

前 言

程序设计是计算机专业十分重要的一门课程，是实践性非常强的一门课程，也应该是一门非常有趣、让学生很有成就感的课程。但在教学过程中，很多学生的反应是：听懂了，但不会做，以至于最后丧失了兴趣。我认为主要的问题是教学过程中过分重视程序设计语言本身，过分强调理解语言的语法，而没有把思路放在解决问题的方法上面。

本书是作者根据多年来在上海交通大学计算机系讲授“程序设计”课程的经验，参考了近年来国内外主要的程序设计教材编写而成的。本书全面介绍结构化程序设计和面向对象程序设计两方面内容，书中秉承以程序设计方法为主、程序设计语言为辅的思想，采用以问题求解引出知识点的方法，在介绍语言要素的同时，更多地强调编程思想。本书的目标是让学生先了解学习的目的，提高学习兴趣，最后能利用学到的知识解决某一应用领域的问题。

C++是业界非常流行的语言，它既支持过程化的程序设计，又支持面向对象的程序设计，可以很好地体现程序设计的思想和方法。因为本书的主旨是强调程序设计的思想，所以选择C++语言作为教学语言恰好服务于这个目标。但是，因为本书以介绍基本的程序设计思想、概念和方法为基础，强调算法、抽象等重要的程序设计技术，所以对于C++的某些特殊成分和技巧将不予重点介绍。

本书内容大体分为两大部分：第1章到第9章为第一部分，它们主要介绍一些基本的程序设计思想、概念、技术、良好的程序设计风格（书中涉及程序设计风格的内容均对其加了波浪线做突出处理）以及过程化程序设计，包括数据类型、控制结构、数据封装、过程封装以及各种常用的算法等；第10章到第16章为第二部分，重点介绍面向对象的思想，包括如何设计及实现一个类，如何利用组合和继承实现代码的重用，如何利用多态性使程序更加灵活，如何利用抽象类制定一些工具的规范，最后为了更好地与数据结构课程衔接，介绍了容器和迭代器的概念。

本书得以顺利地编写和出版首先要感谢上海交通大学计算机系的俞勇教授，是他的鼓励使我有勇气把我的这些经验提供给大家参考；我还要感谢上海交通大学电信学院程序设计课程组的各位老师，与他们经常在一起讨论使我不断加深了对程序设计的理解；我还要感谢可爱的学生们，是他们与我在课上和课后的互动使我了解他们的困惑，清楚他们学习的难点。

由于作者水平有限，本书可能存在很多不足，敬请读者批评指正。

目 录

第 1 章 绪论	1	2.7.1 赋值运算符	25
1.1 计算机硬件	1	2.7.2 赋值时的自动类型转换	25
1.2 计算机软件	2	2.7.3 赋值的嵌套	26
1.3 算法与程序设计	3	2.7.4 多重赋值	26
1.4 程序的编译和调试	3	2.7.5 复合赋值运算	27
小结	5	2.8 自增和自减运算符	28
习题	5	2.9 强制类型转换	29
第 2 章 通过例子学习	6	2.10 数据的输入/输出	29
2.1 第一个程序：输出Hello world.	6	2.10.1 数据的输入	29
2.1.1 注释	6	2.10.2 数据的输出	30
2.1.2 编译预处理	7	2.11 构思一个程序	31
2.1.3 主程序	8	2.11.1 程序设计风格	31
2.1.4 名字空间	9	2.11.2 设计将来的修改	32
2.2 程序示例：计算圆的面积和周长	9	小结	32
2.3 变量定义	11	习题	32
2.4 数据类型	12	第 3 章 逻辑思维及分支程序设计	35
2.4.1 整型	12	3.1 关系运算	35
2.4.2 实型	14	3.1.1 关系运算符	35
2.4.3 字符型	15	3.1.2 关系表达式	35
2.4.4 布尔型	19	3.2 逻辑运算	36
2.4.5 枚举类型	19	3.3 if语句	39
2.4.6 用typedef重新命名类型名	21	3.3.1 if语句的形式	39
2.4.7 定义新的类型	21	3.3.2 if语句的嵌套	40
2.4.8 变量赋初值	21	3.3.3 if语句的应用	40
2.4.9 用sizeof了解占用的内存量	22	3.3.4 条件表达式	42
2.5 符号常量	22	3.4 switch语句及其应用	43
2.6 算术运算	22	小结	48
2.6.1 主要的算术运算符	23	习题	49
2.6.2 各种类型的数值间的混合运算	23	第 4 章 循环控制	50
2.6.3 整数除法和取模运算符	23	4.1 for循环	50
2.6.4 优先级	24	4.1.1 重复n次操作	50
2.6.5 数学函数库	24	4.1.2 for语句的进一步讨论	52
2.7 赋值运算	25	4.1.3 for循环的嵌套	53

4.2	while循环	53	6.5	内联函数	96
4.3	do-while循环	57	6.6	重载函数	97
4.4	循环的中途退出	57	6.7	函数模板	99
4.5	枚举法	58	6.8	变量的作用域	100
4.6	贪婪法	60	6.9	变量的存储类别	101
小结		62	6.9.1	自动变量	102
习题		62	6.9.2	静态变量	102
第5章	批量数据处理——数组	64	6.9.3	寄存器变量	103
5.1	一维数组	64	6.9.4	外部变量	104
5.1.1	一维数组的定义	64	6.10	递归函数	106
5.1.2	数组元素的引用	64	6.10.1	递归函数的基本概念	106
5.1.3	一维数组的初始化	65	6.10.2	递归函数的应用	108
5.1.4	一维数组在内存中的表示	65	6.11	基于递归的算法	113
5.1.5	一维数组的应用	66	6.11.1	回溯法	113
5.2	查找和排序	67	6.11.2	分治法	116
5.2.1	查找	67	6.11.3	动态规划	119
5.2.2	排序	72	小结		122
5.3	二维数组	75	习题		122
5.3.1	二维数组的定义	75	第7章	间接访问——指针	124
5.3.2	二维数组的初始化	75	7.1	指针的概念	124
5.3.3	二维数组在内存中的表示	76	7.1.1	指针变量的定义	125
5.3.4	二维数组的应用	76	7.1.2	指针的基本操作	125
5.4	字符串	79	7.2	指针与数组	128
5.4.1	字符串的存储及初始化	79	7.2.1	指针运算	129
5.4.2	字符串的输入/输出	80	7.2.2	用指针访问数组	131
5.4.3	字符串处理函数	80	7.2.3	数组名作为函数的参数	132
5.4.4	字符串的应用	81	7.3	指针与动态分配	133
小结		82	7.3.1	动态变量的创建	134
习题		82	7.3.2	动态变量的回收	134
第6章	过程封装——函数	84	7.3.3	内存泄漏	135
6.1	自己编写一个函数	84	7.3.4	查找new操作的失误	135
6.1.1	return语句	85	7.4	字符串再讨论	136
6.1.2	函数示例	85	7.5	指针与函数	137
6.2	函数的使用	87	7.5.1	指针作为形式参数	137
6.2.1	函数原型的声明	87	7.5.2	返回指针的函数	140
6.2.2	函数的调用	88	7.5.3	引用与引用传递	141
6.2.3	将函数与主程序放在一起	89	7.5.4	返回引用的函数	143
6.2.4	函数调用过程	90	7.6	指针数组与多级指针	144
6.3	数组作为函数的参数	92	7.6.1	指针数组	144
6.4	带默认值的函数	95	7.6.2	main函数的参数	145

7.6.3 多级指针.....	146	10.3.2 对象的操作.....	199
7.7 多维数组和指向数组的指针.....	147	10.3.3 this指针.....	201
7.8 指向函数的指针.....	148	10.3.4 对象的构造与析构.....	201
小结.....	152	10.4 常量对象与常量成员函数.....	207
习题.....	152	10.5 常量数据成员.....	208
第8章 数据封装——结构体	154	10.6 静态数据成员与静态成员函数.....	208
8.1 记录的概念.....	154	10.6.1 静态数据成员的定义.....	209
8.2 C++语言中记录的使用.....	155	10.6.2 静态成员函数.....	209
8.2.1 结构体类型的定义.....	155	10.6.3 静态常量成员.....	212
8.2.2 结构体类型的变量的定义.....	156	10.7 友元.....	213
8.2.3 结构体变量的使用.....	157	小结.....	215
8.2.4 结构体数组.....	158	习题.....	215
8.3 结构体作为函数的参数.....	160	第11章 运算符重载	216
8.4 链表.....	162	11.1 什么是运算符重载.....	216
8.4.1 链表的概念.....	162	11.2 运算符重载的方法.....	216
8.4.2 单链表的存储.....	163	11.3 几个特殊运算符的重载.....	219
8.4.3 单链表的操作.....	164	11.3.1 赋值运算符的重载.....	219
小结.....	169	11.3.2 下标运算符的重载.....	221
习题.....	169	11.3.3 ++和--运算符的重载.....	221
第9章 模块化开发	171	11.3.4 重载函数的原型设计考虑.....	223
9.1 自顶向下分解.....	171	11.3.5 输入/输出运算符的重载.....	224
9.1.1 顶层分解.....	172	11.4 自定义类型转换函数.....	225
9.1.2 prn_instruction的实现.....	172	11.5 运算符重载的应用.....	226
9.1.3 play函数的实现.....	173	11.5.1 完整的Rational类的定义和 使用.....	226
9.1.4 get_call_from_user的 实现.....	173	11.5.2 完整的IntArray类的定义 和使用.....	228
9.2 模块划分.....	174	小结.....	231
9.3 设计自己的库.....	180	习题.....	231
小结.....	185	第12章 组合与继承	232
习题.....	185	12.1 组合.....	232
第10章 创建功能更强的类型—— 类的定义与使用	187	12.2 继承.....	234
10.1 从过程化到面向对象.....	187	12.2.1 单继承.....	235
10.1.1 抽象的过程.....	187	12.2.2 基类成员在派生类中的访问 特性.....	235
10.1.2 面向对象程序设计的特点.....	188	12.2.3 派生类对象的构造、析构与 赋值操作.....	237
10.1.3 库和类.....	189	12.2.4 重定义基类的函数.....	241
10.2 类的定义.....	195	12.2.5 派生类作为基类.....	243
10.3 对象的使用.....	198	12.2.6 将派生类对象隐式转换为基	
10.3.1 对象的定义.....	198		

类对象.....	244		
12.3 多态性与虚函数.....	246	14.3.2 输入流.....	268
12.3.1 多态性.....	246	14.3.3 格式化的输入/输出.....	271
12.3.2 虚函数.....	246	14.4 基于文件的输入/输出.....	275
12.3.3 虚析构函数.....	249	14.4.1 文件的概念.....	275
12.4 纯虚函数和抽象类.....	249	14.4.2 文件和流.....	275
12.4.1 纯虚函数.....	249	14.4.3 文件的顺序访问.....	278
12.4.2 抽象类.....	250	14.4.4 文件的随机处理.....	280
12.5 多继承.....	250	14.4.5 用流式文件处理含有记录的 文件.....	282
12.5.1 多继承的格式.....	250	14.5 基于字符串的输入/输出.....	287
12.5.2 名字冲突.....	251	小结.....	288
12.5.3 虚基类.....	252	习题.....	288
小结.....	252	第 15 章 异常处理.....	289
习题.....	253	15.1 传统的异常处理方法.....	289
第 13 章 泛型机制——模板.....	254	15.2 异常处理机制.....	289
13.1 类模板的定义.....	254	15.2.1 异常抛出.....	290
13.2 类模板的实例化.....	256	15.2.2 异常捕获.....	291
13.3 模板的编译.....	256	15.3 异常规格说明.....	293
13.4 非类型参数和参数的默认值.....	257	小结.....	294
13.5 类模板的友元.....	258	习题.....	295
13.5.1 普通友元.....	258	第 16 章 容器和迭代器.....	296
13.5.2 模板的特定实例的友元.....	258	16.1 容器.....	296
13.5.3 声明的依赖性.....	259	16.2 迭代器.....	296
13.6 类模板作为基类.....	262	16.3 容器和迭代器的设计示例.....	296
小结.....	263	16.3.1 用数组实现的容器.....	297
习题.....	263	16.3.2 用链表实现的容器.....	299
第 14 章 输入/输出与文件.....	264	小结.....	302
14.1 流与标准库.....	264	习题.....	302
14.2 输入/输出缓冲.....	265	附录.....	303
14.3 基于控制台的输入/输出.....	266	参考文献.....	304
14.3.1 输出流.....	266		

图书在版编目（CIP）数据

JRuby 实战 / (瑞典) 宾尼 (Bini, O.) 著; 丁雪丰译.
—北京: 人民邮电出版社, 2008.8
(图灵程序设计丛书)
书名原文: Practical JRuby on Rails Web 2.0 Projects
ISBN 978-7-115-18375-0

I. J… II. ①宾…②丁… III. ① JAVA 语言—程序设计
②计算机网络—程序设计 IV. TP312 TP393.09

中国版本图书馆CIP数据核字 (2008) 第091945号

内 容 提 要

本书通过 4 个由浅入深的项目, 结合 Rails 向读者全面介绍了 JRuby。内容包括: 如何在 Ruby 中调用 Java 代码, 如何使用 Java 库, 如何实现并访问 EJB, 如何操作 JMS, 如何在 Java 中调用由 Ruby 实现的 Java 类和接口等。同时, 书中给出的代码都很有实用价值, 只需稍做加工就能放进真正的项目中发挥作用。

本书适合各层次 Java Web 开发人员阅读和参考。

图灵程序设计丛书

JRuby实战

◆ 著 [瑞典] Ola Bini
译 丁雪丰
责任编辑 杨 爽

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京 印刷

◆ 开本: 800×1000 1/16
印张: 17.75
字数: 419千字 2008年8月第1版
印数: 1—3500册 2008年8月北京第1次印刷

著作权合同登记号 图字: 01-2008-2675号

ISBN 978-7-115-18375-0/TP

定价: 45.00元

读者服务热线: (010)88593802 印装质量热线: (010)67129223

反盗版热线: (010)67171154

自从第一台计算机问世以来，计算机技术发展得非常迅速，功能不断扩展，性能突飞猛进。特别是微型计算机的出现，使得计算机的应用从早期单纯的数学计算发展到处理各种媒体的信息。计算机本身也从象牙塔进入了千家万户。

计算机系统由硬件和软件两部分组成。硬件是计算机的物理构成，是计算机的物质基础；软件是计算机程序及相关文档，是计算机的灵魂。

1.1 计算机硬件

经典的计算机硬件结构是由计算机的鼻祖冯·诺依曼提出的，因此被称为冯·诺依曼体系结构。冯·诺依曼体系结构主要包括以下3个方面内容。

(1) 计算机的硬件由5大部分组成，即运算器、控制器、存储器、输入设备和输出设备，这些部分通过总线互相连接，如图1-1所示。在现代计算机系统中，运算器和控制器通常集成在一块称为CPU的芯片上。

(2) 数据的存储与运算采用二进制表示。

(3) 程序和数据一样，存放在存储器中。

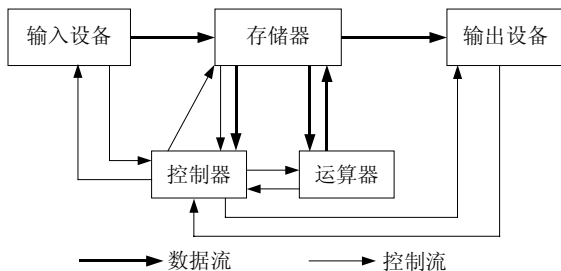


图1-1 计算机硬件系统的组成

运算器是真正执行计算的组件。它在控制器的控制下执行程序中的指令，完成算术运算、逻辑运算和移位运算等。不同厂商生产的机器，由于运算器的设计不同，能够完成的指令也不完全一样。每台计算机能完成的指令集合称为这台计算机的指令系统或机器语言。运算器由算术逻辑

单元 (ALU) 和寄存器组成。ALU 完成相应的运算, 寄存器用来暂存参加运算的数据和中间结果。

控制器用于协调机器其余部分的工作, 是计算机的“神经中枢”。控制器依次读入程序的每条指令, 分析指令, 命令各其他部分共同完成指令要求的任务。控制器由程序计数器 (PC)、指令寄存器 (IR)、指令译码器 (ID)、时序控制电路及微操作控制电路等组成。程序计数器用来对程序中的指令进行计数, 使控制器能依次读取指令; 指令寄存器暂存正在执行的指令; 指令译码器用来识别指令的功能, 分析指令的操作要求; 时序控制电路用来生成时序信号, 以协调在指令执行周期中各部件的工作; 控制电路用来控制各种操作命令。

存储器用来存储数据和程序。存储器可分为主存储器和外存储器。主存储器又称为内存, 用来存放正在运行的程序和数据, 具有存取速度快, 可直接与运算器、控制器交换信息等特点, 但其容量一般不大, 而且一旦断电, 信息将全部丢失。外存储器 (包括硬盘、光盘等) 用来存放长期保存的数据, 其特点是存储容量大、成本低, 但它不能直接与运算器、控制器交换信息, 需要时可成批地与内存交换信息。

存储器内最小的存储单元是比特 (bit)。它可以存放二进制数的一位, 即一个0或一个1。通常8个比特组成一个字节 (byte, 即B)。字节是大部分计算机分配存储器时的最小单位。存储器的容量也是用字节来表示的, 如某台计算机的内存为1GB, 则表明这台计算机的内存是1G字节。

输入/输出设备又称外围设备, 它是外部和计算机交换信息的渠道。输入设备用于输入程序、数据、操作命令、图形、图像和声音等信息。常用的输入设备有键盘、鼠标、扫描仪、光笔及语音输入装置等。输出设备用于显示或打印程序、运算结果、文字、图形、图像等, 也可以播放声音和视频等信息。常用的输出设备有显示器、打印机、绘图仪及声音播放装置等。

事实上, 计算机的工作过程与我们日常生活中的工作过程完全一致。当要求你做一道四则运算题时, 你会通过某个途径获取这道题并记录下来。例如, 你会把这道题目抄到自己的本子上, 那么本子就是主存储器, 你的笔就是输入装置。要计算这道题目, 你会根据先乘除后加减的原则找出里面的乘除部分, 在草稿纸上计算, 结果写回本子上, 再执行加减得到最后结果写回本子。在此过程中, 你的大脑就是CPU, 先做乘除后做加减的过程就是程序, 草稿纸就是ALU中的寄存器, 把答案交给老师的过程就是输出过程。

1.2 计算机软件

计算机硬件是有形的实体, 可以从商店里买到。但如果计算机只有硬件, 那么, 它只能成为一个装饰品。计算机之所以有魅力是因为它有七十二般变化, 可以根据我们的要求变换不同的角色——一会儿是计算器, 一会儿是字典, 一会儿是CD播放机, 一会儿又成了一架照相机。要做到这些, 必须有各种软件的支持。硬件相当于计算机的“躯体”, 而软件相当于计算机的“思想”和处理问题的能力。一个人可能面临各种要处理的问题, 他必须学习相关的知识; 计算机需要解决各种问题, 它需要安装各种软件。

软件可以分为系统软件和应用软件。系统软件居于计算机系统中最靠近硬件的部分, 它将计算机的用户与硬件隔离。系统软件与具体的应用无关, 但其他软件都要通过系统软件才能发挥作

用。操作系统就是典型的系统软件。应用软件是为了支持某一应用而开发的软件，如字处理软件、财务软件等。

1.3 算法与程序设计

要让计算机能够完成某个任务，必须有相应的软件，而软件中最主要的部分就是程序。

程序设计就是教会计算机去完成某一特定的任务，即设计出完成某个任务的程序。它包括两个过程：第一步是设想计算机是如何一步一步地完成这个任务的，第二步是用计算机认识的语言描述这个完成任务的过程。前者称为算法设计，后者称为编码。计算机认识的语言就是程序设计语言，如C++语言。

算法设计就是要设计一个使用计算机提供的基本动作来解决某一问题的方案，是程序设计的灵魂。解决问题的方案要成为一个算法，必须用清楚的、明确的形式来表达，以使人们能够理解其中的每一个步骤，无二义性。算法中的每一个步骤必须有效，以使人们在实践中能够执行它们。例如，若某一算法包含“用 π 的确切值与 r 相乘”这样的操作，则这个方案就不是有效的，因为无法算出 π 的确切值。而且，算法不能无休止地运行下去，必须在有限的时间内给出一个答案。综上所述，算法必须具有以下3个特点。

- (1) 表述清楚、明确，无二义性。
- (2) 有效性，即每一个步骤都切实可行。
- (3) 有限性，即可在有限步骤后得到结果。

有些问题非常简单，一下子就可以想到相应的算法，没有多大的麻烦就可写一个解决该问题的程序；而当问题变得很复杂时，就需要更多的思考才能想出解决它的算法。与所要解决的问题一样，各种算法的复杂性也千差万别。大多数情况下，一个特定的问题可以有多个不同的解决方案（即算法），在编写程序之前需要考虑许多潜在的解决方案，最终选择一个合适的方案。

算法可以用不同的方法表示。常用的有自然语言、传统的流程图、结构化流程图、伪代码和PAD图等方法。本书主要采用伪代码的方法。所谓的伪代码就是介于自然语言和程序设计语言之间的一种表示方法，通常采用程序设计语言的控制结构，用自然语言表示基本的处理。例如，要设计一个算法打印1到100之间的数的平方表，用伪代码表示如下：

```
for ( i=1; i<=100; ++i)
    输出i和i的平方;
```

一旦设计了一个算法，就可以用程序设计语言来描述它，这就是编码。为了让程序员在设计程序时更注重解决问题的算法，而不是某一台计算机上的指令系统，人们希望程序设计语言能独立于计算机系统，更贴近日常的表示。因此，程序设计语言通常都设计得类似于英语，且具有较强的数学计算能力。

1.4 程序的编译和调试

为了让用高级语言编写的程序能够在不同的计算机系统中运行，首先必须将程序翻译成该计

算机特有的机器语言。例如，若为Macintosh机器写了一个C++的程序，这将需要运行一个特殊的程序，该程序将C++语言写的程序转换成Macintosh的机器语言；如果在IBM的PC机上运行该程序，则需要使用另一种翻译程序。在高级语言和机器语言之间执行这种翻译任务的程序叫作编译器。

在大部分计算机系统上，运行程序之前需要先输入程序并将其保存在一个文件中。文件是存储在计算机外存中的信息集合的统称。每个文件都必须有一个文件名，通常用句点将文件名分成两部分，如myprog.cpp。前一部分由文件的创建者指定，通常选择一个能反映文件内容的字符串；后一部分称为扩展名，表示文件的类型，如扩展名“.c”表示文件的内容是C语言编写的程序，“.cpp”表示文件的内容是C++语言编写的程序。包含程序文本的文件称为源文件。

输入文件或修改文件内容的过程称为文件的编辑。各个计算机系统的编辑过程差异很大，不可能用一种统一的方式来描述，因此在编辑源文件之前，必须先熟悉所用的机器上的编辑方法。

一旦有了源文件，下一步就是使用编译器将源文件翻译成计算机可直接读懂的形式。这个过程也因机器而异，但在大多数情况下，编译器将源文件翻译成中间文件，这种中间文件称为目标文件，其中包含适用于特定计算机系统的实际指令，这个目标文件和其他目标文件一起组成可在系统上运行的可执行文件。这里所谓的其他目标文件常常是一些称为库的预定义的目标文件，库中含有许多完成常用操作的指令。将所有独立的目标文件组合成一个可执行文件的过程称为连接，这个过程见图1-2。

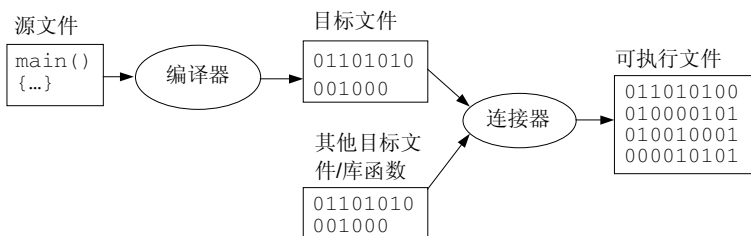


图1-2 编译过程

在编译过程中，编译器会找出源文件中的语法错误和词法错误。用户可根据编译器输出的出错信息来修改源文件，直到编译器生成了正确的目标代码。

语法错误还不是最令人沮丧的。往往程序运行失败不是因为编写的程序包含语法错误，而是程序合乎语法却给出了不正确的答案或者根本没给出答案。检查程序便会发现程序中有些逻辑错误，程序员称这种错误为bug。找出并改正这种逻辑错误的过程称为调试（debug），它是程序设计过程中一个重要的环节。调试一般需要运行程序，通过观察程序的阶段性结果来找出错误的位置和原因。

逻辑错误非常难以察觉。有时程序员非常确信程序的算法是正确的，但随后却发现它不能正确处理以前忽略了一些情况；或者也许在程序的某个地方做了一个特殊的假定，但随后却忘记了；又或者可能犯了一个非常愚蠢的错误。

程序的调试及测试只能发现程序中的错误，而不能证明程序是正确的。因此，在程序的使用过程中可能会不断发现程序中的错误。在使用时发现错误并改正错误的过程称为程序的维护。

小结

本章主要介绍了程序设计所需的下列基础知识和基本概念。

- 计算机系统包括软件和硬件：硬件是计算机的躯壳，软件是计算机的灵魂。
- 计算机硬件主要有5大部分组成：运算器、控制器、存储器、输入设备和输出设备。
- 程序设计包括算法设计、编码、编译、调试及运行维护5个阶段。

习题

1. 简述冯·诺依曼计算机的组成及工作过程。
2. 简述寄存器、主存储器和外存储器的异同点。
3. 所有的计算机能够执行的指令都是相同的吗？
4. 投入正式运行的程序就是完全正确的程序吗？
5. 为什么需要编译器？
6. 调试的作用是什么？

第2章

通过例子学习



2.1 第一个程序：输出 Hello world.

我们先从一个简单的程序开始。

代码清单2-1给出了一个完整的程序的结构。该程序主要由3个部分组成：程序注释、编译预处理命令和主程序。尽管这个程序结构非常简单，但它却是所有程序的典型，可以将它作为C++语言程序组织的范例。

代码清单2-1 输出Hello,world.的C++程序

```
//文件名: hello.cpp                                } 程序注释
//在屏幕上显示 "Hello world."

#include <iostream> } 编译预处理命令

int main()
{
    std::cout << "Hello world." << std::endl; } 主程序
    return 0;
}
```

2.1.1 注释

hello.cpp程序的第一部分为一段注释，描述该程序用来做什么。在C++语言中，注释是从//开始到本行结束。在C++程序中也可以用C语言风格的注释，即在/*与*/之间所有的文字都是注释，可以是连续的几行。

注释是写给人看的，而不是写给计算机的。它们向其他程序员传递该程序的有关信息。C++语言编译器将程序转换为可由机器执行的形式时，注释被完全忽略。

一般来说，每个程序都以一个专门的、从整体描述程序操作过程的注释开头，称为程序注释。它包括源程序文件的名称和一些与程序操作有关的信息。程序注释还可以描述程序中特别复杂的部分，指出可能的使用者，给出如何改变程序行为的一些建议，等等。注释也可以出现在主程序中间，解释主程序中一些比较难理解的部分。

因为注释并不是真正可执行的部分，所以很多程序员往往不愿意写，但注释对将来程序的维护非常重要。给程序添加注释是良好的编程风格。

2.1.2 编译预处理

C++的编译分成两个阶段：预编译和编译。先执行预编译，再执行编译。预编译处理程序中的预编译命令，就是那些以#开头的指令。如代码清单2-1中的 `#include <iostream>`。常用的编译预处理命令主要有库包含和宏定义。

1. 库包含

库包含表示程序使用了某个库。库是实用工具的集合，这些实用工具是由其他程序员编写的，能够完成特定的功能。`iostream`是C++提供的标准输入/输出库。程序中所有数据的输入/输出都由该库提供的功能完成。本书的每个程序都会用到这个库。

C++系统提供了许多标准库，完成各种常用的功能。程序员编写程序时，可以使用这些库提供的实用工具，从而省去自己编写这些实用工具的麻烦。库对于程序设计来说是十分重要的，当你开始编写一些较复杂的程序时，马上将会依赖一些重要的库。

要使用一个库就必须在程序中给出足够的信息，以便使编译器知道这个库里有哪些工具可用，这些工具又是如何使用的。大多数情况下，这些信息以头文件的形式提供。每个库都要提供一个头文件，这种文件为编译器提供了对库所提供的工具的描述，以便在程序中用到这些库的功能时编译器可以检查程序中的用法是否正确。`#include`命令的意思就是把`iostream`头文件加入到现在正在编写的程序中。`iostream`是一个头文件的名称，它主要定义了输入流`cin`和输出流`cout`对象。

`#include`有以下两种格式：

```
#include <文件名>
#include "文件名"
```

用尖括号标记的是系统的标准库，可以通过以下语句包含标准库`iostream`：

```
#include <iostream>
```

个人编写的库用引号标记。例如，某个程序员自己写了一个库`user`，于是`#include`行被写为

```
#include "user"
```

2. 宏定义

宏定义用`#define`实现。宏包括不带参数的宏和带参数的宏。不带参数的宏通常用来定义符号常量。所谓的常量是指在程序执行过程中不变的值，如`hello.cpp`中的`"Hello world"`，而符号常量就是用符号表示的常量。带参数的宏用来定义一些较为复杂的操作。

不带参数的宏的定义格式如下：

```
#define 标识符 替换文本
```

当文件中出现这一行时，以后出现的所有该标识符都会在编译预处理时自动用替换文本取代。例如：

```
#define PI 3.14159
```

在预编译时会用3.14159取代程序中出现的所有符号常量PI。可以用符号常量为某个常量建立一个名字，然后在整个程序中使用这个名字。如果需要修改整个程序中用到的该常量，可以在#define指令中做一次性的修改，然后再重新编译程序就会自动修改出现在程序中的所有这个常量。

在进行宏定义时，可以引用已经定义过的宏名。例如：

```
#define RADIUS 5
#define PI 3.14159
#define AREA PI*RADIUS*RADIUS
```

经过编译预处理后，程序中的RADIUS全部替换成了5，PI全部替换成了3.14159，AREA替换成3.14159*5*5。

在C++中，符号常量一般用大写字母表示。

带参数的宏的处理方式较为复杂。不仅需要进行简单的字符串替换，还要进行参数的替换。带参数的宏的定义格式为

```
#define 宏名(参数表) 替换文本
```

在编译预处理时，首先用替换文本取代程序中的宏名，然后再进行参数的替换。例如，有一个宏定义为

```
#define CIRCLE_AREA(x) (PI* (x) * (x))
```

当程序中出现语句area = CIRCLE_AREA(4)，就会被替换成

```
area = (3.14159 * (4)* (4))
```

用#define定义符号常量和宏是C语言常用的方法。在C++中，这些功能有更好的解决方法。

2.1.3 主程序

代码清单2-1所示的文件hello.cpp的最后部分是程序主体，由以下几行组成：

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

这5行是一个C++语言中函数的例子。函数由一系列独立的程序步骤组成，这些程序步骤集合在一起完成某一功能。每个函数有一个名字。代码清单2-1中的函数名为main。可以把函数看成数学中的函数。数学中的函数有函数名、自变量、函数表达式和函数结果值，C++中的函数也是如此。一个C++程序可以由一个或多个函数组成。在hello.cpp中只有一个函数，名字为main，该函数的执行结果值为一个整型数。main后面的圆括号中的内容是参数，即对应于数学函数中的自变量。空括号表示没有参数。函数表达式（即为所执行的步骤）列于花括号中，称为语句。这些语句共同组成函数的主体，称为函数体。hello.cpp中的main函数的函数体只有两条语句，但大多数函数都有几条连续执行的语句。

名字为main的函数在C++中有独特的作用。每个C++程序至少有一个函数。在组成程序的所

有函数中必须有一个名为main的函数。main函数是程序执行的入口。

当运行C++程序时，计算机从main函数主体开始执行语句，从第一条语句执行到最后一条语句。在main函数的语句中可能调用到其他函数。在hello.cpp中，main的主体由下面两条语句组成：

```
std::cout << "Hello, world" << std::endl;
return 0;
```

cout是标准的输出流对象，它是输入/输出流库的一部分。与cout相关联的设备是显示器。<<称为流插入运算符，表示将其后的数据插入到该流中。std::cout << "Hello, world" << std::endl;表示把Hello, world显示在显示器上，endl表示换行。双引号括起来的Hello, world被称为字符串常量。std是cout和endl所属的名字空间。::称为作用域限定符。return 0表示把0作为函数的执行结果，通常表示函数正常结束。

2.1.4 名字空间

大型的程序通常由很多源文件组成，每个源文件可能由不同的开发人员开发。开发人员可以自由地命名自己的源文件中的实体，如变量名、函数名等。这样很可能造成不同的源文件中有同样的名字。当这些源文件连接起来形成一个可执行文件时，就会造成重名。为了避免这种情况，C++引入了名字空间的概念，把一组程序实体组合在一起，构成一个作用域，称为名字空间。同一个名字空间中不能有重名，不同的名字空间中可以定义相同的实体名。引用某个实体时，需要加上名字空间的限定。

C++的标准库中的实体都是定义在名字空间std中的，因此引用标准库中的名字都要指出是名字空间std中的实体。例如，引用cout必须写成std::cout。

但这种表示方法非常繁琐，为此C++引入了一个使用名字空间的指令using namespace，它的格式如下：

```
using namespace 名字空间名;
```

一旦用了使用名字空间的指令，该名字空间中的所有实体在引用时就不需要再加名字空间的限定了。例如，代码清单2-1所示的程序也可以写成如下形式：

```
//文件名: hello.cpp
//该程序在屏幕上显示 "Hello world."

#include <iostream>
using namespace std;

int main()
{
    cout << "Hello world." << endl;
    return 0;
}
```

2.2 程序示例：计算圆的面积和周长

为了使读者对C++程序的工作过程有更好的了解，下面来看一个复杂一点的例子，计算圆的

面积和周长。程序见代码清单2-2。

代码清单2-2 计算圆的面积和周长的程序

```
//文件名: circle.cpp
//计算圆的面积和周长

#define PI 3.14159 //定义符号常量
#include <iostream>
using namespace std;

int main()
{   double radius, area, circum; } 变量定义

    cout << "请输入圆的半径: " ; } 输入阶段
    cin >> radius;

    area = PI * radius * radius; } 计算阶段
    circum = 2 * PI * radius; }

    cout << endl;
    cout << "圆的面积为: " << area << endl; } 输出阶段
    cout << "圆的周长为: " << circum << endl;

    return 0;
}
```

circle.cpp用到了一些hello.cpp中没有的概念。除了在编译预处理部分增加了一个符号常量外，主要的区别是在main函数中。

在编译预处理命令中定义了常量PI。经过预编译后，main函数中的PI都被3.14159替换。如果该程序要进一步提高计算精度，可以将 π 的值取得更精确些。这只要修改一个地方就可以了，就是把原有的#define指令改成

```
#define PI 3.1415926
```

这样，在计算圆的面积和周长时用的 π 都变成了3.1415926。

circle.cpp的main函数包含了C++函数体的完整内容。函数体分为变量定义部分和语句部分。语句部分又可分为输入阶段、计算阶段和输出阶段。一般各部分之间用一个空行隔开，以便于阅读。

变量（也称为对象）是一些在程序编写时值尚未确定的数据的存放处。例如，在编写求圆的面积和周长的程序时，用户要计算的圆的半径尚未可知，程序运行时用户才输入这个半径值。为了在程序中指明这些目前尚未确定值的对象，可创建一个变量来保存这些需要明确的值，每个变量都有自己的名字。一旦要用到它包含的值，就可使用变量名。变量的名字要精心选定，这样将来阅读程序的程序员就很容易分辨出每个变量的作用。在程序circle.cpp中，变量radius代表圆的半径，变量area和circum分别代表面积和周长。

在C++语言中，变量使用之前必须先定义。定义一个变量就是告知C++编译器创建了一个新的变量名，并具体指定了该变量可以保存的数据类型。例如，在程序circle.cpp中，

```
double radius, area, circum;
```


这一行定义了3个变量，即radius、area和circum，并告知编译器每个变量中保存着一个实型值。类型名double表示实型，用于存储实数。

在输入阶段，计算机要求用户输入圆的半径并将其保存在变量radius中。每个数据的输入过程一般包括两步。首先，程序应在屏幕上显示一个信息以使用户了解程序需要什么，这类信息通常称为提示信息。提示信息可使用cout来显示，如代码清单2-2中的

```
cout << "请输入圆的半径: " ;
```

为了读取数据，程序用了

```
cin >> radius;
```

cin是输入流对象，它也是输入/输出流库的一部分。与cin相关联的设备是键盘。当从键盘输入数据时，形成一个输入流。用流提取运算符>>将数据流存储到一个事先定义好的变量中。

计算阶段包括计算面积和计算周长。在程序设计中，计算是通过表达式来实现的，表达式指定了必需的运算。赋值语句将表达式的结果存储于一个变量中，以备在程序的后面部分使用。2.6节会详细介绍表达式的结构。其实，即使没有一个完整的定义，理解C++语言的表达式也是很容易的，因为它们看上去与传统的数学表达式非常类似。

circle.cpp程序欲计算半径为radius的圆的面积和周长。为了做到这一点，需要使用乘法运算符“*”。为了记录结果，需要将结果存储于专门为之定义的变量area和circum中。执行这些操作的赋值语句是

```
area = PI * radius * radius;  
circum = 2 * PI * radius;
```

同其他语言中的赋值语句一样，计算机计算等式右边的表达式的值，并将结果存储于表达式左边的变量中。

程序的输出阶段是显示计算结果。就像其他输出操作一样，结果的显示是通过使用cout对象来完成的。然而此时有一个新的手法出现，程序circle.cpp的最后两个语句如下：

```
cout << "圆的面积为: " << area << endl;  
cout << "圆的周长为: " << circum << endl;
```

如在hello.cpp程序中一样，双引号内的内容被直接显示在屏幕上。如果输入的radius的值是4，则上述两个输出的结果如下：

```
圆的面积为: 50.2654  
圆的周长为: 25.1327
```

2.3 变量定义

变量是存放数据的地方。变量有3个重要属性：名称、值和类型。为了理解三者之间的关系，可以将变量想象成一个外面贴有标签的盒子。变量的名字写在标签上，以区分不同的盒子。若有3个盒子（即变量），可通过名称来指定其中之一。变量的值对应于盒子内装的东西。盒子标签上的名称从不改变，但盒子中的内容是可变的。变量类型表明该盒子中可存放什么类型的数据。变量定义就是要告诉计算机在本程序执行时要准备多少个盒子，每个盒子里要放什么类型的数据。

因此，C++中变量定义的格式如下：

```
类型名 变量名1, 变量名2, ..., 变量名n;
```

该语句定义了可以存放指定类型数据的 n 个盒子。例如：

```
int num1, num2;
```

定义了两个整型变量num1和num2，而

```
double area;
```

定义了一个实型变量。其中，int和double就是类型名。2.4节会详细介绍C++的内置类型。

定义变量时一项重要的工作是为变量取一个名字，C++语言中变量名的构成遵循以下规则。

- (1) 变量名必须以字母或下划线开头。
- (2) 变量名中的其他字符必须是字母、数字或下划线，不得使用空格和其他特殊符号。
- (3) 变量名不可以是系统的保留字，如int、double、for、return等，保留字在C++语言中有特殊用途。

(4) C++语言中，变量名是区分大小写的，即变量名中出现的大写和小写字母被看作是不同的字符，因此ABC、Abc和abc是3个不同的变量名。

(5) C++没有规定变量名的长度，但各个编译器都有自己的规定。

(6) 变量名应使读者易于明白其存储的值是什么，做到“见名知意”，一目了然。

在C++中，要求所有的变量在使用前都要先定义，这样能保证变量的正确使用。比如，C++中的取模运算（%）只能用于整型数，如果对非整型变量进行取模运算就是一个错误，编译器能检查出这个错误。

2.4 数据类型

C++处理的每一个数据都必须有类型。一个数据类型有两个特征：该类型的数据在内存中是如何表示的，对于这类数据允许执行哪些操作。每种程序设计语言都有自己预先定义好的一些类型，这些类型称为基本类型或内置类型。C++可以处理的基本数据类型有整型、实型、字符型和布尔型。C++也允许用户按照自己的需要定义自己的数据类型。

2.4.1 整型

整型变量中可以存放整数，但整型数和整数并不完全相同。数学中的整数可以有无穷个，但整型数是有穷的。整型数的范围取决于整型数占用的内存空间的大小。

1. 整型数的内部表示

整数在计算机内一般是用补码表示的。正整数的补码是它的二进制表示，负整数的补码是将它的绝对值的二进制表示按位取反后再加1。例如，若一个整数占16位，10在内存中被表示为0000000000001010，-10被表示为111111111110110。

在整数的补码表示中，最高位是符号位。正数的符号位为0，负数的符号位为1。对16位表示而言，正整数的表示范围为0000000000000000~0111111111111111，即0~32 767；负整数的表示

范围为10000000000000000000~1111111111111111，即-32768~-1。

2. 整型的分类

C++中，整型根据占用空间的长度可分为基本整型、长整型和短整型，它们都可用于处理正整数或负整数。在有些应用中，整数的范围通常都是正整数（如年龄、考试分数和一些计数器）。为了充分利用变量的空间，可以将这些变量定义为“无符号”的，即不存储整数的符号，将高位也看成是数据。这样C++一共有6种整型类型。

在C++中，没有具体规定各类整型数所占的内存字节数，只要求长整型不小于基本整型，基本整型不小于短整型。具体如何实现由各编译器自行决定。在Visual C++中各整型所占的字节数如表2-1所示。

表2-1 标准的整型类型

类 型	类 型 名	在Visual C++中占用的空间	表示范围
基本整型	int	4字节	-2 ³¹ ~2 ³¹ -1
短整型	short [int]	2字节	-2 ¹⁵ ~2 ¹⁵ -1
长整型	long [int]	4字节	-2 ³¹ ~2 ³¹ -1
无符号基本整型	unsigned [int]	4字节	0~2 ³² -1
无符号短整型	unsigned short [int]	2字节	0~2 ¹⁶ -1
无符号长整型	unsigned long [int]	4字节	0~2 ³² -1

方括号内的部分是可以省略的。例如，short int与short是等价的。

3. 整型变量的定义

C++规定，每个变量在使用前都必须先定义。变量的定义一般放在函数体的变量定义部分，也可以放在某一个程序块的开头。所谓的程序块就是用花括号（{}）括起来的一组语句。第3章和第4章中将详细介绍。要定义一个短整型数shortnum，可用

```
short int shortnum;
```

或

```
short shortnum;
```

要定义一个计数器，可用

```
unsigned int counter;
```

下面举例说明一下整型变量的定义与使用。编写一个程序来完成两个整型数的相加。所编写的程序应该定义3个整型变量：一个存放加数，一个存放被加数，一个存放结果。输入阶段为加数和被加数赋值，计算阶段计算二者的和，输出阶段输出结果值。实现这一功能的程序如代码清单2-3所示。

代码清单2-3 实现两个整型数相加的程序

```
//文件名：2-3.cpp
//两个整型数相加
```

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

int main()
{   int num1, num2, total;

    num1 = 10;
    num2 = 12;
    total = num1 + num2;

    cout << num1 << '+' << num2 << '=' << total << endl;

    return 0;
}
```

代码清单2-3中程序的运行结果如下：

```
10 + 12 = 22
```

4. 整型数据的溢出

在整型数的内部表示中，正整数的最高位为0，负整数的最高位为1。在Visual C++中，短整型的长度是16位，可表示的数的范围为-32768~+32767。设想一个短整型变量的值为+32767，若对这个变量执行加1的操作，结果将会如何？32767的补码表示为0111111111111111，对它加1，则变成1000000000000000。由于最高位为1，因此，C++不会把这个数解释成+32768，而会把它解释成-32768。这种情况称为“溢出”，但C++在运行时并不报错。这种问题需要程序员靠细心和经验来发现。

5. 整型常量的表示

一旦定义了一个整型变量，就可以把一个整型常量或另一个整型变量的值赋给它。整型常量有3种表示方法：十进制、十六进制和八进制。

十进制与我们日常使用的十进制是一样的，如123、756、-18等。系统按照数值的大小自动将其表示成int或long int。如果需要把一个整数看成是长整型，就可以在这个整数后面加一个“l”或“L”。如100L表示把这个100看成是长整型。

八进制常量以0开头，例如，0123表示八进制数123，它对应的十进制值为83（即 $1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 = 83$ ）。

十六进制数以0x开头，例如，0x123表示十六进制的123，它对应的十进制值为291（即 $1 \times 16^2 + 2 \times 16^1 + 3 \times 16^0 = 291$ ）。

2.4.2 实型

实型又称为浮点型，通常用来存放实数。

1. 实型数的内部表示

在计算机内部，实型数被表示为 $a \times 2^b$ ， a 称为尾数， b 称为指数。在存储实型数时，将存储单元分成两个部分：一部分存放指数，另一部分存放尾数。

在C++中，并没有规定实型数应该用多少位表示，更没有规定用多少位表示指数，用多少位

表示尾数。这些都是由编译器决定。指数部分占据的位数越多，数值表示的范围就越大，尾数部分占据的位数越多，数的精度就越高。

2. 实型的分类

在C++中，实型分为单精度（float）、双精度（double）和长双精度（long double）3种。在Visual C++中，单精度数占4个字节，双精度和长双精度都占8个字节。

在float类型中，1个字节用于存储指数，3个字节用于存储尾数。因此，float可表示的数值范围为 $2^{-128} \sim 2^{127}$ ，相当于 $10^{-37} \sim 10^{38}$ ，精度为十进制的6~7位。double和long double占8个字节：指数占3个字节，尾数占5个字节。

3. 实型变量的定义

同整型变量一样，实型变量在使用前也要定义。例如：

```
float x, y;  
double z;
```

一旦定义了一个实型变量，就可以把一个实型数赋值给它。

由于实型数是用有限的存储单元组成的，所以能提供的有效数字总是有限的。有效位以外的数字将被舍去，这样就会产生一些误差。所以在实型数的应用中不要对两个实型数进行相等比较。

4. 实型常量的表示

实型常量有两种表示方法：十进制小数形式和科学记数法。十进制小数由数字和小数点组成，如123.4、0.0、0.123。科学记数法把实型常量用“尾数 $\times 10^{\text{指数}}$ ”的方式表示。但是因为程序设计语言中不能用上标，所以用了一种替代的方法“尾数e指数”或“尾数E指数”。例如， 123×10^3 可写成123e3或123E3。注意，字母e或E之前必须有数字，而e后面必须是整数，不能有小数。例如，e3、1e3.3、e等都是非法的科学记数法表示。即使实型数是10的幂，如 10^5 ，也不能表示成e5，而要表示成1e5。

2.4.3 字符型

计算机除了能处理数字之外，还能处理文本信息。所有文本信息的基础就是字符。

1. 字符型数据的内部表示

字符在机器内部用一个编号表示。可以把所有可处理的字符写在一个表中，然后对它们顺序编号。例如，可以用整数1代表字母A，整数2代表字母B，依次类推，在用26表示字母Z后，可以继续用整数27、28、29等来表示小写字母、数字、标点符号和其他字符。

尽管每一台计算机可以自己规定每个字符的编码，但这样做会出现一些问题。在当今世界，信息通常在不同的计算机之间共享：你可以用U盘将程序从一台计算机复制到另一台计算机，也可以让你的计算机直接与国内或国际网上的其他计算机通信。为了使这种通信成为可能，计算机必须能够以某种公共的语言“互相交谈”，而这种公共语言的基本特性就在于计算机有同样的字符编码，以免一台机器上的字母A在另一台机器上变成字母Z。

在早期，不同的计算机确实用不同的字符编码。字母A在一台计算机上有一种特定的表示，

但在由另一个生产厂商生产的计算机上却有完全不同的表示。甚至可用字符集也会不同。例如，一台计算机键盘上可能有字符 ϕ ，而另一台计算机则完全不能表示这个字符。

然而，随着时间的推移，许多计算机生产厂商认识到计算机之间相互通信会带来许多的好处，于是他们开始采用统一的字符编码标准。最常用的字符编码标准是ASCII（表示American Standard Code for Information Interchange）字符编码系统。ASCII表请见附录。本书假设所用的计算机系统采用的是ASCII编码。

在大多数情况下，虽然知道字符在内部用什么编码标准是很重要的，但知道各个数字值对应于哪一特定字符并不是很有用。当你键入字母A时，键盘中的硬件自动将此字符翻译成ASCII值65，然后把它发送给计算机。同样，当计算机把ASCII值65发送给显示器时，屏幕上会出现字母A。这些工作并不需要用户的介入。

尽管不需要记住每个字符的具体编码，但ASCII表的以下两个结构特性是值得牢记的，它们在编程中有很重要的用途。

- 表示数字0~9的字符的编码是连续的。尽管不需要知道哪个编码对应于数字字符'0'，但要知道数字'1'的编码是比'0'的编码大1的整数。同样，如果'0'的编码加9，就是字符'9'的编码。
- 字母按字母序分成两段：一段是大写字母（A~Z），一段是小写字母（a~z）。在每一段中，ASCII值是连续的。

2. 字符变量的定义

在C++中，单个字符是用数据类型char来表示。按非正规的说法，数据类型char的值域是一组能在屏幕上显示或能在键盘上输入的符号。这些符号（包括字母、数字、标点符号、空格、回车键等）是所有文本数据的基本构件。要定义字符类型的变量ch，可用以下语句：

```
char ch;
```

变量ch在内存中占一个字节的空間，该字节中存放的是对应字符的ASCII值。

3. 字符常量的表示

C++的字符常量是用单引号括起来的一个字符。例如，'a'、'D'、'1'、'?'等都是字符常量。这些字符被称为可打印字符。例如，在定义了变量ch后，可用

```
ch = 'A';
```

来对ch赋值。此时ch对应的这个字节中存储了十进制值65，这是大写字母A的ASCII值。

由于字符类型变量对应的内存中存放的是字符的编码，C++允许直接将编码赋给字符类型的变量。如要将'A'赋给变量ch，可以直接用ch = 65。

然而，ASCII表也包括许多用来表示某一特定动作的特殊字符，这些特殊字符是以一个“\”开头的字符序列，称为转义序列（escape sequence）。表2-2列出了预定义的转义序列。

将转义序列作为字符常量的一部分就可以在字符常量中包含特殊字符。虽然每个转义序列由几个字符组成，但在机器内部，每个序列被转换为一个ASCII编码。这些特殊字符的编码可见ASCII表。例如，换行符的内部表示为整数10。

表2-2 特殊字符的转义序列

转义序列	功 能
\a	报警声（嘟一声或响铃）
\b	后退一格
\f	换页（开始一新页）
\n	换行（移到下一行的开始）
\r	回车（回到当前行的开始）
\t	Tab（水平移到下一个tab区）
\v	垂直移动（垂直移到下一个tab区）
\0	空字符（ASCII代码为0的字符）
\\	字符\本身
\'	字符'（仅在字符常量中需要反斜杠）
\"	字符"（仅在字符串常量中需要反斜杠）
\ddd	ASCII代码为八进制值ddd的字符

当编译器看见反斜杠字符时，会将其看成是转义序列的第一个字符。如果要表示反斜杠本身，必须在一对单引号中用两个连续的反斜杠，如'\\"'。同样，当单引号被用作为字符常量时，必须在前面加上一个反斜杠'\''。特殊字符也可以用在字符串常量中。例如：

```
cout << "hello, world\nhello, everyone\n";
```

输出两行

```
hello, world
hello, everyone
```

由于双引号作为字符串的开始和结束标记，因此，当双引号作为字符串的一部分时，也必须写成特殊字符。例如，语句

```
cout << "\"Bother,\" said Pooh.\n";
```

的输出将为

```
"Bother," said Pooh.
```

ASCII表中的许多特殊字符没有明确的名字，在程序中可以使用它们的内部编码。在此过程中，唯一的麻烦是这些特殊字符的数字编码是用八进制表示的。例如，字符常量'\177'表示ASCII值为八进制数177对应的字符，这个字符对应于Delete键（在某些键盘上标记为Rubout）的编码在数值上，八进制值177对应于十进制整数127（即1×64+7×8+7=127）。

ASCII编码系统中的许多特殊字符实际上很少使用。对大多数程序设计应用而言，只需要知道换行（'\n'）、tab（'\t'）等有限的几个就够了。

4. 字符运算

在C++中，字符在内部表示为一个整型数，因此字符值能像整数一样参与计算，不需要特别的转换。结果是按其ASCII值计算的。例如，字符'A'，它在内部是用ASCII值65表示的，在运算时被当作整数65处理。

尽管对char类型的值应用任何算术运算都是合法的，但在它的值域内，不是所有运算都是有意义的。例如，在程序中将'A'乘以'B'是合法的，为了得到结果，计算机取它们的内部编码，即65和66，将它们相乘，得到4290。而这个整数作为字符毫无意义，事实上，它超出了ASCII字符的范围。当对字符进行运算时，仅有少量的算术运算是有意义的。下面列举了几种有意义的运算。

- 对一个字符加上一个整数。如果 c 是一个字符， n 是一个整数，表达式 $c+n$ 表示编码序列中 c 后面的第 n 个字符。例如，如果 n 在 $0\sim 9$ ，表达式 $'0'+n$ 得到的是第 n 个数字的字符编码，如 $'0'+5$ 是 $'5'$ 的字符编码。同样，如果 n 在 $1\sim 26$ ， $'A'+n-1$ 表示字母表中第 n 个大写字母的字符编码。
- 从一个字符中减去一个整数。表达式 $c-n$ 表示编码序列中 c 前面的第 n 个字符。例如，表达式 $'Z'-2$ 的结果是字符 $'X'$ 的编码。
- 从一个字符中减去另一个字符。如果 c_1 和 c_2 都是字符，那么表达式 c_1-c_2 表示两个字符在编码序列中的距离。例如，如果再回顾一下ASCII表，并计算每个字符的ASCII值，就可以知道 $'a'-'A'$ 是32。更重要的是，小写字母和它的大写字母之间的距离是固定的，因此 $'z'-'Z'$ 也是32。
- 比较两个字符。用任意的关系运算比较两个字符的值是常用的运算，经常用来确定字母的次序。例如，如果在ASCII表中 c_1 在 c_2 前面，表达式 $c_1 < c_2$ 是true。

为了了解在实际问题中如何应用这些运算，先来想一想计算机是如何执行cin之类的键盘输入的。如果要输入一个整型数（如102）时，计算机将每一次击键作为一个字符接收，因此输入的值为 $'1'$ 、 $'0'$ 、 $'2'$ 。因为结果必须是一个整型数，所以需要将字符转换为相应的整数。要做到这一点，可以利用数字在ASCII表中是连续的这一特点。例如，假设已经从键盘读入了一个字符，并把它存在变量ch中。可以用表达式 $ch-'0'$ 将这个字符转换为数字形式。

假设ch包含一个数字字符，它的ASCII值和 $'0'$ 的ASCII值之间的差正好对应于这个数字的数值。例如，假设变量ch包含字符 $'9'$ ，如果查一下ASCII表，可以知道字符 $'9'$ 的ASCII值为57，数字 $'0'$ 的ASCII值为48，而 $57-48$ 正好等于9。关键在于事先并没有假设 $'0'$ 的ASCII代码为48，这意味着同一种方法可以用于不同字符集的其他计算机上。仅需假设在此字符集中数字的代码是连续的。

但如何判断字符ch是否为数字？再一次利用ASCII表中数字是连续的这一事实。语句

```
if (ch >= '0' && ch <= '9') ...
```

将数字字符与ASCII字符集中的其他字符区分开了。同样，语句

```
if (ch >= 'A' && ch <= 'Z') ...
```

标识出了大写字母，而

```
if (ch >= 'a' && ch <= 'z') ...
```

标识出了小写字母。

2.4.4 布尔型

布尔型用于表示“真”和“假”这样的逻辑值，主要用来表示条件的成立或不成立。“真”表示相应的条件满足，“假”表示相应的条件不满足。C++中用关键字`bool`表示布尔型。布尔型的值只有两个，即`true`和`false`，分别对应逻辑值“真”和“假”。在C++中可以让逻辑值参加算术运算，此时，`true`对应1，`false`对应0。

定义一个布尔型的变量`flag`，可用下列语句：

```
bool flag;
```

在Visual C++中，布尔型的变量占一个字节。当保存`true`时，该字节的值为1；当保存`false`时，该字节值为0。

2.4.5 枚举类型

有时在设计程序时会用到一些特殊的对象，这些对象的取值范围是有限可数的。例如，在一个生成日历的程序中很可能用到一个表示一个星期中的每一天的对象，该对象可能取值的范围就是星期日到星期六，但C++中并没有这样一个数据类型。

解决这个问题有几种方法。一种常用的方法是采用数字编码。假设0表示星期日，1表示星期一，……，6表示星期六，然后用一个整型变量（如`weekday`）表示这个对象。若给这个对象赋值为0，则表示是星期日；给这个对象赋值为1，则表示是星期一；给这个对象赋值为6，则表示是星期六。

这种方法虽然能解决问题，但它也有几问题。首先，编写出来的程序可读性不好。阅读程序时若看见`weekday = 3`这样的语句，我们并不知道这个赋值的涵义是什么，除非有人告诉我们3代表星期三。其次，`weekday`是整型，它可以存放任何整型数。这样，如果我们在录入程序时误将3输入成30，编译器检查不出这个问题，但程序的执行结果肯定不正确。

另外一种方法也是用`int`类型表示这个对象，但为一周中的每一天取一个名字，并为每个名字分配一个数字：星期日是0，星期一是1，星期二是2，依次类推。用`#define`将名字和数字关联起来，表示如下：

```
#define Sunday      0
#define Monday     1
#define Tuesday     2
#define Wednesday  3
#define Thursday   4
#define Friday      5
#define Saturday   6
```

这样可以解决程序的可读性问题。当要给`weekday`赋值为星期三时，可以用`weekday = Wednesday`，而不用`weekday = 3`。

尽管上述两种方法在C++程序中很常见，但它们都没有发挥该语言提供的所有优势。在C++语言中，可以定义一个真正的类型名去表示一种枚举类型。在机器内部，它与用`#define`定义符号常量产生的结果完全相同，枚举类型中的每个元素也是分别用一个整数代码表示的。从程序员

的角度来看，定义自己的枚举类型有下面3个优势。

- 编译器能自动选择整数代码，从而使程序员从中解脱出来。
- 在定义类型时，程序员一般会取一个有意义的类型名，在变量定义时就可以用有意义的类型名，而不是普通的int。这又提高了程序的可读性。
- 在许多计算机系统中，使用明确定义的枚举类型的程序很容易调试，因为编译器可以为调试系统提供有关该类型行为的额外信息，如变量的取值范围或允许的操作。

定义新的枚举类型的语法形式如下：

```
enum 枚举类型名 {元素表};
```

其中元素表是由组成枚举类型的每个值的名字组成的，表中的元素用逗号分开。每个元素也可以紧跟一个等号和一个整数常量，指出特定的内部表示。

用这种方法，可以定义一个表示一个星期中每一天的名字的枚举类型：

```
enum weekdayT { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
```

这个定义引入了一个新的类型名weekdayT，这种类型的变量的取值范围只有花括号中的7个常量值。这些常量类似于用#define引入的常量，事实上，它们有同样的内部表示。不管何时定义一个新的枚举类型，这些元素都将用从0开始的连续的整型数编码。因此，在weekdayT这个示例中，Sunday对应于0，Monday对应于1，依次类推。

作为定义的一个部分，C++语言的编译器也允许明确指出枚举类型的元素的内部表示。例如，若希望从1而不是0开始编号，可以这样定义

```
enum weekdayT { Sunday=1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
```

那么，Sunday对应于1，Monday对应于2，……，Saturday对应于7。也可以从中间某一个元素开始重新指定，例如：

```
enum weekdayT { Sunday, Monday, Tuesday=5, Wednesday, Thursday, Friday, Saturday};
```

那么Sunday对应于0，Monday对应于1，Tuesday对应于5，Wednesday对应于6，依次类推。

任意取值为有限可数的情况都可以使用枚举类型。例如，可以用

```
enum colorT { Red, Orange, Yellow, Green, Violet, Blue, Purple };
```

定义彩虹的颜色，或用

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

定义指南针上的4个基本方向。每个这样的定义都引入了一个新的类型名和对应于该类型的值域。

一旦定义了枚举类型，就可以定义枚举类型的变量，枚举类型变量的定义格式如下：

```
enum 枚举类型名 变量表;
```

或

```
枚举类型名 变量表;
```

例如，enum weekdayT weekday;和weekdayT weekday;都是合法的定义，它们都定义了一个枚举类型weekdayT类型的变量weekday。也可以在定义枚举类型时直接定义变量。例如：

```
enum directionT { North, East, South, West } d;
```

或

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

对枚举类型的对象可以进行赋值和比较运算，例如，可以通过`weekday = Sunday`对变量`weekday`赋值。两个同类的枚举类型的变量也可以相互赋值。对枚举类型的变量可以进行比较操作，如`weekday == Sunday`或`weekday <= Saturday`。枚举类型的比较实际上是对其对应的编码进行比较。

2.4.6 用 `typedef` 重新命名类型名

有些程序员原来可能用的不是C++，而是Pascal语言或FORTRAN语言。习惯用Pascal语言的程序员可能更喜欢用`INTEGER`而不是`int`来表示整型，习惯用FORTRAN语言的程序员可能习惯用`REAL`而不是`double`表示实型。C++提供了一个`typedef`指令，用这个指令可以重新命名类型名。例如，想把整型表示成`INTEGER`，可以用以下语句：

```
typedef int INTEGER;
```

一旦重新命名了类型`int`，就可以用两种方法定义一个整型变量，除了可以用

```
int a;
```

之外，也可以用

```
INTEGER a;
```

2.4.7 定义新的类型

在C++中，当现有的类型不足以满足用户的需求时，允许程序员按自己的需要定义自己的类型，这是通过定义类来实现的。

如前所述，一个类型包括两个含义：一是该类型的数据在内存中是如何标识和存储的，二是对该类型的数据可以做哪些操作。定义一个类就是定义这两个方面。前者称为类的数据成员，后者称为类的成员函数。如何创建和使用类是面向对象技术的关键。本书将从第10章开始详细介绍这方面的内容。

2.4.8 变量赋初值

在C++中，变量定义只是给变量分配相应的存储空间。有时还需要在定义变量时对一些变量设置初值。C++中允许在定义变量的同时给变量赋初值。给内置类型变量赋初值的方法有以下两种：

```
类型名 变量名 = 初值;
```

```
类型名 变量名(初值);
```

例如，`int count = 0;`或`int count(0);`都是定义整型变量`count`，并赋初值0；`float value = 3.4;`或`float value(3.4);`都是定义单精度变量`value`，并赋初值3.4。

虽然上述两种方法达到的目的是相同的，但它们内部采用的处理方法是不同的。第10章将详细讨论其不同之处。

可以给被定义的变量中的一部分变量赋初值。例如：

```
int sum = 0, count = 0, num;
```

定义了3个整型变量，前两个赋了初值，最后一个没有赋初值。

若定义一个变量时没有为其赋初值，然后就直接引用这个变量，是很危险的，因为此时变量的值为一个随机值。给变量赋初值是良好的程序设计风格。

2.4.9 用 `sizeof` 了解占用的内存量

在C++中，每种类型的变量在内存中占用的内存量随编译器的不同而有所不同。要知道某种类型的变量占用多少空间，可以使用`sizeof`运算符。例如，想了解`int`型的变量占用了多少字节，可用`sizeof(int)`；想了解`float`类型的变量占用多少空间，可以用`sizeof(float)`。`sizeof`还可用于表达式。例如，想知道表达式`'a'+15`的结果值占多少空间，可以用`sizeof('a'+ 15)`；想知道变量`x`占用多少空间，可用`sizeof(x)`。

2.5 符号常量

2.1节已经介绍过符号常量以及采用符号常量的好处，也介绍了如何用编译预处理命令`#define`定义符号常量。但是，用`#define`定义符号常量有两个问题：一是所定义的符号常量无法进行类型检查；二是`#define`的处理只是简单的字符串替换，可能会引起一些意想不到的错误。用`#define`定义符号常量是C语言的习惯，在C++中可以给常量取一个名字并指定一个类型。C++中符号常量的定义格式如下：

```
const <类型名> <常量名> = <值>;
```

例如：

```
const double PI = 3.1415926;
```

定义了一个符号常量`PI`，它的值是3.1415926，它是一个双精度的实型数。这样在程序中引用`PI`时，编译器可以检查对`PI`的操作是否为对实型数的合法操作。不过，有了`const`的限定，若在程序中企图修改`PI`的值，编译器就会报错。

必须注意，用`const`定义符号常量时一定要为其指定初值，否则就无法给符号常量指定值。

例如：

```
const double PI;  
PI = 3.14159;
```

是错误的，因为`PI`是常量，在程序中不能改变常量的值。

2.6 算术运算

程序中最重要阶段是计算阶段。计算阶段的主体是数学运算，如代码清单2-2中圆的面积和周长的计算。在程序设计语言中，计算是通过算术表达式实现的。

2.6.1 主要的算术运算符

程序设计语言中的算术表达式和数学中的表达式非常类似。一个表达式由算术运算符和运算数组成。在C++中，适用于所有数值型（包括整型、实型、字符型和布尔型）数据的算术运算符有+（加法）、-（减法）（若左边无值则为负号）、*（乘法）、/（除法）。每一个运算符左右两边各连接一个小的表达式，形成一个新的表达式。这些次级表达式（或称子表达式）称为该运算符的运算数。例如，在表达式 $x+3$ 中，运算符+的运算数为子表达式 x 和 3 。运算数常为变量或常量，但还可以是更复杂的表达式。例如，在表达式 $(2*x)+(3*y)$ 中，+的运算数为子表达式 $(2*x)$ 和 $(3*y)$ 。

同传统的数学中的表达式一样，运算符-有两种含义。当-在两个运算数之间时（例如 $x-y$ ），表示两个数相减；当其左边没有运算数时，表示负号。因此 $-x$ 表示 x 值的相反数。此时运算符-称为一元运算符，因为它只用于一个运算数。其余的运算符（包括表示相减的-）称为二元运算符，因为它们适用于一对运算数。

2.6.2 各种类型的数值间的混合运算

在C++中，整型、实型、字符型和布尔型的数据都可参加算术运算。字符型数据用它的内码参加运算，布尔型数据用0（false）和1（true）参加运算。在进行运算之前，系统会自动将不同类型的数据先转换成同一类型，然后再进行运算。转换的总原则是占用空间少的向占用空间多的靠拢，数值范围小的向数值范围大的靠拢。下面给出具体规则。

- bool、char和short在运算前都必须转换为int。
- int与int运算，结果为int。
- int和float运算，结果为float。
- int和long运算，结果为long。
- int和double运算，结果为double。
- float和float运算，结果为float。
- float和double运算，结果为double。
- double与double运算，结果为double。
- 表达式中，只要有一个运算数是实型常量，结果为double。

根据上述规则，若变量 n 为int型数，则表达式 $n + 1$ 的值为int型数；而表达式 $n + 1.5$ 的值总为double型。这种做法确保计算结果尽可能精确。例如，在表达式 $n + 1.5$ 中，若使用整数运算进行计算，则不能表示出结果中的.5部分。

2.6.3 整数除法和取模运算符

按照自动类型转换规则，若一个二元运算符的左右两边均为整型数，则其结果为整型数。但是当该运算符为除法运算符时，情况就变得有趣了。如果写一个像 $9/4$ 这样的表达式，按C++的规则，此运算的结果必须为整型数，因为两个运算数都为int型数。当程序计算此表达式时，它

用4去除9，将余数丢弃，因此表达式的值为2，而非2.25。若想要计算出从数学意义上来讲正确的结果，应当至少有一个运算数为浮点数。例如，下列3个表达式：

```
9.0 / 4
9 / 4.0
9.0 / 4.0
```

每个表达式都可以得到浮点型数值2.25。只有当两个运算数均为int型数时才会将余数丢弃。

还有一个数学运算符只可用于整型的运算数。这个运算符就是取模运算，即求两个整数除法的余数，在C++中该运算符表示为百分号（%）。它返回的值是第一个运算数除以第二个运算数所得的余数。例如， $9 \% 4$ 的值为1，因为9除以4得2，余数为1。下面是%运算符的一些例子：

```
0 % 4 = 0      19 % 4 = 3
1 % 4 = 1      20 % 4 = 0
4 % 4 = 0      2001 % 4 = 1
```

/和%运算符在程序设计中非常广泛。例如，%运算符常用于检查一个数可否被另一个数整除。若要决定一个整数n能否被3整除，只要看表达式 $n \% 3$ 的结果是否为0即可。

当%的两个运算数中有一个带负号或两个均带负号时，C++的行为就会有些混乱。不同的编译器的行为是不同的。如果完全依赖于一种特定的行为，那么将程序放在另一台机器上运行时，结果将使人大吃一惊。为确保程序在所有机器上都能以相同的方式运行，应避免使用带负号的%运算数。

2.6.4 优先级

若一个表达式中有不止一个运算符，则运算的先后顺序就显得很重要了。在C++中，算术运算符的优先级分成两级：*、/和%是高优先级，+、-是低优先级。当表达式中出现优先级相同的运算符时，按照从左到右的结合方式计算。算术表达式也可以像普通的代数表达式一样用括号改变运算的优先级，但只能用圆括号，不能用方括号或花括号。例如，表达式 $2 * (x + 3) * y$ 表示先执行 $x+3$ ，然后将此结果乘以2，再乘以y。

2.6.5 数学函数库

在C++语言中，除了+、-、*、/、%运算以外，其他的数学运算都是通过函数的形式来实现的。这些数学运算函数都在数学函数库cmath中。cmath包括的主要函数如表2-3所示。要使用这些数学函数，必须在程序开始处写上编译预处理命令：

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

表2-3 cmath中的主要函数

函数类型	cmath中对应的函数
绝对值函数	int abs(int x) double fabs(double x)
e^x	double exp(double x)
x^y	double pow(double x, double y)

(续)

函数类型	cmath中对应的函数
\sqrt{x}	double sqrt(double x)
$\ln x$	double log(double x)
$\lg x$	double log10(double x)
三角函数	double sin(double x)
	double cos(double x)
	double tan(double x)
反三角函数	double asin(double x)
	double acos(double x)
	double atan(double x)

2.7 赋值运算

2.7.1 赋值运算符

在程序中，一个重要的操作就是将计算的结果暂存起来，以备后用。计算的结果可以暂存在一个变量中，因此，任何程序设计语言都必须提供的一个基本的功能就是将数据存放到某一个变量中。大多数程序设计语言都提供一个赋值语句实现这一功能。

在C++中，赋值所用的等号被看成一个二元运算符。赋值运算符=带两个参数，左右各一个。目前可以认为左边的参数一定是一个变量，右边是一个表达式。整个由赋值运算符连起来的表达式称为赋值表达式。执行赋值表达式时，首先计算右边表达式的值，然后将结果存储在赋值运算符左边的变量中。整个赋值表达式的运算结果就是存储在左边变量中的值。因此，

```
total = 0
```

的确是一个表达式，该表达式的结果值为0。一个表达式后面加上一个分号就形成了C++中最简单的语句——表达式语句。赋值表达式后面加上一个分号形成赋值语句。例如：

```
total = 0;
```

就是一个赋值语句，而

```
total = num1 + num2;
```

也是一个赋值语句。这个语句将变量num1和num2的值相加，结果存于变量total中。

在C++中，能出现在赋值运算符左边的表达式称为左值（lvalue）。变量就是最简单的左值。左值是C++中一个重要的概念。

2.7.2 赋值时的自动类型转换

赋值运算是将右边的表达式的值赋给左边的变量，那么很自然地要求右边表达式执行结果的类型和左边的变量类型应该是一致的。当表达式的结果类型和变量类型不一致时，同其他的运算一样会发生自动类型转换。系统会将右边的表达式的结果转换成左边的变量的类型，再赋给左边

的变量。下面列出了基本的转换规则。

- 将实型数赋给整型变量时，舍弃小数部分。例如，若x是整型变量，则在执行赋值语句`x = 1.5;`时，首先将1.5转换成整型数，即去尾变成整型数1，再赋给x。因此，x的值为1。
- 将整型数赋给实型数时，数值不变，但以浮点的形式保存在相应的变量中。例如，若x是double型的变量，则在执行赋值语句`x = 1;`时，首先将1转换成实型数，即加上.0变成实型数1.0，再赋给x。因此，x的值为1.0。
- 将double赋给float时，截取前面7位有效数字存放到float变量中，但应注意数值范围不能溢出。相反，float转换成double，数值不变，将有效位扩展到16位。
- 将字符型数据赋给整型变量时，将字符型数据放入整型变量的最后一个字节。如果所用系统将字符处理成无符号数，则前面补0。如果所用系统将字符处理成有符号数，则扩展符号，将整型变量的最高位设成相应的字符数据的最高位，其他位补0。
- 将整型值赋给字符类型的变量时，直接将整型数据的最低8位赋给字符变量。

2.7.3 赋值的嵌套

C++将赋值作为运算，得到了一些有趣且有用的结果。如果一个赋值是一个表达式，那么该表达式本身应该有值。进而言之，如果一个赋值表达式产生一个值，那么也一定能将这个赋值表达式嵌入到一些更复杂的表达式中去。例如，如果表达式`x = 6`作为另一个运算符的运算数，那么赋给变量x的值就是该赋值表达式的值。因此，表达式`(x = 6) + (y = 7)`等价于分别将x和y的值设为6和7，并将6和7相加，整个表达式的值为13。在C++中，`=`的优先级比`+`低，所以这里的圆括号是必需的。将赋值表达式作为更大的表达式的一部分称为赋值嵌套。

虽然赋值嵌套有时显得非常方便而且很重要，但经常会使程序难以阅读。因为在较大的表达式中的赋值嵌套使变量的值发生的改变很容易被忽略，所以要谨慎使用。

2.7.4 多重赋值

赋值嵌套的最重要的应用就是当你想要将同一个值赋给多个变量的情况。C++语言对赋值的定义允许用以下一条语句代替单独的几条赋值语句：

```
n1 = n2 = n3 = 0;
```

它将3个变量的值均赋为0。它之所以能达到预期效果是因为C++语言的赋值运算是一个表达式，而且它的执行是从右到左的。整条语句等价于：

```
n1 = (n2 = (n3 = 0));
```

表达式`n3 = 0`先被计算，它将n3设为0并将0作为赋值表达式的值传出。随后这个值又赋给n2，结果再赋给n1。这种语句称为多重赋值语句。

当用到多重赋值时，要保证所有的变量都是同类型的，以避免在自动类型转换时出现与预期不相符的结果。可以用一个例子来说明这个问题，假设变量d定义为double，变量i定义为int，那么下面这条语句会有什么效果呢？

```
d = i = 1.5;
```


这条语句很可能使读者产生混淆，以为*i*的值是1，而*a*的值是1.5。事实上，这个表达式在计算时，先将1.5截去小数部分赋给*i*，因此*i*得到值1。表达式*i*=1.5的结果是1，也就是整数1赋给了*a*，而不是浮点数1.5，该值赋给*a*时再引发第二次类型转换，所以最终赋给*a*的值是1.0。

2.7.5 复合赋值运算

假设变量*balance*保存某人银行帐户的余额，他想往里存一笔钱，数额存在变量*deposit*中。新的余额由表达式*balance* + *deposit*给出。于是有以下赋值语句：

```
newbalance = balance + deposit;
```

然而在大多数情况下，人们不愿用一个新变量来存储结果。存钱的效果就是改变银行账户中的余额，因而就是要改变变量*balance*的值，使其加上新存入的数额。与上面的把表达式的结果存入一个新的变量*newbalance*的方法相比，把*balance*和*deposit*的值相加，并将结果重新存入到变量*balance*中更可行，即用下面的赋值语句：

```
balance = balance + deposit;
```

要了解这个赋值语句做些什么，不能将赋值中的=看作数学上的“等于”表达式。在数学上，方程

$$x = x + y$$

只有当*y*取0时可解，除此，*x*不可能等于*x* + *y*。而赋值语句是一个主动的操作，它将右边表达式的值存入左边的变量中。因此，赋值语句*balance* = *balance* + *deposit*;不是断言*balance*等于*balance* + *deposit*，它是一个命令，使得*balance*的值改变为它之前的值与*deposit*的值之和。

尽管语句*balance* = *balance* + *deposit*;能够达到将*deposit*和*balance*相加并将结果存入*balance*的效果，但它并不是C++程序员常写的形式。像这样对一个变量执行一些操作并将结果重新存入该变量的语句在程序设计中使用时十分频繁，因此C++语言的设计者特意加入了它的缩略形式。对任意的二元运算符*op*，

```
变量 = 变量 op 表达式;
```

形式的语句都可以写成

```
变量 op = 表达式;
```

在赋值中，二元运算符与=结合的运算符称为复合赋值运算符。

正因为有了复合赋值运算符，像

```
balance = balance + deposit;
```

这种常常出现的语句便可用

```
balance += deposit;
```

来代替。用自然语言表述即是：把*deposit*加到*balance*中去。

由于这种缩略形式适用于C++语言中所有的二元运算符，所以可以通过下面的语句从*balance*中减去*surcharge*的值：

```
balance -= surcharge;
```

用10除x的值可写作

```
x /= 10;
```

将salary加倍可写作

```
salary *=2;
```

在C++语言中，赋值运算符（包括复合赋值运算符如+=、*=等）的优先级比算术运算符低。如果两个赋值竞争一个运算数，赋值是从右至左进行的。这条规则与算术运算符正好相反，算术运算符都是从左到右进行的。优先级相同的运算符的计算方向称为结合性。加减这样的算术运算符是从左到右计算的，因此称为是左结合的，赋值运算符这样从右到左计算的运算符称为是右结合的。

2.8 自增和自减运算符

除复合赋值运算符外，C++语言还为另外两个常见的操作，即将一个变量加1或减1，提供了更进一步的缩略形式。将一个变量加1称为自增该变量，减1称为自减该变量。为了用最简便的形式表示这样的操作，C++引入了++和--运算符。例如，C++中语句

```
x++;
```

等效于

```
x += 1;
```

这条语句本身又是

```
x = x + 1
```

的缩略形式。同样地有

```
y--;
```

等效于

```
y -= 1;
```

或

```
y = y - 1;
```

自增和自减运算符可以作为前缀，也可以作为后缀。也就是说，++x和x++都是C++中正确的表达式，它们的作用都是将变量x中的值增加1。但当这两个表达式作为其他表达式的子表达式时，它们的作用略有不同。当++作为前缀时，表示先将对应的变量值增加1，然后参加整个表达式的运算；当++作为后缀时，则用对应的变量中的值参加整个表达式的运算，然后再将此变量值加1。例如，若x的值为1，执行下面两条语句的结果是不同的：

```
y = ++x;
```

```
y = x++
```

执行前一语句后，y的值为2，x的值也为2；而执行后一语句后，x的值为2，y的值为1。

当仅要将一个变量值加1或减1时，可以使用前缀的++或--，也可以使用后缀的++或--。但

一般程序员习惯使用前缀的++或--。道理很简单，因为前置操作所需的工作更少，只需加1或减1，并返回运算的结果即可，而后缀的++或--需要先保存变量原先的值，以便让它参加整个表达式的运算，然后变量值再加1。

2.9 强制类型转换

要使得9/4的结果为2.25时，我们可以把此表达式改为9.0/4或9/4.0。但是，如果9和4分别存储于两个整型变量x和y时，如何使x/y的结果为2.25呢？此时不能把这个表达式写为x.0/y，也不能写成x/y.0。

解决这一问题的方法是使用强制类型转换。强制类型转换可将某一表达式的结果强制转换成指定的类型。C++的强制类型转换有两种形式：

(类型名) (表达式)
类型名 (表达式)

因此，想使x/y的结果为2.25，可用表达式double(x)/y或x/(double)y。

强制类型转换实际上就是告诉编译器：“不必检查类型，把它看成是其他类型。”也就是说，在C++类型系统中引入了一个漏洞，并阻止编译器报告那些类型方面出错的问题。更糟糕的是，编译器会相信它，而不执行任何其他的检查来捕获错误。一旦执行了强制类型转换，程序员必须自己面对各种问题。事实上，无论什么原因，任何一个程序如果使用很多强制类型转换都是值得怀疑的。一般情况下，尽可能避免使用强制类型转换，它只是用于解决非常特殊的问题的手段。

一旦理解了这一点，在遇上一个出故障的程序时，第一个反应应该是寻找作为“嫌犯”的转换。为了方便查找这些错误，C++提供了一个更直接的形式，在强制类型转换时指明转换的性质。强制类型转换的性质有4种：静态转换（static_cast）、重解释转换（reinterpret_cast）、常量转换（const_cast）和动态转换（dynamic_cast）。它们的格式如下：

转换类型<类型名>(表达式)

本节将对第一个转换进行较为详细的介绍，后面3个将在用到时再介绍。

static_cast转换可用于允许编译器隐式执行的任何类型转换。例如：

```
int d=10;
char ch = static_cast<char>(d);
```

当需要把一个占用空间较大的类型赋给一个占用空间较少的类型时，使用静态转换非常有用。此时，静态转换告诉程序的读者和编译器：我们知道并且不关心潜在的损失。如果不加静态转换，编译器通常会产生一个警告。加了静态转换后，编译器就会关闭警告信息。

2.10 数据的输入/输出

2.10.1 数据的输入

变量值的来源有很多途径，可以通过赋值运算把一个常量赋给某个变量，可以把某个表达式

的计算结果赋给某个变量，也可以把某个函数执行的结果存放在某个变量中，还可以从某些设备中（磁盘或键盘等）获取数据。当程序中的某些变量值要到程序执行时才能由用户确定时，可以让它在运行时从键盘获取所需要的值。如代码清单2-2中所示，在编程序时不知道用户要计算的圆的半径是多少，一直到运行时，才由用户指定圆的半径。

C++提供多种从键盘输入数据的方式。最常用的方法是利用C++标准库中定义的输入流对象cin和流提取运算符>>来实现。例如：

```
int a;
double d;
cin >> a;
cin >> d;
```

上面的两条输入语句也可以合并成一条：

```
cin >> a >> d;
```

当程序执行到cin时会停下来等待用户的输入。用户可以输入数据，用回车（↵）结束。当有多个输入数据时，一般用空白字符（空格、制表符和回车）分隔。例如，对应上述语句，我们的输入可以是12 13.2↵，那么执行了上述语句后，a的值为12，d的值为13.2；也可以输入12（tab键）13.2↵或者12↵13.2↵，结果都是一样的。但如果输入12a13.2，那么a的值为12，d的值没有意义。

如果要输入的变量是字符型，C++还提供了另外一种常用的方法，即通过cin的成员函数get()输入。get函数的用法为

```
cin.get(字符变量);
```

或

```
字符变量 = cin.get();
```

这两种用法都可以将键盘输入的数据保存在字符变量中。

注意，get函数可以接受任何字符，包括空白字符。如果a、b、c是3个字符型的变量，对应语句

```
a = cin.get();
b = cin.get();
c = cin.get();
```

如果输入为a b c↵，则变量a的内容为'a'，变量b的内容为空格，变量c的内容为'b'。但如果将这个输入用于语句

```
cin >> a >> b >> c
```

那么变量a、b、c的内容分别为'a'、'b'、'c'，因为空格被作为输入值之间的分隔符。

有关cin的详细内容将在第14章进一步介绍。

2.10.2 数据的输出

要将变量的内容显示在显示器上，可以用标准库中的输出对象cout和流插入运算符<<。例如，cout << a可以将变量a的内容显示在显示器上。也可以一下子输出多个变量的值。例如，

`cout << a << b << c`同时输出了变量a、b、c的值。`cout`不仅可以输出变量的值，也可以直接输出表达式的执行结果或输出一个常量。例如，代码清单2-1中的`cout << "Hello world."`直接输出了一个字符串常量。如果变量a等于3，b等于4，想要输出 $3+4=7$ ，可用下列输出语句：

```
cout << a << '+' << b << '=' << a+b << endl;
```

其中，`endl`表示输出一个换行符。

有关`cout`的详细内容将在第14章进一步介绍。

2.11 构思一个程序

任何程序，不管它看起来是多么完美精良，都避免不了将来可能被某人（原先的程序员或项目继承者）做一些修改。原因可能是其中有一些逻辑错误有待纠正，或者是需要加入一些新功能。这被称为程序的维护。作为一名程序员，应该意识到所有的程序都可能在某一天需要改变，自己有责任让修改程序的任务轻松一些。

2.11.1 程序设计风格

要简化所写的程序的维护任务，一条重要途径就是使程序可读性更好。软件开发的一个基本事实是读一个程序的次数总比写它的次数多。然而程序最重要的读者不是机器而是人，那些将在程序生命周期中与它打交道的程序员。让程序变成编译器能接受的状态仅仅是程序设计工作的一部分，好的程序员会花大量的时间在诸如给程序添加注释等一些编译器忽略的工作上。良好的风格和可读性对程序的维护是很重要的。给程序加上精辟的注释并使之对普通读者有所帮助，在一开始虽然会花额外的时间，但这个投入会为以后修改程序节省更多的时间。

那么什么是良好的程序设计风格呢？怎样才能做到呢？从格式上讲，判断一个程序写得好不好的标准又是什么呢？遗憾的是，对这些问题很难给出确切的答案。最简单的方法是审视自己编写的一个程序，想像自己第一次读它，看看是否能读懂。

有很多规则有助于写出较好的程序，下面只列出了一些主要规则。

- 用注释告诉读者需要知道些什么。要向读者解释那些很复杂的或只通过阅读程序本身很难理解的部分。如果希望读者能对程序进行修改，最好简要介绍自己是怎么做的。另一方面，不要过于详细地解释一些很明显的东西。例如，下面这样的注释根本没有什么意义：

```
total += value; //Add value to total
```

最后，也是最重要的，就是确定所加的注释正确地反映了程序当前的状态，当修改程序时，也要及时更新程序的注释。

- 使用缩进来区别程序不同的控制级别。恰当地使用缩进能突出程序中的函数体、循环和条件控制，它们对程序的可读性和结构的清晰性很重要。
- 使用有意义的名字。例如，在支票结算的程序中，变量名`balance`清楚地说明该变量中包含的值是什么。假如使用一个简单字母b，可能会使程序更短更容易输入，但这样却降低

了这个程序的可读性。

- 避免直接使用那些有特殊含义的常量。将这些常量定义成符号常量可以更进一步提高程序的可读性和可维护性。
- 避免不必要的复杂性。为了程序的可读性牺牲一些程序效率是值得的。

我们的宗旨是让程序更容易阅读。为了达到这个目的，最好重新审视自己的程序风格，就像作家校稿一样。在做程序设计作业时，最好早一些开始，做完后把它扔在一边放几天，然后重新拿出来。看看对你来说它容易读懂吗？对别人来说将来它容易维护吗？如果发现程序实际上不易读懂，就应该投入时间来修改它。

2.11.2 设计将来的修改

当编写程序时，可以采用一些能适应变更的设计来使将来对程序的修改容易一些。因为程序员知道哪些地方比其他地方更需要修改，所以程序员常常根据经验判断出程序的哪些部分应该尽可能设计得灵活些。

回想一下本章前面给出的程序circle.cpp。对于这个程序，程序员最可能会修改的部分是什么呢？可能就是 π 的精度。要做这个修改，程序员不得不仔细地检查程序的每个细节，找出 π 在程序的哪些地方曾经出现过，随后发现需要做出两个修改。

考虑到以后其他人也可能会对这个程序做修改，确实希望能通过只做一个修改就将修改效果传递到程序的所有相关部分。这样，程序修改起来更加方便，并且不会因为修改了程序的一部分而未修改其他相应部分而破坏程序。对于这些可能被修改的常量，最好能够采用符号常量。

小结

本章介绍了C++程序的一个完整实例，以使读者了解它们的总体结构及工作方式。主要内容包括一个完整的C++程序的组成以及一个完整的函数的组成。除此之外，着重介绍了构成一个程序必不可少的变量定义、C++的内置数据类型以及算术运算和输入/输出。通过本章的学习，读者应能编写一些简单的程序。

习题

简答题

1. 程序开头的注释有什么作用？
2. 库的作用是什么？
3. 在程序中采用符号常量有什么好处？
4. 有哪两种定义符号常量的方法？C++建议的是哪一种？
5. C++定义了一个称为cmath的库，其中有一些三角函数和代数函数。要访问这些函数，需要在程序中引入什么语句？

6. 每个C++语言程序中都必须定义的函数的名称是什么?
7. 如何定义两个名为num1和num2的整型变量? 如何定义3个名为x、y、z的实型双精度变量?
8. 简单程序通常由哪3个阶段组成?
9. 一个数据类型有哪两个重要属性?
10. 两个短整型数相加后, 结果是什么类型?
11. 说明下列语句的效果, 假设i、j和k声明为整型变量:

```
i = (j = 4) * (k = 16);
```

12. 用怎样的简单语句将x和y的值设置为1.0(假设它们都被声明为double型)?
13. 假如整型数用两个字节表示, 写出下列各数在内存中的表示, 并写出它们的八进制和十六进制表示:

```
10 32 240 -1 32700
```

14. 辨别下列哪些常量为C++语言中的合法常量。对于合法常量, 分辨其为整型常量还是浮点型常量:

```
42 1,000,000 -17 3.1415926 2+3 123456789 -2.3 0.000001 20
1.1E+11 2.0 1.1X+11
```

15. 指出下列哪些是C++语言中合法的变量名?

a. x	g. total output
b. formulal	h. aReasonablyLongVariableName
c. average_rainfall	i. 12MonthTotal
d. %correct	j. marginal-cost
e. short	k. b4hand
f. tiny	l. _stk_depth

16. 在一个变量赋值之前, 可以对它的值做出什么假设?
17. 若k已被定义为int型变量, 当程序执行赋值语句

```
k = 3.14159;
```

后, k的值是什么? 若再执行下面语句, k的值是什么?

```
k = 2.71828;
```

18. 应用相应的优先级法则计算下列每个表达式的值。

```
a. 6 + 5 / 4 - 3
b. 2 + 2 * (2 * 2 - 2) % 2 / 2
c. 10 + 9 * ((8 + 7) % 6) + 5 * 4 % 3 * 2 + 1
d. 1 + 2 + (3 + 4) * ((5 * 6 % 7 * 8) - 9) - 10
```

程序设计题

1. 将本章中的hello.cpp程序原样输入计算机并运行。
2. 设计一个程序完成下述功能: 输入两个整型数, 输出这两个整型数相除后的商和余数。

3. 某工种按小时计算工资。每月劳动时间（小时）乘以每小时工资等于总工资。总工资扣除10%的公积金，剩余的为应发工资。编写一个程序从键盘输入劳动时间和每小时工资，输出应发工资。
4. 编写一个程序，用于水果店售货员结账。已知苹果每斤2.50元，鸭梨每斤1.80元，香蕉每斤2元，橘子每斤1.60元。要求输入各种水果的重量，打印应付钱数。再输入顾客付款数，打印应找的钱数。
5. 编写一个程序完成下述功能：输入一个字符，输出它的ASCII值。

计算机不仅能够做数学运算，还具有逻辑思维的功能，博弈、定理证明甚至某些科学计算都要用到逻辑推理。例如，让计算机解一个一元二次方程，计算机必须能够分析该一元二次方程有解还是无解，然后再根据不同的情况，用不同的方式给出相应的结果。再如，假设上地理课时，4个学生回答我国四大湖的大小时分别有如下表述。

甲：洞庭湖最大，洪泽湖最小，鄱阳湖第三。

乙：洪泽湖最大，洞庭湖最小，鄱阳湖第二，太湖第三。

丙：洪泽湖最小，洞庭湖第三。

丁：鄱阳湖最大，太湖最小，洪泽湖第二，洞庭湖第三。

对于每个湖的大小，已知每个人仅答对一个，如何让计算机通过这些信息去判别4个湖的大小排名。

所有这些问题都必须转换成计算机能看懂的表达式和程序指令。因此，我们必须学习如何让计算机学会人类对于某个问题的思维过程。

3.1 关系运算

逻辑思维最基本的功能就是能分辨各种情况，而关系运算是分辨各种情况的最基本的构件。所谓“关系运算”事实上就是“比较运算”。将两个值进行比较，判断其比较的结果是否符合给定条件。例如， $5 > 3$ 就是一个关系表达式， $>$ 是一个关系运算符。5确实是大于3，因此该表达式的结果为“真”。 $a < 7$ 也是一个关系表达式。当变量 a 的值小于7时，值为“真”，否则为“假”。

3.1.1 关系运算符

C++提供了6个关系运算符： $<$ （小于）、 $<=$ （小于等于）、 $>$ （大于）、 $>=$ （大于等于）、 $==$ （等于）、 $!=$ （不等于）。前4个运算符的优先级相同，后2个运算符的优先级相同。前4个的优先级高于后2个。

3.1.2 关系表达式

用关系运算符可以将两个表达式连接起来形成一个关系表达式，关系表达式的格式如下：

表达式 关系运算符 表达式

参加关系运算的表达式可以是C++的各类合法的表达式，包括算术表达式、逻辑表达式、赋值表达式以及关系表达式本身。因此，下列表达式都是合法的关系表达式：

`a > b` `a + b > c - 3` `(a = b) < 5` `(a > b) == (c < d)`

当关系运算符和算术运算符一起出现时，先执行算术运算。也就是说，算术运算符的优先级比关系运算符高。例如，`a + b > c - 3`表示将`a + b`的结果和`c - 3`的结果进行比较，而不是`a`加上`b > c`的结果，再减去3。当关系运算符和赋值运算符一起出现时，先执行关系运算，再执行赋值运算。如果想要先执行赋值运算，可以用括号改变优先级。例如，`(a = b) < 5`表示先把变量`b`的值赋给`a`，然后再将变量`a`中的值和5进行比较。如果去掉这个表达式中的括号，那么表达式`a = b < 5`表示将关系表达式`b < 5`的运算结果存放在变量`a`中。

关系表达式的计算结果是布尔型的值：`true`和`false`。

在使用关系表达式时，特别要注意的是“等于”比较。“等于”运算符是由两个等号(==)组成的。常见的错误是在比较相等时用一个等号，这样编译器会将这个等号解释为赋值运算。

使用布尔型的数据时需要注意的另外一个问题是，要小心避免冗余，特别是在关系表达式中需要判别布尔型的变量的值时。判别一个布尔变量`flag`的值是否为`true`，初学者常常会用表达式`flag == true`。事实上，只要用一个最简单的表达式`flag`就可以了。

3.2 逻辑运算

除了关系运算符外，C++还定义了3个运算符，即`!`（逻辑非）、`&&`（逻辑与）和`||`（逻辑或），它们的运算对象为布尔型的值，把它们组合起来形成另一个布尔值。这些运算符叫作逻辑运算符。由逻辑运算符连接而成的表达式称为逻辑表达式。逻辑表达式可以用来分辨更复杂的情况。

`&&`、`||`和`!`这3个运算符可以简称为与、或和非。`!`是一元运算符，`&&`和`||`是二元运算符。它们之间的优先级为：`!`最高，`&&`次之，`||`最低。事实上，`!`运算是所有C++运算符中优先级最高的。它们的准确意义可以用真值表来表示。给定布尔值`p`和`q`，`&&`运算符的真值表如表3-1所示。

表3-1 `p&&q`的真值表

<code>p</code>	<code>q</code>	<code>p&&q</code>
false	false	false
false	true	false
true	false	false
true	true	true

表3-1最后一列显示了根据前两列的布尔变量`p`和`q`的值计算出的布尔表达式`p && q`的值。表的第一行表明当`p`的值为`false`，`q`的值为`false`时，表达式`p && q`的值为`false`。

`||`运算的真值表如表3-2所示。要注意不能把`||`运算符按“或”的字面意思理解为有且只有一个成立，而应记住它的数学意义：至少有一个成立。

表3-2 p||q的真值表

p	q	p q
false	false	false
false	true	true
true	false	true
true	true	true

!运算有以下简单的真值表如表3-3所示。

表3-3 !p的真值表

p	!p
false	true
true	false

如果想知道一个很复杂的逻辑表达式是如何计算的，可以先将它分解成这3种最基本的运算，然后再为每一个基本表达式建立真值表，结果就一目了然。

大多数情况下，逻辑表达式不至于复杂到必须用真值表来计算的程度。唯一容易引起混淆的是逻辑运算符!和关系运算符!=与 && 或 || 一起出现的情况。当说起某情况不是真的（即需要用到!或!=的情况），日常用语和数理逻辑表达有时是相悖的，因此需要格外小心避免犯错。比如，在程序中要表达这样一个意思：x不等于2或3。通过阅读这个条件测试的自然语言版，新程序员很可能会写下以下语句：

```
(x != 2 || x != 3) ...
```

如果用数学观点来观察这个条件测试，会发现只要x不等于2或只要x不等于3，测试中的表达式均为true。也就是，无论x取什么值，其中一个表达式必定为真，因为如果x为2，则它不可能同时为3，反之亦然。因此上面写出的测试结果永远为真。

要避免这个问题，需要加强对自然语言的理解，使它能准确地表达各种条件。当只要不满足x为2或x为3，if语句的条件测试就通过，于是可以直接将这句话翻译成C++语句：

```
(!(x == 2 || x == 3)) ...
```

但这条语句不太直观。稍加留意，就会发现真正想测试的是下列条件是否都满足：x不等于2，且x不等于3。如果以这种形式来考虑这个问题，就可以把这个测试写为

```
(x != 2 && x != 3) ...
```

另一个常见的错误是连接几个关系测试时忘记正确地使用逻辑连接。在数学中常可以看到如下表达式：

0 < x < 10

虽然它在数学中有意义，但对C++语言来说却是无意义的。为了测试x既大于0又小于10，需要用下面这样的语句来表达：

```
0 < x && x < 10
```

原则上讲，逻辑运算符的运算对象应该是布尔型的值。但事实上，C++允许运算对象可以是任何类型，0表示false，非0表示true。

有些C++程序员经常喜欢写一些紧凑的表达式,如 $(x = a) < (y = b)$ 。在这个表达式中完成了3项工作:把a的值赋给了x,把b的值赋给了y,比较x和y的值。但在这样的表达式中应用逻辑运算则是很危险的事。因为在计算逻辑表达式时,有时只需要计算一半就能得到整个表达式的结果。例如,对于&&运算,只要有一个运算对象为false,则整个表达式就为false;对于||运算,只要有一个运算对象为true,结果就为true。为了提高计算效益,C++提出了一种短路求值的方法。

当C++程序在计算 $exp1 \&\& exp2$ 或 $exp1 || exp2$ 形式的表达式时,总是从左到右计算子表达式。一旦结果能确定整个表达式的值时,就终止计算。例如,若&&表达式中的 $exp1$ 为false,则不需要计算 $exp2$,因为结果能确定为false。同样,在||表达式的例子中,如果第一个运算对象值为true就不需要计算第二个运算对象的值了。

短路求值最主要的好处在于第一个条件能控制第二个条件的执行。在很多情况下,复合条件的第二部分只有在第一部分满足某个条件时才有意义。比如,要表达以下两个条件:(1)整型变量x的值非零,(2)x能整除y。由于表达式 $y \% x$ 只有在x不为0时才计算,用C++语言可表达这个条件测试为

```
(x != 0) && (y % x == 0)
```

相应的表达式在Pascal程序中得得不到预期的结果,因为无论何时它都要求计算出&&的两个运算对象的值。如果x为0,尽管看起来对x有非零测试,Pascal程序还是会因为除零错误而中止。注意,短路求值也可能使某些程序员精心设计的程序得不到预定的结果。例如:

```
(x=a) && (y=b)
```

程序员想通过这个表达式做3件事:两个赋值和一个逻辑运算。但当a的值为0时,由于短路求值第二个赋值并没有执行。这样的错误是很难发现的。

有了关系表达式和逻辑表达式,就可以表示复杂的逻辑关系。如果想知道某个字符类型的变量ch中的值是否为英文字母,可以用逻辑表达式

```
ch >= 'a' && ch <= 'z' || ch >= 'A' && ch <= 'Z'
```

来判别。想知道某个整型变量num中的值是否为偶数,可用关系表达式

```
m % 2 == 0
```

来判断,想判别某一年year是否为闰年,可用逻辑表达式

```
(year % 4 == 0 && year % 100 != 0) || year % 400 == 0
```

来判断。

回顾本章开头提到的四大湖问题,我们可以把每个学生的答案用一个逻辑表达式表示。如果用a、b、c、d分别表示4个湖,即a表示洞庭湖,b表示洪泽湖,c表示鄱阳湖,d表示太湖,那么甲学生的回答可表示为:

```
a==1 && b==4 && c==3
```

乙学生的回答可表示为:

```
a==4 && b==1 && c==2 && d==3
```

丙学生的回答可表示为:

```
a==3 && b==4
```

丁学生的回答可表示为：

```
a==3 && b==2 && c==1 && d==4
```

C++的一个重要特点是可以将各种类型的数据混合使用。例如，可以把一个逻辑类型的值用于算术表达式。此时，true代表1，false代表0。因此，要表示每个学生仅答对一个，可以分别用下列表达式：

```
((a==1) + (b==4) + (c==3)) == 1
((a==4) + (b==1) + (c==2) + (d==3)) == 1
((a==3) + (b==4)) == 1
((a==3) + (b==2) + (c==1) + (d==4)) == 1
```

3.3 if 语句

3.3.1 if 语句的形式

C++语言中表示按不同情况进行不同处理的最简单办法就是使用if语句，它有以下两种形式：

```
if (条件) 语句
if (条件) 语句1 else 语句2
```

条件部分原则上应该是一个关系表达式或逻辑表达式，语句部分可以是对应于某种情况所需要的处理。如果处理很简单，只需要一条语句就能完成，则可放入此语句。如果处理相当复杂，需要许多语句才能完成，就可以用一个程序块。所谓的程序块就是一组用花括号{}括起来的语句。在语法上相当于一语句。

当某个解决方案需要在满足特定条件的情况下执行一系列语句时，就可以用if语句的第一种形式。如果条件不满足，构成if语句主体的那些语句将被跳过。例如：

```
if (grade >= 60) cout << "passed";
```

当grade的值大于等于60时，输出passed；否则什么也不做。

当程序必须根据测试的结果在两组独立的动作中选择其一时，就可以用if语句的第二种形式。例如：

```
if (grade >= 60) cout << "passed";
else cout << "failed";
```

当grade大于等于60时，输出passed，否则输出failed。

if语句中，条件表达式为true时所执行的程序块叫作if语句的then子句，条件为false时执行的语句叫作else子句。

原则上讲，if语句的条件应该是关系表达式或逻辑表达式。但事实上，在C++的if语句中条件可为任意类型的表达式。可以是算术表达式，也可以是赋值表达式。不管是什么类型的表达式，C++都认为当表达式值为0时表示false，否则为true。

也正因为如此，若要判断x是否等于3，初学者可能会错误地使用

```
if (x = 3) ...
```

而编译器又认为语法是正确的，并不指出错误。这也是初学者常犯的一个错误。程序员会发现当x的值不是3，而是2或5时，then子句照样执行。

3.3.2 if 语句的嵌套

if语句的then子句和else子句可以是任意语句，当然也可以是if语句。由于if语句中的else子句是可有可无的，有时会造成歧义。假设写了几个逐层嵌套的if语句，其中有些if语句有else子句而有些没有，便很难判断某个else子句是属于哪个if语句的。例如：

```
if (x < 100) if (x < 90) 语句1 else if (x<80) 语句2 else 语句3 else 语句4;
```

当遇到这个问题时，C++编译器采取一个简单的规则，即每个else子句是和在它之前最近的一个没有else子句的if语句配对的。按照这个规则，上述语句中的第一个else对应于第二个if，第二个else对应于第三个if，第三个else对应于第一个if。尽管这条规则对编译器来说很简单，但对人来说要快速识别else子句属于哪个if语句还是比较难。这就要求我们通过良好的程序设计风格来解决。例如，通过缩进对齐，清晰地表示出层次关系。上述语句较好的表示方式为

```
if (x < 100)
    if (x < 90) 语句1
        else if (x < 80) 语句2
            else 语句3
    else 语句4;
```

3.3.3 if 语句的应用

有了if语句，就可以解决那些有不同情况的问题。下面通过几个具体的例子来说明if语句的用法。

例3.1 设计一个程序，判断某一年是否为闰年。

这个程序的输入阶段要求输入一个年份，计算阶段判断这一年是否为闰年，输出阶段根据判断结果输出是或不是闰年。具体程序见代码清单3-1。

代码清单3-1 判断闰年的程序

```
//文件名: 3-1.cpp
//判断闰年
#include <iostream>
using namespace std;

int main()
{
    int year;
    bool result;

    cout << "请输入所要验证的年份: ";
```

```

cin >> year;

result = (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;

if (result) cout << year << "是闰年" << endl;
else cout << year << "不是闰年" << endl;

return 0;
}

```

若输入年份为2000，程序的运行结果如下：

```

请输入所要验证的年份：2000
2000是闰年

```

若输入年份为1000，程序的运行结果如下：

```

请输入所要验证的年份：1000
1000不是闰年

```

例3.2 设计一程序，求一元二次方程 $ax^2+bx+c=0$ 的解。

解一个一元二次方程可能遇到下列几种情况：

- (1) $a = 0$ ，蜕化成一元一次方程。
- (2) $b^2 - 4ac = 0$ ，有两个相等的实根。
- (3) $b^2 - 4ac > 0$ ，有两个不等的实根。
- (4) $b^2 - 4ac < 0$ ，无根。

据此，可以写出解一元二次方程的程序，见代码清单3-2。在这个程序中，用if语句判断不同的情况，根据不同的情况采用不同的解法。

代码清单3-2 求一元二次方程解的程序

```

//文件名：3-2.cpp
//求一元二次方程解
#include <iostream>
#include <cmath>           //sqrt所属的库
using namespace std;

int main()
{
    float a, b, c, x1, x2, dlt;

    cout << "请输入3个参数：" << endl;
    cout << "输入 a:";    cin >> a;
    cout << "输入 b:";    cin >> b;
    cout << "输入 c:";    cin >> c;

    if (fabs(a) < 1e-6) cout << "不是一元二次方程" << endl; //fabs(a)<1e-6判a是否为0
    else {
        dlt = b * b - 4 * a * c;
        if (dlt >= 0) {           //有实根
            x1 = (-b + sqrt(dlt)) / 2 / a;
            x2 = (-b - sqrt(dlt)) / 2 / a;
            cout << "x1=" << x1 << "    x2=" << x2 << endl;
        }
    }
}

```

```

    }
    else cout << "无根" << endl;           //无实根
}
return 0;
}

```

若输入0、1、2，程序的运行结果如下：

请输入3个参数：

输入 a: 0

输入 b: 1

输入 c: 2

不是一元二次方程

若输入1、2、3，程序的运行结果如下：

请输入3个参数：

输入 a: 1

输入 b: 2

输入 c: 3

无根

若输入1、-3、2，程序的运行结果如下：

请输入3个参数：

输入 a: 1

输入 b: -3

输入 c: 2

x1=2 x2=1

代码清单3-2所示的程序中有几个地方需要注意。第一个要注意的是条件判断 ($\text{fabs}(a) < 1\text{e-}6$)。这个关系表达式的作用是判断浮点数 a 是否为0。因为浮点数在计算机内无法精确表示，因此一般对浮点数不做等于判断。要判断两个浮点数是否相等，可以转换为两个浮点数的差是否小于一个极小的值。函数 fabs 是C++标准库 cmath 中的一个函数，表示取括号内的浮点数的绝对值。要使用函数 fabs ，必须在程序头上用 $\#include <cmath>$ 把库 cmath 包含进来。

第二个要注意的是变量命名。在第2章中曾经提到变量名不要用简单的单个字母，要有意义。但代码清单3-2的程序中的变量都非常简单： a 、 b 、 c 、 $x1$ 和 $x2$ 。这主要是因为数学中通常一元二次方程的3个系数就是用 a 、 b 、 c 表示，两个根就是用 $x1$ 和 $x2$ 表示。这样命名比较容易理解，因此并没有违反变量的命名规则。

第三个要注意的是 $\text{if} (\text{dlt} > 0)$ 的 then 子句。因为 $\text{dlt} > 0$ 表示有实根， then 子句要计算两个实根并输出，而这不是一条语句能够完成的，需要多条语句才能完成，所以这里将这些语句用一对 $\{\}$ 括起来，形成一个语句块，作为 then 子句。

3.3.4 条件表达式

C++语言提供了另一个更加简练的用来表达条件执行的机制： $?:$ 运算符。（这个运算符被称为问号冒号，但实际这两个符号并不紧挨着出现。）在某些情况下，这个机制非常有用。与C++中的其他运算符不同， $?:$ 在运用时分成两部分且带3个运算数。它的一般形式如下：

(条件) ? 表达式1 : 表达式2

加在条件上的括号从技术上讲是不需要的，但有很多C++程序员用它们来强调测试条件的边界。

当C++程序遇到 ? : 运算符时，首先计算条件的值。如果条件结果为true，则计算表达式1的值，并将它作为整个表达式的值。如果条件结果为false，则整个表达式的值为表达式2的值。由此可以把 ? : 运算符看作以下if语句的缩略形式：

```
if (条件) value = 表达式1;
else value = 表达式2;
```

存储在变量value中的值即为整个 ? : 表达式的值。

例如，将x和y中值较大的一个赋值给max，可以用下列语句：

```
max = (x > y) ? x : y;
```

使用 ? : 运算符的一种最常见的情况是输出时，输出结果可能因为某个条件而略有不同。例如，我们想输出一个布尔变量flag的值，如果直接用

```
cout << flag;
```

那么当flag为“真”时，输出为1；当flag为“假”时，输出为0。如果我们想让flag为“真”时输出true，为“假”时输出false，可以用if语句

```
if (flag) cout << "true";
else cout << "false";
```

这样一个简单的转换需要一个既有then子句又有else子句的if语句来解决，但如果用 ? : 运算符只需要一条语句

```
cout << (flag ? "true" : "false") << endl;
```

3.4 switch 语句及其应用

当一个程序逻辑上要求根据特定条件做出真假判断并执行相应动作时，if语句是理想的解决方案。然而，还有一些程序需要更复杂的判断结构，它有两个以上的可选项，这些选项被划分为一系列互相排斥的情况。比如，在一种情况下，程序应该执行x；在另一种情况下，应该执行y；在第三种情况下，应该执行z；等等。最适合这种情况的就是switch语句，它的语法如下所示：

```
switch (控制表达式) {
    case 常量表达式1: 语句1;
    case 常量表达式2: 语句2;
    ...
    case 常量表达式n: 语句n;
    default:          语句n+1;
}
```

switch语句的主体分成许多独立的由关键字case或default引入的语句组。一个case关键字和紧随其后的下一个case或default之间所有语句合称为case子句。default关键字及其相应语句合称为default子句。

switch语句的执行过程如下。先计算控制表达式的值。当控制表达式的值等于常量表达式1时,执行语句1到语句 $n+1$;当控制表达式的值等于常量表达式2时,执行语句2到语句 $n+1$;依次类推,当控制表达式的值等于常量表达式 n 时,执行语句 n 到语句 $n+1$;当控制表达式的值与任何常量表达式都不匹配时,执行语句 $n+1$ 。

default子句可以省略。当default子句被省略时,如果控制表达式找不到任何可匹配的case子句时,就退出switch语句。

对多分支的情况,通常对每个分支的情况都有不同的处理,因此希望执行完相应的case子句后就退出switch语句。这可以通过break语句实现。break语句的作用就是跳出switch语句。将break语句作为每个case子句的最后一个语句,可以使各个分支互不干扰。这样,switch语句就可写成:

```
switch (控制表达式) {
    case 常量表达式1:    语句1;    break;
    case 常量表达式2:    语句2;    break;
    ...
    case 常量表达式 $n$ :    语句 $n$ ;    break;
    default:              语句 $n+1$ ; break;
}
```

但如果有多分支执行的语句是相同的,则可以把这些分支写在一起,相同的操作只需写一遍。

由于switch语句通常可能会很长,如果case子句本身较短,程序会较容易阅读。如果有足够的空间将case关键字、子句的语句和break语句放在同一行会更好。

例3.3 从键盘输入一个星期的某一天对应的数字:0对应星期天,1对应星期一,……,6对应星期六。然后输出其相应的中文名字。

这是一个典型的switch语句的应用。根据输入的值分成7种情况分别处理。具体程序见代码清单3-3。

代码清单3-3 将整数转换成一星期中的某一天的名字的程序

```
//文件名: 3-3.cpp
//将输入整数转换成星期中的某一天的名字
#include <iostream>
using namespace std;

int main()
{    int day;

    cout << "请输入一个整数(0: 星期天;1: 星期一;……,6: 星期六): ";
    cin >> day;

    switch (day) {
        case 0: cout << "星期天" << endl; break;
        case 1: cout << "星期一" << endl; break;
        case 2: cout << "星期二" << endl; break;
        case 3: cout << "星期三" << endl; break;
```

```

        case 4: cout << "星期四" << endl; break;
        case 5: cout << "星期五" << endl; break;
        case 6: cout << "星期六" << endl; break;
    }

    return 0;
}

```

在这个程序中，控制表达式是一个整型变量，7个常量表达式分别对应于整型数0~6。若输入3，程序的运行结果如下：

```

请输入一个整型数（0：星期天；1：星期一；……，6：星期六）：3
星期三

```

例3.4 设计一个程序，将百分制的考试分数转换为A、B、C、D、E五级计分。转换规则如下：

```

score >= 90      A
90 > score >= 80  B
80 > score >= 70  C
70 > score >= 60  D
score < 60       E

```

解决这个问题的关键也是多分支，它有5个分支。问题是如何设计这个switch语句的控制表达式和常量表达式。初学者首先会想到按照分数分成5个分支，于是把这个switch语句写成以下形式：

```

switch ( score) {
    case score >= 90:  cout << "A";  break;
    case score >= 80:  cout << "B";  break;
    case score >= 70:  cout << "C";  break;
    case score >= 60:  cout << "D";  break;
    default: cout << "E";
}

```

这个switch语句有个严重的错误。case后面应该是常量表达式，而在上述语句的常量表达式中却包含了一个变量score。要解决这个问题，可以修改控制表达式，使之消除case后面的表达式中的变量。观察问题中的转换规则，我们发现分数的档次是和分数的十位数和百位数有关。十位数为9或百位数为1，是A；十位数为8，是B；十位数为7，是C；十位数为6，是D；十位数小于6，是E。因此，只要控制表达式能去除分数的个位数，就能把case后的常量表达式变为10，9，8，7，6，小于6就作为default。去掉一个整型数的个位数只需要通过整数的除法，让分数除以10。这样就可以得到代码清单3-4中的程序。

代码清单3-4 分数转换程序

```

//文件名：3-4.cpp
//将百分制转换成5个等级（A、B、C、D、E）
#include <iostream>
using namespace std;

int main()

```

```
{    int score;

    cout << "请输入分数:";
    cin >> score;

    switch(score / 10) {
        case 10:
        case 9: cout << "A";    break;
        case 8: cout << "B";    break;
        case 7: cout << "C";    break;
        case 6: cout << "D";    break;
        default: cout << "E";
    }
    cout << endl;

    return 0;
}
```

注意在代码清单3-4所示的程序中，case 10后面没有语句，而直接就是case 9，这表示10和9两种情况执行的语句是一样的，就是case 9后面的语句。

若输入100，程序的运行结果如下：

请输入分数:100

A

若输入92，程序的运行结果如下：

请输入分数:92

A

若输入66，程序的运行结果如下：

请输入分数:66

D

若输入10，程序的运行结果如下：

请输入分数:10

E

例3.5 计算机自动出四则运算计算题。

计算机的一个很重要的应用是在教学上。目前，市场上有很多计算机辅助教学软件。应用我们已学到的知识，也可以编一个简单的学习软件。假如想给正在上小学一、二年级的小朋友编一个程序，练习10以内的四则运算。应该怎样解决这个问题呢？首先，程序每次运行时应该能出不同类型的题目，这次出加法，下次可能出乘法；其次，每次出的题目的运算数应该不一样；再次，计算机出好题目后应该把题目显示在屏幕上，然后等待小朋友输入答案；最后，计算机要能批改作业，告诉小朋友答案是正确的还是错误的。

上面4点中关键的是前两点。只要这两点解决了，应用现有的知识就可以编写出这个程序。第三点用一个输出语句就能实现。第四点需要根据题目的类型执行不同的批改方式，这只需要一个switch语句。因此，程序的逻辑如下：

生成题目

switch (题目类型)

```
{
    case 加法: 显示题目, 输入和的值, 判断正确与否
    case 减法: 显示题目, 输入差的值, 判断正确与否
    case 乘法: 显示题目, 输入积的值, 判断正确与否
    case 除法: 显示题目, 输入商和余数的值, 判断正确与否
}
```

问题是如何让程序每次执行的时候都出不同的题目？C++提供了一个称为随机数生成器的工具。随机数生成器能随机生成0~RAND_MAX的整型数，包括0和RAND_MAX。RAND_MAX是一个符号常量，定义在库cstdlib中，它的值与编译器相关。在Visual C++中，它的值是32 767。生成随机数的标准函数是rand()。每次调用rand()都会得到一个0~RAND_MAX的整数，而且这些值的出现是等概率的。

利用随机数生成器，就可以完成题目的生成。运算数可以直接通过调用rand()函数生成，但问题是我们要的整数是10以内，而不是0到RAND_MAX之间。这个问题可以通过一个简单的变换实现，即将0~RAND_MAX的整数等分成10份。如果生成的随机数落在第一份，则映射成0；如落在第二份，则映射成1；……；落在第10份，则映射成9。这可以用一个简单的算术表达式实现 $\text{rand()} * 10 / (\text{RAND_MAX} + 1)$ 。同理可生成运算符。可以把4个运算符用0~3编码。0表示加法，1表示减法，2表示乘法，3表示除法。这样，生成运算符就可以用算术表达式 $\text{rand()} * 4 / (\text{RAND_MAX} + 1)$ 实现。根据上述思想，可以得到代码清单3-5中所示的程序。

代码清单3-5 自动出题程序

```
//文件名: 3-5
//自动出题程序
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

int main()
{
    int num1, num2, op, result1, result2; //num1,num2:操作数, op:运算符,
                                         //result1,result2:结果

    srand(time(NULL)); //随机数种子初始化

    num1 = rand() * 10 / (RAND_MAX + 1); //生成运算数
    num2 = rand() * 10 / (RAND_MAX + 1); //生成运算数
    op = rand() * 4 / (RAND_MAX + 1);    //生成运算符 0--+, 1-- -, 2--*, 3-- /

    switch (op) {
        case 0: cout << num1 << "+" << num2 << "=?"; cin >> result1;
                if (num1+num2==result1) cout << "you are right\n";
                else cout << "you are wrong\n";
                break;
        case 1: cout << num1 << "-" << num2 << "=?"; cin >> result1;
                if (num1-num2==result1) cout << "you are right\n";
                else cout << "you are wrong\n";
    }
```

```

        break;
    case 2: cout << num1 << "*" << num2 << "=?"; cin >> result1;
        if (num1*num2==result1) cout << "you are right\n";
        else cout << "you are wrong\n";
        break;
    case 3: cout << num1 << "/" << num2 << "=?"; cin >> result1;
        cout << "余数为=?"; cin >> result2;
        if ((num1/num2==result1)&&(num1%num2==result2))
            cout << "you are right\n";
        else cout << "you are wrong\n";
        break;
    }

    return 0;
}

```

在代码清单3-5所示的程序头上, 包含了库`cstdlib`。这个库中有随机数生成函数。程序的第一条语句`srand(time(NULL))`的作用是设置随机数的种子。随机数的生成需要种子, 不同的种子可以生成不同的随机数序列。如果用户不设置随机数的种子, 系统会指定一个。系统为每个程序、每次执行指定的随机数的种子都是相同的, 因此程序每次执行生成的随机数序列都是相同的。反映在自动出题程序中, 就是程序每次执行出的题目都是一样的。如今天执行该程序出的题目是 $3+5$, 下一次执行出的题目还是 $3+5$ 。因此希望每次执行生成的题目不一样, 程序每次执行时设置的种子都必须不一样的, C++提供了设置种子的函数`srand`, 允许程序员在程序中设置随机数的种子。但如果程序员设置的种子是一个固定值, 那么该程序每次执行得到的随机数序列还是相同的。如何让程序每次执行时选择的种子都不一样呢? 在一个计算机系统中, 时间总是在变。因此把系统时间设为种子是一个很好的想法。`time(NULL)`就是取当前的系统时间。为了使用时钟, 需要包含头文件`ctime`。

代码清单3-5所示的程序运行过程如下:

```

3+5=?8
you are right

5/2=?2
余数为=?1
you are right

7-5=?8
you are wrong

```

这个程序还是比较粗糙的, 有很多细节没有考虑。例如, 除数为0, 减法的结果可能为负数等。而且程序的使用也不方便, 每次运行只能生成一个题目, 而用户通常希望运行了程序后会一道接一道地出题, 直到用户想退出为止。程序的输出也太单调, 不是`you are right`就是`you are wrong`, 输出信息也能更丰富些就好了。随着学习的深入, 这些问题可以逐步完善。

小结

本章主要介绍了计算机实现逻辑思维的机制, 主要包括两个方面: 如何区分不同的情况, 如

何根据不同的情况执行不同的处理。

简单的情况区分可以用关系表达式实现。通俗地讲，关系运算就是比较。复杂的情况区分可以用逻辑表达式实现。逻辑表达式就是用逻辑运算符连接多个表达式，以表示更复杂的逻辑。

关系运算和逻辑运算的结果是布尔型的值：true和false。但在C++中，布尔型的值可以和其他类型的值混合使用，可以将布尔值用于算术表达式，此时，true表示1，false表示0。我们也可以将其他类型的值用于逻辑表达式，此时，0表示false，非0表示true。

根据逻辑判断执行不同的处理有两种途径：if语句和switch语句。if语句用于两个分支的情况，switch用于多分支的情况。

习题

简答题

1. 怎样用一个逻辑表达式来测试整型变量n的值在0~9之间，包含0和9？
2. 假设myFlag声明为布尔型变量，下面的if语句会有什么问题？

```
if (myFlag == true)...
```

3. 设a=3，b=4，c=5，写出下列各逻辑表达式的值。

- a. `a+b>c && b == c`
- b. `a || b+c && b-c`
- c. `!(a>b) && !c`
- d. `(a!=b) || (b<c)`

程序设计题

1. 从键盘输入3个整数，输出其中的最大值、最小值和平均值。
2. 有一个函数，其定义如下：

$$y = \begin{cases} x & (x < 1) \\ 2x - 1 & (1 \leq x < 10) \\ 3x - 11 & (x \geq 10) \end{cases}$$

编一程序，输入x，输出y。

3. 编写一个计算薪水的程序。某企业有3种工资计算方法：计时工资、计件工资和固定月工资。程序首先让用户输入工资计算类别，再按照工资计算类别输入所需的信息。若为计时工资，则输入工作时间及每小时薪水，计算本月应发工资。职工工资需要缴纳个人所得税，缴个税的方法是：2000元以下免税；2000~2500元者，超过2000元部分按5%收税；2500~4000元者，2000~2500的500元按5%收税，超过2500部分按10%收税；4000元以上者，2000~2500的500元按5%收税，2500~4000的1500元按10%收税，超过4000元的部分按15%收税。最后，程序输出职工的应发工资和实发工资。

第2章中介绍了一个示例，计算圆的面积和周长。这个程序先输入圆的半径，然后计算面积和周长，最后输出结果。换句话说，这个程序的所有语句都是顺序执行的。然而，在解决一些复杂的问题时会发现，严格的顺序执行是不够的。例如，要求计算10个圆的周长和面积，就需要代码清单2-2所示的程序主体执行10遍。要指定重复执行可以使用循环语句。

C++提供了3种循环控制方法：for循环，while循环和do-while循环。

4.1 for 循环

4.1.1 重复 n 次操作

在某些应用中经常会遇到某一组语句要重复执行 n 次。在程序设计语言中通常用for语句来实现。在C++语言中，for语句的最简单的用法如下：

```
for (i=0; i<n; i++) {  
    要重复执行的语句;  
}
```

例如，用5来代替 n ，则大括号中的语句将被执行5次。

在C++语言中，for语句由两个不同的部分构成：循环控制行和循环体。

(1) 循环控制行。for语句的第一行被称为循环控制行。例如：

```
for (i=0; i<N; i++)
```

for语句的控制行用来指定花括号中语句将被执行的次数。它由3个表达式组成：表达式1（上例中为 $i=0$ ）是循环的初始化，指出首次执行循环体前应该做哪些初始化的工作，变量 i 称为循环变量，通常用来记录循环执行的次数，表达式1一般用来对循环变量赋初值；表达式2是循环条件（上例中为 $i < n$ ），满足此条件时执行循环体；表达式3为步长（上例中的 $i++$ ），表示在每次执行完循环体后循环变量如何变化。

(2) 循环体。花括号中的语句构成了for语句的循环体。在for语句中，这些语句将按控制行指定的次数重复执行。为了更明了，循环体内的每一条语句都比控制行多4个空格的缩进，这样for语句的控制范围就一目了然了。

根据for语句的语法规则，上述语句的执行过程为：将1赋给循环变量i；判别i是否小于n，若条件为真，执行循环体；然后i加1。再判别i是否小于n，若条件为真，执行循环体。然后i再加1。如此循环往复，直到i等于n。循环里所有语句的一次完全执行称为一个循环周期。

例4.1 某班级有100个学生，设计一程序统计该班级某门考试成绩中的最高分、最低分和平分。

初看起来，解决这个问题需要先输入100个整型数，保存在各自对应的变量中，然后依次检查这100个数，找出最大的和最小的。在找的过程中顺便可以把所有的数都加起来，最后将总和除以100就得到了平均值。按照这个思路，要定义100个整型变量存放这100个数，还要定义保存最大值、最小值和总和的变量。但要声明100个变量然后每个变量用各自的语句读入数值似乎有些不可思议。

静下心来仔细想想这个问题，想象在没有计算机的情况下，别人依次说出100个数——7,4,6,...，应该如何计算它们的和？可以将听到的数挨个记下来，最后加起来。这个方案和刚才讲的思想是一样的，它的确可行，但却并不高效。另一种可选方案是在说出数字的同时将它们加起来，即7加4等于11，11加6等于17，……。同时记住最小的和最大的值。这样不用保存每一个数据，只存储当前的和以及一个最大值和最小值就可以了。当得到最后一个数时，也就得出了结果。

这个方案不用保存每个单独的数据，也就不需要定义100个变量了。用这个方案只需使用4个变量：一个用于保存每次读入的数据，一个用于保存当前的和，一个用于保存最大值，一个用于保存最小值。每当读入一个新的数值时，只需简单地将它加入到保存之前所有数值和的变量中。然后看看它是否小于最小值，如果是的，则记住它是最小值。如果它大于最大值，则记住它是最大值。之后，又可以用同样的变量保存下一个数值，并按同样的方法处理。

现在可以用新的方案来编写程序了。记住，只需要定义4个变量，分别保存当前输入值、当前和、最大值和最小值。程序的开始是下面的定义：

```
int value, total, max, min;
```

当输入每个数值时必须执行下面的步骤。

- (1) 请求用户输入一个整数值，将它存储在变量value中。
- (2) 将value加入到保存当前和的变量total中。
- (3) 如果value大于max，将value存于max。
- (4) 如果value小于min，将value存于min。

由此可得出代码清单4-1所示的程序。

代码清单4-1 统计考试分数的程序

```
//文件名：4-1.cpp
//统计考试分数中的最高分，最低分和平分
#include<iostream>
using namespace std;

int main()
{   int value, total, max, min, i;//value存放当前输入数据，i为循环变量
```

```
//变量的初始化
total = 0;
max = 0;
min = 100;

for (i=1; i<=100; ++i){
    cout << "\n请输入第" << i << "个人的成绩: ";
    cin >> value;
    total += value;
    if (value > max) max = value;
    if (value < min) min = value;
}

cout << "\n最高分: " << max << endl;
cout << "最低分: " << min << endl;
cout << "平均分: " << total / 100 << endl;

return 0;
}
```

4.1.2 for 语句的进一步讨论

事实上, for语句的循环控制行中的3个表达式可以是任意表达式, 而且3个表达式都是可选的。如果循环不需要任何初始化工作, 则表达式1可以省略。如果循环前需要做多个初始化工作, 可以将多个初始化工作组合成一个逗号表达式, 作为表达式1。在代码清单4-1中, 循环前需要将循环变量i置为1, total和max置为0, min置为100。我们可以把这些工作都放在循环控制行中:

```
for (i=1, total = max = 0, min = 100; i<=100; ++i)
```

逗号表达式由一连串基本的表达式组成, 表达式之间用逗号分开。逗号表达式的执行从第一个表达式开始, 一个一个依次执行, 直到最后一个表达式。逗号表达式的值是最后一个表达式的结果值。逗号运算符是所有运算符中优先级最低的。

也可以把所有的初始化工作放在循环语句之前, 此时表达式1为空。代码清单4-1中的程序也可以做如下修改:

```
total = 0;
max = 0;
min = 100;
i=1
for (; i<=100; ++i){ ... }
```

习惯上, 将循环变量的初始化放在表达式1中, 其他的初始化工作放在循环语句的前面。

表达式2也不一定是关系表达式。它可以是逻辑表达式, 甚至可以是算术表达式。当表达式2是算术表达式时, 只要表达式的值为非0, 就执行循环体, 表达式的值为0时退出循环。

如果表达式2省略, 即不判断循环条件, 循环将无终止地进行下去。永远不会终止的循环称为死循环或无限循环。最简单的死循环是

```
for (;;);
```

要结束一个无限循环, 必须从键盘上输入特殊的命令来中断程序执行并强制退出。这个特殊

的命令因机器的不同而不同，所以应该先了解自己机器的情况。

表达式3也可以是任何表达式，一般为赋值表达式或逗号表达式。表达式3是在每个循环周期结束后对循环变量的修正。表达式3也可以省略，此时做完循环体后直接执行表达式2。

4.1.3 for 循环的嵌套

当程序非常复杂时，常常需要将一个for循环嵌入到另一个for循环中去。在这种情况下，内层的for循环在外层for循环的每一个周期中都将执行它的所有的周期。每个for循环都要有一个自己的循环变量以避免循环变量间的互相干扰。

例4.2 打印九九乘法表。

打印九九乘法表的程序见代码清单4-2。

代码清单4-2 打印九九乘法表的程序

```
//文件名: 4-2.cpp
//打印九九乘法表
#include<iostream>
using namespace std;

void main()
{   int i, j;

    for (i=1; i<=9; ++i){
        for (j=1; j<=9; ++j)    cout << i*j << '\t';
        cout << endl;
    }
}
```

外层for循环，用i作为循环变量，控制乘法表的行的变化。在每一行中，内层for循环用j作为循环变量，控制该行中列的变化，并显示每一个项的值i*j（即行号乘以列号）。注意在每一行结束时必须换行，因此外层循环的循环体由两个语句组成：打印一行和打印换行符。内层循环中的'\t'控制每一项都打印在下一个打印区，使输出能排成一张表。代码清单4-2所示的程序输出如下：

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

4.2 while 循环

例4.1是一个使用for语句的很好的例子，但它现在的这种形式不可能满足很多使用者的要

求。这个程序只能统计100个分数，但实际上并不是每个班级都正好有100个学生。每个班级学生的人数是不同的，因此该程序没有很好的通用性。我们希望有一个程序能对不同人数的班级完成分数统计任务。

要想将例4.1中的程序修改为能解决类似问题的更通用的程序，有一种方法只需要对程序进行一个小的修改。可以在程序的开始部分请求用户输入数据个数，并将之存放在某个变量中，以此来替换for语句控制行中使用的常量100。为此定义一个整型变量n，在程序的开始部分加入以下两行：

```
cout << "请输入人数: ";  
cin >> n;
```

将for语句的控制行改为for (i=0; i<n; i++)，最后求平均分的时候将total/100改为total/n就可以了。

但这种方案的问题是：要统计某个班级的成绩，先要数一数这个班这次参加考试的人数有多少。没人会乐意去做这个工作，所以应该采取另外的方法。

从用户的角度出发，最好的办法就是定义一个特殊的数据，用户可以通过输入该数据来标识输入序列的结束。这个用来结束循环的特殊的值称为标志。选择一个合适的标志值与输入数据的特征有关。选出的标志值不应是一个合法的数据值，也就是说它不可能是用户需要作为正常数据输入的值。在例4.1中，-1就是一个合适的标志，因为合法的考试成绩为0~100。

C++中，提供这种循环控制的语句是while语句。它重复执行一条简单语句或程序块，直到条件表达式变为false。while语句的格式如下：

```
while (条件表达式){  
    要重复执行的语句;  
}
```

与if语句一样，当循环体只由一条简单语句构成时，C++编译器允许去掉加在循环体两边的花括号。整条语句（包括while控制行本身和循环体）一起构成一个while循环。

程序执行while语句时，先计算出条件表达式的值，看它是true还是false。如果是false，循环终止，并接着执行在整个while循环之后的语句；如果是true，整个循环体将被执行，而后又回到while语句的第一行，再次对条件进行检查。

在考查while循环的操作时，有下面两个很重要的原则。

(1) 条件测试是在每个循环周期之前进行的，包括第一个周期。如果一开始测试结果便为false，则循环体根本不会被执行。

(2) 对条件的测试只在一个循环周期开始时进行。如果碰巧条件值在循环体的某处变为false，程序在整个周期完成之前都不会注意它。在下一个周期开始前再次对条件进行计算，倘若为false，则整个循环结束。

例4.3 设计一个程序，统计某个班级某门考试成绩中的最高分、最低分和平均分。当输入的分数为-1时，输入结束。

这个问题的解题思路与例4.1类似，但有两个区别。一是人数不再固定，而是通过在输入数

据中设置一个标志来表示输入结束，因此可用while循环代替for循环；二是必须记住本次执行时输入了多少个分数，并以此来计算平均分，因此需要增加一个变量noOfInput。具体程序见代码清单4-3。

代码清单4-3 统计分数的程序

```
//文件名: 4-3.cpp
//统计考试成绩中的最高分, 最低分和平均分
#include<iostream>
using namespace std;

int main()
{
    int value, total, max, min, noOfInput;

    total = 0; //总分
    max = 0;
    min = 100;
    noOfInput = 0; //人数

    cout << "请输入第1位学生的成绩: ";
    cin >> value;
    while (value != -1){
        ++noOfInput;
        total += value;
        if (value > max) max = value;
        if (value < min) min = value;
        cout << "\n请输入第" << noOfInput + 1 << "个人的成绩: ";
        cin >> value;
    }

    cout << "\n最高分: " << max << endl;
    cout << "最低分: " << min << endl;
    cout << "平均分: " << total / noOfInput << endl;

    return 0;
}
```

与for循环一样，while循环的循环条件表达式不一定要是关系表达式或逻辑表达式，可以是任意表达式。例如：

```
while(1);
```

是一个合法的while循环语句。但它是一个死循环，因为条件表达式的值永远为1，而在C++中任何非0值都表示true。

例4.4 用无穷级数 $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots$ 计算 e^x 的近似值，当 $\frac{x^n}{n!} < 0.000001$ 时结束。

计算 e^x 的近似值就是把该级数的每一项依次加到一个变量中。所加的项数随 x 的变化而变化，

在写程序时无法确定。显然，这是一个需要用while循环解决的问题。循环的条件是判断 $x^n/n!$ 是否大于0.000001。大于时继续循环，小于时退出循环。在循环体中，计算 $x^n/n!$ ，把结果加到总和中。如果令变量ex保存 e^x 的值，由于级数的每一项的值在加完后就没有用了，因此可以用一个变量p保存当前正在处理的项的值，那么该程序的伪代码如下：

```
ex=0;
p = 1;
while (p>0.000001) {
    计算新的p;
    ex += p;
}
```

在这段伪代码中，需要进一步细化的是如何计算当前项p的值。显然，第*i*项的值为 $x^i/i!$ 。 x^i 就是将x自乘*i*次，这是一个重复*i*次的循环，可以用一个for循环来实现：

```
for (xn=1, j=1; j<=n; ++ j) xn *= x;
```

$i! = 1 \times 2 \times 3 \times \cdots \times i$ 可以通过设置一个变量，将1, 2, 3, ..., *i*依次与该变量相乘，结果存回该变量的方式来实现，也可以用for循环实现：

```
for (pi = 1, j=1; j<=i; ++j) pi *=j;
```

最后，令 $p = xn / pi$ 就是第*i*项的值。

这种方法在计算每一项时，需要两个执行*i*次的循环！整个程序的运行时间会很长。事实上，有一种简单的方法。在级数中，项的变化是有规律的。可以通过这个规律找出前一项和后一项的关系，通过前一项计算后一项。如本题中，第*i*项的值为 $x^i/i!$ ，第*i*+1项的值为 $x^{i+1}/(i+1)!$ 。如果p是第*i*项的值，则第*i*+1项的值为 $p*x/(i+1)$ 。这样可以避免两个循环。根据这个思想，可以得到代码清单4-4所示的程序。记住，在程序设计中时刻要注意提高程序的效率，避免不必要的操作；但也不要一味追求程序的效率，把程序写得晦涩难懂。

代码清单4-4 计算 e^x 的程序

```
//文件名: 4-4.cpp
//计算无穷级数
#include <iostream>
using namespace std;

int main()
{
    double ex, x, p; //ex存储 $e^x$ 的值, p保存当前项的值
    int i;

    cout << "请输入x: ";
    cin >> x;

    ex=0;
    p=1;
    i=0;

    while (p>1e-6){
        ex += p;
        ++i;
```

```

        p = p * x / i;
    }

    cout << "e的" << x << "次方等于: " << ex << endl;

    return 0;
}

```

4.3 do-while 循环

在while循环中，每次执行循环体之前必须先判别条件。如果条件表达式为true，执行循环体，否则退出循环语句。因此，循环体可能一次都没有执行。如果能确保循环体至少必须执行一次，那么可用do-while循环。

do-while循环语句的格式如下：

```

do {
    要重复执行的语句;
} while (条件表达式);

```

do-while循环语句的执行过程如下：先执行循环体，然后判别条件表达式，如果条件表达式的值为true，继续执行循环体，否则退出循环。

在例4.4中，由于第1项必定是需要的，因此可以用do-while语句实现。只需要将代码清单4-4所示的程序中的while语句改成do-while语句即可：

```

do {
    ex += p;
    ++i;
    p = p * x / i;
} while (p>1e-6)

```

4.4 循环的中途退出

例4.3考虑一个读入数据直到读到标志值的问题。如果用自然语言描述，基于标志的循环的结构由以下步骤组成：

- (1) 读入一个值；
- (2) 如果读入值与标志值相等，则退出循环；
- (3) 否则，执行在读入那个特定值情况下需要执行的语句。

不巧的是，在循环的最开始没有能决定循环是否应该结束的测试。循环的结束条件要到读入值等于标志值时才出现；为了检查这个条件，程序必须先读入某值。在代码清单4-3中可以看到读入数据的语句在程序中出现两次：一次在循环开始前，一次在循环中。当一个循环中有一些操作必须在条件测试之前执行时，称为中途退出问题。

C++语言中解决中途退出问题的一个方法是使用break语句，该语句除了能用在switch语句中之外，它还有使最内层的循环立即中止的作用。使用break就能用下面很自然的结构来编写标

志问题中的循环:

```
while (true) {  
    提示用户并读入数据;  
    if (value==标志) break;  
    根据数据作出处理;  
}
```

第一行while(true)需要解释一下。按照while循环的定义,循环将一直执行直到括号里的条件值变为false。true是一个常量,它不会变为false。因此就while语句本身而言,循环永远不会结束,是一个死循环。程序退出循环的唯一方法就是执行其中的break语句。

还有一个与break类似的语句:continue。它用于跳出当前的循环周期,回到循环条件判断的地方。

4.5 枚举法

有了循环控制结构,就可以实现一种典型的解决问题的方法——枚举法。枚举法就是对可能是解的众多候选者按某种顺序进行逐一枚举和检验,从中找出符合要求的候选解作为问题的解。

例4.5 用50元钱买了3种水果。各种水果加起来一共100个。西瓜5元一个,苹果1元一个,橘子1元3个。设计一个程序,输出每种水果各买了几个。

这个问题可用枚举法来解。它有两个约束条件:第一是3种水果一共100个,第二是买3种水果一共花了50元。也就是说,如果西瓜有mellon个,苹果有apple个,桔子有orange个,那么mellon + apple + orange等于100, $5 * mellon + 1 * apple + orange/3$ 等于50。我们可以按一个约束条件列出所有可行的情况,然后对每个可能解检查它是否满足第二个约束条件。按照第一个约束条件,至少必须买一个西瓜,至多只能买9个西瓜,因此可能的西瓜数的变化范围是1~9。当西瓜数确定后,剩下的钱是50减去5乘以西瓜数,这些钱可以用来买苹果和橘子。至少必须买一个苹果,至多只能买 $50-5 \times \text{西瓜个数}-1$ 个苹果。剩余的钱都可以用来买橘子,一共可以买 $3 \times (50-5 \times \text{西瓜数}-\text{苹果数})$ 个橘子。这就是一个候选方案。对此方案检查是否满足第二个约束条件,即所有水果数之和为100个。如果满足则输出此方案。按照上述思想可以得到代码清单4-5所示的程序。

代码清单4-5 水果问题的程序

```
//文件名: 4-5.cpp  
//水果问题求解  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int mellon, apple, orange;           //分别表示西瓜数、苹果数和橘子数  
    for (mellon=1; mellon<10; ++mellon) //对每种可能的西瓜数  
        for (apple=1; apple < 50 - 5 * mellon; ++apple) { //当西瓜数给定后可能的苹果数  
            orange = 3*(50-5*mellon-apple);           //剩下的钱全买了橘子  
        }  
}
```



```

        if (mellon+apple+orange == 100){                //3种水果数之和是否为100
            cout << "mellon:" << mellon << ' ';
            cout << "apple:" << apple << ' ';
            cout << "orange:" << orange << endl;
        }
    }
    return 0;
}

```

代码清单4-5所示的程序运行结果如下：

```

Mellon: 1  apple: 18  orange: 81
Mellon: 2  apple: 11  orange: 87
Mellon: 3  apple: 4   orange: 93

```

例4.6 用枚举法解答第3章开头所举的四大湖问题。

可以假设：洞庭湖最大，洪泽湖第二，鄱阳湖第三，太湖第四，然后检查每位同学是否都讲对了一个。如果不是，再尝试下一种情况：洞庭湖最大，洪泽湖第二，鄱阳湖第四，太湖第三，再检查每位同学是否都讲对了一个。尝试所有可能的情况，直到满足每位同学都讲对一个为止。

如果用a、b、c、d分别表示4个湖：a表示洞庭湖，b表示洪泽湖，c表示鄱阳湖，d表示太湖。为了尝试所有情况，需要假设洞庭湖可能是最大，也可能是第二、第三或第四。因此，a的值可能从1变到4。同样，b、c、d的值也都可能从1变到4。为此，我们需要一个控制结构，使a、b、c、d的值能自动从1变到4。这种结构就是for循环结构。a、b、c、d的每种排列都是一个候选解。对每一个候选解，检验4个学生的回答，如果对此候选解，每位同学都答对了一个，则此候选解是正确解。按照此思路设计的程序如代码清单4-6所示。

代码清单4-6 四大湖问题的程序

```

//文件名：4-6.cpp
//求解四大湖问题
#include <iostream>
using namespace std;

int main()
{
    int a, b, c, d;
    for (a=1; a<=4; ++a)
        for (b=1; b<=4; ++b)
            if (a == b) continue;                //洞庭湖和洪泽湖的排名不能并列
            else for (c=1; c<=4; ++c)
                if (c==a||c==b) continue;        //洞庭湖、洪泽湖和鄱阳湖的排名不能并列
                else { d = 10 - a - b - c;        //确定太湖的排名
                    if ((a==1)+(b==4)+(c==3))==1
                        &&((b==1)+(a==4)+(c==2)+(d==3))==1
                        &&((b==4)+(a==3))==1
                        &&((c==1)+(d==4)+(b==2)+(a==3))==1)
                        cout << "洞庭湖第" << a << "\t洪泽湖第" << b
                            << "\t鄱阳湖第" << c << "\t太湖第" << d << endl;
                }
}

```

```
    return 0;  
}
```

代码清单4-6所示的程序运行结果如下：

洞庭湖第2 洪泽湖第4 鄱阳湖第1 太湖第3

这个程序有一个效率的问题，它会尝试所有的情况。事实上，该问题只有一个合法解，当找到了一种合法的排列后就不需要再找下去了。为了达到此目的，可以设置一个标志，当找到答案后，将标志置为true。在每个循环的条件表达式中加入标志的检测。修改后的程序如代码清单4-7所示。

代码清单4-7 四大湖问题的改进版

```
//文件名: 4-7.cpp  
//四大湖问题的改进版  
#include <iostream>  
using namespace std;  
  
int main()  
{    int a, b, c, d;  
    bool flag = false;  
    for (a=1; a<=4 && !flag; ++a)  
        for (b=1; b<=4 && !flag; ++b)  
            if (a == b) continue;  
            else for (c=1; c<=4 && !flag; ++c)  
                if (c==a||c==b) continue;  
                else { d = 10 - a - b - c;  
                    if (((a==1)+(b==4)+(c==3))==1  
                        &&((b==1)+(a==4)+(c==2)+(d==3))==1  
                        &&((b==4)+(a==3))==1  
                        &&((c==1)+(d==4)+(b==2)+(a==3))==1) {  
                        cout << "洞庭湖第" << a << "\t\t洪泽湖第" << b  
                            << "\t\t鄱阳湖第" << c << "\t\t太湖第" << d << endl;  
                        flag = true;  
                    }  
                }  
    return 0;  
}
```

在设计一个算法时，要时刻记住提高它的效率。

4.6 贪焚法

贪焚法是一种不追求最优解，只希望得到较为满意的解的方法。因为它省去了为找最优解而穷尽所有可能所需的时间，所以贪焚法一般可以快速得到满意的解。贪焚法在求解过程的每一步都选取一个局部最优的策略，把问题规模缩小，最后把每一步的结果合并起来形成一个全局解。

下面是贪焚法的基本步骤。

(1) 从某个初始解出发。

(2) 采用迭代的过程，当可以向目标前进一步时，就根据局部最优策略，得到一个部分解，缩小问题规模。

(3) 将所有解综合起来。

贪婪法是一种很实用的方法。在日常生活中，我们经常用贪婪法来寻找问题的解。例如，在平时购物找零钱时，为使找回的零钱的硬币数最少，我们不会穷举所有的方法，而是从最大面值的硬币开始，按递减顺序考虑各种面值的硬币。先尽量用最大面值的硬币，当剩下的找零值比最大硬币值小的时候，才考虑第二大面值的硬币。当剩下的找零值比第二大面值的硬币值小的时候，才考虑第三大面值的硬币。这就是在采用贪婪法。贪婪法在这种场合总是最优的，因为银行对其发行的硬币种类和硬币面额进行了巧妙的设计。但是，如果硬币的面额分别为25、21、10、5、2、1时，贪婪法就不一定能得到最优解。如要找63分钱，用贪婪法得到的结果是25、25、10、1、1、1，一共要6枚硬币。但最优解是3枚21分的硬币。

例4.7 用贪婪法解硬币找零问题。

假设有一种货币，它有面值为1分、2分、5分和1角的硬币，最少需要多少个硬币来找出K分钱的零钱。按照贪婪法的思想，需要不断地使用面值最大的硬币。如要找零的值小于最大的硬币值，则尝试第二大的硬币，依次类推。依据这个思想，可得到代码清单4-8所示的程序。

代码清单4-8 贪婪法解硬币找零问题的程序

```
//文件名: 4-8.cpp
//贪婪法解硬币找零问题
#include<iostream>
using namespace std;

#define ONEFEN 1
#define TWOFEN 2
#define FIVEFEN 5
#define ONEJIAO 10

int main()
{
    int money;
    int onefen = 0, twofen = 0, fivefen = 0, onejiao = 0;

    cout << "输入要找零的钱（以分为单位）: ";    cin >> money;

    //不断尝试每一种硬币
    while (money >= ONEJIAO) {onejiao++; money -= ONEJIAO;}
    while (money >= FIVEFEN) {fivefen++; money -= FIVEFEN;}
    while (money >= TWOFEN) {twofen++; money -= TWOFEN;}
    while (money >= ONEFEN) {onefen++; money -= ONEFEN;}

    //输出结果
    cout << "1角硬币数: " << onejiao << endl;
    cout << "5分硬币数: " << fivefen << endl;
    cout << "2分硬币数: " << twofen << endl;
```

```
    cout << "1分硬币数: " << onefen << endl;  
  
    return 0;  
}
```

小结

计算机的强项是不厌其烦地做同样的操作，这是通过循环语句实现的。本章介绍了C++的循环语句：`while`、`do-while`和`for`。`while`语句用来指示在一定条件满足的情况下重复执行某些操作。`while`语句先判断条件再执行循环体，因此循环体可能一次都不执行。`do-while`类似于`while`循环，其区别是`do-while`循环先执行一次循环体，然后再判断是否要继续循环。`for`语句实现循环次数一定的重复操作。`for`循环一般设置一个循环变量，记录已执行的循环次数，在每个循环周期中要更新循环变量的值。

计算机中许多解决问题的方法都是基于循环的。本章介绍了枚举法和贪婪法：枚举法列举出所有可能的解决方案，从中筛选出正确的答案；贪婪法用于求最优解的情况，它适合的情况是找出问题的解需要多个阶段。贪婪法是在每个阶段寻找最优解，而不考虑全局的优化。

习题

简答题

1. 假设在`while`语句的循环体中有这样一条语句：当它执行时`while`循环的条件值就变为`false`。那么这个循环是将立即中止还是要完成当前周期呢？
2. 当遇到下列4个情况时，你将怎样编写`for`语句的控制行。
 - a. 从1计数到100。
 - b. 从2, 4, 6, 8, ...计数到100。
 - c. 从0开始，每次计数加7，直到成为三位数。
 - d. 从100开始，反向计数，99, 98, 97, ...直到0。
3. 为什么在`for`循环中最好避免使用浮点型变量作为循环变量？
4. 在程序

```
for (i = 0; i < n; ++i)  
    for (j = 0; j < i; ++j) cout << i << j;
```

中，`cout << i << j`；执行了多少次？

程序设计题

1. 编写这样一个程序：先读入一个正整数 N ，然后计算并显示前 N 个奇数的和。例如，如果 N 为4，这个程序应显示16，它是 $1+3+5+7$ 的和。
2. 在数学中，有一个非常著名的斐波那契数列，它是按13世纪意大利著名数学家Leonardo

Fibonacci的名字命名的。这个数列的前两个数是0和1，之后每一个数是它前两个数的和。因此斐波那契数列的前几个数为：

$$F_0=0$$

$$F_1=1$$

$$F_2=1 \quad (0+1)$$

$$F_3=2 \quad (1+1)$$

$$F_4=3 \quad (1+2)$$

$$F_5=5 \quad (2+3)$$

$$F_6=8 \quad (3+5)$$

编写一个程序，顺序显示 F_0 到 F_{15} 。

3. 编写一个程序，要求输入一个整型数 N ，然后显示一个由 N 行组成的三角形。在这个三角形中，第一行一个“*”，以后每行比上一行多两个“*”，三角形像下面这样尖角朝上。

```

      *
     ***
    *****
   *********
  ***********
 *****
*****
*****
*****

```

4. 编写一个程序，要求能自动出1~100范围内的+、-、×、÷四则运算的题目，然后让用户输入答案，并由程序判别是否正确。若不正确，则要求用户订正；若正确，则出下一题。另外还有下列要求：差不能为负值，除数不能为0，用户的输入不管正确与否都要给出一些信息，并且信息不能太单一。例如，输入正确时，可能会输出you are right，也可能会输出it's OK。
5. 编写一个程序，输出3个字母A、B、C的所有排列方式。
6. 编写一个程序求 $\sum_{n=1}^{30} n!$ ，要求只做 n 次乘法和 n 次加法。

例4.3要求编一个程序输出某个班级某次考试中的最高分、最低分和平均分。如果用户还有一个要求，希望为每位学生打印一张成绩单，其中包括该学生的成绩、班级最高分、班级最低分和平均分。这时我们必须保存每位学生的考试成绩，等到最高分、最低分和平均分统计出来后再打印成绩单。在这个程序中，如何保存每个学生的成绩就成为一个难以解决的问题。如果每位学生的成绩用一个整型变量来保存，那么该程序必须定义许多整型变量。有100位学生就要定义100个变量。这样做有两个问题：第一，每个班级有很多学生，就必须定义许多变量，使程序变得冗长，而且每个班级的人数不完全相同，到底应该定义多少个变量也是一个问题；第二，因为每位学生的成绩都是放在不同的变量中，因此统计成绩时循环就无法使用。

为了解决这类处理大批量同类数据的问题，程序设计语言提供了一个叫数组的组合数据类型。C++也不例外。

5.1 一维数组

在程序设计语言中，一维数组是一个有序数据的集合。数组中的每个元素都有同样的类型。用一个统一的数组名和集合中的位置唯一确定一个数组中的元素。集合中的位置称为数组元素的下标。

5.1.1 一维数组的定义

定义一个一维数组要说明3个问题：第一，数组是一个变量，应该有一个变量名；第二，数组有多少个元素；第三，每个元素的数据类型是什么。综合上述3点，C++中一维数组的定义方式如下：

类型名 数组名[常量表达式]；

其中，类型名指出了数组元素的数据类型，数组名是存储该数组的变量名，常量表达式指出数组中元素的个数。在数组定义中特别要注意的是数组的元素个数是用一个常量表达式说明的。也就是说，元素个数在写程序时就已经确定。

5.1.2 数组元素的引用

定义了一个数组，相当于定义了一组变量。例如：

```
int a[10];
```

相当于定义了10个变量a [0],a [1],⋯,a [9]。数组的下标从0开始。数组元素的表示方式为：数组名[下标]。下标可以是常量、变量或任何计算结果为整型的表达式。这样就可以使数组元素的引用变得相当灵活。例如，要访问数组a的10个元素，只需要用一个for循环。让循环变量i从0变到9，引用a[i]，就访问了数组的所有元素。

5.1.3 一维数组的初始化

可以在定义数组时为数组元素赋初值，这称为数组的初始化。数组的初始化可用以下3种方法实现。

(1) 在定义数组时对数组元素赋初值。例如：

```
int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

表示将0, 1, 2, 3, 4, 5, 6, 7, 8, 9依次赋给a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9]。

(2) 可以对数组的一部分元素赋值。例如：

```
int a[10] = { 0, 1, 2, 3, 4};
```

表示数组a的前5个元素的值分别是0, 1, 2, 3, 4，后5个元素的值为0。在对数组元素赋初值时，总是按从下标小的元素到下标大的元素的次序。没有赋到初值的元素的初值为0。因此，想让数组的所有元素的初值都为0，可简单地写成：

```
int a[10] = {0};
```

(3) 在对全部数组元素赋初值时，可以不指定数组大小，系统根据给出的初值的个数确定数组的规模。例如：

```
int a[ ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

表示a数组有10个元素，它们的初值分别为0, 1, 2, 3, 4, 5, 6, 7, 8, 9。

5.1.4 一维数组在内存中的表示

定义数组就是定义了一块连续的空间，空间的大小等于元素数乘以每个元素所占的空间大小。数组元素按序存放在这块空间中。例如，若在Visual C++中定义int intarray[5];，则数组intarray占用了20个字节，因为每个整型数占4个字节。如果这块空间的起始地址为100，那么100~103存放intarray[0]，104~107存放intarray[1]，⋯⋯，116~119存放intarray[4]。

当引用变量intarray[idx]时，系统计算它的地址100+idx×4，对该地址的内容进行操作。数组名本身给出了存储该数组的空间的起始地址。例如，若给intarray[3]赋值为3，则内存中的情况如下：

随机值	随机值	随机值	3	随机值
100	103 104	107 108	111 112	115 116 119

在使用数组时必须注意：C++语言不检查数组下标的越界。例如，定义数组int intarray

[10];合法的下标范围是0~9,但如果你引用intarray[10],系统不会报错。若数组intarray的起始地址是1000,引用intarray[10]时,则系统会对地址为1040的内存单元进行操作,而1040可能是另一个变量的地址。因此在写数组操作的程序时,程序员必须保证下标的合法性,否则程序的运行会出现不可预知的结果。

5.1.5 一维数组的应用

例5.1 定义一个10个元素的整型数组。由用户输入10个元素的值,并将结果显示在屏幕上。代码清单5-1给出了这个程序。

代码清单5-1 数组的输入/输出

```
//文件名: 5-1.cpp

//数组输入/输出示例
#include <iostream>
using namespace std;

int main()
{   int a[10], i;

    cout << "请输入10个整型数: \n";
    for (i=0; i<10; ++i)   cin >> a[i];

    cout << "\n数组的内容为: \n";
    for (i=0; i< 10; ++i)   cout << a[i] << '\t';

    return 0;
}
```

代码清单5-1所示的程序的一次运行结果如下:

```
请输入10个整型数:
1 2 3 4 5 6 7 8 9 0
数组的内容为:
1      2      3      4      5      6      7      8      9      0
```

例5.2 编写一个程序,统计某个班级某次考试中的最高分、最低分和平均分,并为每位学生打印一张成绩单。其中包括该学生的编号、该学生的成绩、班级最高分、班级最低分和平均分。

解决这个问题的关键在于必须把每个学生的成绩保存起来,这可以用一个一维整型数组来实现。但每个班的学生人数不完全一样,数组的大小应该为多少呢?我们可以按照人数最多的班级确定数组的大小。例如,若每个班级最多允许有50个学生,那么数组的大小就定义为50。如果某个班的学生数少于50,如45个学生,就用该数组的前45个元素。在这种情况下,定义的数组大小称为数组的配置长度,而真正使用的部分称为数组的有效长度。

打印成绩单的程序如代码清单5-2所示。

代码清单5-2 打印学生成绩单

```

//文件名: 5-2.cpp
//统计及打印成绩单
#include <iostream>
using namespace std;
#define MAX 50 //定义一个班级中最多的学生数
int main()
{
    int score[MAX], max = 0, min = 100, sum = 0, num, average, i;
    //max为最高分, min为最低分, sum为总分, num为学生数, average为平均分

    cout << "请输入成绩 (-1表示结束): \n";
    for (num = 0; num < MAX; ++num){
        cin >> score[num];
        if (score[num] == -1) break;
        sum += score[num];
        if (score[num] > max) max = score[num];
        if (score[num] < min) min = score[num];
    }

    average = sum / num; //计算平均成绩

    cout << "\n成绩单: \n";
    cout << "编号\t成绩\t最高分\t最低分\t平均分\n";
    for (i = 0; i < num; ++i)
        cout << i << '\t' << score[i] << '\t' << max << '\t' << min << '\t' << average
            << '\n';

    return 0;
}

```

代码清单5-2所示的程序的某次运行结果如下:

请输入成绩 (-1表示结束):

80 70 50 95

67 73 88 91 85 -1

成绩单:

编号	成绩	最高分	最低分	平均分
0	80	95	50	77
1	70	95	50	77
2	50	95	50	77
3	95	95	50	77
4	67	95	50	77
5	73	95	50	77
6	88	95	50	77
7	91	95	50	77
8	85	95	50	77

5.2 查找和排序

5.2.1 查找

一维数组的一个重要的操作就是在数组中找出某个特定的元素, 如果找到了, 则输出该元素

的存储位置。最基本、最直接的查找方法就是顺序查找，但对于已排好序的表，二分查找又比顺序查找更有效。下面就分别介绍这两种查找方法。

1. 顺序查找

从数组的第一个元素开始，依次往下比较，直到找到要找的元素，输出元素的存储位置，若到数组结束还没找到要找的元素则输出错误信息。这种查找方法即为顺序查找。

例5.3 在一批整型数据2, 3, 1, 7, 5, 8, 9, 0, 4, 6中查找某个元素 x 是否出现。

解决这个问题首先要设置一个数组，把这组数据存储起来。然后输入用户要查找的数据，用顺序查找的方法在数组中查找 x 是否出现。解决这个问题的程序如代码清单5-3所示。

代码清单5-3 顺序查找

```
//文件名: 5-3.cpp
//顺序查找
#include <iostream>
using namespace std;

int main()
{   int k, x;
    int array[ ] = { 2, 3, 1, 7, 5, 8, 9, 0, 4, 6};

    cout << "请输入要查找的数据: ";
    cin >> x;

    for (k = 0; k < 10; ++k)
        if (x == array[k]) { cout << x << "的存储位置为: " << k; break;}
    if (k == 10) cout << "没有找到";

    return 0;
}
```

输入为7时，程序的运行结果如下：

```
请输入要查找的数据: 7
7的存储位置为3
```

输入为10时，程序的运行结果如下：

```
请输入要查找的数据: 10
没有找到
```

2. 二分查找

顺序查找的实现相当简单明了。但是，如果被查找的数组很大，要查找的元素又靠近数组的尾端或在数组中根本不存在，则查找的时间可能就会很长。设想一下，在一本5万余词的《新英汉词典》中顺序查找某一个单词，最坏情况下就要比较5万余次。在手工的情况下，几乎是不可能实现的。但为什么我们能在词典中很快找到要找的单词呢？关键就在于《新英汉词典》是按字母序排序的。当要在词典中找一个单词时，我们不会从第一个单词检查到最后一个单词，而是先估计一下这个词出现的大概位置，然后翻到词典的某一页，如果翻过头了，则向前修正，如太靠前面了，则向后修正。

如果待查数据是已排序的，则可以按照查词典的方法进行查找。在查词典的过程中，因为对单词分布情况的了解，所以一下子就能找到比较接近的位置。但是一般的情况下我们不知道待查数据的分布情况，所以只能采用比较机械的方法，每次检查待查数据中排在最中间的元素。如果中间元素等于要查找的元素，则查找完成；否则，确定要找的数据是在前一半还是在后一半，然后缩小范围，在前一半或后一半内继续查找。例如，要在图5-1所示的城市表中查找是否有San Francisco。

0	Atlanta	(亚特兰大)
1	Boston	(波士顿)
2	Chicago	(芝加哥)
3	Denver	(丹佛)
4	Detroit	(底特律)
5	Houston	(休斯敦)
6	Los Angeles	(洛杉矶)
7	Miami	(迈阿密)
8	New York	(纽约)
9	Philadelphia	(费城)
10	San Francisco	(旧金山)
11	Seattle	(西雅图)

图5-1 待查找的城市表

此时不再考虑采用从数组第一个元素开始进行的顺序查找，而是在数组的接近中间的位置取一个值。中间元素的下标值可以通过两个端点的均值得到，即 $(0+11)/2$ 。使用整数进行运算时，这个表达式的值为5。

存储在城市表中的位置5对应的城市名是Houston。因为此时还没有找到San Francisco，因此需要继续查找。另一方面，你知道San Francisco所在位置一定是在Houston的后面，因为这个数组是按照字母序排序的，因此可以立即排除下标值从0到5的城市名（见图5-2），然后在余下的位置中查找。

0	Atlanta
1	Boston
2	Chicago
3	Denver
4	Detroit
5	Houston
6	Los Angeles
7	Miami
8	New York
9	Philadelphia
10	San Francisco
11	Seattle

图5-2 排除下标值从0到5的城市名后的状态

仅仅通过上面的一步操作，已经删去了表中一半的城市名。用同样的方法继续查找。如果San Francisco在城市表中的话，那么它的位置一定是在6~11。用前面提到的方法再计算该范围的中心： $(6+11)/2$ 。由于是整数运算，因此这个表达式的值是8。城市表中的位置8对应的城市名是New York。同样由字母序可知，San Francisco一定在New York的后面，那么又排除了3个城市（见图5-3）。

0	Atlanta
1	Boston
2	Chicago
3	Denver
4	Detroit
5	Houston
6	Los Angeles
7	Miami
8	New York
9	Philadelphia
10	San Francisco
11	Seattle

图5-3 排除下标值为0~8的城市名后的状态

当在剩下的范围中再找中间位置时，得到位置10，而位置10中存储的正好是San Francisco，因此整个查找过程只需要3次比较。

首先在排好序的表中查找中间元素，然后根据这个元素的值确定下一步将在哪一半进行查找，这种查找方法称为二分查找。为了实现这种查找方法，需要记录两个下标值，即分别表示要被搜索范围的两个端点的下标值。这两个值分别存储于变量lh和rh中，表示左边界（小的下标值）和右边界（大的下标值）。开始时，搜索范围覆盖整个数组，而随着查找的继续进行，搜索区间将逐渐缩小。如果最后两个下标值交叉了，那么表示所要查找的值不在数组中。

例5.4 在已排序的一批整型数据0, 1, 2, 3, 4, 5, 6, 7, 8, 9中查找某个元素x是否出现。

使用二分查找解决这个问题的程序如代码清单5-4所示。

代码清单5-4 二分查找程序

```
//文件名: 5-4.cpp
//二分查找
#include <iostream>
using namespace std;

int main()
{   int lh, rh, mid, x;
    int array[ ]={0,1,2,3,4,5,6,7,8,9};

    cout <<"请输入要查找的数据: ";   cin >> x;

    lh = 0;   rh = 9;
    while ( lh <= rh )                    //查找区间存在
```

```
{ mid = ( lh + rh ) / 2;          //计算中间位置
  if ( x== array[mid] ) {
    cout << x << "的位置是: " << mid << endl;
    break;
  }
  if ( x < array[mid] ) rh = mid - 1; else lh = mid + 1;
}
if (lh > rh) cout << "没有找到" << endl;

return 0;
}
```

输入7时，程序的运行结果如下：

请输入要查找的数据：7
7的存储位置为7

输入10时，程序的运行结果如下：

请输入要查找的数据：10
没有找到

由上述讨论可知，二分查找算法比顺序查找算法更有效。在二分查找中，比较的次数取决于所要查找的元素在数组中的位置。对于n个元素的数组，在最坏的情况下，所要查找的元素必须查到查找区间只剩下一个元素时才能找到或者该元素根本不在数组中，那么在第一次比较后，所要搜索的区间立刻减小为原来的一半，只剩下n/2个元素。在第二次比较后，再去掉这些元素的一半，剩下n/4个元素。每次，被查找的元素数都减半。最后搜索区间将变为1，即只需要将这个元素与需要查找的元素进行比较。达到这一点所需的步数等于将n依次除以2并最终得到1所需要的次数，可以表示为如下公式：

$$\underbrace{n/2/2/\cdots/2/2=1}_{k\text{次}}$$

将所有的2乘起来得到以下方程：

$$n = 2^k$$

因此很容易得到k的值：

$$k = \text{lb } n$$

所以，使用二分查找算法最多只需要lb n次比较就可以了。

表5-1给出了不同的n值和它相对应的最精确的lb n的整数值。

表5-1 n与lb n

n	lb n
10	3
100	7
1000	10
1 000 000	20
1 000 000 000	30

从表5-1中的数据可以看出,对于小的数组,两种算法都能很好地完成搜索任务。然而,如果该数组的元素个数为1 000 000 000,在最坏的情况下顺序查找算法需要1 000 000 000次比较才能查找完毕,而二分查找算法最多也仅仅需要30次比较就能查找完毕。

5.2.2 排序

在5.2.1节中我们已经看到,如果待查数据是有序的,则可以大大降低查找时间。因此对于一大批需要经常查找的数据而言,事先对它们进行排序是有意义的。

排序的方法有很多,如插入排序、选择排序、交换排序等。下面介绍两种比较简单的排序方法:直接选择排序法和冒泡排序法。

1. 直接选择排序法

在众多排序算法中,最容易理解的一种就是选择排序算法。应用选择排序时,每次将其中的一个元素放在它最终要放的位置。第一步是找到整个数组中的最小的元素并把它放在它最终要放的位置上,即数组的起始位置,第二步是在剩下的元素中找最小的元素并把它放在第二个位置上。对整个数组继续这个过程,最终将得到按从小到大顺序排列的数组。不同的最小元素选择方法得到不同的选择排序算法。直接选择排序是选择排序算法中最简单的一种,就是在找最小元素时采用最原始的方法——顺序查找。

为了解直接选择排序算法,举以下这个数组作为例子。

31	41	59	26	53	58	97	93
0	1	2	3	4	5	6	7

由于这个数组中最小的元素值是26,它在数组中的位置是3,因此需要将它移动到位置0。经过交换位置0和位置3的数据得到新的数组如下。

正确的位置

26	41	59	31	53	58	97	93
0	1	2	3	4	5	6	7

位置0中就是该数组的最小值,符合最终的排序要求。

现在,可以处理表中的剩下部分。下一步是用同样的策略正确填入数组的位置1中的值。最小的值(除了26已经被正确放置外)是31,现在它的位置是3。如果将它的值和位置为1的元素的值进行交换,可以得到下面的状态,前两个元素是正确的值。

正确的排序

26	31	59	41	53	58	97	93
0	1	2	3	4	5	6	7

在下一个周期中,再将下一个最小值(应该是41)和位置2中的元素值进行交换。

正确的排序							
26	31	41	59	53	58	97	93
0	1	2	3	4	5	6	7

如果继续这个过程，将正确添加入位置3和位置4，依次类推，直到数组被完全排序。

为了弄清楚在整个算法中具体的某一步该对哪个元素进行操作，可以想象用你的左手依次指明每一个下标位置。对每一个左手位置，可以用你的右手找出剩余的元素中的最小元素。一旦找到这样的元素，就可以把两个手指指出的值进行交换。在实现中，你的左手和右手分别用两个变量lh和rh来代替，它们分别代表相应的元素在数组中的下标值。

上述过程可以用下面的伪代码表示：

```
for (数组中每一个下标lh) {  
    设rh是从lh直到数组结束的所有元素中最小值元素的下标，  
    将lh位置和rh位置的值进行交换；  
}
```

要将这段伪代码转换成C++语句不是很难，只要使用一个嵌套的for循环即可。

直接选择排序算法本身有很多优点。首先，它的算法很容易理解；其次，它解决了排序这个问题。但是，还存在一些其他的更有效的排序算法。但效率较高的排序算法需要较高的程序设计技巧，这些复杂的排序算法将在数据结构这门课中学习。

例5.5 采用直接选择排序法对一个其元素分别为2, 5, 1, 9, 10, 0, 4, 8, 7, 6的数组进行排序。

采用直接选择排序法解决这个问题的程序如代码清单5-5所示。

代码清单5-5 直接选择排序的程序

```
//文件名: 5-5.cpp  
//直接选择排序  
#include <iostream>  
using namespace std;  
  
int main( )  
{  
    int lh, rh, k, tmp;  
    int array[ ] = {2, 5, 1, 9, 10, 0, 4, 8, 7, 6};  
  
    for (lh = 0; lh < 10; lh++)  
    {  
        rh = lh;  
        for (k = lh; k < 10; ++k)  
            if ( array[k] < array[rh] )    rh = k;  
        tmp = array[lh]; array[lh] = array[rh]; array[rh] = tmp;  
    }  
  
    for (lh =0; lh<10; ++lh)  cout << array[lh] << ' ';  
  
    return 0;  
}
```

2. 冒泡排序法

冒泡排序法的思想是：从头到尾比较相邻的两个元素，将小的换到前面，大的换到后面。经

过了从头到尾的一趟比较，就把最大的元素交换到了最后一个位置。这个过程称为一趟起泡。然后再从头开始到倒数第二个元素进行第二趟起泡。比较相邻元素，如违反排好序后的次序，则交换相邻两个元素。经过了第二趟起泡，又将第二大的元素放到了倒数第二个位置，……，依次类推，经过第 $n-1$ 趟起泡，将倒数第 $n-1$ 个大的元素放入位置1。此时，最小的元素就放在了位置0，完成排序。冒泡排序法的伪代码如下：

```
for (i=1; i<n; ++i)
    从元素0到元素n-i进行起泡，最大的元素放入位置n-i;
```

一般来讲， n 个元素的冒泡排序需要 $n-1$ 趟起泡，但如果在一趟起泡过程中没有发生任何数据交换，则说明这批数据中相邻元素都满足前面小后面大的次序，也就是这批数据已经是排好序了。这时没有必要再进行后续的起泡了，排序可以结束。因此，冒泡排序法的伪代码可以进一步细化成下面的形式：

```
for (i=1; i<n; ++i) {
    从元素0到元素n-i进行起泡，最大的泡放入元素n-i;
    if (没有发生过数据交换) break;
}
```

例5.6 用冒泡排序法对一个含有11个元素的数组进行排序。

用冒泡排序法排序 n 个整型数据的程序如代码清单5-6所示。为了表示在一趟起泡中有没有发生过交换，我们定义了一个bool类型的变量flag。在起泡前将flag设为false。在起泡过程中如果发生交换，将flag置为true。当一趟起泡结束后，如果flag仍为false，则说明没有发生过交换，可以结束排序。

代码清单5-6 整型数的冒泡排序的程序

```
//文件名: 5-6.cpp
//冒泡排序
#include <iostream>
using namespace std;

int main()
{
    int a[ ] = { 0, 3, 5, 1, 8, 7, 9, 4, 2, 10, 6};
    int i, j, tmp, n = 11;
    bool flag; //记录一趟起泡中有没有发生过交换

    for (i=1; i<n; ++i)
    {
        flag = false;
        for (j=0; j<n-i; ++j)
            if (a[j+1] < a[j])
                {tmp = a[j]; a[j] = a[j+1]; a[j+1] = tmp; flag = true;}
        if (!flag) break; //一趟起泡中没有发生交换，排序结束
    }

    cout << endl;
    for (i=0; i<n; ++i) cout << a[i] << ' ';

    return 0;
}
```


5.3 二维数组

数组的每一个元素又是数组的数组称为多维数组。最常用的多维数组是二维数组，即每一个元素是一个一维数组的一维数组。

5.3.1 二维数组的定义

二维数组可以看成数学中的矩阵，它由行和列组成。声明一个二维数组必须说明它有几行几列。二维数组定义的一般形式如下：

```
类型名 数组名[常量表达式1][常量表达式2];
```

类型名是二维数组中每个元素的类型，常量表达式1给出二维数组的行数，常量表达式2给出了二维数组的列数。当把二维数组看成是元素为一维数组的数组时，也可以把常量表达式1看成是一维数组的元素个数，常量表达式2是每个元素（也是一个一维数组）中元素的个数。例如，定义

```
int a[4][5];
```

表示定义了一个由4行组成，每一行有5个整型元素组成的二维数组a。也可以看成定义了一个有4个元素的一维数组，每个元素的类型是一个由5个元素组成的一维数组。

一旦定义了数组a，就相当于定义了20个整型变量，即a[0][0], a[0][1], ..., a[0][4], ..., a[3][0], a[3][1], ..., a[3][4]。第一个下标表示行号，第二个下标表示列号。例如，a[2][3]是数组a的第二行第三列的元素。同一维数组一样，下标的编号也是从0开始的。

5.3.2 二维数组的初始化

可以用以下3种方法对二维数组初始化。

(1) 对所有的元素赋初值。例如：

```
int a[3][4] = { 1,2,3,4,5,6,7,8,9,10,11,12};
```

编译器依次把花括号中的值赋给第一行的每个元素，然后是第二行的每个元素，依次类推。初始化后的数组元素如下所示：

1	2	3	4
5	6	7	8
9	10	11	12

可以通过花括号把每一行括起来使这种初始化方法表示得更加清晰：

```
int a[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

(2) 对部分元素赋值。同一维数组一样，二维数组也可以对部分元素赋值。计算机将初始化列表中的数值按行序依次赋给每个元素，没有赋到初值的元素初值为0。例如：

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

初始化后的数组元素如下所示：

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(3) 对每一行的部分元素赋初值。例如：

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

初始化后的数组元素如下所示：

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 5 & 0 & 0 & 0 \end{bmatrix}$$

5.3.3 二维数组在内存中的表示

一旦定义了一个二维数组，系统就在内存中准备了一块连续的空间，数组的所有元素都存放在这块空间中。在内存中，二维数组的元素是按行序存放的，即先放第一行的元素，然后放第二行的元素。例如，若在Visual C++中定义了整型数组a[3][4]则该数组在内存中占48个字节。数组元素的存放次序如图5-4所示。

a[0][0]
a[0][1]
...
a[0][4]
a[1][0]
...
a[3][4]

图5-4 二维数组的内部表示

同一维数组一样，在引用二维数组的元素时也不检查下标的合法性。下标的合法性必须由程序员自己保证。

5.3.4 二维数组的应用

例5.7 矩阵的乘法。二维数组的一个主要的用途就是矩阵运算，矩阵的乘法就是其中之一。矩阵 $C=A \times B$ 要求 A 的列数等于 B 的行数。若 A 是 l 行 m 列， B 是 m 行 n 列，则 C 是 l 行 n 列的矩阵。它的每个元素的值为 $c[i][j] = \sum_{k=1}^m a[i][k] \times b[k][j]$ 。试设计一程序输入两个矩阵 A 和 B ，输出矩阵 C 。

设计这个程序的关键是计算 $c[i][j]$ 。对于矩阵 C 的每一行计算它的每一列的元素值，这需要一个两重循环。每个 $c[i][j]$ 的计算是对 A 矩阵的第 i 行和 B 矩阵的第 j 列元素对应相乘后求和，这又需要一个循环。所以程序的主体是由一个三重循环构成。具体程序如代码清单5-7所示。

代码清单5-7 矩阵乘法的程序

```
//文件名：5-7.cpp
//矩阵乘法
#include <iostream>
using namespace std;
#define MAX_SIZE 10 //矩阵的最大规模

int main()
{
    int a[MAX_SIZE][MAX_SIZE], b[MAX_SIZE][MAX_SIZE], c[MAX_SIZE][MAX_SIZE];
    int i, j, k, NumOfRowA, NumOfColA, NumOfColB;
```

```

//输入A和B的大小
cout << "\n输入A的行数、列数和B的列数: ";
cin >> NumOfRowA >> NumOfColA >> NumOfColB;

//输入A
cout << "\n输入A:\n";
for (i=0; i< NumOfRowA; ++i)
    for (j=0; j < NumOfColA; ++j) {
        cout << "a[" << i << "][" << j << "] = ";
        cin >> a[i][j];
    }

//输入B
cout << "\n输入B:\n";
for (i=0; i< NumOfColA; ++i)
    for (j=0; j< NumOfColB; ++j) {
        cout << "b[" << i << "][" << j << "] = ";
        cin >> b[i][j];
    }

//执行A×B
for (i=0; i< NumOfRowA; ++i)
    for (j=0; j< NumOfColB; ++j) {
        c[i][j] = 0;
        for (k=0; k<NumOfColA; ++k)    c[i][j] += a[i][k] * b[k][j];
    }

//输出C
cout << "\n输出C:";
for (i=0; i < NumOfRowA; ++i) {
    cout << endl;
    for (j=0; j< NumOfColB; ++j)    cout << c[i][j] << '\t';
}

return 0;
}

```

例5.8 N 阶魔阵是一个 $N \times N$ 的由1到 N^2 之间的自然数构成的矩阵。它的每一行、每一列和对角线之和均相等。例如，一个三阶魔阵如下所示，它的每一行、每一列和对角线之和均为15：

8	1	6
3	5	7
4	9	2

编写一个程序打印任意 N 阶魔阵。

想必每个人小时候都曾绞尽脑汁填过这样的魔阵。事实上，有一个很简单的方法可以生成这个魔阵。

依次将1到 N^2 填入矩阵，填入的位置由如下规则确定。

- 第一个元素放在第一行中间一列。

- 下一个元素存放在当前元素的上一行、下一列。
- 如上一行、下一列已经有内容，则下一个元素的存放位置为当前列的下一行。

在找上一行、下一行或下一列时，必须把这个矩阵看成是回绕的。也就是说，如果当前行是最后一行时，下一行为第0行；当前列为最后一列时，下一列为第0列；当前行为第0行时，上一行为最后一行。

有了上述规则，生成 N 阶魔阵的算法可以表示为下述伪代码：

```
row = 0; col = N/2;
magic[row][col] = 1;
for (i=2; i<=N; ++i) {
    if (上一行、下一列有空) 设置上一行、下一列为当前位置;
    else 设置当前列的下一行为当前位置;
    将i 放入当前位置;
}
```

其中二维数组magic用来存储 N 阶魔阵，变量row表示当前行，变量col来表示当前列。

这段伪代码中有两个问题需要解决：第一，如何表示当前单元有空；第二，如何实现找新位置时的回绕。第一个问题可以通过对数组元素设置一个特殊的初值（如0）来实现，第二个问题可以通过取模运算来实现。如果当前行的位置不在最后一行，下一行的位置就是当前行加1。如果当前行是最后一行，下一行的位置是0。这正好可以用一个表达式 $(row + 1) \% N$ 来实现。在找上一行时也可以用同样的方法处理。如果当前行不是第0行，上一行为当前行减1。如果当前行为第0行，上一行为第 $N-1$ 行。这个功能可以用表达式 $(row - 1 + N) \% N$ 实现。由此可得到代码清单5-8所示的程序。

代码清单5-8 打印 N 阶魔阵的程序

```
//文件名：5-8.cpp
//打印N阶魔阵
#include <iostream>
using namespace std;

#define MAX 15 //最高为打印15阶魔阵

int main()
{
    int magic[MAX][MAX] = {0}; //将magic每个元素设为0
    int row, col, count, scale;

    //输入阶数scale
    cout << "input scale\n";
    cin >> scale;

    //生成魔阵
    row=0; col = (scale - 1) / 2; magic[row][col] = 1;
    for (count = 2; count <= scale * scale; count++) {
        if (magic[(row - 1 + scale) % scale][(col + 1) % scale] == 0) {
            row = ( row - 1 + scale ) % scale;
```

```

        col = ( col + 1 ) % scale;
    }
    else row = ( row + 1 ) % scale;
    magic[row][col] = count;
}

//输出
for (row=0; row<scale; row++){
    for (col=0; col<scale; col++)    cout << magic[row][col] << '\t';
    cout << endl;
}

return 0;
}

```

5.4 字符串

除了科学计算以外，计算机最主要的用途就是文字处理。在第2章中，我们已经看到了如何保存、表示和处理一个字符，但更多的时候是把一系列字符当作一个单元处理。由一系列字符组成的一个单元称为字符串。字符串常量是用一对双引号括起来、由'\0'作为结束符的一组字符，如在代码清单2-1中看到的"Hello,world"就是一个字符串常量。本节将讨论如何保存一个字符串变量，对字符串有哪些基本的操作，这些操作又是如何实现的。

5.4.1 字符串的存储及初始化

字符串的本质是一系列的有序字符，因此可以用一个字符数组来保存。字符之间的次序由数组的位置来表示。如要将字符串"Hello,world"保存在一个数组中，这个数组的长度至少为12个字符。我们可以用下列语句将"Hello,world"保存在字符数组ch中：

```
char ch[ ] = { 'H', 'e', 'l', 'l', 'o', ',', 'w', 'o', 'r', 'l', 'd', '\0' };

```

系统自动分配一个12个字符的数组，将这些字符存放进去。在定义数组时也可以指定长度，但此时要记住数组的长度是字符个数加1。

对字符串赋初值，C++还提供了另外两种简单的方式：

```
char ch[ ] = {"Hello,world"};

```

或

```
char ch [ ] = "Hello,world";

```

这两种方法是等价。系统都会自动分配一个12个字符的数组，把这些字符依次放进去，最后插入'\0'。

不包含任何字符的字符串称为空字符串。空字符串并不是不占空间，而是占用了1个字节的空间，这个字节中存储了一个'\0'。

'a'和"a"是不一样的。事实上，这两者有着本质的区别。前者是一个字符常量，在内存占1

个字节，里面存放着字符a的ASCII值，而后者是一个字符串，它占2个字节的空間；第一个字节存放了字母a的ASCII值，而第二个字节存放了'\0'。

5.4.2 字符串的输入/输出

字符串的输入/输出有下面3种方法。

- 逐个字符的输入/输出，这种做法和普通的数组操作一样。
- 将整个字符串一次性地用cin和cout输入或输出。
- 通过cin的成员函数getline输入。

如果定义了一个字符数组ch，要输入一个字符串放在ch中可直接用

```
cin >> ch;
```

要输出ch的内容可直接用

```
cout << ch;
```

与其他类型一样，在用cin输入时是以空格、回车或Tab键作为结束符的。在用cin输入时，要注意输入的字符串的长度不能超过数组的长度。如果超过数组长度，可能会出现一些无法预知的错误。因此，在用cin直接输入字符串时，最好在输出的提示信息中告知允许的最长字符串长度。

由于cin输入时是以空格、回车和Tab键作为结束符，因此当一个字符串中真正包含一个空格时将无法输入，此时可用cin的成员函数getline实现。getline函数的格式为：

```
cin.getline(字符数组, 数组长度, 结束标记);
```

它从终端接受一个包含任意字符的字符串，直到遇到了指定的结束标记或到达了数组长度减1（因为字符串的最后一个字符必须是'\0'，必须为'\0'预留空间）。结束标记也可以不指定，此时默认回车为结束标记。例如，ch1和ch2都是长度为80的字符数组，执行语句

```
cin.getline(ch1, 80, '.');  
cin.getline(ch2, 80);
```

如果对应的输入为aaa bbb ccc.ddd eee fff ggg✓，则ch1的值为aaa bbb ccc，ch2的值为ddd eee fff ggg。

5.4.3 字符串处理函数

字符串的操作主要有复制、拼接、比较等。因为字符串不是系统的内置类型，所以不能用系统内置的运算符来操作。例如，要把字符串s1赋给s2，不能直接用s2=s1。因为s1和s2都是数组，数组名表示的是这个数组在内存中的起始地址。与其他类型的数组一样，数组之间的赋值必须是对应的数组元素之间的赋值。同样，也不能直接对两个字符串进行比较，如s1>s2或s1==s2。为方便字符串的操作，C++的函数库中提供了一些用来处理字符串的函数。这些函数在库cstring中。要用这些函数，必须包含头文件cstring。

cstring包含的主要函数如表5-2所示。

表5-2 主要的字符串处理函数

函 数	作 用
strcpy(dst, src)	将字符从src复制到dst。函数的返回值是dst的地址
strncpy(dst, src, n)	至多从src复制n个字符到dst。函数的返回值是dst的地址
strcat(dst, src)	将src拼接到dst后。函数的返回值是dst的地址
strncat(dst, src, n)	从src至多取n个字符拼接到dst后。函数的返回值是dst的地址
strlen(s)	返回s的长度
strcmp(s1, s2)	比较s1和s2。如果s1>s2返回值为正数，s1=s1返回值为0，s1<s2返回值为负数
strncmp(s1, s2, n)	与strcmp类似，但至多比较n个字符
strchr(s, ch)	返回一个指向s中第一次出现ch的地址
strrchr(s, ch)	返回一个指向s中最后一次出现ch的地址
strstr(s1, s2)	返回一个指向s1中第一次出现s2的地址

使用strcpy和strcat函数时必须注意dst应该是一个字符数组，而且该字符数组必须足够大，能容纳被复制或被拼接后的字符串。如果dst不够大，在复制或拼接过程中会出现dst数组的下标越界，程序会出现不可预知的错误。这种情况称为内存溢出。

C++中字符串的比较规则与其他语言中的规则相同，即对两个字符串从左到右逐个字符进行比较（按ASCII值的大小），直到出现不同的字符或遇到'\0'为止。若全部字符都相同，则认为两个字符串相等；若出现不同的字符，则以该字符的比较结果作为字符串的比较结果。若一个字符串遇到了'\0'另一个字符串还没有结束，则认为没有结束的字符串大。例如，"abc"小于"bcd"，"aa"小于"aaa"，"xyz"等于"xyz"。

5.4.4 字符串的应用

例5.9 输入一行文字，统计有多少个单词。单词和单词之间用空格分开。

这个问题可以这样考虑：单词的数目可以由空格的数目得到（连续若干个空格作为一个空格，一行开头的空格不统计在内）。我们可以设置一个计数器num表示单词个数，开始时num=0。从头到尾扫描字符串。当发现当前字符为非空格，而当前字符以前的字符是空格，则表示找到了一个新的单词，num加1。当整个字符串扫描结束后，num的值就是单词数。按照这个思路实现的程序如代码清单5-9所示。

代码清单5-9 统计单词数的程序

```
//文件名：5-9.cpp
//统计一段文字中的单词个数
#include <iostream>
using namespace std;

int main()
{
    char sentence[80], prev = ' '; //prev 表示当前字符的前一字符
    int i, num = 0;
    cin.getline(sentence, 80);
```

```
for (i = 0; sentence[i] != '\0'; ++i) {
    if (prev == ' ' && sentence[i] != ' ') ++num;
    prev = sentence[i];
}

cout << "单词个数为: " << num << endl;

return 0;
}
```

小结

本章介绍了数组的概念及应用。数组通常用来存储具有同一数据类型并且按顺序排列的一系列数据。数组中的每一个值称作元素，通常用下标值表示它在数组中的位置。在C++语言中，所有数组的下标都是从0开始的。数组中的元素用数组名后加用方括号括起来的下标来引用。数组的下标可以是任意的计算结果能自动转换成整型数的表达式，包括整型、字符型或者枚举型。

当定义一个数组时，必须定义数组的大小，而且它必须是常量。如果程序执行时，数组元素的个数会发生变化，则在编写程序时，要考虑为数组分配足够的空间，分配的空间与在程序中需要的最大的空间一致。

在存储器中，数组元素通常是连续存储的。数组第一个元素的地址称为基地址。数据元素在数组中的位置用偏移量表示。

可以定义多维数组。在C++语言中，可以把多维数组想象成数组的数组。第一个下标值表示在最外层的数组中选择一个元素，而第二个下标值表示在相应的数组中再选择元素，依次类推。最常用的多维数组是二维数组，即每个元素是一个一维数组的一维数组。

字符串可以看成是一组有序的字符。当程序中要存储一个字符串变量时，可以定义一个字符数组。每个字符串必须以'\0'结束，因此，字符数组的元素个数要比字符串中的字符多一个。

习题

简答题

1. 数组的两个特有性质是什么？
2. 写出以下的数组变量的定义。
 - a. 一个含有100个浮点型数据的名为realArray的数组。
 - b. 一个含有16个布尔型数据的名为inUse的数组。
 - c. 一个含有1000个字符串，每个字符串的最大长度为20的名为lines的数组。
3. 用for循环实现下述整型数组的初始化操作。

squares

0	1	4	9	16	25	36	49	64	81	100
0	1	2	3	4	5	6	7	8	9	10

4. 在C++中如何确定保存某一数组需要多少字节?
5. 存储一个有20个字符的字符串的数组需要多少字节?
6. 编写一个C++程序实现以下功能: 计算存储一个含有Nelements个double型数据的数组需要多少字节。
7. 什么是数组的配置长度和有效长度?
8. 什么是多维数组?
9. 假如某数组的基地址是1000, 数组元素类型为int型 (每个元素占4个字节), 请用图表示下面数组中各元素的地址:

```
int rectangular [2][3];
```

程序设计题

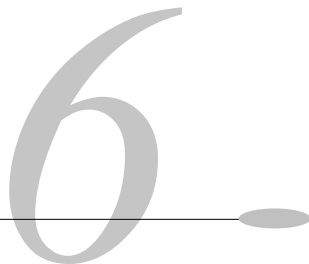
1. 因为每个裁判在评判成绩时都存在主观因素, 所以通常在计算一个运动员的成绩时是按如下方法计算的: 去掉一个最高分, 去掉一个最低分, 然后取剩余的数的平均值。编写一个程序实现读入7个裁判的记分值, 然后去掉一个最高分, 去掉一个最低分, 求剩余元素的平均值。
2. 编写一个程序, 从键盘上输入一篇英文文章。文章的实际长度随输入变化, 最长有10行, 每行80个字符。要求分别统计出其中的字符、数字、空格和其他字符的个数。(提示: 用一个二维字符数组存储文章。)
3. 在公元前3世纪, 古希腊天文学家埃拉托色尼发现了一种找出不大于 n 的所有自然数中的素数的算法, 即埃拉托色尼筛选法。这种算法首先需要按顺序写出 $2 \sim n$ 中所有的数。以 $n=20$ 为例:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

然后把第一个元素画圈, 表示它是素数, 然后依次对后续元素进行如下操作: 如果后面的元素是画圈元素的倍数, 就画 \times , 表示该数不是素数。在执行完第一步后, 会得到素数2, 而所有是2的倍数的数将全被画掉, 因为它们肯定不是素数。接下来, 只需要重复上述操作, 把第一个既没有被圈又没有画 \times 的元素圈起来, 然后把后续的是它的倍数的数全部画 \times 。本例中这次操作将得到素数3, 而所有是3的倍数的数都被去掉。依次类推, 最后数组中所有的元素不是画圈就是画 \times 。所有被圈起来的元素均是素数, 而所有画 \times 的元素均是合数。编写一个程序实现埃拉托色尼筛选法, 筛选范围是 $2 \sim 1000$ 。

第6章

过程封装——函数



函数是程序设计语言中最重要的部分，是模块化设计的主要工具。每一个C++程序都是由一组函数组成的，其中必须有一个名为main的函数，它是程序执行的入口。main函数调用其他函数共同完成任务。

从第2章开始我们就已经接触到了函数。我们用过随机函数rand、求平方根函数sqrt、字符串操作函数strcpy等。函数将一组完成某一特定功能的语句封装起来，作为一个程序的“零件”，为它取一个名字，称为函数名。当程序需要执行这组语句时，只要用它的函数名调用它即可。利用这些“零件”，可以方便地构建功能更强的“零件”。这样可以使程序的主流程变得更短、更简单，逻辑更清晰。当一个功能要被执行多次时，如果没有函数，完成这个功能的语句段就要在程序中反复出现，而有了函数，这个语句段只要出现一次。在程序中每次要执行这一功能就可以调用这个函数。如果没有函数，随着程序的大小和复杂性的增加，程序就会变得越来越难调试。

我们可以将函数想象成数学中的函数。只要给它一组自变量，它就会计算出函数值。自变量称为参数，是程序给被调用函数的输入。函数值称为返回值，是函数输出给调用程序的值。参数相当于是函数的输入，返回值相当于是函数的输出。如果改变输入的参数，函数就能返回不同的值。函数表达式对应于一段语句，它反映了如何从参数得到返回值的过程。函数的参数可以是常数、变量或表达式。例如，求 $\sin x$ 的值就可以写成一个函数，它的参数是一个角度，它的返回值是该角度对应的正弦值。若参数为 90° ，返回值就是1.0。

可以将函数看成上下各有一小孔的黑盒子。在上面的孔中塞入参数，下面的孔中就会流出一个返回值。函数的用户只需要知道什么样的参数应该得到什么样的返回值，而不用去管如何从参数得到返回值。

为了方便程序员，C++提供了许多标准函数，根据它们的用途分别放在不同的库中，如iostream库包含的是与输入/输出有关的函数，cmath库包含的是与数学计算有关的函数。除了这些标准函数以外，用户还可以自己设计函数。在本章中，将会介绍如何自己编写一个函数以及如何使用函数。

6.1 自己编写一个函数

自己写一个函数称为函数定义。一旦定义了一个函数，该函数就可以在程序中反复调用这个

函数。函数定义要说明两个问题，即函数的输入/输出是什么以及该函数如何实现预定的功能。第一个问题由函数头解决，第二个问题由函数体解决。

函数定义的一般形式如下：

```
类型名 函数名(形式参数表)
{
    变量定义部分
    语句部分
}
```

其中第1行是函数头，后3行是函数体。在函数头中，类型名指出函数的返回值的类型。函数也可以没有返回值，这种函数通常被称为过程，此时，返回类型用void表示。函数名是函数的唯一标识。函数名的命名规则与变量名相同。一般变量名用一个名词或名词短语表示，而函数名一般用一个动词短语表示函数的功能。如果函数名是由多个单词组成的，一般每个单词的首字母要大写。例如，将大写字母转成小写字母的函数可以命名为ConvertUpperToLower。形式参数表指出函数有几个参数以及每个参数的类型。

函数体与main函数的函数体一样，由变量定义和语句两个部分组成。变量定义部分定义了语句部分需要用到的变量，语句部分由完成该功能的一组语句组成。

6.1.1 return 语句

当函数的返回类型不是void时，函数必须返回一个值。函数值的返回可以用return语句实现。return语句的格式如下：

```
return 表达式;
或
return(表达式);
```

遇到return语句表示函数执行结束，将表达式的值作为返回值。如果函数没有返回值，则return语句可以省略表达式，仅表示函数执行结束。

return语句后的表达式的结果类型应与函数的返回值的类型一致，或能通过自动类型转换转换成返回值的类型。

6.1.2 函数示例

例6.1 无参数、无返回值的函数示例：编写一个函数打印下面的由5行组成的三角形。

```
 *
***
*****
*****
*****
```

这个函数不需要任何输入，也没有任何计算结果要告诉调用程序，因此不需要参数也不需要返回值。每次函数调用的结果是在屏幕上显示上述一个三角形。函数的实现如下：

```
void PrintStar()
{
    cout << "    *\n";
    cout << "   ***\n";
    cout << "  *****\n";
    cout << " *****\n";
    cout << "*****\n";
}
```

例6.2 有参数、无返回值的函数示例：编写一个函数打印一个由 n 行组成的类似于上例的三角形。

当程序需要打印一个由 n 行组成的三角形时，可以调用此函数，并将 n 作为参数传给它。当 n 等于5时，打印出上例的三角形。因此函数需要一个整型参数来表示行数 n ，但不需要返回值。

在例6.1中，函数体非常简单，直接用5个cout输出5行。在本例中，由于在编写函数时行数 n 并不确定，无法直接用 n 个cout，但可以用一个重复 n 次的循环来实现。在每个循环周期中，打印出一行。那么，如何打印出一行呢？再观察一下每一行的组成。在三角形中，每一行都由两部分组成：前面的连续空格和后面的连续*号。每一行*的个数与行号有关：第1行有1个*，第2行有3个*，第3行有5个*，依次类推，第 i 行有 $2 \times i - 1$ 个*。每一行的前置空格数也与行号有关。第 n 行没有空格，第 $n-1$ 行有1个空格，……，第1行有 $n-1$ 个空格。因此第 i 行有 $n-i$ 个空格。根据上述思路可以得到打印由 n 行组成的三角形的函数如下：

```
void PrintStar(int numOfLine)
{
    int i, j;
    for (i = 1; i <= numOfLine; ++i)
    {
        cout << endl;
        for (j = 1; j <= numOfLine - i; ++j) cout << ' ';
        for (j = 1; j <= 2 * i - 1; ++j) cout << "*";
    }
    cout << endl;
}
```

例6.3 有参数、有返回值的函数示例：计算 $n!$ 。

这个函数需要一个整型参数 n ，函数根据不同的 n 计算并返回 $n!$ 的值，因此它需要一个整型或长整型的返回值。具体的实现如下：

```
int p(int n)
{
    int s = 1, i;

    if (n < 0) return (0);
    for (i = 1; i <= n; ++i) s *= i;

    return(s);
}
```

6.2 函数的使用

6.2.1 函数原型的声明

编写了一个函数后，程序的其他部分就能通过调用这个函数完成相应的功能。但编译器如何知道函数的调用是否正确呢？在C++中，所有的函数在使用前必须被声明。函数声明类似于2.3节的变量定义。变量定义告诉编译器变量的名字和它包含的值的类型，当程序用到此变量时，编译器就会检查变量的用法是否正确。函数声明也类似，只是更详细一些，它告诉编译器对函数的正确使用方法，以便编译器检查程序中的函数调用是否正确。在C++中，函数的声明说明以下几项内容。

- 函数的名字。
- 参数的个数和类型，大多数情况下还包括参数的名字。
- 函数返回值的类型。

上述内容在C++中被称为函数原型，这些信息正好是函数头的内容，因此，C++中函数原型的声明具有下列格式：

返回类型 函数名(形式参数说明)；

返回类型指出了函数结果的类型，函数名指出函数的名字，形式参数说明指出传给这个函数的参数个数和类型，也可以加上参数的名字。每个形式参数之间用逗号分开。例如：

```
char func(int, float, double);
```

说明函数func有3个参数，第一个参数的类型是int，第二个参数的类型是float，第三个参数的类型是double，返回值的类型为char。

例如，在cmath库中的sqrt函数的原型为

```
double sqrt(double);
```

这个函数原型说明函数sqrt有一个double类型参数，返回一个double类型的值。

函数原型只指定了函数调用者和函数之间传进传出的值的类型。从原型中看不出定义函数的真正语句，甚至看不出函数要干什么。要使用sqrt，必须知道返回值是它参数的平方根。然而，C++编译器不需要这个信息，它只需要知道sqrt取一个double类型的值，并返回一个double类型的值。当程序调用此函数时，传给它一个double型的数，编译器认为正确；如果传给它一个字符串，编译器就会报错。同样，当程序将函数的执行结果当作double类型处理时，编译器认为正确；而当作其他类型处理时，编译器会报错。函数的确切作用是以函数名的形式和相关的文档告诉使用该函数的程序员的，至于函数是如何完成指定的功能，使用此函数的程序员无需知道，就如我们在使用计算机时无需知道计算机是如何完成指定的任务的一样，这将大大简化程序员的工作。

在函数原型声明中，每个形式参数类型后面还可以跟一个参数名，该名字可标识特定的形式参数的作用，这可以为程序员提供一些额外的信息。虽然形式参数的名字为使用该函数的程序员

提供了重要的信息，但对程序无任何实质性的影响。例如，在`cmath`中函数`sin`被声明为

```
double sin(double);
```

它仅指出了参数的类型。需要用这个函数的程序员可能希望看到这个函数原型被写为

```
double sin(double angleInRadians);
```

以这种形式写的函数原型提供了一些有用的新信息：`sin`函数有一个`double`类型的参数，该参数是以弧度表示的一个角度。在自己定义函数时，应该为参数指定名字，并在相关的介绍函数操作的注释中说明这些名字。

系统标准库中的函数原型的声明包含在相关的头文件中，这就是为什么在用到系统标准函数时要在源文件头上包含此函数所属的库的头文件。用户自己定义的函数必须在源文件头上声明。

6.2.2 函数的调用

一旦在源文件头上声明了某个函数，就可以在程序的其他部分执行此函数，这称为函数的调用。函数调用形式如下：

函数名 (实际参数表)

其中，实际参数表是本次函数执行的输入。实际参数和形式参数是一一对应的，它们的个数、排列次序、类型要完全相同。实际参数和形式参数的对应过程称为参数的传递。在C++中参数的传递方式有两种：值传递和引用传递。本节先介绍值传递机制。引用传递等到介绍指针类型和引用类型（第7章）时再介绍。

在值传递中，实际参数可以是常量、变量、表达式，甚至是另一个函数调用。在函数调用时，依次将实际参数的值赋给对应的形式参数。赋完值后，形式参数和实际参数就没有任何关系了。在函数中对形式参数的任何变化对实际参数都没有影响。

在C++的函数调用中，如果有多个实际参数要传给函数，而每个实际参数都是表达式。那么，在参数传递前首先要计算出这些表达式的值，然后再将实际参数的值赋给形式参数。但是，C++并没有规定这些实际参数表达式的计算次序。实际的计算次序由具体的编译器决定。因此当实际参数表达式有副作用时，要特别谨慎。例如，`f(++x, x)`，当`x=1`时，如果实际参数表的计算次序是从左到右，则传给`f`的两个参数都是2；如果实际参数表的计算次序是从右到左，则传给`f`的第一个参数为2，第二个参数为1。因此，应避免写出与实际参数计算次序有关的调用。对于上面的问题，可以采用如下的显式方式来解决：

```
++x; f(x, x)
```

或

```
y=x; ++x; f(x, y);
```

函数调用可以出现在以下3种情况中。

- 作为语句。直接在函数调用后加一个分号，形成一个表达式语句。这通常用于无返回值的函数，即过程，如`PrintStar();`。

- 作为表达式的一部分。例如，要计算 $5!+4!+7!$ ，并将结果存于变量 x 。因为已定义了一个计算阶乘的函数 p ，此时可直接用 $x=p(5)+p(4)+p(7)$ 。
- 作为函数的参数，如 $\text{PrintStar}(p(5)+p(4)+p(7))$ 。

6.2.3 将函数与主程序放在一起

函数只是组成程序的一个零件，其本身不能构成一个完整的程序。每个完整的程序都必须有一个名字为`main`的函数。当程序执行时就调用此函数。为了测试某函数是否正确，必须为它写一个`main`函数，并在`main`函数中调用它。例如，要测试有参数的`PrintStar`，可以写一个完整的程序，如代码清单6-1所示。

代码清单6-1 函数的使用

```
//文件名: 6-1.cpp
//该程序说明了多函数程序的组成及函数的使用
#include <iostream>
using namespace std;

void PrintStar(int); //函数原型声明

//主程序
int main()
{   int n;

    cout << "请输入要打印的行数: ";
    cin >> n;

    printstar(n); //函数调用, n为实际参数

    return 0;
}

//函数: PrintStar
//用法: PrintStar(numOfLine)
//作用: 在屏幕上显示一个由numOfLine行组成的三角形
void PrintStar(int numOfLine)
{   int i, j;
    for (i = 1; i <= numOfLine; ++i){
        cout << endl;
        for (j = 1; j <= numOfLine - i; ++j) cout << ' ';
        for (j = 1; j <= 2 * i - 1; ++j) cout << "**";
    }
    cout << endl;
}
```

在一个多函数的程序中，一般每个函数前都应该有一段注释，说明该函数的名字、用法和用途。这样，可以使阅读程序的人将每个函数作为一个单元来理解，更容易理解程序整体的功能。例如，对于代码清单6-1中的程序，我们知道函数`PrintStar(n)`可以打印出一个由 n 行组成的三

角形，因此很容易理解整个程序的功能就是根据用户输入的n打印相应的三角形，而不必关心该三角形是如何打印出来的。

6.2.4 函数调用过程

当在main函数或其他函数中发生函数调用时，系统依次执行如下过程。

- 在main函数或调用函数中计算每个实际参数值。
- 将实际参数赋给对应的形式参数。在赋值的过程中如果类型不一致，则完成自动类型转换。
- 依次执行被调函数的函数体的每条语句，直到遇见return语句或函数体结束。
- 计算return后面的表达式的值，如果表达式的值与函数的返回类型不一致，则完成类型的转换。
- 在函数调用的地方用返回值替代，继续调用函数的执行。

每个函数都可能有形式参数，在函数体内也可能定义一些变量。因此，每次调用一个函数时，都会为这些变量分配内存空间。当发生函数调用时，系统会为该函数分配一块内存空间，称为一个帧。函数的形式参数和在函数体内定义的变量都存放在这块空间上。这些变量仅在定义它们的函数中有意义，因此被称为局部变量。在程序执行时，只有当前函数的帧是有效的。

当函数执行结束时，系统回收分配给该函数的帧，因此所有的局部变量也就都消失了。下面通过代码清单6-2中的程序说明函数的执行过程，以及执行过程中内存的分配情况。

代码清单6-2 函数调用实例

```
//文件名: 6-2.cpp
//函数调用示例
int p( int );
int max( int a, int b );

int main()
{   int x, y;
    cin >> x >> y;
    cout << max(x, y);

    return 0 ;
}

int p( int n )
{   int s =1, i;
    if (n < 0) return(0);
    for (i=1;i<=n; ++i)    s*=i;
    return(s);
}

int max( int a, int b )
{   int n1, n2;
    n1 = p(a); n2 = p(b);
```



```
    return (n1 > n2 ? n1 : n2);  
}
```

在代码清单6-2所示的程序执行时，首先执行的是main函数。系统为main函数分配一个帧。main函数定义了两个变量x和y。如果用户输入的x为2，y为3，则main函数的帧如下所示：

main	x (2)	y (3)
------	-------	-------

当执行到cout << max(x, y)时，main函数调用了函数max，并将x和y作为max函数的实际参数。此时，系统为max函数分配了一个帧，这个帧中存储了4个变量：2个形式参数，2个是函数体内定义的变量。在参数传递时，将x的值传给了a，y的值传给了b。因此，当max函数开始执行时a的值为2，b的值为3。此时内存的情况如下所示，可以看成max函数的帧“覆盖”了main函数的帧，程序能够访问的变量就是max函数中的那些变量，而main函数的那些变量暂时不能访问。

main	x (2)		y (3)	
max	a (2)	b (3)	n1	n2

此时系统能操作的是max函数的帧。max函数的第一条语句是n1 = p(a);，此时又调用了函数p，将a作为实际参数。系统为函数p分配了一个帧，这个帧中存放了3个变量：形式参数n和函数体中定义的s和i。在参数传递时，将a的值传给n。因此，当函数p执行时，n的值为2。此时内存的情况如下所示：

main	x (2)		y (3)	
max	a (2)	b (3)	n1	n2
p	n (2)		s	i

依次执行函数p的语句，直到遇到return语句。此时，s的值为2。当遇到return语句时，函数执行结束，所属的帧被回收，回到max函数，并将return语句中的表达式的值替换函数调用，得到n1的值为2。此时，内存的情况如下所示：

main	x (2)		y (3)	
max	a (2)	b (3)	n1 (2)	n2

继续执行max函数，它的第二个语句是n2 = p(b);，此时又调用了函数p，将b作为实际参数。系统为函数p分配了一个帧。在参数传递时将b的值传给n。因此，当函数p执行时，n的值为3。此时内存的情况如下所示：

main	x (2)		y (3)	
max	a (2)	b (3)	n1 (2)	n2
p	n (3)		s	i

执行函数p的语句，直到遇到return语句。此时，s的值为6。当遇到return语句时，函数执行结束，所属的帧被回收，回到max函数，并将return语句中的表达式的值替换函数调用，得到n2的值为6。此时内存的情况如下所示：

main	x (2)		y (3)	
max	a (2)	b (3)	n1 (2)	n2 (6)

继续执行max函数，此时遇到的语句是return (n1 > n2 ? n1: n2)。max函数执行结束，并将n1和n2中大的一个返回给main函数，回收max的帧。回到main函数，用6取代max函数调用，输出6。此时内存的情况如下所示：

main	x (2)	y (3)
------	-------	-------

继续执行main函数，遇到return语句。main函数执行结束，回收main函数的帧，整个程序执行结束。

6.3 数组作为函数的参数

数组元素可以是表达式的一部分，因此当然可以作为函数的实际参数。它的用法与普通的变量作为实际参数一样，即用值传递的方式。

数组名也可以作为实际参数，此时形式参数和实际参数都是数组名（或指针名，见第7章），这时会有一些有趣的现象。

例6.4 设计一函数，计算10个学生的考试平均成绩。

这个函数的输入是10位学生的考试成绩。在第5章中已知对于一组同类型的数据可以用一个数组来存储，因此，此函数的参数为一个整型数组，函数的返回值是一个整型数，表示平均成绩。函数的实现和使用如代码清单6-3所示。

代码清单6-3 计算10位学生的平均成绩的函数及使用

```
//文件名：6-3.cpp
//计算10位学生的平均成绩的函数及使用
#include <iostream>
using namespace std;

int average(int array[10]); //函数原型声明

int main()
{   int i, score[10];

    cout << "请输入10个成绩: " << endl;
    for ( i = 0; i < 10; i++)   cin >> score[i];

    cout << "平均成绩是: " << average(score) << endl;
```

```
        return 0;
    }

    int average(int array[10])
    {
        int i, sum = 0;

        for (i = 0; i < 10; ++i)    sum += array[i];

        return sum / 10;
    }
```

程序的运行结果如下：

请输入10个成绩：

90 70 60 80 65 89 77 98 60 88

平均成绩是：77

同普通的参数传递一样，形式参数和实际参数的类型要一致。因此，当形式参数是数组时，实际参数也应该是数组，而且形参数组和实参数组的类型也要一致。

但数组传递和普通的变量传递有本质的区别。在第5章中我们已经知道，数组名代表的是数组在内存中的起始地址。按照值传递的机制，当参数传递时将实际参数的值赋给形式参数，即将作为实际参数的数组的起始地址赋给形式参数的数组名，这样形式参数和实际参数的数组具有同样的起始地址，也就是说形式参数和实际参数的数组事实上是一个数组。在函数中对形式参数的数组的任何修改实际上是对实际参数的修改。所以数组传递的本质是仅传递了数组的起始地址，并不是将作为实际参数的数组中的每个元素对应赋给形式参数的数组。那么在被调函数中如何知道作为实际参数的数组的大小呢？没有任何获取途径，数组的大小必须作为一个独立的参数传递。总结一下，传递一个数组需要两个参数：数组名和数组大小。数组名给出数组的起始地址，数组大小给出该数组的元素个数。

由于数组传递本质上是首地址的传递，真正的元素个数是作为另一个参数传递的，因此形式参数中数组的大小是无意义的，通常可省略。如代码清单6-3中的函数原型声明和函数定义中的函数头 `int average(int array[10])` 中的10可以省略，简写为 `int average(int array[])`。

二维数组可以看成是由一维数组组成的数组。当二维数组作为参数传递时，第一维的个数可以省略，第二维的个数必须指定。

数组参数实际上传递的是地址这一特性非常有用，它可以将被调函数内部对形式参数的修改变化传到调用函数的实际参数。

例6.5 编写一个程序，实现下面的功能：读入一串整型数据，直到输入一个特定值为止；把这些整型数据逆序排列；输出经过重新排列后的数据。要求每个功能用一个函数来实现。

除了main函数外，这个程序需要3个函数，即ReadIntegerArray（读入一串整型数）、ReverseIntegerArray（经这组数据逆序排列）和PrintIntegerArray（输出数组），分别完成这3个功能。有了这3个函数，main函数非常容易实现：依次调用3个函数。因此，首先要做的工作就是确定3个函数的原型。

ReadIntegerArray从键盘接收一个整型数组的数据，它需要告诉main函数输入了几个元素以及这些元素的值。输入的元素个数可以通过函数的返回值实现，但输入的数组元素的值如何告诉main函数呢？幸运的是，数组传递的特性告诉我们对形式参数的任何修改都是对实际参数的修改。因此可以在main函数中定义一个整型数组，将此数组传给ReadIntegerArray函数。在ReadIntegerArray函数中，将输入的数据放入作为形式参数的数组中。为了使这个函数更通用和可靠，还需要两个信息：实际数组的规模和输入结束标记。据此可得ReadIntegerArray函数的原型为int ReadIntegerArray(int array[],int max, int flag)。返回值是输入的数组元素的个数，形式参数array是存放输入元素的数组，max是作为实际参数的数组的规模，flag是输入结束标记。

ReverseIntegerArray函数将作为参数传入的数组中的元素逆序排列，这很容易实现。同样因为数组传递的特性，在函数内部对形式参数数组的元素逆序排列也反映给了实际参数。因此ReverseIntegerArray函数的原型可设计为void ReverseIntegerArray(int array[],int size)。

PrintIntegerArray函数最简单，只要把要打印的数组传给它就可以了。因此它的原型为void PrintIntegerArray(int array[], int size)。

按照上述思路得到的程序如代码清单6-4所示。

代码清单6-4 整型数据逆序输出的程序

```
//文件名: 6-4
//读入一串整型数据，将其逆序排列并输出排列后的数据
#include <iostream>
using namespace std;

#define MAX 10

int ReadIntegerArray(int array[ ], int max, int flag );
void ReverseIntegerArray(int array[ ], int size);
void PrintIntegerArray(int array[ ], int size);

int main()
{   int IntegerArray[MAX], flag, CurrentSize;

    cout << "请输入结束标记: ";
    cin >> flag;

    CurrentSize = ReadIntegerArray(IntegerArray, MAX, flag );
    ReverseIntegerArray(IntegerArray, CurrentSize);
    PrintIntegerArray(IntegerArray, CurrentSize);
    return 0;
}

//函数: ReadIntegerArray
//作用: 接收用户的输入，存入数组array, max是array的大小, flag是输入结束标记。
//当输入数据个数达到最大长度或输入了flag时结束
```

```

int ReadIntegerArray(int array[ ], int max, int flag )
{
    int size = 0;

    cout << "请输入数组元素, 以" << flag << "结束: ";
    while (size <= max) {
        cin >> array[size];
        if (array[size] == flag) break; else ++size;
    }

    return size;
}

//函数: ReverseIntegerArray
//作用: 将array中的元素按逆序存放, size为元素个数
void ReverseIntegerArray(int array[ ], int size)
{
    int i, tmp;

    for (i=0; i < size / 2; i++){
        tmp = array[i];
        array[i] = array[size-i-1];
        array[size-i-1] = tmp;
    }
}

//函数: PrintIntegerArray
//作用: 将array中的元素显示在屏幕上。size是array中元素的个数
void PrintIntegerArray(int array[ ], int size)
{
    int i;

    if (size == 0) return;
    cout << "逆序是: " << endl;
    for (i=0; i<size; ++i) cout << array[i] << '\t';
    cout << endl;
}

```

程序的运行结果如下:

```

请输入结束标记: 0
请输入数组元素, 以0结束: 1 2 3 4 5 6 7 0
逆序是:
7    6    5    4    3    2    1

```

6.4 带默认值的函数

对于某些函数, 程序往往会用一些固定的值去调用它。例如, 对于下面的以某种数制输出整型数的函数print:

```
void print(int value, int base);
```

在大多数情况下都是以十进制输出, 因此base的值总是为10。这样, 以十进制输出时总要带第二个参数就显得很累赘。

在C++中, 允许在定义或声明函数时为函数的某个参数指定默认值。当调用函数时没有为它

指定实际参数时，系统自动将默认值赋给形式参数。例如，可以将print函数声明为

```
void print(int value, int base=10);
```

这样，如果要调用该函数以十进制输出变量x，则可省略第二个参数：

```
print(x);
```

编译器会根据默认值把这个函数调用改为

```
print(x, 10);
```

如果要以二进制输出变量x的值，则可用

```
print(x, 2);
```

指定函数的默认值简化了函数调用的书写，但是在使用此功能时应注意以下几点。

(1) 在设计函数原型时，应把有默认值的参数放在参数表的右边。在函数调用时，编译器依次把实际参数赋给形式参数。没有得到实际参数的形式参数取它的默认值。例如，函数声明

```
void f(int a, int b=1, int c=2, int d=3);
```

是合法的。当调用此函数时，至少必须给它一个实际参数，最多给它4个实际参数。该函数可以有4种调用形式：

```
f(0);           //形式参数的值为a=0,b=1,c=2,d=3
f(0,0);         //形式参数的值为a=0,b=0,c=2,d=3
f(0,0,0);        //形式参数的值为a=0,b=0,c=0,d=3
f(0,0,0,0);      //形式参数的值为a=0,b=0,c=0,d=0
```

下面的带默认值参数的函数声明是错误的：

```
void f(int a, int b=1, int c, int d=2);
```

(2) 对参数默认值的指定只有在函数声明处有意义。因为参数的默认值是提供给函数的调用者使用的，而编译器是根据函数原型声明确定函数调用是否合法的，所以在函数定义时指定默认值是没有意义的，除非该函数定义还充当了函数声明的作用。

(3) 在不同的源文件中，可以对函数的参数指定不同的默认值。在同一源文件中对每个函数的声明只能对它的每一个参数指定一个默认值。例如，对于上面的print函数，如果在某一个功能模块中输出的大多是十进制数，那么在此功能对应的源文件中可以指定base的默认值为10；如果在另一个功能模块中经常要以二进制输出，那么在此功能模块对应的源文件中可以指定默认值是2。

6.5 内联函数

从软件工程的角度讲，把程序实现为一组函数很有好处。它不但可以使程序的总体结构比较清晰，而且可以提高程序的正确性、可读性和可维护性，但函数的调用会增加执行时的开销。如果函数比较大，执行时间比较长，相比之下，调用时的额外开销可以忽略。但如果函数本身比较小，执行时间也很短，则调用时的额外开销就显得很可观，使用函数似乎得不偿失。为解决问题，C++提供了内联函数的功能。

内联函数通过将函数代码复制到调用处来避免函数调用。其代价是会产生函数代码的多个副本，并分别插入到每一个调用该函数的位置上，从而加长生成的目标代码。因此，内联函数一般都是些比较短小的函数。

要把一个函数定义为内联函数，只要在函数定义中的返回类型前加一个关键字`inline`就可以了。但在使用内联函数时必须注意内联函数在被调用之前必须进行完整的定义，否则编译器无法知道应该插入什么代码。因此，内联函数通常写在主函数之前。

内联函数一般用来代替传统的C语言中的带参数的宏，但它比带参数的宏更安全。

例6.6 利用内联函数打印1~100的整数的平方表和立方表。

在这个程序中要经常计算某个数的平方和立方，因此可把它们设计成两个函数。由于这两个函数相当简单，所以可把它们设计成内联函数。具体程序见代码清单6-5。

代码清单6-5 用内联函数打印平方表和立方表的程序

```
//文件名: 6-5.cpp
//用内联函数打印平方表和立方表
#include <iostream>
using namespace std;

inline square(int x) {return x*x;}
inline cube(int x) {return x*x*x;}

int main()
{   int i;

    cout << "x" << '\t' << "x*x" << '\t' << "x*x*x" << endl;
    for (i=1; i<=100; ++i) cout << i << '\t' << square(i) << '\t' << cube(i) << endl;

    return 0;
}
```

6.6 重载函数

在传统的C语言中，不允许出现同名函数。当写一组功能类似的函数时，必须给它们取不同的函数名。例如，某个程序要求找出一组数据中的最大值，这组数据最多有5个数据，我们必须写4个函数：求2个值中的最大值、求3个值中的最大值、求4个值中的最大值和求5个值中的最大值。我们必须为这4个函数取4个不同的函数名，如`max2`、`max3`、`max4`和`max5`。这样对函数的用户非常不方便。在调用函数之前先要看一看有几个参数，再决定调用哪个函数。

为解决此问题，C++提供了一个称为重载函数的功能。允许参数个数不同、参数类型不同或两者兼而有之的两个以上的函数取相同的函数名。两个或两个以上的函数共用一个函数名称为函数重载，这一组函数称为重载函数。

有了重载函数，求一组数据中的最大值的函数就可以取同样的函数名，如`max`。这对函数的用户非常方便，他不用去数这组数据的个数，再去找相应的函数，而只需要知道找最大值就是函

数max，只要把这组数据传给max就可以了。这组函数的定义和使用可以写成代码清单6-6。

代码清单6-6 重载函数实例

```
//文件名: 6-6.cpp
//重载函数示例
#include <iostream>
using namespace std;

int max(int a1, int a2);
int max(int a1, int a2, int a3);
int max(int a1, int a2, int a3, int a4);
int max(int a1, int a2, int a3, int a4, int a5);

int main()
{   cout << "max(3,5) is " << max( 3,5) << endl;
    cout << "max(3,5,4) is " << max( 3,5,4) << endl;
    cout << "max(3,5,7,9) is " << max( 3,5,7,9) << endl;
    cout << "max(3,5,2,4,6) is " << max( 3,5,2,4,6) << endl;

    return 0;
}

int max(int a1, int a2) {return a1>a2 ? a1 : a2;}

int max(int a1, int a2, int a3)
{   int tmp;
    if (a1 > a2) tmp = a1; else tmp = a2;
    if (a3 > tmp) tmp = a3;

    return tmp;
}

int max(int a1, int a2, int a3, int a4)
{   int tmp;

    if (a1 > a2) tmp = a1; else tmp = a2;
    if (a3 > tmp) tmp = a3;
    if (a4 > tmp) tmp = a4;

    return tmp;
}

int max(int a1, int a2, int a3, int a4, int a5)
{   int tmp;
    if (a1 > a2) tmp = a1; else tmp = a2;
    if (a3 > tmp) tmp = a3;
    if (a4 > tmp) tmp = a4;
    if (a5 > tmp) tmp = a5;

    return tmp;
}
```

max函数是一组参数个数不同的重载函数。重载函数也可以参数个数相同但参数类型不同。例如，求两个数的最大值。这两个数可以是整型数，也可以是实型数。我们同样可以写出两个重载函数：int max(int, int)和double max(double, double)。

重载函数为函数的用户提供了方便，但给编译器带来了更多的麻烦。编译器必须确定某一次函数调用到底是调用了哪一个具体的函数。这个过程称为绑定（又称为联编或捆绑）。

C++对重载函数的绑定是在编译阶段由编译器根据实际参数和形式参数的匹配情况来决定的。编译器首先会为这一组重载函数中的每个函数取一个不同的内部名字。当发生函数调用时，编译器根据实际参数和形式参数的匹配情况确定具体调用的是哪个函数，用这个函数的内部函数名取代重载的函数名。例如，对代码清单6-6中的程序，函数调用`max(3,5)`调用的是第一个`max`函数，函数调用`max(3,5,7,9)`调用的是第三个`max`函数。

6.7 函数模板

重载函数方便了函数的用户。对于一组功能类似的函数，用户只需要记一个函数名而不是一组函数名；但对于函数的提供者来说，却没有减少任何工作量，他还是要写多个函数。函数模板是使函数的提供者受惠的一项功能。

重载函数通常用于一组功能类似的函数，每个函数可以有自己的处理逻辑。重载函数要求这组函数的参数个数或参数类型是不同的，以便编译器能确定到底调用的是哪个函数。如果一组重载函数仅仅是参数的类型不一样，程序的逻辑完全一样，那么这一组重载函数可以写成一个函数模板。从而将写一组函数的工作减少到了写一个函数模板。

所谓的函数模板就是实现类型的参数化（泛型化），即把函数中某些形式参数的类型定义成参数，称为模板参数。在函数调用时，编译器根据实际参数的类型确定模板参数的值，生成不同的模板函数。

所有的函数模板的定义都以关键字`template`开头，之后是用尖括号括起来的模板的形式参数。每个形式参数之前都有关键字`class`。形式参数后面就是标准的函数定义，只是函数中的某些参数或局部变量的类型不再是系统的标准类型或用户自定义的类型，而是模板的形式参数。

有了函数模板就可以减少函数开发者的工作量。例如，6.6节提到的求两个数的最大值，这两个数可以是整型数，也可以是实型数。这两个函数的程序逻辑完全相同，因此可以用下面的函数模板来解决：

```
template <class T>
T max(T a, T b)
{   return  a>b ? a : b; }
```

这个函数模板适合解决任意类型的两个值求最大值的问题，其应用见代码清单6-7。

代码清单6-7 函数模板的定义及使用

```
//文件名：6-7.cpp
//函数模板的定义及使用
#include <iostream>

using namespace std;
template <class T>
T max(T a, T b)
```

```
{    return  a>b ? a : b; }

int main()
{    cout << "max(3,5) = " << max(3,5) <<endl;
    cout << "max(3.3, 2.5) = " << max(3.3, 2.5) <<endl;
    cout << "max('d', 'r') = " << max('d', 'r') <<endl;

    return 0;
}
```

函数模板的使用和普通函数完全一样。当发生函数调用时，编译器根据实际参数的类型确定模板参数的值，将模板参数的值取代函数模板中的模板的形式参数，形成一个真正可执行的函数。该过程称为模板的实例化。实例化形成的函数称为模板函数。例如，代码清单6-7中的调用`max(3,5)`生成了模板函数

```
int max(int a, int b)

{    return  a>b ? a : b; }
```

函数调用`max('d', 'r')`生成了模板函数

```
char max(char a, char b)

{return  a>b ? a : b; }
```

代码清单6-7中的一个函数模板`max`相当于写了3个重载函数`max`。

在使用函数模板时必须注意，每个模板的形式参数都要在函数的参数表中至少出现一次，这样编译器才能通过函数调用确定模板参数的值。

函数模板本身可以同其他函数模板或普通函数重载。当发生重载时，编译器根据匹配过程确定调用的是哪个函数：首先检查是否有完全匹配的函数，然后再检查是否有匹配的函数模板。

6.8 变量的作用域

6.2节已经介绍过函数调用过程的机制。无论何时一个函数被调用，它定义的变量都创建一个称为栈的独立内存区域。调用一个函数相当于从栈中分配一个新的帧并且放在代表其他活动函数的帧的上面。从函数返回相当于拿掉它的帧，并继续在调用者中执行。

在函数内部定义的变量，包括形式参数，仅仅存活于一个帧中，只能在函数内部引用，因此被称为局部变量。当函数返回时，对应的帧被回收，帧中的变量就完全消失了，其他函数当然无法再引用。

关于C++的局部变量，有下面两点需要说明。

(1) 与其他的一些程序设计语言不同，在C++中主函数`main`中定义的变量也是局部的，只有在主函数中才能使用。

(2) 在一个程序块中也可以定义变量，这些变量值在本程序块中有效。例如：

```
int main()
{    int a, b;
```

```

...
{   int c;
    c = a + b;
    ...
} //c不再有效
...
}

```

然而在C++中，变量定义也可以出现在所有函数定义之外。以这种方式定义的变量称为全局变量。定义本身看起来跟局部变量一样，只是它们是出现在所有函数的外面，通常是在文件开始处。例如，下面的代码段中，变量g是全局变量，变量i是局部变量。

```

int g;
void MyProcedure()
{   int i
    ...
}

```

局部变量i仅在函数MyProcedure内有效，而全局变量g可以被用在该源文件中随后定义的任何函数中。变量可以被使用的程序部分称为它的作用域。这样，局部变量的作用域是定义它的函数或程序块，全局变量的作用域则是源文件中定义它的位置后的其余部分。

与局部变量不同，全局变量以另一种方式保持在内存中，它不受函数调用的影响。全局变量被保存在一个始终有效的独立的内存区域，永远都不会被包含局部变量的帧覆盖。程序中每个函数都可以看到这块独立区域上的这些变量。而且，当函数返回时，这些变量值不会丢失。在函数执行过程中，全局变量保持相同的值直到赋予新值为止。

全局变量可以增加函数间的联系渠道。由于同一源文件中的所有函数都能引用全局变量，因此，当一个函数改变了全局变量的值时，其他的函数都能看见，相当于各个函数之间有了直接的信息传输渠道。但全局变量也破坏了函数的独立性，使得同样的函数调用会得到不同的返回值。全局变量的用途将在第9章详细介绍。

当全局变量和局部变量同名时，在局部变量的作用域中全局变量被屏蔽。如果确实需要在局部变量的作用域中使用全局变量，可以用作用域运算符“::”。例如，下列程序中定义了两个PI，一个是全局的PI，一个是main函数中定义的局部的PI。在main函数中引用全局的PI，可以用::PI。

```

const double PI = 3.14159265358979;
int main()
{   const float PI = static_cast< float >( ::PI );
    cout << setprecision(20)
         << " Local float value of PI = "<< PI
         << "\nGlobal double value of PI = "<< ::PI << endl;
    return 0;
}

```

6.9 变量的存储类别

从变量的作用域来分，变量可分为局部变量和全局变量。在计算机中，内存被分为不同的区

域。按变量在计算机内的存储位置来分,变量又可以分为自动变量(auto)、静态变量(static)、寄存器变量(register)和外部变量(extern)。变量的存储位置决定了变量的存储类别。变量的存储类别也决定了它的生存期。在C++中,完整的变量定义格式如下:

存储类别 数据类型 变量名表;

下面依次对这几种存储类别进行介绍。

6.9.1 自动变量

函数中的局部变量、形式参数或程序块中定义的变量,如不专门声明为其他存储类型,都是自动变量。因此在函数内部以下两个定义是等价的:

```
auto int a, b;
```

```
int a, b;
```

自动变量存储在内存中称为栈的区域中。当函数被调用时,系统会为该函数在栈中分配一块区域,称为帧,所有的自动变量都存放在这块空间中。当函数执行结束时,系统回收该帧,自动变量的值就消失了。当再次进入该函数时,系统重新分配一个帧。由于这类变量是在函数调用时自动分配空间,调用结束后自动回收空间,因此被称为自动变量。在程序执行过程中,栈空间被反复地使用。

6.9.2 静态变量

如果在程序执行过程中某些变量自始至终都必须存在,如全局变量,那么这些变量被存储在内存的全局变量区。如果想限制这些变量只在某一个范围内才能使用,可以用static来限定。

1. 静态的全局变量

如果在一个源文件的头上定义了一个全局变量,则该源文件中的所有函数都能使用该全局变量。不仅如此,事实上该程序中的其他源文件中的函数也能使用该全局变量。但在一个结构良好的程序中,一般不希望多个源文件共享某一个全局变量。要做到这一点,可以使用静态的全局变量。

若在定义全局变量时,加上关键字static,例如:

```
static int x;
```

则表示该全局变量是当前源文件私有的。尽管在程序执行过程中,该变量始终存在,但只有本源文件中的函数可以引用它,其他源文件中的函数不能引用它。

2. 静态的局部变量

静态变量的一种有趣的应用是用在局部变量中。一般的局部变量都是自动变量,在函数执行时生成,函数结束时消亡。但是,如果把一个局部变量定义为static,该变量就不再存放在该函数对应的帧中,而是存放在全局变量区。当函数执行结束时,该变量不会消亡,在下次函数调用时,继续使用空间中的值。这样就能把上一次函数调用中的某些信息带到下一次函数调用中。

考察代码清单6-8中的程序的输出结果。

代码清单6-8 静态局部变量的应用

```
//文件名: 6-8.cpp
//静态局部变量的使用
#include<iostream>
using namespace std;

int f(int a);

int main()
{   int a=2,i;
    for (i=0; i<3; ++i)   cout << f(a);

    return 0;
}

int f(int a)
{   int b=0;
    static int c=3;

    b=b+1;    c=c+1;

    return(a+b+c);
}
```

当第一次调用函数f时，b的初值为0，c的初值为3。函数执行结束时，b的值为1，c的值为4，函数返回7。变量a和b自动消失，但c依然存在。等第二次调用f时，b的初值为0，但c的值为4（上次调用执行的结果）。第二次调用结束时，b的值为1，c的值为5，函数返回8。第三次调用时，c的值为5，函数返回9。

在静态变量使用时必须注意以下几点。

- 未被程序员初始化的静态变量都由系统初始化为0。
- 局部静态变量的初值是编译时赋的。当运行时重复调用函数时，由于没有重新分配空间，因此也不做初始化。
- 虽然局部静态变量在函数调用结束后仍然存在，但其他函数不能引用它。

6.9.3 寄存器变量

一般情况下，变量的值都存储在内存中。当程序用到某一变量时，由控制器发出指令将该变量的值从内存读入CPU的寄存器进行处理，处理结束后再存入内存。由于内存的存取也是需要消耗时间的，如果某个变量被频繁使用，存取时间会非常可观。C++提供了一种解决方案，就是直接把变量的值存储在CPU的寄存器中，用于代替自动变量或形式参数。这些变量称为寄存器变量。

寄存器变量的定义是用关键字register。例如，在某个函数内定义整型变量x：

```
register int x;
```

则表示x不是存储在内存中，而是存放在寄存器中。在使用寄存器变量时必须注意只有局部自动变量或形式参数才能定义为寄存器变量，全局变量和静态的局部变量是不能存储在寄存器中的。由于各个系统的寄存器个数都不相同，程序员并不知道可以定义多少个寄存器类型的变量，因此寄存器类型的声明只是表达了程序员的一种意向，如果系统中无合适的寄存器可用，编译器就把

它设为自动变量。

6.9.4 外部变量

外部变量一定是全局变量。全局变量的作用域是从变量定义处到文件结束。如果在定义点以前的函数或另一源文件中的函数也要使用该全局变量,则在引用之前应该对此全局变量用关键字extern进行外部变量声明,否则编译器会认为使用了一个没有定义过的变量。例如,代码清单6-9所示的程序在编译时将会产生一个“变量x没有定义”的错误。这是因为全局变量x定义在main函数的后面,在main函数输出变量x时没见到x的定义。

代码清单6-9 全局变量的错误用法

```
//文件名: 6-9.cpp
//全局变量的错误用法
#include <iostream>
using namespace std;
void f();

int main()
{   f();
    cout << "in main(): x= " << x << endl;
    return 0;
}

int x;

void f()
{
    cout << "in f(): x= " << x << endl;
}
```

要解决此问题可以在main函数中增加一个外部变量声明:

```
int main()
{   extern int x;

    f();
    cout << "in main(): x= " << x << endl;   return 0;
}
```

外部变量声明extern int x;告诉编译器:这里使用了一个或许还没有见到过的变量,该变量将在别处定义。

外部变量声明最主要的用途是使各源文件之间共享全局变量。一个C++程序通常由许多源文件组成,如果在一个源文件A中想引用另一个源文件B定义的全局变量,如x,该怎么办?如果不加任何说明,在源文件A编译时会出错,因为源文件A引用了一个没有定义的变量x。但如果在源文件A中也定义了全局变量x,在程序连接时又会出错。因为系统发现全局变量x有两个定义,也就是出现了同名变量。

解决这个问题的方法是：在一个源文件（如源文件B）中定义全局变量x，而在源文件A中声明用到一个在别处定义过的全局变量x，就像程序要用到一个函数时必须声明函数是一样的。这样在源文件A编译时，由于声明过x，编译就不会出错。在连接时，系统会将源文件B中的x扩展到源文件A中。源文件A中的x就称为外部变量。外部变量声明的格式如下：

```
extern 类型名 变量名;
```

其中，类型名可省略。例如，可以将代码清单6-9中的两个函数分别存放于两个源文件。main函数存放在源文件file1.cpp中，f函数存放在源文件file2.cpp中。file2.cpp中定义了一个全局变量x，file1.cpp为了引用此变量，必须在自己的源文件中将x声明为外部变量，如代码清单6-10所示。

代码清单6-10 外部变量应用实例

```
//file1.cpp

#include <iostream>
using namespace std;

void f();
extern x;    //外部变量的声明

int main()
{
    f();
    cout << "in main(): x= " << x << endl;
    return 0;
}

//file2.cpp

#include <iostream>
using namespace std;

int x;        //全局变量的定义

void f()
{
    cout << "in f(): x= " << x << endl;
}
```

这样的全局变量的使用应当非常谨慎，因为在执行一个函数时可能会修改全局变量的值，而这个全局变量又会影响到另一个源文件中的其他函数的执行。通常我们希望某一源文件中的全局变量只供该文件中的函数共享，此时可用static把此全局变量声明为私有的。这样其他源文件就不可以用extern来引用它了。如果在代码清单6-10中，把file2.cpp中的x定义为static int x；那么程序连接时会报错“找不到外部符号int x”。

全局变量可以通过static声明为某一源文件私有的，函数也可以。如果在函数定义时前面加上static，那么这个函数只有被本源文件中的函数调用，而不能被其他源文件中的函数调用。这样，每个源文件的开发者可以定义一些自己专用的工具函数。

注意，在使用外部变量时，用术语外部变量声明而不是外部变量定义。变量的定义和变量的声明是不一样的。变量的定义是根据说明的数据类型为变量准备相应的空间，而变量的声明只是说明该变量应如何使用，并不为它分配空间，就如函数原型声明一样。

6.10 递归函数

递归程序设计是程序设计中的一个重要的概念，它的用途非常广泛。递归程序设计的主要实现手段是递归函数。

6.10.1 递归函数的基本概念

某些问题在规模较小时很容易解决，而规模较大时却很复杂。但某些大规模的问题可以分解成同样形式的若干小规模的问题，小规模问题的解可以形成大规模问题的解。

例如，假设你在为一家慈善机构工作。你的工作是筹集1 000 000元。如果你能找到一个人愿意出这1 000 000元，你的工作就很简单了。但是，你不大可能有这么慷慨大方的百万富翁朋友。所以，你可能需要募集很多小笔的捐款来凑齐1 000 000元。如果平均每笔捐款额为100元，你可以用另一种方法完成这项工作：找10 000个捐赠人让他们每个人捐100元。但是，你又不大可能找到10 000个捐赠人，那你该怎么办呢？

当你面对的任务超过你个人的能力所及时，完成这项任务的办法就是把部分工作交给别人做。如果你能找到10个志愿者，你可以请他们每个人筹集100 000元就够了。

筹资100 000元比筹资1 000 000简单得多，但也绝非易事。这些志愿者又怎么解决这个问题呢？他们也可以运用相同的策略把部分筹募工作交给别人。如果他们每个人都找10个筹募志愿者，那么这些筹募志愿者每人就只需筹集10 000元。这种代理的过程可以层层深入下去，直到筹款人可以一次募集到所有他们需要的捐款。因为平均每笔捐款额为100元，志愿者完全可能找到一个人愿意捐献这么多善款，从而无需找更多人来代理筹款的工作了。

可以将上述筹款策略用如下伪代码来表示：

```
void CollectContributions(int n)
{
    if (n<=100) 从一个捐赠人处收集资金；

    else { 找10个志愿者；
           让每个志愿者收集n/10元；
           把所有志愿者收集的资金相加；
        }
}
```

上述伪代码中最重要的是

让每个志愿者收集n/10元；

这一行。这个问题就是原问题的再现，只是规模较原问题小一些。这两个任务的基本特征都是一样的：募捐n元，只是n值的大小不同。再者，由于要解决的问题实质上是一样的，你可以通过同样的方法来解决，即调用原函数来解决。因此，上述伪代码中的这一行最终可以被下列行取代：

CollectContributions (n/10)

需要着重指出的是，如果捐款数额大于100元，函数CollectContributions最后会调用自己。

调用自身的函数称为递归函数，这种解决问题的方法称为递归程序设计。作为解决问题的方法，递归技术是一种非常有力的工具，利用递归不但可以使书写复杂度降低，而且使程序看上去更加美观。

几乎所有的递归函数都有同样的基本结构。典型的递归函数的函数体符合如下范例：

```
if (递归终止的条件测试) return (不需要递归计算的简单解决方案);
else return (包括调用同一函数的递归解决方案);
```

在设计一个递归函数时必须注意以下两点。

- 必须有递归终止的条件。
- 必须有一个与递归终止条件相关的形式参数，并且在递归调用中，该参数有规律地递增或递减（越来越接近递归终止条件）。

数学上的很多函数都有很自然的递归解，如阶乘函数 $n!$ 。按照定义， $n!=1\times 2\times 3\times \cdots \times (n-1)\times n$ ，而 $1\times 2\times 3\times \cdots \times (n-1)$ 正好是 $(n-1)!$ 。因此 $n!$ 可写为 $n!=(n-1)!\times n$ 。在数学上，定义 $0!$ 等于1，这就是递归终止条件。综上所述， $n!$ 可用如下递归公式表示：

$$n! = \begin{cases} 1 & (n=0) \\ (n-1)! \times n & (n>0) \end{cases}$$

其中 $n=0$ 就是递归终止条件，而每次递归调用时， n 的规模都比原来小1，都朝着 $n=0$ 变化。根据定义，很容易写出计算 $n!$ 的函数：

```
long p(int n)
{   if (n == 0) return 1;
    else return n * p(n-1);
}
```

斐波那契数列是计算机学科中一个重要的数列，它的值如下：

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

观察这个数列可以发现：第一个数是0，第二个数是1，后面的每一个数都是它以前的两项的和。因此斐波那契数列可写成如下的递归形式：

$$F(n) = \begin{cases} 0 & (n=0) \\ 1 & (n=1) \\ F(n-1) + F(n-2) & (n>1) \end{cases}$$

该函数的实现可以直接翻译上述递归公式：

```
int Finonacci (int n)
{   if (n == 0) return 0;
    else if (n == 1) return 1;
        else return (Finonacci (n-1) + Finonacci (n-2));
}
```

从上面两个实例可以看出递归函数逻辑清晰,程序简单,整体感强。但要理解递归函数是如何执行的却不太容易。一个递归函数的执行由两个阶段组成:递归调用和回溯。例如,要计算 $n!$ 必须调用此函数本身以计算 $(n-1)!$,这个过程称为递归调用。有了 $(n-1)!$ 的值就可以计算 $n!$,这个过程称为回溯。因此,用递归函数 p 求 $4!$ 的过程如图6-1所示。

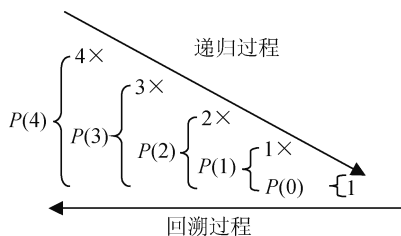


图6-1 递归函数的执行过程

6.10.2 递归函数的应用

例6.7 汉诺塔 (Hanoi) 问题。这是一个古老的问题。相传印度教的天神梵天在创造地球这一世界时,建了一座神庙。神庙里有三根宝石柱子,柱子由一个铜座支撑。梵天将64个直径大小不一的金盘子按照从大到小的次序依次套放在第一根柱子上,形成一座金塔,即所谓的汉诺塔。天神让庙里的僧侣们将第一根柱子上的64个盘子借助第二根柱子全部移到第三根柱子上。同时定下3条规则。

- (1) 每次只能移动一个盘子。
- (2) 盘子只能在三根柱子间移动,不能放在他处。
- (3) 在移动过程中,三根柱子上的盘子必须始终保持大盘在下,小盘在上的状态。

天神说:“当这64个盘子全部移到第三根柱子上之后,世界末日就要到了。”这就是著名的汉诺塔问题。

汉诺塔问题是一个典型的只能用递归(而不能用其他方法)解决的问题。任何天才都不可能直接写出移动盘子的每一个具体步骤。根据递归的思想,我们可以将64个盘子的汉诺塔问题转换为求解63个盘子的汉诺塔问题。如果63个盘子的问题能解决,则可先将上面的63个盘子从第一根柱子移到第二根柱子,再将最后一个盘子直接移到第三根柱子,最后再将63个盘子从第二根柱子移到第三根柱子,这样就解决了64个盘子的问题。依次类推,63个盘子的问题可以转化为62个盘子的问题,62个盘子的问题可以转化为61个盘子的问题,直到1个盘子的问题。如果只有一个盘子,就可将它直接从第一根柱子移到第三根柱子,这就是递归终止的条件。根据上述思路,可得汉诺塔问题的递归程序如下:

```
void Hanoi(int n, char start, char finish, char temp)
{   if (n==1)   cout << start << "->" << finish << '\t';
    else {
        Hanoi(n-1, start, temp, finish);
        cout << start << "->" << finish << '\t';
        Hanoi(n-1, temp, finish, start);
    }
```

```
    }
}
```

当 $n=3$ 时，`Hanoi(3, '1', '3', '2')`的输出如下：

```
1->3  1->2  3->2  1->3  2->1  2->3  1->3
```

例6.8 设计一个打印非负整型数的函数。

利用递归简化程序代码的一个实例就是数的打印。假设我们要以十进制形式打印一个非负的整数 N ，而我们并没有一个可用的数字输出函数。但是，我们有打印一个字符的函数。利用这个函数，就可打印任意整型数。例如，打印数1369，可以首先打印'1'，然后'3'，然后'6'，然后'9'。问题是得到第一位数不太方便：给定一个数 n ，我们需要一个循环判断 n 的第一个数字。相反最后一个数字可以从 $n \% 10$ 立即得到（如果 n 小于10的话，就是 n 本身）。

递归提供了一个很好的解决方案。为打印整型数1369，我们打印出136，然后打印最后一个数字9。正如我们已经提到过的，使用 $\%$ 打印最后一个数字很容易。打印去掉最后一位数以后的整型数也很容易，因为它和打印 n 是同样的问题。因此，用递归调用就可以实现了。

代码清单6-11所示的代码实现了这个打印程序。如果 n 比10小，只有一个数字 $n \% 10$ 被打印；否则，除最后一个数字外的其他所有数字都被递归地打印，然后打印最后一个数字。

代码清单6-11 打印一个十进制整数的函数定义及使用

```
//文件名: 6-11.cpp
//打印一个十进制整数
#include <iostream>
using namespace std;

void printInt(int); //输出一个整型数

int main()
{   int num;

    cout << "请输入一个整型数: " << endl;
    cin >> num;

    printInt(num);
    cout << endl;

    return 0;
}

//作用: 以十进制打印非负整数num
//用法: printInt(1234)
void printInt(int num)
{   if (num < 10)        //递归终止条件
    {   cout << static_cast<char>(num+'0');
    }
    else {
        printInt(num/10);
        cout << static_cast<char>(num%10 + '0');
    }
}
```

注意我们是如何到达递归终止条件（ n 是一位整数）的，因为每次递归调用都少了一位数字，所以，最后总会到达只剩一位数的情况。

为使我们的打印函数更有用，我们把它扩展到能打印二进制、八进制、十进制和十六进制的非负整数。改后的程序如代码清单6-12所示。在代码清单6-12所示的程序中可以看到，该函数和代码清单6-11所示的程序有几个区别。首先，该函数要打印各种数制的整数，因此，必须提供一个表示数制的参数`int base`，它的值可以是2、8、10或16。当实际参数的值为2时，表示以二进制输出。当实际参数为8时，表示以八进制输出。第二个区别是取整数的最后一位。当以十进制输出时，取最后一位可以用表达式`n % 10`表示。当以数制`base`输出时，取最后一位可以用表达式`n % base`得到。最后一个区别是打印某一位的值。在十进制中，每一位的值是从0到9。如果输出的值保存在变量`n`中，则输出时可以用`n + '0'`将这位数字转换成相应的字符，然后用字符输出函数输出。当输出是用二进制、八进制表示时，可以用同样的方法。但当输出是以十六进制表示时，稍有麻烦。因为表示十六进制数的16个符号的ASCII编码是不连续的，不能简单地用一个算术表达式表示。在代码清单6-12所示的程序中，将这16个符号存储在一个字符数组`DIGIT`中。输出时，只要取对应的数组元素即可。另一种方法是用一个条件表达式：如果输出的数字为`n`，则条件表达式`(n < 10) ? n + '0' : n - 10 + 'A'`也完成了这个转换。

代码清单6-12 打印二进制、八进制、十进制或十六进制整数的函数定义及使用

```
//文件名: 6-12.cpp
//打印二进制、八进制、十进制和十六进制整数
#include <iostream>
using namespace std;

const char DIGIT[17]="0123456789abcdef";
void printInt(int,int);
int main()
{   int num,base;

    cout << "请输入一个整型数: " << endl;
    cin >> num;
    cout << "请输入要打印的数制: " << endl;
    cin >> base;

    printInt(num, base);
    cout << endl;

    return 0;
}

//作用: 以数制base打印非负整数num
//用法: printInt(1234, 8)
void printInt(int num, int base)
```

```

{   if (num < base) cout << DIGIT[num];
    else {
        printInt(num/base, base);
        cout << DIGIT[num%base];
    }
}

```

例6.9 设计一个函数，打印任意 n 个字符的全排列。

例如，如果给定3个字母构成的字符串"ABC"，那么可形成的全排列为ABC, ACB, BAC, BCA, CAB, CBA。这可以用一个三重循环来实现。但要设计一个输出任意长度的字符串中的所有字母的全排列，不能沿用循环的思想。

怎样写这个能输出全排列的函数ListPermutations呢？如果被限于循环控制结构（如for或while循环）里，要想找到一个通用的解决方案来处理任何长度的字符串的全排列问题是比较困难了。但是，如果用递归思想来考虑这个问题，就能得到一个相对简单的解决方案。

同一般的递归程序一样，解题过程的难点在于怎样把复杂的原问题分解成简单一些的同类问题。在这个例子中，要为一个字符串找出所有可能的排列，需要搞清楚的是：为一个稍短一些的字符串生成所有排列对解题有没有帮助以及有何帮助。

为了对这个问题有更清楚的认识，我们来探讨一个具体的情况。假设要为一个含5个字母的字符串"ABCDE"生成所有可能的排列，可以假定任何稍短的字符串都能自动生成所有排列。也就是说，假设递归调用是有效的，并且都已经成功返回结果了。现在，关键性的问题出现了：怎样利用稍短字符串的排列解决5字母字符串的排列问题。

解决这个排列问题的关键是看清这样一个事实：5字母字符串"ABCDE"的全排列是由下列字符串组成。

- 字母A后跟着"BCDE"的所有可能排列。
- 字母B后跟着"ACDE"的所有可能排列。
- 字母C后跟着"ABDE"的所有可能排列。
- 字母D后跟着"ABCE"的所有可能排列。
- 字母E后跟着"ABCD"的所有可能排列。

把这个问题一般化来看，也就是说，要显示长度为 n 的字符串的全排列，可以依次把 n 个字母中的每个字母置于列首，其后跟着剩下 $n-1$ 个字母的全排列。

这种解决方法的主要问题在于递归子问题和原问题的形式并不完全相同，而这正违反了递归解题的一个必要条件。原问题要求你显示一个字符串的所有排列，其中没有任何递归信息。而子问题要求你显示一个字母后面接着剩余字母的所有排列。为了使递归方案能够工作，可以对此问题做一个小小的修改，使得递归子问题与原问题完全相同。

在这种情况下，最好的办法就是定义一个新的过程PermuteWithFixedPrefix。在保证前 k 个字母保持不变的条件下，让它产生一个字符串的所有排列。当 $k=0$ 时，所有字母都可以变动，这也就是原问题；随着 k 值的增加，问题就相应变得简单；当 k 值等于字符串长度，就没有字符可以相互变动位置，可以原样显示这个字符串。PermuteWithFixedPrefix过程可以用以下的伪代

码表示:

```
void PermuteWithFixedPrefix(char str[], int k)
{
    if (k等于字符串的长度) 显示该字符串
    else {
        对从k到字符串结束前的每个位置i上的字符{
            交换i和k的字符;
            生成前k+1个字符固定的情况下所有的全排列;
            重新交换i和k的字符, 恢复原先的字符串;
        }
    }
}
```

把这段伪代码转化成C++语言程序代码并不是很困难, 尤其是在你定义了一个交换两个字母的函数后就更简单了。PermuteWithFixedPrefix函数的C++语言代码如下:

```
void PermuteWithFixedPrefix(char str[ ], int k)
{
    int i;

    if ( k == strlen(str) ) cout << str << endl;
    else for (i=k; i<strlen(str); ++i){
        swap(str, k, i);
        PermuteWithFixedPrefix(str, k+1);
        swap(str, k, i);
    }
}
```

这段程序实际上并未完成, 首先, 你需要定义函数swap来完成逐步细化, 该函数交换数组中的两个元素, 其定义如下:

```
void swap(char str[], int k, int i)
{
    int tmp;

    tmp = str[k];
    str[k] = str[i];
    str[i] = tmp;
}
```

由于采用的是数组传递, 形式参数数组和实际参数数组是同一数组, 在函数中将形式参数数组的第k个元素和第i个元素做了交换, 在实际参数数组中的这两个元素也做了交换。

更重要的问题在于你还需要定义函数ListPermutations。函数PermuteWithFixedPrefix完成了所有必需的工作, 但它并不是用户期待的原型。在调用PermuteWithFixedPrefix时, 需要传递一个整数和一个字符串。大多数客户只希望考虑排列一个字符串中的字符, 这里面不包括任何整数。因此, 从用户的角度考虑的话, 应该像下面这样定义函数ListPermutations:

```
void ListPermutations(char str[])
{
    PermuteWithFixedPrefix(str, 0);
}
```

```
}
```

当用递归方案解决问题时，往往需要定义一个原型与原问题稍有不同递归函数。就像在PermuteWithFixedPrefix中一样，新的函数需要带有一些额外的、用于递归控制的参数，但这些参数对于整个问题来说并不是必需的。用户所用的函数调用了这个递归函数，传递这些额外参数的初值，就像在ListPermutations中那样。像ListPermutations这样的函数的目的是为一个更通用的函数提供额外参数，这样的函数叫作包装函数。

6.11 基于递归的算法

求解问题首先需要设计算法，但设计算法是一件非常困难的事。在实际工作中，经常采用的算法设计技术有迭代法、枚举法、贪婪法、回溯法、分治法和动态规划。在第4章中已经介绍了枚举法和贪婪法，在本节中将介绍基于递归的回溯法、分治法和动态规划。

6.11.1 回溯法

回溯法也称试探法。该方法首先暂时放弃问题规模大小的限制，从最小规模开始将问题的候选解按某种顺序逐一枚举和检验。当发现候选解不可能是解时，就选择下一候选解。如果当前候选解除了不满足规模要求外，满足其他所有要求时，则继续扩大当前候选解的规模，继续试探。如果当前的候选解满足包括问题规模在内的所有要求时，该候选解就是问题的一个解。寻找下一候选解的过程称为回溯。扩大当前候选解的规模，并继续试探的过程称为向前试探。八皇后问题和分书问题都是典型的用回溯法解决的问题。

例6.10 八皇后问题。在一个 8×8 的棋盘上放8个皇后，使8个皇后中没有两个以上的皇后会出现在同一行、同一列或同一对角线上。（这样根据国际象棋的规则不会引起冲突。）

求解过程从空配置开始，在第1列到第 m 列为合理配置的基础上再配置 $m+1$ 列，直到第8列的配置也是合理时，就找到了一个解。另外在一列上也有8种配置。开始时配置在第1行，以后改变时，顺序选择第2行、第3行、……、第8行。配置到第8行时还找不到一个合理的配置时，就要回溯，去改变前一列的配置。输出八皇后问题所有解的过程可用如下伪代码表示：

```
{  m = 0;  // 从空配置开始
    good = true;  // 配置中的皇后不冲突
    do {  if (good)
        if (m == 8)
        {  输出解;
            重新寻找下一可行的解;
        }
        else  向前试探，扩展至下一列;
    else  回溯，形成下一候选解;
    good = 检查当前候选解的合理性;
    }  while (m != 0);
}
```

在真正编写程序前，我们还要解决两个问题：如何表示一个棋盘，如何测试解是否合理。比较直观的方法是采用一个二维数组，但仔细考查，就会发现，这种表示方法给调整候选解及检查其合理性会带来困难。对于本题来说，我们关心的并不是皇后的具体位置，而是“某一列上的一个皇后是否已经在某行和某条斜线上合理地安置好了”。因为在每一列上恰好放一个皇后，所以引入一个一维数组(设为`col[9]`)，值`col[j]`表示在棋盘第`j`列上的皇后位置。如`col[3]`的值为4，就表示第3列的皇后在第4行。另外，为了使程序在找完了全部解后回溯到最初位置，设定`col[0]`的初值为0。当回溯到第0列时，说明程序已求得全部解（或无解），结束程序执行。

为了检查皇后的位置是否冲突，引入以下3个布尔型的工作数组。

- 数组`a[9]`，`a[A]=true`表示第`A`行上还没有皇后。
- 数组`b[16]`，`b[A]=true`表示第`A`条右高左低斜线上没有皇后，从左上角依次编到右下角(1~15)。
- 数组`c[16]`，`c[A]=true`表示第`A`条左高右低斜线上没有皇后。从左下角依次编到右上角(1~15)。

当第`i`行第`k`列上放置了一个皇后时，`a[i]=false`，`b[k+i-1]=false`，`c[8+k-i] = false`。

初始时，所有的行和斜线上均没有皇后，从第1列的第1行开始配置第1个皇后。在第`k`列的第`col[k]`行的位置放置了一个合理的皇后后，将`a`、`b`和`c`数组中对应第`k`列、第`col[k]`行的位置设置有皇后标志。当从第`k`列回溯到第`k-1`列，并准备调整第`k-1`列上的皇后的位置时，清除数组`a`、`b`和`c`中设置的关于第`k-1`列，`col[k-1]`行有皇后的标志。一个皇后在第`k`列，第`col[k]`行内的配置是否合理，由`a`、`b`和`c`数组对应的位置值来决定。如果三者都为`true`，则是合理的。具体程序见代码清单6-13。

代码清单6-13 求解八皇后问题的程序

```
//文件名: 6-13.cpp
//八皇后问题
#include <iostream>
using namespace std;

void queen_all(int k);
int col[9];
bool a[9], b[17], c[17];

int main()
{
    int j;
    for(j = 0; j <= 8; j++) a[j] = true;
    for(j = 0; j <= 16; j++) b[j] = c[j] = true;
    queen_all(1);

    return 0;
}
//在8×8棋盘的k列上找合理的配置
void queen_all(int k)
{
    int i, j;
    char awn; //存储是否需要继续寻找的标志
```



```

for (i = 1; i <= 9; i++) //依次在1至8行上配置k列的皇后
    if (a[i] && b[k+i-1] && c[8+k-i]) { //可行位置
        col[k] = i;
        a[i] = b[k+i-1] = c[8+k-i] = false; //置对应位置有皇后
        if (k == 8) { // 找到一个可行解
            for (j = 1; j <= 8; j++) cout << j << col[j] << '\t' ;
            cout << endl << "是否需要继续寻找 (Q -- 退出, 其他键继续: ) ";
            cin >> awn;
            if (awn=='Q' || awn=='q') exit(0);
        }
        else queen_all(k+1); //递归至第k+1列
        a[i] = b[k+i-1] = c[8+k-i] = true; //恢复对应位置无皇后
    }
}

```

例6.11 分书问题。有编号为0,1,2,3,4的5本书，准备分给5个人A,B,C,D,E，每个人的阅读兴趣用一个二维数组描述：

```

like[i][j] = true    //i喜欢书j
like[i][j] = false   //i不喜欢书j

```

编写一个程序，输出所有皆大欢喜的分书方案。

解决这个问题首先要解决信息的存储问题。可以用一个二维的布尔型数组like存储用户的兴趣；用一个一维的布尔型数组book记录书是否被分掉，如果book[i] = true表示第i本书尚未被分掉，等于false则表示已被分掉；用一个一维的整型数组take表示某本书分给了某人，take[i] = j表示第j本书分给了第i个人。解题的思路和八皇后问题类似。先给第1个人分书。在给第i-1个人分配了合理的书后，再尝试给第i个人分配书。如果尝试了所有的书，都不适合分给i，也就是说他喜欢的书都已被分配，此时从第i个人回溯到第i-1个人，重新给第i-1个人分配书。为此设计函数trynext(i)，在已为0到i-1个人分书的基础上，为第i个人分配书。trynext(i)依次尝试把书j分给人i。如果第i个人不喜欢第j本书，则尝试下一本书，如果喜欢，并且第j本书尚未分配，则把书j分配给i。如果i喜欢的书都已被分配，则回溯。如果分配成功且i是最后一个人，则方案数加1，输出该方案；否则调用trynext(i+1)为第i+1个人分书。

由于在trynext中要用到like、book、take以及目前找到的方案数n，而trynext是一个递归函数，因此可将这些变量作为全局变量，以免每次函数调用时都要带一大串参数。trynext函数如下：

```

void trynext(int i)
{
    int j, k;
    for (j=0; j<5; ++j) {
        if (like[i][j] && book[j]){           //如果i喜欢j，并且j未被分配
            take[i] = j; book[j] = false;    //j分给i
            if (i == 4) {                     //找到一种新方案，输出此方案
                n++;
                cout << "\n第" << n << "种方案: " << endl;
            }
        }
    }
}

```

```

        cout << "人\t书" << endl;
        for (k=0; k<5; k++) cout << char(k+'A') << '\t' << take[k] << endl;
    }
    else trynext(i+1);    //为下一个人分书
    book[j] = true;      //尝试找下一方案
}
}
}

```

当like矩阵的值为

false	false	true	true	false
true	true	false	false	true
false	true	true	false	true
false	false	false	true	false
false	true	false	false	true

时，调用trynext(0)的输出如下：

第1种方案：

人	书
A	2
B	0
C	1
D	3
E	4

第2种方案：

人	书
A	2
B	0
C	4
D	3
E	1

6.11.2 分治法

分治法也许是使用最广泛的算法设计技术，其基本思想是将一个大问题分成若干个同类的小问题，然后由小问题的解构造出大问题的解。把大问题分成小问题称为“分”，从小问题的解构造大问题的解称为“治”。

分治法通常都是用递归实现的。求出同类小问题的解就是采用递归调用。

例6.12 快速排序。快速排序是分治法的一个典型的示例，下面是求解问题的主要思路。

- 将待排序的数据放入数组a中，从a[low]到a[high]。
- 从待排序的数据中任意选择一个数据作为分段基准，将它放入变量k。
- 将待排序的数据分成两组，一组比k小，放入数组的前一半；一组比k大，放入数组的后一半；将k放入中间位置。
- 对前一半和后一半分别重复用上述方法排序，直到只剩下一个待排序的数据为止。

在快速排序的实现中主要解决下面两个问题。

- 如何选择作为分段基准的元素？不同的选择对不同的排序数据会产生不同的时间效益。常用的有3种方法：第一种是选取第一个元素，第二种是选取第一个、中间一个和最后一个中的中间元素。最后一种是随机选择一个元素作为基准元素。前一种方法程序比较简单，但当待排序数据比较有序或本身就是有序的时间效率很差。但如果待排序数据很乱、很随机，则时间效率较好。后两种方法能较好地适用各种待排序的数据，但程序更复杂。为简单起见，我们选用第一种方法。
- 如何分段？如何分段也有多种方法。最简单的是再开一个同样大小的数组，顺序扫描原数组，如果比基准元素k小，则从新数组的左边开始放，否则从新数组的右边开始放，最后将基准元素放到新数组唯一的空余空间中。这种方法的空间效益较差。如果待排序的元素数量很大的话，浪费的空间是很可观的。在快速排序中通常采用一种很巧妙的方法，该方法只用一个额外的存储单元。

如果有一个函数divide能实现分段的话，快速排序的函数将非常简单。

```
void quicksort(int a[], int low, int high)
{
    int mid;
    if (low >= high) return;    //待分段的元素只有一个或0个，递归终止
    mid = divide(a, low, high); //low作为基准元素，划分数组，返回中间元素的下标
    quicksort(a, low, mid-1);  //排序左一半
    quicksort(a, mid+1, high); //排序右一半
}
```

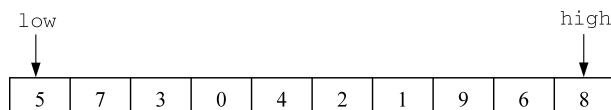
接下去的主要工作就是完成划分。划分工作首先将low中的元素放在一个变量k中，这样low的位置就空出来了。接下去重复下列步骤。

(1) 从右向左开始检查。如果high的值大于k，该位置的值位置正确，high减1，继续往前检查，直到遇到一个小于k的值。

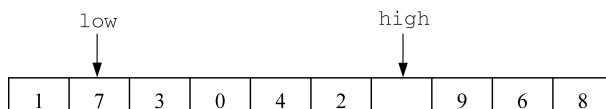
(2) 将小于k的这个值放入low的位置，此时high的位置又空出来了。然后从low位置开始从左向右检查，直到遇到一个大于k的值。

(3) 将low位置的值放入high位置，重复第1步，直到low和high重叠。将k放入此位置。

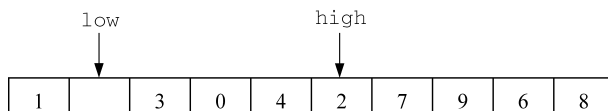
例如，数据5, 7, 3, 0, 4, 2, 1, 9, 6, 8的划分步骤如下所示：



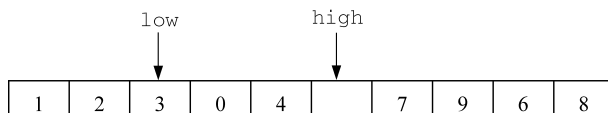
将5放入变量k中。high从右向左进行扫描，遇到比5小的元素停止（此处为元素1）。将此元素放入low中。



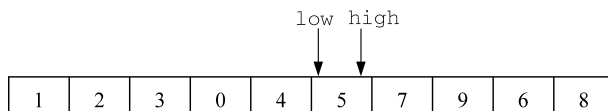
low从左往右扫描，遇到比5大的元素停止。此时遇到的是7，将7放到high中。



high从右到左继续扫描，遇到比5小的停止。此时遇到的是2，将2放到low中。



low从左往右扫描，直到遇到high。将5放入low。



至此，一次划分结束。数据被分成了两半：1, 2, 3, 0, 4和7, 9, 6, 8。5已经在正确的位置上了。

综合上述思想可以得到下面分段的程序：

```
int divide( int a[], int low, int high)
{   int k = a[low];
    do { while (low < high && a[high] >= k) --high;
        if (low < high) { a[low] = a[high]; ++low; }
        while (low < high && a[low] <= k) ++low;
        if (low < high) { a[high] = a[low]; --high; }
    } while (low != high);
    a[low] = k;
    return low;
}
```

例6.13 最长连续子序列和的问题。给定（可能是负的）整数序列 A_1, A_2, A_3, \dots ，寻找（并标识） $\sum_{k=i}^j A_k$ 值为最大的序列。如果所有的整数都是负的，那么最长连续子序列的和是零。

例如，假设输入是-2, 11, -4, 13, -5, 2，那么答案是20，它表示连续子序列包含了第2项到第4项。又如，对于输入1, -3, 4, -2, -1, 6，答案是7，这个子序列包含最后4项。

最长连续子序列和的问题可以用分治法解决。假设输入是4, -3, 5, -2, -1, 2, 6, -2。我们把这个输入划分成两部分，即前4个和后4个。这样最长连续子序列可能出现在下面3种情况中。

- 情况1：最长子序列位于前半部分。
- 情况2：最长子序列位于后半部分。
- 情况3：从前半部开始但在后半部结束。

这3种情况中的最大值就是本问题的解。

在前两种情况中，只需要在前半部分或后半部分找最长连续子序列，这通过递归调用就可解

决。问题是第三种情况如何解决？我们可以从两半部分的边界开始，通过从右到左的扫描来找到左半段的最长序列。类似地，从左到右的扫描找到右半段的最长序列。把这两个子序列组合起来，形成跨越分割边界的最长连续子序列。在上述输入中，通过从右到左扫描得到的左半段的最长序列和是4，包括前半部分的所有元素。从左到右扫描得到的右半段的最长序列和是7，包括-1、2和6。因此从前部分开始到后半部分结束的最长子序列和为4+7=11。

总结一下，用分治法解决最长子序列和问题的算法由4步组成。

- (1) 递归地计算整个位于前半部的最长连续子序列。
- (2) 递归地计算整个位于后半部的最长连续子序列。
- (3) 通过两个连续循环，计算从前半部开始但在后半部结束的最长连续子序列的和。
- (4) 选择上述3个子问题中的最大值，作为整个问题的解。

根据这个算法，可得下面的程序：

```
int maxSum(int a[ ], int left, int right )
{
    int maxLeft, maxRight, center; // maxLeft和maxRight分别为左、右半部的最长子序列和
    int leftSum = 0, rightSum = 0;
    int maxLeftTmp = NEGMAX, maxRightTmp = NEGMAX; // NEGMAX最大负整数

    if (left == right) return a[left] > 0 ? a[left] : 0;
    center = (left + right) / 2;

    maxLeft = maxSum(a, left, center);
    maxRight = maxSum(a, center + 1, right);

    for (int i = center; i >= left; --i){
        leftSum += a[i];
        if (leftSum > maxLeftTmp) maxLeftTmp = leftSum;
    }
    for (i = center + 1; i <= right; ++i){
        rightSum += a[i];
        if (rightSum > maxRightTmp) maxRightTmp = rightSum;
    }

    return max3(maxLeft, maxRight, maxLeftTmp + maxRightTmp);
}
```

6.11.3 动态规划

在实际工作中经常会遇到一个复杂的问题不能简单地分成几个子问题，而是会分解出一系列子问题。如果用分治法的话，会使得递归调用的次数呈指数增长，如斐波那契数列的计算。

因为斐波那契数列是递归定义的，第*i*个斐波那契数是前两个斐波那契数之和。写一个计算 F_N 的递归过程似乎是很自然的事情。这个递归过程如6.10.1节所示，可以工作，但是有一个严重的问题。在相对比较快的机器上，计算 F_{40} 也需要较长的时间，事实上，这个基本的计算只要39次加法，用这么多的时间是荒唐的！

根本的问题就是这个递归过程执行了冗余的运算。为计算 $\text{fib}(n)$ ，递归地计算 $\text{fib}(n-1)$ 。当这个递归调用返回时，通过使用另一个递归调用计算 $\text{fib}(n-2)$ 。但是在计算 $\text{fib}(n-1)$ 的过程中已经计算了 $\text{fib}(n-2)$ ，因此对 $\text{fib}(n-2)$ 的调用是浪费的、冗余的计算。事实上，我们对 $\text{fib}(n-2)$ 调用了两次而不是一次。

其实调用两个函数不只是让程序的运行时间加倍，事实上比这还要糟：每个对 `fib(n-1)` 的调用和每个对 `fib(n-2)` 的调用都会产生一个对 `fib(n-3)` 的调用，这样其实对 `fib(n-3)` 调用了3次。事实上，它还会更加糟糕：每个对 `fib(n-2)` 或 `fib(n-3)` 调用会产生一个对 `fib(n-4)` 的调用，因此对 `fib(n-4)` 调用了5次。这样就产生一个连锁效应：每个递归调用会做越来越多的冗余的工作。

设 $C(N)$ 表示在计算 $\text{fib}(n)$ 的过程中对 fib 函数的调用次数。显然 $C(0)=C(1)=1$ 次调用。对于 $N \geq 2$ ，我们调用 $\text{fib}(n)$ ，计算 $\text{fib}(n)$ ，需要递归地计算 $\text{fib}(n-1)$ 和 $\text{fib}(n-2)$ ，然后将这两个函数的值相加。这样 $C(N) = C(N-1) + C(N-2) + 1$ 。用归纳法很容易证明，对于 $N \geq 2$ 时，这个递归公式的答案是 $C(N) = F_{N+2} + F_{N-1} - 1$ 。这样递归调用的次数比我们要算的斐波那契数还要大，而且是按指数变化的。对于 $N=40$ ， $F_{40}=102\ 334\ 155$ ，总的递归调用的次数大于300 000 000。这个程序永远运行也不足为奇。递归调用次数的爆炸性增长如图6-2所示。

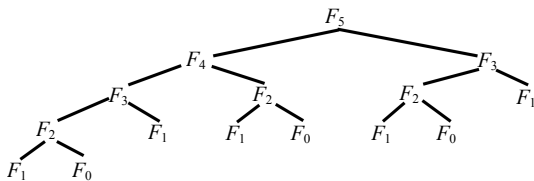


图6-2 fib(5)的计算过程

为了节约重复求相同子问题的时间，可引入一个数组，不管它们对最终解是否有用，把所有子问题的解存于该数组中，这就是动态规划的基本思想。例如，要求 $\text{fib}(n)$ ，可以设置一个 $n+1$ 个元素的数组 f ，并设 $f[0]=0$ ， $f[1]=1$ ，然后依次计算 $f[2]$ ， $f[3]$ ， \dots ， $f[n]$ 。这样计算 $\text{fib}(n)$ 只用了 $n-1$ 次的加法。下面用硬币找零问题进一步说明动态规划的应用。

例6.14 硬币找零问题。对于一种货币，有面值为 C_1, C_2, \dots, C_n （分）的硬币，最少需要多少个硬币来找出 K 分钱的零钱。

怎么来解决硬币找零问题呢？可以采用递归的思想：如果可以用一个硬币找零，这就是答案；否则，对于每个可能的值 i ，独立计算找 i 分钱零钱和 $K-i$ 分钱需要的最小硬币数。然后选择硬币数和最小的 i 这组的方案。

例如，如果有面值为1,5,10,21,25分的硬币，为了找出63分钱的零钱，可以分别尝试下列情况。

- 找出1分钱零钱和62分钱零钱分别需要的硬币数是1和4。因此，63分钱需要使用5个硬币。
- 找出2分钱和61分钱分别需要2和4个硬币，一共是6个硬币。

继续尝试所有的可能性。可以看到一个21分和42分的分解，它可以分别用一个和两个硬币来

找开，因此，这个找零问题就可以用3个硬币解决。

这种方法可以用一个很简单的递归算法来实现，伪代码如下：

```
//参数k为要找零的钱，返回值为所需的硬币数
int coin(int k)
{   int i, tmp, int coinNum = k;

    if(能用一个硬币找零)return 1;
    for (i=1; i<k; ++i)
        if ((tmp = coin(i) + coin(k-i)) < coinNum) coinNum = tmp;
    return coinNum;
}
```

这个算法的效率很低，就如求斐波那契数列一样，在求`coin(k)`的过程中，某些`coin(i)`会被反复调用多次。而且让`i`顺序增长也不合理。例如，硬币的币值为1,5,10,21,25，那么在分解了1和 $k-1$ 后再分解成2和 $k-2$ 是没意义的，因为没有币值为2的硬币，2显然必须被分解成两个1。一个较为合理的方法是按硬币额分解。例如，对于63分钱，可以给出以下找零的办法。

- 一个1分的硬币加上递归地分派62分钱。
- 一个5分的硬币加上递归地分派58分钱。
- 一个10分的硬币加上递归地分派53分钱。
- 一个21分的硬币加上递归地分派42分钱。
- 一个25分的硬币加上递归地分派38分钱。

这个算法的问题仍然是由于重复计算带来的效率问题。

效率低下主要是由于重复计算造成的。因此，可把已有子问题的答案存放起来，当再次遇到这个子问题时就不用重复计算了。这就是动态规划的思想。

在本例中，我们用`coinsUsed[i]`代表找`i`分零钱所需的最小硬币数，而当`i`等于`k`时的解就是我们正在寻找的解。

如果数组`coins`存储某种币制对应的硬币的币值，`differentCoins`表示不同币值的硬币个数，`maxChange`是要找的零钱，则算法过程如下：先找出1分钱的找零方法，把最小硬币数存入`coinUsed[1]`，1分钱的找零就是一个币值为1的硬币，因此`coinUsed[1]`为1。然后再依次找出2分钱、3分钱、……的找零方法。对每个要找的零钱`i`，可以通过尝试所有的硬币，把`i`分解成某个`coins[j]`和 $i - \text{coins}[j]$ ，由于 $i - \text{coins}[j]$ 的解已经存在，存放在`coinUsed[i - coins[j]]`中，如果能够采用`coins[j]`的话，所需硬币数为`coinUsed[i - coins[j]] + 1`。对所有的`j`，取最小的`coinUsed[i - coins[j]] + 1`作为`i`分钱找零的答案。根据该算法得到的函数实现如下：

```
void makechange( int coins[ ], int differentCoins, int maxChange, int coinUsed[ ] )
{   coinUsed[0] = 0;
    for(int cents = 1; cents<= maxChange; cents++){//找出所有1到maxChange之间的找零方案
        int minCoins = cents; //都用1分找零，硬币数最大
        for (int j = 1; j < differentCoins; j++) { //尝试所有硬币
            if(coins[j] > cents) continue; //coin[j]硬币不可用
```

```
        if(coinUsed[cents - coins[j]]+1<minCoins) //分解成coins[i]和cents-coins[j]
            minCoins = coinUsed[ cents - coins[j] ] + 1; //用此硬币
    }
    coinUsed[cents] = minCoins;
}
}
```

这段代码只给出了找零maxChange所需的最少硬币数，而没有给出具体由哪些硬币组成。读者可自行修改这个程序，使之可以得到这个信息。

小结

本章介绍了程序设计的一个重要的概念——函数。函数可以将一段完成独立功能的程序封装起来，通过函数名就可执行这一段程序。使用函数可以将程序模块化，每个函数完成一个独立的功能，使程序结构清晰、易读、易于调试和维护。

C++程序是由一组函数组成。每个程序必须有一个名为main的函数，它对应于一般程序设计语言中的主程序。每个C++程序的执行都是从main函数的第一条语句执行到main函数的最后一条语句。main函数的函数体中可能调用其他函数。

函数中定义的变量和形式参数称为局部变量，它们只在函数体内有效。当函数执行时，生成这些变量；当函数执行结束时，这些变量被销毁。

还有一类变量是定义在所有函数的外面的，被称为全局变量。全局变量的作用域是从定义点到文件结尾。凡是在它后面定义的所有函数都能使用它。全局变量提供了函数间的一种通信手段。

函数也可以调用自己，这样的函数称为递归函数。程序设计中的许多算法设计技术都是基于递归的，如分治法、回溯法和动态规划等。

习题

简答题

1. 为什么需要函数原型声明？
2. 什么是形式参数？什么实际参数？形式参数和实际参数有什么关系？
3. 传递一个数组为什么需要两个参数？
4. C++为什么要用内联函数取代带参数的宏？试举例说明。
5. 什么是值传递？
6. 什么是函数模板？什么是模板函数？
7. C++是如何实现函数重载的？
8. 使用全局变量有什么好处？有什么坏处？
9. 变量定义和变量声明有什么区别？

程序设计题

1. 设计一个函数，判别一个整数是否为素数。
2. 设计一个函数，使用以下无穷级数计算 $\sin x$ 的值。 $\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$ 。舍去的绝对值应小于 ε ， ε 的值由用户选择。
3. 写一个函数stringCopy将一个字符串复制到另一个字符串。
4. 写一个函数实现气泡排序。
5. 设计一函数求两个正整数的最大公约数。
6. 设计一个二分查找的递归函数。
7. 设计一个支持整型、实型和字符型数据的快速排序的函数模板。
8. 修改解决硬币找零问题的程序，使之能输出解的组成。

指针是C++中的重要概念。所谓的指针就是内存的一个地址。利用指针可以尽可能多地访问由硬件本身提供的功能。所以，不理解指针是如何工作的就不能很好地理解C++程序。此外，如果想成为出色的C++程序员，必须学得更深入，学习如何在程序中更加有效地使用指针。

在C++语言中，指针有多种用途。指针可以增加变量的访问途径，使变量不仅能够通过变量名直接访问，也可以通过指针间接访问；此外，指针可以使程序中的不同部分共享数据，也可以通过指针在程序执行过程中动态申请空间。

本章将介绍指针的基本概念和应用。

7.1 指针的概念

在C++语言中，任何一个可以出现在赋值语句左边的表达式称为左值（lvalue）。比如，简单变量就是左值，因为可以写这样的语句：

```
x = 1.0;
```

同样，数组元素也是左值，因为可以直接给其赋值，例如：

```
intarray[2] = 17;
```

但C++语言中的很多值并不是左值。例如，常量不是左值，因为常量不能出现在赋值运算符的左边；同样，算术表达式的计算结果是值，但并不是左值，因为把值赋给算术表达式的结果是非法的。

左值的概念可以更明确地表示成更一般的形式。

- (1) 每个左值在内存中都占据一定的空间，因此必有地址。
- (2) 一旦左值被声明，尽管左值的内容可以改变，但它的地址永远不能改变。
- (3) 按照所保存的数据类型，不同左值需要不同的内存大小。
- (4) 左值的地址本身也是数据，也能在内存进行操作和存储。

最后一条初看并不那么引人注目，但用于程序设计就会显示出深远的意义。考虑这样一个声明：

```
int x;
```

这一定义为整型的x在内存的某处保留了一个存储空间。例如，如果在程序运行的计算机上，存

储整型数需要4个字节的空间，那么变量x可能会得到从1000~1003这4个存储单元。根据第4条原则，变量x的存储地址1000本身也是一个数据值（毕竟根据字面意义，值1000只是一个整数），可以存入内存，存储该地址的变量就称为一个指针。我们可以通过该指针变量间接访问变量x的值。

如果系统给x分配的空间是1000号存储单元，则指向x的指针是另一个变量，该变量中存放的数据为1000，如图7-1所示。

综上所述，指针就是存储左值地址的变量。指针提供了更灵活的访问方式。

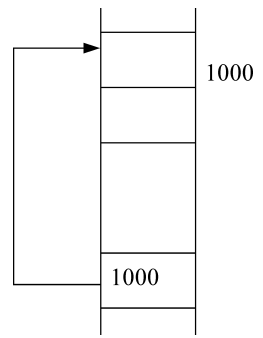


图7-1 指针的含义

7.1.1 指针变量的定义

指针存储的是一个内存地址，它的一个重要用途就是通过指针间接访问所指向的地址中的内容。因此，定义一个指针变量要说明3个问题：该变量的名字是什么，该变量中存储的是一个地址（即是一个指针），该地址中存储的是什么类型的数据。在C++中，指针变量的定义如下：

类型名 *指针变量名；

其中，*表示后面定义的变量是一个指针变量，类型名表示该变量指向的地址中存储的数据的类型。比如，定义

```
int *p;
```

表示定义了一个指针变量p，该指针变量指向的地址中存储的是一个整型数。类似地，定义

```
char *cptr
```

表示定义了一个指向字符型数据的指针变量cptr。虽然变量p和cptr中存储的都是地址值，在内存中占有同样大小的空间，但这两个指针在C++语言中是有区别的。编译器会用不同的方式解释指针指向的地址中的内容。指针指向的值的类型称为指针的基本类型。所以，p的基本类型为int，cptr的基本类型是char。

注意，表示变量为指针的星号在语法上属于变量名，不属于前面的类型名。如果使用同一个定义语句来定义两个同类型的指针的话，必须给每个变量都加上星号标志，例如：

```
int *p1, *p2;
```

而定义

```
int *p1, p2;
```

则表示定义p1为指向整型的指针，而p2是整型变量。

7.1.2 指针的基本操作

指针变量最基本的操作是让它指向某一内存地址，以及访问它指向的地址中的内容。

1. 让指针指向某一内存地址

让指针指向某一内存地址有两种方法。一种是将某一变量的地址赋给指针变量，另一种方法是将一指针变量的值赋给另一个指针变量。

让指针变量指向某一变量，就是将一个变量的地址存入指针变量。但程序员并不知道变量在内存中的地址，而且每次程序执行时变量在内存中的地址都是不同的。为此，C++提供了一个取地址运算符&。&运算符将一个左值作为运算数，返回该左值对应的内存地址。例如，定义

```
int *p, x;
```

可以用`p = &x`将变量x的地址存入指针变量p。指针变量也可以在定义时赋初值，例如：

```
int x, *p = &x;
```

定义了整型变量x和指向整型的指针p，同时让p指向x。

与普通类型的变量一样，C++在定义指针变量时只负责分配空间。除非在定义变量时为变量赋初值，否则该变量的初值是一个随机值。因此，引用该指针指向的空间是没有意义的，甚至是很危险的。为了避免这样的误操作，不要引用没有被赋值的指针。如果某个指针暂且不用的话，可以给它赋一个空指针NULL。NULL是C++定义的一个符号常量，它的值为0，表示不指向任何地址。NULL可以赋给任何指针变量。在引用指针指向的内容时，先检查指针的值是否为NULL是很有必要的，这样可以确保指针指向的空间是有效的。

除了可以直接把某个变量的地址赋给一个指针变量外，同类的指针变量之间也可以相互赋值，表示两个指针指向同一内存空间。例如，有定义

```
int x = 1, y = 2, *p1 = &x, *p2 = &y;
```

系统会在内存中分别为4个变量准备空间，把1存入x，把2存入y，把x的地址存入指针p1，把y的地址存入指针p2。如果在上述语句的基础上继续执行`p1 = p2`，执行完这个赋值表达式后，p1和p2指向同一空间，即指向p2原来指向的变量y，对变量x和y的值没有任何影响。

在指针互相赋值的时候必须注意，赋值号两边的指针类型必须相同。不同类型的指针之间不能赋值。道理很简单，如果p1是指向整型数的指针，而p2是指向double型的指针，执行`p2 = p1`，然后通过*`p2`去引用p2指向的内容，会发生什么问题？由于p2是指向double型的指针，在VisualC++中引用*`p2`时就会取8个字节的内容，然后把它解释成一个double型的数，而整型变量在VisualC++中只占4个字节！即使两个指针类型不同，但指向的空间大小是一样的，这两个指针间的赋值还是没有意义的。如果p1是指向整型数的指针，而p2是指向float型的指针，执行`p2 = p1`，然后通过*`p2`去引用p2指向的内容，则会将一个整型数在内存中的表示解释成一个单精度数。这是没有意义的。因此，C++不允许不同类型的指针之间互相赋值。如果必须在不同类型的指针间相互赋值，必须使用强制类型转换，表示程序员知道该赋值的危险。

2. 引用指针指向的值

定义指针变量的目的并不是要知道某一变量的地址，而是希望通过指针能间接地访问某一变量的值。因此，C++语言定义了一个取指针指向的值的运算符*。*运算符是一元运算符，它的运算对象是一个指针。*运算符根据指针的类型，返回其指向的左值。*运算产生一个左值，可以引用此左值的值或为此左值赋值。例如，有定义

```
int x, y;
int *p;
```

这两个定义为3个变量分配了内存空间，两个是int类型，一个是指向整型的指针。为了更具体一些，假设这些值在机器中存放的地址如图7-2所示。

执行了语句：

```
x = 3; y = 4; intp = &x;
```

之后内存中的情况如图7-3所示。

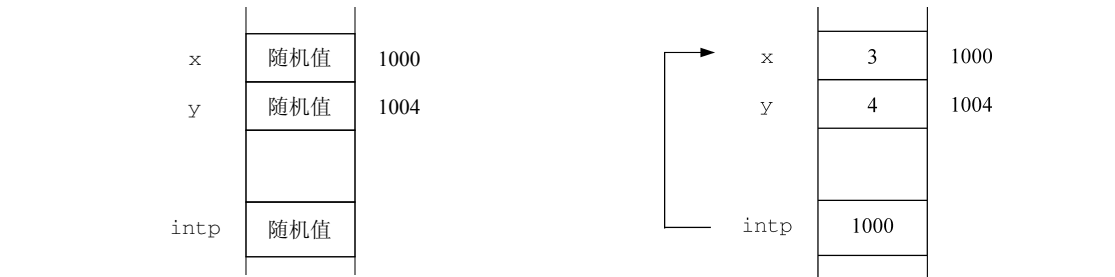


图7-2 为x、y、intp分配内存空间

图7-3 执行x=3;y=4;intp=&x;之后内存的情形

如果执行了语句

```
*intp = y + 4;
```

intp本身的值并没有改变，它仍然为值1000，所以仍然指向变量x。但是intp指向的单元内的值，即x的值，被改变了。改变后的内存情况如图7-4所示。

指针变量可以指向不同的变量。例如，上例中intp指向x，可以通过对intp的重新赋值改变指针的指向。如果想让intp指向y，只要执行intp = &y;就可以了。这时，intp与x再无任何关系。此时*intp的值为4，即变量y的值，如图7-5所示。

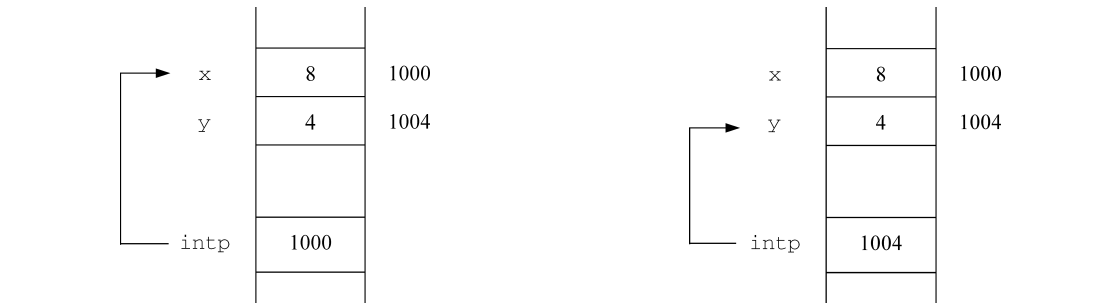


图7-4 又执行*intp=y+4;之后内存的情形

图7-5 又执行intp=&y;之后内存的情形

3. 统配指针类型void

在C++中，可以将指针的基本类型声明为void。void类型的指针只说明了这个变量中存放的是一个地址，但未说明该地址中存放的是什么类型的数据。在标准C++中，只有相同类型的指针之间能互相赋值，但任何类型的指针都能与void类型的指针互相赋值，因此void类型的指针

被称为统配指针类型。

统配指针类型的定义方式如下：

```
void *指针变量名;
```

统配指针类型的应用将在后面用到时介绍。

4. 指针与常量限定符const

有了指针后，变量不仅能通过变量名直接访问，也能通过指针间接访问。这增加了程序的灵活性，但也给程序的调试带来了困难。因为程序员很难确定某个变量是否被修改。为了防止通过指针随意修改变量的值，可以采用常量限定符const来限制指针操作。

const限定符可以用来定义常量，也可以与指针一起使用。const限定符与指针一起使用的组合情况比较复杂，总结起来有以下3种。

(1) 指向常量的指针：是一个指向的内容是常量的指针变量。例如，下列语句定义了一个指向整型常量的指针p，并在初始化时让它指向整型变量x：

```
const int *p = &x;
```

由于使用了const说明指针指向的内容是一个常量，因而不能修改它指向的内容。如果程序中有类似于如下赋值，编译器会指出“左值是一个常量”。

```
*p = 30;
```

但是，由于p是一个普通的指针变量，我们可以让指针指向另一个地址。例如，下面的语句则是正确的：

```
p = &y;
```

(2) 常指针：是指针本身是个常量，它固定指向某一变量。因此它本身的值不能变，但它指向的地址中的值是可变的。常指针的定义方法如下：

```
int *const p = &x;
```

这条语句的含义是定义一个名为p的常指针，指向变量x的地址，而且永远只能指向x，不能指向其他地址。它指向的地址里的内容是可变的。因此，*p = 20是正确的赋值，而p = &y则是不正确的赋值。

(3) 指向常量的常指针：是指指针本身不能改变，指向的地址中的值也不能改变。指向常量的常指针的定义如下：

```
const int * const p = &x;
```

这条语句的含义是定义一个名为p的指向常量的常指针，让它指向变量x的地址，而且永远只能指向x，不能指向其他地址。并且不能通过它修改指向的地址里的内容。因此，*p = 20和p = &y都是不正确的赋值。

7.2 指针与数组

到目前为止，本章中关于指针使用的例子仅限于指针指向某一简单变量的情况。实际上，指

针可以指向任何左值。特别是，数组元素是左值，因此，指针也能指向数组元素。例如，定义了数组

```
int intarray[5];
```

系统会预留5个连续的内存单元，每个单元的大小都足以存放一个int类型的值。假设int型的数据占4个字节，系统为该数组分配的内存起始地址为100，则该数组内存如下所示：

随机值		随机值		随机值		随机值		随机值	
100	103	104	107	108	111	112	115	116	119

5个数组元素都有地址，这些地址可由&运算符得到。例如，表达式

```
&intarray[1]
```

的值为104，因为元素intarray[1]存储于该地址。&intarray[i]是合法的，表示intarray中第i个元素的地址。

由于intarray中第i个元素的地址取决于数组的起始地址和变量i的值，C++语言编译器在编译程序时是无法计算这一地址的。为了确定这一地址，编译器产生一段指令，取出数组的基地址，加上适当的偏移量，该偏移量是将i的值乘以每个数组元素占用的字节数。所以，计算intarray[i]的地址可以由公式100+i×4得到。例如，如果i为2，地址计算的结果即为108，与intarray[2]的地址是一致的。因为计算任何一个数组元素地址的过程是自动的，在写程序时无需考虑计算的细节问题。

在C++中，指针和数组关系密切，几乎可以互换使用。对一维数组来说，数组名是数组的起始地址，也就是第0个元素的地址。因此，数组名可以看成是一个指针，而且是一个常指针，永远指向这一地址。如果定义了整型指针p，执行p=intarray，则p与intarray是等价的，对指针p可以进行任何有关数组下标的操作。例如，可以用p[3]引用intarray[3]。

7.2.1 指针运算

当+和-运算符的运算数为数字时，结果由简单的数学运算决定。在C++语言中，也可以将这些运算符应用于指针。在某些方面，其结果与数学运算相似，但另一些方面又有所不同。对指针值应用加减的过程称为指针运算。

指针运算规则如下：如果指针p指向数组arr的第一个元素且k为整型数，下面的关系总是成立的：p+k等同于&arr[k]。

换句话说，如果在指针值上加上一个整数k，则结果就是起始地址为原指针值的数组的下标值为k的元素的地址。

为了解释指针运算规则，假设有一个函数包含下面的变量声明：

```
int intarray[5];
int *p;
```

在函数帧内会给每个变量都分配空间。对数组变量intarray，编译器为它分配了足够的空间用

于存放5个int型元素的数据。对指针p，编译器为其分配了足够的空间用于存放某个地址。如果帧从地址100开始，内存分配如下所示：

intarray[0]	intarray[1]	intarray[2]	intarray[3]	intarray[4]	p
随机值	随机值	随机值	随机值	随机值	随机值
100	103	104	107	108	111
112	115	116	119	120	123

由于还未给这些变量赋值，所以其初始值都是随机数。可以给它们赋值。例如，可以使用以下的赋值语句将任意值存储到每个数组元素中去：

```
intarray[0] = 0;
intarray[1] = 1;
intarray[2] = 2;
```

要初始化指针变量p，使其指向数组的起始地址，可以执行下面赋值语句：

```
p = &intarray [0];
```

或

```
p = intarray;
```

这些赋值完成后，内存单元中存放的是以下这些值：

intarray[0]	intarray[1]	intarray[2]	intarray[3]	intarray[4]	p
0	1	2	随机值	随机值	100
100	103	104	107	108	111
112	115	116	119	120	123

指针p指向了数组intarray的起始地址。如果给指针p加整数k，其结果就与下标为k的数组元素的地址相对应。例如，如果程序包含表达式p+2，表达式求得的值将是intarray[2]的地址。所以，当p指向地址100时，p+2指向数组中该元素后出现的第二个元素的地址，即地址108。需要注意的是，指针加法和传统加法是不同的，因为指针运算必须考虑到基本类型占用的内存空间。本例中，由于每个int型需要4个字节，所以指针值每增加1个单位，内部数值必须增加4。

C++语言编译器解释指针减去整数时，也采用类似的方法。表达式p-k中，p是一个指针，k是一个整数，计算的是数组中由p当前值指向的地址前的第k个元素的地址。所以，如果设p指向intarray[1]的地址：

```
p = & intarray [1];
```

则与p-1及p+1相对应的地址分别为intarray[0]和intarray[2]的地址。

从程序员的角度来看，重要的是认识到指针运算能自动考虑基本类型的大小。这可以使程序员在操作地址时不需知道每个变量占用的实际内存量。给定任何指针p和整数k，则表达式p+k意味着不管每个元素需要多少内存，结果总是数组中p目前指向的地址后第k个元素的指针。数组元素所需的内存大小只在理解计算机内部如何进行计算操作时才有用。

算术运算符*、/和%对指针来说是没有意义的，不能和指针一起使用。而且，+和-的使用也

是有限的。在C++中，可以对一个指针加上或减去一个整数偏移量，但不可以将两个指针相加。其他唯一可用于指针的算术运算是两个指针相减，表达式`p1-p2`中，`p1`和`p2`是指针，用于返回在`p2`值和`p1`值之间的数组元素的个数。例如，指针`p1`指向`intarray[2]`，`p2`指向`intarray[0]`，则表达式`p1-p2`的值为2，因为两指针值之间有2个元素。

7.2.2 用指针访问数组

C++最不寻常的特征中，最有趣的是数组名作为指向该数组第一个元素的指针的同义词。这个概念很容易用例子解释。

数组定义

```
int array [5];
```

定义了一个含有5个整型数的数组。计算机为数组在内存的某处分配了空间，如地址100，则该数组的内存占用情况如下所示：

随机值		随机值		随机值		随机值		随机值	
100	103	104	107	108	111	112	115	116	119

`array`代表一个数组，但也可以直接用作指针。当作为指针使用时，`array`定义为数组中第一个元素的地址。对于任意数组`array`，下面的关系在C++中是永远成立的：`array`等同于`&array[0]`。

`array+k`就是数组`array`的第`k`个元素的地址。反之，一旦将一个数组名赋给了一个指针后，该指针就可以看作这个数组的起始地址，可以把它当作数组来使用。

了解了数组和指针的关系，数组的操作就更灵活了。例如，要输出数组`array`的5个元素，下面5段代码都是合法的：

```
for ( i=0; i<5; ++i ) cout << array[i] ;
for ( i=0; i<5; ++i ) cout << *(array + i);
for ( p = array; p < array + 5; ++p ) cout << *p ;
for ( p = array, i=0; i<5; ++i ) cout << *(p+i);
for ( p = array, i=0; i<5; ++i ) cout << p[i] ;
```

这是否意味着数组和指针是等价的？不，数组和指针是完全不同的。定义

```
int array[5];
```

和定义

```
int *p;
```

间最基本的区别在于内存的分配。假如整型数在内存中占4个字节，地址的长度也是4个字节，那么第一个定义为数组分配了20个字节的连续内存，能够存放数组元素。第二个定义只分配了4个字节的内存空间，其大小只能存放一个机器地址。

认识这一区别对于程序员来说是至关重要的。如果定义一个数组，则需要有存放数组元素的工作空间；如果定义一个指针变量，则只需要一个存储地址的空间。指针变量在初始化之前与任

何内存空间都无关。

按照目前的理解，将指针作为数组使用的唯一途径是通过将数组的基地址赋给指针变量来初始化指针。如果进行过上述定义后，语句

```
p = array;
```

使指针变量p和array指向同样的地址，两者可以互换使用。

将指针指向一个已经存在的数组的应用是相当有限的。毕竟，如果已经有一个数组名的话就可以直接使用了，没必要再通过指针来访问它。把数组名赋给指针实际上并没有什么好处。将指针作为数组使用的实际优势在于，你可以把指针初始化为未声明的新内存，从而能在程序运行时建立新的数组。这一重要的程序设计技巧将会在下一节详细介绍。

7.2.3 数组名作为函数的参数

数组名可以作为函数的形式参数和实际参数。由6.3节的讨论可知，当数组名作为函数的参数时，形式参数和实际参数实际上是共享了同一块存储空间。函数内对形式参数数组的任何修改都是对实际参数数组的修改。

学习了指针后，对此问题就更容易理解了。因为数组名代表的是数组的首地址，所以，如果将数组名作为函数的参数，在函数调用时是将数组的首地址（而不是数组的值）传给形式参数。例如，如果函数的形式参数是名为arr的数组，实际参数是名为array的数组，在参数传递时将执行arr = array，这样数组arr和array的首地址是相同的，即两个数组共享了一块空间。所以，数组传递的本质是地址传递。

实际上，C++是将形式参数的数组作为指针来处理的。这可以通过代码清单7-1所示的程序做一个简单的测试。

代码清单7-1 数组作为函数的参数的示例程序

```
//文件名: 7-1.cpp
//数组作为函数的参数的示例
#include <iostream>
using namespace std;

void f(int arr[]);

int main()
{   int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

    cout << sizeof(array) << endl;
    f(array);

    return 0;
}

void f(int arr[])
{   cout << sizeof(arr) << endl; }
```

程序的输出如下：

40
4

代码清单7-1所示的程序说明：在main函数中，数组array占用了40个字节，即10个整型数占用的空间；在函数f中，作为形式参数的数组arr占用的内存量是4个字节，即一个指针所占的空间量。因此，当数组名作为函数参数传递时，形式参数表示为数组和指针实质上是一样的。实质上，实际参数写成数组和指针也是一样的，只要作为实际参数的指针指向的空间是存在的。

作为一般规则，声明参数必须能体现出它们的用途。如果想要将一个形式参数作为数组使用，并从中选择元素，那么应该将该参数声明为数组。如果你想要将一个形式参数作为指针使用，并且引用其指向的内容，那么应该将该参数声明为指针。用指针作为形式参数将在7.5.1节详细讨论。

用这个观点来看数组传递的问题，可以使程序更加灵活。例如，有一个排序函数

```
void sort (int p[], int n)
```

可以排序数组p，数组p有n个元素。如果需要排序一个有10个元素的数组a，可以调用sort(a,10)。如果要排序前半元素，则可调用sort(a,5)。如果要排序后半元素，则可调用sort(a+5,5)。

7.3 指针与动态分配

在C++中，每个程序需要用到几个变量在写程序前就应该知道，每个数组有几个元素也必须在写程序时就决定。有时我们并不知道需要多大的数组或需要多少个变量，直到程序开始运行，根据某一个当前运行值才能决定。例如，设计一个打印魔阵的程序，直到输入了魔阵的阶数后才知道数组应该有多大。

在第5章中，我们建议按最大的可能值定义数组，每次运行时使用数组的一部分元素。当元素个数变化不是太大时，这个方案是一个可行的；但如果元素个数的变化范围很大，这个方案就太浪费空间了。

这个问题的一个更好的解决方案就是动态变量机制。所谓动态变量是指：从静态程序中无法确定它们的存在，只有当程序运行起来，随着程序的运行，根据程序的需求动态产生和消亡，因此无法在写程序时定义的变量。由于动态变量不能在程序中定义，也就无法给它们取名字，因此对于动态变量的访问需要通过指向动态变量的指针变量来进行间接访问。要使用动态变量，必须定义一个相应类型的指针，然后通过动态变量申请的功能向系统申请一块空间，将空间的地址存入该指针变量。这样就可以间接访问动态变量了。当程序运行结束时，系统会自动回收指针占用的空间，但并不会回收动态变量的空间，这就需要程序员在程序中释放动态变量的空间。因此要实现动态变量，系统必须提供3大功能。

- 定义指针变量。
- 动态申请空间。
- 动态回收空间。

如何定义一个指针变量在本章前面已经介绍了，下面介绍如何动态申请空间和回收空间。

7.3.1 动态变量的创建

C++的动态变量的创建使用运算符new。运算符new可以创建一个普通变量或一个数组。创建一个普通的动态变量的格式如下：

```
new 类型名;
```

这个操作在内存中称为堆（heap）的区域申请一块能存放相应类型的数据的空间，操作的结果是这块空间的首地址。例如，要动态产生一个int型的变量，将20存于其中，可以用下列语句：

```
int *p;  
p = new int;  
*p = 20;
```

由于new操作是有类型的，它的结果只能赋给同类指针。因此，下面的操作是非法的：

```
int *p;  
p = new double;
```

在创建动态变量时，还可以指定空间中的初值。例如：

```
int *p = new int(10);
```

相当于

```
int *p = new int;  
*p = 10;
```

用new操作也可以创建一个一维数组。它的格式如下：

```
new 类型名[元素个数];
```

这个操作在内存的堆区域申请一块连续的空间，存放指定类型的一组元素，操作的结果是这块空间的首地址。例如，要动态产生一个10个元素的int型的数组，可以用下列语句：

```
int *p;  
p = new int[10];
```

此时，p指向这块空间的首地址，因此，相当于是数组名。如果要将p数组的第二个元素赋值为20，则可用赋值运算 $p[2] = 20$ 。

动态数组和普通数组的最大区别在于，它的长度可以是程序运行过程中某一变量的值或某一表达式的计算结果，而普通数组的长度必须是在编译时就能确定的常量。例如：

```
p = new int [2*n];
```

表示申请了一个动态数组，它的元素个数是变量n当前值的两倍。但下列操作在C++中是非法的。

```
int p[2*n];
```

动态数组不能为数组赋初值。

7.3.2 动态变量的回收

在C++程序运行期间，动态变量不会消亡。甚至，在一个函数中创建了一个动态变量，在该函数返回后，该动态变量依然存在，仍然可以使用。要回收动态变量的空间必须显式地使之

消亡。要回收某个动态变量，可以使用delete操作。对应于动态变量和动态数组，delete也有两种用法。

要回收一个动态变量，可以用

```
delete 指针变量;
```

该操作将会回收该指针指向的空间。例如：

```
int *p = new int(10);
delete p;
```

要回收一个动态数组，可以用

```
delete [] 指针变量;
```

回收由该指针变量值作为数组首地址的数组的空间。

一旦释放了内存区域，堆管理器重新收回这些区域。虽然指针仍然指向这个堆区域，但已不能再使用指针指向的这些区域。在程序中同一个区域只能释放一次。

7.3.3 内存泄漏

在动态变量的使用中，最常出现的问题就是内存泄漏。所谓的内存泄漏就是你用动态变量机制申请了一个动态变量，而后不再需要这个动态变量时没有delete它；或者把一个动态变量的地址放入一个指针变量，而在此变量没有delete之前又让它指向另一个动态变量。这样原来那块空间就丢失了。堆管理器认为你在继续使用它们，而你却不知道它们在哪里。

为了避免出现这种情况，你应该用delete明白地告诉堆管理器这些区域你不再使用。

释放内存对一些程序不重要，但对有些程序非常重要。如果你的程序要运行很长时间，而且存在内存泄漏，这样程序最终可能会耗尽所有内存，直至崩溃。

7.3.4 查找 new 操作的失误

由于计算机内存的空间是有限的，堆空间最终也可能用完，此时new操作就会失败。为可靠起见，在new操作后最好要检查一下操作是否成功。new操作是否成功可以通过它的返回值来确定。当new操作成功时，返回申请到的堆空间的一个地址。如果不成功，则返回一个空指针。在程序中，也可以利用C++的assert()宏来确定new操作是否成功。当检测到new操作不成功时，直接退出程序。

下面的程序中，申请一个动态的整型变量。如果空间申请失败，输出allocation failure，退出main函数；否则，对该动态变量赋值，并输出它的值。

```
int main()
{   int *p;
    p = new int;
    if (!p){        //也可写成 if (p == NULL)
        cout<<"allocation failure\n";
        return 1;
    }
}
```

```
*p=20;
cout << *p;
delete p;

return 0;
}
```

也可以利用assert宏直接退出程序。assert()宏定义在标准头文件cassert中。使用assert()时,给它一个参数,即一个表示断言为真的表达式,预处理器产生测试该断言的代码。如果断言不是真,则在发出一个错误消息后程序会终止。assert()的用法如下所示:

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

int main()
{   int *p;

    p = new int;
    assert (p != 0);
    *p=20;
    cout << *p;
    delete p;

    return 0;
}
```

当空间申请失败时,断言为假,程序会终止。

7.4 字符串再讨论

字符串有两种基本表示:字符数组表示和指向字符的指针表示。在第5章中,已经介绍了字符串的字符数组表示,本节介绍一下字符串的指针表示。

字符串的指针表示是定义一个指向字符的指针,如下例所示:

```
int main()
{   char *string;

    string = "abcde";
    cout << string;

    return 0;
}
```

这里没有定义字符数组,只定义了一个指向字符的指针变量string。接下来的一条语句看起来有点奇怪,把一个字符串赋给一个指针!这个语句应该理解为将存储字符串“abcde”的内存的首地址赋给指针变量string。字符串常量一般与静态变量存放在一个区域,在程序执行过程中始终存在。

由于在C++中,数组名被解释成指向数组首地址的指针。因此,尽管在上例中字符串是用一

个指针变量表示，还是可以把此指针变量解释成数组的首地址，通过下标访问字符串中的字符。例如，`string[3]`的值是d。但由于该指针指向的是一个常量，因此不能修改此字符串。

除了可以直接将字符串常量赋给一个指向字符的指针外，用一个指向字符的指针表示字符串还有下面两种用法。

(1) 将字符数组名赋给指针。

(2) 用动态内存申请的方法申请一块存储字符串的空间，让指针指向这块空间。

前者真正的字符串是存储在栈区域中。后者，真正的字符串是存储在堆中。例如，如果定义

```
char *s1, *s2;
char ch[] = "ffff";
```

执行

```
s1 = ch;
s2 = new char[10];
strcpy(s2, "ghj");
```

则内存情况如图7-6所示。

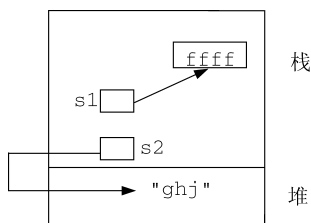


图7-6 用指向字符的指针表示字符的两种用法的对比示例

字符串作为函数参数传递与数组作为函数传递一样，形式参数和实际参数都可写成字符数组或指向字符的指针。但由于字符串有一个特定的结束标志‘\0’，因此与数组作为参数传递不同，传递一个字符串只需要一个参数，即指向字符串中第一个字符的指针，而不需要指出字符串的长度。

7.5 指针与函数

7.5.1 指针作为形式参数

函数的参数不仅可以是整型、实型、字符型等数据，也可以是指针变量。它的作用是将一个变量的地址传到一个函数中。这样可以进一步扩展函数的功能，降低参数传递的代价。

为了对参数的地址传递机制的本质有一个基本的了解，首先来看一个经典的例子：编写一个函数`swap`，使得它能够交换两个实际参数的值。初学者往往会写这样一个函数：

```
void swap(int a, int b)
{   int c = a;
```

```

    a=b;
    b=c;
}

```

如果在某个函数中想交换两个整型变量 x 和 y 的值,可以调用`swap(x, y)`。结果发现,变量 x 和 y 的值并没有交换!这是为什么呢?原因在于C++的参数传递方式是值传递。所谓的值传递就是:在执行函数调用时,将实际参数的值赋给形式参数,以后实际参数和形式参数再无任何关系。不管形式参数如何变化都不会影响实际参数。因此当执行`swap(x, y)`时,系统将 x 的值赋给 a ,把 y 的值赋给 b 。在`swap`函数中将 a 和 b 的数据进行了交换,但这个交换并不影响实际参数 x 和 y 。事实上,由于 a 和 b 是局部变量,当`swap`函数执行结束时,这两个变量根本就不存在。

为了能使形式参数的变化影响到实际参数,可以将形式参数定义成指针类型,在函数调用时,将实际参数的地址传过去,在函数中交换两个形式参数指向的空间中的内容,如下所示:

```

void swap(int *a, int *b)
{
    int c = *a;
    *a = *b;
    *b = c;
}

```

当要交换变量 x 和 y 的值时,可以调用`swap(&x, &y)`。如果 $x=3, y=4$,则调用时内存的情况如图7-7所示。

在函数内交换了 a 指向的单元和 b 指向的单元的内容,即 x 和 y 的内容。当函数执行结束时,尽管 a 和 b 已不存在,但 x 和 y 的内容已被交换。

用指针作为参数可以在函数中修改主调程序的变量值,必须小心使用!

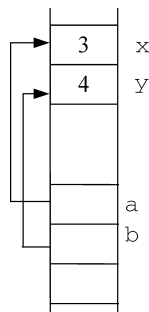


图7-7 调用`swap(&x, &y)`时内存的情形

例7.1 设计一个函数解一元二次方程。

到目前为止我们了解到的函数只能有一个返回值,由`return`语句返回,而一个一元二次方程有两个解,如何让此函数返回两个解?答案是在主函数中为方程的解准备好空间,如定义两个变量 x_1 和 x_2 ,把 x_1 和 x_2 的地址传给解方程的函数,在函数中将方程的解存入指定的地址。因此,函数原型可设计为

```
void SolveQuadratic(double a, double b, double c, double *px1, double *px2)
```

要解方程 $ax+bx+c=0$,可以调用

```
SolveQuadratic(a, b, c, &x1, &x2)
```

调用结束后变量 x_1 和 x_2 中包含方程的两个根。

尽管此函数能够解决一元二次方程返回两个根的问题,但它还有一些缺陷。在解一个一元二次方程时,并不是每个一元二次方程都有两个不同根,有时可能有两个等根,有时可能没有根。函数的调用者如何知道 x_1 和 x_2 中包含的是否是有效的解?我们可以对此函数原型稍加修改,让它返回一个整型数。该整型数表示解的情况。调用者可以根据返回值决定如何处理 x_1 和 x_2 。根据上述思想设计的解一元二次方程的函数及使用如代码清单7-2所示。

代码清单7-2 解一元二次方程的函数及其应用

```
//文件名: 7-2.cpp
//解一元二次方程的函数及其应用
#include <iostream>
#include <cmath>
using namespace std;

int SolveQuadratic(double a, double b, double c, double *px1, double *px2);

int main()
{
    double a,b,c,x1,x2;
    int result;
    cout << "请输入a,b,c: "; cin >> a >> b >> c;
    result = SolveQuadratic(a, b, c, &x1, &x2);

    switch (result) {
        case 0: cout << "方程有两个不同的根: x1 = " << x1 << " x2 = " << x2; break;
        case 1: cout << "方程有两个等根: " << x1; break;
        case 2: cout << "方程无根"; break;
        case 3: cout << "不是一元二次方程";
    }

    return 0;
}

//这是一个解一元二次方程的函数, a,b,c是方程的系数, px1和px2是存放方程解的地址
//函数的返回值表示根的情况: 0--有两个不等根
//                                1--有两个等根, 在px1中
//                                2--根不存在
//                                3--降级为一元一次方程
int SolveQuadratic(double a,double b,double c, double *px1,double *px2)
{
    double disc, sqrtDisc;

    if(a == 0) return 3; //不是一元二次方程

    disc = b * b - 4 * a * c;

    if( disc < 0 ) return 2; //无根

    if ( disc == 0 ) { *px1 = -b / (2 * a); return 1; } //等根

    //两个不等根
    sqrtDisc = sqrt(disc);
    *px1 = (-b + sqrtDisc) / (2 * a);
    *px2 = (-b - sqrtDisc) / (2 * a);
    return 0;
}
```

例7.2 设计一函数用分治法在一个整型数组中找出最大值和最小值。

用分治法解决这个问题的具体方法如下所述。

- 如果数组中只有两个元素, 则大的一个就是最大数, 小的那个就是最小数。(这种情况不需要递归。)
- 否则, 将数组分成两半, 递归找出前一半的最大值和最小值和后一半的最大值和最小值。取两个最大值中的较大者作为整个数组的最大值, 两个最小值中的较小值作为整个数组

的最小值。

按照分治法的思想，可设计出函数的原型。该函数输入的是数组的某一段数据，返回的是这一段数据中的最大值和最小值。因此该函数有4个参数：一个数组，数组中的元素个数，这段数据中的最大值，这段数据中的最小值。前面两个是函数的输入，用值传递，后面两个是函数的输出，用指针传递。因此函数的原型可设计为

```
void minmax ( int a[ ],int n, int *min_ptr, int *max_ptr)
```

函数的伪代码如下：

```
void minmax ( int a[ ], int n , int *min_ptr , int *max_ptr)
{
    switch (n){
        case 1: 最大最小都是a[0];
        case 2: 两者中的大的放入*max_ptr;小的放入*min_ptr;
        default:对数组a的前一半和后一半分别调用minmax;
                取两个最大值中的较大者作为最大值;
                取两个最小值中的较小值作为最小值;
    }
}
```

对这段伪代码进一步细化，可得到完整的程序，如下所示：

```
void minmax ( int a[], int n, int *min_ptr, int *max_ptr)
{
    int min1, max1, min2, max2;
    switch(n){
        case 1: *min_ptr = *max_ptr = a[0]; return;
        case 2: if (a[0] < a[1]) { *min_ptr = a[0]; *max_ptr = a[1]; }
                else { *min_ptr = a[1]; *max_ptr = a[0]; }
                return;
        default: minmax(a, n/2, &min1, &max1);
                minmax( a + n/2, n - n / 2, &min2, &max2 );
                if (min1 < min2) *min_ptr = min1; else *min_ptr = min2;
                if (max1 < max2) *max_ptr = max2; else *max_ptr = max1;
                return;
    }
}
```

7.5.2 返回指针的函数

函数的返回值可以是一个指针。例7.3给出了这样的应用。

例7.3 设计一程序，交换3个变量a、b、c中的最大值和最小值。如果变量a、b、c的值分别为1、2、3，则此程序的执行后，a、b、c的值分别为3、2、1。

这个问题的解决分成3个步骤：找出3个变量中的最大者，找出3个变量中的最小者，交换这两个变量的值。第三个步骤可以用已有的swap函数，我们还必须完成前两个工作。首先看找出最大者的函数。找出最大者的逻辑很简单，相信每位同学都会写，但问题是这个函数的原型如何设计。把它设计成int max(int a,int b,int c)? 不行，该函数返回了a、b、c这3个数中的最大值，但并不知道这个最大值是存放在哪一个变量中。我们希望知道的是哪个变量中的值最大，因此返回的应该是一个变量。标识一个变量可以用它的地址，该函数可以返回包含最大值的变量的地址。所以可把它的原型设计为int *max(int *a, int *b, int *c)。有同学可能要问，

为什么把形式参数设计成指针传递，而不用简单的值传递。确实，找出3个数中的最大值用值传递就可以了，但问题是要找出3个实际参数中哪个参数包含的值最大，并返回包含最大值的变量的地址，因此形式参数只能用指针传递。这3个函数的设计如下：

```
int *max(int *a, int *b, int *c)
{
    if (*a > *b)
        if (*a > *c) return(a); else return(c);
    else if (*b > *c) return(b); else return(c);
}

int *min(int *a, int *b, int *c)
{
    if (*a < *b)
        if (*a < *c) return(a); else return(c);
    else if (*b < *c) return(b); else return(c);
}

void swap(int *a, int *b)
{
    int c;
    c = *a; *a = *b; *b = c;
}
```

如果某个程序要交换x、y、z这3个变量中的最大和最小的两个变量值，只需要调用

```
swap ( max(&x, &y, &z), min(&x, &y, &z));
```

如果x=1, y=2, z=3，执行此调用后，3个变量的值分别为x=3, y=2, z=1。

值得注意的是，当函数的返回值是指针时，返回地址对应的变量可以是全局变量或动态变量，或调用程序中的某个局部变量，但不能是被调函数的局部变量。这是因为，当被调函数返回后，局部变量已消失，当调用者通过函数返回的地址去调用地址中的内容时，会发现已无权使用该地址。在Visual C++中，编译器会给出一个警告。

7.5.3 引用与引用传递

指针类型提供了通过一个变量间接访问另一个变量的能力。特别是，当指针作为函数的参数时，可以使主调函数和被调函数共享同一块空间，同时也提高了函数调用的效率；但是指针也会带来一些问题，如它会使程序的可靠性下降以及书写比较繁琐等。

为了获得指针的效果，又要避免指针的问题。C++提供了另外一种类型——引用类型，也能通过一个变量访问另一个变量，而且比指针类型安全。

1. 引用的定义和使用

所谓的引用就是给变量取一个别名，使一块内存空间可以通过几个变量名来访问。例如：

```
int i;
int &j = i;
```

变量j是变量i的别名，i与j用的是同一个内存单元。通过j可以访问i的空间。例如：

```
i = 1;
cout << j; //输出结果是1
j = 2;
```

```
cout << i; //输出结果是2
```

引用实际上是一种隐式指针。每次使用引用变量时，可以不用书写运算符“*”，因而简化了程序的书写。

在定义和使用引用类型的变量时，必须注意以下几点。

- 定义引用类型的变量时，必须在变量名前加上符号“&”，以示和普通变量的区别。
- 定义引用类型的变量时，必须立即对它初始化，不能在定义完成后再赋值。例如：

```
int i;
int &j; //错误
j=i;
```

- 为引用提供的初始值可以是一个变量或另一个引用。例如：

```
int i = 5;
int &j1 = i;
int &j2 = j1;
```

- 一旦将一个变量定义为某一个变量的别名，就不可使其作为另一变量的别名。这是指针和引用的主要区别。指针可以通过赋值指向另一个变量。例如，定义

```
int i, k, *p = &k;
int &j = i;
```

执行j=&k是错误的，但执行p=&i是正确的。

2. 引用类型作为函数的参数

C++引入引用的主要目的是将引用作为函数的参数。7.5.1节介绍了一个swap函数：

```
void swap(int *a, int *b)
{
    int c;
    c = *a;
    *a = *b;
    *b = c;
}
```

这个函数能交换两个实际参数的值，要交换变量x和y的值，可以调用swap(&x, &y)。但这个函数看起来很繁琐。函数中的3个赋值语句中的指针变量前都要加一个符号*。函数调用看起来也不舒服，在x和y前都要加地址符&。如果把形式参数改成引用类型，可以达到同样的目的，而且形式简单。使用引用类型参数的swap函数如下：

```
void swap(int &a, int &b)
{
    int c;
    c = a;
    a = b; b = c;
}
```

要用此函数交换变量x和y的值，可以调用swap(x, y)。在调用swap(x, y)时，相当于发生了两个引用类型的变量定义int &a = x, &b = y;，即a和x共用一块空间，b和y共用一块空间。因此，在函数内部a和b的交换就是x和y的交换。

在使用引用类型的参数时必须注意，调用时对应的实际参数必须为左值表达式(通常为变量)。

7.5.4 返回引用的函数

函数的返回值也可以是引用类型，它表示函数的返回值是函数内某一个变量的引用。如例7.3中的max和min函数也可以设计成引用类型，如下所示：

```
int &max(int &a, int &b, int &c)
{
    if (a > b)
        if (a > c) return(a); else return(c);
    else if (b > c) return(b); else return(c);
}

int &min(int &a, int &b, int &c)
{
    if (a < b)
        if (a < c) return(a); else return(c);
    else if (b < c) return(b); else return(c);
}

void swap(int &a, int &b)
{
    int c;
    c = a; a = b; b = c;
}
```

要交换x、y、z中的最大值和最小值，可以调用swap(max(x,y,z),min(x,y,z))。

将函数的返回值说明为一个引用的主要目的是将函数用于赋值运算符的左边。示例见代码清单7-3。

代码清单7-3 返回引用值的函数示例

```
//文件名：7-3.cpp
//返回引用值的函数示例
#include <iostream>
using namespace std;

int a[]={1,3,5,7,9};

int &index(int); //声明返回引用的函数

void main()
{
    index(2)=25; //将a[2]重新赋值为25
    cout<<index(2);
}

int &index(int j)
{
    return a[j]; }
```

由于采用了引用返回，函数调用index(i)是a[i]的别名，可以作为左值。赋值index(2)=25相当于a[2]=25。

注意，在定义返回引用值的函数时，不能返回该函数的局部变量。因为局部变量的生存期限限于函数内部，当函数返回时，局部变量就消失了。此时引用该变量就会引起一个无效的引用。

7.6 指针数组与多级指针

7.6.1 指针数组

由于指针本身也是变量，所以它们也可以像其他变量一样存储在数组中。如果一个数组的元素均为指针，则称该数组为指针数组。一维指针数组的定义形式如下：

类型名 *数组名[数组长度];

例如：

```
char *string[10];
```

定义了一个名为string的指针数组，该数组有10个元素，每个元素都是一个指向字符的指针。

为什么要用到指针数组呢？它比较适合用来存储一组字符串。例如，在5.2节中提到的要在一个城市表中寻找San Francisco，那么遇到的问题首先就是如何存储这个城市表。每个城市名是一个字符串，因此城市表就是一个字符串的数组，而字符串又可以用一个指向字符的指针表示，所以城市表可以用一个指向字符的指针数组表示，如图7-8所示。

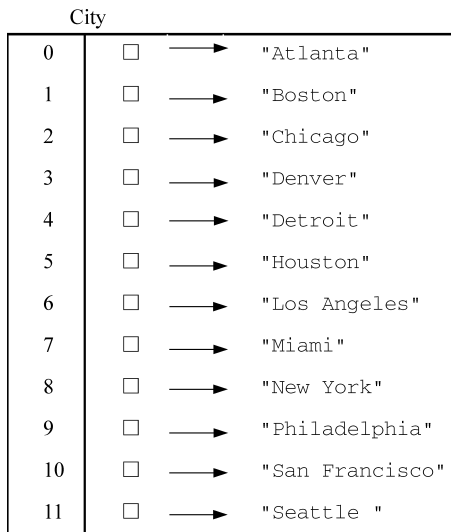


图7-8 城市表的存储

例7.4 写一个函数用二分法查找某一个城市名在城市表中是否出现。要求用递归实现。

按照题意，这个函数有两个参数，即城市表和要查找的城市名，并返回一个整型值，表示所要查找的城市在城市表中的位置。前者是一个字符串的数组；即指针数组；后者是一个字符串，即指向字符的指针。但由于要求用递归实现，在参数中要有表示递归终止的参数，这就是查找的范围，用两个参数表示。因此这个函数有4个参数。函数的实现如下：

```

//该函数用二分查找在cityTable中查找cityName是否出现
//lh和rh表示查找范围
int binarySearch(char *cityTable[], int lh, int rh, char *cityName)
{   int mid, result;   //mid:中间元素的下标值, result:中间元素和cityName的比较结果

    while (lh <= rh) {
        mid =(lh+rh)/2;
        result= strcmp(cityTable[mid], cityName);
        if (result == 0) return mid; //找到
        else if (result > 0) return binarySearch(cityTable, lh, mid-1, cityName);
        //在左一半找
        else return binarySearch(cityTable, mid+1, rh,cityName); //在右一半找
    }
    return -1; //没有找到
}

```

7.6.2 main 函数的参数

指针数组的另一个重要的用途是用在main函数的参数中。如果使用过命令行界面，你会发现输入命令时经常会在命令名后面跟一些参数。例如，DOS中的改变当前目录的命令：

```
cd directory1
```

其中，cd为命令的名字，即对应于改变当前目录命令的可执行文件名，directory1就是这个命令对应的参数。那么，这些参数是怎样传递给可执行文件的呢？每个可执行文件对应的源文件必定有一个main函数，这些参数就是作为main函数的参数传入的。到目前为止，我们设计的main函数都是没有参数的，也没有用到它的返回值。事实上，main函数可以有两个形式参数：第一个参数习惯上称为argc，是一个整型参数，它的值是运行程序时命令行中的参数个数；第二个参数习惯上称为argv，是一个指针数组，它的每个元素是指向一个实际参数的指针。每个实际参数都表示为一个字符串。代码清单7-4中是一个最简单的带参数main函数。

代码清单7-4 带参数的main函数示例

```

//文件名：7-4.cpp
//带参数的main函数示例
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{   int i;

    cout << "argc=" << argc << endl;
    for(i=0; i<argc; ++i) cout << "argv[" << i << "]="<< argv[i] << endl;
    return 0;
}

```

代码清单7-4所示的程序用来检测命令行中有几个实际参数，并把每个实际参数的值打印出来。假如生成的可执行文件为myprogram.exe，在命令行输入myprogram，对应的输出结果如下：

```
argc=1
argv[0]=myprogram
```

注意，在main函数执行时，命令名（可执行文件名）本身也作为一个参数。如果在命令行输入myprogram try this，则对应的输出如下：

```
argc=3
argv[0]=myprogram
argv[1]=try
argv[2]=this
```

例7.5 编写一个求任意 n 个正整数的平均数的程序，它将 n 个数作为命令行的参数。如果该程序对应的可执行文件名为aveg，则可以在命令行输入aveg 10 30 50 20 40↵，对应的输出为30。

这个程序必须用到main函数的参数。通过argc可以知道本次运行输入了多少数字。通过argv可以得到这一组数字。不过这组数字被表示成了字符串的形式，还必须把它转换成真正的数字。程序的实现如代码清单7-5所示。

代码清单7-5 求 n 个正整数的平均值的程序

```
//文件名: 7-5.cpp
//求几个正整数的平均值
#include <iostream>
using namespace std;

int ConvertStringToInt(char *);

int main(int argc, char *argv[])
{
    int sum = 0;
    for (int i = 1; i < argc; ++i) sum += ConvertStringToInt(argv[i]);
    cout << sum / (argc - 1) << endl;
    return 0;
}

//将字符串转换成整型数
int ConvertStringToInt(char *s)
{
    int num = 0;
    while(*s) {num = num * 10 + *s - '0'; ++s;}
    return num;
}
```

7.6.3 多级指针

在下面的定义中，定义了一个元素为指向字符的指针的数组。

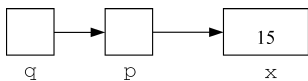
```
char *string[10];
```


在C++中，一维数组的名字string是指向存储数组元素的空间的起始地址，也就是指向数组的第一个元素的指针，而在此数组中每个元素本身又是一个指针，因此string指向了一个存储指针的单元，它是指向指针的指针。

在定义指针变量时，指针变量所指向的变量的类型可以是任意类型。如果一个指针变量所指向的变量的类型是指针类型，则称为多级指针。例如，下面定义中的变量q就是多级指针：

```
int x = 15, *p = &x, **q = &p;
```

变量q指向的内容是一个指向整型的指针，因此它是一个两级指针。上述定义在内存中形成下面的结构：



同理，还可以定义一个三级指针。三级指针的定义如下：

类型名 ***变量名；

普通的数组可以通过指向同类型的指针来访问，同理，指针数组可以用指向指针的指针来访问。如代码清单7-6所示的程序中，用一个指向字符指针的指针访问一个指向字符型的指针数组。

代码清单7-6 用指向指针的指针访问指针数组

```
//文件名：7-6.cpp
//用指向指针的指针访问指针数组
#include <iostream>
using namespace std;

int main()
{
    char **p, *city[] = {"aaa", "bbb", "ccc", "ddd", "eee"};
    for (p = city; p < city + 5; ++p)      cout << *p << endl;
    return 0;
}
```

这段程序的输出结果如下：

```
aaa
bbb
ccc
ddd
eee
```

7.7 多维数组和指向数组的指针

在介绍二维数组时我们讲过，一个二维数组可以看成是一个一维数组的数组。例如，定义

```
int a[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

定义了一个3行4列的矩阵。引用它的第2行第3列的元素可以用`a[1][2]`表示。但从另一个角度来看，也可以看成定义了一个由3个元素组成的一维数组，可以通过`a[0]`、`a[1]`、`a[2]`来引用每个元素。其中每个`a[i]`又是一个由4个元素组成的一维数组，`a[i]`可以看成是数组名，因此`a[i]`又是指向第*i*行第1个元素的指针，是一个指向整型元素的指针。`a[i]+1`指向第*i*行的第2个元素，`a[i]+2`指向第*i*行的第3个元素，……。数组名`a`指向一维数组的第1个元素，即`a[0]`。它们之间的关系如图7-9所示。

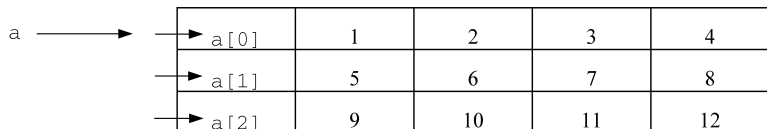


图7-9 二维数组与指针的关系示意图

从一维数组的角度来看，`a`是`a[0]`的地址，也就是第0行的首地址；`a+1`是`a[1]`的地址，也就是第1行的首地址，依次类推。由此可见，对指针`a`加1，事实上是跳到下一行，在本例中即是跳过了4个元素。这样的指针称为指向一维数组的指针。

指向一维数组的指针可以这样定义：

类型名 (*指针变量名) [元素个数];

这个定义中的圆括号不能省略，如果省略了圆括号就变成定义了一个指针数组，而不是一个指针。

二维数组的访问也可以通过指针实现。例如，如果定义

```
int (*p)[4], *q;
```

那么，可以通过以下方法输出数组`a`的所有元素：

```
for (p = a; p < a + 3; ++p){ //让p依次指向每一行
    for (q = *p; q < *p+4; ++q) cout << *q << '\t';
    cout << endl;
}
```

在使用指针访问二维数组时，细心的读者可能会注意到，如果输出`a`和`a[0]`，这两个值是相同的。但是，这两个值的含义是不同的，前者是第0行的首地址，它的类型是指向由4个元素组成的一维数组的首地址，后者是第0行第1个元素的地址，它的类型是整型指针。因此在上述程序的内层循环中，循环控制行的表达式1是`q=*p`，而不是`q=p`。

7.8 指向函数的指针

函数是由指令序列构成的，其代码存储在一块连续空间中。这块空间的起始地址称为函数的入口地址。主函数在调用子函数时，就是让程序转移到函数的入口地址去执行。

在C++中，指针可以指向一个整型变量、实型变量、字符串、数组等，也可以指向一个函数。所谓指针指向一个函数就是让指针指向这个函数的入口地址，以后就可以通过这个指针调用某一

个函数。这样的指针称为指向函数的指针。当通过指针去操作一个函数时，编译器不仅需要知道该指针是指向一个函数的，而且需要知道该函数的原型。因此，在C++中指向函数的指针定义格式如下：

返回类型 (*指针变量)(形式参数表);

注意，指针变量外的圆括号不能省略。如果没有这对圆括号，编译器会认为声明了一个返回指针值的函数，因为函数调用运算符()比表示指针的运算符*的优先级高。例如：

```
int (*p)();
```

定义了一个指向一个没有参数、返回值为int型的函数指针；

```
double (*p)(int);
```

定义了一个指向有一个整型参数、返回值为double型的函数指针。

为了让指向函数的指针指向某一个特定函数，可以通过赋值

指针变量名 = 函数名

来实现。如果有一个函数int f1()，我们可以通过赋值p = f1将f1的入口地址赋给指针p，以后就可以通过p调用f1。例如，p()或(*p)()与f1()都是等价的。

在为指向函数的指针赋值中，所赋的值必须是与指向函数的指针的原型完全一样的函数名。

指向函数的指针的最常见应用是将函数作为另一个函数的参数。例如，在设计一些排序函数（如快速排序）时，我们希望这些函数能排序各种类型的数据。快速排序是基于比较的排序，它通过比较元素之间的大小，将违反排序规则的元素互相交换来达到排序的目的。但是各种数据类型的比较方式是不一样的，例如，系统内置类型可以直接通过关系运算符来比较，而字符串则必须通过strcmp函数来比较，某些用户自定义的类型可能需要特定的比较方法。因此在这些函数中通常设置一个指向函数的指针作为参数，这个函数就是对应于欲排序元素的比较函数。

例7.6 设计一个通用的快速排序函数，可以排序任何类型的数据。

设计这样一个函数需要解决几个问题：如何表示要排序的数据，如何表示不同元素之间的不同比较方法。第二个问题可以用向排序函数传递一个比较函数来解决。该比较函数有两个参数，表示要比较的两个值，返回一个int型的值。当第一个参数比第二个参数大时，返回一个大于0的整数；当第一个参数比第二个参数小时，返回一个小于0的整数；当两个参数相等时，返回0。第一个问题如何解决呢？待排序的数据一般用一个数组表示，但数组是有类型的。怎样才能存储各种不同的类型呢？回忆一下第6章介绍的函数模板。我们可以将快速排序设计成一个函数模板，将待排序的数据类型设计成模板参数。按照这个方法实现的快速排序函数如下：

```
template <class T>
void quicksort(T data[], int low, int high, int (*comp)(T, T))
{
    int mid;
    if (low >= high) return; //待分段的元素只有一个或0个，递归终止
    mid = divide(data, low, high, comp); //low作为基准元素，划分数组，返回中间元素的下标
    quicksort(data, low, mid-1, comp); //排序左一半
    quicksort(data, mid+1, high, comp); //排序右一半
}
```

```

}

template <class T>
int divide(T data[], int low, int high, int (*comp)(T, T))
{
    T k = data[low];
    do {while (low < high && comp(data[high],k)>0) --high;
        if (low < high) {    data[low] = data[high]; ++low;}
        while (low < high && comp(data[low], k) < 0) ++low;
        if (low < high) {    data[high] = data[low]; --high;}
    } while (low != high);
    data[low] = k;
    return low;
}

```

如果需要排序一组字符串，可以直接使用系统提供的字符串比较函数`strcmp`。例如，如果待排序的一组字符串保存在数组`a`中，`a`的定义如下：

```
const char *a[]={"aaa", "nnn", "rrr", "fff", "sss", "ggg", "ddd"};
```

调用

```
quicksort(a, 0, 6, strcmp);
```

可以将数组`a`中的元素排好序。如果要排序一组整型数，则需要定义一个比较函数，如下所示：

```
int intcmp(int a, int b)
{
    if (a == b) return 0;
    if (a < b) return -1; else return 1;
}

```

如果整型数组`a`定义如下：

```
int a[] = {7,9,4,3,8,1,2,5,6,0};
```

调用

```
quicksort(b, 0, 9, intcmp);
```

可以将数组`a`中的元素排好序。

指向函数的指针的第二个应用就是传统的菜单界面的实现。

例7.7 设计一个工资管理系统，要求系统具有如下功能：选择1，添加员工；选择2，删除员工；选择3，修改员工信息；选择4，打印工资单；选择5，打印汇总表；选择0，退出。

在设计中，一般把每个功能设计成一个函数。例如，添加员工的函数为`add`，删除员工的函数为`erase`，修改员工信息的函数为`modify`，打印工资单的函数为`printSalary`，打印汇总表函数为`printReport`。主程序是一个循环，显示所有功能和它的编号，请用户输入编号，根据编号调用相应的函数。具体程序如代码清单7-7所示。

代码清单7-7 实现菜单功能的程序

```

//文件名：7-7.cpp
//菜单功能的实现
int main()

```

```

{   int select;

    while(1) {
        cout << "1--add \n";
        cout << "2--erase\n";
        cout << "3--modify\n";
        cout << "4--print salary\n";
        cout << "5--print report\n";
        cout << "0--quit\n";

        cin >> select;

        switch(select)
        {   case 0: return 0;
            case 1: add(); break;
            case 2: erase(); break;
            case 3: modify(); break;
            case 4: printSalary(); break;
            case 5: printReport(); break;
            default: cout << "input error\n";
        }
    }
}

```

这个程序看上去有点繁琐，特别是当功能很多时，switch语句中会有很多case子句。指向函数的指针可以帮助我们解决这个问题。我们可以定义一个指向函数的指针数组，数组的每个元素指向一个函数。元素1指向处理功能1的函数，元素2指向处理功能2的函数，以此类推。这样，当接收到用户的一个选择后，只要输入是正确的，直接就可以通过相应的数组元素访问该函数。具体程序如代码清单7-8所示。

代码清单7-8 用函数指针实现菜单选择的程序

```

//文件名: 7-8.cpp
//用函数指针实现菜单选择
int main()
{   int select;
    void (*func[6])() = {NULL, add, erase, modify, printSalary, printReport};

    while(1) {
        cout << "1--add \n";
        cout << "2--delete\n";
        cout << "3--modify\n";
        cout << "4--print salary\n";
        cout << "5--print report\n";
        cout << "0--quit\n";

        cin >> select;

        if (select == 0) return 0;
    }
}

```

```
    if (select > 5) cout << "input error\n"; else func[select]();  
    }  
}
```

小结

本章介绍了指针的概念。指针是一种特殊的变量，它的值是计算机内存中一个地址。指针通常是很有用的，因为它允许紧凑地表示大型数据结构，提供程序不同部分间共享数据的功能，能使程序在执行时分配新内存。

像其他变量一样，指针变量使用前必须先定义。定义一个指针变量时，除了说明它里面存储的是一个地址外，还要说明它指向的变量的数据类型。定义指针变量时给它赋初值是一个很好的习惯，可以避免很多不可预料的错误。

指针的基本运算符是&和*。&运算符作用到一个左值并返回一个指向该左值的指针，*运算符作用到一个指针并返回该指针指向的左值。

将指针作为形式参数可以使一个函数与其调用函数共享数据。指针传递的另一种形式是引用传递，它能起到指针传递的作用，但形式更加简洁。指针或引用也可以作为函数的返回值，允许将函数作为左值使用。

在C++语言中，指针和数组密切相关。可以将指针像数组一样使用，反之也行。指针和数组之间的关系使得算术运算符+和-以及++和--也可用于指针。

程序运行时，你可以向一个叫堆的未用内存区中动态地申请新内存，这称为动态内存分配机制。运算符new用于申请动态变量。当申请的动态变量不再需要时，程序必须执行delete运算把内存归还给堆。

习题

简答题

1. 下面的定义所定义的变量类型是什么？

```
double *p1, p2;
```

2. 如果arr被定义为一个数组，描述以下两个表达式之间的区别。

```
arr[2]  
arr+2
```

3. 假设double类型的变量在你使用的计算机系统中占用8个字节。如果数组doubleArray基地址为1000，那么doubleArray+5的地址值是什么？
4. 定义int array[10],*p = array;后，可以用p[i]访问array[i]。这是否意味着数组和指针是等同的？

5. 字符串是用字符数组来存储的。为什么传递一个数组需要两个参数（数组名和数组长度），而传递字符串只要一个参数（字符数组名）？

程序设计题

1. 设计一组字符串处理函数，用动态内存分配的方法实现常用的字符串的操作，包括字符串复制、字符串拼接、字符串比较、求字符串长度和取字符串的子串。作为测试，用这些函数实现文本的双向对齐。
2. 用带参数的main函数实现一个完成整数运算的计算器。例如，输入
`calc 5 * 3`
执行结果为15。
3. 编写一个函数，判断作为参数传入的一个整型数组是否为回文。例如，若数组元素值为10、5、30、67、30、5、10就是一个回文。

在第5章学习数组的时候，我们已经初步理解了计算机程序设计的一个重要理念，即利用聚合型的数据结构表示复杂的信息。在程序中定义一个数组，即可将任意多的数据聚集起来构成一个概念上的整体。需要时可以从数组中选出元素，并单独进行操作，也可以将它们作为一个整体，同时进行操作。

现代程序设计语言的一大特点是能够将各自独立的数据或操作组织成为一个整体。过程和函数可以将一组操作封装成一个整体。在数据处理方面，数组可以将类型相同的一组有序数据封装成一个整体。此外，我们同样需要能够将一组无序的、异质的数据看作一个整体进行操作。在程序设计语言中，这样的一组数据被称为记录。在C++语言中，称为结构体（或结构）。

8.1 记录的概念

为了更好地理解记录的概念，再回顾一下在第5章讲过的打印学生成绩单的例子。我们打印出的成绩单非常简陋，每位同学是用了一个编号来表示的，而且只打印了一门课程的成绩。假设要为每位学生在期末打印一张较为全面的成绩单，包括学号、姓名、各门功课的成绩。如果这学期学的课程有语文、数学和英语，那么相关数据可表示成表8-1所示的形式。

表8-1 学生成绩单示例

学 号	姓 名	语 文 成 绩	数 学 成 绩	英 语 成 绩
00001	张三	96	94	88
00003	李四	89	70	76
00004	王五	90	87	78

每个学生的相关数据（表8-1中的一行）称为一条记录。从表中看出，每条记录都由若干个部分组成，每一部分提供了关于学生某个方面的信息，所有部分组合起来形成了一条完整的学生信息。每个组成部分通常被称为字段，或者被称为成员。譬如，在上例中，对于学生记录，它由5个字段组成：学号和姓名字段是字符串，而3个成绩字段都是整型数。

虽然记录是由若干个单独的字段组成，但它表达了一个整体的含义。在学生信息的例子中，表8-1中第一行的各个字段的内容组成了一组逻辑上一致的数据集合，第一行提供了张三的信息，第二行提供了李四的信息，第三行提供了王五的信息。

问题是如何在程序中保存这些信息呢？由于组成记录的各个字段数据类型都不同，所以数组不能胜任此项工作。在这种情况下，就需要将每个学生的信息定义为一条记录。

真实世界中的信息往往由多个部分组成，对于这样的信息应该用聚集型的数据结构进行处理。如果各个部分是有序的、同类型的，则应该使用数组；如果各个部分是无序的，即使类型相同也需要使用记录。

8.2 C++语言中记录的使用

C++中的记录称为结构体。在C++中使用记录需要进行如下两个步骤：

(1) 定义一个新的结构体类型。记录是一个统称，此记录不同于彼记录，每个记录都有自己的组成内容。如学生记录，可能由学号、姓名、班级、各门课程的成绩组成，而教师信息也必须用一个记录来描述，它的组成部分可能有工号、部门、职称、专业和工资。因此在定义结构体变量前，需要先定义一个新的结构体类型。类型定义指明了该结构体类型的所有变量是由哪些字段组成的，并指明字段的名称以及字段中信息的类型。结构体类型的定义为所有对象定义了一个模式，但并不分配任何存储空间。

(2) 定义新类型的变量。完成了结构体类型的定义后，可以定义该类型的变量。一旦定义了该类型的变量，编译器会参照相应的结构体类型定义分配相应的空间，此时才可将数据存入其中。

这两个步骤是完全不同的操作。刚刚接触程序设计的人通常忘记这两步都是必不可少的。在定义了一个新类型后，他们会认为该类型就是一个变量了，并在程序中使用。然而结构体类型只为定义其他变量提供了一个模板，它本身没有存储空间。

8.2.1 结构体类型的定义

结构体类型定义的格式如下：

```
struct 结构体类型名{  
    字段声明;  
};
```

其中，struct是C++用户定义结构体类型的保留字，字段声明指出了组成该结构体类型的每个字段的类型。例如，存储上述学生信息的结构体类型的定义如下：

```
struct studentT {  
    char no[10];  
    char name[10];  
    int chinese;  
    int math;  
    int english;  
};
```

在定义结构体类型时，字段名可与程序中的变量名相同。在不同的结构体中可以有相同的字段名，而不会发生混淆。例如，定义一个结构体类型studentT，它有一个字段是name，在同一个程序中，还可以定义另一个结构体类型teacherT，它也有一个字段叫name。

结构体成员的类型可以是任意类型，可以是整型、实型，也可以是数组，当然也可以是其他结构体类型。如果在studentT类型中，我们希望增加一个生日字段，日期可以用年、月、日3个部分表示，那么日期也可以被定义成一个结构体：

```
struct dateT{
    int month;
    int day;
    int year;
};
```

在studentT中，可以有一个成员，它的类型是dateT：

```
struct studentT {
    char no[10];
    char name[10];
    dateT birthday;
    int chinese;
    int math;
    int english;
};
```

8.2.2 结构体类型的变量的定义


定义了一个新的结构体类型后，就可以定义该类型的变量了。例如，有了studentT这个类型，就可以通过下列代码定义该类型的一个变量student1：

```
studentT student1;
```

一旦定义了这样一个结构体变量，就可以从两个角度来看待它。如果从整体的角度来看待，即在概念上从一个较远的距离来观察，则得到像下面这样的箱子student1：

student1 

但如果从近处看细节，会发现标签为student1的箱子内部还有着5个单独的箱子：

student1 

事实上，一旦定义了一个结构体类型的变量，系统在分配内存时就会分配一块连续的空间，依次存放它的每一个分量。这块空间总的名字就是结构体变量的名字，每一小块还有自己的名字，即字段名。

结构体变量和其他类型的变量一样，也可以在定义时为它赋初值。但是结构体变量的初值不是一个，而是一组。C++用一对花括号将这一组值括起来，表示一个整体。例如：

```
studentT student1 = {"00001", "张三", 89, 96, 77 };
```

结构体变量也可以在定义结构体类型的同时定义，格式如下：

```
struct 结构体类型名 {
    字段声明;
} 结构体变量;
```

或

```
struct {
    字段声明;
}    结构体变量;
```

这两种方法的区别在于前者可以继续用结构体类型名定义变量，后者却不能。

8.2.3 结构体变量的使用

结构体类型是一个统称，程序员所用的每个结构体类型都是根据需求自己定义的，在C++系统设计时无法预知程序员会定义什么样的结构体类型，因此，除了赋值操作之外，C++无法对结构体变量作整体操作。结构体变量的访问主要是访问它的某一个字段。例如，对结构体变量的赋值是通过对它的每一个字段的赋值来实现的，结构体变量的输出也是通过输出它的每一个字段的值来实现的。当通过语句

```
studentT student1;
```

定义了变量student1后，就可以用student1表示该记录的整体，但如果想了解student1的某个方面的内容，如他的语文成绩等，就可以打开该结构体变量并对它的字段进行单独的操作。要表示结构体变量中的某一个字段，需要写下整个结构体变量的名称，后面跟符号“.”以及该字段的名称。因此，为表示student1的语文成绩，可以写成：

```
student1.chinese
```

结构中的字段可以单独使用，它相当于普通变量。

如果结构体变量的成员还是一个结构体，则可以一级一级用“.”分开，逐级访问。例如：

```
student1.birthday.year
```

结构体变量的赋值通常是通过对它的每一个成员的赋值来实现的。例如，输入student1的内容可用

```
cin >> student1.no >> student1.name
    >> student1.chinese >> student1.math
    >> student1.english
    >> student1.birthday.year
    >> student1.birthday.month
    >> student1.birthday.day;
```

也可以通过对每一个字段赋值来实现结构体的赋值，如student1.birthday.year = 1990。

当两个结构体变量属于同一个结构体类型时，可以互相赋值。例如，如果student1和student2都是studentT类型的变量，则可以用student1= student2将student2的每个字段的值对应地赋给student1的各个字段。如果两个结构体变量属于不同的结构体类型时，则不能互相赋值。从上述讨论中也可以看出数组和结构体的另一个区别，那就是数组不是左值，而结构体是左值。

在很多情况下，即使组成一个对象的所有分量都是同样的类型，但为了表示这是一个有机的整体，我们还是把它定义成一个结构体。如二维平面上的一个点，它是由两个坐标组成。两个坐标都是double类型的。但要定义一个二维平面上的点类型，我们还是把它定义成一个结构体：

```
struct pointT{  
    double x, y;  
};
```

数组通常表示一组对象，而结构体通常表示的是一个对象。

和普通变量一样，结构体除了可以通过变量名直接访问外也可以通过指针间接访问。指向结构体的指针定义与普通指针定义一样：

结构体类型名 *指针变量名；

也可以在定义结构体类型时直接定义指针变量：

```
struct 结构体类型名{  
    字段声明;  
} *结构体指针;
```

指向结构体的指针可以指向一个已有的结构体变量，也可以指向一个通过动态内存申请到的一块用于存储结构体的空间。例如：

```
studentT student1, *sp = &student1;  
sp = new studentT;
```

都是正确的赋值。结构体指针可以像其他指针一样引用，例如，要引用`sp`指向的结构体对象的`name`值，可以表示为

`(*sp).name`

注意，括号是必需的。因为点运算符的优先级比`*`运算符高，如果不加括号，编译器会理解成`sp.name`是一个指针，然后取该指针指向的内容。

这种表示方法显得太过笨拙。指向结构体的指针随时都在使用，使用者在每次选取时都使用括号会使结构体指针的使用变得很麻烦。为此，C++提出了另外一个更加简洁明了的运算符`->`。它的用法如下：

指针变量名`->`字段名

表示取指针变量指向的结构体的指定字段值。例如，取`sp`指向的结构体中的`name`字段，可表示为`sp->name`。C++的程序员一般都习惯使用这种表示方法。

8.2.4 结构体数组

现代程序设计语言的一大特点就是：如果拥有了一个类型，就可以利用它来创建更复杂的新类型。为了达到这一目的，通常可以定义结构体数组，或者包含数组的结构体。同理，这样的新类型同样可以用来创建更加复杂的类型。

例如，在8.1节中定义了一个名为`studentT`的类型。一旦定义了这个类型，就可以定义这个类型的数组。例如：

```
int scores[10];
```

说明了`scores`是具有10个整型值的数组，而

```
studentT studentArray[10];
```

就说明了studentArray是具有10个studentT型值的数组。它可以有如下的内容：

00001	张三	96	94	88
00003	李四	89	70	76
⋮	⋮	⋮	⋮	⋮

选取studentArray的任意元素就像从其他数组中选取元素一样。例如，可以用studentArray[1]来选取李四的整个记录：

00003	李四	89	70	76
-------	----	----	----	----

对于这条记录，可以进一步选取其中的字段，例如，studentArray[1].name就选取了李四的名字：

李四

选取名字字段可能已经满足了应用的需要。但同时也可以进一步进行选取。name字段的值是一个字符串，因此可以继续对其中的元素进行选取。如果想知道studentArray数组中第一个学生的名字首字母是什么，就可以用下面的语句：

```
studentArray [1].name[0];
```

在处理复杂数据结构的程序中，通常可以发现一长串的选取操作，从记录中选取字段，或从数组中选取元素。

在结构体数组中，由于数组的每个分量是相同类型的结构体，因此数组成员之间可以互相赋值。

例8.1 设计一个打印通讯录的程序，该通讯录最多包含100条信息。通讯录包含的内容有姓名、性别、地址、电话号码。信息从键盘输入，直到输入的姓名为@为止。

通讯录中的每条信息是一个有机的整体，可以把它设计为一个结构体。整个程序由两大部分组成：输入信息和输出信息。输入的信息可以存放在一个结构体数组中，然后打印该数组。整个程序如代码清单8-1所示。

代码清单8-1 打印通讯录的程序

```
//文件名：8-1.cpp
//打印通讯录
#include <iostream>
#include <iomanip>
using namespace std;

struct personT
{
    char name[10];
    char sex;
    char addr[30];
    int phonenumber;
};
```

```

const int MAX = 100;

int main()
{
    personT p[MAX];
    int i, num = 0;

    cout << "姓名 性别 地址 电话 ( @表示结束): \n";
    while (num < 100) {
        cin >> p[num].name;
        if (p[num].name[0]=='@' ) break;
        cin >> p[num].sex >> p[num].addr >> p[num].phonenum;
        ++num;
    }

    cout << " Name " << " Sex\t\t" << "Addr\t\t\t" << "PhoneNum\n";
    cout << setiosflags(ios::left);
    for (i=0; i<num; ++i)
        cout << setw(10) << p[i].name << p[i].sex << '\t' << setw(30)
            << p[i].addr << '\t' << p[i].phonenum << endl;
    return 0;
}

```

代码清单8-1所示的程序中的setioflags和setw等都是用于控制输出格式的: setiosflags(ios::left)表示输出信息左对齐, setw(10)表示后面输出的信息占10个空格的位置。格式化输入/输出将在第14章详细介绍。

结构体数组与普通数组一样,也可以在定义时赋初值。由于每个数组成员是一个结构体变量,有一组对应的值,为了表示清晰,可将每个数组成员的值用一对花括号括起来。例如:

```

studentT studentArray[5] = { {"00001", "张三", 80, 90, 98 }, {...}, {...}, {...}};

```

与普通的数组一样,指向结构体的指针也能够用来指向一个结构体数组。此时,对指针加1就是加了该结构体的大小,使指针指向数组的下一个成员。例如,在例8.1中,通讯录的输出也可以用指针实现。例如,如果pp是指向结构体personT的指针,则下列语句可以正确输出通讯录的内容:

```

for (pp=p; pp<p+num; ++pp)
    cout << setw(10) << pp->name << pp->sex << '\t' << setw(30)
        << pp->addr << '\t' << pp->phonenum << endl;

```

8.3 结构体作为函数的参数

当要把一个结构体传给函数时,形式参数和实际参数应具有相同的结构体类型。尽管结构体和数组一样也是由许多分量组成的,但结构体的传递和普通内置类型一样都是值传递。当调用一个形式参数是结构体的函数时,首先会为作为形式参数的结构体分配空间,然后将作为实际参数的结构体中的每个分量复制到形式参数的每个分量中,以后实际参数和形式参数就没有任何关系了。

由于结构体占用的内存量一般都比较大大,值传递既浪费空间又浪费时间,因此常用指针传递

或引用传递。但指针传递形式比较繁琐，所以C++通常用引用传递。

指针传递和引用传递虽然能提高函数调用的效率，但也带来了安全隐患。因为在指针传递和引用传递中，形式参数和实际参数共享了同一块空间。因此，对形式参数的修改也就是对实际参数的修改，这违背了值传递的规则。要解决这个问题，使在函数中不能修改实际参数，可以在此形式参数前加const限定符。这样就表示此形式参数是一个常量，在函数中只能引用，不能修改。这样可以用引用传递达到值传递的目的。

例8.2 某应用经常用到二维平面上的点。点的常用操作包括设置点的位置，获取点的x坐标，获取点的y坐标，显示点的位置，计算两个点的距离。试定义点类型，并实现这些函数。

平面上的点可以由两个坐标表示，因此点类型可定义如下：

```
struct pointT {
    double x, y;
};
```

点操作函数的实现也非常简单。有了这些函数，就可以对点进行操作。这些函数及其应用如代码清单8-2所示。

代码清单8-2 点操作函数的应用

```
//文件名：8-2.cpp
//对平面上点的操作的函数及应用
struct pointT{
double x,y;
};
int main()
{   pointT p1, p2;

    p1 = setPoint(1,1);
    p2 = setPoint(2,2);

    cout << getX(p1) << " " << getY(p2) << endl;
    showPoint(p1);
    cout << " -> " ;
    showPoint(p2);
    cout << " = " << distancePoint(p1, p2) << endl;

    return 0;
}
pointT setPoint(double x, double y)
{   pointT p;

    p.x = x;
    p.y = y;

    return (p);
}

double getX(const pointT &p)
{
    return (p.x);
}
```

```

    }

    double getY(const pointT &p)
    {
        return (p.y);
    }

    void showPoint(const pointT &p)
    {
        cout << "(" << p.x << " , " << p.y << ")";
    }

    double distancePoint(const pointT &p1, const pointT &p2)
    {
        return sqrt((p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y) * (p1.y-p2.y));
    }

```

程序的运行结果如下：

```

1 2
(1, 1)->(2, 2)= 1.41421

```

注意代码清单8-2中的setPoint函数，这个函数的返回类型是PointT类型，这种函数称为返回结构体的函数。它返回的是一个结构体的复制，如代码清单8-2中的调用p1 = setPoint(1,1)；相当于把setPoint函数中的局部变量p的值赋给主调函数中的p1。由于局部变量在函数执行结束时就会消亡，因此在setPoint函数返回前，系统会创建一个临时变量，把p的值赋给该临时变量，然后回收setPoint函数的帧，再把临时变量的值赋给p1。

虽然p是一个局部变量，当setPoint执行结束时该变量就不存在了，但由于是复制，在该变量消失之前已经它的值赋给了p1，因此没有问题。还有一种函数可以返回一个结构体的引用，此时必须注意函数中返回的结构体不能是局部变量。

8.4 链表

8.4.1 链表的概念

链表是一种常用的数据结构，通常用来存储一组数据。第5章已经介绍了用于存储和处理批量数据的工具——数组。但数组在使用时必须事先确定元素的个数，在某些元素个数不确定的场合，特别是个数变化很大的时候，数组较难满足用户的要求。此时，链表是一个很好的替代方案。

链表是一种可以动态地进行内存分配的结构。当需要新增加一个成员时，可以动态地为它申请存储空间，做到按需分配。但是这样就无法保证所有的成员是连续存储的，如何从当前的成员找到它的下一个成员呢？链表提供了一条“链”，这就是指向下一个结点的指针。图8-1给出了最简单的链表——单链表的结构。



图8-1 单链表

图8-1中的每个结点存储一个元素。每个结点由两个部分组成：真正存储数据元素的部分和存储下一结点地址的部分。变量head中存放着存储第一个元素的结点的地址，从head可以找到第一个结点，第一个结点中存放着第二个结点的地址，因此从第一个结点可以找到第二个结点。依次类推，可以找到第三个、第四个结点，一直到最后一个结点。最后一个结点的第二部分存放一个空指针，表示其后没有元素了。因此，链表就像一根链条一样，一环扣一环，直到最后一环。抓住了链条的第一环，就能找到所有的环。

链表有各种形式。除了单链表外，还有双链表、循环链表等。双链表的每一个结点不仅记住后一结点的地址，还记住前一结点的地址。双链表如图8-2所示。循环链表有单循环链表和双循环链表。单循环链表类似于单链表，只是最后一个结点的下一结点是第一个结点，使整条链形成了一个环，如图8-3所示。在双循环链表中，最后一个结点的下一结点是第一个结点，第一个结点的前一结点是最后一个结点。

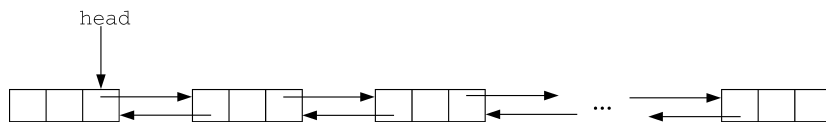


图8-2 双链表

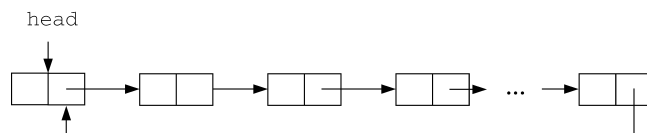


图8-3 单循环链表

在实际应用中，可以根据应用所需的操作选择相应的链表。如果在应用中，常做的操作是找后一结点，而几乎不找前一结点，则可选择单链表。如果既要找前一结点也要找后一结点，则可选择双链表。

本书只给读者介绍最简单的链表，即单链表，那些复杂的链表将在数据结构课程中有详细的讨论。

8.4.2 单链表的存储

在一个单链表中，每个数据元素被存放在一个结点中。每个结点由两部分组成：数据元素本身和指向下一结点的指针。描述这样一个结点的最合适的数据类型就是结构体。我们可以把结点定义为

```
struct linkNode {  
    datatype data;  
    linkNode *next;  
};
```

这里datatype表示任意一种数据类型。第二个成员是指向自身类型的一个指针，这种结构称为自引用结构。如果单链表的每个数据元素是一个整型数，那么该单链表的结点可定义为

```
struct linkNode {  
    int data;  
    linkNode *next;  
};
```

如果单链表的每个数据元素是一个长度为20的字符串，那么该单链表的结点可定义为

```
struct linkNode {  
    char data[20];  
    linkNode *next;  
};
```

单链表的数据元素可以是另一个结构体，如personT类型的变量，则该单链表的结点可定义为

```
struct personNode {  
    personT person;  
    personNode *next;  
};
```

或定义为

```
struct personNode{  
    char name[10];  
    char sex;  
    char addr[30];  
    int phonenum;  
    personNode *next;  
};
```

由于单链表的结点是环环相扣的，保存一个单链表只需要知道单链表的第一个结点的地址。因此，保存一个单链表只需要一个指向第一个结点的指针。

8.4.3 单链表的操作

单链表最基本的操作包括创建一个单链表，插入一个结点，删除一个结点，以及访问单链表的每一个结点。

先看一看插入操作。对于插入操作来说必须指明在哪个结点后插入一个结点。图8-4展示了如何在链表中的结点p后面插入另一个元素x。插入一个结点必须完成下列几个步骤：申请一个结点，将x放入该结点的数据部分，然后将该结点链接到结点p后面。例如，如果结点类型为ListNode，这一连串工作可以用下面几条语句来实现：

```

tmp = new ListNode;    //创建一个新结点
tmp->data = x;          //把x放入新结点的数据成员中
tmp->next = p->next;    //把新结点和p的下一成员相连
p->next = tmp;         //把p和新结点连接起来

```

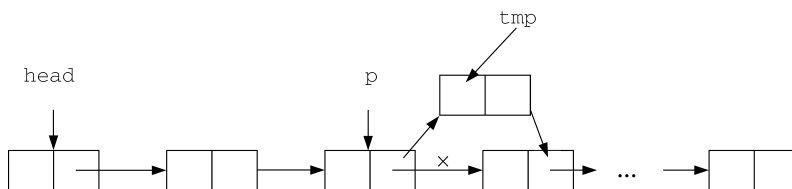


图8-4 单链表的插入

删除命令只需修改一条链即可，图8-5显示了如何在链表中删除项x。将p设置为x前面的那个结点，然后让p的next链绕过x就可以了。这个操作可以由下面的语句来完成：

```
p->next = p->next->next;
```

这条语句确实在单链表中实现了删除，但它有一个严重的问题，那就是内存泄漏，没有回收被删结点的空间。完整的删除应该有两个工作：从链表中删去该结点，回收结点的空间。因此删除操作需要3条语句：

```

delPtr = p->next;      //保存被删结点的地址
p->next=delPtr->next;   //将此结点从链中删除
delete delPtr;         //回收被删结点的空间

```

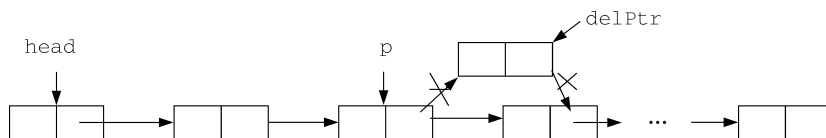


图8-5 单链表的删除

在上述的基本描述中有一个问题：它假定了不论被删除的x在什么位置，在它前面总是有一个项，这样才允许绕过去的操作。而链表中的第一个结点前面是没有结点的，因此删除链表中的第一项就成了一种特殊情况，必须特殊处理。相应地，插入方法也不允许将一个项插入为链表中新首项。原因也在于插入必须是在某个已存在项之后。所以，尽管基本算法工作得很好，但是还有很多令人烦恼的特殊情况要单独处理。

特殊情况经常给算法设计带来很多麻烦，也经常造成代码的漏洞。因此，通常采用避免特殊情况的方法。对于上述问题的一种解决方案就是引入一个头结点。

头结点（header node）是链表中额外加入的一个特殊结点，它不存放数据，只是为了保证链表中每个元素都有一个前驱，如图8-6所示。注意，现在就不再有特殊情况了。首结点也能和其他结点一样通过让p指向它前面的元素来删除它。同样，只要设p等于头结点并调用插入方法，也就可以把一个元素当作链表的新的首元素插入到链表中。使用了头结点，我们只使用一

个额外的结点就使代码大大简化了。在更复杂的应用中，使用头结点不仅能简化代码，而且还会提高速度，这是因为不需要再判别是否是对链表的首结点进行操作，而减少测试量就是节约时间。



图8-6 带头结点的单链表

例8.3 创建并访问一个带头结点的、存储整型数据的单链表，数据从键盘输入，0为输入结束标志。

首先，确定单链表的结点类型。该结点可定义为

```
struct linkRec {
    int data;
    linkRec *next;
};
```

创建一个单链表由如下步骤组成：定义一个单链表，创建一个空的单链表，依次从键盘读入数据链入单链表的表尾。在程序中使用一个单链表只需要定义一个指向头结点的指针。该指针被命名为head。创建一个空的单链表需要申请一个结点，该结点的数据字段不存储有效数据，让head指向该结点。依次从键盘读入数据链入单链表的表尾可由一个循环组成。该循环首先从键盘读入一个整型数，如果读入的数不是0，则申请一个结点，将读入的数放入该结点，将结点链到链表的尾。如果读入的是0，设置最后一个结点的next为NULL，链表创建完成。

读链表的工作比较简单，就像我们数链条上有多少环一样，从第一个环开始，依次找到第二个环，第三个环，……。在此过程中用到了一个重要的工具，那就是我们的手，我们的手总是抓住当前数到的环。在链表的访问中，这个工具就是指向链表结点的指针。该指针首先指向表的第一个结点，看看该结点是否存在。如果存在，则访问此结点，然后让指针移到下一个结点。如果不存在，则访问结束。

实现程序如代码清单8-3所示。

代码清单8-3 单链表的建立与访问

```
//文件名：8-3.cpp
//单链表的建立与访问
#include <iostream>
using namespace std;

struct linkRec {
    int data;
    linkRec *next;
};

int main()
```

```

{   int x; //存放输入的值
    linkRec *head, *p, *rear; //head为表的头指针, rear指向创建链表时的表尾结点,
                                //p是创建和读链表时指向被操作结点的指针

    head = rear = new linkRec; //创建空链表, 头结点也是最后一个结点

    //创建链表的其他结点
    while (true) {
        cin >> x;
        if (x == 0) break;
        p = new linkRec; //申请一个结点
        p->data = x; //将x的值存入新结点
        rear->next = p; //将p链到表尾
        rear = p; //p作为新的表尾
    }

    rear->next = NULL; //设置rear为表尾, 其后没有结点了

    //读链表
    cout << "链表的内容为: \n";
    p = head->next;
    while (p != NULL) {
        cout << p->data << '\t';
        p = p->next; //使p指向下一个结点
    }
    cout << endl;

    return 0;
}

```

例8.4 约瑟夫环问题： n 个人围成一圈，从第一个人开始报数1、2、3，凡报到3者退出圈子。找出最后留在圈子中的人的序号。如果将 n 个人用0到 $n-1$ 编号时，则当 $n=5$ 时，最后剩下的是编号为3的人。试编写一个用单链表解决的约瑟夫环的程序。

首先要考虑的是如何表示 n 个人围成一圈。 n 个人围成一圈意味着0号后面是1号，1号后面是2号，……， $n-1$ 号后面是0号。这正好用一个单循环链表表示，而且该单循环链表不需要头结点。当 $n=5$ 时，该循环链表如图8-7所示。

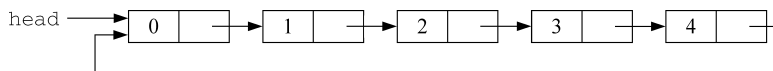


图8-7 约瑟夫环的存储

解决约瑟夫环的问题分成两个阶段：首先根据 n 创建一个循环链表，然后模拟报数过程，逐个删除结点，直到只剩下一个结点为止。创建一个循环链表和例8.3中创建一个单链表基本类似，只有两个小区别：一是约瑟夫环不需要头结点，有了头结点反而会增加报数阶段的处理的复杂性；二是最后一个结点的next指针不再为NULL，而是指向第一个结点。报数阶段本质上是结点的删除，报到3的结点从环上删除。报数的过程是指针移动的过程，让指针停留在被删结点的前一结

点。解决约瑟夫环问题的程序如代码清单8-4所示。

代码清单8-4 求解约瑟夫环问题的程序

```
//文件名: 8-4.cpp
//求解约瑟夫环问题
#include <iostream>
using namespace std;

struct node{
    int data;
    node *next;
};

int main()
{
    node *head, *p, *q; //head为链表头
    int n, i;

    //输入n
    cout << "\ninput n:";    cin >> n;

    //建立链表
    head = p = new node; //创建第一个结点, head指向表头, p指向表尾
    p->data = 0;
    for (i=1; i<n; ++i) {
        q = new node; //q为当前正在创建的结点
        q->data = i;
        p->next = q;  p = q; //将q链入表尾
    }
    p->next = head; // 头尾相连

    // 删除过程
    q=head; //head报数为1
    while (q->next != q) { //只要表非空
        p = q->next;  q = p->next; //p报数为2,  q报数为3
        //删除q
        p->next = q->next; //绕过结点q
        cout << q->data << '\t';    //显示被删者的编号
        delete q; //回收被删者的空间
        q=p->next; //让q指向报1的结点
    }

    // 打印结果
    cout << "\n最后剩下:  " <<  q->data << endl;

    return 0;
}
```

当环中有5个人时, 执行代码清单8-4所示的程序的结果如下:

```
input n: 5
```

2 0 4 1
最后剩下的是：3

小结

在编程过程中，经常会遇到一些复杂的对象，它们不能用一个简单的内置类型来描述，而是需要由一组不同的属性来描述。本章提供了这样的一个工具，即结构体。结构体可以将一组无序的、异质的数据存储在一个聚集类型中，用来描述一个对象。数组通常模拟现实世界中对象的集合，而结构体则用来模拟一个复杂的对象。同时，可以将数组和结构体结合起来，表示任意复杂度的数据集合。

本章的重点为以下几点。

- 一个结构体是由多个组成部分组成的，这些组成部分称为字段。
- 定义一个结构体分为两个步骤：首先要进行类型定义，为结构体提供模板；然后定义该类型的变量。
- 给定一个结构体变量，可以使用点操作符来选取其中的字段。
- 很多情况下，使用结构体指针比结构体本身更合理。使用指向结构体的指针时，可以用->操作符选取字段。
- 结构体的一个重要应用是链表。链表用来存储一组具有线性关系的元素，适合于存储元素个数变化较大的集合，而且它的插入和删除都比较方便。

习题

简答题

1. 判断：数组中的每个元素类型都相同。
2. 判断：结构体中每个字段的类型都必不相同。
3. 定义一个结构体变量的必要步骤是什么？
4. 从结构体中选取某个字段使用哪个运算符？
5. 判断：C++语言中结构体本身是左值。
6. 如果变量p被定义为一个结构体指针，结构体中包括字段cost，要通过指针p选取cost字段时，表达式*p.cost有何错误？在C++语言中完成该操作应该用什么表达式？

程序设计题

1. 用结构体表示一个复数，编写实现复数的加法、乘法、输入和输出的函数，并测试这些函数。
 - 加法规则： $(a+bi)+(c+di)=(a+c)+(b+d)i$ 。
 - 乘法规则： $(a+bi)\times(c+di)=(ac-bd)+(bc+ad)i$ 。
 - 输入规则：分别输入实部和虚部。

- 输出规则：如果a是实部，b是虚部，输出格式为a+bi。

2. 编写函数Midpoint(p1, p2)，返回线段p1、p2的中点。函数的参数及结果都应该为pointT类型，pointT的定义如下：

```
struct pointT {  
    double x, y;  
};
```

3. 可以用两个整数的商表示的数称为有理数。因此1.25是一个有理数，它等于5除以4。很多数不是有理数，比如 π 和2的平方根。在计算时，使用有理数比使用浮点数更有优势：有理数是精确的，而浮点数不是。因此，设计一个专门处理有理数的工具是非常有用的。

- 试定义类型rationalT，用来表示一个有理数。
- 函数CreateRational(num, den)，返回一个rationalT类型的值。
- 函数AddRational(r1, r2)，返回两个有理数的和。有理数的和可以用下面的方程来表示：

$$\frac{num1}{den1} + \frac{num2}{den2} = \frac{num1 \times den2 + num2 \times den1}{den1 \times den2}$$

- 函数MultiplyRational(r1, r2)，返回两个有理数的乘积。乘积可以用下面的方程来表示：

$$\frac{num1}{den1} \times \frac{num2}{den2} = \frac{num1 \times num2}{den1 \times den2}$$

- 函数GetRational(r)，返回有理数r的实型表示。
- 函数PrintRational(r)，以分数的形式将数值显示在屏幕上。

关于有理数的所有计算结果都应化到最简形式，例如，1/2乘以2/3的结果应该是1/3而不是2/6。



回顾一下第6章介绍的函数。使用函数可以把大的计算任务分解成若干个小的计算任务，程序设计人员可以基于函数进一步构建更大规模的程序。一个设计良好的函数可以把程序中不需要让人了解的具体操作细节隐藏起来，使整个程序的结构更加清晰，降低程序维护的难度。

到目前为止，已经介绍的大部分程序都是一些短小的程序，用来说明C++语言中的某些功能的使用。集中介绍单个语句格式或者其他一些语言细节对于初学编程来说是很有意义的，这样能使其专注于理解每个概念，了解它是如何工作的，而不必涉及大段程序中的内在复杂性。但是，真正具有挑战性的是如何掌控这种复杂性。为了达到这个目的，就必须写一些大的程序完成一个较大的任务，在这些程序中，将一些概念和工具结合起来。

处理大程序的最重要的技术是逐步细化策略：当遇到一个较大的、复杂的问题时，可以把问题分解成更为容易解决的几个小部分；如果某个小部分还比较复杂，则再用同样的方法进一步分解。逐步细化的实现工具就是函数，每一个小部分用一个函数来表示。

9.1 自顶向下分解

使用函数可以将一个大的程序设计问题分解为较小的部分，每个较小的部分都是比较容易理解的。将一个问题划分为可管理的分块的过程被称为分解，这是程序设计最基本的策略。然而，找出正确的分解是一项很困难的任务，很好地完成该任务需要相当多的实践。如果分块很合理，会使程序作为一个整体比较容易理解。然而，如果分块不合理，会妨碍分解的结束。虽然本章和后面几章将会给出一些示例，为读者提供一些有用的指导，但对于特定的分解并没有必须遵守的严格的规则，只能通过实践来逐步掌握如何应用这个过程。

当面临一项编写程序的任务时，最好的策略一般是从主程序开始。从主程序的角度出发，将问题作为一个整体考虑，然后试着去标识整个任务的主要分块。一旦确定了程序的大分块，可以沿着这条主线将整个问题继续分解为独立的组件。如果这些组件中的某些组件本身也很复杂，通常可以再将它们分解成更小的分块。这个过程一直持续到每个程序片段都简单到可以独立解决为止。这个过程被称为自顶向下分解，或逐步细化。

为了说明这个过程，本节将介绍一个猜硬币的游戏的实现。实现这个游戏要求完成下列功能：提供游戏指南；计算机随机产生正反面，让用户猜，报告对错结果；重复此过程，直到用户不想

玩了为止。

9.1.1 顶层分解

从主程序开始考虑。程序要做什么？程序要做两件事：显示程序指南；模拟玩游戏的过程。逐步细化的原则指出：一旦有了某个程序的概要描述，你应该在此结束，并把它写下来。因此，可以写出主程序的伪代码表示如下：

```
main( )
{
    显示游戏指南；
    玩游戏；
}
```

主程序的两个步骤是相互独立的，没有什么联系，因此可设计成两个函数：

```
void prn_instruction()
void play()
```

至此完成了第一步的分解，可以完成main函数了，见代码清单9-1。

代码清单9-1 猜硬币正反面游戏的主程序

```
//文件名：9-1.cpp
//猜硬币正反面游戏的主程序
int main()
{   prn_instruction();
    play();

    return 0;
}
```

在main函数实现所示的详细程度上，程序已经具有完整的意义。只要prn_instruction和play完成它们的工作，程序应该工作正常。当然，你还没有写这些函数的实现，因此现在就依赖于它们似乎太早。然而，这步工作是成功分解的关键。只要沿着这条路走下去，你会发明新的函数来解决这些任务中的新的有用的片段，然后，根据这些片段实现这个解决方案的层次结构中的每一层。

9.1.2 prn_instruction 的实现

prn_instruction函数的实现非常简单，只要一系列的输出语句把程序指南显示一下就可以了，见代码清单9-2。

代码清单9-2 prn_instruction函数的实现

```
void prn_instruction()
{   cout << "这是一个猜硬币正反面的游戏。\\n";
    cout << "我会扔一个硬币，你来猜。\\n";
    cout << "如果猜对了，你赢，否则我赢。\\n";
}
```

9.1.3 play 函数的实现

play函数随机产生正反面，让用户猜，报告对错结果，然后询问是否要继续玩。根据用户的输入决定是继续玩还是结束。用伪代码表示如下：

```
void play()
{
    char flag = 'Y'; //flag为是否继续玩的标志
    while (flag == 'Y' || flag == 'y')
    {
        coin = 生成正反面;
        输入用户的猜测;
        if (用户猜测 == coin) 报告本次猜测结果正确;
        else 报告本次猜测结果错误;
        询问是否继续游戏;
    }
}
```

在此函数中，尚未完成的任务有：生成正反面，输入用户的猜测，报告本次猜测结果正确，报告本次猜测结果错误，以及询问是否继续游戏。报告本次猜测正确与否很简单，只要直接输出结果就可以了。询问是否继续游戏也很简单，先输出一个提示信息，然后输入用户的选择并存入变量flag。在了解了C++的随机函数后，生成正反面的工作也相当简单。如果用0表示正面，1表示反面，那么生成正反面就是随机生成0和1两个数。这可以用一个简单的表达式rand()*2/(RAND_MAX+1)实现。最后剩下的问题就是输入用户的猜测。如果不考虑程序的健壮性，这个问题也可以直接用一条输入语句即可。但想让程序做得好一点，就必须考虑得全面一些。比如，用户可以不守游戏规则，既不输入0也不输入1，而是输入一个其他值，程序该怎么办？因此这个任务还可以进一步细化，可以把它再抽象成一个函数get_call_from_user。该函数从键盘接受一个合法的输入，返回给调用者。因此它没有参数，但有一个整型的返回值。有了这个函数以后，就可以完成play函数了，见代码清单9-3。

代码清单9-3 play函数的实现

```
void play()
{
    int coin ;
    char flag = 'Y';

    srand(time(NULL)); //设置随机数种子
    while (flag == 'Y' || flag == 'y')
    {
        coin = rand() * 2 / (RAND_MAX + 1); //生成扔硬币的结果
        if (get_call_from_user() == coin) cout << "你赢了";
        else cout << "我赢了";
        cout << "\n继续玩吗(Y或y)? ";
        cin >> flag;
    }
}
```

9.1.4 get_call_from_user 的实现

实现get_call_from_user函数是本程序的最后一项任务。这个函数接收用户输入的一个

整型数。如果输入的数不是0或1，则重新输入，否则返回输入的值。函数的实现如代码清单9-4所示。

代码清单9-4 get_call_from_user函数的实现

```
int get_call_from_user()
{   int guess;    // 0 = head, 1 = tail
    do { cout << " \n输入你的选择 (0表示正面, 1表示反面) :";
        cin >> guess;
    } while (guess != 0 && guess != 1);
    return guess;
}
```

至此，猜硬币的游戏程序已经全部完成。程序的运行结果如下：

这是一个猜硬币正反面的游戏。
我会扔一个硬币，你来猜。
如果猜对了，你赢，否则我赢。

输入你的选择 (0表示正面, 1表示反面) :1
我赢了
继续玩吗 (Y或y) ?y
输入你的选择 (0表示正面, 1表示反面) :6
输入你的选择 (0表示正面, 1表示反面) :1
你赢了
继续玩吗 (Y或y) ?n

9.2 模块划分

把大问题分解成小问题的方法在程序设计的许多阶段都会用到，猜硬币游戏在只有一个源文件的情况下阐述了此项技术。然而，当程序很复杂或由很多函数组成的时候，要在一个源文件中处理如此众多的函数会变得很困难，就如要一下子理解一个由50行代码组成的函数是比较困难的一样。在这两种情况下，引入一些额外的结构是很有用的。比如，碰到一个由50行代码组成的函数时，最好的方法是进一步细化，从中抽取出几个小函数，该函数调用这些小函数完成所需的任务。这样，该函数本身变得比较短，比较容易理解。当碰到一个包含50个函数的程序时，最好的办法就是把程序分成几个小的源文件。每个源文件包含一组相关的函数。由整个程序的一部分组成的较小的源文件称为模块。每个独立的模块比整个程序要简单。而且，如果在设计的时候考虑得非常仔细的话，可以把同一模块作为许多不同应用程序的一部分。

当面对一组函数时，如何把它们划分成模块是一个问题。模块划分没有严格的规则，但有一个基本原则，即同一模块中的函数功能较类似，联系较密切，不同模块中的函数联系很少。当把一个程序分成模块的时候，选择合适的分解方法来减少模块之间相互依赖的程度是很重要的。下面通过一个石头、剪刀、布游戏来说明这个过程。

石头、剪刀、布是孩子们中很流行的一个游戏。在这个游戏中，孩子们用手表示石头、剪刀、布。伸手表示布，拳头表示石头，伸出两根手指表示剪刀。孩子们面对面地数到3，然后亮出各

自的选择。如果选择是一样的，表示平局，否则就用如下规则决定胜负：

- 布覆盖石头；
- 石头砸剪刀；
- 剪刀剪碎布。

现在我们把这个过程变成计算机和游戏者之间的游戏。游戏的过程如下：游戏者选择出石头、剪子或布，计算机也随机选择一个，评判结果，输出结果，继续游戏，直到游戏者选择结束为止。在此过程中，游戏者也可以阅读游戏指南或看看当前战况。

要解决这个问题，首先进行第一层分解。根据题意，main函数的运行过程如下：

```
while(用户输入 != 退出)
{switch(用户的选择)
    {case 布, 石头, 剪子:
        机器选择;
        评判结果;
        报告结果;
    case 显示游戏战况: 显示目前的战况;
    case 帮助: 显示帮助信息;
    default: 报告错误;
    }
    显示战况;
```

从这个过程中可以提取出第一层所需的6个函数：selection_by_player获取用户输入、selection_by_machine获取机器选择、compare评判结果、report报告结果并记录结果信息、prn_game_status显示目前战况和prn_help显示帮助信息。这6个函数都比较简单，不需要继续分解。因此，自顶向下分解到此结束。同时，为了提高程序的可读性，我们定义两个枚举类型：用户合法的输入p_r_s和评判的结果outcome。这两个枚举类型分别定义如下：

```
enum p_r_s {paper, rock, scissor, game, help, quit} ;
enum outcome {win, lose, tie, error} ;
```

当组成一个程序的函数较多时，需要把这些函数分成不同的源文件，交给不同的程序员去完成。如前所述，模块划分的原则是：同一模块中的函数功能比较类似，联系比较密切，而不同模块中的函数之间的联系尽可能小。按照这个原则，我们可以把整个程序分成4个模块：主模块、获取选择的模块、比较模块和输出模块。主模块（main.cpp）包括main函数。获取选择模块（select.cpp）包括selection_by_player和selection_by_machine两个函数。比较模块（compare.cpp）包括compare函数。输出模块（print.cpp）包括所有和输出相关的函数，有report、prn_game_status和prn_help函数。至此，我们可以考虑第二层模块的设计与实现。先考虑每个模块中的函数的原型。

选择模块包括selection_by_player和selection_by_machine两个函数。selection_by_player从键盘接收用户的输入并返回此输入值，因此，它不需要参数，但有一个p_r_s类型的返回值。selection_by_machine函数由机器产生一个石头、剪子、布的值，并返回，因此，它也不需要参数，但有一个p_r_s的返回值。

比较模块包括compare函数。该函数对用户输入的值和机器产生的值进行比较，确定输赢。因此它要有两个参数，都是p_r_s类型的，它也应该有一个返回值，就是判断的结果，即outcome类型。

输出模块包括所有与输出相关的函数，有report、prn_game_status和prn_help函数。prn_help显示用户输入的指南，告诉用户如何输入它的选择。因此，它没有参数也没有返回值。report函数报告输赢结果，并记录输赢的次数。对report函数的用户来讲，他知道调用该函数是输出输赢结果，因此需要提供一个信息，即本次比赛的比较结果。事实上，该函数还必须将本次的比赛结果记入统计结果值中，因此它必须有4个参数，即输赢结果、输的次数、赢的次数和平局的次数，但没有返回值。prn_game_status函数报告至今为止的战况。对用户来讲，他只要调用prn_game_status函数就能知道战况。但事实上，显示战况必须知道迄今为止的统计结果，因此需要3个参数，即输的次数、赢的次数和平的次数，但没有返回值。观测report和prn_game_status函数的原型，发现用户对函数原型的需求和函数实现对函数原型的需求是有矛盾的。对report函数，用户只需要一个参数，而函数实现需要另外3个参数。prn_game_status函数的情况也是如此。进一步观察表明，统计结果值与其他模块的函数无任何关系，其他模块的函数不必知道这些变量的存在。这样的变量可设计成模块的内部状态，即模块内的函数需要共享而与模块外的函数无关的变量。内部状态可以作为该模块的全局变量，这样report和prn_game_status函数中都不需要这3个参数了。我们把这3个变量对函数的用户隐藏了起来。

综上所述，可以得到该程序中各函数的原型：

```
p_r_s selection_by_player()
p_r_s selection_by_machine()
outcome compare(p_r_s, p_r_s)
void prn_help()
void report(outcome)
void prn_game_status()
```

一般来讲，每个模块都可能调用其他模块的函数，以及用到本程序自己定义的一些类型或符号常量。为方便起见，我们把所有的符号常量定义、类型定义和函数原型声明写在一个头文件（即.h文件）中，让每个模块都include这个头文件。这样，每个模块就不必要再写那些函数的原型声明了。但这样做又会引起另一个问题，当把这些模块连接起来时，编译器会发现这些类型定义、符号常量和函数原型的声明在程序中反复出现多次，编译器会认为某些符号被重复定义，因而会报错。这个问题的解决需要用到一个新的编译预处理命令：

```
#ifndef 标识符
...
#endif
```

这个预处理命令表示：如果指定的标识符没有定义过，则执行后面的语句，直到#endif；如果该标识符已经定义过，则中间的这些语句都不执行。所以头文件都有下面这样的结构：

```
#ifndef _name_h
#define _name_h
    头文件真正需要写的内容
```

```
#endif
```

其中, `_name_h` 是用户选择的代表这个头文件的一个标识。根据上述原则, 石头、剪子、布游戏程序的头文件如代码清单9-5所示。

代码清单9-5 石头、剪子、布游戏程序的头文件

```
//文件: p_r_s.h
//本文件定义了两个枚举类型, 声明了本程序包括的所有函数原型
#ifndef P_R_S
#define P_R_S
    #include <iostream>
    #include <cstdlib>
    #include <ctime>
    using namespace std;

    enum p_r_s {paper, rock, scissor, game, help, quit};
    enum outcome {win, lose, tie, error};

    outcome compare(p_r_s player_choice, p_r_s machine_choice);
    void prn_final_status();
    void prn_game_status();
    void prn_help();
    void report(outcome result);
    p_r_s selection_by_machine(void);
    p_r_s selection_by_player(void);
#endif
```

当编译第一个模块时, 编译器发现没有见过符号 `P_R_S`, 于是定义了符号 `P_R_S`, 并声明了所有的函数原型, 定义了两个枚举类型。当第二个模块编译时, 又遇到这个头文件。此时, 编译器发现 `P_R_S` 已经定义过, 因此跳过中间所有的定义和声明。这样就避免了函数被反复声明多次的问题。

有了这些函数, 我们就可以实现 `main` 函数了。包含 `main` 函数的主模块如代码清单9-6所示。该实现只是把代码清单9-5所示的伪代码中需要进一步细化的内容用相应的函数取代。

代码清单9-6 石头、剪子、布游戏的主模块的实现

```
//文件: 9-6.cpp
//石头、剪子、布游戏的主模块
#include "p_r_s.h"

int main(void)
{
    outcome result;
    p_r_s player_choice, machine_choice;

    // seed the random number generator
    srand(time(NULL));

    while((player_choice = selection_by_player()) != quit)
        switch(player_choice) {
```

```

        case paper:    case rock:    case scissor:
            machine_choice = selection_by_machine();
            result = compare(player_choice, machine_choice);
            report(result); break;
        case game: prn_game_status(); break;
        case help: prn_help(); break;
        default:
            cout<< " PROGRAMMER ERROR: Cannot get to here!\n\n";
            exit(1);
    }
    prn_game_status();
    return 0;
}

```

选择模块的实现如代码清单9-7所示。

代码清单9-7 选择模块的实现

```

//文件: select.cpp
//包括机器选择selection_by_machine和玩家选择selection_by_player函数的实现
#include "p_r_s.h"

p_r_s selection_by_machine( )
{   int select = (rand( ) * 3 / (RAND_MAX + 1)); //产生0到2之间的随机数

    cout << " I am ";
    switch(select){
        case 0: cout << "paper. "; break;
        case 1: cout << "rock. "; break;
        case 2: cout << "scissor. "; break;
    }

    return ((p_r_s) select); //强制类型转换, 将0到2分别转换成paper, rock, scissor
}

p_r_s selection_by_player()
{   char c;
    p_r_s player_choice;

    prn_help(); //显示输入提示
    cout << "please select: "; cin >> c;
    switch(c) {
        case 'p': player_choice = paper;   cout << "you are paper. "; break;
        case 'r': player_choice = rock;    cout << "you are rock. "; break;
        case 's': player_choice = scissor;  cout << "you are scissor. ";break;
        case 'g': player_choice = game;    break;
        case 'q': player_choice = quit;    break;
        default : player_choice = help;    break;
    }
    return player_choice;
}

```


比较模块的实现如代码清单9-8所示。该模块在实现比较时用了个技巧。在本例中，机器有3种选择，玩家也有3种选择，比较时应该有9种情况。但compare函数用了个相等比较player_choice == machine_choice，一下子就包含了3种情况，而且可以使后面的switch语句中的每个case子句都能区分两种情况。

代码清单9-8 比较模块的实现

```
//文件: compare.cpp
//包括compare函数的实现
#include "p_r_s.h"

outcome compare(p_r_s player_choice, p_r_s machine_choice)
{   outcome result;

    if (player_choice == machine_choice)   return tie;
    switch(player_choice) {
        case paper: result = (machine_choice == rock) ? win : lose; break;
        case rock: result = (machine_choice == scissor) ? win : lose; break;
        case scissor: result = (machine_choice == paper) ? win : lose; break;
        default: cout << " PROGRAMMER ERROR: Unexpected choice!\n\n";
                exit(1);
    }
    return result;
}
```

输出模块的实现如代码清单9-9所示。

代码清单9-9 输出模块的实现

```
//文件: print.cpp
//包括所有与输出有关的模块,有prn_game_status、prn_help和report函数
#include "p_r_s.h"

int win_cnt = 0, lose_cnt = 0, tie_cnt = 0; //模块的内部状态

void prn_game_status()
{   cout << endl ;
    cout << "GAME STATUS:" << endl;
    cout << "win:  " << win_cnt << endl;
    cout << "Lose:  " << lose_cnt << endl;
    cout << "tie:   " << tie_cnt << endl;
    cout << "Total:" << win_cnt + lose_cnt + tie_cnt << endl;
}

void prn_help()
{   cout << endl
    << "The following characters can be used:\n"
    << "   p   for paper\n"
    << "   r   for rock\n"
    << "   s   for scissors\n"
```

```
<< "    g    print the game status\n"
<< "    h    help, print this list\n"
<< "    q    quit the game\n";
}

void report(outcome result)
{
    switch(result) {
        case win: ++win_cnt; cout << "You win. \n";    break;
        case lose: ++lose_cnt; cout << "You lose.\n";    break;
        case tie: ++tie_cnt;    cout << "A tie.\n";    break;
        default:    cout <<
            " PROGRAMMER ERROR: Unexpected result!\n\n";
            exit(1);
    }
}
```

9.3 设计自己的库

现代程序设计中，编写一个有意义的程序不可能不调用库函数。在我们写的每一个程序中，至少都包含一个库*iostream*。系统提供的库包含的都是一些通用的函数。如果你的编程工作经常要用到一些特殊的工具，你可以设计自己的库。例如，在9.1节和9.2节中的两个程序都用到了取某一范围的随机数的功能，因此可以设计一个处理随机数的库，供这两个程序使用。

每个库有一个主题。一个库中的函数应该是处理同一类问题的。例如，标准库*iostream*包含输入/输出函数，*cmath*包含数学运算函数。我们自己设计的库也要有一个主题。

设计一个库还要考虑到它的通用性。库中的函数应来源于某一应用，但不应该局限于该应用，而且要高于该应用。在某一应用中提取库内容时应尽量考虑到兼容更多的应用，使其他应用也能共享这个库。

当库的内容选定后，设计和实现库还有两个工作要做：设计库的接口，设计库中函数的实现。

使用库的目的是减少程序设计的复杂性，使程序员可以在更高的抽象层次上构建功能更强的程序。对库的使用者来讲，他只需要知道库提供了哪些函数，如何使用这些函数。至于这些函数是如何实现的，库的用户不必知道。按照这个原则，库被分成了两部分：库的用户需要知道的部分和库的用户不需要知道的部分。前者包括库中函数的原型、这些函数用到的符号常量和自定义类型，这部分内容称为库的接口，在C++中库的接口被表示为头文件（.h文件）。用户不需要知道的部分包括库中函数的实现，在C++中被表示成源文件。库的这种实现方法称为信息隐藏，把库的实现细节对库用户隐藏了起来。

下面以随机函数库为例说明库的设计和实现。

在9.1节中设计了一个掷硬币的程序，该程序用到了随机数的一些特性。如果我们的工作经常需要用到随机数，我们可以把随机数的应用写成一个库。

首先考虑库的功能。在掷硬币的例子中，我们用到了随机生成0和1的功能。为此，我们可以设计库中有一个随机生成0或1的函数。但静下心来仔细想想，除了掷硬币的程序外，其他程序也可能用到随机数。在石头、剪子、布的游戏里，我们需要随机生成0~2的值，在自动出题程序中，

需要随机生成0~3的值。为了使我们的库也能用在这些应用中,库中是否应该也包含随机生成0~2以及随机生成0~3的函数?事实上,这些功能可以用一个函数概括:生成low到high之间的随机数。因此,从掷硬币程序出发,推而广之,我们得到了一个通用的工具: `int RandomInteger(int low, int high)`。

在随机数的应用中,需要设置随机数的种子。随机数种子的选择也是一个复杂的过程。我们需要包含头文件`ctime`,要用到取系统时间函数`time`等。为了避免让使用随机数的用户了解这么多复杂的细节,我们可以在库中设计一个初始化函数`RandomInit()`实现设置随机数种子的功能,而且只告诉库的用户在第一次使用`RandomInteger`函数之前必须要调用`RandomInit`函数做一些初始化的工作。至于初始化工作到底要做些什么,怎么做,用户就不必知道了。除了这两个函数外,随机数还有很多应用,如产生概率事件,产生服从某一分布的随机变量的值。这些函数的实现较为复杂,需要有随机过程方面的知识,这里不再一一介绍。

如果我们设计的库就包含这两个函数,接下去就可以设计库的接口。库接口文件的格式和9.2节提到的接口文件一样,主要包含函数原型声明、有关每个函数的注释以及有关常量定义和类型定义。因此,可得到随机数函数库的接口文件`Random.h`,如代码清单9-10所示。

代码清单9-10 随机数函数库的接口

```
//文件: Random.h
//随机函数库的头文件
#ifndef _random_h
#define _random_h

//函数: RandomInit
//用法: RandomInit()
//作用: 此函数初始化随机数发生器
void RandomInit();

//函数: RandomInteger
//用法: n = RandomInteger(low, high)
//作用: 此函数返回一个low到high之间的随机数, 包括low和high
int RandomInteger(int low, int high);

#endif
```

库的实现在另一个`.cpp`文件中。一般库的实现文件和头文件的名称是相同的。例如,若头文件为`Random.h`,则实现文件为`Random.cpp`。

实现也是从注释开始的,这一部分简单介绍库的功能。接下去列出实现这些函数所需的头文件。例如,这些函数用到了`cstdlib`和`ctime`,必须`include`这些头文件。最后,每个实现文件要包含自己的头文件,以便编译器能检查源文件中的函数定义和头文件中的函数原型声明的一致性。库包含以后是每个函数的实现代码。在每个函数实现的前面也必须有一段注释。它和头文件中的注释的用途不同,头文件中的注释是告诉库的用户如何使用这些函数,而实现文件中的注释是告诉库的维护者这些函数是如何实现的,以便维护者将来容易修改。`Random.cpp`的实现如代码

清单9-11所示。

代码清单9-11 随机数函数库的实现

```
//文件: Random.cpp
//该文件实现了Random库
#include <cstdlib>
#include <ctime>
#include "Random.h"

//函数: RandomInit
//该函数取当前系统时间作为随机数发生器的种子
void RandomInit()
{
    srand(time(NULL));
}

//函数: RandomInteger
//该函数将0到RAND_MAX的区间的划分成high-low+1个子区间。
//当产生的随机数落在第一个子区间时, 则映射成low。当落在最后一个子区间时,
//映射成high。当落在第i个子区间时 (i从0到high-low), 则映射到low + i
int RandomInteger(int low, int high)
{
    return (low + (high - low + 1) * rand() / (RAND_MAX + 1));
}
```

有了Random库就可以使用户进一步远离系统的随机数生成器, 给用户提供更方便的随机数生成工具。读者可以自行修改9.1节和9.2节中的程序, 使它们应用Random库, 而不是直接调用系统提供的随机数生成器。

下面用一个实例说明Random库的应用。

龟兔赛跑是人尽皆知的故事。但你想知道龟兔赛跑的过程是怎样的吗? 是不是在整个过程中乌龟始终领先呢? 是不是每次乌龟肯定都能赢呢? 我们可以让计算机来重演这个过程。

如何让计算机模拟这个过程? 我们可以把兔子和乌龟在赛跑过程中的行为抽象出几种状态, 并按照兔子和乌龟的习性确定每种状态的发生概率。如果根据观察, 乌龟和兔子在赛跑过程中的状态及发生概率如表9-1所示。

表9-1 乌龟和兔子的状态及发生概率

动物	跑动类型	占用时间	跑动量	动物	跑动类型	占用时间	跑动量
乌龟	快走	50%	向前走3点	兔子	睡觉	20%	不动
	后滑	20%	向后退6点		大后滑	20%	向后退9点
	慢走	30%	向前走一点		快走	10%	向前走14点
					小步跳	30%	向前走3点
					慢后滑	20%	向后退2点

在程序中, 分别用变量tortoise和hare代表乌龟和兔子的当前位置。开始时, 大家都在起

点，即位置0。然后模拟在每一秒中乌龟和兔子的动作。乌龟和兔子在每一秒时间内的动作是不确定的。对乌龟来说，可能是快走、后滑或慢走。对兔子来说，可能是睡觉、大后滑、快走、小步跳或慢后滑。因此是一个随机事件，所以我们很容易就想到了随机数。我们可以用随机数来决定乌龟和兔子在每一秒的动作，根据动作决定乌龟和兔子的位置的移动。因此可得到第一层的分解：

```
main()
{
    int hare = 0, tortoise = 0, timer = 0; //timer是计时器，从0开始计时

    while (hare < RACE_END && tortoise < RACE_END) // RACE_END是跑道长度
    {
        tortoise += 乌龟根据它这一时刻的行为移动的距离;
        hare += 兔子根据它这一时刻的行为移动的距离;
        输出当前计时和兔子和乌龟的位置;
        ++timer;
    }
    if (hare > tortoise) cout << "\n hare wins!";
    else cout << "\n tortoise wins!";
}
```

从这段伪代码中可以看出，有3个地方需要进一步分解：确定乌龟的移动距离、确定兔子的移动距离和输出当前位置。我们可以进一步提取出3个函数。

- 乌龟在这一秒的移动距离：int move_tortoise()。
- 兔子在这一秒的移动距离：int move_hare()。
- 输出当前计时和兔子乌龟的位置：void print_position(int timer,int tortoise, int hare)。

第三个函数是非常简单的输出，可以直接写出来，不需要进一步分解。前两个函数的处理非常类似，都是先要确定它们的动作，然后根据动作决定移动的位置。关键是如何确定它们的动作，这和随机数有关。我们已经有了随机数库，因此这两个函数也不需要进一步细化了。

接下去要进行模块划分。由于move_tortoise和move_hare()行为非常类似，他们和print_position的行为完全不同，因此该程序可以划分成3个模块：主模块、移动模块和输出模块。

有了第一层分解出来的函数和Random库，我们可以实现主模块。主模块的实现如代码清单9-12所示。

代码清单9-12 主模块的实现

```
//文件名：9-12.cpp
//主模块的实现
#include "Random.h" //包含随机数库
#include <iostream>
using namespace std;

const int RACE_END = 70; //设置跑道的长度

int move_tortoise();
int move_hare();
void print_position(int, int, int);

int main()
```

```

{   int hare = 0, tortoise = 0, timer = 0; //hare和tortoise分别是兔子和乌龟的位置,
                                     //timer是计时器, 统计比赛用时

    RandomInit(); //随机数初始化
    cout << "timer   tortoise   hare\n"; //输出表头

    while (hare < RACE_END && tortoise < RACE_END) //当兔子和乌龟都没到终点时
    {   tortoise += move_tortoise(); //乌龟移动
        hare += move_hare(); //兔子移动
        print_position(timer, tortoise, hare);
        ++timer;
    }

    if (hare > tortoise) cout << "\n hare wins!\n";
        else cout << "\n tortoise wins!\n";

    return 0;
}

```

接下来关键的问题是如何产生乌龟和兔子在某一时刻的动作。我们首先想到的是利用随机数。以乌龟为例，它的3种状态和概率如下：

快走 50%
后滑 20%
慢走 30%

由于随机数的产生是均匀的，它们的出现是等概率的，为此可以生成0~9的随机数。当生成的随机数为0~4时，认为是第一种情况，5~6是第二种情况，7~9是第三种情况。这样就可以根据生成的随机数确定乌龟的动作。同理，可以生成兔子的动作。根据这个原理可以得到move_tortoise和move_hare的实现，如代码清单9-13所示。

代码清单9-13 移动模块的实现

```

//文件名: move.cpp
//移动模块的实现
#include "Random.h" //本模块用到了随机函数库

int move_tortoise()
{   int probability = RandomInteger(0,9); //产生0~9的随机数
    if (probability < 5) return 3; //快走
    else if (probability < 7) return -6; //后滑
        else return 1; //慢走
}

int move_hare()
{   int probability = RandomInteger(0,9);
    if (probability < 2) return 0; //睡觉
    else if (probability < 4) return -9; //大后滑
        else if (probability < 5) return 14; //快走
            else if (probability < 8) return 3; //小步跳
                else return -2; //慢后滑
}

```

显示当前计时和兔子、乌龟的位置的函数的实现如代码清单9-14所示。

代码清单9-14 输出模块的实现

```
//文件名: print.cpp
#include <iostream>
using namespace std;

void print_position(int timer, int t, int h)
{
    if (timer % 6 == 0) cout << endl; //每隔6秒空一行
    cout << timer << '\t' << t << '\t' << h << '\n';
}
```

小结

本章介绍了如何利用结构化程序设计的技术来解决一个大问题。详细介绍了如何把一个大的程序分解成若干个函数，如何把这些函数组织成一个个模块，如何从程序中抽取出库，以及如何设计和使用库。特别介绍了如何在模块中保存内部状态。

习题

简答题

1. 判断题：每个模块对应于一个源文件。
2. 用自己的话描述逐步细化的过程。
3. 为什么库的实现文件要包含自己的头文件？
4. 为什么头文件要包含`#ifndef...#endif`这个编译预处理指令？
5. 什么是模块的内部状态？内部状态是怎样保存的？
6. 为什么要使用库？

程序设计题

1. 将第7章程序设计题的第1题中的一组字符串处理函数设计成一个库。
2. 哥德巴赫猜想指出：任何一个大于6的偶数都可以表示成两个素数之和。编写一个程序，列出指定范围内的所有偶数的分解。
3. 在每本书中，都会有很多图或表。图要有图号，表要有表号。图号和表号都是连续的，如一本书的图号可以编号为：图1、图2、图3、……。设计一个库seq，它可以提供这样的标签系列。该库提供给用户3个函数：`void SetLabel(const char *)`、`void SetInitNumber(int)`和`char * GetNextLabel()`。第一个函数设置标签，如果`SetLabel("图")`，则生成的标签为图1、图2、图3、……；如果`SetLabel("表")`，则生成的标签为表1、表2、表3、……；如

不调用此函数，则默认的标签为"lable"。第二个函数设置起始序号，如SetInitNumber(0)，则编号从0开始生成；SetInitNumber(9)，则编号从9开始生成。第三个函数是获取标签号。例如，如果一开始设置了SetLabel("图")和SetInitNumber(0)，则第一次调用GetNextLabel()返回“图0”，第二次调用GetNextLabel()返回“图1”，依次类推。

4. 试将第8章程序设计题的第1题改写成一个库。
5. 试将第8章程序设计题的第2题改写成一个库。
6. 试将第8章程序设计题的第3题改写成一个库。

第 10 章

创建功能更强的类型 ——类的定义与使用

10.1 从过程化到面向对象

10.1.1 抽象的过程

抽象是人类处理复杂性问题的基本方法，解决某一问题的复杂性直接与抽象的类型和质量有关。所有的程序设计语言都提供抽象。机器语言和汇编语言是对机器硬件的抽象，使程序员只需要通过一系列二进制的位串去命令机器工作，而不必关心硬件是如何完成这些工作的。高级语言是对汇编语言和机器语言的抽象，使程序员可以在更高的层次上与机器交流。例如，程序员可以命令机器将一个实数保存起来，也可以命令机器将两个实数相加，而不必关心计算机是如何保存一个实数以及如何实现实数的加运算的。

不管是高级语言还是机器语言，它们的主要抽象还是要求程序员按计算机的结构去思考，而不是按要解决的问题的结构去思考。因此，当程序员要解决一个问题时，必须要在机器模型和实际要解决的问题模型之间建立联系。而计算机的结构本质上还是为了支持计算，各种高级语言提供的工具都是对数值的处理。例如，对整数和实数的处理，至多也就是对字符的处理。如果要解决一些非计算问题，这个联系的建立就很困难。在过程化程序设计中，通常采用的是逐步细化的过程。对要解决的问题的功能进行分解，直到能用程序设计语言提供的工具解决为止。在此过程中，我们用了过程抽象的概念。

面向对象程序设计方法为程序员提供了创建工具的功能。在解决一个问题时，程序员首先考虑的是需要哪些工具，然后创建这些工具，用这些工具解决问题。有了合适的工具，问题的解决就自然而然了。这些工具就是所谓的对象。事实上，整型数是一个对象，实型数也是一个对象，这些是解决运算问题的基本工具，在所有的高级语言中都包含这些工具。但高级语言不可能提供解决所有问题所需的工具，所以这些工具需要程序员自己去创建。某些对象之间会有一些共性，可以把它们归为一类。每个对象是这个类的一个实例。因此，类也就是我们通常所说的类型。例如，整型是一种类型，而5是整型这种类型的一个对象。

在面向对象程序设计中，我们的主要工作是创建新的数据类型，用这种类型的变量（对象）完成指定的工作。因此，面向对象方法解决问题时的思考方式与过程化方法不同。例如，对于计算圆的面积和周长的问题。用过程化设计方法，可以从功能着手。解决这个问题过程化实现首先要输入圆的半径或直径，然后利用 $s=\pi r^2$ 和 $C=2\pi r$ 计算面积和周长，最后输出计算结果。但是按面向对象程序设计方法，我们首先考虑需要什么工具。如果计算机能提供给我们一个称为圆的工具，它可以以某种方式保存一个圆，告诉我们有关这个圆的一些特性，如它的半径、直径、面积和周长，那么解决这个问题就简单多了。我们只要定义一个圆类型的变量，以它提供的方式将一个圆保存在该变量中，然后让这个变量告诉我们这个圆的面积和周长是多少。这样，程序员就不必知道圆是如何保存的，也不必知道如何计算圆的面积和周长。就像在过程化程序设计中一样，如果要保存一个整型值，可以定义一个整型变量，把这个值赋给该变量，而不用管这个值在内存是怎样保存的。当我们需要对这个整型变量进行运算时，只需要写一个表达式，而不用管这些运算是怎样实现的。由于程序设计语言事先并不知道要解决什么应用问题，因此不可能提供所有需要的工具，为此，我们需要自己定义圆这样的类型。

10.1.2 面向对象程序设计的特点

定义了圆这样一个类型后，不仅我们自己可以用这个类型的对象完成所需的功能，我们还可以把这个类型提供给那些也需要处理圆的程序员使用。这样就可以使处理圆的这些代码得到重用。

面向对象程序设计可以利用代码重用使软件更好地适合用户需求的变化。对所开发的软件而言，功能常常是易变的，但对象往往是稳定的。例如，对于高校管理系统，各个学校的管理模式有很大的区别（有的是公办的，有的是民办的，有的实现学分制，有的不采用学分制），但对学校系统中的对象（如学生、老师、课程）而言，对于不同的管理模式，他们的变化并不大。在过程化程序设计中，如果学校的管理模式发生了变化，整个系统可能要重做，但如果采用面向对象方法，这些对象在新系统中都能重用。

面向对象程序设计的另一个特征是实现隐藏。在面向对象程序设计中，程序员被分成了两类：类的创建者和类的使用者。类的创建者不断地创造新的工具，而类的使用者则收集已有的工具快速解决要解决的问题。类的使用者不需要知道这些工具是如何实现的，只需要知道如何使用这些工具就可以了。这种工作方式称为实现隐藏，它能减少程序员开发某一应用程序的复杂性。

第9章讲了库的设计和实现，也谈到了实现隐藏问题。设计一个库和设计一个类型的区别将在后面详细介绍。

有了一些工具之后，我们可以在这些工具的基础上加以扩展，形成一个功能更强的工具。例如，有了圆这样一个类，就可以在这个类的基础上扩展出圆柱体这个类。因为圆柱体上下两端都是圆，保存一个圆柱体需要保存一个圆的信息及高度信息。圆柱体的体积和表面积的计算都和圆的面积和周长有关。在已有类的基础上再扩展一个新类称为类的继承或派生。原有的类称为基类或父类，新建的类称为派生类或子类。有了继承，代码又可以进一步得到重用。在创建派生类时，基类已有的这些功能就不用再重写了。例如，在学校管理系统中，我们处理的基本对象是人，因此需要定义一个表示“人”的类。人又进一步被分成各种类型，有教师、学生和教辅人员。每一

类人员都具有人的特性。因此可以从“人”类的基础上派生。在派生类定义时，只需要实现派生类新增加的功能，基类已有的部分就不必重写了。同理，教师又可进一步分为高级职称、中级职称和初级职称。学生又可进一步分为本科生、硕士生和博士生。教辅人员也可进一步分为实验室人员和行政人员。因此，可得到图10-1所示的继承关系。

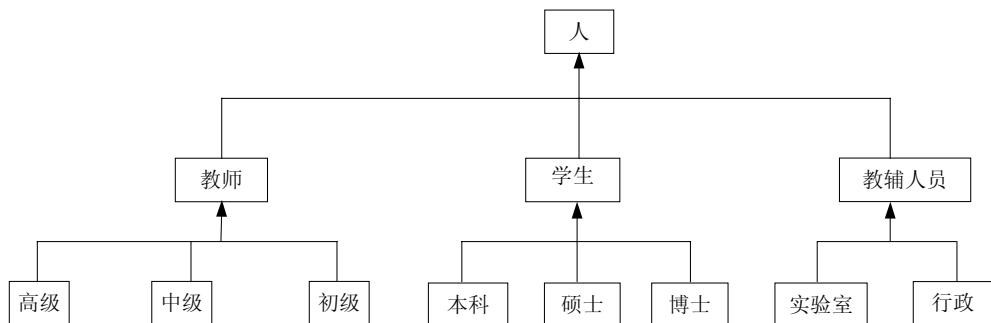


图10-1 学校管理系统中的类的继承关系

处理层次结构的类型时，程序员可能不希望把对象看成是一类特殊的成员，而是想把它看成是基类成员。例如，程序员可以对教师进行处理（如进行考核），而不必管他是什么职称；程序员也可以对学生进行处理（如输出他的期末考试成绩），而不用管是哪一类学生。每个对象自会按照自己的类型作出相应的处理，这样可以进一步简化程序。如果一旦对所有的教师发出考核指令，高级职称的教师会按高级职称的标准进行考核，初级职称的教师会按初级职称的标准进行考核，而不必对每类教师单独发考核指令。这种行为称为多态性。

多态性的另一个好处是程序功能扩展比较容易。例如，如果需增加夜大生的信息处理，只需增加一个夜大生的类型，它也是学生类的一个子类。需要输出学生的期末考试成绩时，整个程序不用修改，夜大生的信息也按规定的格式打印了出来。因此，在程序不变的情况下，功能得到了扩展。

因此，面向对象程序设计的重点就是研究如何定义一个类和如何使用类的对象。

10.1.3 库和类

提高程序生产效率的方法是尽量利用别人已经写好的代码，而利用别人已经写好的代码的一种方法是使用库。我们已经用过系统的*iostream*、*cmath*、*cstdlib*等库，也学过了如何设计自己的库。类则是更合理的库。下面以一个更通用的数组为例，说明库和类的异同。

第5章已经学过数组这个组合类型。C++的数组有两个问题：一是下标必须从0开始，而不能像Pascal语言一样任意指定下标范围；二是C++不检查下标的越界，这会给数组的应用带来一定的危险。我们希望设计一个工具，该工具允许用户在定义整型数组时可以指定数组的下标范围，并且在访问数组元素时检查下标是否越界。例如，我们可以定义一个下标从1开始到10结束的数组*array*，那么引用*array[1]*，…，*array[10]*是正确的，而引用*array[0]*时系统会报错。依据

已学的知识，这个工具可以设计成一个库。

首先考虑如何保存这个数组。与普通数组一样，这个数组也需要一块保存数组元素的空间。但是，在设计这个库时，我们并不知道数组的大小是多少。很显然，这块空间需要在执行时动态分配。与普通数组不同的是，这个数组的下标不总是从0开始，而是可以由用户指定范围。因此，对每个数组还需要保存下标的上下界。综上所述，保存这个数组需要保存3个部分：下标的上下界和一个指向动态分配的用于存储数组元素的空间的指针。这3个部分是一个有机的整体，因此可以用一个结构体把它们封装在一起。

第二个要考虑的是如何实现数组的操作。C++不能直接对结构体进行操作，结构体的操作必须由用户自己编写程序来实现。因此，我们需要考虑提供哪些数组操作函数。对于数组主要有3个操作：定义一个数组，给数组元素赋值，以及取某一个数组元素的值。由于这个数组的存储空间是动态分配的，因此，还必须有一个函数去释放空间。根据上述考虑，可以得到代码清单10-1所示的库的头文件。

代码清单10-1 array库的接口

```
//文件名: array.h
//array库的接口
#ifndef _array_h
#define _array_h

//可指定下标范围的数组的存储
struct IntArray{
    int low;
    int high;
    int *storage;
};

//根据low和high为数组分配空间。分配成功，返回值为true，否则返回值为false
bool initialize(IntArray &arr, int low, int high);

//设置数组元素的值
//返回值为true表示操作正常，返回值为false表示下标越界
bool insert(const IntArray &arr, int index, int value);

//取数组元素的值
//返回值为true表示操作正常，返回值为false表示下标越界
bool fetch(const IntArray &arr, int index, int &value);

//回收数组空间
void cleanup(const IntArray &arr);

#endif
```

库函数的实现如代码清单10-2所示。有了这个库，就可以在主程序中使用可指定下标范围的数组了。array库的应用如代码清单10-3所示。

代码清单10-2 array库的实现

```

//文件名: array.cpp
//array库的实现
#include "array.h"
#include <iostream>
using namespace std;

//根据low和high为数组分配空间。分配成功, 返回值为true, 否则返回值为false
bool initialize(IntArray &arr, int low, int high)
{
    arr.low = low;
    arr.high = high;
    arr.storage = new int [high - low + 1];
    if (arr.storage == NULL) return false; else return true;
}

//设置数组元素的值
//返回值为true表示操作正常, 返回值为false表示下标越界
bool insert(const IntArray &arr, int index, int value)
{
    if (index < arr.low || index > arr.high) return false;
    arr.storage[index - arr.low] = value;
    return true;
}

//取数组元素的值
//返回值为true表示操作正常, 返回值为false表示下标越界
bool fetch(const IntArray &arr, int index, int &value)
{
    if (index < arr.low || index > arr.high) return false;
    value = arr.storage[index - arr.low] ;
    return true;
}

//回收数组空间
void cleanup(const IntArray &arr)
{
    delete [] arr.storage; }

```

代码清单10-3 array库的应用示例

```

//文件名: 10-3.cpp
//array库的应用示例
#include <iostream>
using namespace std;
#include "array.h"

int main()
{
    IntArray array; //IntArray是array库中定义的结构体类型
    int value, i;

    //初始化数组array, 下标范围为20到30
    if (!initialize(array, 20, 30)) { cout << "空间分配失败" ; return 1;}

    for (i=20; i<=30; ++i) { //数组元素的输入
        cout << "请输入第" << i << "个元素: ";
    }
}

```

```

        cin >> value;
        insert(array, i, value);          //将value存入数组array的第i个元素
    }

    while (true) {                        //数组元素的引用
        cout << "请输入要查找的元素序号 (0表示结束): ";
        cin >> i;
        if (i == 0) break;
        if (fatch(array, i, value)) cout << value << endl;
        else cout << "下标越界\n";
    }

    cleanup(array);                      //回收array的空间

    return 0;
}

```

从代码清单10-3所示的程序来看，这个数组的第一个问题是使用相当笨拙。每次调用与数组有关的函数时，都要传递一个结构体。第二个问题是名字冲突。我们可能在一个程序中用到很多库，每个库都可能需要做初始化和清除工作。库的设计者可能都觉得initialize和cleanup是比较合适的名字，因而都写了这两个函数。这样，在主程序中可能引起函数名的冲突。第三个问题是这个数组和系统内置的数组使用起来感觉还是不一样。例如，系统内置的数组在数组定义时就指定了元素个数，系统会根据元素个数自动为数组准备存储空间，而我们这个数组的元素个数要用initialize函数来指定，使用时比内置数组多了一个初始化的操作。同样，数组使用结束后，还需要库的用户显式地归还空间。让用户考虑空间管理问题并不合适。最后，对数组元素的操作不能直接用下标变量的形式表示，而要用函数的形式来表示，这对库的用户来讲同样不太方便。

从C到C++迈出的第一步是允许将函数放入一个结构体。这一步是过程化到面向对象质的变化。假如我们把array库中的函数都放入结构体IntArray中，array库就显得简洁多了。改进以后的array库的接口文件如代码清单10-4所示，实现文件如代码清单10-5所示。

代码清单10-4 改进后的array库的接口

```

//文件名: array.h
//改进后的array接口
#ifndef _array_h
#define _array_h

struct IntArray{
    int low;
    int high;
    int *storage;

    //根据low和high为数组分配空间。分配成功，返回值为true，否则返回值为false
    bool initialize(int lh, int rh);

    //设置数组元素的值
    //返回值为true表示操作正常，返回值为false表示下标越界
    bool insert(int index, int value);
}

```

```

//取数组元素的值
//返回值为true表示操作正常, 返回值为false表示下标越界
bool fatch(int index, int &value);

//回收数组空间
void cleanup();
};

#endif

```

代码清单10-5 改进后的array库的实现

```

//文件名: array.cpp
//改进后的array库的实现
#include "new_array.h"
#include <iostream>
using namespace std;

//根据low和high为数组分配空间。分配成功, 返回值为true, 否则返回值为false
bool IntArray::initialize(int lh, int rh)
{
    low = lh;
    high = rh;
    storage = new int [high - low + 1];
    if (storage == NULL) return false; else return true;
}

//设置数组元素的值
//返回值为true表示操作正常, 返回值为false表示下标越界
bool IntArray::insert(int index, int value)
{
    if (index < low || index > high) return false;
    storage[index - low] = value;
    return true;
}

//取数组元素的值
//返回值为true表示操作正常, 返回值为false表示下标越界
bool IntArray::fatch(int index, int &value)
{
    if (index < low || index > high) return false;
    value = storage[index - low] ;
    return true;
}

//回收数组空间
void IntArray::cleanup()
{
    delete [] storage; }

```

从代码清单10-4和代码清单10-5中可以看出, 由于将这些函数放入了结构体, 原来函数原型中的第一个参数不再需要。编译器自然知道函数体中涉及的low、high和storage是同一结构体变量中的成员, 这样就简化了函数的原型。由于将这些函数放入了结构体, 函数名冲突的问题也得到了解决。现在函数名是从属于某一结构体的, 从属于不同结构体的同名函数不会冲突。就如

人人都可给自己取个名字叫“爱国”，但只要姓不同，这些名字就不会冲突，姓张的叫“张爱国”，姓李的叫“李爱国”。在C++中，我们用“结构体名::函数名”加以限定，如代码清单10-5所示。

由于将这些函数放入了结构体，函数的调用方法就不同了。同引用结构体的成员一样，要用点运算符引用这些函数。具体的示例见代码清单10-6。

代码清单10-6 改进后的array库的应用示例

```
//文件名: 10-6.cpp
//改进后的array库的应用示例
#include <iostream>
using namespace std;
#include "new-array.h"

int main()
{   IntArray array;
    int value, i;

    if (!array.initialize(20, 30)) { cout << "空间分配失败" ; return 1;}
    //数组array的初始化

    for (i=20; i<=30; ++i) { //数组元素的输入
        cout << "请输入第" << i << "个元素:";
        cin >> value;
        array.insert(i, value);
    }

    while (true) { //数组元素的查找
        cout << "请输入要查找的元素序号(0表示结束):";
        cin >> i;
        if (i == 0) break;
        if (array.fetch(i, value)) cout << value << endl;
        else cout << "下标越界\n";
    }

    array.cleanup(); //归还存储数组元素的空间

    return 0;
}
```

将函数放入结构体是从C到C++的根本改变。在C语言中，结构体只是将一组相关的数据捆绑起来，在逻辑上看成一个整体。它除了使程序逻辑更加清晰之外，对解决问题没有任何帮助。但是，一旦将处理这组数据的函数也加入到结构体中，结构体就有了全新的功能。它既能描述属性，也能描述对属性的操作。事实上，它就成了与内置类型一样的一种全新的数据类型。为了表示这是一种全新的概念，C++用了一个新的名称——类来表示。

将函数放入结构体解决了array库的前两个问题，但第三个问题依然没有解决。随着学习的深入，我们可以看到，C++能够做到让自己定义的这样一个数组和系统内置的数组一样使用。

10.2 类的定义

C++提供了一个比结构体更安全、更完善的类型定义方法——类。如前所述，定义一个类就是定义一组属性和一组函数。属性称为类的数据成员，而函数称为类的成员函数。类定义的一般形式如下：

```
class 类名{  
    [private:]  
        私有数据成员和成员函数;  
    public:  
        公有数据成员和成员函数;  
};
```

其中，class是定义类的关键字，类名是正在定义的类的名字，即类型名，后面的花括号表示类定义的内容，最后的分号表示类定义结束。

private和public用于访问控制。列于private下面的每一行，无论是数据成员还是成员函数都称为私有成员，列于public下面的每一行，都称为公有成员。私有成员只能被自己类中的成员函数访问，不能被全局函数或其他类的成员函数访问。这些成员被封装在类的内部，不为外界所知。公有成员能被程序中的其他所有函数访问，它们是类对外的接口。在使用系统内置类型时，我们并不需要了解该类型的数据在内存是如何存放的，而只需要知道对该类型的数据可以执行哪些操作。与系统内置类型一样，我们在定义自己的类型时，一般将数据成员定义为private，而用户必须知道的操作则定义为public成员。private和public的出现次序可以是任意的，也可以反复出现多次。

利用private可以封装类型的实现细节。封装是将低层次的元素组合起来形成新的、高层次实体的技术。函数就是封装的一种形式。函数所执行的细节被封装在函数本身这个更大的实体中。被封装的元素隐藏了它们的实现细节。类也是封装的一种形式，通常将数据的存储和处理的细节隐藏起来。

封装的主要好处是随着时间的推移，如果需求有所改变或发现了一些bug，我们可以根据新的需求以及所报告的bug来完善类的实现，但无需改变用户级的代码。

结构体和类都能用来定义新的类型。这两种方法的最大的区别在于：在结构体中，如果没有指明访问特性，则成员是公有的；在类定义中，默认情况下，成员是私有的。

用类方法实现的IntArray定义如代码清单10-7所示，对应的成员函数的实现与代码清单10-5中的实现完全相同，其应用也与代码清单10-6完全相同。

代码清单10-7 用class定义的IntArray类型

```
class IntArray{  
private:  
    int low;  
    int high;  
    int *storage;  
  
public:
```

```
//根据low和high为数组分配空间。分配成功，返回值为true，否则返回值为false
bool initialize(int lh, int rh);

//设置数组元素的值
//返回值为true表示操作正常，返回值为false表示下标越界
bool insert(int index, int value);

//取数组元素的值
//返回值为true表示操作正常，返回值为false表示下标越界
bool fatch(int index, int &value);

//回收数组空间
void cleanup();
};
```

在代码清单10-7中，由于数据是如何存放的不必让类的用户知道，因此被设计成`private`，而所有的成员函数用户都必须用到，因此被设计成`public`。

类的定义相当于库的接口，因此被写成一个头文件。

完善类的定义还必须包括所有成员函数的实现。成员函数的实现通常有两种方法。第一种方法是在类定义时只给出函数原型，而函数的定义是写在一个实现文件（.cpp）中。这种方法和代码清单10-5的写法完全一样，这里不再重复叙述了。另一种方法是将成员函数的定义直接写在类定义中，直接定义在类中的函数默认为内联函数。因此，直接定义在类中的成员函数都是比较简单的函数。例如，代码清单10-8中，成员函数就默认为内联函数。当然，内联成员函数也可以写在实现文件中，直接用保留字`inline`说明。良好的程序设计习惯是将类定义和成员函数的实现分开，这样可以更好地达到实现隐藏的目的。

代码清单10-8 将成员函数定义为内联函数

```
//文件名: 10-8.h
//将成员函数定义为内联函数
#ifdef _IntArray
#define _IntArray

class IntArray{
private:
    int low;
    int high;
    int *storage;

public:
    //根据low和high为数组分配空间。分配成功，返回值为true，否则返回值为false
    bool initialize(int lh, int rh);
    {   low = lh;
        high = rh;
        storage = new int [high - low + 1];
        if (storage == NULL) return false; else return true;
    }
    //设置数组元素的值
    //返回值为true表示操作正常，返回值为false表示下标越界
```

```

bool insert(int index, int value);
{ if (index < low || index > high) return false;
  storage[index - low] = value;
  return true;
}

//取数组元素的值
//返回值为true表示操作正常, 返回值为false表示下标越界
bool fatch(int index, int &value);
{ if (index < low || index > high) return false;
  value = storage[index - low] ;
  return true;
}

//回收数组空间
void cleanup()
{ delete [] storage; }
};
#endif

```

例10.1 试定义一个有理数类，该类能提供有理数的加和乘运算。要求保存的有理数是最简形式，如 $2/6$ 应记录为 $1/3$ 。

首先考虑怎样保存一个有理数。所谓的有理数就是可以用两整数的商表示的数，因此保存一个有理数就是保存这两个整数。

接下来考虑有理数类应该有哪些操作。根据题意，这个类必须具有加和乘运算，因此需要一个加函数和一个乘函数。除此之外，还应该有一个设置有理数值的函数，用以设置有理数的分子和分母，有一个函数可以输出有理数。这些行为都是类用户需要的，因此都是公有的成员函数。

注意在题目中还有一个要求，就是保存的有理数要是最简形式。因此，如果用户设置有理数的时候给出的不是最简形式，则必须化简，使之成为最简形式。当把两个有理数相加或相乘后，结果也不一定是最简形式，因此也要化简。由于化简的工作在多个地方都要用到，可以将其写成一个函数。在创建有理数时，设置完分子和分母后可以调用化简函数进行化简。执行完加法后，也可以调用化简函数进行化简。由于化简的工作是类内部的工作，有理数类的用户不需要调用，这类函数称为工具函数。工具函数通常设计为`private`的。按照上述思想设计的有理数类Rational如代码清单10-9所示。

代码清单10-9 有理数类

```

//文件名: Rational.h
//有理数类
#ifndef rational_h
#define rational_h

#include <iostream>
using namespace std;

class Rational {
private:

```

```

    int num;                //分子
    int den;                //分母

    void ReductFraction(); //将有理化简化成最简形式

public:
    void create(int n, int d) { num = n; den = d; ReductFraction(); }
    void add(const Rational &r1, const Rational &r2);    //r1+r2,结果存于当前对象
    void multi(const Rational &r1, const Rational &r2); //r1*r2,结果存于当前对象
    void display() { cout << num << '/' << den; }
};

#endif

```

在代码清单10-9所示的Rational类中，create和display函数作为内联函数直接定义在类定义中，还有3个函数没有实现。这3个函数的实现在Rational.cpp中，如代码清单10-10所示。

代码清单10-10 函数的实现

```

//文件名: Rational.cpp
//函数的实现
#include "Rational.h"

//add函数将r1和r2相加，结果存于当前对象
void Rational::add(const Rational &r1, const Rational &r2)
{
    num = r1.num * r2.den + r2.num * r1.den;
    den = r1.den * r2.den;
    ReductFraction();
}

//multi函数将r1和r2相乘，结果存于当前对象
void Rational::multi(const Rational &r1, const Rational &r2)
{
    num = r1.num * r2.num;
    den = r1.den * r2.den;
    ReductFraction();
}

//ReductFraction实现有理数的化简
//方法: 找出num和den的最大公因子，让它们分别除以最大公因子
void Rational::ReductFraction()
{
    int tmp = (num > den) ? den : num;

    for (; tmp > 1; --tmp)
        if (num % tmp == 0 && den % tmp == 0) { num /= tmp; den /= tmp; break; }
}

```

10.3 对象的使用

10.3.1 对象的定义

一旦定义了一个类，就相当于有了一种新的类型，就可以定义这种类型的变量了。在面向对

象程序设计中，这类变量称为对象。

与普通的变量一样，对象也有两种定义方法：直接在程序中定义某个类的对象，或者用动态内存申请的方法申请一个动态变量。

1. 在程序中直接定义类型为某个类的对象

在程序中直接定义对象的方法与定义普通变量的方法一样，它的格式如下：

存储类别 类名 对象列表；

例如，定义两个IntArray类的对象arr1和arr2，可写成

```
IntArray arr1, arr2;
```

要定义一个静态的Rational类的对象r1，可写成

```
static Rational r1;
```

要定义一个有20个对象的Rational类的对象数组array，可写成

```
Rational array[20];
```

用这种方法定义的对象的作用域和存储类别与普通内置类型的变量完全相同。

2. 动态对象

与普通变量一样，对象也可以在程序执行的过程中动态地创建。与普通动态变量的使用一样，要定义一个动态对象必须有一个指向该对象的指针，然后通过new申请一块存储对象的空间，通过delete释放动态对象占用的空间。例如，要动态申请一个存储Rational对象的空间，可以先定义一个指向Rational对象的指针，然后使用new为Rational对象动态申请一个存储空间，如下所示：

```
Rational *rp;  
rp = new Rational;
```

要申请一个20个元素的Rational类的对象数组，可用下面的语句：

```
rp = new Rational[20];
```

释放动态对象，同样是用delete。例如，如果rp是指向动态对象的指针，则可用

```
delete rp;
```

释放该动态对象；如果rp是指向一个动态数组，则可用

```
delete [] rp;
```

释放存储该数组的空间。

10.3.2 对象的操作

与结构体变量一样，对象的操作也是对它的成员的操作。引用对象的成员也是用点运算符。但是在结构体中，默认的情况下所有的成员都是公有的，因此都可以用

结构体变量名.成员名

来访问，而在对象中，对象的用户只能访问公有成员而不能访问私有成员，因此只有公有的成员才能用点运算符来访问。访问对象的公有数据成员，可用

对象名.数据成员名

访问对象的公有成员函数，可以用

对象名.成员函数名(实际参数表)

例10.2 利用例10.1中定义的Rational类，计算两个有理数的和与积。

有了Rational类，这个问题的解决就与计算两个整型数的和与积一样简单。定义3个Rational类的对象：两个存储运算数，一个存储结果。首先，分别将两个加数赋给存储运算数的对象。对第三个对象调用add成员函数，将前两个对象作为实际参数，此时第三个对象存储的就是加的结果。同理可计算两个有理数的积。按照这一思路实现的程序如代码清单10-11所示。

代码清单10-11 有理数类应用示例

```
//文件名: 10-11.cpp
//计算两个有理数的和与积
#include <iostream>
using namespace std;
#include "Rational.h" //使用有理数类

int main()
{
    int n, d;
    Rational r1, r2, r3; //定义三个有理数类的对象

    cout << "请输入第一个有理数（分子和分母）： ";
    cin >> n >> d;
    r1.create(n,d);

    cout << "请输入第二个有理数（分子和分母）： ";
    cin >> n >> d;
    r2.create(n,d);

    r3.add(r1, r2); //执行r3=r1+r2
    r1.display(); cout << " + "; r2.display(); cout << " = "; r3.display(); cout <<
endl;

    r3.multi(r1, r2); //执行r3=r1*r2
    r1.display(); cout << " * "; r2.display(); cout << " = "; r3.display(); cout <<
endl;

    return 0;
}
```

从代码清单10-11可以看出，Rational类的用户并不需要了解如何处理有理数的加和乘，这些细节问题已被封装在Rational类中。但用户很可能对这个类还不满意，因为它用起来不如内置类型那么顺手。例如，不能直接用`r3=r1+r2`，不能直接用`cin`和`cout`来输入和输出。但随着本书介绍，读者能够看到一个能像内置的整型或实型一样处理的Rational类。

与结构体类型的变量一样，除了能用对象名访问对象外，还可以用指向对象的指针访问对象。此时可以用`->`运算符。例如，若`rp`是指向Rational类的对象，要显示该对象可以用

```
rp->display()
```

与结构体类型的变量类似，同类对象之间也可以相互赋值。当把一个对象赋值给另一个对象时，所有的数据成员都会逐位复制。例如，两个Rational类的对象r1和r2之间的赋值：

```
r1 = r2
```

相当于执行

```
r1.num = r2.num;
r1.den = r2.den;
```

10.3.3 this 指针

定义一个对象时，系统会为对象分配空间，用于存储对象的数据成员。例如，Rational类的对象占用的空间是存放两个整型数所需的空间。而类中的成员函数对于该类的所有对象只有一份副本。那么对于类的成员函数来讲，它如何知道要对哪个对象进行操作呢？例如，对于Rational类的成员函数create。它的定义如下：

```
void create(int n, int d) { num = n; den = d; ReductFraction();}
```

当执行这个函数时，系统如何知道其中的num是哪个对象的num，den又是哪个对象的den？实际上，每个成员函数都有一个隐藏的指向本类型的指针形参this，它指向当前调用成员函数的对象。成员函数中对对象成员的访问是通过this指针实现的。因此，create函数的实际形式如下：

```
void create(int n, int d) { this->num = n; this->den = d; this->ReductFraction();}
```

当通过对象调用成员函数时，编译器会把相应对象的地址传给形参this。成员函数通过this指针就知道对哪一个对象进行访问。通常，在写成员函数时可以省略this，编译时会自动加上它们。但是，如果在成员函数中要把对象作为整体来访问时，必须显式地使用this指针。这种情况在函数中返回一个对调用函数的对象的引用时常出现，10.5节中会给出这种用法的示例。

10.3.4 对象的构造与析构

某些类的对象必须被初始化以后才能使用。例如，IntArray类的对象必须通过初始化才能获得存储数组的空间，才能使用。同理，对于某些类的对象在消亡前，往往也需要执行一些操作，做一些善后处理。例如，IntArray类的对象在消亡前必须要归还存储数组的空间，否则会造成内存泄漏。初始化和扫尾的工作给类的用户带来了额外的负担，使他们觉得类和内置类型还是不一样。用户希望使用类的对象就像使用内置类型的变量一样，一旦定义了，就能直接使用，用完了，由系统自动回收。这些工作可有两个特定的成员函数——构造函数和析构函数来完成。构造函数执行初始化的工作，而析构函数执行扫尾的工作。

1. 构造函数

构造函数是一类特殊的成员函数。它不是由类的用户调用，而是由系统在定义对象时自动调用。构造函数的主要工作是保证当前正在定义的对象的数据成员都有合适的初值。构造函数

有以下一些特殊的性质。

- 因为构造函数是系统在定义对象时自动调用的，所以系统必须知道每个类对应的构造函数是哪一个函数。为此，C++规定构造函数的名字必须与类名相同。
- 因为构造函数是系统在定义对象时自动调用的，用户不会检查该函数的返回值，所以构造函数没有返回值。在定义构造函数时，不需要也不能指定返回类型，甚至指定为void也不行。
- 构造函数可以是一组重载函数。这意味着它的对象可以有多种构造方法。
- 构造函数可以设置参数的默认值。

构造函数是在定义对象时自动调用的，因此在定义对象时必须给出构造函数的实际参数。有了构造函数后，对象定义的一般形式如下：

```
类名 对象名(实际参数表);
```

其中，实际参数表必须与该类的某一个构造函数的形式参数表相对应。除非这个类有一个构造函数是没有参数的，才可以用

```
类名 对象名;
```

来定义。不带参数的构造函数称为默认构造函数。

与普通的重载函数一样，如果构造函数是一组重载函数，那么实际参数表就决定了调用哪一个构造函数。例如，可以定义一个Complex类如下：

```
class Complex{
private:
    double real, imag;
public:
    Complex(double r, double i) {real=r; imag=i;}
    Complex() {real = 0; imag = 0; }
    double abscomplex()
    {   double t;
        t=real*real +imag*imag;
        return(t);
    }
};
```

在Complex类中，定义了两个构造函数。因此，在定义Complex的对象时，必须提供与某一构造函数的形式参数表相对应的实际参数表。要定义一个实部为2，虚部为5的复数x，可用下列语句：

```
Complex x(2, 5);
```

根据实际参数表，编译器选择了第一个构造函数。要定义一个实部为0，虚部也为0的复数y，可用下列语句：

```
Complex y;
```

根据实际参数表，编译器选择了第二个构造函数。Complex类的两个构造函数显得非常罗嗦。事实上，这可以通过为形式参数指定默认值来实现。我们可以把构造函数写为

```
Complex(double r = 0.0, double i = 0.0) {real = r; imag = i;}
```

那么，该对象的调用方式可有以下几种：


```
Complex a;           //全部用默认值
Complex b(1.1);      //只传递一个参数，第二个用默认值
Complex c(1.1, 2.2); //传递两个参数，不用默认值
```

每个类都应该至少有一个构造函数。如果在定义类时没有定义构造函数，编译器会自动生成一个默认构造函数。默认构造函数的函数体为空，即不做任何事情。此时生成的对象的所有数据成员的值都为随机值。

有了构造函数，类使用起来就更加方便。例如，我们可以在IntArray类中增加一个构造函数完成初始化的工作。这个函数的实现如下：

```
IntArray :: IntArray(int lh = 0, int rh = 0)
{
    low = lh;
    high = rh;
    storage = new int [high - low + 1];
}
```

有了这个函数，就可以把对象的定义和初始化的工作一起完成。定义一个下标范围可指定的数组，只需要用下面的定义：

```
IntArray array(20,30);
```

这条语句定义了一个下标范围是20~30的整型数组，并已为该数组准备好了空间。

与普通动态变量一样，动态对象也可以初始化。普通动态变量的初始化是在类型后面用一个圆括号指出它的初值。如定义整型的动态变量，并为它赋初值5，可用下列语句：

```
p = new int(5);
```

如果要为一个动态的IntArray数组指定下标范围为20~30，可用下列语句：

```
p = new IntArray(20, 30);
```

括号中的实际参数要与构造函数的形式参数表相对应。

我们也可以为Rational类增加一个构造函数，为它的两个数据成员赋初值。函数的实现如下：

```
Rational::Rational(int n = 0, int d = 0)
{
    num = n;
    den = d;
    ReductFraction();
}
```

这样，就可以在定义Rational对象时为它赋初值了，例如：

```
Rational r1(3,5), r2(1,7);
```

与普通的成员函数一样，构造函数具有名字、形式参数表和函数体，只是它的名字必须与类名完全相同。但它还有一个与普通函数不同的地方，就是可以包含一个构造函数初始化列表。构造函数初始化列表位于函数头和函数体之间。它以一个冒号开头，接着是一个以逗号分隔的数据成员列表，每个数据成员的后面跟着一个放在圆括号中的对应于该数据成员的构造函数的实际参数表。例如，我们可以把IntArray类的构造函数写成初始化列表的形式，如下所示：

```
IntArray :: IntArray(int lh, int rh): low(lh), high(rh)
{
```

```

    storage = new int [high - low + 1];
}

```

我们完全可以在函数体内对数据成员赋初值，为什么还要使用构造函数初始化列表？事实上，不管构造函数中有没有构造函数初始化列表，在执行构造函数体之前，都要调用每个数据成员的构造函数初始化相应的数据成员。如果该数据成员没有出现在初始化列表中，则执行它的默认构造函数。

从概念上讲，可以认为构造函数分两个阶段工作：一是初始化阶段，执行初始化列表，初始化每个数据成员；二是普通的计算阶段，执行构造函数的函数体。因此，采用初始化列表可以提高构造函数的效率，使数据成员在构造的同时完成初值的设置。

在实际应用中，有些时候必须应用初始化列表。例如，数据成员不是普通的内置类型，而是某一个类的对象，可能无法直接用赋值语句在构造函数体中为它赋初值，则可以把它放在初始化列表中，详见第12章。另一种应用是类包含一个常量的数据成员，常量只能在定义时对它进行初始化，而不能对它赋值。因此也必须放在初始化列表中，详见10.5节。

构造函数初始化列表仅给出了要被初始化成员的值，而初始化的次序是类定义中数据成员的定义次序，与初始化列表中的次序无关。

2. 析构函数

与构造函数一样，析构函数是系统在回收对象前自动调用的。因此，它也必须有一个特殊的名字。在C++中，析构函数的名字是“~类名”。它没有参数也没有返回值。

每个类必须有一个析构函数。如果没有显式地定义析构函数，编译器会自动生成一个默认析构函数。该函数的函数体为空，即什么事也不做。

有了构造函数和析构函数，我们可以进一步改进IntArray类。改进后的IntArray类的定义以及构造函数和析构函数的实现如代码清单10-12所示。fetch和insert函数的实现不变。它的使用如代码清单10-13所示。从代码清单10-13可见该对象不再需要显式地执行初始化和收尾工作了。

代码清单10-12 采用构造函数和析构函数的IntArray类的定义

```

class IntArray{
    int low;
    int high;
    int *storage;

public:
    //构造函数根据low和high为数组分配空间
    IntArray(int lh, int rh):low(lh), high(rh)
    {storage = new int [high - low + 1]; }

    //设置数组元素的值
    //返回值为true表示操作正常，返回值为false表示下标越界
    bool insert(int index, int value);

    //取数组元素的值
    //返回值为true表示操作正常，为false表示下标越界
    bool fetch(int index, int &value);
}

```

```

//析构函数
~IntArray() {delete [] storage; }
};

```

代码清单10-13 采用构造函数和析构函数的IntArray类的使用

```

//文件名: 10-13.cpp
//IntArray类的使用
#include <iostream>
using namespace std;
#include "IntArray.h"

int main()
{
    IntArray array(20,30);
    int value, i;

    for (i=20; i<=30; ++i) {
        cout<< "请输入第" << i << "个元素:";   cin >> value;
        array.insert(i, value);
    }

    while (true) {
        cout << "请输入要查找的元素序号 (0表示结束):";
        cin >> i;
        if (i == 0) break;
        if (array.fetch(i, value)) cout << value << endl;
        else cout << "下标越界\n";
    }
    return 0;
}

```

3. 复制构造函数

在创建一个对象时，可以用一个同类的对象对其进行初始化。这需要调用一个特殊的构造函数，称为复制构造函数。复制构造函数以一个同类对象引用作为参数，它的原型如下：

```
类名 (const <类名> &);
```

例如，若需要从一个已有的点对象构建一个新的点对象，新的点对象的坐标值是已有对象的两倍。我们可以在点这个类中定义一个复制构造函数。

```

class point{
    int x, y;
public:
    point(int a, int b) { x = a; y = b;}
    point(const point &p) {x = 2 * p.x; y = 2 * p.y;}
};

```

与构造函数一样，复制构造函数也是由系统自动调用的，它也用类名作为函数名。每个类都有一个复制构造函数，如果类的设计者没有设计复制构造函数，编译器会自动生成一个。该函数将作为形式参数的对象的数据成员值对应地赋给正在创建的对象的数据成员。编译器自动生成的

复制构造函数称为默认的复制构造函数。一般情况下，默认的复制构造函数足以满足要求，但某些情况下可能需要设计自己的复制构造函数。例如，我们希望对IntArray类增加一个功能，能够定义一个和另一个数组完全一样的数组，包括下标范围和数组元素的值。显然，这个工作应该由复制构造函数来完成，但默认的复制构造函数却不能胜任。如果正在构造的对象为arr1，作为参数的对象是arr2，调用默认的复制构造函数相当于执行下列操作：

```
arr1.low = arr2.low;
arr1.high = arr2.high;
arr1.storage = arr2.storage;
```

前两个操作没有问题，第三个操作中，storage是一个指针，这个操作意味着使arr1的storage指针和arr2的storage指针指向同一块空间。以后所有对数组arr1的修改都将变成对arr2的修改。同理，对数组arr2的修改也都将变成对arr1的修改。更为严重的问题是，如果这两个对象的作用域不同，当一个对象析构时，另一个对象也丧失了存储空间。这肯定不是我们的原意。我们的原意是定义一个同arr2一样大小的数组arr1，并把arr2的元素值对应赋给arr1的元素。每个数组应该有自己的存储元素的空间。如果是这样的话，我们可以自己写一个复制构造函数。该函数的定义如下：

```
IntArray(const IntArray &arr)
{
    low = arr.low;
    high = arr.high;
    storage = new int [high - low + 1];
    for (int i = 0; i < high - low + 1; ++i) storage[i] = arr.storage[i];
}
```

该函数首先根据arr的下标范围设置当前对象的下标范围，根据下标的范围申请一块属于自己的空间，最后将arr的每个元素复制到当前对象中。

复制构造函数的应用有以下3种场合。

(1) 对象定义时。复制构造函数用于对象定义时有两种用法：直接初始化和复制初始化。直接初始化将初始值放在圆括号中，直接调用与实参类型相匹配的构造函数。例如，如果已有了一个IntArray类的对象array1，则构造一个新的对象array2可用下面的直接初始化的方法：

```
IntArray array2(array1);
```

根据实参的类型，系统将调用复制构造函数。复制初始化是用=符号。当使用复制初始化时，首先会用=右边的表达式构造一个临时对象，再调用复制构造函数将临时对象复制到正在构造的对象中。例如：

```
IntArray array = IntArray(20,30);
IntArray array1 = array2;
```

(2) 把对象作为参数传给函数时。例如：

```
void f(IntArray array);
IntArray arr;
f(arr);    // 创建一个形式参数对象array，并调用复制构造函数用对象arr初始化array
```

(3) 把对象作为返回值时。例如：

```
IntArray f()
{   IntArray a;
    ...
    return a;
}
```

当执行到`return`语句时，会创建一个`IntArray`类的临时对象，并调用复制构造函数用对象`a`初始化该临时对象，并将此临时对象的值作为返回值。

10.4 常量对象与常量成员函数

自定义类型和内置类型一样，都可以定义常量对象。例如：

```
const int PI = 3.1415926;
```

表示定义了一个整型常量`PI`，它的值为`3.1415926`，一旦将`PI`定义为常量，也就意味着在程序中不能再对`PI`赋值。同理，也能够定义一个常量对象。例如：

```
const Rational r1(1,3);
```

表示对象`r1`是一个常量对象。常量对象只能初始化，不能赋值，而且必须要初始化，否则就无法指定常量的值。如果程序中有试图改变`r1`的语句时，编译器会报错。

问题是对象的操作通常是对它的成员的操作，而很少有对对象的直接操作。如果程序中有一个赋值`r1 = r2`，编译器知道`r1`的值将被改变。如果`r1`被定义为常量，编译器就会报错。如果程序中有诸如“对象名.成员名 = 表达式”这样的语句时，编译器也知道该对象被修改了。但是，如果对象是通过成员函数来操作的，那么编译器如何知道哪些成员函数会改变数据成员？它又如何知道哪些成员函数的访问对常量对象是“安全”的呢？

在C++中可以把一个成员函数定义为常量成员函数，它告诉编译器该成员函数是安全的，不会改变对象的数据成员值，可以被常量对象所调用。一个没有被明确声明为常量成员函数的成员函数被认为是危险的，它可能会修改对象的数据成员值。因此，当把某个对象定义为常量对象后，该对象只能调用常量成员函数。

常量成员函数的定义是在函数头后面加一个保留字`const`。要说明一个函数是常量的，必须在类定义中的成员函数声明时声明它为常量的，同时在成员函数定义时也要说明它是常量的。如果仅在类定义中说明，而在函数定义时没说明，编译器会把这两个函数看成是两个不同的函数，是重载函数。

任何不修改数据成员的函数都应该声明为`const`类型。如果在编写常量成员函数时，不慎修改了数据成员，或者调用了其他非常量成员函数，编译器将指出错误，这无疑会提高程序的健壮性。例如，一个更好的`Rational`类的定义应该写成如下形式：

```
class Rational {
private:
    int num;
    int den;
```

```

void ReductFraction(); //将有理数化简成最简形式

public:
    Rational(int n = 0, int d = 1) { num = n; den = d; ReductFraction(); }
    void create(int n, int d) { num = n; den = d; ReductFraction(); }
    void add(const Rational &r1, const Rational &r2); //r1+r2,结果存于当前对象
    void multi(const Rational &r1, const Rational &r2); //r1*r2,结果存于当前对象
    void display() const { cout << num << '/' << den; }
};

```

一旦将一个Rational类的对象定义为常量的，就不能对该对象调用create、add和multi函数。因为这3个函数都会修改对象的数据成员，前一个函数将重新设置数据成员的值，而后两个函数会把运算的结果存入该对象。该对象允许调用的成员函数只有display函数。另外，它还可以作为add和multi函数的实际参数。

10.5 常量数据成员

类的数据成员也可以定义为常量的。常量数据成员只在某个对象生存期内是常量，即在对象生成时给出常量值，而在此对象生存期内，它的值是不能改变的。而对于整个类而言，不同的对象其常量数据成员的值可以不同。常量数据成员的声明是在该数据成员声明前加保留字const。例如：

```

class Test {
private:
    const int size;
public:
    Test(int sz);
    void display();
};

```

在类Test中，数据成员size就是一个常量数据成员。因为一旦对象生成，常量数据成员的值是不能变的，所以，常量数据成员的值只能在构造函数中设定。常量数据成员只能初始化，即调用构造函数，不能赋值。因此，常量数据成员的初值不能在构造函数的函数体中通过赋值设定，而只能在构造函数的初始化列表中完成。所以，类Test的构造函数可以写成

```
Test::Test(int sz) : size(sz) {
```

而不能写成

```
Test::Test(int sz) {size = sz;}
```

10.6 静态数据成员与静态成员函数

对象是类的实例，类中说明的数据成员在该类的每个对象中都有一个副本。有时，类的对象可能需要共享一些信息。例如，在一个银行系统中，最主要的对象是一个个的账户，为此需要定义一个账户类。每个账户包含的信息有账号、存入日期、存款金额等。为了计算账户的利息，还

需要知道利率。利率是每个账户对象都必须知道的信息，而且对所有账户类的对象，这个值是相同的。因此不必为每个对象设置这样的一个数据成员，只需让所有的对象共享这样一个值就可以了。如果用全局变量来表示这个共享数据，则缺乏对数据的保护。因为全局变量不受类的访问控制的限定，除了类的对象可以访问它以外，其他类的对象以及全局函数往往也能存取这些全局变量。同时也容易与其他的名字相冲突。如果可以把一个数据当作全局变量去存储，但又被隐藏在类中，并且清楚地表示与这个类的联系，那当然是最理想的。

这个功能可以用类的静态数据成员来实现。类的静态数据成员拥有一块单独的存储区，而不管创建了多少个该类的对象。所有这些对象的静态数据成员都共享这一块空间。这就为这些对象提供了一种互相通信的机制。但静态数据成员是属于类的，它的名字只在类的范围内有效，并且可以是公有的或私有的，这又使它免受其他全局函数的干扰。

10.6.1 静态数据成员的定义

在类定义中，可以把一些数据成员说明为静态的，这只需要在此数据成员前加一个保留字 `static`。例如，银行账户类可定义为

```
class SavingAccount{
private:
    char AccountNumber[20];    //账号
    char SavingDate[8];        //存款日期，格式为YYYYMMDD
    double balance;            //存款额
    static double rate;        //利率，为静态数据成员
    ...
};
```

类定义只是给出了对象构成的说明，真正的存储空间是在对象定义时分配。但由于静态数据成员是属于类而不属于对象，因此系统为对象分配空间时并不包括静态数据成员的空间。所以，静态数据成员的空间必须单独分配，而且必须只分配一次。为静态数据成员分配空间称为静态数据成员的定义。静态数据成员的定义一般出现在类的实现文件中。例如，在 `SavingAccount` 类的实现文件中，必须有如下定义：

```
double SavingAccount::rate = 0.05;
```

该定义为 `rate` 分配了空间，并给它赋了一个初值 `0.05`。如果没有这个定义，连接器会报告一个错误。

静态数据成员是属于类的，因此可以通过作用域运算符用类名直接调用。但从每个对象的角度来看，它似乎又是对象的一部分，因此又可以从对象引用它。

10.6.2 静态成员函数

像静态数据成员一样，成员函数也可以是静态的。静态成员函数用于操作静态数据成员，它是为类而不是为类的对象服务的。静态成员函数所做的操作将影响类的所有对象，而不是某一特定对象。把这样的函数放在类的内部，就不需要定义一个全局函数，减少了全局名字空间或局部

名字空间的占用。

静态成员函数的声明只需要在类定义中的函数原型前加上保留字`static`。静态成员函数的定义可以写在类定义中，也可以写在类定义的外面。在类外定义时，函数定义中不用加`static`。

静态成员函数可以用对象来调用。然而，更典型的方法是通过类名来调用，即“类名::静态成员函数名()”的形式。

静态成员函数是为类服务的，它的最大特点就是没有隐含的`this`指针，因此，静态成员函数不能访问一般的数据成员，而只能访问静态数据成员或其他静态成员函数。

定义静态成员函数的主要目的是访问静态数据成员。由于它不属于任何对象，因此，可以用它在任何对象建立之前处理静态数据成员，如初始化静态数据成员的值。这是普通成员函数不能实现的功能。例如，在10.6.1节中提到的`SavingAccount`类中有一个静态数据成员`rate`，它用来保存银行的利率，当利率发生变化时，必须修改这个静态数据成员的值，为此可以设置一个静态的成员函数

```
static void SetRate(double newRate)
{   rate = newRate; }
```

来设置新的利率。当利率发生变化时（如利率变成了5%），不管有没有账户对象存在，都可以通过调用

```
SavingAccount::SetRate(0.05)
```

将利率修改成5%。

例10.3 实现对某类对象的计数。

在程序执行的某个时刻，有时需要知道某个类已创建的对象个数，现在仍存活的对象个数。为了实现这个功能，我们可以在类中定义两个静态数据成员：`obj_count`和`obj_living`。前者保存程序中一共创建过多少对象，后者保存目前还未消亡的对象个数。要实现计数功能，可以在创建一个对象时，对这两个数各加1。当撤销一个对象时，`obj_living`减1。通过这两个静态数据成员就可以得到这两个信息。为了知道某一时刻对象个数的信息，可以定义一个静态成员函数完成此任务。这个类的定义如代码清单10-14所示，它的使用如代码清单10-15所示。

代码清单10-14 `StaticSample`类的定义

```
//文件名: StaticSample.h
//静态数据成员和静态成员函数示例
#ifndef _StaticSample_h
#define _StaticSample_h
#include <iostream>
using namespace std;

class StaticSample {
private:
    static int obj_count;
    static int obj_living;
public:
```



```

    StaticSample() {++obj_count; ++obj_living;}
    ~StaticSample() {--obj_living;}
    static void display()    //静态成员函数
    {cout << "总对象数: " << obj_count << "\t存活对象数: " << obj_living << endl;}
};
#endif

//StaticSample.cpp
#include "StaticSample.h"
int StaticSample::obj_count = 0;    //静态数据成员的定义及初始化
int StaticSample::obj_living = 0;  //静态数据成员的定义及初始化

```

代码清单10-15 StaticSample类的使用

```

//文件名: Static Sample.cpp
//Static Sample类的使用
#include "StaticSample.h"

int main()
{
    StaticSample::display();    //通过类名限定调用静态成员函数

    StaticSample s1, s2;
    StaticSample::display();

    StaticSample *p1 = new StaticSample, *p2 = new StaticSample;
    s1.display();    //通过对象调用静态成员函数

    delete p1;
    p2->display();    //通过指向对象的指针调用静态成员函数

    delete p2;
    StaticSample::display();

    return 0;
}

```

代码清单10-15所示的程序运行结果如下:

总对象数: 0	存活对象数: 0
总对象数: 2	存活对象数: 2
总对象数: 4	存活对象数: 4
总对象数: 4	存活对象数: 3
总对象数: 4	存活对象数: 2

例10.4 在一个货运系统中, 必须保存每件货物的信息。但从全局的角度看, 我们还需要知道所有货物的总重量。设计一个类, 实现上述功能。

对于每件货物来讲, 必须保存货物的重量。因此, 必须有一个保存重量的数据成员weight。由于要保存所有货物的总重量, 这个量是整个类共享的值, 因此可设置一个静态数据成员total_weight。当对象生成时, 需要将当前定义对象的重量加入到total_weight, 这个工作将由构造函数完成。对象消亡前, 必须减去当前对象的重量, 这个工作由析构函数完成。这个类的定义及成员函数的实现如下:

```

//文件名:Goods.h
//货物类的定义
#ifndef _goods_h
#define _goods_h
class Goods{
    int weight;
    static int total_weight;
public: Goods(int w);
    ~Goods();
    int weight();
    static int totalweight();
};
#endif

//文件名: Goods.cpp
//货物类的实现
#include "goods.h"

Goods::Goods(int w) {weight=w; total_weight+=w;}
Goods::~~Goods() {total_weight-=weight;}
int Goods::weight() {return weight;}
int totalweight() {return total_weight;}
int Goods::total_weight = 0;

```

10.6.3 静态常量成员

静态数据成员是整个类共享的数据成员。有时整个类的所有对象需要共享一个常量，此时可把这个成员设为静态常量成员。

注意，常量数据成员和静态常量数据成员的区别。不同对象的常量数据成员的值是不同的，而不同对象的静态常量数据成员值是相同的。

一般而言，类的数据成员不能在类定义时初始化。普通的数据成员是在对象定义时，由构造函数对它们进行初始化的。静态数据成员是在静态数据成员定义时初始化的。这个规则只有一个例外，就是对静态常量数据成员。静态常量数据成员可以并且必须在类定义时初始化。

例如，在某个类中需要用到一个数组，而该数组的大小对所有对象都是相同的，则在类中可指定一个数组规模，并创建一个该规模的数组。数组规模就可作为静态常量数据成员。这个类的定义如下：

```

class Sample {
    static const int SIZE = 10;
    int storage[SIZE];
    ...
};

```

某些旧版本的C++不支持在类中使用static const。如果想在旧环境中达到static const的效果，一个典型的解决方法就是在类中定义一个不带实例的枚举类型。枚举类型在编译时必须要有值，它出现在类中，是一个仅在类中有效的局部类型，而且枚举类型的值可用于常量表达式。

因此，旧版本的C++中，上述功能可用如下定义实现：

```
class Sample {
    enum {SIZE = 10};
    int storage[SIZE];
    ...
};
```

10.7 友元

根据数据保护的要求，类外面的函数不能访问该类的私有成员。对私有成员的访问需要通过类的公有成员函数来进行，而这有时会降低对私有成员的访问效率。在C++中，可以对某些经常需要访问类的私有成员的函数开一扇“后门”，那就是友元。友元可以在不放弃私有成员安全性的前提下，使得全局函数或其他类的成员函数能够访问类中的私有成员。

在C++的类定义中，可以指定允许某些全局函数、某个其他类的所有成员函数或某个其他类的某一成员函数直接访问该类的私有成员，它们分别被称为友元函数、友元类和友元成员函数，统称为“友元”。

友元关系是授予的而不是索取的。也就是说，如果函数 f 要成为类 A 的友元，类 A 必须显式声明函数 f 是它的友元，而不是函数 f 自称是类 A 的友元。

要将函数声明为友元，只需要在类定义中声明此函数，并在函数声明前加上关键字`friend`。此声明可以放在公有部分，也可以放在私有部分。例如，如果类 A 声明全局函数 f 是友元， f 函数的原型是

```
void f();
```

则可在类 A 的定义中加入如下形式的一条声明：

```
friend void f();
```

如果要将类 B 的成员函数

```
int func(double);
```

声明为类 A 的友元，可在类 A 的定义中加入语句

```
friend int B::func(double);
```

如果要将整个类 B 作为类 A 的友元，可在类 A 的定义中加入语句

```
friend class B;
```

尽管友元关系可以写在类定义中的任何地方，但一个较好的程序设计习惯是将所有友元关系的声明放在最前面的位置，并且不要在它的前面添加任何访问控制说明。下面是使用友元函数的一个例子。

例10.5 定义一个存储和操作女孩信息的类`Girl`，保存的信息为姓名和年龄。用友元函数实现对象的显示。

Girl类的定义、友元函数的定义以及Girl类的使用如代码清单10-16所示。友元函数是一个全局函数，它的定义可以写在类定义的外面，像普通全局函数的定义一样，也可以写在类定义的里面。

代码清单10-16 Girl类的定义及使用

```
//文件名: 10-16.cpp
//Girl类的定义及使用
#include <iostream>
#include <cstring>
using namespace std;

class Girl {
    friend void disp(Girl &x);
private:
    char name[10];
    int age;
public:
    Girl(char *n, int d) {strcpy(name,n); age=d;}
};

void disp(Girl &x) {cout<<x.name<<" "<<x.age<<endl;} //友元函数的定义

int main()
{   Girl e("abc", 15);
    disp(e);

    return 0;
}
```

代码清单10-16所示的程序将友元函数的定义写在类的外面。事实上，也可以写在类定义中，如下：

```
class Girl {
    friend void disp(Girl &x) {cout<<x.name<<" "<<x.age<<endl;} //友元函数的定义
private:
    char name[10];
    int age;
public:
    Girl(char *n, int d) {strcpy(name,n); age=d;}
};
```

代码清单10-16所示的程序的运行结果如下：

```
abc      15
```

注意，友元关系既不是对称关系，也不是传递关系。也就是说，如果类A声明了类B是它的友元，并不意味着类A也是类B的友元（友元关系是不对称的）；如果类A是类B的友元，类B是类C的友元，并不意味着类A是类C的友元（友元关系是不传递的）。

友元为全局函数或其他类的成员函数访问类的私有成员函数提供了方便，但它也破坏了类的封装，给程序的维护带来了一定的困难，因此要慎用友元。

小结

本章介绍了面向对象程序设计的基本思想。面向对象程序设计的基本思想是在考虑应用的解决方案时，首先根据应用的需求创造合适的类型，用该类型的对象来解决特定的问题。

本章主要介绍了如何定义一个类，如何通过访问控制实现封装，如何定义和使用类的对象，如何实现对象的初始化等。

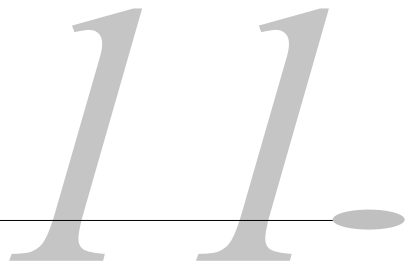
习题

简答题

1. 类与结构体的主要区别是什么？
2. 构造函数和析构函数的作用是什么？它们各有什么特征？
3. 友元的作用是什么？
4. 静态数据成员有什么特征？有什么用途？
5. 在定义一个类时，哪些部分应放在头文件（.h文件）中，哪些部分应放在实现文件（.cpp文件）中？
6. 什么情况下类必须定义自己的复制构造函数？
7. 什么样的成员函数应被说明为公有的？什么样的成员函数应被设为私有的？
8. 常量数据成员和静态常量数据成员有什么区别？如何初始化常量数据成员？如何初始化静态常量数据成员？

程序设计题

1. 创建一个LongLongInt类，用一个128个元素的字符数组存放最多128位的正整数。提供的成员函数有构造函数（根据一个由数字组成的字符串创建一个LongLongInt类的对象）、输出函数、加法函数、把一个LongLongInt类的对象赋给另一个对象的赋值函数。为了比较LongLongInt对象，提供了等于比较、不等于比较、大于比较、大于等于比较、小于比较和小于等于比较。
2. 完善本章提到的SavingAccount类。该类的属性有账号、存款金额和月利率。账号自动生成。第一个生成的对象账号为1，第二个生成的对象账号为2，依次类推。所需的操作有修改利率、每月计算新的存款额（原存款额+本月利息）和显示账户金额。
3. 试定义一个string类，用以处理字符串。它至少具有两个数据成员：字符串的内容和长度。提供的操作有显示字符串、求字符串长度、在原字符串后添加一个字符串等。
4. 为学校的教师提供一个工具，使教师可以管理自己所教班级的信息。教师所需了解和处理的信息包括课程名、上课时间、上课地点、学生名单、学生人数、期中考试成绩、期末考试成绩和平时的课堂练习成绩。每位教师可自行规定课堂练习次数的上限。考试结束后，该工具可为教师提供成绩分析，按优、良、中、差统计。



11.1 什么是运算符重载

C++定义了一组基本的数据类型和适用于这些数据类型的运算符。例如，可以用+运算符把两个整型数相加，也可以用>运算符比较两个字符的大小。这些运算符给程序员提供了简洁的符号，用来表示对基本类型的对象的操作。这些运算符在C++中是有确切的含义，并且是不可改变的。但对于用户定义的类，除了少数的运算符（如赋值、取地址、成员选择等）以外，C++并没有定义它们的含义。一般情况下，它们是不能对对象进行操作的。例如，对于我们定义的Rational类的对象r1和r2，我们不能直接用r1+r1。要把两个Rational类的对象相加，必须在Rational类中定义一个add函数实现加的功能。如果对用户定义的类的对象，也能直接用系统预定义的运算符进行操作，就可以使用户定义的类更像系统的内置类型，使用更加方便。

C++中通过运算符重载可以实现这个功能。通过运算符重载，程序员能够针对特定的类类型定义内置的运算符的运算方法。一旦在某个类中重载了某一运算符，就可以直接对这个类的对象应用此运算符。例如，如果对Rational类重载了+运算符，就可以对两个Rational类对象r1和r2直接用r1 + r2实现加运算。C++中的大多数运算符都能重载，使它们能用于类类型的对象。但常用的主要是一些算术运算符、关系运算符、赋值运算符和输入/输出运算符。

重载运算符只是解释在特定类中如何实现运算符的操作，如有理数的加运算是如何实现的，但不能改变运算符的运算对象数。一元运算符重载后还是一元运算符，二元运算符重载后还是二元运算符。不管运算符的功能和运算符的对象类型如何改变，运算符的优先级和结合性也保持不变。如果在某个类中重载了+运算符和*运算符，则意味着对该类的对象可以直接用这些运算符。例如，如果r1、r2、r3是该类的对象，则可以写如下的表达式：

```
r1 + r2 * r3
```

在计算上述表达式时，编译器会先计算*，再计算+，类的设计者不用考虑这个问题。

11.2 运算符重载的方法

运算符重载就是写一个函数解释某个运算符在某个类中的含义，这个函数可以是所重载的这

个类的成员函数或全局函数。要使得编译器在遇到这个运算符时能自动找到重载的这个函数，函数名必须要体现出和某个被重载的运算符的联系。C++中规定，重载函数名为

`operator@`

其中，@为要重载的运算符。例如，要重载+运算符，该重载函数名为`operator+`；要重载赋值运算符，函数名为`operator=`。

运算符的重载不能改变运算符的运算对象数。因此，重载函数的形式参数个数（包括成员函数的隐式指针`this`）与运算符的运算对象数相同。

大多数运算符重载函数都可以定义为全局函数或成员函数。如果作为类的成员函数，它的形式参数个数比运算符的运算对象数少1。这是因为成员函数有一个隐含的参数`this`。在C++中，隐含参数`this`是运算符的第一个参数。因此，当把一个一元运算符重载成成员函数时，该函数没有形式参数，而把一个二元运算符重载成成员函数时，该函数只有一个形式参数。

例11.1 为`Rational`类增加+和*的重载函数，用以替换现有的`add`和`multi`函数。

这两个函数都可以重载成成员函数或全局函数。当把它们重载成成员函数时，这两个函数都只有一个形式参数，就是运算符的右运算对象。运算符的左运算对象是当前对象。两个`Rational`类的对象相加或相乘的结果还是一个`Rational`类的对象，因此该函数的返回值是一个`Rational`类的对象。添加了重载函数后的`Rational`类的定义、重载函数的实现以及使用如代码清单11-1至代码清单11-3所示。

代码清单11-1 `Rational`类的定义

```
class Rational {
private:
    int num;
    int den;
    void ReductFraction();
public:
    Rational(int n = 0, int d = 1) { num = n; den = d; ReductFraction();}
    Rational operator+(const Rational &r1); //+运算符重载
    Rational operator*(const Rational &r1); /*运算符重载
    void display() { cout << num << '/' << den;}
```

代码清单11-2 `Rational`类的+和*运算符重载函数

```
Rational Rational::operator+(const Rational &r1)
{
    Rational tmp;

    tmp.num = num * r1.den + r1.num * den;
    tmp.den = den * r1.den;
    tmp.ReductFraction();

    return tmp;
}
```

```

Rational Rational::operator*(const Rational &r1)
{
    Rational tmp;

    tmp.num = num * r1.num;
    tmp.den = den * r1.den;
    tmp.ReductFraction();

    return tmp;
}

```

代码清单11-3 重载了加和乘运算后的Rational类的使用

```

//文件名: 11-3.cpp
//重载了+和*运算符的Rational类
#include <iostream>
using namespace std;

#include "Rational.h"

int main()
{
    Rational r1(1,6), r2(1,6), r3;

    r3 = r1 + r2; //r3 = r1.operator+(r2)
    r1.display(); cout << " + "; r2.display(); cout << " = "; r3.display(); cout <<
endl;

    r3 = r1 * r2; //r3 = r1.operator*(r2)
    r1.display(); cout << " * "; r2.display(); cout << " = "; r3.display(); cout <<
endl;

    return 0;
}

```

除了可以用内置运算符进行运算外，运算符重载函数也可以直接用函数的形式调用。例如， $r3 = r1 + r2$ 也可以写成 $r3 = r1.operator+(r2)$ 。但一般情况下，都表示成前者。

加和乘也可以重载成全局函数。由于重载函数主要是对对象的数据成员进行操作，而在一般的类定义中，数据成员都被定义成私有的。所以，当运算符被重载成全局函数时，通常把此重载函数设为类的友元函数，便于访问类的私有数据成员。用全局函数重载的Rational类的定义和实现如代码清单11-4和代码清单11-5所示。它的使用和用成员函数实现的重载完全一样。

代码清单11-4 用友元函数重载的Rational类的定义

```

class Rational {
    friend Rational operator+(const Rational &r1, const Rational &r2); //+运算符重载
    friend Rational operator*(const Rational &r1, const Rational &r2); //*运算符重载

private:
    int num;
    int den;
}

```



```

void ReductFraction();

public:
    Rational(int n = 0, int d = 1) { num = n; den = d; ReductFraction();}
    void display() { cout << num << '/' << den;}
};

```

代码清单11-5 用友元函数重载的Rational类的加和乘运算

```

Rational operator+(const Rational &r1, const Rational &r2)
{
    Rational tmp;
    tmp.num = r1.num * r2.den + r2.num * r1.den;
    tmp.den = r1.den * r2.den;
    tmp.ReductFraction();
    return tmp;
}

Rational operator*(const Rational &r1, const Rational &r2)
{
    Rational tmp;
    tmp.num = r1.num * r2.num;
    tmp.den = r1.den * r2.den;
    tmp.ReductFraction();
    return tmp;
}

```

大多数运算符都可以重载成成员函数或全局函数。但是赋值运算符(=)、下标运算符([])、函数调用运算符(())和成员访问运算符(->)必须重载成成员函数，因为这些运算符的第一个运算对象必须是相应类的对象，定义成成员函数可以保证第一个运算对象的正确性。如果第一个运算对象不是相应类的对象，编译器能检查出此错误。具有赋值意义的运算符，如复合的赋值运算符以及++和--，不一定非要重载为成员函数，但最好重载为成员函数。具有两个运算对象的运算符最好重载为全局函数，这样可以使应用更加灵活。如果在Rational类中把加运算定义成全局函数，r是Rational类的对象，则2+r是一个合法的表达式。在执行这个表达式时，系统会调用作为全局函数的operator+函数，并进行参数传递。在参数传递时，如果形式参数和实际参数类型不同的话，会进行自动类型转换，系统首先调用Rational类的构造函数，将2作为参数，生成了一个num=2，den取默认值1的临时对象，并把这临时对象作为operator+的第一个参数。因此，这是一个合法的函数调用。但当把加重载为成员函数时，2+r等价于调用函数2.operator+(r)，而2不是Rational类的对象，这个调用是非法的，编译器会报错。

11.3 几个特殊运算符的重载

11.3.1 赋值运算符的重载

对任一个类，如果用户没有自定义赋值运算符函数，那么系统会为其生成一个默认的赋值运算符重载函数，在对应的数据成员间赋值。一般情况下，这个默认的赋值运算符重载函数能满足

用户的需求。但是，当类含有类型为指针的数据成员时，可能会带来一些麻烦。回顾前面讲过的IntArray。如果有两个IntArray类的对象array1和array2，我们可以用array1 = array2对array1进行赋值。但这个赋值会引起下面几个问题。

(1) 会引起内存泄漏。在定义array1时，构造函数会根据给出的数组的上下界为array1动态申请一块空间，该空间的首地址存储在数据成员storage中。当执行array1 = array2时，会把array2的storage赋给array1的storage，而array1原有的存储空间就丢失了。

(2) 执行array1 = array2后，array1的storage和array2的storage指向同一块空间，也就是说，当一个数组修改了它的元素后，另一个数组的元素也被修改了，使得两个对象的操作相互影响。

(3) 当这两个对象析构时，先析构的对象会释放存储数组元素的空间，而当一个对象析构时，则无法释放存放数组元素的空间。特别是当两个对象的作用域不同时，一个对象的析构将使另一个对象无法正常工作。

在这种情况下，类的设计者必须写一个赋值运算符重载函数。

赋值运算是二元运算。第一个运算数是赋值号左边的对象，第二个运算数是赋值号右边的表达式。由于第一个运算数必须是类的对象，因此C++规定赋值运算符必须重载为成员函数，IntArray类的赋值运算符重载函数的设计如下：

```
IntArray &IntArray::operator=(const IntArray & a)
{
    if (this == &a) return *this; //防止自己复制自己
    delete [ ] storage; //归还空间
    low = a.low;
    high = a.high;
    storage = new int[high - low + 1]; //根据新的数组大小重新申请空间
    for (int i=0; i <= high - low; ++i) storage[i] = a.storage[i]; //复制数组元素
    return *this;
}
```

一般来讲，需要自定义复制构造函数的类也需要自定义赋值运算符重载函数。但要注意复制构造函数和赋值运算符重载函数是有区别的，主要在于它们的使用场合不同。复制构造函数用于创建一个对象时，用另一个已存在的同类对象对其进行初始化。对于两个已存在的对象，可以通过赋值运算用一个对象的值来改变另一个对象的值。例如，若r1是Rational类的对象，则

```
Rational r2 = r1;
```

调用的是复制构造函数，尽管用的是一个=号；而对于Rational类的对象r3，执行

```
r1 = r3;
```

调用的是赋值运算符重载函数。

对任一类A，它的赋值运算符重载函数的原型如下：

```
A &operator=(const A &a);
```

初学者常常感到困惑的一个问题是：在赋值运算符重载函数中，已经将参数的值赋值给了当前对象，完成了赋值的任务，那为什么还需要返回值呢？记住，在C++中，赋值是一个运算，它

可以构成一个表达式，而该表达式的结果值就是赋给左边的对象的值。因此，赋值运算符重载函数必须返回赋给左边的对象的值。

11.3.2 下标运算符的重载

对于IntArray类，能否使这个类像普通数组那样通过下标运算对数组的成员进行操作呢？这样可以使IntArray类像一个功能更强的数组。在C++中，这可以通过重载下标运算符（[]）来实现。下标运算符是一个二元运算符。第一个运算数是当前对象，第二个运算数是下标值。C++规定下标运算符必须重载成成员函数，因此，下标运算符重载函数的原型如下：

```
数组元素类型 &operator[]( int 下标);
```

对IntArray类，我们可以定义下面的下标运算符重载函数来访问某一个数组成员：

```
int & IntArray::operator[](int index)
{ if (index < low || index > high) {cout << "下标越界"; exit(-1); }
  return storage[index - low];
}
```

这个函数有一个参数，就是数组的下标，返回的是该下标对应的数据元素。由于该函数采用的是引用返回，也就是说，函数调用和某一数组元素是对应的。有了这个重载函数，就可以把IntArray类的对象当作普通数组一样使用。如果有定义

```
IntArray array(20,30);
```

则array是一个11个元素的数组，它的下标从20到30。要给这个数组赋值，可以像普通数组一样用一个for循环实现：

```
for (i=20; i<=30; ++i) {
    cout << "请输入第" << i << "个元素:";
    cin >> array[i];
}
```

要输出这个数组的所有元素，也可以像普通数组一样输出：

```
for (i=20; i<=30; ++i)      cout << array[i] << '\t';
```

11.3.3 ++和--运算符的重载

++和--都是一元运算符，它们可被重载成成员函数或友元函数。但因为这两个运算符改变了运算对象的状态，所以更倾向于将它们作为成员函数。

在考虑重载++和--运算符时，还必须注意一个问题。作为内置运算符，++和--有两种用法，它们既可以作为前缀使用，也可以作为后缀使用。而且这两种用法的结果是不一样的：作为前缀使用时，返回的是修改以后的对象值；而作为后缀使用时，返回的是修改以前的对象值。为了与内置类型一致，重载后的++和--也应具有这个特性。为此，对于++和--运算，每个运算符必须提供两个重载函数。一个处理前缀运算，一个处理后缀运算。

但问题是，处理++的两个重载函数的形式参数个数和类型是完全相同的，处理--的两个重载函数的形式参数个数和类型也是完全相同的。普通的函数重载无法区分这两个函数。

为了解决这个问题，C++规定后缀运算符重载函数接受一个额外的（即无用的）int型的形式参数。使用后缀运算符时，编译器用0作为这个参数的值。当编译器收到一个前缀表示的++或--时，调用正常重载的这个函数；如果收到的是一个后缀表示的++或--，则调用有一个额外参数的重载函数。这样就把前缀和后缀的重载区分开了。

例11.2 设计一个会报警的计数器类。该计数器从0开始计数，当到达预先设定好的报警值时，计数器会发出报警消息，计数器的值不再增加。

这个计数器类需要两个数据成员：一个保存计数器的值，一个保存报警值。它只有一个功能，就是计数并在计数值到达报警值时报警。计数的功能就是计数器加1，因此这个类必须有++操作。除此之外，还需要一个构造函数和一个显示函数。构造函数设置计数器的初值和报警值。显示函数显示计数器的值。该类的定义、实现和使用如代码清单11-6到代码清单11-8所示。

代码清单11-6 Counter类的定义

```
class Counter {
    int value; //计数器的值
    int alarm; //报警值
public:
    Counter(int a) {value = 0; alarm = a;}
    Counter & operator++(); //前缀的++重载
    Counter operator++(int); //后缀的++重载
    void print() {cout << value << endl; }
};
```

代码清单11-7 ++的两个重载函数的定义

```
Counter & Counter::operator++()
{
    if (value == alarm) cout << "已超过报警值\n";
    else { ++value;
        if (value == alarm) cout << "已到达报警值\n";
    }
    return *this;
}

Counter Counter::operator++(int x)
{
    Counter tmp = *this; //保存对象修改前的状态
    if (value == alarm) cout << "已超过报警值\n";
    else { ++value;
        if (value == alarm) cout << "已到达报警值\n";
    }
    return tmp; //返回修改前的状态
}
```

代码清单11-8 Counter类的使用

```
int main()
```

```

{   Counter cnt(3);           //定义一个Counter类的对象，报警值为3

    cnt.print();              //显示对象的当前值，此时输出为0
    ++cnt;
    cnt.print();              //此时输出为1
    (++cnt).print();           //调用前缀的++，输出2
    (cnt++).print();           //调用后缀的++，当前对象的value已经加1，报警，但输出的是2
    cnt.print();              //输出值为3
    return 0;
}

```

前缀的++运算符的实现很容易理解。首先检查value是否已经到达报警值。如果已经到达，则value值不变，再次报警；否则，value加1，再次检查是否到达报警值，如果到达了，则报警。

前缀的++运算符返回当前Counter对象的引用（也就是刚刚自增后的对象）。因为当前对象*this是作为Counter &返回的，这使得前缀自增的Counter对象可以作为左值来使用。正如内置的自增运算符对基本类型的工作方式一样。

后缀的++比较麻烦。为了返回自增前的对象值，我们定义了一个临时对象tmp，保存当前对象的值，然后再修改当前对象，最后返回的是临时对象tmp的值，也就是当前对象修改前的值。请注意，该函数不能返回对tmp的引用，因为tmp是局部变量，在函数执行结束时被删除。

11.3.4 重载函数的原型设计考虑

在前面的例子中，可以看到各种不同的参数传递方式和返回方式。乍一看，有些让人摸不着头脑。虽然可以用任何可行的方式传递参数和返回类型，但在这些例子中所用的方式却不是随意选的。它们遵守一种合乎逻辑的模式，在大多数情况下，我们都应该选择这种模式。下面是一些常用的规则。

(1) 对于任何函数的参数，如果仅需要从参数中读，而不改变它，一般用const引用来传递。普通的算术运算符（如+、-等）和布尔运算不会改变参数，所以const引用是主要的传递方式。当函数是类的成员函数时，就转换为常量成员函数。只有会修改参数的运算符，如自增或自减运算符，在重载成全局函数时，用引用传递，在重载成成员函数时，重载成非常量成员函数。

(2) 返回值的类型取决于运算符的具体含义。如果该运算符的结果产生一个新值，就需要产生一个作为返回值的新对象。例如，operator+必须生成一个操作数之和的新对象。这个对象作为函数的返回值返回。

(3) 所有的赋值运算符（如=、+=等）均改变左值。为了使赋值结果能用于链式表达式（如a = b = c），应该能够返回一个刚刚改变了的左值的引用。但这个引用应该是常量还是非常量呢？虽然我们是从左到右读表达式a = b = c，但编译器是从右到左分析这个表达式，所以并非一定要返回一个非常量来支持链式赋值。然而，人们有时候希望能够对刚刚赋值的对象进行运算，如(a = b).func()，而函数func可能会修改当前对象的值，因此，所有的赋值运算符的返回值对

于左值应该是非常量引用。

(4) 对于逻辑运算符，人们希望至少得到一个int型或bool型的返回值。

因为有前缀和后缀两种版本，所以++和--运算符出现了两难的局面。由于两个版本都改变了对象，所以这个对象不能作为常量类型。在对象改变以后，前缀版本返回改变后的对象值，因此只需要作为一个引用返回*this。因为后缀版本返回改变之前的值，所以必须创建一个存储这个值的对象，并返回它。因此，后缀的++和--必须通过传值的方式返回。

通过传值方式返回时，通常会用如下形式：

```
return Point(left.x + right.x, left.y + right.y);
```

假设Point是处理二维平面上的点的类。初看起来，这个返回和下列语句是等价的：

```
Point tmp;
tmp.x = left.x + right.x;
tmp.y = left.y + right.y;
return tmp;
```

但是，这两种方式的效率是不同的。前者的意思是“根据括号内的参数值创建一个临时对象，并返回它”；而后者的意思是，先创建了一个对象tmp（这将调用构造函数），然后对tmp赋值，最后返回tmp。而在返回tmp时，又要创建一个临时对象，并调用复制构造函数用tmp对它进行初始化。在函数执行结束时，还要调用析构函数析构tmp。可以看出，前者的效率远远高于后者。这种方式常称为返回值优化。

11.3.5 输入/输出运算符的重载

借助于流插入运算符(>>)和流提取运算符(<<), C++能够输入和输出基本类型的数据。流插入运算符和流提取运算符也能重载, 使用户自定义类的对象也能像基本类型的数据一样直接用cin和cout输入和输出。输入/输出运算符必须被重载成全局函数, 一般应将此全局函数声明为类的友元。

1. 输出运算符<<的重载

输出运算符重载函数有两个参数：第一个参数是ostream类的一个引用，第二个参数是对当前类的常量对象的引用。它的返回类型是ostream类的一个引用。输出运算符重载函数的框架如下：

```
ostream & operator<<(ostream & os, const ClassType &obj)
{
    os << 要输出的内容;
    return os;
}
```

这个函数将对象产生的输出写入输出流对象os，然后返回os对象。

例如，对于Rational类，我们可以为它写一个输出运算符重载函数，这个函数必须声明为Rational类的友元：

```
ostream& operator<<(ostream &os, const Rational& obj) //输出重载函数
{
    os << obj.num << '/' << obj.den;
    return os;
}
```

```
}
```

有了输出运算符重载函数，就可以将Rational类的对象直接用cout输出。例如，定义

```
Rational r(2,6);
```

执行cout << r;的结果是1/3。

2. 输入运算符>>的重载

与输出运算符类似，输入运算符重载函数的第一个参数也是一个引用，指向它要读的流，并且返回的也是对同一个流的引用；第二个形式参数是对要读入的对象的非常量引用，该形参必须是非常量的，因为输入运算符重载函数的目的是要将数据读入此对象。输入运算符重载函数的框架如下：

```
istream & operator>>(istream & is, ClassType &obj)
{   is >> 对象的数据成员;
    return is;
}
```

例如，对于Rational类，我们可以为它写一个输入运算符重载函数，与<<重载同理，这个函数必须声明为Rational类的友元：

```
istream& operator>>(istream &in, Rational& obj) //输入重载函数
{   in >> obj.num >> obj.den;
    obj.ReductFraction();
    return in;
}
```

有了输入运算符重载函数就可以将Rational类的对象直接用cin输入。例如，定义

```
Rational r;
```

可以用cin >> r从键盘输入r的数据。例如，输入

```
1 3
```

执行cout << r;的结果是1/3。

11.4 自定义类型转换函数

如第2章所述，系统的内置类型在进行操作时经常会发生自动类型转换。例如，计算表达式

```
3 + 'a' - 4.3
```

时，系统会将3和'a'转换成double类型。再如，如果x是一个整型变量，那么在执行

```
x = 4.3
```

时也会发生自动类型转换，将4.3转换成整型数4，然后再赋给变量x。

对于系统的内置类型，编译器知道如何在这些基本类型之间进行转换，那么用户自定义的类型又如何呢？编译器预先并不知道在用户自定义的类型之间、用户自定义类型和系统的内置类型之间如何进行转换，因此类的设计者必须详细说明该怎样做。

构造函数实现了内置类型到类类型的转换。例如，对于Rational类的对象r，可以执行r=2。此时，编译器隐式地调用Rational类的构造函数，传给它一个参数2。构造函数将构造出一个num=2，den=1的Rational类的对象，并将它赋给r。那么当类类型的对象要转换为内置类型或其他类类型时，该如何做呢？这可以通过定义类型转换函数来实现。

类型转换函数是一类特殊的类成员函数，它定义了如何将类类型转换成其他类型。类型转换函数的格式如下：

```
operator 目标类型名 () const
{
    ...
    return (结果为目标类型的表达式);
}
```

类型转换函数不指定返回类型，因为返回类型就是目标类型名。类型转换函数也没有形式参数，它的参数就是当前对象。这个函数也不会修改当前对象值，因此是一个常量成员函数。例如，我们可以为Rational类增加一个类型转换函数，使Rational类的对象可以转换成一个double类型的对象。函数如下：

```
operator double () const { return (double(num)/den);}
```

有了这个函数，我们可以将一个Rational类的对象r赋给一个double类型的变量x。例如，r的值为(1, 3)，经过赋值x = r后，x的值为0.333333。如果希望Rational类的对象也能转换成其他类型，那么可以再定义相应的转换函数。

11.5 运算符重载的应用

11.5.1 完整的 Rational 类的定义和使用

经过了运算符重载，我们的Rational类与系统的内置类型基本类似。增加了这些重载函数后的Rational类的完整定义和实现如代码清单11-9和代码清单11-10所示。

代码清单11-9 Rational类的完整定义

```
//文件名: Rational.h
//Rational类的完整定义
#ifndef _rational_h
#define _rational_h
#include <iostream.h>

class Rational {
    friend istream& operator>>(istream &in, Rational& obj); //输入重载函数
    friend ostream& operator<<(ostream &os, const Rational& obj); //输出重载函数
    friend Rational operator+(const Rational &r1, const Rational &r2); //+运算符重载
    friend Rational operator*(const Rational &r1, const Rational &r2); //*运算符重载

private:
    int num;
```



```

        int den;

        void ReductFraction();

public:
    Rational(int n = 0, int d = 1) { num = n; den = d; ReductFraction();}
    operator double () const { return (double(num)/den);}

};
#endif

```

代码清单11-10 Rational类的实现

```

//文件名: Rational.cpp
//Rational类的完整实现
#include <iostream.h>
#include "Rational.h"

Rational operator+(const Rational &r1, const Rational &r2) //+重载
{Rational tmp;
 tmp.num = r1.num * r2.den + r2.num * r1.den;
 tmp.den = r1.den * r2.den;
 tmp.ReductFraction();
 return tmp;
}

Rational operator*(const Rational &r1, const Rational &r2) /*重载
{Rational tmp;
 tmp.num = r1.num * r2.num;
 tmp.den = r1.den * r2.den;
 tmp.ReductFraction();
 return tmp;
}

void Rational::ReductFraction() //化简有理数
{int tmp = (num > den) ? den : num;

 for (; tmp > 1; --tmp)
    if (num % tmp == 0 && den % tmp == 0) {num /= tmp; den /= tmp; break;}
}

istream& operator>>(istream &in, Rational& obj) //输入重载函数
{ in >> obj.num >> obj.den;
 obj.ReductFraction();
 return in;
}

ostream& operator<<(ostream &os, const Rational& obj) //输出重载函数
{os << obj.num << '/' << obj.den;
 return os;
}

```

这个Rational类的某次使用如代码清单11-11所示。

代码清单11-11 Rational类的使用

```
//文件名: 11-11.cpp
//Rational类的使用
#include <iostream.h>
#include "Rational.h"

int main()
{
    Rational r1, r2, r3;
    double x;

    cout << "输入r1: "; cin >> r1; //直接用cin输入有理数
    cout << "输入r2: "; cin >> r2;

    r3 = r1 + r2; //调用operator+实现加运算
    cout << r1 << '+' << r2 << " = " << r3 << endl; //直接用cout输出有理数

    r3 = r1 * r2; //调用operator*实现乘运算
    cout << r1 << '*' << r2 << " = " << r3 << endl;

    x = r3; //调用类型转换函数
    cout << "r3的double表示: " << x << endl;

    return 0;
}
```

代码清单11-11所示的程序的某次运行结果如下:

```
输入r1: 1 3
输入r2: 2 6
1/3+1/3=2/3
1/3*1/3=1/9
r3的double表示: 0.111111
```

11.5.2 完整的 IntArray 类的定义和使用

经过了运算符重载, 我们的IntArray类几乎和系统的内置整型数组一样使用。增加了这些运算符重载后的IntArray类的定义和实现如代码清单11-12和代码清单11-13所示。

代码清单11-12 IntArray类的完整定义

```
//文件名: IntArray.h
//IntArray类的定义
#ifndef _array_h
#define _array_h

#include <iostream.h>

class IntArray
{
```

```

friend ostream &operator<<(ostream &os, const IntArray &obj);
friend istream &operator>>(istream &is, IntArray &obj);
friend bool operator==(const IntArray &obj1, const IntArray &obj2);

private:
    int low;
    int high;
    int *storage;

public:
    //根据low和high为数组分配空间。分配成功，返回值为true,否则为false
    IntArray(int lh = 0, int rh = 0):low(lh),high(rh)
    {storage = new int [high - low + 1]; }

    //复制构造函数
    IntArray(const IntArray &arr);

    //赋值运算符重载函数
    IntArray &IntArray::operator=(const IntArray &a);

    //下标运算符重载函数
    int & operator[](int index);
    const int &operator[](int index) const;

    //获取数组规模
    void getSize() {cout << "下标范围为: [" << low << ", " << high << "]\n";}

    //回收数组空间
    ~IntArray() {delete [] storage; }
};

#endif

```

代码清单11-13 IntArray类的实现

```

//文件名: IntArray.cpp
//IntArray类的实现
#include <cassert>
#include "IntArray.h"

IntArray::IntArray(const IntArray &arr)
{low = arr.low; high = arr.high;
 storage = new int [high - low + 1];
 for (int i = 0; i < high - low + 1; ++i) storage[i] = arr.storage[i];
}

IntArray &IntArray::operator=(const IntArray &a)
{ if (this == &a) return *this; //防止自己复制自己
 delete [] storage; //归还空间
 low = a.low; high = a.high;
 storage = new int[high - low + 1]; //根据新的数组大小重新申请空间
 for (int i=0; i <= high - low; ++i) storage[i] = a.storage[i]; //复制数组元素
 return *this;
}

```

```

}

int & IntArray::operator[](int index)
{ assert(index >= low && index <= high);
  return storage[index - low];
}

const int & IntArray::operator[](int index) const
{ assert(index >= low && index <= high);
  return storage[index - low];
}

ostream &operator<<(ostream &os, const IntArray &obj)
{ os << "数组内容为: \n";
  for (int i=obj.low; i<=obj.high; ++i) os << obj[i] << '\t';
  os << endl;
  return os;
}

istream &operator>>(istream &is, IntArray &obj)
{ cout << "请输入数组元素[" << obj.low << ", " << obj.high << "]:\n";
  for (int i=obj.low; i<=obj.high ; ++i)   is >> obj[i] ;
  return is;
}

bool operator==(const IntArray &obj1, const IntArray &obj2)
{ if (obj1.low != obj2.low || obj1.high != obj2.high) return false;
  for (int i = obj1.low; i<=obj1.high; ++i)
    if (obj1[i] != obj2[i]) return false;
  return true;
}

```

这个IntArray类的某次使用如代码清单11-14所示。

代码清单11-14 IntArray的使用

```

//文件名: 11-14.cpp
//IntArray类的使用
#include "IntArray.h"

int main()
{   IntArray array1(20,30), array2;

    cin >> array1; //利用流提取运算符重载输入array1
    cout << "array1 "; cout << array1; //利用流插入运算符重载输出array1

    array2 = array1; //利用赋值运算符重载将array1赋给array2

    cout << "执行 array2 = array1, array2 ";
    cout << array2;

    //利用==重载比较array1和array2
    cout << "array1 == array2 是 " << ((array1 == array2) ? "true" : "false") << endl;

    array2[25] = 0; //利用下标运算符重载为array2的元素赋值

    cout << "执行array[25] = 0后, array1 == array2 是 "

```

```
<< ((array1 == array2) ? "true" : "false") << endl;  
  
    return 0;  
}
```

代码清单11-14所示的程序的某次执行结果如下:

```
请输入数组元素[20, 30]:  
1 2 3 4 5 6 7 8 9 10 11  
array1的内容为:  
1 2 3 4 5 6 7 8 9 10 11  
执行 array2 = array1, array2的内容为:  
1 2 3 4 5 6 7 8 9 10 11  
array1 == array2是true  
执行array2[25] = 0后, array1 == array2是false
```

小结

在本章中,我们学习了如何通过定义运算符重载函数构建功能更加强大的类。运算符重载可以使类的用户将类的对象像C++的内置类型的变量一样操作。我们介绍了运算符重载的基本概念以及C++对运算符重载的一些限制。让读者了解什么时候该重载成成员函数,什么时候该重载成全局函数,并讨论了这两种函数重载的不同之处。通过运算符重载还能实现类类型和内置类型及其他类类型之间的转换。

习题

简答题

1. 重载后的运算符的优先级和结合性与用于内置类型时有何区别?
2. 什么时候会用到new和delete的重载?
3. 如何区分++和--的前缀用法和后缀用法重载函数?
4. 为什么要使用运算符重载?
5. 如果类的设计者在定义一个类时没有定义任何成员函数,那么这个类有几个成员函数?

程序设计题

1. 定义一个时间类Time,通过运算符重载实现时间的比较(关系运算)、时间增加/减少若干秒(+=和-=)、时间增加/减少1秒(++和--)、计算两个时间相差的秒数(-)以及输出时间对象的值(时-分-秒)。
2. 用运算符重载完善第10章程序设计题的第1题中的LongLongInt类。
3. 定义一个保存和处理N维向量空间中的向量的类型,能实现向量的输入/输出、两个向量的加以及求两个向量点积的操作。

第 12 章

组合与继承

12

C++的重要特征之一是代码重用。代码重用不仅仅是指将一段代码复制到程序的另一个地方或另一个程序中，还可以是指利用别人已经创建好的类构建出功能更强大的类。

C++中有两种方法可以完成这个任务。第一种方法很简单，用已有的类的对象作为新定义类的数据成员。因为新的类是已有类的对象组合而成，所以这种方法被称为组合。第二种方法是在一个已存在类的基础上，对它进行扩展，形成一个新类。这种方法称为继承。继承是面向对象程序设计的重要思想，也是运行时多态性的实现基础。

12.1 组合

实际上，创建类的基础工作就是将不同类型的数据组合成一个更复杂的类型。不过到现在为止，我们都是将系统的内置类型组合成一个新类型。其实，使用用户定义的类型组合一个新类也是很容易的，只是把用户定义类的对象作为新类的数据成员而已。

组合表示一种聚集关系，是一种部分和整体的关系。例如，每架飞机都必须有一个发动机，因此，飞机类将包含一个发动机类的对象。

对于含有对象成员的类，它的构造函数有一些限制。因为大多数对象成员不能像内置类型一样直接赋值，对象的初始化是通过构造函数完成的，所以，如果一个类含有对象成员，而新类对象在初始化时也不想用默认构造函数去初始化对象成员，那么必须用初始化列表去初始化对象成员。

例如，我们要定义一个复数类，而复数的虚部和实部都用有理数表示，则复数类是由有理数类组合而成的。它的定义如代码清单12-1所示。

代码清单12-1 Complex类的定义

```
class Complex
{
    friend Complex operator+(const Complex &x, const Complex &y);
    friend istream& operator>>(istream &is, Complex &obj);
    friend ostream& operator<<(ostream &os, const Complex &obj);

    Rational real; //实部
    Rational imag; //虚部
}
```

```
public:
    Complex(int r1 = 0, int r2 = 1, int i1 = 0, int i2 = 1):real(r1, r2), imag(i1, i2) {}
};
```

Complex类有两个数据成员real和imag，都是Rational类型的，分别表示复数的实部和虚部。有一个公有的成员函数，就是构造函数。这个类提供的其他运算有加法和输入/输出，都采用运算符重载的方法来实现。这3个运算符都重载成了友元函数。

注意Complex类的构造函数。由于real和imag都是Rational类型，无法直接赋值，因此这两个数据成员都必须在初始化列表中用它们的构造函数设置初值，构造函数所需的参数来自于Complex类的构造函数的参数表。除了这两个数据成员外，Complex类没有其他的数据成员要被置初值，所以构造函数的函数体为空。

如果对象成员的初始化是由默认构造函数完成的，则该对象成员可以不出现在初始化列表中。

复数的加法是实部和虚部对应相加。由于对Rational类重载过加法，因此可直接将实部与虚部对应相加。输入/输出一个复数就是分别输入/输出它的实部和虚部，由于Rational类重载过输入/输出，因此可直接输入/输出它的实部和虚部。总而言之，由于Rational类重载了这些运算符，我们在设计Complex类时就可以重用这些代码，以简化我们的实现。这3个重载函数的实现如代码清单12-2所示。

代码清单12-2 Complex类的重载函数的实现

```
Complex operator+(const Complex &x, const Complex &y) //加法运算符重载
{
    Complex tmp;
    tmp.real = x.real + y.real;    //利用Rational类的加法重载函数完成两个实部的相加
    tmp.imag = x.imag + y.imag;    //利用Rational类的加法重载函数完成两个虚部的相加
    return tmp;
}

istream& operator>>(istream &is, Complex &obj)    //输入运算符重载
{
    cout << "请输入实部: ";
    is >> obj.real;    //利用Rational类的输入重载实现实部的输入
    cout << "请输入虚部: ";
    is >> obj.imag;    //利用Rational类的输入重载实现虚部的输入
    return is;
}

ostream& operator<<(ostream &os, const Complex &obj)    //输出运算符重载
{
    //利用Rational类的输出重载实现实部和虚部的输出
    cout << '(' << obj.real << " + " << obj.imag << "i" << ')';
    return os;
}
```

有了这样的一个Complex类，就可以将复数对象像内置类型一样使用，见代码清单12-3。

代码清单12-3 Complex类的使用

```

//文件名: 12-3.cpp
//Complex类的使用
int main()
{Complex x1,x2,x3;

    cout << "请输入x1: \n"; cin >> x1;    //利用输入重载输入复数x1
    cout << "请输入x2: \n"; cin >> x2;    //利用输入重载输入复数x2

    x3 = x1 + x2;    //利用加运算符重载完成加法
    cout << x1 << " + " << x2 << " = " << x3 << endl;    //利用输出重载输出复数

    return 0;
}

```

代码清单12-3所示的程序的某次运行结果如下:

```

请输入x1:
请输入实部: 1 4
请输入虚部: 2 5
请输入x2:
请输入实部: 1 4
请输入虚部: 3 5
(1/4+ 2/5i) + (1/4+3/5i) = (1/2 + 1/1i)

```

12.2 继承

继承是软件重用的另一种方式。通过继承,程序员可以利用现有类的数据和行为来创建新类,并增加新的数据和行为来增强此类。软件重用能够节省软件开发的代价,因此希望程序员能够重用经过认可和调试的高质量的软件。

创建新类时,并不一定需要创建全新的数据成员和成员函数。我们可以指明这个新类应当继承现有的某个类的成员。这时,现有的类称为基类,继承实现的新类称为派生类。派生类本身也可能会成为未来派生类的基类。如果派生类只有一个基类,则称为单继承。如果派生类是从多个基类派生出来的,这些基类之间可能毫无关系,则称为多继承。

派生类通常添加了其自身的数据成员和成员函数,因而通常比基类大得多。派生类比基类更具体,它代表了一组外延较小的对象。对于单继承,派生类和基类有相同的起源。继承的真正魅力在于能够添加基类所没有的特点以及取代和改进从基类继承来的特点。

继承机制除了可以支持软件重用之外,它还具有以下作用。

(1) 对事物进行分类。通过类之间的继承关系,可以把事物(概念)以层次结构表示出来。在这个层次结构中,存在着一种一般与特殊的关系。其中,上层表示一般的概念,下层表示一个特殊的概念,它是上层的具体化。如图10-1中所示,“人”是一个一般的概念,而“教师”是一类特殊的“人”。相对于“高级职称”的教师,“教师”又是一个比它更一般的概念。

(2) 支持软件的增量开发。软件的开发往往不是一次完成的,而是随着对软件功能的逐步理

解和改进而不断完善的。继承关系可以用来实现这个完善的过程。当需要扩充功能时，可以通过扩充类的功能来实现，而使整个系统程序的修改量达到最少。

(3) 对概念进行组合。用多继承表示概念的组合。例如，在学校中有一类特殊的人，他们既是教师，又是学生，他们是正在攻读博士学位的在职教师，称为“在职博士生”。在职博士生具有教师的所有特性，又具有博士生的所有特性，这个类可以从教师类和博士生类继承。

12.2.1 单继承

在定义单继承时，派生类只能有一个基类。它的定义格式如下：

```
class 派生类名:[继承方式] 基类名
{ 新增加的成员声明; }
```

其中，派生类名是正在定义的类的名字；基类名是派生类的直接基类的名字；继承方式可以是public、private和protected，它说明了基类的成员在派生类中的访问特性，继承方式可以省略，默认为private；新增加的成员声明是派生类在基类基础上的扩充。例如：

```
class Base
{
    int x;
public:
    void setx(int k);
}

class Derived1:public Base
{
    int y;
public:
    void sety(int k);
}
```

类derived1有两个数据成员x和y，有两个成员函数setx和sety。

12.2.2 基类成员在派生类中的访问特性

第10章介绍了类成员的访问特性。公有成员能够被程序中所有函数访问，私有成员只能被自己的成员函数和友元访问，即使是派生类的成员函数也不能访问基类的私有成员。

有了继承以后，我们引入第三种访问特性protected。protected访问特性介于public访问和private访问之间。protected成员是一类特殊的私有成员，它不可以被全局函数或其他类的成员函数访问，但能被派生类的成员函数和友元函数访问。派生类成员简单地使用成员名就可以引用基类的public成员和protected成员。但是，protected成员破坏了类的封装，基类的protected成员改变时，所有派生类都要修改。

派生类不能直接访问基类的私有成员。其他成员的访问特性取决于它在基类中的访问特性和继承方式。继承基类的方式有3种：public、protected和private。protected继承和private继承不常用，而且使用时必须相当小心。

从基类public派生某个类时，基类的public成员会成为派生类的public成员，基类的

protected成员成为派生类的protected成员。派生类永远也不能直接访问基类的private成员，但可通过基类的public或protected成员函数间接访问。

从基类protected派生一个类时，基类的public成员和protected成员成为派生类的protected成员。

从基类private派生一个类时，基类的public成员和protected成员成为派生类的private成员。

派生类中基类成员的访问特性可参见表12-1。

表12-1 派生类中基类成员的访问特性

基类成员的访问特性	继承类型		
	public继承	protected继承	private继承
public	public	protected	private
protected	protected	protected	private
private	不可访问	不可访问	不可访问

根据表12-1，可以看出Derived1类的组成及访问特性如表12-2所示。

表12-2 Derived1类的访问特性

访问特性	Derived1
不可访问	Int x
private	Int y
public	Setx() Sety()

继承时，也可以不写继承类型。对于用struct定义的类型，默认时为public继承。对于class定义的类型，默认时为private继承。

一般情况下，采用的都是public继承。它可以在派生类中保持基类的访问特性。另外两种派生方法很少使用。

例12.1 定义一个二维平面上的点类型，可以设置点的位置和获取点的位置。在此基础上，扩展出一个三维平面上的点类型。

保存一个二维平面上的点需要保存x和y两个坐标值，因此需要两个数据成员。该类提供3个公有的成员函数，分别完成以下功能：设置点的位置、获取x坐标和获取y坐标。类的定义如下：

```
class Point_2d {
private: int x,y;
public:
    void setpoint2(int a, int b) {x = a; y = b;}
    int getx() {return x;}
    int gety() {return y;}
};
```

三维平面上的点有3个坐标组成，因此可在二维平面的点的基础上再增加一个z坐标。类的定义如下：

```
class Point_3d:public Point_2d {
    int z;
public:
    void setpoint3(int a,int b,int c)    {setpoint2(a,b); z=c;}
    int getz() {return z;}
};
```

Point_3d类有3个数据成员,即x、y和z,有5个公有的成员函数,即setpoint2、setpoint3、getx、gety和getz。其中, setpoint2、getx、gety都是从Point_2d类继承过来的, Point_3d类只定义了两个成员函数。由此可见, Point_2d中的这些函数被重用了。在Point_3d的设计中还有一个要注意的问题: x和y是Point_2d类的私有数据成员, 在Point_3d的成员函数无法直接访问它们, 因此在setpoint3函数中不能直接对它们赋值, 而必须调用在Point_2d的公有成员函数setpoint2实现。如果派生类经常要用到基类的私有数据成员, 则最好将这些私有数据成员定义为Protected, 以提高访问效率。

Point_3d类的使用与普通类完全一样, 用户不用去管这个类是用继承方式从另外一个类扩展而来, 还是完全直接定义的。例如, 可以定义一个在Point_3d的对象p:

```
Point_3d p;
```

调用

```
p.setpoint3(1,2,3);
```

设置p的值。用户只需知道三维空间中的点要给它3个坐标, 而不用管它是如何设置、如何保存的。要获取这个点的位置, 只须调用这个类提供的公有成员函数getx、gety和getz, 也不用去管这个函数是由基类定义的还是派生类定义的。要输出p的位置, 可执行下列语句:

```
cout << "p: (" << p.getx() << ", " << p.gety() << ", " << p.getz() << ")" << endl;
```

12.2.3 派生类对象的构造、析构与赋值操作

派生类对象的初始化由基类和派生类共同完成。基类的数据成员由基类的构造函数初始化, 派生类新增加的数据成员由派生类的构造函数初始化。当创建派生类的对象时, 派生类的构造函数会在进入其函数体之前先调用基类的构造函数。至于调用的是基类的哪个构造函数, 默认情况下调用的是基类的默认构造函数。如果要调用基类的非默认构造函数, 则必须在派生类的构造函数的初始化表中指出。派生类构造函数的一般形式如下:

```
派生类构造函数名(参数表):基类构造函数名(参数表)
{ ... }
```

其中, 基类构造函数中的参数值通常来源于派生类构造函数的参数表, 也可以用常量值。

构造派生类对象时, 先执行基类的构造函数, 再执行派生类的构造函数。如果派生类新增的数据成员中含有对象成员, 则在创建对象时, 先执行基类的构造函数, 再执行对象成员的构造函数, 最后执行自己的构造函数体。

当派生类的对象销毁时, 将自动调用对象的析构函数。先执行派生类的析构函数, 再执行基

类的析构函数。

派生类不会继承基类的构造函数和析构函数，但派生类的构造函数和析构函数可以调用基类的构造函数和析构函数。

例12.2 定义一个表示人的类People。每个人包含一个信息：姓名。在People类的基础上，派生出一个表示学生的类Student。每位学生有一个学号。观察学生类对象的构造和析构过程。

为了说明构造和析构过程，我们只定义了每个类的构造函数和析构函数，并让构造函数和析构函数都输出一条信息。按照这个思想，People类的定义如代码清单12-4所示，Student类的定义如代码清单12-5所示。

代码清单12-4 People类的定义与实现

```
class People {
public:
    People( const char *s )      // 默认构造函数
    { strcpy(name, s);
      cout << "People constructor:" << '[' << name << ']' << endl;
    };

    ~People()                  // 析构函数
    { cout << "People destructor: " << '[' << name << ']' << endl; }

protected:
    char name[20];
};
```

代码清单12-5 student类的定义与实现

```
class Student:public People {
public:
    Student(const char *s, int n): People(s)
    {   s_no = n;
        cout << "Student constructor: student number is" << s_no
            << ", name is " << name << endl;
    }
    ~Student()
    {   cout << "Student destructor: student number is " << s_no
        << ", name is " << name << endl;
    }

private:
    int s_no;
};
```

由代码清单12-5可以看出，Student类在它的构造函数的初始化列表中调用了People类的构造函数，并将形式参数s传递给它，用s构造一个People类的对象，然后再在构造函数体中为s_no赋初值。People类和Student类的使用如代码清单12-6所示。

代码清单12-6 People类和Student类的使用

```
int main()
{
    { People p( "zhang" ); }
    cout << endl;
    Student s1( "li", 29 );
    cout << endl;
    Student s2( "wang", 30 );
    cout << endl;
    return 0;
}
```

代码清单12-6所示程序的运行结果如下：

```
People constructor: [zhang]
People destructor: [zhang]

People constructor: [li]
Student constructor: student number is 29, name is li

People constructor: [wang]
Student constructor: student number is 30, name is wang

Student destructor: student number is 30, name is wang
People destructor: [wang]
Student destructor: student number is 29, name is li
People destructor: [li]
```

这个程序首先在一个程序块中定义了一个People类的对象，此时会调用People类的构造函数，输出上面所示输出结果的第1行。接着就遇到了程序块结束，p的生命周期结束，于是调用了People类的析构函数，输出了输出结果的第2行。接着main函数输出一个空行，然后定义一个Student类的对象s1。该定义先调用基类的构造函数，再调用派生类的构造函数，输出了第4行和第5行。接着main函数又输出一个空行，再定义一个Student类的对象s2。该对象定义输出了第7行和第8行。main函数又输出一个空行，程序结束。此时系统会回收所有变量，这将调用对象的析构函数。析构s2会先执行Student的析构函数，再执行People的析构函数，于是输出了第10行和第11行。最后析构s1，输出最后两行。

派生类不能继承基类的构造函数，同样也不能继承基类的赋值运算。如果派生类没有定义赋值运算符重载函数，系统会为它提供一个默认的赋值运算符重载函数。该函数对派生类中的基类对象调用基类的赋值运算符重载函数，对派生类新增加的数据成员对应赋值。

如果默认的赋值运算符重载函数不能满足派生类的要求，则可以在派生类中重载赋值运算符。在派生类的赋值运算符重载函数中，要显式调用基类的赋值运算符函数来实现基类成员的赋值。

例12.3 定义一个图书馆系统中的读者类，每个读者的信息包括卡号、姓名、单位、允许借书的数量以及已借书记录。学生最多允许借5本书，教师最多允许借10本书。

根据题意,这个系统中有两类读者:学生读者和教师读者。这两类读者有一部分内容是相同的(卡号、姓名和单位),因此,可将这部分内容设计成一个基类。学生读者类readerStudent和教师读者类readerTeacher从基类派生,每个类增加已借书的数量以及已借书记录两个数据成员,并将允许借书的数量定义为整个类共享的常量。类的定义如代码清单12-7所示。

代码清单12-7 读者类的定义

```
class reader{
    int no;
    char name[10];
    char dept[20];
public:
    reader(int n, char *nm, char *d)
    {
        no = n;
        strcpy(name, nm);
        strcpy(dept, d);
    }
};

class readerTeacher :public reader{
    enum {MAX = 10}; //最多允许借的数量,是整个类共享的常量
    int borrowed;
    int record[MAX];
public:
    readerTeacher(int n, char *nm, char *d):reader(n, nm, d) {borrowed = 0;}
    bool bookBorrow(int bookNo); //借书成功,返回true,否则返回false
    bool bookReturn(int bookNo); //还书成功,返回true,否则返回false
    void show(); //显示已借书信息
};

class readerStudent :public reader {
    enum { MAX = 5}; //最多允许借的数量,是整个类共享的常量
    int borrowed;
    int record[MAX];
public:
    readerStudent(int n, char *nm, char *d):reader(n, nm, d) {borrowed = 0;}
    bool bookBorrow(int bookNo); //借书成功,返回true,否则返回false
    bool bookReturn(int bookNo); //还书成功,返回true,否则返回false
    void show(); //显示已借书信息
};
```

注意教师读者类和学生读者类对象的构造。教师读者类和学生读者类的构造函数只对自己新增的数据成员(borrowed)初始化,而基类数据成员的初始化是由基类的构造函数实现。基类构造函数需要的参数是由派生类的构造函数传递给它的。在派生类的构造函数的参数表中列出了派生类对象初始化时需要的所有参数,因而对派生类的用户来说,并不需要知道这个类是从某个类继承过来的还是完全自己开发的。

教师读者类中成员函数的实现如代码清单12-8所示。学生读者类中的这3个成员函数的实现

与教师读者类的完全相同。

代码清单12-8 教师读者类成员函数的实现

```
//借书成功，返回true，否则返回false
bool readerTeacher::bookBorrow(int bookNo)
{   if (borrowed == MAX) return false;
    else record[borrowed++] = bookNo;
    return true;
}

//还书成功，返回true，否则返回false
bool readerTeacher::bookReturn(int bookNo)
{   int i;
    for (i=0; i < borrowed; ++i) if (record[i] == bookNo) break;
    if (i == borrowed) return false;
    while (++i < borrowed) record[i-1] = record[i];
    --borrowed;
    return true;
}

//显示已借书信息
void readerTeacher::show()
{   for (int i = 0; i < borrowed; ++i) cout << record[i] << '\t'; }
```

12.2.4 重定义基类的函数

派生类是基类的扩展，可以是保存的数据内容的扩展，也可以是功能的扩展。当派生类对基类的某个功能进行扩展时，它定义的成员函数名可能会和基类的成员函数名重复。如果只是函数名相同而原型不同，系统认为派生类中有两个重载函数，如果原型完全相同，则派生类的函数会覆盖基类的函数。这称为重定义基类的成员函数。

例12.4 定义一个圆类型，用于保存圆以及输出圆的面积和周长。在此类型的基础上派生出一个球类型，可以给出球的表面积和体积。

保存一个圆，只需要保存它的半径。因此，圆类Circle只有一个数据成员。由于需要提供圆的面积和周长，需要提供两个公有的成员函数。除了这些之外，还需要一个构造函数。Circle类的设计如代码清单12-9所示。

代码清单12-9 Circle类的定义

```
class Circle {
protected:
    double radius;
public:
    Circle(double r = 0) {radius = r;}
    double getr() {return radius;}
    double area() { return 3.14 * radius * radius; }
```

```
double circum() { return 2 * 3.14 * radius;}
};
```

细心的读者可能已经注意到，在Circle类的定义中数据成员radius被声明成protected。这是因为我们将在Circle类的基础上扩展出一个表示“球”的类Ball，而Ball类的操作都需要用到radius。如果这个数据成员被声明成private，则派生类每次用到它时，必须调用getr函数获取它的值。这会大大降低运行的效率，而把它声明成protected，则派生类可直接引用这个数据成员，而其他类的成员函数或全局函数则不可以直接用这个数据成员。

Ball类的定义如代码清单12-10所示。

代码清单12-10 Ball类的定义

```
class Ball:public Circle {
public:
    Ball(double r = 0):Circle(r) {}
    double area() { return 4 * 3.14 * radius * radius; }
    double volumn() { return 4 * 3.14 * radius * radius * radius / 3; }
};
```

Ball类定义了一个求球体表面积的功能area，这个函数的原型与Circle类中的area完全相同，因此它会覆盖Circle类的area函数。当对Ball类的对象调用area函数时，将执行Ball类的area函数，不会引起二义性。

派生类中重新定义基类的成员函数时，它的功能往往是基类功能的扩展。为完成扩展的工作，派生类版本通常要调用基类中的该函数版本。这时必须使用作用域运算符，否则会由于派生类成员函数实际上调用了自身而引起无穷递归，最终使系统用完内存。这是致命的运行时错误。

例12.5 在Circle类的基础上定义一个Cylinder类，可以给出圆柱体的表面积和体积。

存储圆柱体可以在圆的基础上增加一个高度。圆柱体的表面积是上下两个圆的面积加上它的侧面积，圆柱体的体积是底面积乘上高度。而求圆的面积的函数在Circle类中已经存在，因而在Cylinder类中求表面积和体积时可以利用Circle类中已有的求圆面积的功能。Cylinder类的定义如代码清单12-11所示。

代码清单12-11 Cylinder类的定义

```
class Cylinder:public Circle {
    double height;
public:
    Cylinder(double r = 0, double h = 0):Circle(r) {height = h;}
    double geth() {return height;}
    double area() { return 2 * Circle::area() + circum() * height; }
    double volumn() { return Circle::area() * height ; }
};
```

注意代码清单12-11中的area函数。由于计算圆柱体面积用到了计算圆面积，在调用计算圆面积的area函数时，前面要加上了类的限定Circle::，否则会调用Cylinder类的area函数。

这些类的使用如代码清单12-12所示。

代码清单12-12 Circle、Ball和Cylinder类的使用

```
int main()
{
    Circle c(3);
    Ball b(2);
    Cylinder cy(1,2);

    cout << "circle: r=" << c.getr() << endl;
    cout << "area=" << c.area() << "\tcircum=" << c.circum() << endl;

    cout << "ball: r=" << b.getr() << endl;
    cout << "area=" << b.area() << "\tvolumn=" << b.volumn() << endl;

    cout << "cylinder: r=" << cy.getr() << "\th = " << cy.geth() << endl;
    cout << "area=" << cy.area() << "\tvolumn=" << cy.volumn() << endl;

    return 0;
}
```

代码清单12-12所示的程序运行结果如下：

```
circle: r=3
area=28.26      circum=18.84
ball: r=2
area=50.24      volumn=33.4933
cylinder: r=1
area=18.84      volumn=6.28
```

12.2.5 派生类作为基类

基类本身可以是一个派生类，例如：

```
class Base {...}
class D1:public Base {...}
class D2:public D1 {...}
```

每个派生类继承它的直接基类的所有成员，而不用去管它的基类是完全自行定义的还是从某一个类继承的。也就是说，类D1包含了Base的所有成员以及D1增加的所有成员。类D2包括了D1的所有成员及自己新增的成员。对于代码清单12-13中的类，Base包含一个数据成员x，Derive1从Base继承，又增加了一个数据成员y，因此它包含两个数据成员x和y。Derive2从Derive1继承，又增加了一个数据成员z，因此它包含3个数据成员x、y和z。在定义Derive2的时候，不用去管Derive1是如何生成的，只需要知道Derive1有两个数据成员以及这两个数据成员的访问特性。

代码清单12-13 派生类作为基类

```
class Base{
    int x;
public:
    Base(int xx) {x=xx; cout<<"constructing base\n";}
```

```

    ~Base() {cout<<"destructint base\n";}
};
class Derive1:public Base{
    int y;
public:
    Derive1(int xx, int yy): Base(xx) {y = yy; cout<<"constructing derive1\n";}
    ~Derive1() {cout<<"destructing derive1\n";}
};
class Derive2:public Derive1{
    int z;
public:
    Derive2(int xx, int yy, int zz):Derive1(xx, yy) {z = zz;cout<<"constructing
derive2\n";}
    ~Derive2() {cout<<"destructing derive2\n";}
};

```

当构造派生类对象时，同样不需要知道基类的对象是如何构造的，只需知道调用基类的构造函数就能构造基类的对象。如果基类是从某一个其他类继承的派生类，在构造基类对象时，又会调用它的基类的构造函数，依次上溯。例如，D2的构造函数的初始化列表只要指出调用D1的构造函数就可以了，D1的构造函数的初始化列表要指出调用Base的构造函数。当构造D2类的对象时，会先调用D1的构造函数，而D1的构造函数执行时又会先调用Base的构造函数。因此，构造D2类的对象时，最先初始化的是Base的数据成员，再初始化D1新增的成员，最后初始化D2新增的成员。析构的过程正好相反。如果定义了一个代码清单12-13中的类Derive2的对象op，将会输出

```

constructing Base
constructing Derive1
constructing Derive2

```

从中可以看出op的构造过程；而该对象析构时，将会输出

```

destructing Derive2
destructing Derive1
destructing Base

```

12.2.6 将派生类对象隐式转换为基类对象

由于派生类中包含了一个基类的对象，我们可以通过派生类的对象或指向派生类对象的指针来引用基类的成员。

事实上，还可以直接将一个派生类的对象赋给一个基类的对象，或让一个指向基类对象的指针指向派生类的对象，或定义一个基类的对象引用派生类的对象。这类应用也是安全的。

1. 将派生类对象赋给基类对象

由于派生类中包含了一个基类的对象，当把一个派生类对象赋给一个基类对象时，就是把派生类中的基类部分赋给此基类对象。派生类新增加的成员就舍弃了。

2. 基类指针指向派生类对象

当让一个基类指针指向派生类对象时，尽管该指针指向的对象是一个派生类对象，但由于它

本身是一个基类的指针，它只能解释基类的成员，而不能解释派生类新增的成员。因此，从指向派生类的基类指针出发，只能访问派生类中的基类部分。例如，对以下两个类：

```
class Base{
    int x;
public:
    Base(int x1=0) {x=x1;}
    void display() {cout << x << endl;}
};

class Derived:public Base{
    int y;
public:
    Derived(int x1 = 0, int y1=0):Base(x1) {y=y1;}
    void display() {Base::display(); cout << y << endl;}
};
```

在派生类中，有两个数据成员x和y。除了构造函数以外，有两个原型相同的公有的成员函数display。当定义一个派生类的对象，并对此对象调用display函数时，由于派生类的display函数重定义了基类的display函数，该函数覆盖了基类的display函数，因此派生类对象调用到的是派生类的display函数。但事实上，基类的display函数是存在的，只是派生类的对象看不见而已。当用一个基类的指针去指向派生类的对象时，尽管这块空间中有两个数据成员，但基类指针只看得见一个成员x。尽管这个类有两个display函数，但基类指针只看得见基类的display函数。因此对此基类指针调用display函数，调用到的是基类的display函数。

如果试图通过基类指针引用那些只在派生类中才有的成员，编译器会报告语法错误。

3. 基类的对象引用派生类的对象

引用事实上是一种隐式的指针。当用一个基类对象引用派生类对象时，相当于给派生类中的基类部分取了一个名字，从此基类对象看到的也是派生类中的基类部分。例如，如果定义

```
Derived d(1,2);
Base &br = d;
```

则br引用的是d中的基类部分。对br的访问就是对d中的基类部分的访问，对br的修改就是对d中基类部分的修改。

派生类的对象可以隐式地转换成基类的对象，但基类对象无法隐式转换成派生类对象，因为它无法解释派生类新增加的成员。除非在基类中定义了一个向派生类转换的类型转换函数，才能将基类对象转换成派生类对象。同样，也不能将基类地址赋给派生类的指针，即使该基类指针指向的就是一个派生类的对象。例如，有定义

```
Derived d, *dp;
Base *bp = &d;
```

当执行dp = bp 时，编译器依然会报错。

如果程序员能够确信该基类指针指向的是一个派生类的对象，确实想把这个基类指针赋给派生类的指针，这时可以用强制类型转换

```
dp = reinterpret_cast<Derived *> bp;
```

这等于告诉编译器：我知道这个危险，但我保证不会出问题。`reinterpret_cast`是一种相当危险的转换，它让系统按程序员的意思解释内存中的信息。

12.3 多态性与虚函数

12.3.1 多态性

多态性是面向对象程序设计的一个重要特点。所谓的多态性就是对不同对象发出同样的指令时，不同的对象会有不同的行为。例如，运算符的重载就是一种多态性的实现方法。当我们对复数类的对象和有理数类的对象发出一个“加”命令，这两种对象采取的动作是不一样的。编译器会调用不同的函数实现这两个运算。

运算符重载中调用哪一个函数是在编译时决定的，因此被称为编译时的多态性，也称为静态绑定。而虚函数则是实现多态性的另一种手段。在用虚函数实现的多态性中，具体调用哪一个函数，要到运行时才能确定，因此也被称为运行时的多态性，或动态绑定。

12.3.2 虚函数

1. 虚函数的作用

在12.2节中提到，基类的指针或引用可以指向派生类的对象。通过基类指针或基类的引用可以访问派生类对象中的基类部分，而不能访问派生类新增的成员。但如果基类中的某个函数被定义为虚函数的话，则会有完全不同的效果，它表明该函数在派生类中可能有不同的实现。当用基类的指针调用该虚函数时，首先会到派生类中去看一看这个函数有没有重新定义。如果派生类重新定义了这个函数，则执行派生类中的函数，否则执行基类的函数。

如果从该基类派生出多个派生类时，每个派生类都可以重新定义这个虚函数。当用基类的指针指向不同的派生类的对象时，就会调用不同的函数，这样就实现了多态性。而这个绑定要到运行时根据当时基类指针指向的是哪一个派生类的对象，才能决定调用哪一个函数，因而被称为运行时的多态性。

2. 虚函数的定义

虚函数就是在类定义中的函数原型声明前加一个关键字`virtual`。在派生类中重新定义时，它的函数原型（包括返回类型、函数名、参数个数和参数类型）必须与基类中的虚函数完全相同，否则编译器会认为派生类有两个重载函数。

例12.6 定义一个`Shape`类记录任意形状的位置，并定义一个计算面积的函数和显示形状及位置的函数，这些函数都是虚函数。在`Shape`类的基础上派生出一个`Rectangle`类和一个`Circle`类，这两个类都有可以计算面积和显示形状及位置函数。

由于矩形和圆计算面积的方法以及显示的方法都是不同的，因此必须重写基类的这两个函数。这3个类的定义如代码清单12-14所示。

代码清单12-14 虚函数定义示例

```

class Shape{
protected:
    double x, y;
public:
    Shape(double xx, double yy) {x=xx; y=yy;}
    virtual double area() {return 0.0;}
    virtual void display()
        {cout << "This is a shape. The position is (" << x << ", " << y << ")\n";}
};

class Rectangle:public Shape {
protected:
    double w, h;
public:
    Rectangle(double xx, double yy, double ww, double hh): Shape(xx,yy),w(ww),h(hh){}
    double area() {return w * h;} //重定义虚函数area
    void display() //重定义虚函数display
    {   cout << "This is a rectangle. The position is (" << x << ", " << y << ")\t";
        cout << "The width is " << w << ". The height is " << h << endl;
    }
};

class Circle:public Shape {
protected:
    double r;
public:
    Circle(double xx, double yy, double rr): Shape(xx,yy),r(rr){}
    double area() {return 3.14 * r * r;}
    void display()
    {   cout << "This is a rectangle. The position is (" << x << ", " << y << ")\t";
        cout << "The radius is " << r << endl;
    }
};

```

如果定义

```
Shape s(1,2), *sr;
```

执行

```

sr = &s;
sr->display();
cout << "The area is " << sr->area() << endl;

```

此时用基类指针指向基类成员，因此调用的是基类自己的函数。输出的结果如下：

```

This is a shape. The position is (1, 2)
The area is 0

```

如果定义

```
Rectangle rect(3, 4, 5, 6);
```

并执行

```
sr = &rect;
sr->display();
cout << "The area is " << sr->area() << endl;
```

这是用基类指针指向一个派生类的对象。由于在基类中`area`和`display`都是虚函数，因此当通过基类指针找到基类中的这两个函数时，它会到派生类中去检查有没有重新定义。在`Rectangle`类中重新定义了这两个函数，因此执行的是`Rectangle`类中的函数。执行的结果如下：

```
This is a rectangle. The position is (3, 4)   The width is 5. The height is 6
The area is 30
```

如果定义

```
Circle c(7, 8, 9);
```

并执行

```
sr = &c;
sr->display();
cout << "The area is " << sr->area() << endl;
```

这是用基类指针指向`Circle`类的对象，因此执行的也是派生类中的函数。执行结果如下：

```
This is a circle. The position is (7,8)   The radius is 9
The area is 254.34
```

我们甚至可以定义一个指向基类的指针数组，让它的每个元素指向基类或不同的派生类的对象。例如，如果

```
Shape *sp[3] = {&s, &rect, &c};
```

那么对于循环

```
for (int i = 0; i < 3; ++i)
{ sp[i]->display();
  cout << "The area is " << sp[i]->area() << endl;}
```

则循环体中的`sp[i]->display()`和`sp[i]->area()`的3次执行，执行的是不同的函数。而每次执行时执行的是哪一个函数，取决于指针指向的是哪一个类的对象。该语句执行的结果如下：

```
This is a shape. The position is (1,2)
The area is 0
This is a rectangle. The position is (3,4)   The width is 5. The height is 6
The area is 30
This is a circle. The position is (7,8)   The radius is 9
The area is 254.34
```

在使用虚函数时，必须注意以下两个问题。

(1) 在派生类中重新定义虚函数时，它的原型必须与基类中的虚函数完全相同，否则编译器会把它认为是重载函数，而不是虚函数的重定义。

(2) 派生类在对基类的虚函数重定义时，关键字`virtual`可以写也可以不写。不管`virtual`写还是不写，该函数都被认为是虚函数。但最好是在重定义时写上`virtual`。

例如，正方形是一类特殊的矩形，因此，我们可以在`Rectangle`类的基础上派生一个`Square`类，在这两个类中，都有一个显示形状的函数，如下所示：

```
class Rectangle {
    int w, h;
public:
    Rectangle(int ww, int hh): w(ww), h(hh) {}
    virtual void display() {cout << "this is a rectangle\n";}
};

class Square:public Rectangle {
public:
    Square(int ss): Rectangle(ss, ss) {}
    void display() {cout << "this is a square\n";}
};
```

尽管在Square类中，display函数没有指明是虚函数。但由于基类中的这个函数是虚函数，因此它也是虚函数。

12.3.3 虚析构函数

构造函数不能是虚函数，但析构函数可以是虚函数，而且最好是虚函数。

如果派生类新增加的数据成员中含有指针，指向动态申请的内存，那么派生类必须定义析构函数释放这部分空间。如果派生类的对象是通过基类的指针操作的，则delete基类指针指向的派生类的对象时，就会造成内存泄漏。当基类指针指向的对象析构时，通过基类指针会找到基类的析构函数，执行基类的析构函数；但此时派生类动态申请的空间没有释放，要释放这块空间必须执行派生类的析构函数。

要做到这一点，可以将基类的析构函数定义为虚函数。当析构基类指针指向的派生类的对象时，会先找到基类的析构函数。由于基类的析构函数是虚函数，又会找到派生类的析构函数，执行派生类的析构函数。派生类的析构函数在执行时会自动调用基类的析构函数，因此基类和派生类的析构函数都被执行，这样就把派生类的对象完全析构，而不是只析构派生类中的基类部分了。

与其他的虚函数一样，析构函数的虚函数性质都将被继承。因此，如果继承层次树中的根类的析构函数是虚函数的话，所有派生类的析构函数都将是虚函数。

12.4 纯虚函数和抽象类

12.4.1 纯虚函数

有时，基类往往只表示一种抽象的意志，而不与具体事物相联系。如在图12-14中定义的Shape类，它只表示具有封闭图形的东西，但当我们谈起图形时，会讲到圆、三角形、矩形等，但没有一种图形叫“形状”。因此在这个类中定义一个area函数显然是没有意义的。之所以在Shape类中定义这个函数，是为了实现多态性。为了表示这种函数，C++引入了纯虚函数的概念。

纯虚函数是一个在基类中声明的虚函数。它在基类中没有定义，但要求在派生类中定义自己

的版本。纯虚函数的声明形式如下：

```
virtual 返回类型 函数名(参数表)= 0;
```

有了纯虚函数，就不用要去为Shape类中的area函数写一个无用的函数体了，只需要把它声明为纯虚函数：

```
virtual double area() = 0;
```

12.4.2 抽象类

如果一个类至少含有一个纯虚函数，则被称为抽象类。Shape就是一个抽象类，它的定义如下：

```
class Shape{
protected:
    double x, y;
public:
    Shape(double xx, double yy) {x=xx; y=yy;}
    virtual double area()=0;
    virtual void display()
    {    cout << "This is a shape. The position is (" << x << ", " << y << ")\n";}
};
```

在应用抽象类时必须注意，因为抽象类中有没定义全的函数，所以无法定义抽象类的对象。因为一旦对此对象调用纯虚函数，该函数将无法执行。但可以定义指向抽象类的指针，它的作用是指向派生类，以实现多态性。

如果抽象类的派生类没有重新定义此纯虚函数，只是继承了基类的纯虚函数，那么，派生类仍然是一个抽象类。

抽象类的作用是保证进入继承层次的每个类都具有纯虚函数所要求的行为，这保证了围绕这个继承层次所建立起来的类具有抽象类规定的行为，保证了软件系统的正常运行，避免了这个继承层次中的用户由于偶尔的失误（比如，忘了为它所建立的派生类提供继承层次所要求的行为）而影响系统正常运行。

12.5 多继承

如果要在图10-1所示的学校系统中增加一个新的类——在职博士生，可以从教师和博士两个类继承。从多个类派生出一个类称为多继承。

12.5.1 多继承的格式

在定义多继承的派生类时，需要给出两个或两个以上的直接基类，其格式如下：

```
class 派生类名: [继承方法1] 基类名1, [继承方法2] 基类名2, ...
{ 新增的成员说明 };
```

对于多继承，需要说明以下几点。

- 继承方法及访问控制的规定与单继承相同。

- 派生类拥有所有基类的所有成员。
- 如果基类的构造不是采用默认构造函数，则在派生类构造函数的初始化列表中要列出基类的构造函数。
- 基类的声明次序决定基类构造函数的调用次序，基类构造函数的调用次序与初始化列表中基类的构造函数的出现次序无关。

下面来看一个多继承的示例：

```
class A {
    int a;
public:
    A(int aa=0){ a = aa; cout << "A... a = " << a << endl; }
    ~A() { cout << "~A" << endl; }
};

class B {
    int b;
public:
    B(int bb=0){ b = bb; cout << "B... b = " << b << endl; }
    ~B() { cout << "~B" << endl; }
};

class C: public A, public B {
    int c;
public:
    C(int i1=0, int i2=0, int i3=0) :A(i1), B(i2)
        { c = i3; cout << "C... c = " << c << endl; }
    ~C() {cout << "~C" << endl; }
};
```

类C是从类A和类B继承过来，类C包含了类A和类B的所有成员。因此类C有3个数据成员a、b和c。当类C的对象构造时，会按照基类的声明次序先构造A，再构造B，最后执行C的构造函数体。例如，定义

```
C obj(1, 2, 3);
```

则会输出

```
A... a = 1
B... b = 2
C... c = 3
```

当obj析构时，次序正好相反，会输出

```
~C
~B
~A
```

12.5.2 名字冲突

在多继承中，最大的问题就是名字冲突。如果在类A和类B中各增加一个成员函数show，显示各自的数据成员值。A类的show函数如下：

```
void show() {    cout << "a = " << a << endl;}
```

B类的show函数如下：

```
void show() {    cout << "b = " << b << endl;}
```

这样在类C中就继承了两个show函数。这种现象称为名字冲突。当对C类的对象调用show函数时，就会产生二义性。

C++解决名字冲突的方法是采用基类名限定访问，即“基类名::成员名”的形式。

如果在C类中也想定义一个show函数，显示它所有数据成员的值，那么显然该函数应该调用A类的show函数显示a，调用B类的show函数显示b，最后显示自己的c。函数可以定义如下：

```
void show() {    A::show(); B::show(); cout << "c = " << c << endl;}
```

12.5.3 虚基类

在多继承中，如果直接基类本身又是派生类，而且这些直接基类又有公共的基类，则会出现重复继承，即公共基类中的数据成员在派生类中就有多个副本。例如，对于下面的类D：

```
class A { int a; ...};  
class B : public A {...};  
class C : public A {...};  
class D : public B, public C {...};
```

类D中有两个类A的对象，一个是从类B继承过来，一个是从类C继承过来。通常情况下，我们只需要一个类A的副本，这可以用虚基类实现。

我们可以把类A定义成类B和C的虚基类，即

```
class A { int a; ...};  
class B : virtual public A {...};  
class C : virtual public A {...};  
class D : public B, public C {...};
```

这样，在类D中只有一份类A的副本。

在使用虚基类时，要注意派生类的构造函数的写法。一般派生类的构造函数只要调用直接基类的构造函数即可，但如果直接基类有一个虚基类，则派生类必须直接调用虚基类的构造函数。例如，D类的构造函数必须写成：

```
D(int aa,...) : A(aa), B(...),C(...) {...}
```

在派生类对象构造时，先构造虚基类的对象，再构造直接基类的对象，最后执行派生类的构造函数体。因此，当构造D类的对象时，构造的顺序为A、B、C、D。

小结

面向对象程序设计的一个重要目标是代码重用。本章介绍了代码重用的两种方法：组合和继承。组合是将某一个已定义类的对象作为当前类的数据成员，由此对此数据成员操作的代码进行

重用。继承是在已有类（基类）的基础上加以扩展，形成一个新类，称为派生类。在派生类定义时，只需要实现扩展功能，而基类的功能得到了重用。

本章还介绍了基于继承的多态性的实现，即运行时的多态性。

习题

简答题

1. `protected`成员有什么作用？
2. 在多继承的情况下，为什么会出现二义性？如何消除二义性？
3. 定义抽象类有什么意义？
4. 为什么要定义虚析构函数？
5. 试说明派生类对象的构造和析构次序。
6. 试说明虚函数和纯虚函数有什么区别。

程序设计题

1. 定义一个`Shape`类记录任意形状的位置，在`Shape`类的基础上派生出一个`Rectangle`类和一个`Circle`类，在`Rectangle`类的基础上派生出一个`Square`类，必须保证每个类都有计算面积和周长的功能。
2. 定义一个安全的动态整型数组。所谓安全，就是在数组操作中会检查下标是否越界。所谓动态，就是定义数组时，数组的规模可以是变量。在这个类的基础上，派生出一个可指定下标范围的安全的动态数组。
3. 例12.3中给出了一个图书馆系统中的读者类的设计，在教师读者类和学生读者类中，借书信息使用一个数组表示。试修改这些类，将借书信息用一个单链表表示。

所谓泛型程序设计就是以独立于任何特定类型的方式编写代码。使用泛型程序时，必须提供具体的所操作的类型或值。第6章介绍的函数模板就是泛型机制的一种实现方法，本章将介绍类模板，即用泛型机制设计的类。

13.1 类模板的定义

第10章介绍了一个可指定下标范围的安全的整型数组`IntArray`。如果在程序中还需要用到一个可指定下标范围的安全的实型数组，则必须另外定义一个类。事实上，这两个数组除了数组元素的类型不同之外，其他部分完全相同，包括数据成员的设计和成员函数的设计。如果把数组元素的类型定义成一个变量的话，这两个类是完全一样的。

与函数模板一样，可以将数组元素的类型定义成一个模板参数，把这个类写成一个类模板。类模板的定义格式如下：

```
template <模板的形式参数表>
class 类名{...};
```

类模板的定义以关键字`template`开头，后接模板的形式参数表。除了模板的形式参数说明之外，类模板的定义与其他的类定义类似。类模板可以定义数据成员和成员函数，也可以定义构造函数和析构函数，只是这些数据成员的类型可以是模板的形式参数，成员函数的参数类型或返回值类型也可以是模板的形式参数。

模板的形式参数可以是表示类型的类型形参，也可以是表示常量表达式的非类型形参。类型形参跟在关键字`class`或`typename`之后，非类型形参将在13.4节介绍。

例13.1 定义一个泛型的、可指定下标范围的、安全的数组。

这个类模板有一个类型参数，就是数组元素的类型。除了数组元素的类型可变以外，这个类模板与`IntArray`类完全相同。因此，可将数组元素的类型定义为模板参数。该类模板的定义如下：

```
template <class T>
class Array
{ int low;
  int high;
```

```

    T *storage;

public:
    //根据low和high为数组分配空间。分配成功，返回值为true，否则返回值为false
    Array(int lh = 0, int rh = 0):low(lh),high(rh)
    {storage = new T [high - low + 1]; }

    //复制构造函数
    Array(const Array &arr);

    //赋值运算符重载函数
    Array &operator=(const Array &a);

    //下标运算符重载函数
    T & operator[](int index);

    //回收数组空间
    ~Array() {delete [] storage;}
};

```

这个类可用于所有类型的数组，包括系统内置类型的数组和用户定义的类的数组。

这个类模板中，还有3个成员函数没有定义。类模板的成员函数都是形式参数与类模板相同的函数模板，因为它们操作的对象中可能包含有类型为模板参数的对象。

类模板的成员函数的定义具有如下形式。

- 必须以关键字`template`开头，后接类的模板形式参数表，即把类的成员函数定义为函数模板。
- 必须用作用域限定符“`::`”说明它是哪个类的成员函数。
- 类名必须包含其模板形式参数。

从这些规则可以看出，在类外面定义`Array`类的成员函数的格式应该是：

```

template <class T>
返回类型  Array<T>::函数名(形式参数表)
{函数体}

```

类模板`Array`的3个成员函数的定义如下：

```

template <class T>
Array<T>::Array(const Array<T> &arr)
{
    low = arr.low;
    high = arr.high;
    storage = new T [high - low + 1];
    for (int i = 0; i < high - low + 1; ++i)  storage[i] = arr.storage[i];
}

template <class T>
Array<T> &Array<T>::operator=(const Array<T> &a)
{
    if (this == &a) return *this; //防止自己复制自己
    delete [] storage; //归还空间
    low = a.low;
    high = a.high;
}

```

```

        storage = new T[high - low + 1]; //根据新的数组大小重新申请空间
        for (int i=0; i <= high - low; ++i) storage[i] = a.storage[i]; //复制数组元素
        return *this;
    }

    template <class T>
    T & Array<T>::operator[](int index)
    {   if (index < low || index > high) {cout << "下标越界"; exit(-1); }
        return storage[index - low];
    }
}

```

13.2 类模板的实例化

模板是一个蓝图，它本身不是一个类或函数。编译器从模板生成一个特定的类或函数的过程称为模板的实例化。函数模板的实例化由编译器自动完成，函数模板的用户无需关心自己调用的是函数还是函数模板，编译器根据函数调用时的实际参数类型确定模板参数的值，将模板参数的值代入函数模板生成一个真正可执行的模板函数。

类模板没这么幸运。编译器无法根据对象定义的过程确定模板参数的类型，因而需要用户明确指出模板形式参数的值。类模板的实例化格式如下：

类模板名<模板的实际参数> 对象名；

编译器首先将模板的实际参数值代入类模板，生成一个可真正使用的类，然后定义这个类的对象。例如，要定义一个整型的数组array1，它的下标范围是20~30，可用以下语句：

```
Array<int> array1 (20, 30);
```

编译器首先将int代入类模板Array，将Array中的所有的T都替换成int，产生一个模板类。然后定义这个类的一个对象array1。

要定义一个double型的数组array2，它的下标范围是10~20，可用以下语句：

```
Array<double> array2(10,20);
```

可以用下列语句输入array2的值：

```
for (i=10; i<=20; ++i) array2[i] = 0.1 * i;
```

也可以用下列语句输出array1的值：

```
for (i=20; i<=30; ++i) cout << array1[i] << '\t';
```

13.3 模板的编译

当编译器看到模板的定义时，它不立即产生代码。只有看到使用模板（如定义类模板的对象）时，编译器才产生特定类型的模板实例。

类模板的成员函数本身是模板函数。类模板的成员函数只有在为程序所用时才进行实例化。如果成员函数从未被使用，则永远不会被实例化。

13.4 非类型参数和参数的默认值

到现在为止，我们看到的模板参数都是类型参数。事实上，模板的形式参数不一定是类型，也可以是非类型参数。

在模板实例化时，类型参数用一个系统内置类型的名字或一个用户已定义类的名字作为实际参数，而非类型参数将用一个常量表达式作为实际参数。非类型模板实参的值必须是编译时的常量。

例13.2 例13.1中定义了一个类模板Array，它允许指定数组的下标范围，然后再根据下标范围在堆空间中申请一块存放数组元素的空间。这种数组还有一种实现方式，可以将数组元素存放在栈空间中。这可以用带非类型参数的类模板来实现。

这个类模板有3个模板参数：数组元素的类型、数组下标的上下界。前者为类型参数，后者为非类型参数。类模板的定义和实现如下：

```
template <class T, int low, int high>
class Array{
    T storage[high - low + 1];
public:
    //下标运算符重载函数
    T & operator[](int index) ;
};

template <class T, int low, int high>
T & Array<T, low, high>::operator[](int index)
{
    if (index < low || index > high) {cout << "下标越界"; exit(-1); }
    return storage[index - low];
}
```

因为这个类模板不再使用堆空间，因此不再需要构造函数和析构函数。由于模板参数已经给出了下标的上下界，因此也不需要记录下标上下界的数据成员。这个实现比例13.1中的实现更加简单。但唯一不足的是：例13.1中的类模板定义里数组的大小可以在运行时确定，而本例中的类模板定义里数组的大小必须在编译时确定。因为非类型模板实参值在编译时必须为常量。该数组的使用除了数组定义和例13.1中定义的类不同以外，其他的全部相同。如果要定义一个下标范围为10~20的整型数组array，可以用如下语句：

```
Array<int, 10, 20> array;
```

模板参数和普通的函数参数一样，也可以指定默认值。如果例13.1中的类模板Array经常被实例化为整型数组，则可在类模板定义时指定默认值：

```
template <class T = int> class Array
{ ... };
```

这样，在定义整型数组array时，就可以不指定模板的实参：

```
Array<> array;
```

13.5 类模板的友元

类模板可以声明下面两种友元。

- 普通友元：声明某个普通的类或全局函数为所定义类模板的友元。
- 模板的特定实例的友元：声明某个类模板或函数模板的特定实例是所定义类模板的友元。

13.5.1 普通友元

定义普通类或全局函数为所定义类模板的友元的声明格式如下：

```
template <class type>
class A {
    friend class B;
    friend void f();
    ...
};
```

该定义声明了类B和全局函数f是类模板A的友元。B的所有的成员函数和全局函数f可以访问类模板A的所有实例的私有成员。

13.5.2 模板的特定实例的友元

可以定义某个类模板或函数模板的特定实例为友元。例如，定义

```
template <class T> class B;           //类模板的声明
template <class T> void f(const T &); //函数模板的声明
template <class type>
class A {
    friend class B <int>;
    friend void f (const int &);
    ...
};
```

将类模板B的一个实例，即模板参数为int时的那个实例，作为类模板A的所有实例的友元；将函数模板f对应于模板参数为int的实例作为类模板A所有实例的友元。

下面形式的友元声明更为常见：

```
template <class T> class B;           //类模板的声明
template <class T> void f(const T &); //函数模板的声明
template <class type>
class A {
    friend class B <type>;
    friend void f (const type &);
    ...
};
```

这些友元声明说明了使用某一模板实参的类模板B和函数模板f的实例是使用同一模板参数的

类模板A的特定实例的友元。例如，类模板B的模板参数为int的实例是类模板A的模板参数为int的实例的友元，类模板B的模板参数为double的实例是类模板A的模板参数为double的实例的友元，而类模板B的模板参数为int的实例不是类模板A的模板参数为double的实例的友元。

13.5.3 声明的依赖性

当声明类模板B和函数模板f为类模板A的友元时，编译器必须知道有这样一个类模板和函数模板存在，并且知道类模板B和函数模板f的原型。因此，必须在友元声明之前先声明B和f的存在。

例13.3 为例13.1中的类模板Array增加一个输出运算符重载函数，可以直接输出数组的所有元素。

要直接输出数组的所有元素，可以为Array类重载<<运算符。由于Array是一个类模板，可用于不同类型的数组，因此，该输出运算符重载函数也应该是函数模板。这个函数模板可以定义如下：

```
template<class type>
ostream &operator<<(ostream &os, const Array<type> &obj)
{
    os << endl;
    for (int i=0; i < obj.high - obj.low + 1; ++i) os << obj.storage[i] << '\t';
    return os;
}
```

在Array的定义中，必须把这个函数模板声明为模板参数相同的实例的友元。增加了该友元的Array类模板的定义如下：

```
template <class T> class Array;    //类模板Array的声明
template<class T> ostream &operator<<(ostream &os, const Array<T>&obj); //输出重载声明

template <class T>
class Array {
    friend ostream &operator<<(ostream &, const Array<T> &);

private:
    int low;
    int high;
    T *storage;

public:
    //根据low和high为数组分配空间。分配成功，返回值为true，否则返回值为false
    Array(int lh = 0, int rh = 0):low(lh),high(rh)
    { storage = new T [high - low + 1]; }

    //复制构造函数
    Array(const Array &arr);
```

```

//赋值运算符重载函数
Array &operator=(const Array & a);

//下标运算符重载函数
T & operator[](int index) {return storage[index - low];}

//回收数组空间
~Array() {delete [] storage; }
};

```

上述代码中的声明

```
friend ostream &operator<<(ostream &, const Array<T> &);
```

声明了函数模板`operator<<`的一个实参为`T`的实例是类模板`Array`的实参为`T`的实例的友元。有了这样一个声明以后，对于`Array`的任何一个实例，例如：

```
Array<int> array(10,20);
```

我们可以用

```
cout << array;
```

直接输出它的所有元素。

例13.4 定义一个单链表，可以存放任意类型的数据。将单链表的所有操作都封装在单链表类中，使得单链表的用户只需要知道单链表可以做哪些操作，而不用去管这些操作是如何实现的。

如第8章所示，单链表中的每个元素存放在一个单独的结点中，元素之间的关系通过结点中的指针实现。单链表的每个结点存储了一个指向下一个结点的指针。存储一个单链表就是存储一个指向单链表头结点的指针。因此，单链表的实现必须有两个类：一个是处理结点的类（`Node`），一个是处理单链表的类（`linkList`）。结点类有两个数据成员：数据元素及指向下一节点的指针。除了设置两个数据成员初值的操作以外，这个类不需要其他操作，因此它只有构造函数。单链表类只需要一个数据成员，即指向头结点的指针。我们所要求的单链表的功能也非常简单，只包含创建一个单链表和输出链表的所有成员。由于这里要求单链表可以处理任何类型，因此这两个类都是模板。

这两个类的关系相当密切。事实上，它们合起来表示一个对象。链表的操作都涉及结点的数据。这种关系有两种实现方式。一种是定义两个独立的类。由于链表操作经常需要访问结点值，因此，通常将同一模板参数的链表类设置为结点类的友元。另一种方法是将结点类定义成链表类的内嵌类。由于结点类是专门为链表类服务的，除此之外，不会有其他的函数或对象会用到结点的对象。因此，可将结点类定义在链表类中，作为链表类专用的内部类型。按照第一种方案，这两个类的定义如下：

```

template <class elemType> class linkList;
template <class T> ostream &operator<<(ostream &, const linkList<T> &);
template <class elemType> class Node ;

```

```

template <class elemType>
class Node {
    friend class linkList<elemType>;
    friend ostream &operator<< ( ostream &, const linkList<elemType> &);
private:
    elemType data;
    Node <elemType> *next;
public:
    Node(const elemType &x, node <elemType> *N = NULL) { data = x; next = N;}
    Node( ):next(NULL) {}
    ~Node() {}
};

template <class elemType>
class linkList {
    friend ostream &operator<< ( ostream &, const linkList<elemType> &);
protected:
    Node <elemType> *head;
    void makeEmpty();
public:
    linkList() { head = new node<elemType>; }
    ~linkList() {makeEmpty(); delete head;}

    void create(const elemType &flag);
};

```

这两个类的成员函数的实现如下：

```

template <class elemType>
void linkList<elemType>::makeEmpty()
{
    Node <elemType> *p = head->next, *q;
    head->next=NULL;
    while (p != NULL) { q=p->next; delete p; p=q;}
}

template <class elemType>
void linkList<elemType>::create(const elemType &flag)
{
    elemType tmp;
    Node <elemType> *p, *q = head;

    cout << "请输入链表数据, " << flag << "表示结束" << endl;

    while (true) {
        cin >> tmp;
        if (tmp == flag) break;
        p = new Node<elemType>(tmp);
        q->next = p;
        q = p;
    }
    q->next = NULL;
}

```

类模板Node包括两个私有数据成员，一个存放元素的值，一个存放指向下一元素的指针。

类模板linkList包含一个数据成员，就是指向链表头结点的指针。由于所有的链表操作都是通过访问结点元素实现的，因此链表类经常会访问Node的数据成员。为了便于链表类的访问，以及提高访问的效率，Node类将linkList类声明为友元。为了输出链表的所有数据，定义了一个输出运算符重载函数，并把它定义成友元函数。该函数的定义如下：

```
template <class T>
ostream &operator<<(ostream &os, const linkList<T> &obj)
{
    Node <T> *q = obj.head->next;

    os << endl;
    while (q != NULL){ os << q->data; q = q->next; }
    return os;
}
```

由于输出的元素存放在Node类的对象中，因此该函数也必须作为Node类的友元。

有了这些基础，我们就可以使用任何类型的单链表。如果想定义一个整型的单链表，可以用如下定义：

```
linkList<int> intList;
```

这个定义产生了类模板的一个整型实例。如果想创建这个单链表，可以调用create函数：

```
intList.create(0);
```

这个调用将输入链表中的元素值，直到输入0为止。要输出链表的所有元素，可以直接输出：

```
cout << intList;
```

13.6 类模板作为基类

类模板可以作为继承关系中的基类。自然，从该基类派生出来的派生类还是一个类模板，而且是一个功能更强的类模板。

类模板的继承和普通的继承方法基本类似。只是在涉及基类时，都必须带上模板参数。

例13.5 从例13.4中的linkList类模板派生一个栈的类模板。

栈是一类特殊的线性表，它的插入和删除都只能在表的一端进行。允许插入和删除的一端称为栈顶，另一端称为栈底。栈的常用操作有进栈（push）和出栈（pop）。

由于栈是一个特殊的线性表，因此可以将栈建立在单链表的基础上。在单链表中，我们可以将表头定义为栈顶，表尾定义为栈底。因此，栈就是在单链表的基础上增加两个操作：push和pop。栈的类模板的定义如下：

```
template <class elemType>
class Stack:public linkList<elemType> {
public:
    void push(const elemType &data)
    {
        Node <elemType> *p = new Node<elemType>(data);
        p->next = head->next;
```

```
        head->next = p;
    }
    bool pop(elemType &data) //栈为空时返回false, 出栈的值在data中
    {
        Node <elemType> *p = head->next;
        if ( p == NULL) return false;
        head->next = p->next;
        data = p->data;
        delete p;
        return true;
    }
};
```

为了使Stack能访问Node的数据成员, 必须将Stack设为Node的友元。

小结

本章介绍了C++中的泛型程序设计的工具——模板。模板是独立于类型的蓝图, 编译器可以根据模板产生多种特定类型的实例。我们只需要编写一个模板, 编译器会根据模板的使用情况产生不同的实例。模板分为函数模板和类模板。

在本章中, 我们学习了如何写一个类模板, 如何实例化类模板使之成为一个模板类, 如何定义及使用模板的友元, 如何将类模板作为基类。

习题

简答题

1. 什么是模板的实例化?
2. 为什么要定义模板?

程序设计题

1. 设计一个处理栈的类模板, 要求该模板用一个数组存储栈的元素。数组的初始大小由模板参数指定。当栈中的元素个数超过数组大小时, 重新申请一个大小为原来数组一倍的新数组存放栈元素。
2. 设计一个处理集合的类模板, 要求该类模板能实现集合的并、交、差运算。

输入/输出是程序的一个重要部分。输入/输出是指程序与外围设备之间的数据传输。程序运行所需要的数据往往是从外围设备（如键盘或磁盘文件）获得。程序运行的结果通常也是输出到外围设备，如显示器或磁盘文件。

在C++中，输入/输出不包括在语言所定义的部分，而是由标准库提供。C++的设计者提出了一种解决方案，虽然它不属于C++语言定义的范畴，但大多数C++编译器都实现了这个方案，并且也被C++国际标准所接纳。

C++的输入/输出分为基于控制台的输入/输出、基于文件的输入/输出和基于字符串的输入/输出。基于控制台的输入/输出是指从标准的输入设备（如键盘）获得数据，以及把程序的执行结果输出到标准的输出设备（如显示器）；基于文件的输入/输出是指从外存储器上的文件获取数据，或把数据存于外存储器上的文件；基于字符串的输入/输出是指从程序中的某一字符串变量获取数据，或把数据存于某一字符串变量。

14.1 流与标准库

C++的输入/输出是以一连串字节流的方式进行的。在输入操作中，字节从设备（如键盘、磁盘）流向内存。在输出操作中，字节从内存流向设备（如显示器、打印机、磁盘等）。C++同时提供“低层次”和“高层次”的输入/输出。低层次的输入/输出直接对字节流中的字节进行操作，而高层次的输入/输出可以将字节组合成有意义的单位，如整型数、浮点数及用户自定义类型进行操作。

C++还提供了无格式输入/输出和格式化输入/输出两种操作。无格式输入/输出传输速度快，但使用起来较为麻烦。格式化输入/输出按不同类型、不同格式对数据进行处理，因此需要增加额外的处理时间，不适于处理大容量的数据传输。

C++同时支持过程式的输入/输出和面向对象的输入/输出。过程式的输入/输出是通过从C语言保留下来的函数库中的输入/输出函数来实现的，用这些函数可以实现对基本类型数据的输入/输出。面向对象的输入/输出是通过C++的输入/输出类库来实现的。输入/输出类库提供的输入/输出操作是由一些输入/输出的类来实现的。这些类主要包含在3个头文件中：`iostream`定义了基于控制台的输入/输出类型，在本书前面的所有程序中几乎都用到了这个头文件；`fstream`定义

了基于文件的输入/输出类型；sstream定义了基于字符串的输入/输出类型。每个头文件及定义的标准类型如表14-1所示。

表14-1 输入/输出标准库类型及头文件

头 文 件	类 型
iostream	istream从流中读取 ostream写到流中去 iostream对流进行读写，由istream和ostream派生
fstream	ifstream从文件中读取，由istream派生而来 ofstream写到文件中去，由ostream派生而来 fstream对流进行读写，由iostream派生而来
sstream	istringstream从string对象中读取，由istream派生而来 ostringstream写到string对象中去，由ostream派生而来 stringstream对string对象进行读写，由iostream派生而来

所有输入/输出的类都是从一个公共的基类ios派生的。ios派生出istream和ostream类。istream派生出了ifstream和istringstream类，ostream则派生出ofstream和ostringstream。istream和ostream又共同派生出iostream。iostream又继承出fstream和stringstream。这些类之间的继承关系如图14-1所示。

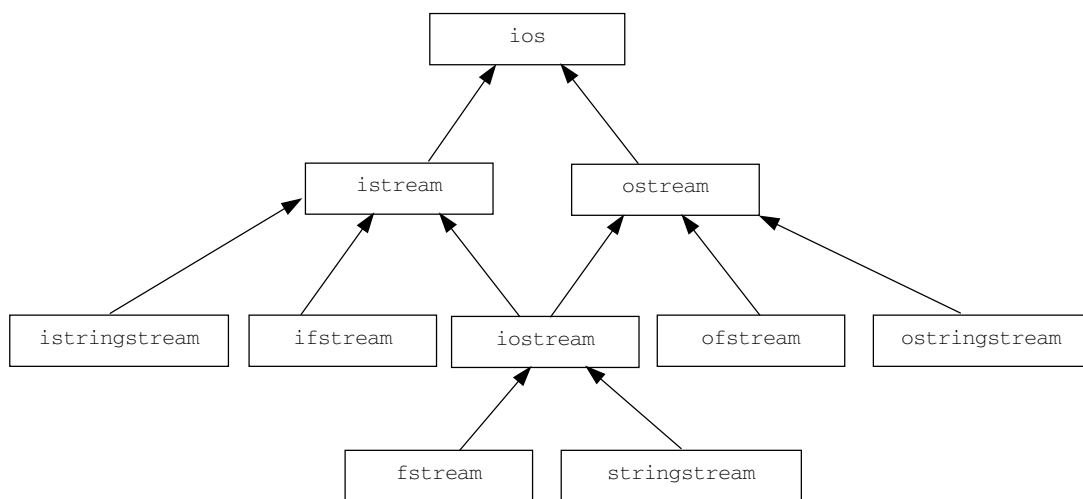


图14-1 输入/输出类的继承关系

从这个继承关系可以看出，C++中的控制台输入/输出和文件操作以及字符串操作的方式是相同的。

14.2 输入/输出缓冲

C++的输入/输出是基于缓冲实现的。每个输入/输出对象管理一个缓冲区，用于存储程序读

写的的数据。当用户在键盘上输入数据时，输入的数据是存储在输入缓冲区中，当执行>>操作时，从输入缓冲区中取数据存入变量，如果缓冲区中无数据，则等待从外围设备取数据放入缓冲区。<<操作是将数据放入输出缓冲区。例如，有下列语句：

```
os << "please enter the value:";
```

系统将字符串常量存储在与流os相关联的缓冲区中。下面几种情况将导致缓冲区的内容被刷新，即写入真正的输出设备或文件：

- (1) 程序正常结束。作为main函数返回工作的一部分，将清空所有的输出缓冲区。
- (2) 当缓冲区已满时，在写入下一个值之前，会刷新缓冲区。
- (3) 用标准库的操纵符，如行结束符endl，显式地刷新缓冲区。
- (4) 可将输出流与输入流关联起来。在这种情况下，在读输入流时，将刷新其关联的输出缓冲区。在标准库中，将cout和cin关联在一起，因此每个输入操作都将刷新cout关联的缓冲区。

14.3 基于控制台的输入/输出

基于控制台的输入/输出我们已经不陌生了。我们用cin和cout输入和输出系统内置类型的数据，通过对>>和<<的重载，我们也可以对用户自己定义的类型用cin和cout输入和输出。

基于控制台的输入/输出的支持主要包含在两个头文件中：iostream和iomanip。头文件iostream声明了所有输入/输出流操作所需要的基础服务，定义了cin、cout、cerr和clog这4个标准对象，分别对应于标准输入流、标准输出流、无缓冲的标准错误流以及有缓冲的标准错误流。cin是istream类的对象，与标准输入设备（通常是键盘）相关联。cout是ostream类的对象，与标准的输出设备（通常是显示器）相关联。cerr是ostream类的对象，与标准的错误设备相关联。cerr是无缓冲的输出，这意味着每个针对cerr的流插入必须立刻送到显示器。clog是ostream类的对象，与标准的错误设备相关联。clog是有缓冲的输出。iostream同时还提供了无格式和格式化的输入/输出服务。格式化的输入/输出通常需要用一些带参数的流操纵符，头文件iomanip声明了带参数的流操纵符。

14.3.1 输出流

ostream提供了格式化和无格式的输出功能。输出功能包括用流插入运算符(<<)执行标准类型数据的输出，通过put成员函数进行字符输出，通过write函数进行无格式的输出，以及格式化的输出。

1. 标准类型数据的输出

C++能自动判别数据类型，并根据数据类型解释内存单元的信息，把它转换成字符显示在显示器上。例如，x为整型变量，它的值是123，它在内存中的表示为3个字节全0，最后一个字节为01111011（如整型数用4个字节表示）。当执行cout << x;时，C++把这4个字节中的值解释为整型数123，然后把它转换成字符输出。此时，显示器会显示123。对于其他类型的输出也是如此。标准输出示例见代码清单14-1。

代码清单14-1 标准输出示例程序

```
//文件名: 14-1.cpp
//标准输出示例
#include <iostream>
using namespace std;

int main()
{   int a = 5, *p = &a;
    double x = 1234.56;
    char ch = 'a';

    cout << "a = " << a << endl;    //输出整型变量a的值
    cout << "x = " << x << endl;    //输出双精度变量x的值
    cout << "ch = " << ch << endl;  //输出字符型变量ch的值
    cout << "*p = " << *p << endl;  //输出整型指针p指向的空间中的值
    cout << "p = " << p << endl;    //输出整型指针p的值, 即一个地址

    return 0;
}
```

代码清单14-1所示的程序对应的输出如下:

```
a = 5
x = 1234.56
ch = a
*p = 5
p = 0012FF7C
```

C++的标准输出对于指针有一个特例。在代码清单14-1所示的程序中, 我们看到了一个指针输出的语句, 该语句输出指针变量p的值。在程序的输出结果中我们看到了一个十六进制的数值0012FF7C, 这就是指针变量p中保存的地址值, 也就是变量a的地址。在C++中, 地址的默认输出方式是十六进制。但如果输出的指针变量是一个指向字符的指针时, C++并不输出该指针中保存的地址, 而是输出该指针指向的字符串。如果确实想输出这个指向字符的指针变量中保存的地址值, 可以用强制类型转换, 将它转换成void*类型, 如代码清单14-2所示。事实上, 如果程序员想输出地址, 最好都把指针转换成void*类型。

代码清单14-2 指向字符的指针输出示例程序

```
//文件名: 14-2.cpp
//指向字符的指针输出示例
#include <iostream>
using namespace std;

int main()
{   char *ptr = "abcdef";

    cout << "ptr指向的内容为: " << ptr << endl;
    cout << "ptr中保存的地址为: " << (void*)ptr << endl;

    return 0;
}
```

代码清单14-2中的程序的输出如下：

```
ptr指向的内容为: abcdef
ptr中保存的地址为: 0046C04C
```

2. 通过put成员函数进行字符输出

字符型数据还可以用成员函数put来输出。put函数有一个字符类型的形式参数，它的返回值是调用put的对象的引用。例如：

```
cout.put('A');
```

将字符A显示在屏幕上，而

```
cout.put(65);
```

输出ASCII码值为65的字符，输出也是字符A。

由于put函数的返回值是当前对象的引用，因此可以连续调用put函数：

```
cout.put('A').put('\n');
```

点运算符(.)从左向右结合，因此，该语句相当于下面两条语句：

```
cout.put('A');
cout.put('\n');
```

即在输出字符A后输出一个换行符。

3. 通过write成员函数进行无格式的输出

成员函数write可实现无格式的输出，这个函数将一定量的字节从字符数组输出到相应的输出流对象。它有两个参数：第一个参数是一个指向字符的指针，表示一个字符数组；第二个参数是一个整型值，表示输出的字符个数。字符数组中的字节都是未经任何格式化的，仅仅是以原始数据形式输出。例如，语句

```
char buffer[] ="HAPPY BIRTHDAY";
cout.write(buffer, 10 );
```

输出buffer中的10个字节，函数调用

```
cout.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ",10);
```

显示了字母表中的前10个字母。

14.3.2 输入流

istream提供了格式化和无格式的输入功能。输入功能包括用流提取运算符(>>)执行标准类型数据的输入，通过get和getline成员函数进行字符和字符串的输入，通过read成员函数进行无格式的输出，以及格式化的输入。

1. 标准类型数据的输入

标准类型数据的输入是通过流提取运算符(>>)实现的。流提取运算符跳过输入流中的空白

字符（如空格、Tab键和回车）。在每个输入操作之后，流提取运算符返回一个当前对象的引用。如果该引用被用作判断条件，如while语句中的循环判断条件，那么将隐式地将它转换为bool类型的值。如果输入操作成功，变量得到了正确的值，则转换成true。如果输入不成功，如遇到文件结束标记（EOF），变量没有得到所需的值，则转换为false。

在第4章中介绍了一个统计某个班级某次考试的最高成绩的问题。由于事先不知道有多少学生，我们选择了一个特殊的输入标记“-1”表示输入结束。但在某些应用中，输入标记很难选择。在进一步了解了流提取运算符以后，我们可以利用流提取运算符的返回值，避免了输入标记选择的问题。当所有学生的成绩都输入后，用户可以输入表示成绩输入完毕的文件结束符，程序将输出这批成绩中的最高分。在Windows系统中，文件结束符是Ctrl+z。程序的实现如代码清单14-3所示。

代码清单14-3 标准输入示例程序

```
//文件名: 14-3.cpp
//标准输入示例
#include <iostream>
using namespace std;

int main()
{
    int grade, highestGrade = -1;
    cout << "Enter grade (enter end-of-file to end): ";
    while ( cin >> grade) {
        if ( grade > highestGrade) highestGrade = grade;
        cout << "Enter grade (enter end-of-file to end): ";
    }
    cout << "\n\nHighest grade is: "<< highestGrade << endl;
    return 0;
}
```

注意代码清单14-3中的while循环的控制条件。当用户输入一个数字后，表达式cin >> grade返回true，则执行循环；当用户输入了EOF后，该表达式返回false，则循环结束。

2. 通过get和getline成员函数进行字符和字符串的输入

get函数有3种格式：不带参数、带1个参数和带3个参数。

不带参数的get函数从当前的输入流对象读入一个字符，包括空白字符以及表示文件结束的EOF，并将读入值作为函数的返回值返回。例如，语句

```
while((ch = cin.get()) != EOF) cout << ch;
```

将重复从标准输入流对象cin读入一字符，并将输入的字符回显在显示器上，直到输入EOF。

第二种格式的get函数带1个字符类型的引用参数，它将输入流中的下一个字符（包括空白字符）存储在参数中，它的返回值是当前输入流对象的引用。例如，下面的循环语句将输入一个字符串，存入字符数组ch，直到输入回车：

```
cin.get(ch[0]);
for (i = 0; ch[i] != '\n'; ++i) cin.get(ch[i+1]);
ch[i] = '\0';
```

第三种格式的`get`函数有3个参数：字符数组、数组规模和表示输入结束的结束符（结束符的默认值为`'\n'`）。这个函数或者在读取比指定的数组规模少一个字符后结束，或者在遇到结束符时结束。

为使字符数组中的输入字符串能够结束，在输入结束时函数会自动将一个空字符`'\0'`插入到字符数组中。结束符仍然保留在输入流中。因此，要输入一行字符，可用语句

```
get(ch, 80, '\n');
```

也可以用

```
get(ch, 80);
```

当输入达到79个字符或读到了回车键时输入结束。要输入一个以句号结尾的句子，可用下面的语句：

```
get(ch, 80, '.');
```

当遇到输入结束符“.”或输入字符数达到79时，函数执行结束。输入结束符没有放在字符数组中，而是保留在输入流中，下一个和输入相关的语句会读入这个输入结束符。例如，对应于语句

```
get(ch, 80, '.');
```

用户输入

```
abcdef.✓
```

则`ch`中保存的是字符串`"abcdef"`，而“.”仍保留在输入缓冲区中，如果继续调用

```
cin.get(ch1);
```

或

```
cin >> ch1;
```

则字符变量`ch1`中保存的是“.”。

成员函数`getline`的功能与第三种形式的`get`函数类似。它也有3个参数，3个参数的类型和作用与第三种形式的`get`函数完全相同。这两个函数的唯一区别在于对输入结束符的处理。`get`函数将输入结束符留在输入流中，而`getline`函数将输入结束符从输入流中删除。例如，对应于语句

```
getline(ch, 80, '.');
```

用户输入

```
abcdef.✓
```

则`ch`中保存的是字符串`"abcdef"`，而“.”从输入缓冲区中被删除，如果继续调用

```
cin.get(ch1);
```

或

```
cin >> ch1 ;
```

因为输入缓冲区为空，程序将会等待用户的键盘响应。

3. 通过read函数进行无格式的输入

调用成员函数read可实现无格式的输入。read函数有两个参数：第一个参数是一个指向字符的指针，第二个参数是一个整型值。这个函数把一定量的字节从输入缓冲区读入字符数组，不管这些字节包含的是什么内容。例如：

```
char buffer[80];
cin.read(buffer, 10 );
```

不管输入缓冲区中有多少个字节，都只读入10个字节，放入buffer。

如果还没有读到指定的字符数就遇到了EOF，则读操作结束。read函数真正读入的字符数可以由成员函数gcount得到。read和gcount函数的应用示例见代码清单14-4。

代码清单14-4 read和gcount函数的示例程序

```
//文件名: 14-4.cpp
//read和gcount函数的应用示例
#include <iostream>
using namespace std;

int main()
{
    char buffer[ 80 ];
    cout << "Enter a sentence:\n";
    cin.read(buffer, 20);
    cout << "\nThe sentence entered was:\n";
    cout.write(buffer, cin.gcount());
    cout << endl;
    cout << "一共输入了" << cin.gcount() << "个字符\n";
    return 0;
}
```

代码清单14-4所示的程序的某次运行结果如下：

```
Enter a sentence:
Using the read, write, and gcount member functions
The sentence entered was:
Using the read, write
一共输入了20个字符
```

尽管用户在键盘上输入的字符串很长，但read函数只读入了20个字符，此时gcount函数的返回值为20。

read和write函数主要用于文件访问。

14.3.3 格式化的输入/输出

C++提供了多种流操纵符来完成格式化输入/输出的问题。流操纵符以一个流引用作为参数，

并返回同一流引用，因此它可以嵌入到输入/输出操作的链中。流操纵符的功能包括设置整型数的基数，设置浮点数的精度，设置和改变域宽，设置域的填充字符等。

1. 设置整型数的基数

输入/输出流中的整型数默认为十进制表示。为了使流中的整型数不局限于十进制，可以插入hex操纵符将基数设为十六进制，插入oct操纵符将基数设为八进制，也可以插入dec操纵符将基数重新设为十进制。

改变输入/输出流中整型数的基数也可以通过流操纵符setbase来实现。该操纵符有一个整型参数，它的值可以是16、10或8，表示将整型数的基数设为十六进制、十进制或八进制。由于setbase有一个参数，所以也称为参数化的流操纵符。使用任何带参数的流操纵符，都必须包含头文件iomanip。流的基数值只有被显式更改时才会变化，否则一直沿用原有的基数。代码清单14-5中的程序演示了这几个流操纵符的用法。

代码清单14-5 设置整型数的基数的示例程序

```
//文件名: 14-5.cpp
//设置整型数的基数的示例
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int n;
    cout << "Enter a octal number: "; //读入八进制表示的整型数
    cin >> oct >> n;
    cout << "octal " << oct << n << " in hexadecimal is:" << hex << n << '\n' ;
    cout << "hexadecimal " << n << " in decimal is:" << dec << n << '\n' ;
    cout << setbase(8) << "octal " << n << " in octal is:" << n << endl;
    return 0;
}
```

代码清单14-5中的程序以八进制读入一个整型数，然后以十六进制和十进制输出。程序的某次运行结果如下：

```
Enter a octal number: 30
Octal 30 in hexadecimal is: 18
Hexadecimal 18 in decimal is: 24
Octal 30 in octal is: 30
```

2. 设置浮点数的精度

设置浮点数的精度（即实型数的有效位数）可以用流操纵符setprecision或基类ios的成员函数precision来实现。一旦调用了这两者之中的某一个，将影响所有输出的浮点数的精度，直到下一个设置精度的操作为止。操纵符setprecision和成员函数precision都有一个参数，表示有效位数的长度。具体示例请见代码清单14-6。

代码清单14-6 设置精度的示例程序

```

//文件名: 14-6.cpp
//设置精度的示例
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double x = 123.456789, y = 9876.54321;

    for (int i = 9; i > 0; --i) {
        cout.precision(i);
        cout << x << '\t' << y << endl;
    }
    //或写成for (int i = 9; i > 0; --i) cout << setprecision(i) << x << '\t' << y << endl;

    return 0;
}

```

代码清单14-6中的程序定义了两个双精度数x和y，它们都有9位精度，然后以不同的精度输出。程序的输出如下：

```

123.456789    9876.54321
123.45679     9876.5432
123.4568      9876.543
123.457       9876.54
123.46        9876.5
123.5         9877
123           9.88e+003
1.2e+002      9.8e+003
1e+002        1e+004

```

由这个输出结果可以看出，每次设置了精度后，所有的输出都是按照这个精度。例如，第一次设置精度为9，则这两个数都输出了9位；第二次设置精度为8，这两个数都输出了8位。

3. 设置域宽

域宽是指数据在外围设备中所占的字符个数。设置域宽可以用基类的成员函数width，也可以用流操纵符setw。width和setw都包含一个整型的参数，表示域宽。设置域宽可用于输入，也可用于输出。设置域宽只适合于下一次输入或输出，之后的操作的域宽将被设置为默认值。当没有设置输出域宽时，C++按实际长度输入/输出。例如，若整型变量a = 123，b = 456，则输出

```
cout << a << b;
```

将输出123456。一旦设置了域宽，该输出必须占满域宽。如果输出值的宽度比域宽小，则插入填充字符填充。默认的填充字符是空格。例如，语句

```
cout << setw(5) << x << setw(5) << y << endl;
```

的输出为

123 456

每个数值占5个位置，前面用空格填充。如果实际宽度大于指定的域宽，则按实际宽度输出。例如，语句

```
cout << setw(3) << 1234 << setw(2) << 56;
```

的输出为

123456

设置域宽也可用于输入。当输入是字符串时，如果输入的字符个数大于设置的域宽，C++只读入域宽指定的字符个数。例如，有定义

```
char a[9] , b[9] ;
```

执行语句

```
cin >> setw(5) >> a >> setw(5) >> b;
```

用户在键盘上的响应为

abcdefghijklm

则字符串a的值为"abcd"，字符串b的值为"defg"。

4. 其他流操纵符

除了前面介绍的流操纵符以外，C++还提供了其他一些常用的流操纵符，如表14-2所示。

表14-2 其他常用的流操纵符

流 操 纵 符	描 述
skipws	跳过输入流中的空白字符，使用流操纵符noskipws复位该选项
left	输出左对齐，必要时在右边填充字符
right	输出右对齐，必要时在左边填充字符
showbase	指明在数字的前面输出基数，以0开头表示八进制，0x或0X表示十六进制，使用流操纵符noshowbase复位该选项
uppercase	指明当显示十六进制数时使用大写字母，并且在用科学记数法输出时使用大写字母E，使用流操纵符nouppercase复位该选项
showpos	在正数前显示加号(+)，使用流操纵符noshowpos复位该选项
scientific	以科学记数法输出浮点数
fixed	以定点小数形式输出浮点数
setfill	设置填充字符，它有一个字符型的参数

5. 自定义输出流操纵符

程序员可以定义自己的流操纵符，定义格式如下：

```
ostream &操纵符名(ostream &obj)
{需要执行的操作}
```

例如，在代码清单14-7中，用户定义了一个流操纵符tab，完成跳到下一打印区域的操作。

代码清单14-7 用户自定义流操纵符的示例程序

```
//文件名: 14-7.cpp
//用户的定义流操纵符示例
#include <iostream>
using namespace std;

ostream &tab(ostream &os) {return os <<'\t';}

int main()
{   int a=5,b=7;

    cout << a << tab << b <<endl;

    return 0;
}
```

代码清单14-7所示的程序的输出如下:

```
5      7
```

14.4 基于文件的输入/输出

14.4.1 文件的概念

文件是驻留在外存储器上、具有标识名的一组信息集合,用来永久保存数据。它可以是有格式的,也可以是无格式的。

与文件相关的概念有数据项(字段)、记录、文件和数据库。数据项是数据的基本单位,表示一个有意义的信息。例如,一个整型数、一个实型数或一个字符串。若干个相关的数据项组成一个记录,每个记录可以看成是一个对象。记录的集合称为文件。一组相关的文件构成了一个数据库。因此一个文件可以看成是二维表格,每一行是一个记录。

例如,在一个图书管理系统中,有一个数据库。这个数据库由书目文件、读者文件及其他辅助文件组成。书目文件中保存的是图书馆中所有书目的信息,每本书的信息构成一条记录。每本书需要保存的信息包括书名、作者、出版年月、分类号、ISBN号、图书馆的馆藏号以及一些流通信息。其中书名是一个字段,作者也是一个字段。

14.4.2 文件和流

C++把文件看成是字节序列,即由一个个字节顺序组成,每一个文件以文件结束符(EOF)结束,这种文件称为流式文件。把每个字节看成是一个ASCII字符时,称为文本文件。把每个字节看成是一个二进制值,则称为二进制文件。C++的文件没有记录的概念,如果要处理含有记录概念的文件,必须由程序员自己想办法解决。

C++把文件看成是一个数据流。要访问一个文件，必须先创建一个文件流。当应用程序从文件中读取数据时，则将文件与一个输入文件流对象（`ifstream`）相关联。当应用程序将数据写入一个文件时，则将文件与一个输出文件流对象（`ofstream`）相关联。如果既要输入又要输出，则与输入/输出文件流对象（`fstream`）相关联。这3个文件流类型定义在头文件`fstream`中。如果一个程序要对文件进行操作，必须在程序头上包含这个头文件。从图14-1中得知，`ifstream`和`ofstream`分别是`istream`和`ostream`派生的，`fstream`是从`iostream`派生的，因此文件的访问与控制台的输入/输出是一样的，除了所访问的流不同以外。控制台是从系统预先定义的输入流对象`cin`提取数据，将数据写到系统预定义的输出流对象`cout`，而文件读写时是读写应用程序定义的数据流对象。

因此，要访问一个文件，首先要将该文件与一个文件流对象相关联。这个操作称为打开文件。文件访问结束时，要断开文件和文件流对象的联系，这称为关闭文件。一旦文件被打开，它的操作与控制台输入/输出是一样的。

总结一下，访问一个文件由4个步骤组成：定义一个文件流对象，打开文件，操作文件中的数据，关闭文件。通常可以在定义文件流对象的同时打开文件，因此，第一和第二个步骤可以合二为一。

1. 定义文件流对象

C++有3个文件流类型：`ifstream`、`ofstream`和`fstream`。`ifstream`是输入文件流类，当要从文件读数据时，必须定义一个`ifstream`类的对象与之关联。`ofstream`是输出文件流类，当要向文件写数据时，必须定义一个`ofstream`类的对象与之关联。`fstream`类的对象既可以读也可以写。例如：

```
ifstream infile;
```

定义了一个输入文件流对象`infile`。一旦将这个对象与一个文件相关联，就可以用

```
infile >> x;
```

从文件中读取数据。

2. 打开和关闭文件

`ifstream`、`ofstream`和`fstream`类除了从基类继承下来的行为以外，还新定义了两个自己的成员函数`open`和`close`，以及一个构造函数。`ifstream`、`ofstream`和`fstream`对象可以调用这些操作，而其他输入/输出类型则不允许。

在打开文件时，无论使用成员函数`open`还是通过构造函数，都需要指定打开的文件的文件名和文件打开模式，因此，这两个函数都有两个形式参数：第一个形式参数是一个字符串，指出要打开的文件名；第二个参数是文件打开模式，它指出要对该文件做什么类型的操作。每个类都定义了一组表示不同打开模式的值，用于指定流打开的不同模式。文件打开模式及其含义如表14-3所示。文件流构造函数和`open`函数都提供了文件打开模式的默认参数。默认值因流类型的不同而不同。

表14-3 文件打开模式

文件打开模式	含 义
in	打开文件，做读操作
out	打开文件，做写操作
app	在每次写操作前，找到文件尾
ate	打开文件后，立即将文件定位在文件尾
trunc	打开文件时，清空文件
binary	以二进制模式进行输入/输出操作

out、trunc和app模式只能用于与ofstream和fstream类的对象相关联的文件。in模式只能用于与ifstream和fstream类的对象相关联的文件。所有的文件都可以用ate和binary模式打开。ate只在打开时有效，文件打开后将定位在文件尾。以binary模式打开的流则将文件以字节序列的形式处理，不解释流中字节的含义。

默认时，ifstream流对象是以in模式打开，该模式只允许对文件做读操作；与ofstream流关联的文件则以out模式打开，使文件可写，以out模式打开的文件时，如果文件不存在，会自动创建一个空文件，否则将被打开的文件清空，丢弃该文件原有的所有数据。对于ofstream对象，如果在打开时想要保存原文件中的数据，可以指定app模式打开，这样，写入文件的数据将被添加到原文件数据的后面。

如果要从文件file1中读取数据，需要定义一个输入流对象，并把它与file1相关联。这可以用下面两个语句实现：

```
ifstream infile;           //定义一个输入流对象
infile.open("file1");      //或infile.open("file1", ifstream::in); ，流对象与文件关联
```

也可以利用构造函数直接打开：

```
ifstream infile("file1");
```

或

```
ifstream infile("file1" , ifstream::in);
```

同样，如果要向文件file2中写数据，需要定义一个输出流对象，并把它与file2相关联。这可以用下面两个语句实现：

```
ofstream outfile;          //定义一个输出流对象
outfile.open("file2");      //或outfile.open("file2", ofstream::out); ，流对象与文件关联
```

也可以利用构造函数直接打开：

```
ofstream outfile("file2");
```

或

```
ofstream outfile("file2" , ofstream::out);
```

当执行上述语句时，如果file2已经存在，则会自动清空该文件。如果file2不存在，则会自动创建一个名为file2的文件。

有时，我们既需要从一个文件中读数据，又需要把数据写回该文件，此时可以定义一个 `fstream` 类的对象：

```
fstream iofile("file3");
```

默认情况下，`fstream`对象以 `in` 和 `out` 方式同时打开。也可以用显式指定文件模式，例如

```
fstream iofile("file3", fstream::in | fstream::out);
```

当文件同时以 `in` 和 `out` 方式打开时，不会清空文件。如果只用 `out` 模式而不指定 `in` 模式，文件会被清空。如果打开文件时指定了 `trunc` 模式，则无论是否指定 `in` 模式都会清空文件。如果以输入方式打开一个文件，但是该文件并不存在，或者以输出方式打开一个文件，但用户对文件所在的目录并无写的权限，那么将无法打开这个文件。如果文件打开不成功，流对象将会得到值 `0`。在打开文件后检查文件打开是否成功是一个良好的程序设计习惯。

当文件访问结束时，应该断开文件与文件流对象的关联。断开关联可以用成员函数 `close`。如果不再从 `file1` 读数据，可以调用

```
file1.close();
```

关闭文件，文件流对象和该文件不再有关。此时可以将此文件流对象与其他文件相关联，访问其他文件。

事实上，当程序执行结束时，系统会自动关闭所有的文件。尽管如此，显式地关闭文件是一个良好的程序设计习惯。特别是在一些大型的程序中，文件访问结束后关闭文件尤为重要。一旦在一个程序模块中没有关闭文件，在另一个模块中，就无法打开这个文件。

14.4.3 文件的顺序访问

C++文件的顺序读写和控制台读写一样，可以用流提取运算符 `>>` 从文件读数据，也可以用流插入运算符 `<<` 将数据写入文件，还可以用数据流类的其他成员函数读写文件，如 `get` 函数、`put` 函数等。

在读文件操作中，经常需要判断文件是否结束（文件中的数据是否被读完）。这可以通过基类 `ios` 的成员函数 `eof` 来实现。`eof` 函数不需要参数，返回一个整型值。当读操作遇到文件结束时，该函数返回 `1`；否则返回 `0`。另一种判断读结束的方法是用流提取操作的返回值。当 `>>` 操作成功时，返回 `true`；否则返回 `false`。

例14.1 将数字 `1~10` 写入文件 `file`，然后从 `file` 中读取这些数据，把它们显示在屏幕上。

首先用输出方式打开文件 `file`。如果文件 `file` 不存在，则自动创建一个；否则打开磁盘上名为 `file` 的文件，并清空。用一个循环依次将 `1~10` 用流插入符插入文件，并关闭文件。然后，再用输入方式打开文件 `file`，读出所有数据，并输出到屏幕上。具体的程序见代码清单14-8。

代码清单14-8 文件的顺序读写

```
//文件名: 14-8.cpp  
//文件的顺序读写
```

```

#include <iostream>
#include <fstream>          //使用文件操作必须包含fstream
using namespace std;

int main()
{
    ofstream out("file"); //定义输出流，并与文件file关联
    ifstream in;          //定义一个输入流对象
    int i;

    if (!out) {cerr << "create file error\n"; return 1;} //如打开文件不成功，则返回
    for (i = 1; i <= 10; ++i) out << i << ' ';          //将1~10写到输出流对象
    out.close();

    in.open("file");          //重新以输入方式打开文件file
    if (!in) {cerr << "open file error\n"; return 1;}
    while (in >> i) cout << i << ' ';                  //读文件，直到遇到文件结束
    in.close();

    return 0;
}

```

用流插入运算符输出信息时，是将数字转换成ASCII字符输出的。因此，如果在操作系统下直接显示文件file，可以看到文件file的内容如下：

```
1 2 3 4 5 6 7 8 9 10
```

注意代码清单14-8所示的程序中的写入部分。在输出每个i后紧接着输出了一个空格，这是为文件的读入做准备，因为在用流提取运算符读数据时是以空白字符作为分隔符的。

除了数字之外，文件也可以写入字符串，甚至可以既有数字，也有字符串。具体的示例见代码清单14-9。

代码清单14-9 写一个包含各种类型数据的文件操作的示例程序

```

//文件名: 14-9.cpp
//写一个包含各种类型的数据的文件操作的示例
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
    ofstream fout("test");

    if (!fout){cerr <<"cannot open output file\n"; return 1;}
    fout<<10<<" "<<123.456<<"\"This is a text file\"\n";
    fout.close();

    return 0;
}

```

执行了代码清单14-9所示的程序后，test文件中的内容如下：

```
10 123.456"This is a text file"
```

要读这个文件的内容，可以用代码清单14-10中的程序。

代码清单14-10 读有各种类型数据的文件的示例程序

```
//文件名: 14-10.cpp
//读有各种类型数据的文件的示例
#include <fstream>
#include <iostream>
using namespace std;

int main()
{   ifstream fin("test");
    char s[80];
    int i;
    float x;

    if (!fin) {cout << "cannot open input file\n"; return 1;}
    fin >> i >> x >> s; cout << i << " " << x << s;
    fin.close();

    return 0;
}
```

代码清单14-10所示的程序的输出是什么呢？某些读者可能会认为应该是

```
10 123.456"This is a text file"
```

但很遗憾，实际的结果是

```
10 123.456"This
```

这是因为在用流提取运算符读字符串时是以空白字符作为结束符，所以字符串s只读到了"This，后面就遇到了空格。如果要读入这个完整的字符串，可用其他的成员函数，如getline。

如果将代码清单14-8中的读入语句改为：

```
fin >> i >> x; fin.getline(s, 80, '\n');
```

则能够得到正确的结果。

14.4.4 文件的随机处理

为了从文件中读取数据，程序一般从文件起始位置开始顺序地读取数据，直到找到了所需的数据为止。例如，以输入方式打开一个文件后，第一次读文件时读入了4个字节，则第二次对此文件发出读操作时，就从第5个字节开始读。系统对每个文件流保存一个下一次读写的位置，这个位置称为文件定位指针。文件定位指针是一个long类型的数据。ifstream和ofstream分别提供了成员函数tellg和tellp返回文件定位指针的当前位置。tellg返回读文件定位指针，tellp返回写文件定位指针。下列语句将输入文件in的定位指针值赋给long类型的变量location：

```
location = in.tellg();
```

当文件用in方式打开时，文件定位指针指向文件头。当文件用app方式打开时，文件定位指针指向文件尾。

在程序执行过程中，有一部分数据可能需要访问多遍，这只需要将文件定位指针重新设回要读写的位置即可。ifstream和ofstream都提供了成员函数来重新设置文件定位指针。在ifstream中，这个函数为seekg，在ofstream中，这个函数称为seekp。seekg设置读文件的位置，seekp设置写文件的位置。seekg和seekp都有两个参数：第一个参数为long类型的整数，表示偏移量；第二个参数可以指定指针移动的参考点，ios::beg（默认）相对于流的开头，ios::cur相对于流当前位置，ios::end相对于流结尾。例如，in.seekg(0)表示定位到输入流in的开始处，in.seekg(10, ios::cur)表示定位到输入流in当前位置后面的第10个字节。

例如，对代码清单14-8中的程序做一个小小的修改，在用输入方式重新打开文件file后，插入语句

```
in.seekg(10);
```

即将读文件指针向后移10个字节，则该程序的输出结果如下：

```
6 7 8 9 10
```

通过重新设置文件定位指针，也可以改写文件中的内容，如代码清单14-11所示。这个程序用输入/输出的方式打开由代码清单14-8的程序生成的文件file，将写文件的位置定在第10个字节，然后重新写入20。也就是将原来6的位置改成了20。将读指针设回文件头，读整个文件。得到的输出如下：

```
1 2 3 4 5 20 7 8 9 10
```

代码清单14-11 文件的随机读写

```
//文件名: 14-11.cpp
//文件的随机读写
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    fstream in("file");
    int i;

    if (!in) {cerr << "open file error\n"; return 1;}

    in.seekp(10); //重新定位写文件指针
    in << 20;     //改写文件的内容

    in.seekg(0); //读文件定位指针移到文件起始处
    while (in >> i) cout << i << ' ';
    in.close();

    return 0;
}
```

从代码清单14-11中的程序的执行结果可以看出,用流插入运算符改写文件的数据可能会破坏文件中其他的数据。例如,程序的原意可能是想把6修改成20,但修改后的20却和原来的7连在一起变成了207。因此想要修改文件的内容,用这种方法是非常危险的。

14.4.5 用流式文件处理含有记录的文件

信息系统中用到的文件一般都有记录的概念,如图书馆系统中的书目文件。当新书到达时,需要将书目信息添加到文件中去。当用户借还书时,需要修改这本书的流通信息。因此,这些文件要求做到以下几点。

- 立即访问到文件甚至大型文件中指定的记录。
- 可以在不破坏其他数据的情况下把数据插入到随机访问文件中。
- 也能在不重写整个文件的情况下更新以前存储的数据。

要实现上述要求,必须要求记录长度是固定的。显然,用流提取运算符和流插入运算符无法达到上述要求,因为它们无法固定记录的长度。例如,用流插入运算符保存整型数123到文件流outFile(即outFile << 123),则这个整型数在文件中占了3个字节。如果保存的整型数的值是12345(即outFile << 12345),则在文件中要占5个字节。一旦程序将整型数123改写成整型数123456,则会影响到文件中的其他数据。事实上,每个整型数,不管它的值是多少,在内存都占4个字节。如果能把整型数在内存中的存储模式原式原样写在文件中,就能保证每个整型数在文件中占用的长度是一样的。这时,不管如何修改这个整型数,它在文件中占用的空间始终是4个字节,不会影响文件中的其他数据。istream中的read函数和ostream中的write函数可以实现这一需求。例如,若number是整型变量,可以用语句

```
outFile.write(reinterpret_cast<const char * > (&number)), sizeof(number));
```

将存储整型数的4个字节(如果整型数以4个字节表示)看成是一个由4个字符组成的字符数组,写入文件流对象outFile。函数write将第一个参数作为一组字节数据,将内存中的整型对象看成是const char *类型的,write函数从这个地址开始,将第二个参数指定的字节数写入文件流。用write函数写入的这4个字节可以用read函数读入。如果这个文件与输入流对象inFile关联,这4个字节可以用下列语句读入,并重新解释成整型数:

```
inFile.read(reinterpret_cast<char * > (&number)), sizeof(number));
```

由于number是一个整型变量,当引用number时会把这4个字节的内容重新解释成整型数。注意上面的reinterpret_cast,它要求编译器将操作数重新解释成目标类型,即尖括号中的类型。在write函数中,将number的地址强制转换成const char *类型的指针,即将number中的内容重新解释成输出到文件的字节序列。在read函数中,将number的地址强制转换成char *类型的指针,将从文件读入的字节序列放入该地址。

例14.2 设计一个图书馆的书目管理系统。如果每本书需要记录的信息有馆藏号(整型数)、书名(最长20个字符的字符串)和借书标记。借书标记中记录的是借书者的借书证号(假设也是整型数),如书未被出借,则借书标记值为0。图书的馆藏号要求自动生成。即系统的第一本书馆

藏号为1，第二本书馆藏号为2，依次类推。该系统需要实现的功能如下：初始化系统、添加书、借书、还书和显示书库信息。初始化系统时，要清除所有保存的书目信息，并且重新从1开始编号。添加书功能要求用户输入书名，系统自动生成馆藏号，设置在库标记，将此信息存入系统的数据库。借书功能输入借书证号及所借书的馆藏号，根据馆藏号从数据库中取出相应的书目信息，将借书证号记入所借书的借书标记，写回数据库。还书功能要求输入书的馆藏号，从数据库中取出相应的书目信息，复位该书的借书标记，写回数据库。显示书库信息将数据库中所有的书目信息按馆藏号的次序显示。

每个信息系统都有自己的数据库，本系统的数据库比较简单。由于本系统只有书目信息需要长期保存，为此，设计了一个文件Book，该文件中的每个记录保存一本书的信息。文件中的记录可按馆藏号的次序存放，这样可方便实现添加书和借还书的功能。添加书时，只要将这本书对应的记录添加到文件尾。借还书时，可以根据馆藏号计算记录的存储位置，修改相应的记录。

按照面向对象程序设计思想，在设计一个软件时首先要考虑需要哪些工具。那么，这个软件需要哪些工具呢？这个软件主要处理的对象是“书”，如果能有一个处理“书”的工具，则软件的实现会简单许多。因此，应该为这个系统设计一个书目类，用于处理一本书的信息。根据题意，保存一本书需要保存3个信息，因此这个类有3个数据成员：馆藏号、书名和借书标记。为了提供馆藏号自动生成，需要提供一个共享的信息，即系统中最大的馆藏号。当添加一本新书时，这个值加1，并作为新加入的书的馆藏号。这个值可以作为书目类的静态成员。对于每一本书，除了构造函数外，可能的操作有借书、还书、显示书的详细信息。由于这个类有一个静态的成员，还应该对静态成员进行操作的函数。这个静态成员有两个操作：一个是当初始化系统时，要清除所有保存的书目信息，并且重新从0开始编号；另一个是，每次添加一本新书时，这个成员值要加1。根据上述考虑设计的类如代码清单14-12所示。

代码清单14-12 Book类的设计

```
//文件名: book.h
//Book类的设计
#ifndef _book_h
#define _book_h

#include <cstring>
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

class Book {
    int no;
    char name[20];
    int borrowed;
    static int no_total;
public:
    Book(const char *s = "") ;
    void borrow(int readerNo);
```

```

        void Return();
        void display();
        static void resetTotal();
        static void addTotal();
    };
#endif

//book.cpp
#include "book.h"

Book::Book(const char *s) {no = no_total; borrowed = 0; strcpy(name,s);}

void Book::borrow(int readerNo)
{   if (borrowed != 0) cerr << "本书已被借，不能重复借\n"; else borrowed = readerNo; }

void Book::Return()
{   if (borrowed == 0) cerr << "本书没有被借，不能还\n";   else   borrowed = 0; }

void Book::display()
{   cout << setw(10) << no << setw(20) << name << setw(10) << borrowed << endl;}

void Book::resetTotal() {   no_total = 0;}
void Book::addTotal() {   ++no_total;}
int Book::no_total = 0;

```

有了合适的工具后，就可以着手整个系统的设计。整个系统由5大功能组成，每个功能定义成一个函数。主函数显示一个功能菜单，并根据用户选择的项目执行相应的函数。main函数的实现如代码清单14-13所示。

代码清单14-13 图书馆系统的主函数

```

//文件名: 14-13.cpp
//图书馆系统的主函数
#include "book.h"
void initialize();    //系统初始化
void addBook();       //添加新书
void borrowBook();    //借书
void returnBook();    //还书
void displayBook();   //显示所有的书目信息

int main()
{   int selector;

    while (true) {
        cout << "0 -- 退出\n";
        cout << "1 -- 初始化文件\n";
        cout << "2 -- 添加书\n";
        cout << "3 -- 借书\n";
        cout << "4 -- 还书\n";
        cout << "5 -- 显示所有书目信息\n";
        cout << "请选择 (0-5): "; cin >> selector;
        if (selector == 0) break;
    }
}

```

```

        switch (selector) {
            case 1: initialize(); break;
            case 2: addBook(); break;
            case 3: borrowBook();break;
            case 4: returnBook(); break;
            case 5: displayBook();break;
        }
    }
    return 0;
}

```

每个功能的实现都非常简单，不需要再继续分解。系统初始化将书目文件清空，并将最大的馆藏号复位。系统初始化函数的实现如代码清单14-14所示。

代码清单14-14 系统初始化函数

```

void initialize() {
    ofstream outfile("book"); //清空文件book
    Book::resetTotal();        //调用Book类的静态成员函数，将最大的馆藏号置为0
    outfile.close();
}

```

添加新书功能将新入库的书的信息添加到文件尾，因此要以app方式打开文件。然后，将用户输入的书名作为参数生成一个Book类的对象，用write函数写入文件。这样，每条记录在文件中所占的字节数都是相同的，即Book类对象的长度。添加新书的实现如代码清单14-15所示。

代码清单14-15 添加新书的实现

```

void addBook() {
    char ch[20];
    Book *bp;
    ofstream outfile("book",ofstream::app); //以app方式打开文件

    Book::addTotal(); //最大的馆藏号加1
    cout << "请输入书名: "; cin >> ch;
    bp = new Book(ch);

    outfile.write( reinterpret_cast<const char *>(bp), sizeof(*bp));
    delete bp;
    outfile.close();
}

```

当用户要借书的时候，需要输入馆藏号和读者的借书证号。根据馆藏号从文件中读出相应的图书记录，更新借书记录字段，将记录重新写回文件。因此，在这个函数中，文件要能读能写，我们定义了一个fstream的对象与之关联。由于在此文件中，每条记录的长度是定长的，因此可以方便地定位到所要读写的记录，更新操作非常方便。借书函数的实现如代码清单14-16所示。

代码清单14-16 借书函数的实现

```

void borrowBook()

```

```
{    int bookNo, readerNo;
    fstream iofile("book");           //以读写方式打开文件
    Book bk;

    cout << "请输入书号和读者号: "; cin >> bookNo >> readerNo;

    iofile.seekg((bookNo - 1) * sizeof(book)); //按照馆藏号定位到所读记录
    iofile.read( reinterpret_cast<char *> (&bk), sizeof(Book) );
                                           //读一条记录, 存入对象bk

    bk.borrow(readerNo);                //调用成员函数修改借书记字段

    iofile.seekp((bookNo - 1) * sizeof(Book)); //按照馆藏号定位到所写记录
    iofile.write( reinterpret_cast<const char *>(&bk), sizeof(Book)); //更新记录

    iofile.close();
}
```

还书函数的实现与借书函数类似, 如代码清单14-17所示。

代码清单14-17 还书函数的实现

```
void returnBook()
{    int bookNo;
    fstream iofile("book");
    Book bk;

    cout << "请输入书号: "; cin >> bookNo ;

    iofile.seekg((bookNo - 1) * sizeof(Book));
    iofile.read(reinterpret_cast<char *> (&bk), sizeof(Book));

    bk.Return();           //复位借书标记

    iofile.seekp((bookNo - 1) * sizeof(Book));
    iofile.write( reinterpret_cast<const char *>(&bk), sizeof(Book));

    iofile.close();
}
```

显示所有的图书信息的程序非常简单, 只要输出文件中的所有记录就可以了。具体的实现如代码清单14-18所示。

代码清单14-18 显示书目信息的实现

```
void displayBook()
{    ifstream infile("book");
    Book bk;

    infile.read(reinterpret_cast<char *> (&bk), sizeof(Book));

    while (!infile.eof()) { //顺序读文件, 直到结束
```

```

bk.display(); //显示当前书目信息的内容
infile.read(reinterpret_cast<char *> (&bk), sizeof(Book));
}
infile.close();
}

```

14.5 基于字符串的输入/输出

`iostream`标准库支持内存中的输入/输出，只要将流与存储在程序内存中的`string`对象捆绑起来即可。此时，可使用`iostream`的输入/输出操作符读写这个`string`对象。标准库定义了3种类型的字符串流：

- `istringstream`：由`istream`派生而来，提供读`string`的功能。
- `ostreamstream`：由`ostream`派生而来，提供写`string`的功能。
- `stringstream`：由`iostream`派生而来，提供读写`string`的功能。

要使用上述类，必须要包含`sstream`头文件。

由于这些类都是由`istream`、`ostream`和`iostream`派生而来的，这意味着`iostream`上的所有操作都适用于`sstream`中的类型。`sstream`中的类型除了继承来的操作外，还各自定义了一个有一个`string`形式参数的构造函数，这个构造函数将形式参数复制给相应的字符串流对象。这些类还定义了一个名为`str`的成员函数，用以读取或设置字符串流对象所操纵的`string`。这些类的一个使用示例如代码清单14-19所示。

代码清单14-19 字符串流使用示例

```

//文件名: 14-19.cpp
//字符串流的使用示例
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main()
{
    string ch;
    ostreamstream os(ch); //或ostreamstream os; 定义一个字符串流类的对象

    for (int i = 0; i<=20; ++i) os << i << ' '; //将1~20写入os
    cout << os.str(); //显示写入os的内容，为1~20，数字间用空格分开
    cout << endl;

    istringstream is(os.str()); //用os对应的字符串初始化输入流is

    while (is >> i) cout << i << '\t'; //从输入流读数据，直到输入流结束

    return 0;
}

```

小结

输入/输出是程序中不可缺少的一部分。在C++中，输入/输出功能是以标准库的形式提供的。输入/输出操作分为控制台输入/输出，文件输入/输出以及字符串输入/输出。由于文件输入/输出和字符串输入/输出类都是从控制台输入/输出类继承的，因此，这3种输入/输出的操作方式是相同的。

本章介绍了如何利用*iostream*库进行格式化的输入/输出，介绍了如何利用文件永久保存信息，并以图书馆系统为例，介绍了如何实现文件的随机读写。

习题

简答题

1. 什么是打开文件？什么是关闭文件？为什么需要打开和关闭文件？
2. 既然程序执行结束时系统会关闭所有打开的文件，为什么程序中还需要用*close*关闭文件？
3. C++有哪4个预定义的流？

程序设计题

1. 编写一个文件复制程序*copyfile*，要求在命令行界面中通过输入

```
copyfile src_name obj_name
```

将名为*src_name*的文件复制到名为*obj_name*的文件中。

2. 修改例14.2使之能实现图书预约功能。
3. 编写一个程序，打印1~100的数字的平方和平方根。要求用格式化的输出，每个数字的域宽为10，实数用小数位置占5位的精度右对齐显示。
4. 编写一个程序，打印所有英文字母（包括大小写）的ASCII值。要求对于每个字符，程序都要输出它对应的ASCII值的十进制、八进制和十六进制表示。
5. 利用第10章的程序设计题的第2题中定义的*SavingAccount*类，设计一个银行系统，要求该系统为用户提供开户、销户、存款、取款和查询余额的功能。

在程序设计中，通常会设计出这样一些程序：在一般情况下，它不会出错，但在一些特殊的情况下，就不能正确运行。例如，一个完成两个数除法的程序。用户从键盘输入两个数 a 和 b ，输出 a/b 的结果。一般情况下，该程序执行是正确的，但当输入的 b 是 0 时，该程序就会出错。为了保证程序的健壮性，需要对程序中可能出现的异常情况进行考虑，并给出相应的处理。例如，在除法程序中，当输入了 a 和 b 后，应该对 b 进行检查：如果 b 等于 0 ，则报错；否则给出除法的结果。

15.1 传统的异常处理方法

传统的异常处理就像除法程序一样，在检测到错误时，就地解决。在这种处理方式中，错误处理代码分布于整个系统代码中。代码中可能出错的地方都要当场进行错误处理。这种做法的好处是：程序员阅读代码时能够直接看到错误处理情况，确定是否实现了正确的错误检查。它的缺点是：代码受到错误处理的“污染”，使应用程序本身的代码更加晦涩难懂，难以看出功能是否正确实现，这样就使代码的理解和维护更加困难。

15.2 异常处理机制

传统的异常处理方法不能满足大规模程序设计的要求。在大规模程序设计中，程序通常被分成了很多函数。写函数的程序员可以检测到函数运行时的错误，但通常却不知道应该如何处理这些错误。例如，把一个除法程序写成一个函数，当在函数中遇到了除数为 0 的情况，函数该如何处理呢？是退出整个程序，还是忽略这次除法让程序继续运行？函数本身很难决定，这取决于函数被用在什么场合。如果该除法运算是某个大型计算中的一个步骤，出现除零错将会使整个计算无法进行下去，此时应该终止程序的运行；但是，如果调用该函数的是一个计算器软件，出现除零错时，将忽略这次除法，请用户输入下一个计算请求。因此，如何处理这个错误必须由调用除法函数的函数来决定。C++异常处理的基本想法就是“踢皮球”，矛盾上交，让一个函数在发现自己无法处理的错误时抛出一个异常，希望它的（直接或间接）调用者能够处理这个问题。这种做法去除了程序中的错误处理代码，可以使程序的“主线条”更加清晰。

基于这个思想，C++把异常处理工作分成了异常抛出以及异常捕获和处理两个部分。

15.2.1 异常抛出

如果程序发生了异常情况，而在当前的环境中获取不到异常处理的足够信息，我们可以创建一个包含出错信息的对象并将该对象抛出当前环境，将错误信息发送到更大的环境中，这称为异常抛出。例如：

```
throw myerror("something bad happened");
```

`myerror`是用户自己定义的一个类，它有一个以字符串为形式参数的构造函数。这条语句构建一个`myerror`类的对象并抛出，而`throw 5`；则抛出了一个整型数。

异常抛出语句的一般形式如下：

```
throw <操作数>;
```

`throw`指定一个操作数，`throw`的操作数可以是任何类型。如果操作数是一个对象，则称为异常对象。`throw`也可以抛出一个结果为任何类型的表达式而不是抛出对象。

抛出异常时，生成和初始化`throw`操作数的一个临时副本，传回处理该异常的程序。

例15.1 定义一个除法函数，当除数为0时，抛出一个用户定义的异常类对象，该对象能告诉用户发生了除零错。

按照题意，定义一个异常类。这个类一般只需要能告诉用户出现了什么异常，因此这个类需要一个数据成员，记录出现的异常，需要一个成员函数，告诉用户出现了什么异常。这个异常类的定义如代码清单15-1所示。

代码清单15-1 异常类的定义

```
//类DivideByZeroException是用户定义的类，用于表示除0错
class DivideByZeroException {
public:
    DivideByZeroException(): message( "attempted to divide by zero" ) { }
    const char *what() const { return message; }
private:
    const char *message;
};
```

当设计一个除法函数时，一旦检测到除数为0，则抛出这个异常类的对象，如代码清单15-2所示。

代码清单15-2 异常对象的抛出

```
double div(int x, int y)
{ if (y == 0) throw DivideByZeroException();

    return static_cast< double > (x) / y;
}
```

在代码清单15-2所示的函数中，如果`y`不等于0，函数返回`x/y`的值。如果`y`等于0，则执行


```
throw DivideByZeroException();
```

这条语句用默认构造函数生成了一个`DivideByZeroException`类的对象，并把这个对象返回给了调用它的函数，`div`函数执行结束。

抛出的异常不一定是对象，可以是一个表达式。例如，代码清单15-3中的函数在检测到错误时，抛出一个整型数。

代码清单15-3 抛出异常表达式

```
int div(int x, int y)
{
    if (y==0) throw y;
    return x/y;
}
```

15.2.2 异常捕获

一个函数抛出异常，它必须假定该异常能被处理它的程序捕获和处理。异常捕获机制使C++可以把问题集中在一处解决。

如果某段程序可能会抛出异常，则必须通知系统启动异常处理机制。这是通过`try`语句块实现的。C++中，异常捕获的格式如下：

```
try{
    可能抛出异常的代码;
}
catch(类型1 参数1){ 处理该异常的代码 }
catch(类型2 参数2){ 处理该异常的代码 }
...
```

`try`块中包含了可能抛出异常的代码。一旦抛出了异常，则退出`try`块，进入`try`后面的异常捕获和处理。异常处理器放在`catch`块中，它的形式如下：

```
catch (捕获的异常类型 [参数]) {
    异常处理代码;
}
```

`catch`处理器定义了自己处理的异常范围。`catch`在括号中指定要捕获的参数类型以及参数名。`catch`处理器中的参数名是可选的。如果给出了参数名，则可以在处理器中引用这个参数名。如果没有指定参数名，只指定匹配抛出对象的类型，则信息不从抛出点传递到处理器中，只是把控制从抛出点转到处理器中。许多异常都可以这样。如果某个异常处理器要捕获所有的异常，可以用“...”代替捕获的类型和参数名。例如：

```
catch(...)
```

可以捕获任意类型的异常。如果`try`块没有抛出异常，则执行完`try`块的最后一个语句后，跳过所有的`catch`处理器，执行所有`catch`后的语句。如果`try`块抛出了异常，则跳出`try`块，开始异常捕获。先将抛出的异常与第一个异常处理器相比较，如果可以匹配，则执行异常处理代码，然

后转到所有catch后的语句继续执行。如果不匹配，则与下一异常处理器比较，直到找到一个匹配的异常处理器。如果找遍了所有的异常处理器，都不匹配，则函数执行结束，并将该异常抛给调用它的函数，由调用它的函数来处理该异常。例如，有了DivideByZeroException这个异常类，我们可以写一个带有异常检测的除法程序，如代码清单15-4所示。

代码清单15-4 带有异常检测的除法程序

```
//文件名: 15-4.cpp
//带有异常检测的除法程序
int main()
{   int number1, number2;
    double result;
    cout << "Enter two integers (end-of-file to end): ";
    while (cin >> number1 >> number2) {
        try {
            if (number2 == 0) throw DivideByZeroException();
            result = static_cast< double > (number1) / number2;
            cout << "The quotient is: " << result << endl;
        }
        catch ( DivideByZeroException ex) {
            cout << "Exception occurred: " << ex.what() << '\n';    }
            cout << "\nEnter two integers (end-of-file to end): ";
        }
    }
    cout << endl;
    return 0;
}
```

代码清单15-4所示的程序重复下列过程：提示用户输入两个数，执行两个数的除法，输出结果。由于在执行除法时，可能会遇到除数为0的情况，因此把执行除法的那段程序放入了一个try块。当number2不等于0时，执行除法，并输出除的结果，退出try块。由于该try块没有抛出异常，因此跳过所有的catch，执行catch后的语句，即显示提示信息，重新开始一次除法。如果遇到number2为0的情况，程序立即跳出try块，开始了异常捕获，计算和显示除法结果的语句都不执行了。第一个异常处理器就是一个匹配的处理器，于是执行该处理器的处理代码，即显示出错的内容，然后执行所有catch后的语句，即显示提示信息。由于该异常捕获时，设置了对对象的名字，因此在异常处理语句中可以用这个对象。这个程序的某次运行结果如下：

```
Enter two integers (end-of-file to end); 100 7
The quotient is: 14.2857
Enter two integers (end-of-file to end); 100 0
Exception occurred: attempted to divide by zero
Enter two integers (end-of-file to end); 33 9
The quotient is: 3.66667
Enter two integers {end-of-file to end):
```

也可以把除法写成一个函数，如代码清单15-2所示。该函数只有抛出异常的语句，而没有异常捕获。当函数抛出异常时，将会回到调用它的函数，由调用它的函数来处理异常。它的使用如代码清单15-5所示。

代码清单15-5 抛出异常的函数的应用

```
//文件名: 15-5.cpp
//抛出异常的函数的应用
int main()
{   int number1, number2;
    double result;
    cout << "Enter two integers (end-of-file to end): ";
    while ( cin >> number1 >> number2 ) {
        try { result = div(number1, number2);
            cout << "The quotient is: " << result << endl;
        }
        catch (DivideByZeroException ex) {
            cout << "Exception occurred: " << ex.what() << '\n';
            cout << "\nEnter two integers (end-of-file to end): ";
        }
    }
    cout << endl;
    return 0;
}
```

由于div函数可能抛出一个异常，因此main函数将它放在了一个try块中。一旦div抛出了异常，则会返回到main函数，而main函数会跳出try块，执行异常处理。这个程序的执行过程与代码清单15-4所示的程序的执行过程完全一样。

异常捕获时也可以只捕获类型，而不指明该类型的对象。例如，对于代码清单15-3所示的函数，可以写一个应用它的main函数，如代码清单15-6所示。

代码清单15-6 只捕获类型，不指明对象的实例

```
int main()
{   try {
        cout << div(6, 3) << endl;
        cout << div(10, 0) << endl;
        cout << div(5, 2) << endl;
    }
    catch (int) {cout << "divide by zero" << endl; }
    cout << "It's Over" << endl;
    return 0;
}
```

代码清单15-6中的程序先执行6/3，并显示除法的结果，然后执行10/0，这时div函数会抛出一个整型的异常。由于div函数没有处理该异常，函数执行结束，返回main函数。程序的控制转到catch语句。第一个异常处理器就是一个匹配的处理器。执行该异常处理语句，输出divide by zero，再继续执行所有catch后面的语句，即一个输出语句，显示It's Over，程序执行结束。

15.3 异常规格说明

如代码清单15-6所示，调用一个函数可能会收到一个异常。调用一个函数时，如何知道是否会收到异常呢？收到的是什么样的异常？这可以通过异常规格说明来实现。

当一个程序用到某一函数时，先要声明该函数。函数声明的形式如下：

返回类型 函数名(形式参数表);

当这样声明一个函数时，表示这个函数可能抛出任何异常。通常我们希望在调用函数时，知道该函数会抛出什么样的异常，我们可以对每个抛出的异常做相应的处理。C++允许在函数原型声明中指出函数可能抛出的异常。例如：

```
void f() throw(toobig,toosmall,divzero);
```

表示函数f会抛出3个异常类toobig、toosmall、divzero的对象。代码清单15-7给出了一个示例。

代码清单15-7 抛出指定异常的函数示例

```
//文件名: 15-7.cpp
//抛出指定异常的函数示例
#include <iostream>
using namespace std;
class up{};
class down{};
void f(int i) throw(up, down);    //f函数可能抛出两类异常: up和down

int main()
{   for (int i=1;i<=3;++i)   try { f(i); }
    catch (up) {cout << "up caught" << endl; }
    catch (down) {cout << "down caught" << endl;}
    return 0;
}

void f(int i) throw(up,down)
{   switch(i) {
        case 1: throw up();
        case 2: throw down();
    }
}
```

在代码清单15-7的程序中，定义了两个异常类up和down，并声明函数f会抛出这两类的异常。main函数知道函数f会抛出这两个异常，因此把函数f的调用写在一个try块中，try块的后面是这两个异常的处理器。这个程序对每种异常都进行了处理，因此可靠性比较高，不太可能出现异常终止。

如果一个函数把所有的问题都自己解决了，不需要抛出异常，则可以在函数声明中用throw()表示不抛出异常。例如：

```
void f() throw();
```

说明函数f不会抛出异常。

小结

为了提高程序的健壮性，程序需要对各种可能的异常进行处理，某些错误可能需要异地处理，

为此，C++提出了一种新的异常处理机制。C++的异常机制由try、throw和catch构成。异常的抛出用throw语句实现。try块把可能抛出异常的代码括在其中，try块后面是一组异常处理器。如果try块中出现了某个异常，则忽略try块中后面的语句，跳出try块，将抛出的异常与异常处理器逐个比较，直到找到一个匹配的异常处理器，执行该异常处理器的语句。如果没有可供匹配的异常处理器，则函数终止，把该异常继续抛向其调用函数。

习题

简答题

1. 采用异常处理机制有什么优点？
2. 是不是每个函数都需要抛出异常？
3. 如何让函数的使用者知道函数会抛出哪些异常？

程序设计题

1. 修改第11章中的IntArray类，使之在下标越界时抛出一个异常。
2. 写一个安全的整型类，要求可以处理整型数的所有操作，且当整型数操作结果溢出时，抛出一个异常。

第 16 章

容器和迭代器

16

16.1 容器

容器是特定类型的对象的集合，是为了保存一组类型相同的对象而设计的类。容器一般需要提供插入对象、删除对象、查找某一对象以及按某种次序访问容器中的所有对象等功能。

一旦定义了一个保存对象的容器，用户可以使用容器提供的插入操作将对象放入容器，用删除操作将对象从容器中删除，而不必关心容器中的对象是如何保存的。这将会给需要处理一组同类对象的用户带来极大的便利。

容器可以存放各种类型的对象。如可以设计一个存放整型对象的容器，也可以设计一个存放实型对象的容器，也可以设计一个存放用户自定义类型的对象的容器。不管容器存放的是什么类型的对象，它们的操作都是类似的。因此，容器通常都被设计成一个模板。

16.2 迭代器

容器可以用多种方法实现。数组是容器的一种实现方法，链表也是容器的一种实现方法。当要访问容器中的某一对象时，必须要有一种方式标识该对象。如果用一个数组保存对象，则可用数组的下标标识正在访问的元素，当用链表保存对象时，可以用一个指向某一结点的指针来标识链表中的某个元素。但容器是抽象的，它保存对象的方法对用户是保密的，因而用户无法知道该如何标识容器中的某一对象。为此，通常为每种容器定义一个相应的表示其中对象位置的类型，称为迭代器。迭代器对象相当于是指向容器中对象的指针。迭代器对象“穿行”于容器，返回容器中的下一对象或对指向的对象执行某种操作。有了迭代器，可以进一步隐藏数据的存储方式。

事实上，可以把迭代器看成是一种抽象的指针。因此，迭代器的常用操作和指针类似，包括给迭代器对象赋值，迭代器对象的比较，让迭代器移到当前对象的下一对象，取迭代器指向的对象，以及判迭代器指向的对象是否存在（即是否为空指针）等。

16.3 容器和迭代器的设计示例

在本节中，我们分别以一个用数组实现的顺序容器和一个用链表实现的顺序容器的设计为

例，说明容器和迭代器的设计。所谓的顺序容器是指容器中的对象之间具有线性关系，可以按某种次序将对象进行排序。

16.3.1 用数组实现的容器

首先看一看用数组实现的容器。由于是用数组实现，因此，该容器可以很容易地重载下标运算符，通过下标运算符（[]）访问容器中的对象，但也可以用迭代器访问容器中的对象。容器的具体功能如下：可以将数据依次放入容器，当数组不够大时，容器自动扩展数组；删除最后放入的对象；在迭代器指出的位置插入一对象；删除迭代器指出的位置的元素。迭代器包括的常规操作如下：设置迭代器的初始位置，向后移一位置，取迭代器指向的对象，以及迭代器对象的等于比较。

要设计这样一个容器，需要设计两个类。一个是用数组实现的容器类，一个是相应的迭代器类。由于这个迭代器类是对应的容器专用的，因此可将迭代器类设计成容器类的内嵌类。根据题意，容器的行为可设计为：在容器尾放入一对象，删除容器尾的对象，在迭代器指出的位置放入一对象，删除迭代器指出的位置中的对象；取容器的首尾位置。由于两个插入行为都可能引起数组空间的扩展，因此，可将扩展数组空间作为容器类的私有行为。迭代器类的行为可设计为：取迭代器对象的值，迭代器向后移动，比较两迭代器是否相同。按照上述思路设计的容器和迭代器类如代码清单16-1所示，成员函数的实现如代码清单16-2所示，该容器的使用如代码清单16-3所示。

代码清单16-1 用数组实现的顺序容器类

```
template <class T>
class seqList {
private:
    int size;                                //数组规模
    int current_size;                        //容器中的对象个数
    T *storage;                              //数组的起始地址
    void doubleSpace();                     //将数组容量扩大一倍

public:
    seqList(int s = 10):size(s) {storage = new T[size]; current_size = 0;}
    ~seqList() {delete [] storage;}
    void push_back(const T &x)              //在容器尾添加对象
    {   if (size == current_size) doubleSpace();
        storage[current_size++] = x;
    }
    void pop_back( )                        //删除容器尾的对象
    { if (current_size == 0) cerr << "这是一个空容器\n"; else --current_size; }
    T &operator[](int i) { return storage[i]; } //下标运算符重载

    class Itr {                             //迭代器类的定义
        T *pos;                             //指向容器中的某一对象
    public:
        Itr(T *obj = NULL) {pos = obj;}
```

```

        Itr &operator++() { ++pos; return *this;} //指向容器中的下一对象
        T &operator*() { return *pos;} //取迭代器指向的对象值
        bool operator!=(const Itr &p) {return pos != p.pos;} //比较两迭代器对象是否相同

        friend class seqList<T>;
    };

    Itr begin() {return Itr(storage);} //返回指向第一个对象的迭代器
    Itr end() { return Itr(storage + current_size); } //返回指向最后一个对象的迭代器
    void insert(Itr &p, const T &x) ; //在迭代器指定的位置上插入对象
    void erase(const Itr &p); //删除迭代器指定的位置中的对象
};

```

代码清单16-2 用数组实现的顺序容器类的成员函数的实现

```

template <class T>
void seqList<T>::doubleSpace()
{
    T *tmp = storage;

    size *= 2;
    storage = new T[size];
    for (int i = 0; i < current_size; ++i) storage[i] = tmp[i];
    delete [] tmp;
}

template <class T>
void seqList<T>::insert( Itr &p, const T &x)
{
    T *q;

    if (size == current_size) {
        int offset = p.pos - storage;
        doubleSpace();
        p.pos = storage + offset; //迭代器指回新空间中的相应对象
    }
    q = storage + current_size;

    while (q > p.pos) { *q = *(q-1); --q;} //迭代器指出的对象后的所有对象后移一个位置

    *p.pos = x;
    ++current_size;
}

template <class T>
void seqList<T>::erase(const Itr &p)
{
    T *q = p.pos ;

    --current_size;
    while (q < storage + current_size) { *q = *(q+1); ++q;}
}

```


代码清单16-3 SeqList类的使用

```

//文件名: 16-3.cpp
//顺序容器的使用示例
int main()
{
    seqList<int> sq(10);           //定义一个存放整型对象的顺序容器, 初始容量为10个对象
    seqList<int>::Itr itr1;        //定义一个对应的迭代器类的对象

    for (int i = 0; i < 10; ++i) sq.push_back(2 * i + 1);

    cout << "用下标运算符输出:\n";
    for (i = 0; i < 10; ++i) cout << sq[i] << '\t';

    cout << "用迭代器输出:\n";
    for (itr1 = sq.begin(); itr1 != sq.end(); ++itr1) cout << *itr1 << '\t';

    //插入0,2,4,6,8,10,12,14,16,18
    for (itr1 = sq.begin(), i = 0; i < 20; ++itr1, ++itr1, i += 2) sq.insert(itr1, i);

    cout << "插入0,2,4,6,8,10,12,14,16,18后:\n";
    for (itr1 = sq.begin(); itr1 != sq.end(); ++itr1) cout << *itr1 << '\t';

    //删除0,2,4,6,8,10,12,14,16,18
    for (itr1 = sq.begin(); itr1 != sq.end(); ++itr1) sq.erase(itr1);

    cout << "删除0,2,4,6,8,10,12,14,16,18后:\n";
    for (itr1 = sq.begin(); itr1 != sq.end(); ++itr1) cout << *itr1 << '\t';

    return 0;
}

```

代码清单16-3所示的程序首先将1, 3, 5, ..., 19依次插入容器, 然后分别用下标运算符和迭代器输出容器中的所有对象, 得到的输出如下:

```

用下标运算符输出:
1 3 5 7 9 11 13 15 17 19
用迭代器输出:
1 3 5 7 9 11 13 15 17 19

```

这两种方式输出的结果是相同的。接下来测试用迭代器指定位置的插入。这里的循环就是让迭代器依次指向1, 3, 5, ..., 然后在当前位置上插入0, 2, 4, ...。由于容器的初始大小是10, 因此插入时自动扩展了数组的空间。插入后容器中的内容为

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

最后, 测试删除功能。依次让迭代器指向0, 2, 4, ..., 然后删除迭代器指向的对象。删除后, 容器中又只剩下1 3 5 7 9 11 13 15 17 19。

16.3.2 用链表实现的容器

下面再来看一下用单链表实现的顺序容器。如果该容器需要提供的功能如下: 在迭代器指出

的位置后插入一对象，迭代器指向新插入对象；删除迭代器指出的位置后的对象。迭代器提供的常规操作如下：设置迭代器的初始位置，向后移一位置，判断迭代器指向的位置是否有对象，取迭代器指向的对象。

如第8章所述，链表中的每一元素存储在一个结点中，因此需要一个结点类。结点类是链表专用的，链表类的用户无需知道有这样的一个类型存在，因此，结点类可设计成链表类的内嵌类，并且是私有的。与容器相关的还有一个迭代器。迭代器也是容器专用的，因此也可以设计成链表的内嵌类。但容器的用户需要使用迭代器访问容器中的对象，因此迭代器类被设计成公有的。链表本身只需要保存一个指向表头的指针。由于对象可以插入到容器中的任何位置，当然也可以插入到第一个位置。为了消除第一个位置插入的这个特例，我们采用了带头结点的单链表。链表和相应的迭代器的所有操作都涉及结点中的成员，为方便起见，把结点类的所有成员都设计成公有的，即把结点类定义成结构体。链表的许多操作也必须访问迭代器的成员，为此将链表类设计成迭代器类的友元。按照这个思想设计的容器如代码清单16-4所示，成员函数makeEmpty的实现如代码清单16-5所示，该类的一个使用示例如代码清单16-6所示。

代码清单16-4 用链表实现的顺序容器

```
template <class elemType>
class linkList {
private:
    struct Node {
        elemType data;
        Node *next;

        Node(const elemType &x, Node *N = NULL) { data = x; next = N;}
        Node():next(NULL) {}
        ~Node() {}
    };

    Node *head;
    void makeEmpty();

public:
    linkList() { head = new Node; }
    ~linkList() { makeEmpty(); delete head;}
    class Itr {
    private: Node *current; //用指向Node的指针表示对象位置
    public:
        Itr(Node *p) {current = p;}
        bool operator()() const { return current != NULL; }
        bool isHead() const {return current == head;}
        const elemType &operator*() const {return current->data;}
        void operator++() {current = current->next; }

        friend class linkList<elemType>;
    };

    void insert(Itr &p, const elemType &x)
```

```

    {   p.current->next = new node(x, p.current->next);
        p.current = p.current->next;
    }
    void erase(Itr &p)
    {   Node *q = p.current->next;
        if (!q) return;
        p.current->next = q->next; delete q;
    }
    Itr begin() {return Itr(head->next);}
    Itr GetHead() {return Itr(head);}
};

```

代码清单16-5 makeEmpty函数的实现

```

template <class elemType>
void linkList<elemType>::makeEmpty()
{   Node *p, *q;

    p = head->next;
    head->next = NULL; //将链表设为空链表

    while ( p != NULL) { q=p->next; delete p; p = q;} //回收链表中的所有结点
}

```

代码清单16-6 linkList类的使用

```

//文件名: 16-6.cpp
//listLink的使用示例
int main()
{   linkList<int> lq;                                //定义一个整型的容器lq
    linkList<int>::Itr itr1 = lq.GetHead(); //定义一个迭代器, 并让它指向lq的头结点

    for (int i = 0; i < 10; ++i) lq.insert(itr1, 2 * i + 1);

    cout << "用迭代器输出:\n";
    for (itr1 = lq.begin(); itr1(); ++itr1) cout << *itr1 << '\t';

    //插入0,2,4,6,8,10,12,14,16,18
    for (itr1 = lq.GetHead(), i = 0; i < 20; ++itr1, i += 2) lq.insert(itr1, i);

    cout << "插入0,2,4,6,8,10,12,14,16,18后:\n";
    for (itr1 = lq.begin(); itr1(); ++itr1) cout << *itr1 << '\t';

    //删除0,2,4,6,8,10,12,14,16,18
    for (itr1 = lq.GetHead(); itr1(); ++itr1) lq.erase(itr1);

    cout << "删除0,2,4,6,8,10,12,14,16,18后:\n";
    for (itr1 = lq.begin(); itr1(); ++itr1) cout << *itr1 << '\t';

    return 0;
}

```

代码清单16-6所示的程序首先将1, 3, 5, ..., 19依次插入容器, 然后用迭代器输出容器中的所有对象。得到的输出如下:

用迭代器输出:

```
1 3 5 7 9 11 13 15 17 19
```

然后测试用迭代器指定位置的插入。下面的一个循环是插入0, 2, 4, ...。首先, 让迭代器指向头结点, 在头结点后面插入0, 此时, 迭代器指向新插入的结点 (即0)。在for循环的表达式3, 执行++itr1, 迭代器指向结点1。然后再执行循环体, 调用插入函数在1后面插入2。执行完循环体后, 容器中的内容为

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

最后, 测试删除功能, 同样是通过一个循环。首先让迭代器指向头结点, 删除头结点后面的元素 (即0); 然后将迭代器移到结点1, 删除它后面的结点 (即2)。执行完这个循环, 容器中又只剩下1 3 5 7 9 11 13 15 17 19。

小结

容器用于存放一组对象。用户不必关心容器是如何保存这组数据的, 只需要通过容器提供的工具访问容器中的对象。有了容器这个工具, 可以进一步减少程序员在处理批量数据时的编程工作量和编程难度。迭代器相当于是一个抽象的指针, 指向容器中的对象。迭代器可以在容器中移动, 用户可以通过迭代器访问容器中的对象。

习题

程序设计题

1. 在linkList类中增加一个删除容器中一部分对象的成员函数erase, 要求该函数有两个迭代器对象的参数itr1和itr2, 删除 (itr1, itr2) 之间的所有对象。
2. 为linkList类重载==运算符。
3. 为seqList类重载赋值运算符, 使之实现两个容器的赋值。

附录

ASCII表

控制符	十进制	十六进制	字符	编码	十进制	十六进制	字符	十进制	十六进制	字符	十进制	十六进制	字符
^@	0	00		NUL	32	20		64	40	@	96	60	*
^A	1	01		SOH	33	21	!	65	41	A	97	61	a
^B	2	02		STX	34	22	..	66	42	B	98	62	b
^C	3	03		ETX	35	23	#	67	43	C	99	63	c
^D	4	04		EOT	36	24	\$	68	44	D	100	64	d
^E	5	05		ENQ	37	25	%	69	45	E	101	65	e
^F	6	06		ACK	38	26	&	70	46	F	102	66	f
^G	7	07		BEL	39	27	,	71	47	G	103	67	g
^H	8	08		BS	40	28	(72	48	H	104	68	h
^I	9	09		HT	41	29)	73	49	I	105	69	i
^J	10	0A		LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B		VT	43	2B	+	75	4B	K	107	6B	k
^L	12	0C		FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D		CR	45	2D	-	77	4D	M	109	6D	m
^N	14	0E		SO	46	2E	.	78	4E	N	110	6E	n
^O	15	0F		SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10		DLE	48	30	0	80	50	P	112	70	p
^Q	17	11		DC1	49	31	1	81	51	Q	113	71	q
^R	18	12		DC2	50	32	2	82	52	R	114	72	r
^S	19	13		DC3	51	33	3	83	53	S	115	73	s
^T	20	14		DC4	52	34	4	84	54	T	116	74	t
^U	21	15		NAK	53	35	5	85	55	U	117	75	u
^V	22	16		SYN	54	36	6	86	56	V	118	76	v
^W	23	17		ETB	55	37	7	87	57	W	119	77	w
^X	24	18		CAN	56	38	8	88	58	X	120	78	x
^Y	25	19		EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A		SUB	58	3A	:	90	5A	Z	122	7A	z
^[27	1B		ESC	59	3B	;	91	5B	[123	7B	{
^\	28	1C		FS	60	3C	<	92	5C	\	124	7C	
^]	29	1D		GS	61	3D	=	93	5D]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^-	31	1F	▼	US	63	3F	?	95	5F	-	127	7F	␣

参 考 文 献

1. Deitel H M. C++大学教程. 张引, 译. 第5版. 北京: 电子工业出版社, 2007.
2. Eckel B. C++编程思想. 英文版. 第2版. 北京: 机械工业出版社, 2002.
3. Lippman S B. C++ Primer中文版. 李师贤等, 译. 第4版. 北京: 人民邮电出版社, 2007.
4. Roberts E S. The Art and Science of C. Boston: Addison-Wesley Publishing Company, 1995.
5. 陈家骏, 郑滔. 程序设计教程. 北京: 机械工业出版社, 2004.
6. 谭浩强. C程序设计. 第2版. 北京: 清华大学出版社, 2005.
7. 吴文虎. 程序设计基础. 第2版. 北京: 清华大学出版社, 2006.