长度为 4 的向量
for( int i=0; i<4; i++ )                 // 遍历向量
{
    vInts[i] = I;                        // 给元素赋值
}
vector<int> iter = vInts.begin();        // 定义一个指向向量开头的迭代器
vInts.erase( iter );                     // 删除向量 vInts 的第一个元素
iter = vInts.begin();                    // 重新指向开头
vector<int> iter2 = iter.end() - 1;      // 获取指向最后一个元素的迭代器
vInts.erase( iter,  iter2 );             // 删除两个迭代器间的元素，只剩最后一个
cout<< vInts.size() <<endl;              // 输出向量长度，结果是 1
```

利用迭代器也可以向容器中插入元素，方法是 insert()。同 erase()一样，insert()既支持单一迭代器的插入，又支持迭代器区间的插入。无论哪种方法，insert()总是需要一个迭代器来指定插入的位置，而这个插入位置迭代器是容器本身的迭代器。例如，在某个位置插入一个值的 insert()方法如下：

```
iterator insert( iterator _Where,  const Type&  _Val );
```

其中_Where 是容器中某个元素的迭代器，_Val 是要插入的值。其结果是从_Where 开始到 end()的所有元素往后移一位，并在_Where 所指处插入新的元素，其值是_Val。容器也支持一次插入多个相同的元素，其方法是：

```
void insert(  iterator _Where,      // 插入位置的迭代器
         size_type _Count,          // 插入元素的数量
         const Type& _Val  );       // 插入元素的值
```

例如：

```
vector<int> a;                       // 定义一个向量 a
a.insert( a.begin(), 1 );            // 在开头插入 1
a.insert( a.begin(), 2 );            // 在开头插入 2，结果是 2，1
a.insert( a.begin()+1, 2, 3 );       // 在第二个元素处插入两个 3，结果是 2，3，3，1
```

除上述 insert()的重载版本外，向量还支持插入另一个容器的一个区段，方法是使用别的容器的两个迭代器指明的区间。不过该方法是一个函数模板，需要用其他容器的迭代器类型来实例化。其声明如下：

```
template<class InputIterator>      // 模板参数
void insert( iterator _Where,      // 插入位置
         InputIterator _First  ,   // 区间头
         InputIterator _Last );    // 区间尾
```

例如：

```
vector<int>  a;                                    // 定义两个向量
vector<int>  b;
……
a.insert( a.begin() , b.begin(), b.end() );    // 在 a 的开始处插入 b 向量的全部
int aryp[10];                                      // 定义一个数组
……
a.insert( a.begin(), &ary[0], &ary[9] );       // 在 a 的开始处插入数组的全部
```

vector 的构造函数也有一个使用迭代器区间的版本。其用法同 insert 的对应版本一致，即可以用一个迭代器区间内的元素构造出一个新的向量对象。

20.6.6 迭代器的有效性

使用迭代器时要保证其有效性。随着容器元素的增删，迭代器指向的元素很可能会发生变化。比较严重的情况是迭代器所指的元素可能已被删除，而迭代器指向的部分则可能已经不属于容器。所以，使用迭代器时要时刻注意其有效性。各种可能情况如图 20.18 所示。

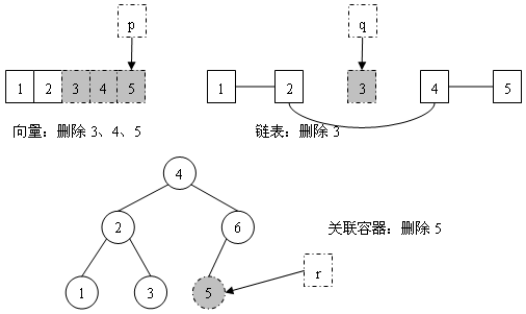


图 20.18 各种可能情况

在上图中，迭代器 p，q，r 分别指向了不同容器中的元素。如果所指元素被删除，而迭代器所指向的位置（内存）并没有变，那么此时迭代器就已经无效了。对这样一个无效的迭代器进行操作有可能导致意想不到的错误，严重的会导致程序崩溃。

20.7 适配器

适配器实际上是一种转换器。其设计初衷是在不改变已有类的原则上，使用已有类的功能，并提供新的接口，从而创建出符合特定目的的新类。这种新类就是所谓的适配器。例如，容器的反向迭代器就是正向迭代器的适配器。

正向迭代器的步进都从容器头开始，向着容器尾的方向。但反向迭代器正好相反，其步进从容器尾开始，向着容器头方向。然而正是这个原因，可以利用正向迭代器实现反向迭代器，只要在反向迭代器的自加运算符“++”的实现中调用正向迭代器的自减运算符“--”即可。同理，可以在反向迭代器的加号运算符“+”的实现中调用正向迭代器的减号运算符“-”。

```
template<typename T>
class iterator                                  // 正向迭代器模板
{
public:
    iterator & operator ++();                   // 步进，前置自加运算符
    iterator & operator ++(int);                // 步进，后置自加运算符
    iterator & operator –();                    // 步退，前置自减运算符
    iterator & operator –(int);                 // 步退，后置自减运算符

    iterator & operator +(int n);               // 正向跳跃，加号运算符
    iterator & operator –(int n);               // 反向跳跃，减号运算符
};
template <typename T>
class reverse_iterator                          // 反向迭代器
{
private:
    iterator<T> m_iter;                         // 用正向迭代器作为内部实现
public:
    reverse_iterator & operator ++()            // 步进，前置自加运算符
    {
        return --m_iter;                        // 使用正向迭代器的步退
    }
    reverse_iterator & operator ++( int )       // 步进，后置自加运算符
    {
        return --m_iter;                        // 使用正向迭代器的步退
    }
    reverse_iterator & operator –()             // 步退，前置自减运算符
    {
        return ++m_iter;                        // 使用正向迭代器的步进
    }
    reverse_iterator & operator –( int )        // 步退，后置自减运算符
    {
        return ++m_iter;                        // 使用正向迭代器的步进
    }
    reverse_iterator & operator +( int n )      // 正向跳跃，加号运算符
    {
        return m_iter - n;                      // 使用正向迭代器的减号运算符
    }
```

```
    reverse_iterator & operator --( int n )// 反向跳跃，减号运算符
    {
        return m_iter + n;                  // 使用正向迭代器的加号运算符
    }
}
```

上述迭代器适配器只是示意，真正的迭代器的实现还有比较多的内容，这里没有一一列出，请读者参考相关的文档，或者 STL 的实现版本。

除了容器适配器，迭代器和算法也有适配器，这些将在相关章节中进行说明。

20.8 小结

本章主要讲述的是 STL 形成的历史、各种主要版本以及主要组成部分。STL 的主要创始人是出生于莫斯科的 Alexander Stepanov。他在上个世纪 80 年代末开始研究 C++语言的泛型编程。当 C++模板成熟之后，Alexander 开发了一套以 C++模板为基础的泛型算法库，这就是 STL 的雏形——HP STL。此后形成的各个 STL 版本大都以 HP STL 为蓝本。在 1994 年的滑铁卢会议上，STL 形成了正式提案，并于 1998 年正式成为 C++语言标准的一部分。

STL 的主要内容包括六大部分——容器、算法、迭代器、函数对象、适配器和配置器。除去配置器比较底层，其他各个部分在日常编程开发中都有非常重要的意义。容器用来存储和组织数据，迭代器为遍历容器提供一个统一的界面，算法通过迭代器访问容器，算法也可以使函数对象改变其计算策略。另外，通过适配器还可以在已有容器、算法、迭代器的基础上，创建出新的容器、算法和迭代器。

◆ ◆ ◆

第 21 章　序列式容器

本章包括

- vector（向量）的定义和使用
- list（双向链表）的定义和使用
- deque（双端队列）的定义和使用
- stack（栈）的定义和使用

本章主要讨论各种序列式容器的定义和使用。所谓序列式容器，就是说其中的元素是按照某种顺序存放的。但是，这里所说的顺序并不是元素之间的大小顺序，而是元素加入容器时的顺序。当在序列式容器中添加元素时，元素会被放置在指定的位置（除非对容器进行排序等操作），这一点是序列式容器与关联式容器最大的区别，后者会在插入元素时进行自动排序。

21.1　向量 vector

使用 vector 的主要目的是替换数组。C++数组最大的优点就是可以使用下标随机访问数组元素。但其缺点也很明显，不仅需要时刻防备下标越界的危险，还要注意空间不足的情况。当然也可以定义一个类，支持动态数组，当内存不足时，自动分配新的内存，将数据复制到新内存后，释放原来的内存。为了能够存储不同类型的数据，可以将该类定义成类模板。这就是向量 vector 的原型。

21.1.1　vector 概述

vector 容器定义在标准头文件 vector 中。其行为非常类似于数组，存储在其中的数据在内存中也是连续的，而且可以通过下标进行随机存取。vector 同数组最大的区别在于其内存空间可以动态增长，即当需要更大的内存空间时，vector 可以自动增长，同时又不破坏原来的数据。

这是一种假象，是由包装在 vector 中的一些成员函数实现的。其真实的过程是：当 vector 发现其容量不够时，首先开辟一块儿新的空间，新空间要比原有的空间大，足可以容纳 vector 原有的元素以及新添加的元素；然后再把原有的元素和新添加的元素复制到新的内存空间中，如图 21.1 所示。

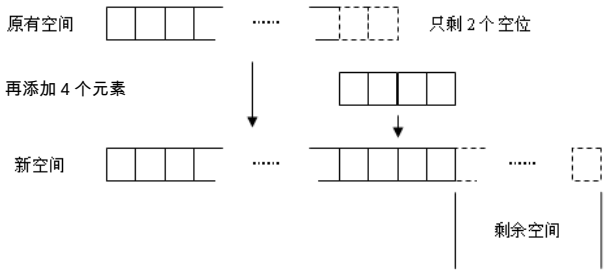


图 21.1　vector 容器的“增长”

虽然用数组也可以实现内存空间的增长，但是需要开发者实现所有的细节，包括内存空间的分配以及元素的复制，使用起来非常不方便。使用 vector 这样的类模板可以将这些底层实现细节包装起来，而不必由用户实现。

vector 的优点是：其内存空间是连续的，查找元素很快，可以像数组一样用下标进行访问，内存空间可以随需要增长。其缺点也同样明显：插入和删除元素较慢，原因是要保证内存空间连续。所以 vector 适用于在元素数量变化不大或者只需要增加元素的情况下，如果需要频繁地插入、删除元素，则应当考虑使用别的容器。

所有的 STL 容器类都是类模板，所以在使用时需要指定其存储元素的类型，vector 也不例外。下面的例子就展示了如何使用 vector 存储整型数据，并用下标进行访问。

```
vector<int> a(2);          // 定义一个保存整型数据的向量，并设置其长度为 2
cout<<a.size()<<endl;      // 输出向量 a 的长度，结果是 2
a[0] = 4;                  // 为第一个元素（下标为 0）赋值
// a[2] = 5;               // 错误，向量 a 中目前只有两个元素，a[2]越界访问
a.push_back( 6 );          // 从后面添加元素，新添加的元素下标为 2
cout<<a.size()<<endl;      // 输出向量 a 的长度，结果是 3
```

向量的长度不等同于向量可存储元素的数量，后者大于等于前者。一般来讲用下标访问向量时，其值不要超过“长度-1”。尽管有时超过了也不会犯错，但不符合逻辑。

21.1.2 构造 vector

向量的构造函数有多个，其功能基本上都是设置向量的初始长度，以及初始化向量的值。向量也有默认的构造函数，其结果是一个长度为 0 的向量对象。另外，向量也有复制构造函数。如下所示：

```
vector();                                  // 默认构造函数，长度为 0
explicit  vector( size_type _Count );      // 构造长度为_Count 的向量
vector( size_type  _Count ,
      const Type&  _Val );                 // 以初始值为_Val、类型为 Type 生成_Count 个元素
vecotor( const  vector&  _Right );         // 用另一个 vector 容器来初始化
```

其中，size_type 和 Type 是 STL 中常用的类型定义。为了能够更好地描述代码逻辑，STL 经常将一些意义不明显的数据类型用一种更有意义的名字来代替，例如：

```
typedef size_t size_type;          // 用 size_type 表示 size_t
```

而 Type 则是向量这个类模板实例化时用的模板参数。比如用 int 型数据实例化 vector 则 Type 的实际类型就是 int，用 string 类型实例化，则 Type 的实际类型就是 string。不过在不同版本的 STL 中，这个定义可能是不同的，比如在 SGI 的版本中就定义成 value_type。下面的代码展示了不

同的构造函数的用法：

```
vector<int> a;                  // 构造一个长度为 0 的向量
vector<int> b( 12 );            // 构造一个长度为 12 的向量
vector<int> c( 3, 0 );          // 构造一个长度为 3 的向量，并将其元素初始化为 0
vector<int> d( c );             // 从向量 c 复制构造一个向量 d
```

向量构造后的长度只是其初始长度，将来随着元素数量的增加，向量的长度也有可能不断增长。另外也可以通过调用成员函数 resize()来改变向量的长度。对于仅指定向量长度、未指定元素初始值的构造函数，生成的向量中将用元素的默认值来初始化。例如对于 C++内置的数据类型，如 int，double 等，就用 0 作为初始值；而对于类，则使用类的默认构造函数。

由于向量在构造时需要调用元素类型的默认构造函数，因此，如果要用向量来保存一个类的对象，则该类应当具有默认构造函数。这一原则适用于所有容器。

21.1.3 处理 vector 的元素

有关 vector 元素的操作无非是增、删、改、查。对于容器的查找 STL 提供了统一的算法，如 find，findif 等。这些算法将在“函数对象和算法”一章中说明，这里只说明如何在 vector 中增加、删除和修改元素。

在 vector 中增加元素可以采用 push_back()方法，其作用是将新元素挂接在向量的尾部。所谓向量的尾部，指的是当前向量中的最后一个元素，新元素将放在紧邻这个元素后的下一个存储单元中。如果需要，还有可能增加向量的内存空间。push_back()方法的声明如下：

```
void push_back( const  Type&  _Val );
```

例如：

```
vector<int> vInts;                  // 定义一个保存整型数的向量
for (int i = 0; i < 10; ++i)        // 从 0 到 9
{
    vInts.push_back(i);             // 依次将元素挂接在向量尾部
}
```

上述程序的作用是将 0 到 9 这 10 个整数存放到一个向量中。上述代码在功能上没有错误，但其实存在比较严重的效率问题。原因是利用 push_back()挂接元素时，很容易引起向量内存的重新分配，如图 21.2 所示。

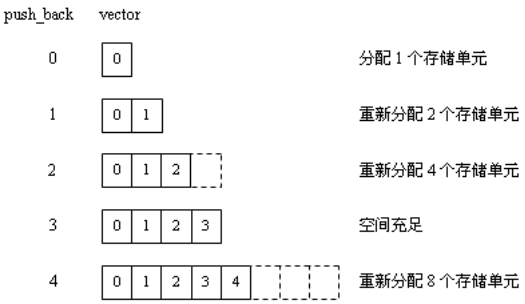


图 21.2 push_back()与空间分配

由图可知，每调用 push_back()一次，就有可能导致向量内存空间的重新分配，并将原有的数据复制到新的内存空间中，同时还要删除原来的内存空间。这样做的效率显然非常低。

向量在重新分配内存空间时，一般新的内存空间的长度是当前长度的两倍。这样做的好处是不用加一个元素就可重新分配一次，提高了效率。新增空间的总长度称做容量，可以通过调用成员函数 capacity()获得；而其中被使用的部分称做向量的长度，可以通过调用成员函数 size()获得。

要解决 push_back()效率低下的问题可以先调用 reserve()方法。该方法的作用是为向量保留指定长度的内存空间。如果当前向量空间的容量充足，即比指定长度还要大，则该方法什么都不做；如果当前向量空间的容量不足，该方法将导致向量空间立即重新分配。如图 21.3 所示就是调用 reserve()方法之后再挂接元素的情形。显然向量的内存空间始终充足，不用重新分配，从而在较大程度上提高了程序的效率。

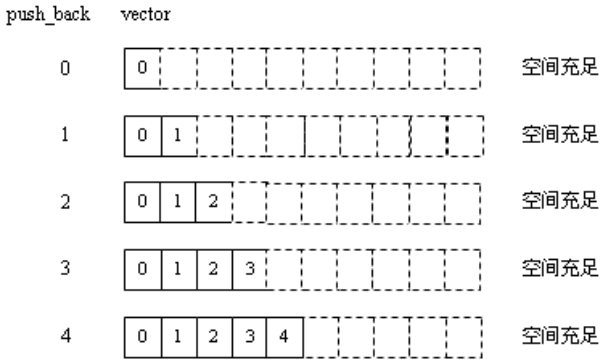


图 21.3 保留空间下的操作

对于删除元素，可选的操作比较多。如果只删除向量尾部的元素，可以使用成员函数 pop_back()，如果要删除全部元素可以使用 clear()。这些函数都只会影响向量的长度，而不会影响向量的容量。修改向量元素的方法非常类似于数组的操作。由于向量可以通过下标进行访问，所以可以直接用向量名加下标找到指定元素，然后直接赋值即可。

假设一个整型向量 a 具有足够的长度，则给每个元素赋值的代码如下：

```
for( int i=0; i<a.size(); i++ )     // 遍历整个向量
{
    a[ i ] = i;                     // 给每个元素赋值
}
```

21.1.4 交换两个容器的元素

STL 中的容器大都支持这样一个方法——交换两个容器中的元素。向量也不例外。交换两个容器中元素的方法是 swap()。该方法接受一个同类型的容器作为参数，其声明如下：

```
swap( vector &other );
```

swap()是类模板 vector 的成员方法。如果一个类模板的成员函数使用模板自身作为参数类型 则不必在模板名后面加上实例化参数。所以 swap()的参数列表中的 vector 没有带<value_type>后缀。

例如，交换两个向量的元素的代码如下：

```
vector<int> a( 3, 1 );      // 定义一个向量，元素是 1，1，1
vector<int> b( 2, 3 );      // 定义一个向量，拥有 3，3
a.swap( b );                // 交换 a 和 b，此时 a 的元素是 3，3，而 b 的元素是 1，1，1
```

除了成员方法之外，序列式容器（如 vector，list，deque 等）还提供一个全局方法 swap()。该方法是上述容器的友元方法，并可以用于交换两个同种容器的元素。

21.1.5 使用向量的实例

在实际生活中很多地方都需要排队，如公交车站、ATM 取款机等。由于很难确定实际排队的人数，所以用数组来模拟一个队伍并不合适，而容器 vector 由于可以自动增长空间，是一个比较好的选择。

下面的程序用来模拟排队。首先用户依次输入排队者的编号，程序依次将这些编号保存到一个 vector 容器中。为了减少一个队伍的压力，当人数达到一定数量时，比如 5，就自动将队伍分成两队。程序如示例代码 21.1 所示。

示例代码 21.1

```
#include <cstdlib>
#include <iostream>
#include <vector>                                   // 使用 vector 容器
using namespace std;                                // 使用名称空间 std
int main(int argc, char *argv[])                    // 主函数
{
    cout<<"——用 vector 管理队伍——"<<endl;          // 输出提示信息
    cout<<endl;

    vector<int> vecA;                               // 定义两个向量
    vector<int> vecB;

    vecA.reserve( 10 );                             // 保留内存空间
    vecB.reserve( 20 );

    int id = 0;                                     // 保留编号的变量
    cout<<"依次输入排队者的编号。输入-1 退出。"<<endl;
    cin>>id;                                        // 输入排队者的编号
```

```
    while( id != -1 )                           // 输入-1 结束循环
    {
        vecA.push_back( id );                   // 优先在 vecA 中排队
        if( vecA.size() >= 10 )                 // 如果 vecA 中排队者的数量超过 10
        {
            vecB.insert( vecB.end(),      // 将第 5 个以后的所有排队者放到 vecB 中
                     vecA.begin() + 5,
                     vecA.end() );
            vecA.erase( vecA.begin() + 5, vecA.end() );// 删除 vecA 中的部分排队者
        }
        cin>>id;                            // 输入排队者编号
    }
    cout<<endl;

    vector<int>::iterator iter;             // 向量的迭代器
    if( vecA.size() > 0 )                   // 如果 vecA 中有排队者
    {
        cout<<"队伍 A 中的排队者：<<endl;
        iter = vecA.begin();
        while( iter != vecA.end() )         // 遍历 vecA
        {
            cout<<*(iter++)<<' ';           // 输出排队者的编号
        }
        cout<<endl;
    }
    if( vecB.size() > 0 )                   // 如果 vecB 中有排队者
    {
        cout<<"队伍 B 中的排队者：<<endl;
        iter = vecB.begin();
        while( iter != vecB.end() )         // 遍历 vecB
        {
            cout<<*(iter++)<<' ';           // 输出排队者的编号
        }
        cout<<endl;
    }

    cout<<endl;
    system("PAUSE");                        // 等待用户反应
    return EXIT_SUCCESS;                    // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 21.4 所示。

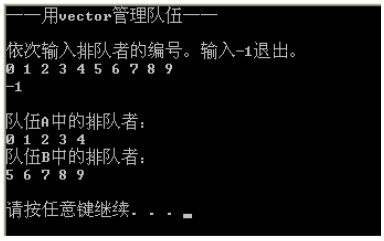


图 21.4　用 vector 管理队伍结果

上述代码使用了容器向量常见的一些操作，包括 reserve，push_back，insert，erase 等，以及用迭代器访问容器的方法。

21.2　双向链表 list

虽然向量解决了数组存在的一些问题，但向量从本质上来讲还是一种数组，其元素在内存上是连续排列的，因此向量的插入和删除操作效率比较低。为了弥补向量的不足，STL 引入了双向链表。

21.2.1　list 概述

所谓链表指的是这样一种容器：元素不是连续排放的，而是靠元素与元素间的联系构成一个容器。在比较典型的链表中，元素与元素之间是靠指针联系的，即每一个元素都保存一个指向下一个元素的指针。由于只有一个方向的联系，所以这种链表也称做单向链表，如图 21.5 所示。

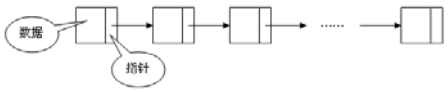


图 21.5　单向链表

由于单向链表只有一个方向的联系，所以只能沿着一个方向遍历，这样在使用的时候未免会受到限制。因此，可以给元素再添加一个指针，指向其前面的元素，这样就构成了一个双向链表，如图 21.6 所示。

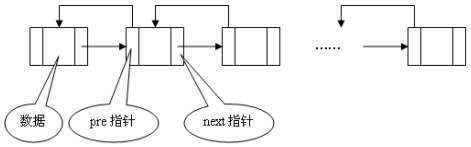


图 21.6　双向链表

链表也可以是环形的。对于非环形的单向链表或者双向链表，其最后一个节点指向下一个节点的指针是一个空指针，但环形链表的这个指针指向链表的第一个节点。同理，对于双向环形链表，其第一个节点的指向前一个节点的指针，应当指向链表的最后一个节点。

链表中的元素也称做“节点”。节点的定义一般有两部分，一部分用于存储数据，另一部分用于链接其他元素。对于双向链表，节点的链接部分则是由两个指针组成的，一个用于指向前面的元素，另一个用于指向后面的元素。

由于链表的内存空间不必是连续的，所以加入新元素受限制较少。只要计算机中有足够的内存空间就可以添加成功。因此，也不用像向量那样预先分配内存空间。所以，链表没有容量的概念，而只有长度的概念，即链表中元素的个数。

链表的优点是节点的插入、删除操作效率较高。这主要是因为链表的节点间不是紧邻排列的，而是通过指针关联起来的，这样在插入、删除时只要重新设定指针的指向即可，而不用移动节点。如图 21.7 所示就是一个删除链表某个节点的示意。

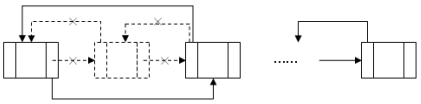


图 21.7 删除链表节点

链表查找比较慢，主要是因为元素间不是紧邻排列的，所以不能使用下标访问某个节点，必须利用元素间的关联指针，逐个对比。

21.2.2 构造 list

STL 中使用的链表就是双向链表，而且所有有关节点定义、节点关联的内容都通过 list 类封装起来了，使用起来非常方便。list 容器定义在头文件 list 中，使用时要引入头文件和命名空间：

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

list 的构造函数同 vector 的构造函数非常相似，而且具有同样的重载版本，都有默认的构造函数、复制构造函数以及利用迭代器区间的构造函数，如下所示：

```
list( );                                        // 默认构造函数
explicit list( size_type _Count );              // 构造几个空节点，但没有给定节点数据
list( size_type _Count, const Type& _Val );// 构造几个节点，并给定数据
list( const _list& _Right );                    // 复制构造函数
template<class InputIterator>                   // 利用迭代器区间的构造函数
list( InputIterator _First, InputIterator _Last );
```

例如：

```
list<int> a;                    // 构造一个长度为 0 的链表
list<int> b(12);                // 构造一个长度为 12 的链表
list<int> c( 3, 0 );            // 构造一个长度为 3 的链表，并将其元素初始化为 0
list<int> d( c );               // 从链表 c 复制构造一个链表 d
list<int> e( d.begin(), d.end() ); // 从一个迭代器区间构造出一个链表 e
```

利用迭代器区间构造一个新的链表，则这个迭代器可以是任意类型容器的迭代器，而不必是链表容器的迭代器。

21.2.3 处理 list 的节点

向量由于其数据存储结构的设计，只能在尾部进行 push 和 pop 操作，即只有 push_back()和 pop_back()方法。如果要在向量头部进行 push 和 pop 操作，则会带来较大的效率问题。链表则没有这个限制，因为在链表头部增删节点并不需要移动后面的节点，而只要重新设置原来头部节点的指针即可。所以链表可以在容器的两端进行 push 和 pop 操作，即除了 push_back()和 pop_back()方法外，链表还有 push_front()和 pop_front()方法，如下所示：

```
void push_front( const value_type & val );
void pop_front();
```

除了 push 和 pop 方法外，向链表中补充元素还可以使用 insert()方法。同向量的插入方法一样，链表的插入也支持插入一个元素、插入多个同值的元素，并且也可以插入某个迭代器区间内的所有元素。

在链表中删除一个元素可以使用 remove()方法。该方法的使用不同于向量的 erase()方法，因为 erase()需要指定元素的下标，而 remove()方法则只要给出元素的值即可。也就是说，给定元素的值，链表会自动查找该值所对应的所有节点，并将其删除。其声明如下：

```
void remove( const value_type &_val );
```

例如：

```
list<int> a;                              // 定义一个链表
int ary[5] = { 1, 2, 3, 2, 4 };           // 定义一个数组
a.insert( a.begin(), &ary[0], &ary[4] + 1 );   // 将数组中的元素插入到链表中
a.remove( 2 );                            // 删除所有值等于 2 的元素
list<int>::iterator iter = a.begin();  // 定义一个迭代器，指向链表的开头
while( iter != a.end() )                  // 遍历链表
{
    cout<< *(iter++) <<' ';               // 输出各个元素
}
```

上述代码的输出结果如下：

```
1 3 4
```

从链表中删除元素还有一个 remove_if()方法，不过该方法需要使用一个函数对象作为是否删除某个元素的条件。函数对象的概念将在后面的相关章节中进行说明。

在链表中删除重复的元素还可以使用 unique()方法。如果链表中某些相邻元素的值是相等的，则调用该方法后，只保留这些元素中的第一个。例如，假设链表 a 中的元素是 1，2，3，2，2，则调用 unique()方法后，a 中的元素将是 1，2，3，2，原来链表中的第 5 个元素因为与跟它紧邻的第 4 个元素相同，所以被删除。

链表也可以排序，方法是调用链表的成员方法 sort()，不过默认的排序方式是从小到大。如果要从大到小排列，则应当使用 sort()的一个重载版本，不过该版本需要使用一个函数对象。从大到小排列一个链表还有一个间接的方法，那就是先从小到大排一遍，然后调用 reverse()方法将链表反向排列。例如：

```
list<int> a;                              // 定义一个链表
```

```
int ary[5] = { 1, 4, 3, 2, 5 };          // 定义一个数组
a.insert( a.begin(), &ary[0], &ary[4] + 1 );   // 将数组中的元素插入到链表中
a.sort();                                // 从小到大排列，a 的内容是 1，2，3，4，5
a.reverse();                             // 反向链表，a 的内容是 5，4，3，2，1
```

21.2.4 链表的拼接和融合

链表的拼接和融合是链表的独特功能，其作用都是将一个（源）链表的全部（或部分）节点加入到另一个（目标）链表中，并将源链表清空（或只删除插入的部分）。这样原来两个链表将合二为一。其中拼接的方法是 splice()，融合的方法是 merge()。splice()方法有三个重载版本，其声明如下：

```
void splice( iterator _Where,  list& _Right );
void splice( iterator _Where,  list& _Right,  iterator _First );
void splice( iterator _Where,  list<Allocator>& & _Right,  iterator _First,
iterator _Last );
```

其中第一个参数都是一个迭代器，表明目标链表插入的位置。其余参数则分别表明插入的源链表以及源链表插入内容的起始位置和结束位置。例如：

```
int ary1[3] = { 1, 2, 3 };               // 定义两个整型数组
int ary2[3] = { 4, 5, 6 };
list<int> a( &ary1[0], &ary1[2]+1 );     // 定义两个链表，并用数组初始化
st<int> b( &ary2[0], &ary2[2]+1 );;
list<int>::iterator iter = a.begin();    // 定义一个链表的迭代器，并指向链表 a 的开头
a.splice( ++iter, b );                   // 将链表 b 拼接到链表 a 中，执行后 b 为空
```

上述代码执行后，a 的内容变为 1，4，5，6，2，3，而 b 的内容则为空。链表拼接并没有对拼接后形成的新链表执行排序操作，而链表融合操作除了将两个链表合二为一外，还会按照指定的顺序对链表进行排序。默认的排序方法是由小到大，当然也可以指定由大到小，不过要使用一个函数对象。链表融合的函数声明如下：

```
void merge();
```

链表的融合操作一般是针对两个已经排好序的链表的。在进行融合时，根据元素间的大小关系，依次将源链表的每个元素插入到目标链表的合适位置。例如，假设一个目标链表是 1，3，5，源链表是 2，4，6，则两个链表融合的结果就是 1，2，3，4，5，6，程序如下：

```
int ary1[3] = { 1, 3, 5 };               // 定义两个整型数组
int ary2[3] = { 2, 4, 6 };
list<int> a( &ary1[0], &ary1[2]+1 );     // 定义两个链表，并用数组初始化
list<int> b( &ary2[0], &ary2[2]+1 );
a.merge( b );                            // 融合链表 a 和 b
```

上述代码执行后，链表 a 就是两个链表融合的结果，而 b 则被清空。

对于未排序的链表，虽然也可以调用 merge()方法，但最终的结果链表并不排序。所以，在调用 merge()之前，最好先对源链表和目标链表调用 sort()方法排序。

21.2.5 list 的反向迭代器

同向量容器一样，链表容器也提供了一个双向迭代器 iterator。不过除此之外，链表还提供了一个反向迭代器 reverse_iterator。严格来讲，反向迭代器也是一个双向迭代器，也可以前后移动。不过反向迭代器的前后方向与一般双向迭代器的前后方向正好相反。reverse_iterator 的前向迭代从容器的尾部开始，向容器头部前进，而其后向迭代则从容器头部开始，向着容器尾部前进。为了配合反向迭代器，容器提供了两个方法用来初始化其开始和结尾，分别是 rbegin()和 rend()。正向迭代和反向迭代示意如图 21.8 所示。

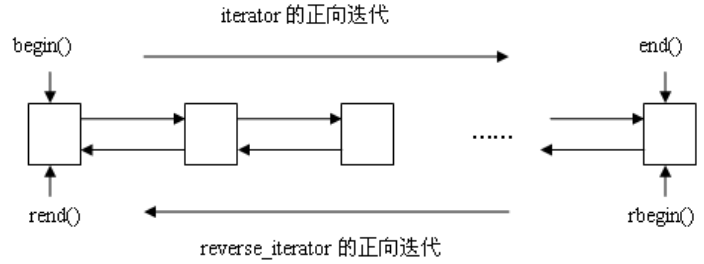


图 21.8　正向迭代和反向迭代示意

例如，可以用反向迭代器，从后向前依次输出每个元素，程序如下：

```
int ary[3] = { 1, 2, 3 };                    // 定义一个数组
list<int> a( &ary[0], &ary[2]+1 );           // 定义一个链表，并用数组初始化
list<int>::reverse_iterator riter = a.rbegin();// 定义一个反向迭代器，并初始化
while( riter != a.rend() )                   // 遍历整个链表
{
    cout<<*( riter++ )<<' ';                 // 反向迭代器向前迭代，并输出每个元素
}
```

虽然链表中的元素是 1，2，3，并且迭代器向前迭代（++），但由于是反向迭代器，所以上述代码实际是从后向前迭代，并依次输出 3，2，1。

向量由于在插入和删除元素时效率较低，所以用向量管理队伍并不是一个好选择，而用链表管理则可以很好地解决效率方面的问题。还可以在需要时对队伍进行排序，将多个队伍按照顺序组成（融合成）一个队伍。在下面的程序中，用户依次输入两个队伍中排队者的编号，输入完之后，按照编号对两个队伍进行排序，并最终将两个队伍合为一个队伍，如示例代码 21.2 所示。

示例代码 21.2

```
#include <cstdlib>
#include <iostream>
#include <list>                                  // 包含 list 头文件
using namespace std;                             // 使用名称空间 std
int main(int argc, char *argv[])                 // 主函数
{
    cout<<"——用链表管理队伍——"<<endl;             // 输出提示信息
    cout<<endl;

    list<int> a, b;                              // 定义两个空链表
    int id = 0;                                  // 定义表示编号的整型变量
```

```
    cout<<"请输入 A、B 两个队伍中排队者的编号：" // 输出提示信息
        <<"（输入-1 结束）"<<endl;
    cout<<"A 队伍："<<endl;

    cin>>id;                                  // 输入编号
    while( id != -1 )                         // 输入-1 结束
    {
        a.push_back( id );                    // 保存编号到队伍中
        cin>>id;                              // 继续输入
    }

    cout<<"B 队伍："<<endl;                   // 输出提示信息
    cin>>id;                                  // 输入编号
    while( id != -1 )                         // 输入-1 结束
    {
        b.push_back( id );                    // 保存编号到队伍中
        cin>>id;                              // 继续输入
    }
    cout<<endl;

    list<int>::iterator iter;                 // 定义一个迭代器
   if( a.size() > 0 )                         // 如果 A 队伍不空
   {
        cout<<"A 队排序："<<endl;
        a.sort();                             // 队伍排序
        iter = a.begin();
        while( iter != a.end() )              // 遍历队伍
        {
            cout<<*(iter++)<<' ';             // 输出排队者编号
        }
        cout<<endl;
    }

   if( b.size() > 0 )                         // 如果 B 队伍不空
   {
        cout<<"B 队排序："<<endl;
        b.sort();                             // B 队伍排序
        iter = b.begin();
        while( iter != b.end() )              // 遍历 B 队伍
        {
            cout<<*(iter++)<<' ';             // 输出排队者编号
        }
        cout<<endl;
    }

    if( a.size() > 0 && b.size() > 0 )        // 如果两个队伍都不空
    {
        cout<<"A、B 两队融合："<<endl;
        a.merge( b );                         // 融合两个队伍
        iter = a.begin();
        while( iter != a.end() )              // 遍历融合后的队伍
        {
            cout<<*(iter++)<<' ';             // 输出排队者的编号
        }
    }

    cout<<endl;
    system("PAUSE");                          // 等待用户反应
    return EXIT_SUCCESS;                      // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 21.9

所示。

图 21.9　用 list 管理队伍结果

上述程序主要是展示 list 的用法，list 在插入和删除元素方面有很高的效率，很适合管理类似队伍这种经常变化的数据结构。

21.3 双端队列 deque

向量和链表在使用中各有优缺点。例如向量能够使用下标，随机访问元素，但在插入、删除元素时效率较低；而链表则与此相反，插入、删除元素效率较高，但不能使用下标随机访问。如果能够融合这两种容器，取长补短，那就既能随机访问元素，又具有较高的效率，这就是设计双端队列 deque 的原因。

21.3.1 deque 概述

与 list 类似，deque 可以在容器两端进行插入、删除操作，即拥有成员函数 push_back()，pop_back()，push_front()，pop_front()。同时，deque 也同 vector 一样，提供了一个利用下标随机访问元素的重载运算符“[]”。所以，deque 可以看做一个两端开口的连续内存空间，如图 21.10 所示。

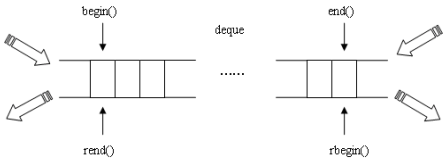


图 21.10　deque 示意图

不过，只是表面看起来是这样，实际上 deque 并不是“一段”连续的内存，而是分段连续的，即 deque 由多段连续的内存组成，各段之间通过某种手段联系起来，然后通过对各种操作的特殊处理，从而使得 deque 的行为既象 vector，又像 list。其内存结构如图 21.11 所示。

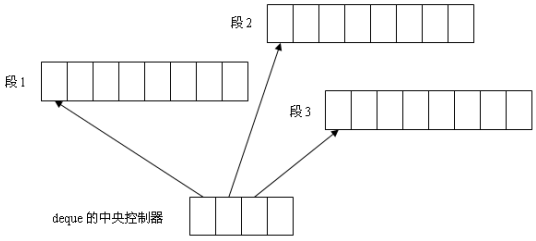


图 21.11 deque 的内存结构

在 deque 中有一个中央控制器，记录 deque 拥有的各段的首地址。每段可以存储 8 个元素，并且每段自身都是连续的内存空间，但段与段之间不是连续的。虽然 deque 的各段不是连续的，但仍然可以像 vector 一样使用下标随机访问。这就需要在重载 deque 的运算符“[]”时，先求得下标所对应的段，再求得在该段中的位置，然后才能求得所需元素。

正因为 deque 是分段的，在 deque 中进行插入和删除操作仅影响元素所在的段，而不会影响整个容器，所以 deque 的效率比 vector 要高，比较类似于 list。另外，在设计 deque 的迭代器时，也需要精心处理跨段的情况。

21.3.2 使用 deque

既然 deque 综合了 vector 和 list 的优点，所以其用法同 vector 和 list 很相似。比如其构造方法基本一致，都具有默认构造函数、复制构造函数、利用迭代器区间的构造函数等；可以利用下标访问元素，较快地进行插入和删除操作，在容器两端都可以进行 push 和 pop 操作。下面的例子集中展现了 deque 的各种操作，如示例代码 21.3 所示。

示例代码 21.3

```
#include <cstdlib>
#include <iostream>
#include <vector>                              // 包含 vector 头文件
#include <deque>                               // 包含 deque 头文件
using namespace std;                           // 使用名称空间 std
int main(int argc, char *argv[])               // 主函数
{
    cout<<"——使用 deque——"<<endl;              // 输出提示信息
    cout<<endl;

    int ary[5] = {1, 2, 3, 4, 5};              // 定义一个数组
    vector<int> v( &ary[0], &ary[4]+1 );       // 定义一个向量，并用数组初始化

    deque<int> a( v.begin(), v.end() );        // 定义一个双端队列，并用向量初始化
    a.push_front( 0 );                         // 在前后两端各插入一个元素
    a.push_back( 6 );

    cout<<"Front: "<<a.front()<<endl;          // 输出前后两个元素
    cout<<"Back: "<<a.back()<<endl;

    deque<int>::iterator iter = a.begin(); // 定义一个迭代器
    while( iter != a.end() )                   // 遍历整个容器
    {
        cout<<*(iter++)<<' ';                  // 输出迭代器所指元素，并指向下一个元素
    }
```

```
    cout<<endl;

    cout<<"a[3] = "<<a[3] << endl;          // 使用下标访问元素
    a.erase( a.begin() );                   // 删除第一个元素
    a.pop_front();                          // 删除头部元素
    a.pop_back();                           // 删除尾部元素
    cout<<"Front: "<<a.front()<<endl;       // 输出前后两个元素
    cout<<"Back: "<<a.back()<<endl;

    cout<<endl;
system("PAUSE");                          // 等待用户反应
return EXIT_SUCCESS;                      // 主函数返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 21.12 所示。

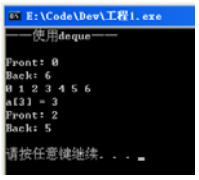


图 21.12　使用 deque 结果

上述代码使用了 deque 的各种操作，其中包括利用下标访问元素，以及在容器两端进行 push 和 pop 操作。

21.4 容器适配器

所谓适配器，就是对已有类的一个封装类。当程序中使用某个类时，由于该类提供的方法不符合要求，或者提供了过多的方法，不能直接应用该类。为了复用该类，可以将这个类封装在另一个类内，后者就是一个封装类。在封装类向外部提供接口时，可以提供同内部类一样的接口，也可以限制提供的接口，甚至可以提供新的接口。

在后文中将要介绍的 stack 容器和 queue 容器，都不是独立的容器，而是建立在之前已经介绍过的容器基础之上的，只是在已存在的容器基础之上提供了一些特殊的接口去处理数据，是一种容器适配器。例如 stack 容器的底层是由 deque 实现的，deque 是双端队列，将其一端封住，只允许在另一端插入、删除数据，便形成了 stack。

stack 和 queue 的底层默认都是由 deque 实现的。在定义一个 stack 或 queue 对象时，也可以选用别的容器，只要其支持相关的操作即可。

21.5 栈 stack

本节将详细讲解另外一种容器——stack，首先讲解 stack 的概念，然后介绍如何使用 stack。

21.5.1 stack 概述

stack 包含在标准头文件 stack 中，使用时要像下面一样引入：

```
#include <stack>
```

我们已经知道 stack 的底层实现是 deque，来看一看模板类 stack 的声明：

```
template <
   class value_type,
   class Container=deque<value_type>
>
class stack;
```

可以看到 template 的第二个参数 Container 默认指定的是 deque ,这就是 stack 是适配容器的证据。stack 容器不提供迭代器，任何元素的进出都必须符合先进后出的规则，每次取用只能使用栈顶元素，而且也没有办法遍历整个容器中的内容，如图 21.13 所示。

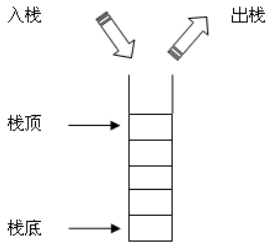


图 21.13 栈的示意图

21.5.2 使用 stack

相对于 deque 本身，stack 提供了非常少的方法。对于元素的增删，通过 push 和 pop 方法去实现，即 push 是放入元素，pop 是弹出元素。如果要访问容器中的元素，每次只能通过 top 方法访问栈顶的元素。例如：

```
stack<int> skInts;
skInts.push(1);
skInts.push(2);
cout << skInts.top() << endl;  // 输出 2
skInts.pop();
cout << skInts.top() << endl;  // 输出 1
skInts.pop();
cout << "stack 大小是：";
cout << skInts.size();         // 输出 0
```

分析上面的程序，第一行声明一个整数类型的 stack 容器，第二行、第三行向容器中顺序放入两个元素 1 和 2，第四行访问栈顶元素，即最后一个进栈的元素 2，第五行将栈顶元素 2 从栈中移除，这时栈中只有一个元素。第六行、第七行执行完以后，栈为空，所以调用 size 方法的返回值为 0。stack 容器也提供了 empty 方法来验证容器是否包含元素，使用这个方法，也可以通过遍历的方法取出栈中的元素：

```
……
while(skInts.size())
{
   cout << skInts.top() << endl;
   skInts.pop();
}
```

但是也必须遵循每次只访问栈顶元素的规则。

如果想改变被配置的容器，例如想用 list 取代 deque 作为底层的容器，可用如下的代码：

```
Stack<int, list<int> > skInts;
```

这样就改变了底层容器。

上面技巧中在<int> 和最后的>之间有一个空格，否则的话，编译器会优先把它当做>>操作符，编译时会引起错误。

假设有这样一个车站，由于被错误地设计成了栈式结构，所有火车只能先进后出。下面的程序将模拟火车进出站的情形，如示例代码 21.4 所示。

示例代码 21.4

```
#include <cstdlib>
#include <iostream>
#include <stack>                                    // 包含 stack
using namespace std;                                // 使用名称空间 std
int main(int argc, char *argv[])                    // 主函数
{
    cout<<"——模拟火车进出站——"<<endl;                 // 输出提示信息
    cout<<endl;

    stack<int> station;                             // 定义火车站对象

    unsigned int trainId = 0;                       // 火车编号
    cout<<"请输入进站火车编号，输入 0 结束："<<endl; // 输出提示信息
    cin>>trainId;                                   // 输出火车编号
    while( trainId != 0 )                           // 输入 0 结束
    {
        station.push( trainId );                    // 保存入站火车
        cin>>trainId;                               // 继续输入
    }
    cout<<endl;

    cout<<"出站火车依次为："<<endl;
    while( !station.empty() )                       // 只要车站不空
    {
        cout<<station.top()<<endl;                  // 输出要出站的火车编号
        station.pop();                              // 出站
    }

    cout<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
```

```
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 21.14 所示。

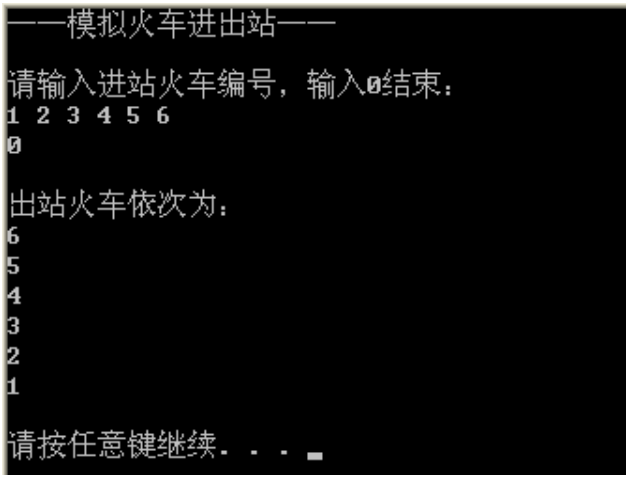


图 21.14　模拟火车进出站程序结果

上述代码正是利用了 stack 容器的先入后出特性，模拟了火车进出站的情形。

21.6 小结

本章主要讨论了各种序列式容器的定义和使用，主要讲解如何使用 vector（向量）、list（双向链表）、deque（双端队列）和 stack（栈）。在讲解各种序列式容器的时候，对各容器的概念都进行了详细的介绍，并结合具体的例子展示了如何使用各种容器。

◆ ◆ ◆

第 22 章　关联式容器

本章包括

- ◆ 关联式容器的存储结构——红黑树
- ◆ map（映射）的定义和使用
- ◆ set（集合）的定义和使用
- ◆ 其他关联式容器

序列式容器加入元素时不立即进行排序，而只是将元素放置在指定的位置上，关联式容器则在元素插入容器之后立即进行排序。在存储结构上，关联式容器与序列式容器也不相同。序列式容器的结构是线性的，而关联式容器则是以二叉树的形式表示的，并且树中的每个节点分为键值和实值两部分。每次插入元素时，关联式容器会根据键值自动排序。关联式容器主要包括集合和映射。

22.1　关联式容器的存储结构

在内部实现上，关联式容器一般采用平衡二叉树。之所以采用这种结构，主要是因为平衡二叉树搜索元素时效率比较高。另外，某些版本的关联式容器也采用哈希表（hash table）作为其存储结构，比如 SGI STL 中有 hash_map，hash_set 等利用哈希表实现的映射和集合。本节主要讲述平衡二叉树。

平衡二叉树也有多种类型，包括 AVL-Tree，RB-Tree 和 AA-Tree，但在 STL 中广泛应用的就是 RB-Tree（红黑树）。

22.1.1　二叉树的概念

平衡二叉树首先是一棵二叉树。所谓二叉树是如图 22.1 所示的一种数据结构的定义。

树是由节点和边组成的。节点之间存在父子关系。连接父节点与子节点的线称做边。边有方向性，由父节点指向子节点。也就是说只能通过父节点访问其子节点，而不是反过来。如果某个节点没有子节点，则称该节点为叶节点。树如果不为空，则只有一个根节点。树中只有根节点没有父节点，其他节点都有且仅有一个父节点。如果限定树中每个父节点最多只能拥有两个子节点，则该树称为二叉树。二叉树中的子节点有左右之分，同一父节点的两个子节点互为兄弟节点，如图 22.1 所示二叉树中各节点的性质如表 22.1 所示。

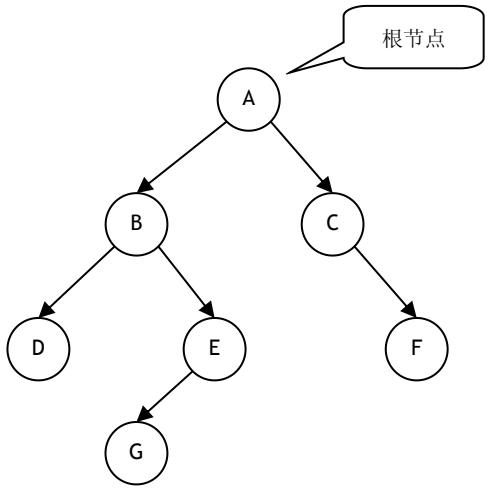


图 22.1　二叉树

表 22.1　二叉树各节点性质

节点	性质
A	根节点
B	A 的左子节点，D 和 E 的父节点
C	A 的右子节点，F 的父节点
D	叶节点，B 的左子节点
E	B 的右子节点，G 的父节点
F	叶节点，C 的右子节点
G	叶节点，E 的左子节点

另外，也可以用递归的方式定义二叉树：如果一棵二叉树不为空，那么该二叉树就是由一个根节点和两棵左右子树构成的，并且左右子树也都是二叉树，子树可以为空。

经典的二叉树，只有从父节点指向子节点的边。然而为了能够快速有效地访问父节点，也可以设立从子节点指向父节点的边。

二叉树中有路径的概念。如果沿着边的方向可以从一个节点到达另外一个节点，则所经过的节点和边称做路径，经过的边数称做路径长度。根节点到任何一个节点都存在唯一一条路径。根节点到某个节点的路径长度，称做该节点的深度；某节点到其最深子节点的路径长度，称做该节点的高度；从根节点到最深叶节点的高度，称做该树的高度或深度。

22.1.2　二叉树的表示

在程序中要表示一棵二叉树，首先要定义树中节点的结构。根据其定义，该节点至少包括三个部分，即数据、左子树索引和右子树索引。在 C/C++语言中，左右子树的索引通常用指针来表示。另外，为了存储不同类型的数据，还应当将该节点设计成类模板。

```
template <typename T>
struct NODE                                  // 节点模板
{
    T m_data;                                // 数据
    NODE *m_pLeft;                           // 左子树指针
    NODE *m_pRight;                          // 右子树指针

    NODE( T value) : m_data( value ),        // 节点构造函数
        m_pLeft( 0 ), m_pRight( 0 )
    {
    }
};
```

有了节点 NODE 的定义后，在程序中构建树就很简单了，只要按照父子关系依次将各个节点串联起来即可。例如要构建二叉树，则可以如下编程：

```
NODE nodeA('A' );                            // 构建节点 A 到 F
NODE nodeB('B' );
NODE nodeC('C' );
NODE nodeD('D' );
NODE nodeE('E' );
NODE nodeF('F' );
NODE nodeG('G' );

nodeA->m_pLeft = &nodeB;                     // 构建节点间的关系
nodeA->m_pRight = &nodeC;

nodeB->m_pLeft = &noddeD;
nodeB->m_pRight = &nodeE;

nodeC->m_pRight = &nodeF;
nodeE->m_pLeft = &nodeG;
```

22.1.3 二叉树的遍历方法

所谓二叉树的遍历，就是依次访问二叉树中的每个节点。按照访问策略，可以分为：

- 广度优先遍历。
- 深度优先遍历。

广度优先遍历指的是按照节点深度由浅到深的次序进行遍历。也就是说，在某一深度的节点访问完毕之前，是不能访问下一深度的子节点的。深度优先遍历指的是优先遍历某棵子树，一般来讲是先遍历左子树，再遍历右子树。按照访问子树根节点的时机，可以分为前序遍历、中序遍历和后序遍历。深度优先遍历过程可以写成如下的递归函数：

```
typedef void (*PFUN) ( NODE *node );          // 访问二叉树节点的函数指针
void pre_visit( NODE *root, PFUN fun)         // 前序遍历函数
{
    fun( root );                              // 访问当前子树的根节点
```

```
    visit( root->m_pLeft, fun );                    // 访问左子树
    visit( root->m_pRight, fun );                   // 访问右子树
}

void mid_visit(NODE *root, PFUN fun)                // 中序遍历函数
{
    visit( root->m_pLeft, fun );                    // 访问左子树
    fun( root );                                    // 访问当前子树的根节点
    visit( root->m_pRight, fun );                   // 访问右子树
}

void post_visit(NODE *root, PFUN fun)               // 后序遍历函数
{
    visit( root->m_pLeft, fun );                    // 访问左子树
    visit( root->m_pRight, fun );                   // 访问当前子树的根节点
    fun( root );                                    // 访问右子树
}

void OutputNode( NODE *pNode )                      // 访问节点的函数
{
    cout<<pNode->m_data<<endl;                      // 输出节点数据
}

nt main(int argc, char *argv[])                     // 主函数
{
    NODE *pRoot = NULL;                             // 二叉树根节点
    ……                                              // 构建以 pRoot 为根的二叉树
    pre_visit( pRoot );                             // 前序遍历以 pRoot 为根的二叉树
    mid_visit( pRoot );                             // 中序遍历以 pRoot 为根的二叉树
    post_visit( pRoot );                            // 后序遍历以 pRoot 为根的二叉树
    system("PAUSE");                                // 暂停程序
    return EXIT_SUCCESS;                            // 主函数返回
}
```

22.1.4 二叉搜索树

顾名思义，二叉搜索树是用于搜索其中元素的二叉树。为了便于搜索，二叉搜索树给出了如下的限制：任何节点的元素一定大于其左子树所有节点的元素，并且小于其右子树所有节点的元素。根据上述限制，在二叉搜索树中查找最小元素和最大元素非常容易，只要从根节点开始，查找到最左的节点，其中的元素即最小元素，从根节点开始，查找到最右的节点，其中的元素即最大元素。如图 **22.2** 所示就是一棵典型的二叉搜索树，其中包含的元素是整数 **1** 到 **9**。

为了保证二叉搜索树始终满足其定义，在树中插入或删除元素时要对树进行重新排序，即要保证根节点大于左子树的所有节点，而小于右子树的所有节点。

根据二叉搜索树的定义，查找一个元素可以采用如下的递归算法：比较目标元素与根元素的关系，如果相等，就直接返回根元素的指针（或者其他指示根元素位置的变量、对象等），如果目标元素小于根元素，则在左子树中查找，否则就在右子树中查找。例如：

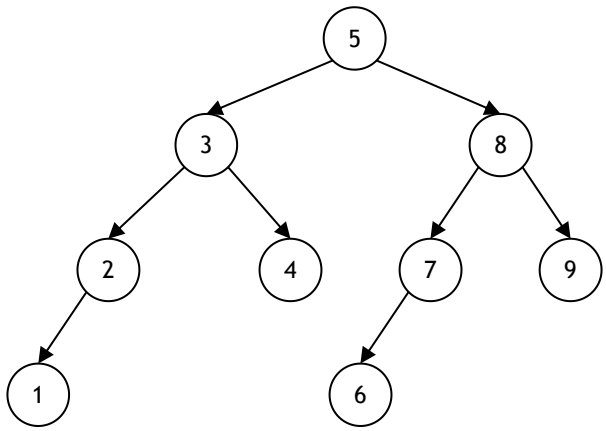


图 22.2 二叉搜索树

```
template< typename T >                    // 函数模板
NODE * find( NODE *root, T value )        // 在以 root 为根的二叉搜索树中查找 value
{
    if( root->m_data == value )           // 如果是当前根节点
    {
        return root;                      // 返回指向根节点的指针
    }
    if( value < root->m_data )            // 如果小于当前根节点
    {
        return find( root->m_pLeft, value );    // 在左子树中查找
    }
    if( value > root->m_data )                  // 如果大于当前根节点
    {
        return find( root->m_pRight, value );   // 在右子树中查找
    }
}
```

22.1.5 平衡二叉树

平衡二叉树是平衡二叉搜索树的缩写。二叉搜索树虽然在查找元素方面比一般二叉树效率要高，但是如果搜索树（或者某个节点）的两棵子树高度差别较大，那么其搜索效率就会大打折扣。例如如图 22.3 所示的二叉搜索树的搜索效率就很低，等同于数组或向量的搜索效率。

如果没有采取特殊措施，并且在插入元素时持续插入越来越大或者越来越小的元素，就会导致如图 22.3 所示的情况，即一边子树的深度极深，而另一边极浅。

为了提高二叉搜索树的查找效率，可以对其进行限定。限定的方法是让根节点（包括任一子树的根节点）的两棵子树的深度差尽量小，也就是说二叉搜索树要尽量平衡，这样的树就称做平衡二叉树。根据不同的平衡策略，可以分为 AVL-Tree、RB-Tree（红黑树）和 AA-Tree。虽然平衡二叉树在构建时要比普通的二叉搜索树复杂，但其搜索效率却大大提高了。据统计，大致可以节省 1/4 的时间。在 STL 中常用的平衡二叉树是 RB-Tree，即红黑树。构建 RB-Tree（红黑树）必须满足如下的条件：

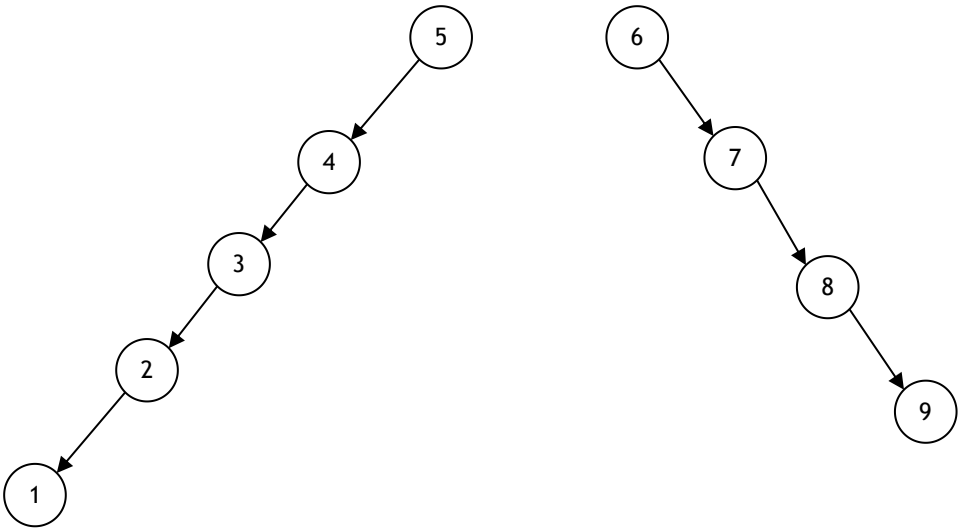


图 22.3　两棵搜索效率低的二叉搜索树

- 红黑树首先是一棵二叉搜索树。
- 每个节点不是红色就是黑色。
- 根节点是黑色。
- 如果某个节点是红色，则其子节点为黑色。
- 任一节点到叶节点（包括叶节点）的任何路径中包含的黑色节点数目必须相同。

如图 22.4 所示的树就是一棵典型的红黑树（浅色底纹为红，深色底纹为黑）。

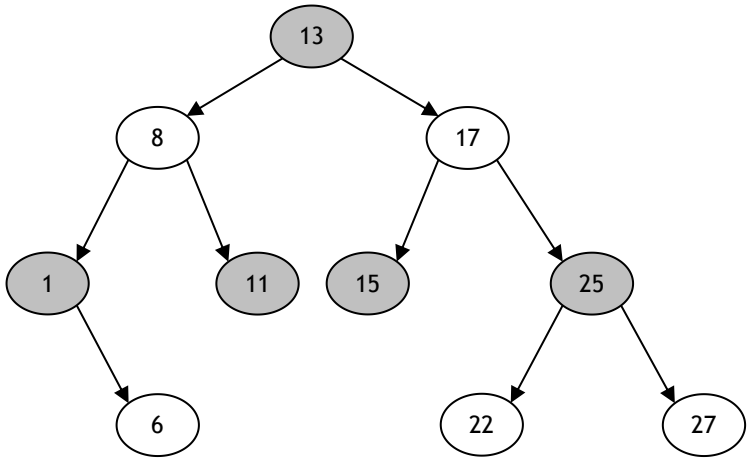


图 22.4　一棵红黑树

类似二叉搜索树，在红黑树中插入或者删除元素时，如果有必要，则应当对树的结构进行调整，以满足红黑树的限制条件。有关调整的方法已经超出了本书的范围，这里不再继续讨论，请读者参考 STL 源码或者相关的书籍。

22.1.6 关联式容器的元素

为了能够进行排序，存储在关联式容器中的元素必须能够比较大小。然而，有的数据可以比较大小，如整数、字符等；也有的数据不能比较大小，如开发者定义的类（重载了关系运算符的除外）。为了能够处理所有类型的数据，关联式容器中的元素一般分为两部分，即键值和实值。其中键值可以比较大小，用于容器中的排序操作，而实值则存放真正感兴趣的数据。因此，关联式容器的元素一般如图 22.5 所示。

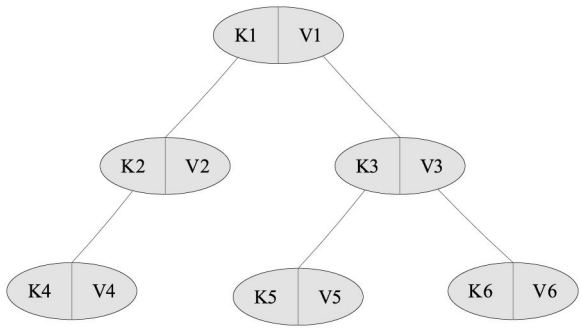


图 22.5 关联式容器的元素

键值和实值合称值对，即 pair。值对一般作为二叉树节点的数据部分而存在。一个具有值对的二叉树节点可如下定义：

```
template <typename KeyType, typename ValueType>
struct pair                                          // 值对
{
    KeyType first;                                   // 键值
    ValueType second;                                // 实值
};

tempalet<typename KeyType, typename ValueType>
struct NODE                                          // 二叉树节点
{
    pair<KeyType, ValueType> m_data;                 // 节点数据
    NODE * m_pLeft;                                  // 指向左子树的指针
    NODE * m_pRight;                                 // 指向右子树的指针
    ……
};
```

上述值对和节点的定义都只是示意，真正的 STL 实现与此并不相同，请读者参考 STL 的实现以及相关的文档。

键值和实值可以不同，也可以相同。STL 有两个主要的关联式容器，即映射 map 和集合 set。在集合中，每个元素的键值和实值是相同的，所以没有必要区分。而在映射中，键值和实值一般是不同的，不仅值可以不同，而且其类型也可以不同。

22.2 映射 map

所谓映射，就是数据之间的对应关系。根据这个对应关系，可以从一个数据迅速查找到另外一个数据。前面的数据即所谓的键值，而后面的数据即所谓的实值。在 STL 中可以用 map 这个容器来存储这些有对应关系的数据，并提供相应的组织和查找等功能。使用时应当包含头文件<map>。map 容器的典型结构如图 22.6 所示。

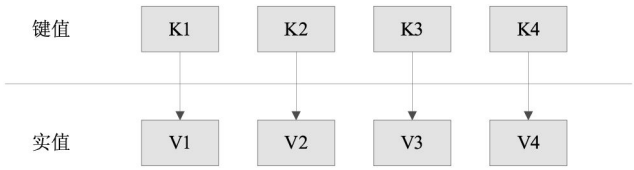


图 22.6 map 容器

容器 map 的行为类似字典，可以根据某个索引值迅速查到目标值。实际上，在某些编程语言中，映射这种容器也被称做字典，即 dictionary。

22.2.1 定义并构造 map

STL 中 map 的声明如下：

```
template <typename _Key, typename _Tp,
        typename _Compare = less<_Key>,
        typename _Alloc = allocator<pair<const _Key, _Tp> > >
                                                    // 注意“>”之间的空格
class map
{
    ……
public:
    typedef _Key                key_type;           // 键值类型
    typedef _Tp                 mapped_type;        // 实值类型
    typedef pair<const _Key, _Tp>  value_type;      // 元素类型
    typedef _Compare            key_compare;        // 键值比较函数对象类型
    ……
};
```

注意第四个模板参数默认值 allocator 后面三个尖括号“>”之间的空格。这些空格是必需的，否则会被当成输入运算符“>>”，从而导致编译错误。

在 map 容器的模板参数列表中，_Key 和_Tp 是两个主要的参数，决定了容器中所能容纳的元素的类型。_Key 是元素键值的类型，_Tp 是元素实值的类型。在定义一个 map 容器的实例时，一般只要有这两个参数即可。例如可以用一个 map 容器来存储学生信息，键值可以是学生的学号，而实值则是表示学生的相关信息的结构体，如下所示：

```
struct Student                              // 表示学生信息的结构体
{
```

```
    string number;                          // 学号
    string name;                            // 姓名
    int grade;                              // 年级
    int cls;                                // 班级
};
map<string, Student> StudentBook;           // 学生名册，模板参数 string 是学号的类型
```

在上述例子中，利用 map 容器建立了一个学生名册，分别用学号和学生信息结构体作为其参数，将来可以通过学号迅速查找到其对应的信息。

除了键值和实值类型外，map 容器还有两个模板参数，分别是_Compare 和_Alloc。前者是用来比较键值大小的，而后者则是一个内存配置器。这两个参数比较受限，只能使用某些类型。

_Compare 是一个函数对象（STL 六大组成部分之一，也称做仿函数）类，实际上是一个重载了括号运算符“()”的类。一般来讲，开发者可以不必自己定义，只要使用 STL 中已有的类即可。例如 map 容器使用的默认_Compare 就是 less<_Key>。less 是一个模板类，可以用键值类型进行实例化。

_Alloc 是一个配置器，用来配置容器中元素的内存。一般用户也不必自己定义，可以直接使用 STL 中的已有配置器，例如 map 中配置器的默认值就是 allocator。值得注意的是，在关联式容器中直接存储的元素是 pair，即键值—实值对。pair 也是一个模板类，可以用键值和实值类型进行实例化。实际上，除了默认构造函数外，map 容器还有如下三个构造函数：

```
// 接受比较函数对象和分配器的构造函数
explicit map(const _Compare& __comp, const allocator_type& __a = allocator_type());
// 复制构造函数
map(const map& __x);
// 接受迭代器区间的构造函数
template <typename _InputIterator>
    map(_InputIterator __first, _InputIterator __last);
```

第一个构造函数显示接受一个函数对象，因为默认情况下 map 内部是按键值的升序进行排序的，通过函数对象可以指定排序规则，这个会在后面有关函数对象的内容中进行介绍。allocator 可以是一个已经存在的 map 容器的分配器，可以通过 get_allocator 方法取得。第二个构造函数接受另一个 map 作为参数进行初始化。第三个构造函数接受两个迭代器所表示的迭代器区间作为参数。

allocator 是 STL 的空间分配器，大多数时候对于使用者是透明的，对于读者的使用不会有任何影响。

例如：

```
map<int, int> m1( greater<int> );               // 从大到小排序的映射容器
map<int, int> m2( m1 );                         // 复制构造
……
map<int, int> m3( m2.begin(), m2.end() );       // 利用迭代器区间构造
```

22.2.2 map 容器的 pair 结构

正如前文所述，关联式容器中存储的数据是一个键值—实值对的 pair 结构，map 容器也是如

此。并且在使用 map 容器时，也会有大量地方用到 pair 结构。因此，在进一步讲述 map 的内容之前，有必要详细了解一下 pair 结构。pair 结构包含在标准头文件 utility 中，正常情况下，map 的头文件中就已经包含了头文件 utility，其实质就是一个键值—实值对。其详细定义如下：

```
template<class _T1, class _T2>
struct pair                                    // pair 结构
{
    typedef _T1 first_type;                    // 键值数据类型定义
    typedef _T2 second_type;                   // 实值数据类型定义

    _T1 first;                                 // 键值
    _T2 second;                                // 实值

    pair(): first(), second() { }              // 默认构造函数
    pair(const _T1& __a, const _T2& __b)       // 接受键值和实值的构造函数
     : first(__a), second(__b) { }

    template<class _U1, class _U2>             // 复制构造函数
      pair(const pair<_U1, _U2>& __p)
    : first(__p.first), second(__p.second) { }
};
```

pair 是一个结构体。虽然在 C++中用 struct 定义的结构体同用 class 定义的类基本相同，但还是略有区别的。默认情况下，用 class 定义的类，其成员的访问属性是 private，外界不能访问，而 struct 结构的默认访问属性是 public，外界可直接访问。例如：

```
pair<int, int> prInts;
prInts.first = 10;
prInts.second = 5;
cout << prInts.first << " " << prInts.second << endl;      // 输出 10 5

pair<int, int> prInts2(3,4);
cout << prInts2.first << " " << prInts2.second << endl;    // 输出 3 4
```

除了 pair 结构声明中的成员之外，STL 中还定义了一些用于 pair 间比较的全局关系运算符，以及一个生成 pair 结构体的函数，如表 22.2 所示。

表 22.2 pair 的重载运算符函数及作用

运算符 / 函数	作用
==	比较两个 pair 是否相等
<	第一个 pair 是否小于第二个 pair
!=	判断两个 pair 是否不相等
>	第二个 pair 是否大于第二个 pair
<=	第一个 pair 是否小于等于第二个 pair
>=	第二个 pair 是否小于等于第一个 pair
make_pair	接受两个参数，生成一个 pair

相关运算符、函数的定义如下：

```
template<class _T1, class _T2>
  inline bool
  operator==(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
    // ==
  { return __x.first == __y.first && __x.second == __y.second; }
template<class _T1, class _T2>
  inline bool
  operator<(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)     // <
  { return __x.first < __y.first
     || (!(__y.first < __x.first) && __x.second < __y.second); }

template<class _T1, class _T2>
  inline bool
  operator!=(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
    // !=
  { return !(__x == __y); }

template<class _T1, class _T2>
  inline bool
  operator>(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)     // >
  { return __y < __x; }

template<class _T1, class _T2>
  inline bool
  operator<=(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
    // <=
  { return !(__y < __x); }

template<class _T1, class _T2>
  inline bool
  operator>=(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
    // >=
{ return !(__x < __y); }

template<class _T1, class _T2>
  inline pair<_T1, _T2>
  make_pair(_T1 __x, _T2 __y) { return pair<_T1, _T2>(__x, __y); } // make_pair
```

22.2.3 使用 insert 插入数据

在 map 容器中插入数据有两种方法，一种是调用成员函数 insert，一种是利用下标运算符“[ ]”。本节首先讲述第一种方法。成员函数 insert 有如下三个重载版本：

```
pair<iterator, bool> insert( const pair<KEY_TYPE,VALUE_TYPE> &val );
iterator insert( iterator pos, const pair<KEY_TYPE,VALUE_TYPE> &val );
void insert( input_iterator start, input_iterator end );
```

第一个版本是在容器中插入一个 pair，该 pair 的模板参数就是该 map 容器的键值类型和实值类型。返回值也是一个 pair，不过其模板参数是迭代器和 bool。在这个返回值中，迭代器表示的

是插入的数据在容器中的位置，bool 值表示的是是否插入成功。

如果参数中键值已经存在于 map 容器中，则 insert 函数不能成功插入数据，而返回值的迭代器成员指向拥有该键值的元素。

例如：

```
map<int, int> m;                          // 定义映射：整数—整数
typedef pair<int, int> VALUE;             // 定义pair类型,等同于map::mapped_type
typedef map<int, int>::iterator MAP_ITER;       // 定义映射的迭代器类型
typedef pair< MAP_ITER, bool> RESULT_PAIR;      // 定义插入 pair 的结果类型
m.insert( VALUE( 0, 10 ) );               // 插入 (0, 10)
m.insert( VALUE( 1, 11 ) );               // 插入 (1, 11)
RESULT_PAIR res = m.insert( VALUE( 0, 12 ) );  // 重复插入键值 0
cout<< res.second <<endl;                 // 显示是否插入成功，结果输出 0
cout<< res.first->first <<endl;           // 输出结果迭代器所指向元素的键值，结果是 0
```

第二个版本也是在容器中插入一个 pair，不过该版本给出了一个表示目标位置的迭代器（第一个参数）。其目的是在目标位置上插入这个 pair。该版本的返回值也是一个迭代器，表示这个 pair 实际插入的位置。同第一个版本一样，如果容器中已经存在与目标 pair 键值相同的元素，则插入不成功，并且返回的迭代器指向已存在元素。

```
// 接上例
MAP_ITER iter = m.insert( m.begin(), VALUE( 2, 12 ) );   // 在容器头处加入(2, 12)
cout<< iter->first << " " << iter->second <<endl;       // 输出 2 12
iter = m.insert( m.begin(), VALUE( 2, 13 ) );      // 重复加入键值为 2 的 pair
cout<< iter->first << " " << iter->second <<endl;
                                          // 输出 2 12（键值为 2 的已存在元素）
```

第三个版本是将一个由输入迭代器表示的区间插入到容器中。该迭代器区间是左闭右开的，即第一个迭代器指向区间的第一个元素，而第二个迭代器指向区间最后一个元素之后的存储单元。

源区间所在的容器可以与目标容器不同质，即键值和实值的类型可以不同，只要两个容器的键值和键值、实值和实值之间存在转换关系即可。

```
// 接上例，映射容器 m 中已存在(0, 10), (1, 11), (2, 12)
map<int, int> m1;                         // 定义容器 m1
m1.insert( m.begin(), m.end() );          // 将容器 m 的内容插入到 m1 中
for(iter = m1.begin(); iter != m1.end(); iter++ )      // 遍历容器 m1
{
    cout<< iter->fist << ' ' << iter->second <<endl;
                                          // 输出容器 m1 的内容，结果同输出容器 m
}
```

在本节的例子中，出于易于理解的目的，为 map 中的元素 pair 定义了一种类型 VALUE。实际上也可以直接使用 map 这个类模板中的类型定义：

```
typedef pair<const _Key, _Tp> value_type;
```

不过在使用前要加上 map 这个类域，例如 map::value_type(1, 2)。

22.2.4 使用下标运算符[]插入数据

map 容器重载了下标运算符“[ ]”。该运算符接受一个键值类型（key_type）的数据，并返回该键值所对应实值的引用（mapped_type &）。如果容器中不存在这样的键值，则生成一个新的元素。其键值就是传入的下标，而实值则是相应实值类型的默认值。

所谓实值类型的默认值，就是调用实值类型的默认构造函数，即 mapped_type()。对于 C++内置的各种数据类型，如 int，char，double 等，其默认值就是 0。

例如：

```
map<int, int> m;                                    // 定义映射：整数—整数
typedef map<int, int>::iterator MAP_ITER;           // 定义映射的迭代器类型

m[0] = 1;                                           // 插入数据（0，1）
m[1] = 2;                                           // 插入数据（1，2）
cout<< m[5] <<endl;                                 // 插入数据（5，0）

MAP_ITER iter = m.begin();                          // 定义迭代器，指向容器头
for( ; iter != m.end(); iter++ )                    // 遍历映射容器
{
    cout<< iter->first <<' '<< iter->second <<endl;    // 输出容器内容
}
```

在上例的第 6 行，程序试图输出键值为 5 的实值。但是由于此时容器中并不存在这样的元素，所以容器会调用实值类型的默认构造函数（即 int()，结果是 0），并结合键值 5 生成一个新的元素（5，0）。

22.2.5 查找数据

由于下标运算符“[]”返回的是实值的引用，即 mapped_type &，所以也可以使用这个运算符来查找数据。但是在使用过程中应当注意该运算符的副作用，即如果目标元素不存在，则会生成一个。

```
map<int, int> m;                                    // 定义映射：整数—整数
……                                                  // 容器操作
cout<< m[1] <<endl;                                 // 查找并输出键值为 1 的实值
cout<< m[2] <<endl;                                 // 查找并输出键值为 2 的实值
```

有一种方法不会带来副作用，那就是利用成员函数 find()来查找数据。find()函数接受一个键值类型的数据，返回一个指向拥有该键值元素的迭代器。如果找不到对应的元素，则返回一个指向容器尾（endl()）的迭代器。该函数有如下两个重载版本：

```
iterator        find(const key_type& __x);          // 返回迭代器的 find 函数
const_iterator  find(const key_type& __x) const;    // 返回常量迭代器 find 函数
```

这两个迭代器接受的参数是一样的，区别在于返回值不同。一个是一般的迭代器，可以通过该

迭代器修改元素的实值。另外一个是常量迭代器，只能访问，不能修改元素的实值。

在关联式容器中，键值是不允许修改的，所以通过关联式容器的迭代器，只能修改实值。

例如：

```
map<int, int> m;                                    // 定义映射：整数—整数
    m[0] = 10;                                      // 插入数据（0, 10）
    m[1] = 11;                                      // 插入数据(1, 11)
    m[2] = 12;                                      // 插入数据(2, 12 )

typedef map<int, int>::iterator MAP_ITER;  // 定义一般迭代器类型
ypedef map<int, int>::const_iterator CONST_ITER;   // 定义常量迭代器类型
MAP_ITER iter = m.find( 5 );                        // 查找键值为 5 的元素
cout<< iter == m.end() <<endl;                      // 输出 1，表示没有找到
iter = m.find( 1 );                                 // 查找键值为 1 的元素
// iter->first = 5;                                 // 尝试修改键值，编译失败
ter->second = 21;                                   // 修改实值
cout<< iter->second;                                // 输出 21
const map<int, int> &m1 = m;                        // 定义容器 m 的常引用
CONST_ITER cIter = m1.find( 1 );                    // 查找键值为 1 的元素，返回常量迭代器
// cIter->second = 31;                              // 修改实值
cout<< cIter->second;                               // 输出实值，结果仍为 21
```

除了利用下标运算符和成员函数 find()之外，还可以利用成员函数 lower_bound()和 upper_bound()进行查找，这两个函数的声明如下：

```
iterator        lower_bound(const key_type& __x);
const_iterator  lower_bound(const key_type& __x) const;
iterator        upper_bound(const key_type& __x);
const_iterator  upper_bound(const key_type& __x) const;
```

lower_bound()接受一个键值类型的参数。如果容器中某个元素的键值等于该参数，那么函数就返回指向该元素的迭代器。如果不存在这样的元素，则返回一个迭代器，指向第一个键值比参数大的元素。也就是说，lower_bound()返回的迭代器，指向键值大于等于（>=）参数的元素，如果所有元素的键值都比参数小，那么函数就返回容器尾，即等于 end()。

upper_bound()正好与 lower_bound()相反。该函数返回的也是一个迭代器，指向第一个键值比参数大的元素，或者键值等于参数的元素。upper_bound()返回的迭代器，指向键值大于参数的元素。如果所有元素的键值都比参数大，那么函数就返回容器尾，即等于 end()。两个函数的作用如图 22.7 所示。

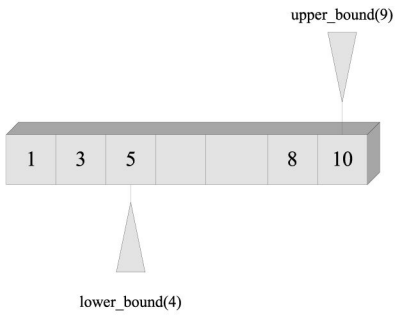


图 22.7 lower_bound()和 upper_bound()的作用

例如：

```
map<int, int> m;                                    // 定义映射容器：整数—整数
m[1] = 1;                                           // 插入数据
m[3] = 3;
m[5] = 5;
m[8] = 8;
m[10] = 10;
map<int, int>::iteator iter = m.end();              // 定义并初始化容器迭代器
iter = m.lower_bound( 4 );                          // 求键值为 4 的迭代器的下限
cout<< iter->first <<' '<< iter->second <<endl;     // 输出容器内容：(5, * )
iter = m.upper_bound( 9 );                          // 求键值为 9 的迭代器的上限
cout<< iter->first <<' '<< iter->second <<endl;     // 输出容器内容：(10, *)
```

22.2.6 遍历 map

遍历一个 map 容器同遍历其他 STL 容器一样，可以利用类模板 map 中定义的四种迭代器类型，即 iterator，const_iterator，reverse_iterator 和 const_reverse_iterator。对于 map 容器有个特殊的方法，可以用来遍历，即利用下标运算符“[ ]”。但前提是键值必须有规律，可以应用在循环中，如键值是依次递增的整数或者有规律的字符串。例如：

```
map<int, int> m;                                    // 定义映射容器：整数—整数
m[1] = 1;                                           // 插入数据
m[3] = 3;
m[5] = 5;
m[7] = 8;
m[9] = 9;
……
for( int i=1; i<=9; i+=2 )                          // 每次键值递增 2，遍历容器
{
    cout<< m[ i ] <<endl;                           // 输出容器内容
}
```

22.2.7 删除数据

与其他 STL 容器一样，在 map 容器中如果要删除全部数据，可以调用成员函数 clear()；如果只是删除其中部分数据，则可以调用函数 erase()。在 map 容器中删除数据需要调用成员函数 erase()。erase()有三个重载版本，分别用于删除用键值表示的元素、用迭代器指定的元素，以及用迭代器区间表示的容器区间，如下所示：

```
size_type   erase(const key_type& __x);
void    erase(iterator __position);
void    erase(iterator __first, iterator __last);
```

第一个版本删除的是容器中所有键值等于参数的元素。当然，对于 map 容器来讲，被删除元素的个数不会超过 1 个。如果容器元素的键值都不等于参数，那么该函数就不删除任何元素，返回 0。第二个版本删除的是参数迭代器所指的元素。第三个版本删除的则是迭代器区间中的所有元素。

例如：

```
map<int, int> m;                              // 定义映射容器：整数—整数
m[1] = 1;                                     // 插入数据
m[3] = 3;
m[5] = 5;
m[7] = 8;
m[9] = 9;
map<int, int>::iterator iter1, iter2;         // 定义两个迭代器
iter1 = m.find( 5 );                          // 查找键值为 5 的元素
m.erase( iter1 );                             // 删除迭代器所指元素
iter1 = m.find( 3 );                          // 查找键值为 3 的元素
iter2 = m.find( 7 );                          // 查找键值为 7 的元素
m.erase( iter1, iter2 );                      // 删除迭代器区间，即删除迭代器是 3 的元素
m.erase( 9 );                                 // 删除键值为 9 的元素，结果为（1，1）
```

22.2.8 其他操作

map 还有一些操作，例如判断容器是否为空、返回比较键值和元素的函数对象等，如表 22.3 所示。

表 22.3 map 的其他操作

成员函数	参数	操作
empty	无	如果 map 为空，返回真(true)，否则返回假(false)
size	无	返回 map 中元素的个数
max_size	无	返回 map 能够保存的最大元素个数
get_allocato r	无	返回 map 的配置器
swap	map &obj	交换两个 map 中的元素
key_comp	无	返回比较键值的函数对象
value_comp	无	返回比较元素的函数对象
equal_range	const key_type &key	返回指向键值为 key 元素的迭代器区间
count	const key_type &key	返回具有指定键值元素的个数

上表中有一些成员函数是比较常见的，在序列式容器中也经常用到，如 empty，size，max_size 等，只有后面四个比较特殊。但对于 map 来讲，equal_range 和 count 也没有什么意义。因为 map 中键值不能重复，所以 equal_range 的效果与 find 的效果基本一致；count 的值要么是 1，要么是 0。最特殊的是 key_comp 和 value_comp。

成员函数 key_comp 返回的是比较键值的函数对象，其类型就是在实例化 map 容器时给定的第三个模板参数_Compare。其默认值是一个用键值类型实例化的模板类 less<key>。less 是 STL 中已经定义的函数对象类模板。还有一些类似的其他类模板，如 less_equal，greater，greater_euqal 等。

在类模板 map 中，类型_Compare 通过类型定义关键字 typedef 定义为 key_compare，所以使用 key_comp 成员函数时，可以将其返回值定义为 key_compare 类型的变量。

成员函数 value_comp 返回的是比较 map 容器元素的函数对象，其实是类模板 map 中的内嵌类型 value_compare。前面已经提到过，关联式容器的元素就是一个 pair，因此比较元素就是比较 map 的键值与实值对。只不过，由于在实例化 map 时没有机会定义实值类型的比较对像，所以比较 map 的元素就是比较元素的键值，实际调用的还是_Compare。

例如：

```
struct Student                                          // 表示学生信息的结构体
{
    string number;                                      // 学号
    string name;                                        // 姓名
    int grade;                                          // 年级
    int cls;                                            // 班级
    Student( string id, string n, int g, int c )    // 构造函数
        : number( id ), name( n ), grade( g ), cls( c )
    { }
};

map<string, Student> studentBook;       // 学生名册，模板参数 string 是学号的类型
studentBook["1001" ] = Student("1001", "张三", 1,2 );  // 加入学生数据
studentBook["1002" ] = Student("1002", "李四", 1,2 );
……
typedef map<string, Student>::key_compare  KEY_CMP; // 定义键值比较函数对象类型
typedef map<string, Student>::value_compare VAL_CMP;// 定义元素比较函数对象类型

KEY_CMP stKCmp = studentBook.key_comp( );           // 获取键值比较函数对象
VAL_CMP stVCmP = studentBook.value_comp();          // 获取元素比较函数对象

map<string, Student>::iterator iter1, iter2;        // 定义迭代器
iter1 = studentBook.find("1001" );                  // 查找学生
iter2 = studentBook.find("1002" );

cout<< "k1 < k2" << stKCmp( iter1->first, iter2->first ) <<endl;// 比较键值
cout<< "stu1 < stu2" << stVCmp( *iter, *iter ) <<endl;          // 比较元素
```

22.2.9 使用 map 容器管理学生名册

管理学生名册时，最重要的是能够快速查找到相应的记录，利用 map 容器可以做到这一点。在 map 容器中键值可以是学生的学号，而实值则是学生的相关信息。map 容器以红黑树作为其底层实现，所以查找数据的效率比较高。

下例使用 map 容器管理学生名册，程序如示例代码 22.1 所示。

示例代码 22.1

```
////////////////////////////////////////////////////////////////////////////
```

```
// 有关学生信息的头文件 student.h 代码如下
#include <string>
#include <iostream>
using namespace std;

struct Student                                          // 表示学生信息的结构体
{
    string id;                                          // 学号
    string name;                                        // 姓名
    int grade;                                          // 年级
    int cls;                                            // 班级
    Student(){}
    Student( string id, string n, int g, int c )    // 构造函数
        : id( id ), name( n ), grade( g ), cls( c )
    { }
    friend istream & operator >> ( istream &, Student & );    // 友元输入函数
    friend ostream & operator << ( ostream &, const Student &);// 友元输出函数
};

istream &
operator >> ( istream &is, Student &stu )           // 输入函数
{
    is >> stu.id;
    if( stu.id == "-1" )                    // 如果输入学号为-1，则表示结束输入
    {
       is.setstate( ios_base::failbit );            // 设置输入流状态
       return is;                                   // 返回
    }
    is >> stu.name;                                 // 输入姓名
    is >> stu.grade;                                // 输入年级
    is >> stu.cls;                                  // 输入班级
    return is;                                      // 返回输入流
}
ostream &
operator << ( ostream &os, const Student &stu)      // 输出函数
{
    os << stu.id << ' ';                            // 输出学号
    os << stu.name << ' ';                          // 输出姓名
    os << stu.grade << ' ';                         // 输出年级
    os << stu.cls << ' ';                           // 输出班级
    return os;                                      // 返回输出流
}
////////////////////////////////////////////////////////////////////////
//主程序文件 main.cpp 代码如下

#include <cstdlib>
#include <iostream>
#include <string>
#include <map>                                  // 包含头文件 map
#include "student.h"                            // 包含学生信息头文件 student.h
using namespace std;                            // 使用名称空间 std
```

```
typedef map<string, Student> STUDENT_BOOK;          // 定义学生名册类型
typedef map<string, Student>::iterator STU_ITER;    // 定义学生名册迭代器类型
typedef map<string, Student>::const_iterator CONST_STU_ITER;
                                                    // 定义学生名册常量迭代器类型

void outputStudentBook( const STUDENT_BOOK & book ) // 输出学生名册的函数
{
    CONST_STU_ITER iter = book.begin();             // 定义迭代器，指向容器头
    while( iter != book.end() )                     // 遍历学生名册
    {
        cout<< iter->second << endl;                // 输出学生信息
        iter++;
    }
}

int main(int argc, char *argv[])                    // 主函数
{
    // ——使用 map 容器管理学生名册——
    STUDENT_BOOK stuBook;                           // 定义学生名册
    STU_ITER iter;                                  // 定义学生名册迭代器

    cout<<"——建立学生名册——"<<endl;
    cout<<"##输入-1 退出##"<<endl;
    Student stu;                                    // 定义学生信息对象

    cin>>stu;                                       // 输入学生信息
    while ( cin )                                   // 输入流状态正确（学号!=-1）
    {
        stuBook[ stu.id ] = stu;                    // 插入学生信息
        cin>>stu;                                   // 继续输入学生信息
    }
    cin.clear();                                    // 恢复输入流状态
    cout<<endl<<"——学生名册——"<<endl;
    outputStudentBook( stuBook );                   // 输出学生名册

    cout<<endl<<"——查找学生——"<<endl;
    cout<<"##请输入学号，输入-1 退出##"<<endl;
    string id;
    cin>>id;                                        // 输入要查找的学号
    while( "-1" != id )                             // 学号 != -1
    {
        iter = stuBook.find( id );                  // 查找学生
        if( iter == stuBook.end() )                 // 如果未找到
        {
            cout<<"未找到学号为"<<id<<"的记录"<<endl;
        }
        else
        {
            cout<<"学生："<< iter->second <<endl;   // 输出学生信息
        }
```

```
        cin>>id;                                          // 继续输入学号
    }

    cout<<endl<<"——删除学生记录——"<<endl;
    cout<<"##请输入学号，输入-1 退出##"<<endl;
    cin>>id;                                              // 输入要删除的学号
    while( "-1" != id )                                   // 学号 != -1
    {
        iter = stuBook.find( id );                        // 查找学生
        if( iter == stuBook.end() )                       // 如果未找到
        {
            cout<<"未找到学号为"<<id<<"的记录"<<endl;
        }
        else
        {
            stuBook.erase( id );                          // 删除学生信息
        }
        cin>>id;
    }
    cout<<endl<<"——学生名册——"<<endl;
    outputStudentBook( stuBook );                         // 输出学生名册

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

建立一个控制台工程以及相应的头文件和源代码文件，将上述示例代码复制到相应的文件中，编译并运行，结果如图 22.8 所示。

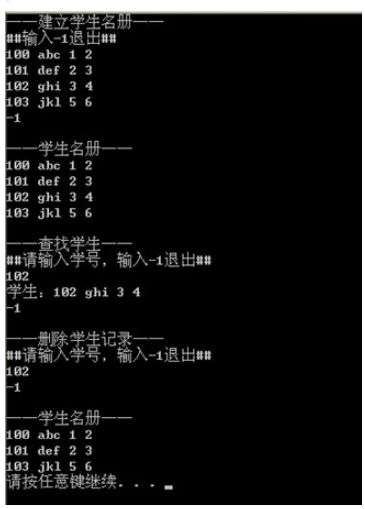


图 22.8　管理学生名册结果

在上述例子中，首先定义了一个表示学生信息的类 Student（为了方便起见，这里用的是结构体，而不是类），并重载了对该类进行输入输出的运算符。然后在主程序中使用 map 容器，并用学号作为键值、学生信息作为实值，建立了学生名册。在学生名册的管理中，分别使用了 map 容器的构造、插入、查找、删除等方法。当然，本例还可以继续扩展，用以对学生信息进行修改，读者

可以以此为题目，自己进行练习。

22.3 集合 set

所谓集合，即将数据汇集在一起的一种容器。将数据存放在集合中，其位置并不重要，重要的是如何快速取出。STL 中容器 set 可以提供这样的功能。set 以红黑树作为其底层数据结构，因此插入数据时可以自动排序。同时，在排好序的红黑树中查找数据效率也非常高。尽管为了能在红黑树中存取数据，set 中的元素也分为键值和实值，但这两者之间不存在对应关系，所以在 set 的元素中，键值和实值是相同的数据。集合示意图如图 22.9 所示。

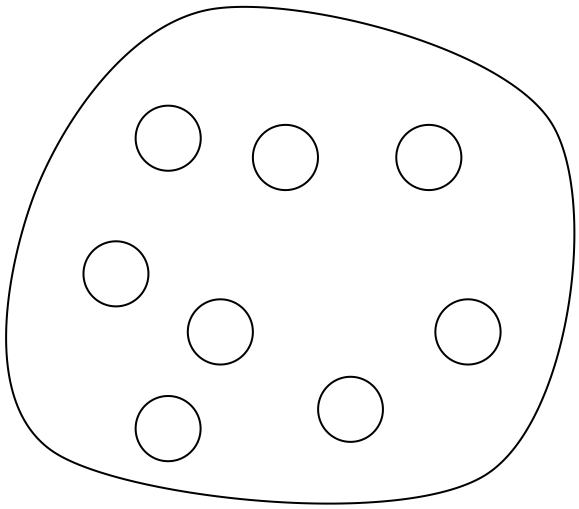


图 22.9 集合示意图

虽然键值和实值相同，但为了满足底层数据结构红黑树的要求，容器 set 不得不将其元素分为键值和实值两部分，这在一定程度上导致了内存空间的浪费。因此，选择使用 set 时，应当尽量使用那些占内存小的数据类型。

22.3.1 定义并构造 set

STL 中 set 的声明如下：

```
template<class _Key,                                    // 值类型
        class _Compare = less<_Key>,                    // 值比较函数对象类型
        class _Alloc = allocator<_Key> >                // 配置器
class set
{
    ……
public:
    typedef _Key    key_type;                           // 键值类型定义
    typedef _Key    value_type;                         // 实值类型定义
    typedef _Compare key_compare;                       // 键值比较函数对象类型定义
    typedef _Compare value_compare;                     // 实值比较函数对象类型定义
```

```
    ......
};
```

在类模板 set 的类型定义中，键值类型和实值类型定义是一样的，都用的是类模板实例化时的值类型参数。因此，在 set 中键值和实值的类型是一样的。不仅如此，set 并没有提供插入实值的操作，而只有插入键值的操作。所以，实值与键值的值也是一样的。

有关 set 的实例化和其他类型定义，同 map 非常相似，这里就不再重复了。下面通过一个例子来讲述 set 的实例化和构造。

```
struct Student                                          // 表示学生信息的结构体
{
    string id;                                          // 学号
    string name;                                        // 姓名
    int grade;                                          // 年级
    int cls;                                            // 班级

    bool operator < ( const Student & other ) const     // 重载小于运算符
    {
        return id < other.id;
    }
};
set<Student> stuBook;                                   // 实例化 set，并构造学生名册
```

本例中用 Student 这个结构体来实例化集合类模板 set，并省略了_Compare 和_Alloc 两个模板参数。值得注意的是，在本例中结构体 Student 重载了小于运算符“<”，其目的是能够利用默认的_Compare，即函数对象 less。在本例中，实例化 set 时构造的比较函数对象是 less<Student>。如果 Student 结构体没有重载小于运算符，虽然能够通过编译，但结果却是没有定义的。

键值必须能够比较大小，这点对所有以红黑树为底层结构的关联式容器都是适用的。试想，如果键值不能比较，那么容器就不能对加入其中的元素进行排序，也就无法建立红黑树。

在上例中，重载的小于运算符被声明为 const。这是必须的，因为 set 容器的迭代器都是常量迭代器，通过常量迭代器只能调用对象的 const 方法。

除了使用默认构造函数，类模板 set 还有另外四个构造函数，分别是复制构造函数、以迭代器区间为参数的构造函数，以及另外附加比较函数对象的构造函数。这里不再一一赘述，请读者参考相关的文档或者 STL 源代码。

22.3.2 set 容器的迭代器

正如前面已经提到过的，set 容器中的元素虽然也包含键值和实值，但是这两个值是相同的。因此，使用迭代器访问 set 容器的元素时，与 map 容器的迭代器有两个不同地方：

- 使用解引用运算符“*”取值，不用 first 和 second。

◆ 所有迭代器都是常量迭代器，不能用来修改所指元素。

因为键值和实值相同，所以在 set 元素中区分键值和实值就没有任何意义，因此也就不必使用 first 和 second 成员了。例如：

```
set<Student> stuBook;                              // 构建学生名册
Student st1, st2;                                  // 定义两个学生
st1.id="100";                                      // 指定学号
st2.id="102";
stuBook.insert( st1 );                                 // 将学生信息入册
stuBook.insert( st2 );
set<Student>::iterator iter = stuBook.begin(); // 指向学号最小的学生
cout<< iter->id <<endl;                            // 输出 100
```

另外，也是因为键值和实值相同，所以也不允许通过迭代器修改实值。因为修改实值，就意味着要修改键值，而修改键值则必然导致红黑树重新排序。如果程序中存在大量修改键值的操作，那么必将严重影响程序的效率。因此，set 容器禁止修改其元素的值，而这一点也可以通过其迭代器的定义看出来，如下所示：

```
typedef typename _Rep_type::const_iterator     iterator;
typedef typename _Rep_type::const_iterator     const_iterator;
typedef typename _Rep_type::const_reverse_iterator reverse_iterator;
ypedef typename _Rep_type::const_reverse_iterator  const_reverse_iterator;
```

_Rep_type 是底层数据结构红黑树的类型定义。在 STL 的实现中，红黑树也是一种类模板。当实例化一个关联式容器时，也需要实例化红黑树。_Rep_type 就是红黑树实例化之后的类型定义。

在上述定义中，形似可读写迭代器的 iterator 和 reverse_iterator，其实质都是常量迭代器。而通过常量迭代器并不能修改其所指的元素，如此就可以限制通过迭代器修改 set 容器的元素。

22.3.3 set 容器的基本操作

set 容器的基本操作包括数据的增、删、查，以及判断容器是否为空、求元素个数、交换两个容器的内容、求指向容器头尾的迭代器等，如表 22.4 所示。这些操作同 map 的操作基本一致，这里不再一一详述，请读者参考相关的文档或者 STL 源代码。

表 22.4 set 的基本操作

成员函数	参数	操作
insert	（重载）	插入数据
erase	（重载）	删除数据
find	const key_type key	查找值为 key 的元素
empty	无	如果 set 为空，返回真(true)，否则返回假(false)
size	无	返回 set 中元素的个数
max_size	无	返回 set 能够保存的最大元素个数

get_allocator	无	返回 set 的配置器
swap	set &obj	交换两个 set 中的元素
key_comp	无	返回比较键值的函数对象
value_comp	无	返回比较元素的函数对象
equal_range	const key_type &key	返回指向键值为 key 元素的迭代器区间
count	const key_type &key	返回具有指定键值元素的个数

22.3.4 使用 set 容器管理学生名册

使用 set 容器也可以管理学生名册。与 map 容器不同的是，可以直接用学生信息本身作为元素，而不需要额外的键值。本例使用 set 容器管理学生名册，程序如示例代码 22.2 所示。

示例代码 22.2

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <set>
using namespace std;

struct Student
{
    string id;
    string name;

    Student(string id, string name)
       :id(id), name(name)
        {}

    bool operator < ( const Student &other ) const
    {
        return id < other.id;
    }
};
typedef set<Student> STUBOOK;
typedef set<Student>::iterator STU_ITER;

void outputStudentBook(const STUBOOK &book)
{
    STU_ITER iter = book.begin();
    for(; iter!=book.end(); iter++)
    {
        cout<<iter->id<<' ' <<iter->name<<endl;
    }
}

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

```
    cout<<"——使用 set 管理学生名册——"<<endl;
    cout<<"——建立学生名册——"<<endl;
    Student st0("100", "abc"),
            st1("101", "def"),
            st2("102", "ghi");
    set<Student> stuBook;
    stuBook.insert( st0 );
    stuBook.insert( st1 );
    stuBook.insert( st2 );
    cout<<"……"<<endl;

    cout<<"——输出学生名册——"<<endl;
    outputStudentBook( stuBook );

    cout<<endl<<"——查找学生（100, abc）——"<<endl;

    STU_ITER iter = stuBook.find( Student("100", "abc"));
    cout<<"找到学生："<<iter->id<<' '<<iter->name<<endl;

   cout<<endl<<"——删除学生（100, abc）——"<<endl;
    stuBook.erase( iter );

   cout<<endl<<"——输出学生名册——"<<endl;
    outputStudentBook( stuBook );

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

建立一个控制台工程以及相应的头文件和源代码文件，将上述示例代码复制到文件中，编译并运行，结果如图 22.10 所示。

图 22.10 用 set 容器管理学生名册结果

为了能够将学生信息（Student 类的对象）保存到 set 容器中，在定义 Student 类时，为其重载了小于运算符“<”。这样当实例化类模板 set 时，就可以用 Student 类来实例化 less 这个比较函数对象。

22.4 其他关联式容器

其他关联式容器主要是建立在 map 和 set 容器基础之上的，本节只介绍两个，一个是建构在 set 基础之上的 multiset，另一个是建构在 map 基础之上的 multimap。multiset 与 set 的最大不同之处是 multiset 允许有相同的键值。multiset 也包含在标准头文件 set 中，看下面的例子：

```
multiset<int> msInts;
msInts.insert(2);
msInts.insert(2);
msInts.insert(1);
msInts.insert(4);
msInts.insert(3);
msInts.insert(3);
multiset <int>::iterator iter = msInts.begin();
for(; iter != msInts.end(); ++iter)
{
     cout << " " << *iter;              // 输出 1 2 2 3 3 4
}
cout << endl;
cout << "键值是 3 的元素个数：";
cout << msInts.count(3) << endl;       // 输出 2
out << "键值是 0 的元素个数：";
cout << msInts.count(0) << endl;       // 输出 0
```

从上面的程序可以看出，insert 方法可以插入相同的元素，count 方法统计个数的时候也是输出 2。

multimap 的方法与 map 也是相同的，但是 multimap 允许有重复的键值。multimap 也包含在标准头文件 map 中，看下面的程序：

```
multimap<int,int> mpInts;
mpInts.insert(multimap<int,int>::value_type(1, 10));
mpInts.insert(multimap<int,int>::value_type(2, 20));
mpInts.insert(multimap<int,int>::value_type(3, 30));
mpInts.insert(multimap<int,int>::value_type(1, 20));
mpInts.insert(multimap<int,int>::value_type(4, 40));
mpInts.insert(multimap<int,int>::value_type(2, 70));
multimap<int,int>::iterator iter = mpInts.begin();
for(; iter != mpInts.end(); ++iter)
{
     cout << " " << iter->first << " " << iter->second;
                                // 输出 1 10; 1 20; 2 20; 2 70; 3 30; 4 40
     cout << endl;
}
```

从上面的输出可以看到键值是 1 和 2 的记录分别有两个。

22.5 综合实例

在银行的自动取款机（ATM）上取款一般需要排队，并且每个人只能取走自己账号里的钱。下面的程序将模仿这样一个过程，其中排队者用 0~9 的数字编号代替，而账号则与该编号相同，并用一个 map 记录每个人账号里面的余额。程序如示例代码 22.3 所示。

示例代码 22.3

```
#include <cstdlib>
#include <iostream>
#include <queue>                          // 使用 deque
#include <map>                            // 使用 map
using namespace std;                      // 使用名称空间 std

int main(int argc, char *argv[])          // 主函数
{
   cout<<"——排队取款——"<<endl;            // 输出提示信息
   cout<<endl;
    map<int, int> accounts;               // 账号
    for( int i=0; i<10; i++ )             // 建立账号
    {
        pair<int, int> a( i, 100 + i );
        accounts.insert( a );
    }

    queue<int> people;                    // 排队者
    int id = 0;                           // 编号
    cout<<"请依次输入排队者编号，"
        <<"输入 0 退出"<<endl;

    cin>>id;                              // 输入排队者编号
    while( id != 0 )                      // 输入 0 退出
    {
        people.push( id );                // 建立队伍
        cin>>id;
    }
    cout<<endl;

    cout<<"取款顺序: "<<endl;             // 输出提示信息
    cout<<"排队者\t 取款数"<<endl;
    map<int, int>::iterator iter;         // 定义账号的迭代器
    while( !people.empty() )              // 遍历整个队伍
    {
        id = people.front();              // 找到队伍最前面的人
        iter = accounts.find(id);         // 取钱
        cout<<id<<'\t'<<iter->second<<' ';      // 输出信息

        people.pop();                     // 出队
```

```
        cout<<endl;
    }
    cout<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 22.11 所示。

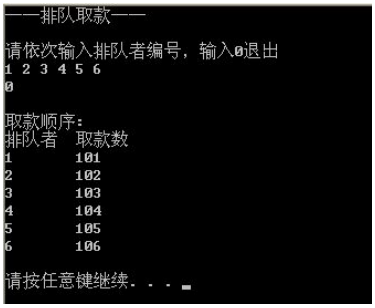


图 22.11 模拟排队取款结果

22.6 小结

本章主要讲解了常见的关联式容器，包括 map（映射）、set（集合）的定义和使用、multimap（多值映射）和 multiset（多值集合）的定义和使用。对于每一个关联式容器，都认真地分析了其概念，并结合具体的例子分析了各容器的功能。

◆ ◆ ◆

第 23 章　函数对象和算法

本章包括

- 函数对象概述
- STL 函数对象分类
- 函数对象适配器
- 算法概述
- STL 算法详解
- 各种常用的算法

函数对象和算法是 STL 中非常重要的两部分，也是联系非常紧密的两部分。算法是用来对容器进行各种计算和处理的，例如复制、查找、排序、删除等；而函数对象则是用来对算法的策略进行配置的，例如设置查找、删除等算法的条件，只对满足条件的元素进行处理。通过本章的学习，读者将掌握大量成熟、高效的算法，从而在实际开发中减少重复劳动，提高效率。

23.1　函数对象概述

简单来说，函数对象就是可以当做函数使用的对象。而使用函数的标志就是在函数名称后面加上函数调用运算符“()”，即一对括号，外加其中的函数参数。因此，如果一个对象能够当做函数使用，也必须能够在其标识符后面加上括号和参数。而要达到这样的目的，则必须为函数对象重载函数调用运算符。

函数对象的英文是 functor，有的文献将其译为仿函数。另外，有的文献并不区别函数对象和函数对象所属的类，在本书中，如不特别说明，也不区别这两个概念。

23.1.1　函数对象的定义

函数对象重载了函数调用运算符“()”，因此定义一个函数对象的关键就是定义该对象所属的类。通常情况下，为了让函数对象能够处理各种类型的参数，应当将该类定义为类模板。其语法如下：

```
template< 模板参数列表 >
class 类名                                  // 也可以使用 struct 关键字
{
    ……
public:
    返回类型 operator () ( 函数参数列表 )  // 返回类型和参数类型可以使用模板参数
    {
        return 返回值;
    }
    ……
};
```

在上述定义函数对象类的语法中可以看出，运算符“()”的重载与其他各种运算符（如小于运算符“<”、等于运算符“==”、大于运算符“>”）的重载没有什么不同。例如定义一个函数对象求两个参数中的较小值，其程序如下：

```
template <typename T>
struct minor                                    // 函数对象类
{
   T operator ( ) ( const T t1, const T &t2 )  // 重载函数调用运算符
   {
      return t1<t2 ? t1 : t2;                   // 返回参数中的较小值
   }
};
minor<int> minFuncObj;                          // 定义函数对象
cout<<minFuncObj( 3, 4 )<<endl;                 // 输出 3
```

在上面的例子中，首先定义了一个函数对象类模板 minor。为了简单起见，这里没有使用 class 关键字，而是使用了 struct 关键字。这样可以在定义公共成员时省略 public 关键字。在 minor 类模板中重载了函数调用运算符“( )”，该运算符接受两个模板类型的参数，并返回其中的较小值。

使用 minor 这个函数对象类模板时，首先用整数类型将其实例化，并定义一个对象 minFuncObj。该对象可以当做一个函数使用，程序在 minFuncObj 之后附加函数调用运算符“( )”，并传入两个参数 3 和 4。

调用函数对象的重载运算符“()”时，要传入参数，这些参数有时也称做函数对象的操作数。

23.1.2 用函数对象替代函数指针

设计函数对象的主要目的是用来配置算法的策略。所谓算法的策略，就是计算或处理数据的方法、目标、条件等。为了将函数设计得更加灵活，并能够尽量重用，通常可以通过函数的参数来指定函数的计算方法、目标、条件等，而该参数就是所谓的算法策略。配置函数的算法策略有如下三种方法：

- 使用数值。
- 使用函数指针。
- 使用函数对象。

如果使用数值方法，那么在设计算法时必须根据不同的数值做出相应的处理，这样做不够灵活。而使用函数指针，则可以由调用者制定算法策略，比较灵活。例如，设计一个查找函数 find_if，至于什么样的数据符合要求，则由调用者通过参数指定。假设使用数值方法来表示算法策略，则应当先定义一个表示各种策略的枚举体，然后在定义函数时为函数传入一个该枚举体的变量，以表示应用的策略，例如：

```
enum eRelation                                  // 表示计算策略的枚举体
{
```

```
    eLess,                                          // 小于
    eEqual,                                         // 等于
    eGreater                                        // 大于
};
template< typename T, typename ITER_TYPE >
ITER_TYPE find_if_number( ITER_TYPE begin, ITER_TYPE end,
                eRelation relType, const T &target )
                                                    // 用数值表示策略的 find_if 函数
{
    for(ITER_TYPE iter = begin; iter!=end; iter++) // 遍历迭代器区间
    {
        switch( relType )                           // 根据不同的策略
        {
        case eLess:
            if( *iter < target )                    // 比较当前元素与目标
            {
                return iter;                        // 返回迭代器
            }
        case eEqual:
          if( *iter == target )
            {
                return iter;
            }
        case eGreater:
          if( *iter > target )
            {
                return iter;
            }
        }
    }
    return end;                                     // 失败，则返回区间尾部
}
int main(int argc, char *argv[])                    // 主函数
{
    int ary[5] = {1,2,3,4,5};                       // 数组
    vector<int> vec(ary, ary+5);                    // 构造向量
    vector<int>::iterator iter;                     // 定义迭代器
    iter = find_if_number( vec.begin(), vec.end(), eEqual, 3);
                                                    // 应用数值策略查找
    cout<< *iter <<endl;                            // 输出查找结果 3
    return EXIT_SUCCESS;
}
```

显然，上述查找策略并不灵活，函数比较复杂，而且维护起来也比较烦琐。一旦表示查找策略的枚举体 eRelation 有什么修改，那么算法函数也要做出相应的修改。假设使用函数指针来表示算法策略，则应当在算法函数中传递一个函数指针，用以传递实现算法策略的函数，例如：

```
bool Equal(const int &ele, const int &target)      // 策略实现函数
{
    return ele == target;
}
```

```
template< typename T, typename ITER_TYPE >          // 模板参数
ITER_TYPE find_if_fun_ptr( ITER_TYPE begin, ITER_TYPE end, // 迭代器区间参数
        bool (*PFUN)(const T &, const T &),         // 表示算法策略的函数指针
        const T& target )                           // 查找目标
{
    for(ITER_TYPE iter = begin; iter!=end; iter++) // 遍历迭代器区间
    {
       if(PFUN(*iter, target))                      // 应用算法策略(函数)
       {
            return iter;                            // 返回迭代器
        }
    }
    return end;                                     // 失败,则返回区间尾部
}
int main(int argc, char *argv[])                    // 主函数
{
    int ary[5] = {1,2,3,4,5};                       // 定义数组
    vector<int> vec(ary, ary+5);                    // 定义向量
    vector<int>::iterator iter;                     // 定义迭代器
    iter = find_if_fun_ptr(vec.begin(), vec.end(), Equal, 3);
                                                    // 使用函数指针表达算法策略
    cout<< *iter <<endl;                            // 输出结果3

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

显然,使用函数指针使得算法函数更加简洁,而且容易维护。由于算法策略都是由调用者提供的,所以给程序开发带来了很大的灵活性,而且也使得算法函数的重用性更好。尽管如此,使用函数指针表达算法策略仍然有一定的局限性:

- 不能存储数据,以用于更复杂的操作。
- 不能使用内联函数,以进一步提高调用效率。

而使用函数对象,则可以解决上述问题。函数对象毕竟是一个对象,可以带有数据,从而可以进行复杂的操作。另外,函数对象重载的函数调用运算符"()"也可以指定为内联函数,从而带来效率方面的提升。使用函数对象表示查找某个目标数据的算法如下:

```
template<typename T>
struct equal2                                       // 函数对象
{
    bool operator()(const int &ele, const int &target) // 重载函数调用运算符
    {
        return ele == target;
    }
};
template< typename T, typename ITER_TYPE, typename Comp >
ITER_TYPE find_if_fun_obj( ITER_TYPE begin, ITER_TYPE end, // 迭代器区间
```

```
        Comp funObj,                                        // 函数对象参数
        const T& target )                                   // 目标
{
    for(ITER_TYPE iter = begin; iter!=end; iter++)          // 遍历迭代器区间
    {
       if(funObj(*iter, target))                            // 应用算法策略
       {
            return iter;
        }
    }
    return end;
}
int main(int argc, char *argv[])                            // 主函数
{
    int ary[5] = {1,2,3,4,5};                               // 定义数组
    vector<int> vec(ary, ary+5);                            // 定义向量
    vector<int>::iterator iter;                             // 定义迭代器
    iter = find_if_fun_obj(vec.begin(), vec.end(), equal2<int>(), 3);
                                                            // 使用函数对象 ,查找目标
    cout<< *iter <<endl;                                    // 输出结果 3

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

通过上面的例子可以看出，使用函数对象作为算法策略，可以使得算法函数很简洁。在这一点上同使用函数指针效果是一样的。但函数对象的定义更加灵活，在函数对象构造时可以传入某些数据，或者将某些算法过程写成成员函数。例如，可以将查找的目标保存在函数对象里，而不是像传统的函数指针那样传入参数。看下面的例子：

```
template<typename T>
struct equal2                                           // 函数对象
{
    equal2(T target):m_target( target )                 // 构造函数
    {}
    bool operator()(const int &ele)                     // 重载函数调用运算符
    {
        return ele == m_target;
    }
private:
    T m_target;                                         // 函数对象数据
};
template<typename ITER_TYPE, typename Comp>
ITER_TYPE find_if_fun_obj( ITER_TYPE begin, ITER_TYPE end,// 迭代器区间
        const Comp &funObj )                            // 函数对象参数
{
    for(ITER_TYPE iter = begin; iter!=end; iter++)      // 遍历迭代器区间
    {
       if(funObj(*iter))                                // 应用算法策略
```

```
        {
            return iter;                                    // 返回目标迭代器
        }
    }
    return end;                                             // 失败，则返回区间尾
}
int main(int argc, char *argv[])                            // 主函数
{
    int ary[5] = {1,2,3,4,5};                               // 数组
    vector<int> vec(ary, ary+5);                            // 定义向量
    vector<int>::iterator iter;                             // 定义迭代器
    equal2<int> eq2( 3 );                                   // 定义函数对象
    iter = find_if_fun_obj(vec.begin(), vec.end(), eq2);// 使用函数对象查找数据
    cout<< *iter <<endl;                                    // 输出结果 3

    system("PAUSE");
    return EXIT_SUCCESS;                                    // 返回
}
```

在上面的例子里，构造函数对象 eq2 时可以传入一个参数，作为查找的目标数据。eq2 将该参数保留在其私有数据中。当程序中使用函数对象调用其重载的运算符“( )”时，即可使用该数据。

23.2 STL 函数对象分类

为了方便用户使用，在 STL 中已经定义了大量的函数对象，这些函数对象涉及了算法的诸多方面，基本可以满足开发者的要求。而且，STL 还提供了一些适配器，可以将已有的函数对象转换成特殊的函数对象。要使用 STL 中的函数对象，应当包含<functional>头文件。

对于 STL 中的函数对象，一般有两种分类方法。一种是按照函数对象所能接受参数的个数，分为一元函数对象和二元函数对象。另外一种是按照函数对象处理问题的范围，分为算数类函数对象、关系类函数对象和逻辑类函数对象。

在有的 STL 版本中，还有不带参数的函数对象。当然，这里所说的参数个数不包括对象本身，也就是说第一个参数 this 指针是不包括在内的。另外，STL 中没有三元以及三元以上的函数对象，因为一般来讲二元已经足够了。如果参数再增加，则算法就比较复杂，不适合用函数对象。

23.2.1 一元函数对象

一元函数对象仅接受一个参数。例如求相反数的函数对象 negate。该函数对象接受一个数值参数，并返回一个该参数的相反数。比如求整数-1 和浮点数 3.14 的相反数，程序如下：

```
int x = negate<int>() ( -1 );       // 使用临时对象，求-1 的相反数，结果为 1
negate<double> dNeg;                // 定义一个求浮点数相反数的函数对象
double y = dNeg( 3.14 );            // 利用函数对象，求浮点数的相反数，结果为-3.14
```

在上述示例代码中，展示了两种使用函数对象的方法。一种是使用临时函数对象，另一种是先定义一个函数对象变量，然后直接使用该变量。这两种方法都是合法的，也许有很多初学者比较习惯第二种使用方法。但实际上，经常使用的是第一种。其优点是比较简单，避免了定义多余的变量。

在 STL 中所有的一元函数对象都派生自 unary_function。unary_function 是一个类模板，接受两个模板参数。这两个参数都与重载的函数调用运算符“()”相关，一个是该运算符参数的类型，另外一个是该运算符的返回值类型。后文中如果没有特别说明，则分别称为函数对象的参数类型和返回值类型。

unary_function 本身定义非常简单，实际上只是使用 typedef 关键字定义了两种类型，即表示运算符“()”参数的类型和表示该运算符返回值的类型：

```
template <class _Arg, class _Result>
struct unary_function
{
    typedef _Arg argument_type;                 // 参数类型
    typedef _Result result_type;                // 返回值类型
};
```

虽然没有实现什么功能，但 STL 定义如此简单的基类并不是没有意义的。事实上，正是有了这两个简单的类型定义，才使得 STL 函数对象可以被适配器修饰，从而表现出新的功能。有关函数对象适配器的详细说明，请参考后面的相关内容。

在 STL 中只有很少几个函数对象是一元的，包括 negate 和 logical_not，分别用于数值取反和逻辑取反。另外还有几个函数对象，如 unary_negate，binder1st，binder2nd，pointer_to_unary_function，这些虽然也是一元函数对象，但都属于适配器，在后文中将用一个单独的小节进行说明。

23.2.2 二元函数对象

二元函数接受两个参数。例如表示加法的函数对象是 plus，该函数对象可以把两个参数相加，并返回相加的结果。如果要定义一个将两个整数相加的函数对象，那么程序应当如下书写：

```
#include <functional>                       // 包含函数对象头文件
std::plus< int > intAddObj;                 // 使用函数对象 plus
    cout<< intAddObj( 3, 4 )<<endl;         // 输出 7
```

在 STL 中二元函数对象都派生自 binary_function。同 unary_function 一样，binary_function 也是一个类模板，该类模板接受三个模板参数。前面两个模板参数表示的是重载运算符“()”两个参数的类型，第三个模板参数表示的是其返回值类型。其定义如下：

```
template <class _Arg1, class _Arg2, class _Result>
struct binary_function
{
```

```
    typedef _Arg1 first_argument_type;          // 第一参数类型
    typedef _Arg2 second_argument_type;         // 第二参数类型
    typedef _Result result_type;                // 返回值类型
};
```

在 STL 中有很多二元函数对象，这里不再一一详述。

23.2.3 算术类函数对象

算术类函数对象支持加、减、乘、除、求余和取反，相应的函数对象分别为 plus，minus，multiplies，divides，modulus 和 negate。如果函数对象实例化的类型是一个类，则该类应当重载相应的运算符，否则会出错。例如：

```
#include <cstdlib>
#include <iostream>
#include <functional>                       // 包含函数对象头文件
using namespace std;
int main(int argc, char *argv[])            // 主函数
{
    cout<< plus<int>()(3, 4) <<endl;        // 3 + 4
    cout<< minus<int>()(5, 3) <<endl;       // 5 - 3
    cout<< multiplies<int>()(4, 6)<<endl;   // 3 * 4
    cout<< divides<int>()(5, 4)<<endl;      // 5 / 4
    cout<< modulus<int>()(5, 3)<<endl;      // 求 5÷3 的余数
    cout<< negate<int>()( 3 )<<endl;        // 求 3 的相反数
    system("PAUSE");                        // 暂停
    return EXIT_SUCCESS;                    // 主函数返回
}
```

23.2.4 关系类函数对象

关系类函数对象支持等于、不等于、大于、大于等于、小于和小于等于。相应的函数对象分别为 equal_to，not_equal_to，greater，greater_equal，less 和 less_equal。同样，实例化函数对象的类，也必须支持相应的运算符。例如：

```
#include <cstdlib>
#include <iostream>
#include <functional>                           // 包含函数对象头文件
using namespace std;
int main(int argc, char *argv[])                // 主函数
{
   cout<< boolalpha;                            // 以文字方式输出 bool 值
   cout<< equal_to<int>()(3, 4) <<endl;         // 3 == 4
   cout<< not_equal_to<int>()(3, 4) <<endl;     // 3 !=4
   cout<< greater<int>()(5, 3) <<endl;          // 5 > 3
   cout<< greater_equal<int>()(5, 3)<<endl;     // 5 >= 3
   cout<< less<int>()(5, 4)<<endl;              // 5 < 4
   cout<< less_equal<int>()(5, 3)<<endl;        // 5 <= 3
   system("PAUSE");                             // 暂停
```

```
    return EXIT_SUCCESS;                                    // 主函数返回
}
```

23.2.5 逻辑类函数对象

逻辑类函数对象支持逻辑与、逻辑或和逻辑非操作。相应的函数对象分别为 logical_and，logical_or 和 logical_not。例如：

```
#include <cstdlib>
#include <iostream>
#include <functional>                                       // 包含函数对象头文件
using namespace std;
int main(int argc, char *argv[])                            // 主函数
{
    cout<< boolalpha;                                       // 以文字方式输出 bool 值
    cout<< logical_and<int>()( 3, 4 ) <<endl;               // 3 && 4
    cout<< logical_or<int>()( 3, 0 ) <<endl;                // 3 || 0
    cout<< logical_not<int>()( 0 ) <<endl;                  // !0
    system("PAUSE");                                        // 暂停
    return EXIT_SUCCESS;                                    // 主函数返回
}
```

23.2.6 STL 函数对象的一般应用

STL 函数对象中最常用的是关系类函数对象，如 less，equal，greater 等。这些函数对象通常用来设定算法的策略。例如，关联式容器的键值排序的默认值是 less，意味着在关联式容器中，元素间是按照键值由小到大的顺序进行排列的。容器头 begin()对应的是最小值，而 end()之前的迭代器对应的则是最大值。如果要改变这种策略，则要在容器模板实例化时，更改这个键值排序函数对象。例如改变为由大到小排列，则应当如下实例化：

```
#include <functional>                                       // 包含函数对象头文件
std::map<int, int, greater<int> > intMap;                   // 由大到小地排列键值
std::set<int, greater<int> > intSet;                        // 由大到小地排列键值
```

STL 中的众多排序函数也可以用这种形式进行配置。例如链表 list 的排序函数 sort()，默认情况下是从小到大进行排序的。如果要从大到小排序 则可以将函数对象 greater 作为参数传给 sort( ) 函数，如下所示：

```
#include <functional>                                       // 包含函数对象头文件
list<int> intList;                                          // 定义链表
……                                                          // 建立链表
intList.sort( greater<int>() );                             // 以从大到小方式排列链表
```

23.3 函数对象适配器

同容器、迭代器一样，函数对象也可以被适配，即通过简单的包装，将函数对象转换为某种特殊用途的函数对象。例如，将本来的二元函数对象包装成一元函数对象，将本来表示“小于”操作的函数对象转换成“大于等于”函数对象。

常见的函数对象适配器包括绑定器（bind）和取反器（negate）。有的版本的函数对象适配器还包括组合器（compose）。

23.3.1 可以适配的函数对象

STL 中的函数对象都是可以适配的。可以适配的关键在于这些函数对象都派生自 unary_function 或者 binary_function。虽然在这两个类模板中并没有定义复杂的数据或操作，而只是用 typedef 关键字定义了参数类型和返回值类型，但是正因为如此，才使得 STL 函数对象获得了可以适配的能力。

在 STL 中，函数对象适配器也是一些函数对象，其定义的形式也是类模板。适配器的模板参数是“被适配”的函数对象类。由于这些函数对象类都派生自 unary_function 或者 binary_function，所以很容易获取其参数类型和返回值类型。定义适配器时，可以将这些类型作为适配器的参数类型和返回值类型。在适配器的函数调用运算符“( )”定义中，可以调用被适配函数对象的功能，并附加一些操作，从而完成适配。函数对象适配器如图 23.1 所示。

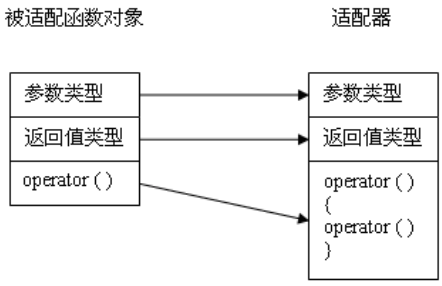


图 23.1　函数对象适配器

例如，在一元取反适配器 unary_negate 中，该类模板接受一个函数对象类（称为 Predicate）作为其模板实参，而该实参函数对象是一个一元函数对象，接受一个参数，取反器 unary_negate 的作用是调用模板实参函数对象_Predicate 的重载运算符“( )”，并对其结果进行逻辑取反。

unary_negate 的定义如下：

```
emplate <class _Predicate>                  // 适配函数对象_Predicate
class unary_negate
    : public unary_function<
        typename _Predicate::argument_type, // 使用_Predicate 的类型定义实例化基类
        bool>
{
protected:
    _Predicate _M_pred;                     // 声明_Predicate 的成员变量
ublic:
```

```
    explicit
    unary_negate(const _Predicate& __x) :  // 构造函数，初始化被适配的函数对象
        _M_pred(__x) {}

    bool                                   // 接受同_Predicate 参数相同类型的参数
    operator( ) (const typename _Predicate::argument_type& __x) const
    {
        return !_M_pred(__x);          // 调用_Predicate 的功能，并对结果取反
    }
};
```

unary_negate 适配器的模板参数通常是结果为 bool 值的函数对象。这些函数对象通常用于判断，因此在定义 unary_negate 时将其模板形参命名为_Predicate。

使用 unary_negate 适配函数对象 logical_not 的代码如下：

```
#include <cstdlib>
#include <iostream>
#include <functional>                  // 包含 STL 函数对象头文件
using namespace std;                   // 使用标准名称空间 std
int main(int argc, char *argv[])       // 主函数
{
    logical_not<bool> lb;              // 定义 logical_not<bool>函数对象
    unary_negate< logical_not<bool> > notNeg( lb );
                                       // 用 unary_negate 适配 logical_not
    cout<<boolalpha<< notNeg( false ) <<endl;
                                       // 求 neg( false )的值并输出，结果为 false
    system("PAUSE");                   // 暂停
    return EXIT_SUCCESS;               // 返回
}
```

在上述例子中，使用函数对象适配器 unary_negate 适配 logical_not 的方法很简单，将后者作为前者的模板实参即可，如第 9 行所示。适配后产生的新类可以用来定义变量，在本例中就是 notNeg。由于 logical_not 是对参数取反，unary_negate 是对 logical_not 的结果取反，反反得正，所以第 10 行 notNeg(false)的结果就是 false，即保持原来参数的值。

在第 9 行中不可以使用临时对象（logical_not<bool>( )），否则在编译过程中，编译器会将该行当做一个函数声明。对编译器来讲，“logical_not<bool>( )”更容易被当做一个返回 logical_not<bool>对象的函数指针，等同于“logical_not<bool> (*)( )”。

23.3.2 绑定器

STL 提供了两个函数对象绑定器 binder1st 和 binder2nd。绑定器针对的是二元函数对象，绑定的过程是将二元函数对象的一个实参限定为一个特殊的值，从而将其转换成一元函数对象。例如 binder1st 函数对象绑定的是第一个参数，即限定第一个参数为指定的值，而 binder2nd 绑定的是

第二个参数，即限定第二个参数为指定的值。

例如，二元函数对象 less 接受两个参数，返回值表示的是第一个参数是否小于第二个参数，其代码如下：

```
template <class _Tp>
   struct less : public binary_function<_Tp, _Tp, bool>
                                          // less 从 binary_function 派生
   {
     bool
     operator()(const _Tp& __x, const _Tp& __y) const // 接受两个参数
     { return __x < __y; }                // 返回第一个参数是否小于第二个参数
   };
```

出于某种特殊目的，要求比较某个数是否小于 10。开发者可以单独设计一个类，比如 less_n，其中定义一个成员变量来保留比较的目标值，然后在构造函数中将 10 作为参数，并初始化这个成员变量。这样做虽然可以达到目的，但显然比较烦琐。另外一种可行的方法就是使用绑定器 binder2nd。绑定之后，将 less 的第二个参数限定为 10，则新的函数对象就只能与 10 进行比较。程序如下：

```
#include <cstdlib>
#include <iostream>
#include <functional>                          // 包含 STL 函数对象头文件
using namespace std;                           // 使用名称空间 Std
int main(int argc, char *argv[])               // 主函数
{
    binder2nd<less<int> > bl( less<int>(), 10 ); // 使用绑定器 binder2nd 适配 less
    cout<<boolalpha<< bl( 8 ) <<endl;          // 8 < 10，true
    cout<<boolalpha<< bl(12) <<endl;           // 12 < 10，false

    system("PAUSE");                           // 暂停
    return EXIT_SUCCESS;                       // 返回
}
```

使用绑定器时要注意其结果所表达的语义。诸如 less，less_euqal，greater 等函数对象，从其语义来讲表示第一个参数与第二个参数的关系，小于、小于等于、大于等。如果使用 binder1st，则表示指定值是否小于、小于等于、大于参数；如果使用 binder2nd，则表示参数是否小于、小于等于、大于指定值。

其实绑定器的实现并没有什么特殊之处，其主要思想就是将目标值保存在成员变量里面，在调用被绑定函数对象时，将该成员变量作为参数。例如 binder2nd 的代码如下：

```
template <class _Operation>                // 适配函数_Operation
   class binder2nd                         // binder2nd 派生自 unary_function
   : public unary_function<typename _Operation::first_argument_type,
               typename _Operation::result_type>
{
```

```
protected:
    _Operation op;                          // 记录被适配的函数对象
     typename _Operation::second_argument_type value;
                                            // 记录被绑定参数的值，目标值
public:
    binder2nd(const _Operation& __x,        // 构造函数，接受被适配的函数对象变量
        const typename _Operation::second_argument_type& __y)
                                            // 目标值，被绑定参数
        : op(__x), value(__y) {}            // 初始化
    typename _Operation::result_type        // 重载运算符“()”
    operator()(const typename _Operation::first_argument_type& __x) const
                                            // 常引用参数
    { return op(__x, value); }              // 调用被绑定函数对象的运算符“()”
    typename _Operation::result_type        // 重载运算符“()”
    operator()(typename _Operation::first_argument_type& __x) const
                                            // 非常引用参数
    { return op(__x, value); }              // 调用被绑定函数对象的运算符“()”
};
```

binder1st 的代码与上述代码非常类似，差别只是在调用被绑定函数对象的重载运算符“()”时，binder2nd 是将目标值（成员变量 value）作为第二参数，而 binder1st 则是将目标值（成员变量 value）作为第一参数。

23.3.3 绑定函数

在实际开发中，绑定器使用比较频繁。为了方便开发者使用，STL 提供了两个函数模板 bind1st 和 bind2nd，分别用于生成相应的绑定器。这两个函数模板的定义如下：

```
template <class _Operation, class _Tp> // 模板参数：被绑定函数对象和目标值
inline binder1st<_Operation>           // 内联函数，返回 binder1st<_Operation>
bind1st(const _Operation& __fn, const _Tp& __x)       // 绑定函数
{
    typedef typename _Operation::first_argument_type _Arg1_type;
    return binder1st<_Operation>(__fn, _Arg1_type(__x));
                                       // 返回 binder1st<_Operation>对象
}
template <class _Operation, class _Tp> // 模板参数：被绑定函数对象和目标值
inline binder2nd<_Operation>           // 内联函数，返回 binder2nd<_Operation>
bind2nd(const _Operation& __fn, const _Tp& __x)       // 绑定函数
{
    typedef typename _Operation::second_argument_type _Arg2_type;
    return binder2nd<_Operation>(__fn, _Arg2_type(__x));
                                       // 返回 binder2nd<_Operation>对象
}
```

使用这两个绑定函数也非常方便，不必像使用类模板一样显式实例化，例如：

```
#include <cstdlib>
#include <iostream>
```

```
#include <functional>                                   // 包含函数对象头文件
using namespace std;                                    // 使用名称空间 std
int main(int argc, char *argv[])                        // 主函数
{
    binder1st<less<int> > b1 = bind1st( less<int>(), 10 );// 调用 bind1st 函数
    cout<<boolalpha<< b1(8) <<endl;                     // 10 < 8，false
    cout<<boolalpha<< b1(12) <<endl;                    // 10 < 12，true

    binder2nd<less<int> > b2 = bind2nd( less<int>(), 10 );// 调用 bind2nd 函数
    cout<<boolalpha<< b2(8) <<endl;                     // 8 < 10，true
    cout<<boolalpha<< b2(12) <<endl;                    // 12 < 10，false
    system("PAUSE");                                    // 暂停
    return EXIT_SUCCESS;                                // 返回
}
```

23.3.4 取反器

STL 提供了两个取反器（取反适配器的简称），unary_negate 和 binary_negate，分别用于对一元函数对象和二元函数对象的结果取反。unary_negate 已经在前面的章节中讲过了，所以这里只讲一下 binary_negate。

binary_negate 作为一个类模板，接受一个函数对象_Predicate 作为其模板参数。binary_negate 的重载运算符“( )”接受两个参数，并调用_Predicate 的运算符“( )”传递这两个参数。binary_negate 对其调用结果取反，并返回。

同 unary_negate 一样，binary_negate 的模板参数通常也是一些返回值类型是 bool 的函数对象，通常用于判断，所以将其模板形参命名为_Predicate。

binary_negate 的定义如下：

```
template <class _Predicate>                     // 类模板形参
class binary_negate                             // binary_negate
   : public binary_function<typename _Predicate::first_argument_type,
                                                // 第一参数类型
                typename _Predicate::second_argument_type, // 第二参数类型
                bool>                           // 返回值类型
{
protected:
    _Predicate _M_pred;                         // 保存被适配函数对象的成员变量
public:
    explicit
    binary_negate(const _Predicate& __x)        // 构造函数
        : _M_pred(__x) { }                      // 使用一个适配函数对象初始化成员变量
    bool                                        // 重载函数调用运算符“( )”
```

```
    operator()(const typename _Predicate::first_argument_type& __x,
           const typename _Predicate::second_argument_type& __y) const
     { return !_M_pred(__x, __y); }       // 调用被适配函数对象的运算符“( )”
};
```

使用取反器 binary_negate 对函数对象 less 进行适配，其代码如下：

```
#include <cstdlib>
#include <iostream>
#include <functional>                          // 包含函数对象头文件
using namespace std;                           // 使用名称空间 std
int main(int argc, char *argv[])               // 主函数
{
    less<int> lsint;                           // 定义 less<int>对象
    binary_negate<less<int> > bneg( lsint );   // 用 binary_negate 适配 less
    cout<<boolalpha<< bneg( 1, 2) <<endl;  // 用适配后的函数对象进行计算！(1<2)
    system("PAUSE");                           // 暂停
    return EXIT_SUCCESS;                       // 返回
}
```

同 unary_negate 一样，在第 9 行不能使用临时函数对象 less<int>()，否则会被当成一个函数声明。

23.3.5 取反函数

同绑定函数一样，STL 也为取反器提供了相应的取反函数 not1 和 not2，分别用于生成一元和二元函数对象的取反器对象。其定义如下：

```
template <class _Predicate>              // 模板参数：被适配的函数对象
inline unary_negate<_Predicate>          // 内联函数
   not1(const _Predicate& __pred)        // 接受被适配函数对象类型的参数
    { return unary_negate<_Predicate>(__pred); }
                                         // 返回 unary_negate<_Predicate>对象
template <class _Predicate>              // 模板参数：被适配的函数对象
   inline binary_negate<_Predicate>      // 内联函数
   not2(const _Predicate& __pred)        // 接受被适配函数对象类型的参数
   { return binary_negate<_Predicate>(__pred); }
                                         // 返回 unary_negate<_Predicate>对象
```

使用取反函数 not1 和 not2，分别对 logical_not 和 less 进行适配，程序如下：

```
#include <cstdlib>
#include <iostream>
#include <functional>                    // 包含函数对象头文件
using namespace std;                     // 使用名称空间 std
int main(int argc, char *argv[])         // 主函数
{
    typedef logical_not<int> LN;         // 定义 logical_not<int>类型
```

```
    unary_negate<LN > uneg = not1( LN() ); // 使用取反函数 not1
    cout<<boolalpha;                    // 设置输出 bool 值的文字形式
    cout<< uneg( 12 ) <<endl;           // !( !12 ), true
    cout<< uneg( 0 ) <<endl;            // !( !0 ), false

    binary_negate<less<int> > bneg = not2( less<int>() );// 使用取反函数 not2
    cout<< bneg( 1, 2 )<<endl;          // !( 1<2 ), false
    cout<< bneg( 2, 1 )<<endl;          // !( 2<1 ), true
    system("PAUSE");                    // 暂停
    return EXIT_SUCCESS;                // 返回
}
```

23.3.6 适配器的级联

将适配器应用于函数对象，其结果也是一个函数对象，而这个新的函数对象可以继续被其他适配器所用。依此类推，适配器可以层层级联下去，从而构造出复杂的表达式。例如为了创建一个“大于 10”的函数对象，可以使用绑定器 binder2nd 适配 greater，也可以配合 unary_negate，使用绑定器 binder2nd 适配 less_equal，原因是“大于 10”也可以表述为“不小于等于 10”。所以下面程序中的两段代码是等价的。

```
#include <cstdlib>
#include <iostream>
#include <functional>                       // 包含函数对象头文件
using namespace std;                        // 使用名称空间 std
int main(int argc, char *argv[])            // 主函数
{
    cout<<boolalpha;                        // 设置输出 bool 值的文字形式
    typedef greater<int> GINT;              // 定义类型
    binder2nd<GINT> bg10= bind2nd(GINT(), 10); // 构建“大于 10”的函数对象
    cout<< bg10( 8 ) <<endl;                // 8 >10
    cout<< bg10( 12 ) <<endl;               // 12>10

    typedef less_equal<int> LE;             // 定义类型
    typedef binder2nd<LE> BLE;              // 定义类型
    unary_negate<BLE> nbl10 = not1(BLE(LE(), 10));// 构建“不小于等于 10”的函数对象
    cout<< nbl10( 8 ) <<endl;               // 8 >10
    cout<< nbl10( 12 ) <<endl;              // 12>10
    system("PAUSE");                        // 暂停
    return EXIT_SUCCESS;                    // 返回
}
```

由于适配器级联，导致生成的函数对象类型非常复杂。所以在实际使用中，通常不会像上述例子那样，定义一个函数对象的变量，而是大量使用临时对象。这样就可以省略变量的定义，从而避免书写复杂的类型。

23.4 算法概述

所谓算法，指的是以有限的步骤解决某种逻辑或数学问题的方法。在计算机编程中，算法是对输入数据的处理，并在有限时间内输出期待的处理结果。算法对输入的数据有格式上的要求，执行过程中，算法按照该格式对数据进行读取、分析和处理。这里的格式即所谓的数据结构。在 STL 中，算法所涉及到的数据结构就是各种容器。

注意区分算法描述和算法实现的区别。描述一个算法可以使用自然语言、伪代码、流程图等手段，而实现一个算法就是将算法描述转换成具体的代码，用一个或多个函数来表示。处于通用的目的，也可以使用函数模板。

23.4.1 算法的特征

算法有如下五个方面的特征：

- 有穷性。
- 确切性。
- 输入。
- 输出。
- 可行性。

有穷性，指的是算法应当保证在有限的步骤内结束。如果一个算法耗时太长，迟迟不能给出有效的输出，那么该算法的设计是不成功的。

确切性，指的是算法的每一个步骤都应当有确切的定义。如果某个步骤没有确切的定义，那么该步骤就是多余的。

输入，一个算法有 0 个或多个输入数据。输入数据确定了算法的起始状态。如果算法没有输入数据，则说明算法本身定义了起始状态。例如，一个求 100 以内所有素数的算法就不需要任何输入。

输出，一个算法有一个或多个输出。输出反映的是算法对输入数据的处理结果。如果算法没有任何输出，那么该算法就没有什么意义。

可行性，指的是算法应当能够在现有计算机软硬件的基础上执行，甚至人工也可以在有限次的计算下完成。如果现有条件不能支持算法的运行，那么算法也没有什么意义。

23.4.2 算法的复杂度

评价一个算法的复杂度，主要包括两个方面：

- 时间复杂度。
- 空间复杂度。

算法的时间复杂度是用来度量算法的运行时间的。通常来讲，执行一个算法所耗费的时间是难

以计算出来的。而且由于现在多数计算机系统都是多任务、多线程并行执行的。因此，在不同的运行环境下，算法的执行时间也是不同的。但是实际上也并不需要知道具体的时间，只需知道影响算法执行时间的因素以及如何影响即可。在算法中，影响执行时间的因素包括：

- 问题的规模。
- 基本操作的重复次数。

问题的规模，即要处理的数据数量，一般用字母 n 来表示。而基本操作的重复次数则是 n 的函数，用 $T(n)$来表示。对于 $T(n)$，可以找到这样一个函数 $f(n)$，使得当 n 趋近于无穷大时，$T(n)/f(n)$ 的极限值趋于一个非零常数，则 $f(n)$是 $T(n)$的同数量级函数，$T(n)$可以表示为 $f(n)$的函数，即 $T(n) = O(f(n))$。$O(f(n))$即为算法的渐进时间复杂度，简称时间复杂度。

$T(n)$往往是一个多项式，而且其各个子式中 n 的次数各不相同。$f(n)$就是这些子式中，n 的次数最高的项去除系数的那部分。例如，假设 $T(n) = 4n^3+n^2+1$，则 $f(n) = n^3$。

常见的时间复杂度有常数阶 $O(1)$、对数阶 $O(\log 2n)$、线性阶 $O(n)$、线性对数阶 $O(n\log_2 n)$、平方阶 $O(n^2)$、立方阶 $O(n^3)$，……，k 次方阶 $O(n^k)$，指数阶 $O(2^n)$。随着问题规模 n 的不断增大，上述时间复杂度不断增大，算法的执行效率逐渐降低。

23.4.3 算法泛化

算法是用来处理数据的，但是不应当限制数据的类型。例如一个能够对整数数组进行排序的算法，也应当可以用来排序浮点数数组、字符串数组，甚至对象数组。另外，算法也要尽量避免对数据结构的依赖。也就是说，一个算法不仅可以应用于向量，也可以应用于数组、链表、队列、映射、集合等各种数据结构，甚至未知的数据结构。

如果一个算法只能应用于某种数据类型或者数据结构，那么为了在其他数据类型或数据结构上应用该算法，则必然要为之编写相应的实现，那将导致代码文本大量增加，从而使得程序难以维护。

使算法独立于数据类型、数据结构，就是算法泛化的主要目的。而算法泛化的主要手段包括：

- 使用模板，抽象数据类型。
- 使用迭代器，抽象数据遍历过程和数据指示方法。

模板自不必说，要支持各种不同类型的数据必然要使用模板。使用迭代器的好处是可以独立于各种容器，遍历数据可以使用统一的迭代器操作方法（前进++，后退--，跳跃±n），获取数据也可以简单地“解引用迭代器”，如果数据是类，访问数据的成员可以通过迭代器重载的指针运算符“->”实现。例如定义一个查找函数 find，如果不对其进行泛化处理，那么 find 的实现形式一般如下：

```
int* find( const int *pAry, int size, int val)     // find 函数
{
```

```
    for(int i=0; i<size; i++)                          // 遍历数组
    {
        if( *(pAry+i) == val )                         // 比较当前元素与目标值
        {
            return const_cast<int*>(pAry+i);           // 如果相等，则返回相应的指针
        }
    }
    return NULL;                                       // 找不到就返回 NULL
}
```

显然上述的 find 函数只能应用于有限的几种数据类型，包括其声明的整型以及其他可以转换为整型的数据类型，如字符型、浮点型等。如果将该方法应用于其他数据类型或者向量、链表等数据结构，则会导致编译失败。所以，为了能够处理各种数据类型，应当对上述算法进行如下的泛化处理：

```
template<typename Iter, typename T>                    // 使用函数模板，数据类型泛化
Iter find( Iter first, Iter last, const T &val )       // 数据区间用迭代器表示
{
    for( Iter it= first; it!=last; it++ )              // 使用迭代器方法遍历数据
    {
        if( *it == val )                               // 解引用迭代器，比较目标值
        {
            return it;                                 // 如果相等，返回相应的迭代器
        }
    }
    return last;                                       // 找不到就返回区间尾
}
```

经过泛化处理后，可以使用该函数处理各种类型的数据，并且可以应用到各种数据结构上，只要该数据结构支持迭代器即可。

23.5 STL 算法详解

经过泛化处理的算法称做泛型算法。STL 中的算法都是泛型算法，这些算法涉及数据处理的诸多方面，包括查找、排序、算术、排列、组合等。全面地了解这些算法，并在实际开发中广泛使用，可以避免很多重复工作，提高编程效率。

使用 STL 算法之前，要包含 STL 算法头文件。如果是算术类算法，应当包含头文件 numeric；如果是其他类型的算法，应当包含头文件 algorithm。

23.5.1 迭代器参数

所有 STL 算法的前两个参数都是一对迭代器，其形参名一般为__first 和__last。这两个迭代器表示了一个前闭后开的区间“[__first, __last)”，表示算法处理的数据范围。__first 指向的是区间中的第一个数据，而__last 指向的则是该区间之后的第一个数据。如果__first 与__last 相等，

则表示该区间为空。

迭代器区间是否有效，取决于是否可以通过迭代器的递增操作（++）由__first 到达__last。如果算法的调用者不能提供一个有效的迭代器区间，则会导致运行时的错误，甚至导致程序崩溃。

例如，查找函数 find 的声明如下：

```
template<typename _InputIterator, typename _Tp>  // 模板参数：迭代器类型、数据类型
    inline _InputIterator                        // 内联函数，返回一个迭代器
    find( _InputIterator __first, _InputIterator __last,  // 两个迭代器参数
        const _Tp& __val );                               // 查找的目标值
```

在算法声明中，迭代器的类型往往暗示了算法对迭代器的最低要求。比如在上述 find 算法的声明中，迭代器的类型名为_InputIterator，这表明要使用 find 算法，使用的迭代器至少应该是输入型迭代器。当然，除此之外，也可以使用功能更强大的前向迭代器、双向迭代器以及随机迭代器。各种迭代器的演化关系如图 23.2 所示。

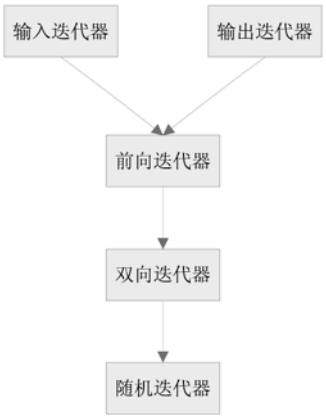


图 23.2　各种迭代器的演化关系

23.5.2 函数对象参数

很多 STL 算法都带有一个函数对象作为参数，该参数用来配置数据的处理方式，或者表示目标数据应当满足的条件。例如在 find_if 算法中就有一个函数对象参数，该参数表示目标数据应当满足的条件。算法 find_if 的声明如下：

```
template<typename _InputIterator, typename _Predicate>       // 模板参数
   inline _InputIterator                                  // 内联函数，返回一个迭代器
   find_if ( _InputIterator __first, _InputIterator __last, // 迭代器参数
         _Predicate __pred );                             // 函数对象参数
```

在上述声明中，第三个函数参数是一个函数对象。其形式类型名是_Predicate，暗示这是一个返回 bool 值的函数对象。在算法的实现中，对迭代器区间中的每一个数据应用该函数对象，如果

返回值为 true，则该数据就是要查找的数据，算法返回一个指向该数据的迭代器。

事实上，算法的函数对象参数也可以用函数指针代替。但是在定义和使用中，函数指针没有函数对象方便，所以一般还是用函数对象。

例如要在一个向量中查找第一个小于 10 的元素，就要使用 find_if 算法。调用该算法之前，首先要建立一个表示小于 10 的函数对象。建立这样一个函数对象，可以使用绑定器 binder2nd 结合函数对象 less，当然也可以使用绑定函数 bind2nd，程序如下：

```
#include <cstdlib>
#include <iostream>
#include <functional>                          // 包含函数对象头文件
#include <algorithm>                           // 包含算法头文件
#include <vector>                              // 包含向量头文件
using namespace std;                           // 使用名称空间 std
int main(int argc, char *argv[])               // 主函数
{
    int ary[5] = { 12, 15, 11, 9, 10 };        // 定义数组
    vector<int> vec(ary, ary+5);               // 用数组构建向量
    vector<int>::iterator iter;                // 定义迭代器变量
    iter = find_if( vec.begin(), vec.end(),    // 使用 find_if 算法 ,查找整个向量
              bind2nd( less<int>(), 10 ));     // 求小于 10 的函数对象
    cout<< *iter <<endl;                       // 输出结果

    system("PAUSE");                           // 暂停
    return EXIT_SUCCESS;                       // 返回
}
```

23.5.3 算法分类

按照迭代器区间内的数据是否会被修改，可以将算法划分为：

- 非质变算法。
- 质变算法。

非质变算法，如查找算法（find，find_if）、计数算法（count，count_if）等，这些算法只是用来对迭代器区间内的数据进行查询和计数，并不修改数据，所以称为非质变算法。质变算法，如复制算法(copy ,copy_n)、替换算法(replace ,replace_if)、交换算法(swap)、删除算法(remove，remove_if ）等，由于这些算法都会对迭代器区间内的数据进行修改，或者将数据复制到别的迭代器区间中，所以称为质变算法。

按照算法的功能，可以将算法分为以下几类。

- 遍历算法：for_each。
- 查找算法 :find ,find_end ,find_first_of ,find_if ,adjacent_find ,binary_search ,count ,

count_if，equal_range，lower_bound，upper_bound，search，search_n。

- 排序算法：sort，stable_sort，partial_sort，partial_sort_copy。
- 整理算法：partition，stable_partition，random_shuffle，reverse，reverse_copy，rotate，rotate_copy。
- 复制算法：copy，copy_n，copy_backward。
- 删除算法：remove，remove_copy，remove_if，remove_copy_if，unique，unique_copy。
- 合并算法：merge，inplace_merge。

在后面的几节中，本书将选择几种比较常用的算法进行讲解，包括遍历算法、查找算法、排序算法、整理算法等。其他的算法读者可以在用到的时候查阅相关材料，或者直接阅读 STL 算法的源代码。在 STL 的源代码中，每个算法都有详细的注释，理解起来并不困难。

23.6 遍历算法

for_each 是 STL 中唯一的一个遍历算法。其参数是一个迭代器区间以及一个函数对象。for_each 的功能是依次遍历该迭代器区间内的每一个数据，并对每一个数据应用参数中的函数对象。其声明如下：

```
template<typename _InputIterator, typename _Function>      // 模板参数
_Function                                                  // 返回一个函数对象(或者函数指针)
for_each( _InputIterator __first, _InputIterator __last,  // 迭代器区间
        _Function __f );                                   // 函数对象(或者函数指针)
```

设计 for_each 算法的主要目的是用来替换开发者手写的循环操作，如 for，while，do…while 等。这些循环操作本身功能比较简单，需要开发者自己处理迭代器的取值、步进。如果没有仔细处理，很容易导致错误。而如果用 for_each 算法，上述操作细节则由 STL 代劳，减轻了开发者的负担。例如，可以利用 for_each 算法输出迭代器区间内的数据，其代码如下：

```
#include <cstdlib>
#include <iostream>
#include <functional>                                      // 包含函数对象头文件
#include <algorithm>                                       // 包含算法头文件
#include <vector>                                          // 包含向量头文件
using namespace std;                                       // 使用名称空间 std
template<typename T>
struct OutPut : public unary_function<T, void> // 定义输出函数对象 OutPut
{
    void operator()( const T &val )                        // 重载函数调用运算符“( )”
    {
        cout<< val << ' ';                                 // 输出数据
    }
};
int main(int argc, char *argv[])                           // 主函数
```

```
{
    int ary[5] = { 12, 15, 11, 9, 10 };        // 定义数组
    vector<int> vec(ary, ary+5);               // 用数组初始化向量
    for_each( vec.begin(), vec.end(), OutPut<int>() );
                                        // 使用 for_each 算法和 OutPut 输出向量

    system("PAUSE");                           // 暂停
    return EXIT_SUCCESS;                       // 返回
}
```

事实上，仅仅是简单的输出，for_each 算法并不比 for 和 while 等循环控制方便。真正能够体现其优点的是：可以通过配置“函数对象参数”更好地控制数据的处理方法，使得程序很容易维护。例如可以为 OutPut 添加一个参数，指明数据间的间隔字符：

```
#include <cstdlib>
#include <iostream>
#include <functional>                          // 包含函数对象头文件
#include <algorithm>                           // 包含算法头文件
#include <vector>                              // 包含向量头文件
using namespace std;                           // 使用名称空间 std
template<typename T>
struct OutPut : public unary_function<T, void> // 定义输出函数对象 OutPut
{
    OutPut(char space) : m_space(space)        // 构造函数，配置间隔字符
    {}
    void operator()( const T &val )            // 重载函数调用运算符“( )”
    {
        cout<< val << m_space;                 // 输出数据
    }
private:
    char m_space;                              // 间隔字符
};
int main(int argc, char *argv[])               // 主函数
{
    int ary[5] = { 12, 15, 11, 9, 10 };        // 定义数组
    vector<int> vec(ary, ary+5);               // 定义并初始化向量
    for_each( vec.begin(), vec.end(), OutPut<int>('+') );
                                               // 以“+”为间隔，输出向量
    cout<<endl;
    for_each( vec.begin(), vec.end(), OutPut<int>('#') );
                                               // 以“#”为间隔，输出向量
    cout<<endl;
    for_each( vec.begin(), vec.end(), OutPut<int>('\t') );
                                               // 以制表符为间隔，输出向量
    cout<<endl;

    system("PAUSE");                           // 暂停
    return EXIT_SUCCESS;                       // 返回
```

```
}
```

在传递给 for_each 的函数对象中，不要试图修改迭代器区间内数据的顺序，否则会使迭代器失效，从而导致不可预知的错误。

23.7 查找算法

查找算法是 STL 算法中的一个大类，包含的算法很多。基本上还可以再细分为四个小类，包括查找单个元素的算法、模式匹配（或者子区间查找）算法、边界查找算法、计数算法。边界查找算法指的是 lower_bound 和 upper_bound，这两个算法在“关联式容器”的内容中已经讲过，这里不再赘述。另外，计数算法也比较简单，这里也不再多说。

23.7.1 查找单个元素

查找单个元素可以使用算法 find 和 find_if。find 和 find_if 基本相同，查找的目标是一个给定的值。find 利用元素本身的“相等”操作（等号运算符“==”）来比较迭代器区间内的数据和目标数据；find_if 则利用一个函数对象参数，提供数据间的比较操作。

find 算法的使用比较简单，只要给出迭代器区间和目标参数即可。但是有一点要注意，迭代器区间内的数据应当可以应用等号运算符“==”。

find_if 参数中的函数对象的作用是用来比较目标数据与迭代器区间内的数据。如果两者相等，则返回 true，否则返回 false。这样的函数对象可以由开发者自己提供，也可以使用适配器 binder1st 或者 binder2nd 适配函数对象 equal_to，并绑定目标数据。例如在一个迭代器区间内查找整数 10，该参数可以写为 bind1st(equal_to <int>(), 10)或者 bind2nd(equal_to<int>(), 10)。

下例利用 find_if 算法，查找学生名册中符合某种条件的学生，例如成绩在 90 分以上的学生，程序如示例代码 23.1 所示。

示例代码 23.1

```
#include <cstdlib>
#include <iostream>
#include <string>                                    // 包含字符串头文件
#include <vector>                                    // 包含向量头文件
#include <algorithm>                                 // 包含算法头文件
using namespace std;                                 // 使用名称空间 std
struct Student                                       // 学生类
{
    Student(int id, string name, int points)         // 构造函数
        : m_id(id), m_name(name), m_points(points) // 初始化学号、名字、成绩
    {}
```

```
    friend ostream &operator<<(ostream& os, const Student &s); // 友元输出函数
    int m_id;                                          // 学号
    string m_name;                                     // 名字
    int m_points;                                      // 成绩
};
ostream &operator<<(ostream& os, const Student &s) // 重载输出运算符“<<”
{
    cout<< s.m_id <<' '<<s.m_name<<' '<<s.m_points;// 输出学生信息
}
struct PointsGE                                        // 成绩大于等于函数对象
{
    PointsGE(int points):m_ptLevel(points)         // 构造函数，初始化成绩
    {}
    bool operator()(Student s)                         // 重载函数调用运算符
    {
        return s.m_points >= m_ptLevel;            // 比较成绩
    }
private:
    int m_ptLevel;                                     // 成绩标准
};
int main(int argc, char *argv[])                       // 主函数
{
    cout<<"———用 find_if 查找 90 分以上的学生———"<<endl;    // 输出提示信息
    vector< Student > stuBook;                         // 学生名册
    stuBook.push_back(Student(0, "a", 60));        // 加入学生
    stuBook.push_back(Student(1, "b", 95));
    stuBook.push_back(Student(2, "c", 84));
    stuBook.push_back(Student(3, "d", 92));
    stuBook.push_back(Student(4, "e", 73));
    stuBook.push_back(Student(5, "f", 97));

    vector< Student >::iterator iter;          // 定义学生名册迭代器
    iter = find_if( stuBook.begin(), stuBook.end(), PointsGE(90) );
                                                       // 第一次查找成绩在 90 以上的学生
    while( iter != stuBook.end() )             // 如果找到
    {
        cout<< *iter <<endl;                   // 输出学生信息
        iter++;                                // 从下一个学生开始
        iter = find_if( iter, stuBook.end(), PointsGE(90) );
                                                       // 继续查找成绩在 90 以上的学生
    }
    system("PAUSE");                           // 暂停
    return EXIT_SUCCESS;                       // 返回
}
```

建立一个控制台工程，并将上述代码复制到源文件中，编译、链接并执行，其结果如图 23.3 所示。

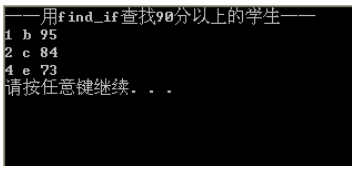


图 23.3　用 find_if 查找学生名册结果

在上述代码中，为了使用 find_if 算法查找目标数据，首先定义了一个函数对象 PointsGE。PointsGE 在构造时接受一个参数，作为比较分数的标准。其重载的函数调用运算符“()”接受一个 Student 对象，并比较该对象的成绩与分数标准。在调用 find_if 算法时，使用 90 构造 PointsGE，则算法 find_if 返回的迭代器就指向成绩在 90 以上的学生。

23.7.2 搜索子区间

搜索子区间，也称做模式匹配，其功能是在目标区间内查找一个子区间，并返回一个指向子区间第一个元素的迭代器。STL 中提供了三个查找子区间的算法，包括 search，search_n，find_end。

search 和 find_end 的声明基本相同，都是接受两对迭代器（标识两个迭代器区间），其重载版本还接受一个二元函数，用来判断数据是否相等。其声明如下：

```
// 使用默认比较方法的 search 和 find_end
  template<typename _ForwardIterator1, typename _ForwardIterator2>
    _ForwardIterator1
   search(_ForwardIterator1 __first1, _ForwardIterator1 __last1,
       _ForwardIterator2 __first2, _ForwardIterator2 __last2);
  template<typename _ForwardIterator1, typename _ForwardIterator2>
    inline _ForwardIterator1
    find_end(_ForwardIterator1 __first1, _ForwardIterator1 __last1,
         _ForwardIterator2 __first2, _ForwardIterator2 __last2);
// 使用函数对象比较数据的 search 和 find_end
  template<typename _ForwardIterator1, typename _ForwardIterator2,
       typename _BinaryPredicate>
    _ForwardIterator1
    search(_ForwardIterator1 __first1, _ForwardIterator1 __last1,
       _ForwardIterator2 __first2, _ForwardIterator2 __last2,
       _BinaryPredicate  __predicate);
  template<typename _ForwardIterator1, typename _ForwardIterator2,
       typename _BinaryPredicate>
    inline _ForwardIterator1
    find_end(_ForwardIterator1 __first1, _ForwardIterator1 __last1,
         _ForwardIterator2 __first2, _ForwardIterator2 __last2,
         _BinaryPredicate __comp);
```

在上述声明中，__first1 和__last1 标识的是目标区间“[__first1, __last1)”，在该区间内查找子区间。__first2 和__last2 标识的是子区间的值区间“[__first2, __last2)”。子区间中的数据应当与该值区间中的数据一一相等。

__first2 和__last2 标识的区间可以是目标区间“[__first1, __last1)”的一部分，也可以是独立的区间。

search 和 find_end 的功能也基本相同，就是在区间“[__first1, __last1)”内查找一个子区间，而子区间的值则在区间“[__first2, __last2)”内指明。如果找到这样一个子区间，则返回一个迭代器，指向该子区间的第一个数据。两个算法不同的是：search 查找的是第一个匹配的子区间，而

find_end 查找的则是最后一个匹配的子区间，如图 23.4 所示。

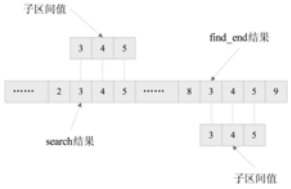


图 23.4 目标区间与子区间

例如，在向量中查找一段子区间，程序如下：

```
#include <cstdlib>
#include <iostream>
#include <algorithm>                                    // 包含算法头文件
#include <vector>                                       // 包含向量头文件
using namespace std;                                    // 使用名称空间 std
int main(int argc, char *argv[])                        // 主函数
{
    int a[] = {0,1,2,3,4,5,6,7,8,3,4,5,9};              // 定义数组
    int len = sizeof(a)/sizeof(a[0]);                   // 求数组长度
    vector<int> vec(a, a+len);                          // 用数组初始化一个向量
    int ary[3] = {3,4,5};                               // 要查找的子区间的值
    vector<int>::iterator iter;                         // 定义向量的迭代器

    iter = search( vec.begin(), vec.end(), ary, ary+3);
                                                        // 在向量中查找第一个匹配的子区间
    cout<< *(iter-1)<<endl;                             // 输出子区间之前数据的值，结果为 2

    iter = find_end( vec.begin(), vec.end(), ary, ary+3);
                                                        // 在向量中查找最后一个匹配的子区间
    cout<< *(iter-1)<<endl;                             // 输出子区间之前数据的值，结果为 8
    system("PAUSE");                                    // 暂停
    return EXIT_SUCCESS;                                // 返回
}
```

算法 search_n 也可以用来搜索子区间，不过其子区间的值比较特殊，是一段相同的数据，而不是用一个迭代器区间标识的。在 search_n 的参数中，用一个参数表示子区间数据的值，另外一个参数表示数据的数量。例如：

```
iter0 = search_n( first_0, last_0, 3, 1 );
                                        // 在[first_0, last_0)中查找 3 个 1 的子区间
iter1 = search_n( first_1, last_1, 4, 'c' );
                                        // 在[first_1, last_1)中查找 4 个 c 的子区间
iter2 = search_n( first_2, last_2, 5, obj );
                                        // 在[first_2, last_2)中查找 5 个 obj 的子区间
```

算法 equal_range 与 search_n 有些类似，不过该算法并不指定要查找的数据个数，而是查找目标区间中数据相等、并且个数最多的子区间。adjacent_find 有点儿像上述两个算法的结合体，不过该算法既不需要指定要查找的数据，也不需要指定数据的个数，其目的是查找目标区间中第一对既相邻又相等的数据。

23.7.3 搜索子区间中的一个值

find_first_of 虽然也可以搜索子区间，但是并不需要匹配整个子区间，只要子区间中有一个值能够在目标区间中找到即可。假设目标区间是{1，2，3，4，5}，而要查找的子区间是{8，9，3}，其中数据 3 可以在目标区间中找到，则算法 find_first_of 返回的迭代器指向目标区间中的 3，如图 23.5 所示。

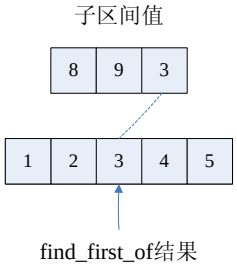


图 23.5　算法 find_first_of

程序如下：

```
#include <cstdlib>
#include <iostream>
#include <algorithm>                          // 包含算法头文件
using namespace std;                          // 使用名称空间 std
nt main(int argc, char *argv[])               // 主函数
{
    int a[] = {1,2,3,4,5};                    // 定义目标数组
    int b[] = {8,9,3};                        // 要查找的子区间
    int *p = find_first_of( a, a+5, b, b+3 );// 在目标数组中，查找子区间中的一个值
    cout<< *p <<endl;                         // 输出查找结果 3
    system("PAUSE");                          // 暂停
    return EXIT_SUCCESS;                      // 返回
}
```

23.7.4　有序区间的查找算法

在 STL 的查找算法中有三个算法比较特殊，即 binary_search，lower_bound 和 upper_bound。这三个算法要求其查找的空间必须是经过排序的，否则不能进行正常的查找，而返回 false 或者一个指向区间尾的迭代器。lower_bound 和 upper_bound 前面已经讲过，这里不再赘述。binary_search 是二分查找算法，其声明如下：

```
template<typename _ForwardIterator, typename _Tp>
  bool                                                    // 返回是否存在
  binary_search(_ForwardIterator __first, _ForwardIterator __last,
   // 查找区间
              const _Tp& __val);                          // 目标值

template< typename _ForwardIterator, typename _Tp,
          typename _Compare>
  bool                                                    // 返回是否存在
  binary_search(_ForwardIterator __first, _ForwardIterator __last,
```

```
    // 查找区间
              const _Tp& __val,                          // 目标值
                _Compare __comp);                        // 用于比较的函数对象
```

binary_search 有一点同大多数查找算法都不同，即其返回值是一个 bool 值，而不是一个迭代器，该返回值表示目标值是否存在于查找区间中。

binary_search 的第一个版本并没有指明其查找区间如何排序。但是在 STL 中，除非特别指明，所有的排序都是由小到大的，所以其参数__first 和__last 标识的区间也必须是从小到大进行排序的。如果不满足该条件，则 binary_search 的返回值就是 false。

binary_search 的第二个版本用一个函数对象指明了排序法则。该函数对象可以是 less，表示区间是从小到大进行排序的；也可以是 greater，表示区间是从大到小进行排序的。注意，该法则必须同区间“[__first，__last)”的排序法则一致，否则其返回值也总是 false。

例如：

```
#include <cstdlib>
#include <iostream>
#include <algorithm>                                     // 包含算法头文件
using namespace std;                                     // 使用名称空间 std
int main(int argc, char *argv[])                         // 主函数
{
    int a[] = {6,5,4,3,2,1};                             // 从大到小排列的数组
    int b[] = {1,2,3,4,5,6};                             // 从小到大排列的数组

    cout<< boolalpha;                                    // 以文字形式输出 bool 值
    cout<<binary_search(a, a+6,3)<<endl;                 // 输出 false
    cout<<binary_search(b, b+6,3)<<endl;                 // true
    cout<<binary_search(a, a+6,3, greater<int>())<<endl;   // true
    cout<<binary_search(b, b+6,3, greater<int>())<<endl;   // false
    system("PAUSE");                                     // 暂停
    return EXIT_SUCCESS;                                 // 返回
}
```

在上述程序中，第 12 行和第 15 行的二分查找的比较算法恰好与数组的顺序相反，所以其结果为 false，而第 13 行和第 14 行的二分查找的比较算法与数组的顺序相同，所以其结果为 true。

23.8 排序算法

STL 的排序算法有 4 个，即 sort，stable_sort，partial_sort 和 partial_sort_copy。这 4 个算法的功能都是将指定区间内的数据进行排序。排序的标准可以是默认的由小到大，也可以由用户指定，例如由大到小排序。

从排序算法的声明中可以看出，传入的迭代器实参必须是随机迭代器。因此，并不是所有容器都可以使用排序算法。例如关联式容器仅支持双向迭代器，链表容器也支持双向迭代器，这两种迭代器都比随机迭代器的功能弱，所有相关的容器也都不支持排序算法。

23.8.1 sort 和 stable_sort

sort 和 stable_sort 是对整个迭代器区间进行排序。这两个算法之间存在差异，这一点从其名称中就可以看出。stable_sort 是一种稳定的排序算法，而 sort 则不能保证稳定。所谓稳定的排序，其含义是指原来相邻并且相等的数据，排序之后其相对位置保持不变。例如在整数序列“{5，4，3(a)，3(b)，6，1，2}”中，3(a)和 3(b)是相邻且相等的两个数据，经过稳定排序，这两个数据的相对位置保持不变，即 3(a)依然保持在 3(b)之前，排序之后的整数序列为“{1，2， 3(a)，3(b)，4，5，6}”。

sort 和 stable_sort 的声明如下：

```
template<typename _RandomAccessIterator>
  inline void
  sort( _RandomAccessIterator __first, _RandomAccessIterator __last);
                                        // 排序区间
template<typename _RandomAccessIterator, typename _Compare>
  inline void
  sort(_RandomAccessIterator __first, _RandomAccessIterator __last,
                                        // 排序区间
       _Compare __comp);                // 比较方法
template<typename _RandomAccessIterator>
  inline void
  stable_sort(_RandomAccessIterator __first, _RandomAccessIterator __last);
                                        // 排序区间
template<typename _RandomAccessIterator, typename _Compare>
 inline void
  stable_sort( _RandomAccessIterator __first, _RandomAccessIterator __last,
                                        // 排序区间
          _Compare __comp);             // 比较方法
```

sort 和 stable_sort 的使用比较简单，这里就不再举例了，请读者自己练习。只是需要注意，迭代器参数的类别必须是随机迭代器，否则不能进行排序。

23.8.2 partial_sort 和 partial_sort_copy

partial_sort 和 partial_sort_copy 的特点是：按照顺序（从小到大或者从大到小）只选取迭代器区间内前面的部分元素，剩余元素的顺序没有定义。partial_sort 排序之后，各数据仍然在原有的迭代器区间内；如果用 partial_sort_copy 进行排序，排好序的数据将被复制到指定的迭代器区间内。至于如何指定排序数据的数量，两个算法是不同的。其声明如下：

```
template<typename _RandomAccessIterator>
   void
```

```
  partial_sort(_RandomAccessIterator __first,           // 区间头
       _RandomAccessIterator __middle,                  // 区间中
       _RandomAccessIterator __last);                   // 区间尾
template<typename _RandomAccessIterator, typename _Compare>
  void
  partial_sort(_RandomAccessIterator __first,           // 区间头
       _RandomAccessIterator __middle,                  // 区间中
       _RandomAccessIterator __last,                    // 区间尾
       _Compare __comp);                                // 排序方法
template<typename _InputIterator, typename _RandomAccessIterator>
  _RandomAccessIterator
  partial_sort_copy(_InputIterator __first, _InputIterator __last,
                                                        // 待排序区间
           _RandomAccessIterator __result_first,        // 排序结果区间
           _RandomAccessIterator __result_last);
template<typename _InputIterator, typename _RandomAccessIterator,
       typename _Compare>
  _RandomAccessIterator
  partial_sort_copy(_InputIterator __first, _InputIterator __last,
                                                        // 待排序区间
           _RandomAccessIterator __result_first,        // 排序结果区间
           _RandomAccessIterator __result_last,
           _Compare __comp);                            // 排序方法
```

partial_sort 是用“[__first, __middle, __last)”指明排序数据的数量的。也就是说在迭代区间“[__first, __last)”中，只对那些在顺序上靠前的“__middle - __first”个元素进行排序，并且将这些元素放在区间“[__first, __middle)”中。其原理如图 23.6 所示。

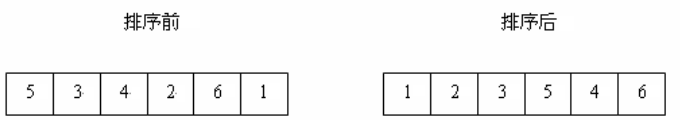


图 23.6 partial_sort 排序原理

partial_sort_copy 排序数据数量的确定略微复杂一些，其数量是“__last - __first”和“__result_first - __result_last)”中的较小值。也就是说如果结果区间足够大（比待排序区间还大），则将全部数据进行排序，否则只选取结果区间中能够容纳的数量。排序结果将被放到结果区间中，其原理如图 23.7 所示。

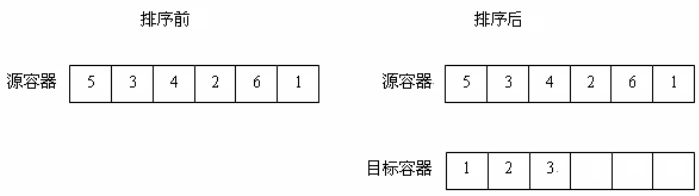


图 23.7 partial_sort_copy 排序原理

请注意 partial_sort 与 sort 的区别。partial_sort 并不是对指定数量的数据进行排序，而是选择那些在顺序上靠前的数据。如果仅仅是对指定数量的数据进行排序，则可以使用 sort 算法，并指定排序的迭代器区间。

23.9 整理算法

整理算法虽然也是用来改变数据的顺序的，但并不会使得数据有序。其功能范围包括分类、随机排列、颠倒和旋转，下面将一一进行说明。

23.9.1 分类

分类算法的目的是按照指定的标准，将迭代器区间内的数据分成两部分。符合标准的数据将被放在迭代器区间的前半部分，不符合标准的数据将被放在迭代器区间的后半部分。两部分的分界线由算法的返回值指明。比如，按照是否为偶数可以将一个整数数组中的数据分为两部分，如图 23.8 所示。

图 23.8 按照是否是偶数进行分类

分类的算法有两个，分别为 partition 和 stable_partition。其区别也在于算法是否稳定，即是否能够保持同类数据之间的相对顺序。算法声明如下：

```
template<typename _ForwardIterator, typename _Predicate>
   inline _ForwardIterator
   partition(_ForwardIterator __first, _ForwardIterator __last, // 待分类区间
         _Predicate   __pred);                                  // 分类标准
template<typename _ForwardIterator, typename _Predicate>
   _ForwardIterator
  stable_partition(_ForwardIterator __first, _ForwardIterator __last,
                                                               // 待分类区间
           _Predicate __pred);                                 // 分类标准
```

在上述分类算法中，表示分类标准的是一个一元函数对象。如果某个数据符合标准，则返回 true，否则就返回 false。例如，用一个判断是否为偶数的函数对象对数组进行分类，其程序如下：

```
#include <cstdlib>
#include <iostream>
#include <algorithm>                          // 包含算法头文件
#include <functional>                         // 包含函数对象头文件
using namespace std;                          // 使用名称空间 std
struct IsEven : public unary_function<int, bool>   // 判断是否是偶数的函数对象
{
```

```
    bool operator()(int val)                // 重载函数调用运算符
    {
        return val%2 == 0;                  // 返回是否是偶数
    }
};
int main(int argc, char *argv[])            // 主函数
{
    int a[] = {1,2,3,4,5,6};                // 定义数组
    partition( a, a+6, IsEven());           // 对数组按照是否是偶数进行分类
    for( int i=0; i<6; i++ )                // 遍历分类后的数组
    {
        cout<<a[i]<<' ';                    // 输出数组
    }
    cout<<endl;                             // 换行
    system("PAUSE");                        // 暂停
    return EXIT_SUCCESS;                    // 返回
}
```

上述程序中使用 partition 算法进行分类，但这并不能保证排序的结果就是{2，4，6，1，3，5}，即同类数据间的相对顺序。如果要做到这一点，可以使用稳定分类算法 stable_partition。

23.9.2 随机排列

随机排列的概念与排序正好相反，其追求的目标不是让数据有序，而是让数据呈现出一种随机的状态，游戏、抽奖等程序恰恰需要这样的随机数据。算法的声明如下：

```
template<typename _RandomAccessIterator>
  inline void
  random_shuffle( _RandomAccessIterator __first,     // 排列区间头
                  _RandomAccessIterator __last );    // 排列区间尾
template<typename _RandomAccessIterator, typename _RandomNumberGenerator>
 void
  random_shuffle( _RandomAccessIterator __first,     // 排列区间头
                  _RandomAccessIterator __last,      // 排列区间尾
                  _RandomNumberGenerator& __rand);   // 随机数发生器
```

在上述两个 random_shuffle 的版本中，第一个版本使用默认的随机数发生器，即函数 rand()，该函数可以获取一个 0～32 767（int 的最大值）之间的任意整数。第二个版本则可以使用用户提供的随机数发生器，要求该发生器接受一个整数 N 作为参数，并返回区间“[0，N)”中的任意一个整数。

由于计算机系统的限制，很难实现真正的随机数，所以这里所谓的随机数通常称为伪随机数。如果不加特殊设置，函数 rand()的返回值也是有规律可循的。所以，为了能够使其返回值尽量随机，可以调用函数 srand()，并以当前时间为基准，设定随机数发生器的起始值。

例如，调用 random_shuffle 算法，随机排列一个整数数组，程序如下：

```
#include <cstdlib>
#include <iostream>.
#include <algorithm>                          // 包含算法头文件
using namespace std;                          // 使用名称空间 std
int main(int argc, char *argv[])              // 主函数
{
    int a[] = {1,2,3,4,5,6};                  // 定义数组
    srand( (unsigned)time( NULL ));           // 设定随机数发生器的基准
    random_shuffle( a, a+6);                  // 随机排列数组
    for( int i=0; i<6; i++)                   // 遍历数组
    {
        cout<<a[i]<<' ';                      // 输出数组
    }
    cout<<endl;
    system("PAUSE");                          // 暂停
    return EXIT_SUCCESS;                      // 返回
}
```

尽管随机排列算法可以得到序列的各种可能排列，但是这样并不能保证每次得到的排列都不重复。如果要取得序列的所有排列方式，并且避免重复，可以使用算法 next_permutation 和 prev_permutation，有关这两个算法的使用请读者参阅相关资料。

23.9.3 颠倒

颠倒算法是将迭代器区间内的数据，以完全对称的方式交换位置。即将长度为 n 的区间中第 i（i 从 0 到 n-1）个元素，与第 n-i-1 个元素进行交换，如图 23.9 所示。

图 23.9　颠倒

颠倒算法有两种形式：一种是在区间上直接进行颠倒，另一种是保持原区间不变，将颠倒的结果输出到别的地方，而复制的目的地则用一个输出迭代器指明。这两种形式的声明如下：

```
  template<typename _BidirectionalIterator>
    inline void
    reverse( _BidirectionalIterator __first,          // 区间头
             _BidirectionalIterator __last);          // 区间尾
  template<typename _BidirectionalIterator, typename _OutputIterator>
    _OutputIterator
    reverse_copy( _BidirectionalIterator __first,     // 区间头
                  _BidirectionalIterator __last,      // 区间尾
                  _OutputIterator __result);          // 输出迭代器
```

例如，颠倒一个数组，并将结果输出到另外一个数组中，程序如下：

```
#include <cstdlib>
```

```
#include <iostream>
#include <algorithm>                                    // 包含算法头文件
using namespace std;                                    // 使用名称空间 std
int main(int argc, char *argv[])                        // 主函数
{
    int a[] = {6,5,4,3,2,1};                            // 定义数组
    int b[6];                                           // 定义目标数组

    reverse_copy( a, a+6, b );                          // 颠倒数组 a，并将结果输出到 b
    for( int i=0; i<6; i++ )                            // 遍历数组 b
    {
        cout<< b[i]<<' ';                               // 输出 b 的元素
    }
    system("PAUSE");                                    // 暂停
    return EXIT_SUCCESS;                                // 返回
}
```

23.9.4 旋转

旋转算法与颠倒算法有点儿类似，都是交换数据的位置。但这两种算法之间存在本质的不同，即“颠倒”是进行完全对称的交换，而旋转则可以指定“中心”,交换中心两侧的数据，如图 23.10 所示。

图 23.10 旋转

从“旋转”的效果来看，其目的是将“中心”两侧的数据进行互换，但并不改变每一侧数据的相对顺序。旋转算法有两种形式，一种是在区间内进行，另外一种是将旋转结果复制到别的地方。其声明如下：

```
  template<typename _ForwardIterator>
    inline void
rotate( _ForwardIterator __first,                       // 区间头
     _ForwardIterator __middle,                         // 旋转中心
     _ForwardIterator __last);                          // 区间尾
  template<typename _ForwardIterator, typename _OutputIterator>
    _OutputIterator
rotate_copy( _ForwardIterator __first,                  // 区间头
         _ForwardIterator __middl                       // 旋转中心,
         _ForwardIterator __last,                       // 区间尾
         _OutputIterator __result);                     // 结果迭代器
```

例如，利用旋转算法，以第三个元素为中心，旋转一个整数数组，并将结果输出到另外一个数组，程序如下：

```
#include <cstdlib>
#include <iostream>
#include <algorithm>                                    // 包含算法头文件
using namespace std;                                    // 使用名称空间 std
int main(int argc, char *argv[])                        // 主函数
{
    int a[] = {1,2,3,4,5,6};                            // 定义数组
    int b[6];                                           // 定义目标数组

    rotate_copy( a, a+2, a+6, b );
                                    // 以第三个元素为中心，旋转数组 a，并将结果输出到 b
```

```
    for( int i=0; i<6; i++ )                    // 遍历数组 b
    {
        cout<< b[i]<<' ';                       // 输出 b 的元素
    }
    system("PAUSE");                            // 暂停
    return EXIT_SUCCESS;                        // 返回
}
```

23.10 小结

本章的主要内容是介绍各种函数对象和算法，以及如何用函数对象来修改算法的计算策略。函数对象一般可以接受一个或两个参数，分别称为一元和二元函数对象。函数对象的功能非常丰富，几乎涉及计算的各个方面，包括算术、关系、逻辑等。另外，还可以使用函数对象适配器，来对函数对象进行调整，产生新的函数对象，进一步丰富函数对象的功能。STL 算法的功能也很丰富，由于使用了模板形式，因此可以看做泛型算法。本章简要介绍了几种常用的算法，至于其他算法，读者可以查阅相关材料或者直接阅读 STL 源代码。

◆ ◆ ◆

第 24 章 名 称 空 间

本章包括

- 为什么要使用名称空间
- 创建名称空间
- 使用名称空间
- 为名称空间创建别名
- 匿名名称空间
- 标准名称空间 std

名称空间是 C++新增加的一种功能，在 C 语言中没有名称空间的概念。在 C++中，不仅各种库函数和类使用名称空间来界定，用户也可以定义自己的名称空间。本章的重点就是讲解名称空间的定义和使用。

24.1 为什么要使用名称空间

名称冲突是 C 和 C++程序中常见的难题。比如在一个程序中要用 Point 这个类来表示二维点，而表示三维点时也要用 Point 这个类名。同样，对于一些函数和全局变量也存在这样的问题。通常，在大规模程序中很容易发生名称冲突，尤其是被分成很多模块的程序。如果开发过程中，对程序中使用的名称没有及时检查，则非常容易导致名称冲突。

例如在下面的程序中，头文件 point2d.h 和 point3d.h 各自定义了一个表示点的类，该程序在编译时会出现编译错误。

```
#include <point2d.h>                // 包含二维点头文件，其中定义了二维点类 Point
#include <point3d.h>                // 包含三维点头文件，其中定义了三维点类 Point
……
int main(int argc, char *argv[])    // 主函数
{
    ……
    Point pt;                       // 编译错误！Point 类名称冲突！
    ……
    return EXIT_SUCCESS;            // 返回
}
```

实际上，只要同时包含这两个头文件就会导致编译错误，因为 C++编译器无法用一个类名同时表示两个不同的类。

为了解决这个问题，C 语言的做法是在名字中加上前缀或后缀，以进行区分。例如，表示二维点的类为 Point2d，表示三维点的类为 Point3d。虽然这样做可以解决一部分问题，但仍然不够完美。比如一个程序要使用其他公司提供的库，而这个库中的类名、函数名、全局变量名也会和本程序中的名字发生冲突。为了解决这个问题，可以为库中的所有名字都加上一个公司名的前缀，例如

ABCPoint2d，DEFPoint3d 等。但这样做往往会使名字变得非常复杂，而且难以阅读。

例如，ABC 公司的 ABCPoint.h 文件代码如下：

```
class ABCPoint2d                        // 二维点类
{
    ……
};
class ABCPoint3d                        // 三维点类
{
    ……
};
```

DEF 公司的 DEFPoint.h 文件代码如下：

```
class DEFPoint2d                        // 二维点类
{
    ……
};
class DEFPoint3d                        // 三维点类
{
    ……
};
```

主程序文件如下：

```
#include <ABCPoint.h>                   // 包含头文件 ABCPoint.h
#include <DEFPoint.h>                   // 包含头文件 DEFPoint.h
……
int main(int argc, char *argv[])        // 主函数
{
    ……
    ABCPoint2d abcPt2d;                 // 定义各种点类的对象
    ABCPoint3d abcPt3d;
    DEFPoint2d defPt2d;
    DEFPoint3d defPt3d;
    ……
    return EXIT_SUCCESS;                // 返回
}
```

在上述程序中，虽然解决了名称冲突的问题。但是由于类名非常复杂，阅读起来非常不便。同样，开发人员使用起来也很不方便。

在编制程序中各种标识符的名字时，应当尽量简洁明了。

为了解决上述问题，C++标准引入了名称空间——namespace。名称空间是一种描述逻辑分组的机制，可以将在逻辑上属于同一个集团的标识符放在同一个名称空间中。一般来讲，名称空间有一个名字（匿名名称空间除外）用来表明本名称空间的逻辑属性，这个名字是唯一的。例如 ABC 公司所提供的程序中，所有标识符都可以置于名称空间 ABC 中，DEF 公司程序中的所有标识符则

可以置于名称空间 DEF 中。ABC 公司的 ABCPoint.h 文件代码如下：

```
namespace ABC
{
    class Point2d
    {
    ……
    }
    class Point3d
    {
    ……
    }
}
```

DEF 公司的 DEFPoint.h 文件代码如下：

```
namespace ABC
{
    class Point2d
    {
    ……
    }
    class Point3d
    {
    ……
    }
}
```

主程序文件代码如下：

```
#include <ABCPoint.h>                    // 包含头文件 ABCPoint.h
#include <DEFPoint.h>                    // 包含头文件 DEFPoint.h
……
int main(int argc, char *argv[])         // 主函数
{
    ……
    return EXIT_SUCCESS;                 // 返回
}
```

在上述主程序中，首先包含了两个头文件。虽然在这两个头文件中都定义了 Point2d 类和 Point3d 类，但是由于这些类分别处在不同的名称空间中，所以不会导致名称冲突，上述程序也不存在编译错误。

在不同的名称空间中，即便有同名的标识符，但因为是在不同的名称空间中，所以也不会引起冲突。

24.2 创建名称空间

创建名称空间之前首先要做好名称空间的划分。一般原则是将在逻辑上属于同一范围的标识符置于同一名称空间中。例如在一个绘图程序中，属于几何构图的部分应当置于名称空间 Geometry

中，属于画面渲染的部分应当置于名称空间 Rendering 中。

24.2.1 创建普通名称空间

创建名称空间要使用关键字 namespace，其后是名称空间的标识符（用来标识该名称空间），然后是由花括号（“{”和“}”）包围的各种声明和定义。例如：

```
namespace Geometry      // 定义一个有关几何元素的名称空间
{
    ……                  // 各种几何元算的声明和定义
}
namespace Rendering     // 定义一个有关图形渲染的名称空间
{
    ……                  // 各种渲染操作的声明和定义
}
```

标识符 Geometry 和 Rendering 唯一地标识了各自的名称空间。一个名称空间的定义可以在程序中出现多次，可以在同一文件中，也可以出现在多个文件中。后出现的名称空间定义不覆盖前面的定义，而是继续前面的定义。例如：

```
////////////////////////////////////////////////////////////////////////
    // Point2d.h   头文件 1
namespace Geometry          // 定义一个名称空间 Geometry
{
    class Point2d           // 名称空间 Geometry 中的类 Point2d
    {
        ……
    };
}
////////////////////////////////////////////////////////////////////////
// Point3d.h   主程序文件
namespace Geometry          // 继续定义名称空间 Geometry
{
    class Point3d           // 名称空间 Geometry 中的类 Point3d
    {
        ……
    };
}
```

在上面例子的两个文件 Point2d.h 和 Point3d.h 中，分别定义了名称空间 Geometry 中的一部分，并各自在名称空间中加入了一个类，即 Point2d 和 Point3d。

尽管引入了名称空间，但在每个名称空间中各个标识符必须保持唯一。只是在各个名称空间之间，标识符可以相同。实际上，由于将标识符置于了名称空间中，标识符实际上是由名称空间名加上原标识符组成的，如 Geometry::Point2d 和 Geometry::Point3d。

即便是在同一文件中，同一名称空间的定义也可以不连续。也就是说，可以先定义名称空间的一部分，再定义名称空间的其他部分。例如，可以先定义名称空间中有关类的部分，再定义名称空间中的函数，如下所示：

```
namespace Geometry              // 创建名称空间
{
    // 定义名称空间中的类
    class Point2d
    {
    ……
    };
    class Point3d
    {
    ……
    };
}                               // 结束名称空间的部分定义
……
namespace Geometry              // 继续定义名称空间
{
    // 定义名称空间中的函数部分
    void DrawLine( const Point2d &start, const Point2d &end );
    void DrawLine( const Point3d &start, const Point3d &end );
}                               // 结束名称空间的部分定义
```

只有在名称空间中声明，才能为名称空间引入一个标识符，在名称空间外部是无法为其加入一个标识符的。例如在名称空间 Geometry 外，语句“int Geometry::newVariable = 0;”就是一种非法的声明，会导致编译错误。

24.2.2 创建嵌套名称空间

在一个名称空间中可以定义另外一个名称空间，即创建嵌套的名称空间。嵌套的名称空间可以在更细的粒度上控制程序的结构。例如，已有一个名称空间 Geometry，其中的标识符（类、全局变量、全局函数等）用于绘制几何图形，图形绘制算法免不了要使用各种矩阵（Matrix），为了更好地组织程序，可以在名称空间 Geometry 中设置一个嵌套的名称空间 Matrix，如下所示：

```
namespace Geometry                                  // 外层名称空间
{
    ……
    namespace Matrix                                // 内层嵌套的名称空间
    {
        class Matrix3                               // 嵌套名称空间中的类
        {
            ……
        };
        Matrix3 MutiplyMatrix( const Matrix3 &m1,   // 嵌套名称空间中的函数
```

```
                        const Matrix3 &m2 );
        ……
    }                                       // 结束内层名称空间 Matrix 的定义
    ……
}                                           // 结束外层名称空间 Geometry 的定义
```

尽管 C++标准并没有对名称空间的嵌套层次做出明确的限制，但是在实际应用中一般不会太多(取决于程序的架构设计)。嵌套层次过多，虽然细化了程序的结构，但是也使得程序的逻辑趋于复杂，反而难以控制。

在程序中，为了引用嵌套名称空间中的名字，需要书写各个层次的名称空间名，中间用域作用符“::”连接。同非嵌套的名称空间一样，可以直接使用嵌套名称空间中的标识符，也可以先使用 using 关键字，引入所需的名称空间，如下所示：

```
int main(int argc, char *argv[])
{
    ……
    Geometry::Matrix::Matrix3 m1, m2, m3;
    ……
    m3 = Geometry::Matrix::MutiplyMatrix( m1, m2 );// 直接使用名称空间中的标识符
    using namespace Geometry:: Matrix;     // 使用 using 关键字引入所需名称空间
    m3 = MutiplyMatrix( m1, m2 );          // 调用引入名称空间中的标识符
    ……
    return EXIT_SUCCESS;
}
```

嵌套名称空间是其外部名称空间的一个子域，其关系类似于语句块与子语句块。例如，外部名称空间中定义的标识符可以在嵌套名称空间中直接访问，而不需要通过域作用符“::”指定外部名称空间名，这有点类似于语句块中的变量可以在子语句块中直接使用。例如：

```
namespace Geometry                          // 外层名称空间
{
    ……
    class Point2d                           // 外部名称空间中定义的类
    {
    public:
        int x;
        int y;
    };
    namespace Matrix                        // 内层嵌套的名称空间
    {
        class Matrix3                       // 嵌套名称空间中的类
        {
        ……
        };
        Matrix3 MutiplyMatrix( const Matrix3 &m1,  // 嵌套名称空间中的函数
                        const Matrix3 &m2 );
```

```
        Poit2d Transform( const Point2d &pt,        // 嵌套名称空间中的函数
                    const Matrix3 &m );
    }                                       // 结束内层名称空间 Matrix 的定义
    ……
                                            // 结束外层名称空间 Geometry 的定义
```

在上面的例子中，类 Point2d 虽然定义在外部名称空间 Geometry 中，但是在内部名称空间 Matrix 中可以直接访问。例如在函数 Transform 的声明中就可以直接使用类名 Point2d，而不需要附加上名称空间前缀“Geometry::”。

另外，如果嵌套名称空间定义的标识符在外部名称空间中已经存在，则在嵌套名称空间中访问到的标识符就是新定义的标识符。假设在上述例子中，Matrix 名称空间中也定义了 Point2d 类，则在函数 Transform 声明中使用的 Point2d 类就是 Matrix 中的，如下所示：

```
namespace Geometry                          // 外层名称空间
{
    ……
    class Point2d                           // 外部名称空间中定义的类
    {
    public:
        int x;
        int y;
    };
    namespace Matrix                        // 内层嵌套的名称空间
    {
        class Matrix3                       // 嵌套名称空间中的类
        {
            ……
        };
        class Point2d                       // 覆盖外部名称空间的 Point2d 类
        {
        public:
            double x;
            double y;
        };
        Matrix3 MutiplyMatrix( const Matrix3 &m1, const Matrix3 &m2 );
                            // 嵌套名称空间中的函数
        Poit2d Transform( const Point2d &pt, const Matrix3 &m );
                            // 嵌套名称空间中的函数，使用 Matrix 中的 Point2d 类
    }                       // 结束内层名称空间 Matrix 的定义

    Point2d Rotate( const Point2d &pt, const Matrix::Matrix3 &m );
                            // 外部名称空间的函数，使用 Geometry 中的 Point2d 类
}
```

在上述例子中，在嵌套名称空间 Matrix 中，内部定义的 Point2d 类覆盖了外部定义的 Point2d 类。但是，一旦脱离了 Matrix，引用的 Point2d 类仍然是名称空间 Geometry 中定义的那个。

24.2.3 定义名称空间的成员

名称空间的定义过程其实就是为名称空间引入名称，包括变量名、函数名、类名等。引入名称的操作包括声明和定义变量、函数和类。变量的声明和定义是同时的，但函数、类的声明和定义则可以分开进行。定义函数和类可以放在名称空间内，也可以放在名称空间外。例如：

```
namespace Geometry                          // 定义名称空间 Geometry
{
    int gNum = 0;                           // 名称空间 Geometry 中的全局变量
    class Point2d                           // 声明类 Point2d
    {
    public:
        void draw();
    };
    class Line2d
    {
        ……
    };
    Line2d drawLine( const Point2d &pt1,    // 声明函数 drawLine
                const Point2d &pt2 );
}
void Geometry::Point2d::draw()              // 定义类 Point2d 的成员函数 draw
{
    ……
}
Geometry:: Line2d                           // 返回一个 Geometry:: Line2d 对象
Geometry::drawLine( const Point2d &pt1,     // 定义函数 drawLine
                const Point2d &pt2 )
{
    ……
    Line2d line( pt1, pt2 );                // 构造 Geometry:: Line2d 对象
    pt1.draw();                             // 调用 Geometry::Point2d 的成员函数
    pt2.draw();                             // 调用 Geometry::Point2d 的成员函数
    line.darw();                            // 调用 Geometry::Line2d 的成员函数
    return line;                            // 返回 Geometry:: Line2d 对象
}
```

在上述例子中，名称空间 Geometry 中的全局函数 drawLine 是在 Geometry 外部定义的。值得注意的是，drawLine 的返回值类型需要指定名称空间，即在 Line2d 之前用“Geometry::”进行修饰；而参数列表和函数体中用到的标识符却不需要这样的修饰，例如 Point2d 和 Line2d 可以直接使用。这是因为编译程序会在 drawLine 所在的名称空间中查找相关的标识符，如果找不到，再到其外部名称空间、全局空间中去查找。

虽然名称空间中的函数和类可以在外部定义，但也不是没有限制。C++标准要求函数和类必须在其外部名称空间或者全局空间中定义，而不可以在其他名称空间（包括嵌套名称空间）中定义。

24.3 使用名称空间

由于函数、类的定义通常放在名称空间之外，所以定义时要用域限定符标明函数、类所属的名称空间。例如对于前面小节中的例子，如果要使用名称空间 Geometry 中的标识符，则需要加上“Geometry::”。如果代码较少，问题还不是很大。如果代码很多，则书写时未免有些麻烦。而且，如果要给名称空间改名，则需要改很多次，很容易因疏忽而遗漏。为了解决这个问题，C++标准引入了关键字“using”，从而可以避免书写名称空间。

24.3.1 使用整个名称空间

使用 using 关键字加 namespace 关键字，然后加一个名称空间的标识符，表示使用整个名称空间。例如常见的使用标准名称空间 std 方法：

```
using namespace std;
```

使用整个名称空间，就是将目标名称空间中的所有名字看做在当前作用域中，访问时可不必用相应的名称空间来限定。例如：

```
namespace Geometry                          // 定义名称空间 Geometry
{
    int gNum = 0;                           // 名称空间 Geometry 中的全局变量
    class Point2d                           // 声明类 Point2d
    {
    public:
        void draw();
    };
    void drawLine( const Point2d &pt1, const Point2d &pt2 );
                                            // 声明函数 drawLine
}
using namespace Geometry;                   // 使用名称空间 Geometry
gNum = 123;                                 // 为 Geometry 中的全局变量 gNum 赋值
void  Point2d::draw()                       // 定义类 Point2d 的成员函数 draw
{
    ……
}
void  drawLine( const Point2d &pt1, const Point2d &pt2 )
                                            // 定义函数 drawLine
{
    ……
}
```

using 命令的作用域从其声明之处开始，直至当前作用域结束。在 using 命令的作用域之外使用名称空间中的名字是非法的。例如：

```
namespace Geometry                          // 定义名称空间 Geometry
{
    int gNum;                           // 名称空间中的变量 gNum
}
```

```
void draw()                          // 名称空间 Geometry 之外的函数
{
    {                                // 语句块开始
        using namespace Geometry;    // 使用名称空间 Geometry
    }                                // 语句块结束，也结束了上述 using 命令的作用域
    gNum = 123;                      // 编译错误，因为 gNum 未定义
}
```

使用 using 关键字引入的标识符只有在使用时才会进行有关二义性的判断。如果仅仅是定义，则并不会导致程序编译错误。例如：

```
#include <cstdlib>
#include <iostream>
namespace Geometry                   // 定义名称空间 Geometry
{
    int gNum;                        // 定义全局变量 gNum
}
using namespace Geometry;            // 使用名称空间 Geometry
int gNum = 0;                        // 定义同名的全局变量 gNum ,此处不会导致编译错误
int main(int argc, char *argv[])     // 主函数
{
    gNum = 1;                        // 编译错误！gNum 存在二义性
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

在上述例子中，虽然在第 8 行引入了 Geometry 中的全局变量 gNum，同时在第 9 行定义了一个重名的全局变量 gNum。但是此时编译程序不进行二义性检查，所以不会导致编译错误。但是在第 12 行使用该全局变量时，由于无法确知 gNum 到底是来自名称空间 Geometry 还是全局空间，所以会导致编译错误。

虽然在上述程序中直接使用全局变量 gNum 会导致编译错误。但是，可以通过域作用符“::”明确使用全局空间中的变量 gNum，而不是名称空间 Geometry 中的变量 gNum，即将第 12 行的语句改为“::gNum = 1;”，则不会再导致编译错误。

24.3.2 使用名称空间中的名字

如果只使用某个名称空间中的一小部分内容，则不应当使用整个名称空间，否则会引入大量的、不必要的名字，容易导致冲突。可以利用 using 关键字引入名称空间中的部分名字，其语法如下：

```
using 名称空间的标识符::名字;
```

例如：

```
using Geometry::gNum;
gNum = 123;
```

在这种情况下，using 将目标名字引入到当前作用域中，而对名称空间中的其他名字则没有影响。如果要使用其他名字，还是应当加上限定符（如“Geometry::”）。

用 using 引入的名字，可以在全局空间中，可以在任意名称空间中，也可以在任何局部域中（如函数、某个用花括号“{”和“}”包围起来的语句块）。其作用域将从声明处开始，直至本作用域结束。如果引入之前，已经有一个同名的标识符，则引入的操作就是错误的，反过来也一样，都将导致标识符的重复定义。例如：

```
// 某个程序的 cpp 文件
namespace Geometry                          // 定义名称空间 Geometry
{
    int g1;                                 // 全局变量 g1
    int g2;                                 // 全局变量 g2
    int g3;                                 // 全局变量 g3
}
using Geometry::g1;                         // 在全局空间中引入 g1
void fun()                                  // 函数 fun
{
    ……
    using Geometry::g2;                     // 在函数空间中引入 g2
    ……
    {// 语句块开始
        ……
        using Geometry::g3;                 // 在语句块中引入 g3
        ……
    } // 语句块结束                         // g3 的作用域结束
    ……
                                            // g2 的作用域结束
……
// 文件结尾                                 // g1 的作用域结束
```

用 using 关键字引入目标名称空间中的所有名字，显然削弱了名称空间的作用，所以应当谨慎使用。一般使用整个名称空间，只适合在一个类的实现文件中，该文件只包含某类的定义，而该类则声明在一个名称空间中。

24.4 为名称空间创建别名

有时候为了准确地描述一个名称空间，不得不使用一个长而且复杂的标识符。但是，这样的的标识符往往由于过长，使用起来很不方便，而且也容易写错。如果可以使用一个较短的标识符来代替长而复杂的标识符，则会方便得多。

C++的名称空间提供了这样的功能，即为一个名称空间定义一个别名。定义的方法如同定义一个变量，而这个“变量”的标识符就是别名，“变量”的值则是名称空间原来的名字，“变量”的类型就是

namespace。其语法如下：

```
namespace 别名 = 名称空间原名;
```

一旦定义了别名，那么以后在使用这个名称空间时，就完全可以使用别名来代替原有的名字。例如，原来要写成“using 名称空间原名”的地方，现在可以写成“using 别名”；原来写成“名称空间原名::成员名”的地方，现在可以写成“别名::成员名”。如此，使用原来名称空间中成员就方便多了。

例如，有一个名称空间，其中包含了一些图形类以及有关图形操作的函数，为了能够准确地说明该名称空间的内容，所以将其命名为 Geometry_Representation_and_Operation。

```
namespace Geometry_Representation_and_Operation
{
    int  gNum;
    ……
}
```

如果真要使用这么长的名称空间，无疑非常烦琐，而且容易写错。因此，可以在使用该名称空间之前，先给这个名称空间起一个较短的别名，如 GRO，以方便使用。

```
namespace GRO = Geometry_Representation_and_Operation;
GRO::gNum = 123;
```

显然，使用名称空间别名后，使得程序代码简洁多了。但是不要将名称空间的别名定义在头文件中，否则所有包含该头文件的源文件都拥有该别名的定义，反而失去了定义别名的意义。可以将名称空间的别名定义在源代码文件里，在所有函数之外，那么该别名将是文件级别的别名，在整个文件内都有效。另外也可以将名称空间的别名定义在某个代码块中，那么该别名将是代码块级别的别名，只对该代码块有效。

例如在下面的程序中，包含了两个文件，一个是定义名称空间的头文件 ns.h，另外一个是主程序文件 main.cpp，在主程序文件中展示了如何定义和使用名称空间的别名。定义名称空间的头文件 ns.h 的代码如下：

```
namespace Geometry_Representation_and_Operation        // 名称空间
{
    int gNum;                                          // 名称空间成员
}
namespace Sound_Representation_and_Operation           // 名称空间
{
    int gCount;                                        // 名称空间成员
}
```

主程序文件 main.cpp 的代码如下：

```
#include <cstdlib>
#include <iostream>.
#include "ns.h"                                  // 包含定义名称空间的头文件
using namespace std;                             // 使用名称空间 std
namespace GRO = Geometry_Representation_and_Operation;
                                                 // 定义文件级别的名称空间别名
void f()
{
    namespace SRO = Sound_Representation_and_Operation;
```

```
                                                    // 定义代码块级别的名称空间别名
    SRO::gCount = 456;                              // 使用名称空间中的成员
    cout<<SRO::gCount<<endl;
}
int main(int argc, char *argv[])
{
    GRO::gNum = 123;                                // 使用文件级别的名称空间别名
    cout<<GRO::gNum<<endl;

    //SRO::gCount = 456;                            // 别名 SRO 未定义
    cout<<endl;
    system("PAUSE");                                // 暂停
    return EXIT_SUCCESS;                            // 返回
}
```

当然，在定义时就直接使用简短的名称，也可以避免长且复杂的名称空间标识符。但简短的名称有两方面的问题：一是不能够明确表达空间的内容，二是容易导致名称空间的名称冲突。如果使用名称空间别名，那么受影响的将只是局部的代码块。

24.5 匿名名称空间

如果在全局变量和函数前面加上 static 关键字，则该变量和函数只能在其定义的文件中使用。但是，这是继承自 C 语言的用法。按照 C++标准，在这种情况下，应当使用匿名名称空间来代替 static 关键字。而所谓的匿名名称空间，就是没有名字的名称空间。

24.5.1 定义匿名名称空间

匿名名称空间没有名字，除此之外，其定义方法跟有名字的名称空间的定义方法是一样的。在关键字 namespace 之后，用花括号“{ }”将需要放置在其中的名称包围起来即可，例如：

```
namespace
{
    int gNum;
}
```

在一个程序中，每个源代码文件都可以定义自己的匿名名称空间。在这些众多的匿名名称空间中，只要保证每个标识符在各自的名称空间中唯一即可，而不需要保证在所有匿名名称空间中都唯一。这是因为一个匿名名称空间只对其所在的文件有效，而对其他文件无效。而且，因为匿名名称空间没有名字，所以在其他源代码文件中也无法引用其他的匿名名称空间。

从编译器实现的角度来讲，匿名名称空间实际上是有名字的，而且各不相同，只是对用户不可见，而只对编译器有效。

例如，下面的程序中有两个源代码文件 A.cpp 和 B.cpp，每个源代码文件都定义了一个匿名名

称空间，每个匿名名称空间中都定义了一个全局的整型变量 gNum，尽管这两个全局变量同名，但是并不会导致编译错误。

```
//////////////////////////////////////////////////////////////////////////
// A.cpp   源文件 1
namespace
{
    int gNum;           // 源文件 A.cpp 中的全局变量
}
void fun()
{
    gNum = 123;         // 使用 A.cpp 中的 gNum
}
//////////////////////////////////////////////////////////////////////////
// B.cpp   源文件 2
namespace
    {
    int gNum;           // 源文件 B.cpp 中的全局变量
}
int main(int argc, char *argv[])   // 主函数
{
    gNum = 456;         // 使用 B.cpp 中的 gNum
    return 0;
}
```

24.5.2 匿名名称空间与 static 的区别

匿名名称空间中的标识符只能在定义该空间的源代码文件中使用，这一点同 static 关键字是相同的。但是有一点，两者是完全不同的。用 static 关键字修饰的全局变量和函数的链接属性是内部链接，而在匿名名称空间中的全局变量和函数则是外部链接。

一般而言，具有外部链接属性的全局变量和函数可以在其他文件中使用，而内部链接的全局变量和函数只能在本文件中使用。但是，由于不能跨文件访问匿名名称空间，所以虽然匿名名称空间中的全局变量和函数是外部链接，仍然不能跨文件访问。

在函数模板和类模板的模板参数中，可以使用非类型参数，即用常量等作为模板参数。但是，非类型的常量必须具有外部链接属性。所以，使用 static 修饰的全局变量和函数不能作为模板的非类型参数。如果匿名名称空间中的全局变量、函数被 static 修饰，则该全局变量、函数仍然是内部链接，不能作为模板的非类型参数。例如，在下面的程序中，只有匿名名称空间中的全局变量才能作为模板的非类型参数。

```
#include <cstdlib>
#include <iostream>
using namespace std;        // 使用名称空间 std
template <char *p>
```

```
struct foo                  // 模板类 foo
{
  void bar(){};
};
static char a ='a';         // 静态全局变量
namespace                   // 匿名名称空间
{
  char b = 'b';             // 匿名名称空间中的全局变量
  static char c = 'c';      // 匿名名称空间中的静态全局变量
  template <int NUM> struct ABC {};// 匿名名称空间中的模板
  void foobar()
  {
    const int x = 123;      // 局部变量，不具有外部链接属性
    // ABC<x>();            // 此语句会导致编译错误
}
}
int main(int argc, char *argv[])   // 主函数
{
    // foo< &a >().bar();   // a 是静态变量，属于内部链接，所以会导致编译错误
       foo< &b >().bar();   // b 是匿名名称空间中的全局变量，属于外部链接，可以编译
    // foo<&c>().bar();     // c 是匿名名称空间中的静态变量，属于内部链接，编译错误
    foobar();               // 调用匿名名称空间中的全局函数

    system("PAUSE");        // 暂停
    return EXIT_SUCCESS;    // 返回
}
```

24.6 标准名称空间 std

C++标准定义了一个名称空间 std，很多标准中规定的函数、类等都声明在这个名称空间中，在本书中也大量使用了这个名称空间。例如：

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

不过，正如前面所说的，这样做其实削弱了名称空间的作用。由于引入了 std 中的全部名字，所以在后面的程序中声明各种标识符时，容易引起冲突。虽然本书中大量使用了这种方法，那只是为了简要地说明例子，而不是鼓励读者这么做。比较好的方法是仅引入所需的名字，例如：

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
```

或者在使用 std 中名字时都加上 std 限定符，例如：

```
std::cout<<"Hello World"<<std::endl
std::cin>>x;
```

如果源文件规模比较小，可以使用上述的第一种方法。如果名称空间名字比较短小，或者使用名称空间别名，则可以使用上述的第二种方法。

24.7 小结

本章的主要内容是介绍如何使用名称空间。名称空间的主要作用是为了防止标识符冲突。将逻辑上属于同一空间的标识符置于同一名称空间中，这样使用时就可以在标识符之前附加上一个前缀“名称空间名::”。创建名称空间时，可以是非连续的，即将一个名称空间分成多次进行定义，而且同一名称空间也可以在不同的文件中进行定义。

如果要引用名称空间中的标识符，可以使用“名称空间::嵌套名称空间::标识符”的方法，也可以通过关键字 using 访问指定名称空间中的全部标识符或者部分标识符。名称空间也可以不指定名字，这样的名称空间就是所谓的匿名名称空间。匿名名称空间是用来替代原来 C 标准的 static 关键字的。与 static 关键字不同的是，匿名名称空间中的标识符可以用做模板的非类型参数。

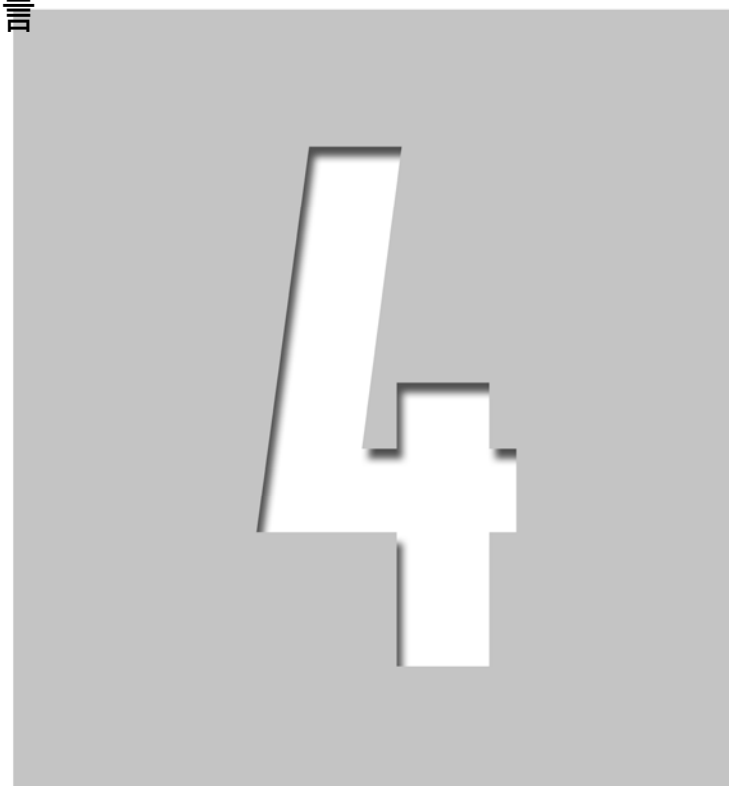
◆ ◆ ◆

Part

第 4 部分　底 层 开 发

第 25 章　位操作

第 26 章　在 C++中嵌入汇编语言



第 25 章 位 操 作

本章包括

- 什么是数据进制
- 数据进制间转换的规则
- 数据存储及数值表式方式
- 位运算的各种运算符
- 位运算的应用
- 位段的定义及应用

通过前面的学习，读者已经掌握了 C++的基本语法。这一章将介绍数值数据和位运算的基本知识。位操作使得 C++语言具有了汇编语言所能完成的一些功能，因此它具有广泛的用途和很强的生命力。数据进制是位运算的基础，因此本章将先介绍数据的表示和编码。

25.1 数据的表示和编码

数制是人们利用符号进行计数的科学方法，它广泛应用于计算机、电子领域里。在计算机中，所有的数据都是按照二进制方式存储的，因此有必要对数据的进制转换以及数据的存储方式有所了解。

25.1.1 数据进制

C++程序中出现的所有数据都必须明确指定其数据类型。数值数据用数制的方式来表示。数制是进位记数制的简称，用数字量表示物理量的大小时，需要用多位数码，我们把多位数码中每一位的构成方法以及从低位到高位的进位规则称为数制，表达每位数码的字符的个数称为基数。数位是数码在一个数中所处的位置。例如，十进位计数制中，每个数位可以使用的数码为 0~9，即基数为十。在计算机中常用的数制有十进制、二进制、八进制和十六进制。

1. 十进制（decimal）

日常生活中常用的是十进制，它有 0，1，2，…，9 十个不同的数码，即基数为 10，而各个数码处于十进制的不同数位时，所代表的数值不同，即对应的权值不同。其运算规律为“逢十进一，借一当十”。其权展开式为 $(N)_{10}=\sum_{i=-m}^{n-1}a_i\times 10^i$。

2. 二进制（binary）

二进制数由两个基本数字 0 和 1 组成，运算规律是“逢二进一，借一当二”。权展开式每位的权是 2 的幂，$(N)_2=\sum_{i=-m}^{n-1}a_i\times 2^i$。例如：

$(10101.01)_2=1\times 2^4+0\times 2^3+1\times 2^2+0\times 2^1+1\times 2^0+0\times 2^{-1}+1\times 2^{-2}$

为了区别于其他进制数，二进制数的书写通常在数的右下方注上基数 2，或在后面加 B 表示。

3. 八进制（Octal）

由于二进制数据的基数较小，所以二进制数据的书写和阅读很不方便，为此，在小型机中引入了八进制。八进制的基数=8=2^3，有数码 0，1，2，3，4，5，6，7，并且每个数码正好对应三位二进制数，所以八进制能很好地反映二进制。按权展开式为 $(N)_8=\sum_{i=-m}^{n-1}a_i\times 8^i$，其运算规律为“逢八进一，借一当八”。

4. 十六进制数（hexadecimal）

十六进制数有两个基本特点：它由十六个字符 0～9 以及 A，B，C，D，E，F 组成（它们分别表示十进制数 10～15），十六进制数的基数=16=2^4，通常在表示时用尾部标志 H 或下标 16 以示区别。按权展开式为 $(N)_{16}=\sum_{i=-m}^{n-1}a_i\times 16^i$，其运算规律为“逢十六进一，借一当十六”。

25.1.2 数据存储

大多数计算机系统的内存储器是由许多“字节”（Byte）单元组成的。字节是计算机信息技术用于计量存储容量和传输容量的一种计量单位，1 个字节由 8 位二进制位组成。例如，在 ASCII 码中，一个英文字母（不分大小写）占一个字节的空间，一个中文汉字占两个字节的空间。计算机系统中，以字节为单位的换算关系如下：

```
1 艾字节（EB，Exabyte）=1024 拍字节（2 的 60 次方字节），即 1EB=1024PB
1 拍字节（PB，Petabyte）=1024 太字节（2 的 50 次方字节），即 1PB=1024TB
1 太字节（TB，Terabyte）=1024 吉字节（2 的 40 次方字节），即 1TB=1024GB
1 吉字节（GB，Gigabyte）=1024 兆字节（2 的 30 次方字节），即 1GB=1024MB
1 兆字节（MB，Megabyte）=1024 千字节（2 的 20 次方字节），即 1MB=1024KB
1 千字节（KB，Kilobyte）=1024 字节（2 的 10 次方字节），即 1KB=1024B
1 字节（Byte）= 8 位（bit）
```

对于数值表示方法，可以采用不同的方法，一般有原码、反码和补码三种。

1. 原码

原码（true form）是一种计算机中对数字的二进制的定点表示方法。原码表示法在数值前面增加了一位符号位（即最高位为符号位），该位为 0 表示正数，该位为 1 表示负数，其余位表示数值的大小。例如，用 8 位表示一个数，则+11 的原码为 00001011，-11 的原码就是 10001011。

原码的优点是简单直观，缺点是不能直接参加运算，可能会在运算中出错。看这个例子：在十进制中 1+（-1）=0，而在二进制中 00000001+10000001=10000010，换算成十进制为 130，显然出错了。

采用原码表示方式，8 位 bit 可以表示的数据范围是-127～+127。所以原码的符号位不能直接参与运算，必须和其他位分开，这就增加了硬件的开销和复杂性。

2. 反码

正整数的反码就是其自身，负整数的反码可以通过对其绝对值逐位求反来求得。在反码表示法中，符号位仍然是 0 表示正号，1 表示负号。例如：

```
[+7]反 = 0 0000111 B
[-7]反 = 1 1111000 B
```

值得注意的是，数 0 的反码也有+0 和-0 两种形式，即：

```
[+0]反 = 00000000B
[-0]反 = 11111111B
```

采用反码表示方式，8 位 bit 可以表示的数据范围是-127～+127。

3. 补码

正数的补码与原码相同，负数的补码符号位为 1，其余位为该数绝对值的原码按位取反，然后整个数加 1。例如：

```
[+9]补 = 00001001
[-7]补 = 11111001
```

采用补码表示方式，8 位 bit 可以表示的数据范围是-128～+127。

使用补码，可以将符号位和其他位统一处理，而且减法也可按加法来处理。另外，两个用补码表示的数相加时，如果最高位（符号位）有进位，则进位被舍弃。出于这些原因，在计算机系统中，数值一律用补码来表示（存储）。

在计算加法时，按如下公式：

```
[X+Y]补 = [X]补 + [Y]补
```

假设 X=+0110011，Y=-0101001，则：

```
[X]补 = 00110011
[Y]补 = 11010111
```

所以，[X+Y]补 = [X]补 + [Y]补 = 00110011+11010111=00001010。

因为计算机中运算器的位长是固定的，上述运算中产生的最高位进位将丢掉，所以结果不是 100001010，而是 00001010。

在计算减法时，按如下公式：

```
[X-Y]补 = [X]补 - [Y]补 = [X]补 + [-Y]补
```

例如计算十进制数 1-1，因为

```
[+1]补 = 00000001
[-1]补 = 11111111
```

所以，[+1-1]补 = 00000001+111111111 = 00000000。

25.2 位运算

所谓位运算是指进行二进制位的运算。在计算机领域，经常需要处理二进制的问题。例如，将一个存储单元中的各二进位左移或右移一位、两个数按位比较等。在某些情况下，位运算可以改善性能，提高程序运行速度。

25.2.1 位运算简介

在很多系统程序中常要求在位（bit）一级进行运算或处理。C++语言提供了位运算的功能，这使得 C++语言也能像汇编语言一样用来编写系统程序。

所谓位运算，是指对整型或字符型数据进行的二进制位的运算。比如将一个存储单元中的各二进制位左移或右移 1 位、两个数按位相加等。具体而言，位运算的主要用途包括：

- 判断一个数据的某位是否为 1。
- 屏蔽掉一个数据中的某些位。
- 清零。
- 保留若干位。
- 把一个数据的某位置为 1。
- 把一个数据的某位翻转，即 1 变为 0，0 变为 1。

表 25.1 给出了 C++语言支持的位运算操作，下面将针对每个位运算进行详细描述。

表 25.1 位运算符及其含义

位运算符	含义	举例
&	按位与	a&b
|	按位或	a|b
∧	按位异或	a∧b
~	按位取反	~a
<<	左移	a<<1
>>	右移	a>>2

25.2.2 按位与“&”

按位与运算符“&”是双目运算符，其功能是将参与运算的两数各对应的二进位相与。只有对应的两个二进位均为 1 时，结果位才为 1，否则为 0。参与运算的数以补码方式出现。

例如，9&5 可写成算式 00001001（9 的二进制补码）& 00000101（5 的二进制补码）= 00000001（1 的二进制补码），可见 9&5=1。测试程序如下：

```
main(){
    int a=9,b=5,c;
    c=a&b;
    printf("a=%d\nb=%d\nc=%d\n",a,b,c);
```

```
}
```

按位与运算通常也可用来对某些位清 0 或保留某些位。例如把 a 的高八位清 0，保留低八位，可做 a&255 运算（255 的二进制数为 0000000011111111）。

25.2.3 按位或“|”

按位或运算符“|”是双目运算符，其功能是将参与运算的两数各对应的二进位相或。只要对应的两个二进位有一个为 1，结果位就为 1。参与运算的两个数均以补码出现。

例如，9|5 可写成算式 00001001 | 00000101 = 00001101（十进制为 13），可见 9|5=13。测试程序如下：

```
main(){
    int a=9,b=5,c;
    c=a|b;
    printf("a=%d\nb=%d\nc=%d\n",a,b,c);
}
```

25.2.4 按位异或“^”

按位异或运算符“^”是双目运算符，其功能是将参与运算的两数各对应的二进位相异或。当两对应的二进位相异时，结果为 1，否则为 0，参与运算数仍以补码出现。

例如，9^5 可写成算式 00001001^00000101 = 00001100（十进制为 12）。测试程序如下：

```
main(){
    int a=9;
    a=a^15;
    printf("a=%d\n",a);
}
```

25.2.5 按位取反“~”

求反运算符“~”为单目运算符，具有右结合性，其功能是对参与运算的数的各二进位按位求反。例如，~9 的运算为~(0000000000001001)，结果为 1111111111110110。

25.2.6 按位左移“<<”

左移运算符“<<”是双目运算符，其功能是把“<<”左边的运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数，高位丢弃，低位补 0。

例如，a<<4 指把 a 的各二进位向左移动 4 位，如 a=00000011（十进制 3），左移 4 位后为 00110000（十进制 48）。测试程序如下：

```
main(){
    char a='a',b='b';
    int p,c;
    p=a;
    p=(p<<8)|b;
    c=p&0xff;
    printf("a=%d\nb=%d\nc=%d\n",a,b,c);
}
```

25.2.7 按位右移“>>”

右移运算符“>>”是双目运算符，其功能是把“>>”左边的运算数的各二进位全部右移若干位，“>>”右边的数指定移动的位数。

例如，设 a=15，a>>2 表示把 000001111 右移为 00000011（十进制 3）。应该说明的是，对于有符号数，在右移时，符号位将随同移动。当为正数时，最高位补 0，而为负数时，符号位为 1，最高位补 0 还是补 1 取决于编译系统的规定，很多系统规定为补 1。测试程序如下：

```
main(){
    unsigned a,b;
    printf("input a number: ");
    scanf("%d",&a);
    b=a>>5;
    b=b&15;
    printf("a=%d\tb=%d\n",a,b);
}
```

针对一位二进制数，表 25.2 给出了位运算的运算结果。

表 25.2 一位二进制数位运算结果

a	b	a&b	a|b	a∧b	~a	~b
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

25.2.8 位赋值运算符

位运算符与赋值运算符结合，可以组成扩展的赋值运算符，如表 25.3 所示。

表 25.3 位赋值运算符及其含义

位赋值运算符	含义	举例	等同于
&=	位与赋值	a&=b	!a
!=	位或赋值	a|=b	a && b
∧=	位异或赋值	a∧=b	a=a∧b
<<=	左移赋值	a<<=b	a=a<<b
>>=	右移赋值	a>>=b	a=a>>b

25.3 位运算的应用

在前面章节里学习的各种运算（如+，-，*等）都是以字节作为最基本位进行的。但在很多系统程序中常要求在位（bit）一级进行运算或处理。位操作使得 C++语言也能像汇编语言一样用来编写系统程序，因此应用十分广泛，是学习 C/C++语言的基础。下面我们对位运算的应用做一个简

介。

25.3.1 设置位

位运算符的常见应用是在变量中使用单个或多个位存储信息。每一个位上的值是二进制的，只有 0 和 1，因此可用来表示某种只有两个状态的信息。例如，可以使用一个位来描述有两个状态的值开或关、男或女、真或假等，由于只需要一个 bit 来存储这些信息，所以极大地节省了开销。

设置位就是指将一个字节中的某一位或者某几位设置为 0 或者 1，该位通常是个标识位。下面具体分析把某一特定位设为 0 或 1 的方法。

1. 设置某一特定位为 0

利用&运算符，使要设置的数 s 与同类型 mask 做&运算。如果第 n 位要清零，则设置 mask 的第 n 位为 0，其他位为 1，然后运算：

```
s=s&mask
```

例如，把整数 5 的第三位清零，因为 $(5)_{10}=(00000101)_2$，要置其第三位为 0，则 mask = $(11111011)_2$，所以 $(5)_{10}$ 的第三位置为 0 的结果为 $(00000101)_2$ & $(11111011)_2$ = $(00000001)_2$。

2. 设置某一特定位为 1

利用|运算符，使要设置的数 s 与同类型 mask 做|运算。如果第 n 位要清零，则设置 mask 的第 n 位为 1，其他位为 0，然后运算：

```
s=s|mask
```

例如，把整数 5 的第二位置为 1，因为 $(5)_{10}=(00000101)_2$，要置其第二位为 1，则 mask = $(00000010)_2$，所以 $(5)_{10}$ 的第二位置为 1 的结果为 $(00000101)_2$ & $(00000010)_2$ = $(00000111)_2$。

3. 设置某几个特定位为 1

利用|运算符，使要设置的数 s 与同类型 mask 做|运算。某几位要设置为 1，则设置 mask 的这几位为 1，其他位为 0，然后运算：

```
s=s&mask
```

例如，把整数 5 的第一、二、三、四位置为 1，因为 $(5)_{10}=(00000101)_2$，要置其前四位为 1，则 mask = $(00001111)_2$，所以 $(5)_{10}$ 的前四位置为 1 的结果为 $(00000101)_2$ | $(00001111)_2$ = $(00001111)_2$。

4. 设置某几个特定位为 0

利用&运算符，使要设置的数 s 与同类型 mask 做&运算。某几位要清零，则设置 mask 对应的几个位为 0，其他位为 1，然后计算：

```
s=s&mask
```

例如，把整数 5 的第三位清零，因为 $(5)_{10}=(00000101)_2$，要置其前三位为 0，则 mask =

$(11111000)_2$，所以 $(5)_{10}$ 的第三位置为 0 的结果为 $(00000101)_2$ & $(11111000)_2$ = $(00000000)_2$。

下面我们来举一些在程序设计中经常用到的例子。

◆ 乘法运算转化成移位运算（在不产生溢出的情况下）：移位运算的特点是速度快，而乘法运算速度较慢，把乘法运算转化为移位运算可以提高程序运行效率。a * (2^n) 等价于 a<< n，代码如下：

```
main(){
     int a = 8;
     printf("a= %d\n  a*2= %d\n a*4= %d\n",a,a<<1,a<<2 );
}
```

运行结果如下：

```
a= 8
a*2= 16
a*4= 32
```

◆ 除法运算转化成移位运算（在不产生溢出的情况下）：a / (2^n) 等价于 a>> n，代码如下：

```
main(){
     int a = 16;
     printf("a= %d\n  a/2= %d\n a/4= %d\n",a,a>>1,a>>2 );
}
```

运行结果如下：

```
a= 16
a/2= 8
a/4= 4
```

25.3.2 取指定位

取指定位是指利用运算符取一个位串信息的某一位或某几位，是位运算符的一种典型应用。下面我们举例进行说明。

1. 利用&和|结合可取出位串中指定的某一位

取 int 型变量 a 的第 k 位（k=1,2,…,sizeof(int)）：先将 a 右移 k-1 位，再与 1 进行按位与，其结果即为变量 a 的第 k 位的值，代码如下：

```
main(){
    int a=9;
    int k = 4;
    printf("a= %d\n  a 的第%d 位为：%d\n",a, k, (a>>(k-1)) & 0x01 );

}
```

运行结果如下：

```
a= 9
a 的第 4 位为： 1
```

2. 判断 int 型变量 a 是奇数还是偶数

a 如果是奇数，则 a 的最后一位是 1，反之为 0。因此可利用只取最后一位来判断是奇数还是

偶数，即把变量 a 与整数 1 相与，代码如下：

```
main(){
    int a=9;
    if((a&1) == 1) //奇数
        printf("a= %d\n  a 为奇数",a);
    else
        printf("a= %d\n  a 为偶数",a);
}
```

运行结果如下：

```
a= 9
a 为奇数
```

3. 判断一个正整数是不是 2 的幂

如果一个正整数 a 是 2 的 n 次幂，那么它一定可以表示成 a = $(\underbrace{00...10...0}_{32})_2$，其中，1 之前有 31-n 个 0，1 的后面有 n 个 0，其他非 2 的幂的整数都不会只有一个 0。那么 a-1 = $(00..011...1)_2$，其中第一个 1 出现之前的 0 有 32-n 个，1 的个数为 n，因此 a&(a - 1) = 0。我们可以利用这个特点来判断一个正整数是不是 2 的幂，代码如下：

```
main(){
    int a = 8;
    if(a&(a-1)==0)
        printf("a= %d\n  a 是 2 的整数次幂 \n",a );
    else
        printf("a= %d\n  a 不是 2 的整数次幂 \n",a );
}
```

运行结果如下：

```
a= 8
a 是 2 的整数次幂
```

25.3.3 特定位取反

特定位取反是指将某一位串中的某一位或几位进行取反的运算。这种操作利用^可以实现，在加密解密算法中经常用到。我们知道，0^1 = 1，0^0 = 0，因此，某一位与 0 做异或运算不会对该位的值有影响；反之，1^1 = 0，1^0 = 1，因此，某一位与 1 做异或运算会对该位取反。利用这个特点，欲对位串中的某一位或某几位取反，则设置 mask 的对应位为 1，其余位为 0。

如欲求整型变量 a 的前四位的反，则设置 mask=$(00001111)_2$，就能求得 a 最右 4 位的信息的反，即原来为 1 的位，结果是 0，原来为 0 的位，结果是 1。

例如，对整数 9 的第三、四位取反，代码如下：

```
main(){
    int a=9;
    int mask = 12;
    printf("a= %d\n  a 的第三、四位取反的结果为：%d\n",a, a&mask );

}
```

运行结果如下：

```
a= 9
a 的第三、四位取反的结果为：5
```

25.4 位段的定义及应用

在对内存资源耗费、运算效率、实时性以及同硬件资源的配合等要求比较高、资源有限的领域，例如嵌入式设备中，经常会用到位段。

25.4.1 位段的定义

位段是把一个字节中的二进位划分为几个不同的区域，并说明每个区域的位数。位段是 C 语言特有的数据结构，它允许我们定义一个由位组成的段，并可为它赋一个名字。位段的定义与结构体定义类似，用 struct 关键字，其形式为：

```
struct 位段结构名
{ 位段列表 };
```

其中位段列表的形式为：

```
类型说明符［位段名］：位段长度
```

对于任一位段，必须存储在同一个字节中，不能跨字节存储。其中，位段名不是必需的，也就是说，可以没有位段名，直接用“：位段长度”这种方式即可。这时这个位段是不可以使用的，这样的位段可以用来调整、填充，以避免跨字节的问题。例如：

```
struct bitseries
{
    int a:4;
    int b:4;
    int c:8;
    };

    struct blankbit
{
    int d:5;
    int :3;
    int e:8;
    };
```

在 bitseries 这个位域定义中，a 占第一字节的 4 位，b 占第一字节的 4 位，c 占第二字节的 8 位。在 blankbit 位段定义中，d 占 5 位，而 e 要占到 8 位，第一字节的位数不够用，所以第一字节里的后 3 位填充，用第二字节来存储 e。

位域变量的声明和 C++里的结构体变量的声明是类似的。例如：

```
struct bitseries1
{
int a:4;
int b:4;
int c:8;
}  data;
```

位段的使用和结构体成员的使用相同，其一般形式为：

```
位段变量名·位域名
```

或者：

```
位段变量指针->位域名
```

例如：

```
main()
{
    struct bitseries
    {
    int a:4;
    int b:4;
    int c:8;
    } bitVari;
    // 赋值
    bitVari.a = 1;
    bitVari.b = 2;
    bitVari.c = 3;
    struct blankbit
    {
    int d:5;
    int :3;
    int e:8;
    } bit, *bitPtr;
    // 赋值
    bitPtr = &bit;
    bitPtr->d = 4;
    bitPtr->e = 5;
    printf("bitVari:a= %d\n b= %d\n c= %d\n bitPtr:d= %d\n e= %d\n", bitVari.a,
bitVari.b,bitVari.c, bitPtr->d,bitPtr->e);
}
```

运行结果如下：

```
bitVari:a= 1
b= 2
c= 3
bitPtr:d= 4
e= 5
```

25.4.2 位段的应用

位段在嵌入式设计中有很广泛的应用，在很多对内存、CPU 的运算速度等有很高的要求，同时又对一些事件的响应速度有很高的要求时经常会用到。有些信息在存储时，并不需要占用一个完整的字节，而只需占一个或几个位。例如在存放一个开关量时，只有 0 和 1 两种状态，用一位二进位即可，那么剩余的 7 个位就会被浪费。每个域有一个域名，允许在程序中按域名进行操作，这样就可以把几个不同的对象用一个字节的二进制位域来表示。

25.5 小结

本章主要介绍了计算机中数据存储和表示的方式。详细地解释了数据进制之间的转换，以及原码、反码和补码的概念。计算机中的数据都是用反码二进制方式储存的，在表示方式上经常用到十六进制，而在现实生活中我们用十进制更多。因此了解这些进制及其表现形式是很有必要的。这是我们理解位运算的基础。

位运算使得C/C++语言也能像汇编语言一样用来编写系统程序，可以进行单独的位操作，在许多对速度要求高，以及需要对硬件进行操作的场合，都离不开位运算。位运算极大地提高了C++语言的灵活性，在位运算的应用这一节里，介绍了很多位运算的应用场合及小技巧。读者若能理解并掌握这些应用，对编写高效简洁的代码会有很大的帮助。

本章最后介绍了位段的概念和应用。位段在嵌入式等对内存、运算效率、实时性以及同硬件资源的配合等方面要求比较高、资源有限的领域，经常会用到。

第 26 章　在 C++中嵌入汇编语言

本章包括

- 什么是汇编语言
- 汇编语言的特点及其应用领域
- 汇编语言的基本语法
- 汇编语言在 C++中的应用

前一章介绍了用 C++进行底层操作时经常用到的位操作，这一章将介绍底层操作必备的汇编语言知识，以及如何在 C++的代码中使用汇编语言。

26.1 汇编语言的基本概念

计算机程序设计语言的发展，经历了从机器语言、汇编语言到高级语言的历程。汇编语言（Assembly Language）是面向机器的程序设计语言。在进行底层开发时，汇编语言是不可或缺的。因此，有必要学习一下汇编语言。

26.1.1 什么是汇编语言

汇编语言是一种功能很强的程序设计语言，也是利用了计算机所有硬件特性并能直接控制硬件的语言。在汇编语言中，用助记符（Memoni）代替操作码，用地址符号（Symbol）或标号（Label）代替地址码。这样用符号代替机器语言的二进制码，就把机器语言变成了汇编语言。

汇编语言比机器语言易于读写、调试和修改，同时也具有机器语言执行速度快、占用内存空间少等优点。但在编写复杂程序时，相对高级语言来说汇编语言代码量较大，而且汇编语言依赖于具体的机型，不能通用，因此不能直接在不同处理机型之间移植。虽然其移植性不好，但效率非常高，针对计算机特定硬件而编制的汇编语言程序，能准确地发挥计算机硬件的功能和特长，程序精炼而质量高，所以汇编语言至今仍是一种常用而强有力的底层开发语言。

26.1.2 汇编语言的特点

汇编语言指令是用一些具有相应含义的助忆符来表达的，所以，它要比机器语言容易掌握和运用。但因为要直接使用 CPU 资源，所以相对高级程序设计语言来说它又显得相对复杂。汇编语言程序归纳起来大概有以下几个主要特点。

- 与硬件相关：汇编语言指令是机器指令的一种符号表示，而不同类型的 CPU 有不同的机器指令系统，也就有不同的汇编语言，所以汇编语言程序与机器有着密切的关系。也就是说，不同型号的 CPU 之间是无法通用相同汇编代码的，因此导致汇编语言的移植性和通用性降低，这是汇编语言天生的缺陷。
- 保持了机器语言的优点，具有直接和简捷的特点：正因为汇编语言有“与机器相关性”的特性，程序员用汇编语言编写程序时，可充分发挥自己的聪明才智，对机器内部的各种资源

进行合理的安排，让它们始终处于最佳的使用状态，这样做的最终效果就是程序的执行代码短，执行速度快，所以，汇编语言是高效的程序设计语言。另外汇编语言可有效地访问、控制计算机的各种硬件设备，如磁盘、存储器、CPU、I/O 端口等，实现资源利用的最大化。

- 编写程序复杂：汇编语言是一种面向机器的语言，其汇编指令与机器指令基本上一一对应，所以，汇编指令也同机器指令一样具有功能单一、具体的特点。要想完成某件工作，就必须安排 CPU 的每步工作。另外，在编写汇编语言程序时，还要考虑具体机型的限制、汇编指令的细节和限制等。
- 经常与高级语言配合使用，应用十分广泛：在某些情况下，比如直接操作 CPU 执行中断以实现线程调度、保存 CPU 寄存器以存储/恢复线程状态等，仅仅使用高级语言是完不成的，需要借助于汇编语言，但是仅使用汇编语言的话，大型程序恐怕需要付出比高级语言几倍的工作量，有时候也是没有必要的。因此，可以在高级语言里嵌入汇编语句，让仅仅一部分需要高效率的代码用汇编语言来完成，其余的框架搭建等用高级语言来完成，这样既保证了效率又降低了代码的复杂程度。这种配合使用在大型软件开发里经常遇到，应用十分广泛，后面会详细讲述其用法。

26.1.3 汇编语言的应用领域

汇编语言是面向机器的低级程序设计语言。它可以直接控制硬件的最下层，如寄存器、标志位、存储单元等，因而能充分发挥机器硬件的性能，具有其他高级语言不可替代的作用。汇编语言是计算机能够提供给用户的最快的、最有效的语言，也是能够利用计算机所有硬件特性并且能够直接控制硬件的唯一语言。也正因为汇编语言具有这些特性，在对于程序的空间和时间要求很高的场合，以及需要直接控制硬件的应用场合，使用汇编语言是必不可少的，它能够完成许多其他高级语言所无法完成的功能。比如 Linux 操作系统的内核，大多数代码是用 C 语言完成的，但是在某些关键的与硬件关系密切的部分，仍不可避免地使用了汇编语言。汇编语言适用的应用领域如下。

- 要求执行效率高、反应快的领域：在操作系统内核、实时系统、智能仪器仪表的控制程序等领域中最主要的要求是效率高和反应快。汇编语言是直接作用于机器硬件的，它所设计的程序具有较小的时间复杂度和空间复杂度。而高级语言是面向问题来设计的，它所设计的程序运行时间长，并且占用存储空间大，在反应和效率上远远不如汇编语言。
- 与硬件资源密切相关的软件开发领域：汇编语言是面向机器的，而高级语言是面向问题的，对于一些与硬件资源密切相关的软件也只能采用汇编语言。如外部设备的底层驱动程序、单片机控制、图像处理软件、加密解密算法等领域。
- 受存储容量限制的应用领域：诸如家用电器的控制领域等，控制系统功能单一、行为简单，且受内存、CPU 等硬件条件限制，需要使用汇编语言进行嵌入式控制。

不过，相比高级语言，汇编语言调试困难，工作量大，其不宜使用的领域有大型软件的整体开发、没有特殊要求的一般应用系统的开发等。

26.2 汇编语言的基本语法

在讲述了汇编语言的基本概念后，有必要对汇编语言的基本语法进行了解，这样才能深入理解汇编语言的概念及其特点，也为后面学习如何在C++的程序里嵌入汇编代码打好基础。本节我们以8086系统为例详细讲解汇编语言的基本语法和用法。

按功能分，8086系统的汇编指令包括数据传送指令、算数指令、逻辑指令、串处理指令、控制转移指令和处理机控制指令6大类，下面逐一进行讲述。

26.2.1 通用数据传送指令

8086系统有4类数据传送指令，以实现CPU的内部寄存器之间、CPU和存储器之间、CPU和I/O端口之间的数据传送。这4类指令是通用数据传送指令、累加器专用传送指令、地址传送指令和标志传送指令。通用数据传送指令中包括最基本的传送MOV(Move)指令、堆栈PUSH(Push onto the stack)和POP (Pop from the stack)指令、数据交换XCHG (Exchange)指令。

1. 最基本的传送指令MOV指令

MOV 指令为双操作数指令，它将一个字节或一个字的操作数从源操作数复制至目的操作数，其语法格式为：

```
MOV  DST,SRC
```

其中，SRC为源操作数，它可以是立即数、寄存器以及各种寻址的内存单元内容。DST为目的操作数，它可以是寄存器或者各种寻址的内存单元，不可以是立即数。

执行的操作：

```
(DST)<-(SRC)
```

对于最基本的传送指令的使用，有几点需要说明：

- 目的数可以是通用寄存器、存储单元和段寄存器（但不允许用CS段寄存器）。
- 立即数不能直接传送至段寄存器。
- 不允许在两个存储单元间直接传送数据。
- 不允许在两个段寄存器间直接传送信息。

以8086为例：

```
MOV   CL , AL            ;AL 中的 8 位数据到 CL
MOV   DS , AX            ;AX 中的 16 位数据到 DS
```

2. 堆栈操作指令

堆栈是按照后进先出（LIFO—Last In First Out）原则组织的一段内存区域。在子程序调用和中断处理时，分别要保存返回地址和断点地址，在进入子程序和中断服务程序后，还通常需要保护通用寄存器原先的内容，子程序返回或中断处理返回主程序前，则要恢复通用寄存器原先的内容，并分别将返回地址或断点地址恢复到指令指针寄存器中。这些功能都要通过堆栈来实现。另外，在高级语言中除中断及子程序调用外，参数的传递也是靠堆栈来实现的。除返回地址和断点地址的保

护及恢复操作是由 CPU 自动完成的以外，寄存器的保存、恢复以及参数的传递都需要由堆栈指令来完成。

堆栈操作指令包括入栈 PUSH 指令和出栈 POP 指令。

PUSH 是入栈指令，完成将源操作数中的一个字推入（也称压入）堆栈的操作。其堆栈指针 SP 的值始终指向刚刚入栈的数据处，每进一个字，栈顶指针 SP 的值减 2。其语法格式为：

```
PUSH SRC
```

其中 SRC 为源操作数，可以是除立即数之外的 16 位的寄存器或者内存字单元的内容（两个字节）。对于入栈 PUSH 指令的使用，需要说明的是：

- 入栈的操作数除不允许用立即数外，可以为通用寄存器、段寄存器（全部）和存储器。
- 入栈时高位字节先入栈，低位字节后入栈。

以 8086 为例：

```
PUSH   AX                ;CPU 通用寄存器入栈
PUSH   CS                ;段寄存器入栈
PUSH   [BX+DI]           ;存储器单元入栈
```

POP 是出栈指令，用于将 SP 所指的堆栈顶处的一个字取出（也称弹出），送至目的 DST 中，并且 SP 的值加 2。其语法格式为：

```
POP DST
```

DST 是目的操作数，可以是除立即数之外的 16 位的寄存器或者内存字单元的内容（两个字节）。对于出栈 POP 指令的使用，需要说明的是：

- 出栈操作数除不允许用立即数和 CS 段寄存器外，可以为通用寄存器、段寄存器和存储器。
- 执行 POP SS 指令后，堆栈区在存储区的位置要改变。
- 执行 POP SP 指令后，栈顶的位置要改变。

以 8086 为例：

```
POP   AX          ;CPU 通用寄存器出栈
POP   CS          ;段寄存器出栈
POP   [BX+DI]     ;存储器单元出栈
```

3. 数据交换指令：XCHG

数据交换指令 XCHG 可在寄存器之间、寄存器与存储器之间进行数据交换，但不允许在两个存储单元之间执行交换过程，并且段寄存器和 IP 寄存器不能参与数据交换。此外，该指令的执行不影响标志位。其语法格式为：

```
XCHG OPR1,OPR2
```

上述数据交换指令实现操作数 OPR1 和 OPR2 中的一个字或一个字节的数据交换。对于数据交换指令的使用，需要说明的是：

- 必须有一个操作数在寄存器中。
- 不能与段寄存器交换数据。
- 存储器与存储器之间不能交换数据。

以 8086 为例：

```
XCHG   AL , BL          ;AL 和 BL 间进行字节交换
XCHG   BX , CX          ;BX 与 CX 间进行字交换
```

26.2.2 累加器专用传送指令

累加器专用传送指令包括 I/O 端口向 CPU 输入信息指令 IN（Input）、CPU 向端口发送信息指令 OUT（Output）以及换码指令 XLAT（Translate）。

在 IBM PC 机里，所有 I/O 端口与 CPU 之间的通信都由 IN 和 OUT 指令来完成。其中 IN 完成从 I/O 到 CPU 的信息传送，而 OUT 完成从 CPU 到 I/O 的信息传送。CPU 只能用累加器（AL 或 AX）接收或发送信息。

1. 输入 IN 指令

输入指令完成将信息从 I/O 通过累加器传送到 CPU 的操作。该指令有长格式和短格式之分。长格式语法格式为：

```
IN AL,PORT(每次输入一字节内容。字节传送)
IN AX,PORT(字传送)
```

执行的操作：

```
(AL)<-(PORT)(字节)
(AX)<-(PORT+1,PORT)(字)
```

短格式语法格式为：

```
IN AL,DX(字节)
IN AX,DX(字)
```

执行的操作：

```
AL<-((DX))(字节)
AX<-((DX)+1,DX)(字)
```

CPU 与外设通信时，输入/输出通信必须经过特殊的端口进行。外部设备最多可有 65 536 个端口，端口号（即外设的端口地址）为 0000～FFFFH。其中前 256 个端口（0～FFH）可以直接在指令中指定，这就是长格式指令中的 PORT，此时机器指令用两个字节表示，第二个字节就是端口号。所以用长格式时可以在指令中直接指定端口号，但只限于外设的前 256 个端口。当端口号大于 255 时，只能使用短格式指令。短格式指令要用寄存器 DX 来间接传送，此时 DX 的内容是端口号，而 IN 指令的作用是将 DX 中指示的端口号内的信息送至 AL 或 AX，OUT 指令的作用是将 AL 或 AX 的内容送至 DX 中的内容所指示的端口中。当所送内容是一个字时，必须使用连续的两个端口号，端口地址（即 DX 中的内容）只能取偶数。

总而言之，IN 和 OUT 指令提供了字和字节两种使用方式，选用哪一种，则取决于外设端口宽度。8088 是准 16 位机，外部数据总线是 8 位，因而只有字节传送指令，而 8086 还有字传送指令。以 8086 为例，将 12 位 A/D 变换器所得数字量输入，这时，A/D 变换器应使用一个字端口，输入

数据的程序为：

```
MOV   DX , 02F0H          ;设该端口为 02F0H
IN    AX , DX
```

2. 输出 OUT 指令

输出 OUT 指令完成将信息从 CPU 通过累加器传送到 I/O 的操作。与输入 IN 指令同样的原因，输出 OUT 指令的语法格式也有长格式和短格式之分。

长格式语法格式为：

```
OUT PORT,AL(字节)
OUT PORT,AX(字)
```

执行的操作：

```
(PORT)<-(AL)(字节)
(PORT+1,PORT)<-(AX)(字)
```

短格式语法格式为：

```
OUT DX,AL(字节)
OUT DX,AX(字)
```

执行的操作：

```
((DX))<-(AL)(字节)
((DX)+1,(DX))<-AX(字)
```

以 8086 为例：

```
OUT   PORT , AL           ;直接的字节输出，PORT 规定与 IN 指令相同
OUT   PORT , AX
OUT   DX , AL             ;间接的字节输出
OUT   DX , AX
MOV   AL , 05H
OUT   27H , AL            ;将字节 05H 传送到地址 27H 的端口
```

3. XLAT 换码指令

换码指令一般用来实现编码之间的转换，又称为查表转换指令，它对于一些无规律代码的转换非常方便。其语法格式为：

```
XLAT [OPR]
```

其中，OPR 为转换表名（即转换表的首地址）。执行的操作：

```
(AL)<-((BX)+(AL))
```

该指令执行时将 BX 寄存器和 AL 寄存器中的内容相加，把得到的值作为地址，然后将此地址所对应的内存单元中的内容送到 AL 中。

在使用这条指令以前，应先建立一个字节表格，将字节表格在内存中的首地址先置于 BX 中，欲查字节距其首地址的偏移值也提前置于 AL 中，表格的内容则是所要换取的代码，该指令执行后就可在 AL 中得到转换后的代码。该指令可用 XLAT OPR 或 XLAT 两种格式中的任一种，使用 XLAT OPR 时，OPR 为表格的首地址（一般为符号地址），但这里的 OPR 只是为提高程序的可读性而设置的，指令执行时只使用预先已存入 BX 中的表格首地址，而不用汇编格式中指定的值。在使用 XLAT 指

令时，应注意：

- 由于 AL 寄存器只有 8 位，所以表格的长度不能超过 256 个字节。
- 该指令的执行不影响标志位。

以 8086 为例，数字 0~9 对应的格雷码依次存放在内存以 TABLE 开始的区域，当#10 端口输入一位十进制数码时，要求 CPU 将之转换为相应的格雷码再输出给该端口，源程序为：

```
MOV   BX , TABLE
IN     AL , 10
XLAT   TABLE
OUT    10 , AL
```

26.2.3 地址传送指令

IBM PC 机的指令系统中，有三条专用于传送地址的指令，即 LEA（Load Effective Address），LDS（Load pointer into register and DS）和 LES（Load pointer into register and ES）。这类指令是用来对寻址机构进行控制的指令，它传送到 16 位模板寄存器中的是存储器操作数的地址，而不是它的内容。

1. LEA 指令

该指令把源操作数的地址偏移量传给某 16 位的通用寄存器，这条指令常用来建立操作所需要的寄存器地址指针。其语法格式为：

```
LEA REG,SRC
```

执行的操作：

```
(REG)<-SRC
```

需要注意的是，SRC 只能是各种寻址方式的存储器操作数，REG 只能是 16 位寄存器。

以 8086 为例：

```
MOV   BX , OFFSET OPER_ONE;等价于 LEA BX , OPER_ONE
MOV   SP , [BX]           ;将 BX 间接寻址的相继的两个存储单元的内容送入 SP 中
LEA   SP , [BX]           ;将 BX 的内容作为存储器有效地址送入 SP 中
```

2. LDS 指令

该指令把一个存放在 4 个存储单元中共计为 32 位的目标指针（段地址和偏移量）传送到两个目的寄存器中。其中后两个字节（高地址）内容，即段地址，送到 DS 中；前两个字节（低地址）内容，即偏移量，送到指令中所出现的目的寄存器中。其语法格式为：

```
LDS REG,SRC
```

执行的操作：

```
(REG)<-(SRC)
(DS)<-(SRC+2)
```

LDS 指令把源操作数指定的 4 个相继字节送到由指令指定的寄存器及 DS 寄存器中。该指令常指定 SI 寄存器。将 SRC 指出的前两个存储单元的内容送入指令中指定的寄存器中，后两个存储单元送入 DS 段寄存器中。

以 8086 为例：

```
LDS   SI , [10H]
;如指令执行前（DS）=C000H,（C0010H）=0180H,（C0012H）=2000H
;则指令执行后（SI）=0180H,（DS）=2000H
```

3. LES 指令

该指令除把目标段地址送至 ES 外，其他与 LDS 相同。其语法格式为：

```
LES REG,SRC
```

执行的操作：

```
(REG)<-(SRC)
(ES)<-(SRC+2)
```

以上三条指令指定的寄存器不能使用段寄存器，且源操作数必须使用除立即数方式及寄存器方式以外的其他寻址方式。这些指令不影响标志位。

本组指令把变量的偏移地址（LEA）或段地址和偏移地址（LDS 和 LES）送给寄存器，以提供访问变量的工具。

26.2.4 标志寄存器传送指令

IBM PC 机的指令系统中提供了四条标志寄存器的传送指令，通过这些指令的执行可以读出当前标志寄存器中的内容，也可以对标志寄存器设置新的值。这四条指令是读取标志指令 LAHF、设置标志指令 SAHF、标志寄存器内容入栈指令 PUSHF 以及标志出栈指令 POPF。

- 读取标志指令 LAHF：该指令将标志寄存器 PSW 中的低 8 位传送到 AH 中对应的位。具体地说，就是将 SF（符号标志）、ZF（零标志）、AF（辅助进位标志）、PF（奇偶标志）和 CF（进位标志）传送到 AH 寄存器的相应位，即 D7，D6，D4，D2 和 D0 位。执行 LAHF 指令后，AH 寄存器的 D5，D3，D1 位没有意义。

  ```
  语法格式：LAHF
  执行操作：(AH)<-(PWS 的低字节)
  ```

- 设置标志指令 SAHF：该指令的操作与 LAHF 相反，SAHF 指令将 AH 寄存器的相应位传送到标志寄存器 PSW 的低 8 位。这条指令不影响 FLAG 的 OF（溢出标志）、DF（方向标志）、IF（中断允许标志）和 TF（跟踪标志）。

  ```
  语法格式：SAHF
  执行操作：(PWS 的低字节)<-(AH)
  ```

- 标志寄存器内容入栈指令 PUSHF：把标志寄存器 F 推入堆栈指针所指的堆栈顶部，同时（SP）- 2→SP。此指令的执行不影响标志位。

  ```
  语法格式：PUSHF
  执行操作：(SP)<-(SP)-2
            ((SP)+1,(SP))<-(PSW)
  ```

- 标志出栈指令 POPF：与 PUSHF 相反，该指令把堆栈指针所指的一个字传送给标志寄存器，同时(SP)+2→SP。这条指令执行后标志寄存器的标志位取决于原堆栈顶部单元的内容。

  ```
  语法格式：POPF
  ```

```
执行操作：(PWS)<-((SP)+1,(SP))
          (SP)<-(SP+2)
```

PUSHF 和 POPF 指令一般用于子程序和中断处理程序的首尾，分别起保存主程序标志和恢复主程序标志的作用。

26.2.5 算术指令

IBM PC 机的算术运算指令包括二进制数运算指令及无符号十进制数运算指令。算术指令用来执行算术运算，它们中有双操作数指令，也有单操作数指令。如前所述，双操作数指令的两个操作数中除源操作数为立即数的情况外，必须有一个操作数在寄存器中。单操作数指令不允许使用立即数方式。算术指令的寻址方式均遵循这一规则。

8086 的算术运算指令包括加、减、乘、除四种基本运算指令。所有算术运算指令都涉及两种类型的数据，即无符号数和有符号数。

无符号二进制数将所有的数位都看成数据位，所以无符号数只有正数没有负数。8 位无符号二进制数表示的无符号十进制数的范围为 0～255，16 位无符号二进制数表示的无符号十进制数的范围为 0～65 535。

有符号二进制数将最高位定义为符号位，而数据本身用补码表示，因此，有符号数既可以表示正数，也可以表示负数。8 位有符号二进制数表示的有符号十进制数的范围为 - 128～ + 127，16 位有符号二进制数表示的有符号十进制数的范围为 - 32 768～ + 32 767。

对加法或减法来说，无符号数和有符号数可以采用同一套指令，对乘法或除法来说，无符号数和有符号数不能采用同一套指令。

无符号数和有符号数采用同一套加法指令及减法指令要满足两个条件。其一，要求参与加法或减法运算的两个操作数必须同为无符号数或有符号数；其二，要求使用不同的方法检测无符号数或有符号数的运算结果是否溢出。

对无符号数进行运算时，只要在执行运算后判断 CF 是否为 1，便可知道结果是否溢出；而对有符号数进行运算时，只要在执行运算后判断 OF 是否为 1，便可知道是否产生了溢出。

所有的算术运算指令，都会影响状态标志位。状态标志位满足下列这些规则：

- 当无符号数运算产生溢出时，CF = 1。
- 当有符号数运算产生溢出时，OF = 1（除法指令除外）。
- 如果运算结果为 0，则 ZF = 1。
- 如果运算结果为负数，则 SF = 1。
- 如果运算结果中有偶数个 1，则 PF = 1。

这里还要指出一点，无符号数运算结果产生溢出只有一种原因，就是超过了最大表示范围，因此溢出也就是有进位。实际当中，在进行多字节无符号数运算时，正是利用溢出来传递低位字节往高位字节的进位的。而有符号数运算产生溢出表示出现了错误，这与无符号数运算产生溢出的情况不同。

1. 加法指令

加法指令包括三个，分别是不带进位的加法指令 ADD、带进位的加法指令 ADC 以及操作数增量指令 INC。

◆ 不带进位的加法指令 ADD：该指令将源操作数和目的操作数中同为一个字节或一个字的数相加，其和送至目的操作数中，且该指令的执行影响标志。DST 可以是寄存器或内存单元的 8 位或者 16 位二进制数，SRC 也可以是寄存器或内存单元的 8 位或 16 位二进制数，还可以是立即数，但 DST 和 SRC 不能同时为存储单元的内容。

```
语法格式： ADD DST,SRC
执行操作：(DST)<-(SRC)+(DST)
```

使用该指令时值得注意的是：

- 两个存储器操作数不能通过 ADD 指令直接相加 ,即 DST 和 SRC 必须有一个是通用寄存器操作数。
- 段寄存器不能作为 SRC 和 DST。
- 影响标志位 Auxiliary Crray Flag（AF）, Carry Flag（CF）, Overflow Flag（OF）, Parity Flag（PF）, Sign Flag（SF）和 Zero Flag（ZF）。

以 8086 为例：

```
ADD   DI , SI              ;SI 和 DI 内容相加，结果放入 DI 中
ADD   AX , [BX+2000H]
          ;BX+2000H 和 BX+2001H 所指的两个存储单元的内容与 AX 相加，结果放在 AX 中
ADD   AL , 3FH             ;立即数 3FH 与 AL 内容相加，结果放入 AL 中
```

◆ 带进位的加法指令 ADC：该指令将源操作数、目的操作数与进位标志 CF 的值相加，和送入目的操作数，其余同 ADD 指令。

```
语法格式：ADC DST,SRC
执行操作：(DST)<-(SRC)+(DST)+CF
```

需要强调的是，该指令在 ADD 指令的基础上还将 CF 的值考虑进来，参与求和运算。

以 8086 为例：

```
ADC   AX , DX              ;AX，DX 与 CF 内容相加，结果放入 AX 中
```

◆ 操作数增量指令 INC：该指令将其后的一个字节或一个字的操作数的值加 1 ，再送回到该操作数中。其中 OPR 既是源操作数，也是目的操作数。这条指令一般用在循环程序中修改指针和循环次数。

```
语法格式： INC OPR
执行操作：(OPR)<-(OPR)+1
```

需要说明的是，OPR 可以是寄存器和存储器操作数，但不能是立即数和段寄存器。影响标志位 OF，SF，ZF，PF 和 AF，不影响 CF。

以 8086 为例：

```
INC   CX                        ;将 CX 的内容加 1 后，送回此单元
```

2. 减法指令

减法指令包括 5 个，他们分别是不考虑借位的减法指令 SUB、带借位的减法指令 SBB、操作数减量指令 DEC、求补指令 NEG 以及比较指令 CMP。

- 不考虑借位的减法指令 SUB：将同为一个字节或一个字的两个操作数相减，用目的操作数减去源操作数，其差存放在目的操作数中，若是有借位发生，则 CF = 1，说明此时有两个无符号数相减溢出；如果作为两个有符号数相减时，OF = 1 说明带符号数的减法溢出，结果是错误的。操作数 DST 和 SRC 同加法指令一样有一定的限制。减法运算不论减数为正还是为负，都要将减数变为补码，然后与被减数相加。

```
语法格式：SUB DST,SRC
执行操作：(DST)<-(DST)-(SRC)
```

对于该指令的使用，需要说明的是，DST 和 SRC 寻址方式及规定与 ADD 相同。

以 8086 为例：

```
SUB   AX , BX              ;AX 中内容减去 BX 中内容，结果放在 AX 中
SUB   AL , 01H             ;AL 中的字节数减去立即数 01H，结果放在 AL 中
SUB   WORD PTR[DI] , 30H ;DI 所指的字单元中的 16 位数减去立即数 30H，结果放在字单元中
```

- 带借位的减法指令 SBB：将同为一个字节或一个字的目的操作数和源操作数相减，再减去进位标志值，其差存放在目的操作数中。其余操作与 SUB 指令相同。和带进位的加法指令类似，SBB 主要用在多字节减法运算中。

```
语法格式： SBB DST,SRC
执行操作：(DST)<-(DST)-(SRC)-CF
```

需要强调的是，该指令与 SUB 指令功能基本相似，区别在于 SBB 在完成两个字节或字相减的同时，还要将较低位字节或字相减时的借位 CF 减去。

以 8086 为例：

```
SBB   AX , 1000H
;相当于 AX 里的内容减去立即数 1000H，减去 CF 里的内容
```

- 操作数减量指令 DEC：该指令将其后的一个字或一个字节的操作数的值减 1，再将结果送回操作数。

```
语法格式： DEC OPR
执行操作：(OPR)<-(OPR)-1
```

需要说明的是，与 INC 类似，OPR 可以为寄存器或存储单元，不能为立即数。除 CF 标志位，其余标志位都受影响。

以 8086 为例：

```
DEC   CX                   ;CX 内容减 1 后，放回 CX 里
DEC   BYTE  PTR[DI+1]      ;将 DI 下一个所指单元内容加 1 后，放回该单元
```

- 求补指令 NEG：NEG 对指令中给出的操作数（一个字节或一个字）取补码，再将结果送回。因为对一个操作数取补码相当于用 0 减去此操作数，所以 NEG 指令执行的也是减法

操作。

```
语法格式：NEG OPR
执行操作：(OPR)<- -(OPR)
```

需要说明的是，NEG 指令执行后，对 OF，SF，ZF，AF，PF，CF 都会产生影响。其中 CF 的内容总会为 1。

- 比较指令 CMP：将同为一个字节或一个字的目的操作数和源操作数相减，但差并不存放到目的操作数中，只是根据结果设置条件标志位 AF，CF，OF，PF，SF 和 ZF。CMP 指令后往往紧跟着一条条件转移指令，根据比较结果产生不同的程序分支。

```
语法格式： CMP OPR1,OPR2
执行操作：(OPR1)-(OPR2)
```

该指令与 SUB 指令一样执行减法操作，但不保存结果，只是根据结果设置标志位 OF，SF，ZF，PF，CF。

3. 乘法指令

进行乘法时，如果两个 8 位数据相乘，那么会得到一个 16 位的乘积。与此类似，如果两个 16 位数据相乘，会得到一个 32 位的乘积。IBM PC 机指令系统规定，两个 8 位数相乘，有一个乘数放在 AL 中，另一个乘数是出现在指令中的 8 位寄存器操作数或内存操作数，乘积在 AX 中；两个 16 位数相乘，有一个乘数必在 AX 中，另一个乘数是出现在指令中的 16 位寄存器操作数或内存操作数，乘积的高 16 位在 DX 中，低 16 位在 AX 中。

乘法指令又分无符号数乘法和有符号数乘法。为了使在两种情况下分别获得正确的结果，IBM PC 机对无符号数和有符号数相乘提供了不同的乘法指令 MUL 及 IMUL。

对于除法运算，也有同样的情况。因此，PC 机的指令系统中也有对无符号数的除法指令和对有符号数的除法指令。

- 无符号数乘法指令 MUL：该指令把预置在 AL（字节）或 AX（字）中的被乘数与源操作数（也应同为字节或字）中的乘数相乘，积存放于 AX（两字节相乘后积为字）中，或 DX，AX（两字相乘后积为双字）中。指令中的源操作数 SRC 可以使用除立即数方式以外的任一种寻址方式。

```
语法格式：MUL SRC
执行操作：字节操作为(AX)<-(AL)*(SRC)
          字操作为  (DX,AX)<-(AX)*(SRC)
```

值得注意的是，目的数必须是累加器 AX 或 AL，指令中不需要写出。源操作数 SRC 可以是通用寄存器和各种寻址方式的存储器操作数，而不允许是立即数或段寄存器。

以 8086 为例：

```
MUL   BL              ;AL 中和 BL 中的 8 位数相乘，乘积放在 AX 中
MUL   CX              ;AX 中和 CX 中 16 位数相乘，结果放在 DX 和 AX 中
MUL   BYTE PTR[DI]    ;AL 中和 DI 所指的字节单元中的 8 位数相乘，乘积放在 AX 中
```

◆ 有符号数的乘法指令 IMUL：对于有符号数的乘法，需要使用相应的有符号数乘法指令 IMUL。该指令的操作与 MUL 类似。

```
语法格式：IMUL SRC
执行操作：字节操作为(AX)<-(AL)*(SRC)
          字操作为  (DX,AX)<-(AX)*(SRC)
```

该指令执行的操作与 MUL 相同，但 IMUL 必须是带符号数，而 MUL 是无符号数。

以 8086 为例：

```
IMUL   BL          ;AL 中和 BL 中的 8 位数相乘，乘积放在 AX 中
IMUL   AX          ;AX 中的数取平方，结果放在 DX 中
```

4. 除法指令

8086 PC 机执行除法运算时，规定除数必须为被除数的一半字长，即被除数为 16 位时，除数为 8 位，被除数为 32 位时，除数为 16 位。指令格式中给出除数的长度和形式，计算机根据给定的除数为 8 位还是 16 位来确定被除数为 16 位还是 32 位。

根据数据类型分，除法指令可分为 4 种，分别是无符号数的除法指令 DIV、有符号数的除法指令 IDIV、将字节扩展成字的指令 CBW 以及将字扩展成双字的指令 CWD。

◆ 无符号数的除法指令 DIV：该指令完成无符号数的除法。

```
语法格式：            DIV SRC
执行操作：字节操作为(AL)<-(AX)/(SRC)的商
                    (AH)<-(AX)/(SRC)的余数
          字操作为  (AX)<-(DX,AX)/(SRC)的商
                    (AX)<-(DX,AX)/(SRC)的余数
```

以字节方式操作时，16 位被除数在 AX 中，8 位除数为源操作数，结果的商在 AL 中，8 位余数在 AH 中。而以字方式操作时，32 位被除数在 DX，AX 中，其中 DX 为高位字，16 位除数为源操作数，结果的 16 位商在 AX 中，16 位余数在 DX 中。

在使用该指令时，存储器操作数必须指明数据类型 BYTE PTR src 或 WORD PTR src。以 8086 为例：

```
DIV   CL              ;AX 中的数除以 CL 中的数，所得的商放在 AL 中，余数放在 AH 中
DIV   WORD PTR[DI]
     ;DX 和 AX 中的 32 位数被 DI 和 DI+1 中所指的 16 位数除，商放在 AX 中，余数放在 DX 中
```

◆ 有符号数的除法指令 IDIV：该指令实现有符号数的除法。

```
语法格式：            DIV SRC
执行操作：字节操作为(AL)<-(AX)/(SRC)的商
                    (AH)<-(AX)/(SRC)的余数
          字操作为  (AX)<-(DX,AX)/(SRC)的商
                    (AX)<-(DX,AX)/(SRC)的余数
```

该指令与 DIV 相同，但操作数必须是带符号数，商和余数也均为带符号数，且余数的符号

与被除数的符号相同。

以 8086 为例：

```
IDIV   CX
       ;将 DX 和 AX 中的 32 位数除以 CX 中的 16 位数，所得的商放在 AX 中，余数放在 DX 中
```

◆ 将字节扩展成字的指令 CBW :该指令把 AL 中的一个字节的值变成一个字的值 ,但值不变。一个带符号数如要增加位数 ,只要将符号位向高位方向延伸即可 ,其数值不变。因此 ,CBW 也可以是符号延伸指令。

```
语法格式：CBW
执行操作：AL 的内容符号扩展到 AH
```

如果(AL)的最高有效位为 0 ,则(AH)=00 ;如(AL)的最高有效位为 1 ,则(AH)=0FFH。

◆ 将字转换为双字的指令 CWD：该指令把 AX 中的一个字变成（DX，AX）中的双字，其值不变。遇到两个字相除时，要预先执行 CWD 指令将 AX 中的被除数扩展成双字，即把 AX 中的符号扩展到 DX 中。

```
语法格式：CWD
执行操作：AX 的内容符号扩展到 DX
```

如(AX)的最高有效位为 0，则(DX)=0；否则，(DX)=0FFFFH。

CBW 和 CWD 指令都不影响条件码。

26.2.6 逻辑指令

逻辑指令在这里包含逻辑运算指令和逻辑移位指令。这些逻辑操作在概念上与上一章的位运算一致，只是在汇编中引入了新的操作符。下面对汇编中涉及的这些逻辑指令进行讲解。

1. 逻辑运算指令

IBM PC 机的逻辑运算指令包括 AND（与）、OR（或）、NOT（非）、XOR（异或）和 TEST（测试）指令。逻辑运算指令可以对字或字节执行逻辑运算。由于逻辑运算是按位操作的，因此操作数应是位串而不是数。

◆ 逻辑与指令 AND：将源操作数和目的操作数中同为一个字或一个字节的相应各位按位相与，结果放在目的操作数中。该运算经常用来屏蔽某些指定位（即置 0 ）或保留某些位。

```
语法格式： AND DST,SRC
执行操作：(DST)<-(DST)^(SRC)
```

AND 指令执行后，将使 CF=0，OF=0，AF 位无定义，指令执行结果影响 SF，ZF 和 PF 标志位。以 8086 为例，屏蔽 AL 的高 4 位，即将高 4 位和 0000 B 相与，低 4 位和 1111 B 相与，

代码如下：

```
MOV   AL , 39H           ;AL= 0011 1001B[39H]
ADD   AL , 0FH
;AL= 0000 1001B[09H]，即 0011 1001B[39H] & 0000 1111B[0FH] = 0000 1001B[09H]
```

- 逻辑或指令 OR :将源操作数和目的操作数中同为一个字或一个字节的相应各位按位相或，结果放在目的操作数中。此运算经常用来对某操作数的指定位“置位”(即将该位上的值设置为 1)。

```
语法格式：OR DST,SRC
执行操作：(DST)<-(DST)V(SRC)
```

OR 指令执行后，将使 CF=0，OF=0，AF 位无定义，指令执行结果影响 SF，ZF 和 PF 标志位。以 8086 为例，将 AL 的第 5 位置 1：

```
MOV   AL , 4AH           ;AL=0100 1010B[4AH]
OR    AL , 10H
;AL=0101 1010B[5AH]，即 0100 1010B[4AH] | 0001 0000B[10H] =0101 1010B [5AH]
```

- 逻辑非指令 NOT：将操作数 (可为字或字节) 中各位求反 (即 0 变 1，1 变 0)，该操作数 (可为寄存器或存储器) 兼作为源操作数和目的操作数，该指令不影响任何标志位。

```
语法格式： NOT OPR
执行操作：(OPR)<-(OPR)
```

操作数不能使用立即数或段寄存器操作数，可使用通用寄存器和各种方式寻址的存储器操作数。以 8086 为例，将 AL 各位取反：

```
MOV   AL , 65H       ;AL=0110 0101B[65H]
NOT   AL          ;AL=1001 1010B[9AH]，即 ~ 0110 0101B[65H]=1001 1010B[9AH]
```

- 异或指令 XOR :将源操作数和目的操作数中同为一个字或一个字节的相应各位按位相异或，结果放在目的操作数中。此运算可以使某些操作数的若干位保持不变，另外若干位取反。

```
语法格式： XOR DST,SRC
执行操作：(DST)<-(DST)V(SRC)
```

以 8086 为例，将 AL 的高 4 位维持不变，低 4 位取反：

```
MOV   AL, B8H        ;AL=1011 1000B[B8H]
XOR   AL , 0FH
;AL=1011 0111B[B7H] 即 1011 1000B[B8H] ^ 0000 1111[0FH]=1011 0111B[B7H]
;测试某一个操作数是否与另一确定操作数相等
XOR   AX , 042EH
JZ .... //如果 AX==042EH， 则 ZF=TRUE(1)， 执行 JZ。
```

- 测试指令 TEST：该指令与 AND 类似，区别是不将结果送至目的操作数，只根据其特征置条件码。

```
语法格式：TEST OPR1,OPR2
执行操作：(DST)^(SRC)
```

需要说明的是，两个操作数相与的结果不保存，结果影响标志位 PF，SF 和 ZF，使 CF=0，

OF=0，而 AF 位无定义。

TEST 指令常用于在不改变原有操作数的情况下，检测某一位或某几位的条件是否满足。只要令用来测试的操作数对应检测位为 1，其余位为 0，相与后判断零标志 ZF 值的真假即可。以 8086 为例：

```
TEST AL , 0000 00001B
```

这段代码的含义是：如果 AL 的最低位为 1，则运算结果必为 0，因而影响 ZF，使其置 1，故只要检查零标志位是否为 1 ，就可以判断 AL 中的第三、第五位的情况，且又不改变 AL 中的内容。

2. 移位指令

移位指令可使(字或字节)操作数中各位上的值在它依存的寄存器或存储器中进行移动。包括逻辑左移指令 SHL、算术左移指令 SAL、逻辑右移指令 SHR、算术右移指令 SAR、循环左移指令 ROL、循环右移指令 ROR、带进位循环左移指令 RCL 以及带进位循环右移指令 RCR。所有的移位指令都影响标志位 CF，OF，PF，SF 和 ZF。

- 逻辑左移指令 SHL :该指令将目的操作数 OPR 的每一位都同时左移由“移位次数 CNT”所指出的若干位数，每左移的一位都进入 CF 标志位，左移时右边空出的位置置 0。

```
语法格式：SHL OPR,CNT
执行操作：使 OPR 左移 CNT 位，并使最低 CNT 位全为 0
```

其中 OPR 可以是除立即数以外的任何寻址方式。移位次数由 CNT 决定，CNT 可以是 1 或 CL。CNT 为 1 时只移一位，如需要移位的次数大于 1，则可以在该移位指令前把移位次数置于 CL 寄存器中，而移位指令中的 CNT 写为 CL 即可。

- 算术左移指令 SAL：该指令的操作与 SHL 完全相同。

```
语法格式：SAL OPR , CNT
执行操作：使 OPR 左移 CNT 位，并使最低 CNT 位全为 0
```

- 逻辑右移指令 SHR：该指令把操作数看成无符号数，逻辑右移一位相当于除以 2，移到 CY 中的数作为小数。

```
语法格式：SHR OPR , CNT //Byte/Word
执行操作：同 SHL，每次将 OPR 的最低位 D0 移出并移到 CF。最高位补 0
```

- 算术右移指令 SAR：该指令将目的操作数 OPR 中的每一位右移由“移位次数 CNT”所指出的若干位数，移位时最高位参加右移的同时，却不改变其值，即如原来是 0 则仍为 0，原来是 1 则仍为 1，最后移出的位送入 CF 中。

```
语法格式：SAR OPR , CNT //Byte/Word
执行操作：移位时最高位参加右移的同时仍保持其值不变，即如原来是 0 则仍为 0，原来是 1 则仍为 1，最后移出的位送入 CF 中
```

算术右移指令对有符号数来讲，右移一位相当于除以 2，移到 CY 中的数作为小数处理。

- 循环左移指令 ROL：该指令将目的操作数 OPR 中的各位同时左移由“移位次数 CNT”所指

出的若干次数，每移动一次，最高位上的值就移至 CF，且右移到右边的最后一位上。

语法格式：ROL OPR , CNT //Byte/Word

执行操作：每次移位时，最高位移出并同时移到 CF 和最低位 D0

- 循环右移指令 ROR：该指令类似于 ROL，只是方向不同，每移动一次，最低位上的值就移向 CF，右移至最高位上。

语法格式：ROR OPR,CNT //Byte/Word

执行操作：每次移位时，最低位 D0 移出并同时移到 CF 和最高位，带进位循环移位指令

- 带进位循环左移指令 RCL：该指令类似于 ROL，但 CF 中的值与目的操作数 OPR 中的 8 位值联合起来形成一个有 9 位长的二进制串进行循环左移。

语法格式：RCL OPR,CNT

执行操作：每次移位时相当于 CF 中的值与目的操作数 OPR 中的 8 位值联合起来形成一个有 9 位长的二进制串进行循环左移

- 带进位循环右移指令 RCR：该指令与 RCL 相似，只是方向相反。

语法格式：RCR OPR ,CNT //Byte/Word

执行操作：每次移位时相当于 CF 中的值与目的操作数 OPR 中的 8 位值联合起来形成一个有 9 位长的二进制串进行循环右移

可以看出，这 8 种指令可以分为两组：前 4 种为移位指令，后 4 种为循环移位指令。循环移位指令可以改变操作数中所有位的位置，在程序中是很有用的。移位指令则常常用来完成乘以 2 或除以 2 的操作。实际上，十进制数中左移 1 位值扩大 10 倍，右移 1 位则值缩小 10 倍。对于二进制来说，当然左移一位就扩大 2 倍，右移一位则缩小 2 倍。其中算术移位指令适用于带符号数运算，SAL 用来乘 2，SAR 用来除以 2；而逻辑移位指令则适用于无符号数运算，SHL 用来乘 2，SHR 用来除以 2。以 8086 为例：

```
MOV  CL,5
SAR  BYTE PTR[DI],CL
;如指令执行前：(DS)=F800H,(DI)=18A0H,(F980AH)=0064H
;则指令执行后：(F980AH)=0003H，CF=0
;相当于 100d/32d=3d
MOV  CL,2
SHL  SI,CL
;如指令执行前：(SI)=1450H
;则指令执行后：(SI)=5140H，CF=0
;相当于 5200d×4d=20800d
```

26.2.7 控制转移指令

程序执行时，一般来说是顺序往下逐条执行的，但有时会改变这种顺序，如循环时会反复执行一段指令；又如当需要按不同情况分别执行不同程序段时，会往前或往后跳过一些指令段，去执行所需要的指令；有时侯甚至暂时不执行本程序，而去执行另一程序段，当执行完后又会返回原程序处，再继续执行下去。总的来说，执行程序的流程会中途有所改变，这里将要介绍的控制转移指令

就是用来控制程序的执行流程的。汇编语言中转移指令分为无条件转移指令和条件转移指令。

1. 无条件转移指令

无条件转移指令（JMP）将程序流程转移到由目的地址说明所指示的存储器地址处，并且从那里开始继续执行程序。执行时 CPU 将从目标地址说明中取出转移目标的段地址送 CS，偏移地址送指令指针 IP。然后从 IP 指向的地址中取出指令并往下执行。可以看出 JMP 指令必须指定转移的目标地址（或转向地址）。

```
语法格式：JMP  目标地址说明
```

事实上，转移可以分成两类：段内转移和段间转移。段内转移是指在同一段的范围之内进行转移，此时只需改变 IP 寄存器的内容，即用新的转移目标地址代替原有的 IP 的值就可达到转移的目的。段间转移则是要转到另一个段去执行程序，此时不仅要修改 IP 寄存器的内容，还需要修改 CS 寄存器的内容才能达到目的，因此，此时的转移目标地址应由新的段地址和偏移地址两部分组成。下面简要给出各种转移情况下的指令格式。

- 段内直接短转移，应用如下：

```
语法格式：JMP SHORT OPR
执行操作：(IP)<-(IP)+8 位位移量
```

- 段内直接近转移，应用如下：

```
语法格式：JMP NEAR PTR OPR
执行操作：(IP)<-(IP)+16 位位移量
```

- 段内间接转移，应用如下：

```
语法格式：JMP WORD PTR OPR
执行操作：(IP)<-(EA)
```

- 段间直接（远）转移，应用如下：

```
语法格式：JMP FAR PTR OPR
执行操作：(IP)<-OPR 的段内偏移地址
          (CS)<-OPR 所在段的段地址
```

- 段间间接转移，应用如下：

```
语法格式：JMP DWORD PTR OPR
执行操作：(IP)<-(EA)
          (CS)<-(EA+2)
```

使用时需要注意的是：

- 无条件转移到指定的地址去执行从该地址开始的指令。
- 段内转移是指在同一代码段的范围内进行转移，只需改变 IP 寄存器的内容。
- 段间转移则要转移到另一个代码段执行程序，此时要改变 IP 寄存器和 CS 段寄存器的内容。

2. 条件转移指令

条件转移指令在改变程序时，需要满足一定的条件，否则将不发生转移，而是继续执行下一条指令。转移是否发生的条件主要由标志寄存器的某些标志位的值来决定，而使标志位发生变化的指

令都能作为条件转移指令的前置指令。每一种条件转移指令都有它的测试条件，满足测试条件则转移到由指令指出的转向地址去执行那里的程序，如不满足条件则顺序执行下一条指令。

以 8086 为例：

```
test ax,0010 0011B  ;测试 AX 中的数的第 0，1，5 位上的数是否为零
JZ   exit
;如果为零，则 ZF = 1，而 JZ 指令正是当 ZF=1 时，转移到标号 EXIT 处去继续执行程序的条件
mov bx,0
……
exit:  ret           ;转移到标号为 EXIT 处
```

上述第一条指令为转移前置指令，第二条指令为条件转移指令，转移条件是上一条指令的运算结果为零，即 ZF=1，否则并不跳转而继续执行下一条指令。

所有的条件转移指令都是段内直接短转移方式，当要求较远的条件转移时可以用条件转移指令转移到附近某处，再在该处用一条无条件转移指令转移到目的地址。

各种不同条件转移的构造和格式如下：

```
J**  LABEL 或 J**OPR   ;其中**表示某种条件
```

条件中，N（Not）表示否定，E（Equal）表示等于，G（Greater）表示带符号数比较时的大于，L（Less）表示带符号数比较时的小于，A（Above）表示无符号数比较时的大于，B（Below）表示无符号数比较时的小于，C（CarryFlag）是进位标志置位，CF = 1，S（SignFlag）是符号标志置位，SF = 1，P（ParityFlag）是奇偶标志置位，PF = 1，O（OverflowFlag）是溢出标志置位，OF = 1。这些条件是可以根据实际情况进行组合的。

（1）根据单个条件标志的设置情况转移。这种条件转移一般适用于测试某一次运算的结果，并根据其不同特征产生程序分支做出不同处理的情况。如：

- JZ 或 JE（Jump if zero,or equal）

  功能说明：ZF = 1 转移，即结果为零（或相等）则转移。
- JNZ 或 JNE（Jump if not zero,or not equal）

  功能说明：ZF = 0 转移，即结果不为零（或不相等）则转移。
- JS（Jump if sign）

  功能说明：SF = 1 转移，即结果为负数则转移。
- JNS（Jump if not sign）

  功能说明：SF = 0 转移，即结果为正数则转移。
- JO（Jump if overflow）

  功能说明：OF = 1 转移，即有符号数有溢出则转移。
- JNO（Jump if not overflow）

  功能说明：OF = 0 转移，即有符号数无溢出则转移。
- JP 或 JPE（Jump ifparity,or parity even ）

  功能说明：PF = 1 转移，即结果的低 8 位有偶数个 1 则转移。

◆ JNP 或 JPO (Jump if not parity,or parity odd)

功能说明：PF = 0 转移，即结果的低 8 位有奇数个 1 则转移。

◆ JC (Jump if carry)

功能说明：CF = 1 转移，即无符号数有进位或借位则转移。

◆ JNC (Jump if not carry)

功能说明：CF = 0 转移，即无符号数无进位或借位则转移。

(2) 比较两个无符号数，并根据比较的结果转移。可以用下面这 4 个条件转移指令进行无符号数的判断转移。

◆ JBE 或 JNA (Jump if below or equal,or above)

功能说明：低于或等于，或者不高于则转移，此时 CFvZF = 1，即 CF = 1 或 ZF = 1。

◆ JNBE 或 JA (Jump if not below or equal,or above)

功能说明：不低于或等于，或者高于则转移，此时 CFvZF = 0，即 CF = 0 且 ZF = 0。

◆ JB 或 JNAE (Jump if below,or not above or equal)

功能说明：小于，或者不大于或等于则转移，此时 CF = 1。

◆ JNB 或 JAE (Jump ifnot below,or above or equal)

功能说明：不低于，或者高于或等于则转移，此时 CF = 0。

(3) 比较两个带符号数，并根据比较结果转移。可以用下面这 4 个条件转移指令进行有符号数的判断转移。

◆ JL 或 JNGE (Jump if less,or not greater or equal)

功能说明：小于，或者不大于或等于则转移，此时 SF⊕OF = 1，即 SF≠OF。

◆ JNL 或 JGE (Jump if not less,or greater or equal)

功能说明：不小于，或者大于或等于则转移，此时 SF⊕OF = 0，即 SF = OF。

◆ JLE 或 JNG (Jump if less or equal,or not greater)

功能说明：小于或等于，或者不大于则转移，此时 (SF⊕OF) vZF = 1，即 ZF = 1 或 SF≠OF。

◆ JNLE 或 JG (Jump if not less or equal,or greater)

功能说明：不小于或等于，或者大于则转移，此时 (SF⊕OF) vZF = 0，即 ZF = 0 或 SF = OF。

26.2.8 循环控制指令

在设计循环程序时可以用控制指令来控制循环是否继续。PC 机指令系统提供了三种形式的循环控制指令，它们是循环指令 LOOP、当为零或相等时循环指令 LOOPZ/LOOPE 以及当不为零或不相等时循环指令 LOOPNZ/LOOPNE。在使用循环指令之前必须先对 CX 寄存器预置初值。

◆ LOOP：循环指令。

```
语法格式： LOOP OPR
测试条件：(CX)<>0
```

◆ LOOPZ/LOOPE：当为零或相等时循环指令。

```
语法格式： LOOPZ(或 LOOPE) OPR
测试条件：(CX)<>0 且 ZF=1
```

◆ LOOPNZ/LOOPNE：当不为零或不相等时循环指令。

```
语法格式： LOOPNZ(或 LOOPNE) OPR
测试条件：(CX)<>0 且 ZF=0
```

这三条指令的步骤是：

step 1 (CX)<-(CX)-1。

step 2 检查是否满足测试条件，如满足，则(IP)<-(IP)+D8 的符号扩充。

26.2.9 子程序调用和返回指令

在汇编语言的程序设计中，常采用子程序结构。子程序结构相当于高级语言中的过程。为了便于模块化程序设计，往往把程序当中某些具有独立功能的部分编写成独立的功能模块，称之为子程序。程序中可由调用程序（或称主程序）调用这些子程序，而在子程序执行完后又返回调用程序继续执行。为实现这一功能，8086 PC 机提供了两个指令：调用指令 CALL 和返回指令 RET。

1. 调用指令 CALL

由于子程序和调用程序可以在一个段中，也可以不在一个段中，PC 机提供了 4 种寻址方式的调用指令。

◆ 段内直接调用

使用方法如下：

```
语法格式：CALL  DST
执行操作：SP←(SP)-2
          ((SP)+1,(SP))←(IP)
          IP←(IP)+D16
```

这条指令的第一步操作是把 CALL 指令下一条指令的地址（即子程序的返回地址）压入堆栈保护起来，以便子程序返回主程序时使用。第二步操作则是转移到子程序的入口地址去继续执行。指令中 DST 给出转向地址（即子程序的入口地址，亦即子程序的第一条指令的地址），D_{16} 即为机器指令中的位移量，它是转向地址和返回地址之间的差值。

◆ 段内间接调用

使用方法如下：

```
语法格式：CALL  DST
执行操作：SP←(SP)-2
          ((SP)+1,(SP))←(IP)
          IP←(EA)
```

其中 EA 是由 DST 的寻址方式确定的有效地址。

例如：

```
CALL  AX  ;段内间接调用，调用地址由 AX 给出
```

◆ 段间直接调用

它同样是先保留返回地址，然后转移到由 DST 指定的转向地址去执行。由于调用程序和子程序不在同一个段内，因此返回地址的保存以及转向地址的设置都必须把段地址考虑在内。

```
语法格式：CALL  DST
执行操作：SP←(SP)-2
          ((SP)+1,(SP))←(CS)
          SP←(SP)-2
          ((SP)+1,(SP))←(IP)
          IP←偏移地址（机器指令的第二、三个字节）
          CS←段地址（机器指令的第四、五个字节）
```

例如：

```
CALL  2500H:1000H        ;段间直接调用，CS 和 IP 出现在指令中
```

◆ 段间间接调用

使用方法如下：

```
语法格式：CALL  DST
执行操作：SP←(SP)-2
          ((SP)+1,(SP))←(CS)
          SP←(SP)-2
          ((SP)+1,(SP))←(IP)
          IP←(EA)
          CS←(EA+2)
```

其中 EA 是由 DST 的寻址方式确定的有效地址。

2. 返回指令 RET

和调用指令相对应的是返回指令 RET。RET 指令总是放在子程序的末尾，它使子程序在执行完后返回调用程序继续执行，对应段间和段内调用的返回指令形式上相同，均是 RET，但汇编时机器码是不同的。对应段内调用的返回指令执行时，从堆栈顶弹出两个字节传送到 IP 中；对应段间调用的返回指令执行时，从堆栈顶弹出 4 个字节，先弹出两个字节至 IP，再弹出两个字节至 CS，然后返回主程序。与调用指令类似，PC 机提供了 4 种寻址方式的返回指令。

◆ 段内返回

使用方法如下：

```
语法格式：RET
执行操作：IP←((SP)+1,(SP))
```

```
        SP←(SP)+2
```

◆ 段间返回

使用方法如下：

```
语法格式：RET
执行操作：IP←((SP)+1,(SP))
        SP←(SP)+2
        CS←((SP)+1,(SP))
        SP←(SP)+2
```

◆ 段内带立即数返回

使用方法如下：

```
语法格式：RET  EXP
执行操作：IP←((SP)+1,(SP))
        SP←(SP)+2
        SP←(SP)+D16
```

其中 EXP 是一个表达式，根据它的值计算出来的常数成为机器指令中的位移量 D_{16}，它应是 0～FFFFH 中的一个偶数。这条指令表示从堆栈顶弹出返回地址后，再使 SP 值加上 EXP 的值。这个值的大小一般是调用子程序前压入堆栈的参数所占字节数，这些参数供子程序用。当子程序返回后，这些参数不再有用，就可以修改指针使其指向参数入栈以前的值。RET EXP 形式的返回指令一般用在这样一种情况下：主程序为某个子程序提供一定的参数或者参数的地址，在进入子程序前，主程序将这些参数或者地址先送到堆栈中，通过堆栈传递给子程序。子程序运行过程中，使用了这些参数或参数地址，子程序返回时，这些参数或参数地址已经没有在堆栈中保留的必要，因而，可以在返回指令后面加上参数 EXP 的值，这样，使得在返回的同时，将堆栈指针自动移动几个字节，以腾出那些已经无用的参数或参数地址所占用的单元。

◆ 段间带立即数返回

使用方法如下：

```
语法格式：RET  EXP
执行操作：IP←((SP)+1,(SP))
        SP←(SP)+2
        CS←((SP)+1,(SP))
        SP←(SP)+2
        SP←(SP)+D16
```

这里 EXP 的含义及使用情况与段内带立即数返回指令相同。

26.3 汇编语言在 C++中的应用

本节将详细介绍如何在 C++语言程序里嵌入汇编代码，达到高效和简单的双重目的。

26.3.1 内联汇编的优点

因为在 Visual C++中使用内联汇编不需要额外的编译器和联接器，且可以处理 Visual C++中不能处理的一些事情，同时可以使用在 C/C++中的变量，所以非常方便。

内联汇编代码不易于移植，如果你的程序打算在不同类型的机器(比如 x86 和 AIpha)上运行，应当尽量避免使用内联汇编，这时可以使用 MASM，因为 MASM 支持更方便的宏指令和数据指示符。

26.3.2 __asm 语法

__asm 关键字用来调用内联汇编，可以出现在任何合法的 C 或 C++声明中。它不能单独出现，后面必须有汇编指令，可以是一条汇编指令、大括号括起来的一组代码，或者至少是大括号括起来的空代码。术语“__asm 块”指的是任何单独的一条指令或一组指令，可以不包含在大括号里。

第一种语法格式：

```
__asm 汇编指令
```

第二种语法格式：

```
__asm
{
汇编指令列表
 }
```

例如，下面的代码是一个简单的大括号里的__asm 块：

```
__asm
{
   mov al, 4
   mov dx, 0xB008
   out dx, al
}
```

另外，在每一条汇编指令前加上__asm，与前面的方法是一样的作用。例如：

```
__asm mov al, 4
__asm mov dx, 0xB008
__asm out dx, al
```

上面的两个例子所生成的代码是相同的，但是在括号里的__asm 块这种方式更具优势，因为大括号可以使汇编指令很清楚地和 C 或 C++代码分开，避免了无意义的__asm 关键字重复。另外，大括号还可以避免引起歧义。如果想把 C 或 C++代码和__asm 块放在同一行，则必须把这个__asm 块放在括号里。如果没有括号，编译器就不能确定汇编代码结束和 C 或 C++代码起始的位置。

另外，由于大括号里的语句和一般的 MASM 语句格式一样，所以可以很方便地从现有的 MASM 源程序里复制。

不像 C 或 C++中的“{}”，__asm 块中的“{}”不会影响 C 或 C++变量的作用范围。同时，__asm 块可以嵌套，嵌套也不会影响变量的作用范围。

26.3.3 在__asm 块里使用汇编语言

内联汇编和其他的汇编有很多共同之处，比如说，它可以是任意的合法 MASM 表达式。下面说明如何在__asm 块中使用汇编语言。

- 内联汇编指令集：内联汇编完全支持的 Intel 486 指令集，允许使用 MMX 指令。不支持的指令可以使用_EMIT 伪指令定义。_EMIT 伪指令相当于 MASM 中的 DB，但一次只能定义一个字节。比如：

```
__asm
{
    JMP _CodeOfAsm
    _EMIT 0x00 ;定义混合在代码段中的数据
    _EMIT 0x01
    _CodeOfAsm:
    ; 这里是代码
    _EMIT 0x90 ;NOP 指令
}
```

- MASM 表达式：内联汇编可以使用 MASM 中的表达式。比如：

```
MOV EAX, 1
```

- 数据指示符和操作符：虽然__asm 块中允许使用 C 或 C++的数据类型和对象，但它不能用 MASM 指示符和操作符定义数据对象。这里特别指出，__asm 块中不允许有 MASM 中的定义指示符 DB，DW，DD，DQ，DT 和 DF，也不允许 DUP 和 THIS 操作符。MASM 结构和记录也不再有效，内联汇编不接受 STRUC，RECORD，WIDTH 或者 MASK。
- EVEN 和 ALIGN 指示符：尽管内联汇编不支持大多数 MASM 指示符，但它支持 EVEN 和 ALIGN，当需要的时候，这些指示符在汇编代码里面加入 NOP（空操作）指令使标号对齐到特定边界。这样可以使某些处理器取指令时具有更高的效率。
- MASM 宏指示符：内联汇编不是宏汇编，不能使用 MASM 宏指示符（MACRO，REPT，IRC，IRP 和 ENDM）和宏操作符（<>，!，&，%和 TYPE）。
- 段说明：必须使用寄存器来说明段，跨越段必须显式地说明，如 ES:[BX]。
- 类型和变量大小：我们可以使用 LENGTH 来取得 C 或 C++数组中的元素个数，如果不是一个数组，则结果为 1。使用 SIZE 来取得 C 或 C++中变量的大小，一个变量的大小是 LENGTH 和 TYPE 的乘积。TYPE 用来取得一个变量的大小，如果是一个数组，它得到的是一个数组中的单个元素的大小。例如，如果程序中定义了一个包含 8 个元素的整型数组：

```
int arr[8];
```

如表 26.1 所示用 C 和汇编语言的表达式计算出了数组的大小和所含元素的数目。

表 26.1　计算数组大小及元素数目

__asm	C	Size
LENGTH arr	sizeof(arr)/sizeof(arr[0])	8
|SIZE arr	sizeof(arr)	32
TYPE arr	sizeof(arr[0])	4

- 注释：可以使用 C 或 C++的注释，但推荐用 ASM 的注释，即“;”号。

26.3.4 在__asm 块中使用 C/C++元素

C/C++与汇编可以混合使用，在内联汇编中可以使用 C/C++的变量和很多其他 C/C++元素。在__asm 块中可以使用以下 C 或 C++元素：

- 符号，包括标号、变量和函数名。
- 常量，包括符号常量和枚举型（enum）成员。
- 宏定义和预处理指示符。
- 注释，包括“/**/”和“//”。
- 类型名，包括所有 MASM 中合法的类型。
- typedef 名称， 像 PTR、TYPE、特定的结构成员或枚举成员这样的通用操作符。

1. 在__asm 块中使用操作符

__asm 块中不能使用像<<一类的 C/C++操作符。C/C++和 MASM 通用的操作符，比如“*”和“[]”操作符，都被认为是汇编语言的操作符。举个例子：

```
int array[10];
__asm mov array[6], bx ;  Store BX at array+6 (not scaled)
array[6] = 0;        /* Store 0 at array+24 (scaled) */
```

上例中，第一次使用数组时没有根据数组类型调整指针偏移，而第二次是调整过的。注意，可以在__asm 块中使用 TYPE 操作符获得正确的指针偏移。例如，下面这个表达式：

```
__asm mov array[6 * TYPE int], 0 ; Store 0 at array + 24
array[6] = 0;                /* Store 0 at array + 24 */
```

2. 在__asm 块中使用 C/C++符号

__asm 块可以引用其作用域内的任何 C/C++符号（C/C++符号可以是变量名、函数名以及标签，不可以是常量和成员变量）。下面是一些使用 C/C++符号的规则：

- 每一条汇编语句可以只包含一个 C/C++符号。只有使用 LENGTH，TYPE 和 SIZE 这些表达式时，多符号才可以出现在一条汇编指令中。
- 一个__asm 块中的函数引用必须在这之前别声明。否则，编译器将无法区分__asm 块中的函数名和标签 labels。
- __asm 块不能使用与 MASM 保留字相同的 C/C++符号。MASM 保留字包括指令名，比如 PUSH，以及寄存器名称，比如 SI。

3. 在__asm 块中使用 C/C++ 数据

内联汇编能通过变量名直接引用 C/C++的变量。__asm 块中可以引用任何符号，包括变量名。如果 C/C++中的类、结构体或者枚举成员具有唯一的名称，且在“.”操作符之前不指定变量或者 typedef 名称，则__asm 块中只能引用成员名称。然而，如果成员不是唯一的，必须在“.”操作符之前加上变量名或 typedef 名称。例如，下面的两个结构体都具有 same_name 这个成员变量：

```
struct first_type
{
char *weasel;
int same_name;
```

```
};
struct second_type
{
int wonton;
long same_name;
};
```

如果如下声明变量：

```
struct first_type hal;
struct second_type oat;
```

那么，所有引用 same_name 成员的地方都必须使用变量名，因为 same_name 不是唯一的。另外，上面的 weasel 变量具有唯一的名称，可以仅仅使用它的成员名称来引用它。例如：

```
__asm
{
MOV EBX, OFFSET hal
MOV ECX, [EBX]hal.same_name ; 必须使用 'hal'
MOV ESI, [EBX].weasel ; 可以省略 'hal'
}
```

4. 在内联汇编中调用 C 的函数

asm 块中可以调用 C 的函数，包括 C 库函数。下面的例子调用了 printf 库函数。

```
// InlineAssembler_Calling_C_Functions_in_Inline_Assembly.cpp
#include <stdio.h>              // 标准输入输出库函数
char format[] = "%s %s\n";      // 定义变量，用于 printf 函数所使用到的格式化字符串
char hello[] = "Hello";         // 定义变量，用于输出的 Hello 字符串
char world[] = "world";         // 定义变量，用于输出的 World 字符串
int main( void )                // 主函数
{
  __asm                         // in-line 汇编关键字
  {
    mov  eax, offset world
    push eax                    // 以上两行将变量 world 入栈
    mov  eax, offset hello
    push eax                    // 以上两行将变量 hello 入栈
    mov  eax, offset format
    push eax                    // 以上两行将 format 格式化字符串入栈
    call printf                 // 调用函数 printf
    // clean up the stack so that main can exit cleanly
    // use the unused register ebx to do the cleanup
    pop  ebx
    pop  ebx
    pop  ebx                    // 反向出栈
  }
}
```

由于函数的参数是在堆栈上传递的，所以上例中，在调用函数前可以把所需要的参数 string 的指针 push 进去。函数参数的传递是逆序的，所以它们在出栈时才能按照正确的顺序出栈。要想输出 hello world，例子中入栈的参数顺序依次是指向 world 的指针、指向 hello 的指针，然后是 format 函数，最后再调用 printf。

5. 内联汇编中调用 C++函数

__asm 块只能调用 C++的未重载的全局函数,如果调用重载的全局函数或者 C++的成员函数,编译器会报错。

26.3.5 一个例子

下面的例子是在 VS.NET(即 VC7)中用 C 语言写的。先建一个工程,将下列代码放到工程中的.c 文件中编译,无须做特别的设置,即可编译通过。程序实现了对输入字符串求 CRC 校验码功能。

循环冗余编码又名多项式编码,也称 CRC,一般用于通信过程中。在发送端产生一个循环冗余码,附加在信息位后面一起发送到接收端,接收端收到的信息按发送端形成循环冗余码同样的算法进行校验。CRC 校验码的编码方法为:用待发送的二进制数据 $t(x)$除以生成多项式 $g(x)$,将最后的余数作为 CRC 校验码。程序代码如下:

```
#include <stdio.h>
#include <WTYPES.H>         // windows 程序所需要的头文件
////////////////////////////////////////////////////////////////////////
// 全局变量
HWND g_hWnd;                // 主窗口句柄
HINSTANCE g_hInst;          // 进程句柄
TCHAR szTemp[1024];         // 用于存放用户输入的文本字符串
TCHAR szAppName[] = "CRC32 Sample";    // 用于存放进程名称的变量
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// 函数声明
DWORD GetCRC32(const BYTE *pbData, int nSize);
int  WINAPI  WinMain(HINSTANCE  hInstance,  HINSTANCE  hPrevInstance,  LPSTR
lpCmdLine, int iCmdShow);
LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// 主函数,程序的入口函数
int  WINAPI  WinMain(HINSTANCE  hInstance,  HINSTANCE  hPrevInstance,  LPSTR
lpCmdLine, int iCmdShow)
{
    MSG msg;                            // 存放消息的变量
    WNDCLASSEX wndClassEx;              // 窗口类变量

    g_hInst = hInstance;

    wndClassEx.cbSize = sizeof(WNDCLASSEX);
    wndClassEx.style = CS_VREDRAW | CS_HREDRAW;
    wndClassEx.lpfnWndProc = (WNDPROC) WindowProc;
    wndClassEx.cbClsExtra = 0;
    wndClassEx.cbWndExtra = 0;
    wndClassEx.hInstance = g_hInst;
    wndClassEx.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndClassEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndClassEx.hbrBackground = (HBRUSH) (COLOR_WINDOW);
```

```
    wndClassEx.lpszMenuName = NULL;
    wndClassEx.lpszClassName = szAppName;
    wndClassEx.hIconSm = NULL;

    RegisterClassEx(&wndClassEx);      // 注册窗口类

    g_hWnd = CreateWindowEx(0, szAppName, szAppName, WS_OVERLAPPED | WS_CAPTION
| WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX,
        CW_USEDEFAULT, CW_USEDEFAULT, 300, 70,
        NULL, NULL, g_hInst, NULL);        // 创建窗口

    ShowWindow(g_hWnd, iCmdShow);          // 将创建的窗口显示出来
    UpdateWindow(g_hWnd);

    while (GetMessage(&msg, NULL, 0, 0))   // 进入消息循环，获取消息并进行分发
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return ((int) msg.wParam);             // 退出程序
}
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
// 主窗口回调函数，用于处理和窗口有关的消息
LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_CREATE:  // 窗口被创建后，在窗口之上创建一个 EDIT 控件和一个 BUTTON 控件
        CreateWindowEx(WS_EX_CLIENTEDGE, "EDIT", NULL, WS_CHILD | WS_VISIBLE
| WS_BORDER | ES_AUTOHSCROLL | ES_AUTOVSCROLL | ES_NOHIDESEL | WS_OVERLAPPED,
            7, 12, 220, 22,
            hWnd, (HMENU)1000, g_hInst, NULL);
        CreateWindowEx(0,  "BUTTON",  "&OK",  WS_CHILD  |  WS_VISIBLE  |
BS_PUSHBUTTON | WS_OVERLAPPED | BS_FLAT,
            244, 12, 40, 20,
            hWnd, (HMENU)IDOK, g_hInst, NULL);

        break;

    case WM_COMMAND:
        switch (LOWORD(wParam))
        {
        case IDOK:      // 当窗口上的 BUTTON 控件被按下后， 进入相应消息
            // 获得 EDIT 内用户填入的文本
            GetDlgItemText(g_hWnd, 1000, szTemp + 100, 800);
            // 首先通过调用函数获得对所输入文本的 CRC 校验码，然后将该校验数据打印到字符
            // 串变量 szTemp 中
            wsprintf(szTemp, "当前文本框内的字符串的 CRC32 校验码是：0x%lX",
GetCRC32(szTemp + 100, (int)strlen(szTemp + 100)));
            // 将上述格式化后的 szTemp 字符串显示在一个弹出的提示对话框内
            MessageBox(g_hWnd, szTemp, szAppName, MB_OK|MB_ICONINFORMATION);
        }
```

```
        break;

        case WM_DESTROY:            // 处理窗口销毁消息
            PostQuitMessage(0);     // 发消息给主线程，以退出程序
            break;

        default:                    // 其他消息调用默认窗口函数
            return (DefWindowProc(hWnd, uMsg, wParam, lParam));
    }
    return (0);
}
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
// GetCRC32: 求字节流的 CRC32 校验码
// pbData: 指向字节流缓冲区首地址
// nSize: 字节流长度
//
// 返回值: 字节流的 CRC32 校验码
//
// 这里使用查表法求 CRC32 校验码，具体算法可参考相应文章
// 下面使用内联汇编求 CRC32 校验码，充分使用了 CPU 中的寄存器，速度和方便性都是使用 C/C++
// 所不能比拟的
//
DWORD GetCRC32(const BYTE *pbData, int nSize)
{
    DWORD dwCRC32Table[256];

    __asm       // 这片内联汇编是初始化 CRC32 表
    {
        MOV ECX, 256

_NextTable:
        LEA EAX, [ECX-1]
            PUSH ECX
            MOV ECX, 8

_NextBit:
        SHR EAX, 1
            JNC _NotCarry
            XOR EAX, 0xEDB88320
_NotCarry:
        DEC ECX
            JNZ _NextBit

            POP ECX
            MOV [dwCRC32Table + ECX*4 - 4], EAX
            DEC ECX
            JNZ _NextTable
    }

    __asm       // 下面是求 CRC32 校验码的代码
    {
        MOV EAX, -1
```

```
            MOV EBX, pbData
            OR EBX, EBX
            JZ _Done
            MOV ECX, nSize
            OR ECX, ECX
            JZ _Done

_NextByte:
        MOV DL, [EBX]

            XOR DL, AL
            MOVZX EDX, DL
            SHR EAX, 8
            XOR EAX, [dwCRC32Table + EDX*4]

            INC EBX
            LOOP _NextByte
_Done:
        NOT EAX
    }
}
```

26.4 小结

本章介绍了汇编语言的特点和应用领域，详细介绍了汇编语言的基本语句，并举了很多小例子帮助读者理解这些汇编语句的用法。掌握了这些汇编语言的基本概念之后，我们就可以在 C++中使用它们了。内联汇编不需要额外的编译器和链接器，且可以处理 Visual C++中不能处理的一些事情，而且可以使用 C/C++中的变量，非常方便。内联汇编的应用，使得代码简洁而高效，读者可根据实际的需要使用它。

Part

第 5 部分　综 合 案 例

第 27 章　图书管理系统

本章包括

- 需求分析
- 系统设计
- 类设计
- 图书管理
- 读者管理
- 借书模块
- 还书模块
- 系统集成

学习了前面章节的 C++语言相关内容后，该部分将通过两个综合实例来向读者具体介绍通过 C++语言如何实现应用系统的编写和实现。该部分实现了两个简易的管理信息系统，为读者介绍一个完整的数据库应用系统的实现，从而使读者快速掌握应用系统设计的流程，并能单独开发类似管理信息系统。

为简单起见，本章以简单的图书管理系统的设计为例。图书管理系统是一个典型的小型管理信息系统，也是毕业设计中非常典型的一种。在教育行业中，图书的管理最为烦琐，数据量也最大，管理工作非常复杂，图书管理系统就是被设计用来解决这些问题的。随着信息化管理的深化，在一些其他行业和单位也可以使用图书管理系统来管理类似的文档和资料等。

为使读者对数据库应用系统的实现有一个完整的理解，本章从软件工程的角度，按照软件的生命周期，对图书管理系统的实现做一个较完整的讲解。

27.1　需求分析

按照软件的生命周期，实现系统的第一个步骤就是需求分析。需求分析主要对系统需要实现的功能做一个概要说明，一般以数据流图的形式体现。

图书管理系统主要用于实现对图书馆信息的管理。因此，根据对图书馆管理信息的调查分析，归纳出图书管理系统的几个通用功能，分别是管理有关读者、书籍、借阅的信息，其数据流图 DFD 的顶层图如图 27.1 所示。

图 27.1　数据流图 DFD 的顶层图

由于本实例中的图书管理系统相对来说规模较小，所以此处不再对该 DFD 图的顶层图做进一步细化了，直接进入软件生命周期的下一个环节——系统设计。

27.2　系统设计

为简单起见，将该图书管理系统的结构分为四个模块：图书管理模块、读者管理模块、借书模块和还书模块。

27.2.1 总体设计

通过需求分析中得到的数据流图和对功能的分析，此处可以将该简单图书管理系统的功能确定为如下 4 个部分。

- 图书管理：对图书基本信息进行维护，主要包括新增图书、更改图书基本信息、删除图书、查找图书等功能模块。
- 读者管理：对读者的基本信息进行维护，主要包括增加读者、删除读者信息、查找特定读者等功能模块。
- 借书：进行图书借阅操作，该部分是整个图书管理系统中最主要的部分，需要对图书文件和读者文件同时进行读写。
- 还书：进行图书归还操作，该功能模块也需要对图书文件和读者文件同时进行读写。

根据以上的功能叙述与需求分析，将上述分析结果以功能结构图的形式表示，就形成了如图 27.2 所示的系统结构图。

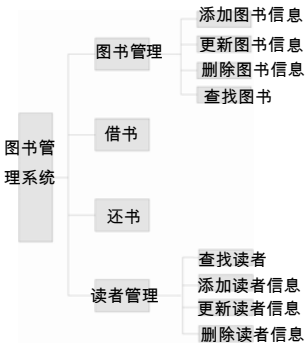


图 27.2 系统结构图

27.2.2 详细设计

得到如图 27.2 所示的系统的总体设计图后，就可以针对其中各个模块进行详细设计了。详细设计中以结构图的方式描述前台程序与后台数据库之间的关系，它能够细化与后台数据库及用户之间的关系。下面给出几个重要模块的程序结构图。

图书管理是管理系统中一个必不可少的模块，它用于对图书的基本信息进行管理。一般来说，图书管理的程序流程如图 27.3 所示。

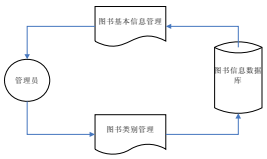


图 27.3 图书管理流程

从上图可看出，图书管理只发生在图书信息基本表与管理员之间，管理员发送图书基本信息管理指令给数据库，数据库以图书信息形式返回处理结果。

类似地，借书模块也可以通过类似图 27.3 所示的程序流程图来表示。借书模块是整个图书管理系统中涉及数据最多的一个模块，主要包括读者数据和图书数据。一般来说，可以将借书模块的程序流程以如图 27.4 所示的流程图来表示。

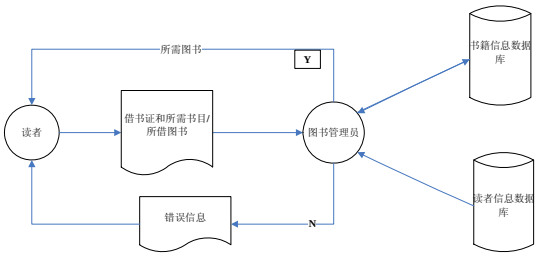


图 27.4　借书流程

借书模块的工作流程如下：读者向管理员申请借书后，管理员根据读者数据库判断该读者是否符合借书条件；如果符合，再判断图书数据库中读者所借的书是否符合外借条件；如都符合则借阅成功，否则返回错误信息。

读者管理和还书等其他一些模块的实现流程都与上述模块有类似之处，此处就不再赘述。数据库应用系统的详细设计中除了对模块的结构进行设计外，更重要的是对数据库的设计，下面将介绍一个具体的数据库应用系统的数据库设计。

27.2.3 数据库设计

数据库设计（Database Design）是指对于一个给定的应用环境，构造最优的数据库模式，建立数据库及其应用系统，使之能够有效地存储数据，满足各种用户的应用需求（信息要求和处理要求）。一般来说，数据库的设计可以分为如下 6 个阶段。

- 需求分析阶段：准确了解与分析用户需求（包括数据与处理），是整个设计过程的基础，是最困难、最耗费时间的一步。
- 概念结构设计阶段：通过对用户需求进行综合、归纳与抽象，形成一个独立于具体 DBMS 的概念模型，是整个数据库设计的关键。
- 逻辑结构设计阶段：将概念结构转换为某个 DBMS 所支持的数据模型，并对其进行优化。
- 数据库物理设计阶段：为逻辑数据模型选取一个最适合应用环境的物理结构（包括存储结构和存取方法）。
- 数据库实施阶段：运用 DBMS 提供的数据语言、工具及宿主语言，根据逻辑设计和物理设计的结果建立数据库，编制与调试应用程序，组织数据入库，并进行试运行。
- 数据库运行和维护阶段：数据库应用系统经过试运行后即可投入正式运行，但在数据库系统运行过程中必须不断地对其进行评价、调整与修改。

在本章的简单图书管理系统实例中，由于其涉及的数据量较小，且 C++对于文件的处理非常方

便，因此此处使用文件来存储图书、读者等数据。数据文件的建立可以在类定义的时候通过构造函数来实现。例如，下面的 C++语句即可建立并打开一个文本文件 book.txt：

```
fstream file("book.txt",ios::in)
```

上述语句使用到文件输入输出流，因此在库函数中应包含对应输入输出文件流 fstream，即通过包含命令#include 来实现，语句如下：

```
#include<fstream>
```

在选择数据存储形式后，就应该为具体数据文件定义字段及相关操作了。读者知道，图书基本信息主要包括图书编号、图书名称等关键字段。此外，考虑到在系统中需要标记该图书是否可借和是否删除，需要为其增加两个标识字段。因此，可以定义如下数据结构表示图书基本信息：

```
int tag;                          // 删除标记，1:已删，0:未删
int no;                           // 图书编号
char name[20];                    // 书名
int onshelf;                      // 是否可借，1:可借，2:不可借
```

同样，读者基本信息主要包括读者编号、读者姓名和所借图书等信息，也需要为其增加一个是否删除标记。因此，可以定义如下数据结构表示读者基本信息：

```
int tag;                          // 删除标记，1:已删，0:未删
int no;                           // 读者编号
char name[10];                    // 读者姓名
int borbook[Maxbor];              // 所借图书
```

在上述数据结构中，用数组存储读者所借图书，其中所借图书最大数由全局变量 Maxbor 来确定。至此，数据库设计就完成了，接下来该进行具体的系统实现了。

读者可以看出，数据库设计是整个数据库应用系统中非常重要的一个步骤，只有将数据库设计完整了，在以后的具体实现中才能事半功倍，否则将加大具体实现的烦琐程度，甚至可能导致整个系统的失败。因此，包括数据库在内的系统设计是整个数据库应用系统设计的基础。

27.3 类设计

从本节开始进入具体的系统实现，后续小节将针对不同的模块分别进行设计和编码。类的设计是面向对象程序设计的基础，设计一个好的类对于程序的结构优化有很大促进作用。本节将给出图书类和读者类的私有成员和公有成员。

27.3.1 创建应用程序

在进行具体的类设计前，读者首先需要在 Dev-C++中新建一个 Console Application 应用程序，其具体步骤如下：

step 1 选择 Dev-C++集成开发环境中的“文件”→“新建”→“工程”命令，打开“新工程”对话框，在其“Basic”选项卡中选中“Console Application”选项，并在下方的“名称”文本框中输入该工程

的名称，这里输入“图书管理系统”，选择创建工程类型为“C++工程”，如图 27.5 所示。

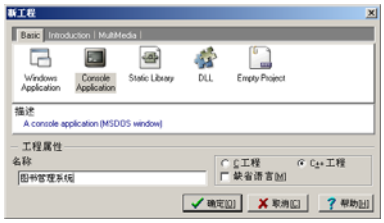


图 27.5　新建工程

step 2　在上述对话框中单击“确定”按钮后，即弹出工程保存对话框，要求用户选择该应用程序的保存路径，选择后如图 27.6 所示。

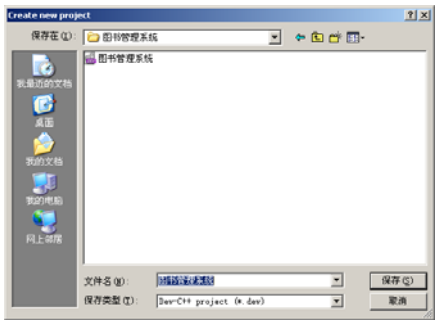


图 27.6　保存工程

step 3　工程保存完成后，Dev-C++的集成开发环境将自动打开一个名称为“main.cpp”的 C++源程序文件，其中包含了部分初始代码，如图 27.7 所示。

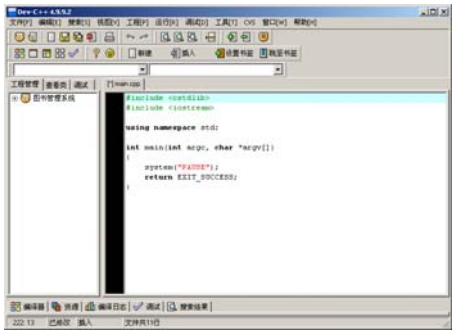


图 27.7　工程初始界面

至此，新建应用程序的步骤就完成了。用户可以在该 main.cpp 源程序文件中进行类设计、程序编码和实现等操作。此外，用户还可以通过选择 Dev-C++集成开发环境中的“文件”→“新建”→“源文件”命令，为该工程添加新的 C++源程序文件。

27.3.2　设计图书类

图书类是对图书的描述，主要包括图书的编号、名称、是否可借和是否删除等基本信息，还包括对图书的借出和还入等基本操作。其中，图书的基本信息以私有成员形式出现在类中，而图书的

借还等基本操作以成员函数形式出现在类中。

根据前面小节中定义的数据字段，可在私有成员中定义图书的编号、名称、是否可借和是否删除 4 个成员，而成员函数（即对图书的操作）则主要有新增图书、借书、还书和输出图书信息等。因此，设计图书类 Book 如下：

```
class Book              // 图书类，实现对图书的描述，图书的编号、书名、借出、还入等
{
private:
    int tag;                                  // 删除标记，1:已删，0:未删
    int no;                                   // 图书编号
    char name[20];                            // 书名
    int onshelf;                              // 是否可借，1:可借，2:不可借
public:
    Book(){}                                  // 构造函数
    char *getname() { return name; }          // 获取姓名
    int getno(){ return no; }                 // 获取图书编号
    int gettag(){ return tag; }               // 获取删除标记
    void setname(char na[])                   // 设置书名
    {
        strcpy(name,na);
    }
    void delbook(){ tag=1;}                   // 删除图书
    void addbook(int n,char *na)              // 增加图书
    {
        tag=0;
        no=n;
        strcpy(name,na);
        onshelf=1;
    }
    int borrowbook()                          // 借书操作
    {
        if (onshelf==1)                       // 当前图书可借
        {
            onshelf=0;
            return 1;
        }
        return 0;
    }
    void retbook()                            // 还书操作
    {
        onshelf=1;
    }
    void disp()                               // 输出图书
    {
        cout << setw(6) << no << setw(18) << name << setw(10)
            <<(onshelf==1? "可借":"不可借") <<endl;
    }
};
```

可以看出，类 Book 分别定义了该图书管理系统中图书的属性和操作，如果通过类图的形式将其表示出来，可画出如图 27.8 所示的类的成员和函数图。

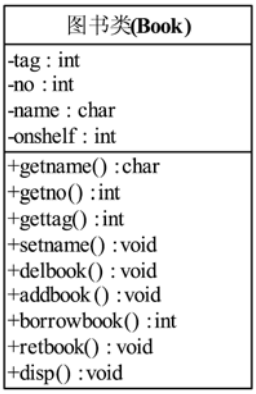


图 27.8　图书类的成员和函数图

可以看出，该类包含了删除标记、图书编号、名称和是否可借等 4 个属性作为其私有成员，包含了获取姓名、获取图书编号、获取删除标记、设置书名、删除图书、新增图书、借书、还书和显示图书等 9 个操作作为其公有成员函数。在具体的使用中，通过成员函数对图书类进行相关操作，这也是面向对象程序设计的优势。

27.3.3 设计图书库类

与图书类不同的是，图书库类被设计用来描述以某种形式存储的图书数据记录，此处以文本文件作为存储图书数据的文件。显然，图书库类中包含的私有成员只有图书对象，为了更好地描述对图书库内图书的操作，可以引进一个图书记录指针 top。

同样地，对图书库内图书的操作主要包括读取文本文件中数据记录到内存中、新增图书到图书库中、图书库维护、在图书库中查找图书等。因此，设计图书库类时将上述操作以成员函数的形式进行描述即可。设计图书库类 BDatabase 如下：

```
class BDatabase                          // 图书库类，实现对图书的维护、查找、删除等
{
private:
   int top;                              // 图书记录指针
    Book book[Maxb];                     // 图书记录
public:
    BDatabase()                          // 构造函数，将 book.txt 读到 book[]中
    {
            Book b;                      // 定义图书对象
            top=-1;                      // 初始化图书记录指针
            fstream file("book.txt",ios::in); // 以读取方式打开数据文件 book.txt
            while (1)                    // 读取数据文件 book.txt 中的图书记录
            {
                file.read((char *)&b,sizeof(b));
```

```
            if (!file) break;
            top++;
            book[top]=b;
        }
        file.close();
    }
    void clear( )                       // 全删
    {
        top=-1;
    }
    int addbook(int n,char *na);        // 增加图书
    Book *query(int bookid);            // 查找图书
    void bookdata();                    // 图书库维护
    void disp()
    {
        for (int i=0;i<=top;i++)
          if (book[i].gettag()==0)
              book[i].disp();           // 显示该图书记录
    }
    ~BDatabase()                        // 析构函数,将 book[]写到 book.txt 文件中
    {
        fstream file("book.txt",ios::out); // 以写入方式打开数据文件 book.txt
       for (int i=0;i<=top;i++)
            if (book[i].gettag()==0)
                file.write((char *)&book[i],sizeof(book[i]));
            file.close();               // 关闭文件
    }
};
```

可以看出，上述图书库类的构造函数从数据文件 book.txt 中读取图书记录，通过 addbook，query 和 bookdata 等一系列成员函数实现对图书库内图书进行操作，最后调用析构函数将维护结果写入到数据文件 book.txt 中并关闭。用类图来表示该图书库类，如图 27.9 所示。

图 27.9 图书库类图

27.3.4 设计读者类

读者类是对读者的描述，主要包括读者的编号、名称、所借图书和是否删除等基本信息，还包括对图书的借出和还入等基本操作。其中，读者基本信息以私有成员形式出现在类中，而读者的借还等基本操作以成员函数形式出现在类中。

根据前面小节中定义的数据字段，可在私有成员中定义读者的编号、名称、所借图书和是否删除等 4 个成员，而成员函数（即对图书的操作）则主要有新增读者、借书、还书和输出读者信息等。因此，设计读者类 Reader 如下：

```
class Reader                                  // 读者类，实现对读者信息的描述
{
   private:
    int tag;                                  // 删除标记，1:已删，0:未删
    int no;                                   // 读者编号
    char name[10];                            // 读者姓名
    int borbook[Maxbor];                      // 所借图书
   public:
        Reader() {}
    char *getname() {return name;}            // 获取姓名
        int gettag() {return tag;}            // 获取删除标记
       int getno() {return no;}               // 获取读者编号
    void setname(char na[])                   // 设置姓名
        {
           strcpy(name,na);

        }
    void delreader(){  tag=1;  }              // 设置删除标记，1:已删，0:未删
    void addreader(int n,char *na)            // 增加读者
    {
        tag=0;
        no=n;
        strcpy(name,na);                      // 设置读者姓名
        for(int i=0;i<Maxbor;i++)
            borbook[i]=0;                     // 设置所借书籍为空
    }
    void borrowbook(int bookid)               // 借书操作
    {
        for(int i=0;i<Maxbor;i++)             // 小于最大借书数目
        {
            if (borbook[i]==0)
            {
                borbook[i]=bookid;            // 设置所借书籍的图书编号
                return;
            }
        }

    }
    int retbook(int bookid)                   // 还书操作
    {
        for(int i=0;i<Maxbor;i++)
        {
            if(borbook[i]==bookid)            // 设置读者所借书籍字段中对应的图书编号为空
            {
            borbook[i]=0;
```

```
            return 1;
            }
        }
        return 0;
    }
    void disp()                          // 读出读者信息
    {
        cout << setw(5) << no <<setw(10) << name<<"借书编号：[";
        for(int i=0;i<Maxbor;i++)
            if(borbook[i]!=0)            // 输出所借图书
                cout << borbook[i] << "|";
        cout << "]"<<endl;

    }
};
```

可以看出，读者类 Reader 与图书类 Book 非常类似，其私有成员和成员函数也都类似，因此也可以用如图 27.10 所示的类图将其表示出来。

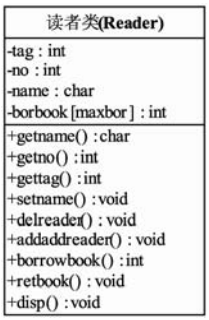


图 27.10 读者类图

27.3.5 设计读者库类

与图书库类类似，读者库类被设计用来描述以某种形式存储的读者数据记录，此处以文本文件作为存储读者基本信息的数据文件。显然，读者库类中包含的私有成员只有读者对象，为了更好地描述对读者库内读者的操作，可以引进一个读者记录指针 top。

同样，对读者库内读者的操作主要包括读取文本文件中的数据记录到内存中、新增读者到读者库中、读者库维护、在读者库中查找读者等。因此，设计读者库类时将上述操作以成员函数的形式进行描述即可。设计读者库类 RDatabase 如下：

```
class RDatabase                          // 读者库类，实现建立读者的个人资料
{
private:
   int top;                              // 读者记录指针
   Reader read[Maxr];                    // 读者记录
public:
    RDatabase()                          // 构造函数，将 reader.txt 读到 read[]中
    {
        Reader s;
        top=-1;
        fstream file("reader.txt",ios::in);// 打开一个输入文件
        while (1)
        {
            file.read((char *)&s,sizeof(s));
            if (!file)break;
            top++;
            read[top]=s;
        }
        file.close();                    // 关闭 reader.txt
```

```
    }
    void clear()                          // 删除所有读者信息
    {
        top=-1;
    }
    int addreader(int n,char *na);        // 添加读者时先查找是否存在
    }
    Reader *query(int readerid);          // 按编号查找
    void disp()                           // 输出所有读者信息
    {
        for (int i=0;i<=top;i++)
            read[i].disp();
    }
    void readerdata();                 // 读者库维护
    ~RDatabase()                       // 析构函数，将 read[]写到 reader.txt 文件中
    {
        fstream file("reader.txt",ios::out);
        for (int i=0;i<=top;i++)
            if (read[i].gettag()==0)
                file.write((char *)&read[i],sizeof(read[i]));
        file.close();
    }
};
```

可以看出，上述读者库类的构造函数从数据文件 book.txt 中读取读者记录，通过 addreader，query 和 readerdata 等一系列成员函数实现对读者库内读者进行操作，最后调用析构函数将维护结果写入到数据文件 book.txt 中并关闭。用类图来表示该读者库类，如图 27.11 所示。

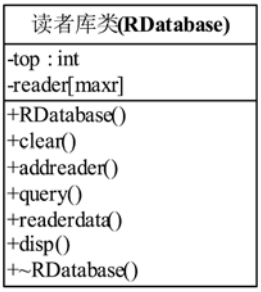


图 27.11　读者库类图

该图书管理系统所需要用的几个类的设计已经完成了，接下来要做的工作就是将图书管理、读者管理、借书和还书等操作以类成员函数的形式表示出来。在结构化程序设计中，可以将其认为是详细设计和编码的步骤。

27.4 图书管理

图书管理是指对图书基本信息进行维护，主要包括新增图书、更改图书基本信息、删除图书、查找图书等功能模块，本节将先介绍各个功能模块的实现，最后将各模块集成在图书管理的界面下进行统一操作。

27.4.1 查找图书

查找图书是指在图书库中查找指定图书编号的图书，如果找到则返回该图书所在地址。在图书库类 BDatabase 的设计中已经声明了查找图书成员函数 query，C++允许在类外对公有成员函数进行定义，因此此处定义该函数的具体功能。

```
Book *BDatabase::query(int bookid)                  // 查找图书
{
    for (int i=0;i<=top;i++)
        if (book[i].getno()==bookid &&book[i].gettag()==0)
        {
            return &book[i];                        // 返回查找结果
        }
        return NULL;
}
```

上述成员函数判断指定的图书编号在图书库中是否存在，且其删除标记是否为 0，即是否未被删除，如果两个条件都满足则返回该图书所在地址，否则返回空。为了便于读者更好地理解上述函数的流程，下面给出该函数的执行流程图，如图 27.12 所示。

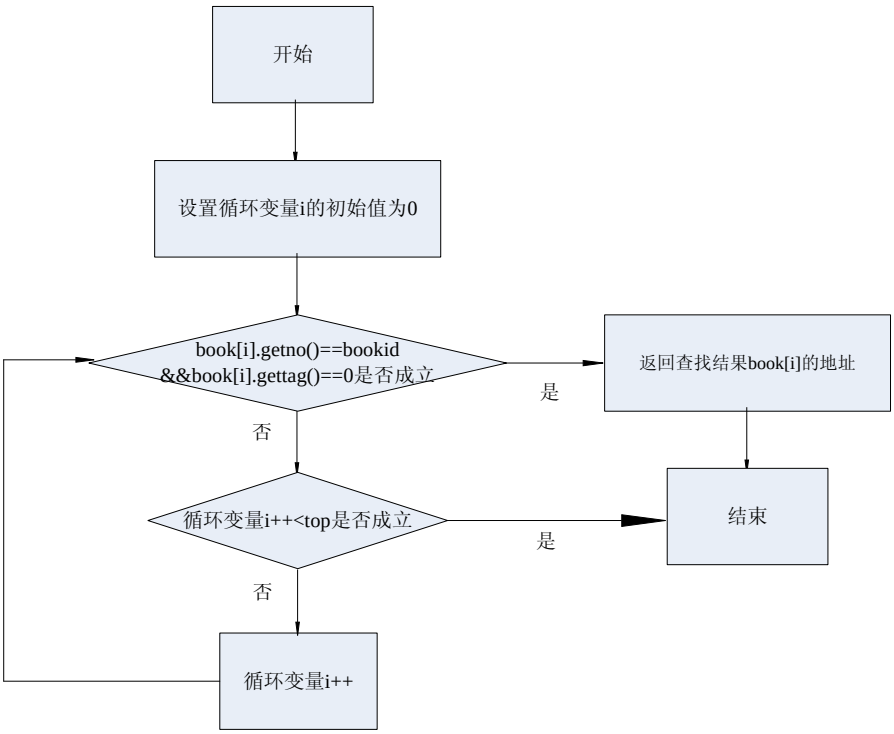


图 27.12 执行流程图

当在主程序 main 中调用该类的 query 成员函数后，系统执行上述程序，如查找成功则返回指定编号的图书地址，否则返回空。例如，如图 27.13 所示为在主程序中调用该函数查找图书编号为 00001 的图书并显示其全部信息的结果。

图 27.13　查找图书结果

由于图书编号为主关键字，因此在数据文件中不允许出现图书编号重复的情况，否则系统将出现错误。

27.4.2　增加图书

增加图书是指输入新的图书信息，将这些信息写入到图书库类指定的数据文件中。当图书管理系统第一次运行后，一般数据文件为空，这就需要通过增加图书成员函数来为系统添加图书记录。

显而易见，根据前面定义的图书基本信息包括图书的编号、名称、是否可借和是否删除，此处新增图书只需指定图书的编号和名称，其他两个字段可在成员函数中自行指定，即是否可借标记为 1，是否删除标记为 0。此外，由于系统中不允许有重复的图书编号，因此在增加图书前需要先确定图书库中没有该图书，这就需要调用 27.4.1 节中定义的 query 成员函数了。因此，根据如上分析，可以定义增加图书成员函数 addbook 如下：

```
int BDatabase::addbook(int n,char *na)          // 增加图书
    {
        Book *p=query(n);                       // 查找是否有重复图书编号
        if (NULL==p)                            // 该图书编号无重复
        {
            top++;                              // 记录指针加 1
            book[top].addbook(n,na);            // 调用图书类的增加图书成员函数
            return 1;
        }
        return 0;                               // 该图书编号重复
    }
```

上述成员函数调用了两个已经定义的类的成员函数，分别为图书库类 BDatabase 的查找图书函数 query 和图书类 Book 的增加图书函数 addbook，如增加成功则返回 1，否则返回 0。该函数的执行流程如图 27.14 所示。

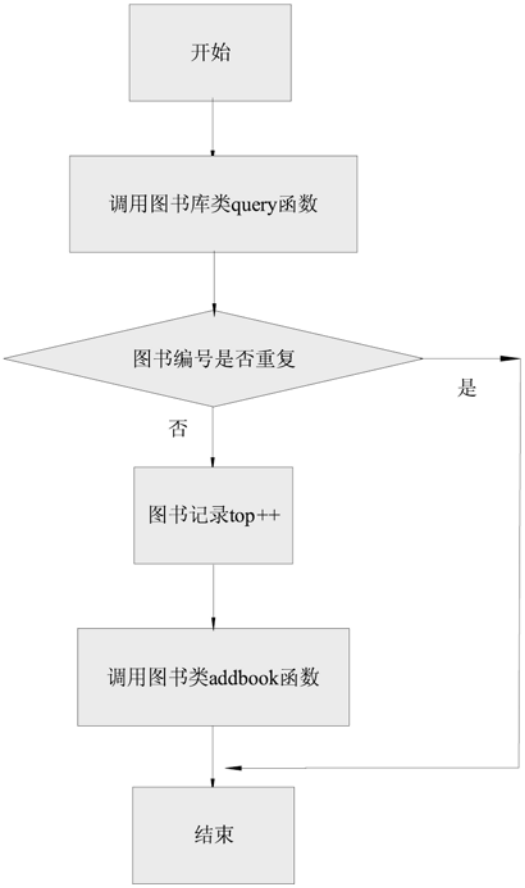


图 27.14　执行流程图

可以看出，该函数定义了图书类 Book 的对象，并调用了该类的成员函数，因此 BDatabase 类的成员函数应定义在 Book 类的成员函数之后。在主函数中调用该函数后，就可以从键盘输入接收图书的编号和名称，完成增加图书的操作。例如，如图 27.15 所示为增加一本编号为 00001、名称为“C 语言”的图书的运行结果。

图 27.15　增加图书结果

27.4.3 维护图书

维护图书成员函数是一个包含图书查询、增加图书、删除图书和显示图书等函数的函数，它提供一个集成界面，供用户操作。当用户进入维护图书模块后，将显示如图 27.16 所示的程序界面，用户通过输入选项来执行相关的成员函数。

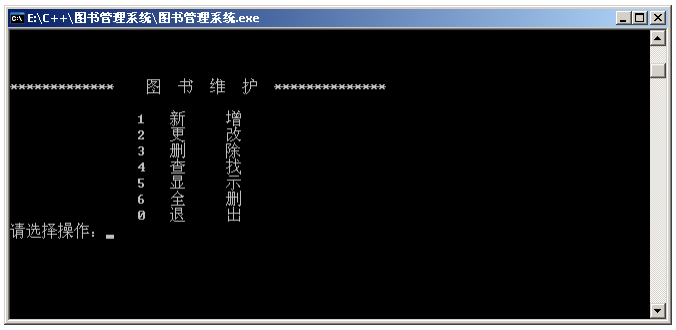


图 27.16 图书维护界面

为实现该功能模块，需要在用户界面上画出该界面，并提供人机交互，根据用户选择的功能调用不同的成员函数。为确保用户能多次调用其中的函数，可设计一个循环，当用户输入选项 0 时退出循环。因此，该函数 bookdata 设计如下：

```
void BDatabase::bookdata()                    // 维护图书成员函数
{
   char choice;                               // 定义变量
    char bname[40];
    int bookid;
    Book *b;                                  // 创建图书对象
    while (choice!='0')                       // 当输入不为 0 时
    {
        cout <<"\n\n\n\**********    图  书  维  护  ********* "<<endl<<endl;
        cout<<"\t\t1   新     增\n \t\t2   更     改\n\t\t3   删     除\n\t\t4
查    找\n\t\t5   显     示\n\t\t6   全     删\n\t\t0   退     出"<<endl;
        cin >> choice;                        // 输入选项
        switch (choice)
        {
          case '1':                           // 选择新增图书功能
                cout << "输入图书编号:"<<endl;
                cin >> bookid;
               cout << "输入图书书名:"<<endl;
                cin >> bname;
                addbook(bookid,bname);        // 调用 addbook 函数
                break;
          case '2':                               // 选择更改图书功能
                cout << "输入图书编号:"<<endl;
                cin >> bookid;
                b=query(bookid);                  // 调用 query 函数
                if (b==NULL)                      // 未找到该图书
                {
                    cout << " 该图书不存在  "<<endl;
                   break;
                }
                cout << "输入新的书名:"<<endl;
```

```
                cin >> bname;
                b->setname(bname);              // 重新设置图书名称
                break;
            case '3':                           // 选择删除图书功能
                cout <<"请输入要删除的图书编号:"<<endl;
                cin >> bookid;
                b=query(bookid);                // 调用 query 函数
                if (b==NULL)
                {
                   cout <<" 该图书不存在" << endl;
                   break;
                }
                b->delbook();                   // 删除图书
                break;
            case '4':                           // 选择查找图书功能
                cout << "请输入要查找的图书编号:"<<endl;
                cin >> bookid;
                b=query(bookid);                // 调用 query 函数
                if (b==NULL)
                {
                    cout <<"  该图书不存在"<< endl;
                    break;
                }
                b->disp();                      // 显示找到的图书信息
                break;
            case '5':                           // 选择显示所有图书功能
                disp();
                break;
            case '6':                           // 选择全部删除图书功能
                clear();
                break;
            default:cout<<"输入错误，请从新输入:";// 错误的输入
        }
    }
}
```

可以看出 ,如上成员函数 bookdata 通过一个多重分支结构提供多个选项 ,并设计了一个 while 循环，只有当用户输入字符 0 时才退出。该程序结构较为复杂，为帮助读者理解该函数，下面给出其执行流程，如图 27.17 所示。

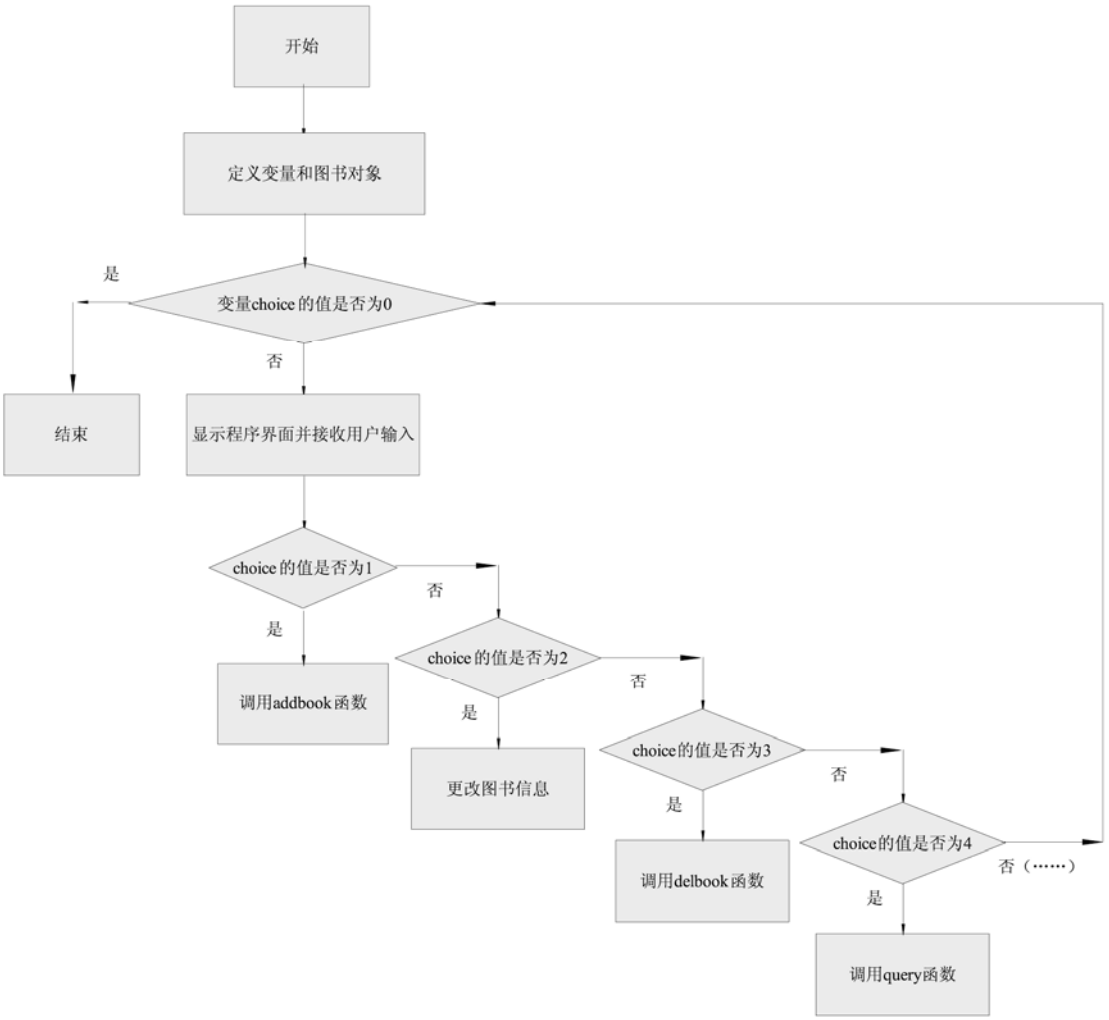


图 27.17　图书维护执行流程

至此，图书管理中的功能模块查找图书、增加图书和维护图书都已经完成了。读者可以看出，这三个模块涉及到了图书类 Book 和图书库类 BDatabase，主要工作是定义其成员函数，这里面向对象的程序设计思想是读者需要仔细理解的。

27.5 读者管理

读者管理是对图书管理系统的读者进行管理的功能部分，主要是指对读者基本信息的管理，其实现与图书管理类似。

27.5.1 查找读者

查找读者是指在读者库中查找指定读者编号的读者，如果找到，则返回该读者所在读者库的地

址。在读者库类 RDatabase 的设计中已经声明了查找读者成员函数 query，下面给出该函数的具体定义。

```
Reader * RDatabase::query(int readerid)            // 按编号查找
{
    for (int i=0;i<=top;i++)
        if (read[i].getno()==readerid &&
        read[i].gettag()==0)                       // 查找成功
        {
            return &read[i];                       // 返回读者地址
        }
        return NULL;
}
```

可以看出，上述成员函数判断指定的读者编号在读者库中是否存在，且其删除标记是否为 0，即是否未被删除，如果两个条件都满足则返回该读者所在地址，否则返回空，其程序执行流程如图 27.18 所示。

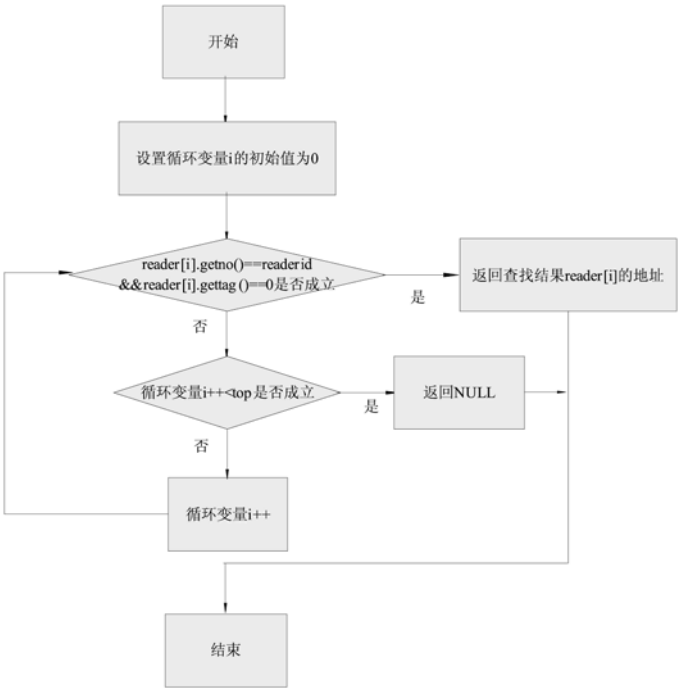


图 27.18　查找读者执行流程

当在主程序 main 中调用该类的 query 成员函数后，系统执行上述程序，如查找成功则返回指定编号的读者地址，否则返回空。例如，如图 27.19 所示为在主程序中调用该函数查找读者编号为 10001 的读者并显示其全部信息的结果。

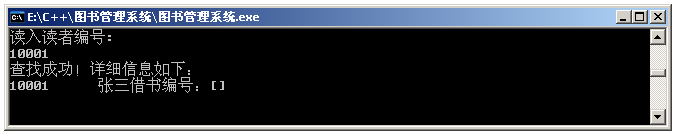


图 27.19　查找读者结果

由于读者编号为主关键字，因此在数据文件中不允许出现读者编号重复的情况，否则系统将出现错误。

27.5.2 增加读者

与增加图书模块实现类似，增加读者是指输入新的读者信息，将这些信息写入到读者库类指定的数据文件中。当图书管理系统第一次运行后，一般数据文件为空，这就需要通过增加读者成员函数来为系统添加读者记录。

同样，在增加读者的编号、姓名、所借书籍和是否删除等信息前，也需要判断该读者编号是否已经存在，如存在则不能添加重复的读者编号。因此，增加读者成员函数的实现代码如下：

```
int addreader(int n,char *na)          // 添加读者时先查找是否存在
{
    Reader *p=query(n);                // 首先查找读者
    if (p==NULL)                       // 未找到该读者编号
    {
      top++;                           // 添加
      read[top].addreader(n,na);
      return 1;
    }
    return 0;
```

上述成员函数调用了两个已经定义的类的成员函数，分别为读者库类 RDatabase 的查找读者函数 query 和读者类 Reader 的增加图书函数 addreader，如增加成功则返回 1，否则返回 0。该成员函数的执行流程如图 27.20 所示。

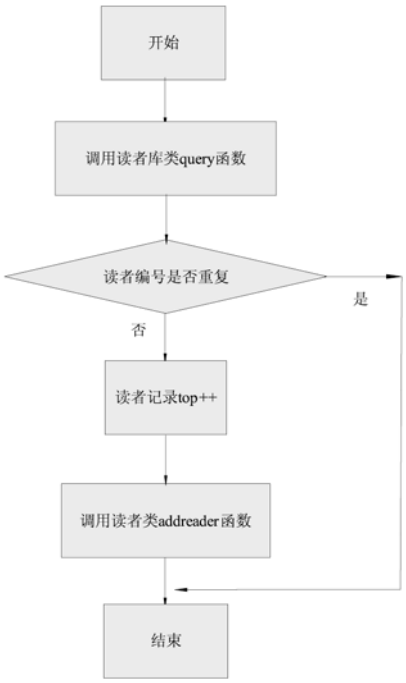


图 27.20　增加读者执行流程

当在主程序 main 中调用该成员函数后，会在打开的数据文件中写入用户指定的读者的编号和姓名，如图 27.21 所示为写入读者编号为 10001、姓名为“张三”的记录的结果。

图 27.21 增加读者结果

27.5.3 维护读者

维护读者成员函数是一个包含读者查询、增加读者、删除读者和显示读者等函数的函数，它提供一个集成界面，供用户操作。当用户进入维护读者模块后，将显示如图 27.22 所示的程序界面，用户通过输入选项来执行相关的成员函数。

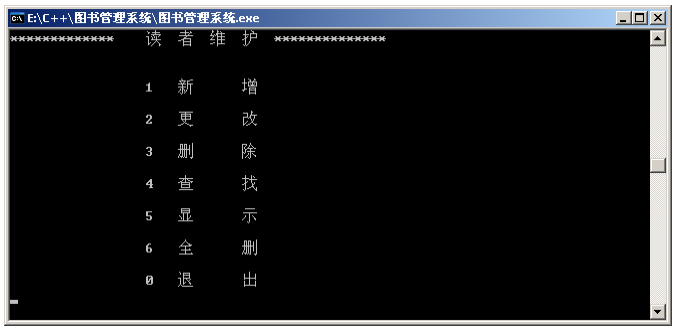


图 27.22 维护读者界面

为了实现该功能模块，需要在用户界面上画出该界面，并提供人机交互，根据用户选择的功能调用不同的成员函数，为确保用户能多次调用其中的函数，可设计一个循环，当用户输入选项 0 时退出循环。因此，该函数 readerdata 设计如下：

```
void RDatabase::readerdata()
{
    char choice;                              // 定义变量
    char rname[20];
    int readerid;
    Reader *r;                                // 创建对象
    while (choice!='0')
    {
       cout <<"\n\n**********    读 者 维 护  ********** \n\n\n\t\t 1
新      增\n\n\t\t 2   更      改\n\n\t\t 3   删      除\n\n\t\t 4   查      找
\n\n\t\t 5   显      示\n\n\t\t 6   全      删\n\n\t\t 0   退      出"<<endl;
        cin >> choice;                        // 接收用户输入的选项
        switch (choice)
        {
          case '1':                           // 选择新增读者功能
               cout << "输入读者编号:";
               cin >> readerid;
```

```
            cout << "输入读者姓名:";
            cin >> rname;
            addreader (readerid,rname);// 调用 addreader 函数
            break;
        case '2':                          // 选择更改读者功能
            cout << "输入读者编号:";
            cin >> readerid;
            r=query(readerid);             // 首先查找读者编号
            if (r==NULL)
            {
                cout << " 该读者不存在  "<<endl;
               break;
            }
            cout << "输入新的姓名:";
            cin >> rname;
            r->setname(rname);             // 重新设置读者姓名
            break;
        case '3':                          // 选择删除读者功能
            cout << " 输入读者编号:";
            cin >> readerid;
            r=query(readerid);
            if (r==NULL)                   // 未找到该读者
            {
               cout <<" 该读者不存在" << endl;
               break;
            }
            r->delreader();                // 调用 delreader 函数
            break;
          case '4':                        // 选择查找读者功能
            cout << "读入读者编号:";
            cin >> readerid;
            r=query(readerid);
            if (r==NULL)                   // 未找到该读者
            {
                cout <<"该读者不存在"<< endl;
                break;
            }
            r->disp();                     // 显示读者信息
            break;
          case '5':                        // 选择显示所有读者功能
            disp();
            break;
          case '6':                        // 选择删除所有读者功能
            clear();
            break;
          default:cout<<"输入错误，请从新输入：";break;  // 错误的输入
      }
  }
```

```
}
```

可以看出，如上成员函数 readerdata 通过一个多重分支结构提供多个选项，并设计了一个 while 循环，只有当用户输入字符 0 时才退出，其执行流程与图书维护的模块类似，此处就不再赘述了。

27.6 借书模块

借书模块是图书管理系统中的关键模块之一，在完成借书操作的同时，需要对图书数据文件和读者数据文件进行读者操作，主要包括如下两个步骤：

step 1 根据图书编号在图书库中查找该读者需要借出的图书，如存在并且可借，则将其可借标记置为不可借，将图书借出；如不存在或已借出，则给出相应提示信息。

step 2 根据读者编号在读者库中查找该读者，如存在则将其所借图书字段中加上步骤 1 中的图书编号，完成借阅操作。

根据如上分析，可以通过函数的形式将如上步骤表示出来，或直接写在主函数 main 中也可。此处为体现结构化程序设计思想，以函数 borrow 表示，代码如下：

```
int borrow()                                        // 借书模块
{
    int bookid,readerid;                            // 定义变量
    RDatabase ReaderDB;                             // 创建对象
    Reader *r;                                      // 定义对象指针
    BDatabase BookDB;
    Book *b;
    cout <<"  借书\n 请输入读者编号:"<<endl;
    cin >>readerid;                                 // 接收用户输入
    cout <<"请输入图书编号："<<endl;
    cin >>bookid;
    r=ReaderDB.query(readerid);                     // 按编号查找读者
    if (NULL==r)                                    // 读者编号不存在
    {
        cout <<"  不存在该读者，不能借书"<< endl;
        exit(0);                                    // 正常结束程序
    }
    b=BookDB.query(bookid);                         // 按编号查找图书
    if (b==NULL)                                    // 图书编号不存在
    {
        cout <<"  不存在该图书，不能借书"<< endl;
        exit(0);
    }
    if (b->borrowbook()==0)                         // 是否借出
    {
        cout << "  该图书已借出，不能借书"<< endl;
```

```
        exit(0);                                    // 正常退出
    }
    r->borrowbook(b->getno());                      // 结束
    cout<<"借书成功！"<<endl;                        // 显示提示信息
    return 0;
}
```

可以看出，上述程序分别调用了图书库的 query 成员函数和读者库的 query 和 borrowbook 成员函数，实现了借书操作，其执行流程如图 27.23 所示。

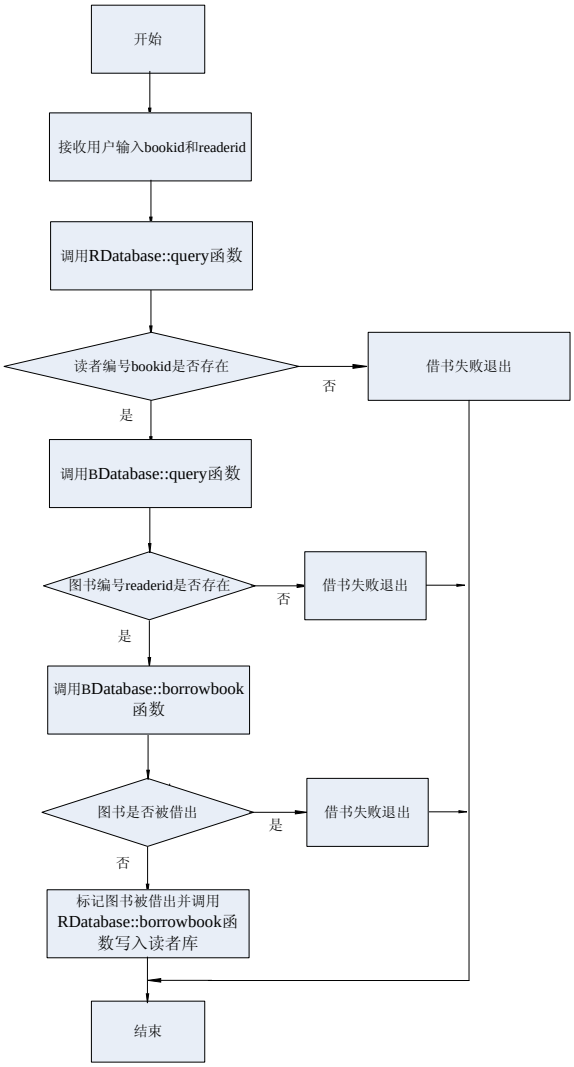


图 27.23　借书执行流程

在主函数 main 中调用该函数后，根据要求输入读者编号和图书编号，即可完成借书操作，此时图书库和读者库中的数据都已做出相应的修改。例如，编号为 10001 的读者借出了编号为 00001 的图书，结果如图 27.24 所示。

图 27.24 借书结果

可以看出，进行借书之前需要确认读者编号存在，还需要确认图书编号存在且该图书未被借出，即可借标记为 0，三个条件中任意一个不满足都无法完成借书操作。

27.7 还书模块

与借书操作类似，还书操作同样是对图书数据文件和读者数据文件进行操作，它要做的是在图书库中将对应编号的图书的可借标记设置为 0，即可借，还需要将读者库文件中对应编号的读者的所借图书字段中的该图书编号删除。

相比起来，还书操作不需要确认图书是否可借，只需确认图书和读者存在即可。因此，还书操作的实现代码如下：

```
int ret()                                              // 还书模块
{
    int bookid,readerid;
    RDatabase ReaderDB;
    Reader *r;                                         // 定义对象指针
    BDatabase BookDB;
    Book *b;
    cout<<"还书\n 请输入读者编号:";
    cin >>readerid;                                    // 接收用户输入
    cout << "请输入图书编号:";
    cin >>bookid;
    r=ReaderDB.query(readerid);                        // 调用读者库的 query 函数
    if (r==NULL)                                       // 未找到该读者
    {
        cout <<" 不存在该读者，不能还书" << endl;
        exit(0);                                       // 正常退出
    }
    b=BookDB.query(bookid);                            // 调用图书库的 query 函数
    if (b==NULL)
    {
        cout <<"  不存在该图书，不能还书" <<endl;
        exit(0);                                       // 正常退出
    }
    b->retbook();                                      // 图书类成员函数还书
    r->retbook(b->getno());                            // 读者类成员函数还书
    cout<<"还书成功！"<<endl;
}
```

上述代码分别调用了读者库的 query 函数和图书库的 query 函数确认读者和图书是否存在，都存在则调用各自的 retbook 函数进行还书操作，其执行流程如图 27.25 所示。

在主函数 main 中调用该函数后，根据要求输入读者编号和图书编号后，即可完成还书操作，此时图书库和读者库中的数据都已做出相应的修改。例如，编号为 10001 的读者还回了编号为 00001 的图书，如图 27.26 所示。

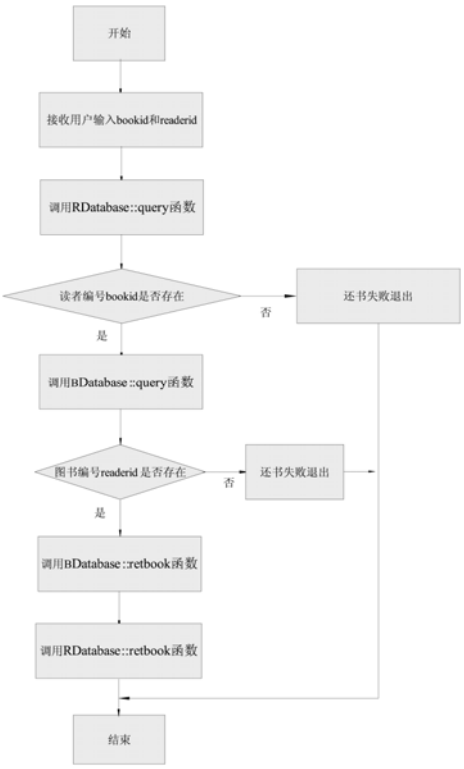


图 27.25　还书执行流程

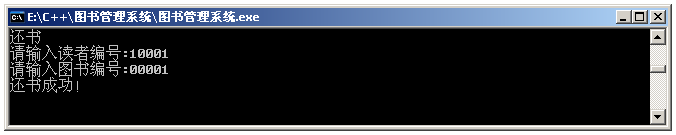


图 27.26　还书结果

27.8 系统集成

经过如上介绍，图书管理系统的各个模块均已设计完成。一般来说，将系统的各功能模块全部完成后，即可将其各部分集成到主窗体上来，以便用户在主窗体中操作所有功能。如果有主菜单，将其添加到主菜单的事件驱动上。因此，为了将这些模块集成起来供用户使用，还需要画出主界面，如图 27.27 所示。

图 27.27 主界面

为了让用户能够对图书管理系统的各个模块进行多项操作,需要设计一个循环,直到用户输入 0 才退出主界面。此外,为了更直观地体现系统的各项功能,此处将借书和还书模块代码嵌入到 main 函数中实现。

```
int main()                                  // main 函数的实现,程序的主界面的引导
{
    char choice;
    int bookid,readerid;
    RDatabase ReaderDB;
    Reader *r;                              // 定义对象指针
    BDatabase BookDB;
    Book *b;
    while(choice!='0')
    {
            cout <<endl<<endl<<"*******   图 书  管 理  系 统  *******\n\n\n";
            cout <<"\t\t\t1    借       书\n\n\t\t\t2    还       书 \n\n\t\t\t3
图 书 管 理\n\n\t\t\t4    读 者 管 理\n\n\t\t\t0    离       开"<<endl;
            cout
<<endl<<endl<<"************************************************\n\n\n";
            cout<<"请选择需要进行的操作序号:";
            cin >> choice;                          // 接收用户输入的选项
            switch (choice)
            {
            case '1':                               // 选择借书模块
                borrow();                           // 调用 borrow 函数
                break;
            case '2':                               // 选择还书模块
                ret();                              // 调用 ret 函数
                break;
            case '3':                               // 选择图书管理模块
                BookDB.bookdata();                  // 调用图书管理函数
                break;
            case '4':                               // 选择读者管理模块
                ReaderDB.readerdata();
                break;
            default:cout<<"输入错误,请从新输入:";// 错误的输入
            }
    }
}
```

可以看出,上述代码使用了一个 switch 多分支语句为用户提供选择功能,当用户选择不同的选项时调用不同的函数实现各功能,上述代码的执行流程如图 27.28 所示。

图 27.28　系统集成执行流程

需要注意的是，在该图书管理系统中，使用到了输入输出流、文件输入输出流、字符串处理等函数，还在具体程序中使用到了定义的命名空间，因此在主程序 main 的开始需加上这些头文件和命名空间的定义。

```
#include <iostream>
#include <iomanip>
#include <string>
#include <fstream>                    // 输入/输出文件流类
using namespace std;
const int Maxr=100;                   // 最多的读者
const int Maxb=100;                   // 最多的图书
const int Maxbor=5;                   // 每位读者最多借 5 本书
```

27.9　小结

本章具体介绍了一个实际的数据库应用系统实例——图书管理系统的实现。本章从软件生命周期的角度，分别对系统的实现进行了需求分析、系统设计和系统实现等步骤。需求分析阶段给出了系统的顶层 DFD 图，总体设计阶段给出了系统结构图，详细设计阶段对常用模块给出了程序结构图，最后详细讲解了各个功能模块的具体实现。本章在具体的模块讲解中，针对几个程序结构较为复杂的模块，画出了执行流程图供读者参考，重点是要理解其中的分支、循环之间的相互嵌套。学习完本章后，读者应对使用 C++设计一个完整的数据库应用系统有了实际的理解，并提高了独自分析设计一个数据库应用系统的能力。

第 28 章　学生管理系统

本章包括

- 需求分析
- 总体设计
- 数据结构设计
- 类设计
- 详细设计
- 系统集成

学生管理系统是较为通用的一类数据库应用程序，许多毕业设计和程序设计的初学者都是通过学生管理系统来入门的。随着社会的发展，学校的规模不断扩大，学校经常需要从日常教学活动中提取相关信息，以反映教学情况。传统的手工操作方式，易发生数据丢失、统计错误，且劳动强度高、速度慢。使用计算机可以高速、快捷地完成以上工作。

本章通过在 Dev-C++集成开发环境下使用 C++完成一个简单通用的学生管理系统，将类和链表的概念和操作融合在其中，读者需要仔细体会。

28.1　需求分析

一个通用学生管理系统中每个学生应该包含地址、出生日期、学科成绩（语文、英语、数学、物理）、个人信息（姓名、性别、民族、国籍、学历）、联系方式（手机、家庭电话、学校电话）等一系列信息。

此外，学生管理系统要求能够任意添加学生、删除学生、编辑学生信息，并且具有保存和读入学生信息的功能。在实际的应用中，需求分析要结合现有的资源和客户的需求，以便根据需求分析的结果设计出合理的系统结构。

根据如上的分析，下面给出简单学生管理系统的主要实现功能：

- 提供添加学生、删除学生、编辑学生信息的操作界面和功能。
- 提供保存和读入学生信息的功能。
- 提供菜单界面。
- 提供学生信息的显示。
- 提供帮助信息。

事实上，一个完整的学生管理系统应该是一个真正意义上的数据库应用程序，它必须拥有数据库管理和用户管理等功能，为简单起见并突出 C++的重点，本章介绍的简单学生管理系统不考虑这些功能。

28.2　总体设计

总体设计阶段即系统的概要设计，需要完成对系统结构的分析和设计，以及设计系统需要的主

要数据结构。本节将基于需求分析的结果，给出简单学生管理系统的总体结构。根据需求分析的结果，本系统至少要分为以下几个模块：main 函数模块、添加学生信息模块、删除学生信息模块、编辑学生信息模块、保存和读入学生信息模块以及显示学生信息模块等 6 个功能模块。各模块的功能说明如下。

- 添加学生信息：添加学生的基本信息到数据文件中，这些基本信息主要包括个人信息、学科成绩和联系方式等。
- 编辑学生信息：对读者的基本信息进行维护，编辑完成后能够保存到数据文件中。
- 删除学生信息：对数据文件中已经存在的学生记录进行删除。
- 保存和读入学生信息：将学生信息写入到数据文件中，或从数据文件中读入学生信息到系统中，以供用户操作。
- 显示学生信息：将数据文件中已经存在的学生记录显示在应用程序界面中。
- main 函数：提供用户操作界面，并以菜单方式提供各个功能模块的调用。

根据以上的功能叙述与需求分析，将上述分析结果以功能结构图的形式表示，就形成了如图 28.1 所示的系统结构图。

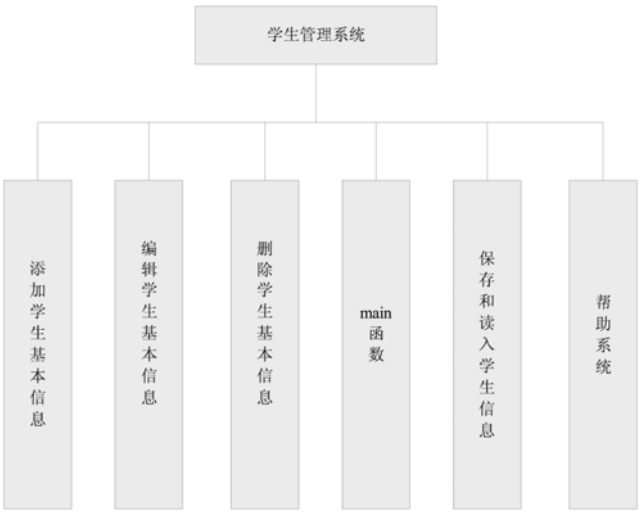


图 28.1 系统结构图

28.3 数据结构设计

与图书管理系统不同的是，学生管理系统中涉及的学生基本信息较多，根据大类可分为地址、个人信息、学科成绩和联系方式等，而每个大类中又包含若干个元数据。同时，考虑到对学生基本信息能够方便地进行添加、删除等操作，此处采用链表这种数据结构来实现。

28.3.1 链表概述

简单地说，链表是由节点组成的非连续的动态线性数据结构，适合于处理数据序列中数据数目

不定且插入和删除较多的问题。相对于数组，链表具有更好的动态性，建立链表时无须预先知道数据总量，在程序中可动态添加或删除数据。

链表是动态地进行存储分配的一种结构，它可以根据需要开辟内存单元。链表有一个“头指针”变量，以 head 表示，它存放一个地址，该地址指向一个元素。链表中每一个元素称为“节点”，每个节点都应包括两部分：一部分为用户需要用的实际数据，另一部分为下一个节点的地址。因此，head 指向第一个元素，第一个元素又指向第二个元素，……直到最后一个元素，该元素不再指向其他元素，它称为“表尾”，它的地址部分放一个“NULL”（表示“空地址”），链表到此结束。链表可以用如图 28.2 所示的“链”来表示。

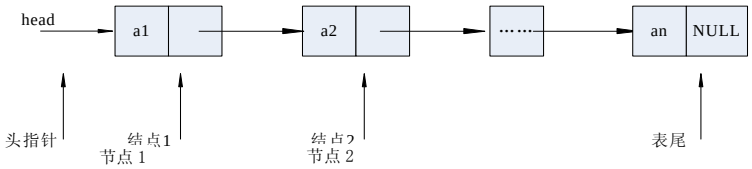


图 28.2　链表

由于链表的特点是动态申请存储空间，并通过指针按照线性表的前驱关系把节点链接起来。因此，链表克服了顺序表无法改变长度的缺点，所需要的存储空间可以根据线性表元素的多少而动态地改变。一般来说，根据节点中指针的数目和链接方式，链表可以分为如下三种形式。

- 单链表：节点由两部分组成，一部分存放线性表节点的数据，另一部分存放指向后继节点的指针。
- 双链表：在单链表的基础上，为每个节点增加一个指向前驱的指针。
- 循环链表：改变单链表和双链表的最后一个节点和第一个节点的指针值，使之分别指向第一个节点和最后一个节点，形成一个循环链。这样在不增加额外开销的情况下，给某些操作带来了方便。其主要优点是从循环表的任一节点出发都能访问到表中的其他节点。

此处主要介绍的是单链表。所谓单链表，是指数据节点是单向排列的。一个单链表节点，其结构类型分为两部分：数据域和链域。其中，数据域用来存储本身数据；链域也称为指针域，用来存储下一个节点地址，或者说指向其直接后继的指针。

单链表的每个节点中除信息域以外还有一个指针域，用来指出其后续节点，单向链表的最后一个节点的指针域为空（NULL）。单向链表由头指针唯一确定，因此单向链表可以用头指针的名字来命名，例如头指针名为 head 的单向链表称为表 head，头指针指向单向链表的第一个节点。

28.3.2　构造单链表

在 C++语言中，单链表是通过声明结构体类型来进行构造的。在用 C++语言实现时，首先说明一个结构体类型，在这个结构体类型中包含一个（或多个）信息成员以及一个指针成员。例如，下面的语句就构造了一个单链表 node。

```
typedef struct node
{
char name[20];
struct node *link;
```

```
}student;
```

上述 C++语句定义了一个单链表的结构体，其中 char name[20]是一个用来存储姓名的字符型数组，指针*link 是一个用来存储其直接后继的指针。

定义好链表的结构体之后，在程序运行的时候，其数据域中就可以存储适当的数据，如有后继节点，则把链域指向其直接后继，若没有，则置为 NULL。例如，如图 28.3 所示为一个包含 3 个节点的单链表 node。

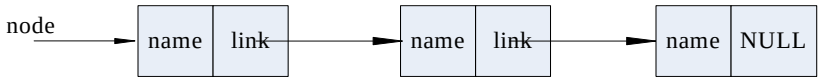


图 28.3 单链表结构

可以看到，链表结构中包含指针型的结构体成员，类型为指向相同结构体类型的指针。根据 C++语言的语法要求，结构体的成员不能是结构体自身类型，即结构体不能自己定义自己，因为这样将导致一个无穷的递归定义，但结构体的成员可以是结构体自身的指针类型，通过指针引用自身类型的结构体。

由于链表在 C++语言中以结构体类型进行构造，因此在构造链表时也可以同时声明链表变量、指针等，如前面声明的变量 student 就是链表 node 的变量。此外在程序中，用户还可以通过如下语句来声明指向该链表类型的指针 p。

```
struct node *p;
```

至此，用户就可以用变量 student 或指针 p 引用该链表中的成员了，其方式与引用结构体成员相同。如引用其中的 name 成员，只需用语句 p->name 即可。

28.3.3 设计数据结构

了解了链表的相关知识后，下面开始针对学生管理系统进行具体数据结构的设计。链表与结构体、共用体等数据结构不同的地方在于，在程序中构造链表后，在程序中要使用该链表还必须先建立该链表结构。只有建立了链表后，才能对该链表进行节点查询、节点插入、节点删除等基本操作。

根据前面对学生基本信息的地址、个人信息、学科成绩和联系方式的分类，下面给每个分类都建立一个结构体，最后将这些结构体作为数据域建立单链表，以方便用户进行添加、删除等操作。建立地址、个人信息、学科成绩和联系方式等 4 个结构体代码如下：

```
struct address                          // 家庭地址
{
        char city[10];                  // 城市
        char town[10];                  // 县城
        char village[10];               // 乡镇
};
struct telephone                        // 联系方式
{
        char SJ[50];                    // 手机
        char JD[30];                    // 家庭电话
        char XD[30];                    // 学校电话
};
struct person                           // 个人信息
{
```

```
        char name[20];                          // 名字
        char sex[10] ;                          // 性别
        char MZ[16];                            // 民族
        char GJ[17];                            // 国籍
        char XL[19];                            // 学历
};
struct score                                    // 学科成绩
{
        char num[20];                           // 学号
        char english[20];
        char chinese[20];
        char math[20];
        char physics[20];
};
```

读者知道，在具体的单链表节点中，如上的 4 个结构体只能作为数据域而存在，还需要建立链域来实现各个节点的相互链接，因此，设计学生基本信息节点如下：

```
typedef struct linknode                         // 定义节点的类型
{
        char address[100];                      // 地址
        char birthday[100];                     // 出生日期
        struct score sc;                        // 成绩
        struct person pe;                       // 个人信息
        struct telephone te;                    // 联系方式
        bool flag;
        struct linknode* next;
}nodetype;
```

在上述节点中，数据域 address 表示学生的地址，birthday 表示学生的出生日期，它们都是字符型数据类型；而数据域 sc 表示学生的各科成绩，pe 表示学生的个人信息，te 表示其联系方式，它们都是结构体数据类型，其各自结构体类型在前面的代码中已定义；flag 为标识，它为逻辑性数据类型；next 为指针型数据类型，它指向下一个学生基本信息节点。通过如上数据结构的设计后，学生信息将以如图 28.4 所示的形式存储在内存中。

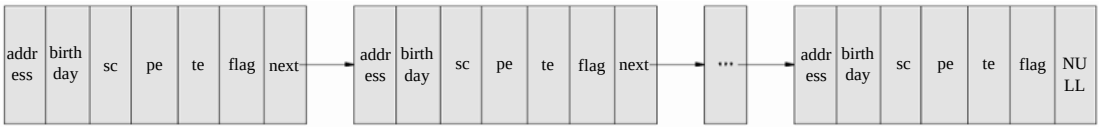


图 28.4　数据结构

至此，数据结构的设计就完成了，用户在具体模块实现中可以通过增加、删除链表的节点来实现添加学生信息、删除学生信息等操作。

28.4　类设计

从本节开始进入具体的系统实现，后续小节将针对不同的模块分别进行设计和编码。类的设计是面向对象程序设计的基础，设计一个好的类对于程序的结构优化有很大促进作用。

28.4.1　创建应用程序

在进行具体的类设计前，读者需要首先在 Dev-C++中新建一个 Console Application 应用程序，其创建步骤如下：

step 1 选择 Dev-C++集成开发环境中的“文件”→“新建”→“工程”命令，打开“新工程”对话框，在其“Basic”选项卡中选中“Console Application”选项，并在下方的“名称”文本框中输入该工程的名称，这里输入“学生管理系统”，选择创建工程类型为“C++工程”，如图 28.5 所示。

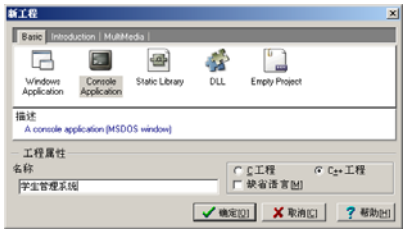


图 28.5 新建工程

step 2 在上述对话框中单击“确定”按钮后，即弹出工程保存的对话框，要求用户选择该应用程序的保存路径，选择后如图 28.6 所示。

图 28.6 保存工程

step 3 在上述对话框中单击“保存”按钮后，Dev-C++的集成开发环境将自动打开一个名称为“main.cpp”的 C++源程序文件，其中包含了部分初始代码，如图 28.7 所示。

至此，新建应用程序的步骤就完成了。读者可以在该 main.cpp 源程序文件中进行类设计、程序编码和实现等操作。此外，读者还可以通过选择 Dev-C++集成开发环境中的“文件”→“新建”→“源文件”命令，为该工程添加新的 C++源程序文件。

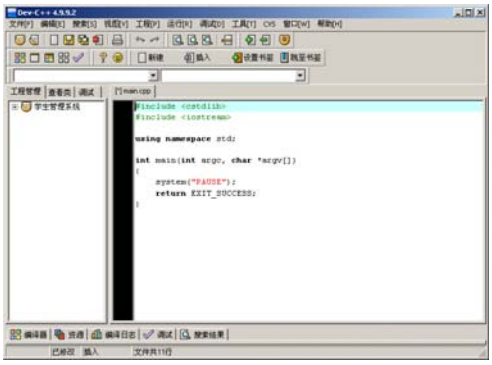


图 28.7 工程初始界面

28.4.2 设计 Student 类

由于学生管理系统中涉及到的数据都是学生的相关信息，没有与其他对象产生联系，因此，该系统只需设计一个类。显而易见，学生类的私有成员即学生的个人信息、学科成绩、联系方式等信息，而这些信息都定义在了链表节点 nodetype 中，因此私有成员定义指向该链表的头指针即可，而公有成员即对链表操作的各个函数。

为了方便读者理解各个成员函数对链表的操作，即对学生管理系统中学生节点的操作，下面给出这些成员函数的声明，具体定义则在具体模块中再详细介绍。根据如上分析，可设计学生类 Student 如下：

```
class Student
{
private:
    nodetype* head;
public:
    Student();
    Student::~Student();
    linknode* creatStudent(int);           // 创建链表
    int Studentlen();                      // 返回链表长度
    nodetype* findnode(int);               // 通过查找序号返回节点的指针
    nodetype* find(char c[]);              // 通过查找姓名返回节点的指针
    int find2(char c[]);                   // 通过查找姓名返回节点的序号
    nodetype* insnode(int);                // 插入节点
    void delnode(int);                     // 删除节点
    nodetype* load();                      // 初始化：从外部读入数据
    void readstr(FILE *f,char *string);    // 读行函数
    bool check(char *a, char *b);          // 对比两个字符串是否相等
    void help();                           // 显示帮助菜单
    void editperson(nodetype*);            // 编辑个人信息
    void editscore(nodetype*);             // 编辑学科成绩
    void edittelephone(nodetype*);         // 编辑联系方式
    void dispname();                       // 显示所有学生姓名
    void dispnode(nodetype* p);            // 显示一个学生的所有信息
    void dispperson(nodetype*);            // 显示一个学生的个人说明
    void dispscore(nodetype*);             // 显示一个学生的学科成绩
    void disptelephone(nodetype*);         // 显示一个学生的联系方式
};
```

该类设计中只给出了成员函数的声明，具体定义则在类外进行。如果用类图来表示上述 Student 类，可画出如图 28.8 所示的类图。

图 28.8　学生类图

可以看出，该类包含了 head 指针作为其私有成员，公有成员函数则包括了对学生信息进行添加、删除、修改、读入和保存等一系列操作。在具体的使用中，通过如上的成员函数对学生类进行

相关操作，这也是面向对象程序设计的优势。

28.5 详细设计

在软件工程生命周期中，详细设计阶段主要将总体设计中的各个模块细化，以流程图的形式表示出来，使其能够在编码阶段快速地用代码体现。本节将讲解几个主要模块（添加学生、删除学生、编辑学生信息、读入学生信息和保存学生信息）的设计过程和编码实现。

28.5.1 创建链表

在具体介绍各个功能模块的实现之前，需要先建立链表，后续的添加、删除等操作都是在已有的链表上来进行的。

在创建链表过程中，可以根据指定的节点数，逆序地创建链表。每个节点在开始创建时，其 next 指针都为 NULL，直到将其指向下一个节点，用 C++语言表述该过程如下：

```
nodetype* Student::creatStudent (int n)    // 创建链表
{
    nodetype *h=NULL, *s, *t;           // 定义对象指针
    for(int i=1; i<n; i++)
    {
        if(i==1)                        // 创建第一个节点
        {
            h=(nodetype*)malloc(sizeof(nodetype)); // 分配空间
            h->next=NULL;               // 节点指针为空
            t=h;
        }
        Else                            // 创建其余节点
        {
            s=(nodetype*)malloc(sizeof(nodetype)); // 创建节点
            s->next=NULL;
            t->next=s;                  // 将节点加入链表
            t=s;                        // t 始终指向生成的单链表的最后一个节点
        }
        i++;                            // 节点个数累加
    }
    head=h;
    return h;                           // 返回头指针
}
```

上述成员函数判断用户指定的节点个数，如果个数为 1，则其 next 指针为 NULL，如果不为 1，则该节点的 next 指针要指向其下一个创建的节点，最后返回头指针。为了便于读者更好地理解上述函数的流程，下面给出该函数的执行流程图，如图 28.9 所示。

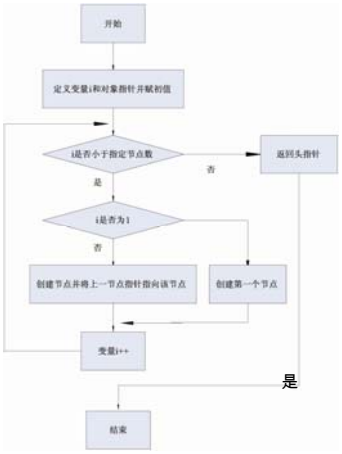


图 28.9　执行流程图

当在程序中调用该类的 creatStudent 成员函数创建链表后，系统将创建一个节点数为 n 的单向链表，链表的数据域可通过用户输入或读取已存在的数据文件来填充。

28.5.2　插入节点

所谓链表的插入是指在程序中动态地插入一个节点到链表中的特定位置。如果链表是有序的，那么插入该节点后，链表仍必须维持有序的状态。因此，此处可先建立一个节点，然后将该节点插入到指定的位置，实现代码如下：

```
nodetype* Student::insnode(int i)
{
    nodetype *h=head, *p, *s;
    s=(nodetype*)malloc(sizeof(nodetype)); // 创建节点 s
    s->next=NULL;
    if(i==0)                               // i=0 时 s 作为该单链表的第一个节点
    {
        s->next = h;
        h=s;                               // 重新定义头节点
    }
    else
    {
        p=findnode(i);                     // 查找第 i 个节点，并由 p 指向该节点
        if(p!=NULL)                        // 找到节点
        {
            s->next=p->next;               // 插入节点
            p->next=s;
        }
        else cout<<"输入的 i 值不正确"<<endl; // 未找到节点
    }
    head=h;
    return s;
}
```

可以看出，上述程序首先创建节点 s，如果需要将该节点插入到链表的开头，则重新定义头节点，否则将其插入到指定位置 i，并返回链表头指针，其流程如图 28.10 所示。

该程序采用了双分支结构实现，在判断指定位置不为 0 后还需查找该位置 i 的节点，这是因为要操作链表中的节点，每次都必须从头指针开始顺序遍历找到该节点，它不能像数组那样直接定义到第 i 个元素，这也是链表的不足之处。因此，上述程序使用到了查找第 i 个节点的函数 findnode，

该函数定义如下：

```
nodetype* Student::findnode (int i)          // 通过查找序号返回节点的指针
{
    nodetype* p=head;
    int j=1;
    if( i>Studentlen()||i<=0 )               // i 上溢或下溢
        return NULL;
    else
    {
        while( p!=NULL &&  j<i )             // 查找第 i 个节点并由 p 指向该节点
        {
            j++;
            p=p->next;                       // 指向下一个节点
        }
        return p;
    }
}
```

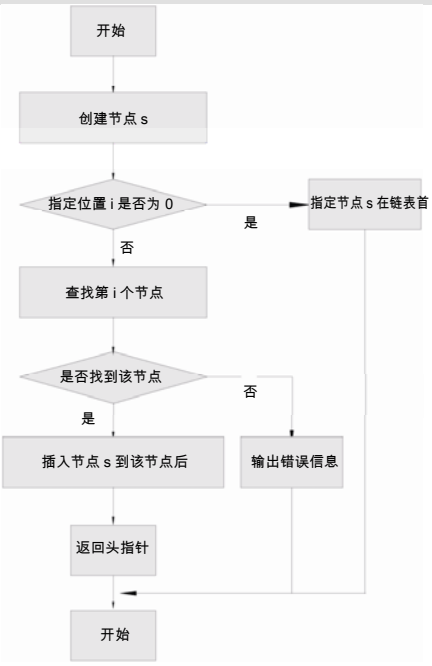


图 28.10　添加学生执行流程

函数 findnode 返回位置为 i 的节点的指针，便于在添加学生函数 insnode 中调用该指针，完成节点插入操作。其中，Studentlen 函数用于确定链表长度，即其节点个数，可以通过一个简单的遍历完成该操作。

```
int Student::Studentlen()                    // 返回链表长度
{
    int i=0;
    nodetype* p=head;                        // 定义对象指针并初始化
    while(p!=NULL)                           // 链表未结束
    {
        p=p->next;
        i++;                                 // 节点个数累加
    }
    return i;
}
```

至此，添加学生信息的成员函数的实现就完成了。

28.5.3 添加学生信息

添加学生信息是指输入新的学生信息，将这些信息写入到学生基本信息数据文件中。事实上，由于该学生管理系统采用的数据结构为单链表，所以添加学生模块所做的操作就是在已存在的链表中插入一个新的学生节点。

在生成节点并插入到指定位置后，用户需要输入该学生的基本信息到节点的数据域中，这就需要设计几个接收用户输入个人信息、学科成绩和联系方式的成员函数。其实现代码如下：

```
void Student::editperson(nodetype* p)          // 编辑学生个人信息函数
{
    char c[100];
    cout<<"请输入姓名: ";
    cin>>c;                                    // 接收用户输入姓名
    strcat(c, "\n");                           // 加上回车分隔符
    strcpy(p->pe.name, c);                     // 写入到链表节点数据域中
    cout<<"请输入性别：";
    cin>>c;                                    // 接收用户输入性别
    strcat(c, "\n");
    strcpy(p->pe.sex, c);                      // 写入到链表节点数据域中
    cout<<"请输入生日（格式举例：1982-1-1）: ";
    cin>>c;                                    // 接收用户输入生日
    strcat(c, "\n");
    strcpy(p->birthday, c);                    // 写入到链表节点数据域中
    cout<<"请输入民族：";
    cin>>c;                                    // 接收用户输入民族
    strcat(c, "\n");
    strcpy(p->pe.MZ, c);                       // 写入到链表节点数据域中
    cout<<"请输入国籍：";
    cin>>c;                                    // 接收用户输入国籍
    strcat(c, "\n");
    strcpy(p->pe.GJ, c);                       // 写入到链表节点数据域中
    cout<<"请输入学历：";
    cin>>c;                                    // 接收用户输入学历
    strcat(c, "\n");
    strcpy(p->pe.XL, c);                       // 写入到链表节点数据域中
    cout<<"请输入家庭住址";
    cin>>c;                                    // 接收用户输入地址
    strcat(c, "\n");
    strcpy(p->address, c);                     // 写入到链表节点数据域中
    cout<<"编辑个人信息完成!";
}
void Student::editscore(nodetype* p)           // 编辑学生学科成绩函数
{
    char a[50];
    cout<<"请输入学号: ";
    cin>>a;                                    // 接收用户输入学号
    strcat(a, "\n");
    strcpy(p->sc.num, a);                      // 写入到链表节点数据域中
    cout<<"请输入大学语文成绩: ";
```

```
    cin>>a;                                          // 接收用户输入语文成绩
    strcat(a, "\n");
    strcpy(p->sc.chinese, a);                        // 写入到链表节点数据域中
    cout<<"请输入英语成绩: ";
    cin>>a;                                          // 接收用户输入英语成绩
    strcat(a, "\n");
    strcpy(p->sc.english, a);                        // 写入到链表节点数据域中
    cout<<"请输入数学成绩: ";
    cin>>a;                                          // 接收用户输入数学成绩
    strcat(a, "\n");
    strcpy(p->sc.math, a);                           // 写入到链表节点数据域中
    cout<<"请输入物理成绩: ";
    cin>>a;                                          // 接收用户输入物理成绩
    strcat(a, "\n");
    strcpy(p->sc.physics, a);                        // 写入到链表节点数据域中
    cout<<"编辑学科成绩完成!";
}
void Student::edittelephone(nodetype* p)             // 编辑学生练习方式函数
{
    char c[50];
    cout<<"请输入手机号码: ";
    cin>>c;                                          // 接收用户输入手机号码
    strcat(c, "\n");
    strcpy(p->te.SJ, c);                             // 写入到链表节点数据域中
    cout<<"请输入家庭电话号码: ";
    cin>>c;                                          // 接收用户输入家庭电话
    strcat(c, "\n");
    strcpy(p->te.JD, c);                             // 写入到链表节点数据域中
    cout<<"请输入学校电话号码: ";
    cin>>c;                                          // 接收用户输入学校电话
    strcat(c, "\n");
    strcpy(p->te.XD, c);                             // 写入到链表节点数据域中
    cout<<"编辑联系方式完成!";
}
```

如上三个函数分别实现了学生的个人信息输入、学科成绩输入和联系方式输入，在主程序中只需直接调用即可。例如，简单的调用代码如下：

```
    cout<<"************** 添加一个学生信息 ******************"<<endl;
    cout<<"下面输入个人信息: "<<endl;
    L1.editperson(p);                                // 调用编辑学生个人信息成员函数
    cout<<"下面输入学科成绩: "<<endl;
    L1.editscore(p);                                 // 调用编辑学生学科成绩成员函数
    cout<<"下面输入联系方式: "<<endl;
```

```
    L1.edittelephone(p);                                    // 调用编辑学生联系方式成员函数
```

其中，L1 为一个学生对象，指针 p 指向链表中的当前节点。例如，需要在学生管理系统中添加一个学生姓名为“张三”、学号为“990001”的学生，并输入相关信息，其实现如图 28.11 所示。

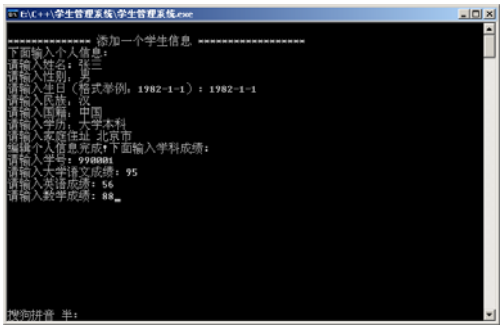


图 28.11　添加学生

28.5.4　显示学生信息

显示学生信息是指按照要求显示学生的个人信息、学科成绩或联系方式等信息。同样，要显示所有学生的信息，只需在链表中遍历所有节点即可。在学生管理系统中，学号和姓名是使用率比较高的，因此下面给出这两个字段的显示代码。

```
void Student::dispname()                                    // 显示所有学生姓名和学号
{
    nodetype* p=head;                                       // 定义对象指针
    cout<<"现有的学生:  "<<endl;
    if(p==NULL)                                             // 链表为空
        cout<<"没有任何学生数据"<<endl;
    while(p!=NULL)                                          // 链表未结束
    {
        cout<<"  姓  名  :  "<<p->pe.name<<"  学  号  :  "<<p->sc.num;   // 输出姓名和学号
        p=p->next;                                          // 指向下一节点
    }
}
```

该程序段通过一个 while 循环对整个链表的节点进行了遍历，将其姓名和学号输出。其执行流程如图 28.12 所示。

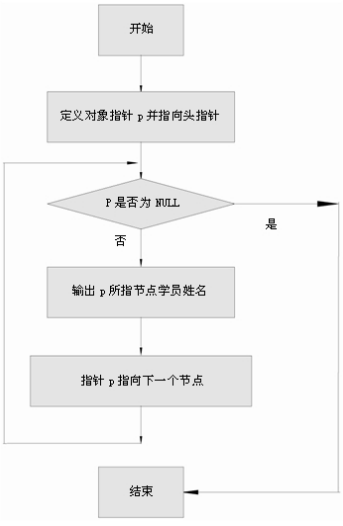


图 28.12　显示学生信息执行流程

当在主程序中调用该类的 dispname 成员函数后，系统执行上述程序，如果数据文件中已有学生信息，则返回其学号和姓名，否则返回空。例如，如图 28.13 所示为在主程序中调用该函数显示数据文件中全部学生学号和姓名信息的结果。

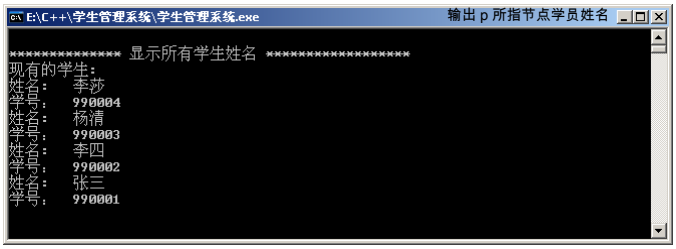


图 28.13 显示全部学生信息

28.5.5 读入学生信息

读入学生信息是指当数据文件中存在学生信息时，系统启动后自动从数据文件中将这些学生的信息读入到链表的节点中，供用户在学生管理系统中对其进行添加、删除和编辑等操作。显而易见，此处需要用到 C++中的文件输入输出流操作。

此处以文本文件为例，如果学生信息存储在文本文件中，每一行都为一个学生的数据，则可以通过调用文件读入函数 fgets 来实现，设计如下的 readstr 函数来读取信息：

```
void Student::readstr(FILE *f,char *string)
{
    do  // 先读入一行文本
    {
        // fgets():从文件 f 读入长度为 255-1 的字符串并存入到 string 中
        fgets(string, 255, f);
    } while ((string[0] == '/') || (string[0] == '\n'));
    return;
}
```

可以看出，上述函数只是从文本文件中读取数据，而这些数据是学生的哪一个数据项还不能确定，这就需要再设计一个函数，将读入的信息赋给具体的数据项，才可真正实现链表中的数据项为数据文件中对应的信息。该成员函数实现如下：

```
nodetype* Student::load()                    // 读入学生信息成员函数
{
    FILE *fp;                                // 定义文件指针
    nodetype *p;
    char c[255];
    int num;
    if((fp=fopen("student.txt", "r"))==NULL) // 以只读方式打开 student.txt 文件
    {
        cout<<"打开文件失败"<<endl;
        return 0;
    }
    readstr(fp, c);                          // 读取文件内容
    sscanf(c, "The Length Of Link: %d", &num); // 获取链表长度
    p=creatStudent(num);                     // 创建链表
    for(int i=0; i<num; i++)                 // 循环读取数据放入节点数据域中
    {
        readstr(fp, c);                      // 读取地址
```

```
        strcpy(p->address, c);                  // 存储到节点数据域中
        readstr(fp, c);                         // 读取生日
        strcpy(p->birthday, c);                 // 存储到节点数据域中
        readstr(fp, c);                         // 读取学号
        strcpy(p->sc.num, c);                   // 存储到节点数据域中
        readstr(fp, c);                         // 读取语文成绩
        strcpy(p->sc.chinese, c);               // 存储到节点数据域中
        readstr(fp, c);                         // 读取英语成绩
        strcpy(p->sc.english, c);               // 存储到节点数据域中
        readstr(fp, c);                         // 读取数学成绩
        strcpy(p->sc.math, c);                  // 存储到节点数据域中
        readstr(fp, c);                         // 读取物理成绩
        strcpy(p->sc.physics, c);               // 存储到节点数据域中
        readstr(fp, c);                         // 读取姓名
        strcpy(p->pe.name, c);                  // 存储到节点数据域中
        readstr(fp, c);                         // 读取性别
        strcpy(p->pe.sex, c);                   // 存储到节点数据域中
        readstr(fp, c);                         // 读取国籍
        strcpy(p->pe.GJ, c);
        readstr(fp, c);                         // 读取民族
        strcpy(p->pe.MZ, c);
        readstr(fp, c);                         // 读取学历
        strcpy(p->pe.XL, c);
        readstr(fp, c);                         // 读取手机号码
        strcpy(p->te.SJ, c);
        readstr(fp, c);                         // 读取家庭电话号码
        strcpy(p->te.JD, c);
        readstr(fp, c);                         // 读取单位电话号码
        strcpy(p->te.XD, c);
        p=p->next;                              // 指向下一个节点
    }
    fclose(fp);
    return p;
}
```

上述函数代码看起来较多，事实上其结构非常简单，即将读入的每一行数据赋值给节点的特定数据域，直到数据文件中所有信息都读取完成为止。其执行流程如图 28.14 所示。

当在主程序 main 中调用该成员函数后，会在打开的数据文件 student.txt 中读入所有的数据到链表中，读取完成后，读者可以试着调用 dispname 函数将其信息显示出来。例如，在学生管理系统的当前目录下有 student.txt 文件，其中数据信息如图 28.15 所示。

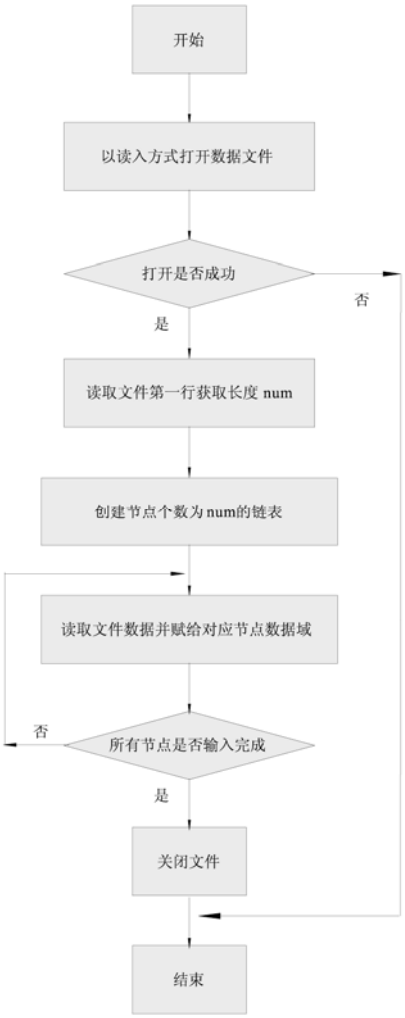


图 28.14　读入学生信息执行流程

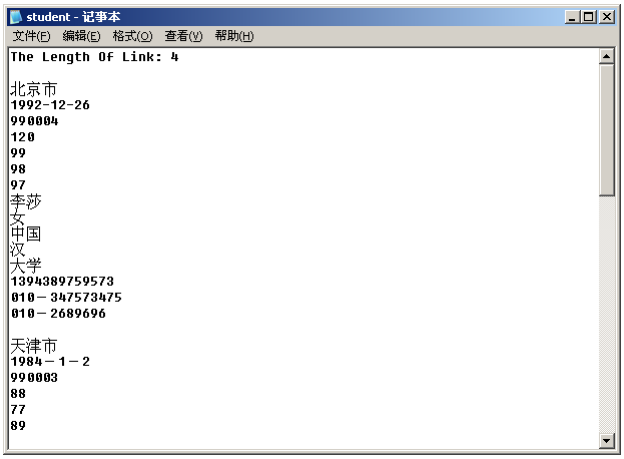


图 28.15　数据文件

在主程序中调用 load 函数后将这些信息读入到程序中，再调用 dispperson 函数将其个人信息

全部显示出来，其实现代码如下：

```
p=L1.load();                                    // 调用读入信息成员函数
dispname();
```

其中，函数 dispname 为前面小节定义的，用循环输出所有学生学号和姓名的信息。通过如上代码调用后，其执行结果如图 28.16 所示。

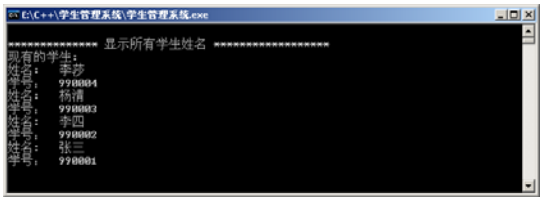


图 28.16　显示学生信息

28.5.6 编辑学生信息

编辑学生信息模块实现的主要功能是对指定的学生进行其个人信息、学科成绩和联系方式的修改和更新，一般需要提供具有唯一性的关键字来指定某个学生，此处采用姓名作为指定学生的关键字。当需要编辑一个学生的信息时，用户可能需要修改学生的个人信息、学科成绩或联系方式中的任意一个字段，因此，此处可以设计一个简单的菜单供用户选择。

基于如上分析，在编辑学生信息的成员函数中画出一个包含选择功能的菜单，而编辑学生的三类信息的实现函数 editperson，editscore 和 edittelephone 在 28.5.3 节中已经具体定义了，在编辑学生信息函数中只需直接调用。因此，该函数 editstudent 的实现代码如下：

```
void Student::editstudent(nodetype* p)          // 编辑学生信息成员函数
{
    Student L1;                                 // 创建 Student 对象
    char c[20];
    cout<<endl;
    cout<<"************** 根据姓名对单个学生进行编辑 ******************"<<endl;
    L1.dispname();                              // 显示所有学生姓名和学号
    cout<<"请输入学生姓名: "<<endl;
    cin>>c;                                     // 接收用户输入姓名
    p=L1.find(c);                               // 查找指定姓名的学生
    cout<<endl<<endl;                           // 画出菜单
    cout<<"*********************************************************"<<endl;
    cout<<"1:          编辑个人信息"<<endl;
    cout<<"2:          编辑学科成绩"<<endl;
    cout<<"3:          编辑联系方式"<<endl;
    cout<<"4:          显示个人信息"<<endl;
    cout<<"5:          显示学科成绩"<<endl;
    cout<<"6:          显示联系方式"<<endl;
    cout<<"7:          显示该学生所有信息"<<endl;
```

```
cout<<"8:           帮助菜单"<<endl;
cout<<"9:           返回上一级菜单"<<endl;
cout<<"*****************************************************"<<endl;
while(1)
{
    cout<<endl<<endl;                              // 换行
    cout<<"请输入选择(帮助选项--> 8 ): "<<endl;
    cin>>c;                                        // 输入菜单选项
    system("cls");                                 // 清屏
    if(L1.check(c, "1"))                           // 选择编辑个人信息菜单项
    {
        system("cls");
        cout<<endl;
        cout<<"************** 编辑个人信息 ******************"<<endl;
        L1.editperson(p);                          // 调用编辑个人信息成员函数
    }
    else if(L1.check(c, "2"))                      // 选择编辑学科成绩菜单项
    {
        system("cls");
        cout<<endl;
        cout<<"************** 编辑学科成绩 ******************"<<endl;
        L1.editscore(p);                           // 调用编辑学科成绩成员函数
    }
    else if(L1.check(c, "3"))                      // 选择编辑联系方式菜单项
    {
        system("cls");
        cout<<endl;
        cout<<"************** 编辑联系方式 ******************"<<endl;
        L1.edittelephone(p);                       // 调用编辑联系方式成员函数
    }
    else if(L1.check(c, "4"))                      // 选择显示个人信息菜单项
    {
        system("cls");
        cout<<endl;
        cout<<"************** 显示个人信息 ******************"<<endl;
        L1.dispperson(p);                          // 调用显示个人信息成员函数
    }
    else if(L1.check(c, "5"))                      // 选择显示学科成绩菜单项
    {
        system("cls");
        cout<<endl;
        cout<<"************** 显示学科成绩 ******************"<<endl;
        L1.dispscore(p);                           // 调用显示学科成绩成员函数
    }
    else if(L1.check(c, "6"))                      // 选择显示联系方式菜单项
    {
        system("cls");
        cout<<endl;
```

```
            cout<<"************** 显示联系方式 ******************"<<endl;
            L1.disptelephone(p);                    // 调用显示联系方式成员函数
        }
        else if(L1.check(c, "7"))                   // 选择显示所有信息菜单项
        {
            system("cls");
            L1.dispnode(p);                         // 调用显示所有信息成员函数
        }
        else if(L1.check(c, "8"))                   // 选择显示帮助信息菜单项
        {
            system("cls");
            L1.help();                              // 调用帮助成员函数
        }
        else if(L1.check(c, "9"))                   // 选择返回上一级菜单项
        {
            display();                              // 调用显示主菜单成员函数
            break;                                  // 用 break 跳出本循环
        }
    }
}
```

上述函数首先通过输入输出流 cout 函数画出了包含多个选项的菜单，再通过一个循环供用户多次进行菜单选择，同时在循环中使用了多重分支，当用户选择不同的菜单后调用不同的成员函数来实现指定的功能。其执行流程如图 28.17 所示。

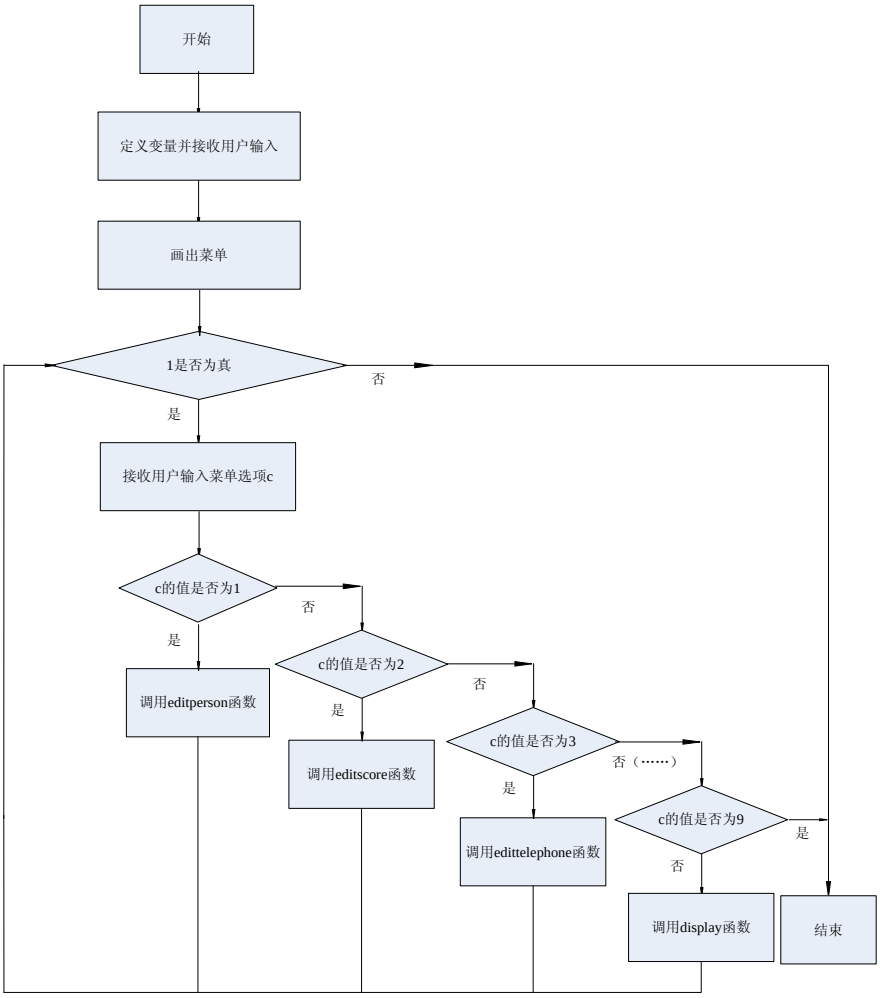


图 28.17　编辑学生信息执行流程

可以看出，编辑学生信息函数通过一个多重分支语句调用了众多前面定义的功能函数。初次运行该函数时，它会画出菜单并显示当前数据文件中存在的学生姓名和学号，如图 28.18 所示。

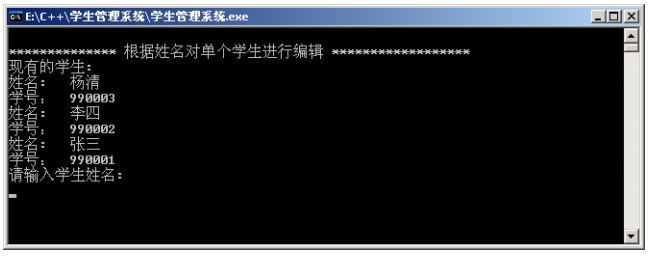


图 28.18　编辑学生信息主界面

例如，在上图中需要将学生姓名为“张三”的学生的学号改为“991111”，其各科成绩均置为 0，调用该函数后，在输入学生姓名处输入“张三”，在弹出的如图 28.19 所示的菜单界面中选择“2：编辑学科成绩”选项，即可重新输入该学生的学号和所有成绩，重新输入完成后其执行结果如图 28.20 所示。

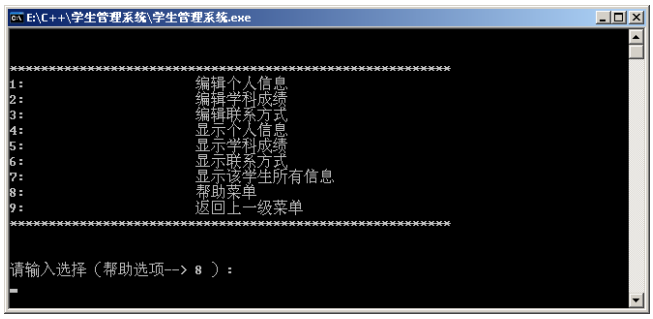


图 28.19　选择菜单

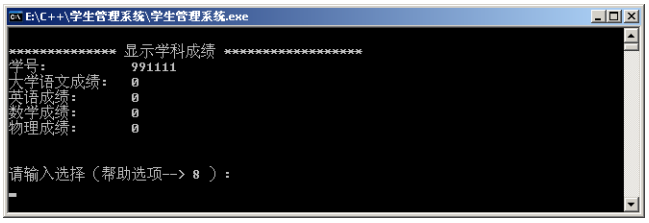


图 28.20　编辑完成

28.5.7 删除学生信息

在学生管理系统中，删除一个或多个学生的信息即将该学生的所有个人信息、学科成绩和联系方式等一起删除。在链表数据结构中，只需将该学生对应的节点从链表中删除，并释放该节点内存空间即可。因此，删除学生信息的实现代码如下：

```
void Student::delnode(int i)            // 删除第 i 个节点
{
    nodetype *h=head, *p=head, *s;
    if(i==1)                            // 删除第一个节点
    {
        h=h->next;                      // 断开节点
        free(p);                        // 释放节点空间
    }
    else
    {
        p=findnode(i-1);                // 查找第 i-1 个节点，并由 p 指向这个节点
        if(p!=NULL && p->next!=NULL)    // 找到节点
        {
            s=p->next;                  // s 指向要删除的节点
            p->next=s->next;            // 断开节点
            free(s);                    // 释放节点空间
        }
        else                            // 未找到该节点
            cout<<"输入的 i 值不正确"<<endl;
    }
    head=h;
}
```

上述成员函数删除第 i 个节点，如果 i 不为 1，需要调用前面定义的 findnode 成员函数找到第 i-1 个节点，即找到待删除节点的前一个节点，将其指针指向第 i 个节点的下一个节点，就将第 i 个节点从链表中断开了。最后将第 i 个节点的内存空间释放，就完成了删除操作。根据如上代码和分析，其执行流程如图 28.21 所示。

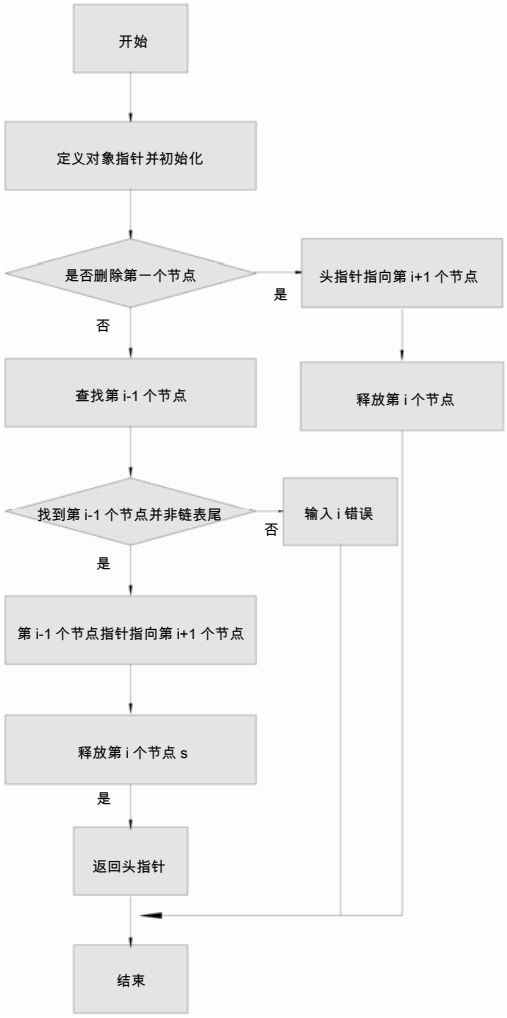


图 28.21 删除学生信息执行流程

可以看出，由于使用链表作为数据结构，某一个学生信息的删除事实上是链表中对应节点的删除。例如，将学号为 990004、姓名为“李莎”的学生删除，在主程序中通过如下代码实现：

```
cout<<"请输入学生姓名: "<<endl;
cin>>ch;
i=L1.find2(ch);                          // 查找指定节点
L1.delnode(i);                           // 调用删除节点成员函数
```

其中，函数 find2 用于找出姓名为“李莎”的学生在链表中节点的位置，并将其值赋给变量 i，再调用 delnode 函数删除该节点，删除后再调用 dispname 函数，其实现结果如图 28.22 所示。

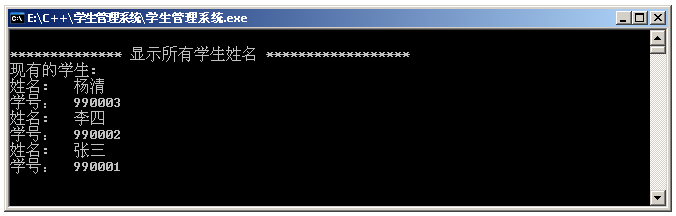


图 28.22 删除学生信息结果

28.5.8 保存学生信息

与读入学生信息成员函数相对应 ,保存学生信息用于将修改后的链表节点数据域写入到数据文件 student.txt 中。根据前面章节的学习，读者知道文件必须以写方式打开，并通过文件输入输出函数实现写入操作，其实现代码如下：

```
nodetype* Student::save()                    // 保存学生信息成员函数
{
    FILE *fp;                                // 定义文件指针
    if((fp=fopen("student.txt", "w"))==NULL)   // 以写方式打开 student.txt 文件
    {
        cout<<"打开文件失败"<<endl;
        return;
    }
    int i;
    char t[255];
    sprintf(t, "The Length Of Link: %d\n", L1.Studentlen());
                                             // 将 L1.Studentlen() 赋予字符串中的数字
    fputs(t, fp);                            // 写入数字
    strcpy(t, "\n");
    fputs(t, fp);
    p=L1.findnode(1);                        // 将链表头指针赋予 p
    for(i=0; i<L1.Studentlen(); i++)         // 循环写入所有节点
    {
        fputs(p->address, fp);               // 输出地址
        fputs(p->birthday, fp);              // 输出生日
        fputs(p->sc.num, fp);                // 输出学号
        fputs(p->sc.chinese, fp);            // 输出语文成绩
        fputs(p->sc.english, fp);            // 输出英语成绩
        fputs(p->sc.math, fp);               // 输出数学成绩
        fputs(p->sc.physics, fp);            // 输出物理成绩
        fputs(p->pe.name, fp);               // 输出姓名
        fputs(p->pe.sex, fp);                // 输出性别
        fputs(p->pe.GJ, fp);                 // 输出国籍
        fputs(p->pe.MZ, fp);                 // 输出民族
        fputs(p->pe.XL, fp);                 // 输出学历
```

```
        fputs(p->te.SJ, fp);            // 输出手机
        fputs(p->te.JD, fp);            // 输出家庭电话
        fputs(p->te.XD, fp);            // 输出学校电话
        fputs(t, fp);
        p=p->next;                      // 执行下一个节点
    }
    p=head;                             // p 指向头指针
    fclose(fp);                         // 关闭文件
    return p;                           // 返回头指针
}
```

可以看出，保存学生信息的成员函数与读入学生信息的成员函数类似，其执行流程也是相似的，区别在于保存学生信息的函数需要对数据文件进行写操作。例如，将上述删除学生“李莎”的结果保存到数据文件 student.txt 中后，打开该文件，读者可以发现该学生已被删除，如图 28.23 所示。

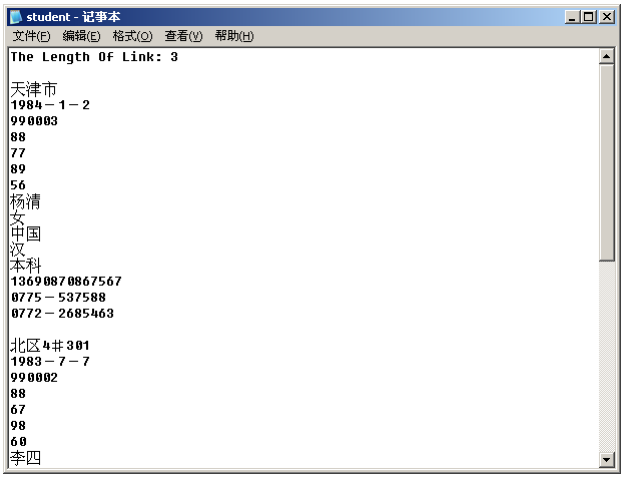


图 28.23 保存学生信息结果

28.6 系统集成

经过如上的介绍，学生管理系统的各个模块均已设计完成。一般来说，将系统的各功能模块全部完成后，即可将其各部分集成到主窗体上来，以便用户在主窗体中操作所有功能。如果有主菜单，即将其添加到主菜单的事件驱动上。

28.6.1 设计菜单

与可视化程序或基于窗体的应用程序不同的是，Console Application 应用程序不能设计出类似 Windows 窗体的菜单。因此，学生管理系统的菜单以数字和字符的形式画出，根据用户输入不同的数字选择相应的功能模块。

根据如上分析，结合前面小节实现的各个功能，通过输入输出流 cout 在应用程序中画出菜单，

以函数形式实现的代码如下：

```
void Operater::display()                          // 显示主菜单成员函数
{
    cout<<endl<<endl;                             // 画出菜单
    cout<<"********************** 学生管理系统 ********************"<<endl;
    cout<<"1:           添加一个学生信息"<<endl;
    cout<<"2:           删除一个学生信息"<<endl;
    cout<<"3:           显示所有学生的姓名"<<endl;
    cout<<"4:           根据姓名显示单个学生所有信息"<<endl;
    cout<<"5:           根据姓名对单个学生进行编辑"<<endl;
    cout<<"6:           帮助菜单"<<endl;
    cout<<"7:           保存数据"<<endl;
    cout<<"0:           退出系统"<<endl;
    cout<<"*************************************************"<<endl;
}
```

在主程序中直接调用 display 函数后即可在应用程序中画出该菜单，这就是学生管理系统应用程序运行后的主界面，如图 28.24 所示。

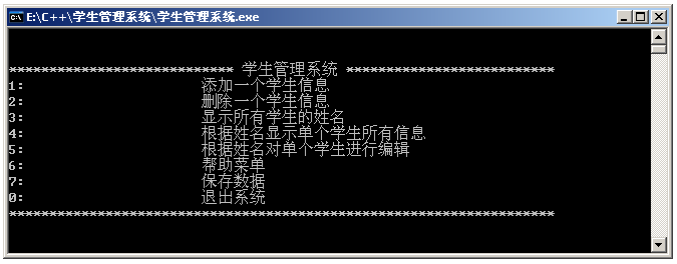


图 28.24　主菜单

28.6.2 绑定菜单功能

设计好主菜单后，应用程序并不能选择这些菜单实现相应功能，要实现选择某一个菜单项实现指定功能，需要为这些菜单项逐个绑定相应的类的成员函数。此处也可以通过循环语句内置多分支语句来实现绑定，具体实现代码如下：

```
void Operater::Loop()                    // 绑定菜单功能成员函数
{
    Student L1;                          // Student 对象
    char ch[20];
    nodetype *p, *head;
    int i;                               // 存放节点序号
    p=L1.load();                         // 初始化：从外部读入数据创建链表
    head=p;
    display();                           // 显示主菜单
    while(1)
    {
        cout<<endl<<endl;
```

```
cout<<"请输入选择（帮助选项--> 6 ）: "<<endl;
cin>>ch;                          // 接收用户输入菜单选项
system("cls");
if(L1.check(ch, "1"))             // 选择添加学生菜单项
{
    p=L1.insnode(0);
    head=p;
    system("cls");
    cout<<endl;
    cout<<"************** 添加一个学生信息 ******************"<<endl;
    cout<<"下面输入个人信息: "<<endl;
    L1.editperson(p);
    cout<<"下面输入学科成绩: "<<endl;
    L1.editscore(p);
    cout<<"下面输入联系方式: "<<endl;
    L1.edittelephone(p);
}
if(L1.check(ch, "2"))                 // 选择删除学生菜单项
{
    system("cls");
    cout<<endl;
    cout<<"************** 删除一个学生信息 ******************"<<endl;
    L1.dispname();
    cout<<"请输入学生姓名: "<<endl;
    cin>>ch;
    i=L1.find2(ch);
    L1.delnode(i);
}
if(L1.check(ch, "3"))                 // 选择显示所有学生姓名菜单项
{
    system("cls");
    cout<<endl;
    cout<<"************** 显示所有学生姓名 ******************"<<endl;
    L1.dispname();
}
if(L1.check(ch, "4"))                 // 选择显示单个学生菜单项
{
    system("cls");
    cout<<endl;
    cout<<"********* 根据姓名显示单个学生所有信息 *************"<<endl;
    L1.dispname();
    cout<<"请输入学生姓名: "<<endl;
    cin>>ch;
    p=L1.find(ch);
    L1.dispnode(p);
}
if(L1.check(ch, "5"))                 // 选择编辑学生菜单项
{
    L1.editstudent(p);
}
if(L1.check(ch, "6"))                 // 选择帮助菜单项
```

```
        {
            help();
        }
        if(L1.check(ch, "7"))              // 选择保存菜单项
        {
            L1.save();
        }
        else if(L1.check(ch, "0"))         // 选择退出菜单项
            return;
    }
}
```

上述程序中，函数 check 用于对比用户输入的字符选择的是哪个菜单项。事实上，此处可以用“==”来实现。同样，根据前面多分支语句的讲解，此处的结构可以通过 switch 语句来实现，读者可自行将上述程序中的 if…else 结构用 switch 结构进行改写。同样，为了方便读者理解，下面给出该函数的执行流程图，如图 28.25 所示。

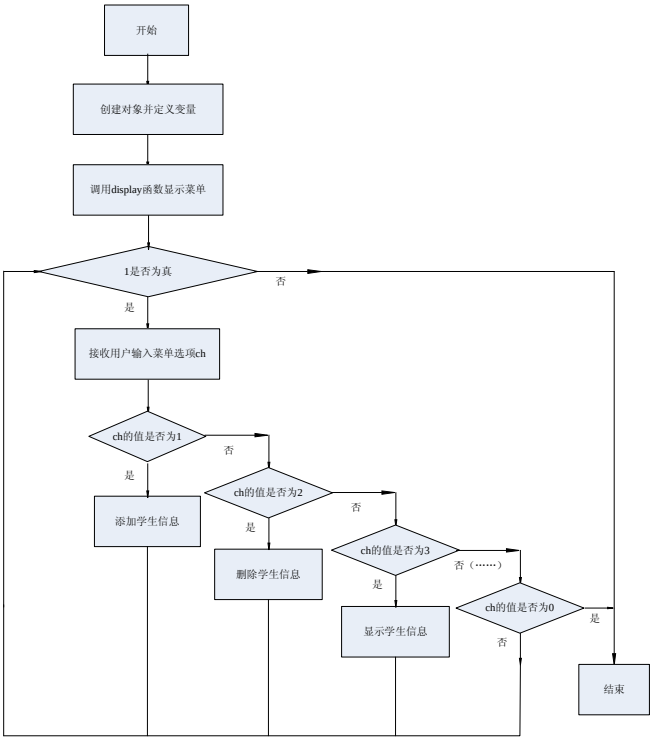


图 28.25　系统集成执行流程

其中，帮助系统 help 函数用于显示每个菜单项所实现的功能，它只需提供显示信息，而不需要提供具体功能。因此，可以设计实现代码如下：

```
void Student::help()                    // 帮助成员函数
{
    cout<<endl<<endl;                   // 画出帮助项
    cout<<"*****************************************************"<<endl;
    cout<<"1:          编辑个人信息"<<endl;
    cout<<"2:          编辑学科成绩"<<endl;
    cout<<"3:          编辑联系方式"<<endl;
    cout<<"4:          显示个人信息"<<endl;
```

```
    cout<<"5:           显示学科成绩"<<endl;
    cout<<"6:           显示联系方式"<<endl;
    cout<<"7:           显示该学生所有信息"<<endl;
    cout<<"8:           帮助菜单"<<endl;
    cout<<"9:           返回上一级菜单"<<endl;
    cout<<"*****************************************************"<<endl;
}
```

28.6.3 设计主函数

完成上节中的菜单设计和菜单功能绑定工作后，就可以在主函数中调用主菜单，完成整个应用系统的设计了。

根据面向对象的程序设计思想，所有操作都是以对象来进行的。因此，在主函数中创建一个学生对象，并用该对象调用其中的 Loop 成员函数，即可显示学生管理系统绑定功能后的主菜单，通过其中菜单项的选择再调用该对象的其他成员函数，实现相应的功能。因此，可简单地设计主函数如下：

```
int main()                                  // 主函数
{
    Student Stu;                            // 创建对象
    Stu.Loop();                             // 调用主程序成员函数
}
```

至此，整个学生管理系统的设计就完成了。用户运行该应用程序后，将打开如图 28.26 所示的程序主界面，通过输入主界面中的菜单项可实现不同的功能。

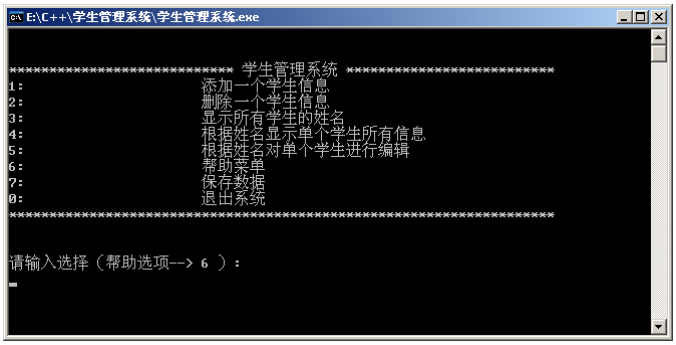


图 28.26 学生管理系统主界面

28.7 小结

本章通过一个简单的学生管理系统的实现，从软件工程生命周期的角度，分别从需求分析、总体设计、数据结构设计和详细设计介绍了其实现。为提高读者对面向对象程序设计思想的理解，该实例以类的设计和成员函数的设计为主，以链表的节点存储学生信息，方便地实现了学生节点的添加、删除和编辑等功能。学习完本章后，读者应该具有了一定的使用 C++实现一个具体的数据库应用程序的分析和实践能力。

反侵权盗版声明