

C++ 程序设计 基础教程

李青 编著

上海大学出版社

C++程序设计 基础教程

责任编辑 王悦生

封面设计 柯国富

ISBN 978-7-5671-2481-3



9 787567 124813 >

定价：65.00元

C++程序设计基础教程

李 青 编著

上海大学出版社

• 上海 •

内 容 提 要

本书以程序案例为线索，以问题需求驱动的方式深入浅出地介绍了 C++ 语言的基本语法、程序设计规范和实用技法。书中的例题程序按照科学的原则、完整的结构、规范的格式设计，并经过多方面的测试，可供读者精读和模仿。各章配有一定的实用程序和趣味程序。大部分章后有基本语法练习题、程序设计练习题和程序设计竞赛题。

本书内容丰富，详略得当。全书共 15 章分成三个部分：第一部分（第 1 章）计算与算法基础；第二部分（第 2~8 章）面向过程程序设计；第三部分（第 9~15 章）面向对象程序设计。

本书可作为计算机等理工科专业大学生或研究生学习高级语言程序设计相关课程的教材或教学参考书，也可供从事软件开发的人员学习或使用 C++ 语言时参考。

图书在版编目 (CIP) 数据

C++ 程序设计基础教程 / 李青编著. — 上海 : 上海大学出版社, 2016.9

ISBN 978-7-5671-2481-3

I . ①C… II . ①李… III . ①C++语言—教材 IV .
①TP312.8

中国版本图书馆 CIP 数据核字(2016)第 205871 号

责任编辑 王悦生

封面设计 柯国富

技术编辑 章斐

C++ 程序设计基础教程

编著 李青

上海大学出版社发行

(上海市上大路 99 号 邮政编码 200444)

(<http://www.press.shu.edu.cn> 发行热线 021—66135112)

出版人：郭纯生

*

江苏省句容市排印厂印刷 各地新华书店经销

开本 787 × 1092 1/16 印张 26 字数 649 千字

2016 年 9 月第 1 版 2016 年 9 月第 1 次印刷

ISBN 978-7-5671-2481-3/TP · 064 定价：65.00 元

前　　言

计算机是人类集其智慧之大成的杰作，是脑力劳动机械化、自动化的成功典范。在当今及未来，计算机都是科技进步、社会发展不可或缺的得力助手。然而计算机——这个神通广大的人造精灵的一举一动全都是靠人来掌控的。

掌控计算机用的是计算机语言程序。计算机程序设计语言——尤其现代高级语言的出现和发展是计算机科学中最富于智慧的成就之一。计算机程序设计语言是人为制订的一整套功能近乎完美的算法思想表达体系和计算机行为规范准则。学会一门计算机语言便是掌握了一种掌控计算机的本领。

C++ 从 C 语言进化而来，是 C 语言的超集，它同时吸收了许多著名语言最优秀的特征。C++ 新增加的特点和机制体现了它对高级抽象的支持，它是一门适合各种应用的计算机程序设计高级语言。C++ 既支持面向过程程序设计，又支持面向对象程序设计。它既适合作为教学及训练的计算机语言（适合作为大学相关专业第一门程序设计课程的语言进行学习），又能胜任大型软件开发，特别是众人集体开发大型软件。C++ 是众多学习程序设计和从事软件开发人员的首选语言。

在计算机专业课程体系中，高级语言程序设计课程具有基础性和工具性。通过该课程的学习，读者应该达到以下三个基本目标，并为进一步学习数据结构等课程或进行应用开发做准备。

- (1) 掌握 C++ 语言的基本语法规则；
- (2) 掌握基本的程序设计技术规范；
- (3) 建立朴素的算法设计思维模式。

全书共分 15 章，第 1 章为导论，第 2~8 章为面向过程程序设计，第 9~15 章为面向对象程序设计。

书中的全部程序源代码均适用于 MinGW Developer Studio 2.05 和 Microsoft Visual C++ 6.0 等集成开发环境。这些源代码及本书的电子教案皆挂在上海大学出版社网站上 (<http://www.press.shu.edu.cn>) 供使用者下载。作者推荐读者在互联网上下载 MinGW Developer Studio 2.05 集成开发环境。它是一种可运行于 Windows 操作系统下的 C++ 应用程序集成开发环境，其中含 GCC 编译器，可在互联网上自由下载并发布它。

本书的特点

- (1) 详略得当

本教材将函数的概念分散到各章节，并在第 6 章给出函数调用及返回时的栈操作过程描述；强调多文件结构；强化引用的概念和用法，弱化了指针（尤其是多级指针），并提倡对指针的封装和屏蔽；突出对象的构造（特别是对象的拷贝构造、析构）和重载赋值运算符；强调多态性和虚析构函数的重要性。本书不介绍 STL（标准模板类库）。

书中的一些趣味程序具有一定设计技巧，值得读者钻研。本书还照顾到程序设计竞赛练习的需要，在一些章节后专设有竞赛题型。这些题目难度不高，并给出了其中一些题目的解答全过程，目的是给读者提供一些读取输入数据、处理输出结果的简便方法。

(2) 需求牵引

本书的面向过程程序设计部分以程序中最活跃的变量设计为线索，通过所谓的“评委评分”案例程序的逐步扩充和优化引出 C++ 的基本语法。即随着程序版本按“单变量→数组→堆数组→函数→链表”不断升级，教学内容从易到难逐步展开。这种“需求牵引”式方法有利于读者弄清为什么及怎样做的问题。

面向对象程序设计部分朝着“与基本数据类型看齐”的目标逐步推进，并将“评委评分”程序的功能扩充至最强（参赛选手人数、评委人数皆可随时变动，并可轻松处理多场同时进行的比赛）。

(3) 前后照应

本书的取材及在内容编排上注重循序渐进、前后照应。第 1 章（导论）的主要目的是使读者大致了解计算机的工作原理；懂得信息数字化及其标准化的必要性；理解将数据分成各种基本类型的“无奈”；了解计算机内存字节地址编址的线性和程序执行的时序性。第 2 章与第 15 章的“小学算术测验程序”相映成趣。章节（包括练习）中的前后引证，可使读者逐步加深对知识点的理解，达到融会贯通。

(4) 科学规范

本书尽量做到叙述严格、条理清楚，严格区分“定义”与“声明（描述）”的含义，希望读者关注这一用词的区别。作者不仅注重程序的宏观结构，而且非常强调精致的技术细节。本书中的示例程序皆根据科学的设计原则进行设计，特别注重函数的形式参数类型、函数的返回类型。程序中回避那些难理解的表达式^[1]，避免因编译器不同而造成的编译障碍^[2]。本书中的程序完整，并且注重了许多容易被忽视的细节。这些程序都经过了多方面的测试，以保证程序的质量。例如，在第 8 章的链表程序中，既然假设所有结点皆为堆结点以便于删除结点、释放结点操作的统一性，那么插入结点时就应该申请新的堆空间，而不能武断地将结点直接链接进链表。

笔者认为，仅仅为了解释某一语法现象而设计一些“纯粹”的语句不如将语法知识点放在一定的“语境”之中。因此，本书中的大部分示例程序均有其应用背景。这些示例程序能起到一定的示范作用，包括程序的书写格式，程序中的注解等。程序中的保留字一律排成粗体字；书中的反例程序则用加背景的方式排版，以引起读者的注意。

致教师

教师的主要任务是引导和教会学生如何学习^[3]。即教师应把握好如下关键点：引入概念、剖析难点、解答疑点、启发学生思考、指导学生归纳、组织学生研讨^[4]，尤其是引导学生养成良好的学习习惯和良好的程序设计风格（包括书写风格）。例如，在函数首部设计中，需要特别考虑引用型参数、考虑是否该加 `const` 保护；设计类时，是否应该提供

^[1] 作者认为程序员主动编写程序时应该讲究程序的可读性。在被动地阅读他人编写的程序时，若遇见类似 `a+=a+++++b;` 故弄玄虚的语句时，则应该予以抵制。

^[2] 例如：Visual C++ 与标准 C++ 关于 `for` 循环中所定义变量的生命期是不相同的。

^[3] “教会学生自学，也就是教会学生战胜自己。”——钱伟长

^[4] 由于一般的中学生尚没有很强的归纳能力，故优秀的中学教师是善于代学生做归纳者。

拷贝构造函数、析构函数和重载赋值运算符，成员函数是否应该设计成常量成员函数等。教师要告诫学生，不要使用全局变量；提醒学生注意，开发程序常常是要与他人分工合作的，程序的最终用户也往往不是程序员自己。

本书的主要内容一般可在两个学期共约 80~100 学时内讲完。对于书中比较长的程序代码可由教师提出若干关键问题，让学生分小组讨论并报告。

只需要或只准备学习 C 语言者也可以使用本教材前两个部分，它们与 C 语言几乎是相同的。不同之处（如：I/O, new, delete 等）正是 C 语言的不足之处。因此，使用本教材本质上是在先学习 C 语言，再学面向对象程序设计。

致读者

中学阶段学生学习的主要任务是通用基础知识的奠基。这一阶段通过教师主导、示范，教师为学生归纳科学思想、总结科学方法和结论。学生通过学习、解练习题达到掌握这些基础知识的要求。

显然，大学学习方法不是中学学习方法的“惯性延长”。大学学习活动是针对某专门知识与创造能力的自主建构。大学学习内容的显著特点是需要抽象性思维，分析问题和解决的问题常常需要构造性方法。在大学，学生在教师的引导下应初步了解专门领域的知识/能力结构框架（即专业导论）；初步有序地充实专门领域知识/能力结构框架；及时自主地对当前知识/能力结构进行综合评价，发现问题后有针对性地自主调整学习、优化知识/能力结构；根据新的发展，更新领域知识/能力结构并展望新的目标。这一过程是一个不断重复、螺旋式上升的过程。这一过程就是“让学生在研究中学习和成长”的本科研究型教学。

人类的学习是经验的重组，是认知结构的获得和建构过程。学习是学习者在原有经验的基础上，主动、积极地进行知识和能力构建的过程。学习发生于个体与周围环境的相互作用。

与中学学习方式不同，自主学习是大学生的主要学习方式，在自主研究中学习和成长，在团队合作中学习和成长。自我发现自己的兴趣特长、自主确定目标方向。在某具体知识学习过程中，应自主预习、自主问为什么并自主地尝试解答、自主地归纳总结、主动地与他人交流。

本书的读者可能主要是大学低年级的学生。这一阶段的学生需要主动探索适合自己的学习方法——“学习怎样学习和学习怎样思考”^[1]。在教学活动中，学是矛盾的主要方面。作者强调学生自学能力的自我培养，因为一个合格的大学生应该有能力通过自学学会那些已经成熟的、并经过他人总结的知识。精读教材（逐字逐句、反复研读）是训练自学能力的有效方法。把握程序结构、精读具有一定长度的源代码是十分必要的。

本书涉及的 C++ 语法规规定是容易接受的，并且可以通过课堂教学的方式了解这些语法机制的背景。然而学习 C++ 语言的目的决不仅为了懂得一些规定、会用“排除法”做单项选择题。学习语言的目的是为了应用。与学习外国语一样，应该将知识点放在“语境”中，与计算机进行实际的“情景对话”。

[1] “全世界在争论着这样一个问题：学校应该教什么？在我们看来，最重要的应该是两个‘科目’：学习怎样学习和学习怎样思考。这首先意味着学习你的大脑是怎样工作的，你的记忆是怎样工作的，你是怎样存储信息、找回信息、将它与其他概念相连并在你需要时马上查出新知识。”——《学习的革命》

学生仅听老师讲而不加以练习是无法提高程序设计能力的，而且还可能导致越来越听不懂。读者应该抓紧一切机会在计算机上验证所学的语法，通过调试程序不断总结经验以逐步提高自己的程序设计能力。与有些课程不同，本课程实验课前的准备工作比课堂上课前的准备工作更加重要。

学习程序设计语言需要多模仿、多上机练习。本书所设计的例题程序均有一定的代表性，读者在弄懂它们的基础上应该做到举一反三、触类旁通。

读者即使有了本教材上的所有程序源代码，作者仍然建议读者自己按语句块将一些源代码录入计算机并亲自进行调试（要按“意群”而非按行的顺序录入，即当录入到左花括号时，应立即换行输入右花括号，然后在其中插入语句块语句）。

当一个人所编写的程序代码总行数达到一定数量时，其程序设计能力就自然会有质的提高，程序设计思想就自然而然地建立起来了。知识可以通过传授获取并积累，能力只能依靠自己勤练来提高。

致谢

本书是作者在总结多年教学经验的基础上逐步形成的。在长期的教学工作中，作者与本课程组的同仁们经常互相交流经验。本书的初稿承蒙他们的指点和试用，他们提出了许多宝贵的、富有建设性的意见均被作者采纳。在此，作者对他们的帮助表示衷心的感谢！

作者十分感谢上海大学出版社的大力支持和帮助，他们为本书的策划、质量控制、出版等方面做出了卓有成效的贡献。

由于作者的水平有限，本书难免存在疏漏和缺点，敬请广大读者批评指正。

李　青

2016年8月

目 录

第一部分 计算与算法基础

第1章 导论	3
1.1 计算系统	3
1.1.1 珠算系统	3
1.1.2 电子计算机基本原理	5
1.1.3 信息数字化及其标准化	10
1.2 计算机程序设计语言概述	14
1.2.1 计算机低级语言与高级语言	14
1.2.2 高级语言程序要素	16
1.2.3 高级语言程序设计方法	16
1.3 算法基础	18
1.3.1 算法的概念	18
1.3.2 算法的表示	18
1.4 小结	20
练习1	21

第二部分 面向过程程序设计

第2章 C++ 概貌	25
2.1 基本程序设计	25
2.1.1 “算术测验”程序之一	25
2.1.2 C++ 程序基本元素	28
2.1.3 输入输出及赋值操作	31
2.2 基本程序改进	33
2.2.1 “算术测验”程序之二	33
2.2.2 C++ 基本运算	34
2.2.3 C++ 程序流程控制	38
2.3 基本程序扩展	44
2.3.1 简单函数	44
2.3.2 多文件结构	46
2.4 C++ 程序开发流程	48
2.5 C++ 应用程序集成开发环境简介	49
2.6 趣味程序——变换的字符	53
2.7 小结	53
练习2	54

第3章 数据的表示及I/O流格式控制	61
3.1 数据的表示	61
3.1.1 常量	61
3.1.2 变量	62
3.1.3 变量的引用	67
3.1.4 常量的引用	68
3.2 函数	69
3.2.1 函数的形式参数	70
3.2.2 函数的返回类型	75
3.3 运算表达式	77
3.3.1 C++运算符汇总	77
3.3.2 单目运算	79
3.3.3 二进制位运算	80
3.3.4 迭代赋值运算	80
3.3.5 抽取及插入运算	80
3.3.6 三目条件运算	81
3.3.7 逗号运算	81
3.3.8 区分作用域	81
3.4 语句	82
3.5 I/O流格式控制	82
3.6 应用举例	86
3.6.1 深入理解ASCII字符集	86
3.6.2 深入理解整型数据	87
3.6.3 输出字符图案	89
3.7 趣味程序——行走的字符串	90
3.8 小结	90
练习3	91
第4章 变量设计	97
4.1 穷举计算	97
4.1.1 “百钱买百鸡”问题	97
4.1.2 判定素数	100
4.2 迭代计算	101
4.2.1 牛顿迭代法	101
4.2.2 级数计算	102
4.2.3 最大公因数和最小公倍数	105
4.3 标志变量的设计与应用	107
4.3.1 整除问题	107
4.3.2 三角形的周长及面积	109
4.4 单变量版“评委评分”程序设计	109
4.4.1 问题描述及算法分析	109
4.4.2 程序实现	110

4.5 趣味程序——击打字母游戏.....	112
4.6 小 结.....	114
练习 4.....	114
第 5 章 数组与指针.....	118
5.1 数组.....	118
5.1.1 数组的定义	118
5.1.2 访问数组元素	119
5.1.3 多维数组.....	119
5.2 数组版“评委评分”程序设计.....	120
5.2.1 问题描述及算法分析.....	120
5.2.2 程序实现.....	121
5.3 指针	122
5.3.1 定义指针变量	123
5.3.2 指针运算.....	123
5.4 动态变量和动态数组——堆变量和堆数组.....	125
5.5 地址值在函数之间传递	127
5.5.1 传递地址值——值传递	127
5.5.2 传递指针变量——引用传递	130
5.5.3 返回地址.....	131
5.6 堆数组版“评委评分”程序设计.....	132
5.7 字符数组与 C-字符串	134
5.7.1 字符数组.....	134
5.7.2 C-字符串	135
5.7.3 字符串 I/O 操作	136
5.7.4 C-字符串处理函数.....	138
5.8 指针数组与数组指针.....	140
5.8.1 指针数组.....	140
5.8.2 数组指针.....	142
5.9 趣味程序	143
5.9.1 生日的概率问题	143
5.9.2 匹配的概率问题	144
5.9.3 模仿密码输入	146
5.10 小 结.....	147
练习 5.....	148
第 6 章 函数	157
6.1 函数概述	157
6.2 函数的调用机制	158
6.2.1 函数调用的栈操作过程	158
6.2.2 函数原型纵览	162
6.3 函数版“评委评分”程序设计	165
6.3.1 功能模块设计	165

6.3.2 功能实现——函数定义	167
6.4 递归函数	170
6.5 函数重载	172
6.6 参数带默认值的函数	173
6.7 内联函数	174
6.8 函数模板	175
6.8.1 描述函数模板	175
6.8.2 模板函数的使用	175
6.8.3 重载模板函数	177
6.9 函数应用	177
6.9.1 静态局部变量的特性	177
6.9.2 排序	179
6.9.3 定积分计算	183
6.9.4 矩阵乘积	185
6.9.5 动态二维数组	187
6.10 趣味程序——高爾頓钉板实验模拟	189
6.11 小 结	191
练习 6	191
第 7 章 程序结构	199
7.1 多文件结构	199
7.1.1 同一编译单元中的共享变量及函数	199
7.1.2 不同编译单元中的共享变量及函数	199
7.1.3 头文件	200
7.2 编译预处理指令	202
7.2.1 文件包含指令	202
7.2.2 宏定义指令	202
7.2.3 条件编译指令	203
7.3 名字空间	205
7.4 隐藏函数的定义	209
7.5 小 结	210
练习 7	210
第 8 章 链 表	211
8.1 结构体	211
8.1.1 数据组织形式描述	211
8.1.2 创建结构体对象	211
8.1.3 访问对象的成员	212
8.2 链表的概念	213
8.2.1 结点的结构	213
8.2.2 单向链表	214
8.3 链表操作	215
8.3.1 遍历	215

8.3.2 插入一个结点	218
8.3.3 删除一个结点	218
8.3.4 链表版“评委评分”程序清单	219
8.4 小 结	230
练习 8	231

第三部分 面向对象程序设计

第 9 章 类与类的对象	239
--------------------	-----

9.1 类的声明	240
9.1.1 成员的访问控制	241
9.1.2 数据成员和成员函数	241
9.2 创建类的对象	242
9.2.1 创建对象	242
9.2.2 对象的基本空间	243
9.3 对象的自我表现	243
9.3.1 this 指针常量	244
9.3.2 常量成员函数	245
9.4 封装与隐藏	246
9.4.1 屏蔽类的内部实现	246
9.4.2 隐藏类的内部实现	249
9.5 类模板与模板类	251
9.5.1 类模板声明	251
9.5.2 模板类及其对象	252
9.6 小 结	253
练习 9	253

第 10 章 构造函数及赋值运算	258
------------------------	-----

10.1 构造函数	258
10.1.1 默认构造函数	258
10.1.2 转换构造函数	259
10.1.3 构造函数的使用	261
10.2 析构函数	263
10.2.1 析构函数的概念	263
10.2.2 对象构造和析构的顺序	263
10.3 拷贝构造函数	263
10.3.1 浅拷贝构造——拷贝对象基本空间的数据成员	265
10.3.2 对象的资源空间	268
10.3.3 深拷贝构造——构造属于自己的资源空间	269
10.4 赋值运算	271
10.5 组合成员的构造	274
10.5.1 成员的构造时机	274
10.5.2 组合成员的构造——冒号语法	275
10.6 趣味程序——模拟银行打印储户存折	280

10.7 小结.....	283
练习 10.....	284
第 11 章 静态成员及友元.....	288
11.1 静态成员	288
11.1.1 静态数据成员	289
11.1.2 静态成员函数	290
11.2 友元	294
11.2.1 友元函数.....	294
11.2.2 友元类.....	299
11.3 趣味程序——自动单向链表类.....	300
11.4 小结.....	303
练习 11.....	303
第 12 章 运算符重载.....	308
12.1 运算符概述	308
12.2 重载运算符	309
12.2.1 重载双目运算符	313
12.2.2 重载单目运算符	315
12.3 自定义版字符串类——String	317
12.4 趣味程序——“评委评分”程序之类模板应用	322
12.5 小结.....	329
练习 12.....	329
第 13 章 继承与多态性.....	331
13.1 继承与派生概述	331
13.1.1 抽象与具体	331
13.1.2 组合与继承	332
13.1.3 派生类成员的访问属性	338
13.2 派生类对象的构造	338
13.2.1 派生类对象的构造与析构	338
13.2.2 派生类对象的空间	339
13.2.3 派生类对基类的赋值兼容性	340
13.3 多态性	340
13.3.1 虚函数	342
13.3.2 重载运算符享受多态性	345
13.3.3 虚析构函数	345
13.3.4 纯虚函数与抽象类	348
13.3.5 关于虚函数的说明	349
13.4 多重继承	350
13.4.1 多重继承的一般形式	350
13.4.2 虚拟继承	351
13.5 构造顺序	353
13.6 小结	355

练习 13	355
第 14 章 I/O 流	361
14.1 标准 I/O 流	361
14.1.1 操作系统关于标准 I/O 及其重新定向	361
14.1.2 常用输入流成员函数	364
14.1.3 常用输出流成员函数	366
14.2 文件 I/O 流	367
14.2.1 文本文件	368
14.2.2 二进制文件	370
14.2.3 应用举例	373
14.3 字符串 I/O 流	375
14.3.1 C 语言中的字符串生成与解析	375
14.3.2 C++ 字符串流类	376
14.4 趣味程序——探究文件字节内容	377
14.5 小 结	382
练习 14	382
第 15 章 异常处理	383
15.1 异常处理的概念	383
15.2 异常处理的方法	383
15.2.1 抛掷异常	383
15.2.2 围定及捕捉处理	385
15.3 趣味程序——正整数算术运算测验程序	387
15.4 小 结	395
练习 15	395
参考文献	396
附录A ASCII 字符集	397
附录B 常用库函数参考	398
B.1 C 字符串函数	398
B.2 转换函数	399
B.3 随机数发生器	399
B.4 数学函数	400

第一部分

计算与算法基础

第1章 导论

我国以计算为主的古代科学，在相当长的时期里处于世界领先地位。其中一个重要原因，就是我们的先祖们很早就创造了一种简便的计算系统——算筹记数及其操作。后来，又进一步发明了更为简捷的珠算，大大地推动了古代科学技术的进步。珠算系统一直沿用至今。本章立足珠算系统看现代计算机系统，其结论是现代计算机系统与古老的珠算系统是一脉相承的。

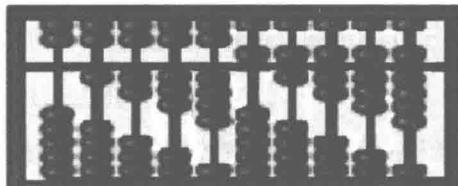
学习本章，读者通过类比珠算系统，应该初步了解电子计算机系统的基本原理和工作方式；了解信息数字化及其标准化的必要性和重要性；了解电子计算机存储器空间字节地址的线性性；了解算法的基本概念和算法执行的时序性。初步形成计算机科学与技术的时空观、方法论。

1.1 计算系统

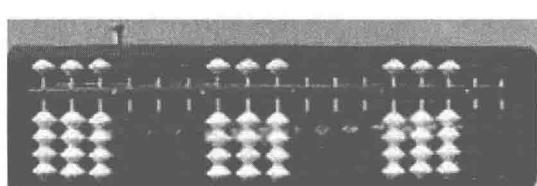
在本节中，将分析珠算系统的特点，通过类比方法“导出”电子计算机系统的基本组成，力求使读者初步理解电子计算机系统的基本特点。

1.1.1 珠算系统

珠算系统由算盘（硬件，如图 1-1 所示）、珠算口诀（软件）和人（操作者，软硬兼备）三个方面紧密结合而成。



(a) 10 档 7 珠算盘



(b) 17 档 5 珠算盘

图 1-1 算盘

1. 算珠、档及记数法

算盘由若干个档按空间位置顺序排列而成；每一档上有个数相同的算珠。算珠在其档中可以被上下移动，算珠的不同位置可用于表示不同的数码（如可表示数码 0, 1, 2, ……, 9）。

按顺序排列的档，从右至左依次编号为第 1 档、第 2 档……根据计算时的具体需要，这些档可被分成若干段（位组），每一段可存放一个数据。这样，在有限的算盘盘面上可以同时存放多个数据。算盘上的档数越多，存储数据的能力就越强。理论上，算盘的档数可以无限制地扩充。珠算系统的组成如图 1-2 所示。

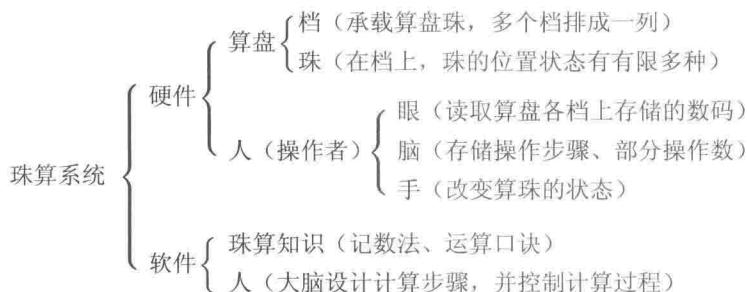


图 1-2 珠算系统的组成

2. 珠算方法及特点

在算盘上能够进行的操作（动作）只有有限多种。

- ① 选择某一档。
- ② 读取该档上算珠的状态。
- ③ 移动该档上的算珠（即改变算珠的状态）。

对于一个可由珠算系统计算的问题，其解题过程被分解成有限多个上述操作的有序动作集合。这个有穷动作序列存储在人脑中，由人脑根据其时序控制其逐步执行。显然，人脑中需要记录当前动作在整个动作序列中的位置（指令计数器或指令指针，标示动作序列的当前位置）。

算珠的移动过程所展现的就是珠算解题的计算过程。参与计算的原始数据、计算的中间结果、最终结果常常存放在同一段（位组）中，即从原始数据起“就地”逐步演变成为结果。这表明，一些操作步可能覆盖原始数据和中间结果（应用于那些不需要保留中间结果的情形）。这样可使有限的算盘空间得到充分地利用。

例如，人们学习珠算加法时，常进行的练习是 $1 + 2 + 3 + \dots + 36$ 的累加计算。其过程如下（中间结果均未保留）。

- 步1 开始；
- 步2 在算盘上选择若干档（即位组，“分配算盘空间”对应累加的结果）并清0；
- 步3 读取算盘盘面上的数字（0），运用口诀加1，将结果（1）仍存放在本位组；
- 步4 读取算盘盘面上的数字（1），运用口诀加2，将结果（3）仍存放在本位组；
- 步5 读取算盘盘面上的数字（3），运用口诀加3，将结果（6）仍存放在本位组；
- 步6 读取算盘盘面上的数字（6），运用口诀加4，将结果（10）仍存放在本位组；
- ⋮ ⋮
- 步38 读取算盘盘面上的数字（630），运用口诀加36，将结果（666）仍存放在本位组；
- 步39 读取算盘盘面上的数字（666），即运行结果；
- 步40 结束。

综上所述，算盘是数据的载体，能够存放多个数据；将计算问题分解成一系列的算盘可进行的操作是珠算的关键；珠算口诀是设计动作分解的基础和依据。

采用珠算系统解题，实质上是将复杂的计算问题转化为一系列简单操作的顺序执行。这表明一些计算过程可以机械化。但是，珠算系统中需要人的大量介入：除了“有穷的动

作序列”存放在人的大脑中以外，一些操作数、辅助量可能也存放在人的大脑中；读取算盘上的信息、算珠的移动等都需要人工实现。

“脑力劳动机械化”是人类的一种愿望。目前，许多脑力劳动已经实现了机械化、自动化。计算工具的产生和发展是脑力劳动机械化成功的典范。然而，一把算盘的存储空间容量是极其有限的，手工操作的速度是低下的。

【提出问题】拥有一把存储空间容量大、能快速操作、且自动化程度高的“算盘”是人们所追求的。

【分析问题】

(1) 这样的“算盘”应具有如下特点。

① 尽可能多的“档”数，每个“档”有多种稳定性好的可控状态（用不同的状态表示不同的数码）。

② 有一只“手”，它能左右移动，选择指定的“档”，还能改变该“档”上的状态。

(2) 用什么材料作“档”及“珠”？显然，要求这种材料具有稳定性好、可检测其状态、可控制其状态的改变等特点。例如：

① 电子元件（利用电平信号“低”“高”两种状态分别表示0,1两个数码）——电子计算机。

② 光学器件（利用光信号的几种不同的可检测、可控制的稳定状态表示数码）——光学计算机。

③ 其他器件（如：量子、生物信号等）。

1.1.2 电子计算机基本原理

本小节介绍抽象的计算机模型，主要内容有二进制算术、图灵机和冯·诺依曼存储程序原理。以便使读者理解计算机内存字节编址的线性性、程序执行的时序性。

1. 二进制算术

数字是客观存在的。通过抽象及符号表示，数字系统已经被人们认识。对于数字系统，人们最熟悉和习惯的是用十进制表示的形式。其实，人们同时也在使用数字系统的其他不同进制形式。例如，计时就采用了多种进制：60秒=1分钟，60分钟=1小时，24小时=1天，7天=1周。

事实上，对于任何大于1的整数 p ，均可以将数字按 p 进制表示及运算。在 p 进制数表示中， p 被称为“基数”。首先确定 p 个符号依次对应从0开始的前 p 个1位 p 进制数。“逢 p 进1，借1当 p ”是 p 进制数运算时的进位法则和借位法则。

同一个数在不同进制下的表现形式不同，它们之间可以转换。

将一个整数部分有 n 位的 p 进制数 $(d_{n-1}d_{n-2}\dots d_1d_0.d_{-1}d_{-2}\dots)_p$ 转换成十进制数表示，直接将其按基数展开即可。

$$(d_{n-1}d_{n-2}\dots d_1d_0.d_{-1}d_{-2}\dots)_p$$

$$= d_{n-1}p^{n-1} + d_{n-2}p^{n-2} + \dots + d_1p + d_0 + d_{-1}p^{-1} + d_{-2}p^{-2} + \dots$$

反过来，将一个十进制数转换成 p 进制数表示的方法如下。

(1) 整数部分：反复除以 p 取余，直至商为 0，依次得到 p 进制表示法的各位（从低位到高位，即第 0, 1, ……位）。

(2) 小数部分：反复将剩下的纯小数部分乘 p 取其整数部分，依次得到小数点后的第 1, 2, ……位，直到剩下的纯小数部分为 0，或者达到一定位数的精度为止。

数字的二进制表示中，基数为 $p = 2$ ，两个数码分别为 0 和 1，“逢 2 进 1，借 1 当 2”。采用二进制表示数值的优点是：易于物理实现且可靠性高；运算规则少且简单^[1]。不足之处是：与其他进制形式相比表示同一数值时需要更多的位数。

例 1.1 将十进制数 123.45 转换成二进制数。

解：(1) 整数部分。设将该十进制整数部分 123 转换成的二进制数为

$$(b_{n-1}b_{n-2}\cdots b_1b_0)_2 = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_12 + b_0 = 123$$

其中 $b_{n-1}, b_{n-2}, \dots, b_1, b_0$ 只取 0 或 1。我们的任务是确定 $b_{n-1}, b_{n-2}, \dots, b_1, b_0$ 。

将上式各端都除以 2，则其商为

$$(b_{n-1}b_{n-2}\cdots b_1)_2 = b_{n-1}2^{n-2} + b_{n-2}2^{n-3} + \cdots + b_1 = 61$$

余数为 $b_0 = 1$ 。再对所得到的商做类似的处理，可依次得到 $b_1, b_2, \dots, b_{n-2}, b_{n-1}$ 。即

2	1	2	3		余 数
2	6	1		$b_0 = 1$
2	3	0		$b_1 = 1$
2	1	5		$b_2 = 0$
2	7			$b_3 = 1$
2	3			$b_4 = 1$
2	1			$b_5 = 1$
0				$b_6 = 1$

因此， $123 = (1111011)_2$ 。验证

$$\begin{aligned}(1111011)_2 &= 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\&= 64 + 32 + 16 + 8 + 2 + 1 \\&= 123\end{aligned}$$

(2) 小数部分。设将该十进制小数部分 0.45 转换成的二进制数如下。

$$(0.b_{-1}b_{-2}b_{-3}\cdots b_{-m})_2 = b_{-1}2^{-1} + b_{-2}2^{-2} + b_{-3}2^{-3} + \cdots + b_{-m}2^{-m} = 0.45$$

上式各端都乘以 2，则产生的整数部分 $b_{-1} = 0$ ，剩下的纯小数部分如下。

$$(0.b_{-2}b_{-3}\cdots b_{-m})_2 = b_{-2}2^{-1} + b_{-3}2^{-2} + \cdots + b_{-m}2^{-(m-1)} = 0.9$$

上式各端都再乘以 2，则产生的整数部分 $b_{-2} = 1$ ，剩下的纯小数部分如下。

$$(0.b_{-3}\cdots b_{-m})_2 = b_{-3}2^{-1} + \cdots + b_{-m}2^{-(m-2)} = 0.8$$

^[1] 1 位二进制数的乘法口诀表只有 $1 \times 1 = 1$ ，而 1 位十进制数乘法有“九九乘法表”。

依此类推，得到 $0.45 \approx (0.01110011001100\cdots)_2$ ，从计算过程中可以看出 1100 是其循环节。

(3) 综合。最后得到结果是 $123.45 = (1111011.01110011001100\cdots)_2$ 。

从上述例子可以看出，一个小数位数有限的十进制数，可能表示成无穷循环的二进制小数。

我们已经了解了实数十进制、二进制表示的形式。人们习惯于使用十进制表示法，而电子计算机采用二进制数表示法，这是“人—机矛盾”之一。这一矛盾是可调和的：计算机在输入输出数据时，进行适当转换，使得呈现在人们面前的是熟悉的十进制数，这种转换由计算机系统完成。不仅如此，人们还在 10 附近找了两个“基数”——8 和 16，它们之中一个小于 10，另一个大于 10。由于 $8 = 2^3$, $16 = 2^4$ ，因此计算机系统可以非常方便地将数据按照八进制数或者十六进制数^[1]的形式输出。其转换过程无须计算，只要查表即可（如表 1-1、表 1-2 所示）。

表 1-1 二进制数与八进制数转换

3位 二进制数	1位 八进制数
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

表 1-2 二进制数与十六进制数转换

4位 二进制数	1位 十六进制数	4位 二进制数	1位 十六进制数
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

具体过程：以小数点为中心，整数部分从右向左，小数部分从左向右，每 3 位二进制数对应 1 位八进制数，或每 4 位二进制数对应 1 位十六进制数。

例如， $(1111011.01110011001100)_2 = (173.34630)_8 = (7B.7330)_{16}$ 。

2. 图灵机

图灵机（Turing Machine）为一种抽象机器，是 20 世纪 30 年代中期英国数学家图灵（A M Turing）首先提出来（由后人命名）的。这种机器由一个控制部件、一条存储带和一个读写头构成（如图 1-3 所示）。这种机器所能进行的操作有（请与算盘类比）：

- ① 左移（读写头在存储带上向左移一格）。
- ② 右移（读写头在存储带上向右移一格）。
- ③ 在存储带的某一格内写下或清除一个符号。
- ④ 条件转移。

这种机器的结构虽然简单，但在理论上它却能够模拟现代计算机的一切运算，因此，它可以被看作是现代计算机的一种抽象模型。

[1]十六进制数的 16 个数字分别是 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F（亦可用 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f）。常在数字后加后缀 H 或 h 表示十六进制数。如 $7FH=(01111111)_2=127$ 。

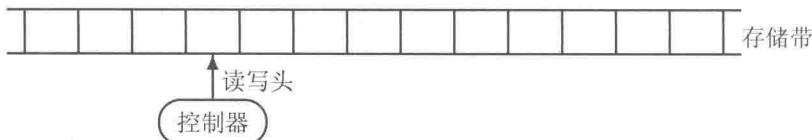


图 1-3 图灵机的构成

最简单的情形是存储带上的每一格只能记录 0 或 1。换言之，存储带是由一系列具有双稳态的元器件线性排列而成的。此时的存储带便可视为现代计算机的内存储器 (memory, 内存) 的模型。计算机内存中实质上只是一个很长的、可控制其变化的、仅由 0 与 1 组成的串，它是信息的载体。我们可以将计算机视为一把有很多档的算盘，而每一档上却仅有一粒算珠，算珠仅有“下”或“上”两种状态来表示一个二进制位 0 或 1。

上面叙述的由 0 与 1 组成的串是没有间隔的。其中，不同位置上的 0 或者 1 均可以表示不同的含义。为了有效地使用、方便地控制这条存储带，须将它按位分组、按组分段，并约定每一个 0 或 1 的含义。

位 (bit)：二进制的一个位，可表示 0 或 1 两种状态之一。

字节 (byte)：也称为位组，作为一个单位来处理的一串二进制数位。最常用的是 8 位二进制数，即 $1 \text{ byte} = 8 \text{ bit}$ ，亦即：1 字节由 8 位组成。

字节地址 (byte address)：在存储器中，以字节（位组）为单位进行编址的地址。即一个字节赋予一个地址，字节地址将按线性方式顺序编排。

计算机运行时，还可按字节地址将内存分成一些片段，供不同的程序使用。在计算机高级语言程序中，将更多地使用抽象的内存单元名称（由程序员命名，编译系统将某一部分内存空间与之联系），而不一定直接使用内存的字节地址。

例如，一台内存容量为 512MB（兆字节^[1]）的计算机内存由 $512 \times 2^{20} \times 8 = 2^{32}$ 个双稳态的元器件组成。 2^{32} 就是这种串的长度，这种串的所有不同状态共有 $2^{2^{32}}$ 种，可以存放许多不同的信息，可以刻画世间的万事万物。

3. 存储程序原理

美籍匈牙利数学家冯·诺依曼 (John von Neumann) 于 1945 年提出了存储程序的概念：程序和数据一样存储在内存中，并可以由机器来加工及修改。存储程序被认为是近代通用计算机最重要的概念之一。从此还出现了一门新的分支学科——程序设计，为构造和开发现代计算机奠定了基础。

4. 计算机系统组成

计算机系统由硬件系统和软件系统组成（如图 1-4 所示）。硬件系统主要有中央处理器 (central processing unit, CPU)、存储器（内存、外存）和输入输出设备等组成；软件系统由系统软件和应用软件构成，其中操作系统是最重要的系统软件。

在计算机系统中，输入输出设备属于计算机外部设备，如图 1-5 所示。输入是指数据从外部设备流向计算机内存的操作；输出是指数据从计算机内存流向外部设备的操作。标

^[1] 几个纯粹的数量（无量纲）记号： $1K = 2^{10} = 1\,024$ ； $1M = 2^{10}K = 2^{20} = 1\,048\,576$ ； $1G = 2^{10}M = 2^{30} = 1\,073\,741\,824$ 。

准输入设备特指键盘，标准输出设备特指显示器。外存储器（如磁盘、光盘和 U 盘）和网络设备既可以作为输入设备（如从磁盘上读取文件内容到内存），又可以作为输出设备（将内存中的数据存入磁盘文件以长期保存）。



图 1-4 计算机系统的组成

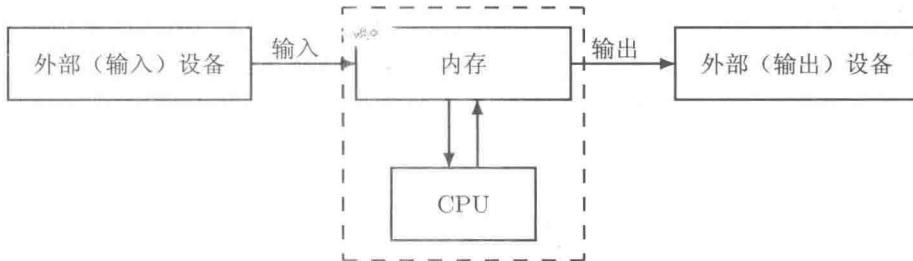


图 1-5 输入与输出

5. 计算机执行程序的基本步骤

通常，一些应用程序以文件的形式存放在外存储器（如磁盘）上。每当启动一个程序时，进行如下操作。

① 首先将该程序文件载入（load）到计算机内存：将该程序中的数据放置到数据区，机器指令存放到代码区。

② 按照指令的顺序依次执行：根据指令的操作码和地址码将数据区中的操作数送至 CPU 中运算器进行运算。同样根据指令将计算结果存回到计算机内存单元中（可以覆盖原操作数，或者存放到其他允许存放的内存单元中）。

③ 除了驻留内存的程序外，程序应该在其结束后从内存中退出。

计算机内存是数据及其操作指令的载体，而数据及其操作指令是所讨论的实际问题的计算机表达方式。程序设计的任务就是控制数据在计算机内存中的活动过程：在计算机内存中何处存放数据？何处存放机器指令？如何进行输入输出操作？如何实现基本的运算？如何将结果存放到内存的特定单元？在程序设计高级语言中，这些问题均已经被高度抽象化，用类似于英语的语句（如输入语句、输出语句、表达式语句和赋值语句等）表示。例如，在 C++ 语言中：

```

int n;           // 定义变量(意味着为 n 分配内存空间单元)
n = 1;           // 变量赋值
n = n + 1;       // 运算及赋值
  
```

其中，第1行为变量定义语句。此处定义了一个只能存放整数数值的变量，`n`为变量名。变量`n`与内存某单元联系（可以认为`n`是该单元的名字）。第2, 3行中符号“=”被称为赋值运算符，它将其右边表达式的值存入其左边变量所联系的内存单元中。换言之，赋值符号“=”左边的量获得一个新值。显然要求赋值符号左边的量必须联系内存中的某单元，并且该单元还是允许写的，这种量在C++语言中被称为左值(left value)。第3行中，变量`n`既出现在赋值符号的左边，又出现在赋值符号的右边。出现在赋值符号右边的变量`n`是右值(right value)，此时仅读取变量`n`所表示的数值。第3行中，符号“+”表示算术加法运算符，其运算优先级高于赋值运算。因此，第3行的执行过程是，将右值`n`的值与1进行加法运算（在CPU中进行），得到结果2；再将结果赋值给左值`n`（`n`的结果为2）。

前面介绍珠算计算 $1 + 2 + 3 + \dots + 36$ 的过程，用符号表示如下。

```
int s;           // 定义一个整型变量s(由系统安排某一内存单元与之关联)
s = 0;          // 累加器, 清零
s = s + 1;      // 累加计算, 覆盖存储
s = s + 2;
|
s = s + 36;    // 中间省略了若干行
                // 得到结果s为666, 最后可输出s的值
```

综合上面的两个部分，计算 $1 + 2 + 3 + \dots + 36$ 的过程可进一步表示如下。

```
int s, n;       // 定义两个整型变量(由系统安排两个内存单元分别与之关联)
s = 0;          // 累加器, 清零
n = 1;          // 计数器, 同时又是加数, 置初始值
s = s + n;      // 累加计算, 覆盖存储
n = n + 1;      // 计数器(即加数)增1
s = s + n;
n = n + 1;
|
s = s + n;      // 中间省略了若干行
                // 得到结果s为666, 最后可输出s的值
```

细心的读者可能已经发现：上述过程中的一些部分在形式上是完全相同的重复。这种重复过程有更加简捷的表达方式（参见第1.2.3小节）。须注意的是上述过程是自上而下顺序执行的，具有时序性。

1.1.3 信息数字化及其标准化

计算机内存是数据的载体。任何信息进入计算机中进行处理的第一步便是信息数字化：图形、图像、声音和文字，包括数字本身，都需要数字化，计算机指令也需要数字化。只有将各种信息变成了数字，才能被存入计算机内存，才能利用计算机进行处理。这里只讨论数值数字化和字符数字化的问题，其他信息的数字化问题超出本书的范围。

1. 数值数字化

我们知道，实数有无穷多个，无理数的小数部分是无限不循环的。显然，在有限的计算机内存中精确表示所有的实数（即使只表示一个无理数，如圆周率π）是不可能的。因此，计算机中的数系不可能是完备的，而是残缺不全的。这是一对“有限—无穷”的矛盾，此矛盾是无法调和的。缓解此矛盾的方法是选择某种精度，近似地表示一些实数。然

而，实际计算问题又不能“容忍”所有的数都是近似的。因此，又必须选择一些数值在计算机系统中精确地表示。最终的解决之道是一种折中方案：将数据分成不同的类型，并按不同的方式存储及加工处理，有些是近似表示的，有些是精确表示的。

不同的数据类型在计算机内存中所占用的单元数量多少、可表示的数值范围、具体的存放格式均有所不同（参见第2章表2-1）。程序员在使用数据时，需要考虑选择某一种数据类型，使其既能满足应用的需要又尽可能地节省内存空间。在应用中，若选用的数据类型不恰当，则可能导致数据溢出（超出可表示的范围）或者精度过低；若将某种类型的数据强行地按另一种类型的数据使用（如将较大的无符号数当成带符号数，则数据可能表示负数；或将四字节的整数理解为单精度浮点数）将导致错误。

在计算机系统中，为了便于算术运算，通常整型数值按其补码形式存放在计算机内存中。对于带符号的整数（包括 `signed char`, `signed short int` 和 `signed long int`）用其最高位表示符号（称为符号位）：最高位为1表示负数，最高位为0则表示非负数。

① 原码：二进制码。

② 补码：非负数的补码为其原码；负数的补码是其原码按位求反（0-1互反）后，在末位加1。

2. 字符数字化

(1) 字符编码

常用的英文大写字母（26个）、小写字母（26个）、阿拉伯数字0~9（10个）、标点符号和算术运算符号若干，总数超过 $64 = 2^6$ 个。因此，为这些字符编号至少需要7位二进制数。7位二进制数共有 $2^7 = 128$ 个（对应到十进制数为0~127，能区分128种不同的情形），亦即采用7位二进制编码可以将128个字符进行编码。计算机中使用的编码方案有多种，最多的是采用“美国信息交换用标准代码（American Standard Code for Information Interchange, ASCII）”（参见附录A）。

ASCII字符集采用7位编码，而一个字节有8位（能编码256个字符），因此，ASCII编码的最高位为0者对应ASCII基本字符集（详见附录A）；最高位为1者称为ASCII扩展字符集（如图1-6所示）。这样，一个字节恰好存放一个ASCII字符的编号。换言之，字符在计算机内存中按照其编码存放、运算。



图1-6 ASCII字符集

当然，若将字符'A'（ASCII码为65）作为整数时就是整数65。因此，将其作为颜

色值时，可能是某种具体的颜色；作为声音信号时，可能表示某种频率或某种音量；作为机器指令时，可能对应某种操作。即字符与整数是兼容的。亦即字符是一字节的整数。

例 1.2 给出整型数据的存放格式，即讨论其每一个二进制位的情况^[1]。

解：一字节的无符号整数（即 `unsigned char` 型）和一字节带符号整数（即 `signed char` 型）在内存中均各占一个字节（均各 8 个位，均分别能精确表示 $2^8 = 256$ 个不同的整数）。它们所精确表示的数值列表如表 1-3 所示。

表 1-3 一字节整数（即字符）在内存中的表示

一字节 8 个位情况	<code>unsigned char</code>	<code>signed char</code>
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
⋮	⋮	⋮
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
⋮	⋮	⋮
1111 1110	254	-2
1111 1111	255	-1

对于一字节的无符号整数，其表示的整数均为非负数，取值范围为 $0 \sim 2^8 - 1$ ；而一字节的带符号整数，其最高位为符号位（0 表示非负数，1 表示负数），取值范围为 $-2^7 \sim 2^7 - 1$ 。两字节、四字节以及扩展长整型的带符号或无符号整型数在内存中的存放形式是类似的，故略。

汉字编码利用了 ASCII 扩展字符集中的编码，在非中文系统下，原先的汉字编码将不会对应汉字库字模，而显示出所谓的“乱码”（实为原 ASCII 扩展字符集字符）。

《信息交换用汉字编码字符集——基本集》（即 GB 2312—80）精选了一些常用汉字，将其分成 94 个区（实际使用 87 个区），每区 94 位，共可给 $94 \times 94 = 8836$ 个汉字编码。在计算机系统中采用两字节的汉字编码实现 GB 2312—80 标准。即利用两个连续的、范围在 161~254 的数字给一个汉字编码。其中，第一个字节对应汉字的区号（160+区号），第二个字节对应汉字的位号（160+位号）。

2000 年 3 月，国家信息产业部和质量技术监督局在北京联合发布了《信息技术和信息交换用汉字编码字符集、基本集的扩充》，国家标准号为 GB 18030—2000，收录了 27 000 多个汉字，还收录了藏、蒙、维等主要少数民族的文字。该标准于 2000 年 12 月 31 日强制执行。GB 18030—2000 作为 GBK for Unicode 3.0 的更新而诞生，并且作为 GB 2312—80 的扩展，向下兼容 GBK 和 GB 2312—80 标准。

GB 18030 是我国继 GB 2312—80 和 GB 13000—93 之后最重要的汉字编码标准，是我国计算机系统必须遵循的基础性标准之一。编码空间超过 150 万个码位，为解决邮

^[1]关于浮点型数据的存放格式，请参考 IEEE 754 标准（可作为课外研讨题研究）。

政、户政、金融和地理信息系统等迫切需要的人名、地名用字问题提供了解决方案，也为汉字研究、古籍整理等领域提供了统一的信息平台基础。

(2) 字库——字模数字化

在显示器、打印机上所看到的字符，实质上是该字符的字形（字模）。同样也需要对字形进行数字化形成字库。在计算机进行文字处理时，通常只涉及字符的编码；当需要显示、打印输出时，由系统自动在字库中查询该编码字符对应的字模，并将字模显示或打印；当输出到磁盘时，通常只存储字符的编码，而不存储其字模数据（由于采用一定的编码标准，字符编码与字库中字模形成固定的对应关系，这种处理及存储方式既灵活又节省存储空间）。同一编码的字符，可有不同字体的字模与之对应。这样，根据字符的编码，再附加一定的字体、字号等属性便能演绎出多种变化。

对于两字节的汉字编码，现连续输出两个字符 `char(160+16)` 和 `char(160+1)`，若在汉字系统下，系统将在汉字字库中查找对应的汉字字模并显示一个汉字“啊”（其国标区位码为 1601，两个字节值依次为 176，161 被称为该汉字的机内码）；若在非汉字系统下，系统则分别查找扩展 ASCII 字符集中对应的两个字符的字模，显示结果为“𠀤”即所谓的“乱码”，请参照第 3.6 节。

一个 16×16 点阵的汉字字模需要 16×16 (bit) = $16 \times 16 / 8$ (byte) = 32 (byte) 字节的存储空间。汉字“汉”的编码仅需要两个字节，这两个字节机内码分别为 186，186（因为其国标区位码为 2626）。图 1-7 是根据该汉字的机内码分别从 16×16 点阵的简体汉字库文件、繁体汉字库文件中读取字模数据（各 32 字节，见每个汉字的左侧用十六进制数表示的数据），再根据这些点阵信息模拟“显示”的结果。

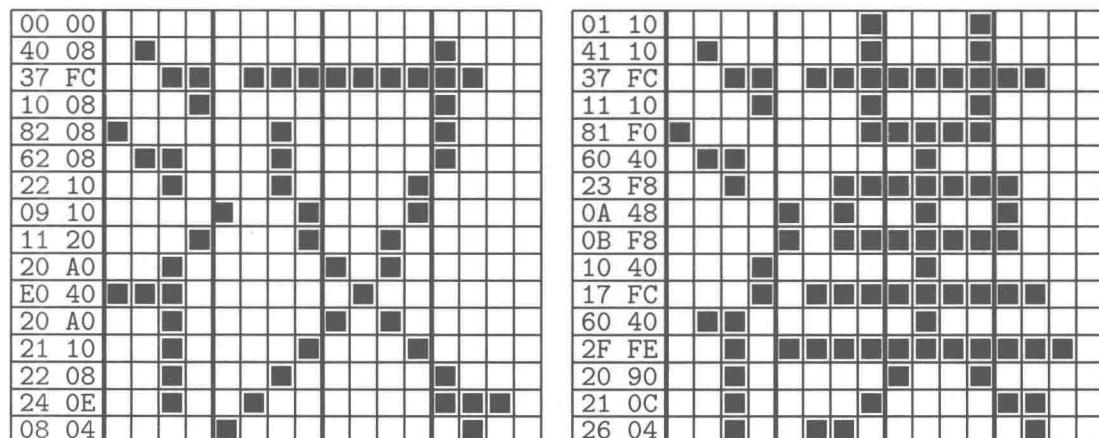


图 1-7 不同字库中的“汉”字点阵数据

3. 计算机指令数字化

计算机的所有指令也必须数字化，其数字化的结果是一系列二进制编码。计算机硬件系统不同，其机器指令集也不同。在此，我们仅指出，计算机只能按其所能识别的机器语言程序（机器指令序列）执行。不同类型计算机的可执行程序一般是不能通用的。计算机机器语言强烈地依赖于计算机硬件。

计算机高级语言（如 C++）程序是一种抽象程度十分高的程序设计语言，它接近人

类的自然语言（英语），且基本上独立于具体的计算机硬件系统和操作系统。高级语言程序可以方便地在不同的计算机系统之间移植。但需经相应的软件工具将其编译（翻译）成相应计算机所能识别的机器语言程序之后才可以运行。

4. 其他信息的数字化及其标准化

对于多媒体信息（如声音、图像等）、网络通信中的信息等，其数字化是它们进入计算机系统，利用计算机高速、程序化自动处理的第一步。数字化还需要有标准、协议来规范、约定这些数字的含义，以便各种信息的数字编码在多种系统中兼容，并对这些数字进行一致的理解、有效地利用和正确地进行交换。这些内容有专门的书籍讨论，故不在此赘述。

1.2 计算机程序设计语言概述

1.2.1 计算机低级语言与高级语言

人类的自然语言是人们在生产等社会活动中逐步形成的，自然语言的发展变迁经历了漫长的岁月。不同种族、不同地域、不同时代的语言皆有差异。有些自然语言之间的差别之大，以至于在使用不同语言进行交流时出现障碍。跨越这种障碍的方法有学习并直接使用对方的语言；或者聘请精通不同语言者充当翻译。翻译员的工作方式有两种：口译和笔译。

与人类自然语言的情况类似，不同的计算机硬件系统（如 CPU类型不同）其指令集及其数字化——指令的编码是不同的，即其机器语言是不同的。

从计算机的原型——图灵机可见，计算机能够做的事情十分简单和原始，如同人们手工操作算盘一样。信息数字化、存储程序及数据，并按指令序列操控计算机、操控计算机外部设备。这一切，从外表看来显得如此的“理所当然”！其实，计算机的高速处理能力、系统软件及应用软件的强大功能及友好的用户界面，将计算机内部巨量操作的繁忙景象隐藏到了背后。想象一下计算机内部，其内存是一本只有 0 和 1 两种符号写成的“天书”。这部“天书”的内容随时间变化而变化。但这却是一部由人写就的“天书”！在内存中，何处安排指令、何处存放数据，积累了人类的大智慧。

使用计算机最“原始”（也是最根本）的方法，是编写或设法生成能够被计算机直接执行的仅由 0 和 1 序列构成的程序——计算机机器语言程序。显然，用机器语言编写程序既难写也难修改，编写这种“天书”者非是精通具体型号机器指令者不可。另外，计算机执行程序的高速度与手工编制机器语言程序的琐碎、低效率形成巨大的反差。

为了方便编程，计算机专家们将机器指令编码（0-1串）进行命名，即用英文缩写词与机器指令一一对应，以帮助记忆，这就是汇编语言。用汇编语言编写的程序其可读性、易维护性、编写效率等方面均较机器语言程序的编写有较大的提高。汇编语言程序需要翻译成机器语言程序后才能在计算机上运行。不过，这个翻译系统是不太复杂的，它是一种简单的字面上的“笔译”。

机器语言和汇编语言程序都依赖于具体的计算机硬件系统和操作系统，它们是面向具体机器的程序设计语言，被称为计算机低级语言（实为底层语言）。用低级语言编写程序

时，必须对程序的执行过程描写得十分具体。例如，必须详尽地描述具体数据的存放地址、数据的存放与读取、算术运算、数据转移等过程。这种程序的编写效率仍然是较低的，程序仍然不够直观，但是，这种低级语言程序的执行效率往往是最高的。

屏蔽具体操作细节、面向问题表达及求解过程描述的程序设计语言——计算机高级语言应运而生。高级语言采用接近人类自然语言的、规范化了的符号系统，高度抽象地表达问题及其解题过程。与人类词汇丰富、存在多义性、存在模糊性的自然语言不同，计算机程序设计高级语言用少量的字符和单词（字符集、保留字等）按照严格的语法规则、明确的语句含义和规定的流程描述问题的求解过程。

用高级语言编写的计算机程序需要经过翻译（解释或者编译）生成机器语言程序（机器指令序列）才能在计算机上运行。

早期的 BASIC 语言是解释型的，其程序的执行过程是取一条语句、检查该语句是否有错，有错即停，无错则立即（现场）将该语句翻译成机器指令序列然后执行，再取下一条语句……重复上述过程直至程序结束。这种现场查错、现场翻译、立即执行的方式类似于“口译”。解释型程序设计语言易学，程序的执行速度慢。

现在的大多数程序设计语言是编译型的，其程序翻译和执行过程是先检查整个程序中是否有错误，有错即报告之，无错则将整个程序翻译成机器语言程序，生成可执行程序；需要时再启动所生成的可执行程序执行。需要指出的是，所生成的可执行程序以文件的形式存放在外部设备（如磁盘）上。该可执行文件运行时，将由操作系统将其装入计算机内存（将指令装入内存的代码区，操作数存放在内存的数据区），并根据指令顺序执行程序。可执行程序运行时将不再需要源程序（源程序由开发者或著作权拥有者保留以便以后的维护、移植等）。这种方式是程序员编写源程序文件；再由编译系统（一种软件）将源程序文件翻译（编译、连接）成机器语言程序的可执行文件；需要时启动该可执行文件运行程序。相当于程序员撰写“原著”；编译系统充当“笔译”；需要时，计算机直接“阅读译著”。

计算机程序设计高级语言数以百计、种类繁多，涉及科学计算、事务处理、人工智能等。经典的高级语言有 BASIC、FORTRAN、ALGOL60、COBOL、LISP（LISP 用于符号演算、公式推导、博弈等非数值处理）和 Pascal（该语言的创始人是瑞士 N Wirth 教授，N Wirth 教授是图灵奖获得者，他将其创造的计算机程序设计语言命名为 Pascal 是为了纪念 17 世纪法国著名的科学家，世界上第一台机械加法器的创造者 Blaise Pascal）。

C 语言是一个良好的结构化程序设计语言，它具有高级语言的特点和功能，还具有像汇编语言那样的低级语言的部分功能。C 语言具有丰富的数据类型，且数据的构造能力较强；C 语言本身简洁，且表达能力强；C 语言为众人集体开发大型软件、系统软件提供了有力的支撑；C 语言程序的可移植性好，执行速度快。

C++ 是 C 语言的超集，即 C 语言是 C++ 的子集。C++ 保留了 C 语言的灵活性、高效性及可移植性等；C++ 又吸收了许多著名语言的最优秀的特征。例如，C++ 从 Simula 语言中吸取了类，从 ALGOL 语言中吸取了引用，综合了 Ada 语言的类属。C++ 是一种面向对象范型的通用程序设计语言，程序的代码可重用性好。

学习计算机高级语言，不仅要掌握其语法规则，更重要的是通过多阅读（阅读范例程

序)、多仿写(模仿范例编写自己的程序)、多调试(在计算机上调试、优化程序)来掌握程序设计规范和原则,来训练算法设计思想。使读者掌握精确的C++语法规则、掌握基本的程序设计规范、建立朴素的算法设计思想,使读者达到这三位一体的要求是本书的宗旨和目标。

1.2.2 高级语言程序要素

计算机程序设计高级语言包装并屏蔽了计算机硬件、操作系统中的诸多细节,用高度抽象的文字符号描述计算过程,便于程序员将主要精力集中到问题的描述和求解中来。N Wirth教授提出如下经典公式:

$$\text{程序} = \text{算法} + \text{数据结构}$$

数据是所描述问题的载体,算法则是对数据加工方法和步骤的表达。大多数数据在程序的执行过程中是可以变化的,它们被称为变量。变量所承载的数据的逐步变化,标志着问题求解过程的时序递进。

计算机高级语言及程序均有如下要素。

- (1) 语言基本元素。字符集、保留字(关键词)集,自定义标识符。
- (2) 程序结构概貌。包括程序的首、尾,语句及其分隔,程序的书写规范和书写风格。
- (3) 数据表示及运算。基本数据类型及其运算,新数据类型的构造、使用方法。
- (4) 输入输出操作。标准输入、标准输出,文件输入、文件输出。
- (5) 算法流程控制。程序执行的起点、终点,执行顺序、分支选择、循环。
- (6) 子程序。标准库函数及自定义函数的声明、定义和调用方法。

学习一种计算机程序设计高级语言,常常亦按上述顺序进行。

1.2.3 高级语言程序设计方法

程序设计的实践中,起初人们好像是走在一 片从林之中,不断探索,却常常迷路。直到有一天,人们意识到是滥用 `goto` 导致迷茫,进而人们意识到要研究程序设计方法。程序设计方法的根本任务是解决如何组织程序的问题。下面从微观(结构化)和宏观(面向过程、面向对象)层次分别介绍有关程序设计方法的基本概念。

1. 结构化程序设计——微观技术

N Wirth提出的结构化程序设计思想对程序的微观结构控制做了本质的描述。他指出,描述任何过程的操作只需要顺序、选择、循环这三种基本控制结构(如图1-8所示)。仅用这三种结构及其相互嵌套,可以实现任何复杂的计算机程序。

结构化程序设计着眼于程序局部结构的规范化,强调每一种结构应具有“单入口、单出口”的性质,以便使一种结构嵌入其他结构时能够完全被嵌入。

循环是“唯一”使程序语句(或多个语句组成的复合语句)反复执行的结构,重复执行的部分被称为循环体。一般而言,循环体中所进行的操作是不变的,所变化的是参与重复计算的数据的值,往往对应数学中的迭代计算,或数据集合(如数组)各元素的依次处理。

例如，在 $1+2+3+\cdots+36$ 的计算过程中，累加操作是反复进行的，变化的是加数、部分和的值。前面分别用了近40行（详见第1.1.1小节）和近80行（详见第1.1.2小节）表示了这一计算过程，有些语句是完全相同的（操作数的值在逐步改变）。幸运的是，算法及程序中的循环结构能够用极少的几行语句表示这一计算过程^[1]，具体如下。

```
int s, n; // 定义两个整型变量由系统安排两个内存单元分别与之关联
s = 0; // 累加器。清零
n = 1; // 计数器，同时又是加数。置起始值
while(n<=36) // 当n小于或等于36时，执行循环体
{
    s = s + n;
    n = n + 1; // 花括号内的语句被称为循环体
} // 循环结束后，n为37（即n<=36不成立时），s为666
cout << "Sum = " << s << endl; // 输出计算结果：Sum = 666
```

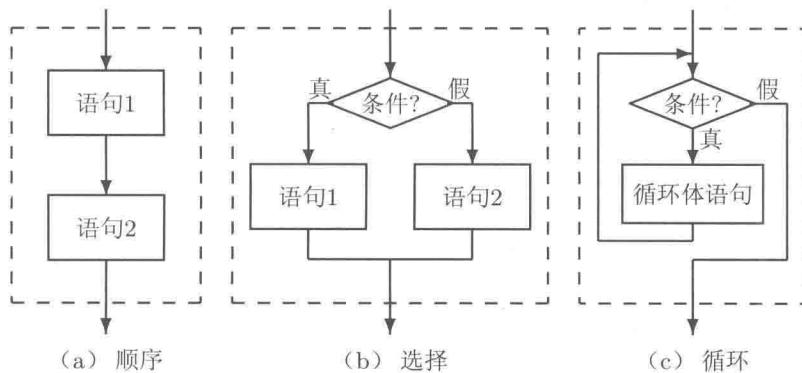


图1-8 结构化程序中的三种控制结构

循环结构是解决“手工编程低效率与计算机运算高速度”这一矛盾的武器之一。问题求解的算法设计中，发现并利用循环不变式是设计循环结构程序的关键。

2. 面向过程程序设计方法

自顶向下，逐步求精是过程化程序设计的基本思想方法，它将待解决的问题由粗到细地逐层分解（参见第4.1.1小节例4.1）。究竟将待解决的问题分解到何种地步，取决于所使用的高级语言提供的最基本的数据表达方式、最基本的数据操作方法，以及提供的标准函数库。

3. 面向对象程序设计方法

C++是既支持面向过程的程序设计，又支持面向对象的程序设计语言。本书第2~8章介绍面向过程程序设计，第9~15章介绍面向对象程序设计。

分类、分层是人们认识客观事物的一个有效方法，是对事物的聚类和抽象。正确的分类标志着人们深刻地认识了该类事物。面向对象程序设计语言和方法直接描述问题域中的类、类的对象，以及对象之间的相互关系。

例如，“运载工具”就是对一类事物的高度抽象，“运载工具”都有一些共同的属性（如自重、载重量和设计最高时速等），还有一些相同的行为（如启动、行驶、加速、倒

^[1]在本书的源程序中，字符串中的空格字符皆用符号“_”表示，以便读者精确地分析程序及程序的运行结果。

车和停止等)。“汽车”“小轿车”仍然是抽象的。这几个类中，“小轿车”是“汽车”的子类，“汽车”是“运载工具”的子类，它们呈现出一种继承关系：子类具有其父类的属性和行为；子类可能还有自己特殊的属性和行为。当谈到具体的某辆小轿车时，这便是“小轿车”这个类的一个具体对象。

面向对象程序设计方法从对客观事物的分类和分层出发，更加符合人们的认知规律、符合人们的思维方式，更加有利于描述和刻画纷繁的客观事物。面向对象程序设计将数据(即属性)以及对数据的操作方法(即行为)封装在一起，成为一个相互依存、不可分离的整体——类。类通过一些简单的接口与外界联系。对象是类的实体。这样，程序模块间的关系更加简单；界面更加清晰；程序中数据的安全性、程序模块之间的独立性、程序的移植性等都有了良好的保障。通过继承与多态性还可以大大提高程序的重用性，使所开发的大型软件产品的质量得到好的保障。

C++、Java 等是优秀的面向对象程序设计语言。面向对象程序的公式为

$$\left\{ \begin{array}{l} \text{对象} = (\text{算法} + \text{数据结构}) \\ \text{程序} = \text{对象} + \text{对象} + \dots \end{array} \right.$$

公式中的圆括号表示封装。从上述公式可见，面向对象程序设计方法需要面向过程程序设计方法为基础。不应该将它们割裂开，更不能将它们对立起来。

1.3 算法基础

算法学是计算机科学最重要的内容，有的计算机学者甚至认为，计算机科学就是算法的科学。算法的设计、验证、分析是算法学的主要内容。算法设计指创作算法的过程和研究有代表性的、好的创作策略；算法验证是指证明算法的正确性，即证明它满足欲求解问题的要求；算法分析是指对算法效率的确定，如算法所需要执行时间之长短(时间复杂度)和运行空间之多少(空间复杂度)。如果同一问题存在几个不同的算法解，算法分析的任务之一就是比较这些算法效率的高低。

算法分为数值计算方法和非数值计算方法两个方面。计算方法有直接法、迭代法等；算法设计策略有穷举、回溯、仿生和拟物等。新的算法层出不穷。

本书所涉及的算法是最基本的，其设计思想是朴素的。本书不讨论算法验证与算法分析方面的内容。

1.3.1 算法的概念

算法是解决问题方法的精确描述。比较正式的定义是，算法是一组具有明确步骤的有序动作的集合，它产生结果并在有限的时间内终止。

具体地，算法具有如下性质：①解题算法是一有穷动作序列；②动作序列仅有一个初始动作；③序列中每个动作的后继动作是确定的；④序列的终止表示问题得到解答或者问题没有解答。

1.3.2 算法的表示

求解问题的算法不依赖于具体的计算机程序设计语言。换言之，算法可以用图表或文

字表示，以描述待解问题从开始到结束的整个计算过程的时序流程。常用的算法描述方式有流程图和伪代码。流程图是算法的图形表示方式，它用处理框和箭头等表示计算的流程；伪代码是算法的文字表示方式，其中可使用自然语言（如英文及中文）、符号和数学语言等方式描述算法的各个步骤。

例 1.3 求解一元二次方程 $ax^2 + bx + c = 0$ ($a \neq 0$)。

【分析】 一元二次方程及其性质由其系数唯一决定，未知量变元的记号 x 是次要的。因此，在程序中只需要存储系数 a, b, c 的值和可能的两个解 x_1, x_2 的值（共五个变量）。这五个变量是所论问题的载体。除此以外，我们可能还需要一些中间变量。例如，在下面的算法中，我们使用了一个变量 d 存放方程的判别式，这个变量起辅助作用。我们还需要根据变量的作用，估计其可能的取值范围并为其选定适当的数据类型。

解：求解一元二次方程的算法描述如下（计算流程图如图 1-9 所示）。

步1（开始）

步2（输入） 输入方程的系数 a, b, c （双精度浮点型）；

步3（计算） $d = b^2 - 4ac$ ；

步4（判断） 若 $d \geq 0$ ，则

$$(\text{计算}) \quad x_1 = \frac{-b - \sqrt{d}}{2a}, x_2 = \frac{-b + \sqrt{d}}{2a};$$

(输出) 显示 x_1, x_2 的值；

否则

(输出) 显示“原方程无实数解。”；

步5（结束）

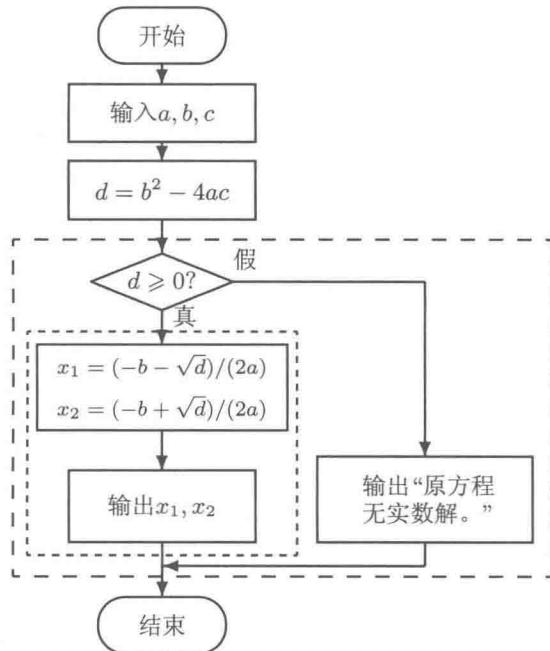


图 1-9 求解一元二次方程的计算流程图

算法描述中省略了变量的定义，使用了汉字、数学公式。至于如何进行算术四则运

算、乘方和开方，如何进行输入、输出等操作，以及如何将这些操作步骤转化为计算机机器指令等细节均不必考虑。

在高级语言（如 C++）中，算术四则运算、整数相除求商、整数相除求余、数值大小关系比较和赋值等基本操作均已经高度抽象化，并且十分接近自然语言；乘方、开方、指数、对数和三角函数等数学函数也由编译系统提供。程序员均可直接使用而不必了解其内部的实现过程。从源程序到计算机机器语言程序的翻译工作由编译程序完成。高级语言程序设计将困难留给编译器，将方便让给程序员。

【附注】 \sqrt{d} 到底是如何计算出来的。这个问题涉及数值计算方法的内容，在此，直接给出计算非负实数算术平方根的方法——牛顿迭代法。牛顿迭代公式如下：

$$\begin{cases} x_0 = 1 \\ x_n = \left(x_{n-1} + \frac{d}{x_{n-1}} \right) / 2 \quad n = 1, 2, 3, \dots \end{cases}$$

具体计算中，给定一定的精度控制，如 $\varepsilon = 10^{-8}$ ，当 $|x_n - x_{n-1}| < \varepsilon$ 时，将 x_n 当作 \sqrt{d} 的近似值。以计算 $\sqrt{2}$ ， $\sqrt{10}$ 为例，进行列表计算如下。

表 1-4 牛顿迭代法计算非负数算术平方根过程

迭代次数 n	计算 $\sqrt{2}$, x_n	计算 $\sqrt{10}$, x_n
0	1	1
1	1.5	5.5
2	1.416 666 666 666 67	3.659 090 909 090 91
3	1.414 215 686 274 51	3.196 005 081 874 65
4	1.414 213 562 374 69	3.162 455 622 803 89
5	1.414 213 562 373 09	3.162 277 665 175 68
6		3.162 277 660 168 38
标准函数计算结果	1.414 213 562 373 1	3.162 277 660 168 38

由表 1-4 可见，计算非负实数算术平方根的牛顿迭代法确实非常神奇高效，它仅仅用少量的几次算术四则运算，便可以获得精度高的计算结果。计算 $\sqrt{2}$ 仅用 5 次迭代（共 15 次算术运算），计算 $\sqrt{10}$ 仅用 6 次迭代（共 18 次算术运算）便达到较高的精度。

上述公式只是牛顿迭代公式的一个具体应用，牛顿迭代公式的一般理论还有许多用途。建立牛顿迭代公式所用到的数学原理涉及函数的导数等微积分知识（参见任何一本《计算方法》或《数值分析》的教材）。

上面的例子中，解一元二次方程采用的是直接法（直接运用求根公式），计算非负实数算术平方根的算法被称为迭代法（通过建立迭代公式进行迭代计算）。

1.4 小结

计算机（computer）是一种具有存储能力、能进行高速运算（操作）的自动电子装置。

计算机系统由紧密相关的硬件和软件组成。硬件系统由运算器、控制器、存储器和输

入输出设备组成。中央处理器 (CPU) 中包含了运算器和控制器两个重要部件，是计算机的“心脏”。

可以粗略地理解为 CPU 是执行计算的场所，操作数来自计算机内存，计算结果又回到内存中。

计算机的内存空间以字节为单位按线性方式编地址——字节地址。计算机程序按时序逐步执行。计算机系统与珠算系统的原理一脉相承。

数值数字化、字符数字化（包括字模数字化）、多媒体信息数字化和计算机机器指令数字化等，是利用计算机进行信息处理的第一步。对具体信息如何进行数字化除了涉及数学、物理等学科知识外，还需要建立一些协议、标准，以便在信息处理、存储、传输过程中进行正确的交换和识别。在计算机中，所有的信息均被称为数据。

电子计算机采用二进制表示各种数字化了的信息。按位看待计算机内存，它是仅仅由 0 与 1 组成的序列。对于某个固定的位，其所有变化归根到底就是将 0 变成 0 或 1，1 变成 0 或 1。因此，计算机是简单的。而计算机应用已经渗透到各个领域，能处理许多极其复杂的事情。我们感叹：复杂是重复简单的结果。

算法是计算机科学的灵魂。算法设计是一项最能体现人类智力的活动。进行算法设计常常需要数学、物理等方面的知识。然而，算法设计的基本思想是朴素的、自然的，除了需要细致周密的逻辑思维以外，常常还需要大智慧。

如果说算法是计算机科学的灵魂，那么，变量就是算法及程序中的“小精灵”。这些“小精灵”中，有的代表实际的物理量；有的则表示计算过程中的某些特殊状态、标志，起着辅助作用。这些“小精灵”们可以“招之即来、挥之即去”，任凭程序员差遣。在实际问题中，到底设置哪些“小精灵”，并控制好这些“小精灵”需要一定的技巧。“实践—实践—再实践”是练就技巧的重要途径，也是唯一途径。

练习 1

- (1) 将十进制数 78.5 分别转换成二进制、八进制和十六进制形式。
- (2) 写出直接将十进制数化成十六进制数的方法。
- (3) 设 $x = 1\ 234\ 567\ 890, y = 987\ 654\ 321$ ，分析如下计算 $x - y$ 方法的特点。

$$\begin{aligned}x - y &= (x - 1\ 000\ 000\ 000) + (999\ 999\ 999 - y + 1) \\&= 234\ 567\ 890 + 12\ 345\ 679 \\&= 246\ 913\ 569\end{aligned}$$

- (4) 设 $x = (10011000)_2, y = (01110101)_2$
 - ① 仿照上题设计一种方法计算 $x - y$ ，并按所设计的方法计算之。
 - ② 写出 $-y$ 的补码。
- (5) 在 ASCII 字符集的可打印字符中，空格 (' ')、数码 ('0'~'9')、大写字母 ('A'~'Z')、小写字母 ('a'~'z') 编码大小的基本顺序是什么？
- (6) 在 ASCII 字符集中，英文大写字母与其对应的小写字母的编号相差多少？内存中若存有字符 'M'，如何将其转换成对应的小写字母 'm'？（提示：在内存中字符以其编码——整数的格式存放。）

(7) 给定一个正整数 n , 设计一个判断 n 是否为素数的算法。若 n 是素数, 则显示“是”, 否则显示“否”。写出算法步骤并画出流程图。

(8) 给定一个数组, 数组中的元素为一些同学的考试成绩, 要求计算他们的名次, 并将计算的结果存入另一个数组中(注意其中有并列名次; 不必对原始数据进行排序)。例如: 若有 8 位同学的考试成绩存放在数组 `double score[8]={85.5, 95, 85.5, 78, 78, 78, 80, 70};` 中, 需将他们对应的名次存放在数组 `int rank[8]` 中, 结果应该是 `{2, 1, 2, 5, 5, 5, 4, 8}`。试写出算法步骤并画出流程图。

第二部分

面向过程程序设计

第2章 C++ 概貌

一个人走路，用不着把路上所有的石头搬掉再走。有时候要绕过去，等你走远了回头看，看到的只是走过的路，有些石头早已看不见了。

——钱伟长

本章中，通过一个具体例子的逐步改进和扩展介绍 C++ 程序的概貌，使读者了解 C++ 语言的基本要素、C++ 程序的基本结构；掌握 C++ 程序的书写规范，并有意识地培养良好的程序书写习惯；掌握 C++ 程序的开发步骤，并学会一种 C++ 语言程序集成开发环境的使用及初步的程序调试方法。

2.1 基本程序设计

2.1.1 “算术测验”程序之一

【“算术测验”问题】以“求给定的两个整数之和”为题，首先在显示器上输出一道具体算式，然后要求应试者从键盘输入一个数作为答案，最后由程序给出正确与否的评判。

【分析】我们知道“程序=算法+数据结构”，因此，我们从数据设置、动作序列设计两个方面开始。

数据：本问题涉及两个加数（ x ， y ）及其和（ z ），共三个整数。

算法：（有穷的动作序列）

- (1) 定义三个整型变量 x ， y ， z 。
- (2) 确定两个加数 x 及 y 的值。
- (3) 输出题目（在显示器上显示）。
- (4) 输入答案（从键盘输入一个整数，存放到变量 z 中）。
- (5) 判断 $x + y$ 是否等于 z

若是则输出“正确”；否则输出“错误”。

根据上述分析，编写如下源程序（见下页）。

程序运行时^[1]，将在显示器上显示“ $3 + 5 =$ ”及一个闪烁的光标，等待操作者从键盘输入一个整数。输入 8 并回车，则程序输出“正确！”，随后整个程序结束。再次启动程序运行，从键盘输入一个不为 8 的其他整数（如输入 7）并回车，则程序输出“错误。”，随后整个程序结束。

连同空行在内的源程序共有 22 行，其中的行号不是程序中的部分，它是为了便于解释源程序而添加上去的，程序员在编辑 C++ 源程序时不要输入行号。

[1]指将该源程序编译、连接生成可执行文件后，运行其可执行文件。

源代码 2.1 算术测验程序之一

```

1 // test1.cpp      源程序文件名
2 #include <iostream>
3 using namespace std;
4
5 int main()           // 主函数首部
6 {
7     int x, y, z;      // (1) 定义三个整型变量 x, y, z
8
9     x = 3;  y = 5;      // (2) 确定两个加数 x 及 y 的值
10
11    cout << x << " + "
12        << y << " = ";
13
14    cin >> z;          // (4) 输入答案, 从键盘输入一个
15                           // 整数, 存放到变量 z 中
16    if(x+y == z)       // (5) 判断 x+y 是否等于 z
17        cout << "正确!" << endl; // 若是则输出“正确!”字样
18    else
19        cout << "错误。" << endl; // 否则输出“错误。”字样
20
21    return 0;           // 退出程序, 返回到操作系统
22 }

```

从程序可见, C++ 程序由英文字母、数字、运算符号、标点符号、空格和换行字符等组成。由字符组成具有一定含义的单词(被称为**标识符**)。有些标识符的含义是固定的(被称为**保留字**或**关键词**。本书的所有程序中, 我们将保留字按粗体字排版, 便于辨认)。由标识符、数字、运算符号及分号组成语句。由语句的顺序排列及其他成分组成一个完整的程序。

C++ 程序按照“缩进”方式编排: 将语法上属于不同层次的部分从不同的列开始对齐, 以使程序的层次结构清晰。使用一个或多个空格字符、制表符, 或者换行符(连续两个换行符, 则出现一个空行)将程序中的一些部分隔开, 便于阅读。可以将多个语句写在同一行(第 9 行上有两条语句, 语句之间用分号分隔), 也可以将一条语句写在多行中(如第 11、12 行为一条语句)。

以符号“//”引出的, 直至该行结束的部分称为程序的**注释**。编译器在将源程序翻译成机器语言程序时将忽略所有的程序注释。因此, 注释的具体内容不影响程序中的任何语句。注释部分的作用是帮助阅读者理解程序, 编写适当的注释将有助于程序的维护。程序注释的另一个用途是在调试程序时——可在某些语句上添加(或去掉)注释符号, 使该部分语句不作为(或成为)程序中的执行语句。另一种编写注释的符号是 /*注释部分*/。这种注释可以将多行文字作为程序的注释内容。

上述程序中的第 1 行是一个注释行。本书中的绝大多数程序都将其文件名作为注释内容写在第 1 行上。程序中的第 2、3 行只需照搬, 暂不必深究。

一个 C++ 程序必须有一个, 且只能有一个 `main` 函数, 该函数被称为**主函数**。标识函数的一对圆括号“()”是不可缺少的。函数体用一对花括号包围起来(此对花括号也是不可缺少的)。标准 C++ 要求主函数的返回类型为 `int` 型。上例中, 第 5~22 行为主函数。其中第 5 行为主函数首部, `int` 表示主函数的返回类型; 第 6~22 行为主函数的函数

体。函数体语句大致上可以分成如下三个部分。

(1) 变量定义。数据是所论问题的数字化表述形式。计算机内存是数据的载体，可以承载（即存放）大量的数据。将数据存放到计算机内存的第一步是定义变量。在 C++ 程序中，**定义一个变量就意味着为该变量分配内存空间**，使该内存空间成为存放一个数据的场所，该空间的大小取决于变量的数据类型，变量名则成为该内存空间的标识，程序中通过变量名使用内存空间。**变量必须先定义后使用**。

程序中第 7 行为变量定义语句，该语句定义了三个变量， x , y , z 是变量的名字（由程序员自行命名），它们分别联系到计算机内存中三个不同的单元。

(2) 执行语句。存放数据的内存空间准备好后，便可以利用这些内存空间将数据存入其中（称为写），或将其中所存放的数据取出（称为读）。读、写操作统称为访问。随着变量的访问，其中的数据将由起始值经一系列演变（中间结果）最终得到问题的解，即数据的逐步变化标志着问题的求解历程。数据的演变是通过执行语句实现的。

程序中第 9~19 行为执行语句。第 9 行和第 14 行分别对变量 x , y , z 进行了写的访问操作。第 9 行两条语句分别将整数 3 及 5 存入变量 x 及变量 y 联系的内存空间中（该语句称为**变量赋值语句**）。程序执行到第 14 行时，将暂停并等待用户从键盘输入一个整数，将用户输入的整数存入变量 z 所联系的内存空间中（该语句称为**输入语句**）。程序中 `cin` 指键盘，`>>` 为输入操作符。赋值语句和输入语句均能将一个新数值写入指定的变量（即变量联系的内存空间）中，覆盖其原有的数值。上述程序中变量 x , y , z 原有的数值是不可预知的。

第 11、12 和 16 行中涉及读取相应变量的当前值的访问操作，并进行运算加工。执行这些语句不会修改这些变量的值。

程序中 `cout` 指显示器，`<<` 为输出操作符，该操作符将其右边的数据输出到显示器上，并且可以连续输出多种数据（称为**输出语句**）。双引号包围起来的称为字符串常量，字符串常量在输出时会“原样照印”，但不输出双引号本身。`endl` 表示换行。第 11、12 行为输出语句，输出题目作为提示信息（不换行），第 17 和 19 行的输出语句为输出汉字字符串后换行，作为反馈给用户的结果。

第 16~19 行为一条语句（`if` 语句），它将表达式 $x+y == z$ （即 x 加 y 的结果与 z 的值进行是否相等的比较）作为分支的判断条件，选择进一步执行第 17 行或者第 19 行。

(3) 返回语句。主函数中的返回语句将引起整个程序结束，并返回到操作系统。标准 C++ 要求主函数的返回类型为 `int`。即用一个整数作为程序执行结束后报告给操作系统的信息（第 21 行），该语句中的整数 0 对应第 5 行函数首部的函数返回类型 `int`。

建议读者录入源程序时按如下方式先编写一个完整的框架。

```
1 // myprog.cpp          源程序文件名
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     return 0;
8 }
```

然后在“`return 0;`”语句前插入程序的其他语句，如变量定义语句、执行语句。

2.1.2 C++ 程序基本元素

汉语的基本元素是汉字，由汉字组成词语（既有约定俗成的成语，也可以为一些事物自行命名）；由词语组成语句；由语句组成段落；由段落组成节；由节组成章；由章组成一部完整的作品。英语中的情况也是类似的。

在文学创作中，对整部作品的谋篇布局，章节标题“画龙点睛”般的命名；各种人物角色的名字设计；人物的特点、事件、命运变化和对他人的影响等，皆需要精心构思、设计出框架。接下来就是按照语言的语法遣词造句，按照事情发展的时序写作。

在C++语言中，从最基本的元素到程序，大致上可以与自然语言对应地表述为“字符集 → 标识符（保留字、标准名字和自定义名字）→ 语句（表达式语句、流程控制语句）→ 函数（标准函数、自定义函数）→ 类（数据及其操作的封装体）→ 源程序文件（编译单元）→ C++程序（多文件结构）”。

与文学创作一样，程序各部分的层次结构、功能的划分，各种函数的命名，各种变量的命名、存储类型、数据类型和变量值的变化等，皆需要精心构思、画出流程框图。接下来按照 C++ 的语法，按照算法的执行顺序编写程序。同小学生开始学写作文一样，“记流水账”式的结构是朴素而有效的。

与自然语言相比，计算机程序设计语言中字词少而精——数量少、语义精确（没有二义性）。在这个意义上，计算机程序设计语言是容易学习和掌握的，主要靠多思勤练。善于把握整体结构、精于刻画细节者为创作高手。

1. 字符集

规定 2.1 (C++字符集) C++字符集包含如下字符。

小写字母 (26个) a b c d e f g h i j k l m n o p q r s t u v w x y z

大写字母 (26个) A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

数 字 (10个) 0 1 2 3 4 5 6 7 8 9

其他符号 (若干) 空格 制表符 换行符 + - * / % = , . ; : ? ! | \ # & ~ ^ _ < > [] { }

C++ 语言对英文字母的大、小写是敏感的，即大写字母与小写字母是严格区分的。因此，`num`，`Num`，`NUM` 是三个不同的标识符。

说明：作为字符型数据（字符常量的值、字符变量的值和字符串的内容），可取 ASCII 表中的所有字符（参见第 3.1.1 小节）。例如：

<code>const char c = '@';</code>	// 等价于 <code>const char c = 64;</code>
<code>char str[9] = "程序设计";</code>	// 包含汉字的字符串

2. 标识符

由 C++ 字符集中一定字符组成的单词被称为标识符。有些标识符的含义是固定的，有些标识符已经由系统命名，有些标识符由程序员自行命名。

规定 2.2 (C++ 保留字) 保留字亦称为关键词，它是系统预先定义的标识符，有其固定的特殊含义，程序员不能挪作他用。C++ 的保留字如下。

auto	bool	break	case	catch
char	class	const	const_cast	continue
default	delete	do	double	dynamic_cast
else	enum	explicit	extern	false
float	for	friend	goto	if
inline	int	long	mutable	namespace
new	operator	private	protected	public
register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch
template	this	throw	true	try
typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	while

不同的 C++ 编译器还可能添加一些保留字。这些保留字的意义和用法将在以后逐步介绍，其中排成斜体字的保留字本书将不做介绍。

有一些标识符已经由系统命名（如 `main`, `cin`, `cout`, `cerr`, `endl`, `sqrt`, `std`, `NULL` 和 `RAND_MAX` 等）。尽管我们可以将其中的一些名字挪作他用，但在程序中最好还是使用它们已经存在的含义。

规定 2.3 (自定义标识符的命名规则) 程序员在给自定义变量、函数等命名时，须遵守如下规则：

- (1) 不能与保留字相同。
- (2) 必须以字母或者下画线开头，后面可跟字母、下画线或数字。
- (3) 名字的长度不宜过长（编译器一般对标识符名字的长度有限制）。

例如，`New`、`NEW`、`newRecord`、`_new`、`new_recored`、`_`、`_new1`、`first` 和 `First` 均是合法的自定义标识符的名字。不能用保留字 `new` 做变量名。`$x`、`1st`、`2nd`、`#123` 和 `new-record`、`TEST.H` 都不是合法的名字。

3. 基本数据类型

由第 1 章关于数字化的讨论可知，计算机系统中的数系是残缺不全的。为了缓解“有限的内存容量与无限的数字空间”之间的矛盾，以满足一般应用的需求，C++ 语言提供了如下基本数据类型。不同数据类型的数据在计算机内存中所占用空间单元数量（字节数）及存放格式是不同的。程序员在实际应用中，应该选择适当的类型来刻画实际数据，既要满足实际问题的需要，又能尽可能地节省内存空间。

规定 2.4 (C++ 基本数据类型) C++ 基本数据类型的名称、数据尺寸及其取值范围如表 2-1 所示。

表 2-1 C++ 基本数据类型

类 型	名 称	占 用 字 节	取 值 范 围	说 明
布尔型	bool	1	false, true	分别与 0, 1 兼容
无符号字符型	unsigned char	1	0~255	$0 \sim (2^8 - 1)$
带符号字符型	[signed] char	1	-128~127	$-2^7 \sim (2^7 - 1)$
无符号短整型	unsigned short [int]	2	0~65 535	$0 \sim (2^{16} - 1)$
带符号短整型	[signed] short [int]	2	-32 768~32 767	$-2^{15} \sim (2^{15} - 1)$
无符号整型	unsigned int	4	0~4 294 967 295	$0 \sim (2^{32} - 1)$
带符号整型	[signed] int	4	-2 147 483 648~2 147 483 647	$-2^{31} \sim (2^{31} - 1)$
无符号长整型	unsigned long [int]	4	0~4 294 967 295	$0 \sim (2^{32} - 1)$
带符号长整型	[signed] long [int]	4	-2 147 483 648~2 147 483 647	$-2^{31} \sim (2^{31} - 1)$
单精度浮点型	float	4	$-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$	7 个十进制有效位
双精度浮点型	double	8	$-1.7 \times 10^{308} \sim 1.7 \times 10^{308}$	15 个十进制有效位
扩展的浮点型	long double	12	$-1.2 \times 10^{4932} \sim 1.2 \times 10^{4932}$	19 个十进制有效位

表 2-1 是 GCC 对 C++ 标准的实现^[1]。不同的编译系统可能有不同的实现，如 Visual C++ 中 long double 与 double 相同，占用 8 字节，有的编译系统中 long double 为 10 字节（即 80 位）。在 GCC 中，还可以使用无符号扩展长整型 unsigned long long，它占用 8 字节，取值范围是 $0 \sim 18\,446\,744\,073\,709\,551\,615$ （即 $0 \sim 2^{64} - 1$ ）；以及带符号扩展长整型 signed long long，它占用 8 字节，取值范围是 $-9\,223\,372\,036\,854\,775\,808 \sim 9\,223\,372\,036\,854\,775\,807$ （即 $-2^{63} \sim 2^{63} - 1$ ）。

在表 2-1 中，方括号本身不是数据类型名称的一部分，它表示其所包围的部分是可以省略的。例如，long 表示 signed long int。另外，程序中常用的 int 型，在 Visual C++ 及 MinGW GCC 等编译系统中相当于 signed long int 型。

4. 变量定义

定义变量就意味着给该变量分配内存空间。定义变量的格式为

[存储类型] 数据类型 变量名；

或者

[存储类型] 数据类型 变量名 = 初始值；

还可以用一条语句定义多个数据类型相同的变量，并且可以初始化它们中的部分或者全部。关于变量的存储类型将在第 3 章介绍。

例如：

```
int a, b=2, c=b; // 变量 b 被初始化为 2, c 用 b 的值进行初始化
double x, y, z=0; // 变量 z 被初始化为 0.0
```

^[1] 关于基本数据类型的存放格式，标准 C++ 并未做严格的规定。仅规定
 $\text{sizeof(char)} < \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$
 和
 $\text{sizeof(float)} < \text{sizeof(double)}$
 作为 C++ 的入门教材，本书并不在类似的这些细节上做太多的介绍。

程序中定义的所有变量均有各自独立的内存单元，它们之间是相互独立的。多个变量参与同一个运算操作并不能建立变量之间的长期联系，更不存在变量之间的自动联动机制。

2.1.3 输入输出及赋值操作

规定 2.5 (C++ 运算符) 运算符亦称为操作符，是表示数据加工的符号。对于基本数据类型，这些运算符有其固定的含义。

规定 2.6 (运算表达式) 运算表达式（简称为表达式）是由操作符和操作数（常量、变量或函数调用）组成的序列，用来说明一个计算过程。运算表达式后加分号便成为一条语句，称为表达式语句。

C++ 提供了丰富的运算符，所有运算符被分成 18 个优先级（第 1 级最高，第 18 级最低）（参见第 3.3.1 小节）。在一个表达式中，优先级高的运算先进行，优先级低者后进行；同级运算则按其结合方向（从左至右或从右至左）依次进行。

根据参与运算的操作数的个数，C++ 运算分为单目运算（只需要一个操作数）、双目运算（需要两个操作数）和三目运算（需要三个操作数）。

本节先介绍一些简单的运算符及运算表达式，其他的运算符将在后续的章节中逐步予以介绍。

1. 赋值运算表达式及赋值语句

规定 2.7 (左值与右值) 左值是指具有存储单元的量，并且该单元是可写的。右值是指其值可以被读取的量。

左值的本质是变量。变量之所以被称为左值是因为它可以出现在赋值运算符“=”的左边。输入操作、自增操作、自减操作能对左值进行，也只能对左值进行。这些操作能使变量获得一个新数值，以覆盖旧值。

右值是指可以出现在赋值运算符右边的值。常量、变量和函数调用表达式均可做右值。左值可以用做右值，作为右值的量参与表达式运算时，只取其值不改其值。

C++ 语言使用一个等号“=”作为赋值运算符。赋值表达式的一般形式为

变量名 = 右值表达式

赋值表达式执行时，将其中“右值表达式”的值存入“变量名”对应的变量所联系的内存单元中，该变量的原值将被覆盖，是对变量的写访问操作。整个赋值表达式的值为其左值。

赋值表达式后加分号则成为一个语句，称其为赋值语句。

赋值表达式；

例如，语句

```
int a, b, c;
double x, y, z;
a = b = c = 0;      // 即 a=(b=(c=0)); 赋值运算的结合方向从右至左
x = (y = 1) + 2;  // 相当于 y = 1; x = y + 2;
```

```

z = sqrt(x);      // 将 x 的算术平方根（函数调用）赋值给变量 z
x = 5.5;         // x 与 z 独立。仅修改 x 的值，不影响 z

```

表达式中有多个运算符时，根据其优先级和结合方向按步骤依次进行。其中，表达式 `(y = 1)` 中的 `y` 为左值，而表达式 `(y = 1) + 2` 为右值。表达式 `sqrt(x)` 中的 `x` 为右值（仅仅使用 `x` 的值，而不会改变 `x` 的值）。

2. 输入输出语句

输入输出 (input/output, I/O) 操作涉及计算机外部设备与计算机内存之间交换数据。程序中涉及 I/O 操作时，需要先用包含文件指令 (`#include`) 将 I/O 流头文件 (`iostream`) 的内容插入到程序中。在该头文件 (header file) 中，声明了一些与外部设备关联的名字及其操作符 (如表 2-2 所示)，并将这些名字及操作符放在标准 (standard) 名字空间 `std` 中。因此，程序中涉及 I/O 操作时，需要如下两行 (参见上例程序第 2、3 行，初学者只需模仿照搬即可)。

```

#include <iostream>
using namespace std;

```

表 2-2 常用的 I/O 名字及操作符

名 字	意 义	说 明
<code>cout</code>	控制台输出 console output	显示器，可重新定向到文件
<code>cin</code>	控制台输入 console input	键盘，可重新定向到文件
<code><<</code>	插入运算符 (输出操作符)	可连续输出
<code>>></code>	抽取运算符 (输入操作符)	可连续输入
<code>flush</code>	刷新输出流缓冲区	将输出流缓冲区中尚未显示的内容立即显示并清空
<code>endl</code>	输出换行并刷新输出流缓冲区	相当于输出 ' <code>\n</code> ' 及 <code>flush</code>

“`<<`”被称为插入运算符，它将其右边的表达式的值插入到输出流之中，即将表达式的值输出到其左边的设备 (如 `cout`) 上。“`>>`”被称为抽取运算符，它通过输入设备 (如 `cin`) 获取数据，并将数据存入其右边的变量中。

(1) 插入运算表达式的一般格式为

```
cout << 表达式
```

上述插入运算表达式除输出其中表达式的值外，其结果仍为 `cout`，因此可实现连续插入。插入运算表达式后加分号，便成为输出语句。一般地，在控制台标准输出设备 (即显示器) 上，每行能输出 80 个字符 (一个汉字占两个字符宽)。输出时，一般只能从左至右 (除输出退格字符 '`\b`' 其 ASCII 编码为 8，回车字符 '`\r`' 其 ASCII 编码为 13 外)、从上至下依次输出。

执行输出语句，实际上先将所输出的信息存放到内存某个区域 (称为输出流缓冲区)。当输出流缓冲区充满时，由系统进行刷新。刷新的实际效果是将输出流缓冲区中尚未呈现在显示器 (外部设备) 上的内容立即呈现在显示器上，随后清空该缓冲区。若输出流缓冲区未满可使用语句：

```
cout << flush;
```

进行强制刷新，使之及时在显示器上呈现输出语句执行的结果。

`endl`是（end of line）的缩写，表示输出换行字符'`\n'（ASCII编码为10）并刷新输出流缓冲区。若要求输出的信息立即在显示器上显示，而又不需要换行，可用如下语句：

```
cout << x << " " << y << " " << flush;
```

常需要在输出的数据项中插入一些空格字符（' '）、制表符（'\t'）或换行符（'\n'）以分隔各数据项。

（2）抽取运算表达式的一般格式为

```
cin >> 变量名
```

上述抽取运算表达式除在输入流缓冲区^[1]中抽取数据存放到指定的变量中外，其结果仍为`cin`，因此可实现连续抽取。抽取运算表达式后加分号，便成为输入语句。可以通过输入语句修改变量的值（覆盖其原值），是对变量的写访问操作。

执行抽取操作时，系统根据一个或多个空白字符（' '、'\t' 或 '\n'）分隔各项数据。若抽取的数据不合法（例如，需要抽取一个数字时，却抽取到字符或字符串；或者按`Ctrl+Z`键），则表达式（`cin >> 变量名`）的值为`NULL`。

2.2 基本程序改进

2.2.1 “算术测验”程序之二

前面的基本程序只能测验用户一道固定的算术题。下面，我们从两个方面对该基本程序进行改进。

- ① 测试题目是随机变化的（两个加数均不超过 20）。
- ② 提供 10 道测验题（每题 10 分），统计并输出测验成绩。

分析如下。

（1）在基本程序`test1.cpp`中，第 9~19 行“确定两个加数、显示测验题目、接收答案输入、判断答案是否正确”是其主要功能实现部分。也是如下改进程序中需要重复执行的部分，即将其作为循环体执行 10 次（参见程序`test2.cpp`第 14~19 行）。

（2）C++ 提供了一个产生伪随机数的标准函数`rand`，它产生 0~`RAND_MAX`^[2]之间的一个伪随机整数。表达式`rand() % 21`将所取得的伪随机整数除以 21 取余数，结果为 0~20 之间的伪随机整数（参见程序`test2.cpp`第 14、15 行）。

（3）条件分支语句修改成“当答案正确时加 10 分”；答案不正确时不需做任何操作。因此，程序中使用缺省`else`分支的判断语句（参见程序`test2.cpp`第 18、19

[1] 用户从键盘输入的数据首先被存放到输入流缓冲区中等待被抽取。若输入流缓冲区中尚有数据，则抽取操作直接从其中抽取；若输入流缓冲区为空且遇到抽取操作，则程序暂停等待用户从键盘输入数据。若需要清空输入流缓冲区中尚未被抽取的数据可使用语句“`cin.sync();`”强制刷新输入流缓冲区。

[2] 在不同的编译系统下，`RAND_MAX`的值可能不同。程序员可用`cout << RAND_MAX << endl;`语句查看其值。在 MinGW C++ 中该值为 32 767。

行)。第 19 行中使用的运算符“`+=`”(被称为迭代赋值运算符)累加得分。第 20 行用运算符“`++`”(被称为增量运算符)使循环控制变量 `i` 随循环次数增加 1, 起到循环计算次数的作用。

改进的程序如下。

源代码 2.2 算术测验程序之二

```

1 // test2.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int x, y, z;           // 再定义两个整型变量
8     int i, score;
9
10    score = 0;            // 开始答题前, 成绩为 0
11    i = 0;                // i 为循环控制变量, 初值为 0
12    while(i<10)          // 继续循环的条件为 i<10
13    {
14        x = rand() % 21;   // 给 x 赋 0~20 之间的某个随机整数值
15        y = rand() % 21;   // 给 y 赋值。x 与 y 的值不一定相同
16        cout << x << "□+□" << y << "□=□";
17        cin >> z;
18        if(x+y == z)
19            score += 10;    // 回答正确, 加 10 分
20        i++;              // i 增加 1, 相当于 i = i+1;
21    }
22
23    cout << "成绩: " << score
24    << "分" << endl;      // 完成测试后, 输出成绩
25
26    return 0;
}

```

多次运行该程序就会发现: 产生随机数的规律不变。这就是称其为“伪随机数”的原因。因为这些所谓的“随机数”是按照某个公式计算出来的。计算公式中用到一个被称为“种子”的整数, 程序员可以通过设定不同的“种子”使 `rand` 函数产生随机性更好的随机数。选定一个与时间相关的量做“种子”是一个常用的好方法。具体做法如下(只需要模仿)。

(1) 在第 2 行与第 3 行之间添加文件包含指令 `#include <ctime>`。

(2) 在第 8 行后增加如下两行语句:

```

time_t t;           // 类型 time_t 即为类型 long, 定义一个变量 t
srand(time(&t));  // 用获取的系统时间做“种子”

```

2.2.2 C++ 基本运算

1. 算术运算

算术运算符有 + (加)、- (减)、* (乘)、/ (除) 以及 % (整除取余, 只能对整型数据进行操作)。

一般地，本质上，系统只提供了数据类型相同的数据之间的算术四则运算。不同数据类型的数据之间的算术运算须通过类型自动转换或强制转换成相同数据类型后进行。

```

int a, b, c;
double x, y, z;
unsigned int u=3, v=4;
b = 5;
c = 2;
a = b + c + b/c;      // a 的值为 9
y = 5;
z = 2;
x = y + z + y/z;      // x 的值为 9.5

if(u-v >= 0)          // 无论 u,v 取何值，条件始终成立。此处 u-v 为 4294967295
    cout << "u>=v" << endl;    // 永远输出 u>=v
else
    cout << "u<v" << endl;

if(u >= v)            // 此 if 语句正确
    cout << "u>=v" << endl;
else
    cout << "u<v" << endl;

```

由于变量 **b** 与变量 **y**，变量 **c** 与变量 **z** 在计算机内存中的存放格式不同，计算机执行加法运算 **b+c** 与 **y+z** 的具体操作是不同的（尽管使用相同的加法运算符号“+”）。

需要注意的是两个整数相除的结果为它们的整数商，舍弃余数。

如 $5/2$ 及 $(-5)/(-2)$ 为 2 ， $-5/2$ 及 $5/(-2)$ 的值为 -2 。表达式 m/n 的绝对值及符号用数学公式表示为

$$\begin{cases} |m/n| = |m|/|n| \\ \text{sgn}(m/n) = \text{sgn}(m)\text{sgn}(n) \end{cases}$$

运算符 **%** 只能对两个整型数操作，即两个整数相除，取其余数。表达式 $5\%2$ 的值为 1 。事实上 $x\%y$ 按照公式 $x-(x/y)*y$ 计算。因此，表达式 $5\%(-2)$ 为 1 。表达式 $-5\%2$ 及 $-5\%(-2)$ 为 -1 。

2. 类型转换

一般来说，运算符对操作数的数据类型是有要求的。当操作数的数据类型不符合要求时，编译系统将尝试类型的自动转换（或称为隐式转换）。若转换不成功，则报错。我们建议程序员在程序中使用类型转换运算符进行强制转换（或称为显式转换），以保证计算按程序员的要求进行。

基本数据类型自动转换的规则是向字节数更长的数据类型转换。其他方向的转换需要使用类型转换运算符进行强制转换，此时可能造成数据精度的损失。

类型转换运算符为单目运算，其一般形式为

数据类型(右值表达式)

或者

(数据类型)右值表达式

其中的括号不可缺少。类型转换操作只将表达式做右值处理，即将其值取出后按指定的数据类型建立并初始化一个临时变量，让该临时变量参与其他的运算。因此，类型转换不改变变量定义时所规定的数据类型，也不改变变量的值。例如：

```
int b, c;
double x, y, z;
b = 5;
c = 2;
x = b + c + double(b/c);           // x 的值为 9.0。b, c 的值不变
y = b + c + (double)b/c;          // y 的值为 9.5。b, c 的值不变
z = b + c + 1.0*b/c;              // z 的值为 9.5。b, c 的值不变
```

表达式 $y = b + c + (\text{double})b/c$ 的计算涉及多个运算。各个运算按其优先级从高到低依次为类型转换（取 b 的值并将其显式地转换成 `double` 型数值 5.0，取 c 的值并将其隐式地转换成 `double` 型数值 2.0）；计算两个双精度浮点型临时变量的值的除法（5.0/2.0）得到 2.5；两个整数（ b 与 c ）相加，得到整数 7；然后将整数 7 隐式地转换成 `double` 型数值 7.0；计算两个双精度浮点数的和 7.0+2.5；最后将计算结果 9.5 赋值给左值 y 。共计执行 7 个运算操作，如图 2-1 所示。

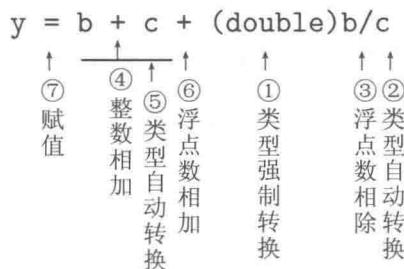


图 2-1 运算操作顺序示例

3. 自增及自减运算

变量增 1 及减 1 运算是计算机程序中频繁使用的操作。C++语言提供了变量增 1 或减 1 的运算。它们是单目运算，优先级为 2 与 3。根据运算符在操作数的前后分为前置运算、后置运算两种。也分别被称为前（后）增（减）量运算。自增及自减运算符只能对左值操作，是对变量既读又写的访问操作。

前置自增运算 (`++变量名`) 首先将变量的值增加 1，然后以新值参加表达式的其他运算。前置自增的结果可以做左值。如 `++++a` 即 `++(++a)` 合法。

前置自减运算 (`--变量名`) 首先将变量的值减去 1，然后以新值参加表达式的其他运算。前置自减的结果可以做左值。如 `----a` 即 `--(--a)` 合法。

后置自增运算 (`变量名++`) 先以变量的原值参加所在表达式的其他一次运算，然后变量的值增加 1。后置自增运算的结果不能做左值。`++(a++)` 及 `a++++` 均有语法错。

后置自减运算 (`变量名--`) 先以变量的原值参加所在表达式的其他一次运算，然后变量的值减去 1。后置自减运算的结果不能做左值。`--(a--)` 及 `a----` 均有语法错。

例如：

```

int a = 0;
cout << ++a << endl;           // 输出 1 (先增 1 以新值 1 输出)
cout << a << endl;             // 输出 1
a = 0;
cout << a++ << endl;           // 输出 0 (先以原值 0 输出)
cout << a << endl;             // 输出 1 (表明 a 的值被改变)

```

4. 关系运算

比较两个量之间的大小、相等或不等的运算被称为关系运算。C++ 语言中，如下六个运算符号 `>`、`>=`、`<`、`<=`、`==` 和 `!=` 被称为关系运算符，用于比较两个量之间的关系，分别为大于、大于或等于、小于、小于或等于、等于和不等于。关系运算的结果为逻辑值（即 `bool` 型）：`true` “真”，或者 `false` “假”。若关系成立，则关系运算的结果为 `true`（同整数 1）；若关系不成立，则关系运算的结果为 `false`（同整数 0）。

请读者务必严格区分、正确使用赋值运算符“`=`”和关系运算符“`==`”。

5. 逻辑运算

关于逻辑（`bool` 型）值 `false`, `true` 的运算有逻辑非！（单目运算）、逻辑与 `&&`、逻辑或 `||`。它们的运算法则（真值表）如下：

!	真		假			真		假	
	真	假	真	假		真	假	假	真
	0	1	真	1	0	真	1	1	
			假	0	0	假	1	0	
逻辑非				逻辑与				逻辑或	

逻辑运算的结果为逻辑值。系统给出的逻辑值与整数兼容，“逻辑真（`true`）”用 1 表示，“逻辑假（`false`）”用 0 表示。系统判断真假时，将非零数当成“逻辑真（`true`）”，将零当成“逻辑假（`false`）”。

例如，给定年份 `int year = 2016;` 判断该年是否为闰年。

历法规定，年份能被 4 整除但不能被 100 整除，或者年份能被 400 整除者为闰年。如：

```

int year;
bool flag;
year = 2016;
flag = year%4==0 && year%100!=0 || year%400==0;
cout << flag << endl;           // 输出 1, 表示 2016 年为闰年
year = 2017;
flag = year%4==0 && year%100!=0 || year%400==0;
cout << flag << endl;           // 输出 0, 表示 2017 年为平年

```

对于多个逻辑表达式的“或者”运算（`||`），只需要判断到其中（从左到右）有逻辑真（`true`）便可知整个逻辑表达式的结果为真。对于多个逻辑表达式的“并且”运算（`&&`），只要判断到其中（从左到右）有逻辑假（`false`）便可知整个逻辑表达式的结果为假。C++ 语言充分利用了逻辑计算的这个特点，用所谓的逻辑表达式的短路计算法省去不必要的计算。

虽然“或者 (||)”运算、“并且 (&&)”运算皆满足交换律，但是逻辑表达式的短路计算对 C++ 程序的运行会产生一定的影响，程序员要善于利用其有利影响，避免其不利影响的发生。例如：

```
int x, y, z;
x = rand() % 21;
y = rand() % 21;
cout << x << "/" << y << "u=u";
cin >> z;
if(y!=0 && x/y==z)           // 短路求法避免了除数为 0 的计算发生
                                // 因此，不能写成 if(x/y==z && y!=0)
    cout << "回答正确！" << endl;
else if(y!=0)
    cout << "回答错误。" << endl;
else
    cout << "题目错误（除数为零）。" << endl;
```

6. 迭代赋值运算

`+=`、`-=`、`*=`、`/=` 和 `%=` 为算术迭代赋值运算符。其一般格式为

变量名 <code>+=</code>	右值表达式
变量名 <code>-=</code>	右值表达式
变量名 <code>*=</code>	右值表达式
变量名 <code>/=</code>	右值表达式
整型变量名 <code>%=</code>	整型右值表达式

计算这些运算表达式时，首先将其左边的量作为右值（即读取变量的当前值）与运算符右边的右值表达式进行相应的（`+`、`-`、`*`、`/` 或 `%`）运算，再将计算结果赋值给左值变量。即先对变量进行读访问操作，然后进行相应的运算，最后对该变量进行写访问操作。显然要求该变量事先具有有意义的数值供读取。例如：

```
double x=5, y=10.5;
int a=15, b=3;
x += y;                      // 相当于，但效率高于 x = x + y;
x -= 5;                      // 相当于，但效率高于 x = x - 5;
x *= 3;                      // 相当于，但效率高于 x = x * 3;
x /= 2;                      // 相当于，但效率高于 x = x / 2;
a /= 2;                      // 相当于，但效率高于 a = a / 2;
a %= b;                      // 相当于，但效率高于 a = a % b;
```

2.2.3 C++ 程序流程控制

1. 条件分支语句

规定 2.8（分支语句） C++ 提供的刻画选择结构的语句为分支语句（或称为 `if` 语句）。其语法格式和流程图如图 2-2 所示。

分支语句对应算法中的选择结构。语句中的圆括号不能缺少。若语句 2 为空语句，则连同 `else` 在内均可省略。

若某分支需要执行多条语句，则必须用一对花括号将这些语句包围起来，使之在语法上作为一条单语句（复合语句，参见第 3.4 节规定 3.10）。

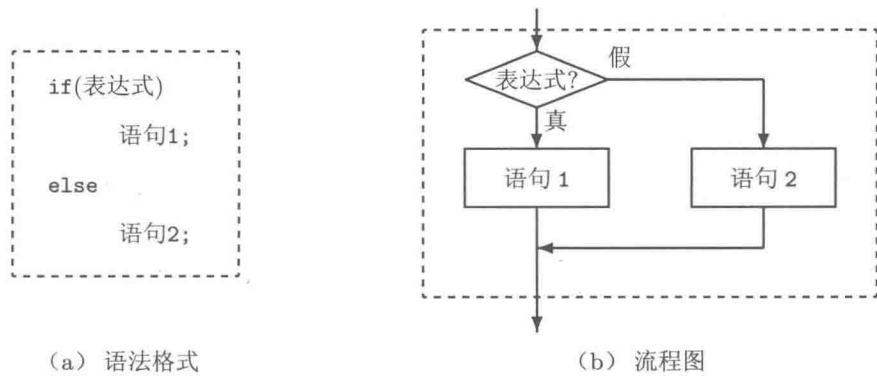


图 2-2 if 语句

`if` 语句可以嵌套，以实现多路选择。`if` 语句嵌套时，`else` 与 `if` 配对的规定为 `else` 与其上面最近的、同一层次的、尚未与其他 `else` 配对的 `if` 配对。例如，若有语句：

```
if(x>=0)
    if(x>0)
        cout << x << "大于0." << endl;
    else
        cout << x << "小于0." << endl;
```

则编译器将上面的程序段按如下方式编译：

```
if(x>=0)
{
    if(x>0)
        cout << x << "大于0." << endl;
    else
        cout << x << "小于0." << endl;
}
```

显然，上面的程序段不是程序员的本意。正确的程序段应该如下，尽管花括号内仅为一个单语句，此处的花括号不可缺少。

```
if(x>=0)
{
    if(x>0)
        cout << x << "大于0。" << endl;
}
else
    cout << x << "小于0。" << endl;
```

C++ 程序的书写格式是自由式的，书写格式不能代替语法规规定。应该按照一定的书
写规范编辑源程序代码，养成良好的程序书写习惯。程序决不会因为书写格式紧凑而执行
效率高。

根据 `else` 与 `if` 配对的规定，不难分析并“推导”出如下多路分支结构。

```
if( 表达式 1 )
    语句 1;
else if( 表达式 2 )
    语句 2;
```

```

    :
else if( 表达式 n )
语句 n;
else
语句 n+1;

```

例如：

```

char grade;           // 五级分制成绩
double score;        // 百分制分数
cout << "请输入百分制分数: ";
cin >> score;
if(score>=90)
    grade = 'A';
else if(score>=80)
    grade = 'B';
else if(score>=70)
    grade = 'C';
else if(score>=60)
    grade = 'D';
else
    grade = 'E';
cout << "转换成五级分制成绩: " << grade << endl;

```

请读者思考，在上面的条件判断中为什么不必用 `(score>=80 && score<90)`，而只需要简单地用 `(score>=80)`？

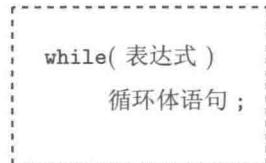
2. 循环语句

C++ 提供的刻画循环结构的语句有三种形式，这三种形式能够相互转换。程序员可以只用其中的任意一种便能编写任何循环结构的程序。这三种形式各有其最便利的应用场合。

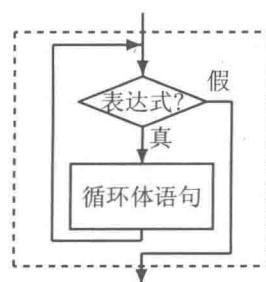
在循环结构中，循环体内的语句执行完后，称为结束本轮（本次）循环，准备进入下一轮循环。整个循环语句是否结束，依赖于：

- ① 准备进入下一轮循环的条件判断（即继续循环的条件判断）。
- ② 循环体内是否有机会执行中断循环的 `break` 语句，甚至 `return` 语句。

规定 2.9 (while 循环) while 循环的语法格式及其对应的流程图如图 2-3 所示。



(a) 语法格式



(b) 流程图

图 2-3 while 循环

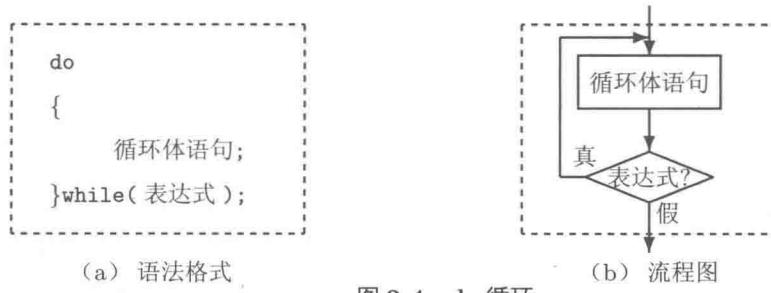


图 2-4 do 循环

规定 2.10 (do 循环) do 循环的语法格式及其对应的流程图如图 2-4 所示。

do 循环的特点是先执行循环体语句，再判断继续循环的条件。因此，在 do 循环中，循环体语句至少被执行一次。

do 循环语句中的一对花括号可能是不必要的。建议读者在书写 do 循环语句时写下一对花括号，并且将右花括号写在 while 之前（写在同一行）。这样更容易区分该语句是 do 循环语句块中的最后一行，而不是一个 while 循环语句。

规定 2.11 (for 循环) for 循环的语法格式及其对应的流程图如图 2-5 所示。

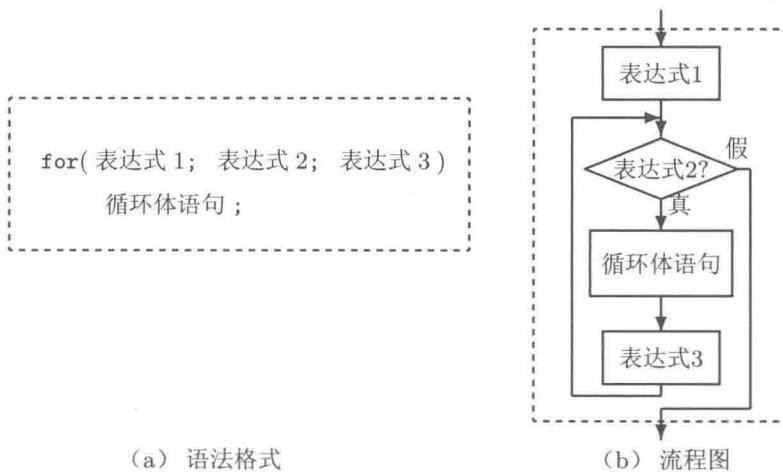


图 2-5 for 循环

其中“表达式 1”“表达式 2”和“表达式 3”都可以缺省，但其中的两个分号不可缺少。缺省“表达式 2”时，默认为 true（即“真”，亦即 1），称其为“永真循环”。此时，循环体内应该有机会执行到 break 语句（如图 2-7 所示），以避免程序中出现“死循环”。“表达式 1”只执行一次，常用于为进入循环体做准备；“表达式 3”也可以看成循环体中的最后一条语句；也可以将循环体语句放在“表达式 3”的位置。for 循环的循环体语句，有可能一次都不执行，而“表达式 1”总是会执行一次的。

显然，在 for 循环中“表达式 1”和“表达式 3”均缺省时，便相当于 while 循环。

3. 开关语句

规定 2.12 (开关语句) C++ 提供分情况多路选择的开关语句，其语法格式为

```

switch( 整型或枚举型表达式 )
{
    case 常量表达式 1 :
        语句组 1 ;
    case 常量表达式 2 :
        语句组 2 ;
    .
    .
    case 常量表达式 n :
        语句组 n ;
    default :
        语句组 n+1 ;
}

```

`switch` 语句块中的 `case` 相当于语句标号。根据进入该语句块时所提供的整型或枚举型表达式的值，跳转到相应的 `case` 处开始执行其后的所有语句组语句直到整个 `switch` 语句块结束，或者遇到 `break` 语句而结束整个 `switch` 语句块。其执行流程如图 2-6 所示。`default` 及其“语句组`n+1`；”是可以缺省的。

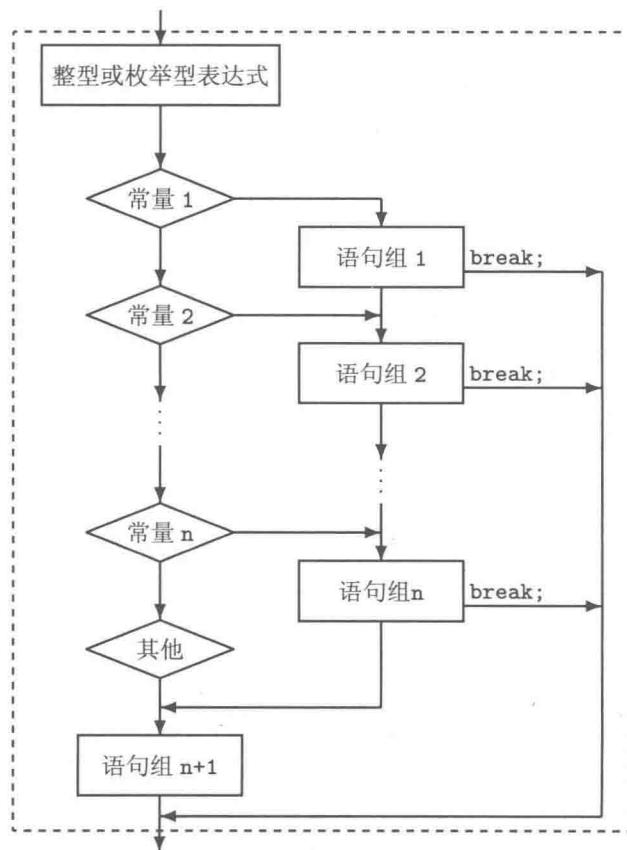


图 2-6 `switch` 开关语句

`case` 后面所跟的必须是常量，而且各常量值不能重复（这意味着同一 `switch` 语句块中不能有重复的标号）。流程图中借用菱形框表示各标号，并不表示条件判断。若需要用浮点型数据的关系运算作为判断条件，只能使用 `if` 语句嵌套实现多路分支选择。

4. 跳转语句

规定 2.13 (break 语句) 结束其所在的一层循环语句或者开关语句，从本层结构之后的语句起执行。若有多重循环，则 break 仅中断其所在的那一层循环。

规定 2.14 (continue 语句) 提前结束本轮循环，准备进入下一轮循环。即本轮循环体中所剩下的语句将不予执行。

图 2-7 给出了 break 和 continue 在 for 循环语句中的作用，在其他两种循环中的作用是类似的。

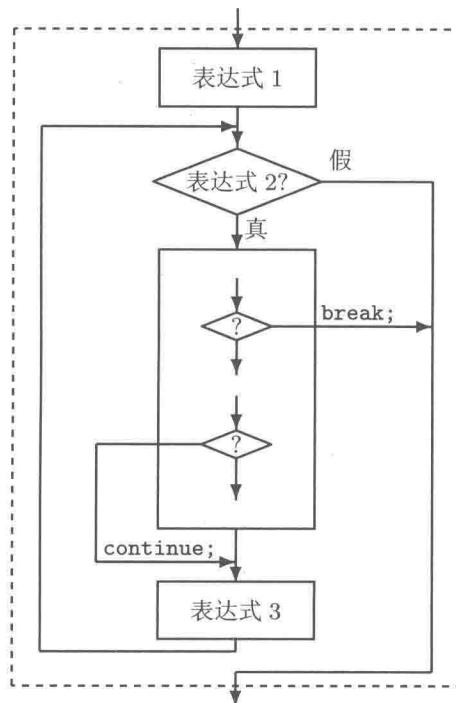


图 2-7 for 循环中的跳转

规定 2.15 (return 语句) return 语句也被称为返回语句，其语法格式为

```
return 表达式;
```

或者

```
return ;
```

在程序的执行过程中，用函数名及一定的实际参数调用函数时，系统将保存当前的状态，并将程序的控制权转移给该函数。在该函数体内若遇到 return 语句或执行完该函数体所有的语句时，该函数立即结束，并将控制权交还给调用它的函数。若在主函数 main 中遇到 return 语句，则将引起整个程序结束，返回到操作系统。

返回语句中的表达式的类型应该与函数的返回类型相匹配（必要时通过类型转换）。若函数的返回类型为 void，则函数体内可以编写 return; 语句（无表达式的返回语

句），也可以不编写 `return`；语句（函数体内的语句执行完毕后，自动返回）。如：

```

1 #include <iostream>
2
3 void greeting()
4 {
5     std::cout << "Hello." << std::endl;
6     return ;           // 交还程序控制权。此语句可缺省。
7 }
8
9 int main()          // 程序执行的起点
10 {
11     greeting();    // 调用函数，程序控制权转移给 greeting 函数
12     return 0;        // 程序执行的终点
13 }
```

规定 2.16（语句标号与 `goto` 语句） C++ 程序中，可以在某语句前设置一个标号（标识符后跟冒号）。`goto` 语句可以从其所在的语句，或向前或向后跳转到任意的标号处。注意：此处的语句标号与 `switch-case` 语句中相当于语句标号的 `case` 是不相同的。

`goto` 语句的滥用将导致程序结构的混乱，是结构化程序设计所忌讳的。我们不提倡使用 `goto` 语句。仅限制在一种情形^[1] 下使用：从多重循环的深处直接跳出多重循环。这样处理，实际上有利于程序的结构。因为，倘若要一层层地退出，势必要增加一些标志量及标志值的判断，使程序的结构反而复杂。

2.3 基本程序扩展

本节将前面的“算术测验”程序改编成“主控模块/功能模块”结构的程序，目的是使程序具有良好的可扩展性。具体地，首先将 `test2.cpp` 程序按照功能拆分成两个函数；然后将程序写成多文件结构并进行扩展使之能够进行算术减法运算的测验。

2.3.1 简单函数

将某些完成一定功能的程序语句段包装成一个整体，形成一个功能单一、相对独立的模块，以便灵活地组装到其他功能模块中或者主控模块中。程序的功能模块在 C++ 语言中体现为 **函数**（或者称为子程序）。

一个 C++ 程序由一个主函数和若干个其他函数构成，即 **C++ 用函数组织程序**。编写 C++ 程序就是编写函数。一个函数是否被执行或执行多少次与其编写时的位置无关，而取决于是否有机会执行调用该函数的语句。因为，**程序运行是由函数的相互调用所驱动的**。除了给全局变量分配内存空间及初始化在主函数执行前完成外，主函数是程序执行的起点，主函数执行完毕（在主函数中返回）也就意味着程序即将结束（之后，还要销毁已经定义的全局变量）。

本节介绍简单的函数：无形式参数。下面将 `test2.cpp` 程序改编成如下形式。

^[1] 在 C 语言中，一般还用 `goto` 语句使程序流程跳转到出错处理处。由于 C++ 有性能卓越的“异常处理”机制（参见第 15 章），而不再在这种场合下使用 `goto` 语句。

源代码 2.3 算术测验程序之三

```

1 // test3.cpp
2 #include <iostream>
3 #include <ctime>
4 using namespace std;
5
6 int add_test();           // ①函数声明（函数原型）
7
8 int main()
9 {
10    int score;
11
12    score = add_test();      // ②函数调用（执行语句）
13                                // 函数调用表达式 add_test() 本身表示函数的返回值
14    cout << "成绩: " << score << "分" << endl;
15    return 0;
16 }
17
18 int add_test()           // ③函数定义（函数功能实现）
19 {
20    int x, y, z;
21    int i, score;
22    time_t t;
23    srand(time(&t));      // 设置随机数发生器的“种子”
24
25    score = 0;             // 开始答题前，成绩为 0
26    i = 0;                 // i 为循环控制变量，初值为 0
27    while(i<10)            // 继续循环的条件为 i<10
28    {
29        x = rand() % 21;   // 给 x 赋 0 ~ 20 之间的某个随机值
30        y = rand() % 21;   // 给 y 赋值。x 与 y 的值不一定相同
31        cout << x << " + " << y << " = ";
32        cin >> z;
33        if(x+y == z)
34            score += 10;     // 回答正确，加 10 分
35        i++;              // i 增加 1，相当于 i = i+1;
36    }
37    return score;          // 完成测验后，将成绩（分数）返回
38 }

```

本程序与程序 test2.cpp 的功能相同。本程序由函数 `add_test` 和函数 `main` 组成。其执行过程如下。

- (1) 执行主函数，定义主函数中的 `score` 变量（其初值不可预知）。
- (2) 执行程序第 12 行前半段：调用 `add_test` 函数（即转子程序），将程序的控制权转移给该函数（从第 18 行起）。
 - ① 定义 6 个变量 `x`, `y`, `z`, `i`, `score`, `t`。此处的 `score` 与主函数中的 `score` 相互独立，即既无关联也无冲突，是不同的两个变量。
 - ② 随机产生 10 道算术题、显示题目、接收答案输入、判题评分等（第 25~36 行）。
 - ③ 返回测验成绩，销毁本函数中定义的 6 个变量 `x`, `y`, `z`, `i`, `score`, `t`（第 37、38 行）。
 - (3) 程序的控制权返回给主函数，执行第 12 行后半段：用 `add_test` 函数的返回值

(该返回值就用函数调用表达式 `add_test()` 本身表示) 给主函数的变量 `score` 赋值。

(4) 输出测验成绩 (第 14 行)。

(5) 执行主函数中的返回语句 (第 15 行), 销毁主函数中的变量 `score`, 结束整个程序返回到操作系统。

在这两个函数中各自定义了一些变量, 这些变量皆为局部自动 `auto` 变量 (简称为局部变量)。局部自动变量的生命期随所在函数的调用及函数体中变量定义语句的执行而开始, 至所在函数返回而结束。局部变量的可见性也仅为其实所在函数内。因此, 尽管两个函数中都定义了变量 `score` (数据类型、变量名皆相同), 事实上, 这两个变量是相互独立的, 它们有各自所联系的内存单元。它们虽有重叠的生命期, 却没有同时的可见性, 故它们之间不会引起任何冲突。

函数必须先声明或先定义才能被调用。

① 程序中第 6 行为 **函数原型** 用于函数声明。函数声明的作用是告诉编译器 (同时也告知程序员) 如何使用该函数。从函数原型可见, 调用该函数时不需要提供实际参数 (即圆括号中不需要表达式, 但表示函数的圆括号不可缺少)。

② 调用该函数后将得到一个整型数值 (这个数值是执行函数的结果) 直接用函数调用表达式本身表示 (参见程序第 12 行)。

③ 函数定义 (函数功能的具体实现过程) 见程序第 18~38 行。

此外, 程序中并不在 `add_test` 函数中直接输出测试成绩, 而是将测试成绩返回给调用该函数的函数 (`main`)。这样做好处是: 主函数 (`main`) 中用变量 `score` 保存了 `add_test` 函数的执行结果, 不仅仅可以进行输出, 而且还可以做更多其他的操作。

2.3.2 多文件结构

在第 1 章中, 提到“C/C++ 语言支持众人集体开发大型软件”。显然众人集体开发程序时, 不会将所有的函数都写进同一个文件。一个 C++ 程序由多个文件组成, 主要有源程序文件和头文件两种。一般地, 将这些文件组织成一个工程文件 (也被称为项目文件)。可以向工程文件中添加新的文件或者添加已经存在的文件; 也可以从工程文件中剔除一些文件。

规定 2.17 (编译单元) C++ 程序以源程序文件为单位, 进行分割编译。源程序文件被称为**编译单元**。一个源程序文件可以包含多个头文件。头文件不是编译单元, 编译系统不编译头文件。头文件中的内容常在编译前插入到某源程序文件中进行编译。

本节仅介绍基本的多文件结构。读者只需要模仿这种结构。程序中使用了 I/O 流格式控制以便使测验题目在显示器上整齐地排列输出 (其语法详见第 3.5 节)。多文件结构的“算术测验”程序由如下三个文件组成。

① `test.cpp`: 源程序文件, 其内容为函数的定义。

② `test.h`: 头文件, 其内容为函数的声明。

③ `test_main.cpp`: 源程序文件, 其中有主函数的定义。

文件 `test.cpp` 与 `test.h` 为一组文件。可假定这两个文件是由程序开发小组的其他程序员所完成, 现移植到此进行组装。三个文件的具体内容如下。

源代码 2.4 算术测验程序之四

```

1 // test.cpp
2 #include <iostream>
3 #include <iomanip>      // 为了使用 I/O 流格式控制。见第 18 行、第 41 行
4 #include <ctime>
5 using namespace std;
6
7 int add_test()          // 函数定义
8 {
9     int x, y, z, score=0;
10    time_t t;
11    srand(time(&t));
12
13    cout << "加法测验 (共 10 题)" << endl;
14    for(int i=0; i<10; i++)
15    {
16        x = rand() % 21;
17        y = rand() % 21;
18        cout << setw(2) << x << "+" << setw(2) << y << "=";
19        cin >> z;
20        if(x+y == z)
21            score += 10;
22    }
23    return score;
24 }
25
26 int sub_test()          // 函数定义
27 {
28     int x, y, z, score=0;
29     time_t t;
30     srand(time(&t));
31
32     cout << "减法测验 (共 10 题)" << endl;
33     for(int i=0; i<10; i++)
34     {
35         x = rand() % 21;
36         y = rand() % 21;
37         if(x<y)
38         {
39             z = x; x = y; y = z;           // 借助 z 交换 x 与 y 的值
40         }
41         cout << setw(2) << x << "-" << setw(2) << y << "=";
42         cin >> z;
43         if(x-y == z)
44             score += 10;
45     }
46     return score;
47 }

```

```

1 // test.h
2 #ifndef TEST_H          // 采用判断某标识符是否没有定义
3 #define TEST_H          // 以避免重复包含 (详见第 7 章)
4
5 int add_test();          // 函数声明
6 int sub_test();          // 函数声明
7 #endif

```

```

1 // test_main.cpp
2 #include <iostream>
3 #include "test.h"           // 包含头文件，完成函数声明
4 using namespace std;
5
6 int main()
7 {
8     int choice=1, score;
9
10    while(true)
11    {
12        cout << "算术测验\n"
13        << "1——加法\n"
14        << "2——减法\n"
15        << "0——退出\n"
16        << "请选择：";           // 显示“菜单”
17        cin >> choice;          // 输入选项
18        if(choice<=0) break;
19        if(choice>2) continue;
20        switch(choice)         // 根据选项进行调度，调用不同函数
21        {
22            case 1: score = add_test(); break;
23            case 2: score = sub_test(); break;
24        }
25        if(score>=60)
26            cout << "成绩：" << score << "分，通过。" << endl;
27        else
28            cout << "成绩：" << score << "分，不及格。" << endl;
29    }
30    return 0;
31 }

```

2.4 C++ 程序开发流程

用 C++ 语言编写的源程序必须经过编译、连接，生成可执行文件后才能在操作系统的控制下运行。C++ 程序的基本开发流程如图 2-8 所示。

一个 C++ 程序可能由多个源程序文件、多个头文件等组成。而头文件的内容将在编译之前的预处理阶段，根据包含指令“插入”到源程序文件或其他头文件的当前位置。C++ 以源程序文件为编译单元分别进行编译（分割编译），分别生成目标代码文件。连接程序将各个目标代码文件，以及标准函数库代码连接起来，产生可执行文件。

系统在程序的编译、连接、运行阶段都具有检查错误的能力。只要出现错误就应该分析源程序、修改源程序，再重新编译、连接、运行程序。程序中的错误或问题被分成三类：第一类是错误 error(s)，遇此情况，将不会生成可执行文件；第二类是警告 warning(s)，若程序中仅有警告，则可以生成可执行文件。程序员应该认真对待编译系统给出的警告，仔细分析其原因（有些警告并不表示有错，只是系统的“温馨提示”）；第三类是程序运行时产生的错误。例如，计算过程中数据溢出（数据超出该类型所能表示的范围）、用 0 做除数、当应该输入数值时却输入了字母等。读者需要通过在计算机上多实践，不断积累经验，以提高自己调试程序的能力。

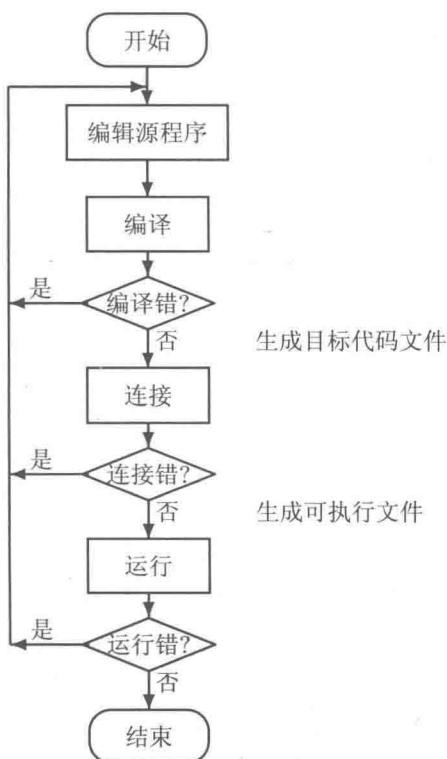


图 2-8 C++ 程序开发流程

2.5 C++ 应用程序集成开发环境简介

为方便程序员使用高级语言开发应用软件，一些公司或团体根据不同计算机语言研制开发了一些应用程序集成开发环境(Integrated Development Environment, IDE)。IDE 集成了编辑器(用于录入、修改、保存源程序文件)、编译器、连接器、调试及运行工具为一体，方便程序员使用计算机语言进行软件开发。常用的 C++ IDE 有 MinGW Developer Studio、Code Blocks 和 Microsoft Visual C++ 等。本教材中的所有程序均在这些 IDE 下编译通过。

需要指出的是 C++ IDE 只是 C++ 语言的开发工具，不等于 C++ 语言；C++ IDE 中集成的编译器是 C++ 语言的一种实现，并不一定完全符合 C++ 标准。

本教材中涉及非标准 C++ 的内容主要有“控制台输入输出”的几个函数，它们的原型在头文件 `conio.h` 中，如表 2-3 所示。

表 2-3 控制台输入输出函数

函数原型	功 能
<code>int getch();</code>	暂停，等待用户按任意键(不必回车)并返回所按键的编码
<code>int getche();</code>	同上，且回显(echo)用户所按键的字符
<code>int kbhit();</code>	判断是否按键，若是则返回 1，否则返回 0

本节提到的 C++ IDE 均支持这些函数。但使用了这些函数的程序不能直接移植到其他系统（如 UNIX/Linux 操作系统）中。

下面以 MinGW Developer Studio 为例介绍 C/C++ IDE 的一般使用方法。其他 IDE 的使用步骤大致相同。

MinGW Developer Studio 是一个小巧的可运行于 Windows 操作系统下的 C/C++ 应用程序集成开发环境。它集成了：

① Minimalist GNU for Windows 软件^[1]（简记 MinGW C++）即 Windows 版的 GCC 编译器。

② 一个功能完善的编辑器，支持代码折叠等。

③ 其他工具（如一个图形用户界面设计工具箱、一个资源编辑器等）。

该集成开发环境可以通过互联网免费获取、自由发布^[2]。

下面简要介绍运用 MinGW Developer Studio C/C++ IDE 开发 Windows 操作系统下控制台应用程序（Win32 Console Application）的过程。

(1) 启动 IDE (如图 2-9 所示)。

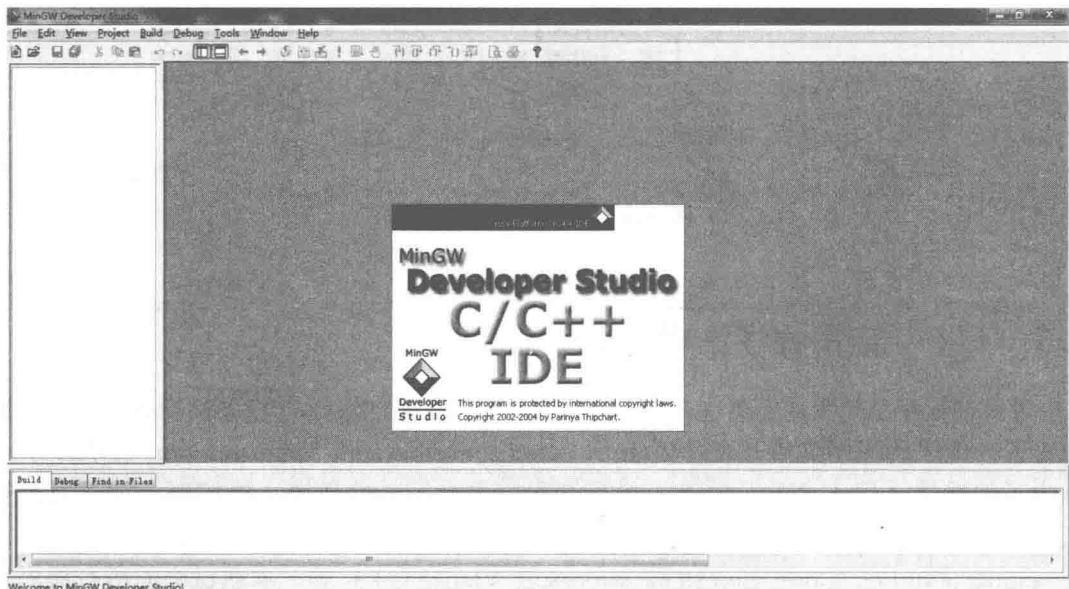


图 2-9 MinGW Developer Studio C/C++ IDE 主界面

(2) 新建工程文件。

在主界面的主菜单上选择 File | New (如图 2-10(a) 所示)，则弹出如图 2-10(b) 所示的对话框（系统自动进入 Projects 选项卡），选择 Win32 Console Application；然后点击 Location 的右下方省略号处，选择工程文件存放的文件夹（即子目录）。例如：D:\CPP（假定文件夹 D:\CPP 已经存在，请事先建立该文件夹）表示将工程文件的工作目录创建在该文件夹下；再在 Project name 的编辑栏中直接输入 test1 作为工程文件的文件名，同时该工程文件的工作文件夹为 D:\CPP\test1；最后单击 OK 按钮。

^[1]<http://www.mingw.org>

^[2]<http://koti.mbnet.fi/vaultec/mingwstudio.php>

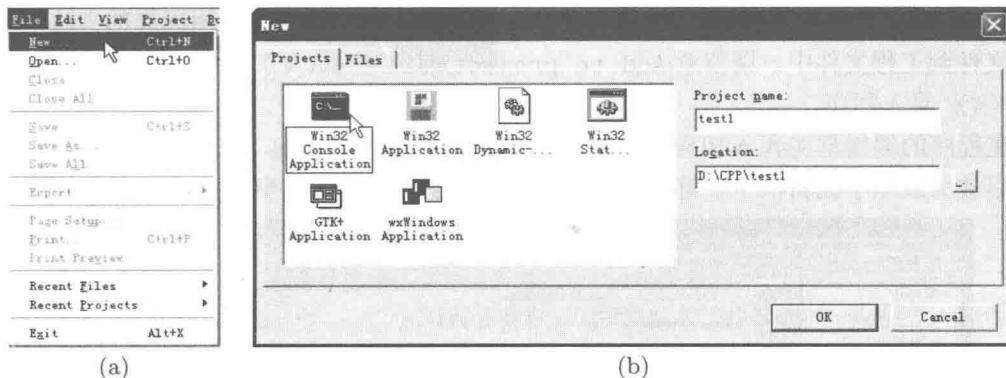


图 2-10 创建工程文件

此时将在 D:\CPP 文件夹中再自动建立一个文件夹 D:\CPP\test1，并在新文件夹中建立一个工程文件 test1.mdsp。

新建工程文件成功后，该工程文件便处于“已打开”的状态。之后，便可以添加新的源程序文件、头文件，或添加已经存在的源程序文件、头文件到该工程文件中。

须注意的是：一个 C/C++ 程序往往由多个源程序文件、多个头文件等组成，工程文件能够很好地管理它们：①自动打开所有相关程序文件；②分割编译所有的编译单元；③连接所有分割编译的目标代码生成可执行程序。即，一个程序对应一个工程文件。一个程序（一个工程文件）包含多个源程序文件或头文件。因此，打开一个程序的方法是打开其工程文件，而不是孤零零地单独打开一个源程序文件（即使整个程序仅由一个源程序文件组成）。

打开一个已经存在的工程文件的方法：鼠标双击扩展名为 .mdsp 的文件图标。或者在主菜单上选 Project | Open Project，再根据所弹出的对话框选择工程文件。

当打开了工程文件后，主界面的左侧将不再是“灰暗的”，其中列有该工程文件管理的所有文件，将其中的加号“+”展开，可见具体的文件。主要有源程序文件、头文件两类。

(3) 添加新源程序文件或头文件到工程文件中。

当某工程文件打开后，仍然在主界面的主菜单上选 File | New，则弹出的对话框自动跳至“新建文件 Files”选项卡（如图 2-11 所示）。

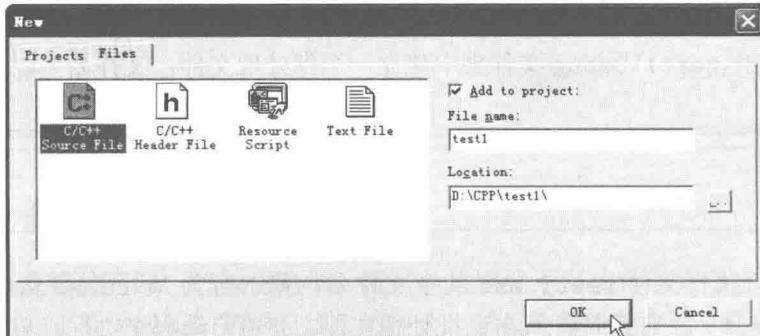
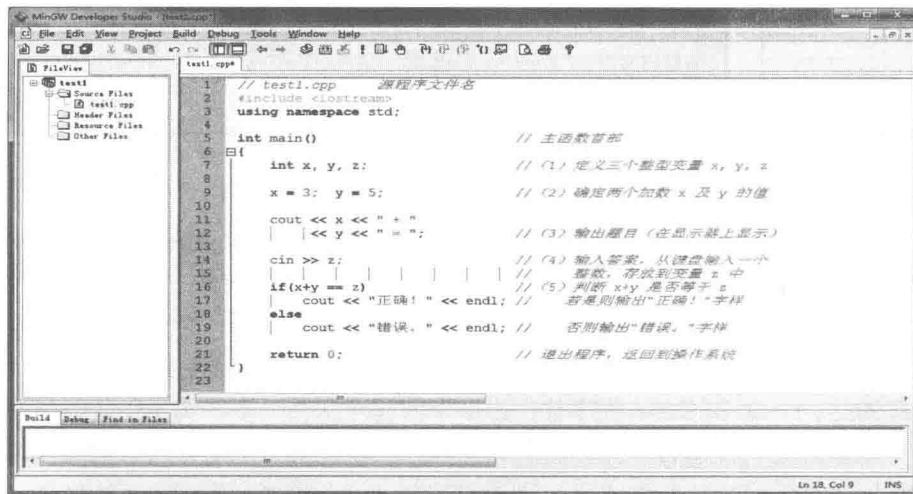


图 2-11 添加新源程序文件到工程文件中

此时将在 D:\CPP\test1 文件夹中新建一个文件 test1.cpp。采用同样的方法可以添加头文件到工程文件中。请留意 Add to project 前的对勾（√）是否出现。

（4）录入程序。

在程序的编辑框中录入相应文件的内容。此时可用键盘上的 [Ctrl] 键，配合鼠标滚轮，以放大或缩小编辑框中字符的大小，特别适合于课堂讲解（参见图 2-12）。



```

// test1.cpp  源程序文件名
#include <iostream>
using namespace std;

int main() // 主函数首部
{
    int x, y, z; // (1) 定义三个整型变量 x, y, z
    x = 3; y = 5; // (2) 确定两个加数 x 及 y 的值

    cout << x << " + "
    | << y << " = ";
    // (3) 输出题目(在显示器上显示)

    cin >> z; // (4) 输入答案, 从键盘输入一个
    // 整数, 存放到变量 z 中
    if(x+y == z)
        cout << "正确!" << endl; // (5) 判断 x+y 是否等于 z
    else
        cout << "错误." << endl; // 否则输出"错误."字样

    return 0; // 退出程序, 返回到操作系统
}

```

图 2-12 录入程序

（5）编译链接及运行程序。

主菜单上有三个图标分别是编译、编译并连接、编译连接并运行。直接用鼠标单击其中的感叹号便可以由系统决定是否需要重新编译、重新连接、直至运行。其中任何一步出现错误，均需要程序员进一步修改程序。对于编译系统给出的警告错误（温馨提示）也需要认真对待。

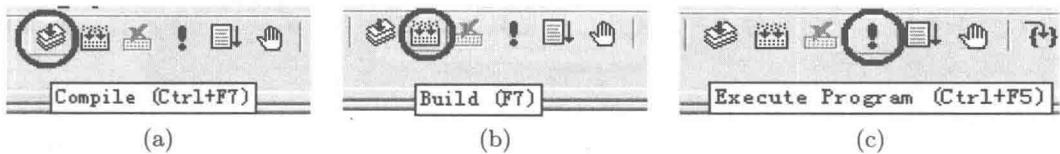


图 2-13 编译、连接与运行程序

请读者观察 D:\CPP\test1 文件夹中的文件，及其子文件夹 Debug 中的文件，以了解系统将用户的程序存放在何处。

可在 D:\CPP\test1\Debug 文件夹中建立一个批处理文件，文件名可取为 run.bat。文件内容如下。

```

test1
pause

```

其中第一行为可执行文件 test1.exe 的主文件名；第二行为 Windows 命令（功能为暂停，按任意键继续）。存盘后，双击该批处理文件，亦可启动程序执行。与直接双击文件 test1.exe 启动程序执行不同之处是程序执行完毕后还执行暂停命令，以便操作者有足够的时间看清程序的输出结果，待用户按任意键后再关闭 Windows 命令窗口。

2.6 趣味程序——变换的字符

源代码 2.5 变换的字符

```

1 #include <iostream>
2 #include <ctime>           // 为了使用函数 clock
3 #include <conio.h>         // 为了使用函数 kbhit 和函数 getch
4 using namespace std;
5
6 void delay(int n)          // 延时 n 毫秒
7 {
8     clock_t t=clock();      // 记录进入本函数的时刻
9     while(clock()-t<n)    // 继续循环条件：时间差小于 n
10        ;                  // 循环体为空语句，因为无其他事情要做
11 }
12
13 void trans_char()
14 {
15     int n = 0;
16     cout << "按任意键……" << endl;
17     while(kbhit()==false)  // 继续循环条件：未按键
18     {
19         n = (n+20) % 200; // 设置延时时间，0~200 毫秒，步长 20 毫秒
20         cout << "-\r";   delay(n); // 转义字符 \r 为回车（不换行）
21         cout << "/\r";   delay(n);
22         cout << "| \r";  delay(n);
23         cout << "\\\r"; delay(n);
24     }
25     getch();               // “吃掉”所按键产生的输入
26     cout << endl;
27
28     while(!kbhit())        // 继续循环条件：未按键
29     {
30         n = (n+20) % 200;
31         cout << "☆\r";  delay(n);
32         cout << "★\r";  delay(n);
33     }
34     getch();
35     cout << endl;
36 }
37
38 int main()
39 {
40     trans_char();
41     return 0;
42 }
```

2.7 小结

学习语言从模仿开始，边模仿边学。语言的运用绝不应该待学完全部语法后才开始。“主动出击，在实战中成长”是学习语言的要诀。学习计算机程序设计语言也是一样的。

学习C++语言程序设计，先模仿其外观、风格，“照葫芦画瓢”是常用方法。字母的大小写、空格空行、分号括号均需认真对待。C++ 程序的书写格式是自由的。虽然格

式“规范”不是必需的，但是，格式优美的程序犹如一首诗，它能体现其设计者的专业素养。

程序员犹如一个钟表匠：

- ① 钟表几乎都是为他人而制作的。
- ② 尽量使钟表的外表简洁美观、含义清晰。
- ③ 留给用户一些简单的操作接口（按钮、旋钮），将复杂的操作包装在内部。

因此，程序员也要时刻考虑为使用者（他人）提供简洁明了、符合问题描述规范的接口界面，如输出适当的提示信息以指引操作者进行操作。

C++ 应用程序开发集成环境（C++ IDE）的使用方法应该在第一次上机实践就学会。C++ 支持众人集体开发大型软件，一个程序可能包含多个文件，在 C++ IDE 中这些文件被组织到一个工程文件中进行有效的集中管理。即使程序中只有一个源程序文件，在打开该程序时，也不要只孤零零地打开该一个源程序文件，而应该打开一个工程文件（双击 MinGW C++ 中的 .mdsp 文件，或者 Code Blocks 中的 .cbp 文件），工程文件会很好地管理其中的所有文件。

一个工程文件（可能包含多个文件）对应一个程序。一个程序中有且仅能有一个主函数 main。（程序中没有全局变量时）主函数常常是程序执行的起点及终点，其他函数执行与否取决于是否有机会执行调用它的表达式，与该函数所在的文件及编排前后次序无关，只需保证先声明或先定义后调用。

调试程序是程序员的基本工作内容。调试程序磨砺程序员的意志，锻炼程序员的思维。C++ 集成开发环境为程序员提供了方便的调试工具。学会调试程序，也是本课程的目的之一。

练习 2

1. 指出下列各程序中的错误

(1)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "HELLO,world." << endl;
7     return 0;
8 }
9
10 int main()
11 {
12     cout << "I am a programmer." << endl;
13     return 0;
14 }
```

(2)

```

1 #include <iostream>
2 using namespace std;
3
```

```

4 Int Main()
5 {
6     cout << "HELLO,world.";
7     << "你好。" << endl;
8     return 0;
9 }
```

(3)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int x, y, z;
7     z = x + y;
8     cout << "Input two integers:" ;
9     cin >> x, y;
10    cout << x << " + " << y << " = " << z << endl;
11    return 0;
12 }
```

(4)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int x, y, z;
7
8     x = rand() % 100;
9     y = rand() % 100;
10    cout << x << " + " << y << " = ";
11    cin >> z;
12    if(z == x + y)
13        cout << "正确！" << endl;
14    else
15        cout << "错误。" << endl;
16    return 0;
17 }
```

(5)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     double x = 0.5, y = -x;
7
8     if(1 < x < 2)
9         cout << x << "在区间(1,2)中。" << endl;
10    else
11        cout << x << "不在区间(1,2)中。" << endl;
12
13    if(-1 < y < 0)
14        cout << y << "在区间(-1,0)中。" << endl;
```

```

15     else
16         cout << y << "不在区间(-1,0)中。" << endl;
17     return 0;
18 }
```

(6)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int i, sum;
7
8     i = 1;
9     while(i<=100)
10    {
11        sum += i;
12    }
13    cout << "Sum=" << sum << endl;
14    return 0;
15 }
```

(7)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     greeting();
7     return 0;
8 }
9
10 void greeting()
11 {
12     cout << "Hello." << endl;
13 }
```

(8)

```

1 #include <iostream>
2 using namespace std;
3
4 int func();
5
6 int main()
7 {
8     int sum=0;
9     func();
10    cout << "Sum=" << sum << endl;
11    return 0;
12 }
13
14 int func()
15 {
16     for(int i=1; i<=100; i++)
17         sum += i;
```

```

18     return sum;
19 }
```

(9)

```

1 #include <iostream>
2 using namespace std;
3
4 int func();
5
6 int main()
7 {
8     sum = func();
9     cout << "Sum=" << sum << endl;
10    return 0;
11 }
12
13 int func()
14 {
15     int sum = 0;
16     for(int i=1; i<=100; i++)
17         sum += i;
18     return sum;
19 }
```

(10)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int choice;
7
8     while(choice)
9     {
10         cout << "=====主菜单====" << endl;
11         cout << "1---Hello" << endl;
12         cout << "2---Hi" << endl;
13         cout << "3---Ok" << endl;
14         cout << "0---Bye-bye" << endl;
15         cout << "======" << endl;
16         cout << "Your choice:" ;
17         cin >> choice;
18         switch(choice)
19         {
20             case 1: cout << "Hello." << endl;
21             case 2: cout << "Hi." << endl;
22             case 3: cout << "Ok." << endl;
23             case 0: cout << "Bye-bye!" << endl;
24         }
25     }
26     return 0;
27 }
```

2. 基本题

(1) 编写一个程序实现第 1 章中分析的 $1 + 2 + 3 + \dots + 36$ 迭加计算，并使累加计

算的中间结果都显示出来。

(2) 编写一个程序实现迭加计算 $1 + 2 + 3 + \dots + n$, 其中整型变量 n 的值用 `cin >> n;` 语句从键盘输入。执行输入语句前, 用 `cout << "Input n:";` 输出提示信息, 不要显示中间结果。

(3) 第 2.2.3 小节介绍的 `if` 语句嵌套实现多路分支例题中, 为什么只需要将条件表达式简单地写成关系运算(`score>=80`), 而不必要写成(`score>=80 && score<90`)?

3. 扩展题 (程序设计竞赛题型)

下面的两个题目按程序设计竞赛的常见题型给出。这种题型中包含“问题描述”“输入说明”“输出说明”“输入样例”和“输出样例”。参赛者通过网络将编写的源程序提交到竞赛判题系统进行自动评判。自动判题系统中储备了一定数量的测试数据(与题目中样例的格式相同, 常特别增加一些非常特殊的测试用例, 考查编程者是否考虑全面)用于判断参赛者的程序运行结果是否正确。

下面的第 1 题给出完整的解答以及递交前的自我测试方法。

(1) 计算两个整数的和。

问题描述 对于给定的两个整数, 要求计算它们的和。

输入说明 输入数据有多行, 每一行有两个整数。

输出说明 对于每一行的两个数据, 要求先输出“Case 序号:”(序号从 1 起), 然后输出算式及结果。

输入样例

```
15 3
42 89
51 201
303 -99
755 800
```

输出样例

```
Case 1: 15 + 3 = 18
Case 2: 42 + 89 = 131
Case 3: 51 + 201 = 252
Case 4: 303 - 99 = 204
Case 5: 755 + 800 = 1555
```

【解答】

源代码 2.6 简单的竞赛题型程序

```
1 // addition.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int x, y, k=0;
8     while(cin >> x >> y)
9     {
10         cout << "Case " << ++k << ":" << x;
11         if(y>=0)
```

```

12         cout << "□+□" << y;
13     else
14         cout << "□-□" << -y;
15     cout << "□=□" << x+y << endl;
16 }
17 return 0;
18 }
```

说明：

① 当读完输入数据文件中的数据后，再执行 `cin >> x >> y` 时将读到文件结束标志，此时返回 NULL（即 0，亦即 false “假”），导致第 8 行继续循环条件不成立而结束循环，进而执行第 17 行使程序结束。

② 从输出样例看，对输出格式是有要求的。当第二个加数为负数时，需要特殊处理。

【递交前的自测方法之一】 利用 C++ 集成开发环境将上述程序编译、连接、生成可执行文件。设可执行文件在文件夹 D:\CPP\addition\Debug 中，文件名为 `addition.exe`。现用 Windows “记事本” 建立如下两个文件。

① 输入数据文件。文件名为 `input.txt`，内容同输入样例。

```

15 3
42 89
51 201
303 -99
755 800
```

② 批处理文件。文件名为 `run.bat`（注意将文件的扩展名改为 `.bat`），内容为

```
addition < input.txt > output.txt
```

其一般格式如下。

```
可执行文件名 < 输入数据文件名 > 输出数据文件名
```

执行批处理文件（双击文件 `run.bat` 的图标），将启动可执行文件 `addition.exe` 并将标准输入设备重定向到文件 `input.txt`（即输入设备不是键盘，而是该文件）。同时将标准输出设备重定向到文件 `output.txt`（即输出设备不是显示器，而是该文件）。若文件 `output.txt` 不存在则产生该文件，否则将覆盖该文件原有的内容。这样可使输入与输出分隔开来。

若所产生的 `output.txt` 文件内容与输出样例一致，则可初步认为程序正确，可考虑将源程序递交到自动判题系统。

【递交前的自测方法之二】 直接运行程序，按输入样例的格式从键盘依次输入测试数据，程序随即输出计算结果（请与输出样例比较。不必顾虑输入与输出交织在一起）。最后按 `Ctrl+Z` 组合键（适用于 Windows 系统。UNIX/Linux 系统中为 `Ctrl+D` 组合键）表示测试数据输入完毕，此时表达式 (`cin >> x >> y`) 为 NULL 而结束循环，进而结束程序运行。若测试结果正确，可考虑将源程序递交到自动判题系统。

(2) 计算一系列整数的和。

问题描述 对于给定的一系列整数，要求计算它们的和。

输入说明 输入数据文件有多行，每行上可能有 0 个、一个或多个整数，直到文件结束。

输出说明 输出数据的个数、逗号、空格、总和。

输入样例

```
15    42
```

```
51  
303  
755
```

输出样例

```
5, 1166
```

第3章 数据的表示及I/O流格式控制

在本章中，将进一步介绍C++的运算，介绍数据的表示、数据的访问、函数定义、I/O流的格式控制等语法。最后给出一些具有一定功能的程序模块，希望读者精读它们并将它们有机地组织起来进行调试，以提高阅读程序和调试程序的能力。阅读程序的方法是跟踪变量的变化，以及外部设备（如键盘、显示器）上的反应。

3.1 数据的表示

3.1.1 常量

规定 3.1 (常量) 常量是指其值在程序的运行中不允许被改变的量。常量有字面常量、符号常量两种。

1. 字面常量

(1) 字符常量：用单引号包围起来的一个字符。字符在内存中按其ASCII码（整数）存放，占`sizeof(char)`（一个字节）。如'a'，'A'，'?'，'\$'等；一些特殊的字符用转义字符表示（如表3-1所示）；所有字符都可用十进制形式、八进制形式和十六进制形式表示，还可以利用类型转换运算符表示（见表3-1最后三行）。

表3-1 字符常量（转义字符）表

字符形式	十六进制值	意 义
\a	0x07	响铃
\b	0x08	退格
\t	0x09	制表符（横向跳至下一制表站）
\n	0x0A	换行
\v	0x0B	纵向跳行
\r	0x0D	回车
\"	0x22	双引号
\'	0x27	单引号
\\	0x5C	反斜杠“\”
\ddd	0x00 ~ 0xFF	ddd为1~3位八进制数
\hhh	0x00 ~ 0xFF	hh为1~2位十六进制数
char(ddd)或(char)ddd	0x00 ~ 0xFF	ddd为从0~255之间的整数

(2) 字符串常量：用双引号包围起来的0个、一个或多个字符。如""、"□"、"A"、"A\tB\tC\n"、"程序设计"、"双引号\" 和 "单引号\'"等（注意其中的转义字符）。

字符串是程序中常用的一种数据类型（数据结构），它不是基本数据类型，而是内存中连续存放的一系列字符的集合。C++语言有两种字符串：C-String（被称为C字符串，它从C语言沿袭而来）；class string（字符串类，这种字符串将在面向对象程序设计部分讨论）。

(3) 整型常量：可用十进制形式表示（如123, -123）、十六进制形式表示（以0x开头，如0x7B, -0x7B）和八进制形式（以零0开头，如0173, -0173）。以L（或l）为后缀表示长整型数据，以U（或u）表示无符号数据。例如，123L表示长整型数据，123U表示无符号整型数据，123LU或123UL表示无符号长整型数据。

(4) 浮点型（实型）常量：只能用十进制表示。可用小数形式或指数形式表示。如123.45、-123.45、1.2345E+2（表示 1.2345×10^2 ）、12345E-2（表示 12345×10^{-2} ）。其中的E可用小写字母e。E（或e）前面的数字不能缺省。浮点数常量默认为double型。以F（或f）为后缀表示单精度浮点型（float）数据，如123.45f。

(5) 布尔常量：也称为逻辑常量。只有true（表示“真”）和false（表示“假”）两个值。将数值（字符型、整型、浮点型，带符号或无符号）型数据视为逻辑值时，非零表示“真”，零表示“假”。系统给出的逻辑值“真”用1表示，“假”用0表示。例如：while(true)与while(1)、while(0.5)、while(!0)、while(!false)皆是等效的。while(x!=0)与while(x)是等效的。while(x==0)与while(!x)是等效的。

2. 符号常量

程序员在程序中通过如下语句定义自行命名的标识符为常量（其中const为保留字）。

```
const 数据类型 常量名 = 初始化值;
```

例如定义圆周率π、定义MAX_NUMBER为10000、定义LF为换行字符如下。

```
const double PI = 3.1415926;
const unsigned long MAX_NUMBER = 10000;
const char LF = '\n';
```

符号常量在定义时必须对其初始化，这里的“=”为初始化记号，不是赋值运算符。由于常量的值不允许被改变，因此常量不能做左值，试图对其赋值是错误的。

3.1.2 变量

规定3.2（变量） 在程序的运行中其值可以被改变的量称为**变量**。

变量是程序中最活跃的基本元素。它既联系到计算机内存单元，又承载着所计算的问题。更重要的是，变量值的变化过程还标志着问题求解的进展。完整的变量定义语句为

```
存储类型 数据类型 变量名 = 初始化值;
```

值得注意的是：这里的“=”为初始化记号，不是赋值运算符。定义变量时可以不对变量进行显式的初始化，而采用系统默认的初始化处理。

程序中变量必须先定义，后使用。定义变量意味着为变量分配内存空间。定义变量需要确定其如下五要素。

- ① 存储类型。
- ② 数据类型。
- ③ 变量名。
- ④ 变量的初始值。
- ⑤ 变量的地址。

其中变量的地址由系统确定，程序中可以通过取变量地址运算操作获取变量的地址值。其余的四个要素皆由程序员根据需要自行确定。

在一个程序中可以使用许许多多存储在不同内存区域（存储类型不同）、不同存放格式（数据类型不同）的变量。本质上，程序中的所有变量都是相互独立的。对变量的读或写操作称为对该变量的访问。变量以右值参与某一表达式运算时，只读取其数值，不会影响该变量本身。变量以左值参与运算时（如赋值、迭代赋值、抽取、自增和自减），可能会改变其值。一些变量参与某表达式运算，并不能建立它们之间的长久关系。变量之间的相关关系是算法逻辑上的，是由程序员努力控制的。

1. 变量的存储类型

一个程序在操作系统的控制下启动执行时，由操作系统分配给该程序一定的内存空间，同时允许程序申请使用堆空间。在逻辑上，这些空间可分成两大部分：代码区（用于存放程序代码，按照函数的入口地址——函数名来使用代码区）和数据区（用于存放数据）。数据区又细分为三块：全局数据区（存放全局变量、一些常量）；栈区（函数调用时存放局部自动变量，函数执行完毕返回时清除）；堆区（程序运行中根据需要可申请使用的内存空间，不需要继续使用它们时予以释放。因此，又称堆区为动态数据区）。一般地，堆区由操作系统直接掌管，其空间容量非常大，而全局数据区、栈空间的容量相对较小。程序可使用的内存空间如图 3-1 所示。



图 3-1 程序可使用的内存空间

堆、栈分别是某种数据结构，是数据（如变量）的容器。它们对存储在其中的数据元

素有特殊的处理方式，适合不同的应用场合。栈的结构特点是数据先入后出——最先入栈的数据将被压在栈底，最后才能被弹出。函数调用时，局部变量（包括非引用型的形式参数）的创建和销毁经历栈操作处理（参见第6.2节）。

变量的存储类型决定了该变量在内存中的存储区域，决定了该变量的生命期（何时产生、何时销毁）和可见性（何种情形下可访问）。

规定3.3（全局变量^[1]） 在所有函数之外定义的变量被称为全局变量，定义全局变量时无存储类型名。全局变量存放在全局数据区中。在定义全局变量时若未显式地进行初始化，则系统将其初始化为各位（bit）全0。

全局变量的生命期是全局的。全局变量在程序启动后，主函数执行之前创建，主函数结束后销毁。在多文件结构的程序中，全局变量应该在某一个编译单元（.cpp源程序文件）中定义，在其他需要访问该全局变量的编译单元中用如下方式进行外部参照访问声明：

```
extern 数据类型 某全局变量名;
```

一般地，该语句只是声明在本编译单元中需要访问指定的全局变量，而不一定是定义变量，故一般不能对这种声明进行初始化。这种声明最好编写到头文件（.h）中，以便通过文件包含预处理操作插入到不同的编译单元中，使得相应的编译单元能正确地通过独立的分割编译，实现所谓的一处定义，多处声明。

全局变量的作用域从该变量被定义或外部参照访问声明之处起，直至本编译单元结束。利用外部参照访问声明，可以跨编译单元使用某个全局变量。亦即全局变量的作用域是可以跨编译单元的。在其作用域范围内的所有函数体内均可以直接访问该全局变量。

规定3.4（静态全局变量） 在所有函数之外定义的存储类型为static的变量被称为静态全局变量。

静态全局变量与全局变量的唯一区别：静态全局变量的作用域和可见性为其所在的编译单元（不能跨越编译单元）。

规定3.5（静态局部变量） 在某函数内定义的存储类型为static的变量被称为静态局部变量。

静态局部变量在其所在函数第一次被调用时创建，存放在全局数据区。直到整个程序结束时，静态局部变量才被销毁。在定义静态局部变量时若未显式地进行初始化，则系统将其初始化为各位（bit）全0。从定义静态局部变量的函数返回后，该变量处于“休眠”状态，仍保留其所占用的空间，保存其数值，任何其他函数都难以访问该变量。当再次调用其所在的函数时，静态局部变量被“唤醒”（跳过该变量的定义，也不再进行初始化，即静态局部变量最多只被定义及初始化一次）。静态局部变量具有全局生命期、局部可见性。

规定3.6（局部自动变量） 在某函数内定义的存储类型为auto的变量被称为局部自动变量（简称为局部变量），其中保留字auto可被省略。

^[1]此处我们撇开名字空间的概念和用法讨论变量。有关名字空间namespace的内容见第7.3节。

局部变量的生命期随其所在函数的调用而产生，存放在栈空间中，随其所在函数的返回而结束。定义局部变量时若未显式地对其初始化，则其初值是不可预知的。因而，未显式初始化的局部自动变量不宜直接将其作为右值使用。

变量的存储类型能够很好地解决变量名之间的冲突问题^[1]。局部变量的作用域范围只在其函数内部，不同函数中定义的局部变量不会引起任何冲突。即使函数体内调用了本函数自己——递归函数（参见第 6.4 节），其不同层次函数调用的局部自动变量也是相互独立的，不会引起任何冲突。全局变量可以直接在一些函数中使用，给函数定义带来一点点方便，但是它破坏了函数的相对独立性、不利于函数代码的移植。因此，我们不提倡使用全局变量。事实上，C++ 语言的一些机制可以彻底消灭程序中的全局变量。

全局变量可能被局部变量屏蔽。此时需使用作用域区分符 (::) 指示全局变量。

例 3.1 各种存储类型的变量使用举例（多文件结构）。

源代码 3.1 变量的存储类型

```

1 // VarExample.h 头文件 (编写声明语句)
2 #ifndef VAREXAMPLE_H
3 #define VAREXAMPLE_H
4
5 extern int n;           // 全局变量外部参照访问声明，并非定义变量
6 void function1();       // 函数原型，用于函数声明
7 void function2();
8 void function3();
9
10#endif

```

```

1 // Variable_Test.cpp 源程序文件 (编写定义语句)
2 #include <iostream>
3 #include "VarExample.h"          // 声明全局变量 n 和三个函数
4 using namespace std;
5
6 static float a=800.8f;          // 定义静态全局变量，分配内存空间
7
8 int main()
9 {
10     int a=10000, x=500, n=800;  // 定义局部自动变量，分配内存空间
11
12     function1();
13     function2();
14     function3();
15     function1();
16     function2();
17     function3();
18     cout << "in main() n:" << n      // 访问全局变量
19     << ", a=" << a                // 访问局部自动变量
20     << ", a=" << a                // 访问静态全局变量
21     << ", a=" << a                // 访问局部自动变量
22     << ", x=" << x << endl;      // 访问局部自动变量
23
24 }
```

^[1]众人集体开发软件中变量名冲突问题的彻底解决方案是使用名字空间。参见第 7.3 节。

```

1 // VarExam1.cpp 源程序文件 (编写定义语句)
2 #include <iostream>
3 #include "VarExample.h"    // 将文件 VarExample.h 的内容插入至此
4 using namespace std;
5
6 int n;           // 定义全局变量, 分配内存空间, 默认初始化为 0
7 static int a=10000; // 定义静态全局变量, 分配内存空间, 初始化为 10000
8
9 void function1()
10 {
11     static int a;      // 定义静态局部变量 (最多只执行一次)
12     double x=5.5;    // 随函数每次调用定义局部自动变量
13     int n=100;        // 随函数每次调用定义局部自动变量
14     x++;             // 访问局部变量
15     ::n++;           // 访问全局变量
16     ::a++;           // 访问静态全局变量
17     a = a + ::a + (int)x + ::n + n;
18     cout << "in function1(): n=" << ::n
19         << ", a=" << n           // 访问局部自动变量
20         << ", A=" << ::a
21         << ", a=" << a           // 访问静态局部变量
22         << ", x=" << x << endl;
23 }

```

```

1 // VarExam2.cpp 源程序文件 (编写定义语句)
2 #include <iostream>
3 #include "VarExample.h"    // 将文件 VarExample.h 的内容插入至此
4 using namespace std;
5
6 static double a=200.5;       // 定义一个静态全局变量
7
8 void function2()
9 {
10     static double a=2.71828;   // 定义一个静态局部变量
11     int x = 200;
12     a += ++n + ++x;          // 等效于 ++n; ++x; a=a+n+x;
13     cout << "in function2(): n=" << n   // 全局变量 n
14         << ", x=" << x           // 局部变量 x
15         << ", A=" << ::a        // 静态全局变量 a
16         << ", a=" << a           // 静态局部变量 a
17 }
18
19 void function3()
20 {
21     double a=3.1415926;
22     static char x='C';
23     int n = 300;
24     ::a++;
25     x++;
26     a = a + ::a + n + ::n + x;
27     cout << "in function3(): n=" << ::n // 全局变量 n
28         << ", x=" << n           // 局部变量 n
29         << ", A=" << ::a        // 静态全局变量 a
30         << ", a=" << a           // 局部变量 a
31         << ", x=" << x           // 静态局部变量 x
32 }

```

程序的运行结果如下。

```
in function1(): N = 1, n = 100, A = 10001, a = 10108, x = 6.5
in function2(): N = 2, x = 201, A = 200.5, a = 205.718
in function3(): N = 2, n = 300, A = 201.5, a = 574.642, x = D
in function1(): N = 3, n = 100, A = 10002, a = 20219, x = 6.5
in function2(): N = 4, x = 201, A = 201.5, a = 410.718
in function3(): N = 4, n = 300, A = 202.5, a = 578.642, x = E
in main()      : N = 4, n = 800, A = 800.8, a = 10000, x = 500
```

从运行结果可见，各种变量（尽管有的名字相同，甚至有同时的可见性）也未引起冲突。

2. 变量的数据类型

变量的数据类型决定该变量的数据在内存中的存放格式：占用的字节数、取值范围。定义变量时所规定的数据类型是不会改变的。对变量使用类型转换操作时，只是将该变量的值（作为右值）读出，建立并初始化一个另一种数据类型的临时变量，用该临时变量参与表达式运算。这一操作并不影响变量原来的数据类型及其数值。

3. 变量名

变量的命名应符合自定义标识符的命名规则。变量名可视为相应内存单元的名字，是程序中访问数据的依据之一。在变量（原名）的可见范围内访问它是简单的；在变量的生命周期内，但在变量的原名不可见处访问该变量，则需要借助于变量的别名（参见第 3.1.3 小节）、或变量的间接名（参见第 5.3.2 小节）。因此，一个变量名称可能以多种形式呈现。

4. 变量的值

变量定义时可以对其进行初始化，使其在创建时便具有对问题求解有意义的初始值。变量在其整个生命期内，值的演变过程代表着问题求解的历程。程序运行过程中，变量的值的变化规律，尤其是变量的当前值是值得特别关注的，是阅读理解程序的关键之所在。

5. 变量的地址

变量所在的存储区域（参见图 3-1）由其存储类型决定，变量在其存储区域中的具体地址由系统安排。C++ 语言提供了获取变量具体地址值的运算操作。通常情况下，我们并不关心变量在其存储区域中的具体地址，只需要能够获取变量的地址值并能够根据变量地址值“间接地”访问变量即可。与变量地址相关的概念和操作详见第 5 章。

3.1.3 变量的引用

变量的引用是 C++ 语言的一个重要特色。C 语言中没有变量引用的概念。

规定 3.7（变量的引用） 变量的引用是变量实体（一个已经存在的变量）的一个别名。

引用只是声明，不是定义，引用不占内存空间。声明引用时，必须用一个变量实体对其初始化。这样的“绑定”从引用声明开始，直到引用的生命期结束不能更改。对引用的访问就是对其“绑定”的变量的访问（包括读和写）。程序中应该保证实体变量的生命期包含其引用的生命期，否则“皮之不存，毛将焉附”。

声明引用的语法格式为

```
数据类型 & 引用名 = 已存在的变量名;
```

其中符号“`&`”是声明引用时的标记。作为标记的符号不是运算符。数据类型必须与被引用的变量实体的数据类型一致。这里的“`=`”为初始化记号，不是赋值运算符。

例如：

```
int x; // 定义一个整型变量，分配一定的内存空间
int &rx = x; // 声明一个引用，不占用新的内存空间
x = 100;
cout << rx << endl; // 将输出 100
rx = 200;
cout << x << endl; // 将输出 200
```

【变量的名称】 我们常说：算法是计算机科学的灵魂；变量是算法（程序）中的小精灵。这些小精灵们可以“招之即来，挥之即去”。这些多变的小精灵们，有时我们知道其变量名（原名），有时我们只知道其“绰号”（别名），有时我们仅知道其住址编号（间接名）。因而，访问变量的方法有通过变量名直接访问；或者通过变量的别名引用访问；或者通过变量的存储单元起始地址间接访问（见第5章）。

编写或阅读程序时，要正确地辨明那些五花八门的“间接名”“别名”所指代的真正变量实体。

3.1.4 常量的引用

C++ 中除了变量可以被引用外，常量也可以被引用。常量引用的声明如下。

```
const 数据类型 & 引用名 = 常量或已存在的变量;
```

其中符号“`&`”是声明引用时的标志，不是运算符；符号“`=`”为初始化记号，不是赋值运算符。

例如：

```
double x = 2.5;
const double PI = 3.1415926;
const double &rPI = 1/PI, &E = 2.17281728;
const double &rx = x; // 允许将变量视为常量
cout << rx << endl; // 输出 2.5
x++;
// rx++; // 错误，因为 rx 为常量，不能做左值
cout << rx << endl; // 输出 3.5，说明“绑定”
// double &r = PI; // 错误，不能将常量视为变量
```

可见，可以将变量作为常量使用，但不允许将常量作为变量看待。

— 变量引用的主要作用之一是实现变量在函数之间的双向传递。在函数体内通过别名访问原名不可见的变量。

3.2 函数

规定 3.8 (函数) 函数是程序按功能划分的基本单位，俗称为子程序。函数是对程序中某种功能的包装和抽象。

C++ 中的函数有三个方面的内容：

① **函数原型** 是编译器检查源程序中调用函数语句语法正确性的依据，亦是程序员编写调用函数语句的依据。

② **函数定义** 是指函数功能的具体实现过程，它被编译成目标代码后，是连接器连接诸目标代码生成可执行文件的依据。

③ **函数调用** 是指实际使用函数，必要时需提供待加工的实际数据（被称为实际参数，用于初始化形式参数）。函数通过其返回类型获得计算结果，并用函数调用表达式本身表示结果。

只要用函数原型声明了某个函数，即使该函数尚未定义，调用该函数的语句便可以通过编译。若该函数始终未定义，则连接时出错。编译系统所提供的标准函数的定义已经被编译成函数库，直接由连接器连接。其函数原型已经写在标准头文件中，用 `#include` 指令包含了头文件后，相应的函数就被声明了，程序中便可以直接调用它们了。

函数原型的一般格式如下（注意其中的分号“;”）。

```
返回类型 函数名(形式参数表);
```

函数必须先声明后调用，函数原型用于函数声明。当函数定义在前调用在后时，函数定义可以起到函数声明的作用。

我们将经常使用的、具有一定功能的、相对独立和完整的程序段组织成函数。函数定义完成，经由其原型声明后，便可以反复使用。调用函数时，只关心其功能和使用方法，不必关心其具体的实现细节。例如，计算非负实数算术平方根，数学函数为 $y = f(x) = \sqrt{x}$ 。C++ 提供了根据给定“自变量” x 计算这个数学函数值的标准函数，其原型在头文件 `cmath` 中声明为

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

设有变量 `double x=2.0, y;` 欲计算 x 的算术平方根，并将结果存入变量 y 中，用 C++ 的语句表示为

```
y = sqrt(x); // 不能写成 y = sqrt(double x);[1]
```

至于 `sqrt(x)` 函数是如何计算出“自变量”（C++ 中称为参数） x 的算术平方根的，程序员不必关心其中的细节（有兴趣的读者可参见第 1.3.2 小节的【附注】和第 4.2.1 小节）。程序员关心的是该函数的功能、函数的使用方法。函数的功能必须查阅相关的资料（反过来说明了程序员编写技术文档是重要的）；函数的使用方法从函数原型便可直接看出。

^[1] 写成 `y = sqrt(double x);` 是初学者容易犯的错误。这被理解成①定义变量 `x`；②执行 `y = sqrt();`；由于没有提供被开方数（即实际参数）而错。

C++用函数组织程序，通过函数调用驱动程序运行。函数有标准库函数、自定义函数。编写C++程序就是编写各种各样的函数，并将它们有机地组织起来。常用的标准库函数及声明其原型的头文件参见附录B。

第2章介绍的函数基本上是最简单的“不带参数的函数”。本节介绍带参数的函数。与数学中的一元函数、二元函数或多元函数类似，C++语言中也允许带一个参数、两个参数或多个参数的函数。各个参数的数据类型必须分别明确；函数可以通过返回类型返回计算结果。

本节主要介绍函数设计方面的语法规规定。侧重数值、变量本身传递给函数，加工计算后返回数值、返回变量的相关规定。

3.2.1 函数的形式参数

1. 值传递型参数——传递数值

例如，绝对值函数。其函数原型在头文件 `cmath` 中的声明如下。

```
double fabs(double x);
```

该函数的定义（已经被编译成库函数）为

```
double fabs(double x)
{
    if(x>=0)
        return x;
    else
        return -x;
}
```

函数的原型声明及函数定义中的 `double x` 为形式参数（简称形参）。从该形式参数的语法格式上看，属于变量定义。形式参数变量存放在栈区，其存储类型为 `auto`，因而其生命期是局部的：它在函数每次调用时定义（意味着为其分配内存空间），并用调用该函数时的一个实际数值（被称为实际参数，简称实参）对形式参数进行初始化，该形式参数变量在函数返回时被销毁。

这种形式的参数之所以被称为“值传递型”的，是因为每次调用该函数时，创建了一个新的形参变量，用实参（某变量的值、某常量的值、某表达式的值）的右值初始化之。该形参变量有其独立的存储空间，其生命期、可见性皆限于本次调用之中，最多可称为实参的一个副本。对形参（副本）的任何操作均不影响实参。

绝对值函数不需要程序员重新定义。若要使用它，只需包含头文件 `cmath` 即可。如：

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 int main()
6 {
7     double a=-5.5, b=-8.5;
8     cout << fabs(a) << endl           // 输出 5.5
9                  << fabs(b) << endl           // 输出 8.5
```

```

10      << fabs(a+b+3) << endl        // 输出 11
11      << fabs(-10.5) << endl;       // 输出 10.5
12      cout << a << '\t' << b << endl;   // 输出 -5.5 -8.5
13      return 0;
14  }

```

显然，主函数中调用绝对值函数时使用了实际参数。对于值传递型的形式参数，实际参数体现的是实参的右值特性。本例中，只读取了变量 `a`, `b` 的值作为实际参数调用函数。实际参数还可以是一个运算表达式的值，或者为一个常量值。这些右值分别用于 4 次调用 `fabs(x)` 函数时定义形式参数变量 `x` 的初始化，亦即形式参数 `x` 经历了 4 次“创建—销毁”的生命历程。在形式参数每次的生命周期内，形式参数都独立于实际参数，因而对形参的操作不会影响到实参。因而，函数中的值传递型参数是单向的，即单向向函数传递数值。程序最后的输出结果表明变量 `a`, `b` 的值未被修改。

2. 引用型参数——传递变量

将函数的形式参数设计为引用（即引用型参数）是引用的主要用途。用已经存在的变量作为实际参数调用函数时，便是声明了对实参变量的引用。即用已经存在的变量初始化引用型形式参数的声明。引用型形式参数不占用内存空间，使形式参数成为实际变量参数的别名。函数体内对形式参数的访问实质上就是对具有生命期、但不具备可见性的实参变量的访问。因而，引用型参数在函数间进行的数据传递实质上是变量传递（当然，变量的 5 要素全部传入），可形成双向的效果：实际参数带值进入函数，函数体内可通过实际参数的别名（即形参）对实参进行读、写操作。即通过引用型参数，还可以“返回”计算结果，尤其是实现要求“返回”多个计算结果的任务（此处的所谓“返回”不是指函数返回类型的返回）。

引用型形参（即别名）的生命周期起始于所在函数被调用时，形参“绑定”实参变量，这种绑定随其所在函数返回时，即形参生命周期结束而自动解除。函数再次被调用时，形参又可以再次“绑定”同一变量，或者另一变量。

另外，将形式参数声明为引用型时，该形式参数不占用内存空间，也不存在大数据体的传递，故执行效率高。须注意的是此时的实际参数应该是能够被引用的量，是一个变量实体。

例 3.2 设计自定义函数求解一元二次方程 $ax^2 + bx + c = 0$ ($a \neq 0$)，并测试之。

【分析】 该问题涉及 3 个系数 (`a`, `b`, `c`) 单向传入给处理函数，从函数“返回”获得 2 个可能的根 (`x1`, `x2`，它们传入给函数的值并不重要)。当方程无实数根时，`x1`, `x2` 的值是无意义的（它们总是有数值的）。为了区分这种情况，还需要设计一个量表示方程的类型。故函数需要“返回”或返回标志方程类型的标志值。为此，我们约定标志值为 0、1 和 2 分别表示方程无实数根、方程有重根和方程有两个不同的根。故本例有两种设计方案如下。

源代码 3.2 求解一元二次方程的函数

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;

```

```

4
5 void Solver(double a,double b,double c, double &x1,double &x2,int &flag)
6 { // 函数定义(6个参数, 其中3个值传递、3个变量传递; 函数无返回类型)
7     double d = b*b - 4*a*c;
8
9     if(d<0)
10    {
11        flag = 0;
12        return;
13    }
14    d = sqrt(d);
15    if(a>0)
16    {
17        x1 = (-b - d)/(2*a);
18        x2 = (-b + d)/(2*a);
19    }
20    else
21    {
22        x1 = (-b + d)/(2*a);
23        x2 = (-b - d)/(2*a);
24    }
25    if(d>0)
26        flag = 2;
27    else
28        flag = 1;
29 }
30
31 int Solver(double a, double b, double c, double &x1, double &x2)
32 { // 函数定义(5个参数, 其中3个值传递、2个变量传递; 函数返回方程类型标志值)
33     int flag; // 定义局部自动变量
34     double d = b*b - 4*a*c;
35
36     if(d<0) return 0;
37     d = sqrt(d);
38     if(a>0)
39     {
40         x1 = (-b - d)/(2*a);
41         x2 = (-b + d)/(2*a);
42     }
43     else
44     {
45         x1 = (-b + d)/(2*a);
46         x2 = (-b - d)/(2*a);
47     }
48     if(d>0)
49         flag = 2;
50     else
51         flag = 1;
52     return flag;
53 }
54
55 void Display(int flag, double x1, double x2)
56 {
57     switch(flag)
58     {
59     case 1: cout << "该方程有重根:x1=x2=" << x1; break;
60     case 2: cout << "该方程的根为:x1=" << x1
61             << ",x2=" << x2; break;
62     default: cout << "该方程无实根。"; break;
}

```

```

63     }
64     cout << endl;
65 }
66
67 int main()
68 {
69     double a, b, c, x1, x2;           // 定义五个 double 型变量
70     int flag;
71
72     flag = Solver(1, -2, 1, x1, x2);    // 调用 5 个参数的函数
73     Display(flag, x1, x2);
74     Solver(1, 2, -3, x1, x2, flag);    // 调用 6 个参数的函数
75     Display(flag, x1, x2);
76
77     a = 1; b = 2; c = 3;
78     flag = Solver(a, b, c, x1, x2);
79     Display(flag, x1, x2);
80     Solver(a, b, c, x1, x2, flag);
81     Display(flag, x1, x2);
82     return 0;
83 }
```

调用 `Solver` 函数时，前三个实际参数可以是任何表达式（使用其右值特性）；后面的参数必须是可以被引用的量（必须是变量实体，能作为左值的量）。

因为引用不占用内存空间，所以利用函数的引用型参数传递实参数据具有较好的空间利用率及时间效率。另一方面，函数能获得对实参变量的访问权，使函数的作用范围扩大到本函数以外（这既有有利的方面，又有不利的方面）。使用引用型参数后，实参处于可能被函数修改的“危险境地”。有时我们不得不使用引用型参数（如面向对象程序设计部分的拷贝构造函数的形式参数），同时又不希望函数体对实参进行修改，此时可以用保留字 `const` 对引用型形式参数加以限制，成为常量引用以达到：

① 使函数体内不能修改实参的值（使程序员敢于大胆地调用该函数，因为从函数原型可知，传递给函数的实参值不会被修改）。

② 可以用真正的常量作为实参初始化常量的引用。

例如，可将上述函数做如下修改。

```

1 int Solver(const double &a, const double &b, const double &c,
2             double &x1, double &x2)
3 {
4     int flag;                      // 定义局部自动变量
5     double d = b*b - 4*a*c;
6
7     if(d<0) return 0;
8     d = sqrt(d);
9     if(a>0)
10    {
11        x1 = (-b - d)/(2*a);
12        x2 = (-b + d)/(2*a);
13    }
14    else
15    {
16        x1 = (-b + d)/(2*a);
17        x2 = (-b - d)/(2*a);
18    }
```

```

19     if(d>0)
20         flag = 2;
21     else
22         flag = 1;
23     return flag;
24 }
```

主函数中仍然可以使用语句 `flag = Solver(1, 2, -3, x1, x2);` 和 `flag = Solver(a, b, c, x1, x2);` 调用函数，甚至可以使用语句 `flag = Solver(a+1, -b, 10*c, x1, x2);` 调用函数。

综上所述，形式参数在函数的每次调用时均定义（值传递型，分配内存空间）或声明（引用型传递，引用不占空间），并用实际参数对其进行初始化。函数形式参数的存储类型为 `auto`，其定义或声明及其初始化完全遵循局部变量定义、局部引用声明及其初始化的语法规规定。我们将值传递型的参数称为 **传递数值**（函数传递数值型的形式参数是新定义的变量，其值与实参相同，因此常称此时的形参为实参的一个副本，函数体内只操作该副本而与实参无关，体现数据单向传入给函数）；将引用型参数称为 **传递变量本身**（函数体内能操作实参变量，体现变量的双向传递）。

例 3.3 交换两个实参的值。这是一个经典例子。

源代码 3.3 交换两个实参的值

```

1 // swap.cpp
2 void swap(int &a, int &b)           // 正确的函数，传递实参变量本身
3 {
4     int temp;
5     temp = a; a = b; b = temp;
6 }
7 void swap1(int a, int b)           // 无用的函数，仅操作实参的副本
8 {
9     int temp;
10    temp = a; a = b; b = temp;
11 }
12
13 #include <iostream>
14 using namespace std;
15
16 int main()
17 {
18     int x=3, y=5;
19     cout << "x=" << x << ", y=" << y << endl;
20     swap1(x, y);      // 实参值传递
21     cout << "x=" << x << ", y=" << y << endl;
22     swap(x, y);      // 实参引用传递
23     cout << "x=" << x << ", y=" << y << endl;
24
25 }
```

程序的运行结果：

```
x = 3, y = 5
x = 3, y = 5
x = 5, y = 3
```

可见，函数 `void swap1(int a, int b);` 是一个无用的函数。其形式参数 `a, b` 分别是实参 `x, y` 的副本（复制品），函数体中对 `a, b` 的操作与 `x, y` 无关。函数 `void swap(int &a, int &b);` 中的形式参数 `a, b` 分别是实参 `x, y` 的引用（别名），函数体中对 `a, b` 的操作就是对实参 `x, y` 的操作。

3.2.2 函数的返回类型

函数原型、函数定义中需要指明函数的返回类型。函数的返回类型有无返回、值返回、引用返回（变量返回）。函数返回类型的所有形式详见第 6 章。通过引用型形式参数双向传递的数据被称为所谓的“返回”（打引号的“返回”），本质上不属于函数的返回类型。

函数通过返回类型只能返回某种类型的一个量，这个量就用函数调用表达式本身表示。

1. 无返回

返回类型被指定为 `void` 者为无返回函数。这种函数通常只完成某个动作而不返回任何量。当然，这种函数仍能通过引用型形式参数“返回”多个量。

```

1 void greeting()
2 {
3     cout << "Hello." << endl;
4     return ;           // 此语句可省略
5 }
```

2. 值返回（返回临时无名变量）

函数值返回是最常见的一种形式。这种函数的返回类型为某种数据类型。如：函数原型在头文件 `cmath` 中的部分标准函数如表 3-2 所示。

表 3-2 头文件 `cmath` 中声明的部分函数

函数原型	功 能
<code>int abs(int x);</code>	绝对值函数
<code>long labs(long x);</code>	绝对值函数
<code>double fabs(double x);</code>	绝对值函数
<code>double sqrt(double x);</code>	算术平方根函数（要求 x 非负）
<code>double pow(double x, double y);</code>	x 的 y 次幂函数（要求 x 非负）
<code>double exp(double x);</code>	e 的 x 次幂函数
<code>double sin(double x);</code>	正弦函数

调用数值返回的函数返回时，将创建一个与返回数据类型相同的临时变量^[1]，用于存放函数的返回值；该临时变量是无名的，用函数调用表达式本身表示；该临时变量的生命期是短时的，不宜或不能将其作为左值使用。例如：

^[1]这是标准 C++ 的规定。有些编译系统的一些编译优化选项可能对可执行代码进行“优化”而不一定创建临时变量。

```
double x=2, y;
y = sqrt(x); // 函数返回时创建临时变量，存放返回值，再将临时变量的值赋给 y
y = sqrt(x+2); // 临时变量是无名的，直接用函数调用表达式本身表示
```

例 3.4 加法运算函数。

```
1 #include <iostream>
2 using namespace std;
3
4 int Add(int a, int b)           // 模拟“+”运算，值返回
5 {
6     a += b;                   // 故意改变了形参 a 的值
7     return a;                 // a 的生命期仅存在于本函数被调用时
8 }
9
10 int main()
11 {
12     int a = 3, b;             // Add(a, 2) 表示一个临时变量
13     b = Add(a, 2);           // cout << a << ", " << b << endl; // 输出 3, 5 (注意 a 的值不变)
14     cout << a << ", " << b << endl; // 输出 3, 5 (注意 a 的值不变)
15     return 0;
16 }
```

其中 Add 函数的定义本应该写成如下更加清晰的形式（写成上述形式的目的是请读者比较后面的 AddAssign 函数）。

```
int Add(int a, int b) // 模拟“+”运算
{
    return a+b;        // 与上面的函数等效
}
```

3. 引用返回（返回变量）

函数的返回类型可以被设计成某种数据类型的引用。与函数值返回不同，函数引用返回不创建临时变量，直接返回变量本身，使函数调用表达式成为左值。因此，引用返回也被称为变量返回。引用返回的变量仍然用函数调用表达式表示。这样，该函数调用表达式本身就成为了该变量“五花八门”的变量名的表现形式之一。自然要求引用返回的变量是可以被引用的，其生命期和作用域应该包含该函数调用语句所在的表达式。

例 3.5 迭加计算函数。

```
1 #include <iostream>
2 using namespace std;
3
4 int & AddAssign(int &a, int b) // 模拟“+=”运算，引用返回
5 {                           // a 所“绑定”的变量的生命期长于本函数的执行期
6     a += b;
7     return a;                // 返回一个生命期长于本函数执行期的变量
8 }
9
10 int main()
11 {
12     int a = 3;
```

```

13     AddAssign(a, 2);      // a 的值被修改，增加 2
14     cout << a << endl;    // 输出 5
15     AddAssign(a, 2)++;    // 函数调用表达式成为左值，此处即为变量 a
16     cout << a << endl;    // 输出 8
17     return 0;
18 }
```

例 3.6 返回静态局部变量。

【说明】本例仅用于说明这样的函数在语法上是可行的。但不提倡这样的函数，因为这种函数中的静态局部变量“局部可见性”的特性丧失了。注意：整个程序中仅定义了一个变量（静态局部变量 a），主函数中没有定义任何变量，对变量 a 是不可见的（即不知其原名）。主函数中的 ra 和函数调用表达式 func() 均是变量 a “五花八门”的别名。

```

1 #include <iostream>
2 using namespace std;
3
4 int & func()
5 {
6     static int a;      // 静态局部变量 a 的生命期是全局的
7     return a;          // 返回一个生命期长于本函数执行期的变量
8 }
9
10 int main()
11 {
12     int &ra = func();      // ra 为 func 函数中静态局部变量的别名
13     cout << func() << endl; // 输出 0，静态局部变量默认初始化为 0
14     func() += 8;
15     cout << func() << endl; // 输出 8
16     ra += 10;            // 变量 a 丧失“局部可见性”
17     cout << func() << endl; // 输出 18
18     return 0;
19 }
```

3.3 运算表达式

在第 2 章中介绍了一些基本的运算符和基本的运算表达式。主要涉及类型转换运算、算术运算、关系运算和逻辑运算。本节继续介绍一些运算符及运算表达式。

3.3.1 C++ 运算符汇总

表 3-3 列出了 C++ 的各种运算符。由表可见，C++ 语言提供了丰富的运算操作符，运算表达能力强。这些运算符按运算优先级从高到低（从第 1 级到第 18 级）共分成 18 级。在 C++ 程序中，可以将多种运算符组织到一个表达式中。在一个表达式中，优先级高者先计算，优先级低者后计算；优先级相同者根据其结合方向（从左到右或从右到左）依次进行。

需要指出的是，有些运算符由多个字符组成，使用时应将其作为一个整体，不能分开。运算符中出现圆括号、方括号时，其中的左、右括号应分开；唯一的三目运算符中的两个字符用于分隔三个操作数。

表 3-3 C++ 运算符汇总表

优先级	运算符	结合方向	描述
1	::	左→右	作用域区分符
2	.		对象取成员
	->		指针取目标对象成员
	[]	左→右	访问数组元素
	()		组织运行次序
	++		后置自增 1
	--		后置自减 1
3 (单目运算)	++		前置自增 1
	--		前置自减 1
	~		二进制位反(波浪号)
	!		逻辑非
	+		正号
	-		负号
	*		指针目标内容
	&		取变量地址
	sizeof		数据类型的尺寸(以字节为单位)
	new		在堆内存中申请一片可写的空间并返回其首地址
	delete		释放由new申请的堆内存空间
	(类型名)		显式(强制)转换数据类型, 或类型名(表达式)
4	* ->*	左→右	直接访问指向对象成员的指针(参见8.1.3小节) 间接访问指向对象成员的指针(参见8.1.3小节)
5 (算术运算)	*		乘法
	/		除法
	%		整数相除取余
6 (算术运算)	+	左→右	加法, 还用于指针加减整数(参见5.3.2小节)
	-		减法, 还用于两个同类指针相减(参见5.3.2小节)
7 (位运算)	<< >>	左→右	整数按其二进制位左移, 还用做输出(插入运算) 整数按其二进制位右移, 还用做输入(抽取运算)
8 (关系运算)	< <=	左→右	小于 小于等于
	>		大于
	>=		大于等于
9 (关系运算)	==	左→右	等于
	!=		不等于

续表

优先级	运算符	结合方向	描述
10 (位运算)	&	左→右	整型数按其二进制位进行与运算
11 (位运算)	~	左→右	整型数按其二进制位进行异或运算(尖帽子)
12 (位运算)		左→右	整型数按其二进制位进行或运算
13 (逻辑运算)	&&	左→右	逻辑与(并且)
14 (逻辑运算)		左→右	逻辑或(或者)
15 (三目运算)	? :	左←右	三目条件运算
16 (赋值运算)	= += -= *= /= %= &= ^= = <<= >>=	左←右	赋值运算 迭代加 迭代减 迭代乘 迭代除 整型量迭代除取余 迭代按位与 迭代按位异或 迭代按位或 迭代按位左移 迭代按位右移
17	throw	左→右	抛掷异常
18	,	左→右	逗号运算

3.3.2 单目运算

优先级为3的是单目运算，正号“+”、负号“-”运算符与数学符号的含义相同。自增“++”、自减“--”、逻辑非“!”、类型转换运算符“(类型名)表达式”或者“类型名(表达式)”见第2章。

运算符*、&、new和delete的用法涉及指针的概念，将在第5章介绍。

变量的尺寸(以字节为单位)：sizeof(数据类型名)或sizeof(变量名)。如：

```
double x;
cout << sizeof(int) << endl;      // 输出 4
cout << sizeof(x) << endl;        // 输出 8
```

表明所使用的编译器中int型数据同long型数据，在内存中占4字节。double型的变量x在内存中占用8字节。

整数按其二进制位求反“~”运算符见下面的位运算。

3.3.3 二进制位运算

优先级为3的位反运算“~”是单目运算，只能对整型数据操作，它将操作数按二进制位求反。

优先级为7的运算是二进制位移运算，均为双目运算。设n, k均为整数，表达式 $n \ll k$ 表示将n代表的整数按照二进制位左移k位，左侧被移出的部分直接丢弃，右侧补零。左移k位相当于乘以 2^k 。表达式 $n \gg k$ 表示将n代表的整数按照二进制位右移k位，右侧被移出的部分直接丢弃，左侧则补符号位（负数补1，非负数及无符号数则补0）。值得注意的是：负整数n右移k位不一定等于n除以 2^k 。如： $-1 \gg 1$, $-1 \gg 2$, $-1 \gg 3$ 等，皆为-1。

优先级为10的是位与运算；优先级为11的是位异或运算；优先级为12的是位或运算。因为这三种位运算均不进位，故仅列出一位二进制数的位操作法则即可。

\sim	0 1	$\&$	0 1	\wedge	0 1	$ $	0 1
	1 0	0 0	0 0	0 0 1	0 0	0 0 1	0 0 1
	1 0 1	1 0 1	1 1 0	1 1 0	1 1 1	1 1 1	1 1 1
位反	位与	位异或	位或				

3.3.4 迭代赋值运算

在第2章介绍了算术运算的迭代赋值运算，其优先级为16。本优先级的运算符还有二进制位运算的迭代赋值运算 $\&=$ 、 $\wedge=$ 、 $|=$ 、 $\ll=$ 和 $\gg=$ 。例如：

```
char c='A';           // 字符'A'的ASCII码为65, 即01000001
c |= ' ';            // 空格字符的ASCII码为32, 即00100000
cout << c << endl; // 输出字符'a', 其ASCII码为97, 即01100001
c >>= 1;             // 右移1位, 相当于除以2
cout << c << endl; // 输出字符'0', 其ASCII码为48, 即00110000
```

3.3.5 抽取及插入运算

C++重载了运算符“ $>>$ ”和“ $<<$ ”分别用于输入与输出操作。“ $>>$ ”被称为抽取运算符，“ $<<$ ”被称为插入运算符。插入运算操作除了在输出设备上输出指定表达式的值外，运算的结果为其左边的输出设备名。这样，才使连续输出成为可能。即

```
cout << "The result is " << 666 << endl;
```

相当于

```
((cout << "The result is ") << 666) << endl;
```

因为表达式`(cout << "The result is ")`的结果为`cout`（别名）。

输入操作的情况是类似的。抽取运算操作不仅将从输入设备上输入的数值存入了相应的变量，而且运算的结果为其左边的输入设备名。这样，也才允许连续输入。如：

```

int m, n;           // 定义两个整型变量
cout << "请输入两个整数: ";
cin >> m >> n;    // 输出提示信息
(cin >> m) >> n; // 接收键盘输入
                    // 与上一行等价

```

因为表达式 `(cin >> m)` 的结果是 `cin` (别名)。

【注】 若输入的数据不符合数据类型的要求 (例如, 若要求输入一个整数或实数却输入了一个字母或字符串, 或者输入了如 `Ctrl+Z` 键等) 则 `(cin>>x)` 为 `NULL` (即 0)。此时需要清除错误状态标志并忽略输入缓冲区中的信息 (参见 14.1.2 小节例 14.2)。

3.3.6 三目条件运算

优先级为 15 的运算是唯一的一个三目运算, 它需要三个操作数参与运算:

```
表达式1 ? 表达式2 : 表达式3
```

若表达式 1 为“真”(非零), 则运算结果为表达式 2 的值, 否则为表达式 3 的值。

三目条件运算的结合方向为从右到左。该运算符可以嵌套使用, 有些场合下可代替条件分支或多分支语句。如:

```

int abs(int x)      // 绝对值函数
{
    return x>=0 ? x : -x;
}
int sgn(int x)      // 符号函数
{
    return x>0 ? 1 : (x<0? -1 : 0);    // 此行的圆括号可以省略
}                   // 保留它则便于程序理解

```

3.3.7 逗号运算

优先级最低的是逗号运算。逗号运算是一系列用逗号连接起来的表达式。

```
表达式1, 表达式2, ..., 表达式n
```

逗号运算将依次计算各表达式的值, 并将最后一个表达式的值作为整个逗号运算表达式的值。

3.3.8 区分作用域

优先级最高的运算符是双冒号 “`::`”, 它被称为作用域区分符, 用于区分全局变量 (全局变量以双冒号开头。参见例 3.1 表达式 `::n`, `::a` 表示访问被局部变量屏蔽了的全局变量), 或某名字空间中的名字, 或者某个类的对象等。例如, 若程序中有包含指令 `#include <iostream>` 而未使用语句 `using namespace std;` 即未声明使用标准名字空间, 则需要使用如下格式:

```
std::cout << "OK" << std::endl;
```

以指明 `cout` 和 `endl` 属于 `std` 名字空间。

3.4 语句

第二章已经介绍了大部分语句，包括最常见的赋值语句、输入/输出语句、程序执行流程控制语句等。

规定 3.9（表达式语句） 表达式后加分号。

规定 3.10（复合语句） 用一对花括号包围起来的若干条（0 条、一条或多条）语句被称为复合语句。复合语句在语法上作为一个单语句使用，可出现在任何可以使用语句的地方。

规定 3.11（空语句） 仅用一个分号或仅用一对花括号表示。其主要作用是满足语法的要求：需要一个语句，但不需要执行任何操作。

3.5 I/O 流格式控制

将参数化了的 I/O 流格式控制标识符添加到 I/O 流中可以控制 I/O 的格式，以实现将数据按一定的格式进行输入或输出操作。这些控制标识符在 I/O 流控制头文件 `iomanip` (input/output manipulators) 中声明，且属于标准名字空间 `std`。因此，使用 I/O 流格式控制的程序需要用 `#include <iomanip>` 包含该头文件，并需要使用名字空间 `std`。C++ 的 I/O 流格式控制标识符及其作用如表 3-4 所示。

表 3-4 I/O 流格式控制标识符及其作用

格式控制标识符	作用
<code>std::left</code> <code>std::internal</code> <code>std::right</code>	输出的数据在本域宽内左对齐（填充字符填在右边） 两端对齐（符号靠左，数值靠右，填充字符填在中间） 输出的数据在本域宽内右对齐（填充字符填在左边）
<code>std::fixed</code> <code>std::scientific</code>	定点形式（小数形式）输出 指数形式（科学记数法）输出
<code>std::dec</code> <code>std::oct</code> <code>std::hex</code>	置基数为 10，整数按十进制数输入或输出 置基数为 8，整数按八进制数输入或输出 置基数为 16，整数按十六进制数输入或输出
<code>std::uppercase</code> <code>std::nouppercase</code>	十六进制数中的字母按大写字母输出 十六进制数中的字母按小写字母输出
<code>std::showbase</code> <code>std::noshowbase</code>	八进制输出时，输出前缀（0）；十六进制输出时，输出前缀（0x或0X，与 <code>uppercase</code> 设置有关） 不输出基数（即取消 <code>showbase</code> 设置）
<code>std::showpoint</code> <code>std::noshowpoint</code>	强制输出浮点数的小数点 取消强制输出浮点数的小数点设置
<code>std::showpos</code> <code>std::noshowpos</code>	强制输出数值的符号（特指输出“+”号） 取消强制输出数值的符号设置（当然，负数会输出“-”号）
<code>std::boolalpha</code> <code>std::noboolalpha</code>	逻辑值输出字符 <code>true</code> 或 <code>false</code> 逻辑值输出字符 1（表示 <code>true</code> ）或 0（表示 <code>false</code> ）
<code>std::skipws</code> <code>std::noskipws</code>	忽略前导空白字符（用于输入控制） 不忽略前导空白字符（用于输入控制）

续表

格式控制标识符	作用
std::unitbuf std::nouunitbuf	每次输出后刷新所有的流 取消每次输出后刷新所有的流设置
std::setiosflags(标志位) std::resetiosflags(标志位)	直接设置格式控制标志位 (标志位值如表 3-5 所示) 直接取消格式标志位设置
std::setprecision(精度位数)	设置输出数值的位数 (默认 6 位)。设置 fixed 后为输出 小数精度位数, 小数位被截断时, 最后的小数位四舍五入
std::setfill(字符)	设置填充字符。默认值为空格字符 ' '。
std::setw(域宽)	设置下一个输出项宽度 (占字符数)。若实际数据超宽, 则不受此限制。该设置仅对其下一个输出数据起作用

表 3-4 中的横线将 I/O 流格式控制标识符分成若干组, 同一组中的两种或三种设置之间是互斥的。例如, 设置了 left 后, 将自动取消可能曾经设置的 internal 和 right。表中, 最后一项 setw 仅对其后的一个输出项起作用。

源代码 3.4 I/O 流格式控制示例程序

```

1 // iomanipTest.cpp
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5 int main()
6 {
7     int n, left = 101, right = 303; // 此处 left, right 为局部变量
8     const double E = 2.718281828459045;
9     cout << "请输入一个十六进制数: ";
10    cin >> hex >> n >> dec; // 接收十六进制数输入后还原成默认状态
11    cout << left << "\t" << right << endl;
12        // 控制标识符 left, right 被屏蔽
13    cout << setfill('*') << showbase << showpos << uppercase
14        << std::left << "(" << oct << setw(8) << n << ")"
15            // std::不可省略
16        << internal << dec << setw(8) << n << "]"
17        << std::right << hex << setw(8) << n << "]"
18            // std::不可省略
19        << dec << nouppercase << noshowpos << noshowbase
20        << setfill(' ') << endl;
21    cout << E << "\n" << setprecision(10) << E << "\n"
22        << fixed << E << "\n" << scientific << E
23        << setprecision(6) << fixed << endl;
24    cout << setiosflags(ios::scientific) << E << endl;
25    cout << resetiosflags(ios::fixed) << E << endl;
26    cout << boolalpha << (3>2) << "\t" << (3<2) << "\n"
27        << noboolalpha << (3>2) << "\t" << (3<2) << endl;
28    return 0;
29 }
```

程序的运行结果:

```

请输入一个十六进制数: 7f
101      303
(0177*****)[+*****127]{*****0X7F}
2.71828

```

```

2.718281828
2.7182818285
2.7182818285e+000
2.71828
2.718282e+000
true    false
1        0

```

事实上，C++编译器用一个长整型变量的不同位分别记录程序中所设置的各种输入格式控制，系统将在每一次输入操作时根据该变量的具体情况实施格式控制。同理，C++编译系统用另一个长整型变量的不同位分别记录程序中所设置的各种输出格式控制，并在每一次输出操作时根据该变量的内容实施格式控制。这种变量被称为标志变量，标志变量的各个位可以独立地进行操作。

对于输入流、输出流格式控制变量，其默认值均为 (`ios::dec | ios::skipws`)。可用 `cin.flags` 函数和 `cout.flags` 函数调用分别获取输入流及输出流格式控制标志变量的当前值。

为便于记忆，C++编译系统还提供了若干常量（参见表 3-5 第 1 列）。不过，在不同的编译器中，这些常量的值可能不同（参见表 3-5 的最后两列），因此，在程序中不应使用这些具体数值，而应该使用常量标识符以便使程序能在多种编译器中正确地移植。表 3-5 列出 MinGW C++ 和 Visual C++ 中具体常量的值是为了展示标志变量的使用方法。

表 3-5 I/O 格式标志位及功能描述

格式标志	默认	功 能 描 述	MinGW	VC++
<code>ios::skipws</code>	✓	忽略前导空白字符	4096	1
<code>ios::unitbuf</code>	✗	每次输出后刷新所有的流	8192	2
<code>ios::uppercase</code>	✗	十六进制数中字母大写 (A, B, C, D, E, F)	16384	4
<code>ios::showbase</code>	✗	输出整数的基数。按八进制输出时，输出前缀 (0)；按十六进制输出时，输出前缀 (0x或0X，与uppercase设置有关)	512	8
<code>ios::showpoint</code>	✗	强制输出浮点数的小数点	1024	16
<code>ios::showpos</code>	✗	强制输出数值的符号 (特别是“+”号)	2048	32
<code>ios::left</code>	✗	输出的数据在本域宽内左对齐	32	64
<code>ios::right</code>	✗	输出的数据在本域宽内右对齐	128	128
<code>ios::internal</code>	✗	输出的数据在本域宽内两端对齐 (符号左对齐，数值右对齐，中间由填充字符填充)	16	256
<code>ios::dec</code>	✓	置基数为 10，整数按十进制操作	2	512
<code>ios::oct</code>	✗	置基数为 8，整数按八进制操作	64	1024
<code>ios::hex</code>	✗	置基数为 16，整数按十六进制操作	8	2048
<code>ios::scientific</code>	✗	浮点数以科学记数法格式输出	256	4096
<code>ios::fixed</code>	✗	浮点数以定点格式 (小数形式) 输出	4	8192
<code>ios::boolalpha</code>	✗	逻辑值输出字符 <code>true</code> 或 <code>false</code> ，默认输出 1 或 0	1	16384
<code>ios::adjustfield</code>	✗	<code>ios::left ios::internal ios::right</code>	176	448
<code>ios::basefield</code>	✗	<code>ios::dec ios::hex ios::oct</code>	74	3584
<code>ios::floatfield</code>	✗	<code>ios::fixed ios::scientific</code>	260	12288

另一种控制 I/O 流格式的方法是用 `setiosflags`(标志位) 或 `resetiosflags`(标志位) 直接设置格式控制标志变量的标志位（参见表 3-4 的倒数 4、5 行）。这些标志位的标识符参见表 3-5 的第 1 列。它们可以用位操作进行组合，甚至将互斥的设置组合在一起，

例如：

```
cout << setiosflags(ios::hex|ios::oct|ios::left|ios::right);
// 出现互斥设置
cout << cout.flags() << endl; // 输出当前格式控制标志变量的值
cout << resetiosflags(ios::hex|ios::oct|ios::left|ios::right);
// 取消上述设置
cout << cout.flags() << endl; // 输出当前格式控制标志变量的值
```

用 `setiosflags` 和 `resetiosflags` 设置或取消格式控制变量的标志位时无法自动保证应该的互斥。当格式控制标志变量中 `ios::dec`、`ios::oct` 和 `ios::hex` 位设置不唯一时，系统按 `ios::dec` 处理。当格式控制标志变量中 `ios::left`、`ios::internal` 和 `ios::right` 位设置不唯一时，系统按 `ios::right` 处理。当格式控制标志变量中 `ios::fixed` 和 `ios::scientific` 位设置不唯一时，系统按 `ios::fixed` 处理。因此，执行上述程序第 24 行的结果仍按 `ios::fixed` 格式输出。第 25 行的作用是取消 `ios::fixed` 设置，使在其之前第 24 行设置的 `ios::scientific` 生效。

请注意控制标识符 `std::left`（参数化了的 I/O 流格式控制，实际上为函数）与 `ios::left`（标志位值）的区别。

```
cout << ios::left << endl; // 输出一个数值
cout << std::left; // 设置左对齐输出模式
```

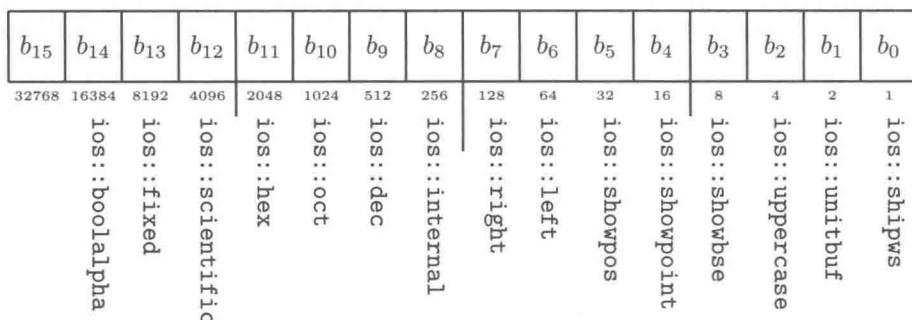


图 3-2 Visual C++ 中的 I/O 流格式控制标志变量各标志位

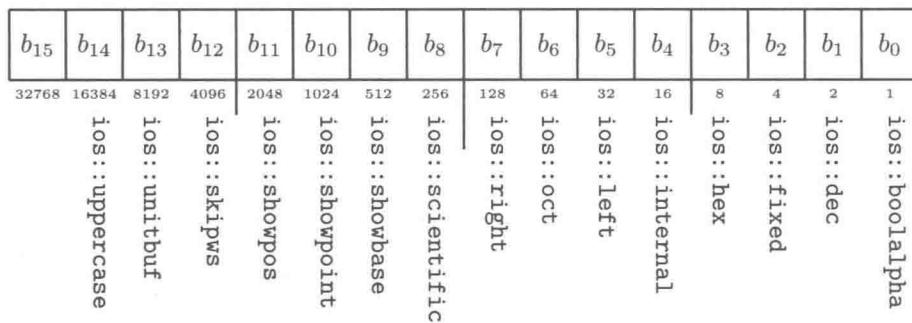


图 3-3 MinGW C++ 中的 I/O 流格式控制标志变量各标志位

Visual C++ 与 MinGW C++ 中，I/O 流格式控制标志变量各标志位如图 3-2 和图 3-3 所示。显然，Visual C++ 对这些标志位的值是按照功能“意群”设计的，而 MinGW C++ 则是以标志位的标识符名称按字典顺序设计的。

3.6 应用举例

本节综合运用C++的基本语法介绍数据转换、I/O操作等程序的设计。我们仅给出相应的函数，请读者仿照第2.3.2小节源代码2.4按多文件结构将下面的所有函数组织到同一个程序中，另编写主函数用简易菜单及switch语句对这些函数进行调度（模仿源代码2.4的test_main.cpp）。必要时包含头文件*iomanip*。

阅读理解程序的能力也是程序员所必须具备的。本节直接给出一些未加注释的自定义函数。希望读者精读它们以提高阅读理解程序的能力。阅读程序的方法：按照程序执行的时序，跟踪记录各变量的变化情况，注意各种控制状态的改变情况、显示器上输出的情况（一般地，显示器每行能输出80个字符，超过80个字符时自动换行）。

3.6.1 深入理解ASCII字符集

1. 显示ASCII字符集

源代码3.5 显示ASCII字符集

```

1 void ShowASCII()
2 {
3     int i;
4     cout << setfill('0') << uppercase << showbase;
5     for(i=0; i<256; i++)
6         cout << setw(3) << dec << i << '\t'
7             << setw(4) << oct << i << '\t'
8                 << setw(4) << hex << i << '\t'
9                     << char(i) << endl;
10    cout << setfill(' ') << dec << nouppercase
11        << noshowbase;                                // 函数返回前还原为默认设置
12 }
```

2. 显示GB 2312—80汉字字符集

请参见第1.1.3小节关于两字节汉字机内码的内容。

源代码3.6 显示GB 2312—80汉字字符集

```

1 void ShowGBHZ()
2 {
3     int section, position;
4
5     cout << setfill('0');
6     while(true)
7     {
8         cout << "\n请输入区码(1--87,0返回): ";
9         cin >> section;
10        if(section<1 || section>87) break;
11        for(position=1; position<=94; position++)
12            cout << setw(2)<<section << setw(2)<<position<<' '
13                << char(160+section)<<char(160+position)<<' ';
14        cout << endl;
15    }
16    cout << setfill(' ');
17 }
```

3. 英文字母大小写转换

源代码 3.7 英文字母大小写转换

```

1 char upper(char c)           // 不改变实参
2 {
3     if(c>='a' && c<='z')
4         return c - 'a' + 'A';
5     else
6         return c;
7 }
8
9 char lower(char c)           // 不改变实参
10 {
11    if(c>='A' && c<='Z')
12        return c - 'A' + 'a';
13    else
14        return c;
15 }
```

3.6.2 深入理解整型数据

1. 整数按八、十、十六进制输入输出

整数在计算机内存中均按二进制形式存放，输入输出时默认按十进制形式。通过 I/O 流格式控制可实现八、十、十六进制整数之间的任意转换而不需要程序员编写其他转换计算语句。

源代码 3.8 整数八、十、十六进制输入输出

```

1 void InputInt(int base)
2 {
3     int n;
4     if(base==8)
5         cin >> oct;
6     else if(base==16)
7         cin >> hex;
8     else
9     {
10         base = 10;
11         cin >> dec;
12     }
13     cout << "请输入一个" << base << "进制形式的整数: ";
14     cin >> n >> dec;
15     cout << oct << n << "(O)↓=↓"
16         << hex << n << "(H)↓=↓"
17         << dec << n << endl;
18 }
```

2. 输出整数的二进制各位

源代码 3.9 整数的二进制各位

```

1 void CBits(char x)
2 {
3     int len = sizeof(x)*8;
```

```

4     cout << x << ":" << int(x) << ":";
5     for(int i=len-1; i>=0; i--)
6     {
7         cout << int(x>>i & 1);
8         if(i%4==0) cout << ' ';
9     }
10    cout << endl;
11 }
12
13 void UCBits(unsigned char x)
14 {
15     int len = sizeof(x)*8;
16     cout << x << ":" << int(x) << ":";
17     for(int i=len-1; i>=0; i--)
18     {
19         cout << int(x>>i & 1);
20         if(i%4==0) cout << ' ';
21     }
22     cout << endl;
23 }
24
25 void IBits(int x)
26 {
27     int len = sizeof(x)*8;
28     cout << x << ":" << " ";
29     for(int i=len-1; i>=0; i--)
30     {
31         cout << (x>>i & 1);
32         if(i%4==0) cout << ' ';
33         if(i%8==0) cout << ' ';
34     }
35     cout << endl;
36 }
37
38 void UIBits(unsigned int x)
39 {
40     int len = sizeof(x)*8;
41     cout << x << ":" << " ";
42     for(int i=len-1; i>=0; i--)
43     {
44         cout << (x>>i & 1);
45         if(i%4==0) cout << ' ';
46         if(i%8==0) cout << ' ';
47     }
48     cout << endl;
49 }

```

3. 整数的十进制各位

源代码 3.10 整数的十进制各位

```

1 int SumDigits(int n)           // 十进制整数各位之和
2 {
3     int sum = 0;
4     while(n!=0)
5     {
6         sum += n % 10;          // 取n当前的个位数字
7         n /= 10;                // 除以10, 取整数商

```

```

8      }
9      return sum;
10 }
11
12 int reverse(int n)          // 十进制整数各位倒置
13 {
14     int m = 0;
15     while(n!=0)
16     {
17         m = 10*m + n%10;           // m乘以10，再添加个位数字
18         n /= 10;
19     }
20     return m;
21 }
```

3.6.3 输出字符图案

1. 九九乘法表

源代码 3.11 九九乘法表

```

1 void Multiply()
2 {
3     int i, j;
4
5     for(i=1; i<10; i++)
6     {
7         for(j=1; j<10; j++)
8             cout << i << "*" << j << "="
9             << setw(2) << i*j << " ";
10        cout << endl;           // 无换行
11    }                         // 换行
12 }
```

2. 字符图案

源代码 3.12 字符菱形图案

```

1 void pattern()
2 {
3     int i, j;
4     for(i=0; i<9; i++)
5     {
6         cout << setw(2*(10-i)) << "";
7         for(j=0; j<2*i+1; j++)
8             cout << "#";
9         cout << endl;
10    }
11    for(i=9; i>=0; i--)
12    {
13        cout << setw(2*(10-i)) << "";
14        for(j=0; j<2*i+1; j++)
15            cout << "#";
16        cout << endl;
17    }
18 }
```

3.7 趣味程序——行走的字符串

源代码 3.13 行走的字符串

```

1 #include <iostream>
2 #include <ctime>           // 为了使用函数 clock
3 #include <conio.h>         // 为了使用函数 kbhit 和函数 getch
4 using namespace std;
5
6 void delay(int n)          // 延时 n 毫秒
7 {
8     clock_t t=clock();
9     while(clock()-t<n)
10    ;
11 }
12
13 void walking_string()
14 {
15     int n = 100, w=1, i;
16
17     cout << "按任意键....." << endl;
18     while(kbhit()!=true)
19     {
20         cout << "\右行的文字";  delay(n);
21         for(i=0; i<10; i++)
22             cout << '\b';      // 转义字符：退格
23         if(w==68) cout << "\oooooooooooo\r";
24         w = w % 68 + 1;
25     }
26     getch();
27     cout << endl;
28
29     while(!kbhit())
30     {
31         for(i=0; i<w; i++)
32             cout << '\u';
33         cout << "左行的文字\u\r";  delay(n);
34         w = (w+66) % 68 + 1;
35     }
36     getch();
37     cout << endl;
38 }
39
40 int main()
41 {
42     walking_string();
43     return 0;
44 }
```

3.8 小结

本章的内容带有资料的性质，并不要求读者立即理解并熟练掌握，但要求读者经常翻阅，本课程结束时，应该最终完全掌握。到本章为止，已经介绍了C++程序设计语言的基本语法。这些基本语法是程序设计的基础，需要读者尽快掌握，并且学会灵活运用。

与学习自然语言中的外国语一样，学习计算机程序设计语言也需要多读、多写。首先需要阅读一些简单的程序，然后立即动手编写类似的程序。当自己动手编写的总代码行数达到一定量的时候，自己的程序设计能力和水平就自然提高了。

事实上，主动地编写程序比被动地阅读程序更加容易。本章给出了一些源代码（即源程序），希望读者精读它们以培养和提高自己阅读程序的能力。

练习 3

1. 指出下列各程序中的错误

(1) 希望根据半径计算圆面积。

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     const double PI;
7     double r, area;
8
9     PI = 3.1415926;
10    area = PI*r*r;
11
12    cout << "请输入圆半径: ";
13    cin >> r;
14    cout << "圆面积为" << area << endl;
15    return 0;
16 }
```

(2) 希望输出字符及其 ASCII 编码。

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     char c = 48;
7
8     cout << 'ASCII 码为' << int(c)
9         << '的字符为' << c << endl;
10    return 0;
11 }
```

(3) 希望制作一张数学用表——平方根表。

```

1 #include <iostream>
2 #include <iomanip>
3 #include <cmath>
4 using namespace std;
5
6 void PrintSQRT()
7 {
8     double x, y;
9     int i, j;
10    cout << setw(44) << "平方根表" << endl; // 输出表的名称
11 }
```

```

12     cout << "sqrt() 0.0";
13     for(x=0.1; x<=0.9; x+=0.1)
14         cout << setw(7) << x;           // 输出表头
15
16     cout << '\n' << setfill('-') << setw(79) << ""
17             << setfill('_) << endl;        // 输出横线
18
19     cout << fixed << setprecision(4) << showpoint;
20     for(i=0; i<=20; i++)
21     {
22         cout << setw(5) << i << "|";      // 当前行左侧部分
23         for(j=0; j<10; j++)
24         {
25             x = i + j/10.0;
26             y = sqrt(double x);
27             cout << setw(7) << y;           // 输出表中的数值
28         }
29         cout << endl;                      // 换行
30     }
31 }
32
33 int main()
34 {
35     PrintSQRT();
36     return 0;
37 }
```

(4) 希望计算 100 以内的所有偶数之和。

```

1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int i, sum;
6     for(i=2; i<=100; i+=2)
7         sum += i;
8     cout << "100 以内的偶数和为" << sum << endl;
9     return 0;
10 }
```

(5) 希望返回多个数据。

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 double triangle(double a, double b, double c)
6 {
7     double len, area;
8     double s = (a+b+c)/2;
9     if(s>a && s>b && s>c)
10    {
11        len = a + b + c;
12        area = sqrt(s*(s-a)*(s-b)*(s-c));
13    }
14    else
15        len = area = 0;
16    return len, area;
```

```

17 }
18
19 int main()
20 {
21     cout << "三角形的周长及面积分别为" <<
22         triangle(3, 4, 5) << endl;
23     return 0;
24 }
```

(6) 希望判断给定字符是否为数码字符。

```

1 #include <iostream>
2 using namespace std;
3
4 int isDigit(char c)
5 {
6     if(c>='0' && c<='9')
7         return 1;
8 }
9
10 int main()
11 {
12     if(isDigit('3')) cout << "Yes." << endl;
13     return 0;
14 }
```

(7) 指出下列函数中的错误。

```

1 int & f(int a)
2 {
3     return ++a;
4 }
```

(8) 指出下列函数中的错误。

```

1 int & f(int &a)
2 {
3     return a+1;
4 }
```

(9) 修改函数 f 中的错误（不要修改主函数）。

```

1 #include <iostream>
2 using namespace std;
3
4 int f(int &a)
5 {
6     return 10*a;
7 }
8
9 int main()
10 {
11     cout << f(3) << endl;
12     return 0;
13 }
```

(10) 指出下列程序中的错误。

```

1 #include <iostream>
2 using namespace std;
3 int f(int &a)
4 {
5     return 10*a;
6 }
7 int main()
8 {
9     double x = 3;
10    cout << f(x) << endl;
11    return 0;
12 }
```

2. 基本题

(1) 第3.6.1小节的ShowASCII函数中,若将变量i定义成unsigned char型,程序中需要进行哪些修改?以保证函数功能不变。

(2) 将代数式 $\frac{-b - \sqrt{b^2 - 4ac}}{2a}$,写成C++表达式。

(3) 有某变量a,请分析if(a!=0)与if(a)的关系;if(a==0)与if(!a)的关系。

(4) 编写一个函数,函数原型为int digit(unsigned int n);其功能是将非负整数n按十进制形式反复累加其各位数字之和,直到其和数在0~9范围内,并返回该结果。再编写一个主函数,根据从键盘输入的无符号整数测试该函数的正确性。

(5) 编程求“水仙花数”。“水仙花数”是指一个三位数,其各位数字的立方和等于该数本身。例如,153是一个水仙花数,因为 $153 = 1^3 + 5^3 + 3^3$ 。

(6) 编程输出如下“下三角”形式和“上三角”形式的“九九乘法表”。

```

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

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

(7) 已知某年元旦为星期一, 请编程输出该年一月份的日历。要求按星期日、星期一、……、星期六编排。

(8) 编写一个程序, 从键盘输入顾客所购买商品的单价 p 和数量 n , 计算出该顾客应付款 total 金额。然后根据用户所交纳的金额 m , 计算出找零金额数 c 。

(9) 修改上题以实现顾客购买多件商品的情形。提示: 可以约定当输入的商品单价为 0 时, 表示该顾客所购买的商品统计完毕。

3. 扩展题 (程序设计竞赛题型)

(1) 验证 $3n + 1$ 问题。

问题描述 对于给定的一个正整数, 若该数为偶数则将其除以 2, 若为奇数则将其乘 3 再加 1。反复进行上述过程, 直到结果为 1 时停止。这就是著名的 “ $3n + 1$ ” 问题。要求计算对于给定的整数按 $3n + 1$ 规则变换到 1 所需要的变换次数。

输入说明 输入数据有多个, 每个数据对应一种情形。

输出说明 对于每一种情形, 要求先输出 “Case 序号: ”, 然后输出读入的整数、逗号、空格和计算结果。若输入的数据为非正整数, 规定结果为 -1。

输入样例

```
1024 1023 0 100 66
```

输出样例

```
Case 1: 1024, 10
Case 2: 1023, 62
Case 3: 0, -1
Case 4: 100, 25
Case 5: 66, 27
```

(2) 输出由字符构成的 16 行 8 列图案。

问题描述 对于每一种情形, 给定标志 (负号 “-” 表示输出阴图, 正号 “+” 或者缺省标志时输出阳图)、一个字符及图案数据。将每个图案数据的 8 个二进制位对应图案一行中的 8 列。对于阳图, 当二进制位为 1 时输出给定字符, 否则输出空格字符。对于阴图, 当二进制位为 1 时输出空格字符, 否则输出给定字符。

输入说明 输入数据有多行。每两行对应一种情形, 其中的第一行为标志 (可能缺省) 及一个字符, 第二行为 16 个由两位十六进制形式表示的图案数据。

输出说明 对于每一种情形输出由给定字符及空格字符组成的 16 行 8 列字符图案。

输入样例

```
-0
00 00 38 6C C6 C6 C6 C6 C6 6C 38 00 00 00 00
1
00 00 18 38 78 18 18 18 18 18 18 7E 00 00 00 00
-E
00 00 FE 66 62 68 78 68 60 62 66 FE 00 00 00 00
```

输出样例 (其中的行号不必输出, 此处是为帮助理解特意添加的)

```
1 00000000
2 00000000
3 00 000
4 0 0 00
5 000 0
6 000 0
7 000 0
8 000 0
9 000 0
10 000 0
11 0 0 00
12 00 000
13 00000000
14 00000000
15 00000000
16 00000000
17
18
19 11
20 111
21 1111
22 11
23 11
24 11
25 11
26 11
27 11
28 111111
29
30
31
32
33 EEEEEEEE
34 EEEEEEEE
35 E
36 E EE E
37 E EEE E
38 E E EEE
39 E EEE
40 E E EEE
41 E EEEEE
42 E EEE E
43 E EE E
44 E
45 EEEEEEEE
46 EEEEEEEE
47 EEEEEEEE
48 EEEEEEEE
```

第4章 变量设计

在本章中，通过几种不同类型的算法实现着重讨论算法实现中的变量设计问题，包括函数的参数、局部变量、标志辅助变量的设计与应用。从本章起，将围绕所谓的大奖赛“评委评分”程序的设计和优化、程序功能的扩充逐步展开讨论。本章涉及一些基本算法，这些算法的设计思想是朴素的。

变量设计是程序设计的重要环节，初学者应该重视这一环节。熟练后，这一环节将成为程序员程序设计时的一种“自觉行为”。

4.1 穷举计算

通过一个具体实例，介绍“实际问题 → 数学模型 → 计算方法 → C++程序 → 计算机运行程序”这一计算机求解问题的全过程。本节重点介绍穷举法程序设计。

4.1.1 “百钱买百鸡”问题

例 4.1 【“百钱买百鸡”问题】公元前 5 世纪，我国古代数学家张丘建在《算经》一书中提出了“百鸡问题”：鸡翁一值钱五，鸡母一值钱三，鸡雏三值钱一。百钱买百鸡，问鸡翁、母、雏各几何？

【数学模型】记 x, y, z 分别表示购买鸡翁，鸡母，鸡雏的数量 ($x, y, z \in \mathbb{Z}$)，则有

$$\begin{cases} x + y + z = 100 \\ 5x + 3y + z/3 = 100 \end{cases}$$

上述方程组有三个未知数仅有两个方程，是一个不定方程。它在非负整数范围内可能有多组解。求解不定方程的数学方法需要较多的理论知识，存在较大的困难。

【算法设计】显然有 $0 \leq x \leq 100/5$, $0 \leq y \leq 100/3$, 且 $x, y \in \mathbb{Z}$ 。即

$$\begin{cases} x = 0, 1, 2, \dots, 20 \\ y = 0, 1, 2, \dots, 33 \\ z = 100 - x - y \end{cases}$$

亦即，该问题可以被限制在如上所述的一个较小范围——包含解集的某有限集合内讨论。利用计算机快速运算的能力，在该有限集合中逐个进行搜索（试算），以找出该问题的所有解。这种处理（算法创作）策略被称为穷举法或枚举法。这种以简御繁的策略是计算机解题的一种常用方法。穷举法将求解问题质的困难转化成量的重复。但是，当可能的解集中元素个数巨大时，该方法失效。尽量裁减搜索范围是穷举法成败的关键。

本问题可能的解集是三维空间中的点，由于有约束条件 $x + y + z = 100$ ，可将解集投影到任意一个二维坐标平面上。例如，将解集投影到 xOy 平面上，得到的点应该被包含在集合 $A = \{(x, y) | x = 0, 1, 2, \dots, 20; y = 0, 1, 2, \dots, 33\}$ 中，该集合共有 $21 \times 34 = 714$

个元素。因此，仅需要对这 714 种情形逐个试算，判断是否满足题意，并将满足者输出即可。即需要对 (x, y) 每一对组合反复地进行判断操作（必要时进行输出）。

值得指出的是，若考察 (y, z) 取值的各种组合，将有 $34 \times 101 = 3434$ 种情况。显然此时的穷举操作计算量将近是考察 (x, y) 组合的 5 倍。由于有约束条件，更不必去讨论 (x, y, z) 各自独立取值的 $21 \times 34 \times 101 = 72114$ 种组合，而且还要另外判断 $x + y + z$ 是否等于 100，其计算量是考察 (x, y) 组合的 100 多倍。这说明在穷举计算中，搜索范围的不同裁减可导致计算效率的巨大差别。

【计算流程】 将集合 A 的 714 个元素描在平面坐标系中，将形成一个矩形点阵。遍历这 714 个元素可按“行”或按“列”依次扫描。例如，按“列”的具体扫描过程如下：

- ① 确定购买鸡翁的数量 x （从 0 起，逐次增加 1，直到 20 为止）。
- ② 对于每一个确定的 x 而言，再确定购买鸡母的数量 y （从 0 起，逐次增加 1，直到 33 为止）。

③ 对于每一个确定的 (x, y) 组合，计算购买鸡雏的数量 $z = 100 - x - y$ ；计算所需要的金额 $s = 5x + 3y + z/3$ ；判断 s 是否恰好为 100，若是，则对应的 (x, y, z) 便是该问题的一组解（输出该组解）。处理下一个组合。

显然，上述的三个过程是逐层嵌套的。按照自顶向下、逐步求精的设计方法分解成图 4-1 所示的计算流程图。

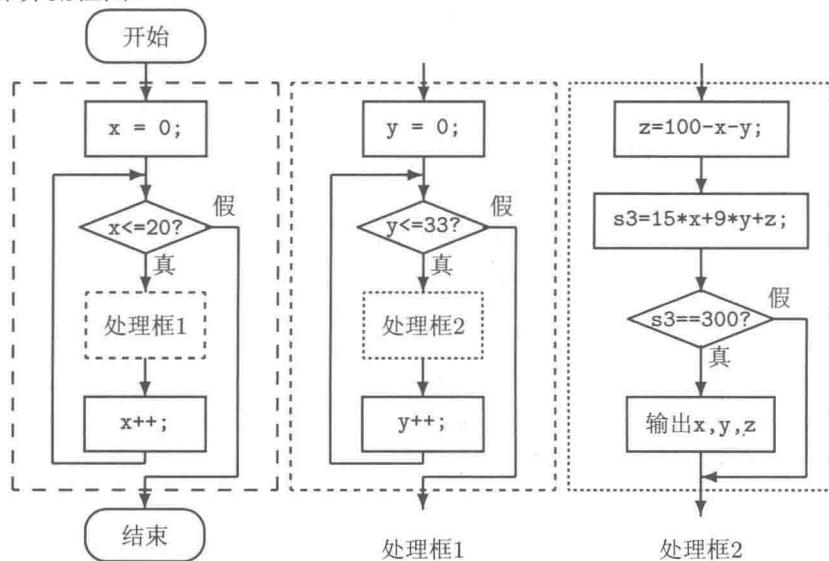


图 4-1 “百钱买百鸡”问题计算流程图

在左侧的总框图中，留有一个待进一步细化的部分（处理框 1）；同样，在处理框 1 中，也留有一个待细化的部分（处理框 2）。它们是按层次完全嵌套的，从上一层看待下一层时，处理框是一个整体。逐层细化的部分见图 4-1 中用不同虚矩形框表示的部分。每一个虚矩形框内部均又由若干语句组成，从宏观来看，虚矩形框是“单入口、单出口”的，能保证该处理框完全嵌入到其上一层框图之中。

【C++ 程序】 根据上述计算流程图，按照 C++ 语言提供的表达方式，采用合适的循环结构、选择结构等 C++ 语句，写成如下 C++ 源程序。

源代码 4.1 百钱买百鸡问题

```

1 // HundredChicken.cpp 源程序文件名
2 #include <iostream>
3 using namespace std;
4
5 int chicken100(); // 函数原型用于函数声明
6
7 int main()
8 {
9     chicken100(); // 主函数起调度作用，尽可能简单
10    return 0;
11 }
12
13 int chicken100()
14 {
15     int cocks, hens, chicks, n=0;
16     for(cocks=0; cocks<=20; cocks++) // 可能购买公鸡的数量 0~20
17         for(hens=0; hens<=33; hens++) // 可能购买母鸡的数量 0~33
18         {
19             chicks = 100 - cocks - hens; // 购买小鸡的数量
20             if(3*100 == 3*(5*cocks + 3*hens) + chicks)
21                 { // 找到一个解
22                     cout << "鸡翁" << cocks
23                         << "只，鸡母" << hens
24                         << "只，鸡雏" << chicks
25                         << "只。" << endl; // 输出解
26                     n++; // 解集元素个数增 1
27                 }
28         }
29     cout << "共有" << n << "个解。" << endl;
30     return n;
31 }

```

【运行程序】 程序经过编译、连接后生成可执行文件，运行结果如下。

```

鸡翁 0 只，鸡母 25 只，鸡雏 75 只。
鸡翁 4 只，鸡母 18 只，鸡雏 78 只。
鸡翁 8 只，鸡母 11 只，鸡雏 81 只。
鸡翁 12 只，鸡母 4 只，鸡雏 84 只。
共有 4 个解。

```

将总框图中的虚矩形框写成函数 `chicken100` 由主函数调用，其主要部分是两重循环及一个分支选择语句的嵌套，与流程图（如图 4-1 所示）对应。整个循环结束时 `cocks` 的值为 21（即继续循环条件 `cocks <= 20` 不成立时）；此时 `hens` 的值为 34（这已经是该变量第 21 次，从 0 逐步递增到 34，总共被改变 714 次）；此时 `chicks` 的值应该是 $100 - 20 - 33 = 47$ 。

值得指出的是，若将第 20 行中的条件判断改成 $100 == 5*cocks + 3*hens + chicks/3$ ，则程序的运行结果将给出 7 组解，其中有三组是错误的。若改成 $100 == 5*cocks + 3*hens + chicks/3.0$ ，则程序运行正确。但是，这有赖于两个近似表示的浮点数之间的精确比较，这种做法是应该尽量避免的。本例中，将比较的双方均扩大到三倍成为整数后进行精确的比较。

另外，将常量写在关系运算符“`==`”的左侧有利于减少将关系运算符误用成赋值运算符“`=`”的机会，因为编译器会指出“`if(3=a)`”中的语法错，而“`if(a=3)`”却能够通过编译。由于赋值表达式“`a=3`”的值为3（非零，即`true`），上述判断相当于“`if(true)`”，这不一定是程序员的本意。

4.1.2 判定素数

例 4.2 给定一个正整数 n ，判断其是否为素数（素数亦称为质数，即只有1及自身两个平凡因数的正整数。正整数1不是素数）。

【算法设计】为了判断给定的正整数 n 是否有非平凡的因数，简单的方法是穷举搜索。因为 n 的非平凡因数一定不超过 n 。故可从2起到 $n-1$ 逐个进行判断。这样处理的计算量大约为 n 。进一步分析以裁减搜索范围。若 n 有一个非平凡因数 a ，则 n/a 亦为 n 的一个非平凡因数，并且其中必有一个因数不超过 \sqrt{n} （用反证法可证明之）。从而，实际计算只需要从2起穷举判断到 $\lfloor \sqrt{n} \rfloor$ （向下取整）即可。这样处理的计算量约为前一种处理方法的 $1/\sqrt{n}$ ，节省的计算量十分可观！

源代码 4.2 判定素数

```

1 // isprime.cpp 判定素数
2 #include <iostream>
3 #include <iomanip>
4 #include <cmath>
5 using namespace std;
6
7 int isprime(unsigned int n)
8 {
9     if(n<2) return 0;
10    unsigned int k, m;
11    m = (unsigned int)sqrt(double(n)); // 数据类型强制转换
12    for(k=2; k<=m; k++)
13        if(n%k==0)
14            return 0;
15    return 1;
16 }
17 int main()
18 {
19     const unsigned int N = 1000;
20     unsigned int n, m=0;
21     for(n=2; n<=N; n++)
22         if(isprime(n))
23         {
24             cout << setw(4) << n;
25             m++;
26         }
27     cout << "\n\n" << N << "以内共有素数" <<
28             << m << "个。" << endl;
29     return 0;
30 }
```

程序中有如下两种情形结束第12~14行循环语句。

- ① 在该循环体内有机会执行到`return`语句，表明此时的`k`是`n`的一个非平凡因数，故`n`是合数（不是素数），此时已无必要判断到底，可立即返回结果。

② 继续循环的条件不成立，此时 $k==m+1$ 成立。可见，变量 k 除了计数、做除数外，它还可充当了标志作用。

在用循环语句实现穷举计算，并且有提前结束循环语句情形时，循环控制变量常常也用做标志。在循环语句结束后，通过判断循环控制变量的值可知该循环语句的实际执行情况。上述函数可以改写成如下形式。

```

1 int isprime(unsigned int n)
2 {
3     if(n<2) return 0;
4     unsigned int k, m;
5     m = (unsigned int)sqrt(double(n));
6     for(k=2; k<=m; k++)
7         if(n%k==0) break; // 提前结束循环，但不立即退出函数
8     if(k<=m)           // 循环语句结束后根据 k 的值进行判断
9         return 0;
10    else
11        return 1;
12 }
```

程序中的辅助变量 m 具有重要的作用。若去掉该变量而将循环语句的首部写成 `for(k=2; k<=sqrt(double(n)); k++)` 则将执行许多重复的开方计算。当然，可写成 `for(k=(unsigned int)sqrt(double(n)); k>=2; k--)` 而只执行一次开方计算。

关于 `for` 语句中定义的变量的生命期，标准 C++ 规定 `for` 语句中定义的变量的生命期及作用域仅限于其所在的循环语句。例如：

```

1 for(int i=1; i<10; i++)
2 {
3     for(int j=1; j<10; j++)
4         cout << i << "*" << j << "=" << setw(2) << i*j;
5     cout << endl;           // 不能访问 j
6 }                         // 不能访问 i 及 j
```

4.2 迭代计算

迭代是一种常用的有效算法。本节介绍几个典型迭代计算的程序实现。

4.2.1 牛顿迭代法

在第 1 章中提到计算非负实数算术平方根 (\sqrt{d}) 的牛顿迭代法^[1]：

$$\begin{cases} x_0 = 1 \\ x_n = x_{n-1} - \left(x_{n-1} - \frac{d}{x_{n-1}} \right) / 2 \quad n = 1, 2, 3, \dots \end{cases}$$

^[1]解方程 $f(x) = 0$ 的牛顿迭代公式为

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \quad n = 1, 2, 3, \dots$$

任取一满足 $f'(x_0) \neq 0$ 的 x_0 作为迭代计算的初值。对于预先给定的适当小的正数，如 $\epsilon = 10^{-8}$ ，若前后两次迭代计算的差别 $|x_n - x_{n-1}| < \epsilon$ ，则可将 x_n 当作方程的近似解。考虑方程 $f(x) = x^2 - d = 0$ ($d \geq 0$)，有导函数 $f'(x) = 2x$ ，即得到上述计算非负实数算术平方根的迭代公式。

为了与 cmath 中声明的标准函数区别，将函数原型设计成如下形式：

```
double mysqrt(double); // 形参类型是重要的，形参名称可缺省
```

函数定义如下：

源代码 4.3 自定义的平方根函数

```
1 double mysqrt(double x)
2 {
3     double delta, y=1, epsilon=1e-8;
4
5     do
6     {
7         delta = (x/y - y)/2;
8         y += delta;
9     }while(delta>=epsilon || delta<=-epsilon);
10    return y;
11 }
```

4.2.2 级数计算

1. 指数函数

由高等数学的无穷级数理论可知，一些常用的数学函数可表示成无穷级数的和。通过计算无穷级数的前若干项部分和达到近似计算函数值的目的。本书不讨论级数的收敛性、收敛速度等数学问题，而是直接应用数学结论进行程序实现。

例 4.3 根据如下级数计算指数函数的值。

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots \quad (-\infty < x < \infty)$$

【分析】 在计算机上计算无穷级数的部分和，到底需要计算多少项呢？一般采用精度控制法。取一个适当小的正数，例如， $\varepsilon = 10^{-8}$ ，当前后两次部分和的差（即某一项）的绝对值小于 ε 时便截断，并以此部分和当作级数的近似值。

级数求和计算是一个重复过程。一般地，根据级数项的递推公式依次计算出各项，再将其累加到部分和中。上述数学公式经过变形，可写成下面的迭代（递推）公式：

$$\begin{cases} s_0 = 1, p_0 = 1 \\ p_n = p_{n-1} \times x/n & n = 1, 2, 3, \dots \\ s_n = s_{n-1} + p_n \end{cases}$$

这种递推公式中涉及的运算是不变的，只有数据在发生变化，这就是所谓的循环不变式。

由于我们并不关心计算的中间结果，可将逐步计算出的项 p_n 及部分和 s_n “原地覆盖存储”：用计算得到的新值 p_n 及 s_n 分别覆盖前一步的旧值 p_{n-1} 及 s_{n-1} 。因此，只需要少量的存储单元便能实现上述计算。也就是说，将上述数学迭代公式中的下标去掉，就是算法的伪代码。

步1 $\text{epsilon}=1e-8;$
 $s = p = n = 1;$
 步2 若 $\text{fabs}(p) \geq \text{epsilon}$ 则循环
 $p *= x/n;$
 $s += p;$
 $n++;$
 步3 s 即为所求。

源代码 4.4 自定义的指数函数

```

1 double myexp(double x)
2 {
3     int n;
4     double s, p, epsilon=1e-8;
5
6     s = p = n = 1;
7     do
8     {
9         p *= x/n;
10        s += p;
11        n++;
12    }while(p>=epsilon || p<=-epsilon);
13
14    return s;
15 }
```

特别值得注意的是，计算第 n 项时，切勿分别计算 x^n 及 $n!$ ，然后计算它们的商。因为它们均可能是值非常大的数，可能出现数据溢出或出现较大的误差（参见练习 4 的 2(2)）；再做除法，又可能出现更大的误差。

2. 正弦函数

例 4.4 计算正弦函数的值。正弦函数的级数展开式为交错级数：

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + (-1)^{n-1} \frac{x^{2n-1}}{(2n-1)!} + \cdots \quad (-\infty < x < \infty)$$

【算法描述】

步1 $\text{epsilon}=1e-8;$
 $s = p = x;$
 $n = 3;$
 步2 若 $\text{fabs}(p) \geq \text{epsilon}$ 则循环
 $p *= -x*x/(n*(n-1));$
 $s += p;$
 $n += 2;$
 步3 s 即为所求。

源代码 4.5 自定义的正弦函数

```

1 double mysin(double x)
2 {
3     int n;
4     double s, p, epsilon=1e-8;
5     s = p = x;
6     n = 3;
7     do
8     {
9         p *= -x*x/(n*(n-1));
10        s += p;
11        n += 2;
12    }while(p>=epsilon || p<=-epsilon);
13    return s;
14 }
```

【综合测试】采用多文件结构测试上述自定义的数学函数。`mymath.cpp`, `mymath.h` 为一对文件(可将其移植到其他程序中)。将上述三个自定义函数录入到 `mymath.cpp` 文件中(略); 头文件 `mymath.h` 中的内容为函数原型, 作用是函数声明。

```

1 // mymath.h
2 #ifndef MYMATH_H
3 #define MYMATH_H
4
5 double mysqrt(double);
6 double myexp(double);
7 double mysin(double);
8
9 #endif
```

测试函数及主函数编辑到另一个源程序文件 `MyMathTest.cpp` 中。

源代码 4.6 自定义数学函数综合测试程序

```

1 // MyMathTest.cpp
2 #include <iostream>
3 #include <cmath>           // 用标准函数与之比较
4 #include "mymath.h"
5 using namespace std;
6
7 int mysqrtTest()
8 {
9     double x, err, max_err=0;
10    for(x=0; x<=100; x++)
11    {
12        err = mysqrt(x) - sqrt(x);      // 自定义函数与标准函数之差
13        if(fabs(err) > fabs(max_err))
14            max_err = err;             // 找出误差绝对值的最大值
15    }
16    cout << "mysqrt(x)-sqrt(x), (0≤x≤100), Max_err="
17    << max_err << endl;
18    return 0;
19 }
20 int myexpTest()
21 {
22     double x, err, max_err=0;
```

```

23     for(x=-10; x<=10; x++)
24     {
25         err = myexp(x) - exp(x);
26         if(fabs(err) > fabs(max_err))
27             max_err = err;
28     }
29     cout << "myexp(x)-exp(x), (-10 ≤ x ≤ 10), Max_err: "
30     << max_err << endl;
31     return 0;
32 }
33 int mysinTest()
34 {
35     const double PI=3.1416;
36     double x, err, max_err=0;
37     for(x=-2*PI; x<=2*PI; x+=0.1)
38     {
39         err = mysin(x) - sin(x);
40         if(fabs(err) > fabs(max_err))
41             max_err = err;
42     }
43     cout << "mysin(x)-sin(x), (-2PI ≤ x ≤ 2PI), Max_err: "
44     << max_err << endl;
45     return 0;
46 }
47
48 int main()
49 {
50     mysqrtTest();
51     myexpTest();
52     mysinTest();
53     return 0;
54 }
```

程序的运行结果：

```

mysqrt(x)-sqrt(x), ( 0 ≤ x ≤ 100),  Max_err : 7.45058e-009
myexp(x) - exp(x), (-10 ≤ x ≤ 10),  Max_err : -1.67e-009
mysin(x) - sin(x), (-2PI ≤ x ≤ 2PI), Max_err : -3.9544e-010

```

从结果可见，自定义的几个数学函数与标准数学函数的计算结果误差不超过所取的精度控制 $\varepsilon = 10^{-8}$ 。

4.2.3 最大公因数和最小公倍数

例 4.5 求两个整数 m, n 的最大公因数 (m, n) 和最小公倍数 $[m, n]$ 。

【分析】 首先有 $(m, n) = (|m|, |n|)$ 及 $[m, n] = [|m|, |n|]$ ；其次考虑非负整数 m, n

$$(m, n) = \begin{cases} m & n = 0, m \neq 0 \\ \text{不存在} & n = 0, m = 0 \\ (n, m \% n) & n \neq 0 \end{cases}$$

$$[m, n] = \begin{cases} \frac{m \times n}{(m, n)} & n \neq 0, m \neq 0 \\ \text{不存在} & n = 0 \text{ 或 } m = 0 \end{cases}$$

最后仅需讨论 m, n 全为正整数的情形。由带余除法：

$$m = q_1n + r_1 \quad 0 \leq r_1 < n$$

可知， m, n 的公因数集合与 n, r_1 的公因数集合相同，因而其最大公因数相等。同理，由

$$n = q_2r_1 + r_2 \quad 0 \leq r_2 < r_1$$

可知， $(m, n) = (n, r_1) = (r_1, r_2)$ 。重复这个过程（称为辗转相除法），由于 $0 \leq \dots < r_2 < r_1 < n$ ，必定有某 $r_k = 0$ ，则 r_{k-1} 即为所求。

根据上述讨论，计算两个非负整数的最大公因数的函数设计如下。

源代码 4.7 计算最大公因数

```

1 // 最大公因数 Greatest Common Divisor
2 unsigned int gcd(unsigned int m, unsigned int n)
3 {
4     unsigned int r;
5     while(n)
6     {
7         r = m%n;
8         m = n;
9         n = r;
10    }
11    return m;
12 }
```

m, n 的最小公倍数存在时其函数可如下定义。

源代码 4.8 计算最小公倍数

```

1 // 最小公倍数 Lowest Common Multiple
2 unsigned int lcm(unsigned int m, unsigned int n)
3 {
4     unsigned int gcd(unsigned int, unsigned int);
5         // 函数原型用于函数声明，声明 gcd 函数为局部函数
6     unsigned int g = gcd(m, n);
7     if(g==0)
8         return 0;
9     else
10        return m/g*n;    // 显然 m%g 为 0 (能除尽)，尽量避免可能由 m*n 造成的溢出
11 }
```

例如：

$m = 36, n = 14$ 时，

$$36 = 2 \times 14 + 8$$

$$14 = 1 \times 8 + 6$$

$$8 = 1 \times 6 + 2$$

$$6 = 3 \times 2 + 0$$

亦即 $(36, 14) = (14, 8) = (8, 6) = (6, 2) = (2, 0) = 2$ 。得到 2 为 36 与 14 的最大公因数。36 与 14 的最小公倍数为 $36/2 \times 14 = 252$ 。

4.3 标志变量的设计与应用

4.3.1 整除问题

例 4.6 编程输入一个整数，判断其是否能被 3、5 或 7 整除，并输出以下信息之一。

- (1) 能同时被 3, 5, 7 整除。
- (2) 能被两个数（要指出哪两个数）整除。
- (3) 仅能被一个数（要指出哪一个数）整除。
- (4) 不能被 3, 5, 7 整除。

【分析】 判断整数 n 能否被 3 整除的方法是判断 n 除以 3 的余数 $n \% 3$ 是否为 0。

【方法一】

源代码 4.9 整除问题

```

1 // 整除问题1.cpp
2 #include <iostream>
3 using namespace std;
4
5 void f(int n)
6 {
7     if(n%(3*5*7)==0)
8         cout << n << "能同时被3,5,7整除。";
9     else if(n%3==0 && n%5!=0 && n%7!=0)
10        cout << n << "仅能被一个数(3)整除。";
11    else if(n%3!=0 && n%5==0 && n%7!=0)
12        cout << n << "仅能被一个数(5)整除。";
13    else if(n%3!=0 && n%5!=0 && n%7==0)
14        cout << n << "仅能被一个数(7)整除。";
15    else if(n%3==0 && n%5==0 && n%7!=0)
16        cout << n << "能被两个数(3,5)整除。";
17    else if(n%3==0 && n%5!=0 && n%7==0)
18        cout << n << "能被两个数(3,7)整除。";
19    else if(n%3!=0 && n%5==0 && n%7==0)
20        cout << n << "能被两个数(5,7)整除。";
21    else
22        cout << n << "不能被3,5,7整除。";
23    cout << endl;
24 }
25
26 int main()
27 {
28     int n;
29
30     while(1)          // 设置循环，以便重复执行下面的程序段
31     {
32         cout << "\n请输入一个整数(负数退出)：";
33         cin >> n;
34         if(n<0) break;
35         f(n);
36     }
37     return 0;
38 }
```

【方法二】 先完成所有的判断，然后输出，使计算与输出两个环节分离。为此，我们设计一个标志量（整型变量 flag），记录 n 被 3, 5, 7 整除的情况。我们约定：这个标志变量的低 3 位（从最低位起）分别表示整数 n 被 3, 5, 7 整除的情况（用 0 表示不能整除，用 1 表示能整除），其余的位全置 0。共有 $2^3 = 8$ 种情形。这样，若标志值为 5（二进制为 101），则表示整数 n 能被 3, 7 整除，但不能被 5 整除。程序如下。

源代码 4.10 整除问题（推荐方法）

```

1 // 整除问题2.cpp
2 #include <iostream>
3 using namespace std;
4 int mod_3_5_7(int n)
5 {
6     int flag = 0;
7     if(n%3 == 0) flag += 1;           // 二进制 001 或 flag |= 1
8     if(n%5 == 0) flag += 2;           // 二进制 010 或 flag |= 2
9     if(n%7 == 0) flag += 4;           // 二进制 100 或 flag |= 4
10    return flag;
11 }
12 void show_result(int n)
13 {
14     switch(mod_3_5_7(n))
15     {
16         case 0: cout << n << "不能被3,5,7整除。"; break;
17         case 1: cout << n << "仅能被一个数(3)整除。"; break;
18         case 2: cout << n << "仅能被一个数(5)整除。"; break;
19         case 3: cout << n << "能被两个数(3,5)整除。"; break;
20         case 4: cout << n << "仅能被一个数(7)整除。"; break;
21         case 5: cout << n << "能被两个数(3,7)整除。"; break;
22         case 6: cout << n << "能被两个数(5,7)整除。"; break;
23         default: cout << n << "能同时被3,5,7整除。";
24     }
25     cout << endl;
26 }
27 int main()
28 {
29     int n;
30     while(1)
31     {
32         cout << "\n请输入一个整数(负数退出): ";
33         cin >> n;
34         if(n<0) break;
35
36         show_result(n);
37     }
38     return 0;
39 }
```

第二个程序中之所以将计算与输出分离，是因为有可能采用与 cout 不同的其他输出方法，此时，只要修改输出部分的语句，计算部分的语句不变；或者计算结果有其他用途而根本不进行输出操作。

程序第 14 行直接用函数的返回值（整型临时变量，用函数调用表达式本身表示）作为 switch 语句的入口表达式，其值只能是 0~7 之间的整数。最后的 default 情形后使用或者缺省 break 都是等效的。

4.3.2 三角形的周长及面积

例 4.7 给定三条线段的长度 a, b, c , 判断它们是否能够组成一个三角形, 若能则计算并“返回”该三角形的周长和面积两个值, 否则令其周长和面积全为零。

我们知道, 三角形的周长 (perimeter) 和面积 (area) 分别为

$$\text{perimeter} = a + b + c$$

$$\text{area} = \sqrt{s(s - a)(s - b)(s - c)}$$

其中 $s = (a + b + c)/2$ 。

【分析】设计一个函数完成该项计算。该项计算涉及的数据有三条线段的长度、周长、面积、辅助量 s 共六个数据。其中线段的长度单向传递给函数即可; 周长和面积需要由函数传回给函数调用者; 辅助量 (s) 在所论问题中不是独立的量, 不必在函数之间来回传递, s 应该作为函数的局部变量。

源代码 4.11 三角形的周长及面积

```

1 #include <cmath>
2 int triangle(double a,double b,double c,double &perimeter,double &area)
3 {
4     double s = (a + b + c)/2;
5     if(s>a && s>b && s>c)
6     {
7         perimeter = a + b + c;
8         area = sqrt(s*(s-a)*(s-b)*(s-c));
9         return 1;
10    }
11    perimeter = area = 0;
12    return 0;
13 }
```

我们为本函数设计了一个返回类型: 若 a, b, c 能构成一个三角形, 则函数返回 1 (相当于逻辑值 `true`), 否则返回 0 (相当于逻辑值 `false`)。这样的处理几乎不增加函数的设计工作量和难度, 却给使用该函数的程序员带来便利: 程序员不必在函数调用后通过判断近似表示的浮点型数据 `area` 是否精确地等于 0 来获知 a, b, c 能否构成一个三角形。

第 3.2.1 小节中所介绍的求解一元二次方程的函数 `int Solver(double a, double b, double c, double &x1, double &x2)`; 与上述函数是类似的。

4.4 单变量版“评委评分”程序设计

从本节开始, 将在后续的若干章节中逐步由浅入深地讨论所谓的大奖赛“评委评分”程序设计。随着这个具体例子的不断扩充逐步介绍 C++ 的语法及程序设计技术, 目的是使读者了解相关的 C++ 语法为什么需要 (必要性) 以及如何使用的问题。

4.4.1 问题描述及算法分析

例 4.8 【“评委评分”问题的提出】有 n 名选手参加的某大奖赛, 组委会聘请了 m 个评委为每一位参赛选手评分。每位选手在其所得的 m 个分数中按“去掉一个最高分,

去掉一个最低分，剩下的 $m - 2$ 个分数的算术平均分”确定最后得分。现要求编写一个程序，帮助大奖赛组委会计算并输出各选手的最后得分。

【算法分析】我们认为，计算机解题的过程与算盘解题过程是一脉相承的。设想这样的情景：大奖赛组委会决定请读者您充当竞赛现场的秘书，组委会仅提供一把算盘。那么如何处理这项工作呢？

(1) 这是一项重复劳动，对每一位选手的数据进行相同的处理（重复次数等于参赛选手人数 n ）。

(2) 对于每一位参赛选手，需要：

① 将算盘上的档分成五段，分别用于存放分数 (x)、最高分 (\max)、最低分 (\min)、总分 (\sum) 和平均分 (aver)。并且对每一位选手计算前都需要先将算盘盘面清零。

② 依次读取各位评委为当前选手所亮出的分数；边读边记录 (x)，边比较以得到目前的局部最高分 (\max)、局部最低分 (\min) 及总分的部分和 (\sum)。这一过程也是重复劳动（重复次数等于评委人数 m ）。当读完全部评委所亮的分数后，这些局部最高分、局部最低分、部分和，便演变成了该选手的最高分、最低分、总分。

③ 计算 $\text{aver} = (\sum - \max - \min) / (m - 2)$ 并宣布当前选手的最后得分。

其中求最高分、最低分的过程就如同“擂台赛”，下一位评委所亮分数与“擂主”比较，更大（或更小）者成为新的“擂主”，因而俗称这种处理方法为“打擂台算法”。

4.4.2 程序实现

1. 变量设计

实际问题中的数据需要以变量的形式存放在计算机内存中。另外还需要一些起辅助作用的变量。解决本问题所需要的主要变量如表 4-1 所示。

表 4-1 单变量版“评委评分”程序的变量设计

变量名	数据类型	存储类型	作用
referees	int	auto	存放评委人数
players	int	auto	存放参赛选手人数
x	double	auto	存放评委给参赛选手的分数
\max	double	auto	存放选手的最高分
\min	double	auto	存放选手的最低分
\sum	double	auto	存放选手的总分
aver	double	auto	存放选手的最后得分

程序的大致结构是二重循环，外层循环对应于参赛的各选手；内层循环对应于各评委给当前选手评分，以及计算该选手的最高分、最低分、总分，为计算最后得分做准备。

程序运行时，若输入的评委人数为 10，参赛选手人数为 20，则共有 $10 \times 20 = 200$ 个

原始分数。如此多的数据，若皆从键盘输入则是十分不便的。为此，程序中利用伪随机数发生器产生各个原始数据，并且将所产生的随机数变换到 8.0 ~ 10.0 之间且仅有一位小数。

2. 源程序代码

源代码 4.12 单变量版“评委评分”程序

```

1 // contest0.cpp          评委评分（单变量版）
2 #include <iostream>
3 #include <ctime>
4 #include <conio.h>           // 参见第 2.5 节
5 using namespace std;
6
7 int main()
8 {
9     int referees, players;
10    double x, min, max, sum, aver;
11
12    cout << "请输入评委人数:\u2022";
13    cin >> referees;
14    if(referees<=2)
15    {
16        cout << "评委人数太少。" << endl; // 无法继续运行
17        return -1;                  // 返回 -1 给操作系统表示出现了某种状况
18    }
19    cout << "请输入参赛选手人数:\u2022";
20    cin >> players;
21
22    time_t t;
23    srand(time(&t));
24    for(int i=0; i<players; i++)
25    {
26        cout << "\n\n====\u2022No.\u2022" << i+1 << endl;
27        min = 10;                   // 最低分一定不超过 10
28        max = sum = 0;              // 最高分一定高于 0
29        for(int j=0; j<referees; j++)
30        {
31            x = (80 + rand()%21)/10.0; // 产生8.0~10.0之间随机数
32            cout << x << '\t';       // 取代键盘输入语句 cin>>x;
33            if(x > max) max = x;      // 打擂台算法求最大值
34            if(x < min) min = x;      // 打擂台算法求最小值
35            sum += x;                // 计算总分
36        }
37        aver = (sum-max-min)/(referees-2);
38
39        cout << "\n\u2022去掉一个最高分\u2022" << max
40        << "\u2022分,\u2022去掉一个最低分\u2022" << min
41        << "\u2022分, 最后得分\u2022" << aver << "\u2022分。" << endl;
42        getch();
43    }
44    return 0;                      // 返回 0 给操作系统表示正常
45 }
```

本程序中没有定义全局变量，因此程序开始执行时直接执行主函数 main。

(1) 首先定义两个整型变量和 5 个双精度浮点型变量（意味着分配内存空间）。

(2) 第 12 行为输出提示信息，以指引操作员在接下来的输入语句中正确地输入。倘若没有适当的提示信息，操作员将可能无所适从。

(3) 执行第 13 行语句时，系统将暂停、等待操作员输入数据并以回车结束输入。倘若输入的数值太小，则将会输出一个信息指出：“评委人数太少。”，进一步遇到 `return -1;` 语句而终止程序，将 -1 返回给操作系统。操作系统或者用户可以检测该程序的返回码（此处为 -1）根据事先的约定以了解程序的执行情况，程序结束的原因。

(4) 若输入的评委人数大于 2，则继续执行第 19 行的语句，输出提示信息；及第 20 行的语句：暂停、等待操作员输入参赛选手人数并回车。

(5) 程序第 24~43 行为外层循环，循环控制变量 `i` 的初值为 0；步长为 1（即每循环一次，循环控制变量 `i` 的值增加 1）；终值为 `players-1`。外层循环将执行 `players` 次。循环结束时 `i` 的值为 `players`，即继续循环的条件不再成立之时。

(6) 立足循环体内（第 26~42 行）看变量 `i`，则 `i` 是暂时不变的，这里特指第 `i+1` 位参赛选手。第 27、28 行便相当于“算盘”盘面上的清零操作：为了计算最低分，我们首先在“擂台”上放置一个最大值（本例中 10.0 为最大值）；为了计算最高分，“擂台”上放置了一个低于 8.0 的分数 0；计算总分的累加器 `sum` 清零，这都是合理的或必需的。

(7) 第 29~36 行为内层循环，其功能是处理第 `i+1` 位选手的分数。其中，第 31 行产生一个随机数并转换成 8.0~10.0 之间的数，且仅包含一位小数。内层循环结束后，便计算该选手的“最后得分”（第 37 行），然后输出该选手的有关信息。

大赛结束后，“算盘”上留下来的数字应该有：`x` 是最后一位评委给最后一名选手的分数，`max`, `min`, `sum`, `aver` 是最后一名选手的相关数据。上述程序结束前，这些变量的值与“算盘”上的数值一样。此时变量 `j` 的值等于 `referees`，变量 `i` 的值等于 `players`。

若 `referees` 取 10, `players` 取 20，则有 200 个原始分数都存放在同一内存单元 `x`（反复覆盖前一个数据值）；也分别各仅用一个变量存放过各位选手的最高分、最低分、总分和最后得分。这就是我们将该程序称为“单变量版”程序的原因。遗憾的是，前 199 个原始分数，以及前 19 位选手的所有信息都不存在了（注：显示器属于外部设备，虽然此时在显示器上仍然显示着有关数据，但是计算机内存中已经只有最后少量的几个数据了）。

我们知道，计算机是一个拥有巨量“档”数的“算盘”，有大容量的内存供我们支配使用。如何改进上述程序，使所有原始数据得以保留以便再利用，且看下章分解。

4.5 趣味程序——击打字母游戏

例 4.9 击打字母游戏。在显示器第二行中部若干列的随机位置、随机地出现一个大写或小写英文字母。操作员在规定的延时时间（`delay=2000 ms`）内在键盘上击打对应的键，击打正确则可得分。可修改本程序使得分及格者过关（每关 20 个字母），每过一关则适当缩短延时时间。

源代码 4.13 击打字母游戏

```

1 // typing.cpp                                击打字母游戏
2 #include <iostream>
3 #include <iomanip>                         // 为了使用 setw
4 #include <ctime>
5 #include <conio.h>
6 using namespace std;
7
8 int hitted(int n, int delay)    // 共被要求击 n 个字符，延时 delay 毫秒
9 {                               // 返回正确击打字母的个数
10    int i, m=0, w;
11    char ch, key;
12    clock_t t;
13    cout << endl;
14    for(i=0; i<n; i++)
15    {
16        ch = 'A' + rand() % 26;           // 随机产生一个大写字母
17        if(rand() % 2) ch += 'a' - 'A'; // 可能被转换成小写字母
18        w = rand() % 40 + 20;           // 随机产生输出位置
19
20        cout << setw(w) << ch << '\r'; // 输出字符
21
22        t = clock() + delay;           // 确定延时终止时间
23        while(clock() <= t)           // 延时循环
24        {
25            if(kbhit())                // 判断是否有击键
26            {
27                key = getch();         // 接收键盘输出
28                if(key == ch)          // 判断是否正确
29                    m++;              // 累计正确字符数
30                else
31                    cout << "\a\r";      // 不正确时响铃
32                break;                // 不等待延时期满
33            }
34        }
35        cout << setw(w) << ' ' << '\r'; // 清除原来的字符
36    }
37    return m;
38 }
39
40 int main()
41 {
42     int n = 10, delay=2000, m;           // 可修改 n, delay 的值
43     time_t t;
44     srand(time(&t));
45
46     m = hitted(n, delay);
47
48     cout << "\n" << m << "/" << n << " = " << " "
49     << 100.0*m/n << "%" << endl;
50
51     cout << "Press  . . ." << endl;
52     while(getch()!=27);                // ESC 键的编码为 27
53     return 0;
54 }
```

4.6 小结

计算机内存是数据的载体，变量是计算机内存的抽象。计算机高级语言程序通过变量使用计算机内存，同时通过变量刻画所计算的问题。程序设计中，变量设计是关键，变量所代表的实际量的逐步变化标志着问题求解过程的递进。我们必须关注变量的值的变化规律——初始化值、当前值、最终值。每个变量都有自己独立的运行轨迹，把握了变量的变化规律就能很好地把握程序。我们称变量是程序中的“小精灵”。程序员是这些“小精灵”们的主人：创造它们、给它们住所、为它们命名、规定它们的角色，等等。控制好这些“小精灵”们是程序员的职责，也能体现程序员控制、驾驭它们的能力。

对于一个命题作文，不同的人写出不同的作品。程序设计也一样，对同一问题可以有多种不同的算法、不同的方案来解决。读者应该善于总结基本的程序设计方法和技巧，做到举一反三。

练习 4

1. 指出下列各程序或函数中的错误

(1) 希望判定整数是否为素数。

```

1 int isprime(unsigned int n)
2 {
3     unsigned int k;
4     for(k=2; k<n; k++)
5     {
6         if(n%k==0)
7             return 0;
8         else
9             return 1;
10    }
11 }
```

(2) 希望根据 $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$ 计算圆周率 π 。

```

1 double pi()
2 {
3     int n=1;
4     double s=0, p=1, epsilon=1e-8;
5     do
6     {
7         s += p;
8         n += 2;
9         p *= -1/n;
10    }while(p>=epsilon || p<=-epsilon);
11    return 4*s;
12 }
```

(3) 希望根据 $e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^{20}}{20!}$ 以级数前 20 项之和为 e^x 的近似值。

```

1 double Exp20(double x)
2 {
3     double s=0, p;
4     int i, n, q;
```

```

5   for(n=1; n<=20; n++)
6   {
7       p = q = 1;
8       for(i=1; i<=n; i++)
9       {
10          p *= x; q *= i;
11      }
12      s += p/q;
13  }
14 return s;
15 }
```

(4) 希望根据二分法求解方程 $f(x) = e^{-x} - (100x)^{-3} = 0$ 的在区间 $(1, 50)$ 之间的根 (事实上 $f(1) > 0, f(50) < 0$, 精度较高的解为 $23.2551, f(23.2551) = 9.5163 \times 10^{-20}$)。

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 double f(double x){ return exp(-x)-pow(100*x, -3); }
6 double g1(double a, double b)
7 {
8     double c;
9     if(a>b)
10    {
11        c=a; a=b; b=c;           // 使 a 为区间左端点
12    }
13    do
14    {
15        c = (a+b)/2;
16        if(f(c)==0) break;
17        if(f(a)*f(c)<0) b = c;
18        else             a = c;
19    }while(fabs(f(c))>=1e-8);
20    return c;
21 }
22 double g2(double a, double b)
23 {
24     double c;
25     if(a>b)
26    {
27        c=a; a=b; b=c;           // 使 a 为区间左端点
28    }
29    while(true)
30    {
31        c = (a+b)/2;
32        if(f(c)==0) break;
33        if(f(a)*f(c)<0) b = c;
34        else             a = c;
35    }
36    return c;
37 }
38
39 int main()
40 {
41     cout << g1(1, 50) << endl; // 输出25.5
42     cout << g2(1, 50) << endl; // 调用函数陷入无穷循环无法返回
}
```

```

43     return 0;
44 }
```

(5) 希望求两个正整数的最大公因数及最小公倍数。

```

1 void gcd_lcd(unsigned int m, unsigned int n
2             , unsigned int &gcd, unsigned int &lcm)
3 {
4     unsigned int temp;
5     while(n)
6     {
7         temp = n;  n = m%n;  m = temp;
8     }
9     gcd = m;
10    lcm = m*n/gcd;
11 }
```

(6) 给定 4 个数据类型相同的变量（如 double a1, a2, a3, a4;），希望将它们的值进行交换使其满足 $a_1 \leq a_2 \leq a_3 \leq a_4$ （即对它们进行从小到大排序）。

```

1 void Sort(double a1, double a2, double a3, double a4)
2 {
3     double temp;
4     if(a1 > a2){ temp = a1; a1 = a2; a2 = temp; }
5     if(a2 > a3){ temp = a2; a2 = a3; a3 = temp; }
6     if(a3 > a4){ temp = a3; a3 = a4; a4 = temp; }
7     if(a1 > a2){ temp = a1; a1 = a2; a2 = temp; }
8     if(a2 > a3){ temp = a2; a2 = a3; a3 = temp; }
9     if(a1 > a2){ temp = a1; a1 = a2; a2 = temp; }
10 }
```

2. 基本题

(1) 阅读程序，分析运行结果。

```

1 // Fibonacci.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int a=1, b=1, i;
8     cout << a << ", " << b;
9     for(i=3; i<16; i+=2)
10    {
11        a = a + b;
12        b = a + b;
13        cout << ", " << a << ", " << b;
14    }
15    cout << endl;
16    return 0;
17 }
```

(2) 编程计算 $1! + 2! + \dots + 15!$ ，要求输出每个部分和的值。

(3) 编程求 1000 之内的所有“完数”。“完数”是指一个数恰好等于它的所有因数（不包括该数本身）之和。例如，6 是完数，因为 6 的因数有 1, 2, 3, 6 且 $1+2+3=6$ 。

(4) 编写一个函数原型为 `int f(int n);` 的函数, 对于给定的正整数 n 计算并返回不超过 n 的被 3 除余 2, 并且被 5 除余 3, 并且被 7 除余 5 的最大整数, 若不存在则返回 0。

(5) 设 $f(x) = \cos(x) - x$, 容易验证 $f(0)f(1) < 0$, 根据连续函数的性质可知在区间 $(0, 1)$ 中, 方程 $f(x) = 0$ 至少有一个根。编程用二分法求其一个根, 要求精度为 10^{-8} 。【提示】二分法的计算过程: 计算区间中点 c 的函数值, 若 $f(c)$ 为 0, 则 c 即为所求; 否则判断 $f(a)$ 与 $f(c)$ 是否异号, 若它们异号则去掉右半区间 (即令区间右端点 $b=c$), 否则去掉左半区间 (即令区间左端点 $a=c$), 直到区间长度小于预先给定的精度控制值 (如 $\varepsilon = 10^{-8}$)。

(6) 计算 $\sqrt[3]{x}$ 的牛顿迭代公式为

$$\begin{cases} y_0 = 1, & x \neq -2 \\ y_n = y_{n-1} + \left(\frac{x}{y_{n-1}^2} - y_{n-1} \right) / 3 & n = 1, 2, \dots \end{cases}$$

试编写一个函数 `double cuberoot(double x);` 计算浮点型数值的立方根, 并编写一个主函数测试它 (包括计算正数、零、负数的立方根)。请注意: 当 $x = -2$ 时, 初值 y_0 不能取 1, 取 2 即可。

(7) 编程求解问题: 有一种细菌, 从其产生的第 4 分钟后, 每分钟都产生一个同种细菌。若某初始时刻仅有一个这种细菌, 那么此后第 n 分钟时共有多少个这种细菌? 【提示】将产生后 1, 2, 3 分钟及大于等于 4 分钟的细菌数分别记为 a, b, c, d , 然后分析其变化规律。

3. 扩展题 (程序设计竞赛题型)

判断算式正确性。

问题描述 给定一个算式, 该算式中只含一个四则运算符号, 操作数及结果均为整数。要求判断该算式的正确性 (规定: 除法必须除尽才可能正确)。

输入说明 输入数据有多行, 每行为一个算式。

输出说明 对于每一种情形, 要求先输出 “Case 序号: ”, 然后输出 `correct` (表示算式正确) 或者 `incorrect` (表示算式错误)。

输入样例 (有意不将算式编排整齐)

```
1+2 = -3
1 - 2 = -1
5/ 2 = 2
4 /2=2
```

输出样例

```
Case 1: incorrect
Case 2: correct
Case 3: incorrect
Case 4: correct
```

第5章 数组与指针

本章从改进第4章的单变量版“评委评分”程序出发，介绍一系列变量（数组、堆数组）版“评委评分”程序设计。学习本章后，读者应该理解为什么需要使用数组，掌握怎样使用数组的方法；理解指针的概念，掌握指针与堆数组的使用方法。本章还介绍C++中C-字符串的概念与操作方法。

5.1 数组

5.1.1 数组的定义

数组是具有相同数据类型，顺序排列并连续存放的一系列变量的集合，其中的变量皆被称为数组元素。定义数组的语法格式为

```
存储类型 数据类型 数组名[元素个数] = {初始化数据表};
```

数组是一系列变量的容器，元素个数是该容器的容量。定义数组意味着定义一系列变量，意味着分配一系列内存空间。数组各元素在计算机内存中连续存放，共占用“元素个数*sizeof(数据类型)”字节连续的内存空间。数组名是该内存空间的首地址，亦即首元素的地址。数组名所代表的内存地址值为地址常量，也就是说，数组名不能做左值，不能对数组名赋值。

定义数组时，数组的元素个数必须为整型常量（在编译时能确定其值）。元素个数不能小于1。定义数组时可以对数组的全部或部分元素进行初始化。有初始化数据时，元素个数可以缺省，此时系统将按实际的初始化数据的个数认定数组的元素个数。

数组及其各元素的存储类型、数据类型、生命期、可见性和作用域等皆与普通变量的规定和特性一样。例如：

```
double a[200];           // 全局数组。定义了 200 个全局变量
void f()
{
    const int N = 10;      // N 为常量
    int a[N];             // 局部自动数组。定义了 10 个局部自动变量
    int b[] = {12, 45, 32, 66, 56}; // 元素个数为 5
    // ...
}
```

常量数组是具有相同数据类型，顺序排列并连续存放的一系列常量的集合。例如：

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

其中初始化数据表中常量的个数应该与元素个数一致，局部常量数组必须提供全部元素的初始化值。

5.1.2 访问数组元素

相同的数据类型保证了这些变量在内存中所占用单元字节数相等、数据的存放格式一致，将它们连续存放有利于对这一系列变量进行统一地、集中地管理。数组元素是一系列相互独立的变量，它们有形式统一的变量名字（变量的原名）：

数组名[下标]

数组的下标始终从 0 起，到“元素个数-1”^[1]。下标可以通过表达式计算得到。例如：

```
const int N = 10;
int i, a[N];
for(i=0; i<N; i++)
    a[i] = i + 1;                                // 变量 a[i] 做左值
for(i=0; i<N; i++)
    cout << a[N-1-i] << ' ';                  // 数组元素做右值，表达式做下标
```

其中定义数组 a 相当于定义了 10 个变量，这些变量的原名依次为

a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9]

特别需要指出的是，C++ 不提供对整个数组的直接表达方法。数组定义后，企图用数组名 a 表示整个数组是不行的，因为数组名是一个地址常量。另一方面，企图用 a[10] 表示整个数组更是错误的，因为它表示“第 11 个元素”（下标已经越界），访问该元素（尤其是给该元素赋值）的后果将不可预料。

定义数组时，方括号 “[]” 不是运算符，它仅仅是一个记号。访问数组元素时所用的方括号 “[]” 是运算符。定义数组、访问数组元素操作中的方括号有不同的含义。读者应该善于从程序的上下文正确地辨别。指明数据类型者为定义数组（方括号中的常量为元素个数）。无指明数据类型者为访问数组元素（方括号中的表达式为下标）。注意：(float)a[9] 或者 float(a[9]) 不是定义数组而是读取第 10 个（下标为 9）元素的值（右值）进行类型转换运算。

既然 C++ 不提供对整个数组的直接表达方法，若要把握一个数组，就必须知晓数组的三要素：数据类型、首地址和元素个数。处理数组时这三要素缺一不可。此外，系统还必须提供地址运算和根据地址访问变量的运算（参见第 5.3 节）。

5.1.3 多维数组

一系列具有相同数据类型、相同元素个数，顺序排列并连续存放的数组的集合为二维数组。即二维数组是一维数组的数组。以此类推，可定义三维数组及更高维的数组。从这个意义上说，二维数组的概念是“推导”出来的而不是规定的。同时也说明，理解了一维数组，高维数组的概念与使用方法也就掌握了。

由于计算机内存编址的线性性，在实际应用中，常用所谓的升维/降维方法处理多维数组。

存储类型 数据类型 数组名[元素个数1][元素个数2] = {初始化值列表}；

^[1]C++ 的编译器并不检查程序中访问数组元素时的下标是否越界。因此，程序员应该尽量保证访问数组元素时下标不越界。

二维数组由(元素个数1×元素个数2)个元素组成,按照最右边的下标先变化的方式在内存中连续排列。俗称二维数组的元素按行存放。

例如, `double a[3][4]`;共定义了 $3 \times 4 = 12$ 个元素,这些元素在内存中的排列顺序为:`a[0][0], a[0][1], a[0][2], a[0][3], a[1][0], a[1][1], a[1][2], a[1][3], a[2][0], a[2][1], a[2][2], a[2][3]`。其中`a[0], a[1], a[2]`是3个“元素个数为4的一维数组”的数组名,仍然表示3个一维数组的起始地址。而`a`是二维数组的数组名表示首个一维数组的首地址——二级地址。

可将上述二维数组看成数组名为`a[0]`(一级地址),拥有12个元素的一维数组。`a[0][0], a[0][1], ……, a[0][11]`为其所有元素的“五花八门”的名字之一。

5.2 数组版“评委评分”程序设计

5.2.1 问题描述及算法分析

例 5.1 【问题的提出】 在第4章中讨论了所谓的某大奖赛“评委评分”问题的程序实现。在那个程序中,实现其基本功能时仅使用了数量极少的变量。现在,大赛组委会进一步提出按选手们的最后得分排列各选手的名次。

显然,第4章的单变量版程序无法实现此功能。因为比赛结束时仅有最后一名选手的部分信息还未被覆盖,大量的原始数据和计算结果已经被反复覆盖。

避免数据被覆盖的办法很简单:将不同的数据存放在不同的内存单元!因此,需要让系统为程序分配一系列内存单元——即定义一系列变量;并且要求这一系列变量是有组织的、能被集中而统一地管理。例如,希望这一系列变量有形式统一的变量名,以便在循环结构程序中方便地访问到它们。

回到“评委评分”问题上来,除了计算各选手的最后得分以外,还需要根据各选手的最后得分排列名次。因此,需要定义一个专门的一维数组作为容器存放各选手的最后得分(浮点型数据)、一个专门的一维数组作为容器存放各选手的名次(整型数据)。

为了使程序在最后能输出一张完整而详细的数据表,还可以考虑将所有的原始分数也都用数组来存放。由于C++编译器要求数组的元素个数为常量,故选择两个适当的整数分别作为评委人数的最大值及参赛选手数的最大规模(这里隐含该程序有最大处理能力的限制)。所需要的常量、变量和数组设计如表5-1所示。

表 5-1 数组版“评委评分”程序的常量及变量设计

标 识 符	数据类型	存储类型	描 述
MAX_REFEREES	int	auto	常量, 定为 50
MAX_PLAYERS	int	auto	常量, 定为 100
referees	int	auto	实际评委数, 键盘输入其值
players	int	auto	实际参赛选手数, 键盘输入其值
x[MAX_PLAYERS][MAX_REFEREES]	double	auto	二维数组, 存放原始分数
aver[MAX_PLAYERS]	double	auto	一维数组, 存放选手的最后得分
rank[MAX_PLAYERS]	int	auto	一维数组, 存放选手的名次
max	double	auto	计算最高分用
min	double	auto	计算最低分用
sum	double	auto	计算总分用

5.2.2 程序实现

源代码 5.1 数组版“评委评分”程序

```

1 // contest1.cpp      数组版“评委评分”程序
2 #include <iostream>
3 #include <iomanip>
4 #include <ctime>
5 using namespace std;
6
7 int main()
8 {
9     const int MAX_REFEREES = 50, MAX_PLAYERS = 100;
10    int referees, players;
11    double x[MAX_PLAYERS][MAX_REFEREES], aver[MAX_PLAYERS];
12    double min, max, sum;
13    int rank[MAX_PLAYERS], i, j;
14
15    cout << "请输入评委人数(例如:10):";
16    cin >> referees;
17    if(referees<=2 || referees > MAX_REFEREES)
18    {
19        cout << "评委人数太少或太多。" << endl;
20        return -1;           // 返回-1给操作系统表示出现状况之一
21    }
22    cout << "请输入参赛选手人数(例如:20):";
23    cin >> players;
24    if(players > MAX_PLAYERS)
25    {
26        cout << "参赛选手太多。" << endl;
27        return -2;           // 返回-2给操作系统表示出现状况之二
28    }
29    time_t t;
30    srand(time(&t));
31    for(i=0; i<players; i++)
32    {
33        cout << "\n\n====No." << i+1 << endl;
34        min = 10;
35        max = sum = 0;
36        for(j=0; j<referees; j++)
37        {
38            x[i][j] = (80 + rand() % 21)/10.0;
39            cout << x[i][j] << '\t';
40            sum += x[i][j];
41            if(x[i][j] > max) max = x[i][j];
42            if(x[i][j] < min) min = x[i][j];
43        }
44        aver[i] = (sum-max-min)/(referees-2);
45        cout << "\n去掉一个最高分" << max
46                << "分, 去掉一个最低分" << min
47                << "分, 第" << i+1 << "位选手最后得分"
48                << aver[i] << "分" << endl;
49    }
50
51    for(i=0; i<players; i++)           // 根据最后得分计算名次
52    {
53        rank[i] = 1;
54        for(j=0; j<players; j++)
55            if(aver[j] > aver[i]) rank[i]++;

```

```

56     }
57     cout << setprecision(3) << showpoint;
58     cout << "\nNo";
59     for(j=0; j<referees; j++)
60         cout << "uuuuu" << char('A'+j);
61     cout << "uuuAver.uRank" << endl;           // 输出表头
62     for(i=0; i<players; i++)
63     {
64         cout << setw(3) << i+1;
65         for(j=0; j<referees; j++)
66             cout << setw(6) << x[i][j];
67         cout << setw(6) << aver[i]
68             << setw(6) << rank[i] << endl; // 输出一行数据
69     }
70     return 0;
71 }
```

与第4章中的程序相比，程序的结构、算法大致上相同。原来的单变量x换成了数组元素x[i][j]，原来的aver换成了aver[i]。程序中增加了如下两个部分。

(1) 第51~56行，根据选手的最后得分aver[0]~aver[players-1]计算各选手的名次。计算结果存放在数组元素rank[0]~rank[players-1]中。这个算法的设计思想来源于分数公布后，各选手看榜时暗自计算自己名次的过程。该段程序代码模拟了这个过程：每位选手首先记住自己的最后得分(aver[i])，在开始扫描别人的分数前，认为自己是第1名(rank[i]=1)。然后与榜上所有选手的最后得分逐一比较，每遇到一个超过自己分数的人时，自己的名次便下降一位(rank[i]++)。扫遍所有选手后，名次也就确定了。这个算法很好地解决了出现并列名次时的排名问题：若有两个第二名，则不出现第三名。

(2) 按横行及纵栏输出所有数据，每一行为一名参赛选手的数据。纵栏包括选手的序号、评委所给出的分数、最后得分和名次等数据信息。在输出数据前，输出一个表头，表头中，用A, B, C, D, ……表示评委。为了使输出的数据对齐，程序中采用了I/O流格式控制符。

当程序运行时，若输入的referees值大于MAX_REFERES或者小于2；或者输入的players值大于MAX_PLAYERS，则超出本程序的处理能力而无法继续运行，只能输出一定信息后终止程序运行，返回操作系统。倘若输入的referees值小于MAX_REFERES，或者输入的players值小于MAX_PLAYERS，则显得预设的数组所占用的内存空间过大，造成宝贵的内存空间资源的浪费。解决这一问题的方法见第5.6节。

5.3 指针

C++语言能够直接操作内存地址。很多情形下，需要根据内存地址值及数据类型访问该内存单元处的数据。访问数组元素实际上就是利用数据类型、首地址和元素序号（元素地址与首元素地址的偏移量）进行的（即变量的间接名，参见第3.1.2小节）。

数据在内存中的具体存放格式、占用的字节数由数据类型决定。当我们获得某内存地址后，自然要求了解以此处为首地址的若干字节所存放的数据是什么类型（也将了解到占用多少字节），要求有一种表达方式表示该内存处的数据，以便访问它。

5.3.1 定义指针变量

存放地址值的变量被称为指针变量（简称为指针）。定义指针变量的语法格式为

```
存储类型 目标数据类型 * 指针变量名 = 初始地址值;
```

可以定义无类型的指针变量：

```
存储类型 void * 指针变量名 = 初始地址值;
```

无类型的指针变量只能存放一个内存地址值，不能直接进行指针运算。必须将其显式转换成某种具体的数据类型的指针后才能进行运算。

因为“定义变量就意味着为该变量分配内存空间”，那么，对于指针变量而言，无论以何种目标数据类型定义指针变量，系统均为每个指针变量分配同样字节数的内存单元，即 `sizeof(void *)` 字节^[1] 的内存单元。该单元能够存放，也只能存放一个内存地址值。可用语句

```
cout << sizeof(void*) << endl;
```

查看所使用的系统中一个地址值需要用多少字节的单元来存放。

若指针变量的值是某个变量存储单元的首地址（简称为该变量的地址），我们称指针指向了该变量，亦称该变量是**指针的目标变量**。

指针变量的存储类型及初始化、生命期、可见性和作用域等属性与普通变量的属性规定一致。

定义指针变量时若不显式地进行初始化，根据该指针变量的存储类型，则其存放的初始地址值是不可预知的（称为指向不明）或为 0（即 `NULL`，称为空指针）。此时，指针的目标变量均是不可预知的，是不宜访问的。贸然访问指向不明或空指针的目标变量是危险的，可能引起不可预料的严重后果（参见第 5.5.1 小节例 5.4）。

5.3.2 指针运算

1. 获取变量的地址

C++ 提供的获取变量地址值的操作符为“`&`”^[2]（单目运算，优先级为 3）。

2. 访问指定内存处的数据

C++ 提供的根据内存地址及数据类型访问该内存处数据的操作符为“`*`”^[3]（单目运算，优先级为 3）。我们称该运算符为**取目标内容运算符**（简称**取内容运算符**），或者称为**间接访问运算符**。

例如：

^[1]Visual C++ 及 MinGW C++ 中 `sizeof(void*)` 均为 4。

^[2]运算符“`&`”作为双目运算时为二进制位与运算符（优先级 10）；作为声明引用的记号（包括引用型形式参数、引用返回类型）也使用符号“`&`”；此处为取变量地址运算符。

^[3]运算符“`*`”作为双目运算时为算术乘法（优先级 5）；作为定义指针变量的记号（包括指针型形式参数、返回指针）也使用符号“`*`”；此处为取目标内容运算符。读者应该善于从程序的上下文加以辨别。

```

1 int a[10], b, *pa=a;      // 定义一个指针变量并用数组名初始化之
2 double x, y[10], *px=&x;  // 定义一个指针变量并用 x 的地址初始化之
3 int *pb;                  // 定义的指针变量未初始化，目标不明
4 double *py;                // 定义的指针变量未初始化，目标不明
5 pb = &b;                  // 用变量的地址给指针变量赋值
6 py = y;                   // 用数组名（地址值）给指针变量赋值

```

上面出现的星号“*”全部是定义指针变量的记号，符号“&”全是取变量地址运算符。

表达式“*指针变量名”可视为指针变量的目标变量名，可称其为变量的间接名，可以做左值或右值。例如，接上面的语句：

```

7 *pa = 100;           // 相当于给 a[0] 赋值 100
8 *pb = 200;           // 相当于给变量 b 赋值 200
9 pb[0] = 200;         // 与上一行等效，此时下标只能取 0
10 *px = 3.14;          // 相当于给变量 x 赋值 3.14
11 px[0] = 3.14;        // 与上一行等效，此时下标只能取 0
12 *py = 2.718;         // 相当于给变量 y[0] 赋值 2.718

```

在上面出现的星号“*”全部是取目标内容运算符。`*pa`, `*pb`, `*px` 和 `*py` 分别被称为变量 `a[0]`, `b`, `x` 和 `y[0]` 的间接名。这里，可以看到指针变量目标数据类型的作用。

3. 地址的偏移

C++ 中地址值可以与整数进行加、减运算，运算的结果为一个新地址值。接上例：

```

13 pa++;             // 相当于 pa = pa+1; 改变了指针 pa 的目标指向
14 *pa = 101;          // 相当于给 a[1] 赋值 101, 给新目标变量赋值
15 pa += 4;            // 相当于 pa = pa+4; 改变了指针 pa 的目标指向
16 *pa = 105;          // 相当于给 a[5] 赋值 105, 给新目标变量赋值
17 *(py+5) = 5.5;     // 相当于给变量 y[5] 赋值 5.5, 指针 py 的指向不变
18 *(py+9) = 9.9;      // 相当于给变量 y[9] 赋值 9.9, 指针 py 的指向不变

```

需要特别指出的是：地址值与整数 `n` 相加（减）得到的新地址值是在原地址值上增加（减少）了 `n*sizeof(数据类型)` 字节地址。C++ 语言的这项规定实质上是将困难留给编译器，将方便让给了程序员。这种规定与数组的下标运算遵循同样的规则，即整数 `n` 表示数据的项数。

两个目标数据类型相同的指针可以相减，结果为两个地址之间所存放的数据类型数据的个数。该运算仅对一片连续的内存空间中的两个地址值相减有实际意义。

4. 指针与数组

定义了如下数组和指针，并将指针指向数组首地址：

```
double a[10], *p=a;
```

此时变量原名（第 1 排）及其间接名（第 2~4 排）依次对应为

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
*a	*(a+1)	*(a+2)	*(a+3)	*(a+4)	*(a+5)	*(a+6)	*(a+7)	*(a+8)	*(a+9)
p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]	p[7]	p[8]	p[9]
*p	*(p+1)	*(p+2)	*(p+3)	*(p+4)	*(p+5)	*(p+6)	*(p+7)	*(p+8)	*(p+9)

变量地址的四种表示方法分别依次为

a	a+1	a+2	a+3	a+4	a+5	a+6	a+7	a+8	a+9
&a[0]	&a[1]	&a[2]	&a[3]	&a[4]	&a[5]	&a[6]	&a[7]	&a[8]	&a[9]
p	p+1	p+2	p+3	p+4	p+5	p+6	p+7	p+8	p+9
&p[0]	&p[1]	&p[2]	&p[3]	&p[4]	&p[5]	&p[6]	&p[7]	&p[8]	&p[9]

可以看到，利用指针变量可能对计算机内存中的任何地方进行操作，俗称“指向哪里就可以打到哪里”。指针的这种特点给程序设计带来极大的方便，尤其是访问一片连续的内存空间中的数据集合。但是，指针操作不当又会给程序运行带来很大的隐患和危害（例如，指针变量所指向的目标不明、空地址、数据集合越界或目标不可写等）。因此，访问指针变量的目标前，必须先使该指针变量获得一个安全（即可以被访问）的地址。另外，变量原名的生命期应该不短于其间接名的生命期，因为当目标变量不存在时，指向目标的指针是无用的，贸然地访问该指针目标变量内容是危险的。例如：

```

1 void f()
2 {
3     double x, *px;      // 指针未初始化，指向目标不明（不可预知）
4     *px = x;            // 错误！贸然访问指针的目标
5     *px = 3.14;
6 }
```

请注意：上述第3行语句中的“*”为定义指针变量的记号（不是运算符）。第4行语句中的“*”为运算符，该语句直接访问指向不明的指针目标变量。这种错误是初学者最容易犯的。第4行应该改为给指针变量赋一个安全的地址值，例如使其指向变量x：

```
px = &x;
```

之后的第5行才是正确的语句，间接地给x赋值3.14。

取变量地址运算与指针变量取内容运算是一对互逆运算。例如：

```

double x, *px;      // 指针未初始化，指向目标不明
px = &x;            // 明确指针的目标
if(*(&x) == x && &(*px)==px && *(&px)==px) // 条件成立，输出Ok
    cout << "Ok" << endl;
else
    cout << "Sorry" << endl;
```

须注意的是，对于非指针变量x，表达式`&(*x)`是错误的。另外，即使指针的目标不是数组元素而是一个普通变量，指针px皆可写成`&px[0]`；指针的目标变量（间接名）*px皆可写成`px[0]`。

5.4 动态变量和动态数组——堆变量和堆数组

堆(heap)是程序数据区中的一种（另外两种分别为全局数据区、栈区）。我们将堆内存视为可按需分配、随用随取、用后归还的一种系统资源。实际上，堆是一种数据结构（即数据的一种组织及使用方式）。堆内存空间一般不属于某个程序所专有，所有的程序（特别指多个程序同时运行时）都可以多次向操作系统申请使用堆内存空间。因此，在程

序中，当所申请的堆内存不再需要时，应该及时将其释放——归还给操作系统，以利其他程序或本程序其他时候使用。

C++ 提供了堆内存的操作方法。申请堆内存空间的运算符为 `new`，释放堆内存空间的运算符为 `delete`。它们的语法格式如下。

(1) 申请堆变量（动态变量）：

```
new 数据类型
```

(2) 申请堆数组（动态数组）：

```
new 数据类型 [元素个数]
```

执行上述运算表达式时，系统将在堆内存空间中“切下”一片连续的内存空间。该片空间占“`sizeof(数据类型)`”字节（对应于申请堆变量）或“`元素个数*sizeof(数据类型)`”字节（对应于申请堆数组）。若分配堆内存成功，则表达式的结果为该片堆内存空间的首地址；若分配内存空间不成功（如，元素个数为非正数或者太大）则表达式的结果为 `NULL`（空地址）。`new` 操作的结果是否为空地址常作为申请堆内存是否成功的判断依据。

动态变量没有原名，动态数组也无数组原名，只能利用 `new` 返回的地址值使用其间接名。因此，`new` 操作所返回的堆空间首地址需要妥善保存并保护好。通常用一个目标数据类型相同的指针变量存储该首地址值，丢失或改变该指针变量的指向将导致难以访问堆变量或堆数组的元素；也将导致所申请的堆空间无法被释放而造成内存泄漏。

当程序中不再需要所申请的堆内存时，应该及时归还给操作系统，称为释放堆内存空间。释放堆内存空间的运算符为 `delete`，其语法格式如下。

(1) 释放堆变量：

```
delete 指向堆地址的指针变量
```

(2) 释放堆数组：

```
delete [] 指向堆地址的指针变量
```

由于堆变量、堆数组的生命期取决于 `new` 与 `delete` 操作，因此堆变量被称为动态变量、堆数组被称为动态数组，而堆空间也被称为动态空间。例如：

```
int n = 10; // 定义一个整型变量，并初始化
double *ptr, *array; // 定义两个指向双精度浮点型数据的指针变量

ptr = new double; // 申请分配堆内存，即定义一个动态变量
// ptr 存放首地址，*ptr 为间接名，没有原名
array = new double[n]; // 在堆中申请一片连续空间，即定义动态数组
// array 存放首地址
*ptr = 3.1415926; // 使用“间接名”访问堆变量
array[0] = 1*3.1415926; // 访问堆数组元素，亦即 *array=1*3.1415926;
array[1] = 2*3.1415926; // 亦即 *(array+1) = 2*3.1415926;
// ...
array[9] = 10*3.1415926; // 亦即 *(array+9) = 10*3.1415926;
delete ptr; // 释放堆变量
delete [] array; // 释放堆数组
```

值得指出的是动态数组的元素个数可以是变量，或一个表达式的值，不必要求它是在编译时就须确定的常量，这样便能做到在程序运行过程中，根据运行时所确定的元素个数按需分配内存资源。

5.5 地址值在函数之间传递

指针和函数都是 C/C++ 语言中的重要概念。显然，地址值（右值）、指针变量（左值）在函数之间的传递是重要的。本节的内容是指针变量套用函数参数传递、返回的直接推论，并非新规定。所需注意的是形参指针变量的目标变量是谁。

5.5.1 传递地址值——值传递

指针型形式参数本质上是数据值（实参右值）的传递。调用函数时，建立形参指针变量，占用 `sizeof(void*)` 字节，用实参（一个地址值）初始化之。函数体内对形参指针的指向改变不影响实参地址（属于单向的值传递，参见例 5.3）。

由于函数获得了某地址值以及相关数据类型，便能用指针的目标变量（即利用变量的间接名）间接地访问变量（称为间接双向传递，参见例 5.2）。

1. 变量的地址值作为实参

例 5.2 交换实参目标的值。

源代码 5.2 交换实参目标的值

```

1 #include <iostream>
2 using namespace std;
3 void swap_p(double *a, double *b)
4 {
5     double temp;
6     temp = *a;      // 等价于 temp = a[0]; *a 为实参目标变量的间接名
7     *a = *b;        // 等价于 a[0] = b[0]; *b 为实参目标变量的间接名
8     *b = temp;      // 等价于 b[0] = temp;
9 }
10 int main()
11 {
12     double x=3, y=5;
13     cout << "x=" << x << ",y=" << y << endl;
14     swap_p(&x, &y);    // 实参值传递(传递地址值)
15     cout << "x=" << x << ",y=" << y << endl;
16     return 0;
17 }
```

程序的运行结果为：

```
x = 3, y = 5
x = 5, y = 3
```

本例中，实际参数 `&x`, `&y` 分别为变量 `x`, `y` 的地址值用于初始化指针型形式参数（严格匹配）。`swap_p` 函数在获得了主函数中变量 `x`, `y` 地址后，通过其间接名（`swap_p` 函数体内不可见 `main` 函数中的变量 `x`, `y`，不知其原名，只能通过间接名）对主函数中变量 `x`, `y` 进行了读写访问。

请将本例与第 3.2.1 小节例 3.3 中的函数 swap 进行比较。例 5.2 的程序是一个 C 语言风格的程序，本程序的语句“不太通顺”：第 14 行语句的“字面意思”似乎是交换变量的地址值！这是由于 C 语言没有引用的概念，交换两个变量的值的函数只能采用上述暴露变量的地址给函数，允许函数根据地址间接操作变量来实现。

例 5.3 错误使用指针型形参的反例（无用的函数）。

源代码 5.3 无法交换实参指针指向（反例）

```

1 #include <iostream>
2 using namespace std;
3 void swap2(double *a, double *b)
4 {
5     double *temp;
6     temp = a;
7     a = b;
8     b = temp;
9 }
10 int main()
11 {
12     double x=3, y=5;
13     double *px = &x, *py = &y;
14     swap2(px, py);           // 等价于 swap2(&x, &y);
15     cout << "x=" << x << ", y=" << y << endl;
16     cout << "*px=" << *px << ", *py=" << *py << endl;
17     return 0;
18 }
```

程序的运行结果：

```
x = 3, y = 5
*px = 3, *py = 5
```

可见，上述函数既不能交换实参目标的值，也不能交换实参（指针）的指向，请对照本例与第 3.2.1 小节例 3.3 中的函数 swap1。

例 5.4 借道不明处交换实参目标的值（反例，危险的函数）。

源代码 5.4 借道不明处交换实参目标的值（反例）

```

1 #include <iostream>
2 using namespace std;
3 void swap3(double *a, double *b)
4 {
5     double *temp;    // 指针未初始化，目标不明
6     *temp = *a;      // 贸然访问指针 temp 的目标
7     *a = *b;
8     *b = *temp;
9 }
10 int main()
11 {
12     double x=3, y=5;
13     swap3(&x, &y);
14     cout << "x=" << x << ", y=" << y << endl;
15     return 0;
16 }
```

因为局部自动指针变量 `temp` 未初始化，其指向不明。其目标变量 `*temp` 可能是内存中十分重要的单元，直接将 `double` 型数据 `*a` 写入其中可能会导致重大错误。或者该处不允许写入。因此，该程序可能出现运行时错。

可以将函数 `swap3` 修改为

```

1 void swap3(double *a, double *b)
2 {
3     double *temp = new double;
4     *temp = *a;    *a = *b;    *b = *temp;
5     delete temp;
6 }
```

这样修改后的函数是正确的，它借道堆空间实现参目标变量值的交换。但有“大动干戈”或“弄巧成拙”之嫌。因而，例 5.2 中的 `swap_p` 函数借道局部自动变量 `temp` 是最明智的。

本章中关于交换变量数值的函数是 C 语言程序设计的经典例子。由于 C++ 语言增加了引用的概念和处理方法，交换变量数值的函数还是以例 3.3 为佳。

2. 数组在函数之间传递

将数组的所有元素值都按值传递方式传递给函数（建立数组所有元素的副本）是不必要的，而且是耗时费空间的。由于 C++ 不提供对整个数组的直接表达方法，而不能建立数组的引用，所以要想将数组的一系列变量“传递”给函数，实际上只要将数组的三要素传递给函数（设定数据类型，暴露首地址，告知元素个数）即可。这是一种间接的传递方式，这种方式的时间效率高（速度快，仅传递一个地址值和一个整数值）、空间效率高（不建立数组元素的副本，不复制数组元素）。

例 5.5 数组（数组首地址）在函数之间传递。

```

1 #include <iostream>
2 using namespace std;
3
4 void display(const double *a, int n)
5 {
6     cout << a[0];
7     for(int i=1; i<n; i++)
8         cout << ", " << a[i];
9     cout << endl;
10 }
11
12 int main()
13 {
14     double a[10];
15     for(int i=0; i<10; i++)
16         a[i] = 10*i;
17     display(a, 10);
18     display(a+2, 5);
19     return 0;
20 }
```

程序的运行结果：

```
0, 10, 20, 30, 40, 50, 60, 70, 80, 90
20, 30, 40, 50, 60
```

从程序的运行结果可见，函数所获得的数据类型、首地址、元素个数不一定是主函数中数组的首地址、元素个数。函数体内从给定的地址处开始、处理接下来的连续 n 个元素。

若执行函数调用 `display(a+5, 6)` 则函数体内访问的第 6 个元素已经越界，将输出一个不可预料的数据值。出现这种错误的原因在函数调用，而不应归咎于函数定义，因为函数没有义务，也不可能判断调用本函数时所提供的实际参数的合理性和正确性。

指向常量的指针（常量指针）的定义格式为：

```
const 数据类型 * 常量指针变量名 = 初始化地址值;
```

常量指针可以指向常量，也可以指向变量。从指针的角度看常量指针的目标时，目标为常量（参见例 5.5）。又如：

```
const int a = 3;
int b = 5;
const int *p = &a;
p = &b;           // 指针 p 的指向可以修改
*p = 10;         // 错误。*p 为常量。尽管其目标为变量 b
int *p1 = &a;    // 错误。不能用非常量指针指向常量
```

上述语句中，常量指针 p 被初始化为常量 a 的地址。后改变其指向使之指向变量 b。从指针的角度上看，其目标 *p 为常量，即所指之处皆常量。`*p` 不能做左值。b 是变量仍然可以做左值，即可以将变量视为常量，但不允许将常量视为变量（一个例外，参见第 5.7.2 小节的注）。

函数的指针型形式参数用 `const` 加以限制，即形式参数为常量指针，能有效地保护实参地址处的数据不被修改，同时也给程序员信心使程序员放心地调用该函数（参见例 5.5）。进一步，还可用常量数组名做实参，处理真正的常量数组。

通过上述一系列函数可见，指针型参数本质上是数值传递（传递地址值），这种形式的参数不可能改变实参指针的指向，但是可以修改指针的目标变量的值。可将指针型形式参数的这种特性称为数据间接地双向传递。有时，需要保护实参地址处的目标变量不被修改，可使用指向常量的指针型形式参数。

5.5.2 传递指针变量——引用传递

应用程序有时需要通过函数修改作为实际参数的指针变量的指向，即要求直接的双向传递指针变量。实现这种要求需要将函数的形式参数设计成指针引用型的。

例 5.6 传递指针的引用。

```
1 // 引用传递指针变量.cpp
2 #include <iostream>
3 using namespace std;
4
5 void GetMemory(double *&p, int n)      // 动态一维数组
6 {
```

```

7     p = new double[n];           // 指针 p 是实参指针变量的别名
8 }
9
10 void FreeMemory(double *&p)
11 {
12     if(p != NULL)
13         delete [] p;          // 释放空间，指针 p 的指向不变
14     p = NULL;                // 修改指向，将指针赋值成空指针
15 }
16
17 int main()
18 {
19     double *p;                // 局部自动指针变量，初始指向不明
20     int n = 10;               // n 为变量
21     GetMemory(p, n);         // 申请堆内存空间，改变 p 的指向
22     for(int i=0; i<n; i++)
23         cout << (p[i] = 10*i) << " ";
24     cout << endl;
25     FreeMemory(p);          // 释放堆数组，同时使 p 为空指针
26     return 0;
27 }

```

主函数中定义指针变量 p 未初始化，其指向不明。调用函数 `GetMemory`，在该函数体内通过别名使主函数中的指针变量指向所申请的堆内存某地址。若不传递指针的引用，则 `GetMemory` 函数体内申请的堆内存的首地址将丢失（请读者在计算机上验证），其后也无法释放。

5.5.3 返回地址

函数的返回类型可以设计成指针（地址值）返回。这种函数调用本身只能做右值（函数调用表达式是一个临时指针变量），而函数调用的目标可能成为左值。

例 5.7 返回地址值的函数示例。

```

1 // 返回地址值.cpp
2 #include <iostream>
3 using namespace std;
4
5 double *ElementAddr(double *p, int size, int index)
6 {                           // 返回数组指定元素的地址值
7     if(index<0 || index >= size)
8         return (double*)NULL;
9     return p+index;
10 }
11
12 int main()
13 {
14     const int N = 3;
15     double a[N];           // 元素个数须为常量
16     int i;
17     for(i=0; i<N; i++)
18     {
19         // ElementAddr(a, N, i)++; 错误！函数调用本身不能为左值
20         *ElementAddr(a, N, i) = 10*i; // 函数调用的目标成为左值
21         cout << *ElementAddr(a, N, i) << '\t'
22             << ElementAddr(a, N, i)[0] << '\t'

```

```

23             << ElementAddr(a, N, 0)[i] << endl;
24     }
25     return 0;
26 }
```

程序的运行结果:

0	0	0
10	10	10
20	20	20

还可以将函数的返回类型设计成指针的引用，这样的函数调用不仅能够使其目标成为左值，函数调用本身也能成为指针变量左值，用于改变所返回的指针变量的指向（通过函数的形参“返回”指针变量新的指向、通过函数引用返回类型返回指针变量新的指向的程序，请参见第6章第6.9.5小节动态二维数组）。本章例5.6实为处理动态一维数组。

5.6 堆数组版“评委评分”程序设计

例5.8 堆数组版“评委评分”程序设计。

为了更合理地使用计算机内存空间，做到“按需分配”“有借有还”。可利用C++的new操作在堆内存中创建动态数组用于存放一系列数据。为此，程序中所需的变量设计如表5-2所示。

表5-2 堆数组版“评委评分”程序的变量设计

标识符	数据类型	存储类型	描述
referees	int	auto	实际评委人数，从键盘输入其值
players	int	auto	实际参赛选手人数，从键盘输入其值
*x	double	auto	指针，将存放new操作返回的堆中某地址值
*aver	double	auto	指针，将存放new操作返回的堆中某地址值
*rank	int	auto	指针，将存放new操作返回的堆中某地址值
max	double	auto	计算最高分用
min	double	auto	计算最低分用
sum	double	auto	计算总分用

实现上述计算的C++源程序如下：

源代码5.5 堆数组版“评委评分”程序

```

1 // contest2.cpp      指针—动态数组版“评委评分”程序
2 #include <iostream>
3 #include <iomanip>
4 #include <ctime>
5 using namespace std;
6
7 int main()
8 {
9     int referees, players;
10    double *x, *aver, min, max, sum;
11    int *rank, i, j;
12 }
```

```

13     cout << "请输入评委人数(例如:10):";
14     cin >> referees;
15     if(referees<=2)
16     {
17         cout << "评委人数太少。" << endl;
18         return -1;
19     }
20     cout << "请输入参赛选手人数(例如:20):";
21     cin >> players;
22
23     x = new double[referees * players];      // 申请使用堆内存
24     aver = new double[players];
25     rank = new int[players];
26     if(x==NULL || aver==NULL || rank==NULL) // 若申请不成功
27     {
28         if(x) delete [] x;
29         if(aver) delete [] aver;
30         if(rank) delete [] rank;    // 释放可能已申请成功的部分空间
31         return -2;                // 终止程序，返回到操作系统
32     }
33     time_t t;
34     srand(time(&t));
35     for(i=0; i<players; i++)
36     {
37         cout << "\n\n====No." << i+1 << endl;
38         min = 10;
39         max = 0;
40         sum = 0;
41         for(j=0; j<referees; j++)
42         {
43             x[i*referees+j] = (80 + rand() % 21)/10.0;
44             cout << x[i*referees+j] << '\t';
45             sum += x[i*referees+j];
46             if(x[i*referees+j]>max) max = x[i*referees+j];
47             if(x[i*referees+j]<min) min = x[i*referees+j];
48         }
49         aver[i] = (sum-max-min)/(referees-2);
50         cout << "\n去掉一个最高分" << max
51             << ",去掉一个最低分" << min
52             << "分, 第" << i+1 << "位选手平均得分"
53             << aver[i] << "分" << endl;
54     }
55     for(i=0; i<players; i++)           // 根据最后得分计算名次
56     {
57         rank[i] = 1;
58         for(j=0; j<players; j++)
59             if(aver[j] > aver[i]) rank[i]++;
60     }
61     cout << setprecision(3) << showpoint;
62     cout << "\nNo";
63     for(j=0; j<referees; j++)
64         cout << "....." << char('A'+j);
65     cout << ".....Aver.Rank" << endl;    // 输出表头
66     for(i=0; i<players; i++)
67     {
68         cout << setw(3) << i+1;
69         for(j=0; j<referees; j++)
70             cout << setw(6) << x[i*referees+j];
71         cout << setw(6) << aver[i]

```

```

72             << setw(6) << rank[i] << endl;
73     }
74     delete [] x;
75     delete [] aver;
76     delete [] rank;           // 释放所有申请的堆内存空间
77     return 0;
78 }
```

该程序不仅做到了需要多少内存空间就申请使用多少空间，而且还突破了第 5.2 节数组版程序中“最大处理能力”的限制，本节的程序与之相比有以下不同。

(1) 定义变量时，除了普通变量外，定义了三个指针变量，系统自然会为它们分配内存空间，每个指针变量均各仅占用 `sizeof(void*)` 字节，用于存放一个内存地址值。定义这三个局部自动指针变量时未对其进行初始化，其初始指向是不可预知的。直到程序中执行 `new` 操作（第 23~25 行）并将所返回的地址值赋值给这些指针变量后，这些指针变量才有了明确的指向。须指出的是，这三个地址值不能被丢失，否则将无法正确地用 `delete` 操作释放程序运行时所申请的这几部分堆内存空间。

(2) `new` 操作不成功时所返回的值为 `NULL`，程序中利用这一点进行判断。当申请堆内存不成功时，则需要释放那些已经申请成功的堆内存，然后终止程序的执行，返回到操作系统，以保证不造成内存泄漏。

(3) 对于堆数组中的元素，如同普通数组元素一样，可以用“首地址[下标]”进行访问，如程序中的 `aver[i]`，`rank[i]`。本程序中的 `new` 操作所返回的堆地址是一个一级指针，程序中用 `x[i*referees+j]` 访问堆数组中的元素是匹配的。这种方法称为升维处理方法——用一维数组模拟二维数组。

(4) 程序结束前，不再需要这些堆数组了，便用 `delete[]` 操作将它们释放，返还给操作系统。这一点是重要的。

5.7 字符数组与 C-字符串

实际上，数组是利用已经存在的数据类型进一步构造的数据集合类型。它同时提供了数据集合中元素的存储及访问方式，属于一种数据结构——顺序表的一种简单形式。这是因为数组的各元素在内存中按顺序连续存放而得名。把握一个数组（包括堆数组），需要把握其数据类型、首地址、元素个数。有了这三要素，数组的一切便在我们的掌控之中。字符串是一系列有序字符的集合。下面首先讨论字符数组，再介绍字符串。

5.7.1 字符数组

数据类型为字符型（`unsigned char` 或 `char`）的数组被称为字符数组。它是一系列字符变量的集合，其每一个元素都能够独立地做左值。例如，可以定义一个字符数组存放某班级全体同学某课程的考试成绩等级 '`A`'，'`B`'，'`C`'，'`D`' 或 '`E`'。

字符型数据在内存中按字符的 ASCII 码存放，与整数兼容。即字符数组的各元素能像整数一样地运算。因此，还可以利用字符型数据作为标志变量，一个这样的标志量共可区分 $2^8 = 256$ 种不同的状态（ $0 \sim 255$ 或 $-128 \sim 127$ ）。

可以定义字符数组

```
char c[14]={'I','\\','m','u','a','u','s','t','u','d','e','n','t','.'};
```

上述语句定义了 14 个变量的数组并对全部元素进行了初始化。其中单引号字符使用了其转义形式 '\\' 来表达。要把握该数组，除了数据类型为字符型已经明确外，数组名 c 代表数组的起始地址，即首元素的地址（&c[0]）、元素个数 14 均不可忽视。

上述数组定义还等价于下面的定义语句（请对照附录 A）：

```
char c[]={'I','\\','m','u','a','u','s','t','u','d','e','n','t','.'};  
或  
char c[]={ 73, 39, 109, 32, 97, 32, 115, 116, 117, 100, 101, 110, 116, 46};
```

因为缺省元素个数时，编译系统将根据实际初始化的数据个数确定数组的元素个数。

若要对字符数组进行操作，与一般的数组一样，须逐个元素地进行。如：

(1) 字符变换。c[6]+='A'-'a'；则将 c[6] 由 's' 转换成 'S'。c[13]='!'；则将 c[13] 原来的圆点句号改成感叹号。

(2) 字符 I/O。例如：

```
for(int i=0; i<14; i++)  
    cin >> c[i]; // 顺序接收从键盘输入的字符，并存放到数组各元素中  
for(int i=0; i<14; i++)  
    cout << c[i]; // 顺序输出数组各元素
```

这种按元素逐个字符地处理是最根本的。下面介绍的 C-字符串将一些操作进行了包装，使用起来简捷方便。但其本质仍是按元素逐个处理。

5.7.2 C-字符串

C++ 中有两种字符串，一种是从 C 语言沿袭下来的，被称为 **C-字符串** (**C-string**)；另一种是字符串 (**string** 类) 类型。**string** 并不是 C++ 的保留字，这种类型属于自定义的数据类型，C++ 编译器都支持它。这种类型将在面向对象程序设计部分予以介绍。下面，主要讨论 C-字符串。

为了使字符串表达方便，C/C++ 将存放字符串的数组的三要素“减少”为一要素（其他要素隐藏于其中）。

规定 5.1 (C-字符串) C-字符串（简称字符串）是存放在字符数组中的一系列有序字符，并以特殊的控制字符 ('\0'，其 ASCII 码为 0) 作为字符串的结束标志。

字符 '\0' 也被称为 C-字符串的串结束标志。该字符兼容整数 0 和逻辑假 **false**。

C-字符串以字符数组作为其存放内容的容器。既然是数组就有数组三要素：

① 数据类型是显然的。

② 容器的容量（数组的元素个数）需要足够大即可（实际字符串不一定存满。由于约定了串结束标志，常常只需要处理到串结束标志之前的数组元素为止）。

③ 唯有字符串首地址是必须给出的。因此，C-字符串最重要的是其首地址，存放字符串的数组应该事先定义足够大的空间。在确认了有足够的存放空间后，便将字符型的地址（即字符型的指针）与 C-字符串等同看待。

特别提醒：存放 C-字符串的数组元素个数至少应该比字符串的长度多 1，以便存放串结束标志字符。

规定 5.2（字符串常量） 用双引号包围的一系列字符被称为字符串常量。字符串常量被存放在内存的全局数据区中常量池里，并追加串结束标志。实为字符型常量数组，其首地址值就用字符串字面常量本身表示。

例如，若有语句

```
char *pStr = "I\m\ua\ustudent."; // 注[1]
```

则定义了一个指针变量（仅占内存 `sizeof(void*)` 字节）。指针变量 `pStr` 的值为全局数据区常量池中的某地址值。从该地址处起，共占用 15 个字节（字符串的长度为 14）。这 15 个字节的内容均为常量，不能被修改。故企图进行 `cin>>pstr;` 或 `pStr[6]='S';` 操作都是不可行的。这两个语句虽然能够通过编译，但会出现运行时错误，因为该内存处是不允许写的。

前面提到的字符数组 `char c[14];` 要成为 C-字符串，需要定义成

```
char c[15]={'I','\','m','\','a','\','s','\','t','\','u','\','d','\','e','\','n','\','t','\','.','\0'};
```

或者

```
char c[15]={73, 39, 109, 32, 97, 32, 115, 116, 117, 100, 101, 110, 116, 46, 0};
```

或者更为简捷地

```
char c[15] = "I\m\ua\ustudent.;"
```

此时不应该误解为用一个字符型地址值去初始化数组元素。应该理解为用字符串的内容（包括串结束标志）初始化数组各元素。这是编译系统内部悄悄进行的适应性调整。

5.7.3 字符串 I/O 操作

C/C++ 编译系统对字符型指针所做的适应性调整不仅体现在初始化字符型数组元素上，它对字符型指针（即字符型地址）的操作给予了特别的关照。

例如：

```
1   char str[100];           // 准备足够大的容器存放字符串
2   cin >> str;             // 接收键盘输入一个字符串
3   cout << str;            // 输出字符串的内容
4   cout << str + 4;         // 跳过前 4 个字符，输出其余字符
5   cin.getline(str, 100, '\n');
```

细心的读者可能对第 2~4 行语句产生疑惑。`str` 为数组名，是一个地址值（且为地址常量），难道要从键盘输入一个地址值？第 3 行语句将会输出一个地址值？原来，C/C++ 语言在处理字符型指针（或称为字符型地址）时，有别于其他数据类型的指针（如 `int*`, `double*`），也有别于字符型（`char`, `unsigned char`）数据。

^[1]之所以允许用字符串字面常量初始化一个非常量指针，究其原因是由于 C 语言中无 `const` 保留字，而 C++ 为了完全兼容 C 语言而故意“视而不见”并“网开一面”。

由于字符串由其首地址决定，故 C++ 中的插入运算符、抽取运算符，以及一些字符串处理函数将 C-字符串等同于字符型地址。它们并非对地址值进行操作，而是从该地址处起对内存中的字符内容逐个进行操作，直到遇到串结束标志 '\0' 而不问下标是否越界、该空间是否可读写。保证有足够的空间并保证该空间是可读或可写的是程序员的责任。

在 C++ 中用抽取运算符进行输入操作时，以一个或多个空格字符 (' ') 或制表字符 ('\t') 或换行字符 ('\n') 分隔各个数据项。在执行上述第 2 行时，若输入的字符串中包含上述分隔字符，则数组 str 存放的字符串内容只到输入的第一个分隔字符前为止，系统自动追加串结束标志。余下的数据存放在输入缓冲区，留给后面的输入语句（可能导致后面的语句接收到错误的数据，参见第 2.1.3 小节的注）。

第 5 行调用函数接收键盘输入时，可接收包含空格等字符的字符串。此处的第 2 个参数 100 表示接收字符串的最大长度。第 3 个参数 '\n' 表示遇到换行为止，并自动将该字符转换成串结束标志。其中的圆点 “.” 运算符的用法将在面向对象程序设计部分介绍，此处只需要读者模仿即可。

若输入的字符个数超过存放该字符串的存储空间容量时，超出的部分字符及串结束标志将破坏内存中的其他数据。这种情形是程序员应该警觉的。

第 3、4 行均为输出字符串的内容，而不是输出地址值。这两个字符串有共同的结束标志，但它们的起始地址不同（地址与整数相加减的运算结果为一个新地址值）。

例 5.9 一个有趣的英文单词。

源代码 5.6 一个有趣的英文单词

```

1 // smart.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     char *p, str[100] = "smart";
8     for(p=str; *p; p++)
9         cout << p << endl;
10    cout << p - str << endl;
11    return 0;
12 }
```

程序的运行结果：

```

smart
mart
art
rt
t
5
```

还可以将字符串存放在堆空间中，即利用堆数组作为字符串的容器。如：

```

char *pStr;           // 定义一个指针变量，仅占 sizeof(void*) 字节
pStr = new char [100]; // 申请堆数组，并将首地址值存放到指针 pStr 中
```

```
cin.getline(pStr, 100, '\n');
cout << pStr;
cin >> pStr;
// .....
delete [] pStr;           // 用完后释放
```

须注意的是，对一般非 `unsigned char*`, `char*` 类型的其他指针变量的输出操作确实为对一地址值的操作，其中有些输入操作是不支持的。如：

```
char *pStr=new char[100];
char str[100]={"abcd"};
int a[100];
double *p = new double[100];
cin >> pStr;      // 接收从键盘输入一个字符串, OK
cout << pStr;    // 输出字符串的内容, OK
//cin >> a;       错误。a 为地址常量
//cin >> p;        错误。p 虽为变量, 但 C++ 不支持对 (double*) 型的抽取操作
cout << a;         // 可行。输出一个地址值! 一般而言, 无意义
cout << p;         // 可行。输出一个地址值! 一般而言, 无意义
delete [] p;       // 用完后释放
delete [] pStr;   // 用完后释放
```

5.7.4 C-字符串处理函数

字符串处理的本质是数组元素处理。C++ 提供了一些操作 C-字符串的标准函数（参见附录 B），函数原型在头文件 `cstring` 中。自定义版的 C-字符串处理函数是读者练习指针操作的极佳素材。通过这种练习能够加深对指针的理解，加强对指针的掌控能力。建议读者独立完成附录 B.1 所列的 C-字符串处理函数的自定义版本。本节介绍部分自定义版字符串处理函数的设计原理。为了区别于标准函数，自定义版函数名按“驼峰”（英文字母大小写混合）方式命名。

1. 字符串的长度

从给定的起始地址开始，顺序统计不等于串结束标志字符的个数，直到遇到串结束标志为止。字符串的长度不包括串结束标志字符。例如：

空串。长度为 0，在内存中占用 1 个字节，存放的内容为串结束标志 '\0'（整数 0）
空格串。长度为 1，在内存中占用 2 个字节：' '（整数 32）和串结束标志 '\0'（整数 0）
长度为 4，在内存中占用 5 个字节，存放字符 '3', '2', '1', '0'（整数 51, 50, 49, 48）
和串结束标志 '\0'（整数 0）

例 5.10 计算字符串长度的自定义版本函数。

源代码 5.7 字符串长度自定义函数（数组版）

该函数还可以用如下不同的方法实现。

源代码 5.8 字符串长度自定义函数（指针版）

```

1 int StrLen(const char *str)
2 {
3     int len;
4     for(len=0; *str++; len++)           // 形式参数指针指向逐步移动
5     ;
6     return len;
7 }
```

2. 字符串复制

将给定字符串的内容复制到另一个起始地址处，即字符串的内容复制。这种复制实际上是数组元素间的赋值（复制）。需要注意的是字符串结束标志必须复制到目标数组元素中。

例 5.11 字符串复制的自定义版本函数。

源代码 5.9 字符串复制自定义函数（数组版）

```

1 char *StrCpy(char *dest, const char *source)
2 {
3     int i;
4     for(i=0; source[i]!='\0'; i++)
5         dest[i] = source[i];      // 复制串结束标志前的字符
6     dest[i] = '\0';            // 追加串结束标志字符
7     return dest;
8 }
```

该函数还可以写成如下形式。

源代码 5.10 字符串复制自定义函数（指针版）

```

1 char *StrCpy(char *dest, const char *source)
2 {
3     char *temp = dest;        // 保存目标字符串的首地址
4     while(*source)           // 循环条件：尚未遇到源串的串结束标志
5     {
6         *dest = *source;    // 字符串内容，数组元素间的赋值
7         dest++;             // 指向目标字符串的形式参数指针指向逐步移动
8         source++;            // 指向源字符串的形式参数指针指向逐步移动
9     }
10    *dest = '\0';
11    return temp;            // 返回目标字符串的首地址
12 }
```

上述函数还可以进一步简写成如下形式。

源代码 5.11 字符串复制自定义函数（指针版）

```

1 char *StrCpy(char *dest, const char *source)
2 {
3     char *temp = dest;        // 保存目标字符串的首地址
4     while(*dest++ = *source++) // 此处为赋值运算而不是关系运算
5     ;
```

```

6     return temp;           // 返回目标字符串的首地址
7 }

```

【测试程序】 编写一个程序测试上述两个自定义版的字符串处理函数。要求从键盘输入一个字符串，计算其长度，并将其复制到另一字符串。

【分析】 字符串以字符数组为容器。程序中需要定义元素个数足够大的数组用于存放字符串的内容。本例中，分别采用数组和堆数组存放字符串。其中数组的元素个数取一个“足够大的整数常量” 1000，而堆数组则按需申请。源程序如下：

源代码 5.12 自定义版字符串函数测试程序

```

1 #include <iostream>
2 using namespace std;
3 // 请插入 StrLen, StrCpy 函数定义
4 int main()
5 {
6     char str[1000];      // 定义元素足够多的字符数组做字符串的容器
7     char *p;              // 定义字符指针存放字符串的首地址
8     int len;
9     cout << "请输入一个字符串: ";
10    cin.getline(str, 1000, '\n');        // 键盘输入字符串, 可含空格
11    len = StrLen(str);                // 计算字符串长度
12    cout << "字符串长度: " << len << "字节" << endl;
13    p = new char [len+1];            // 申请适量的堆内存空间
14    StrCpy(p, str);                // 复制字符串
15    cout << "\"" << p << "\"" << endl; // \" 为转义字符
16    delete [] p;
17    return 0;
18 }

```

程序执行时输入“C++ Program.”，则输出结果为

```

请输入一个字符串: C++ Program.
字符串长度: 12 字节
"C++ Program."

```

程序中在输出字符串时，特意输出两个双引号字符 '\"'（使用转义字符）将字符串的内容包围在其中。

5.8 指针数组与数组指针

其实指针数组、数组指针等概念也不需要规定。它们都能够从最基本的概念“推导”出来。

5.8.1 指针数组

规定 5.3（指针数组） 目标数据类型相同的一系列指针的集合为指针数组。其元素可以指向不同的地址。指针数组定义的格式为

```
目标数据类型 * 指针数组名[指针个数] = {地址值列表};
```

例如，指向多个字符串的指针数组。在某函数中定义

```
char str1[80] = "Hello,";
static char str2[80] = "World.";
char *pStr1 = "C++";
char *pStr2 = new char[100];
strcpy(pStr2, "Program.");
char *strings[5]={str1, str2, pStr1, pStr2, "ok."}; // 指针数组
```

其中，字符串 `str1` 在栈空间中，字符串 `str2` 在全局数据区中，字符串 `pStr1` 在全局数据区的常量池中（指针本身在栈空间中），字符串 `pStr2` 在堆空间中（指针本身在栈空间中），字符串 `"ok."` 在全局数据区的常量池中。指针数组 `strings` 在栈空间中，其 5 个元素存放了上述 5 个字符型的地址值^[1]。

命令行参数

操作系统在启动程序执行时，可将一些量作为实际参数传递给程序的主函数。主函数通过形式参数接收。主函数的首部可以是如下形式：

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

其中：

- ① `argc` 为命令行参数的个数（含命令名本身）。
- ② `argv` 是字符串数组（指针数组），`argv[0]~argv[argc-1]` 为命令行各个参数字符串的首地址，`argv[0]` 为命令名本身。
- ③ `env` 为操作系统环境变量及其值的字符串数组（指针数组）。

第二种格式是常见的，其典型的用法如下。

源代码 5.13 命令行参数示例

```
1 // comm_line.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main(int argc, char *argv[])
6 {
7     int width, height, area;
8
9     if(argc<3)
10    {
11         cout << "参数太少" << endl;
12         cout << "命令格式：" << argv[0] << "宽高" << endl;
13         return -1;
14    }
15
16    for(int i=0; i<argc; i++)
17        cout << argv[i] << endl; // 输出命令行的所有参数
18
19    width = atoi(argv[1]); // 将数码字符串转换成对应的整数
```

[1]指针数组像“刺猬”：是指针的集合，各元素（指针）可以指向不同的地址。

```

20     height = atoi(argv[2]); // 将数码字符串转换成对应的整数
21
22     area = width * height;
23     cout << "面积: " << area << endl;
24
25 }
```

编译连接生成可执行文件（如 `comm_line.exe`），直接在操作系统的提示符下输入如下命令。

```

D:\CPP\comm_line\Debug>comm_line 123 456
comm_line
123
456
面积: 56088

D:\CPP\comm_line\Debug>comm_line
参数太少
命令格式: comm_line 宽 高
```

若输入可执行文件名后还带有 "123456"，则程序中的 `argc` 为 3 表示共有三个参数：

- ① `argv[0]` 为命令名称（可执行文件名的主名）"`comm_line`"。
- ② `argv[1]` 为字符串 "`123`"。
- ③ `argv[2]` 为字符串 "`456`"。

程序中用函数 `int atoi(char *)`；将这两个数码字符串转换成整型数据，便可以进行数值计算（参见附录 B.2）。若执行程序时不带命令行参数，或者命令行参数太少，则执行第 11~13 行给出反馈信息后结束程序。

5.8.2 数组指针

规定 5.4（数组指针）存放数组地址的变量称为数组指针^[1]。定义数组指针的格式为

```
目标数据类型 (*数组指针名)[目标数组的元素个数] = 目标数组地址值;
```

数组指针可指向同种类的一系列数组（包括数据类型、元素个数均相同）。数据类型、数组名、元素个数是数组的三要素。指向数组的指针移动时，将根据这三要素改变指向。例如：

有二维数组 `int a[3][4]`；将其看成三个一维数组的集合，这三个一维数组名分别是 `a[0]`，`a[1]`，`a[2]`，它们皆是元素个数为 4 的整型数组。定义一个数组指针，专门指向具有 4 个元素的一维整型数组。

```

const int N=4;
int a[3][4]={ {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} };
int (*p)[N] = a; // 定义了一个指针变量，N 必须为常量
```

^[1] 数组指针像“独角兽”：只有一个指针，其步幅很大，一步跨过一个数组，因而必须确定目标数组的数据类型及元素个数。

则这个指针 p 与二维数组名 a 等效^[1]。p[i][j] 等于 a[i][j]。指针 p 的下标取 1（或者 p++），意味着指针指向下一个具有 4 个元素的一维数组，即等效于一维数组 a[1]。数组指针的主要作用是：将二维数组作为实际参数传递给某函数，函数相应的形式参数应该为数组指针（参见第 6.9.4 小节）。

例 5.12 创建二维堆数组。

源代码 5.14 二维堆数组

```

1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     const int N = 4;
6     int m = 5, i, j;
7     int (*p)[N];           // 定义一个变量（即数组指针）
8     p = new int [m][N];    // 二维堆数组。m 可为变量，N 必须为常量
9     for(i=0; i<m; i++)
10    {
11        for(j=0; j<N; j++) // 用双方括号访问二维堆数组元素
12            cout << (p[i][j] = 10*(i+1) + j+1) << ' ';
13        cout << endl;
14    }
15    delete [] p;
16    return 0;
17 }
```

这样创建二维堆数组仍然不够灵活，因为除最左端的一个元素个数可以是变量外，其余低维的元素个数都必须是常量。更加灵活方便的二维堆数组参见第 6.9.5 小节。

5.9 趣味程序

5.9.1 生日的概率问题

例 5.13 某班级有 n 名同学 ($n \leq 365$)，问至少有两名同学的生日在同一天的概率为多大？该问题的理论结果为

$$p(n) = 1 - \frac{N!}{N^n(N-n)!}$$

若一年按 $N = 365$ 天计算，则有

人数 n	10	20	23	30	40	50
概率 $p(n)$	0.12	0.41	0.51	0.71	0.89	0.97

这是概率论中一个有趣的问题，当人数仅为 23 时其概率就超过 50%；当人数为 50 时其概率竟高达 97%。下面，利用随机数发生器模拟 n 个人的生日。通过大量（如 $K = 1000$ ）次模拟，统计出现生日相同事件的频数。请读者在程序的语句后添加适当的注释，并分析程序中是如何利用标志变量的。

^[1]之所以称等效，而不是等价，是因为 a 是地址常量而 p 是指针变量。

源代码 5.15 生日的概率问题

```

1 // Birthday.cpp
2 #include <iostream>
3 #include <ctime>
4 using namespace std;
5
6 int isSameBirthday(int n)
7 {
8     const int N = 365;
9     int i, day[N], r;
10
11    for(i=0; i<N; i++)
12        day[i] = 0;
13
14    for(i=0; i<n; i++)
15    {
16        r = rand() % N;           // 随机产生第 i 个人的生日
17        if(day[r]==0)
18            day[r] = 1;
19        else
20            return 1;
21    }
22    return 0;
23 }
24
25 int main()
26 {
27     const int K = 1000;
28     int n, m, k;
29     time_t t;
30     srand(time(&t));
31
32     cout << "请输入你所在班级的人数: ";
33     cin >> n;
34
35     for(k=m=0; k<K; k++)
36         m += isSameBirthday(n);
37     cout << "人数: " << n
38             << ", 你班出现有生日相同现象的可能性为 "
39             << m << "/" << K << " = "
40             << double(m)/K << endl;
41     return 0;
42 }
```

5.9.2 匹配的概率问题

例 5.14 某人一次写了 n 封内容互不相同的信，又写了 n 个收件人互不相同的信封，如果他任意地将 n 封信装入 n 个信封中。问至少有一封信装正确的概率是多少？该问题的理论结果为

$$p(n) = 1 - \frac{1}{2!} + \frac{1}{3!} - \cdots + (-1)^{n-1} \frac{1}{n!}$$

当 n 充分大时， $p(n) \approx 1 - e^{-1}$ 。

下面的程序模拟了该问题，计算了其频数和频率。最后还计算了 $1/(1 - p(n))$ （即 e 的近似值）。请读者在程序的语句后添加适当的注释。分析程序中为什么需要使用动态数组，程序中是如何利用标志变量的。

源代码 5.16 匹配的概率问题

```
1 // matching.cpp
2 #include <iostream>
3 #include <ctime>
4 using namespace std;
5
6 int matching(int n)
7 {
8     int *x, *y, i, r, flag = 0;
9     x = new int [n];
10    y = new int [n];
11
12    for(i=0; i<n; i++)
13        x[i] = 0;
14    for(i=0; i<n; i++)
15    {
16        while(true)
17        {
18            r = rand() % n;
19            if(x[r]==0)
20            {
21                y[i] = r;
22                x[r] = 1;
23                break;
24            }
25        }
26    }
27    for(i=0; i<n; i++)
28        if(y[i]==i)
29        {
30            flag = 1; break;
31        }
32    delete [] x;
33    delete [] y;
34    return flag;
35 }
36
37 int main()
38 {
39     const int K = 10000;
40     int n = 100, m = 0, k;
41     time_t t;
42     srand(time(&t));
43     for(k=0; k<K; k++)
44         m += matching(n);
45
46     cout << m << "/" << K << "≈"
47         << double(m)/K << endl;
48     cout << double(K)/(K-m) << endl;
49
50 }
```

5.9.3 模仿密码输入

例 5.15 模仿密码输入。许多应用系统中，在进入某些有一定密级的功能模块前需要用户通过密码验证。通常不回显用户所输入的字符，代之以星号“*”输出，以回车结束。并限时或不限时地给用户三次机会输入密码。

下面的程序模仿密码输入过程。通过调用 `getch` 函数接收用户的键盘输入，并将输入的字符依次存入一数组中。允许用户使用退格键修改输入的字符；自动将用户输入的回车字符替换成串结束标志并结束输入。程序中将判断是否有键盘输入与计时器相结合实现限时和不限时功能。最后将输入的结果与给定的用 C-字符串存放的密码（可包含空格，'\t' 等字符）进行内容比较。

源代码 5.17 模仿密码输入

```

1 // Password.cpp      模仿密码输入示例
2 #include <iostream>
3 #include <cstring>    // 为了使用 strcmp 函数比较两个 C-字符串的内容
4 #include <conio.h>    // 为了使用 getch 函数, kbhit 函数
5 #include <ctime>      // 为了使用 clock 函数
6 using namespace std;
7
8 int InputPasswd(const char *passwd, double timelimit)
9 {           // 在限定的时间内输完正确的密码则返回1, 否则返回0
10    char str[80], c=0;
11    int i=0;
12    double t = clock() + timelimit*1000;    // t 为终止时间
13
14    do
15    {
16        if(kbhit())
17        {
18            c = getch();          // 接收键盘输入一个字符, 不回显
19            switch(c)
20            {
21                case '\b':           // 按退格键
22                    if(i>0)
23                    {
24                        cout << c << ' ' << c << flush;
25                        // 覆盖一个星号, 光标退一格
26                        i--;             // 计数器(下标)减1
27                    }                   break;
28                case '\r':           // 按回车键
29                    str[i] = '\0';       // 换成串结束标志
30                    cout << endl;     break;
31                default:             // 按其他键
32                    str[i++] = c;    // 将字符存入数组元素, 下标增1
33                    cout << "*" << flush; // 输出一个星号
34            }
35        }
36    }while( i<sizeof(str) && c!='\r'
37          && (clock()<=t || timelimit<=0) );
38
39    str[sizeof(str)-1] = '\0';           // 保证有串结束标志
40    return strcmp(passwd, str)==0;        // 字符串比较
41 }
42

```

```
43 int main()
44 {
45     char passwd[80] = "How\uare\uyou!";
46     int n;
47
48     for(n=0; n<3; n++)
49     {
50         cout << "\n\u请输入密码 (限时 10 秒) : ";
51         if(InputPasswd(passwd, 10)==1)
52             break;
53     }
54     if(n<3)
55         cout << "\n\u密码正确。" << endl;
56     else
57         cout << "\n\u密码错误" << endl;
58
59     for(n=0; n<3; n++)
60     {
61         cout << "\n\u请输入密码 (不限时) : ";
62         if(InputPasswd(passwd, 0)==1)
63             break;
64     }
65     if(n<3)
66         cout << "\n\u密码正确。" << endl;
67     else
68         cout << "\n\u密码错误" << endl;
69     return 0;
70 }
```

5.10 小 结

数组是一系列变量的集合，用于存放大量数据的容器。数组各元素相同的数据类型，连续的存储单元安置使大量数据集中统一管理（如定义和销毁）及访问（如读和写）成为可能。数据类型、首地址和元素个数是数组三要素。访问数组元素、数组在函数之间传递，本质上是通过指针操作实现的。将指针与数组联系起来思考和应用是十分有利的。

指针既让人喜欢又让人头痛。学习 C 语言而不会操作指针者，等于没有学习 C 语言。C++语言引入了一些好的机制（例如引用、类的封装、流），利用这些机制，可以将指针操作隐藏在幕后，使程序员少接触指针。如同木偶戏表演中，木偶们漂亮的动作造型背后，隐藏着其最基本的操作。学习高级语言程序设计的人，至少计算机专业的学生需要到后台看个究竟，揭密其内幕。

自定义版 C-字符串处理函数的实现是读者练习指针操作的极佳素材。通过这项练习，读者将能深刻体会指针与数组与函数的关系；深刻理解 C/C++对字符指针（目标类型为字符型的地址值）所做的适应性调整的细节和本质。

本章出现的符号稍多，且同一个符号（如 []，*，&）皆有多种含义。它们有时是作标志用的记号，有时是运算符。读者要善于从程序的上下文根据语法规规定正确地辨析，在程序中正确地使用它们。

练习 5

1. 指出下列各程序中的错误

(1)

```

1 #include <iostream>
2 using namespace std;
3
4 double average(double *x)
5 {
6     int n = sizeof(x) / sizeof(double);           // 此处 n 将为 0
7     double s = 0;
8     for (int i = 0; i < n; i++)
9         s += x[i];
10    return s / n;
11 }
12
13 int main()
14 {
15     double a[10];
16     int i, n = sizeof(a) / sizeof(double);      // 此处 n 为 10, 正确
17     for (i = 0; i < n; i++)
18         a[i] = rand() / (double)RAND_MAX;
19     cout << "average: " << average(a) << endl;
20     return 0;
21 }
```

(2)

```

1 #include <iostream>
2 using namespace std;
3
4 double average(double *x, int n)
5 {
6     double s = 0;
7     for (int i = 0; i < n; i++)
8         s += x[i];
9     return s / n;
10 }
11
12 int main()
13 {
14     double a[10];
15     int i;
16     for (i = 0; i < 10; i++)
17         a[i] = rand() / (double)RAND_MAX;
18     cout << "average: "
19     << average(a[10], sizeof(a) / sizeof(*a)) << endl;
20     return 0;
21 }
```

(3)

```

1 #include <iostream>
2 using namespace std;
3
4 void swap_r(int &a, int &b)
5 {
```

```

6     int temp;
7     temp = a; a = b; b = temp;
8 }
9
10 void swap_r(int *a, int *b)
11 {
12     int temp;
13     temp = *a; *a = *b; *b = temp;
14 }
15
16 int main()
17 {
18     int a=3, a=5;
19     cout << "a=" << a << ", b=" << b << endl;
20     swap_r(&a, &b);
21     cout << "a=" << a << ", b=" << b << endl;
22     swap_p(*a, *b);
23     cout << "a=" << a << ", b=" << b << endl;
24     return 0;
25 }
```

(4)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     char *p;
7     cout << "请输入一个字符串: ";
8     cin >> p;
9     cout << p << endl;
10    return 0;
11 }
```

(5)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     char str[100];
7     str[0] = '0'; str[1] = 'k'; str[2] = '!';
8     cout << str << endl;
9     return 0;
10 }
```

(6)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     double x=1, *px=&x;
7     px++;
8     cout << *px << endl;
```

```

9     return 0;
10 }
```

(7)

```

1 #include <iostream>
2 #include <cstring>
3 using namespace std;
4
5 int main()
6 {
7     char *p = "Hello,world.";
8     char str[7] = "Hello.";
9     strcpy(p, str);
10    cout << p << endl;
11    return 0;
12 }
```

(8) 希望实现字符串复制、字符串拼接。

```

1 char *StrCpy(char *dest, const char *source)
2 {
3     while(*dest++ = *source++)
4         ;
5     return dest;
6 }
7
8 char *StrCat(char *dest, const char *source)
9 {
10     char *temp=dest;
11     while(*dest++)
12         ;
13     while(*dest++ = *source++)
14         ;
15     return temp;
16 }
```

(9)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     char *p = new char[100];
7     cout << "请输入一个字符串: ";
8     cin.getline(p, 100, '\n');
9     cout << p << endl;
10    return 0;
11 }
```

(10)

```

1 #include <iostream>
2 using namespace std;
3
4 void GetMemory(double **x, int n)
```

```

5  {
6      x = new double [n];
7  }
8
9 void FreeMemory(double *x)
10 {
11     delete [] x;
12     x = NULL;
13 }
14
15 int main()
16 {
17     double *p;
18     GetMemory(p, 100);
19     for(i=0; i<100; i++)
20         cout << (p[i]=rand()/(double)RAND_MAX) << endl;
21     FreeMemory(p);
22     return 0;
23 }

```

2. 基本题

(1) 阅读下述程序，指出其中各函数的功能。

```

1 #include <iostream>
2 using namespace std;
3
4 int isLeapyear(int year)
5 {
6     return year%4==0 && year%100 || year%400==0;
7 }
8
9 void days_month(int year)
10 {
11     int days[12]={31,28,31,30,31,30,31,31,30,31,30,31};
12     days[1] += isLeapyear(year);
13     for(int i=0; i<12; i++)
14         cout << i+1 << "月" << days[i] << "天" << endl;
15 }
16
17 int main()
18 {
19     days_month(2008);
20     return 0;
21 }

```

(2) 编写一个原型为 `int week(int year, int month, int day);` 的函数，其功能是计算并返回 `year` 年 `month` 月 `day` 日的星期（0、1、……、6 依次表示星期日、星期一、……、星期六）。【提示】从“1 年 1 月 1 日”起到“`year` 年 1 月 1 日（含）”所经历的“总天数”为（该算法不适用于 1582 年以前，但今后的千年均正确，这是由于历法的原因） $n=365*(year-1)+(year-1)/4-(year-1)/100+(year-1)/400+1$ ；则 `year` 年 1 月 1 日为星期 $w=n\%7$ 。

(3) 编写一个程序根据输入的年份 (`year`) 输出该年的日历。

(4) 编写一个输出杨辉三角的函数 void Yanghui(int n); 如语句 Yanghui(6); 的执行结果应该输出 (每个数据占 4 字符宽)。

```

1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10  10  5   1
1   6   15  20  15  6   1

```

(5) 指出函数 int pos(char, const char*); 的功能及程序的运行结果。

```

1 #include <iostream>
2 using namespace std;
3
4 int pos(char c, const char *str)
5 {
6     for(int i=0; *str; i++)
7         if(*str++==c)
8             return i;
9     return -1;
10 }
11
12 int main()
13 {
14     cout << pos('a', "abcd") << "\n"
15     << pos('c', "abcd") << "\n"
16     << pos('e', "abcd") << "\n"
17     << pos('\0', "abcd") << "\n"
18     << pos(98, "abcd") << endl;
19     return 0;
20 }

```

(6) 阅读程序, 指出程序的运行结果。

```

1 #include <iostream>
2 #include <cstring>
3 using namespace std;
4
5 int f(const char *str)
6 {
7     int len = strlen(str);
8     const char *p=str+len-1;
9     while(p>=str)
10        cout << p-- << endl;
11     return len;
12 }
13
14 int main()
15 {
16     cout << f("SMART") << endl;
17     return 0;
18 }

```

(7) 在一个程序的主函数中，定义了两个字符型指针变量，它们分别指向两个字符串的首地址。通过调用函数实现两个字符型指针交换指向。请完成函数 `exchange` 的设计和定义。注意：不要交换字符串的内容（下面的程序中无法交换字符串的内容，因为它们都是字符串常量）。

```

1 #include <iostream>
2 using namespace std;
3
4 // 定义 exchange 函数
5
6 int main()
7 {
8     char *p1 = "I'm a programmer.";
9     char *p2 = "Hello, World.";
10    cout << p1 << '\n' << p2 << endl;
11    exchange(p1, p2);
12    cout << p1 << '\n' << p2 << endl;
13    return 0;
14 }
```

(8) 在一个程序的主函数中，定义了两个元素个数相等的字符型数组，它们分别存放两个字符串。通过调用函数实现两个字符串内容的交换。请完成函数 `StrSwap` 的设计与定义。注意：不要交换字符型地址值（下面的程序中无法交换字符串的首地址，因为它们都是数组名，为地址常量）。

```

1 #include <iostream>
2 #include <cstring>
3 using namespace std;
4
5 // 定义 StrSwap 函数
6
7 int main()
8 {
9     char str1[100] = "I'm a programmer.";
10    char str2[100] = "Hello, World.";
11    cout << str1 << '\n' << str2 << endl;
12    StrSwap(str1, str2);
13    cout << str1 << '\n' << str2 << endl;
14    return 0;
15 }
```

(9) 编程输入一个字符串，将其倒置，然后输出倒置的结果。

(10) 设计用户自定义版的 C-字符串处理函数，实现与相应的标准函数相同的功能（参见附录 B.1）。为了与标准函数相区别，函数命名采用“驼峰”表示法。请定义如下函数：

```

int StrLen(const char *str); // 计算字符串的长度
char *StrCpy(char *dest, const char *source);
                           // 字符串复制
char *StrN Cpy(char *dest, const char *source, int size);
                           // 复制指定长度字符串
char *StrCat(char *dest, const char *str);
                           // 字符串拼接
```

```

char *StrUpr(char *str);
// 字符串中小写字母转换成对应的大写字母
int StrCmp(const char *str1, const char *str2); //字符串比较
// 当第一个字符串大于、等于、小于第二个字符串时，分别返回 1、0、-1

```

3. 扩展题（程序设计竞赛题型）

(1) 计算若干个整数的和。

问题描述 对于给定的若干个整数，要求计算它们的和。

输入说明 输入数据有多行，每一行有若干个整数，希望计算它们的项数及总和。

输出说明 对于每一行中的数据，要求先输出“Case 序号：”，然后输出数据个数、逗号、空格、它们的和、换行。

输入样例

```

15 3 42 89
51 201 303 9 755 800

```

输出样例

```

Case 1: 4, 149
Case 2: 6, 2119

```

【解答】 本题的难点在于如何读取一行中的数据（数据个数不定）。本程序中使用的串流类将在面向对象程序设计部分介绍（参见第 15.3.2 小节），程序中涉及串流类的操作只需模仿。

源代码 5.18 竞赛题型·读取一整行上的多个数据

```

1 #include <iostream>
2 #include <sstream> // 因需要定义一个串流类对象
3 #include <string> // 因需要使用 C++ 字符串
4 using namespace std;
5
6 int main()
7 {
8     string str; // 定义一个 C++ 字符串变量
9     int x, sum, n, k=0;
10
11    while(getline(cin, str)) // 当作字符串读取一整行到 str 中
12    {
13        istringstream istr(str); // 根据 str 定义一个串流类的对象
14        sum = 0;
15        n = 0;
16        while(istr >> x) // 依次从串流类中读取各个数据
17        {
18            sum += x;
19            n++;
20        }
21        cout << "Case " << ++k << ":" <<
22             << n << ", " << sum << endl;
23    }
24    return 0;
25 }

```

(2) 判断回文串。

问题描述 判断给定的字符串是否为回文。所谓回文是指一个字符串，将其从左到右读和从右到左读都是相同的。

输入说明 输入数据的第一行为一个整数 n，其后共有 n 行待判断的字符串表示有 n 种情况（每行的字符总数不超过 1000 个字符）。

输出说明 对于每一种情况，要求先输出“Case 序号：”，然后输出“YES.”（若对应的字符串为回文）或“NO.”（若对应的字符串不是回文）、换行。

输入样例

```
5
123 456 789 0 987 654 321
abcdefghijklmnopqrstuvwxyz
level
Abba
x
```

输出样例

```
Case 1: YES.
Case 2: NO.
Case 3: YES.
Case 4: NO.
Case 5: YES.
```

【解答】 本题除了回文判断算法外，容易出错处是抽取操作 (`cin>>n;`) 执行之后输入流缓冲区中仍留有换行符（整数 n 之后可能还有的一系列空格字符 ' '、一系列制表字符 '\t' 以及一个换行字符 '\n'）的处理。若不处理，直接使用 `cin.getline(str, N)` 则读取的是上述内容，将其当作第一种需要判断回文的情况将是错误的。

源代码 5.19 竞赛题型·判断回文串

```
1 // Palindrome.cpp
2 #include <iostream>
3 using namespace std;
4
5 int Palindrome(const char *str)
6 {
7     int left=0, right;
8     for(right=0; str[right]; right++)
9     ;
10    right--;           // 找到最右端字符的下标
11    while(left<right && str[left]==str[right])
12    {
13        left++;
14        right--;
15    }
16    return !(left<right);
17 }
18
19 int main()
20 {
21     const int N = 1000;
22     int i, n=0;
```

```

23     char str[N];
24
25     cin >> n;           // 读取一个整数
26     cin.ignore(N, '\n'); // 忽略整数后直到换行符的所有字符
27 //    cin.getline(str, N); // 或者读取后丢弃
28     for(i=1; i<=n; i++) // 处理 n 种情况
29     {
30         cin.getline(str, N); // 读取待处理的字符串
31         cout << "Case " << i << ":" <<
32             << (Palindrome(str) ? "YES." : "NO.")
33             << endl;
34     }
35     return 0;
36 }
```

(3) 字符串变换（字符串加密与解密）

问题描述 对某字符串（原文）进行加密并形成密文，或者将这种密文解密还原成原文。给定一个密码字符串（例如 "3721"）。加密时，循环利用密码字符串的各字符依次与字符串原文字符进行位异或运算得到密文字符串。请根据上述规定编写程序实现字符串的加密与解密。

输入说明 输入数据有多行，每两行为一种情形。每两行中的第一行为密码字符串，其下一行是待加密的字符串（原文）。密码和原文字符串中皆可能出现空格等字符，它们的长度均不超过 100 个字符。

输出说明 对于每一种情形输出三行。要求三行中的第一行输出“Case 序号：”，密码字符串；其下一行输出密文字符串各字符的 ASCII 值，每个数字后输出一个空格（这是考虑到密文中可能含有非可打印字符）；再下一行为解密的结果（应该与原文一致）。

输入样例

```

3721
Hello, world.
VC++ 6.0
这是汉字字符串原文。
密码
This is a string.
```

输出样例

```

Case 1: 3721
123 82 94 93 92 27 18 70 92 69 94 85 29
Hello, world.
Case 2: VC++ 6.0
131 161 225 236 154 140 249 230 129 149 156 208 148 152 250 157 152 135 138 136
这是汉字字符串原文。
Case 3: 密码
151 180 171 152 227 181 177 203 162 252 177 159 177 181 172 140 237
This is a string.
```

第6章 函数

函数是 C++ 语言的重要内容。本书将函数的基础知识分散到了前面的各章节之中。本章介绍函数调用操作的内部原理——栈操作机制。学习本章后，读者应该了解函数调用的栈操作过程，理解传递数值（包括传递地址值）、传递变量（包括传递指针变量）的本质；理解函数数值返回（包括地址值返回）、变量返回（包括指针变量返回）的本质；进一步理解局部自动变量和函数形式参数的生命期。

本章将对函数的形式参数、返回类型的各种情形进行汇总综述。还将介绍函数的其他语法，包括内联函数、递归函数、函数重载和参数带默认值等。本章将继续讨论“评委评分”程序的改编。

6.1 函数概述

设计程序是为了完成一定的计算、实现一定的功能。一个比较大的项目总是可以将其按照功能划分成若干个模块进行相对独立地处理，使程序模块化。一个好的模块划分方案本身就能体现对欲求解问题的理解和把握。例如，在“评委评分”程序中，我们可以将程序划分成配置、输入、计算、输出 和善后 5 个模块。其中计算模块中还可以进一步细分成若干个子模块，如计算最后得分、计算选手名次。

程序模块是功能模块，整个程序是由大小不同、各俱特色的功能模块拼装而成的。就像用积木搭建建筑物模型一样，具有一定功能的各种积木块可以用在任何需要它的地方。一个设计好的模块，可能具有一定的通用性，能够将其应用到其他需要此功能模块的地方，使我们设计的程序模块可以重复使用。

显然，要使程序模块具有较高的灵活性、可重用性，在划分功能模块时，应该尽量使各功能模块具有相对独立、功能单一且模块之间接口清晰等特点。程序的功能模块在 C++ 语言中体现为函数。主函数为主控模块。

常将具有一定功能，仅暴露少量用于信号流入流出接口的装置称为黑箱，在使用黑箱时并不关心其内部结构。从函数外部以使用函数的角度看，一个函数便是一个“黑箱”，给它提供一定的数据信息，经其处理后，可得到一定的结果。

函数原型是对函数内部与外部之间接口界面的一种描述。其作用是声明函数，它告诉编译器如何编译调用该函数的语句。函数原型也是程序员调用函数，进行数据传递并获取返回值的依据。根据函数的功能和函数原型，就能够很好地使用函数。当然，调用函数时应该给函数提供可靠的实际数据，因为在函数内部接收数据时，一般没有义务或者没有能力判断这些数据的来源及其正确性。事实上，函数体内常常无法考证数据的来源，也不便或根本无法判断这些数据的正确性（例如，字符串复制函数 `strcpy` 所接收的第一个参数——目标串首地址，该地址处是否可写，是否有足够大的空间，这些问题对该函数是无法考证和保证的）。函数内部只管“来料加工”“完工交货”。

6.2 函数的调用机制

6.2.1 函数调用的栈操作过程

在程序执行过程中调用函数 A，则函数 A 获得程序控制权。若在函数 A 内又调用了另一个函数 B，则函数 A 将程序的控制权暂时移交给函数 B。此时，函数 A 必须等待函数 B 返回后，重新获得程序的控制权执行函数 A 剩下的语句，执行完毕后才可能返回到调用函数 A 的函数。这是一个“先进后出”的过程。这种先进后出的操作就像将子弹压入弹匣，而发射时则是最后压入的子弹最先发射。在计算机学科中，将这种先进后出的操作称为栈操作；称实现栈操作的容器为栈。

栈是一种数据结构，数据存放至栈中称为将数据压入栈中，被压入的数据将存放在当时的栈顶之上；当前栈顶数据在栈空间中取出称为数据从栈中退出。栈操作过程与函数嵌套调用逐层返回的要求是相吻合的。因此，函数调用适合用栈结构操作来实现。

调用函数时还涉及“保护现场”、程序控制权转移等一系列操作；函数返回时也涉及“恢复现场”、转移程序的控制权等一系列操作。值返回的函数返回值并不存放在栈空间中，在不影响理解的前提下，本章将函数的返回值画在栈中。这些方面的内容超出本课程的范围，故简略地称“保存状态，转移控制”。

下面我们将第 3.2.1 小节例 3.2 改编完整，用程序配合图解的方式介绍函数调用的栈操作机制。详细说明函数调用、参数传递、函数返回等操作在栈空间中的活动情况。

例 6.1 求解一元二次方程。对于一元二次方程 $ax^2 + bx + c = 0$ ，我们知道，它由其三个系数唯一决定。为简单起见，始终认为 $a \neq 0$ 。源程序及程序的运行结果如下。

源代码 6.1 求解一元二次方程

```

1 // solver.cpp 求解一元二次方程
2 #include <iostream>
3 #include <cmath> // 其中含有 sqrt 等标准函数声明
4 using namespace std;
5
6 int Solver(double a, double b, double c, double &x1, double &x2);
7 double Delta(double a, double b, double c); // 函数原型用于声明
8
9 int main()
10 {
11     double a=1, b=2, c=-3, x1, x2;
12     int flag; // 准备存放数据的变量
13
14     flag = Solver(a, b, c, x1, x2); // 计算
15     cout << "方程" << a << "x^2"; // 输出方程
16     if(b>0)
17         cout << "±" << b << "x";
18     else if(b<0)
19         cout << "±" << -b << "x";
20     if(c>0)
21         cout << "±" << c;
22     else if(c<0)
23         cout << "±" << -c;
24     cout << "±0";
25 }
```

```

26     switch(flag)                                // 输出结果
27     {
28         case 0: cout << "无实数根。" << endl;      break;
29         case 1: cout << "有重根: x1=x2=" << x1 << endl; break;
30         case 2: cout << "有两个根: x1=" << x1
31             << ",x2=" << x2 << endl;
32     }
33     return 0;                                    // 返回操作系统
34 }
35
36 double Delta(double a, double b, double c)
37 {
38     return b*b - 4*a*c;
39 }
40
41 int Solver(double a,double b,double c, double &x1,double &x2)
42 {
43     int flag = 0;
44     double d = Delta(a, b, c);                  // 调用函数
45     if(d >= 0)
46     {
47         d = sqrt(d);                           // 调用标准函数
48         x1 = (-b - d)/(2*a);
49         x2 = (-b + d)/(2*a);
50         flag = (d>0) ? 2 : 1;
51     }
52     return flag;
53 }
```

程序的运行结果：

方程 $1x^2 + 2x - 3 = 0$ 有两个根: $x1 = -3, x2 = 1$

执行该程序时栈空间的活动情况如图 6-1 所示。本程序中没有使用全局变量，也没有使用堆变量。因此，图中没有画出全局数据区、堆区、代码区的情况。

图 6-1 (a) 表现了程序执行时栈空间持续压栈的活动过程。

① 程序开始执行时首先将主函数压入栈底；建立命令行参数（本例不处理命令行参数）；定义局部自动变量 $a, b, c, x1, x2, flag$ （其生命期开始）。

② 执行第 14 行：保存主函数状态；将 `Solver` 函数压栈；程序的控制权交给 `Solver` 函数。

③ 执行 `Solver` 函数（第 41 行起），值传递型形式参数 a, b, c 的定义（分配内存空间）及初始化、引用型形参 $x1, x2$ 声明（不另占空间）及初始化^[1]；定义局部自动变量 $flag$ （它们的生命期起始点）。

④ 第 44 行完成变量 d 的定义之前需要调用 `Delta` 函数。保存 `Solver` 函数的状态；将 `Delta` 函数压栈；程序控制权转移给 `Delta` 函数。

⑤ 执行 `Delta` 函数（第 36 行起），定义函数 `Delta` 的值传递型形式参数 a, b, c （分配内存空间）并初始化（其生命期开始）。此时 `Delta` 函数位于栈顶。`Solver` 函数中的局部自动变量 d 仍在待建中。

^[1] 关于函数参数入栈的顺序，图 6-1 表示成从左至右，也有的编译器按从右至左处理。

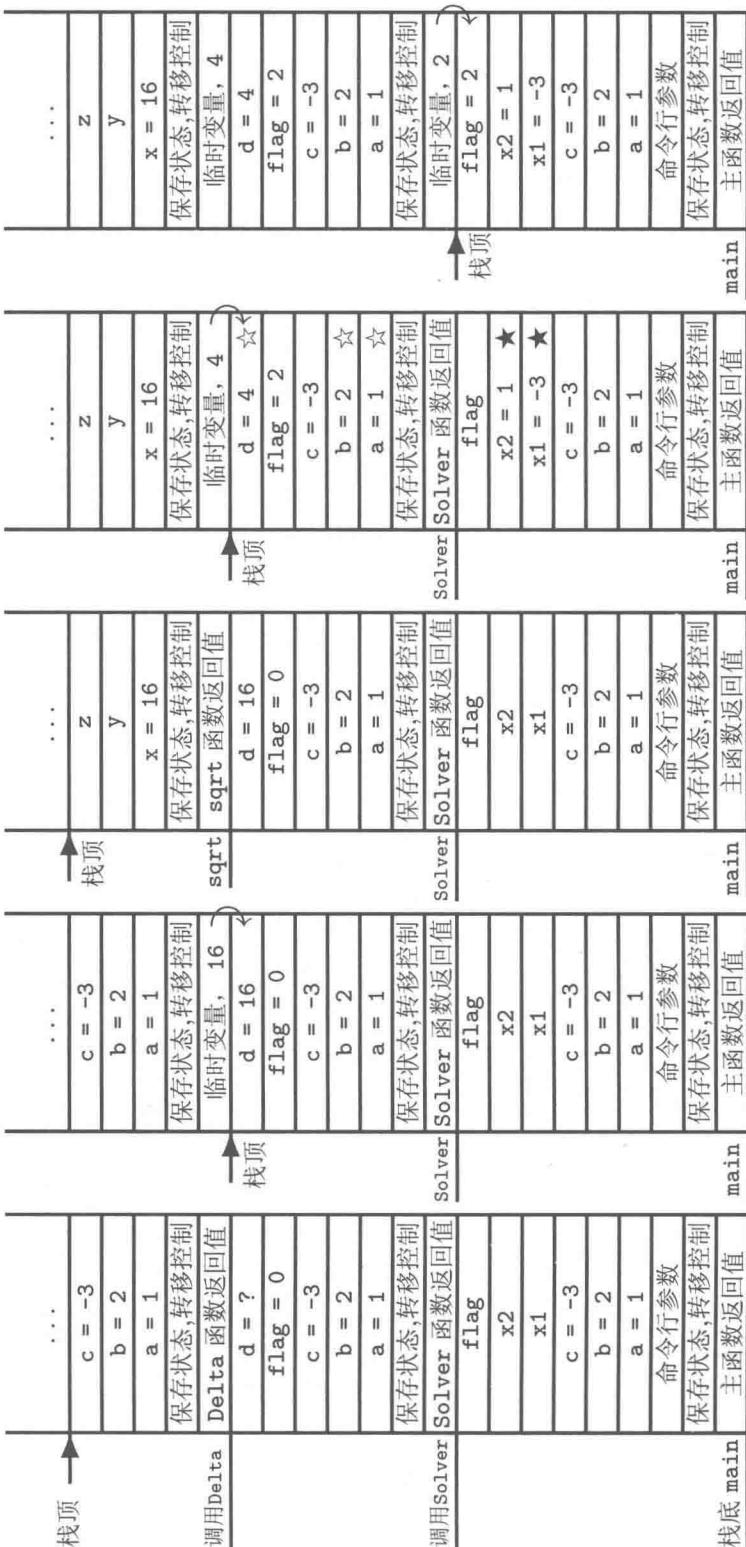


图 6-1 函数调用的栈操作过程

可以看到，栈空间中有主函数的局部变量 `a`, `b`, `c` 及其副本（`Solver` 函数的形参）以及副本的副本（`Delta` 函数的形参）。它们存放于栈空间的不同地址处，彼此间相互独立，彼此间不能相互访问，不会引起冲突。栈空间中主函数的局部变量 `x1`, `x2` 只有一份，`Solver` 函数的形式参数 `x1`, `x2` 分别是其别名，不另占空间。

图 6-1 (b) 展示的是 `Delta` 函数返回时的栈空间状态。此时，标志栈空间当前位置的栈顶指针回落到 `Solver` 函数所占的空间处，表示程序的控制权交还给 `Solver` 函数，也表明 `Delta` 函数的形式参数的生命期终止。值返回的 `Delta` 函数的返回值 16 画在栈顶上方的临时变量中^[1]，该临时变量用表达式 `Delta(a, b, c)` 本身表示。在完成了第 44 行初始化变量 `d` 之后该临时变量随即消失。变量 `d` 创建完成后，程序将在 `Solver` 函数中从第 45 行起继续执行。

图 6-1 (c) 展示的是程序执行到第 47 行时，调用标准函数 `sqrt`，将其压栈的情形。“保存状态，转移控制”均是类似的。作为标准函数 `sqrt` 有一个形式参数（姑且认为形式参数名为 `x`），至于 `sqrt` 函数内部“黑箱”中定义了什么变量，我们不得而知，只好用 `y`, `z` 代替。栈顶指针再次上升。

图 6-1 (d) 表现当 `sqrt` 函数返回时，销毁 `sqrt` 函数执行时创建的形式参数、局部自动变量后，将其返回值 4 存放在一临时变量（该临时变量用表达式 `sqrt(d)` 本身表示）中的情形。返回值 4 通过赋值语句仍然给变量 `d` 赋值覆盖其原值。接下来执行的第 48、49 行对 `Solver` 函数中声明的两个引用赋值，也就是对它们所引用的变量——主函数中定义的两个变量 `x1`, `x2` 分别赋值（参见图中的五角星，空心五角星做右值构成求根公式表达式，实心五角星做左值）。程序中第 50 行为 `Solver` 中定义的局部变量 `flag` 赋值 2。第 52 行的 `return flag;` 语句结束 `Solver` 函数的执行。其中的局部自动变量 `d`, `flag` 被销毁，形式参数 `a`, `b`, `c` 及 `x1`, `x2` 的生命期皆终止，将控制权交还给主函数 `main`。同时，将返回值 2 存放在一临时变量中，以便程序第 14 行给主函数中定义的变量 `flag` 赋值。

图 6-1 (e) 表示 `Solver` 函数返回后，主函数中定义的变量 `flag` 获得 `Solver` 函数的返回值 2 的情形。主函数最后获得程序控制权，从“恢复现场”后的第 15 行继续执行余下的语句^[2]。直到执行主函数中的 `return 0;` 语句（第 33 行）而结束整个程序，返回到操作系统。

【说明】 图 6-1 各子图中栈顶指针之上的存储空间的内容是不可访问的。其中的内容不一定被清除，应该理解为“未被使用”的区域。

上述 `Solver` 函数的第 4, 5 个形参为传递变量（引用型形参），在函数体内直接使用变量的别名访问其绑定的实体变量，在主函数中则直接用主函数中可见的两个变量分别初始化形参。

若将第 4, 5 个形参数改成值传递（传递地址值，即指针型形参。参见下面的 `SolverP` 函数），则函数体内为间接访问指针的目标变量（用间接名访问主函数中不可见的实体变量）。而在主函数中，则需要用取地址运算，将两个实体变量的地址作为实参传递之。

^[1]事实上，数值返回的函数返回值（临时变量）并不存放在栈空间中。示意图 6-1 简单地将其画在栈空间中并影响对此问题的理解。

^[2]C++ 中运算操作（包括输入输出）也是函数调用，同样需要进行压栈、退栈操作，其栈操作原理及过程是类似的，暂不做介绍，故略。

```

1 int SolverP(double a, double b, double c, double *x1, double *x2);
2
3 int main()
4 {
5     double a=1, b=2, c=-3, x1, x2;
6     int flag;                                // 准备存放数据的变量
7     flag = SolverP(a, b, c, &x1, &x2);      // 取地址运算的结果做实参
8     // ...
9     return 0;
10 }
11
12 int SolverP(double a, double b, double c, double *x1, double *x2)
13 {
14     int flag = 0;
15     double d = Delta(a, b, c);
16     if(d >= 0)
17     {
18         d = sqrt(d);
19         *x1 = (-b - d)/(2*a);                // 取指针的目标变量：间接名
20         *x2 = (-b + d)/(2*a);                // 间接访问
21         flag = (d>0) ? 2 : 1;
22     }
23     return flag;
24 }
```

(1) SolverP 函数体内对两个形参指针的目标变量 `*x1`, `*x2` 进行赋值操作（参见程序段第 19~20 行, 即该地址处变量的间接名）。

(2) 主函数中调用函数时（第 7 行）的第 4, 5 个实参应该传递变量的地址值，用于初始化调用函数 SolverP 时创建的两个形式参数指针变量（指针变量本身占用栈空间，分别各占用 `sizeof(void*)` 字节）。

程序的其他部分请读者补充。并请读者仿照前面的方法分析这样修改后的程序在运行时的栈空间活动情况，以掌握传递地址值的操作过程。

C 语言没有引用的概念。当需要令一个函数“返回”多个数值时，常采用这种向函数“暴露变量地址”的方法，将变量的地址值传递给函数的指针型形式参数，让函数通过内存地址间接地访问变量。C++ 是 C 语言的超集，完全包容了 C 语言的一切。上面的传递指针的做法能够在 C++ 编译器中编译通过，但它带有浓厚的 C 语言风格。

请读者分析第 3.2.1 小节例 3.3, 第 5.5.1 小节例 5.2, 以及例 5.3 (反例)、例 5.4 (反例) 和改正反例 5.4 的函数 `swap3` (涉及堆内存空间) 运行时栈空间的活动情况。

上面的例子只讨论了函数值返回时的情形（创建临时变量存放返回值）。函数引用返回时不创建临时变量，直接返回变量本身，并用函数调用表达式表示所返回的变量。请读者思考这种函数调用返回时的栈操作情况。

全局变量、静态全局变量、静态局部变量和堆变量均不存放在栈空间中，它们的生命周期不因为函数的调用而存在（静态局部变量在其所在函数第一次调用时定义），也不因为函数的返回而销毁。

6.2.2 函数原型纵览

根据形式参数的不同形式，返回类型的不同形式，将函数原型归纳如下。

1. 函数形式参数类型

在函数原型中，形式参数名称可以被省略，类型名称不可缺少。函数的形式参数分为三类，共五种情形，如表 6-1 所示。

表 6-1 函数形式参数类型

实际要求		形式参数	例如	说明
无		无，或 void	void	
传数值	普通数值	数据类型	int x	定义变量 (创建副本)
	地址值	数据类型 * void * 数据类型 []	int *Ptr void *dest int array[]	
传变量	普通变量	数据类型 &	int &rx	声明引用
	指针变量	数据类型 *&	int *&rPtr	

特别需要说明的是，在函数的形式参数中写下方括号，甚至在方括号中写下所谓的元素个数，都不代表传递整个数组，而只是代表传递数组的地址。函数的形式参数仅获得了数组的地址，数组的元素个数仍需要另外用一个整型参数传递给函数。这种传递方式与指针型的形式参数是等效的。传递多维数组时，最高维的元素个数不必指明，其他低维的元素个数必须明确指出，此时仍为传递一个地址值，即用该地址值（实参）初始化一个数组指针（形参）。最高维的元素个数需要另外传递给函数。

函数形式参数的存储类型为 auto，其生命周期存在于函数被调用之时。函数的形式参数在调用该函数时产生（定义变量或声明引用），用实际参数对其进行初始化（初始化变量或初始化引用）。即函数形式参数定义或声明及其初始化完全遵循局部自动变量定义、局部引用声明及其初始化的语法规规定。函数返回时销毁形式参数。

(1) 无形式参数的函数也就是形式参数为 void 的函数。调用该函数时不需要提供任何实际参数，但是，表示函数的一对圆括号不可缺少。

(2) 传递数值是将数值（右值）单向地传入给函数，所创建的函数形参可视为实参的副本，其数值与实参的数值相等，但副本与实参是相互独立的、无关联的。

(3) 利用传递变量（引用型形式参数）的方法，可通过形式参数“返回”多个数据，实现数据的双向传递。特别地，对于大数据体（第 8 章及其后的结构体变量，或者类的对象）实际参数，即使只要求单向传入给函数，也常采用引用型传递（避免建立副本）以节省时间和空间。当然，实参必须是可以被引用的量。

(4) 采用传递指针值（仅传递一个地址值，形参仅占 sizeof(void*) 字节）也能起到节省时间和空间的作用。就指针本身而言，传递的指针值是单向的，所创建的形参指针副本与实参指针的指向一致，但与实参指针相互独立。函数体内无法修改实参指针的指向。就指针的目标而言，可实现指针目标的双向传递（间接双向传递）。然而这种传递常显得有些“语句不够通顺”（参见第 5.5.1 小节）。

(5) 还可以传递指针的引用，不仅可以操作指针目标处的内容（间接双向传递），还可以改变实参指针本身的指向（直接双向传递）。

直接或间接的双向传递都具有较高的效率，它避免了目标变量副本值的构造（间接双向传递时需要构造指针变量）。常利用这一特点处理那些单向传递（仅读取目标的值）的应用问题。此时，为了保护实参目标不被修改，常用保留字 `const` 进行限制。亦即将形式参数设计成常量指针（所指之处皆常量）或者常量引用（所“绑定”之对象皆常量）。如函数 `int strlen(const char *str);` 就采用了这种保护措施。

2. 函数的返回类型

函数返回类型有三种形式，共分五种情况，如表 6-2 所示。

表 6-2 函数的返回类型

实际要求	例如		说明
无返回	<code>void display(const double *a, int size);</code>		
返回数值	普通数值	<code>int strlen(const char *str);</code>	创建临时变量 只能做右值
	地址值	<code>void *memcpy(void*, const void*, size_t);</code> <code>char *strcpy(char*, const char*);</code>	
返回变量	普通变量	参见例 3.5 或例 6.2	不创建临时变量 直接返回左值量
	指针变量	一般无此必要。参见例 6.16	

其中，`void *memcpy(void *dest, const void *source, size_t length);` 是一个标准函数，原型在头文件 `cstring` 中。其功能是将内存从 `source` 起，`length` 字节的内容复制到内存 `dest` 为起始处的空间中，返回值为 `dest`。数据类型 `size_t` 是某种基本数据类型，一般地将其定义为 `unsigned int`。该函数不管其中的数据是什么类型。需指出的是，程序中不能直接访问 `void *` 处的内容，应该先转换成某种类型的指针，再进行访问，如：

```
int x[100] = {/* 初始化数据（略） */}, y[100];
int *p = (int*)memcpy((void*)y, (void*)x, 100*sizeof(int));
```

例 6.2 返回数值（值返回）与返回变量（引用返回）。请读者比较第 5.5.3 小节例 5.7 程序中的函数 `double *ElementAddr(double *p, int size, int index);`。

源代码 6.2 返回数值与返回变量

```
1 // ref.cpp
2 #include <iostream>
3 using namespace std;
4 double element(double *p, int size, int index)
5 {
6     if(index < 0 || index >= size)
7         exit(-1); // 终止程序运行，返回到操作系统
8     return p[index];
```

```

9 }
10 double &Element(double *p, int size, int index)
11 {                               // 返回数组指定元素本身
12     if(index < 0 || index >= size)
13         exit(-1);           // 终止程序运行, 返回操作系统
14     return p[index];
15 }
16 int main()
17 {
18     double array[10];
19     int i;
20     for(i=0; i<10; i++)
21     {
22         Element(array, 10, i) = 10*i; // 函数调用成为左值
23         cout << element(array, 10, i) << " ";
24     }
25     cout << endl;
26     for(i=0; i<10; i++)
27     {
28         // element(array, 10, i) = 100*i;           此行有错
29         // error '=': left operand must be l-value   编译时的错误信息
30         cout << array[i] << " ";
31     }
32     cout << endl;
33     return 0;
34 }

```

程序中函数 `Element` 是返回变量（引用返回）的，直接用函数调用表达式表示其所返回的变量。该变量是可以进行读写访问的，即函数调用表达式成为所返回的变量的别名，可以做左值。而函数 `element` 为返回数值的，函数返回时将新创建（定义）一个临时变量暂存其返回值，该临时变量稍后便被销毁。一般地，这种量不能做左值，或者做左值是无意义的。下面的代码在编译时将给出警告。

```

1 int &f(int a)
2 {
3     return a;
4 }

```

因为变量 `a` 的生命期是局部的。该函数调用表达式将成为变量 `a` 的别名，而变量 `a` 的生命期短于其别名的生命期。因而是不正确的。

6.3 函数版“评委评分”程序设计

6.3.1 功能模块设计

以堆数组版“评委评分”程序为蓝本进行改编，使程序模块化。可以考虑将程序按功能划分为如下 7 个模块。

模块名称	初始化 <code>Init</code>
功 能	系统配置（确定程序中的参数）：评委人数、选手人数
输入信息	两个待修改值的变量（双向传递）
返回信息	评委人数小于 2 时返回 -1，否则返回 0
函数原型	<code>int Init(int &referees, int &players);</code>

模块名称	分配堆内存空间 <code>GetMemory</code>
功 能	根据评委人数值、选手人数值，申请适当的堆空间
输入信息	评委人数值、选手人数值(单向传递)；三个待明确指向的指针变量(直接双向传递)
返回信息	分配不成功时返回 -1，否则返回 0
函数原型	<pre>int GetMemory(int referees, int players, double *&x, double *&aver, int *&rank);</pre>
模块名称	释放所分配的堆内存空间 <code>FreeMemory</code>
功 能	释放由 <code>GetMemory</code> 模块所申请的堆空间
输入信息	三个指向堆内存的指针变量(释放堆空间后，置为空指针)(直接双向传递)
返回信息	无
函数原型	<pre>void FreeMemory(double *&x, double *&aver, int *&rank);</pre>
模块名称	获取原始分数 <code>GetData</code>
功 能	生成各种原始分数
输入信息	评委人数值，选手人数值(单向传递)，存放原始分数的地址值(间接双向传递)
返回信息	无
函数原型	<pre>void GetData(int referees, int players, double *x);</pre>
模块名称	计算各选手的最后得分 <code>ComputingAver</code>
功 能	根据原始分数计算各选手的最后得分
输入信息	评委、选手人数(单向)，原始分数地址(受保护的间接双向)，最后得分地址(间接双向)
返回信息	无
函数原型	<pre>void ComputingAver(int referees, int players, const double *x, double *aver);</pre>
模块名称	计算各选手的名次 <code>ComputingRank</code>
功 能	根据各选手的最后得分计算各选手的名次
输入信息	选手人数(单向传递)，最后得分地址(受保护的间接双向传递)，名次地址(间接双向传递)
返回信息	无
函数原型	<pre>void ComputingRank(int players, const double *aver, int *rank);</pre>
模块名称	输入详细的数据表 <code>Display</code>
功 能	输出一张详细数据表
输入信息	评委、选手人数(单向传递)，各种数据地址(受保护的间接双向传递)
返回信息	无
函数原型	<pre>void Display(int referees, int players, const double *x, const double *aver, const int *rank);</pre>

6.3.2 功能实现——函数定义

上述模块基本上符合“相对独立、功能单一且模块之间接口清晰”的要求。一般地，具体的且功能单一的模块实现（函数定义）是简单的。此时可抛开其他所有的干扰——因为能够被利用的信息仅有那几个以形式参数表示的从函数输入接口接收到的数据，根据合适的算法编写语句即成。

源程序开始的部分为包含一些头文件，使用 std 名字空间。其后是各模块的具体实现，其中未列出的行号为各模块之间的空行。

源代码 6.3 函数版“评委评分”程序

```

1 // contest3.cpp          函数版“评委评分”程序
2 #include <iostream>
3 #include <iomanip>
4 #include <ctime>
5 using namespace std;
```

1. 初始化

```

7 int Init(int &referees, int &players)
8 {
9     cout << "请输入评委人数(例如10)：" ;
10    cin >> referees;
11    if(referees<=2)
12    {
13        cout << "评委人数太少。" << endl;
14        return -1;           // 此处不退出程序的运行，返回 -1 给调用者处理
15    }
16
17    cout << "请输入参赛选手人数(例如20)：" ;
18    cin >> players;
19    if(players < 1)
20    {
21        cout << "参赛选手人数错误。" << endl;
22        return -1;
23    }
24    return 0;
25 }
```

2. 分配堆内存空间

```

27 int GetMemory(int referees, int players,
28                 double *&x, double *&aver, int *&rank)
29 {
30     x = aver = NULL;
31     rank = NULL;
32     x = new double[referees * players];
33     if(x==NULL) return -1;           // 动态内存分配失败
34
35     aver = new double[players];
36     if(aver==NULL)
37     {
```

```

38     delete [] x;
39     x = NULL;
40     return -1;
41 }
42
43 if( (rank = new int[players])==NULL )
44 {
45     delete [] x;
46     delete [] aver;
47     x = aver = NULL;
48     return -1;
49 }
50 return 0;                                // 动态内存分配成功
51 }

```

3. 释放所分配堆内存空间

```

53 void FreeMemory(double *&x, double *&aver, int *&rank)
54 {
55     if(x) delete [] x;
56     if(aver) delete [] aver;
57     if(rank) delete [] rank;
58     x = aver = NULL;
59     rank = NULL;
60 }

```

4. 获取原始分数

```

62 void GetData(int referees, int players, double *x)
63 {                                         // 随机产生原始分数
64     int i, j;
65     time_t t;
66     srand(time(&t));
67     for(i=0; i<players; i++)
68         for(j=0; j<referees; j++)
69             x[i*referees + j] = (80 + rand() % 21)/10.0;
70 }

```

5. 计算各选手的最后得分

```

72 void ComputingAver(int referees, int players, const double *x,
73                      double *aver)
74 {
75     int i, j;
76     double min, max, sum;
77
78     for(i=0; i<players; i++)
79     {
80         min = 10;
81         max = sum = 0;
82         for(j=0; j<referees; j++)
83         {
84             sum += x[i*referees + j];
85             if(x[i*referees + j]>max) max = x[i*referees + j];

```

```

86             if(x[i*referees + j]<min) min = x[i*referees + j];
87         }
88         aver[i] = (sum-max-min)/(referees-2);
89     }
90 }
```

6. 计算各选手的名次

```

92 void ComputingRank(int players, const double *aver, int *rank)
93 {
94     int i, j;
95     for(i=0; i<players; i++)
96     {
97         rank[i] = 1;
98         for(j=0; j<players; j++)
99             if(aver[j]>aver[i]) rank[i]++;
100    }
101 }
```

7. 输出详细的数据表

```

103 void Display(int referees, int players, const double **x,
104                 const double *aver, const int *rank)
105 {
106     int i, j;
107     cout << setprecision(3) << showpoint;
108     cout << "\nNo";
109     for(j=0; j<referees; j++)
110         cout << "....." << char('A'+j);
111     cout << ".....Aver.....Rank" << endl;
112     for(i=0; i<players; i++)
113     {
114         cout << setw(3) << i+1;
115         for(j=0; j<referees; j++)
116             cout << setw(6) << x[i*referees + j];
117         cout << setw(6) << aver[i] << setw(6) << rank[i]<<endl;
118     }
119 }
```

8. 功能组装

```

121 int Competition()
122 {
123     int referees, players;          // 定义变量
124     double *x, *aver;              // 定义指针变量
125     int *rank;                    // 定义指针变量
126     if(Init(referees, players)==-1) return -1;
127     if(GetMemory(referees, players, x, aver, rank)==-1) return -2;
128     // 分配堆内存空间
129     GetData(referees, players, x);
130     ComputingAver(referees, players, x, aver);
131     ComputingRank(players, aver, rank);
132     Display(referees, players, x, aver, rank);
133     FreeMemory(x, aver, rank);   // 释放堆内存空间
}
```

```

134     return 0;
135 }
136 }
```

9. 主函数

```

138 int main()
139 {
140     Competition();
141     return 0;
142 }
```

由于所有的函数均在被调用之前定义，故程序中省略了用函数原型进行函数声明。

6.4 递归函数

递归函数是指在函数体内部直接或间接地调用该函数自身的函数。

例 6.3 递归计算阶乘 $n!$ 。有数学公式

$$\begin{cases} 0! = 1! = 1 \\ n! = n \times (n-1)! \quad n = 1, 2, 3, \dots \end{cases} \quad \text{即} \quad \begin{cases} f(0) = f(1) = 1 \\ f(n) = n \times f(n-1) \quad n = 1, 2, 3, \dots \end{cases}$$

写成 C++ 函数如下。

源代码 6.4 计算阶乘的递归函数

```

1 unsigned long fact(int n)
2 {
3     if(n < 0) return 0;           // 排除实参可能的意外情况
4     if(n == 0 || n == 1)         // 终止进一步递归的条件
5         return 1;
6     return fact(n-1)*n;          // 递归调用
7 }
```

由函数调用的栈操作机制可知，每进行一次递归调用，都会产生一个压栈过程，都会“保存状态，转移控制”并定义一个形式参数 n 。不同层次中的形式参数变量 n 存放在栈空间中的不同地址处，存放着不同的数值。当不再进一步递归调用时，函数才逐步从栈空间中按后进先出的顺序依次退栈（包括依次建立及销毁各层函数值返回的临时变量），返回到调用它的函数。若递归的深度太大，可能导致栈溢出而错。

以计算 $\text{fact}(3)$ 为例分析其递归调用过程、返回过程：

① 递归调用过程。调用 $\text{fact}(3)$ ；由于 $3 > 1$ 而继续递归调用 $\text{fact}(2)$ ；同理，继续递归调用 $\text{fact}(1)$ 。如图 6-2 自下而上所示。

② 返回过程。当执行 $\text{fact}(1)$ 调用时由于满足“终止进一步递归”的条件而直接返回常数 1，按照退栈的次序，返回到 $\text{fact}(2)$ 中的表达式 $\text{fact}(1)*2$ （其中 $\text{fact}(1)$ 表示值为 1 的一个临时变量，2 为所在函数的形参 n 的值）计算得到乘积 2 紧接着返回到 $\text{fact}(3)$ 中的表达式 $\text{fact}(2)*3$ ，进一步计算乘积得到 6，返回。

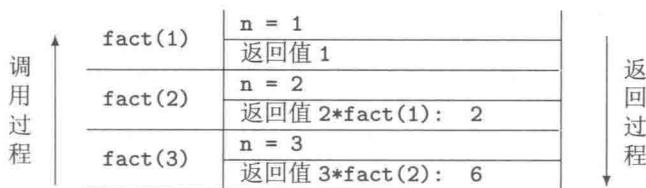


图 6-2 递归调用及返回过程的栈空间结构示意图

例 6.4 给定两个非负整数 m, n , 计算它们的最大公因数。第 4.2.3 小节例 4.5 给出了其递归公式并用循环算法实现。直接根据递归公式的递归函数如下。

```

1 unsigned int gcd1(unsigned int m, unsigned int n)
2 {
3     if(n==0) return m;
4     return gcd1(n, m%n);
5 }
```

递归函数在执行时具有较高的空间复杂度（需要较多的内存空间）和时间复杂度（因函数调用的辅助操作需要的计算时间稍长）。但是，递归程序结构清晰，可读性好。理论上已经证明所有的递归算法都能用循环结构实现。

递归函数设计时，应该先进行是否进一步递归调用的条件判断，确有必要时才进行进一步递归调用。先进行判断十分重要，使用不当则可能造成无穷递归导致栈溢出而使程序崩溃。

再举一个经典的递归函数的例子。

例 6.5 (汉诺塔问题) 有三根针 A 、 B 、 C ，如图 6-3 所示。 A 针上有 n 个大小不等的盘子，大盘子在下小盘子在上。要求将这 n 个盘子从 A 针借助 B 针移动到 C 针。规定每次只能移动一个盘子，并且始终保持大盘子在下小盘子在上。

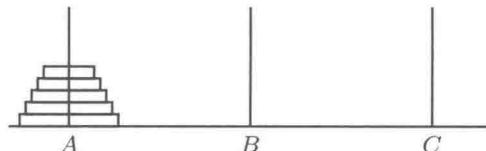


图 6-3 汉诺塔问题

【分析】 将 n 个盘子从 A 针借助 B 针移动到 C 针可以分解为下面的三个步骤：

- ① 将 A 针上的 $n - 1$ 个盘子借助 C 针移动到 B 针。
- ② 将 A 针上的第 n 个盘子直接移动到 C 针。
- ③ 将 B 针上的 $n - 1$ 个盘子借助 A 针移动到 C 针。

经过上述分析，将 n 个盘子的移动问题，缩减成 $n - 1$ 个盘子的移动问题。即问题的类型不变但规模缩减。当问题的规模小到一定程度时直接得到问题的解。

于是，有解汉诺塔问题的递归算法程序如下。

源代码 6.5 汉诺塔

```

1 // Hanoi.cpp      汉诺塔
2 #include <iostream>
3 using namespace std;
4
5 void move(char x, char y)
6 {
7     cout << x << "--->" << y << endl;
8 }
9
10 void Hanoi(int n, char x, char y, char z)
11 {                                // 将n个盘子从x针借助y针移动到z针
12     if(n==1)
13         move(x, z);
14     else
15     {
16         Hanoi(n-1, x, z, y);
17         move(x, z);
18         Hanoi(n-1, y, x, z);
19     }
20 }
21
22 int main()
23 {
24     int n;
25     cout << "请输入盘子的个数: ";
26     cin >> n;
27     cout << "移动过程: " << endl;
28     Hanoi(n, 'A', 'B', 'C');
29     return 0;
30 }
```

程序的运行结果：

请输入盘子的个数：3

移动过程：

A --> C
 A --> B
 C --> B
 A --> C
 B --> A
 B --> C
 A --> C

6.5 函数重载

在 C 语言中，不同的函数不能使用相同的函数名，以保证编译系统能正确地区分和处理。例如，在 C 语言中，计算给定参数的绝对值函数须有多个：

<pre>int abs(int);</pre>	// 计算整型数据的绝对值
<pre>long labs(long);</pre>	// 计算长整型数据的绝对值
<pre>double fabs(double);</pre>	// 计算浮点型数据的绝对值

对于程序员而言，当然希望有一个统一名称的函数能分别实现上述三个函数的功能。本着将方便让给程序员，把困难留给编译器的精神，C++语言提供了函数重载的支持。本质上，在编译系统进行实质性编译时仍然不允许不同的函数使用相同的函数名。编译系统根据函数的形式参数数据类型、顺序、个数信息，悄悄地修改了程序员给函数的命名，然后进行编译和连接。

因此，程序员在进行函数重载时，必须满足下列要求之一。

- (1) 函数的形式参数个数不同。
- (2) 形式参数个数相同时，参数的数据类型不同，或数据类型出现的次序不同。

例如：

```
char Max(char, char);
char *Max(char*, char*);
int Max(int, int);
double Max(double, double);
int Max(int, int, int);
int Max(int, int, int, int);
```

均构成函数重载。但是 `int Max(int&, int&);` 与上述函数却不能构成重载，因为

```
int a=1, b=2, c;
c = Max(a, b);
```

无法确定匹配 `int Max(int, int);` 还是匹配 `int Max(int&, int&);` 函数。

仅形式参数名称不同，仅函数的返回类型不同均不构成函数重载。

规定 6.1（匹配重载函数的规则） C++ 按下列三个步骤的次序寻找匹配的重载函数：

- (1) 根据数据类型寻找一个严格的匹配。
- (2) 通过数据类型兼容性的隐式转换（自动转换）寻找一个匹配。
- (3) 通过用户定义的类型转换寻找一个匹配（参见第 13.4 节）。

倘若经过上述三种寻找仍不能匹配一个重载函数，则编译系统放弃匹配并给出错误报告。

6.6 参数带默认值的函数

仍然是将方便让给程序员，把困难留给编译器。C++编译器支持参数带默认值的函数。例如，在下面的函数中，形式参数类型为字符指针，其后有一设定的值为该参数的默认值。该函数被称为参数带默认值的函数。

例 6.6 参数带默认值的函数示例。

```
1 #include <conio.h>
2 int Pause(char *prompt="按任意键继续....."); // 函数原型
3
4 int Pause(char *prompt) // 函数定义
5 {
6     cout << prompt << flush;
7     return getch();
8 }
```

该函数在调用时，可以有以下两种形式。

(1) 不提供实参。此时编译系统使用默认值。如：

```
Pause();
```

(2) 提供实参。此时编译系统不使用默认值，而使用程序员提供的实参数。如：

```
Pause("Press any key to continue...");  
int flag = Pause("确实要删除该记录吗？(y/[N])");
```

在一个函数中可以有多个参数带默认值。调用这种函数时，无默认值的参数不能缺省实参，实参从左边依次初始化形参，其余的形参使用其默认值。因此，函数设计时应将带默认值的形参全部安排在参数表的右边。

参数的默认值只应出现在函数声明中。若无函数原型声明函数（即函数定义兼做函数声明）时，才允许在函数定义时声明参数默认值。有函数原型声明时，函数定义中不应出现参数默认值。

6.7 内联函数

调用函数及函数返回都需要一些辅助操作而耗费一定的时间。为了提高程序执行的速度，C++ 语言提供了一种机制，允许一些小巧的、常用的函数在编译时直接内嵌到调用它的函数体内。即在调用点“就地展开”。这样的函数已经成了调用点处的一段代码，本质上已经不是函数。之所以仍将其称为函数，是因为在形式上它和函数几乎是相同的。C++ 采用这样的做法是用空间换时间——程序最后生成的可执行文件的尺寸增大，但程序的执行时间缩短了。在兼做函数声明的函数定义前加上保留字 `inline` 可将该函数设计成内联函数。

例 6.7 内联函数示例。

源代码 6.6 内联函数示例程序

```
1 // ex_inline.cpp  
2 #include <iostream>  
3 #include <conio.h>  
4 using namespace std;  
5  
6 inline int isdigit(char ch) // 内联函数定义  
7 {  
8     return (ch>='0' && ch<='9') ? 1 : 0;  
9 }  
10  
11 int main()  
12 {  
13     int key, n=0, digit=0;  
14  
15     cout << "请输入字符，按ESC键退出：" << endl;  
16     do  
17     {  
18         key = getch(); // 暂停，等待用户按任意键  
19         n++;  
20         if(isdigit(key)) digit++;  
21     }while(key!=27); // 27 为 ESC 键的编码
```

```

22     cout << "\n共按键" << n << "次，其中数字键"
23         << digit << "次。" << endl;
24     return 0;
25 }
```

内联函数中不能含有复杂的流程控制语句（如 `switch`, `while`）；递归函数不可能在调用点“就地展开”因而不能是内联函数。如果编译系统不能将程序员设计的内联函数“就地展开”，编译系统就按照普通的函数进行编译。也就是说，程序员在程序中将某个函数设置成内联的，只是程序设计者对编译器提出的一个建议。对程序员建议的内联函数，编译系统不是一定将该函数按内联函数处理的。

6.8 函数模板

函数模板更加体现了C++语言将方便让给程序员，把困难留给编译器的精神。C++允许程序员在源程序级将那些语句完全相同，仅数据类型不同的函数做成模具——函数模板（`template`）。在编译过程中遇到相应的函数调用时，由编译系统根据函数模板自动生成模板函数然后进行编译。省却了程序员编写一批重载函数还不一定能穷尽所有数据类型的麻烦。

6.8.1 描述函数模板

函数模板描述声明的格式为

```

template <typename 形式数据类型表> 返回类型 函数模板名(形式参数表)
{
    // 函数模板体
}
```

例 6.8 返回给定的两个参数之中的较大值的函数模板。

```

1 template <typename T> T Max(T a, T b)
2 {
3     return (a>b) ? a : b;
4 }
```

这里，`template`, `typename` 都是 C++ 的保留字。`T` 是程序员自行命名的标识符，代表某种数据类型。`T` 只是形式上的，因此称其为**形式数据类型**。该函数模板名为 `Max` 有两个形式参数，其数据类型均为 `T` 所代表的数据类型；该函数有返回类型，仍然为 `T` 所代表的数据类型。在函数模板中不变的是操作，可变的是数据类型。

6.8.2 模板函数的使用

函数模板只是描述了对某种假想的数据类型的数据将要进行的操作，编译系统不会编译函数模板本身，因此，我们称模板为声明（它仅描述了可能进行的操作），而非定义。当编译系统遇到用具体的实际参数使用函数模板时，便用实际参数的数据类型取代形式数据类型 `T`、根据函数模板体的执行语句，自动地为程序员定义一个函数——称为**模板**

函数，亦称为函数模板的实例化。然后对所生成的模板函数进行编译。接例 6.8 中的程序代码：

```

5 #include <iostream>
6 using namespace std;
7
8 int main()
9 {
10     int a=3, b=5;
11     char c='3', d='5';
12     double x=3.5, y = 5.5;
13     char s[10]="ABCD", t[10]="abcd";
14
15     cout << Max(a, b) << endl;
16     cout << Max(c, d) << endl;
17     cout << Max(x, y) << endl;
18     cout << Max(s, t) << endl;
19
20     return 0;

```

程序的运行结果：

```

5
5
5.5
ABCD

```

编译系统在编译主函数时，将遇到四次实际参数数据类型均不相同的 Max 函数调用。编译系统会根据实际参数的数据类型、函数模板体的执行语句，自动生成如下四个模板函数（四种不同的实例化）并进行编译。

```

1 int Max(int a, int b)          // 比较两个整型数值的大小, T 为 int
2 {
3     return (a>b) ? a : b;
4 }
5
6 char Max(char a, char b)       // 比较两个字符的大小, T 为 char
7 {
8     return (a>b) ? a : b;
9 }
10
11 double Max(double a, double b) // 比较两个浮点数的大小, T 为 double
12 {
13     return (a>b) ? a : b;
14 }
15
16 char *Max(char *a, char *b)    // 比较两个地址值的大小, T 为 char*
17 {
18     return (a>b) ? a : b;
19 }

```

由此可见，实际参数的数据类型应该支持函数模板中描述的各种运算操作，否则编译系统生成的模板函数将不能通过编译。从结果可见 `char *Max(char *a, char *b);` 函数并非比较字符串的内容而仅比较了意义不大的存放地的先后（地址值的大小），改进方

法见第 6.8.3 小节。

例 6.9 交换两个实参变量值的函数模板。

```

1 template <typename T> void Swap(T &x, T &y)      // 引用型形参
2 {
3     T temp;
4     temp = x; x = y; y = temp;
5 }
6
7 template <typename T> void Swap(T *x, T *y)      // 指针型形参
8 {
9     T temp;
10    temp = *x; *x = *y; *y = temp;
11 }
```

6.8.3 重载模板函数

再看上面的例 6.8，其中有一个比较两个字符串地址值大小的模板函数。一般情况下，这样的比较是没有意义的。当我们希望比较两个字符串内容的大小时，还需要重载模板函数。

在上面的例子中，若事先声明且定义了如下函数。

```

1 char *Max(char *a, char *b)
2 {
3     return strcmp(a, b)>=0 ? a : b;      // 字符串内容比较
4 }
```

则编译系统首先匹配重载模板函数，而不会生成前面的第四个函数去比较两个字符串的起始地址值的大小。

6.9 函数应用

6.9.1 静态局部变量的特性

静态局部变量是在函数内部定义的静态变量。静态局部变量于其所在函数第一次被调用时创建，它存放在全局数据区，其生命期不受函数调用栈操作的影响，直到整个程序结束才结束。因此，静态局部变量最多只被初始化一次。当再次调用其所在函数时，将跳过其定义语句，该静态局部变量被“唤醒”，其值是上次函数返回时它所保存的值。一般地，静态局部变量只能在定义它的函数内部被访问。即静态局部变量具有全局生命期和局部可见性。

下面介绍一个应用静态局部变量的例子。其中用到在头文件 `ctime` 中声明的函数

```
clock_t clock(void);      // clock_t 被定义为 long
```

该函数返回程序自启动之时起，到当前时刻所用的时间（以毫秒为单位）。

例 6.10 测试计算时间。要求编写一个函数 `gettime` 具有如下功能：调用该函数可设置计时起点，可返回从所设置的计时起点到目前所间隔的时间（秒）。然后编写一个程序测试该函数。

下面的程序模拟“摇奖”选数过程。程序运行时不断地产生 $[0, 1]$ 区间上的随机数，直到用户按任意键。此时的随机数被“选中”。程序中顺便统计了所经历的时间、所产生的随机数个数、最大随机数值、最小随机数值和平均值。用户可反复按键选数，直到按 ESC 键结束程序。

源代码 6.7 “摇奖” 程序

程序运行时可出现类似下面的结果，若按 ESC 键则程序运行结束。

```

Time: 1.612s    0.757775
n: 3207, 选中 0.757775 [min: 0.000183111, max: 0.999786, aver: 0.501226]
Time: 3.515s    0.171941
n: 7448, 选中 0.171941 [min: 6.1037e-005, max: 0.999969, aver: 0.500125]
Time: 2.063s    0.844356
n: 4346, 选中 0.844356 [min: 0.000366222, max: 1, aver: 0.500661]

```

6.9.2 排序

给定元素个数为 n 的数组，要求将元素按照其值从小到大或者从大到小的顺序重新排列，这一过程被称为排序。排序是计算机科学与技术领域中的一个基本问题，应用十分广泛。

排序操作总是通过比较和交换进行的。不同的排序算法采用不同的策略，其比较和交换的次数是不尽相同的，有些算法的效率低，有些算法的效率高。下面介绍两种排序算法，一种是效率不高但容易理解的冒泡排序法，另一种是效率高的快速排序法。

排序算法只用到元素值的比较和交换两种操作。为了适应更多的数据类型，将排序算法设计成函数模板。只要某种数据类型能进行比较及赋值运算便可以使用本模板。

1. 冒泡排序法

例 6.11 冒泡排序函数模板（升序）。

冒泡排序法的基本思想是反复比较相邻两个元素的值，必要时交换它们的位置。经过第一轮从头到第 n 个元素的处理，其中最大值必然被置换到最后（沉底）。经过第二轮从头到第 n-1 个元素的处理，其中次最大元素必然位列 n-1。依此类推，直到经过 n-1 轮后，全部数据排列有序。因为这种算法的每一次交换都将值小者向前交换，故称为冒泡排序法。函数模板设计如下：

函数名	Bubble
数组数据类型	形式类型 typename T
形式参数	数组首地址指针，元素个数
返回类型	void

源代码 6.8 冒泡排序（升序）函数模板

```

1 template <typename T> void Bubble(T *a, int size)
2 {
3     T temp;      // 定义一个局部变量，数据类型与形式数据类型相同
4     int i, j;
5     for(i=1; i<size; i++)          // 共进行size-1轮比较和交换
6     {
7         for(j=0; j<size-i; j++)
8         {
9             if(a[j]>a[j+1])      // 相邻元素之间比较，必要时
10            {
11                temp = a[j];    // 交换 a[j] 与 a[j+1]
12                a[j] = a[j+1];
13                a[j+1] = temp;
14            }
15        }
}

```

```

16 }
17 }
```

测试举例（代码片段）。

```

1 // 插入必要的头文件及冒泡排序函数模板
2 template <typename T> void Display(const T *a, int size)
3 {
4     cout << *a++;           // 输出数组各元素的值
5     for(int i=1; i<size; i++)
6         cout << ", " << *a++;
7     cout << endl;
8 }
9
10 int main()
11 {
12     int iArray[10] = {12, 56, 34, 87, 11, 3, 6, 36, 34, 89};
13     char cArray[10] = {'A', 'b', '+', ' ', '2', '!', 'Q', 'S', 'X', 'z'};
14
15     Display(iArray, 10);    // 输出原始数据
16     Bubble(iArray, 10);    // 排序
17     Display(iArray, 10);    // 输出排序结果
18     Display(cArray, sizeof(cArray)/sizeof(char));
19     Bubble(cArray, sizeof(cArray)/sizeof(cArray[0]));
20     Display(cArray, sizeof(cArray)/sizeof(*cArray));
21     return 0;
22 }
```

请读者比较上述函数模板 `Display` 与例 5.5 相应函数在实现技术上的不同点。并分析表达式 `*a++` 与 `(*a)++` 的区别。程序中第 18~20 行函数调用语句中的第二个实参均为 10，以第 19~20 行者为佳。

2. 快速排序法

例 6.12 快速排序函数模板（升序）。

快速排序法是一种高效的排序算法。它建立在数据分组的基础上，采取“分而治之”的思想。在元素个数为 n 的数组中选择一个分界值，将数组元素按该分界值分成两个部分：小于等于分界值、大于等于分界值。将问题变成两个规模较小的同类问题，采用递归算法处理。因此，需要

- ① 选定分界值。
- ② 分别从左右两端开始与分界值比较、交换，形成左右两个小组。
- ③ 递归处理左右两个小组。

源代码 6.9 快速排序（升序）函数模板

```

1 template <typename T> void Qsort(T *a, int size)
2 {
3     T pivot, temp;
4     int left=0, right=size-1;           // 下标（整数）
5     if(size<=1) return;
6     pivot = a[right];                  // 选择最后一个值为分界值
7     do
8     {
```

```

9      while(left<right && a[left] <= pivot) left++;
10     while(left<right && a[right] >= pivot) right--;
11     if(left < right)
12     {
13         temp=a[left]; a[left]=a[right]; a[right]=temp;
14     }
15 }while(left < right);
16 a[size-1] = a[left]; a[left] = pivot; // 找到分界点 left
17 Qsort(a, left); // 递归调用 (左侧部分)
18 Qsort(a+left+1, size-left-1); // 递归调用 (右侧部分)
19 }

```

下面编写一个测试程序，测试三个排序函数的执行时间。这三个排序函数分别是：本书中的冒泡排序模板 Bubble、本书中的快速排序模板 Qsort 和系统提供的快速排序函数 qsort。为节省篇幅，程序中用到的已经在教材前面出现的函数、函数模板未列入，请读者实验时自行补上。

例 6.13 几种排序算法函数执行时间的比较。

源代码 6.10 排序算法时间测试程序

```

1 // SortTest.cpp
2 #include <iostream>
3 #include <ctime>
4 using namespace std;
5
6 double gettime(int restart=0);
7 template <typename T> void Qsort(T *a, int size);
8 template <typename T> void Bubble(T *a, int size);
9
10 // 需要将前面介绍的相应函数及模板插入到此处
11
12 int compare(const void *a, const void *b)
13 { // 为使用标准快速排序函数 qsort 设计的比较函数
14     return (*((double*)a) > *((double*)b)) ? 1 :
15         (((double*)a) < *((double*)b)) ? -1 : 0; // 为了升序
16 }
17
18 int main()
19 {
20     int n, i;
21     double *data0, *data, t;
22     time_t t1;
23     srand(time(&t1));
24     for(n=1024; n<=65536; n*=2)
25     {
26         data0 = new double[n];
27         data = new double[n];
28         for(i=0; i<n; i++)
29             data0[i] = data[i] = rand()/(double)RAND_MAX;
30
31         gettime(1);
32         Bubble(data, n); // 冒泡排序
33         t = gettime();
34         cout << "BubbleSort: n=" << n << ", t=" << t << endl;
35
36         for(i=0; i<n; i++)

```

```

37         data[i] = data0[i];    // 恢复原始数据
38         gettimeofday(&t);
39         Qsort(data, n);        // 本教材中设计的快速排序模板
40         t = gettimeofday();
41         cout << "QuickSort: n=" << n << ", t=" << t << endl;
42
43         for(i=0; i<n; i++)
44             data[i] = data0[i];    // 恢复原始数据
45         gettimeofday(&t);
46         qsort((void*)data, n, sizeof(double), compare);
47         t = gettimeofday();      // 系统提供的标准快速排序函数
48         cout << "quickSort: n=" << n << ", t=" << t << endl;
49         delete [] data0;
50         delete [] data;
51     }
52     return 0;
53 }
```

该程序某次在微型计算机上的运行结果如下， t 为将元素个数为 n 的双精度浮点型数组排序所需要的时间（秒），各种排序函数处理的同一规模的数据是完全相同的。

```

Bubble Sort: n = 1024, t = 0.01
Quick Sort: n = 1024, t = 0
quick Sort: n = 1024, t = 0
Bubble Sort: n = 2048, t = 0.04
Quick Sort: n = 2048, t = 0
quick Sort: n = 2048, t = 0.01
Bubble Sort: n = 4096, t = 0.18
Quick Sort: n = 4096, t = 0
quick Sort: n = 4096, t = 0.01
Bubble Sort: n = 8192, t = 0.661
Quick Sort: n = 8192, t = 0
quick Sort: n = 8192, t = 0
Bubble Sort: n = 16384, t = 2.543
Quick Sort: n = 16384, t = 0
quick Sort: n = 16384, t = 0.011
Bubble Sort: n = 32768, t = 10.224
Quick Sort: n = 32768, t = 0.01
quick Sort: n = 32768, t = 0.01
Bubble Sort: n = 65536, t = 40.879
Quick Sort: n = 65536, t = 0.02
quick Sort: n = 65536, t = 0.04
```

由运行结果可见，快速排序比冒泡排序算法快许多倍，并且数据元素个数越多，它们之间的耗时差别越大。因此，实际应用程序中切勿使用冒泡排序算法。

系统提供的快速排序标准函数原型为

```
void qsort(void *, size_t nelem, size_t width,
           int (*fcmp)(const void *, const void *));
```

其中，第一个参数为待排序数组首地址；第二个参数 $nelem$ 为数组的元素个数；第三个参数 $width$ 是元素数据类型的长度（字节数）；第四个参数 $fcmp$ 是一个函数指针（参见第 6.9.3 小节规定 6.2）指向用户设计的比较函数。

比较函数的返回类型应该为 `int`，该函数名由用户命名，函数的形式参数类型必须为 `const void*`, `const void*`。如果两个形式参数的目标变量值相等则返回 0，如果前者大于后者则返回正数（负数），否则返回负数（正数），为升（降）序。

由于比较函数通过形式参数接收到的两个地址 `a` 和 `b` 是无类型的，在比较函数体内应该将它们强制转换成待排序的数据类型指针（还其本来面目）。在上面的程序中，由于待排序的数组为 `double` 型，所以比较函数体内首先将形参强制转换成 `double*` 类型，再取其内容（间接访问），即等价于如下函数。

```

1 int compare(const void *a, const void *b)
2 {
3     double *x, *y;
4     x = (double*)a;           // 还其本来“面目”
5     y = (double*)b;           // 之后进行运算
6     return (*x > *y) ? 1 : ((*x < *y) ? -1 : 0);
7 }
```

更简捷，但稍难理解的写法是程序中采用的。若换成如下定义，则排序的结果为降序。

```

1 int compare(const void *a, const void *b)
2 {
3     return (*(double*)a > *(double*)b) ? -1 :
4         (*(double*)a < *(double*)b) ? 1 : 0; // 为了降序
5 }
```

由于系统提供的标准函数 `qsort` 具有通用性，辅助操作稍多（如调用比较函数），故稍慢于 `Qsort` 模板。

6.9.3 定积分计算

C++ 程序运行时，编译后的函数代码存放在内存的代码区，函数名是函数的入口地址。可以定义指向函数的指针变量间接地访问函数，亦即间接地调用函数。

规定 6.2 (函数指针) 指向函数入口地址的指针变量被称为函数指针。函数指针变量定义的语法格式为

目标函数返回类型 <code>(*函数指针名)(目标函数形式参数表)</code> = 已存在的函数名；
--

一个指向函数的指针变量，可以指向一类函数中的任意一个。其目标函数应该具有相同形式参数类型、相同返回类型。例如，如下语句定义了一个指向函数的指针变量 `f`。该指针只能指向带有一个 `double` 型形式参数且返回类型为 `double` 的函数。

<code>double (*f)(double) = sin;</code>

上述语句还用正弦函数 `sin` 的入口地址 `sin` 对该指针进行了初始化，即该指针指向正弦函数的入口地址。那么，函数指针间接访问其目标就是调用该目标函数。函数调用表达式 `f(x)` 与 `sin(x)` 等效。之所以称等效而不是等价，是因为指针 `f` 是变量而 `sin` 是地址常量。变量还可以改变其值，如赋值运算 `f=cos;` `f=sqrt;` 等。

若函数 $f(x)$ 在区间 $[a, b]$ 上可积, 取一个适当小的正数, 如 $\Delta x = 0.0001$, 则

$$I = \int_a^b f(x)dx \approx \sum_{a \leq x_i \leq b} f(x_i)\Delta x \quad x_i = a + (i + 0.5)\Delta x, \quad i = 0, 1, 2, \dots$$

希望编写一个函数用于计算一元数学函数的定积分, 此时作为函数形式参数的应该有指向被积函数的指针变量(函数指针)、积分下限和积分上限。函数返回积分值。

上述公式是计算定积分的中矩形公式, 其计算的速度、精度均不太高。若需要高速度、高精度的定积分计算, 可参考任何一本有关数值分析方面的教材(《数值分析》或《计算方法》)。本例展示如何将函数做实际参数传给另一个函数的方法。

例 6.14 定积分计算。

源代码 6.11 定积分计算

```

1 // integral.cpp
2 #include <iostream>
3 #include <cmath>
4 using namespace std;
5
6 double integral(double (*f)(double), double a, double b)
7 {   // 函数的形式参数f为指向函数入口地址的指针
8     double s=0, dx=0.0001, x;
9     for(x=a+dx/2; x<b; x+=dx) // 循环初值, 继续循环条件、步长
10       s += f(x);           // 用指针访问函数, 即调用函数
11     return s*dx;
12 }
13 double f1(double x)
14 {
15     return 4/(1+x*x);      // 其原函数为 4*arctan(x)
16 }
17 double f2(double x)
18 {   // 标准正态分布密度函数, 该函数的原函数不能表成有限形式
19     const double u = 1/sqrt(2*3.1415926);
20     return u*exp(-x*x/2);
21 }
22
23 int main()
24 {
25     cout << integral(f1, 0, 1) << endl;
26     cout << integral(f2, -3, 3) << endl;
27     cout << integral(sin, 0, 3.1415926/2) << endl;
28     cout << integral(sqrt, 0, 1) << endl;
29     cout << integral(exp, 0, 1) << endl;
30     return 0;
31 }
```

程序的运行结果:

```

3.14159
0.9973
1
0.666667
1.71828
```

对运行结果逐行说明如下：

$$\int_0^1 \frac{4}{1+x^2} dx = 4 \arctan x \Big|_0^1 = \pi \approx 3.14159 \quad (1)$$

$$\int_{-3}^3 \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx \approx 0.9973 \quad (2)$$

为正态分布的“ 3σ 原则”。即服从正态分布 $N(\mu, \sigma^2)$ 的随机变量取值在区间 $[-3\sigma, 3\sigma]$ 的概率约为 99.73%。

$$\int_0^{\frac{\pi}{2}} \sin x dx = -\cos x \Big|_0^{\frac{\pi}{2}} = 1 \quad (3)$$

$$\int_0^1 \sqrt{x} dx = \frac{2}{3} x^{\frac{3}{2}} \Big|_0^1 = \frac{2}{3} \approx 0.666667 \quad (4)$$

$$\int_0^1 e^x dx = e^x \Big|_0^1 = e - 1 \approx 1.71828 \quad (5)$$

6.9.4 矩阵乘积

存放矩阵元素常用二维数组，也可以用一维数组。本小节介绍这两种不同的方式将存放矩阵元素的数组首地址传递给函数。

在第 5.8.2 小节中讨论过指向一维数组的指针变量——数组指针。这种指针变量能指向一类数组（其目标数组应具有相同的数据类型、相同的元素个数）。当然，这种指针在计算机内存中仅占用 `sizeof(void*)` 字节。C++ 程序中的二维数组是一系列同类一维数组的集合。可定义一个数组指针变量间接地访问二维数组的所有元素。若将函数的形式参数设计成数组指针，则可将二维数组地址传递给函数。这种传递方法的好处是可使用双方括号（即双下标）访问二维数组元素。

用二维数组刻画矩阵，借用矩阵的“行”与“列”的概念，称二维数组在内存中按行存放。各行的首地址便是那些一维数组的首地址，一维数组的元素对应为“列”。那么矩阵同一列上的元素在内存中不是连续存放的，它们之间相隔的元素个数恰好为一维数组的元素个数。

例 6.15 计算两个矩阵之积。

源代码 6.12 矩阵乘积

```

1 // MultiMatrix.cpp
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 int MultiMatrix(double (*pa)[4], int arow,// 需要传递行数
7                  double (*pb)[5], int brow,
8                  double (*pc)[5], int crow)
9 {

```

```

10     int acol, bcol, ccol;           // 用于存放矩阵的列数
11     acol = pa[1] - pa[0];          // 两个地址相减
12     bcol = pb[1] - pb[0];          // 参见第 5.3.2 小节
13     ccol = pc[1] - pc[0];          // 计算矩阵的列数
14     if(acol!=brow || arow!=crow || bcol!=ccol)
15         return -1;                // 无法计算, 返回
16     for(int i=0; i<crow; i++)
17     {
18         for(int j=0; j<ccol; j++)
19         {
20             pc[i][j] = 0;
21             for(int k=0; k<acol; k++)
22                 pc[i][j] += pa[i][k]*pb[k][j];
23         }
24     }
25     return 0;
26 }
27
28 template <typename T>
29 void Display(const T *a, int row, int col)           // 形参 a 为一级指针
30 {
31     for(int i=0; i<row; i++)
32     {
33         for(int j=0; j<col; j++)
34             cout << setw(5) << a[i*col+j]; // 注意下标的表达式
35         cout << endl;                  // 输出一行后换行
36     }
37     cout << endl;
38 }
39
40 int main()
41 {
42     int flag;
43     double a[3][4] = {{5,7,8,2}, {-2,4,1,1}, {1, 2, 3, 4}};
44     double b[4][5] = { {4, -2, 3, 3, 9}, {4, 3, 8, -1, 2},
45                       {2, 3, 5, 2, 7}, {1, 0, 6, 3, 4} };
46     double c[3][5];
47     Display(a[0], 3, 4);               // 第一个参数为一级地址
48     Display(b[0], 4, 5);
49     flag = MultiMatrix(a, 3, b, 4, c, 3); // 参数a,b,c为二级地址
50     if(flag==0)
51         Display(c[0], 3, 5);
52     else
53         cout << "矩阵的阶数不正确, 无法计算。" << endl;
54     return 0;
55 }

```

程序的运行结果:

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

66	35	123	30	123
11	19	37	-5	1
22	13	58	19	50

程序中，函数 MultiMatrix 的第一个形式参数为一个指向数组的指针，这种指针将其所指之处看成一系列具有 4 个元素的一维 double 型数组的集合。因此，不必传递矩阵的列数（它可在函数体内算出，参见第 11~13 行），但矩阵的行数必须传递给函数。在函数体内通过指向数组的指针直接访问二维数组元素（参见第 20、22 行）。

函数模板 Display 则采用传递一级指针的方法处理二维数组。程序第 34 行则通过表达式计算出矩阵对应元素的下标。这是因为二维数组仍然在内存中连续存放。此时，矩阵的行数和列数均需要传递给该函数。并且需要用一级指针（一级地址）`a[0]` 或 `b[0]` 或 `c[0]` 为实参初始化形式参数一级指针（参见第 47、48、51 行）。

对于这个函数模板，还可以写成如下形式。需要注意的是形式参数指针变量 `a` 不断地移动，当两重循环结束时，该指针指向二维数组最后一个元素之后的地址，函数返回时指针 `a` 被销毁。

```

1 template <typename T>
2 void Display(const T *a, int row, int col)
3 {
4     for(int i=0; i<row; i++) // 形式参数 a 为一级指针
5     {
6         for(int j=0; j<col; j++)
7             cout << setw(5) << *a++;
8         cout << endl;
9     }
10    cout << endl;
11 }
```

6.9.5 动态二维数组

在矩阵乘积问题的计算中，我们采用传递数组指针（该函数只能计算指定列数的矩阵乘积，灵活性欠佳）、传递一级指针（下标计算复杂）的方法处理了二维数组。本小节介绍如何在堆空间中建立和使用二维数组的实现方案。采用函数模板编写，以适应所有的数据类型。这个例子有助于读者深入理解二级指针和二维数组。

C++ 的 `new` 操作运算成功的结果是返回堆中的某地址值，一般为一个一级地址，对应一维数组。若另外建立一个地址的索引——指针数组，用于存放一系列偏移地址，则通过两次寻址便能像访问二维数组元素一样用双下标访问所申请的堆空间中的元素。

例 6.16 动态二维数组。

源代码 6.13 动态（分配、释放）二维数组模板

```

1 template <typename T>
2 T ***New2DArray(T ***array, int row, int col)
3 {
4     array = new T*[row];           // 存放一系列行首址
5     if(array==NULL) return array;
6     array[0] = new T[row*col];      // 分配一片连续的堆内存单元
7     if(array[0]==NULL)
```

```

8   {
9     delete [] array;
10    return array=NULL;
11  }
12  for(int i=1; i<row; i++)
13    array[i] = array[0] + i*col; // 计算并保存各“行”首地址
14  return array;
15 }
16
17 template <typename T> void Delete2DArray(T **&array)
18 {
19   if(array) // 释放动态二维数组
20   {
21     if(array[0]) delete [] array[0];
22     delete [] array;
23     array = NULL;
24   }
25 }
```

上述函数模板的形式参数传递二级指针变量（引用型参数，直接双向传递），其中第一个函数模板还返回所传递的二级指针变量（引用返回^[1]）。函数模板的功能是定义（创建）一个二维动态数组和释放一个二维动态数组。测试程序如下。

```

1 // DynamicArray.cpp
2 #include <iostream>
3 #include <iomanip>
4 #include <ctime>
5 using namespace std;
6
7 // 请插入函数模板
8
9 template <typename T> void Display(T **a, int row) // 只需要传递行数
10 {
11   int col = a[1]-a[0]; // 列数可通过计算得到
12   for(int i=0; i<row; i++)
13   {
14     for(int j=0; j<col; j++)
15       cout << a[i][j] << ' ';
16     cout << endl;
17   }
18   cout << endl;
19 }
20
21 int main()
22 {
23   int **ip; // 定义一个二级指针，仅占 sizeof(void*) 字节空间
24   double **dp; // 定义一个二级指针，仅占 sizeof(void*) 字节空间
25   int m, n, i, j;
26   double iAver = 0, dAver = 0;
27
28   cout << "请输入二维数组的行数\列数\m\n:" ;
29   cin >> m >> n;
30
31   New2DArray(ip, m, n); // 动态定义二维数组，其元素个数可为变量
32   if(ip==NULL)
33   {
```

^[1]指针引用返回的作用不大。因为并不需要将该函数调用做左值；将其目标作为左值时采用返回地址值即可。

```

34         cout << "内存分配不成功。" << endl;
35         exit(-1); // 直接结束程序返回到操作系统
36     }
37
38     if(New2DArray(dp, m, n)==NULL) // 应用于不同的数据类型
39     {
40         cout << "内存分配不成功。" << endl;
41         Delete2DArray(ip); // 释放已经分配的动态二维数组
42         exit(-1);
43     }
44     time_t t;
45     srand(time(&t));
46
47     for(i=0; i<m; i++)
48         for(j=0; j<n; j++)
49             dp[i][j] = (ip[i][j]=rand())/(double)RAND_MAX;
50
51     Display(ip, m);
52     Display(dp, m);
53
54     for(i=0; i<m; i++)
55     {
56         for(j=0; j<n; j++)
57         {
58             iAver += ip[i][j];
59             dAver += dp[i][j];
60         }
61     }
62     iAver /= m*n;
63     dAver /= m*n;
64     cout << iAver << "\t\t" << dAver << endl;
65
66     Delete2DArray(ip); // 释放动态二维数组
67     Delete2DArray(dp); // 释放动态二维数组
68     return 0;
69 }

```

从上述程序可以看出，这样的动态数组建立在堆空间中。创建二维堆数组时，元素个数都不必是常量；访问数组元素时比采用 $a[i*col+j]$ 的下标计算“升”维的处理方法简单。这种动态数组建立在堆空间中，系统不会自动释放它。当程序中不再需要这种动态二维数组时应该调用 `Delete2DArray` 函数将其释放。

6.10 趣味程序——高尔顿钉板实验模拟

例 6.17 高尔顿钉板实验模拟。

高尔顿（Galton）钉板是统计学中的一个基本实验。在有若干排钉子的木板上方有一个漏斗形的入口。钉子的排列如图 6-4 所示。将一系列小球从入口处使其自由下落，小球在下落的过程中碰到板上的钉子后，以等可能的概率从钉子的左侧或者右侧继续下落，最终落至底部的槽中。统计各槽中的小球数量，可得到一个分布表。

下面的程序利用随机数发生器模拟了高尔顿钉板实验。程序中模拟了 $m = 10\,000$ 个小球落入 $n = 20$ 个槽中的情形。

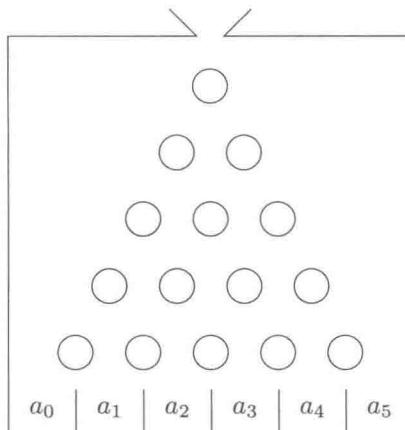


图 6-4 高尔顿钉板

源代码 6.14 高尔顿钉板实验模拟

```

1 // Galton_board.cpp
2 #include <iostream>
3 #include <ctime>
4 using namespace std;
5
6 void Galton_board(int m, int *a, int n)
7 {                                         // 总球数, 各槽中的球数, 槽数
8     int i, j;
9     double x;
10
11    for(j=0; j<n; j++)
12        a[j] = 0;
13    for(i=0; i<m; i++)
14    {
15        x = (n-1)/2.0;                  // 第 i 个小球的初始位置
16        for(j=1; j<n; j++)
17            x += (rand()%2) ? 0.5 : -0.5;
18        a[(int)x]++;                  // 落入第 x 个槽
19    }
20 }
21 void display(const int *a, int n)
22 {
23     cout << a[0];
24     for(int i=1; i<n; i++)
25         cout << "," << a[i];
26     cout << endl;
27 }
28 int main()
29 {
30     const int N = 20;
31     int m=10000, a[N];
32     time_t t;
33     srand(time(&t));
34     Galton_board(m, a, N);
35     display(a, N);
36     return 0;
37 }
```

某次运行程序的结果：

```
0, 0, 1, 16, 71, 218, 542, 939, 1444, 1760, 1775, 1413, 962, 546, 222, 76, 15, 0, 0, 0
```

6.11 小 结

函数是 C++ 的重要概念之一。理解函数调用的栈操作机制及操作过程可以：

- ① 彻底理解数据或变量在函数之间传递、返回的本质。
- ② 彻底理解形式参数、自动变量的生命周期。
- ③ 有助于理解全局变量、静态全局变量、静态局部变量和堆变量的生命周期。

因此，理解函数调用的机制也有助于程序员设计变量、设计函数。

函数的其他特性：重载、内联、参数带默认值和模板等在不增加程序员负担的同时带给程序员极大的方便。我们应乐意接受这些特性，并加以充分利用。

下面归纳在函数设计时的一些基本原则，供读者参考：

- (1) 函数的功能应小而单一，尽量将计算与 I/O 分离。
- (2) 函数命名应尽量表明函数的功能。
- (3) 不要使用全局变量。应让数据皆通过参数传递，考虑数据传递的方向是单向还是需要双向的。
- (4) 形式参数的个数应尽可能地少（参数之间在算法上相互独立、无冗余，中间结果和辅助量不要作为参数），尝试使用参数带默认值。
- (5) 加强参数的安全保护措施（如采用 `const` 限制）。
- (6) 充分利用函数的返回类型，即使仅返回一个函数执行状态值。
- (7) 考虑函数调用时的效率（尽可能传递引用、引用返回）。
- (8) 尽量编写函数模板，使操作能穷尽所有可能的数据类型。

本章所介绍的函数应用例子程序都具有典型性，值得精读。

练习 6

1. 指出下列各程序或函数中的错误

(1)

```

1 #include <iostream>
2 using namespace std;
3 void display(const double *x, int m, int n) // 不修改函数首部
4 {
5     int i, j;
6     for(i=0; i<m; i++)
7     {
8         for(j=0; j<n; j++)
9             cout << x[i][j] << ' ';
10        cout << endl;
11    }
12 }
```

(2)

```

1 #include <iostream>
2 using namespace std;
3
4 void GetData(double *x, int m, int n)
5 {
6     int i, j;
7     for(i=0; i<m; i++)
8         for(j=0; j<n; j++)
9             x[i*n+j] = rand()/(double)RAND_MAX;
10 }
11
12 int main()
13 {
14     double a[4][5];
15     GetData(a, 4, 5);
16     return 0;
17 }
```

(3)

```

1 #include <iostream>
2 using namespace std;
3
4 void f(int a){ cout << a << endl; }
5 int f(int x){ return x*x; }
6
7 int main()
8 {
9     int m=3, x=5, n;
10    f(m);
11    n = f(x);
12    return 0;
13 }
```

(4)

```

1 #include <iostream>
2 using namespace std;
3
4 void f(int a){ cout << a << endl; }
5 int f(int &x){ return x*x; }
6
7 int main()
8 {
9     int m=3, x=5, n;
10    f(m);
11    n = f(x);
12    return 0;
13 }
```

(5)

```

1 #include <iostream>
2 using namespace std;
3 int f(int n)
4 {
```

```

5     return f(n-1)+f(n-2);
6     if(n==1 || n==2) return 1;
7 }
8
9 int main()
10 {
11     cout << f(5) << endl;
12     return 0;
13 }
```

(6)

```

1 #include <iostream>
2 using namespace std;
3
4 int f(int n)
5 {
6     return f(n-1) + n;
7     if(n==1) return 1;
8 }
9
10 int main()
11 {
12     cout << f(5000) << endl;
13     return 0;
14 }
```

(7)

```

1 #include <iostream>
2 using namespace std;
3
4 void f(int n=10);
5 void f(int n=10){ cout << n << endl; }
6
7 int main()
8 {
9     f();
10    f(100);
11    return 0;
12 }
```

(8)

```

1 #include <iostream>
2 using namespace std;
3
4 void f(int a=3, int b)
5 {
6     cout << a << '\t' << b << endl;
7 }
```

(9)

```

1 #include <iostream>
2 using namespace std;
3
```

```

4 void f(int a, int b=a)
5 {
6     cout << a << '\t' << b << endl;
7 }

```

(10)

```

1 template <typename T> void swap(T &a, T &b)
2 {
3     int temp;
4     temp = a; a = b; b = temp;
5 }

```

2. 基本题

(1) 编写计算两点间距离的函数 `double distance(double x1, double y1, double x2, double y2);`

(2) 编写一个函数，其功能是根据表示年月日的三个整型变量 `year, month` 和 `day` 计算并“返回”下一天的日期。函数名定为 `NextDay`，形式参数及返回类型请自行设计。

(3) 产生一个 5×5 的矩阵（矩阵元素取 $0 \sim 99$ 之间的随机整数），输出该矩阵并将该矩阵转置后再输出。

(4) 产生一个 5×5 的矩阵（矩阵元素取 $0 \sim 99$ 之间的随机整数），求该矩阵的鞍点。所谓矩阵的鞍点是矩阵的（行，列）位置，该位置上的元素是其所在行的最大值并且是其所在列的最小值，或者该位置上的元素是其所在行的最小值并且是其所在列的最大值。若该矩阵无鞍点，则输出“该矩阵无鞍点”。

(5) 根据递推公式编写一个递归函数计算 Fibonacci 数列的第 n 项：

$$\begin{cases} f(1) = f(2) = 1 \\ f(n) = f(n-1) + f(n-2), \quad n = 3, 4, 5, \dots \end{cases}$$

(6) 阅读下列程序，指出函数模板 `template <typename T> void ShowBits(T x);` 的功能。请在计算机上运行该程序，观察其结果（比较第 3.6.2 小节的“整数的二进制各位”程序）。

```

1 #include <iostream>
2 using namespace std;
3
4 template <typename T> void ShowBits(T x)
5 {
6     unsigned char *p;
7     int n, i, j;
8     n = sizeof(T);
9     p = (unsigned char*)(&x);
10    for(i=0; i<n; i++)
11    {
12        for(j=7; j>=0; j--)
13            cout << (*p >> j & 1);
14            cout << ' ';
15            p++;
16    }
17    cout << endl;

```

```

18 }
19
20 int main()
21 {
22     ShowBits(char(-1));
23     ShowBits(short(-1));
24     ShowBits(long(100));
25     ShowBits(char(100));
26     ShowBits(short(100));
27     ShowBits(long(100));
28     ShowBits(3.1415926f);
29     ShowBits(3.1415926);
30     return 0;
31 }
```

(7) 编写函数模板 `template <typename T> void ShowTypeSize(const char* type, T x);` 将根据实际参数的数据类型输出其所占用的字节数。主函数如下：

```

1 #include <iostream>
2 using namespace std;
3
4 // 添加函数模板
5 int main()
6 {
7     ShowTypeSize("int", int(1));
8     ShowTypeSize("double", double(1));
9     // 补充对其他数据类型的处理
10    return 0;
11 }
```

并希望按如下形式输出。

```

sizeof(int): 4 byte(s)
sizeof(double): 8 byte(s)
...
```

(8) 阅读程序，写出运行结果。

```

1 #include <iostream>
2 using namespace std;
3
4 void ShowInt(unsigned int n, int base=10)
5 {
6     static char BASE[17] = "0123456789ABCDEF";
7     if(n!=0) ShowInt(n/base, base);
8     else      return;
9     cout << BASE[n%base];
10 }
11 int main()
12 {
13     ShowInt(127);      cout << endl;
14     ShowInt(127, 2);   cout << endl;
15     ShowInt(127, 8);   cout << endl;
16     ShowInt(127, 16);  cout << endl;
17     return 0;
18 }
```

(9) 改编“摇奖”程序（源代码6.7）。要求在显示器上随机显示预设的若干城市名称，按键后确定所选择的城市名。

(10) 编写程序，使用标准函数 `qsort`，对各类数组进行排序。

① 对整数数组进行排序。比较是以一个整数的各位数字之和的大小为依据，从小到大排列。数组中的元素为：12, 32, 42, 51, 8, 16, 51, 21, 19, 9。

② 对浮点型数组进行排序。从小到大排列。数组中的元素为：32.1, 456.87, 332.67, 442.0, 98.12, 451.79, 340.12, 54.55, 99.87, 72.5。

③ 对字符串数组进行排序。比较是以各字符串的长度为依据，如果长度相等，再比较字符串的内容，从小到大排列。数组中的元素为：“enter”, “number”, “size”, “begin”, “of”, “cat”, “case”, “program”, “centain”, “a”。

3. 扩展题（程序设计竞赛题型）

(1) 计算逆序对的数目。

问题描述 给定 $1 \sim n$ 这 n 个自然数的一个排列： $a_0a_1\cdots a_{n-1}$ ，若 $i < j$ 且 $a_i > a_j$ ，则称 $\langle i, j \rangle$ 为该排列的一个“逆序对”。对于给定排列，计算其总逆序对的个数，要求在尽可能短的时间内完成计算（注意：穷举法不符合要求，因为穷举法所需的计算时间较长。请参考快速排序的设计思想设计算法）。

输入说明 输入数据有多行，每一行为一种情形有若干个自然数形成一个排列。

输出说明 对于每一种情形输出“Case 序号：”，空格，结果。

输入样例

```
3 1 4 5 2
1 2 3 4 5 6
7 6 5 4 3 2 1
```

输出样例

```
Case 1: 4
Case 2: 0
Case 3: 21
```

(2) 高精度计算。

问题描述 对于给定的两个不超过 $N - 1$ 位的十进制整数（例如 $N = 51$ ），要求计算它们的和与差。计算过程中，若有数据溢出（数据超过 $N - 1$ 位）时，直接丢掉溢出的部分。

输入说明 输入数据有两行，每一行为一个不超过 50 位的十进制整数。

输出说明 输出的第一行为输入数据的和，第二行为输入数据的差。

输入样例

```
112233445566778899001234567890
-987654321097531864209876543210
```

输出样例

```
-875420875530752965208641975320
109988776666431076321111111100
```

【参考提示】定义常量 `const int N=51;` 用元素个数为 N 的字符数组倒序存放十进制整数。不妨称这种数据为 `VeryLong`。约定：

① 数组首元素为 0 表示正数，为 1 表示负数。

② 下标为 1 的元素存放十进制数的个位，下标为 2 的元素存放十进制数的十位，依此类推，下标为 $N - 1$ 的元素存放十进制数的第 $N - 1$ 位。

因此，这种数组能够存放多达 50 位的十进制整数。根据上述存储方式，要求对这种数据进行加工。下面给出从字符串到这种数据的转换函数、输出函数。请读者完成这种数据的加、减等功能的其他函数。

源代码 6.15 高精度计算程序片段

```

1 #include <iostream>
2 using namespace std;
3
4 const int N = 51;
5
6 void output(const char *d)           // 输出函数，根据约定①和②
7 {
8     int i;
9     for(i=N-1; i>=1 && d[i]==0; i--) // 扫描非零的最高位
10    ;
11    if(i==0)                      // 若全为 0
12    {
13        cout << 0;      return;
14    }
15    if(d[0]) cout << "-";          // 若为负数
16    for(; i>0; i--)
17        cout << (int)d[i];
18 }
19
20 void outputline(const char *d)       // 输出并换行
21 {
22     output(d);
23     cout << endl;
24 }
25
26 char *atovl(char *d, const char *str) // 转换函数
27 {                                     // 字符串表示的整数串转换到约定的 VeryLong 格式
28     int i, j, k;
29     d[0] = 0;                         // 默认为正数
30     for(i=0; str[i]==' ' || str[i]=='\t' || str[i]=='0'; i++) // 跳过前导空格、前导制表符、前导 0
31     ;
32     if(str[i]=='-')                  // 出现负号时
33     {
34         i++;
35         d[0] = 1;                   // 按约定 ①，表示负数
36     }
37     else if(str[i]=='+') // 出现正号时
38         i++;                     // 处理符号（若有正号或者负号时）
39     for(; str[i]==' ' || str[i]=='\t' || str[i]=='0'; i++)

```

```
40 ; // 跳过符号后的空格、制表符及 0
41 // 此时, i 为最高位
42 for(j=i; str[j]; j++)
43 ; // 找到串 str 的结束标志处
44 // 此时, j-1 为最低位。因此, 下面的语句中, 首先 j--
45 for(k=1, j--; k<N && j>=i; j--)
46 {
47     if(str[j]>='0' && str[j]<='9') // 过滤掉不合法的字符
48         d[k++] = str[j] - '0'; // 将合法的数码字符转换成数值
49 }
50 for(; k<N; k++) // 剩余的高位全置 0
51     d[k] = 0;
52 return d;
53 }
54
55 int main()
56 {
57     char x[N], y[N], p[N], q[N]; // 分别存放原始数据和计算结果
58     char str[2*N]; // 用于读取数据, 字符串
59
60     cin.getline(str, 2*N); // 读取第一个字符串
61     ctovl(x, str); // 转换
62     cin.getline(str, 2*N); // 读取第二个字符串
63     ctovl(y, str); // 转换
64
65     add(x, y, p); // 待读者完成该函数的定义
66     sub(x, y, q); // 待读者完成该函数的定义
67     outputline(p); // 输出计算结果
68     outputline(q);
69     return 0;
70 }
```

第7章 程序结构

C++以源程序文件（.cpp）为编译单元进行分割编译，支持众人集体开发大型软件。一个C++程序往往由一系列文件组成，每个编译单元中均可能有一系列函数、一系列变量定义。不同开发人员自行命名的变量、函数等标识符可能相同，有些变量、函数和模板需要共享。C++语言提供了良好的保障机制既能避免冲突发生，又能实现变量、函数等的共享。学习本章后，读者应该掌握多文件结构，并主动地加以实践。

7.1 多文件结构

7.1.1 同一编译单元中的共享变量及函数

静态全局变量的生命期是全局的：从程序启动时（主函数执行之前）产生，到程序结束时（主函数返回后）销毁。静态全局变量存放在全局数据区中（不受函数调用压栈、函数返回退栈影响）。静态全局变量的作用域为其所在的编译单元中，该变量定义语句之后（其后定义的函数体内均能直接访问该变量，在该变量定义语句之前的函数体内不可访问该静态全局变量）。

在不同编译单元中定义的静态全局变量（即使变量名称等均相同）是完全不同的变量，它们之间不存在任何关系，故不会引起任何冲突（参见第 3.1.2 小节例 3.1）。

声明或定义函数时也可以将函数的存储类型设置为静态（`static`）的，这种函数被称为静态函数。静态函数的作用域将被限制在其所在的编译单元中。不同编译单元定义的静态函数也不会引起冲突。C++ 不允许在一个函数体内定义另一个函数。

设置静态全局变量、静态函数的作用是将它们规定为本编译单元“私有”，为本编译单元“共享”。

7.1.2 不同编译单元中的共享变量及函数

C++程序的不同编译单元之间总是需要相互联系的。我们在前几章编写的所有函数都是全局的，当需要使用其他编译单元中定义的函数时，只需要在本编译单元中用其函数原型声明，告诉编译器如何检查函数调用语句的语法正确性，由连接器寻找该函数的代码。同样，当需要使用其他编译单元中定义的全局变量时，也只需要在本编译单元中用保留字 `extern` 对其进行外部参照访问声明（参见第 3.1.2 小节规定 3.3），告诉编译器这种全局变量的数据类型，由连接器寻找其定义。

全局变量的生命期是全局的：从程序启动时（主函数执行之前）产生，到程序结束时（主函数返回后）销毁。全局变量存放在全局数据区中（不受函数调用压栈、函数返回退栈影响）。全局变量的作用域为其所在的编译单元中定义或外部参照访问声明语句之后。

值得注意的是：全局变量不能只有外部参照访问声明而从未定义，也不能在多个编译单元中重复定义。倘若一个程序有 30 个编译单元，其中有 10 个编译单元需要共享某一个全局变量，则应该在这 10 个编译单元中的某一个编译单元中定义它（意味着为它分配内存空间），而在其他的需要访问该全局变量的 9 个编译单元中（可包括定义它的编译单元，即 10 个编译单元）对该全局变量进行外部参照访问声明。即一处定义，多处声明。在剩下的 20 个编译单元中不允许用该全局变量名定义另一个全局变量。这样，在定义或声明过该全局变量的编译单元中便可以共享它。既未定义也未声明的编译单元则不能访问该全局变量。

7.1.3 头文件

头文件不是编译单元，编译器不会单独地编译它。C++ 程序中，常常用编译预处理指令 `#include` 将指定的头文件的内容插入到该包含指令所在的位置。即，头文件中所书写的内容可能会随包含指令插入到多个编译单元中进行编译。因此，头文件中应该只写声明的内容，不要写定义的部分。

应该写入头文件的成分：

- (1) 全局变量的外部参照访问声明。
- (2) 函数原型（函数声明）。
- (3) 数据类型的描述。
- (4) 内联函数。
- (5) 模板描述。

由于变量或函数不能重复定义，故如下定义不宜写入头文件中。

- (1) 全局变量的定义。
- (2) 函数的定义（即函数实现）。

若将函数模板写入源程序文件，则该模板仅能在该编译单元中使用，不能跨编译单元共享。

例 7.1 求解一元二次方程的多文件结构程序。将例 6.1 改编成多文件结构的程序。整个程序由表 7-1 所示的三个文件组成。

表 7-1 求解一元二次方程程序的文件组成

文 件 名	说 明
<code>solver.h</code>	头文件，编写函数声明
<code>solver.cpp</code>	源程序文件，求解方程的若干函数定义
<code>solverTesting.cpp</code>	源程序文件，主函数，测试函数定义

显而易见，`solver.h`, `solver.cpp` 是一组文件，它们构成相对独立的程序块，可移植到其他程序中。

在 C++ 应用程序集成开发环境中，创建一个新的工程文件；新建两个源程序文件和一个头文件并添加到该工程文件中；输入下面所列文件的内容；最后编译、连接、运行。

源代码 7.1 头文件 solver.h

```

1 // solver.h
2 #ifndef SOLVER_H
3 #define SOLVER_H
4
5 int Solver(double a,double b,double c,double &x1,double &x2);
6 void ShowEquation(double a, double b, double c);
7 void ShowSolution(int flag, double x1, double x2);
8
9 #endif

```

源代码 7.2 源程序文件 solver.cpp

```

1 // solver.cpp          求解一元二次方程
2 #include <iostream>
3 #include <cmath>
4 using namespace std;
5 static double Delta(double a, double b, double c)
6 {                  // 静态函数, 仅限于本编译单元中使用
7     return b*b - 4*a*c;
8 }
9 int Solver(double a,double b,double c,double &x1,double &x2)
10 {
11     int flag = 0;
12     double d = Delta(a, b, c);
13     if(d >= 0)
14     {
15         d = sqrt(d);
16         x1 = (-b - d)/(2*a);
17         x2 = (-b + d)/(2*a);
18         flag = (d>0) ? 2 : 1;
19     }
20     return flag;
21 }
22 void ShowEquation(double a, double b, double c)
23 {
24     cout << "方程: " << a << "x^2";
25     if(b>0)
26         cout << "+";
27     else if(b<0)
28         cout << "-";
29     if(c>0)
30         cout << "+";
31     else if(c<0)
32         cout << "-";
33     cout << "x=";
34 }
35 void ShowSolution(int flag, double x1, double x2)
36 {
37     switch(flag)
38     {
39     case 0: cout << "无实数根。" << endl; break;
40     case 1: cout << "有重根: x1=x2=" << x1 << endl; break;
41     case 2: cout << "有两个根: x1=" << x1
42                 << ",x2=" << x2 << endl;
43     }
44 }

```

源代码 7.3 源程序文件 solverTesting.cpp

```

1 // solverTesting.cpp
2 #include "solver.h"           // 本编译单元不必包含其他头文件
3
4 int main()
5 {
6     double a=1, b=2, c=-3, x1, x2;
7     int flag;
8     flag = Solver(a, b, c, x1, x2);
9     ShowEquation(a, b, c);
10    ShowSolution(flag, x1, x2);
11    return 0;
12 }
```

7.2 编译预处理指令

顾名思义，编译预处理是指编译单元在被编译之前预先进行的处理。常见的编译预处理指令有文件包含指令、宏定义指令和条件编译指令。

7.2.1 文件包含指令

文件包含指令有两种格式：

```
#include <文件名>
#include "文件名"
```

第一种格式表示将在系统约定的子目录中查找指定的文件，若找到则相当于执行将该文件的内容“全选”“复制”并“粘贴”到包含指令处（简称包含该文件）。若找不到，则停止处理并给出出错信息。第二种格式将首先在当前目录中查找指定的文件，若找到则包含该文件；若在当前目录中找不到指定的文件，则再到系统约定的子目录中查找，若找到则包含之，否则才停止处理并给出出错信息。因此，建议读者将系统提供的头文件用第一种格式包含；自己编写的头文件（存放在当前工程文件的工作目录中）用第二种格式。

7.2.2 宏定义指令

C 语言中没有 `const` 保留字，除了字面常量外，符号常量采用宏定义。如：

```
#define M_PI 3.1415926
```

C 语言中还常用带参数的宏描述一些简单的计算表达式，如：

```
#define ABS(x) ((x)>0?(x):-(x))      // 不管 x 的数据类型
```

其中 `x` 被称为参数，三目条件运算表达式为计算参数的绝对值。这里的 `ABS(x)` 既不是函数定义，又不是语句（此处无分号）。程序中的语句 `y=ABS(-2);` 也不是函数调用。这个语句将在编译前被处理（编译预处理）成如下语句。

```
y = ((-2)>0?(-2):-(-2));
```

即编译系统在编译前首先扫描整个源程序，遇到使用宏的地方，如 MPI, ABS(表达式)，则将其进行“智能查找并替换”，处理后再进行编译。当然，编译系统并不修改用户的源程序文件，它将预处理的结果保存为一个临时文件，并对该临时文件进行编译。因此在编译的目标代码文件中，已经不可能有类似 MPI, ABS 的身影。ABS(x) 不是函数调用，而是在编译前就“就地展开”了。

C++ 中有 const 定义符号常量，有 inline 内联函数实现“就地展开”，还有函数模板能适应多种不同的数据类型。这些新特性均超过 C 语言中的宏定义指令。因此，在 C++ 程序中宏定义指令最多出现在头文件中用于“挡驾”重复包含头文件（如头文件 solver.h 中第 3 行），或者用于下面介绍的条件编译。

7.2.3 条件编译指令

条件编译指令有 #if, #else、#elif、#endif、#ifdef 和 #ifndef。在程序中使用条件编译指令可以协调头文件之间的相互包含。例 7.1 求解一元二次方程的头文件中，使用了条件编译指令 #ifndef SOLVER_H 判断是否没有定义标识符 SOLVER_H。显然，作为分割编译的一个编译单元起初是没有定义该标识符的，此时，便将其后直到 #endif 之间的语句（第 3~8 行）插入到该编译单元中。需要注意的是，其中的指令 #define SOLVER_H (第 3 行) 便使得标识符 SOLVER_H 被定义了。因此，倘若再包含该头文件，便会被条件编译指令（第 2, 9 行）“挡驾”以避免重复包含。

例 7.2 利用条件编译指令处理程序的兼容性问题。

问题描述 本教材的一些程序中使用了一个不属于标准 C++ 的头文件 conio.h 和其中的三个函数 getch、getche 和 kbhit。Visual C++ 6.0、MinGW C++ 和 Dev-C++ 均提供了这样的头文件和函数（为叙述方便，将其称为第一种编译系统）。假设有某编译系统提供了一个功能类似的头文件 otherHead.h 其中声明了一个功能相同或类似的函数，原型为 int getkey();（注意假想的头文件名、函数名皆不同，我们将其称为第二种编译系统）。或者某编译系统只能用标准 C++ 函数（我们将其称为第三种编译系统）。如何使相关程序能方便地在这三种编译系统中编译？

我们不能“因噎废食”而放弃使用 getch 函数，也不愿意重复地为每个编译系统重新写一遍程序。利用条件编译便能很好地解决这个问题。具体做法是将其他编译系统中功能相近的函数“包装”成函数 int getch(); 的形式。

源代码 7.4 条件编译示例程序

```

1 // myconio.h
2 #define COMP 3           // 将标识符定义成不同的值对应不同的编译器
3
4 #if COMP==1
5     #include <conio.h>
6 #else
7     int getch();        // 函数声明
8 #endif

```

```

1 // myconio.cpp
2 #include "myconio.h"
3
4 #if COMP==1          // 适用于 VC++、MinGW C++ 和 Dev-C++
5   #include <conio.h>
6 #elif COMP==2        // 适用于某种假想的 C++ 编译器
7   #include <otherHead.h>
8   int getch(){ return getkey(); }
9 #else                // 适用于标准 C++
10  #include <cstdio>
11  int getch(){ return getchar(); }
12 #endif

```

```

1 // test.cpp
2 #include <iostream>
3 #include "myconio.h"
4 using namespace std;
5
6 int main()
7 {
8     int i, key;
9
10    for(i=0; i<5; i++)
11    {
12        cout << "Press any key..." << flush;
13        key = getch();
14        cout << i+1 << '\t' << key << endl;
15    }
16    return 0;
17 }

```

程序中的一些语句在某一种编译器下是正确的，在另一种编译器下却可能是错误的。显然这些语句不能全部被编译，必须“删除”其中的一部分。

(1) 若将标识符 COMP 定义成 3，则只需要标准的 C++ 编译器。实际调用的是标准 C++ 函数 `getchar`，执行该函数时需要按回车键。其他函数未被编译。

(2) 若将标识符 COMP 定义成 1，则直接使用非标准的函数 `getch`。

(3) 若将标识符 COMP 定义成 2，则出现编译错误！因为这种情形是我们假设的，根本没有这样的头文件，也没有函数 `getkey`。

条件编译预处理指令还常用于程序的调试，例如：

```

#define DEBUG // 调试完成后，仅去掉此行即可

// 程序语句

#ifndef DEBUG
// 编写调试用的语句

#endif

// 程序语句

```

条件编译指令与条件分支选择语句是不同的。条件分支选择语句的每一个分支都是要被编译的（只是可能没有机会执行其某一分支），而条件编译指令是一种预处理，一些语句组可能不参加编译（相当于删除了这些不符合条件的语句），更谈不上被执行。条件编译预处理指令中条件的不同实为将源程序处理成几个不同的程序。

7.3 名字空间

在众人集体开发大型软件时，每个程序员可能编写有多个编译单元。现假设程序员 A 编写了五个编译单元都需要使用一个表示最大客户数的全局变量，程序员 A 将该变量命名为 `MaxNumber`；另有程序员 B 编写了三个编译单元都需要使用一个表示某物品最大件数的变量，程序员 B 将该变量命名为 `MaxNumber`。程序员 A、程序员 B 所编写的共八个编译单元都能通过分割编译。但是在连接时将会导致该变量被定义两次的错误。不仅如此，这两个变量的含义和作用不相同，将会导致程序出现混乱。

名字空间是 C++ 引入的可以由程序员命名的作用域，用来避免程序中标识符名字冲突的问题。名字空间可跨越编译单元。名字空间中声明的标识符具有一定的封装屏蔽特性，不同名字空间中的同名变量不会发生冲突。

名字空间中定义的非静态变量是一种受名字空间限制的全局变量，可以跨越不同的编译单元。这种“受限制的全局变量”与全局变量一样，只能且必须在某编译单元中被定义一次，其他编译单元中用 `extern` 进行参照访问声明。

下面，举例说明名字空间的用法。程序中三组文件声明了两个名字空间，两个名字空间中有些标识符完全相同，但是它们的功能不尽相同（为了区别起见，有的函数故意不按常理设计，敬请读者留意）。另外设计一个测试用的主函数。共 7 个文件如表 7-2 所示。

表 7-2 名字空间示例程序的文件组成

文件名	说明
<code>myMath.h</code>	头文件，将一个数学函数原型，一个函数模板声明，一个常量定义放入 <code>mySpace</code> 空间中
<code>myMath.cpp</code>	定义 <code>mySpace::Sqrt</code> 函数
<code>myString.h</code>	头文件，将两个字符串处理函数放入 <code>mySpace</code> 空间中
<code>myString.cpp</code>	源程序文件，定义函数
<code>OtherSpace.h</code>	头文件，将一些函数声明等放入 <code>OtherSpace</code> 空间中
<code>OtherSpace.cpp</code>	源程序文件，定义函数
<code>testSpace.cpp</code>	源程序文件，主函数，测试程序

源代码 7.5 头文件 myMath.h

```

1 // myMath.h
2 #ifndef MYMATH_H
3 #define MYMATH_H
4

```

编写各种声明

```

5 namespace mySpace
6 {
7     const double PI = 3.1415926;           // 定义常量
8     extern int max;                      // 外部参照访问声明
9     template <typename T> inline T Abs(T x) // 函数模板声明(描述)
10    {
11        return x>=0 ? x : -x;
12    }
13    double Sqrt(double);                // 函数声明
14 }
15 #endif

```

源代码 7.6 源程序文件 myMath.cpp

```

1 // myMath.cpp          编写各种定义
2 #include "myMath.h"
3
4 int mySpace::max = 1000;      // 定义“受限制的全局变量”
5                               // 注意作用域及作用域区分符
6 double mySpace::Sqrt(double x) // 函数定义
7 {
8     double y=1, yold=0;
9     do
10    {
11        yold = y;
12        y = (y+x/y)/2;
13    }while(Abs(y-yold)>1e-8);
14    return y;
15 }

```

源代码 7.7 头文件 myString.h

```

1 // myString.h
2 #ifndef MYSTRING_H
3 #define MYSTRING_H
4 namespace mySpace
5 {
6     int StrLen(const char *);           // 函数声明
7     char *StrCpy(char *, const char *); // 函数声明
8 }
9 #endif

```

源代码 7.8 源程序文件 myString.cpp

```

1 // myString.cpp
2 #include "myString.h"
3 int mySpace::StrLen(const char *str)
4 {
5     int len;
6     for(len=0; str[len]; len++)
7         ;
8     return len;
9 }
10
11 char *mySpace::StrCpy(char *dest, const char *source)

```

```

12 {
13     char *p = dest;
14     while(*dest++ = *source++)
15         ;
16     return p;
17 }

```

源代码 7.9 头文件 OtherSpace.h

```

1 // OtherSpace.h
2 #ifndef OTHERSPACE_H
3 #define OTHERSPACE_H
4 namespace OtherSpace
5 {
6     const double PI = 3.14;
7     extern int max;           // 外部参照访问声明
8     template <typename T> inline T Abs(T x) // 函数模板声明
9     {
10         return x>=0 ? x : -x;
11     }
12     int StrLen(const char *);    // 函数声明
13     char *StrCpy(char *, const char *); // 函数声明
14 }
15 #endif

```

源代码 7.10 源程序文件 OtherSpace.cpp

```

1 // OtherSpace.cpp
2 #include "OtherSpace.h"
3 /// 本例仅用作说明标识符不冲突。故意将下列函数的功能进行了修改
4 /// 不合常理，切勿模仿
5
6 int OtherSpace::max = 2000;      // 定义“受限制的全局变量”
7
8 int OtherSpace::StrLen(const char *str)
9 {                           // 将串结束标志计入串长度（不合常理，切勿模仿）
10     int len;
11
12     for(len=0; str[len]; len++)
13         ;
14     return len+1;
15 }
16
17 char *OtherSpace::StrCpy(char *dest, const char *source)
18 {                           // 将字符串颠倒复制（不合常理，切勿模仿）
19     char *p = dest;
20     int i;
21
22     for(i=0; source[i]; i++)
23         ;
24     for(i--; i>=0; i--)
25         *dest++ = source[i];
26     *dest = '\0';
27     return p;
28 }

```

源代码 7.11 源程序文件 testSpace.cpp

```

1 // testSpace.cpp
2 #include <iostream>
3 #include "myString.h"
4 #include "myMath.h"
5 #include "OtherSpace.h"
6
7 using namespace std;
8 using namespace mySpace;
9 using namespace OtherSpace;
10
11 int max = 3000;      // 全局变量
12
13 int main()
14 {
15     char str[80];
16     double x = 2.0;
17
18     cout << "::max" << ::max << endl << endl;
19
20     cout << "mySpace::max" << mySpace::max << endl;
21     cout << mySpace::PI << endl;
22     cout << mySpace::StrCpy(str, "C++Program.") << endl;
23     cout << mySpace::StrLen(str) << endl;
24     cout << Sqrt(x) << endl;
25     cout << mySpace::Abs(3) << endl;
26     cout << mySpace::Abs(-3.5) << endl << endl;
27
28     cout << "OtherSpace::max" << OtherSpace::max << endl;
29     cout << OtherSpace::PI << endl;
30     cout << OtherSpace::StrCpy(str, "C++Program.") << endl;
31     cout << OtherSpace::StrLen(str) << endl;
32     cout << OtherSpace::Abs(3) << endl;
33     cout << OtherSpace::Abs(-3.5) << endl;
34
35 }

```

程序的运行结果：

```

::max : 3000

mySpace::max : 1000
3.14159
C++ Program.
12
1.41421
3
3.5

OtherSpace::max : 2000
3.14
.margorP ++C
13
3
3.5

```

从程序及运行结果可知，可以从不同的编译单元向同一名字空间中添加标识符。不同名字空间中的标识符可以重名，还可以与全局变量重名。访问时只要不引起二义性，就能正确地匹配。用名字空间名和作用域区分符（`::`）可将它们严格地区分开来。若名字空间中的标识符与全局量相同，则“`::`标识符”表示访问全局量。在不引起二义性时，名字空间名称和双冒号运算符可省略。

回到本节开始所提出的问题。只要程序员 A 和程序员 B 分别设置各自的名字空间，并将各自所需要的全局变量放入其中成为受限制的全局变量即可解决标识符相冲突的问题。

7.4 隐藏函数的定义

软件开发过程包含许多环节。虽然编写源代码这一环节所占用的工作量不是最多的，但是所编写的源代码却是多个环节的具体体现。源代码中包含了多方面的成果，具有很高的技术含量。尤其是对于那些提供给用户进行二次开发的基础软件，如果不是开源（公开源代码）的，就很有必要将功能模块的实现技术细节隐藏起来，仅留一些接口给二次开发的用户使用。即给从事二次开发的用户一些黑箱，类似编译系统所提供的标准数学函数库一样，仅能看到头文件 `cmath` 中的内容。

下面，在 Visual C++ 6.0 集成开发环境中以第 7.1.3 小节求解一元二次方程为例，介绍具体处理方法。其他开发环境的处理方法是类似的。需要注意的是不同编译器所生成的目标代码文件的格式可能不同，不一定相互兼容。

将编译单元 `solver.cpp` 单独编译生成目标代码文件 `solver.obj`（该文件在 Debug 子目录中）。将文件 `solver.h`, `solver.obj` 提供给二次开发的用户使用（不必提供源程序文件 `solver.cpp`，该文件由版权所有者保留）。

对于二次开发者，使用 `solver.h`, `solver.obj` 的步骤如下：

① 创建一个工程文件，将文件 `solver.h`, `solver.obj` 复制至其工程文件的工作目录中，另外添加一个源程序文件，如第 7.1.3 小节中的 `solverTesting.cpp`。

② 将头文件 `solver.h` 添加到当前的工程文件中。从 Visual C++ 6.0 主菜单起，依次选择 Project | Add To Project | Files，在弹出的对话框中选择文件 `solver.h`，单击 OK 按钮。

③ 将目标代码文件 `solver.obj` 添加到当前工程文件的连接选项设置中。从 Visual C++ 6.0 主菜单起，依次选择 Project | Settings，在弹出的对话框中选择 Link 选项卡，在其中的 Object | library modules 编辑栏中添加文件名 `solver.obj`（用空格分隔已经存在的项目），单击 OK 按钮。

④ 编译、连接生成可执行文件，最后运行之。

由以上的步骤可见，用户仅根据头文件 `solver.h`（以及另外的说明文档、帮助文档）中的信息进行二次开发，并且不再需要源程序文件 `solver.cpp`。用户程序中调用函数时，仅能参考的是一些说明文档、帮助文档和头文件中的函数原型。这也说明函数原型是程序员之间交流信息的接口。同时，还看到 `Delta` 函数仅“内部使用”，其接口也不暴露给二次开发的用户，隐藏得更深。

MinGW Developer Studio 集成开发环境中目标代码文件的扩展名为 .o，其他的具体操作与 Visual C++ 的类似，参见第 9.4.2 小节。

7.5 小 结

存储类型决定了标识符在内存中的位置、生命期和可见性等。它规定了在多文件结构中的连接特性。非静态全局名字具有外部存储属性，它能被多个编译单元共享，也容易引起冲突；同时，也处于随时随地可能被修改的危险境地。使用全局变量可能影响程序的移植性。建议读者在程序设计中不使用全局变量，或者只使用属于某名字空间中“受限制的全局变量”。由于静态全局变量具有在其编译单元内“私有”及“共享”的特性，可以适当加以利用。

在本教材中，虽然我们所设计的程序基本上没有使用全局变量，但是第 2~8 章中所设计的函数均是全局的。对于众人集体开发同一个程序时，全局变量或外部函数的名字可能发生冲突。若所有设计者将其设计的全局变量、外部函数放入各自设计的名字空间中，或者使用面向对象方法将函数及数据封装起来，可消除不同设计者对于标识符命名的冲突。使用多文件结构将声明部分放入头文件，将全局变量定义、函数定义（即函数的实现细节）部分放入源程序文件，并编译成目标代码文件后隐藏起来（仅提供头文件和目标代码文件），可有效地保护源代码。

练习 7

(1) 设计一个名字空间 myString 将练习 6 中用户自定义版的字符串处理函数放入该名字空间中。编写一个头文件和相应的源程序文件，再编写一个含主函数的源程序文件对名字空间进行测试。

(2) “24点”计算。给定四个不超过 10 的正整数 a、b、c 和 d，允许交换其次序并在其间添加括号及运算符 +、-、*、/（浮点数除法）使其结果为 24。若无法使算式的结果为 24 则输出 Impossible。

【提示】 不区分给定的四个数中是否有相同的，也不利用加法和乘法满足的交换律，所有的情形共 $4! \times 4^3 \times 5 = 7680$ 种。**①** 四个数的全排列共有 $4! = 24$ 种。**②** 四个数需要三次算术运算（从四种运算符中可重复地任选三个运算符，共有 $4^3 = 64$ 种情形）。**③** 共有如下五种添加括号的方式，其中“▲”“■”“★”分别表示某种算术运算符：

[(a ▲ b) ■ c] ★ d	例如 $[(1+2)+3]*4 = 24$
[a ▲ (b ■ c)] ★ d	例如 $[5-(1/5)]*5 = 24$
(a ▲ b) ■ (c ★ d)	例如 $(1+2)*(3+5) = 24$
a ▲ [(b ■ c) ★ d]	例如 $2*[(3+4)+5] = 24$
a ▲ [b ■ (c ★ d)]	例如 $6*[8-(1+3)] = 24$

建议将运算设计成函数，利用如下指向函数的指针数组对自定义的 add（加）、sub（减）、mul（乘）和 div（除）函数进行有效而统一的管理和使用：

```
double (*op[4])(double, double) = {add, sub, mul, div};
```

第8章 链 表

本章通过“评委评分”程序的进一步改编，介绍一种数据结构——单向链表。这一部分内容是高级语言程序设计的传统“保留节目”。是读者今后深入学习数据结构课程的基础，也作为从过程化程序设计到面向对象程序设计的过渡。希望读者仿照本章的程序，设计一个“通讯录”程序或“学生成绩管理”程序或“图书资料管理”程序以进一步掌握指针、函数的概念和用法。通过链表应用程序的调试真正掌握链表的操作技术。

8.1 结构体

程序设计高级语言允许程序员利用已经存在的数据类型（包括基本数据类型、其他构造数据类型）自行构造新的数据类型。结构体是一种构造数据类型。构造一种新的数据类型需要：

- ① 规定这种新类型的数据组织形式。
- ② 提供这种新数据类型的操作方法。

面向对象程序设计部分将对此问题进行深入细致的介绍。

8.1.1 数据组织形式描述

将一些数据类型不尽相同但有紧密关系的数据组织在一起是很有必要的。例如，描述一名“学生”的基本情况。“学生”的基本属性有学号（整型或者字符串型）、姓名（字符串型）、各科成绩（整型）和平均成绩（浮点型）等。将这些属性组织起来作为整体形成一种新的数据类型。

```
struct Student
{
    int id;
    char name[9];           // 可存放四个汉字
    int Chinese, math, English, physics, chemistry;
    double average;
};
```

这样，便完成了数据组织形式的描述，形成了一种新的构造数据类型——结构体。新的数据类型名称 `Student` 是由程序员根据标识符的命名规则自行命名的。在 `Student` 中描述的属性（如 `id`, `name`, `Chinese`, `math`, `English`, `physics`, `chemistry` 和 `average`）被称为结构体成员。结构体成员有其自己的数据类型。

8.1.2 创建结构体对象

与基本数据类型一样，可以用这种新的数据类型定义变量、定义数组、定义指针变量、申请堆变量和申请堆数组，可以声明引用，以及在函数之间传递这种新数据类型的数据。

所描述的结构体并不占用内存空间。只有当用这种新数据类型定义变量、数组、申请堆变量或申请堆数组时才意味着分配内存空间。每个这种变量占用 `sizeof(Student)` 字节的内存空间。今后，称定义结构体变量、结构体数组、申请堆变量、申请堆数组分别为创建结构体对象、创建结构体对象数组、创建结构体堆对象和创建结构体堆对象数组。

可以定义目标数据类型为结构体类型的指针变量。当然，指向结构体对象的指针变量仍然仅占 `sizeof(void*)` 字节的内存空间。可以声明结构体对象的引用，引用同样不占用内存空间。例如：

```
Student x, y;           // 创建两个 Student 对象;
Student *p = &x;         // 定义一个指针变量;
Student &rx = x;        // 声明一个引用(不占空间);
Student a[50];          // 创建结构体数组;
p = new Student [100];   // 创建堆对象数组
delete [] p;            // 释放堆对象数组所占用的堆内存资源
```

其中，对象 `x, y` 分别各占用 `sizeof(Student)` 字节；指针变量 `p` 在内存中仍然只占用 `sizeof(void*)` 字节，并用对象 `x` 的地址对该指针进行初始化；`rx` 为声明的引用，不另占用内存空间，它是对象 `x` 的别名；数组名 `a` 仍然为数组的首地址；用 `new, delete` 操作申请及释放堆对象数组。

还可以将结构体对象的值、指向结构体对象的指针值（即某结构体对象的地址），结构体对象本身（引用型）和指向结构体对象的指针本身（指针的引用）作为实际参数传递给函数或作为函数的返回类型。参见第 8.2 节中的函数。

8.1.3 访问对象的成员

优先级为 2 的圆点运算符“.”用于结构体对象访问其成员；箭头运算符“->”（由减号及大于号组成）用于结构体指针访问其目标对象的成员。如：

```
Student s, *ps=&s, &rs=s; // 仅创建了一个对象 s
s.id      = 1;
strcpy(s.name, "张三");    // 字符串复制。不能用 s.name="张三";
ps->Chinese = 85;          // 指针变量访问其目标对象的成员
(*ps).math = 86;            // 等价于 ps->math = 86;
rs.English = 82;            // 引用访问其“绑定”的目标对象的成员
rs.physics = 80;
s.chemistry = 83;
ps->average = (s.Chinese + rs.math + s.English
+ ps->physics + s.chemistry)/5.0;
```

优先级为 4 的运算符为指向成员的指针，用法如下。例如，定义一个专用的指向 `Student` 类型成员 `average` 的指针 `pAver`。

```
// 接上面的对象、指针变量
double Student::*pAver = &Student::average;
cout << "对象 s 的平均成绩为 " << s.*pAver << endl;
cout << "对象 s 的平均成绩为 " << ps->*pAver << endl;
```

输出结果均为“对象 `s` 的平均成绩为 83.2”。

8.2 链表的概念

前面几章谈到某“大奖赛”，若其组委会决定：允许参赛选手中途退出比赛（退出比赛者将不参加排名，即该选手已经取得的部分成绩应该不对其他选手的名次产生影响）；比赛进行中仍可接收新报名参赛的选手。

这两个方面问题的共同点是参赛选手人数的变化（增加或减少）。用以前的方法（如数组、堆数组）解决这个问题是不方便的。在第5, 6章介绍的程序中，首先配置的是评委人数、参赛选手人数（隐含地假定人数不变），然后分配适当的空间。若选手人数发生变化，则需要“重新分配空间、复制已有的数据、释放原来的空间”使相关数据在内存中连续存放。这样将导致大量的数据在内存中经常移动。

一种行之有效的解决方案是让选手们依次记录其下一位选手的座位号码，各选手的这种驻扎地址号码不一定连续、不一定有序，但固定不动，直至被销毁。具体地，现场主持人只需要找到首位选手的座位号码，选手们各负责记录其下一位选手的座位号码，末尾的选手所记录的座位号码为特殊的NULL（空）即可。这种单方向联络的连接方式被称为单向链表。构成单向链表的元素被称为结点。结点应该具有承载数据及指向下一个结点的能力。

在这种链式连接结构中，删除某个结点只需要将该结点保存的下一个结点地址“交给”其上一个结点即可。链表各结点在内存中不移动位置。添加一个结点到链表中，也需要局部地调整相关结点所应该记录的下一个结点地址，也不必移动各个结点在内存中的位置。

由于约定了单向链表尾结点的结束标志，故只需要把握单向链表的链首结点地址即可。结点个数、各结点在内存中的地址都容易得到。这也说明将单向链表传递给函数时，只需要传递链首地址。这个特性有些像具有串结束标志的C-字符串。与字符串不同的是，字符串各元素在内存中连续存放，依靠与字符串首元素地址的偏移量（下标）便能找到指定的字符元素；而链表各结点在内存中不一定连续存放，要依靠存放在各结点间的索引“按图索骥”访问所需要的数据。

8.2.1 结点的结构

形成链式结构的结点可采用如下数据组织形式的描述。

```
const int REFEREES = 10;
struct Player
{
    int num;                                // 选手编号
    char name[7];                            // 选手姓名（存放三个汉字）
    double x[REFEREES], aver;                // 选手分数
    int rank;                                // 选手名次
    Player *next;                            // 同类指针，下一位选手的地址
};
```

【注意】 结构体中不能嵌套地将本结构体对象作为成员。但可以将指向本结构体对象的指针变量作为成员（因为指针成员所占用的`sizeof(void*)`字节数是个定值）。在结点的成员中，除指向下一个同类结点的指针`next`外，其他成员统称为数据域成员。

这样一来，就描述了一种新的数据类型。这种数据类型名为 Player（由程序员命名），一个这种数据类型的对象在内存中占用 `sizeof(Player)` 字节。

8.2.2 单向链表

我们约定：为了链表处理的方便和统一，本章中凡作为链表结点的变量（称为结点对象）均为堆变量（即堆对象）。链表的一般形式如图 8-1 所示。

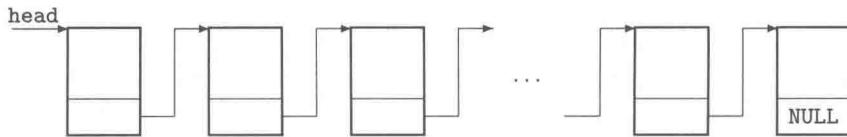


图 8-1 单向链表

单向链表是一种能够存储数据的容器，数据以结点（存放在结点的数据域中）的面貌出现，以“手拉手”的链式结构联络。需要设计一些对链表进行操作的函数使之便于应用。各结点都无原名，皆通过间接名访问，结点的地址存储在其前一个结点的指针域中，首结点的地址存放在 `head` 指针变量中。

【说明】 另一种单向链表被称为带头结点的单向链表。带头结点的链表中设有一个专门结点，被称为头结点。头结点数据域中的数据无实际意义，头结点的 `next` 指针指向的结点为链表真正的第一个结点。头结点可以不在堆空间中。带头结点的单向链表更加容易控制，因为不管链表结点如何增减，头结点可以始终不动。头结点在函数间的传递是简单的，不必使用引用。本书不讨论这种链表。

1. 空链表

定义一个结点类型的指针变量并初始化为空指针，便是构建成功了一条单向链表（有头有尾）。该链表的结点数为 0。构造一条空链表是进行链表操作的起点，虽简单但重要。如：

```
Player *head = NULL;
```

2. 首结点处的操作

单向链表在首结点处的操作是最简单的，这种操作有许多应用。若仅在链首进行操作，则此时的单向链表就是栈结构的一种具体实现——链首为栈顶，链尾为栈底。

链表创建（一般只需要创建一条空链表即可）后，添加一个新结点到链首前成为新的链首。用函数实现如下。

```

1 void InsertBeforeHead(Player *&head, int num, char *name)
2 {
3     Player *p = new Player; // 创建一个新结点（堆对象）
4
5     p->num = num; // 确定数据成员 num 的值
6     strncpy(p->name, name, sizeof(p->name));
7         // 为了不破坏其他数据，限定复制字符个数
8     p->name[sizeof(p->name)-1] = '\0'; // 确保有串结束标志
9         // 其他数据成员赋值（略）
10
  
```

```

11     p->next = head;           // 新结点将原链首结点当成下一个结点
12     head = p;                // head 记录新结点的地址
13 }
```

其中第 3 行为创建一个堆结点，第 11、12 行完成将该结点插入到链表中，并成为新的链首结点。第 5~8 行为给新结点的数据域成员赋值。

删除链首结点的函数为

```

1 void DeleteHead(Player *&head)
2 {
3     Player *p = head;          // 记录当前链首结点地址
4     if(head != NULL)          // 如果不是空链表
5     {
6         head = head->next;   // 下一个结点成为新的链首结点，可能为NULL
7         delete p;             // 释放原链首结点，隐含假定结点在堆空间中
8     }
9 }
```

特别需要指出的是上述两个函数在链首操作，需要修改链首指针的指向，使其指向新的链首，需要链首指针变量直接双向传递。因此函数的形式参数都应该设置成 `Player *&` 型，即传递指针变量本身。同时，前一个函数还设置（修改）了首结点的数据域成员的值和 `next` 指向（间接双向传递）。

8.3 链表操作

8.3.1 遍历

遍历是指将链表中的所有结点均按某种方式处理一次。遍历单向链表的方法是从链首结点开始，沿结点的指向依次处理每个结点，直到尾结点为止。典型的遍历操作采用与下面类似的循环语句。

```

1 Player *p;
2 for(p=head; p!=NULL; p=p->next)
3 {
4     // 处理当前结点的数据成员, p->数据成员
5 }
```

下面的链表版“评委评分”程序中，有如下遍历操作：输出所有结点的所有数据；释放所有结点；计算各选手的名次。

1. 输出所有结点的数据

请读者分析下面的函数中为什么可以使用语句 `head=head->next;` 而不担心实参（链首指针）被修改。

```

1 void Show(const Player *node)
2 {
3     cout << node->num << " " << node->name;
4     for(int i=0; i<REFEREES; i++)
5         cout << " " << node->x[i];
```

```

6     cout << " " << node->aver << " " << node->rank << endl;
7 }
8
9 void ShowList(const Player *head)
10 {
11     if(head==NULL)
12     {
13         cout << "链表为空。" << endl;
14         return;
15     }
16     while(head)
17     {
18         Show(head);           // 输出的二维表格中数据可能未对齐
19         head = head->next;   // 参见下一节程序有格式控制的输出
20     }
21 }
```

2. 释放所有结点

这一操作与结点的数据域无关，故可设计成函数模板。

```

1 template <typename T> void FreeList(T *&head)
2 {
3     T *p;
4     while(head!=NULL)
5     {
6         p = head;
7         head = head->next; // 要求指向下一个结点的指针成员名为 next
8         delete p;          // 隐含假定所有的结点皆为堆结点
9     }
10 }
```

上述函数模板适用于多种类型的单向链表，只要求结点中指向下一个结点的指针成员名称为 `next`，且链表中的所有结点均为堆结点即可。

3. 统计计算

统计链表中的结点数。由于此项操作与结点的数据域无关，故也可写成函数模板。

```

1 template <typename T> int Count(const T *head)
2 {
3     int n=0;
4     const T *p;           // 常量指针（所指之处皆被视为常量）
5     for(p=head; p!=NULL; p=p->next)
6         n++;
7     return n;
8 }
```

统计满足如下条件的选手数：有三名及三名以上的评委给该选手的分数低于 8.5 分。

```

1 int Stat(const Player *head)
2 {
3     int n=0, m, i;
4     const Player *p;
5     for(p=head; p!=NULL; p=p->next)
6     {
```

```

7     m = 0;
8     for(i=0; i<REFEREES; i++)
9         if(p->x[i]<8.5)
10            m++;
11        if(m>=3)
12            n++;
13    }
14    return n;
15 }
```

4. 定位某结点

给定某整数 num，要求在链表中查找编号等于 num 的结点。从链首起，依次比较结点数据域中的相应数据项，若找到则返回该结点的地址；若查完所有结点都未找到，我们约定返回 NULL。这种定位操作不一定遍历整个链表。

```

1 Player *Search(Player *head, int num)
2 {
3     Player *p;
4     for(p=head; p!=NULL; p=p->next)
5         if(p->num == num)
6             break;
7     return p;
8 }
```

5. 定位及继续定位

链表中满足某条件的结点可能没有或可能有多个，此时需要定位和继续定位。例如，查找所有第 k 名的选手，并输出他们的信息。这个函数应实现“返回满足条件的结点的地址，并为继续定位确定搜索新起点”。请注意分析其中静态局部指针变量 p 的特性，注意局部自动指针变量 temp 以及带默认值的参数的作用。

```

1 Player *Locate(Player *head, int rank, int restart=0)
2 {
3     static Player *p=head; // 注意这里采用了静态局部指针变量
4     Player *temp;
5
6     if(restart) p = head; // restart非零时从头开始
7
8     for( ; p!=NULL; p=p->next)
9         if(p->rank == rank)
10            break;
11     temp = p;
12     if(p) p = p->next; // 确定继续定位搜索的新起点
13     return temp;
14 }
```

下面的测试函数说明了上述定位及继续定位函数的使用方法。

```

1 void Find(Player *head)
2 {
3     int n, rank;
4     Player *p;
```

```

5     cout << "查询所有第 k 名的结点。请输入 k: ";
6     cin >> rank;
7     n = 0;
8     p = Locate(head, rank, 1); // 从头开始
9     while(p!=NULL)
10    {
11        Show(p);           // 输出指针 p 的目标对象的数据成员值
12        n++;
13        p = Locate(head, rank); // 继续定位
14    }
15    cout << "共有 " << n << " 位选手并列第 "
16    << rank << " 名。 " << endl;
17 }

```

8.3.2 插入一个结点

插入一个结点的操作（如图 8-2 所示），需按插入到链首前，插入其他处（包括链表中间结点、链表尾结点后）分别处理。其中插入到链首前的操作需要修改链表首地址。因此，插入函数中需要引用型形式参数 Player *&head。

插入一个结点到链表首结点前的操作已经在前面做了介绍。插入一个结点到链表中间或者尾部的操作如下。

① 找到插入点的前一个结点的地址，称其为“前哨”，记其地址为 pGuard。

② 创建一个新的堆结点并对其数据域成员赋值。

③ 将新结点的 next 指针指向“前哨”的下一个结点，即“前哨”原下一个结点成为新结点的下一个结点。

④ 令“前哨”的 next 指针改向指向新结点，不再指向原来的结点。

插入一个结点到尾部是同样的，只是此时 pGuard->next 为 NULL。

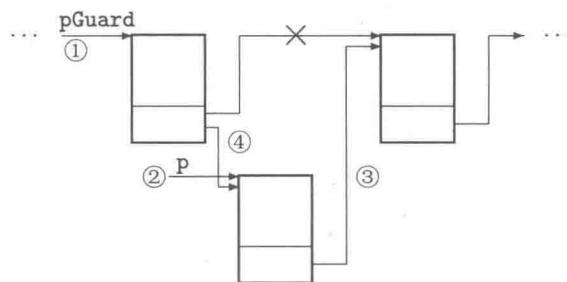


图 8-2 插入一个结点到单向链表中

请读者将以上四步与下一节的源程序对应。

8.3.3 删 除一个结点

删除一个结点的操作（如图 8-3 所示），需按删除首结点，删除其他结点（包括链表中间结点、链表尾结点）分别处理。其中删除链首结点将改变链首指针，因此删除结点的函数需要引用型形式参数 Player *&head。

删除链首结点的操作已经在前面做了介绍。删除链表中间或者尾部的一个结点的操作如下。

① 找到待删结点的前一个结点，称其为“前哨”，记其地址为 pGuard。

- ② 记录下“前哨”的下一个结点（即待删结点）的地址 `p=pGuard->next`。
 ③ 将“前哨”的 `next` 指针指向待删结点的下一个结点 `pGuard->next=p->next` 使待删结点脱链。
 ④ 释放待删结点所占用的堆内存单元资源。
- 删除尾结点的操作是完全相同的，只是此时的待删结点的 `p->next` 为 `NULL`，即 `pGuard->next->next` 为 `NULL`。请读者将以上四步与下一节的源程序对应。

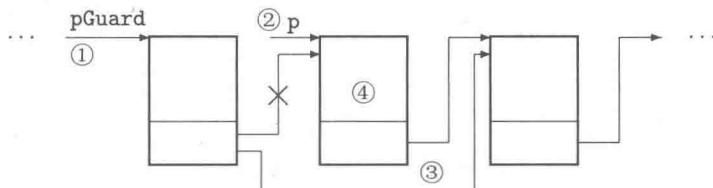


图 8-3 从链表中删除一个结点

8.3.4 链表版“评委评分”程序清单

程序采用多文件结构：`contest.h`, `contest.cpp` 是一组文件，头文件 `contest.h` 中描述了一种新的数据类型 `Player` 并声明了各函数。以便被两个编译单元包含。源程序文件 `contest.cpp` 中对各函数进行了定义。其中有关文件读写操作暂不要求读者弄懂。文件操作能将数据永久性保存在磁盘文件中，便于程序或其他工具软件读写。

源程序文件 `Main.cpp` 中，主函数的结构为循环，可反复执行“输出菜单、接收选择、根据选择调用不同函数”的操作。

读者可有不同的次序阅读本程序。

① 按程序的执行次序阅读：从主函数出发，遇到函数调用时阅读相应的函数，返回后继续阅读主函数，直到主函数结束。

② 先阅读各个函数，掌握函数的功能，最后阅读主函数。

本节中的程序没有使用任何全局变量，各函数所需要的数据皆可根据形式参数获得；函数的处理结果皆从函数返回或通过参数“返回”。

源程序清单如下。

源代码 8.1 链表版“评委评分”程序头文件

```

1 // contest.h
2 #ifndef CONTEST_H
3 #define CONTEST_H
4
5 const int REFEREES = 10;
6
7 struct Player
8 {
9     int num;           // 选手编号
10    char name[7];     // 选手姓名
11    double x[REFEREES], aver; // 选手分数
12    int rank;          // 选手名次
13    Player *next;      // 同类指针，存放下一位选手的地址
14 };
15

```

```

16 int Pause(const char *prompt="按任意键继续……");
17 char *trim(char *str); // 去掉 C-字符串前后的空白字符
18
19 // 结点操作函数的函数原型
20 void Show (const Player *node); // 输出结点数据
21 Player *ComputeAver (Player *node); // 计算结点最后得分
22 Player *Modify (Player *node); // 修改结点数据
23
24 // 链表操作函数的函数原型
25 void FreeList(Player *&head); // 释放所有结点
26 int Insert (Player *&head, Player *p); // 插入一个结点
27 int Delete (Player *&head, int num); // 删除一个结点
28 Player *Search (Player * head, int num); // 查询结点
29 Player *Locate (Player * head, int rank, int restart=0);
30 // 定位、继续定位
31 Player *Modify (Player * head, int num); // 修改结点数据
32 void ShowList(const Player * head); // 输出所有结点数据
33 void ComputeRank(Player *head); // 计算各选手的名次
34
35 enum SORT_FLAG { SORT_NUM=0, SORT_NAME=1, SORT_RANK=2 };
36 // 枚举型声明, 用于排序
37 int compare(const Player *p1,const Player *p2,SORT_FLAG flag);
38 // 用于排序的比较函数
39 void SortList(Player *&head, SORT_FLAG flag=SORT_RANK);
40 // 按指定的数据项排序
41 // 文件操作函数的函数原型(供参考)
42 int TSave(const Player * head, const char *filename);
43 int TLoad(Player *&head, const char *filename); // 按文本文件格式
44 int BSave(const Player * head, const char *filename);
45 int BLoad(Player *&head, const char *filename); // 按二进制格式
46
47 #endif

```

源代码 8.2 链表版“评委评分”函数定义源程序文件

```

1 // contest.cpp
2 #include <iostream>
3 #include <iomanip>
4 #include <cstring>
5 #include <conio.h> // 为了使用 getch 函数
6 #include "contest.h"
7 #include <fstream> // 为了文件操作
8 using namespace std;
9
10 int Pause(const char *prompt)
11 {
12     int key;
13     cout << prompt << flush;
14     key = getch();
15     cout << endl;
16     return key;
17 }
18
19 char *trim(char *str) // 去掉 C-字符串前后的空白字符
20 {
21     int i, j;
22     for(i=0; str[i]==' ' || str[i]=='\t'; i++)
23         ;

```

```

24     for(j=0; str[j]==str[j+i]; j++)
25         ;
26     for(j--; j>=0 && (str[j]=='\u' || str[j]=='\t'); j--)
27         ;
28     str[j+1] = '\0';
29     return str;
30 }
31
32 void FreeList(Player *&head)
33 {
34     Player *p;
35     while(head!=NULL)
36     {
37         p = head;
38         head = head->next;
39         delete p;           // 隐含假定结点均为堆对象
40     }
41 }
42
43 int Insert(Player *&head, Player *p)
44 { // 假设链表已经按编号顺序连接，则不会出现重号
45     // 若链表未按编号排序，则将新结点插入到首个更大的编号结点前
46     Player *heapNode, *pGuard;
47
48     if(head==NULL || p->num < head->num)
49     {
50         heapNode = new Player;
51         // p 指向的目标对象“来历不明”，不宜将其直接插入到链表中
52         // 故新建对象，以保证所有的结点均为堆结点
53         *heapNode = *p;           // 录用其数据域中的数据
54
55         heapNode->next = head;    // 插至链首前
56         head = heapNode;         // 成为新链首，需要传递引用
57
58         ComputeAver(heapNode);   // 计算最后得分
59         ComputeRank(head);      // 计算名次
60         return 1;                // 插入结点成功
61     }
62
63     if(p->num == head->num)        // 此时，链表至少有一个结点
64     {                                // 访问 head 目标对象成员合法
65         Show(head);
66         return 0;                   // 与首结点重号，拒绝插入
67     }
68
69     pGuard = head;                  // 链表非空，pGuard->next 合法
70     while(pGuard->next!=NULL && pGuard->next->num < p->num)
71         // 在 pGuard->next 不为空的前提下才可进一步访问成员
72         pGuard = pGuard->next;       // ① 循环结束后，确定插入点前哨
73
74     if(pGuard->next!=NULL && pGuard->next->num == p->num)
75     {                                // pGuard->next!=NULL 不能少
76         Show(pGuard->next);
77         return 0;                   // 与其他结点重号，拒绝插入
78     }
79
80     heapNode = new Player;          // ② 新建堆对象
81     *heapNode = *p;
82

```

```

83     heapNode->next = pGuard->next; // ③ 指向下一个结点
84     pGuard->next = heapNode; // ④ 加入到前哨后
85
86     ComputeAver(heapNode); // 计算最后得分
87     ComputeRank(head); // 计算名次
88     return 1; // 插入结点成功
89 }
90
91 int Delete(Player *&head, int num)
92 {
93     Player *p, *pGuard;
94     char yn;
95
96     if(head == NULL) return -3; // 链表为空
97
98     if(head->num == num) // 此时链表一定非空, 可访问首结点数据成员
99     {
100         Show(head);
101         yn = Pause("是否删除该结点? [y/[N]]"); // 默认 N
102         if(yn=='y' || yn=='Y')
103         {
104             p = head; // 暂存首结点地址
105             head = head->next; // 首结点脱链
106             delete p; // 删除原首结点, 需要传递引用
107             ComputeRank(head); // 重新排名次
108             return -1;
109         }
110         return 0; // 结点未被删除
111     }
112
113     pGuard = head; // 此时至少有一个结点, 且首结点不是待删结点
114     while(pGuard->next!=NULL && pGuard->next->num != num)
115         pGuard = pGuard->next; // ① 确定待删结点的前哨
116
117     if(pGuard->next != NULL) // 此条件判断是必须的
118     {
119         p = pGuard->next; // ② p 指向待删结点
120         Show(p);
121         yn = Pause("是否删除该结点? [y/[N]]");
122         if(yn=='y' || yn=='Y')
123         {
124             pGuard->next = p->next; // ③ 待删结点脱链
125             delete p; // ④ 释放堆结点
126             ComputeRank(head); // 重新排名
127             return -1;
128         }
129         return 0; // 结点未被删除
130     }
131     else
132         return -2; // 查无此人
133 }
134
135 Player *Search(Player *head, int num)
136 {
137     while(head!=NULL && head->num!=num)
138         head = head->next;
139     return head; // 与上节不同的实现方法
140 }
141

```

```

142 Player *Locate(Player *head, int rank, int restart)
143 {
144     static Player *p=head;           // 注意静态局部指针变量的特点
145     Player *temp;
146     if(restart) p = head;          // 若 restart 非零，则从头开始
147     for( ; p!=NULL; p=p->next)
148         if(p->rank == rank)
149             break;
150     temp = p;
151     if(p!=NULL) p = p->next;
152     return temp;
153 }
154
155 Player *Modify(Player *node)
156 {
157     char str[80];
158     if(node==NULL) return node;
159
160     cout << "原有记录" << endl;
161     Show(node);
162
163     cout << "不允许修改编号，可修改其他属性" << endl;
164     cout << "请输入姓名: ";
165     cin.sync();                  // 刷新输入缓冲区
166     cin.getline(str, 80);
167     strncpy(node->name, str, sizeof(node->name));
168                         // 允许用户输入超长字符串
169     node->name[sizeof(node->name)-1] = '\0'; // 截断超长部分
170
171     cout << "重新输入新成绩: " << endl;
172     for(int i=0; i<REFEREES; i++)
173         cin >> node->x[i];
174     ComputeAver(node);          // 重新计算最后得分
175     return node;
176 }
177
178 Player *Modify(Player *head, int num)
179 {
180     Player *p;
181
182     p = Search(head, num);
183     if(p!=NULL)
184     {
185         Modify(p);            // 调用重载函数，并非递归调用
186         ComputeRank(head);   // 重新排名
187     }
188     return p;
189 }
190
191 void Show(const Player *node)      // 输出一个结点的数据
192 {
193     cout << setprecision(3) << showpoint;
194     cout << setw(4) << node->num << " "
195         << setw(6) << node->name << " ";
196     for(int i=0; i<REFEREES; i++)
197         cout << setw(5) << node->x[i];
198     cout << setw(6) << node->aver
199         << setw(6) << node->rank << endl;
200 }

```

```
201
202 void ShowList(const Player *head)
203 {
204     int n;
205     const Player *p;
206
207     if(head==NULL)
208     {
209         cout << "链表为空。" << endl;
210         return;
211     }
212
213     cout << setprecision(3) << showpoint;
214     cout << "\n\u201cNo.\u201cNum.\u201cName\u201d";
215     for(int i=0; i<REFEREES; i++)
216         cout << "\u201c\u201c" << char('A'+i);
217     cout << "\u201c\u201cAver.\u201cRank" << endl;
218     n = 0;
219     for(p=head; p!=NULL; p=p->next)
220     {
221         cout << setw(3) << ++n;      // 输出序号
222         Show(p);
223     }
224 }
225
226 Player *ComputeAver(Player *node)
227 {
228     double max, min, sum;
229     max = min = sum = node->x[0];
230     for(int i=1; i<REFEREES; i++)
231     {
232         if(node->x[i]>max) max = node->x[i];
233         if(node->x[i]<min) min = node->x[i];
234         sum += node->x[i];
235     }
236     node->aver = (sum-max-min)/(REFEREES-2);
237     return node;
238 }
239
240 void ComputeRank(Player *head)
241 {
242     Player *p, *q;
243
244     for(p=head; p!=NULL; p=p->next)
245     {
246         p->rank = 1;
247         for(q=head; q!=NULL; q=q->next)
248         {
249             if(q->aver > p->aver)
250                 p->rank++;
251         }
252     }
253 }
254
255 int compare(const Player *p1, const Player *p2, SORT_FLAG flag)
256 {                                         // 用于排序的比较函数
257     int result = 0;
258     switch(flag)
259     {
```

```

260     case SORT_NUM : result = p1->num - p2->num;
261     break;
262     case SORT_NAME: result = strcmp(p1->name, p2->name);
263     break;
264     case SORT_RANK: result = p1->rank - p2->rank;
265   }
266   return result;
267 }
268
269 void SortList(Player *&head, SORT_FLAG flag) // 链表插入排序法
270 {
271     // 结点不移动, 仅修改结点间的连接顺序
272     if(head==NULL) return;
273
274     // 将原链表拆成两条链表
275     Player *oldLink = head->next; // “旧”链表从原第二个结点起
276     head->next = NULL;           // “新”链表为单（原链首）结点链表
277
278     Player *p, *pGuard;
279     while(oldLink!=NULL)
280     {
281         p = oldLink;
282         oldLink = oldLink->next; // 旧链表链首结点脱链
283
284         if(compare(p, head, flag)<0)
285         {
286             p->next = head;
287             head = p;           // 脱链结点插入到新链表首结点前
288             continue;          // 提前结束本轮循环
289         }
290         pGuard = head;
291         while(pGuard->next && compare(pGuard->next,p,flag)<0)
292             pGuard = pGuard->next;
293         p->next = pGuard->next; // 脱链结点插入到适当位置
294         pGuard->next = p;
295     }
296
297 // 文件操作（供参考）
298 int TSave(const Player *head, const char *filename)
299 {
300     // 将链表各结点的数据按文本文件格式写入文件
301     ofstream outfile(filename); // 建立文件
302     const Player *p;
303     int i, n=0;
304     outfile << setprecision(3) << showpoint;
305     for(p=head; p; p=p->next)
306     {
307         outfile << p->num << " "
308             << p->name << "\t"; // 注意字符 '\t' 的作用
309         for(i=0; i<REFEREES; i++)
310             outfile << setw(5) << p->x[i] << " ";
311         outfile << setw(6) << p->aver << " "
312             << setw(6) << p->rank << endl;
313         // 注: p->next 所存放的地址值不必存储, 参见下面的函数
314         n++;
315     }
316     outfile.close();           // 关闭文件
317     return n;
318 }
```

```

319 int TLoad(Player *&head, const char *filename)
320 { // 从指定的文本格式文件中读取数据，形成结点插入到链表中
321     ifstream infile(filename); // 打开文件
322     Player x;
323     char str[80];
324     int i, n=0;
325
326     while(infile>>x.num)
327     {
328         infile.getline(str, 80, '\t');// 遇到字符 '\t' 结束
329         trim(str);
330         strncpy(x.name, str, sizeof(x.name));
331         x.name[sizeof(x.name)-1] = '\0';
332         for(i=0; i<REFEREES; i++)
333             infile >> x.x[i];
334         infile >> x.aver >> x.rank;
335         Insert(head, &x); // 仅提供数据域数据，交给插入函数处理
336         n++;
337     }
338     infile.close();
339     return n;
340 }
341
342 int BSave(const Player *head, const char *filename)
343 { // 将链表各结点的数据按二进制格式写入文件
344     ofstream outfile(filename, ios::binary); // 建立文件
345     const Player *p;
346     int n=0;
347     for(p=head; p; p=p->next)
348     {
349         outfile.write((char*)p, sizeof(*p));
350         // 注：每个结点占sizeof(*p)字节。p->next 存放的地址值也被存储
351         n++;
352     }
353     outfile.close(); // 关闭文件
354     return n;
355 }
356
357 int BLoad(Player *&head, const char *filename)
358 { // 从指定的二进制格式文件中读取数据，形成结点插入到链表中
359     ifstream infile(filename, ios::binary); // 打开文件
360     Player x;
361     int n=0;
362     while(infile.read((char*)&x, sizeof(x)) != NULL)
363     {
364         trim(x.name);
365         Insert(head, &x);
366         n++;
367     }
368     infile.close();
369     return n;
370 }

```

源代码 8.3 链表版“评委评分”程序主程序文件

```

1 // Main.cpp
2 #include <iostream>
3 #include <iomanip>

```

```

4 #include <ctime>
5 #include "contest.h"
6 using namespace std;
7
8 int Menu()
9 {
10    cout << "\n\t1--添加多个结点(N)" // 输出菜单
11    << "\n\t2--插入一个结点(I)"
12    << "\n\t3--删除一个结点(D)"
13    << "\n\t4--输出链表数据(V)"
14    << "\n\t5--查询一个结点(L)"
15    << "\n\t6--根据名次查询(R)"
16    << "\n\t7--修改结点数据(M)"
17    << "\n\t8--释放所有结点(F)"
18    << "\n\t9--排序(S)"
19    << "\n\tW--数据按两种格式分别存盘(W)"
20    << "\n\tT--从文本文件中读取数据(T)"
21    << "\n\tB--从二进制文件中读取数据(B)"
22    << "\n\t0--退出([ESC],Q)" << endl;
23    return Pause("\t请选择: "); // 接收选择
24 }
25
26 void InsertNodes(Player *&head)
27 {
28    int n, k;
29    Player x, *p;
30    cout << "添加若干个结点" << endl;
31    for(n=0; n<20; n++)
32    {
33        x.num = rand()%99 + 1;
34        p = Search(head, x.num);
35        if(p==NULL)
36        {
37            // C 语言风格的语句
38            sprintf(x.name, "%c%05d", rand()%26+'A', x.num);
39            // 以大写字母开头后跟 5 位数码 (与编号相同, 不足补 0)
40            for(k=0; k<REFEREES; k++)
41                x.x[k] = (80 + rand() % 21)/10.0; // 准备数据
42            Insert(head, &x);
43        }
44        else
45        {
46            Show(p);
47            cout << "编号为" << x.num << "的结点已经存在。" << endl;
48        }
49    }
50 }
51
52 void InsertNode(Player *&head)
53 {
54    int k;
55    char str[80];
56    Player x, *p;
57
58    cout << "插入一个结点, 请输入新编号: ";
59    cin >> x.num;
60    p = Search(head, x.num);
61    if(p==NULL)
62    {

```

```

63         cout << "请输入姓名: ";
64         cin.getline(str, 80);    // “吃掉”前面输入的换行字符
65         cin.sync();           //或刷新输入缓冲区
66         cin.getline(x.name, sizeof(x.name));
67         for(k=0; k<REFEREES; k++)
68             x.x[k] = (80 + rand() % 21)/10.0;
69         Insert(head, &x);
70     }
71     else
72     {
73         Show(p);
74         cout << "编号为" << x.num << "的结点已经存在。" << endl;
75     }
76 }
77
78 void DeleteNode(Player *&head)
79 {
80     int k, num;
81     cout << "删除一个结点, 请输入其编号: ";
82     cin >> num;
83     k = Delete(head, num);
84     if(k==-3)
85         cout << "链表为空。" << endl;
86     else if(k==-2)
87         cout << "查无此人。" << endl;
88     else if(k==-1)
89         cout << "该结点已经被删除。" << endl;
90     else
91         cout << "该结点未被删除。" << endl;
92 }
93
94 void SearchNode(Player *head)
95 {
96     int num;
97     Player *p;
98
99     cout << "查询结点, 请输入编号: ";
100    cin >> num;
101    p = Search(head, num);
102    if(p)
103        Show(p);
104    else
105        cout << "查无此人。" << endl;
106 }
107
108 void SearchRank(Player *head)
109 {
110     int n, k;
111     Player *p;
112
113     cout << "查询/继续查询"
114         << " (因为可能存在并列名次或空缺名次) \n"
115         << "请输入欲查询的名次: ";
116     cin >> k;
117     n = 0;
118     p = Locate(head, k, 1);
119     while(p)
120     {
121         n++;

```

```

122     Show(p);
123     p = Locate(head, k);
124 }
125 switch(n)
126 {
127     case 0: cout << "无第" << k << "名" << endl; break;
128     case 1: break;
129     default: cout << "共有" << n << "人并列第"
130                 << k << "名" << endl; break;
131 }
132 }
133
134 void ModifyNode(Player *head)
135 {
136     int num;
137     char str[80];
138     cout << "修改结点数据, 请输入其编号: ";
139     cin >> num;
140     cin.getline(str, 80); // “吃掉” 换行字符
141     Modify(head, num);
142 }
143
144 void Sort(Player *&head)
145 {
146     int choice = Pause("\n按编号[I]按姓名[N]按名次[R]:");
147     if(choice=='i' || choice=='I')
148         SortList(head, SORT_NUM);
149     else if(choice=='n' || choice=='N')
150         SortList(head, SORT_NAME);
151     else
152         SortList(head, SORT_RANK);
153 }
154
155 void Write2File(Player *head)
156 {
157     int n;
158
159     n = TSave(head, "Contest.txt");
160     cout << "共" << n << "行记录写入文件Contest.txt\n"
161             << "该文件可用Windows的记事本打开。" << endl;
162
163     n = BSave(head, "Contest.dat");
164     cout << "共" << n << "个结点写入文件Contest.dat\n"
165             << "该文件只能用本程序有意义地打开。" << endl;
166 }
167
168 int main()
169 {
170     int choice = 1, n, yn;
171     Player *head = NULL;           // 创建一条空链表
172
173     time_t t;
174     srand(time(&t));
175     while(choice)
176     {
177         choice = Menu();          // 输出菜单, 返回输入的选项
178
179         switch(choice)          // 分情况处理
180         {

```

```

181     case '1':
182     case 'n':
183     case 'N': InsertNodes(head); break;
184     case '2':
185     case 'i':
186     case 'I': InsertNode(head); break;
187     case '3':
188     case 'd':
189     case 'D': DeleteNode(head); break;
190     case '4':
191     case 'v':
192     case 'V': ShowList(head); break;
193     case '5':
194     case 'l':
195     case 'L': SearchNode(head); break;
196     case '6':
197     case 'r':
198     case 'R': SearchRank(head); break;
199     case '7':
200     case 'm':
201     case 'M': ModifyNode(head); break;
202     case '8':
203     case 'f':
204     case 'F': yn = Pause("释放所有结点, 是否继续(y/[N]):");
205             if(yn=='y' || yn=='Y') FreeList(head);
206             break;
207     case '9':
208     case 's':
209     case 'S': Sort(head);      break;
210     case 'w':
211     case 'W': Write2File(head); break;
212     case 't':
213     case 'T': n = TLoad(head, "Contest.txt");
214             cout << "共读取" << n << "行记录。" << endl;
215             break;
216     case 'b':
217     case 'B': n = BLoad(head, "Contest.dat");
218             cout << "共读取" << n << "个结点。" << endl;
219             break;
220     case '0':
221     case 'q':
222     case 'Q':
223     case 27: yn = Pause("退出本系统吗? (y/[N]):");
224             if(yn=='y' || yn=='Y') choice = 0;
225         }
226     }
227     FreeList(head);
228     return 0;
229 }
```

8.4 小 结

结构体（struct）是一种自定义的数据类型，它声明了数据的构成，即描述了数据的组织形式。作为一种自定义的数据类型名并不占用内存空间，只有当用这种数据类型定义变量（亦称为创建对象）或定义数组、申请堆变量、申请堆数组时才分配内存空间。结构体变量（对象）能和基本数据类型变量一样在函数中传递、返回。

其实，C++ 结构体（**struct**）与类（**class**）几乎没有区别。本章只涉及结构体的数据成员，免去成员函数等许多概念。本章的程序中假设评委人数是事先确定的常量，不能在程序执行时修改。对进一步允许评委人数可变的情形，我们将在面向对象程序设计中讨论。

本章讨论的单向链表是最基本的，在数据结构课程中将以此为基础进一步讨论双向链表、队列、栈和树等数据结构。

练习 8

1. 指出下列各程序或片段中的错误

(1)

```
1 struct Student
2 {
3     char id[9], name[9];
4     double score;
5 }
```

(2)

```
1 struct Point
2 {
3     double x=0, y=0;
4 };
```

(3)

```
1 struct Person
2 {
3     char name[9];
4     int age;
5 };
6
7 #include <cstring>
8
9 int main()
10 {
11     Person x, *p;
12     x.name = "张三";
13     x.age = 18;
14     strcpy(p->name, "李四");
15     p->age = 19;
16     // ...
17     return 0;
18 }
```

(4)

```
1 struct Node
2 {
3     double x;
4     Node *next;
5 };
6
7 Node *AddNode(Node *head, double x)
```

```

8  {
9      Node *p = new Node;
10     p->x = x;
11     p->next = head;
12     head = p;
13     return head;
14 }
15
16 int main()
17 {
18     Node *head;
19     AddNode(head, 3);
20     AddNode(head, 2);
21     AddNode(head, 1);
22     return 0;
23 }
```

(5)

```

1 #include <iostream>
2 using namespace std;
3
4 struct Node
5 {
6     double x;
7     Node *next;
8 };
9
10 void ShowList(Node *&head)
11 {
12     cout << "head";
13     while(head)
14     {
15         cout << "->" << x;
16         head = head->next;
17     }
18     cout << "->NULL" << endl;
19 }
20
21 int main()
22 {
23     Node *head=NULL, *p;
24     head = new Node;
25     p->x = 1.1;
26     head->next = p = new Node;
27     p->x = 2.2;
28     p->next=NULL;
29     ShowList(head);
30     ShowList(head);
31     FreeList(head); // 假设已经定义了FreeList 函数
32     return 0;
33 }
```

(6)

```

1 #include <iostream>
2 using namespace std;
3
```

```

4 struct Node
5 {
6     int x;
7     Node *next;
8 };
9
10 void Create(Node *head)
11 {
12     Node *p, *pEnd;
13     int x;
14     head = NULL;
15     while(true)
16     {
17         cout << "创建链表。请输入一个数(输入0结束)：" ;
18         cin >> x;
19         if(x==0) break;
20         p = new Node;
21         p->x = x;
22         if(head==NULL)
23             head = p;
24         else
25             pEnd->next = p;
26         pEnd = p;
27     }
28 }

```

(7)

```

1 struct Node
2 {
3     int x;
4     Node *next;
5 };
6
7 template <typename T> void FreeList(T *head)
8 {
9     T *temp;
10    while(head)
11    {
12        temp = head;
13        head = head->next;
14        delete temp;
15    }
16 }
17
18 int main()
19 {
20     Node *head=new Node;
21     head->x = 1; head->next = NULL;
22     FreeList(head);
23     FreeList(head);
24     return 0;
25 }

```

(8)

```

1 struct Node
2 {

```

```

3     int x;
4     Node *next;
5 };
6
7 void Insert(Node *&head, int x)
8 {
9     Node *p;
10    if(x < head->x)
11    {
12        p = new Node;
13        p->x = x;
14        p->next = head;
15        head = p;
16        return;
17    }
18    // ...
19 }
```

(9)

```

1 struct Node
2 {
3     int x;
4     Node *next;
5 };
6
7 void Insert(Node *&head, int x)
8 {
9     Node *p, *pGuard;
10    if(head==NULL || x < head->x)
11    {
12        p = new Node;
13        p->x = x;
14        p->next = head;
15        head = p;
16        return;
17    }
18    for(pGuard=head; pGuard->next->x > x; pGuard=pGuard->next)
19    ;
20    // ...
21 }
```

(10)

```

1 struct Node
2 {
3     int x;
4     Node *next;
5 };
6
7 void Delete(Node *&head, int x)
8 {
9     Node *p;
10    if(x == head->x)
11    {
12        p = head;
13        head = head->next;
14        delete p;
```

```
15         return;  
16     }  
17     // ...  
18 }
```

2. 基本题

(1) 链表综合练习。编写一个“××管理系统”，内容不限。要求有插入结点、删除结点、修改结点数据、输出结点数据、输出链表所有结点数据、释放所有结点等功能；要求在主函数中设置循环以反复对链表进行操作。

(2) 编写一个函数统计“评委评分”程序中最后得分不低于 9.50 分的选手人数。

(3) 编写一个函数模板，实现链表结点倒置。要求不移动结点在内存中的位置。假设结点的结构中有指向本类对象的指针 `next`。提示（拆旧链、造新链）：

① 先将原首结点做成单结点链表（新链表），同时将原第二个结点起的链表称为“旧链表”。

② 将新旧两条链表分别看成两个栈，始终在它们的栈顶操作，将从旧链表退栈的结点（不能释放）压入新链表。

【提示】 想象某教师批改学生的作业本。取未批改的作业本（最上面）一本进行批改，批改后放置到已批改的作业本（最上面）。批改结束时，学生的作业本倒序。

(4) 设有单向链表（结点的数据域为一个整型数），编写一个函数对其进行排序（按结点数据域的数值从小到大）。提示：按照“拆旧链、造新链”的方法，将旧链表中退栈的结点（即首结点）插入到新链表的适当位置处。

(5) 设有两条同类单向链表（结点的数据域为一个整型数），首先用上题的方法将它们排序，然后合并这两条链表（并保证各结点有序）。

(6) 设有单向链表（结点的数据域为一个整型数），编写一个函数消去链表中数据域数据重复的结点。即，若链表中结点数据域数据等于 10 的结点有三个，要求删除后两个结点数据域数据为 10 的结点，保留前一个结点。

第三部分

面向对象程序设计

第9章 类与类的对象

从本章起至第 12 章，将围绕类（class）的基本特征——封装，以及由此而引出的新问题和解决这些新问题的新机制展开讨论。学习这些内容时应重点关注两个方面^[1]：问题的提出（理解为什么）；语法规规定（掌握怎样用）。读者将时刻体会到程序设计高级语言的高度抽象性，以及 C++ 语言的“将方便让给程序员，把困难留给编译器”的设计思想。

封装是对数据及其操作方法进行包裹；更重要的是，封装还意味着某种程度上的内部细节的屏蔽、外部联络接口的公开。

在第 8 章中，我们用结构体对一些紧密相关的、数据类型不尽相同的、集中描述某一事物多种属性的数据进行了组织。结构体中的数据被称为数据成员。事实上，常量、变量、数组、指针变量和引用均可以作为数据成员。这样的组织方式便是声明了一种新的数据类型，具有一定的封装作用。

显然，对于描述某具体事物的数据，其操作具有其特定性；反之，一种特定的操作往往是针对描述某种特定事物属性的数据进行的。总之，数据及其加工方法具有内在的必然联系。因此，数据的组织及其基本的加工方法有必要作为一个整体加以封装。公式对象 = (算法 + 数据结构) 中的括号就是封装的意思。

在第 8 章中，为简单起见，在结构体中仅对数据进行了包装。利用这种新数据类型创建对象（即定义变量）、创建对象数组、创建堆对象和创建堆数组均意味着分配内存空间，由所分配的内存空间承载对象的具体属性（对象的数据成员值）。对这种数据类型的对象进行操作（如对其数据成员进行输入输出、基本运算等），以及对由结构体构造的新数据结构——链表的操作（如输入输出、基本运算等）都通过外部函数进行。亦即这些数据及其操作并没有语法意义上的封装，数据及其操作的联系是通过程序员外在地维持的。所编写的应用程序带有明显的 C 语言风格。

另一方面，第 8 章所讨论的结构体虽对数据进行了包装，但未对数据进行有效地封闭，在程序中可直接操作结构体变量的数据成员。即未对数据进行隐藏和保护处理，此时对象的数据域是一个不设防的开放空间。

C++ 的类及 C++ 的结构体对 C 语言的结构体进行了变革式地扩充。C++ 的类中可以包含数据（被称为数据成员）和函数（被称为成员函数，或者方法）。数据成员和成员函数统称为成员。数据成员反映事物的属性，成员函数体现事物的行为。C++ 的类中可以通过设置其成员的不同访问属性实现切实有效地封装：屏蔽或隐藏内部细节，公开外部接口。

从设计到使用一个类，需要如下“三部曲”（三个步骤）：

① **类的声明**，即类的设计或描述。对某一类事物的属性和行为进行抽象地描述：描述数据的组织形式、描述数据可能的加工方法。

^[1]至于编译系统是如何具体处理相应问题的，因超出本课程的范围而不予讨论。

② 创建类的对象。建立类的具体实例，占用内存空间，承载具体数据。

③ 传递消息给对象。令对象自己表现自己，即通过对象调用成员函数，表现对象的行为，实现具体问题求解。

9.1 类的声明

将类的数据成员组织形式的描述、成员函数的原型声明和成员函数的定义统称为类的声明（或称为类的设计）。声明类的成员的同时，需要明确该成员的访问属性。

例 9.1 时间（Time）的描述及处理方法。

(1) 属性：时（h）、分（m）和秒（s）。

(2) 行为：调整时间（设置新时间）、获取时间和输出时间。

显然，时间的“时、分、秒”属性对于一个具体的时间来说至关重要，需要对它们加以保护——进行封装且对外屏蔽。具体地，只允许该类的一些方法（成员函数）有权对h、m和s进行访问（读或写），不能通过其他方式直接对h、m和s进行操作。就像我们应该通过手表上提供的外部接口（旋钮）调整时间，而不要打开手表的面盖直接转动时针、分针、秒针一样。

```

1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 class Time
6 {
7 public: // 公开的（访问属性）
8     Time(int hour=0, int minute=0, int second=0) // 构造函数
9     {
10         h = hour; m = minute; s = second;
11     }
12     void SetTime(int hour, int minute, int second) // 设置时间
13     {
14         h = hour; m = minute; s = second;
15     }
16     int GetHour() { return h; } // 获取“时”的处理方法
17     int GetMinute() { return m; } // 获取“分”的处理方法
18     int GetSecond() { return s; } // 获取“秒”的处理方法
19     void Show() // 输出时间的方法
20     {
21         cout << setfill('0') << setw(2) << h
22             << ":" << setw(2) << m
23             << ":" << setw(2) << s
24             << setfill('0') << endl;
25     }
26
27 private: // 私有的（访问属性）
28     int h, m, s; // 数据成员
29 };

```

上述程序段完成了对新数据类型Time的抽象描述。其中有三个私有的整型数据成员和六个公开的成员函数。关于其中的构造函数将在第10章进行专门讨论。

9.1.1 成员的访问控制

上面的程序段中 `class` 为标识类声明的保留字。数据成员或成员函数的访问控制设定通过成员的访问属性设置实现。成员的访问属性有三种，它们分别用三个保留字 `public`（公开的）、`protected`（受保护的）和 `private`（私有的）标识。

C++ 的结构体与类的唯一区别是：`struct` 成员的访问属性默认为 `public`，`class` 成员的访问属性默认为 `private`。类声明中，成员的访问属性可以出现在任何成员前，且对成员连续有效，直至遇到新的访问属性设置。属性设置的次序、次数不限。有时将数据成员放在成员函数之前声明，有时则相反。

使用一个已经设计完好的类时，仅需要了解和掌握（通常，也只能了解）该类的具有 `public` 访问属性的成员。就好比钟表，调整时间的功能旋钮、显示时间值的钟面读数是钟表设计者留给用户进行输入输出操作的外部接口。钟表内部的各种子功能的实现及使用不必用户了解，也不必由用户来直接操作。因此，会操作旋钮、会读取钟表所指示的时间值便是会使用钟表的人了。

上例中，成员函数的访问属性是 `public` 的，是类的外部接口，其“外观”设计比较清晰、美观——成员函数形式参数名称的含义清楚。相比之下，数据成员的访问属性是 `private` 的，并不直接对外，数据成员的名称设计就可以逊色一些。这样，将方便让给类的使用者，将“不便”留给类的设计者自己。

类的设计应该遵循一定的原则，以使类的功能齐全，外部接口简单清晰、美观，使用方便。一般地，只将必须对外的部分设计成公开的；不必对外的成员尽量设计成受保护的或者私有的。称属性为 `protected` 或者 `private` 的成员为内部的。

9.1.2 数据成员和成员函数

类的成员有数据成员和成员函数之分。类的声明完成对数据组织形式的抽象描述，完成对数据可能的加工方法的抽象描述。

数据成员的访问属性可以是公开的、受保护的或私有的。通常，将数据成员设置成 `protected` 或 `private` 以保护类的对象的属性。

成员函数的访问（被调用）属性可以是公开的、受保护的或私有的。访问属性为 `protected` 或 `private` 的成员函数为内部的（参见练习 9 的 2(2) 中成员函数 `Delta`），仅限类的其他成员函数或友元函数（参见第 11.2.1 小节）调用。类的成员函数对同类的所有对象的所有数据成员具有无限制的访问能力^[1]。

成员函数可以在类体中定义，同时具有函数声明的作用。在类体中定义的成员函数含默认的 `inline` 建议。通常，在类体中用成员函数原型进行函数声明，而在类体外完成成员函数定义。在类体外定义成员函数时，必须使用函数全名“名字空间名`::`类名`::`成员函数名”，显式地指明成员函数所在的类，类所在的名字空间（参见练习 9 的 2(3)）。若无名字空间，则仅需要指明类名，其中的双冒号“`::`”为作用域区分符。通常将成员函数的定义写在某源程序文件中分割编译，以实现对具体操作方法的隐藏（参见第 9.4.2 小节）。

^[1] C++ 的这项规定是合理、必要的。后续章节（如第 9.3.1 小节、第 10.3 节、第 10.4 节和第 11.1 节）对该规定有直接的依赖。但由继承而来的基类的私有成员除外（参见第 13.1.3 节）。

例 9.1 中，类 `Time` 中有一个特殊的成员函数：它没有返回类型（函数体内可以使用 `return`; 语句），函数名与类名相同。这个函数被称为构造函数，在创建具体对象时由系统自动调用它。从该函数原型可见，其三个参数均带默认值。关于构造函数的详细讨论将在第 10 章进行。

9.2 创建类的对象

类的声明（即设计）是描述性的、抽象的：数据的组织形式、数据可能的操作方法。简单地说，声明一个类就构造了一种新的数据类型，同时提供了对这种数据类型的一些操作方法。从数据类型的角度上看，这种新的数据类型是对基本数据类型的补充，可能提供给其他程序员使用。

既然所声明的类是一种数据类型，便可以用这种数据类型定义变量——用面向对象程序设计的语言称为创建对象。创建对象意味着为所创建的对象分配内存空间。只有分配了内存空间才能够承载具体的数据，才能进行实际的数据处理。

9.2.1 创建对象

首先回顾基本数据类型 `int`。我们知道，类型名 `int` 所描述的是一定范围内的整数在内存中的存放形式，以及一定的运算操作。类型名 `int` 本身并不占用内存空间，只有用它定义了变量、数组后，这些变量、数组元素才占用内存空间，每一个 `int` 型的变量占用 `sizeof(int)` 字节。

与之相仿，例 9.1 所声明的类 `Time` 的类名并不占用内存空间。只有用它创建了对象、创建了对象数组后，这些对象、对象数组的元素才占用内存空间，每一个 `Time` 型的对象占用 `sizeof(Time)` 字节。

```

int n=0, m(10);           // 定义两个整型变量，各分别占 sizeof(int) 字节
                          // 初始化 n 为 0，用另一种语法格式初始化 m 为 10
Time t0, t1(10), t2(10, 20), t3(10, 20, 30);      // 创建四个对象，各分别占 sizeof(Time) 字节空间
Time tArray[10];          // 创建有十个元素的对象数组，                                         // 占用连续的 10*sizeof(Time) 字节内存空间
Time *tPtr = &t1;          // 定义一个指针变量，指针占 sizeof(void*) 字节，                                // 并未创建新的 Time 对象
Time &tRef = t0;          // 声明一个引用（对象的别名），不占用内存空间，                                // 并未创建新的 Time 对象
tPtr = new Time;           // new 操作创建一个动态对象（堆对象）
delete tPtr;               // delete 操作释放一个动态对象
tPtr = new Time [m];       // 创建有 m 个元素的动态对象数组（堆对象数组）
delete [] tPtr;            // 释放动态对象数组

```

可以创建全局对象、静态全局对象、静态局部对象、局部自动对象和动态对象（堆对象）；可以定义类类型的指针变量，声明类类型对象的引用；可以将类对象的值（右值）、地址值、对象本身（左值）作为实参或函数返回值使类的对象的值或对象本身在函数之间传递。关于对象的生命期、作用域和可见性等，均与基本数据类型的规定相同。请读者体会。

我们还希望将类类型设计成像基本数据类型一样的可以方便地进行初始化、赋值运

算、关系运算、可能的算术运算和 I/O 操作等。例如，系统所提供的字符串类 `class string;` 就基本上“与基本数据类型看齐了”。有关这些内容的讨论将在后续的各章中逐步展开。

9.2.2 对象的基本空间

创建对象意味着给对象分配内存空间。例 9.1 所设计的类 `Time` 的每一个对象所占用的内存空间为 `sizeof(Time)` 字节，具体地为 $12 = 3 * sizeof(int)$ 字节。例如：

```
Time t; // 创建一个对象
cout << sizeof(t) << endl; // 输出 12, 因为这里 sizeof(int) 为 4
```

可见，对象 `t` 的空间是三个数据成员所必需的空间。我们将对象的非静态^[1]数据成员所占用的空间总和称为对象的基本空间。不同对象的基本空间在内存中的起始地址是不同的，但其大小（字节数）是相同的。对象的基本空间占 `sizeof(类型名或对象名)` 字节。

由上述结果还可见，对象的成员函数并不占用对象的空间。系统将对象的成员函数代码与对象的数据分离，存放至内存的代码区，使之成为类的成员函数。成员函数为该类的所有对象共享。这样的处理显然能够节省宝贵的内存空间。

9.3 对象的自我表现

对象的自我表现体现在执行类的成员函数。以下代码接例 9.1。

```
30 int main()
31 {
32     Time t1, t2(10, 20, 30), *p=&t1; // 创建两个对象
33     t1.Show();
34     t1.SetTime(8, 30, 0); // 对象直接访问(调用)成员函数
35     p->Show(); // 指针变量间接访问成员函数
36     t2.Show();
37     return 0;
38 }
```

程序的运行结果：

```
00:00:00
08:30:00
10:20:30
```

称对象调用其成员函数为发送消息给对象，令对象自己表现自己。如语句 `t1.Show()`；及语句 `t1.SetTime(8,30,0);`，均为发送消息给对象 `t1`，令对象 `t1` 表现自己（输出对象 `t1` 时间值，调整、修改对象 `t1` 的时间值）。称语句 `t2.Show()`；是将消息发送给对象 `t2`，令对象 `t2` 表现自己（输出时间值）；语句 `p->Show();` 等价于 `(*p).Show();`，为发送消息给指针变量 `p` 的目标变量（目标对象）`*p`，此处即为对象 `t1`，因此语句 `p->Show();` 等价于 `t1.Show();`。

^[1]关于静态数据成员参见第 11 章。

9.3.1 this 指针常量

前面提到类的成员函数为该类的所有对象共享，而不同的对象有各自的对象空间。作为“身处代码区”的成员函数（如成员函数 void Time::Show();）是如何区分具体的对象，操作不同对象的数据空间的呢？

原来，C++ 编译器悄悄地为所有的非静态成员函数添加了一个隐含的形式参数，这个隐含的参数的数据类型为该类的指针常量^[1]，参数名为 C++ 保留字 this，意味着 this 的指向被“锁定”不能更改。并且在该成员函数体中访问数据成员或成员函数时在成员名前隐含地添加了 this->（意指本对象的）。成员函数的定义相当于（下面用加下划线的斜体字表示编译系统悄悄添加的隐含部分）：

```
Time::Time( Time * const this, int hour, int minute, int second)
{
    this->h = hour;
    this->m = minute;
    this->s = second;
}

void Time::SetTime( Time * const this,
                     int hour, int minute, int second)
{
    this->h = hour;
    this->m = minute;
    this->s = second;
}

int Time::GetHour ( Time * const this) { return this->h; }
int Time::GetMinute( Time * const this) { return this->m; }
int Time::GetSecond( Time * const this) { return this->s; }
void Time::Show( Time * const this)
{
    cout << setfill('0') << setw(2) << this->h
        << ":" << setw(2) << this->m
        << ":" << setw(2) << this->s
        << setfill(' ') << endl;
}
```

而在具体对象调用该类的非静态成员函数时，编译器又悄悄地将该对象的地址作为第一个实参传递给成员函数。亦即相当于：

```
int main()
{
    Time t1, t2(10, 20, 30), *p = &t1;
    t1.Show(&t1);           // this 指针常量锁定对象 t1
    t1.SetTime(&t1, 8, 30, 0);
    p->Show(p);           // this 指针常量锁定 p 的目标，即对象 t1
    t2.Show(&t2);           // this 指针常量锁定对象 t2
    return 0;
}
```

^[1]指向不能被改变（即目标固定）的指针为指针常量（如 int x=3; int * const p = &x;），指针常量的目标须为变量。常量指针常量的目标固定（如：const int * const p = &x; 或 int const * const p = &x;），且将目标视为常量。

这样，由于提供了具体对象的起始地址，类的成员函数虽不跟随对象数据而“身处代码区”，仍能正确而有效地访问具体对象的数据空间。自然地，要求成员函数对本类所有对象的所有成员具有无限制的访问能力（参见第 9.1.2 小节的注）。在后续的章节中，将看到显式地使用 `this` 指针的情形。

9.3.2 常量成员函数

有一些成员函数（如返回或输出对象的属性值函数）对于对象的属性只进行读访问，不进行写操作。在这种成员函数体内，仅靠程序员“自觉遵守”保证不修改对象的属性是不够的。C++ 提供了这个问题的语法上的保证：将成员函数声明成常量成员函数。

常量成员函数是类的成员函数，这种函数的隐含形式参数为指向本对象的常量指针常量（`const * const this`），故在函数体内只能读取但不能修改本对象的任何属性^[1]。设计这种函数的语法格式是在所声明及定义的成员函数首部后添加保留字 `const`。

例如，在类体中声明常量成员函数格式为 `int GetHour() const;`，在类体外定义常量成员函数的格式为

```
1 int Time::GetHour() const // 定义常量成员函数。隐含参数为
2 {                         // 常量指针常量 const Time * const this
3     return h;
4 }
```

常量成员函数的主要作用是处理常量对象。应用程序中直接定义常量对象并不多见，最常见的是出现在函数形式参数位置上的。

- ① 对象的常量引用（`const Time &t`）。
- ② 指向常量对象的指针变量（`const Time *p`）。

例如，定义如下外部函数：

```
void Print(const Time &t, const Time *p)
{
    t.Show();
    p->Show();
}
```

若类 `Time` 的成员函数 `Show` 为非常量成员函数，则上述函数编译时出错：认为使用了常量对象（实为对象的常量引用）`t` 的地址初始化非常量成员函数 `Show` 的隐含指针常量 `this`；认为使用了指向常量对象的指针 `p` 初始化了非常量成员函数 `Show` 的隐含指针常量 `this`。这是编译器不允许的，哪怕调用 `Print` 函数的实参对象确实不是常量对象。

C++ 可以将变量视为常量，但不能将常量视为变量^[2]。关于对象，不能给常量对象发消息命其调用非常量成员函数。否则，这将意味着用常量对象的地址初始化指向非常量对象的指针常量 `this`。常量对象只能调用常量成员函数。

读者可能会问，为什么要在函数的形式参数中用 `const` 限制，即为什么不将函数设计成 `void Print(Time &t, Time *p);` 呢？其原因是：调用该函数时，实参通过引用型

[1] 常量成员函数可以修改类的对象以外的属性。如可以修改类的静态数据成员的值，参见第 12.2 节源代码 12.2 中的“`int Date::DaysOfYear() const;`”函数。

[2] 一个例外的情形参见第 5.7.2 小节的注。

(双向传递)、指针型(间接双向传递)形式参数传递后，在函数体内存在被修改的危险。其他程序员仅看到函数原型，不能确认函数体是否修改了实参对象的值时，是不敢大胆使用该函数的，担心传入的对象被函数体语句修改。而大多数情况下，其他程序员也只能看到函数原型，不一定能看到函数体语句。另一方面，有时可能确实会用常量(常对象)做实参调用函数。

那么，读者可能会再问，为什么要采用引用型、指针型形式参数呢，即为什么不将函数设计成 void Print(Time t1, Time t2); 呢？的确，仅将实际对象的值(单向)传递给函数时将创建局部对象(实参对象的副本)，在函数体内无论怎样操作对象的副本均不影响实际对象。然而，创建对象的副本可能是一项费时间、耗空间的操作，需要通过拷贝构造(参见第 10.3 节)，是程序中应该尽量回避的。

将类中所有不修改对象属性的成员函数都设计成常量成员函数总是有好处的。在类 Time 中，将成员函数 Show 设计成常量成员函数，则上述 void Print(const Time &t, const Time *p); 函数便能通过编译。该类中的 GetHour 函数、GetMinute 函数和 GetSecond 函数最好也设计成常量成员函数。

由此可见，C++ 的机制具有强烈的应用需要，许多机制是从时间效率、空间效率和数据的安全性方面考虑的。希望读者能够将相关的知识点融会贯通。

9.4 封装与隐藏

9.4.1 屏蔽类的内部实现

不管钟表内部结构如何设计和改变，甚至由机械装置更换成电子芯片，只要钟表上调整时间的旋钮等外部接口的功能及使用方法不变，对于钟表的用户而言是差别不大的(或者认为没有差别)。当然，在保留原有的接口功能及使用方法不变的前提下，还可以新增若干功能接口。

软件设计及其升级维护的情形也是类似的。在应用软件升级中可以修改类的声明，增加一些新的功能。为了与旧系统保持兼容，常常需要保持类的原有外部接口不变。下面的例子是一个类的两种不同设计方案，其数据成员的含义不同，但是它们有着相同的外部接口，对那些基于该类进行二次开发的用户来说，它们是相同的。另一方面，若作为基础的类在保持原有接口不变的情况下增加了新的功能时，则应用软件仅需要增加对新功能的处理，原有的部分不变。这将使软件维护、升级的成本较低。

声明一个类已经将数据以及对数据的操作包裹在一起了。类的设计还要讲究功能齐全，接口清晰、简单。利用成员的访问属性控制可达到屏蔽甚至隐藏某些不必公开的部分，使所设计的类的外观简洁、使用方便。

例 9.2 复数类(Complex)的描述及处理方法。

(1) 属性：复数的实部(real)及虚部(imag)；或复数的模(radius)及复角(angle)。

(2) 行为：修改复数(设置新的复数值)，计算复数的实部、虚部、模和复角，输出复数。

我们知道，复数有一般表示、三角函数表示和指数表示多种形式。记 $i = \sqrt{-1}$ ，则

$$a + ib = r(\cos \theta + i \sin \theta) = re^{i\theta}$$

其中

$$a = r \cos \theta, \quad b = r \sin \theta$$

或者

$$r = \sqrt{a^2 + b^2}, \quad \theta = \arctan(b/a)$$

设计方案之一：

```

1 // Complex.h
2 #ifndef COMPLEX_H
3 #define COMPLEX_H
4 #include <iostream>
5 #include <cmath>
6 using namespace std;
7
8 class Complex           // 声明一个复数类
9 {
10 public:                 // 公开的
11     Complex(double real=0, double imag=0) // 构造函数
12     {
13         re = real; im = imag;
14     }
15
16     double Real() const { return re; }      // 返回复数的实部的值
17     double Imag() const { return im; }      // 返回复数的虚部的值
18
19     double Radius() const {return sqrt(re*re+im*im);} // 返回模
20     double Angle() const { return atan2(im, re); } // 返回复角
21
22     void Set1(double real, double imag) // 设置复数的实部及虚部
23     {
24         re = real; im = imag;
25     }
26     void Set2(double radius, double angle) // 设置复数的模与复角
27     {
28         re = radius * cos(angle);
29         im = radius * sin(angle);
30     }
31
32     void Show1() const                // 按一般形式输出复数
33     {
34         cout << re;
35         if(im > 0)      cout << " + " << im << "i";
36         else if(im < 0) cout << " - " << -im << "i";
37     }
38     void Show2() const                // 按指数形式输出复数
39     {
40         double r, angle;
41         r = sqrt(re*re + im*im);
42         angle = atan2(im, re);
43         if(r > 0) cout << r << " * exp(" << angle << "i)";
44         else       cout << 0;
45     }
46 private:                  // 私有的
47     double re, im;             // 数据成员 (实部, 虚部)
48 };
49 #endif

```

设计方案之二：

```

1 // Complex.h
2 #ifndef COMPLEX_H
3 #define COMPLEX_H
4 #include <iostream>
5 #include <cmath>
6 using namespace std;
7
8 class Complex           // 声明一个复数类
9 {
10 public:                // 公开的
11     Complex(double real=0, double imag=0) // 构造函数
12     {
13         _radius = sqrt(real*real + imag*imag);
14         _angle = atan2(imag, real);
15     }
16     double Real() const { return _radius*cos(_angle); }          // 返回复数的实部
17
18     double Imag() const { return _radius*sin(_angle); }          // 返回复数的虚部
19
20     double Radius() const {return _radius;} // 返回复数的模
21     double Angle() const {return _angle;} // 返回复数的复角
22
23     void Set1(double real, double imag)    // 设置实部及虚部
24     {
25         _radius = sqrt(real*real + imag*imag);
26         _angle = atan2(imag, real);
27     }
28     void Set2(double radius, double angle) // 设置复数的模与复角
29     {
30         _radius = radius; _angle = angle;
31     }
32     void Show1() const                  // 按一般形式输出复数
33     {
34         double re, im;
35         re = _radius * cos(_angle);
36         im = _radius * sin(_angle);
37         cout << re;
38         if(im > 0)      cout << "+" << im << "i";
39         else if(im < 0) cout << "-" << -im << "i";
40     }
41     void Show2() const                  // 按指数形式输出复数
42     {
43         if(_radius>0) cout << _radius << "e^(" << _angle << "i)";
44         else           cout << 0;
45     }
46 private:                 // 私有的
47     double _radius, _angle;           // 数据成员（模，复角）
48 };
49 #endif

```

在这两种设计方案中，作为内部的（私有的）数据成员是不同的，成员函数的定义也不同。然而，这两种设计方案中的外部接口：公开的成员——成员函数的函数名、函数的形式参数类型、函数的返回类型、函数的功能均相同。因此，对于类 Complex 的使用者而言是不必区分这两种设计方案的。请读者设计程序分别测试上述两个类。

9.4.2 隐藏类的内部实现

既然类的内部实现方案可以有多种方式，而且类的内部实现不必让使用该类的其他程序员了解。更进一步，为了保护知识产权、保护类实现中的技术秘密，我们需要将类的内部实现与类的声明相分离，再进一步将成员函数的定义隐藏起来（参见第 7.4 节）。以复数类的第一种设计方案为例。

1. 分离类的声明与成员函数定义

采用多文件结构，编写一个头文件和一个源程序文件如下。

```

1 // Complex.h          // 类体声明头文件
2 #ifndef COMPLEX_H
3 #define COMPLEX_H
4                         // 无须包含任何头文件，亦不需声明使用某名字空间
5 class Complex          // 声明复数类
6 {
7 public:                // 公开的
8     Complex(double real=0, double imag=0); // 构造函数
9     double Real() const;           // 返回复数的实部的值
10    double Imag() const;           // 返回复数的虚部的值
11    double Radius() const;         // 返回复数的模
12    double Angle() const;          // 返回复数的复角
13    void Set1(double real, double imag); // 设置复数的实部及虚部
14    void Set2(double radius, double angle); // 设置复数的模与复角
15    void Show1() const;            // 按一般形式输出复数
16    void Show2() const;            // 按指数形式输出复数
17
18 private:               // 私有的
19     double re, im;             // 数据成员
20 };
21 #endif

```

头文件中的注释是供其他程序员参考的，最好另写详细的说明文档同时提交给其他程序员。而下面的源程序文件不一定直接提供给其他程序员，其中的注释是该类的开发者为方便自己以后维护而编写的。

```

1 // Complex.cpp        类 Complex 的内部实现源程序文件
2 #include <iostream>
3 #include <cmath>
4 #include "Complex.h"           // 包含头文件
5 using namespace std;
6
7 Complex::Complex(double real, double imag)
8 {                           // 构造函数。注意参数默认值只能出现在函数声明中
9     re = real; im = imag;
10 }
11
12 double Complex::Real() const { return re; } // 返回复数的实部的值
13 double Complex::Imag() const { return im; } // 返回复数的虚部的值
14 double Complex::Radius() const { return sqrt(re*re+im*im); } // 返回复数的模
15 double Complex::Angle() const { return atan2(im, re); } // 返回复数的复角
16
17 void Complex::Set1(double real, double imag) // 设置复数的实部及虚部
18 {

```

```

19     re = real; im = imag;
20 }
21
22 void Complex::Set2(double radius, double angle) // 设置复数的模与复角
23 {
24     re = radius * cos(angle);
25     im = radius * sin(angle);
26 }
27
28 void Complex::Show1() const // 按一般形式输出复数
29 {
30     cout << re;
31     if(im > 0)
32         cout << " + " << im << "i";
33     else if(im < 0)
34         cout << " - " << -im << "i";
35 }
36
37 void Complex::Show2() const // 按指数形式输出复数
38 {
39     double r, angle;
40     r = sqrt(re*re + im*im);
41     angle = atan2(im, re);
42     if(r > 0)
43         cout << r << "exp(" << angle << "i)";
44     else
45         cout << 0;
46 }

```

对基于上述类进行开发的其他程序员来说，只需要将上述两个文件移植到相应的程序（如某工程文件）中即可。程序员只需要研究头文件 Complex.h 中的内容以及一些必要的说明文档，而不必关心源程序文件 Complex.cpp 的具体实现细节。下面，进一步将源程序文件 Complex.cpp 隐藏起来。

2. 单独编译源程序文件

C++ 程序按源程序文件分割编译分别生成目标代码文件，再由连接器将各目标代码文件、其他库文件中相关的函数目标码连接生成可执行文件。

将源程序文件 Complex.cpp（编译前进行预处理时将头文件 Complex.h 的内容包含到源程序文件中）编译生成目标代码文件 Complex.o（Visual C++ 等系统中的目标代码文件的后缀为 .obj）。

将头文件 Complex.h 和目标代码文件 Complex.o（位于 Debug 或 Release 文件夹中）一起提供给需要使用 Complex 类的其他程序员（不提供源程序文件 Complex.cpp）。此时，其他程序员只能通过头文件中类声明时的 public（公开的）部分了解并使用该类。在新开发的程序中只需要将 Complex.h 和 Complex.o 复制到程序文件所在的工作目录中，将头文件添加到工程文件中，并在工程文件的连接选项中指定连接 Complex.o 文件即可。具体操作方法如下：

- (1) MinGW Developer Studio 集成开发环境。在主菜单中依次选择 Project | Settings，则弹出工程设置的对话框，选其中的 Link 选项卡，在 Libraries 下的编辑栏中输入 Complex.o，再单击 OK 按钮即可。

(2) Visual C++ 6.0 集成开发环境。在主菜单中依次选择 Project | Settings，则弹出工程设置的对话框，选其中的 Link 选项卡，在 Object | library modules 下的编辑栏中增加 Complex.obj，再单击 OK 按钮即可。还可以直接创建 Win32 Static Library 生成静态库 Complex.lib，其他操作与前面的相同。

值得注意的是，不同编译器编译的目标代码文件一般是不能互换的。

9.5 类模板与模板类

在面向过程程序设计中，介绍了函数模板——在源代码级，数据类型待定而操作确定的函数描述。函数模板是程序员提供给编译系统的描述性声明。编译系统遇到相应的函数调用语句时，首先根据实际参数的数据类型自动生成（即由编译系统替程序员定义）一个实际的函数——模板函数，然后再编译这个模板函数。编译系统不编译函数模板只编译模板函数。因此，最好将函数模板写在头文件中供其他文件包含。描述函数模板时使用保留字 template, typename（或 class）以及尖括号“<”和“>”。

类似地，在源代码级将数据及其操作进行封装时，也可以用待定的数据类型声明一个类，将这种含有待定数据类型，操作描述确定的类称为类模板。

9.5.1 类模板声明

例如，描述二维向量类模板可以设计如下。

```
1 // Vec2.h          类模板须写在头文件中
2 #ifndef VEC2_H
3 #define VEC2_H
4 #include <iostream>
5 using namespace std;
6
7 template <typename TYPE> class Vec2           // 类模板声明
8 {           // TYPE 为待定数据类型，或称为形式数据类型
9 public:
10     Vec2(const TYPE &x=0, const TYPE &y=0);    // 构造函数
11     void Show() const                         // 可在类体中描述成员函数要进行的操作
12     {
13         cout << '(' << a << ", " << b << ')';
14     }
15     TYPE & GetX();
16     TYPE & GetY();
17     // 其他成员函数（略）
18
19 private:
20     TYPE a, b;                                // 数据成员的数据类型待定
21 };
22
23 // 在类声明体外描述成员函数要进行的操作
24 template <typename TYPE>
25 Vec2<TYPE>::Vec2(const TYPE &x, const TYPE &y)
26 {
27     a = x; b = y;
28 }
29
30 template <typename TYPE> TYPE & Vec2<TYPE>::GetX()
```

```

31 { // 采用引用返回, 可使函数调用成为左值
32     return a;
33 }
34
35 template <typename TYPE> TYPE & Vec2<TYPE>::GetY()
36 {
37     return b; // 采用引用返回, 可使函数调用成为左值
38 }
39 #endif

```

其中的成员函数可以在类模板体内描述，也可以在类模板体外描述。编译系统并不编译类模板，只有遇到全部确定了的待定数据类型后，由编译系统先自动生成一个实际的类——称为**模板类**，然后再对该模板类进行编译。因此，类模板及其所有成员的声明、成员函数体的描述（无论写在类模板体内还是写在类模板体外）均应该放在头文件中供其他文件包含。

请注意类模板成员函数在类声明体外描述时的书写格式。其中，`template <typename TYPE>` 表示模板，`TYPE` 表示形式数据类型，而“`模板名<TYPE>`”则是类模板完整的形式类名——带形式数据类型的类名。

9.5.2 模板类及其对象

将类模板的形式类名中的形式参数（即形式数据类型）换成实际数据类型，便是**模板类**。模板类是实际的类，将由编译系统对其进行编译。实际数据类型可以是基本数据类型，也可能是其他已经存在的类类型。模板类是对类模板的第一次实例化。

一个模板类是否能够通过编译，就要看类模板的描述中所涉及的形式数据类型数据运算操作是否对这个具体的实际数据类型可进行。就上述类模板而言，对形式数据类型的数据进行过插入操作（第 13 行）、赋值运算（第 27 行）。当然，基本数据类型的变量均能进行这几种操作。

用模板类创建对象是对类模板的第二次实例化，其语法格式如下。

模板名<实际数据类型名> 对象名(初始化表) ;

例如，接上面的程序。

```

1 // Testing.cpp
2 #include "Vec2.h"
3
4 int main()
5 {
6     Vec2<int> a, b(10); // Vec2<int> 为模板类类名
7     Vec2<char> c('A', 'B'); // Vec2<char> 为模板类类名
8     Vec2<double> *p; // Vec2<double> 为模板类类名, p 不是对象
9
10    p = new Vec2<double>(2.72, 3.14); // 创建动态对象
11
12    a.Show(); cout << endl; // a, b, c, *p 为对象名
13    b.Show(); cout << endl;
14    c.Show(); cout << endl;
15    p->Show(); cout << endl;
16

```

```

17     a.GetX() = 5;
18     a.GetY() = 8;
19     a.Show(); cout << endl;
20     c.GetX() += 32;
21     c.GetY() += 'a' - 'A';
22     c.Show(); cout << endl;
23     delete p;
24     return 0;
25 }
```

程序的运行结果：

```
(0, 0)
(10, 0)
(A, B)
(2.72, 3.14)
(5, 8)
(a, b)
```

9.6 小 结

类是对事物的抽象，对象是类的具体化实例。在面向过程程序设计中，我们将程序中的变量比作“小精灵”。由公式“**程序=对象+对象+…**”可知，与变量一样，对象是程序中的“精灵”。类的设计规定了对象的属性构成及对象自我表现的方法。类的设计是面向对象程序设计中最关键的部分。从数据类型的角度上看，类是一种构造数据类型，自然地，我们希望所设计的类类型能与基本数据类型一样的使用方便，甚至包括对象之间的关系运算、I/O 操作、可能的算术运算和赋值运算等。这些要求是 C++ 语言新机制出现的原动力，我们将在后续的章节中逐步讨论这些内容。

对象有其存放数据成员值的空间，不同对象的数据空间位于内存中的不同地址处。类的成员函数存放在内存代码区为所有该类的对象共享。非静态成员函数通过隐含传递“本对象地址”给隐含的形式参数 `this` 指针，以识别具体的对象。

数据成员的访问属性常设置成 `private` 或 `protected`（这两种访问属性的差别在第 14 章介绍继承的概念时予以讨论）。一些仅内部使用的成员函数也应该设置成 `private` 或 `protected` 的。利用访问属性控制可使类的设计内外有别。设计一个类时，应该尽可能使其功能齐全，对外的接口简单、清晰，使类本身“干净利落”。

练习 9

1. 指出下列各程序片段中的错误

(1) 下面的代码中有多处错误，请指出。

```

1 #include <iostream>
2 using namespace std;
3
4 class Test
5 {
```

```

6     int a=0;
7 public:
8     void Set(int x=0);
9     int Get(){return a;}
10 }
11
12 void Set(int x=0)
13 {
14     a = x;
15 }
16
17 Test::int Get()
18 {
19     return a;
20 }

```

(2) 指出下面的多文件结构程序中的错误。

```

1 // Point.h
2 #ifndef POINT_H
3 #define POINT_H
4 class Point
5 {
6     double x, y;
7 public:
8     Point(double x=0, double y=0);
9     void Move(double x=0, double y=0);
10    void Show();
11    double GetX(), GetY();
12 };
13 #endif

```

```

1 // Point.cpp
2 #include <iostream>
3 #include "Point.h"
4 Point::Point(double a, double b)
5 {
6     x = a; y = b;
7 }
8 void Point::Move(double a, double b)
9 {
10    x = a; y = b;
11 }
12 void Point::Show()
13 {
14     std::cout << "(" << x << ", " << y << ")";
15 }
16 void Point::GetX() {return x;}
17 void Point::GetY() {return y;}

```

```

1 // Point_test.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     Point a, b(3, 4), *p;

```

```

8     a.x = 5;
9     a.y = 8;
10    p->Show();
11    cout << b << endl;
12    return 0;
13 }

```

2. 基本题

(1) 编写一个日期类，具有设置日期，返回年、月、日，输出日期，计算本日期的下一天的日期等方法（成员函数）。要求按多文件结构编写，并注意将一些函数设计成常量成员函数。最后编写主函数测试之。

(2) 阅读下面两个文件中的代码。再编写一个包含主函数的源程序文件，在主函数中创建若干个对象测试各种一元二次方程求解的情况。

源代码 9.1 求解一元二次方程的类 CSolver

```

1 // CSolver.h
2 #ifndef CSOLVER_H
3 #define CSOLVER_H
4 class CSolver
5 {
6 public:           // 外部接口
7     CSolver(double a=1, double b=0, double c=0); // 构造函数
8     void Set(double a, double b, double c);
9     int Solve(), GetFlag() const;
10    double GetA() const, GetB() const, GetC() const;
11    double GetX1() const, GetX2() const;
12    void ShowEquation() const, ShowSolution() const;
13 protected:        // 受保护的
14     double Delta(); // 成员函数
15 private:          // 私有的
16     double _a, _b, _c, x1, x2; // 数据成员
17     int flag;
18 };
20 #endif

```

```

1 // CSolver.cpp
2 #include <iostream>
3 #include <cmath>
4 #include "CSolver.h"
5 using namespace std;
6
7 CSolver::CSolver(double a, double b, double c)
8 {
9     _a = a; _b = b; _c = c; flag = -3;
10 }
11 void CSolver::Set(double a, double b, double c)
12 {
13     _a = a; _b = b; _c = c; flag = -3;
14 }
15 double CSolver::GetA() const { return _a; }
16 double CSolver::GetB() const { return _b; }
17 double CSolver::GetC() const { return _c; }

```

```

18 double CSolver::GetX1() const { return x1; }
19 double CSolver::GetX2() const { return x2; }
20 int CSolver::GetFlag() const { return flag; }
21
22 void CSolver::ShowEquation() const
23 {
24     if(_a==1) cout << "x^2";
25     else cout << _a << "x^2";
26     if(_b>0) cout << "+"
27     else if(_b<0) cout << "-"
28     if(_c>0) cout << "+"
29     else if(_c<0) cout << "-"
30     cout << " = 0";
31 }
32 void CSolver::ShowSolution() const
33 {
34     switch(flag)
35     {
36         case -3: cout << "尚未进行求解。"; break;
37         case -2: cout << "该方程无实数根。"; break;
38         case -1: cout << "该方程是矛盾式。"; break;
39         case 0: cout << "该方程为一次方程。其解: "
40                 << "x=" << x1; break;
41         case 1: cout << "该方程有重根: "
42                 << "x1=x2=" << x1; break;
43         case 2: cout << "该方程有两个不同的实数根: "
44                 << "x1=" << x1
45                 << ", x2=" << x2; break;
46         case 3: cout << "该方程为恒等式。"; break;
47     }
48     cout << endl;
49 }
50 int CSolver::Solve()
51 {
52     double d;
53     if(_a==0)
54     {
55         if(_b==0)
56             flag = (_c==0) ? 3 : -1;
57         else
58         {
59             flag = 0;
60             x1 = x2 = -_c/_b;
61         }
62     }
63     else
64     {
65         d = Delta();
66         if(d >= 0)
67         {
68             d = sqrt(d);
69             x1 = (-_b - d)/(2*_a);
70             x2 = (-_b + d)/(2*_a);
71             flag = (d>0) ? 2 : 1;
72         }
73         else flag = -2;
74     }
75     return flag;
76 }
```

```
77 double CSolver::Delta()
78 {
79     return _b*_b - 4*_a*_c;
80 }
```

(3) 在计算机上调试下面所设计的类，再增加一个包含主函数的源程序文件，在主函数中创建若干个对象测试之。

```
1 // Test.h
2 #ifndef TEST_H
3 #define TEST_H
4 namespace mySpace
5 {
6     class Test
7     {
8         public:
9             Test(int x = 0);
10            void Set(int x);
11            int Get() const;
12            void Show() const;
13     private:
14         int a;
15     };
16 }
17 #endif
```

```
1 // test.cpp
2 #include <iostream>
3 #include "Test.h"
4 using namespace std;
5 mySpace::Test::Test(int x) { a = x; }
6 void mySpace::Test::Set(int x) { a = x; }
7 int mySpace::Test::Get() const { return a; }
8 void mySpace::Test::Show() const { cout << a << endl; }
```

第 10 章 构造函数及赋值运算

构造函数、拷贝构造函数和析构函数是 C++ 类中具有特殊地位的函数，它们与对象的创建、销毁相关。构造函数或者拷贝构造函数用于创建对象，析构函数用于销毁对象。它们的形式特别容易辨认。它们的执行常常是由系统自动调用的，识别何时调用了哪个函数却需要读者一定的细心和耐心，其中涉及有关对象的生命期、对象在函数之间传递等概念。

执行赋值运算并不创建新对象，作为参与赋值运算的左操作数（左值对象）是早已存在的对象，赋值运算操作只是可能改变该左值对象的数据成员的值而已。

学习本章后，读者应该对面向过程程序设计中介绍的变量的生命期、数据在函数之间传递（值传递、引用传递、值返回和引用返回）有更加深刻的认识。学习本章后，读者应该对对象的数据空间，尤其是带资源的对象的资源空间有一个全面而深刻的认识。对于对象可能带资源的类的设计，要时刻警醒深拷贝构造、深赋值运算及析构操作的必要性。

10.1 构造函数

创建对象时由系统自动调用适当的构造函数来完成处理对象空间、构造结构和初始化数据成员等工作。没有构造函数就不能构造对象，任何类都至少有两个构造函数，其中一个是拷贝构造函数。构造函数没有返回类型（函数体内可以使用 `return;` 语句），函数名与类名相同，可以没有或有一个或有多个参数。构造函数可以被重载以方便多种场合、多种条件下创建对象。

10.1.1 默认构造函数

不需要实参的构造函数被称为默认的构造函数。声明一个类时，若未定义任何构造函数，则系统会为该类提供一个默认的构造函数，系统提供的这种默认构造函数将不对对象的数据成员进行初始化。此时，所创建的对象数据成员的初值取决于对象的存储类型（全局对象、静态全局对象和静态局部对象的数据成员初始化为各位全为 0；局部自动对象、动态对象各数据成员的值是不可预知的）。只要在类声明时定义了构造函数，系统将不再提供默认的构造函数，倘若仍需要默认的构造函数，则需要类的设计者重载默认构造函数。通常的做法是：设计所有参数均带默认值的构造函数。当不提供实参时，该构造函数便成为默认的构造函数。

例如，第 9 章例 9.1、例 9.2 中均使用了参数带默认值的构造函数。下面的语句：

```
Time t;  
Complex z;
```

分别为用类 `Time` 的默认构造函数创建对象 `t`；用类 `Complex` 的默认构造函数创建对象 `z`。由于这两个类的构造函数的所有参数均带默认值 0，因此，无论这两个对象的存储类

型是什么（即使是局部自动对象），它们的数据成员的初始化值均为 0。

需要指出的是如下语句：

```
Time t();
Complex z();
```

不是创建对象，而是函数原型用于函数声明。即声明了参数个数为 0，返回类型为 Time 值返回的函数 t；声明了参数个数为 0，返回类型为 Complex 值返回的函数 z。

10.1.2 转换构造函数

带一个实参的构造函数被称为转换构造函数。之所以称这种构造函数为转换构造函数，是因为可以用它实现数据类型的隐式转换或强制转换。可以为一个类设计多个转换构造函数。

例 10.1 学生 (Student) 的描述及处理方法。

- (1) 属性：学号 (id)、姓名 (name)、性别 (gender) 和年龄 (age)。
- (2) 行为：构造函数、输出学生信息，其余行为暂略。

```
1 // Student.h
2 #ifndef STUDENT_H
3 #define STUDENT_H
4 #include <string>                                // 为了使用 C++ 字符串
5 using namespace std;
6 class Student
7 {
8 public:
9     Student(char *Id="00000000", string Name="NoName",
10             char Gender='m', int Age=20);
11    Student(string Id, char *pName="无名氏",
12             char Gender='f', int Age=20);
13    void Show() const;
14
15 private:
16    char id[9];                                     // 字符数组，存放 C-字符串
17    string name;                                    // C++ 字符串
18    char gender;
19    int age;
20 };
21 #endif
```

```
1 // Student.cpp
2 #include "Student.h"
3 #include <iostream>
4 #include <iomanip>
5 #include <cstring>                                // 为了使用 C-字符串处理函数
6 using namespace std;
7 Student::Student(char *Id, string Name, char Gender, int Age)
8 {
9     strncpy(id, Id, sizeof(id)); id[sizeof(id)-1] = '\0';
10    name = Name;
11    gender = (Gender=='M' || Gender=='m')? 'm' : 'f';
12    age = Age;
13 }
```

```

14 Student::Student(string Id, char *pName, char Gender, int Age)
15 {
16     strncpy(id, Id.c_str(), sizeof(id)); id[sizeof(id)-1]='\0';
17     name = pName; // 自动转换后赋值
18     gender = (Gender=='M' || Gender=='m')? 'm' : 'f';
19     age = Age;
20 }
21 void Student::Show() const
22 {
23     cout << setw(8) << id << " "
24         << setw(8) << name << " "
25         << (gender=='m'? "男" : "女")
26         << " " << age << endl;
27 }
```

其中 `c_str` 是 `string` 类的成员函数，其功能是将 C++ 字符串转换成 C-字符串。

```

1 // StudentTest.cpp          测试程序
2 #include "Student.h"
3 void test1()
4 {
5     Student s1;           // 调用默认的构造函数
6     // 利用 C-字符串转换构造 Student 对象
7     Student s2 = "00001357"; // 调用转换构造函数（格式一）
8     Student s3("00001357"); // 调用转换构造函数（格式二）
9     // 利用 C-字符串转换构造 C++ 字符串对象
10    string id("00002468"); // 利用 C-字符串转换构造 string 对象
11    // 利用 C++ 字符串转换构造 Student 对象
12    Student s4 = id;       // 调用另一个转换构造函数（格式一）
13    Student s5(id);       // 调用另一个转换构造函数（格式二）
14    s1.Show();
15    s2.Show();
16    s3.Show();
17    s4.Show();
18    s5.Show();
19 }
20 int main()
21 {
22     test1();
23     return 0;
24 }
```

程序的运行结果：

00000000	NoName	男	20
00001357	NoName	男	20
00001357	NoName	男	20
00002468	无名氏	女	20
00002468	无名氏	女	20

从运行结果可见，利用转换构造函数可以用其他数据类型（即参数的数据类型）的数据初始化新创建的类对象。初始化有两种格式：

类名 对象名(实参, 实参, ...);	
类名 对象名 = 转换构造函数的实参;	// “=” 为初始化记号，不是赋值运算符

10.1.3 构造函数的使用

第 9.2 节中已经提到可以创建各种存储类型的对象、对象数组等，本小节侧重构造函数及其应用，特别是构造对象数组时参数的传递问题。

通常情况下，构造函数是在创建对象时由系统自动调用的，系统根据实际参数选择一个匹配的构造函数来创建对象、初始化数据成员。程序员也可以在程序中直接使用构造函数初始化正在创建的新对象。

在这里，顺带提到了拷贝构造函数。关于拷贝构造函数的详情见第 10.3 节。

1. 创建对象数组

创建对象数组时，需要提供一个常量作为数组的元素个数。在依次创建数组的每个元素时均要调用一次构造函数或者拷贝构造函数。

可以显式地使用构造函数初始化数组元素；可以用已经存在的对象利用拷贝构造函数初始化数组元素；可以利用转换构造函数（即直接提供转换构造函数实参，这种情况下每个元素对象仅分别接受一个实参数）初始化数组元素：

```
类名 对象数组名[元素个数] = {转换构造函数的实参数} ;
类名 对象数组名[元素个数] = {构造函数名(实参数)} ;
```

2. 创建动态对象或动态对象数组

C++ 运算符 `new` 和 `delete` 具有很强的功能^[1]。用 `new` 运算符创建堆对象时，该操作运算具有调用类的相应构造函数或拷贝构造函数的功能，使分配空间、构造结构和初始化对象数据成员等工作一气呵成。

```
new 类名;
new 类名(实参, 实参, ...);
new 类名[元素个数](实参, 实参, ...);
```

同样，运算符 `new` 创建堆对象（堆对象数组）成功后，返回从堆空间所“切”下的那一片连续空间的首地址。程序中需要用指针变量记录下这个地址值，以便能方便地访问堆对象或堆对象数组的元素，以及使用完毕后释放所申请的堆内存资源。若 `new` 操作不成功，则返回空指针 `NULL`。创建动态数组的每一个元素时都要调用一次构造函数。

释放堆内存资源仍然使用 C++ 操作符 `delete`。关于 `delete` 释放动态对象时的特性参见第 10.2 节。

接例 10.1，补充几个函数并在主函数中调用。

```
1 // StudentTest.cpp
2 #include "Student.h"
3
4 void test2()
5 {
```

^[1] C 语言中通过调用 `calloc` 或 `malloc` 等函数申请堆内存资源。这些函数的返回类型为 `void*`，即它们仅从堆内存空间中“切”下一片连续的空间，而并不清楚该空间的结构和用途，因而无法构造 C++ 类的对象的数据成员。例如，用语句 `Student *p = (Student*)calloc(10, sizeof(Student))`；能从堆空间中切下 `10*sizeof(Student)` 字节的连续空间，可以存放 10 个 `Student` 类的对象。但是这样的处理绕开了构造函数，本类又没有提供其他修改私有数据成员的接口，因而数据成员的值是无意义的。

```

6   Student s1("00012345", string("Zhang"));
7   Student s2("00013456", string("Li"), 'm');
8   Student s3("00014567", string("Zhao"), 'f', 18);
9   s1.Show();
10  s2.Show();
11  s3.Show();
12 }
13
14 void test3()
15 {
16     int i;
17     Student s("00020001", string("Zhou"), 'f', 16);
18     Student array[5] = {
19         "00020002",                                // 使用转换构造函数
20         string("00020003"),                        // 使用另一个转换构造函数
21         Student("00020004", string("Wang"), 'f'), // 直接调用构造函数
22                               // 直接调用构造函数
23         Student(string("00020005"), "Chen", 'm', 19),
24                               // 调用拷贝构造函数
25         s
26     };
27     for(i=0; i<5; i++)
28         array[i].Show();
29
30     Student *p = new Student(string("00030001"), "Sun", 'm', 18); // 调用适当的构造函数
31     p->Show();
32     delete p; // 释放堆内存资源
33
34
35     Student *p1 = new Student[2](string("00040008"), "Qian"); // 调用适当的构造函数 (VC++ 只能用默认的构造函数创建堆对象数组)
36     for(i=0; i<2; i++)
37         p1[i].Show();
38     delete [] p1; // 释放堆内存资源
39 }
40
41
42 int main()
43 {
44     test2();
45     test3();
46     return 0;
47 }

```

程序的运行结果：

00012345	Zhang	男	20
00013456	Li	男	20
00014567	Zhao	女	18
00020002	NoName	男	20
00020003	无名氏	女	20
00020004	Wang	女	20
00020005	Chen	男	19
00020001	Zhou	女	16
00030001	Sun	男	18
00040008	Qian	女	20
00040008	Qian	女	20

10.2 析构函数

10.2.1 析构函数的概念

析构函数也是类中特殊的成员函数。析构函数没有返回类型、没有参数、不能被重载，一般不应显式地调用它。对于某个对象而言，当其生命期结束时由系统自动调用析构函数完成一些必要的善后处理工作。类的析构函数是容易辨认的，其函数名为~类名（波浪号与类名），常称析构函数为逆构造函数。辨别何时调用了析构函数需要对对象的生命期有清晰的认识。

在声明一个类时，若未设计析构函数，则系统会为该类提供一个默认的析构函数。由系统提供的默认析构函数仅完成对象基本数据空间的清理工作。若对象带有资源，则应该定义析构函数以完成对象所带资源的清理工作。

C++ 的运算符 `delete` 在释放堆对象或 `delete []` 在释放堆对象数组时，具有自动调用对象的析构函数的功能^[1]。析构对象数组时，将为每一个元素调用一次析构函数。

10.2.2 对象构造和析构的顺序

- (1) 全局对象、静态全局对象在主函数执行之前被构造，直至主函数结束后析构。
- (2) 静态局部对象在其所在的函数第一次被调用时创建（该对象的基本空间在全局数据区，其所在函数返回后，静态局部对象为不可见，对象仍然存在且保留当前属性值，当再次调用其所在函数时，该对象成为可见的，不重复创建），直至主函数结束后析构。
- (3) 局部自动对象（包括值传递的形参对象）在其所在函数每一次调用时创建，函数返回后析构。
- (4) 函数值返回的临时对象在函数返回时创建，在参与其所在表达式运算后析构。
- (5) 动态对象（堆对象）在执行 `new` 操作时创建，在执行 `delete` 时析构。

在上述基本原则下，先创建的对象后析构，后创建的对象先析构。

10.3 拷贝构造函数

对于基本数据类型 `int`，定义整型变量及初始化方式有

```
int a, b = 3;           // 采用默认方式定义变量 a, 用常量初始化变量 b
int c = a, d = b;       // 用已经存在的变量 a,b 分别初始化变量 c,d
```

变量 `a` 的初始化值取决于其存储类型，变量 `b` 的初始化值为 3。变量 `c` 的初始化值与变量 `a` 的值相同（它们的 `sizeof(int)` 字节的各个位皆相同）；变量 `d` 的初始化值与变量 `b` 的值相同（皆为 3）。

对于类类型，也希望能像基本数据类型一样，除了用构造函数创建对象外，也可以用已经存在的对象的值来初始化新创建的对象。这是一种新情况：呼唤一种新的“构造函

^[1]C 语言中释放堆内存是通过调用 `void free(void*);` 函数完成的。若用该函数释放堆对象、堆对象数组，则绕过了析构函数，只能清理对象基本空间，无法处理对象所带的资源等必要的善后操作。

数”，要求这种构造函数的形式参数为本类对象的引用（请读者思考：为什么不能用传值型的形式参数，也不能用指针型的形式参数？）。

在类体中声明拷贝构造函数，其函数原型的格式为

```
类名(const 类名&);
```

拷贝构造函数可以在类体中声明时定义，也可以在类体外定义。在类体外定义的格式如下：

```
类名::类名(const 类名& 形式参数)
{
    // 执行语句
}
```

这样的函数被称为拷贝构造函数或复制构造函数。拷贝构造函数也是类的一个特殊成员函数：它没有返回类型、函数名与类名相同、函数的形式参数为该类对象的常引用（即用 `const` 对实参进行保护）。用拷贝构造函数创建对象时，将不再调用其他的构造函数。亦即，构造函数、拷贝构造函数均是创建对象时所用的函数，它们使用的场合不同。

声明一个类时若未定义拷贝构造函数，则编译系统会为该类提供一个默认的拷贝构造函数。默认的拷贝构造函数按对象的数据成员（数组成员则按每一个元素）依次调用成员的拷贝构造函数进行拷贝构造。若在声明类时定义了拷贝构造函数，则系统将不再提供拷贝构造函数。

与构造函数、析构函数一样，拷贝构造函数的形式是容易辨认的。拷贝构造函数由系统自动调用。然而，何时调用了拷贝构造函数则需要读者概念清楚加上细心。拷贝构造函数被调用的总原则：当用一个已经存在的对象初始化一个正在创建的对象时，调用拷贝构造函数。有如下三种常见场合。

(1) 创建新对象时，用一个已经存在的对象作为初始化值。

(2) 调用函数时，用实参（已经存在的对象）初始化值传递创建的形参。

(3) 函数类类型对象值返回时，用函数所返回的表达式（一个已经存在的对象）初始化值返回时创建的临时对象（参见第 3.2.2 小节的注）。

对于上述(2)、(3)两点，常称创建了对象的副本。值传递型形式参数对象（实参对象的副本）、函数值返回时创建的临时对象（返回值的副本）都不是实参本身，对副本的操作不会影响实参；另一方面，创建副本需要花费时间和内存空间。因此，在今后的函数设计中，应该尽可能地使用引用型形式参数、引用返回类型来避免拷贝构造对象。一般是在不得已的情况下才采用值传递、值返回。

然而，采用引用型形式参数传递即传递了对象本身，对引用型形式参数的修改，就是对实参对象的修改。许多情况下，为了避免拷贝构造而采用引用型形式参数，又不希望函数体内对实参对象进行修改，此时，可加保留字 `const` 对实参予以保护。不仅如此，调用这样的函数还能用常量对象做实参。

拷贝构造函数可以访问另一个对象的所有（包括公开的、受保护的、私有的）成员。因为类的成员函数对该类所有对象的所有成员有无限制的访问权限（参见第 9.1.2 小节）。

10.3.1 浅拷贝构造——拷贝对象基本空间的数据成员

在拷贝构造对象时，仅根据对象基本空间的数据成员拷贝者被称为浅拷贝构造。系统提供的默认拷贝构造函数只执行浅拷贝操作（取决于其各个成员的拷贝构造函数）。

例 10.2 名称 (Name) 类的设计。

下面将围绕名称类的设计，按“初步设计→发现问题→解决问题”螺旋式递进，逐步展开讨论。为简单起见，仅设计一个数据成员 name。为了看清何时调用了何种函数，在成员函数中添加一定的输出语句。

本例相当于设计自定义的字符串类。下面先用字符数组（第 10.3 节），然后用字符指针（第 10.4 节）进行设计。在第 10.5 节组合 C++ 的 `string` 类设计方案实质上只是外加了一个“壳”，没有实质意义。更为一般的设计参见第 12.3 节或练习 13（3）。

例 10.2 (A) 字符数组做数据成员（正例，但容量受限）。

```

1 // ArrayName.cpp
2 #include <iostream>
3 #include <cstring>
4 using namespace std;
5
6 class ArrayName
7 {
8 public:
9     ArrayName(const char *pName="NoName")      // 构造函数
10    {
11        strncpy(name, pName, sizeof(name));
12        name[sizeof(name)-1] = '\0';
13        cout << "Constructing an object of ArrayName ["
14            << name << "]." << endl;
15    }
16    ArrayName(const ArrayName &an)                // 拷贝构造函数
17    {
18        strcpy(name, an.name);
19        cout << "COPY constructing an object of ArrayName ["
20            << name << "]." << endl;
21    }
22    ~ArrayName()                                // 析构函数
23    {
24        cout << "Destructing an object of ArrayName ["
25            << name << "]." << endl;
26    }
27    void ChangeName(const char *pName)
28    {
29        cout << "Change ArrayName [" << name;
30        strncpy(name, pName, sizeof(name));
31        name[sizeof(name)-1] = '\0';
32        cout << "] -> [" << name << "]." << endl;
33    }
34    void Show() const
35    {
36        cout << "ArrayName: [" << name << "]." << endl;
37    }
38 private:
39     char name[10];
40 };
41

```

```

42 ArrayName fAN(ArrayName x, ArrayName *p, ArrayName &r)
43 {           // 测试函数。值传递，地址值传递，引用传递，值返回
44     cout << "In_fAN()..." << endl;
45     x.ChangeName("Architecture");
46     p->ChangeName("Software");
47     r.ChangeName("Application");
48     x.Show();
49     p->Show();
50     r.Show();
51     cout << "Return_from_fAN()..." << endl;
52     return r;
53 }
54 int main()
55 {
56     ArrayName an1("Java_Program");
57     ArrayName an2("C++_Program");
58     ArrayName an3("C#_Program");
59     cout << "\nBegin_fAN()..." << endl;
60     fAN(an1, &an2, an3);
61     cout << "End_of_fAN()\n..." << endl;
62     an1.Show(); an2.Show(); an3.Show();
63
64     an2 = an1;           // 赋值操作
65     cout << "\nAfter_assign_an2_=an1;" << endl;
66     an2.Show();
67     cout << "Return_to_Operating_System." << endl;
68     return 0;
69 }

```

ArrayName 类的特点：对象的基本空间就是对象全部数据空间，即用对象基本空间字符串数组为容器存放字符串的内容；字符串有最大长度的限制。

(1) 由于数据成员 **name** 是一个数组的数组名，不能对其赋值，故在成员函数体中用函数 **strcpy** 或函数 **strncpy** 将字符串的内容拷贝至数组各元素。

(2) 构造函数中使用 **strncpy** 函数是为了将从参数处传入的可能超长的字符串截短，因为每一个对象仅能存放长度不超过 **sizeof(name)-1**（此处为 9）个字符的 C-字符串。又因为 **strncpy** 函数不一定拷贝串结束标志（'\0'），故至少应保证字符数组的最后一个元素为串结束标志。成员函数 **ChangeName** 的情形是类似的。

(3) 拷贝构造函数仅需要使用 **strcpy** 函数即可，因为已经存在的对象的数据成员字符串肯定不会超长。

(4) 析构函数无须做其他事情，故仅输出一些信息。

其实，若去掉类中的拷贝构造函数、析构函数，直接利用系统提供的默认拷贝构造函数、析构函数，除了无输出语句外，程序的功能是相同的。

请读者分析如下的运行结果，注意何时调用了拷贝构造函数及析构五个对象的顺序。

```

Constructing an object of ArrayName [Java_Prog].
Constructing an object of ArrayName [C++_Progr].
Constructing an object of ArrayName [C#_Progra].
Begin fAN() ...
COPY constructing an object of ArrayName [Java_Prog].

```

```

In fAN() ...
Change ArrayName [Java Prog]->[Architect].
Change ArrayName [C++ Progr]->[Software].
Change ArrayName [C# Progra]->[Applicati].
ArrayName: [Architect].
ArrayName: [Software].
ArrayName: [Applicati].
Return from fAN() ...
COPY constructing an object of ArrayName [Applicati].
Destructing an object of ArrayName [Applicati].
Destructing an object of ArrayName [Architect].
End of fAN() ...

ArrayName: [Java Prog].
ArrayName: [Software].
ArrayName: [Applicati].

After assign an2 = an1;
ArrayName: [Java Prog].
Return to Operating System.
Destructing an object of ArrayName [Applicati].
Destructing an object of ArrayName [Java Prog].
Destructing an object of ArrayName [Java Prog].

```

例 10.2 (B) 字符型指针变量做数据成员 (反例之一)。

```

1 // PointName1.cpp
2 #include <iostream>
3 #include <cstring>
4 using namespace std;
5
6 class PointName1
7 {
8 public:
9     PointName1(char *pName="NoName") { name = pName; }
10    void UpperName() { strupr(name); }
11    void Show() const { cout << "\"" << name << "\"" << endl; }
12 private:
13     char *name;
14 };
15
16 int main()
17 {
18     char str[10] = "Tom", *ptr = new char[10];
19     strcpy(ptr, "Jerry");
20     PointName1 pn1(str), pn2(ptr), pn3("Snoopy");// 创建三个对象
21     pn1.Show();
22     pn2.Show();
23     pn3.Show();
24     cout << "Do_something..." << endl;
25     strcpy(str, "Winnie");
26     pn1.Show();           // 名字已被修改，而对象却浑然不知
27     delete [] ptr;
28     pn2.Show();           // 目标已不存在，而对象也浑然不知
29     pn3.UpperName();      // 此行编译通过，运行时出错
30     pn3.Show();           // 因为数据成员指针所指的区域为常量
31     return 0;
32 }

```

每个 PointName1 类的对象的基本空间为一个指针，仅占 `sizeof(void*)` 字节，仅能存放一个内存地址值，这个地址值是对象的直接属性。然而真正重要的是以该地址为起始的 C-字符串的内容（间接属性）。对象对字符串的长度却没有限制。

这个类的设计虽无语法错误，但存在一个非常严重的问题——**封装不严**（实际上是没有封装）。出现如下荒唐的情况。因此，我们将这样设计的类作为反例示众。

① 对象不能对自己的（间接）属性作主（对象 `pn3` 数据成员所指之处为字符串常量，不允许被修改）。

② 尽管对象的数据成员是私有的，对象的（间接）属性却可以被外部任意修改（如对象 `pn1` 的属性），甚至被销毁（对象 `pn2` 的属性），而对象却浑然不知。

10.3.2 对象的资源空间

那么，如何突破字符串最大长度的限制，并且封装严密呢？一个可行的解决方案是：创建对象时由构造函数为该对象申请堆内存空间、在该对象生命周期终止时由析构函数释放对象所占用的堆内存资源。我们称这种情形为**对象带资源**。

对象的空间=对象的基本空间+对象的资源空间

堆内存空间是一种常用的资源（除此以外，数据文件、图标、鼠标指针等都有可能成为对象的资源），本教材主要讨论堆内存资源。对象的基本空间占用 `sizeof(类型名或对象名)` 字节，对象的资源空间的大小可因对象不同而异，这两部分在内存中不一定连续。例如，基本空间可能在全局数据区、栈区或堆区，而堆内存资源在堆区。

例 10.2 (C) 字符型指针变量做数据成员 (反例之二)。

```

1 // PointName2.cpp
2 #include <iostream>
3 #include <cstring>
4 using namespace std;
5
6 class PointName2
7 {
8 public:
9     PointName2(const char *pName="NoName") // 构造函数
10    {
11        name = new char[strlen(pName)+1]; // 根据需要量身定做
12        strcpy(name, pName); // 无须截短处理
13    }
14    ~PointName2() // 析构函数
15    {
16        if(name!=NULL) delete [] name;
17    }
18    void UpperName() { strupr(name); }
19    void Show() const { cout << "\"" << name << "\"" << endl; }
20
21 private:
22     char *name;
23 };
24
25 int main()
26 {
27     char str[10] = "Tom", *ptr = new char[10];
28     strcpy(ptr, "Jerry");

```

```

29     PointName2 pn1(str), pn2(ptr), pn3("Snoopy");
30     pn1.Show();
31     pn2.Show();
32     pn3.Show();
33     cout << "Do something..." << endl;
34     strcpy(str, "Winnie"); // 这个修改不影响对象 pn1
35     pn1.Show();
36     delete [] ptr;        // 这个释放不影响对象 pn2
37     pn2.Show();
38     pn3.UpperName();      // 对象属性的修改由对象自己作主
39     pn3.Show();
40     PointName2 pn4 = pn1; // 浅拷贝构造 pn4, 引出新问题
41     cout << "Return to Operating System." << endl;
42     return 0;
43 }

```

在创建 PointName2 类的对象时，构造函数根据实参的尺寸“量身定做”申请堆内存空间。该空间的首地址仅由该对象掌握（存放在私有数据成员 name 指针变量中）保证良好的封装性。一般地，外界无法知道对象所带堆内存资源的首地址。该堆内存资源也应该由其析构函数释放。

`sizeof(PointName2)` 等于 `sizeof(void*)` 为一个对象基本空间的尺寸。对象资源空间的尺寸与创建对象时的实参有关。这两部分之和为对象的空间。

从上面的测试程序（主函数）可见，直到执行主函数的 `return 0;` 语句，上述程序均不出现错误。但是程序结束时需要依次析构对象 pn4, pn3, pn2 和 pn1。首先析构对象 pn4，其析构函数将对象 pn1 的资源释放了（因为，程序第 40 行对象 pn4 是由对象 pn1 通过默认的拷贝构造函数创建的。默认的拷贝构造函数为浅拷贝构造，仅拷贝了对象的基本空间的内容，造成指针 `pn4.name` 与指针 `pn1.name` 存放相同的堆内存地址值，即它们的目标是相同的，如图 10-1 所示）；紧接着析构对象 pn3, pn2（未暴露错误）；最后析构对象 pn1 执行析构函数中的 `delete [] name;` 时，由于再次释放已经释放的空间而导致出错。因此，我们也将这个类作为反例示众。

造成这种错误的根本原因是：不同的对象共享同一资源。类的对象的资源是对象的属性，应该具有一定的私有性。对象之间的这种“资源共享”是应该避免的。

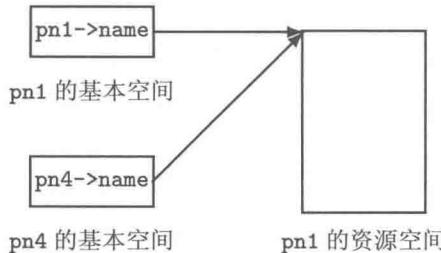


图 10-1 利用对象 pn1 浅拷贝构造对象 pn4 的结果

10.3.3 深拷贝构造——构造属于自己的资源空间

对象带资源时，若不同的对象共享同一个资源，析构对象时将已经释放的资源再度释放是错误的。解决这个问题的办法是让每个对象各自带自己独立的资源。因此，拷贝构造

函数在创建对象时，还需要申请属于自己的资源空间，然后仿照已经存在的对象拷贝一份数据到自己的资源空间中来。这种拷贝资源空间数据的拷贝构造被称为深拷贝构造。

因此，上面的类设计中必须设计深拷贝构造函数，以取代系统提供的默认拷贝构造函数。

例 10.2 (D) 字符型指针变量做数据成员（反例之三）。

```

1 // PointName3.cpp
2 #include <iostream>
3 #include <cstring>
4 using namespace std;
5
6 class PointName3
7 {
8 public:
9     PointName3(const char *pName="NoName")           // 构造函数
10    {
11        name = new char[strlen(pName)+1];
12        strcpy(name, pName);
13    }
14    PointName3(const PointName3 &pn)                 // 深拷贝构造函数
15    {
16        name = new char[strlen(pn.name)+1];
17                                // 构造属于自己的资源空间
18        strcpy(name, pn.name);           // 复制已存在对象资源内容
19    }
20    ~PointName3(){ if(name) delete [] name; } // 析构函数
21    void UpperName() { strupr(name); }
22    void Show() const { cout << "\"" << name << "\"" << endl; }
23 private:
24     char *name;
25 };
26
27 int main()
28 {
29     char str[10] = "Tom", *ptr = new char[10];
30     strcpy(ptr, "Jerry");
31     PointName3 pn1(str), pn2(ptr), pn3("Snoopy");
32     pn1.Show();
33     pn2.Show();
34     pn3.Show();
35     cout << "Do something..." << endl;
36     strcpy(str, "Winnie"); // 这个修改不影响对象 pn1
37     pn1.Show();
38     delete [] ptr;         // 这个释放不影响对象 pn2
39     pn2.Show();
40     pn3.UpperName();      // 对象属性的修改由对象自己作主
41     pn3.Show();
42     PointName3 pn4 = pn1; // 深拷贝构造对象 pn4, 有其自己的资源空间
43     pn2 = pn1;            // 浅赋值运算, 又引出新问题
44     // 导致对象 pn2 原来的资源丢失而无法释放 (错误之一)
45     // 同时导致对象 pn2 共享 pn1 的资源, 析构 pn1 时出错 (错误之二)
46     cout << "Return to Operating System." << endl;
47     return 0;
48 }
```

与反例之二比较，上述程序在类的声明中定义了深拷贝构造函数，使拷贝构造的对象

pn4 拥有自己独立的资源。解决了反例之二中的问题。

上述程序的主函数中新增加的赋值语句 `pn2 = pn1;` 又引出一个新的问题。赋值操作的结果也导致不同的对象共享同一资源，而且左值对象原有的资源被丢失（`pn2.name` 存放的堆内存地址被修改）且无法释放。解决这个新问题的方法请看下一节。

10.4 赋值运算

与拷贝构造函数创建新对象不同，赋值运算是用右值对已经存在的左值对象的属性进行修改的操作，不创建新对象。虽然赋值操作不创建新对象，我们将赋值运算的有关内容纳入本章是因为对象带资源时需要深赋值的根本原因与需要深拷贝构造一样——希望对象所带的资源是各自独立、独享的。

与类的几种特殊函数类似，在声明一个类时若未定义赋值运算符，则系统会为类提供一个默认的赋值运算符。系统所提供的默认赋值运算符按对象的数据成员（数组成员则按每一个元素）依次赋值（称其为浅赋值运算）。若对象不带资源，则浅赋值运算就足以满足赋值操作的需要。

若对象带有资源，则简单的浅赋值运算将会导致左值对象“遗忘”（即“丢失”）自己原有的资源而导致无法释放它们，并且“共享”另一对象的资源将导致析构资源错误。因而，对于对象可能带资源的类需要设计深赋值运算操作。

可以想象，深赋值运算需完成如下操作^[1]：

- (1) 主动释放对象原有的资源。
- (2) 重构对象的新资源空间。
- (3) 复制右值对象资源的数据以填充新资源。
- (4) 引用返回本对象 (`*this`)，这是因为赋值运算表达式应该有运算结果，其结果就是该左值对象本身。

C++ 中，运算符是一种特殊的函数（被称为运算符函数），函数名为保留字 `operator` 及运算符；对于双目运算符，其两个操作数就是运算符函数的两个参数；执行运算表达式操作就是调用运算符函数，此时可将两个实参分别放置到运算符的左右两边（也可以用一般的函数调用格式执行运算符操作）。关于运算符重载的详细讨论将在第 13 章进行，本节仅讨论赋值运算符函数。读者只需将“`operator=`”视为函数名即可，赋值表达式（即赋值运算符函数调用格式）为：

对象名.`operator=(右值)`

或者，更多地是使用格式：

对象名 = 右值

C++ 中，赋值运算符函数必须为非静态成员函数。我们知道，类的非静态成员函数的第一个形式参数是隐含的 `this` 指针常量，在用对象调用这样的成员函数时系统隐含地

^[1] 应该注意到右值与左值为同一对象的自我赋值运算，尽管右值对象（形参）设计为常引用，此时修改左值对象（如贸然地释放其原有资源），亦改变了右值对象。因此，当右值对象与左值对象为同一对象（即它们的起始地址相同）时，则应直接返回本对象。

传递本对象的地址。而在使用运算表达式格式或运算符函数调用时，编译系统将实参对象本身（左值）兼容地处理成对象的地址（初始化 this 指针）。

有了深拷贝构造函数、深赋值运算符函数和析构函数，字符型指针变量做数据成员，带资源的类可正确地设计如下。

例 10.2 (E) 字符型指针变量做数据成员（正确的程序）。

```

1 // PointName.cpp
2 #include <iostream>
3 #include <cstring>
4 using namespace std;
5
6 class PointName
7 {
8 public:
9     PointName(const char *pName="NoName")           // 构造函数
10    {
11        name = new char[strlen(pName)+1];
12        strcpy(name, pName);
13        cout << "Constructing an object of PointName \""
14            << name << "\n" << endl;
15    }
16
17    PointName(const PointName &pn)                  // 深拷贝构造函数
18    {
19        name = new char[strlen(pn.name)+1];
20        strcpy(name, pn.name);
21        cout << "COPY constructing an object of PointName \""
22            << name << "\n" << endl;
23    }
24
25    ~PointName()                                     // 析构函数
26    {
27        cout << "Destructing an object of PointName \""
28            << name << "\n" << endl;
29        if(name!=NULL)
30            delete [] name;
31    }
32
33    PointName & operator=(const PointName &pn) // 深赋值运算符函数
34    {
35        cout << "ASSIGN an object of PointName \""
36            << pn.name << "\n" << endl;
37        if(&pn != this)                            // 防止自我赋值而丢失资源
38        {
39            if(name) delete [] name;             // (1) 主动释放原资源
40            name=new char[strlen(pn.name)+1]; // (2) 申请自己的资源
41            strcpy(name, pn.name);            // (3) 复制右值数据
42        }
43        return *this;                         // (4) 返回赋值结果
44    }
45
46    void UpperName()
47    {
48        cout << "Upper \" " << name << " -> \" ";
49        strupr(name);
50        cout << name << "\n" << endl;
51    }

```

```

52     void Show() const
53     {
54         cout << "\\" << name << "\\" << endl;
55     }
56 private:
57     char *name;
58 };
59
60 int main()
61 {
62     char str[10] = "Tom", *ptr = new char[10];
63     strcpy(ptr, "Jerry");
64     PointName pn1(str), pn2(ptr), pn3("Snoopy");
65     pn1.Show();
66     pn2.Show();
67     pn3.Show();
68     cout << "Do something..." << endl;
69     strcpy(str, "Winnie"); // 这个修改不影响对象 pn1
70     pn1.Show();
71     delete [] ptr; // 这个修改不影响对象 pn2
72     pn2.Show();
73     pn3.UpperName(); // 对象属性的修改由自己作主
74     pn3.Show();
75     PointName pn4 = pn1; // 深拷贝构造 pn4, pn4 有自己的资源空间
76     pn2 = pn1; // 深赋值运算, 重构 pn2 的资源
77                         // 或 pn2.operator=(pn1);
78     cout << "Return to Operating System." << endl;
79     return 0;
80 }
```

程序的运行结果：

```

Constructing an object of PointName "Tom".
Constructing an object of PointName "Jerry".
Constructing an object of PointName "Snoopy".
"Tom"
"Jerry"
"Snoopy"
Do something...
"Tom"
"Jerry"
Upper "Snoopy"->"SNOOPY".
"SNOOPY"
COPY constructing an object of PointName "Tom".
ASSIGN an object of PointName "Tom".
Return to Operating System.
Destructing an object of PointName "Tom".
Destructing an object of PointName "SNOOPY".
Destructing an object of PointName "Tom".
Destructing an object of PointName "Tom".
```

请读者分析程序的运行结果。特别注意深拷贝构造函数、深赋值运算、析构函数的作用和具体的执行情况。

最后需强调的是，在类的设计中，若类的对象可能带有资源，则应该定义深拷贝构造函数；重载赋值运算符实现深赋值运算；定义析构函数释放资源。

10.5 组合成员的构造

既然类是一种数据类型，其对象便可以作为另一个类的数据成员。即作为类的数据成员者是另一个类的对象，称这种情形为类的组合，该数据成员被称为组合成员。一个很自然的问题是组合成员的数据成员由谁来构造，答案也是很自然的——自己的事情自己做。这是 C++ 的一个基本原则。接下来的问题是如何实现自己的事情自己做，亦即何时调用组合成员的构造函数，构造函数的参数如何转递给组合成员的构造函数？

10.5.1 成员的构造时机

定义变量的同时给该变量确定初值的过程是初始化。也就是说，在构造变量空间的同时，就为该空间各个字节的各个位确定了指定的初始状态。

```
double x=3.1415926; // 定义变量, 显式初始化。"=" 不是赋值运算符
double y;           // 定义变量, 隐式初始化
y = 3.1415926;     // 执行赋值运算, 修改变量的值
```

其实，定义变量中系统在分配存储单元空间时对变量进行了“初始化”（对局部自动变量可能“不作为”）。上面的变量 x 的初始化值由程序员指定为 3.1415926；变量 y 的初始化值按系统约定（存储类型为全局、静态全局和静态局部者默认初始化为各位全为 0；其他存储类型变量的初始化值不可预知）。上面的语句 y = 3.1415926；是一条赋值语句，此时的变量 y 已经存在，赋值语句修改了变量 y 原有的值，不属于初始化。

我们先回顾前面所设计的构造函数，如第 10.1.2 小节例 10.1 中的代码。

```
8 Student::Student(char *Id, string Name, char Gender, int Age)
9 {
10    strncpy(id, Id, sizeof(id)); id[sizeof(id)-1] = '\0';
11    name = Name;
12    gender = (Gender=='M' || Gender=='m')? 'm' : 'f';
13    age = Age;
14 }
15 Student::Student(string Id, char *pName, char Gender, int Age)
16 {
17    strncpy(id, Id.c_str(), sizeof(id)); id[sizeof(id)-1]='\0';
18    name = pName;                      // 自动转换后赋值
19    gender = (Gender=='M' || Gender=='m')? 'm' : 'f';
20    age = Age;
21 }
```

粗略地，我们说构造函数“完成对象数据空间处理、构造结构、初始化数据成员”。但实际上，上面的构造函数应该理解为“完成对象数据空间处理、构造结构、初始化数据成员，再修改数据成员的值”。因为，当进入到构造函数的函数体时，编译系统已经“完成对象数据空间处理、构造结构、初始化数据成员”。那么编译系统完成这些事情的时机何在？用 C++ 语句表述：其时机应该在构造函数首部与左花括号之间！我们赞叹 C++ 语言设计者的智慧！

准确地说，在进入到构造函数或拷贝构造函数体左花括号之前，对象数据成员业已构造并初始化完毕。进入函数体后所进行的赋值运算是调整、修改数据成员的值。

10.5.2 组合成员的构造——冒号语法

分析了类的数据成员的初始化时机，并指出了 C++ 语句表述的位置。C++ 具体的语法格式是冒号语法，C++ 规定：

(1) 先调用组合成员的构造函数，构造组合成员。若有多个组合成员，则依照它们在数据成员描述时声明的顺序构造之，与冒号后面书写顺序无关。

(2) 再进入本类的构造函数的函数体，执行函数体语句，以调整、修改成员的值。

冒号语法只能用在构造函数、拷贝构造函数中，位置在函数首部与左花括号之间。构造函数中的冒号语法格式如下。

```
构造函数(总参数表) : 组合成员名1(参数表), 组合成员名2(参数表), ...
{
    // 函数体语句
}
```

对于那些显式的冒号语法中未提及的数据成员，或者没有使用显式的冒号语法时，系统自动采用隐式的冒号语法，即自动调用相应的默认构造函数构造并初始化数据成员。

析构含组合成员的对象时，遵循析构与构造顺序相反的原则。

(1) 先执行本对象的析构函数体。

(2) 然后执行组合成员的析构函数。有多个组合成员时，按其声明时的相反顺序依次析构。

上面的构造函数最好写成如下形式。

```
8 Student::Student(char *Id, string Name, char Gender, int Age)
9             : name(Name), gender(Gender), age(Age)
10 {
11     strncpy(id, Id, sizeof(id)); id[sizeof(id)-1] = '\0';
12     gender = (gender=='M' || gender=='m')? 'm' : 'f';
13 }
14 Student::Student(string Id, char *pName, char Gender, int Age)
15             : name(pName), gender(Gender), age(Age)
16 {
17     strncpy(id, Id.c_str(), sizeof(id)); id[sizeof(id)-1]='\0';
18     gender = (gender=='M' || gender=='m')? 'm' : 'f';
19 }
```

从上面修改后的构造函数来看，其效率更高，因为它们对数据成员name, gender, age实行了真正的初始化，成员name, age 的构造一步到位。成员gender 需要调整其值（考虑到兼容实参'M' 及容错既非'm' 又非'f' 的字符）。数组id 的元素尚不能一步到位。

另外，原来的构造函数中使用string 类的深赋值运算操作修改组合成员的值亦不符合“自己的事情自己做”的设计原则。与此类似的还有，如第 9 章的复数类中，虽有Set 函数可用于设置私有数据成员的值。若新声明的类中包含该复数类的对象作为组合成员，我们也不提倡先构造组合成员，再用 Set 函数修改其组合成员值的做法。因为这是“你的事情由我来做”的不明之举。

这里，还看到基本数据类型成员（如age）能使用冒号语法进行初始化（注意圆括号格式）。

从上述程序行第 9 行可见, `Student` 类的构造函数作为“二传手”将参数 `Name` 转递给 `string` 类的拷贝构造函数深拷贝构造组合成员 `name`。第 15 行, `Student` 类的构造函数作为“二传手”将参数 `pName` 转递给 `string` 类的转换构造函数构造组合成员 `name`。

在上面的例子中, 显式使用或隐式使用冒号语法(例 10.1)都能够达到同样的效果, 它们在效率上略有差别。必须指出, 在一些情况下, 必须显式使用冒号语法:

(1) 初始化类的常量数据成员(因为常量必须在定义时初始化。常量不能先定义, 后赋值)。

(2) 初始化类的引用成员声明(引用是其目标的别名, 必须在声明时绑定目标, 且不能改变绑定)。

(3) 组合成员所属的类没有默认的构造函数时, 构造组合成员必须带参数, 这个参数只能通过冒号语法传递。

例 10.3 使用冒号语法示例。本程序仅说明冒号语法的作用, 无实际意义。

```

1 // ColonRule.cpp
2 #include <iostream>
3 #include <iomanip>
4 #include <ctime>
5 using namespace std;
6
7 class A    // 无默认构造函数, 创建对象必须提供参数
8 {
9 public:
10    A(int x) : a(x) {}
11 private:
12    int a;
13 };
14
15 class ColonRule
16 {
17 public:
18    ColonRule(int size=5) : n(size), len(n), num(size)
19    {                               // 这三个量都必须用冒号语法
20        x = new double[(n>0)?n:1]; // 使其至少有一个元素
21        x[0] = (rand()%101)/10.0;
22        for(int i=1; i<n; i++)
23            x[i] = (rand()%101)/10.0;
24    }
25    ColonRule(const ColonRule &cr)
26        : n(cr.n), len(n), num(cr.num)
27    {                               // 这三个量都必须用冒号语法
28        x = new double[(n>0)?n:1]; // 使其至少有一个元素
29        x[0] = cr.x[0];
30        for(int i=1; i<n; i++)
31            x[i] = cr.x[i];
32    }
33    ColonRule & operator=(const ColonRule &cr)
34    {                               // n 为常量, 资源空间大小不便改变
35        int i;
36        x[0] = cr.x[0];
37        for(i=1; i<n && i<cr.n; i++) // 实际复制的数据个数受限制
38            x[i] = cr.x[i];
39        for( ; i<n; i++)           // 可能有剩余, 全置 0
40            x[i] = 0;

```

```

41         return *this;
42     }
43     ~ColonRule()
44     {
45         delete [] x;
46     }
47     void Show() const
48     {
49         cout << "n=" << setw(3) << n << ":" ;
50         cout << fixed << setprecision(1)
51             << setw(4) << x[0];      // 至少输出一个数据
52         for(int i=1; i<len; i++)
53             cout << "," << setw(4) << x[i];
54         cout << ")" << endl;
55     }
56 private:
57     const int n;           // 常量数据成员
58     const int &len;        // 引用成员
59     A num;                // 组合成员。类 A 无默认构造函数
60     double *x;
61 };
62 int main()
63 {
64     time_t t;
65     srand(time(&t));
66     ColonRule a, b(8), c(-1), d(a); //不同的对象可以有不同的常量值
67     cout << "a.Show();"; a.Show();
68     cout << "b.Show();"; b.Show();
69     cout << "c.Show();"; c.Show();
70     cout << "由对象 a 深拷贝构造对象 d 的结果" << endl;
71     cout << "d.Show();"; d.Show();
72     a = b;
73     b = d;
74     cout << "执行赋值运算 a=b; 深赋值的结果" << endl;
75     cout << "a.Show();"; a.Show();
76     cout << "执行赋值运算 b=d; 深赋值的结果" << endl;
77     cout << "b.Show();"; b.Show();
78     return 0;
79 }

```

程序的运行结果（其中的一些数据是随机产生的）：

```

a.Show(); n = 5:( 8.2, 8.1, 4.3, 6.0, 7.3)
b.Show(); n = 8:( 9.7, 9.2, 6.9, 3.8, 4.7, 0.4, 6.8, 7.9)
c.Show(); n = -1:( 8.7)
由对象 a 深拷贝构造对象 d 的结果
d.Show(); n = 5:( 8.2, 8.1, 4.3, 6.0, 7.3)
执行赋值运算 a = b; 深赋值的结果
a.Show(); n = 5:( 9.7, 9.2, 6.9, 3.8, 4.7)
执行赋值运算 b = d; 深赋值的结果
b.Show(); n = 8:( 8.2, 8.1, 4.3, 6.0, 7.3, 0.0, 0.0, 0.0)

```

本程序还说明，不同对象的常量成员的值可以不同，引用所绑定的目标也随对象不同而异。这种不能更改的指定或绑定只能通过显式的冒号语法实现。

请注意构造函数及拷贝构造函数中均应该用 `len(n)` 使常量引用 `len` 绑定本对象自己的常量 `n`。尤其是拷贝构造函数中，不应使用 `len(cr.len)` 来绑定其他对象的常量 `n`。因

为其他对象可能先于本拷贝构造的对象析构，这将使 len 绑定的目标先消失而可能导致错误。

如果去掉本程序中的赋值运算符函数定义，仅使用由系统提供的默认赋值运算符，则上述程序的第 73、74 行出现编译错（常量成员间赋值错）。在定义的赋值运算符函数中，也只能赋值能够复制的部分（由左值对象的常量 n 及右值对象的常量 n 值的大小决定）。

本节最后介绍 C++ 字符串类 `string` 的对象为成员的名字类。

例 10.4 类 `string` 的对象做组合数据成员（正确的程序。本例仅展示拷贝构造、赋值运算和析构函数依赖于成员的拷贝构造、赋值运算的实现）。

C++ 提供的字符串类 `string` 是一个设计完善的类。该类的对象是带资源的，自然地，`string` 类有深拷贝构造函数、深赋值运算符和析构函数。该类还重载了关系运算符、I/O 操作符和字符串拼接（+）等其他运算符。现在我们讨论用 `string` 类的对象作为组合成员的名字类。

```

1 // StringName.cpp
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 class StringName
7 {
8 public:
9     StringName(const char *pName="NoName") : name(pName)
10    {                                     // 冒号语法。调用 string 类的转换构造函数
11        cout << "Constructing an object of StringName | "
12        << name << "." << endl;
13    }
14    StringName(const StringName &sn) : name(sn.name)
15    {                                     // 冒号语法。调用 string 类的深拷贝构造函数
16        cout << "COPY constructing an object of StringName | "
17        << name << "." << endl;
18    }
19    StringName & operator=(const StringName &sn)
20    {
21        name = sn.name; // 赋值运算。调用 string 类的深赋值运算。
22        return *this;
23    }
24    ~StringName()           // 析构函数。自动调用 string 类的析构函数
25    {
26        cout << "Destructing an object of StringName | "
27        << name << "." << endl;
28    }
29    void ChangeName(const char *pName)
30    {
31        cout << "Change StringName | " << name;
32        name = pName;
33        cout << "| ->| " << name << "." << endl;
34    }
35    void Show() const
36    {
37        cout << "StringName: | " << name << "." << endl;
38    }
39 private:

```

```

40     string name;
41 };
42
43 StringName fSN(StringName x, StringName *p, StringName &r)
44 {
45     cout << "In fSN()..." << endl;
46     x.ChangeName("Architecture");
47     p->ChangeName("Software");
48     r.ChangeName("Application");
49     x.Show();
50     p->Show();
51     r.Show();
52     cout << "Return from fSN()..." << endl;
53     return r;
54 }
55
56 int main()
57 {
58     StringName sn1("Java Program");
59     StringName sn2("C++ Program");
60     StringName sn3("C# Program");
61
62     cout << "\nBegin fSN()..." << endl;
63     fSN(sn1, &sn2, sn3);
64     cout << "End of fSN()\n" << endl;
65     sn1.Show();
66     sn2.Show();
67     sn3.Show();
68     sn2 = sn1;
69     cout << "\nAfter assign sn2=sn1;" << endl;
70     sn2.Show();
71     cout << "Return to Operating System." << endl;
72     return 0;
73 }

```

程序的运行结果：

```

Constructing an object of StringName |Java Program|.
Constructing an object of StringName |C++ Program|.
Constructing an object of StringName |C# Program|.

Begin fSN() ...
COPY constructing an object of StringName |Java Program|.
In fSN() ...
Change StringName |Java Program|->|Architecture|.
Change StringName |C++ Program|->|Software|.
Change StringName |C# Program|->|Application|.
StringName: |Architecture|.
StringName: |Software|.
StringName: |Application|.
Return from fSN() ...
COPY constructing an object of StringName |Application|.
Destructing an object of StringName |Application|.
Destructing an object of StringName |Architecture|.
End of fSN() ...

StringName: |Java Program|.
StringName: |Software|.

```

```

StringName: |Application|.

After assign sn2 = sn1;
StringName: |Java Program|.
Return to Operating System.
Destructing an object of StringName |Application|.
Destructing an object of StringName |Java Program|.
Destructing an object of StringName |Java Program|.

```

Visual C++ 实现的 C++ 字符串 `string` 类对象的尺寸（基本空间尺寸）为 `sizeof(string)` 等于 16 字节，它除记录了存放字符串的首地址外，还记录了字符串的长度等。而 GCC 实现的 C++ 字符串 `string` 类对象的尺寸（基本空间尺寸）为 `sizeof(string)` 等于 4 字节，它将字符串的长度等数据成员藏于其内部结构体中，真正的字符串内容是资源的资源。

在此，归纳三个设计正确的名字类的特点如下。

类名	对象基本空间	对象资源空间
ArrayName	10 字节	无
PointName	4 字节	堆内存
StringName	<code>sizeof(string)</code> 字节	堆内存

`ArrayName` 类的每个对象都不带资源，只能存放长度不超过 9 个字符的字符串，超长部分被截断。该类仅需要系统提供的默认拷贝构造函数、默认赋值运算符函数和默认析构函数即可。

`PointName` 类的对象是带资源的，资源空间用于存放字符串的内容；资源的首地址存放在对象的基本空间中（私有的指针成员）。所存储的字符串长度几乎不受限制，因而不必截断。该类必须定义深拷贝构造函数、深赋值运算符函数和析构函数。

`StringName` 类的对象不直接带资源（其资源由组合成员间接地拥有，所存储的字符串长度也几乎不受限制）。因此 `StringName` 类可以直接利用系统所提供的默认的拷贝构造函数、默认的赋值运算符函数、默认的析构函数而不必重新定义，即删除拷贝构造函数、赋值运算符函数和析构函数（请读者在计算机上进行相应的测试）。全仰仗于 `string` 类的较完善的功能设计和 C++ 语言的机制：系统会根据组合成员的拷贝构造、赋值运算、析构而自动调用 `string` 类的深拷贝构造函数、深赋值运算符函数、析构函数进行相应的操作。

可见，几乎可以将一个设计完善的类像使用基本数据类型一样地使用它，即使该类的对象带有资源。在设计一个类时，应该使其尽善尽美。

10.6 趣味程序——模拟银行打印储户存折

问题描述 本例设计一个类，用于处理多个储户的存取款操作，最后输出所有数据。为简单起见，每次输出均从换折记录开始，假定一个存折最多可以记录 MAX 次存取款活动情况（调试程序时取较小的值，如 `MAX = 5`）。采用多文件结构的源程序如下。

源代码 10.1 模拟银行打印储户存折（头文件·类声明）

```

1 // Bank.h
2 #ifndef BANK_H
3 #define BANK_H
4 #include <string>
5 using namespace std;
6 const int MAX = 5;    // 每本存折最大记录条数，较小的值供调试
7
8 class Bank
9 {
10 public:
11     Bank(const char *AccountName, const char *AccountNumber,
12           const char *Date, double Money=0);           // 开户
13     void Deposit(const char *Date, double Money);   // 存款
14     void Withdraw(const char *Date, double Money);  // 取款
15     void Print() const;                            // 打印
16 private:
17     string name, num;                           // 户名，账号
18     int booknum, top;                          // 存折编号，本册记录号
19     struct Record;                           // 内部结构体（可以无名）
20 {
21     string date;                            // 操作日期
22     double money;                          // 本次金额
23     double balance;                         // 余额
24 } record[MAX];                           // 记录业务活动
25 };
26 #endif

```

源代码 10.2 模拟银行打印储户存折（源程序文件·类的内部实现）

```

1 // Bank.cpp
2 #include <iostream>
3 #include <iomanip>
4 #include "Bank.h"
5
6 Bank::Bank(const char *AccountName, const char *AccountNumber,
7             const char *Date, double Money)
8 : name(AccountName), num(AccountNumber), booknum(1), top(1)
9 {
10     record[0].date = Date;
11     record[0].money = Money;
12     record[0].balance = Money;
13 }
14
15 void Bank::Deposit(const char *Date, double Money)
16 {
17     record[top].date = Date;
18     record[top].money = Money;
19     record[top].balance = record[top-1].balance + Money;
20     if(++top==MAX)      // 记录存满，换折处理
21     {
22         Print();          // 打印
23         booknum++;
24         record[0].date = Date;
25         record[0].money = 0;
26         record[0].balance = record[MAX-1].balance;
27         top = 1;

```

```

28     }
29 }
30
31 void Bank::Withdraw(const char *Date, double Money)
32 {
33     record[top].date = Date;
34     record[top].money = -Money;
35     record[top].balance = record[top-1].balance - Money;
36     if(++top==MAX)      // 记录存满, 换折处理
37     {
38         Print();        // 打印
39         booknum++;
40         record[0].date = Date;
41         record[0].money = 0;
42         record[0].balance = record[MAX-1].balance;
43         top = 1;
44     }
45 }
46
47 void Bank::Print() const
48 {
49     cout << "\n户名:" << name << "账号:" << num
50     << "存折序号:" << booknum;
51     cout << "\n序号 日期 存入 余额"
52     << "取出" << endl;
53     for(int i=0; i<top; i++)
54     {
55         if(i==0)
56             cout << (booknum==1 ? "开户" : "换折");
57         else
58             cout << setw(5) << i+1;
59         cout << setw(12) << record[i].date;
60         if(record[i].money>0)
61             cout << setw(10)<<record[i].money <<setw(10)<<' ';
62         else if(record[i].money<0)
63             cout << setw(10)<<' '<<setw(10)<<-record[i].money;
64         else
65             cout << setw(20) << ' ';
66         cout << setw(10) << record[i].balance << endl;
67     }
68 }

```

源代码 10.3 模拟银行打印储户存折（源程序文件·主函数）

```

1 // main.cpp
2 #include "Bank.h"
3
4 int main()
5 {
6     Bank Zhang("张三", "0000372100003721", "2008.02.05", 1000);
7     Bank Li    ("李四", "0123456789876543", "2008.02.08", 1500);
8
9     Li.Withdraw  ("2008.03.01", 600);
10    Zhang.Deposit ("2008.03.02", 2000);
11    Li.Deposit   ("2008.03.03", 1000);
12    Zhang.Withdraw("2008.04.01", 500);
13    Zhang.Withdraw("2008.04.05", 800);
14    Zhang.Deposit ("2008.04.20", 1000);

```

```

15     Zhang.Withdraw("2008.05.08", 400);
16     Zhang.Deposit ("2008.05.20", 1000);
17
18     Zhang.Print();
19     Li.Print();
20     return 0;
21 }

```

程序的运行结果：

户名：张三	账号：0000372100003721	存折序号：1		
序号	日期	存入	取出	余额
开户	2008.02.05	1000		1000
2	2008.03.02	2000		3000
3	2008.04.01		500	2500
4	2008.04.05		800	1700
5	2008.04.20	1000		2700
户名：张三	账号：0000372100003721	存折序号：2		
序号	日期	存入	取出	余额
换折	2008.04.20			2700
2	2008.05.08		400	2300
3	2008.05.20	1000		3300
户名：李四	账号：0123456789876543	存折序号：1		
序号	日期	存入	取出	余额
开户	2008.02.08	1500		1500
2	2008.03.01		600	900
3	2008.03.03	1000		1900

10.7 小结

本章介绍的对象构造过程中的若干问题是面向对象程序设计的重要基础，是本课程教学工作的重要内容。

构造函数的形式多样“丰富多彩”，赋值运算符函数也可以“丰富多彩”。拷贝构造函数、析构函数均有点“深沉”。数据成员尤其是组合成员的初始化采用“十分巧妙”的位置表达了其构造的时机。对象的数据成员的构造总是通过冒号语法（显式地或默认地）进行的。析构函数的调用次序与构造函数的恰好相反。

构造函数（特别是默认的构造函数、转换构造函数）、拷贝构造函数和赋值运算符函数的功能及用法，令我们有理由认为 C++ 的机制会使类类型向基本数据类型靠拢。基本数据类型变量的初始化由一般的 `int a=3, b=a;` 格式发展到 `int a(3), b(a);` 格式也使我们看到基本数据类型融合类类型特征的迹象，似乎基本数据类型也有其默认构造函数、转换构造函数和拷贝构造函数等。

阅读理解程序能力的培养是本课程教学工作中的一个重要方面。提高阅读理解程序的能力需要读者积极主动，多思勤练。建议读者在阅读 C++ 程序时采用“扫读”的方法，扫读的基本顺序如下。

- ① 浏览整个程序的结构、概貌及大致功能。

- ② “扫描”类的数据成员（特别注意是否有指针成员）。
- ③ 构造函数、拷贝构造函数、析构函数和赋值运算符函数。
- ④ 其他成员函数。
- ⑤ 对于应用或测试程序，注意程序中的“精灵”——类的对象的生命期（何时创建，如何自我表现，何时销毁）。

练习 10

1. 指出下列各程序片段中的错误

(1)

```

1 #include <iostream>
2 using namespace std;
3
4 class Test
5 {
6 public:
7     void Test(int x=0) {a = x;}
8     Test(const Test &t) { a = t; }
9     ~Test(int x) { delete a; }
10    void Set(int x) : a(x) {}
11    int Get() {return a;}
12 private:
13     int a;
14 };

```

(2)

```

1 #include <iostream>
2 using namespace std;
3
4 class A
5 {
6 public:
7     A(int x) {a = x;}
8 private:
9     int a;
10 };
11
12 class B
13 {
14 public:
15     B(int x=0) {a.a = x;}
16     B(const B &x) {a.a = x.a;}
17 private:
18     A a;
19 };

```

2. 基本题

- (1) 设计一个人民币类 (class RMB;) 其中有表示“元、角、分”的数据成员 int yuan, jiao, fen;。要求编写默认的构造函数；转换构造函数（形式参数为以“元”为单位的双精度浮点型金额值）；三个参数的构造函数，其他必需的基本成员函数。

(2) 阅读下面的程序，指出程序的运行结果。

```

1 #include <iostream>
2 using namespace std;
3
4 class Sample
5 {
6 public:
7     Sample() { x = y = 0; }
8     Sample(int a, int b) { x = a; y = b; }
9     ~Sample()
10    {
11        if(x==y)
12            cout << "x==y" << endl;
13        else
14            cout << "x!=y" << endl;
15    }
16    void Display() const
17    {
18        cout << "x=" << x
19        << ",y=" << y << endl;
20    }
21 private:
22     int x, y;
23 };
24
25 int main()
26 {
27     Sample s1, s2(2, 3);
28     s1.Display();
29     s2.Display();
30     return 0;
31 }
```

(3) 设计一个雇员类 Employee，该类中有表示姓名（name）、地址（addr）和邮政编码（zip）的字符数组。要求有默认的构造函数、转换构造函数和其他构造函数，定义 ChangeName 和 Display 等函数，函数的参数自定。测试用的主函数如下。

```

1 #include <iostream>
2 #include "Employee.h"
3 using namespace std;
4
5 int main()
6 {
7     Employee s("张三", "中山路18号", "200020");
8     Employee staff[3] = {"李四", s,
9                           Employee("王五", "中南路100号")};
10
11    staff[1].ChangeName("赵六");
12    s.Display();
13    for(int i=0; i<3; i++)
14        staff[i].Display();
15    return 0;
16 }
```

(4) 将上题数据成员的数据类型改成字符型指针，重新编制 Employee 类。测试的主函数同上。

(5) 将上题数据成员的数据类型改成 C++ 字符串类型 `string`, 重新编制 `Employee` 类。测试的主函数同上。

(6) 阅读下面的程序, 指出程序的运行结果。

```

1 #include <iostream>
2 using namespace std;
3
4 class A
5 {
6 public:
7     A(int x=0) : a(x)
8     {
9         cout << "Constructing an object of class A (" 
10            << a << ")" << endl;
11    }
12    A(const A &x) : a(x.a)
13    {
14        cout << "COPY constructing an object of class A (" 
15            << a << ")" << endl;
16    }
17    void Show() const
18    {
19        cout << "class A, (" << a << ")" << endl;
20    }
21
22 private:
23     int a;
24 };
25
26 class A1
27 {
28 public:
29     A1(int x=0) : a(x)
30     {
31         cout << "Constructing an object of class A1 (" 
32            << a << ")" << endl;
33    }
34    A1(const A1 &x) : a(x.a)
35    {
36        cout << "COPY constructing an object of class A1 (" 
37            << a << ")" << endl;
38    }
39
40    void Show()
41    {
42        cout << "class A1, (" << a << ")" << endl;
43    }
44
45 private:
46     int a;
47 };
48
49 class B
50 {
51 public:
52     B(int x=0, int y=0) : data1(y), data(x) {}
53     void Show()
54     {

```

```
55         data.Show();
56         data1.Show();
57     }
58
59 private:
60     A data;
61     A1 data1;
62 };
63
64 int main()
65 {
66     B x(10, 20), y(x);
67     x.Show();
68     y.Show();
69     return 0;
70 }
```

第 11 章 静态成员及友元

本章讨论类的静态成员，它与整个类相关，不默认联系具体对象（即只认类型不认对象）。静态成员是 C++ 封装全局变量，以达到消灭全局变量目的的重要机制。本章讨论的友元是 C++ 的另一个机制，其目的是为了提高程序的执行效率。友元不是类的成员，它仅是类的朋友。学习本章后，读者应该懂得为什么需要静态成员、友元；掌握静态数据成员在何处定义、初始化及其生命期；掌握静态成员函数处理具体对象的方法；掌握友元函数、友元类相关的声明、定义和使用方法。

11.1 静态成员

在面向过程程序设计部分，曾指出过不提倡在程序中使用全局变量，也曾提到过 C++ 的一些机制可以彻底消灭程序中的全局变量。然而在某些情况下，全局变量确实给程序设计带来方便。显然，全局变量也是数据的载体，亦即它也是与某问题相联系的，对全局变量的操作也具有所论问题的特定性。由于“对象 = (算法 + 数据结构)”，我们希望能将程序中的全局变量封装在类里面，成为类的所有对象所共享的量。当然，封装在类中的这种量不再是全局变量。这样，通过严密的封装既能使类的安全性、可移植性得到保障，而被封装在类中的“共享变量”又能给应用程序带来便利。

在程序中，一个全局变量可以被许多函数直接访问。大多数情况下，一个全局变量是一类事物的共用属性。例如，考察一个班级，该班的人数、班长的姓名、班导师姓名等信息是全班学生共用的属性。并不是每一个学生专有一个班长、专有一个班导师。这种共有属性如何封装？请先看一种不好的设计：

```
1 #include <string>
2 using namespace std;
3 class Student
4 {
5 public:
6     // 略
7 private:
8     string id, name;           // 学生对象本人的学号、姓名
9     string monitor, tutor;    // 该班的班长、班导师姓名
10    int num;                  // 该班的学生人数
11 };
12
13 int main()
14 {
15     Student Zhang, Li, array[8]; // 共创建 10 个对象，学生人数为 10
16     Student *p = new Student;   // 第 11 名同学，学生总人数应该增 1
17     // ...
18     delete p;                 // 学生总人数应该减 1
19     // ...
20     return 0;
21 }
```

显然，这样的设计有两个大问题：

① 数据冗余度大（每个对象的数据空间均存放有全班人数 num、班长姓名 monitor 和班导师姓名 tutor）。

② 数据的一致性难以保证（当该班的学生人数变化、班长易人或更换班导师时，需要给每个对象发送消息令其修改各自的属性）。关于数据的一致性保证是费时的、困难的，甚至是无法实现的。例如上面代码中第 17 行增加了一名学生，此时并无一个机制发送消息通知以前的所有十个对象，令其修改其 num 成员；第 19 行删除一名学生导致人数变化也是类似的。对于这种情况，只能由程序员另写语句进行处理。因此，上述设计方案是不可取的。仔细分析这些属性的特点，得出我们所希望的解决方案，如表 11-1 所示。

表 11-1 对象的属性与类的属性

属性	特点	解决方案
学号 姓名	对象的属性（每个对象特有）	占对象数据空间（已解决）
人数 班长 班导师	类的属性（全体对象共用）	仅需要一份且与对象分离 不占用对象的数据空间 (呼唤新机制)

既然所讨论问题的属性有“每个具体对象特有的”与“全体对象共用的”之分，又希望将这些属性封装进类中（拒绝全局变量便于程序移植等），必须有新的机制。

C++ 提供静态成员机制便是为了解决上述问题的。C++ 类中的静态成员仍借用保留字 static，与以前的静态全局变量、静态局部变量及静态函数的概念无任何关联。类的静态成员有静态数据成员和静态成员函数之分。

11.1.1 静态数据成员

类声明时，在所描述的数据成员的数据类型前加保留字 static，则该数据成员便是该类全体对象所共用的属性，被称为静态数据成员。编译系统只为类的每个静态数据成员创建一份，存放于计算机内存的全局数据区，它不占具体对象的数据空间（请回顾第 9.2.2 小节对象的基本数据空间只包含非静态数据成员）。

1. 静态数据成员的定义及初始化

静态数据成员的创建和销毁处分权不属于任何对象。对象有权利访问（读或写）静态成员，更有义务维护静态数据成员。静态数据成员的改变可能影响到该类的所有对象。

类体中描述静态数据成员是数据组织形式的描述，堪称“静态数据成员参照访问声明”。一个很自然的要求：在创建具体对象时应该可以访问到静态数据成员。这就要求静态数据成员在创建类的任何对象之前就已经存在（犹如全局变量、全局对象在 main 函数执行前先创建一样）。因此静态数据成员应该也在主函数之前先行定义及初始化。

静态数据成员不能没有定义，也不能重复定义。与以前处理全局变量时一样，将类的静态数据成员定义及初始化语句放在类的内部实现源程序文件中，格式为

数据类型 类名::静态数据成员名 = 初始值；

或者

数据类型 类名::静态数据成员名(初始化值);

此处不能再写保留字 static。

2. 访问静态数据成员

静态数据成员的生命期是全局的，与是否创建了对象无关。当有具体对象存在时可以采用对象访问成员的方式访问类的静态数据成员（当然还取决于该静态数据成员的访问控制属性）。

对象名.静态数据成员名

此时系统并不关心是哪个对象访问了静态数据成员而只关心对象所在的类。即只认类型，不认对象。访问静态数据成员的更一般方法是只用类名不用对象名。

类名::静态数据成员名

其中须用作用域区分符“::”。

11.1.2 静态成员函数

有些处理方法（成员函数）并不一定针对具体的某个对象而是针对整个类的。例如：在没有任何对象存在（即尚未创建任何具体对象时），且类的静态数据成员是受保护的或私有的情况下，要访问该静态数据成员只能通过公开的被称为静态成员函数的函数来实现。这说明静态成员函数机制是必要的。

类体中声明静态成员函数的语法格式是

static 返回类型 静态成员函数名(形式参数表);

在类体外定义静态成员函数的方法与定义非静态成员函数的方法一样。注意：不能添加保留字 static。

静态成员函数与非静态成员函数一样均存放在计算机内存的代码区，被该类所有对象共享。它们之间的不同之处在于静态成员函数不隐含与任何具体对象联系。亦即静态成员函数没有隐含传递所谓“本对象地址”的指针形式参数 this。若需要静态成员函数在类的层次高度处理某个具体对象，则需要将该对象（或对象的地址、或对象的引用）作为实参进行显式地传递。

与访问静态数据成员时相似，可以通过具体对象访问成员的方式调用静态成员函数

对象名.静态成员函数名(实际参数表)

因为静态成员函数不可能隐含地接收对象的地址，此时，静态成员函数仍然“只认类型，不认对象”。更为一般的调用静态成员函数的格式是

类名::静态成员函数名(实际参数表)

例 11.1 学生类 (Student) 中的静态成员及处理方法。

(1) 属性

- ① 非静态数据成员：学号 (id)、姓名 (name)。
- ② 静态数据成员：学生人数 (num)、班长 (monitor)、班导师 (tutor)。

(2) 行为

- ① 非静态成员函数：构造函数、拷贝构造函数、析构函数、输出信息。
- ② 静态成员函数：获取学生人数，修改、获取班长姓名、班导师姓名。

【分析】 作为静态数据成员学生人数 num 应该在定义时初始化为 0。然后，num 在创建对象、析构对象时发生改变。由于创建对象的途径有构造和拷贝构造两种。因此，学生人数由构造函数或拷贝构造函数随对象创建而增 1，由析构函数随某对象销毁而减 1。由此可见，本类的对象虽不带资源，但定义拷贝构造函数和析构函数却是必须的。因为赋值运算不创建对象，不改变学生人数，故不需要重载赋值运算符函数。

源代码 11.1 静态成员示例程序

```

1 // Student.h
2 #ifndef STUDENT_H
3 #define STUDENT_H
4 #include <string>
5 using namespace std;
6
7 class Student
8 {
9 public:
10    Student(string Id, string Name);
11    Student(const Student &s);           // 必须有拷贝构造函数
12    ~Student();                         // 必须有析构函数
13    void Show() const;
14    static int GetNum();                // 静态成员函数 (下同)
15    static string GetMonitor();
16    static string GetTutor();
17    static void SetMonitor(string Monitor);
18    static void SetTutor(string Tutor);
19 private:
20    string id, name;
21    static string monitor, tutor; // 静态数据成员 (下同)
22    static int num;
23 };
24 #endif

```

```

1 // Student.cpp
2 #include <iostream>
3 #include "Student.h"
4 using namespace std;
5
6 int Student::num = 0;           // 定义静态数据成员并初始化
7 string Student::monitor = ""; // 没有对象，班长空缺
8 string Student::tutor = "张三"; // 可以先指定班导师
9
10 Student::Student(string Id, string Name) : id(Id), name(Name)
11 {
12     cout << "构造对象" << name

```

```

13         << " " << ++num << ")" << endl; // 创建对象 num 增 1
14     }
15 Student::Student(const Student &s) : id(s.id), name(s.name)
16 {
17     cout << "拷贝构造对象" << name
18     << " " << ++num << ")" << endl; // 创建对象 num 增 1
19 }
20 Student::~Student()
21 {
22     cout << "析构对象" << name
23     << " " << --num << ")" << endl; // 销毁对象 num 减 1
24 }
25 void Student::Show() const
26 {
27     cout << "学号: " << id << ", 姓名: " << name << endl;
28 }
29 int Student::GetNum()
30 {
31     return num;
32 }
33 string Student::GetMonitor()
34 {
35     return monitor;
36 }
37 string Student::GetTutor()
38 {
39     return tutor;
40 }
41 void Student::SetMonitor(string Monitor)
42 {
43     monitor = Monitor;
44 }
45 void Student::SetTutor(string Tutor)
46 {
47     tutor = Tutor;
48 }

```

```

1 // main.cpp
2 #include <iostream>
3 #include "Student.h"
4 using namespace std;
5 void Display()
6 {
7     cout << "学生人数: " << Student::GetNum()
8     << ", 班长: " << Student::GetMonitor()
9     << ", 班导师: " << Student::GetTutor()
10    << endl;
11 }
12 void f(Student s, Student *p, Student &r)
13 { // 拷贝构造 s
14     cout << "In f()..." << endl;
15     static Student Li("004", "Li"); // 静态局部对象
16     Student x("005", "Xiao"); // 局部自动对象
17     cout << "Return from f()..." << endl;
18 }
19 int main()
20 {
21     Display(); // 未创建任何对象时

```

```

22     Student Zhao("001", "Zhao"), Qian("002", "Qian");
23     Student *p = new Student("003", "Sun"); // 动态对象（堆对象）
24     Student::SetMonitor("Qian"); // 设置班长
25     cout << "Begin_of_f()..." << endl;
26     f(Zhao, &Qian, *p); // 第一次调用 f 函数
27     cout << "End_of_f()..." << endl;
28     cout << "Begin_of_f()_again..." << endl;
29     f(Zhao, &Qian, *p); // 第二次调用 f 函数
30     cout << "End_of_f()..." << endl;
31     p->Show();
32     delete p; // 释放堆对象
33     Zhao.SetTutor("张珊"); // 更换班导师。不认对象 Zhao，只认 Zhao 的类型
34     Zhao.Show();
35     Display();
36     cout << "Return_to_Operating_System." << endl;
37     return 0;
38 }

```

程序的运行结果：

```

学生人数：0，班长：，班导师：张三
构造对象 Zhao (1)
构造对象 Qian (2)
构造对象 Sun (3)
Begin f() ...
拷贝构造对象 Zhao (4)
In f() ...
构造对象 Li (5)
构造对象 Xiao (6)
Return from f() ...
析构对象 Xiao (5)
析构对象 Zhao (4)
End of f() ...
Begin f() again ...
拷贝构造对象 Zhao (5)
In f() ...
构造对象 Xiao (6)
Return from f() ...
析构对象 Xiao (5)
析构对象 Zhao (4)
End of f() ...
学号：003，姓名：Sun
析构对象 Sun (3)
学号：001，姓名：Zhao
学生人数：3，班长：Qian，班导师：张珊
Return to Operating System.
析构对象 Qian (2)
析构对象 Zhao (1)
析构对象 Li (0)

```

请读者分析上述运行结果，特别注意如下几点。

- ① 静态数据成员的定义及初始化。
- ② 函数 f 传值型形式参数 (s)、局部自动对象 (x) 和静态局部对象 (Li) 的生命周期。
- ③ 主函数中动态对象 (*p) 的生命周期。

11.2 友元

利用类成员的 `protected` 及 `private` 访问控制属性能很好地屏蔽对象的内部细节，起到保护作用。通常将对象的属性（数据成员）加以保护，通过类的成员函数访问它们。

若对象带有大量数据，而访问这些数据时全都通过专门的成员函数周转，则将导致程序执行的效率不高。希望在保证数据安全的前提下适当开放数据的访问权限给指定的“朋友”，使之不通过成员函数周转就能直接访问类中的受保护的或私有的成员（包括数据成员和成员函数）。这就是提出友元机制的背景。友元不是类的成员，仅仅是类的朋友。友元分为友元函数和友元类。

11.2.1 友元函数

友元函数可以是一个普通的函数，也可以是另一个类的成员函数。在类声明中，在函数返回类型前使用保留字 `friend` 表示该函数不是本类的成员，而是本类的友元函数。友元函数可以在类体内声明并定义，也可以在类体内声明后，在类体外定义。在类体外定义时不能用保留字 `friend`，也不存在类名及作用域区分符 “`::`”，因为友元函数可以是一个普通的 C++ 函数（若该友元函数是另一个类的成员函数，则按其类的格式定义之）。

由于类的友元函数不是该类的成员函数，自然不默认联系具体的对象。友元函数访问类的具体对象及对象的成员时，应该将该对象（最好采用引用传递以避免拷贝构造对象的副本）作为实参显式地传递给友元函数。

例 11.2 矩阵与向量相乘。这个问题涉及如下两个类。

(1) 向量类 (`Vector`)

① 属性：向量维数 (`n`)，向量分量顺序存放的首地址（指针 `x`）及其资源。

② 行为：构造函数、深拷贝构造函数、深赋值运算符函数、析构函数、设置分量值和输出向量。

(2) 矩阵类 (`Matrix`)

① 属性：矩阵阶数 (`row, col`)，矩阵元素按行顺序存放的首地址（一级指针 `x`，并升维处理）及其资源。

② 行为：构造函数、深拷贝构造函数、深赋值运算符函数、析构函数、设置矩阵元素值和输出矩阵。

再设计一个矩阵乘向量的普通函数，并成为上述两个类共同的友元函数。程序采用多文件结构，由如表 11-2 所示的 5 个文件组成。

表 11-2 矩阵乘以向量程序的文件组成

文件名	说 明
<code>Vector.h</code>	向量类声明（头文件）
<code>Vector.cpp</code>	向量类内部实现（源程序文件）
<code>Matrix.h</code>	矩阵类声明（头文件）
<code>Matrix.cpp</code>	矩阵类内部实现（源程序文件）
<code>main.cpp</code>	矩阵乘向量测试程序（源程序文件）

具体源程序如下（请读者注意两个类的提前声明在两个头文件中的书写方法）。

源代码 11.2 矩阵乘向量（友元函数应用）

```

1 // Vector.h
2 #ifndef VECTOR_H
3 #define VECTOR_H
4
5 class Matrix; // 提前声明
6
7 class Vector
8 {
9 public:
10    Vector(int dimension=0);
11    Vector(const Vector &vec);
12    Vector & operator=(const Vector &vec);
13    virtual ~Vector(); // 虚析构函数，详见第 14.3.3 小节
14    void Set();
15    int GetDim() const;
16    void Show() const;
17    friend Vector Multiply(const Matrix &m, const Vector &v);
18 private:
19    int n;
20    double *x;
21 };
22 #endif

```

```

1 // Vector.cpp
2 #include <iostream>
3 #include "Vector.h"
4 #include "Matrix.h"
5 using namespace std;
6
7 Vector::Vector(int dimension)
8 {
9     if(dimension <= 0)
10    {
11        n = 0; x = NULL;
12        return;
13    }
14    x = new double[n=dimension];
15    for(int i=0; i<n; i++)
16        x[i] = 0;
17 }
18
19 Vector::Vector(const Vector &vec)
20 {
21     x = NULL;
22     if((n=vec.n)!=0) // 赋值表达式作为判断条件
23    {
24        x = new double[n];
25        for(int i=0; i<n; i++)
26            x[i] = vec.x[i];
27    }
28 }
29 Vector & Vector::operator=(const Vector &vec)
30 {
31     if(&vec == this) return *this;

```

```

32     if(x!=NULL) delete [] x;    // 主动释放原有资源
33     x = NULL;
34     if((n=vec.n)!=0)
35     {
36         x = new double[n];
37         for(int i=0; i<n; i++)
38             x[i] = vec.x[i];
39     }
40     return *this;
41 }
42 Vector::~Vector()
43 {
44     if(x!=NULL) delete [] x;
45 }
46 void Vector::Set()
47 {
48     for(int i=0; i<n; i++)
49         x[i] = rand() % 10;      // 为简单起见, 随机取值
50 }
51 int Vector::GetDim() const
52 {
53     return n;
54 }
55 void Vector::Show() const
56 {
57     if(n==0) return;
58     cout << "(" << x[0];
59     for(int i=1; i<n; i++)
60         cout << ", " << x[i];
61     cout << ")" << endl;
62 }
63
64 Vector Multiply(const Matrix &mat, const Vector &vec)
65 {                                         // 普通 C++ 函数
66     if(mat.col != vec.n)      // 无法相乘
67         return Vector(0);    // 直接用构造函数创建一个无名对象返回
68
69     Vector result(mat.row); // 创建向量类局部自动对象, 分量皆为 0
70     for(int i=0; i<mat.row; i++)
71     {
72         for(int j=0; j<vec.n; j++)
73             result.x[i] += mat.x[i*mat.col+j] * vec.x[j];
74     }
75     return result;
76 }
```

```

1 // Matrix.h
2 #ifndef MATRIX_H
3 #define MATRIX_H
4
5 class Vector;                                // 提前声明
6
7 class Matrix
8 {
9 public:
10    Matrix(int Row=0, int Col=0);
11    Matrix(const Matrix &mat);
12    Matrix & operator=(const Matrix &mat);
```

```

13     virtual ~Matrix();
14     void Set();
15     int GetRow() const, GetCol() const;
16     void Show() const;
17     friend Vector Multiply(const Matrix &m, const Vector&v);
18 private:
19     int row, col;
20     double *x;
21 };
22 #endif

```

```

1 // Matrix.cpp
2 #include "Vector.h"
3 #include "Matrix.h"
4 #include <iostream>
5 using namespace std;
6
7 Matrix::Matrix(int Row, int Col)
8 {
9     if(Row<=0 || Col<=0)
10    {
11        row = col = 0; x = NULL;
12        return;
13    }
14    row = Row; col = Col;
15    x = new double[row*col];           // 一级指针
16    for(int i=row*col-1; i>=0; i--)
17        x[i] = 0;                   // 一维数组
18 }
19
20 Matrix::Matrix(const Matrix &mat)
21 {
22     if(mat.row==0 || mat.col==0)
23    {
24        row = col = 0; x = NULL;
25        return;
26    }
27    row = mat.row; col = mat.col;
28    x = new double[row*col];
29    for(int i=row*col-1; i>=0; i--)
30        x[i] = mat.x[i];
31 }
32
33 Matrix & Matrix::operator=(const Matrix &mat)
34 {
35     if(&mat == this) return *this;
36     if(x!=NULL) delete [] x;           // 主动释放原有资源
37     if(mat.row==0 || mat.col==0)
38    {
39        row = col = 0; x = NULL;
40        return *this;
41    }
42    row = mat.row; col = mat.col;
43    x = new double[row*col];
44    for(int i=row*col-1; i>=0; i--)
45        x[i] = mat.x[i];
46    return *this;
47 }

```

```

48
49 Matrix::~Matrix()
50 {
51     if(x!=NULL) delete [] x;
52 }
53
54 void Matrix::Set()
55 {
56     for(int i=row*col-1; i>=0; i--)
57         x[i] = rand() % 10;           // 为简单起见，随机取值
58 }
59
60 int Matrix::GetRow() const
61 {
62     return row;
63 }
64
65 int Matrix::GetCol() const
66 {
67     return col;
68 }
69
70 void Matrix::Show() const
71 {
72     for(int i=0; i<row; i++)
73     {
74         cout << '[' << x[i*col];
75         for(int j=1; j<col; j++)
76             cout << " " << x[i*col+j]; // 一级指针处理二维数组
77         cout << ']' << endl;
78     }
79 }
```

```

1 // main.cpp
2 #include "Vector.h"
3 #include "Matrix.h"
4 #include <iostream>
5 #include <ctime>
6 using namespace std;
7
8 int main()
9 {
10     Vector vec(5);
11     Matrix mat(4, 5);
12     Vector result;           // 暂时不指定其维数
13     time_t t;
14     srand(time(&t));
15     mat.Set();
16     vec.Set();
17     mat.Show();
18     vec.Show();
19
20     result = Multiply(mat, vec);
21     cout << "The result is: ";
22     result.Show();
23     return 0;
24 }
```

程序的运行结果（矩阵和向量的元素均随机取自 0~9 之间的整数）：

```
[1 2 5 6 5]
[5 7 8 1 2]
[2 7 6 5 6]
[4 2 1 3 1]
(8, 6, 2, 3, 9)
The result is : (93, 119, 139, 64)
```

11.2.2 友元类

可以声明一个类是另一个类的友元，称为友元类。这样一来，友元类中的所有成员函数均成为本类的友元函数，均能访问本类的所有成员。须注意的是友元类的友元函数不一定是本类的友元。如：

```
1 // friendTest.cpp
2 #include <iostream>
3 using namespace std;
4
5 class A; // 提前声明
6
7 class B
8 {
9 public:
10    B(int x=100) : b(x) {}
11    friend class A; // 声明类 A 为类 B 的友元类
12 private:
13    int b;
14 };
15 class A
16 {
17 public:
18    A(int x=0) : a(x) {}
19    void g(B &b) // 类 A 的成员函数访问类 B 对象的私有成员
20    {
21        cout << "[" << ++b.b << "]" << endl;
22    }
23    friend void f(A &a, B &b) // 声明并定义友元函数
24    {
25        cout << "(" << (a.a+=10) << ")" << endl; // ok
26 //        cout << b.b << endl; // error. 不能访问 b 的私有成员
27    }
28 private:
29    int a;
30 };
31 int main()
32 {
33    A a; // 创建类 A 的对象
34    B b; // 创建类 B 的对象
35    f(a, b);
36    f(a, b);
37    a.g(b);
38    a.g(b);
39    return 0;
40 }
```

程序的运行结果：

```
(10)
(20)
[101]
[102]
```

从结果可见，类 A 的成员函数 g 可以访问（读或写）类 B 对象的所有成员。但是，类 A 的友元函数 f 不再是类 B 的友元函数，不能访问类 B 对象的受保护的或私有的成员。

11.3 趣味程序——自动单向链表类

本节例子所设计的类能够处理单向链表，但对任何一种实际数据类型实例化的类（例如 `AutoLink<int>` 或 `AutoLink<char>`，亦或 `AutoLink<Student>`）自始至终都有且仅有唯一的一条单向链表。因此，这种设计方案不适合作为单向链表类的普遍设计模式（更为一般的单向链表类模板设计方案见练习 11 的 3(1)）。

本节设计的类有一个特点：无论通过何种途径创建（构造、拷贝构造）的对象——包括全局对象、静态全局对象、静态局部对象、局部自动对象、堆对象、对象数组各元素、堆对象数组各元素、对象值传递的参数（实参对象的副本）、对象值返回时函数创建的临时对象及组合成员对象等——都将自动地插入到这唯一的一条单向链表中（对于某一种实际数据类型）。若某对象的生命期结束，则该对象会自动从链表中退出，并在临退出前负责维护好单向链表的连接。总之，该链表全程记录了该类所有对象的创建、销毁事件。

读者可以利用这个类来验证各种存储类型对象生命期的特性；验证构造函数、析构函数的调用顺序；验证递归函数执行过程中函数参数的生命期及其值的变化情况（见练习 11 的 2(3)）。

源代码 11.3 自动单向链表类

```
1 // AutoLink.h 自动单向链表类模板
2 #ifndef AUTOLINK_H
3 #define AUTOLINK_H
4 #include <iostream>
5 using namespace std;
6
7 template <typename T> class AutoLink
8 {
9 public:
10     AutoLink<T>(const T &x=0);
11     AutoLink<T>(const AutoLink<T> &a);
12     AutoLink<T> & operator=(const AutoLink<T> &a);
13     virtual ~AutoLink<T>();           // 虚函数（参见第13章）
14     operator T() const;             // 重载类型转换运算符
15     static void ShowLink();         // 静态成员函数
16 private:
17     T data;
18     AutoLink<T> *next;
19     static AutoLink<T> *head;      // 静态数据成员，链首指针
20     static int num;                // 静态数据成员，结点个数
21 };
```

```

22 template <typename T> AutoLink<T> *AutoLink<T>::head = NULL;
23                                     // 定义静态数据成员。即创建一条空链表，见第 8.2.2 小节
24 template <typename T> int AutoLink<T>::num = 0;           // 定义静态数据成员
25
26 template <typename T> AutoLink<T>::AutoLink<T>(const T &x)
27 {
28     num++;
29     data = x;
30     next = head;                                // 等价于 this->next=head;
31     head = this;                               // 将本对象挂在链首成为新首结点
32     cout << "构造一个对象" << data << endl;
33 }
34 template <typename T> AutoLink<T>::AutoLink<T>(const AutoLink<T> &a)
35 {
36     num++;
37     data = a.data;                            // 仅用对象 a 的一个数据成员的值
38     next = head;                                // 等价于 this->next=head;
39     head = this;                               // 将本对象挂在链首成为新首结点
40     cout << "拷贝构造一个对象" << data << endl;
41 }
42
43 template <typename T>
44 AutoLink<T> & AutoLink<T>::operator=(const AutoLink<T> &a)
45 {
46     cout << "赋值运算，将" << data
47         << "改成" << a.data << endl;
48     data = a.data;                            // 仅修改数据，不能改变连接次序和 num 值
49     return *this;
50 }
51
52 template <typename T> AutoLink<T>::~AutoLink<T>()
53 {                                         // 当“本对象”生命期即将结束时
54     AutoLink<T> *pGuard = head;
55     if(head == this)                      // 若“本对象”为链首结点
56         head = next;
57     else
58     {
59         while(pGuard->next != this) // 寻找“本结点”的前哨结点
60             pGuard = pGuard->next;
61         pGuard->next = next;        // 负责任地维护好链表
62     }
63     num--;
64     cout << "析构一个对象" << data << endl;
65 }
66
67 template <typename T> AutoLink<T>::operator T() const
68 {           // 数据类型转换运算符重载。将 AutoLink<T> 类型的对象转换成 T 类型的变量
69     return data;
70 }
71 template <typename T> void AutoLink<T>::ShowLink()
72 {
73     AutoLink<T> *p;
74     cout << "结点数: " << num << ", " << head;
75     for(p=head; p!=NULL; p=p->next)
76         cout << " -> " << p->data;
77         cout << " -> NULL" << endl;
78 }
79 #endif

```

```

1 // main.cpp
2 #include "AutoLink.h"
3
4 void f1()
5 {
6     AutoLink<int> x(1), array[3], *p; // 局部自动对象、数组
7     AutoLink<int>::ShowLink();
8     AutoLink<int> y = x; // 复制构造对象
9     AutoLink<int>::ShowLink();
10    p = new AutoLink<int>(5); // 动态对象（堆对象）
11    array[1] = *p; // 赋值运算
12    AutoLink<int>::ShowLink();
13    static AutoLink<int> s(6); // 静态局部对象
14    AutoLink<int>::ShowLink();
15    delete p; // 释放堆对象
16    AutoLink<int>::ShowLink();
17 }
18
19 class Test // 测试静态数据成员和组合成员
20 {
21 public:
22     Test(char x='A') : a(x) {}
23     Test(const Test &t) : a(t.a) {}
24 private:
25     static AutoLink<char> sa; // 只要是 AutoLink 类的对象
26     AutoLink<char> a; // 不管身在何处，皆“一网打尽”
27 };
28
29 AutoLink<char> Test::sa('Z'); // Test 类的静态数据成员创建及初始化
30
31 void f2()
32 {
33     Test t; // 创建对象 t 时创建组合成员 a (AutoLink 类的对象)
34     AutoLink<char>::ShowLink();
35 } // 析构 t 时，析构其组合成员 a
36
37 int main()
38 {
39     cout << "Begin..." << endl;
40     AutoLink<int>::ShowLink();
41     f1();
42     AutoLink<int>::ShowLink();
43
44     AutoLink<char>::ShowLink();
45     f2();
46     AutoLink<char>::ShowLink();
47
48     cout << "Return to Operating System." << endl;
49     return 0;
50 }

```

程序的运行结果：

```

构造一个对象 (Z)
Begin ...
结点数: 0, head-> NULL
构造一个对象 (1)

```

```
构造一个对象 (0)
构造一个对象 (0)
构造一个对象 (0)
结点数: 4, head -> 0 -> 0 -> 0 -> 1-> NULL
拷贝构造一个对象 (1)
结点数: 5, head -> 1 -> 0 -> 0 -> 0 -> 1-> NULL
构造一个对象 (5)
赋值运算, 将 0 修改成 5
结点数: 6, head -> 5 -> 1 -> 0 -> 5 -> 0 -> 1-> NULL
构造一个对象 (6)
结点数: 7, head -> 6 -> 5 -> 1 -> 0 -> 5 -> 0 -> 1-> NULL
析构一个对象 (5)
结点数: 6, head -> 6 -> 1 -> 0 -> 5 -> 0 -> 1-> NULL
析构一个对象 (1)
析构一个对象 (0)
析构一个对象 (5)
析构一个对象 (0)
析构一个对象 (1)
结点数: 1, head -> 6-> NULL
结点数: 1, head -> Z-> NULL
构造一个对象 (A)
结点数: 2, head -> A -> Z-> NULL
析构一个对象 (A)
结点数: 1, head -> Z-> NULL
Return to Operating System.
析构一个对象 (6)
析构一个对象 (Z)
```

运行结果的第一行及最后一行表明：类的静态数据成员在 `main` 函数执行之前就已经被构造了，并且最后被析构。静态数据成员 `num` 虽显得有一点冗余，但有了它，可省去计算结点数的遍历操作，使程序的执行效率更高。

11.4 小 结

C++ 静态成员机制进一步完善了类的封装性。利用这一机制能彻底消灭程序中的全局变量，进一步增强了程序的安全性、可移植性。静态数据成员在 `main` 函数执行之前定义并初始化，其生命期是全局的，直到 `main` 函数结束后才被销毁。从本章所介绍的程序实例可见，有静态数据成员的类应该考虑是否需要定义拷贝构造函数、析构函数。另一方面，拷贝构造函数、析构函数也不一定只是做深拷贝、释放资源之类的事情。它们的需要性及应该完成的操作是由实际问题决定的。

严密的封装可能带来效率上的损失。为了弥补这一损失，除了可以采用内联 (`inline`) 建议外，C++ 还提供了一种有效的机制——友元函数及友元类。友元只是类的朋友而不是类的成员，“朋友的朋友”不一定是朋友。友元在类的封装上开了一扇小门，作为朋友身份者应该谨慎行事，切忌滥用权力。友元将在第 12 章中大显身手，大放异彩。

静态成员函数、友元函数在处理具体对象时均需要将对象（最好是对象的引用）显式地传递给函数。

练习 11

1. 指出下列各程序片段中的错误

(1)

```

1 #include <iostream>
2 using namespace std;
3 class A
4 {
5 public:
6     static A(int x=0){ s++; a=x; }
7     void Show() const;
8     void Set(int x);
9     static void Set(A &a, int x);
10    static int GetS();
11    static int GetA();
12 private:
13     int a;
14     static int s = 0;
15 };
16 void A::Show() const
17 {
18     cout << "a=" << a
19         << ",s=" << s << endl;
20 }
21 void A::Set(int x){ a = x; }
22 static void A::Set(A &a, int x){ a.a = x; }
23 static int A::GetS(){ return s; }
24 static int A::GetA(){ return a; }
25
26 int main()
27 {
28     A obj;
29     A::Set(obj, 100);
30     obj.Show();
31     return 0;
32 }
```

(2)

```

1 #include <iostream>
2 using namespace std;
3 class A
4 {
5 public:
6     A(int x=0):a(x) {}
7     friend void Set(const A &a, int x)
8     {
9         a = x;
10    }
11    friend void Show(const A &a);
12 private:
13     int a;
14 };
15 friend void Show(const A &a)
16 {
17     cout << a.a << endl;
18 }
```

2. 基本题

(1) 设计一个日期 (Date) 类, 其中有数据成员 `int year, month, day;`, 有静态数据成员数组 `int days[12];`, 用于分别存放一年 12 个月的天数, 初始化为 {31, 28, 31, 30, 31, 30, 31, 31, 30, 31}。在具体应用时, 根据对象的成员 `year` 年是否为闰年调整 2 月份的天数 `days[1]` 的值。请设计一个成员函数计算对象的日期在当年是第几天。

(2) 设计 `Point` 类描述平面直角坐标系中的点, 其中有数据成员 `double x, y;`, 再设计一个该类的友元函数用于计算两个给定的 `Point` 类对象之间的距离。

(3) 利用自动单向链表类设计测试程序, 分别观察计算阶乘的递归函数、计算最大公因数的递归函数的调用及返回过程。

3. 研讨设计题

(1) 单向链表类模板设计。

第 8 章介绍了单向链表的概念, 并用结构体描述了结点类型, 用一系列函数实现了对结点及整个链表的操作。用面向对象的设计方法将所有这些内容封装在类中是可行的, 并且也无太大的困难。

在对整个链表及对结点的许多操作中, 并不直接涉及结点中具体的数据类型及其具体操作。例如, 统计链表中结点的个数、释放链表中的所有结点等函数就几乎与结点中的数据域成员无关 (只要求指向下一个结点的指针变量名为 `next`)。我们曾在第 8.3.1 小节将上述函数写成函数模板的形式。

单向链表类模板框架设计

构成单向链表涉及数据、结点、链表三层结构。

① 底层: 数据域类。数据域成员类无须特别设计, 可将其想象成 `int` 即可。

● 抽象化: 认为已经将各种组分封装成一个类。

● 统一化: 认为数据类型已经“向基本数据类型看齐”了 (具有一系列处理实际应用问题的数据成员及对数据成员的基本操作, 如: 构造、拷贝构造、析构、赋值、计算以及 I/O 操作等)。

② 中间层: 结点类。

● 数据成员 (两个): 数据域类的对象 (组合成员) `data`, 指向下一个结点的指针变量 `next`。

● 成员函数: 一些必要的外部接口, 或者将最高层的链表类作为友元类以便链表类的成员函数能直接访问结点的数据成员。

③ 最高层: 链表类。

● 数据成员 (三个): 链首指针 `head` (必须), 记录“当前结点”的指针 `cur_node` (为方便应用), 结点数 `num` (此成员是冗余的, 但增加该成员将获得运行效率方面的好处)。

● 成员函数: 构造链表、复制链表、析构链表、链表赋值运算, 插入、删除结点, 定位“当前结点”、获取“当前结点”状态, 结点数据操作, I/O 操作, 排序、归并等。

在上述三层结构中, 位于中间层的结点类是重要的, 但它的结构及功能比较简单, 它

是底层到最高层的纽带，起“二传手”的作用：数据域的具体操作由数据域自己完成、链表的操作由最高层负责。这种设计方案也体现了自己的事情自己做的设计原则。

单向链表类模板部分代码

其中链表类模板的 31 个成员函数实现为练习的主要内容。请统计函数实现中对形式数据类型的数据所进行的具体操作（如：构造、转换构造、拷贝构造、赋值运算、算术运算、关系运算、I/O 操作、类型转换运算等）。

源代码 11.4 单向链表类模板（头文件）

```

1 // LinkList.h
2 #ifndef LinkList_H
3 #define LinkList_H
4 #include <iostream>
5 #include <fstream>           // 因为文件操作
6 using namespace std;
7
8 template <typename T> class LinkList; // 提前声明
9
10 template <typename T> class Node      // 结点类
11 {
12 public:
13     Node(const T &t=0) : data(t)      // 冒号语法初始化组合成员
14     {
15     }
16     friend class LinkList<T>;        // 声明友元类
17 private:
18     T data;
19     Node *next;
20 };
21
22 template <typename T> class LinkList
23 {
24 private:
25     Node<T> *head, *cur_node;        // 链首指针、当前结点指针
26     int num;                         // 链表中的结点数
27
28 public:
29     LinkList();                      // 默认的构造函数
30     LinkList(const T *t, int n);    // 利用连续元素创建链表
31     LinkList(const LinkList &list); // 拷贝构造链表
32     LinkList & operator=(const LinkList &list); // 赋值运算
33     virtual ~LinkList();           // 析构函数（虚函数）
34
35     // 定位当前结点
36     Node<T> *GoTop();             // 绝对位置：首结点
37     Node<T> *Go(int n);          // 绝对位置：第 n(从 0 起)个结点
38     Node<T> *GoBottom();         // 绝对位置：尾结点
39     Node<T> *Skip(int n=1);       // 相对定位：偏移 n(可为负数)个结点
40     template <typename T1>
41     Node<T> *Locate(const T1 &t, bool restart=false);
42     // 依条件(将结点数据域强制转换成 T1 类型)定位/继续定位。restart!=0 时从链首开始
43
44     // 判断当前结点位置及其他
45     bool isEmpty() const;         // 当前链表是否为空链表
46     bool isBegin() const;         // 当前结点是否为首结点
47     bool isLast() const;          // 当前结点是否为尾结点

```

```

48     Node<T> *CurNode() const;           // 返回当前结点的地址
49     T & CurData() const;                // 引用返回当前结点本身
50     int CurPos() const;               // 返回当前结点序号。0 为头结点, -1 无当前结点
51     int NumNodes() const;             // 返回当前链表结点数
52     // 插入结点
53     void InsBeforeHeadNode(const T &t); // 链首结点前, 成为新链首
54     void InsBeforeCurNode(const T &t); // 当前结点之前
55     void InsAfterCurNode(const T &t); // 当前结点之后
56     void Append(const T &t);          // 尾结点后, 成为新尾结点
57     // 删除结点
58     void DeleteCurNode();            // 当前结点
59     void FreeList();                // 释放所有结点
60     // 修改
61     void Change(const T &t);         // 修改当前结点数据
62     // 输出
63     void ShowCurData() const;        // 输出当前结点数据
64     void ShowList(int LinePerNode=0) const; // 输出所有结点数据
65     // 链表其他操作
66     void Reverse();                 // 结点顺序倒置
67     template <typename T1>
68     void Sort(const T1 &t, bool ascending=true);
69         // 根据指定(转换)类型的关系运算排序。默认升序, 否则降序
70     template <typename T1> static void Merge(LinkList &linkA,
71         LinkList &linkB, const T1 &t, bool ascending=true);
72         // 静态成员函数。将有序的两条链表归并至 linkA; 而 linkB 置空
73     // 文件 I/O 处理
74     void Save(const char *filename,
75             ios_base::openmode OpenMode=ios::out);
76             // 链表各结点数据存入磁盘文件
77     void Load(const char *filename,
78             ios_base::openmode OpenMode=ios::in);
79             // 读数据文件, 追加结点到链表
80 };
81
82 /*****
83 * 类模板成员函数操作描述(应该编写在头文件中)! *
84 *****/
85
86 #endif

```

(2) 单向链表类模板应用。

① **约瑟夫 (Josephus)** 问题。有 n 个小孩 (编号 $1 \sim n$) 按顺时针围成一圈, 从第一个小孩开始按顺时针方向从 $1 \sim m$ 循环报数, 凡报数字 m 的小孩皆退出该圈。直至最后剩下的一个小孩为胜利者。这便是约瑟夫问题。要求从键盘输入 n 和 m , 逐次输出出圈小孩的编号以及尚留在圈中的小孩的编号。

② **链表结点的奇偶二分**。给定一条单向链表, 要求按结点顺序号的奇偶性拆分结点。具体地, 要求创建两条新链表: 偶链表和奇链表 (要求不破坏原链表, 包括保持其原“当前记录”位置)。要求编写二分操作的函数 (写成函数模板) 以适用于尽可能多的数据类型, 在主函数中对整型、字符型数据进行了测试。

③ **删除两条链表的最大相同前缀**。给定两个同类单向链表, 要求删除它们数据域数据值相同的最大前缀结点。

第 12 章 运算符重载

本章介绍的运算符重载将给类的封装做最后的修饰，使我们所设计的类类型更加“向基本数据类型看齐”。运算符重载不仅仅是为了书写程序时的直观和方便，在第 10 章中我们已经感受到重载赋值运算符的必要性。类类型向基本数据类型看齐后，可以直接应用到已经设计好了的类模板中。本章介绍运算符重载的概念、语法和应用。希望读者特别关注本章的程序代码中函数形式参数、函数返回类型的设计，它们体现了向基本数据类型看齐，并保持运算表达式左值或右值特性的设计理念。

12.1 运算符概述

运算符是计算机程序设计语言中表达数据加工操作的符号。运算符由语言本身规定其意义，由编译系统内部定义其基本功能。C++ 语言不仅运算符的符号丰富，而且还用相同的运算符对不同数据类型的数据执行不同的操作。

例如，基本算术运算符 +、-、*、/ 对 C++ 的十几种基本数据类型就有十几种不同的操作，因为不同数据类型的数据在内存中所占用的字节数多少、存放格式是各异的，对它们的操作自然就不会完全相同（最多只能说有些操作的原理相同，但实际操作确实是不同的）。可见，编译系统早就能根据程序的上下文正确地辨析用同一个符号表示的多种不同的操作了。其中的除法运算比较明显：两个整数相除的结果为其整数商（舍弃余数部分）；而两个浮点数相除为其含小数的商。对指针进行运算时，其结果与指针的目标数据类型有关，指针加减整数 n 的结果为偏移 $n * \text{sizeof(数据类型)}$ 字节地址值；两个同类指针相减的结果不是字节地址的差而是数据的项数（字节地址的差 / sizeof(数据类型) ）。

另一个明显的例子是运算符 “`>>`” 和 “`<<`”。在 C/C++ 语言中，它们是整数的位操作符（右移、左移）；在 C++ 中，它们还被用作 I/O 操作——将它们重载为输入输出操作符。对于不同的数据类型更有具体的不同：

① 对于 `signed char*`, `unsigned char*` 类型的操作数进行操作时，系统将运算符 “`<<`” 重载为“输出从操作数所指向地址处开始的一系列连续字符（字符串），直至遇到字符 ‘\0’ 为止”。另将运算符 “`>>`” 重载为“从输入流缓冲区中抽取一系列字符，并将抽取到的字符从操作数所指向的地址处开始依次存放，直至遇到空格字符（‘ ’）、或者 `<Tab>` 字符（‘\t’）、或者换行字符（‘\n’）为止，最后追加存放串结束标志字符（‘\0’）”。参见第 5.7.3 小节所述的“适应性调整”，实为运算符重载。

② 对其他基本数据类型或其指针类型，则按一般的 I/O 流进行处理。其实际进行的操作自然也会因数据类型不同而异。

在 C++ 中，可以称运算符为运算符函数，即运算符是一种特殊的函数，它们的名称、定义和调用格式与普通函数有些差别。运算符函数可以作为某个类的成员函数，也可以作为普通的 C++ 函数（常作为类的友元函数）。

在第 10 章已经介绍过赋值运算符函数。其函数名为 `operator=`，其中 `operator` 是 C++ 的保留字，在保留字与具体的运算符之间可以插入空格。可以像使用一般函数那样向运算符函数传递实参来调用它。可以用更加直观方便的格式调用它：将实参作为运算操作数分别写在运算符的左侧或右侧（如前、后置自增、自减运算），或运算符的左右两侧（如双目运算）；函数的返回值便是运算操作表达式的运算结果。

程序员可以自行重载 C++ 的某些运算符用于自定义的类类型。关于运算符重载，C++ 规定如下。

① C++ 不允许用户自己定义新的运算符，只能对已有的部分运算符进行重载；不允许改变运算符操作数的个数（自然不允许使用带默认值的参数）；不能改变运算符的运算优先级；不能改变运算符的运算结合方向。

② 下列 5 个运算符不允许被重载。

<code>::</code>	作用域区分符
<code>.</code>	成员访问运算符
<code>*</code>	成员指针访问运算符
<code>sizeof</code>	数据尺寸运算符
<code>? :</code>	三目条件运算符（唯一的三目运算）

③ 重载运算符时，至少要有一个操作数为用户自定义的类类型（因为对基本数据类型及其指针而言，其定义已经存在，而构成重载必须有不同于基本数据类型的参数）。

④ 重载的运算符函数不能为类的静态成员函数。

须指出的是系统不会将运算符“+”与运算符“=”自动组合成运算符“+=”。需要使用运算符“+=”时，应该单独重载它。

关于运算符重载，有如下基本原则。

① 尽可能地使用引用型形式参数，并尽可能地加以 `const` 限制（其作用是尽可能地避免拷贝构造形参操作数；尽可能地保护实参操作数；同时使操作符具有与常量运算的功能）。

② 尽可能地采用引用返回（其作用是尽可能地避免拷贝构造临时对象，并使运算表达式成为左值）。

③ 若第一个操作数可能为非本类的对象时，必须将运算符重载成类的友元函数（例如希望日期类的对象 `Date d(2008, 8, 8);` 与整数 `int n=6;` 进行相加运算。运算表达式 `d+n` 与 `n+d` 执行的是两个不同的运算符函数。前者可以作为类的成员函数，隐含对象 `d` 作为第一个操作数。而后者只能作为友元函数定义，将整数 `n` 作为第一个操作数）。

④ 尽可能地保持运算符原有的含义、保持运算符的直观可视性。

⑤ 充分利用类型转换函数（参见第 12.4 节）、转换构造函数（参见第 10.1.2 小节）。

12.2 重载运算符

在设计单向链表类模板时，希望类模板具有更强的通用性——支持更多的数据类型。然而，在具体设计类模板时，假想的数据类型是基本数据类型，甚至是 `int` 类型。倘若希望程序员自己设计的类类型也能利用上类模板，则自然要求该类类型能够进行类模

板中所进行的操作（如：构造、拷贝构造、一定的转换构造、赋值运算，可能需要的输入/输出、关系运算等）。因此，只要自定义的类类型能进行相关操作，则这个类类型便能使用单向链表类模板，亦即，这个自定义的类类型就向基本数据类型“看齐了”。

本节以日期类的设计为例，按照前面提到的设计原则设计一个“干净利落”的、向基本数据类型看齐的类类型。最后还用单向链表类模板进行了测试。

例 13.1 日期 (Date) 的描述及处理方法。

(1) 属性：年 (year)、月 (month)、日 (day)。

(2) 行为：构造函数、算术运算（日期与整数相加减、两个日期相减）、关系运算（约定：以前的日期小，之后的日期大）、I/O 操作（插入运算、抽取运算）、增量运算（前增量、后增量、前减量、后减量），以及与日期有关的其他函数。

开始，仅能给出类 Date 的声明体的基本框架，其中一些成员函数、友元函数的函数原型暂时无法给出，下面将边分析边设计，最后需要读者根据本节中所定义的成员函数和友元函数完成一个完整的日期类声明体。

源代码 12.1 日期 Date 类声明（头文件·部分）

```

1 // Date.h
2 #ifndef DATE_H
3 #define DATE_H
4 #include <iostream>
5 using namespace std;
6
7 class Date
8 {
9 public:
10    Date(int Year=2008, int Month=1, int Day=1);
11    bool isLeapyear() const;           // 判断是否为闰年
12    friend bool isLeapyear(int y);    // 判断是否为闰年
13    Date & SetDate(int Year, int Month, int Day);
14        // 根据年、月、日设置日期
15    Date & SetDate(int Year, int DaysOfYear);
16        // 根据年、当年的天数设置日期。DaysOfYear 可为负数、较大的数
17    int GetYear() const, GetMonth() const, GetDay() const;
18    int GetWeek() const;             // 计算当前日期的星期数
19    int DaysOfYear() const;         // 计算当前日期在当年的天数
20    // 重载的运算符函数原型（声明）待读者根据后续的设计逐步补充
21
22 protected:
23    static bool isValid(int y, int m, int d);      // 静态成员函数
24    static int getTotalDays(int y, int m, int d); // 静态成员函数
25    int getTotalDays() const; // 计算“总天数”，仅内部使用
26
27 private:
28    static int days[12];           // 静态数据成员
29    int year, month, day;
30 };
31 #endif

```

有如下几点说明。

(1) 静态数据成员 days 为一个数组，用于存放 12 个月每月的天数。唯有 days[1] 的值可能经常被改变（会影响所有对象），其他元素保持其初始化值不变。

(2) 由于该类的对象不带资源，其中的静态数据成员的值基本上稳定，直接利用编译系统所提供的默认拷贝构造函数、默认析构函数和默认赋值运算符即可。

(3) 判断闰年的函数 `isLeapyear` 有两个。其一是成员函数，另一为友元函数。

(4) 设置日期的函数 `SetDate` 有重载。

(5) 获取“总天数”^[1]的函数 `GetTotalDays` 有重载。其一是静态成员函数、另一个是非静态成员函数。

(6) 函数 `GetWeek` 为计算当前日期的星期数，返回值为 0、1、……、6 依次表示星期日、星期一、……、星期六。

对于上述已经列入类体中声明的成员函数、友元函数的定义并不是本章的重点，读者不必细究，只需了解其函数原型（以掌握其调用格式）和功能。具体代码如下。

源代码 12.2 日期 Date 类内部实现（源程序文件·部分）

```

1 // Date.cpp
2 #include "Date.h"
3 #include <iomanip>
4
5 int Date::days[12]={31,28,31,30,31,30,31,31,30,31,30,31};           // 定义及初始化静态数据成员
6
7 Date::Date(int Year, int Month, int Day)
8 {
9     SetDate(Year, Month, Day);
10 }
11
12 bool Date::isLeapyear() const
13 {
14     return year%4==0 && year%100!=0 || year%400==0;
15 }
16
17 bool isLeapyear(int year)
18 {
19     return year%4==0 && year%100!=0 || year%400==0;
20 }
21
22 bool Date::isValid(int y, int m, int d)
23 {
24     if(y<=0 || m<=0 || m>12 || d<=0) return false;
25     days[1] = ::isLeapyear(y) ? 29 : 28;
26     if(d > days[m-1]) return false;
27     return true;
28 }
29
30 int Date::GetTotalDays(int year, int month, int day)
31 {
32     int y = year-1, n;
33     n = 365*y + y/4 - y/100 + y/400;
34     days[1] = ::isLeapyear(year) ? 29 : 28;
35     for(int i=1; i<month; i++)
36         n += days[i-1];
37     n += day;
38     return n;

```

[1] 这里计算“总天数”所选择的参照点是“1年1月1日”。即从“1年1月1日”起（含），至“当前日期（含）”所经历的天数。该算法不适用于1582年以前的日期，但其后及今后数千年均正确，这是由于历法的原因。

```
39 }
40
41 int Date::GetTotalDays() const
42 {
43     return GetTotalDays(year, month, day);
44 }
45
46 int Date::GetYear() const {return year;}
47 int Date::GetMonth() const {return month;}
48 int Date::GetDay() const {return day;}
49 int Date::GetWeek() const {return GetTotalDays() % 7;}
50
51 int Date::DaysOfYear() const
52 {
53     int i, n = day;
54     days[1] = ::isLeapyear(year) ? 29 : 28;
55                                         // 常量成员函数可以修改静态数据成员的值
56     for(i=1; i<month; i++)
57         n += days[i-1];
58     return n;
59 }
60
61 Date & Date::SetDate(int Year, int Month, int Day)
62 {
63     if(isValid(Year, Month, Day))
64         year = Year, month = Month, day = Day; // 逗号运算表达式
65     else
66         year = month = day = 0;
67     return *this;
68 }
69
70 Date & Date::SetDate(int Year, int Daysofyear)
71 {
72     int y, n;
73     n = GetTotalDays(Year, 1, 1) + Daysofyear - 1;
74     y = n / 365;
75     y -= (y/4-y/100+y/400)/365;
76     if(n <= (365*y+y/4-y/100+y/400))
77         year = y;
78     else
79         year = y+1;
80     y = year-1;
81     n -= 365*y + y/4 - y/100 + y/400;
82     days[1] = ::isLeapyear(year) ? 29 : 28;
83     for(month=1; month<=12; month++)
84     {
85         if(n > days[month-1])
86             n -= days[month-1];
87         else
88         {
89             day = n;
90             break;
91         }
92     }
93     return *this;
94 }
95
96 // 重载的运算符函数定义待补充
```

下面按双目运算、单目运算依次介绍重载运算符函数的定义^[1]，请读者将其添加到源程序文件 Date.cpp 中，并将运算符函数的原型写入头文件 Date.h 中。

需提醒读者注意的是将成员函数的函数原型写入头文件中时，类名及其作用域区分符可以省略；对于友元函数必须加上保留字 friend。

12.2.1 重载双目运算符

双目运算符需要两个操作数。运算符函数做类的成员函数时，由于隐含传递本对象的地址给 this 指针（运算符函数将实参对象兼容地处理成对象的地址），故双目运算符成员函数只有一个显式形式参数。而作为友元的双目运算符函数，其两个形式参数均应该为显式的。

重载双目运算符时，要特别注意当两个操作数联系同一个实参对象时的处理，要避免因修改左操作数（同时也是右操作数，即使右操作数有 const 限制）而“无意”中改变了右操作数导致计算错误（见练习 12 的 1，以及第 10~12 章有关赋值运算符重载）。

1. 重载算术运算符

算术运算中一般不应该修改操作数，必须将运算结果存放在其他对象或变量中。这些运算符函数的返回类型只能为值返回。与日期相关的算术运算如下。

- ① 日期加整数、日期减整数、整数加日期，结果为一个新的日期对象。
- ② 两个日期对象相减，结果为一整数。

接前面的源程序文件 Date.cpp。

```

97 Date operator+(const Date &d, int n)      // 友元函数
98 {
99     Date temp(d);
100    int m = temp.DaysOfYear() + n;
101    temp.SetDate(d.year, m);
102    return temp;
103 }
104 Date operator-(const Date &d, int n)      // 友元函数
105 {
106     return d + (-n);           // 利用已经重载的运算符函数
107 }
108
109 Date operator+(int n, const Date &d)      // 友元函数
110 {
111     return d + n;
112 }
113 int Date::operator-(const Date &d) const // 常量成员函数
114 {
115     return GetTotalDays() - GetTotalDays(d.year, d.month, d.day);
116 }
```

上述四个函数中，由于第三个函数的第一个操作数为非 Date 对象，因而不能将该函数设计成成员函数，而将其作为友元函数。其余的函数可以设计成成员函数或友元函数。

^[1] 第 12.3 节有“重载方括号运算符”的内容。第 12.4 节有“重载（强制）类型转换运算符”的内容。

2. 重载迭代赋值运算符

迭代赋值运算中一般不应该修改右操作数，计算结果存入左操作数。因此，这些运算符函数可以是成员函数（也可以是友元函数），并且应该设计成引用返回本对象本身。它们是日期加整数并赋值和日期减整数并赋值。

```

118 Date & Date::operator+=(int n)
119 {
120     *this = *this + n;    // 利用 operator+ 和默认的赋值运算符函数
121     return *this;
122 }
123
124 Date & Date::operator-=(int n)
125 {
126     *this = *this + (-n);
127     return *this;
128 }
```

3. 重载关系运算符

重载了两个日期相减的运算符函数后，便可以利用它重载如下六个关系运算符。

```

130 bool Date::operator> (const Date &d) const {return *this-d>0;}
131 bool Date::operator>=(const Date &d) const {return *this-d>=0;}
132 bool Date::operator< (const Date &d) const {return *this-d<0;}
133 bool Date::operator<=(const Date &d) const {return *this-d<=0;}
134 bool Date::operator==(const Date &d) const {return *this-d==0;}
135 bool Date::operator!=(const Date &d) const {return *this-d!=0;}
```

上述重载的关系运算符函数设计为成员函数。读者可以将其改成友元函数设计。

4. 重载 I/O 流操作运算符

插入运算（运算符“`<<`”）是双目运算，其第一个操作数常常为 `cout` 并且可以连续插入（相当于“复合函数”调用）。抽取运算（运算符“`>>`”）也是双目运算，其第一个操作数常常为 `cin` 并且可以连续抽取（相当于“复合函数”调用）。如：

```

int n;
double x;
cout << "Input n x: ";
cin >> n >> x;
cout << n << ", " << x << endl;
```

可见，上面的语句在调用插入运算符函数时使用了实参 `cout`，它是类 `ostream` 的对象；插入运算的第二个操作数可以为常量。在调用抽取运算符函数时使用了实参 `cin`，它是类 `istream` 的对象。抽取运算的第二个操作数必须是左值，其形式参数应该为引用（双向传递）。使用类 `ostream` 和类 `istream` 需要包含头文件 `<iostream>`。

重载插入、抽取运算符使其能处理 `Date` 类的对象，相应的函数定义如下。

```

137 ostream & operator<<(ostream &out, const Date &d)
138 {
139     out << setfill('0') << setw(4) << d.year
140         << '/' << setw(2) << d.month
```

```

141         << '/' << setw(2) << d.day
142         << setfill('0');
143     return out;
144 }
145
146 istream & operator>>(istream &in, Date &d)
147 {
148     int year, month, day;
149     char str[100];
150
151     in.getline(str, 100, '/'); // 抽取操作, 遇到字符 '/' 为止
152     if(in==NULL) return in;
153     year = atoi(str);
154
155     in.getline(str, 100, '/'); // 抽取操作, 遇到第二个 '/' 字符为止
156     if(in==NULL) return in;
157     month = atoi(str);
158
159     in.getline(str, 100, '\n'); // 抽取操作, 遇到字符 '\n' 为止
160     if(in==NULL) return in;
161     day = atoi(str);
162
163     d.SetDate(year, month, day);
164     return in;
165 }

```

【注意】 插入运算符函数传递 `ostream` 类对象的引用并且引用返回, 这是实现连续插入的关键。抽取运算符函数传递 `istream` 类对象的引用并且引用返回, 也是实现连续抽取的关键。上述运算符不仅仅可以分别对 `cout` 与 `cin` 进行操作, 也可以对输出输入数据文件进行操作。抽取运算符从数据文件中抽取数据时强烈地依赖于数据文件的格式, 因此需要精心设计插入运算操作的格式, 确保插入运算和抽取运算相匹配。

抽取运算符函数中语句 `if(in==NULL) return in;` 的作用是处理输入非正常结束的情形。执行抽取操作时, 若遇到文件结尾或者在 Windows 操作系统下按 `Ctrl+Z` 键, 或者在 UNIX/Linux 操作系统下按 `Ctrl+D` 键, 则返回 `istream` 类的空对象 (`istream`)`NULL`, 此时上述抽取运算不改变对象 `d` 的原有值。

12.2.2 重载单目运算符

单目运算符仅需要一个操作数。运算符函数作为类的成员函数时, 由于隐含传递了一个参数, 故单目运算符成员函数不再需要显式形式参数。而作为友元的单目运算符函数其形式参数应该为显式的。

1. 重载前增（减）量运算符

前增量、前减量运算符都是单目运算符, 其特点是: 先修改操作数的值, 然后以新值参与所在表达式的其他运算。并且前增量、前减量运算的结果都可以做左值。这就要求引用传递对象及引用返回对象本身, 所返回的对象具有新值。

```

166 Date & Date::operator++()
167 {
168     days[1] = isLeapyear() ? 29 : 28;
169     day++;

```

```

170     if(day > days[month-1]) { day = 1; month++; }
171     if(month>12)           { month = 1; year++; }
172     return *this;
173 }
174
175 Date & Date::operator--()
176 {
177     days[1] = isLeapyear() ? 29 : 28;
178     if(day==1 && month==1)
179     {
180         year--; month = 12; day = 31;
181         return *this;
182     }
183     day--;
184     if(day < 1){ month--; day = days[month-1]; }
185     return *this;
186 }
```

2. 重载后增（减）量运算符

后增量、后减量运算符也都是单目运算符。为了与前增量、前减量运算符区别，C++语言规定在其操作符函数原型及函数首部中添加一个纯形式上的参数 int。这个参数仅仅起记号作用：告知编译器作为后增量、后减量运算符处理。

后增量、后减量运算的特点是先以操作数的原值参与所在表达式的其他运算一次，然后修改操作数的值。例如：

```

int a, b=2;
cout << (a = b++) << ", " << b << endl; // 输出 2, 3

```

上面的第二行代码是一条表达式语句，参与运算的操作数有 cout, a, b, ", " , endl，运算符有 <<, () , =, ++。根据运算优先级确定最先计算圆括号中的赋值运算：将 b 的原值（整数 2）赋给变量 a（这是变量 b 以原值参与的所在表达式的一次其他运算），然后变量 b 增 1（此时，a 的值为 2，变量 b 的值为 3，表达式 (a = b++) 的值为 2，输出该值）。

这就要求函数返回的是操作数的“原值”（因为函数调用表达式正是函数返回值的表现形式），“之后”操作数的值却要发生变化（实际上，这个变化只能在函数返回之前，即在运算符函数体内发生）。因此，若为友元函数，则函数的形式参数应该为引用型（需要修改操作数的值），函数无法用“本对象”返回修改前的原值。具体定义如下。

```

189 Date Date::operator++(int)      // 对象值返回，不宜做左值
190 {
191     Date temp(*this);          // 拷贝构造对象以保存本对象“原值”
192     ++(*this);                // 利用前增量运算符改变对象的值
193     return temp;              // 返回所保存本对象的原值
194 }
195 Date Date::operator--(int)
196 {
197     Date temp(*this);
198     --(*this);
199     return temp;
200 }
```

由于 `temp` 是一个局部自动对象，其生命期随函数返回而结束，故上述两个运算符函数不能采用引用返回（参见第 6.2 节最后的有错误的代码）。采用对象值返回需要拷贝构造一个临时对象存放“原值”以参与可能有的其他运算。C++ 中，基本数据类型的后增（减）量运算表达式不能做左值。用户自定义的类类型后增（减）量运算表达式虽可做左值，但被操作者却不是对象本身而是临时对象（参见练习 12 的 2(3)）。

在完成 `Date` 类的设计后，来做个测试——打印日历。

源代码 12.3 打印万年历

```

1 // Calendar.cpp    万年历
2 #include "Date.h"
3 #include <iomanip>
4 int main()
5 {
6     Date d;
7     int year, day, week;
8
9     cout << "Year?" << flush;
10    cin >> year;
11
12    d.SetDate(year, 1, 1);      // 从当年的元旦起
13    week = d.GetWeek();        // 计算元旦的星期数
14
15    while(d.GetYear()==year)
16    {
17        day = d.GetDay();
18        if(day==1 || week==0)
19        {
20            if(day==1)
21            {
22                cout << "\n=====" << setw(2)
23                    << d.GetMonth() << "月=====\n";
24                cout << "SunMonTueWedThuFriSat";
25            }
26            cout << "\n" << setw(4*week) << "";
27        }
28        cout << setw(4) << d.GetDay();
29        week = (week+1)%7;
30        d++;                      // 日期按农历增 1 (下一天的日期)
31    }
32    return 0;
33 }
```

12.3 自定义版字符串类——String

本节的例子中有关于重载方括号运算符的语法及使用方法。重载方括号运算符后，对于对象 `s` 而言，需注意对象的地址 `&s` 与 `&s[0]` 是不同的，`&s[0]` 也不是 `s`。

C++ 提供的字符串类 `string` 具有很强的功能。在本节中，直接将 C-字符串进行封装，设计一个自定义版的字符串类——`String`。该类的设计方法与功能皆与 `string` 类有较大差别。此处并不是想用 `String` 取代 `string`，而是通过这个较完整、涉及较多知识点的例子使读者加深对相关知识点，特别是 C-字符串及 `string` 的理解。

在设计类 `String` 时尽量模仿类 `string` 的功能。例如，为类设计了多个常用的构造函数，重载了一些运算符用于字符串操作，还另增加了几个实用的成员函数或友元函数，如函数原型在头文件中第 48~50 行的函数。`String` 类的另一种设计方案参见练习 13 (3)。

源代码 12.4 自定义版字符串类 class String

```

1 // String.h
2 #ifndef MY_STRING_H
3 #define MY_STRING_H
4 #include <iostream>
5 using namespace std;
6
7 class String
8 {
9 public:
10     String(const char *s="");           // 构造函数
11     String(const char *s, int n);
12     String(int n, char c);
13     String(const String &Str);        // 复制构造函数
14     String(const String &Str, int pos, int n);
15     String & operator=(const String &Str); // 必须为成员函数
16     virtual ~String();                // 析构函数（虚函数）
17
18     String & insert(int p0, const char *s);
19             // 将 s 所指向的字符串插入在本串位置 p0 之前
20     String substr(int pos, int n) const;
21             // 取子串，取本串位置 pos 开始的 n 个字符，构成新对象
22     int find(const String &Str) const;
23             // 查找并返回 Str 在本串中第一次出现的位置
24     int length() const;               // 返回串的长度（字符个数）
25     const char * c_str();            // 转换为 C-字符串
26     void swap(String &Str);          // 将本串与 Str 交换
27     char & operator[](int index);    // 方括号（下标）运算符
28             // 引用返回可作左值、右值
29     friend String operator+(const String &str1,
30                           const String &Str2); // 友元函数，字符串拼接
31     String & operator+=(const String &Str); // 字符串拼接及赋值
32 // 重载关系运算
33     friend bool operator==(const String &Str1, const String &Str2);
34     friend bool operator!=(const String &Str1, const String &Str2);
35     friend bool operator> (const String &Str1, const String &Str2);
36     friend bool operator>=(const String &Str1, const String &Str2);
37     friend bool operator< (const String &Str1, const String &Str2);
38     friend bool operator<=(const String &Str1, const String &Str2);
39 // 重载 I/O 流运算符
40     friend ostream& operator<<(ostream &out, const String &Str);
41     friend istream & operator>>(istream &in, String &Str);
42 // 附加的成员函数
43     friend istream & getline(istream &in, String &Str,
44                           int num, char delim='\'\n' );
45     String & trim();                // 删除字符串前后的空白（空格、制表）字符
46
47 private:
48     char *str;
49 };
50 #endif

```

```
1 // String.cpp
2 #include "String.h"
3 #include <cstring>
4
5 String::String(const char *s)
6 {
7     str = new char[strlen(s)+1];
8     strcpy(str, s);
9 }
10
11 String::String(const char *s, int n)
12 {
13     int i, m = strlen(s);
14     if(m < n) n = m;
15     if(n<0) n = 0;
16     str = new char[n+1];
17     for(i=0; i<n; i++)
18         str[i] = s[i];
19     str[i] = '\0';
20 }
21
22 String::String(int n, char c)
23 {
24     int i;
25     if(n<0) n = 0;
26     str = new char[n+1];
27     for(i=0; i<n; i++)
28         str[i] = c;
29     str[i] = '\0';
30 }
31
32 String::String(const String &Str)
33 {
34     str = new char[strlen(Str.str)+1];
35     strcpy(str, Str.str);
36 }
37
38 String::String(const String &Str, int pos, int n)
39 {
40     int i, m;
41     m = strlen(Str.str);
42     if(pos > m)
43     {
44         str = new char[1];
45         str[0] = '\0';
46         return;
47     }
48     if(m-pos < n) n = m-pos;
49     if(n<0) n = 0;
50     str = new char[n+1];
51     for(i=0; i<n; i++)
52         str[i] = Str.str[pos+i];
53     str[i] = '\0';
54 }
55
56 String & String::operator=(const String &Str)
57 {
58     if(&Str == this) return *this;
```

```
59     delete [] str;
60     str = new char[strlen(Str.str)+1];
61     strcpy(str, Str.str);
62     return *this;
63 }
64
65 String::~String()
66 {
67     if(str!=NULL)
68         delete [] str;
69 }
70
71 String & String::insert(int p0, const char *s)
72 {
73     int n = strlen(str);
74     if(p0 > n) p0 = n;
75     char *p = new char[strlen(str)+strlen(s)+1];
76     strncpy(p, str, p0); // 原字符串内容的第一部分
77     p[p0] = '\0';
78     strcat(p, s); // 插入的部分
79     strcat(p, str+p0); // 原字符串的剩余部分
80     delete [] str; // 释放原字符串
81     str = p; // 保存新字符串的首地址
82     return *this;
83 }
84
85 String String::substr(int pos, int n) const
86 {
87     String temp(*this, pos, n);
88     return temp;
89 }
90
91 int String::find(const String &Str) const
92 {
93     int i, j, m, n, flag;
94     m = strlen(Str.str);
95     n = strlen(str);
96     if(m > n) return -1;
97     for(i=0; i<n-m; i++)
98     {
99         flag = 1;
100        for(j=0; j<m; j++)
101            if(str[i+j]!=Str.str[j])
102            {
103                flag = 0; break;
104            }
105        if(flag==1)
106            return i;
107    }
108    return -1;
109 }
110
111 int String::length () const
112 {
113     return strlen(str);
114 }
115
116 const char * String::c_str()
117 {
```

```
118     return str;
119 }
120
121 void String::swap(String &Str)
122 {
123     char *temp = Str.str;
124     Str.str = str;
125     str = temp;
126 }
127
128 char & String::operator[](int index)      // 重载方括号运算符
129 {
130     return str[index];
131 }
132
133 String operator+(const String &Str1, const String &Str2)
134 {
135     String temp;
136     temp.str = new char[strlen(Str1.str)+strlen(Str2.str)+1];
137     strcpy(temp.str, Str1.str);           // 自动扩展资源空间
138     strcat(temp.str, Str2.str);
139     return temp;
140 }
141
142 String & String::operator+=(const String &Str)
143 {
144     *this = *this + Str;
145     return *this;
146 }
147 bool operator==(const String &Str1, const String &Str2)
148 {
149     return strcmp(Str1.str, Str2.str)==0;
150 }
151 bool operator!=(const String &Str1, const String &Str2)
152 {
153     return strcmp(Str1.str, Str2.str)!=0;
154 }
155 bool operator> (const String &Str1, const String &Str2)
156 {
157     return strcmp(Str1.str, Str2.str) > 0;
158 }
159 bool operator>=(const String &Str1, const String &Str2)
160 {
161     return strcmp(Str1.str, Str2.str) >= 0;
162 }
163 bool operator< (const String &Str1, const String &Str2)
164 {
165     return strcmp(Str1.str, Str2.str) < 0;
166 }
167 bool operator<=(const String &Str1, const String &Str2)
168 {
169     return strcmp(Str1.str, Str2.str) <= 0;
170 }
171
172 ostream & operator<<(ostream &out, const String &Str)
173 {
174     out << Str.str;
175     return out;
176 }
```

```

177 istream & operator>>(istream &in, String &Str)
178 {
179     char str[1000];
180     in >> str;
181     Str = str; // 利用转换构造函数（自动转换），再利用赋值运算
182     return in;
183 }
184
185 // 附加的成员函数
186 istream & getline(istream &in, String &Str, int num, char delim)
187 {
188     if(num<=0) return in;
189
190     if(Str.str!=NULL) delete [] Str.str;
191     Str.str = new char[num+1];
192     in.getline(Str.str, num, delim);
193     return in;
194 }
195
196 String & String::trim()
197 {
198     int i, j = strlen(str);
199
200     if(j==0) return *this;
201
202     for(j--; j>=0 && (str[j]==' ' || str[j]=='\t'); j--)
203         ;
204     str[j+1] = '\0';
205     for(i=0; str[i]==' ' || str[i]=='\t'; i++)
206         ;
207     if(i > j)
208     {
209         delete [] str;
210         str = new char[1];
211         str[0] = '\0';
212     }
213     else
214     {
215         char *temp = new char[j-i+2];
216         strcpy(temp, str+i);
217         delete [] str;
218         str = temp;
219     }
220     return *this;
221 }

```

请读者设计一个程序测试类 String。

12.4 趣味程序——“评委评分”程序之类模板应用

本节的例子中有关于重载（强制）类型转换运算符函数的语法及使用方法。类型转换运算符函数重载有如下特殊性。

(1) 类型转换运算符函数必须为非静态成员函数。

(2) 类型转换运算符函数无返回类型，因为函数名“operator 数据类型”中已经指定了转换结果的数据类型。

(3) 若只重载了一个类型转换运算符函数, 系统将在适当的时候进行自动转换(称为默认的类型转换)。若定义了多个类型转换运算符函数, 则需要使用强制类型转换运算符进行强制转换。

(4) 若声明一个类时, 未定义类型转换运算符函数, 系统不会提供任何类型转换运算符函数。

常常将类类型对象转换成基本数据类型的数据, 然后充分利用基本数据类型的关系运算实现类类型的关系运算。类型转换运算符函数与转换构造函数的配合使用, 可以实现更多的自动转换及运算(参见练习 12 的 2(2))。

下面我们讨论所谓“评委评分”问题。设计 Contest 类如下。

- (1) 属性: 选手编号、姓名、评委人数、平均分、名次和各个分数(堆资源)。
- (2) 行为: 构造函数、深拷贝构造函数、析构函数、深赋值运算符, 设置各个分数、计算平均分, 重载插入、抽取运算符, 重载类型转换运算符, 引用返回数据成员名次。

【说明】 引用返回数据成员名次是因为本对象自身无法计算名次, 因此, 将其引用返回以便可以直接对该数据成员进行操作。

源代码 12.5 “评委评分”之竞赛类

```

1 // Contest.h
2 #ifndef CONTEST_H
3 #define CONTEST_H
4 #include <iostream>
5 #include <string>
6 using namespace std;
7
8 class Contest
9 {
10 public:
11     Contest(int Referees);           // 转换构造函数
12     Contest(string Id="000", string Name="NoName",
13             int Referees=10);
14     Contest(const Contest &c);      // 深复制构造
15     Contest & operator=(const Contest &c); // 深赋值运算符
16     virtual ~Contest();            // 析构函数
17     void Average();                // 计算“平均分”
18     void SetScores();
19     int & Rank();                  // 引用返回可为左值
20     operator double() const; // 类型转换运算符, 将本类数据转换成 double
21     operator int() const; // 类型转换运算符, 将本类数据转换成 int
22     operator string() const; // 类型转换运算符, 将本类数据转换成 string
23     friend ostream & operator<<(ostream &out, const Contest &c);
24     friend istream & operator>>(istream &in, Contest &c);
25
26 private:
27     string id, name;              // 选手编号、姓名
28     int referees;                 // 评委人数
29     double *score, average;       // 各个分数、平均分
30     int rank;                     // 名次
31 };
32 #endif

```

```
1 // Contest.cpp
2 #include "Contest.h"
3 #include <iomanip>
4
5 Contest::Contest(int Referees)
6     : id("NoId"), name("NoName"), referees(Referees)
7 {
8     if(referees<=2) referees = 10;
9     score = new double[referees];
10    for(int i=0; i<referees; i++)
11        score[i] = 0;
12    average = 0;
13    rank = 1;
14 }
15
16 Contest::Contest(string Id, string Name, int Referees)
17     : id(Id), name(Name), referees(Referees)
18 {
19     if(referees<=2) referees = 10;
20     score = new double[referees];
21     for(int i=0; i<referees; i++)
22         score[i] = 0;
23     average = 0;
24     rank = 1;
25 }
26
27 Contest::Contest(const Contest &c) : id(c.id), name(c.name)
28 {
29     score = new double[referees=c.referees];
30     for(int i=0; i<referees; i++)
31         score[i] = c.score[i];
32     average = c.average;
33     rank = c.rank;
34 }
35
36 Contest & Contest::operator=(const Contest &c)
37 {
38     if(&c == this) return *this;
39     if(score!=NULL) delete [] score;
40     id = c.id; name = c.name;
41     score = new double[referees=c.referees];
42     for(int i=0; i<referees; i++)
43         score[i] = c.score[i];
44     average = c.average;
45     rank = c.rank;
46     return *this;
47 }
48
49 Contest::~Contest()
50 {
51     if(score!=NULL) delete [] score;
52 }
53
54 void Contest::Average()
55 {
56     double max, min, sum;
57     max = min = sum = score[0];
58     for(int i=1; i<referees; i++)
```

```
59     {
60         if(score[i]>max) max = score[i];
61         if(score[i]<min) min = score[i];
62         sum += score[i];
63     }
64     average = (sum-max-min)/(referees-2);
65 }
66
67 void Contest::SetScores()
68 {
69     for(int i=0; i<referees; i++) // 为简单起见, 分数随机取值
70         score[i] = (80 + rand() % 21)/10.0;
71     Average();
72 }
73
74 int & Contest::Rank()
75 {
76     return rank;
77 }
78
79 // 重载运算符
80 Contest::operator double() const // 重载类型转换运算符
81 {
82     return average; // 将对象转换成“平均分”
83 }
84 Contest::operator int() const
85 {
86     return rank; // 将对象转换成“名次”
87 }
88
89 Contest::operator string() const
90 {
91     return id; // 将对象转换成“选手编号”
92 }
93
94 ostream & operator<<(ostream &out, const Contest &c)
95 {
96     out << setprecision(3) << showpoint;
97     out << setw(3) << c.id << '\t' << setw(6) << c.name
98         << '\t' << setw(6) << c.average << setprecision(2)
99         << setw(4) << c.rank << "[" << setw(2) << c.referees
100        << "]";
101    for(int i=0; i<c.referees; i++)
102        out << setw(4) << c.score[i];
103    return out;
104 }
105
106 istream & operator>>(istream &in, Contest &c)
107 {
108     Contest temp;
109     char str[80];
110
111     if(in.getline(str, 80, '\t') == NULL) return in;
112     temp.id = str;
113     if(in.getline(str, 80, '\t') == NULL) return in;
114     temp.name = str;
115     if(in>>temp.average == NULL) return in;
116
117     if(in.getline(str, 80, '[') == NULL) return in;
```

```

118     temp.rank = atoi(str);
119     if(in.getline(str, 80, '\n') == NULL) return in;
120     temp.referees = atoi(str);
121
122     if(temp.score != NULL) delete [] temp.score;
123     temp.score = new double[temp.referees];
124     for(int i=0; i<temp.referees-1; i++)
125     {
126         if(in>>temp.score[i] == NULL) return in;
127     }
128     if(in.getline(str, 80, '\n') == NULL) return in;
129     temp.score[temp.referees-1] = atof(str);
130     temp.Average();
131     c = temp;
132     return in;
133 }
```

【说明】 本例中，重载的抽取运算符函数强烈地依赖于重载的插入运算符所输出的格式。特别当字符串中含空格字符时，需要特殊处理。本程序中采用字符 '\t' 做字符串与其他数据的分隔符，以便正确地输入 id 和 name。

(1) 测试之一（数组应用）

下面的代码中利用数组存放六个对象进行处理。

源代码 12.6 “评委评分” 测试程序（数组应用）

```

1 // ArrayTest.cpp
2 #include "Contest.h"
3
4 int main()
5 {
6     int i, j, n;
7     Contest a[] = { Contest("001", "Zhang"),
8                     Contest("002", "Wang"),
9                     Contest("003", "Li", 8),
10                    Contest("004", "Liu", 5),
11                    Contest("005", "Chen", 6),
12                    Contest("006", "Zhao") };
13     n = sizeof(a)/sizeof(*a);
14     time_t t;
15     srand(time(&t));
16     for(i=0; i<n; i++)           // 设置各个分数（随机取值）
17         a[i].SetScores();        // 计算名次
18     for(i=0; i<n; i++)
19     {
20         a[i].Rank() = 1;
21         for(j=0; j<n; j++)
22             if((double)a[j] > (double)a[i]) a[i].Rank()++;
23             // 利用类型（强制）转换后的 double 型数据的关系运算结果
24     }
25     for(i=0; i<n; i++)
26         cout << a[i] << endl;      // 输出结果
27     return 0;
28 }
```

程序的运行结果：

001	Zhang	8.81	5[10]	8.8	8.4	8.4	9.4	8.4	8.4	9.9	9.4	8.1	9.3
002	Wang	8.67	6[10]	8.5	9.3	8.8	8.0	10.	9.0	8.0	8.7	8.5	8.6
003	Li	8.85	3[8]	9.2	8.6	9.2	9.7	9.3	8.5	8.3	8.3		
004	Liu	9.17	1[5]	8.1	9.5	8.6	10.	9.4					
005	Chen	9.15	2[6]	9.0	8.1	9.4	8.5	9.9	9.7				
006	Zhao	8.84	4[10]	8.0	8.7	9.4	8.7	9.0	8.4	8.9	8.7	9.7	8.9

从运行结果可见，对每一位参赛选手而言，为其评分的评委人数（见方括号中的数字）还可以不同。

(2) 测试之二（单向链表类模板应用）

在本教材配套的电子文档中，提供有单向链表类模板头文件 `LinkList.h`。直接利用它进行测试。

源代码 12.7 “评委评分”测试程序（单向链表类模板应用）

```

1 // LinkTest.cpp
2 #include "Contest.h"
3 #include "LinkList.h"
4
5 int main()
6 {
7     int i, j, n, rank;
8     double x;
9     Contest a[] = { Contest("001", "Zhang"),
10                     Contest("002", "Wang"),
11                     Contest("003", "Li", 8),
12                     Contest("004", "Liu", 5),
13                     Contest("005", "Chen", 6),
14                     Contest("006", "Zhao")      };
15     n = sizeof(a)/sizeof(*a);
16     time_t t;
17     srand(time(&t));
18     for(i=0; i<n; i++)
19         a[i].SetScores();
20
21     LinkList<Contest> link(a, sizeof(a)/sizeof(*a)); // 创建链表
22     n = link.NumNodes();                                // 计算名次
23     for(i=0; i<n; i++)                                // 计算名次
24     {
25         link.Go(i);
26         rank = 1;
27         x = (double)link.CurData();                  // 利用重载的转换运算符
28         link.GoTop();
29         for(j=0; j<n; j++)
30         {
31             if((double)link.CurData()>x) // 利用重载的转换运算符
32                 rank++;
33             link.Skip();
34         }
35         link.Go(i);
36         link.CurData().Rank() = rank;
37     }
38     link.ShowList(1);                                // 输出结果，每行一条记录
39     cout << "Save\to\"Contest.txt\"" << endl;
40     link.Save("Contest.txt");                         // 将数据写入文件（存盘）

```

```

41     cout << "FreeList..." << endl;
42     link.FreeList();
43     cout << "Load from \"Contest.txt\" " << endl;
44     link.Load("Contest.txt"); // 从文件中读取数据，插入结点到链表
45     link.ShowList(1);
46
47     cout << "====Sort by rank====" << endl;
48     link.Sort(int(1)); // 升序
49     link.ShowList(1);
50     cout << "====Sort by average====" << endl;
51     link.Sort(double(1), 1); // 升序
52     link.ShowList(1);
53     cout << "====Sort by id====" << endl;
54     link.Sort(string(""), 0); // 降序
55     link.ShowList(1);
56     return 0;
57 }

```

程序的运行结果：

```

001    Zhang    9.03   3[10] 9.8 9.1 9.3 8.9 8.8 8.9 8.1 9.3 8.0 10.
002    Wang     9.06   2[10] 9.6 8.8 9.1 9.2 8.0 8.5 9.4 8.7 10. 9.2
003    Li       8.97   4[ 8] 9.9 8.0 9.0 8.6 9.2 9.7 9.0 8.3
004    Liu      8.60   5[ 5] 9.6 8.1 8.7 8.5 8.6
005    Chen     8.43   6[ 6] 8.9 8.2 8.1 8.5 8.2 8.8
★006   Zhao     9.34   1[10] 9.6 9.7 8.2 9.6 9.5 9.7 8.8 9.1 9.6 8.8
Save to "Contest.txt"
FreeList ...
Load from "Contest.txt"
001    Zhang    9.03   3[10] 9.8 9.1 9.3 8.9 8.8 8.9 8.1 9.3 8.0 10.
002    Wang     9.06   2[10] 9.6 8.8 9.1 9.2 8.0 8.5 9.4 8.7 10. 9.2
003    Li       8.97   4[ 8] 9.9 8.0 9.0 8.6 9.2 9.7 9.0 8.3
004    Liu      8.60   5[ 5] 9.6 8.1 8.7 8.5 8.6
005    Chen     8.43   6[ 6] 8.9 8.2 8.1 8.5 8.2 8.8
★006   Zhao     9.34   1[10] 9.6 9.7 8.2 9.6 9.5 9.7 8.8 9.1 9.6 8.8
==== Sort by rank ====
★006   Zhao     9.34   1[10] 9.6 9.7 8.2 9.6 9.5 9.7 8.8 9.1 9.6 8.8
002    Wang     9.06   2[10] 9.6 8.8 9.1 9.2 8.0 8.5 9.4 8.7 10. 9.2
001    Zhang    9.03   3[10] 9.8 9.1 9.3 8.9 8.8 8.9 8.1 9.3 8.0 10.
003    Li       8.97   4[ 8] 9.9 8.0 9.0 8.6 9.2 9.7 9.0 8.3
004    Liu      8.60   5[ 5] 9.6 8.1 8.7 8.5 8.6
005    Chen     8.43   6[ 6] 8.9 8.2 8.1 8.5 8.2 8.8
==== Sort by average ====
005    Chen     8.43   6[ 6] 8.9 8.2 8.1 8.5 8.2 8.8
004    Liu      8.60   5[ 5] 9.6 8.1 8.7 8.5 8.6
003    Li       8.97   4[ 8] 9.9 8.0 9.0 8.6 9.2 9.7 9.0 8.3
001    Zhang    9.03   3[10] 9.8 9.1 9.3 8.9 8.8 8.9 8.1 9.3 8.0 10.
002    Wang     9.06   2[10] 9.6 8.8 9.1 9.2 8.0 8.5 9.4 8.7 10. 9.2
★006   Zhao     9.34   1[10] 9.6 9.7 8.2 9.6 9.5 9.7 8.8 9.1 9.6 8.8
==== Sort by id ====
★006   Zhao     9.34   1[10] 9.6 9.7 8.2 9.6 9.5 9.7 8.8 9.1 9.6 8.8
005    Chen     8.43   6[ 6] 8.9 8.2 8.1 8.5 8.2 8.8
004    Liu      8.60   5[ 5] 9.6 8.1 8.7 8.5 8.6
003    Li       8.97   4[ 8] 9.9 8.0 9.0 8.6 9.2 9.7 9.0 8.3
002    Wang     9.06   2[10] 9.6 8.8 9.1 9.2 8.0 8.5 9.4 8.7 10. 9.2
001    Zhang    9.03   3[10] 9.8 9.1 9.3 8.9 8.8 8.9 8.1 9.3 8.0 10.
析构一条链表。

```

12.5 小结

运算符重载给类做的华丽修饰不只是外观上的表现美、书写上的简洁美，更重要的是与基本数据类型相一致的统一美。

通过重载运算符函数的定义还能使读者进一步加深对运算符、运算表达式和函数的本质的理解。

将运算符设计成成员函数还是友元函数，函数的参数是否需要保护，函数是值返回还是引用返回，这些问题都在运算符函数重载中细致地呈现出来，好的设计体现了程序员的微观精细品质，锻炼着程序员的宏观分析和综合能力。

练习 12

1. 指出下面函数中的错误

定义一个复数类 Complex（参见第 9.4 节），若为该类增加运算符函数 operator*= 重载，请指出下面重载运算符（成员函数）中的算法错误，并改正。

```

1 Complex & Complex::operator*=(const Complex &c)
2 {
3     Complex temp = *this;
4     re = temp.re * c.re - temp.im * c.im;
5     im = temp.re * c.im + temp.im * c.re;
6     return *this;
7 }
```

2. 基本题

(1) 设计时间 (class Time) 类，要求重载算术运算符（时间与以秒为单位的整数相加减，两个时间对象相减）、插入运算符、抽取运算符（按“时:分:秒”格式）和前(后)增(减)量运算符。

(2) 有如下设计的人民币类 class RMB；其中转换构造函数以“元”（double 型）为单位。请完成类的类型转换运算符函数的重载，以使下面的主函数正确运行（不必重载其他运算符）。

```

1 // RMB.cpp
2 #include <iostream>
3 using namespace std;
4
5 class RMB
6 {
7 public:
8     RMB(double x=0)
9     {
10         int n = int(100*(x+0.005));
11         yuan = n / 100;
12         jiao = n / 10 % 10;
13         fen = n % 10;
14     }
15     operator double() const;      // 待定义
16 }
```

```

17 private:
18     int yuan, jiao, fen;
19 };
20
21 int main()
22 {
23     RMB x = 1.23;
24
25     cout << x << "\n"
26     << x+2 << "\t" << 2+x << "\n"
27     << x-2 << "\t" << 2-x << "\n"
28     << x*2 << "\t" << 2*x << "\n"
29     << x/2 << "\t" << 2/x << "\n"
30     << x*x << "\n"
31     << (x>1) << "\t" << (x<1) << endl;
32
33     RMB y = x + 2;
34     cout << (x<y) << "\t" << (x<=y) << "\t"
35     << (x>y) << "\t" << (x>=y) << "\t"
36     << (x==y) << "\t" << (x!=y) << endl;
37
38     x = 2 + 2*x +3.5;
39     cout << x << endl;
40
41 }

```

(3) 测试本章设计的日期类 Date 中的运算符。请先指出程序的运行结果，再在计算机上进行验证，并对结果进行分析。

```

1 // DateTest.cpp
2 #include <iostream>
3 #include "Date.h"
4 using namespace std;
5
6 void main()
7 {
8     Date d;
9     d.SetDate(2008, 5, 1);
10    cout << ++d << endl;      cout << d << endl;
11    d.SetDate(2008, 5, 1);
12    cout << d++ << endl;      cout << d << endl;
13    d.SetDate(2008, 5, 1);
14    cout << ++(++d) << endl; cout << d << endl;
15    d.SetDate(2008, 5, 1);
16    cout << (++d)++ << endl; cout << d << endl;
17    d.SetDate(2008, 5, 1);
18    cout << ++(d++) << endl; cout << d << endl;
19    d.SetDate(2008, 5, 1);
20    cout << (d++)++ << endl; cout << d << endl;
21
22 }

```

(4) 设计 class VeryLongInt；能表示及处理 50 位的十进制整数。要求能进行算术四则运算、关系运算、赋值及迭代赋值运算、前（后）增（减）量运算、插入运算和抽取运算等，并且能够将字符串转换成本类类型的对象（参考练习 6 的 3（2））。

第 13 章 继承与多态性

继承与多态性是面向对象程序设计的重要特征。如果一种语言只支持类而不支持多态性（如 VB、Ada），便不能称这种语言是面向对象的，只能称其是基于对象的。

本章简要介绍 C++ 语言中关于继承和多态性方面的基本概念及语法要点。希望读者理解派生类对象的构造、空间结构；理解虚函数的作用；掌握虚函数的语法与多态性的基本应用技术。

13.1 继承与派生概述

13.1.1 抽象与具体

C++ 的类（class）是对一类事物的抽象，类的对象是事物的具体实例。从前面各章的学习中，对此应该有较为深刻的认识。本节将在一个更高的层次上讨论抽象与具体。

分类、分层是人们表达和处理复杂事物的基本手段，正确的分类及分层是人们掌握事物间相互关系，深刻认识了事物之间异同的具体体现。分类是人类认知过程中的一个很自然的环节，知识存在于比较和分类之间。

以“生物分类学”为例。生物分类学是认识生物多样性的基础，各种各样的生物将按“界、门、纲、目、科、属、种”七个层次归类。整个分类结构庞大，图 13-1 仅是其中的几个自底上溯至顶的小分支。

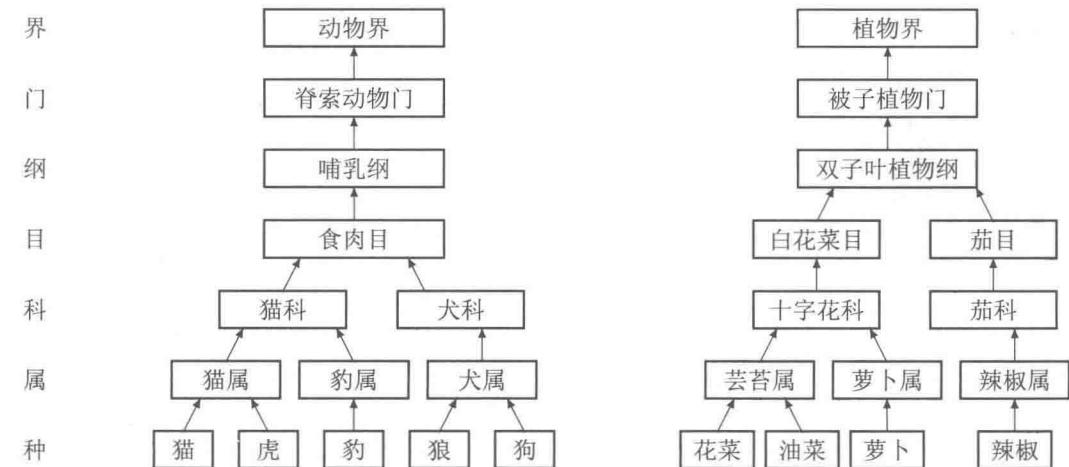


图 13-1 抽象与具体的层次结构

显然“界”的抽象程度高，“门”次之，依此类推，逐步从抽象到具体。这里所说的具体还不是指某个具体的一只猫或具体的某只狗，仍然是指一类对象的总称，仍然是一种抽象。用 C++ 的语言表示：仍然是类（class），而不是类的对象。

13.1.2 组合与继承

若要刻画“猫 (cat)”和“豹 (leopard)”，有如下方法。

(1) 方案一

分别独立地设计猫类 (`class Cat;`) 和豹类 (`class Leopard;`)。这两个类之间会出现许多相同的部分。由于类的封装性，它们之间相同之处无法借用，也不宜相互成为“朋友”（友元类），只能分别从头开始设计并实现之。

(2) 方案二

先设计一个猫类 (`class Cat;`)，然后组合到豹类 (`class Leopard;`) 中。即豹类中有一个猫类的对象做组合成员。这犯了逻辑上的错误，因为，“豹有重量、有年龄、有性别”不错，但说“豹有猫”是不正确的。

(3) 方案三（可行方案）

先找到它们的共性——“猫科动物 (feline)”，提取猫科动物的共性设计一个猫科动物类 (`class Feline;`) 包含猫科动物的基本属性、基本行为。然后由类 `Feline` 分别派生出 `Cat` 类和 `Leopard` 类。此时我们说：“猫是猫科动物，豹是猫科动物”。这完全符合生物学的分类。本方案是可行的，它克服了方案一中设计不能重用的不足，纠正了方案二中的逻辑错误。

从类 `Feline` 派生出类 `Cat`，我们也称类 `Cat` 继承了类 `Feline`。称 `Feline` 为基类，`Cat` 为派生类。`C++` 中派生类承袭了基类的所有属性和行为（除构造函数、拷贝构造函数和析构函数外），在派生类中仅需要补充一些新的属性、新的行为或扩充基类的行为（重载或覆盖基类的成员函数）。面向对象设计技术强调软件的可重用性。`C++` 语言提供的继承机制可用于解决软件重用的问题。

类的组合成员所刻画的是有 (has) 该属性的问题，类的继承或派生所刻画的是派生类的对象是 (is) 基类的对象。

同理，进一步考虑到刻画“狗 (dog)”的需要，需要上溯到“食肉目”。当然，比较彻底的、科学的设计方案是方案四。

(4) 方案四（推荐方案）

设计一个动物类 (`class Animal;`)，派生脊索动物门类 (`class Chordate;`)，再继续派生哺乳纲类 (`class Mammal;`)，依此类推。

这种分层、分类的设计方案有许多优点：避免了大量重复的工作；在每个层次中，能专心致志地处理好本层次及相关层次的事情；当需要修改某个层次的内容时，只要公开的 (`public`) 部分不变，则不会影响其他层次及整个系统结构。

再看一个例子：某学校有本科生 (Undergraduate) 和研究生 (Postgraduate)。他们的共同属性有：学号、姓名、性别、专业、学分和应缴纳的学费等；共同行为有：设置数据、返回数据、计算学费和输出数据等。本科生可能有选修第二专业的特殊属性和行为；研究生有导师（组合成员）、研究方向等特殊属性和行为。

因此，需要设计教师类 `class Teacher;` 学生类 `class Student;` 本科生类 `class Undergraduate;` 研究生类 `class Postgraduate;` 等。其中学生类作为基类，由它派生出本科生类、研究生类。按多文件结构共有 8 个文件，如表 13-1 所示。

表 13-1 继承与组合示例程序的文件组成

文件名	说明
Teacher.h	教师类声明文件
Student.h	学生类（基类）声明文件
Student.cpp	学生类成员函数、友元函数定义
Undergraduate.h	本科生（由学生类派生）类声明文件
Undergraduate.cpp	本科生类成员函数、友元函数定义
Postgraduate.h	研究生（由学生类派生）类声明文件
Postgraduate.cpp	研究生类成员函数、友元函数定义
Main.cpp	测试用的主程序

源代码 13.1 继承与组合示例程序（教师类）

```

1 // Teacher.h
2 #ifndef TEACHER_H
3 #define TEACHER_H
4 #include <iostream>
5 #include <cstring>
6 using namespace std;
7 class Teacher
8 {
9 public:
10    Teacher(char *Id="NoID", char *Name="NoName",
11             char *Title="Prof.", char *ResearchField="None")
12    {
13        strncpy(id, Id, sizeof(id));
14        id[sizeof(id)-1] = '\0';
15        strncpy(name, Name, sizeof(name));
16        name[sizeof(name)-1] = '\0';
17        strncpy(title, Title, sizeof(title));
18        title[sizeof(title)-1] = '\0';
19        strncpy(research_field, ResearchField,
20                 sizeof(research_field));
21        research_field[sizeof(research_field)-1] = '\0';
22    }
23    const char *Id() const { return id; }
24    const char *Name() const { return name; }
25    const char *Title() const { return title; }
26    const char *ResearchField() const { return research_field; }
27    friend ostream & operator<<(ostream &out, const Teacher &t)
28    {
29        out << t.id << " " << t.name << " "
30                << t.title << " " << t.research_field;
31        return out;
32    }
33
34 private:
35     char id[9], name[9], title[11];           // 工号, 姓名, 职称
36     char research_field[20];                  // 研究领域 (方向)
37 };
38 #endif

```

源代码 13.2 继承与组合示例程序（学生类）

```

1 // Student.h
2 #ifndef STUDENT_H
3 #define STUDENT_H
4 #include <iostream>
5 using namespace std;
6
7 class Student
8 {
9 public:
10    Student(char *Id="NoID", char *Name="NoName",
11             char Gender='M', char *Specialty="NoSpecialty");
12    void SetCredit(double Credit);      // 设置学分
13
14    void Tuition();                  // 计算学费
15    void Show(ostream &out) const;    // 输出信息
16    friend ostream& operator<<(ostream &out, const Student &s);
17
18 protected:                      // 受保护的，而不是私有的
19    char id[9], name[9], gender, specialty[20];
20                                // 学号，姓名，性别，专业
21    double credit, tuition;        // 学分，学费
22 };
23 #endif

```

源代码 13.3 继承与组合示例程序（本科生类）

```

1 // Undergraduate.h
2 #ifndef UNDERGRADUATE_H
3 #define UNDERGRADUATE_H
4 #include "Student.h"
5
6 class Undergraduate : public Student // 以 Student 为基类派生
7 {
8 public:
9    Undergraduate(char *Id="NoID", char *Name="NoName",
10                  char Gender='M', char *Specialty="NoSpecialty",
11                  char *Specialty1="No2ndSpec");
12
13    void Set2ndSpec(char *Specialty1); // 处理新增属性等的新行为
14    void Tuition();                // 覆盖所继承的 Tuition 函数
15    void Show(ostream &out) const;  // 覆盖所继承的 Show 函数
16    friend ostream & operator<<(ostream &out, const Undergraduate &ug);
17
18 private:                      // 私有的
19    char specialty1[20];          // 新增属性：第二专业
20 };
21 #endif

```

源代码 13.4 继承与组合示例程序（研究生类）

```

1 // Postgraduate.h
2 #ifndef POSTGRADUATE_H
3 #define POSTGRADUATE_H
4 #include "Student.h"
5 #include "Teacher.h"

```

```

6
7 class Postgraduate : public Student // 以 Student 为基类派生
8 {
9     public:
10    Postgraduate(Teacher &Advisor, char *Id="NoID", char *Name=
11        "NoName", char Gender='M', char *Specialty="NoSpecialty");
12
13    void SetAdvisor(Teacher &Advisor); // 处理新增属性等的新行为
14    void Tuition(); // 覆盖所继承的 Tuition 函数
15    void Show(ostream &out) const; // 覆盖所继承的 Show 函数
16
17    friend ostream & operator<<(ostream &out, const Postgraduate &pg);
18
19 private: // 私有的
20     Teacher advisor; // 新增属性。组合成员：导师
21 };
22 #endif

```

上述设计的本科生类 (Undergraduate) 是学生类 (Student) 的派生类，研究生类 (Postgraduate) 也是学生类 (Student) 的派生类。研究生类中拥有教师类 (Teacher) 的对象作为组合成员。这符合“本科生是学生，研究生是学生，研究生有导师”这种逻辑关系。继承是“纵向”的，而组合是“横向”的。

有关继承与派生的语法格式如下。

```

class 派生类名 : 继承方式 基类类名
{
    // 新增加的属性和行为，或者基类成员函数的覆盖、重载。
};

```

【说明】

① 基类是已经存在的类，任何已存在的类均能做基类。所声明的派生类是一个新类。

② 继承方式有公开继承、保护继承和私有继承。这三种继承方式分别借用保留字 `public`、`protected` 和 `private`。无论哪种继承方式，派生类都全部继承了基类的一切成员（基类的构造函数、拷贝构造函数和析构函数除外）。从基类继承而来的成员已经成为派生类的成员，在不引起混淆的情况下，我们仍称它们为基类的成员。这些成员的访问属性变化如表 13-2 所示。

③ 从基类继承而来的成员函数可以被覆盖（此时显式参数不变）^[1]，也可以被重载（此时显式参数不同）。

④ 派生类中可以增加新的数据成员和新的成员函数。

在上述派生类中，对于直接从基类继承而来的函数不要再声明（如设置学分的函数 `SetCredit`）；对于基类成员函数的重载或覆盖，必须在派生类中声明（如派生类中覆盖计算学费的函数 `Tuition`、输出信息的函数 `Show`）。在上述两个派生类中还各自添加了一个新的数据成员和一个新的成员函数。

下面给出各个类的成员函数定义。

^[1]实际上，在派生类新声明及定义的非静态成员函数中，其隐含参数 `this` 的数据类型与基类中的不同。本质上是重载从基类继承的成员函数。若需要访问基类的该成员函数可以用基类名及作用域区分符指明。参见 `Undergraduate::Show` 及 `Postgraduate::Show` 函数的定义。

源代码 13.5 继承与组合示例程序（学生类成员函数定义）

```

1 // Student.cpp
2 #include "Student.h"
3 #include <cstring>
4
5 Student::Student(char *Id, char *Name, char Gender,
6                   char *Specialty)
7 {
8     strncpy(id, Id, sizeof(id)); id[sizeof(id)-1] = '\0';
9     strncpy(name, Name, sizeof(name));
10    name[sizeof(name)-1] = '\0';
11    gender = (Gender=='m' || Gender=='M')? 'M' : 'F';
12    strncpy(specialty, Specialty, sizeof(specialty));
13    specialty[sizeof(specialty)-1] = '\0';
14    credit = tuition = 0;
15 }
16 void Student::SetCredit(double Credit)
17 {
18     credit = Credit;
19 }
20 void Student::Tuition()
21 {
22     // 不必执行任何语句
23 }
24 void Student::Show(ostream &out) const
25 {
26     out << id << " " << name << " "
27     << (gender=='M'? "男" : "女")
28     << " " << specialty;
29 }
30 ostream & operator<<(ostream &out, const Student &s)
31 {
32     s.Show(out);
33     return out;
34 }

```

派生类构造函数的格式如下。

```

派生类名::派生类构造函数名(总参数表)
    : 基类构造函数名(部分参数表),组合成员名(部分参数表),其他成员初始化
{
    // 派生类新增数据成员处理语句。
}

```

这里仍用冒号语法在最适当的时机构造从基类继承的数据成员。注意其中使用基类的构造函数名，组合成员则用成员名。

源代码 13.6 继承与组合示例程序（本科生类成员函数定义）

```

1 // Undergraduate.cpp 派生类——本科生类的成员函数定义
2 #include "Undergraduate.h"
3 #include <cstring>
4
5 Undergraduate::Undergraduate(char *Id, char *Name,
6                               char Gender, char *Specialty, char *Specialty1)
7   : Student(Id, Name, Gender, Specialty)
8 {

```

```

9     strncpy(specialty1, Specialty1, sizeof(specialty1));
10    specialty1[sizeof(specialty1)-1] = '\0';
11 }
12
13 void Undergraduate::Set2ndSpec(char *Specialty1)
14 {
15     strncpy(specialty1, Specialty1, sizeof(specialty1));
16     specialty1[sizeof(specialty1)-1] = '\0';
17 }
18
19 void Undergraduate::Tuition()          // 覆盖所继承的函数
20 {
21     tuition = credit * 100;           // 本科生每学分 100 元
22 }
23
24 void Undergraduate::Show(ostream &out) const
25 {
26     Student::Show(out);             // 直接调用基类的函数
27     out << "学分: " << credit << "学费" << tuition;
28     out << "第二专业: " << specialty1;
29 }
30
31 ostream & operator<<(ostream &out, const Undergraduate &ug)
32 {
33     ug.Show(out);
34     return out;
35 }

```

源代码 13.7 继承与组合示例程序（研究生类成员函数定义）

```

1 // Postgraduate.cpp 派生类——研究生类的成员函数定义
2 #include "Postgraduate.h"
3
4 Postgraduate::Postgraduate(Teacher &Advisor, char *Id,
5                             char *Name, char Gender, char *Specialty)
6     : Student(Id, Name, Gender, Specialty), advisor(Advisor)
7 {
8 }
9 void Postgraduate::SetAdvisor(Teacher &Advisor)
10 {
11     advisor = Advisor;           // 只需要浅赋值运算即可
12 }
13 void Postgraduate::Tuition()          // 覆盖所继承的函数
14 {
15     tuition = credit * 500;      // 研究生每学分 500 元
16 }
17 void Postgraduate::Show(ostream &out) const
18 {
19     Student::Show(out);         // 直接调用基类的函数
20     out << "学分: " << credit << "学费" << tuition;
21     out << "导师: " << advisor;
22 }
23
24 ostream & operator<<(ostream &out, const Postgraduate &pg)
25 {
26     pg.Show(out);
27     return out;
28 }

```

【说明】

① 按照 C++ “自己的事情自己做”的原则，派生类的构造函数应该接收足够多的参数，以便传递给基类的构造函数，用于构造对象从基类继承的数据成员^[1]。仍采用冒号语法在最适当的时机直接调用基类的构造函数。默认冒号部分时系统自动调用基类默认的构造函数构造基类数据成员的空间，并初始化它们。

② 研究生类 Postgraduate 的构造函数中，用冒号语法调用了基类的构造函数，还调用了教师类 Teacher 默认的拷贝构造函数来构造派生类的组合成员。

③ 派生类覆盖或重载从基类继承而来的成员函数时必须重新声明和定义。函数定义中仍可以使用基类的成员函数。如研究生类的 Show 函数，它不仅调用了基类的 Show 函数还调用了组合成员的插入运算符友元函数。请注意它们的格式不同，它们应该分别理解为 `this->Student::Show();` 和 `operator<<(out, this->advisor);`。

13.1.3 派生类成员的访问属性

表 13-2 基类成员在派生类中的访问属性

基类成员访问属性	公开继承	保护继承	私有继承
<code>public</code>	<code>public</code>	<code>protected</code>	<code>private</code>
<code>protected</code>	<code>protected</code>	<code>protected</code>	<code>private</code>
<code>private</code>	被隔离，不可访问	被隔离，不可访问	被隔离，不可访问

从表 13-2 的规定可以看到类中成员的访问属性为 `protected` 与 `private` 的区别。基类的私有成员在派生类中是存在的，但是被隔离起来，不能直接访问。若要访问它们，只能通过基类的成员函数。绝大多数情况下，程序设计都采用公开继承，很少使用保护继承和私有继承。因此，本章仅使用和讨论公开继承方式的继承。

13.2 派生类对象的构造

13.2.1 派生类对象的构造与析构

在构造派生类对象时，系统将先调用基类适当的构造函数或拷贝构造函数来构造从基类继承的数据成员部分（显式或隐式的冒号语法），其次调用组合成员的相应构造函数或拷贝构造函数来构造组合成员，然后构造并初始化其他数据成员（显式或隐式的冒号语法），最后进入派生类的构造函数体执行其中的语句。析构派生类对象的顺序与构造顺序相反：先执行派生类的析构函数体语句，然后执行组合成员的析构函数，最后再执行基类的析构函数。

在设计完成所有的类后，编写一个测试用的文件，包含如下三个函数。

^[1] 虽然基类公开的数据成员和受保护的数据成员可以由派生类的构造函数来赋值，但是，若基类有私有数据成员，则在派生类的构造函数体中将无法访问它们。因此，将参数传递给基类的构造函数是最好的解决方案，实现真正的初始化。

源代码 13.8 继承与组合示例程序（主程序文件）

```

1 // Main.cpp
2 #include "Undergraduate.h"
3 #include "Postgraduate.h"
4
5 void UGOffice(Undergraduate &x, double credit)
6 {                                // 模拟财务部门专为本科生开设的办事处
7     x.SetCredit(credit);
8     x.Tuition();
9     cout << x << endl;
10 }
11
12 void PGOffice(Postgraduate &x, double credit)
13 {                                // 模拟财务部门专为研究生开设的办事处
14     x.SetCredit(credit);
15     x.Tuition();
16     cout << x << endl;
17 }
18
19 int main()
20 {
21     Teacher t1("1001", "张明", "教授", "计算机科学");
22     Teacher t2("1002", "李亮", "研究员", "数学");
23     Undergraduate s1("2001", "赵小刚", 'm', "力学");
24     Undergraduate s2("2002", "钱程锦", 'f', "数学", "管理科学");
25     Postgraduate g1(t2, "3001", "周有才", 'm', "数学");
26     Postgraduate g2(t1, "3002", "吴岚", 'f', "计算机科学");
27
28     UGOffice(s1, 28);           // 函数调用，实参与形参类型严格匹配
29     UGOffice(s2, 30);
30     PGOffice(g1, 10);
31     PGOffice(g2, 12);
32     return 0;
33 }
```

将上述八个文件添加到一个项目文件（或称为工程文件）中，编译连接生成可执行文件后，运行结果如下。

2001 赵小刚 男 力学 学分: 28 学费2800 第二专业: No2ndSpec
2002 钱程锦 女 数学 学分: 30 学费3000 第二专业: 管理科学
3001 周有才 男 数学 学分: 10 学费5000 导师: 1002 李亮 研究员 数学
3002 吴岚 女 计算机科学 学分: 12 学费6000 导师: 1001 张明 教授 计算机科学

【说明】 整个程序中创建了两个教师对象、两个本科生对象及两个研究生对象。并没有创建纯粹的“学生”对象。

13.2.2 派生类对象的空间

前面已经提到派生类继承了基类的所有数据成员，还可以添加新的数据成员。因此派生类对象的数据空间中的一部分可以看成基类对象的空间。图 13-2 是以上面的研究生类对象 g1 为例的对象空间结构示意图。

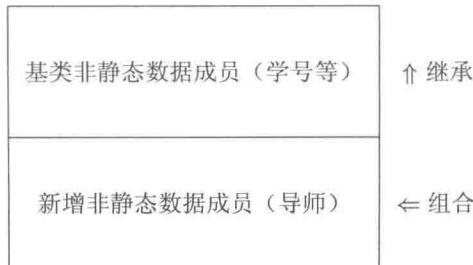


图 13-2 派生类对象的空间结构

13.2.3 派生类对基类的赋值兼容性

强调“本科生是学生，研究生是学生”，而“研究生有导师”。反之不然：不能认为“学生一定是本科生”，也不能认为“学生一定是研究生”。那么将“本科生”或者“研究生”作为“学生”看待时是“完整的”学生。本科生对象可以：

① 初始化（拷贝构造）或赋值给学生类的对象（当然只用到基类的那一部分非静态数据成员）。

② 可以初始化学生类的引用。

③ 本科生对象的地址可以给学生类的指针初始化或赋值。

即派生类的对象可以初始化（拷贝构造）基类的对象、初始化基类的引用，可以赋值给基类的对象。派生类对象的地址可以初始化基类的指针变量，可以赋值给基类的指针变量。这种特性称为派生类对象对基类对象的初始化及赋值兼容性。

13.3 多态性

上一节中，介绍了继承与派生的概念。程序中留下了三个伏笔（下面最先讨论第三方面的内容）。

(1) 基类 `Student` 中的成员函数 `Tuition` 是无用的。的确，在上面的程序中这个函数是可以去掉的。我们并不打算去掉它，而是改变其形式（参见第 13.3.4 小节）。

(2) 重载插入运算符时没有直接利用友元的身份访问对象的数据成员，而是调用成员函数 `Show`。并且 `Show` 函数的形式参数虽好理解，但有些怪异（原因参见第 13.3.2 小节）。

(3) 两个函数：

```

void UGOffice(Undergraduate &x, double credit)
{
    // 模拟财务部门专为本科生开设的办事处
    x.SetCredit(credit);
    x.Tuition();
    cout << x << endl;
}

void PGOffice(Postgraduate &x, double credit)
{
    // 模拟财务部门专为研究生开设的办事处
    x.SetCredit(credit);
    x.Tuition();
    cout << x << endl;
}

```

分别代表财务部门两个不同的服务窗口，本科生、研究生不能去错服务窗口。倘若学校的学生有更多的层次（如还有专科生、进修学生和博士研究生等），则要设置更多的专门窗口，岂不是“机构臃肿”。学校的财务部门能否做到“接待所有的学生，按具体学生办理相关事情”。这里的“所有的学生”可能是抽象的，而“具体学生”一定是具体的。

即要求只设计一个统一的函数（由于需要操作实参对象，因此需要引用型或指针型形式参数）处理各类学生。

```
void Office(Student &x, double credit)
{
    // 模拟财务部门为学生开设的办事处
    x.SetCredit(credit);
    x.Tuition();
    cout << x << endl;
}
// 或者
void Office(Student *x, double credit)
{
    // 模拟财务部门为学生开设的办事处
    x->SetCredit(credit);
    x->Tuition();
    cout << *x << endl;
}
```

在主函数中调用格式相应地改成如下形式。

```
Office(s1, 28);           // 形参为基类引用，实参为派生类对象
Office(s2, 30);
Office(g1, 10);
Office(g2, 12);
cout << endl;

Office(&s1, 28);          // 形参为基类指针，实参为派生类对象地址
Office(&s2, 30);
Office(&g1, 10);
Office(&g2, 12);
cout << endl;
```

程序的运行结果：

```
2001 赵小刚 男 力学
2002 钱程锦 女 数学
3001 周有才 男 数学
3002 吴岚 女 计算机科学
```

```
2001 赵小刚 男 力学
2002 钱程锦 女 数学
3001 周有才 男 数学
3002 吴岚 女 计算机科学
```

从运行结果可知，没有达到预期的目标：仅输出了基类的数据成员，派生类添加的新数据成员均未输出。即只执行了基类的友元插入运算符函数。事实上在调用函数 `Office` 时，将本科生或研究生皆视为学生，而无法处理实参对象的差异。

回顾函数重载（包括运算符重载），或者考虑函数模板（相当于由系统自动重载函数）。

```
template <typename T> void Office(T &x, double credit)
{
    x.SetCredit(credit);
    x.Tuition();
    cout << x << endl;
}
```

这样似乎解决了问题。而事实上这种处理与最初的程序一样，本质上是由系统根据模板自动生成两个模板函数（分别用类型 `Undergraduate`, `Postgraduate` 取代形式数据类型 `T`）。总之，利用函数重载、模板技术的本质还是定义了多个函数。即服务部门专门为各种类型的对象分别开设不同的服务窗口。与所希望的“设置统一的服务窗口，来者都是学生，之后再看对象类型”相距甚远。

上述各种处理方法均能在程序编译阶段就完全确定所将要调用的函数，用面向对象程序设计的术语称为先期联编或静态多态性或编译时的多态性。前面各章节介绍的函数重载、运算符重载及模板技术均为静态多态性。

与之相对应的是迟后联编或动态多态性或运行时的多态性。迟后联编所调用的函数不是在编译阶段确定的，而是在程序运行过程中根据实际对象动态地确定的。运行时的多态性是面向对象程序设计的一个重要特征。这种特性的实现机制是虚函数。

13.3.1 虚函数

设计一个类时，使用保留字 `virtual` 可将成员函数声明为虚函数^[1]。虚函数并非是“虚假的”，虚函数是实实在在的成员函数。使用虚函数会略微增加一些空间开销和很少的时间开销。总体来讲，虚函数将给程序带来方便，实现“一个接口，多种方法”，同时由于采用指针或引用操作，程序的执行效率依然很高。

再讨论上述程序，将基类 `Student` 中计算学费和输出信息的成员函数（`Tuition` 和 `Show`）声明成虚函数。仅修改头文件 `Student.h` 中第 14 和 15 行，其他有关类设计的文件的内容一律不变。

源代码 13.9 继承与组合示例程序（含虚函数的学生类）

```
1 // Student.h
2 #ifndef STUDENT_H
3 #define STUDENT_H
4 #include <iostream>
5 using namespace std;
6
7 class Student
8 {
9 public:
10     Student(char *Id="NoID", char *Name="NoName",
11             char Gender='M', char *Specialty="NoSpecialty");
12     void SetCredit(double Credit);           // 设置学分
13 }
```

^[1]对于含有虚函数、纯虚函数的类，编译系统为该类建一个虚函数表（virtual function table, vtable），它是一个指针数组，存放每个虚函数的入口地址，供该类的所有对象共享。同时在类中悄悄添加一个特殊指针成员指向该虚函数表。每当对象调用成员函数时，将首先访问该特殊指针，找到虚函数表，最后找到所要执行的函数。实现运行时的多态性。

```

14     virtual void Tuition();           // 计算学费（虚函数）
15     virtual void Show(ostream &out) const; // 输出信息（虚函数）
16     friend ostream& operator<<(ostream &out, const Student &s);
17
18 protected:                      // 受保护的，而不是私有的
19     char id[9], name[9], gender, specialty[20];
20                               // 学号，姓名，性别，专业
21     double credit, tuition;          // 学分，学费
22 };
23 #endif

```

主程序修改成如下形式。

源代码 13.10 继承与组合示例程序（使用虚函数的主程序）

```

1 // Main.cpp
2 #include "Undergraduate.h"
3 #include "Postgraduate.h"
4
5 void Office(Student &x, double credit)
6 {                           // 模拟财务部门为学生开设的办事处
7     x.SetCredit(credit);
8     x.Tuition();
9     cout << x << endl;
10 }
11
12 void Office(Student *x, double credit)
13 {                           // 模拟财务部门为学生开设的办事处
14     x->SetCredit(credit);
15     x->Tuition();
16     cout << *x << endl;
17 }
18
19 void Office1(Student x, double credit)
20 {                           // 形参为值传递，创建学生对象，不办实事
21     x.SetCredit(credit);
22     x.Tuition();
23     cout << x << endl;
24 }
25
26 int main()
27 {
28     Teacher t1("1001", "张明", "教授", "计算机科学");
29     Teacher t2("1002", "李亮", "研究员", "数学");
30     Undergraduate s1("2001", "赵小刚", 'm', "力学");
31     Undergraduate s2("2002", "钱程锦", 'f', "数学", "管理科学");
32     Postgraduate g1(t2, "3001", "周有才", 'm', "数学");
33     Postgraduate g2(t1, "3002", "吴岚", 'f', "计算机科学");
34
35     cout << "Part_1:" << endl;           // 呈现多态性
36     Office(s1, 28);                   // 形参为基类引用，实参为派生类对象
37     Office(s2, 30);
38     Office(g1, 10);
39     Office(g2, 12);
40
41     cout << "\nPart_2:" << endl;           // 呈现多态性
42     Office(&s1, 28);                   // 形参为基类指针，实参为派生类对象地址
43     Office(&s2, 30);

```

```

44     Office(&g1, 10);
45     Office(&g2, 12);
46
47     cout << "\nPart_3:" << endl;      // 呈现多态性
48     Student &rs = s1, *ps = &s2, &rg = g1, *pg = &g2;
49         // 定义基类的指针或声明基类的引用，用派生类进行相应初始化
50     rs.SetCredit(29);      rs.Tuition();
51     cout << rs << endl;
52     ps->SetCredit(31);    ps->Tuition();
53     cout << *ps << endl;
54     rg.SetCredit(11);      rg.Tuition();
55     cout << rg << endl;
56     pg->SetCredit(13);    pg->Tuition();
57     cout << *pg << endl;
58
59     cout << "\nPart_4:" << endl;      // 基类对象的自我表现
60     Student x = s1, y = g1; // 创建基类对象，用派生类对象初始化
61     x.SetCredit(28);      x.Tuition();
62     cout << "x:_" << x << endl;
63     y.SetCredit(30);      y.Tuition();
64     cout << "y:_" << y << endl;
65
66     cout << "\nPart_5:" << endl;      // 基类对象的自我表现
67     Office1(s1, 28);        // 形参为值传递型基类对象
68     Office1(s2, 30);
69     Office1(g1, 10);
70     Office1(g2, 12);
71     cout << endl;
72
73     return 0;
74 }

```

程序的运行结果：

Part 1:

```

2001 赵小刚 男 力学 学分: 28 学费2800 第二专业: No2ndSpec
2002 钱程锦 女 数学 学分: 30 学费3000 第二专业: 管理科学
3001 周有才 男 数学 学分: 10 学费5000 导师: 1002 李亮 研究员 数学
3002 吴岚 女 计算机科学 学分: 12 学费6000 导师: 1001 张明 教授 计算机科学

```

Part 2:

```

2001 赵小刚 男 力学 学分: 28 学费2800 第二专业: No2ndSpec
2002 钱程锦 女 数学 学分: 30 学费3000 第二专业: 管理科学
3001 周有才 男 数学 学分: 10 学费5000 导师: 1002 李亮 研究员 数学
3002 吴岚 女 计算机科学 学分: 12 学费6000 导师: 1001 张明 教授 计算机科学

```

Part 3:

```

2001 赵小刚 男 力学 学分: 28 学费2800 第二专业: No2ndSpec
2002 钱程锦 女 数学 学分: 30 学费3000 第二专业: 管理科学
3001 周有才 男 数学 学分: 10 学费5000 导师: 1002 李亮 研究员 数学
3002 吴岚 女 计算机科学 学分: 12 学费6000 导师: 1001 张明 教授 计算机科学

```

Part 4:

```

x: 2001 赵小刚 男 力学
y: 3001 周有才 男 数学

```

Part 5:

```

2001 赵小刚 男 力学

```

2002 钱程锦 女 数学
3001 周有才 男 数学
3002 吴岚 女 计算机科学

Part 1, 2, 3 的本质是类似的，它们采用基类引用派生类对象，或者基类指针变量访问派生类目标对象。由于虚函数的作用实现了运行时的多态性。

Part 4, 5 的本质上是相同的，它们都是创建了基类的对象，具体操作时已经与派生类的对象无关。因而输出如上结果。

基类 **Student** 中两个函数 (**Tuition** 和 **Show**) 分别设计成虚函数或非虚函数共有四种组合。请读者就四种情形分别编译运行之，并分析运行的结果。

【注意】

(1) 基类中的虚函数会自动地遗传给其派生类。在派生类中覆盖（参数及返回类型不变）基类的虚函数时已经隐含为虚函数，因此可省略派生类中的保留字 **virtual**。建议在派生类中明确地用保留字 **virtual** 表示虚函数，以增加程序的可读性。

(2) 虚函数在类体外定义时不能用保留字 **virtual**。

(3) 根据赋值兼容性，用基类的引用或基类的指针访问派生类的对象时，可以实现迟后联编。但是，用派生类的对象初始化基类对象后，再访问基类对象则与派生类的对象无关，仅是基类对象的自我表现。

13.3.2 重载运算符享受多态性

现在讨论本节开始时所述的第二个伏笔。若被重载的运算符是成员函数，则可以将该运算符声明成虚函数以实现动态多态性。

但是，有些运算符的第一个操作数不是本类的对象（例如插入运算符、抽取运算符等），常将这种运算符声明并定义成类的友元函数。友元函数不能是虚函数，但在友元函数中可以调用虚函数。因此，常声明并定义一个与运算符功能相同的成员函数并将该函数声明成虚函数（如 **Student** 类中的 **Show** 函数）；然后在重载运算符函数中调用该虚函数，便可让运算符函数享受多态性带来的便利（如 **Student** 类中的插入运算符友元函数）。

若该虚函数的访问属性是公开的，则重载的运算符函数还不必是友元。若该虚函数是受保护的或私有的，则需要声明运算符函数为友元。读者可以将例子中类 **Student**，类 **Undergraduate** 及类 **Postgraduate** 所声明的友元函数（插入运算符重载）的声明删除，改在类体外声明并定义成普通的 C++ 函数（不是任何类的友元函数），程序的其他部分不需做任何改变，也可直接编译连接并运行。

13.3.3 虚析构函数

前面曾介绍过对象带资源的类需要深拷贝构造函数、深赋值运算符函数，还需要析构函数。在前面的有些程序中我们将析构函数设计成虚函数（如 **LinkList** 类模板，自定义版字符串类 **String**）。

下面举例说明，可能做基类的类应该定义其析构函数，而且还要将该析构函数声明为虚函数。

源代码 13.11 虚析构函数的必要性

```

1 // TestVirtualDestructor.cpp
2 #include <iostream>
3 using namespace std;
4
5 #define VirtualDestructor
6 // 本程序实为两个程序：有或无上述编译预处理宏定义。请分别测试。
7 // 请参阅第 7.2.3 小节
8
9 class BASE                                // 基类
10 {
11 public:
12     BASE(){ cout << "构造基类对象。" << endl; }
13     #ifdef VirtualDestructor
14         virtual ~BASE();                  // 虚析构函数
15     #else
16         ~BASE();                      // 非虚析构函数
17     #endif
18 };
19
20 BASE::~BASE()
21 {
22     cout << "析构基类对象。" << endl;
23 }
24
25 //-----
26 class Derived : public BASE                // 派生类
27 {
28 public:
29     Derived(int n=10)
30     {
31         x = NULL;
32         if((size=n) > 0)
33         {
34             x = new double[size];
35             for(int i=0; i<size; i++)
36                 x[i] = 0;
37         }
38         else size = 0;
39         cout << "构造派生类对象: [Size]=[" << size << endl;
40     }
41
42     Derived(const Derived &d)
43     {
44         if((size=d.size)==0) x = NULL;
45         else x = new double[size];
46         for(int i=0; i<size; i++)
47             x[i] = d.x[i];
48         cout << "拷贝构造派生类对象: [Size]=[" << size << endl;
49     }
50
51     Derived & operator=(const Derived &d)
52     {
53         if(&d == this) return *this;
54         if(x!=NULL) delete [] x;
55         if((size=d.size)==0) x = NULL;
56         else x = new double[size];
57         for(int i=0; i<size; i++)

```

```

58         x[i] = d.x[i];
59         cout << "派生类对象间赋值: Size=" << size << endl;
60         return *this;
61     }
62
63     ~Derived()
64     {
65         if(x!=NULL) delete [] x;
66         cout << "析构派生类对象: Size=" << size << endl;
67     }
68
69 protected:
70     int size;
71     double *x; // 带堆资源
72 };
73
74 int main()
75 {
76     BASE *pBB = new BASE;
77     delete pBB;
78     cout << "BASE,OK.\n" << endl;
79
80     Derived *pDD = new Derived(8);
81     delete pDD;
82     cout << "Derived,OK.\n" << endl;
83
84     BASE *pBD = new Derived(5);
85     delete pBD;
86     cout << "请观察是否调用了派生类的析构函数。" << endl;
87     return 0;
88 }
```

上述程序实际上包含两个版本，本质上是两个程序。

(1) 程序之一

保持程序中第 5 行的编译预处理宏定义指令，则编译器将编译第 14 行的语句，第 16 行的语句不参加编译，即基类的析构函数为虚函数。程序的运行结果如下。

```

构造基类对象。
析构基类对象。
BASE, OK.

构造基类对象。
构造派生类对象: Size = 8
析构派生类对象: Size = 8
析构基类对象。
Derived, OK.

构造基类对象。
构造派生类对象: Size = 5
析构派生类对象: Size = 5
析构基类对象。
请观察是否调用了派生类的析构函数。
```

将其中的一行输出特意用黑体字排版。有此行输出则表明派生类的析构函数已经被调用，其对象所带的堆内存资源被安全释放。

(2) 程序之二

删除程序中第 5 行的编译预处理宏定义指令（只要将该行作为注释行即可），则编译器将编译第 16 行的语句，第 14 行的语句不参加编译，即基类的析构函数不是虚函数。程序的运行结果如下。

```
构造基类对象。  
析构基类对象。  
BASE, OK.  
  
构造基类对象。  
构造派生类对象: Size = 8  
析构派生类对象: Size = 8  
析构基类对象。  
Derived, OK.  
  
构造基类对象。  
构造派生类对象: Size = 5  
析构基类对象。  
请观察是否调用了派生类的析构函数。
```

从程序的运行结果上看，最后的派生类的析构函数未被调用，即派生类对象所带的堆内存资源未被释放（将造成内存泄露）。

造成这种结果的根本原因是没有将基类的析构函数声明成虚函数。当使用基类的指针操作派生类对象时，对非虚函数仅操作基类部分，无法处理派生类新增的数据空间；而对于虚函数，则根据对象的类型操作对象的数据空间。

从类的设计来看，派生类的设计是无懈可击的（深拷贝构造、深赋值运算和析构函数一应俱全）；而基类非常单纯，其设计也似乎是“没有问题”。这个例子告诉我们，若某个类可能成为基类，最好将其析构函数声明为虚函数。

13.3.4 纯虚函数与抽象类

现在讨论本节开始时所述的第一个伏笔。

我们再次讨论前面提到的 `Student` 类。若不需要运行时的多态性，则其中计算学费的函数 `Tuition` 可以去掉；若要使计算学费函数 `Tuition` 能迟后联编，则基类中的这个函数是必须的且应该声明为虚函数。与另一个输出信息的虚函数 `Show` 不同的是基类中计算学费的函数“只占位置、不做事情”。这样的函数体不要也罢，让其成为一个纯粹的虚函数——纯虚函数。

纯虚函数是一种特殊的虚函数。在虚函数声明的函数首部后（分号前）写下记号“=0”后该虚函数就是一个纯虚函数。纯虚函数没有函数体（当然也不需要定义），其作用只是“占个位置”，便于实现多态性。

至少含有一个纯虚函数的类被称为抽象类。抽象类的唯一作用就是用来被继承的。在派生类中可以覆盖定义基类的纯虚函数。抽象类的派生类也有可能仍然是抽象类（只要其中还有纯虚函数未被覆盖定义）。当一个类中没有纯虚函数时，这个类便成为具体类。

不能创建抽象类的对象；可以定义抽象类的指针变量指向由其派生的具体类的对象；可以声明抽象类的引用，声明引用时必须用其派生的具体类的对象进行初始化。

因此，可以将 `Student` 类的计算学费函数 `Tuition` 声明成纯虚函数。这样一来，类 `Student` 就是一个抽象类，此时不能创建 `Student` 类的对象，即“不存在抽象的学生”。

源代码 13.12 继承与组合示例程序（抽象类）

```

1 // Student.h
2 #ifndef STUDENT_H
3 #define STUDENT_H
4
5 class Student
6 {
7 public:
8     Student(char *Id="NoID", char *Name="NoName",
9             char Gender='M', char *Specialty="NoSpecialty");
10    void SetCredit(double Credit);           // 设置学分
11    virtual void Tuition() = 0;            // 计算学费（纯虚函数）
12    virtual void Show();                  // 输出信息（虚函数）
13
14 protected:                         // 受保护的，而不是私有的
15     char id[9], name[9], gender, specialty[20];
16                                         // 学号，姓名，性别，专业
17     double credit, tuition;            // 学分，学费
18 };
19#endif

```

当然，应该删除 `Student.cpp` 文件中原来的关于 `Tuition` 函数的定义。

13.3.5 关于虚函数的说明

C++ 编译器默认是在先期联编的状态下编译程序的，只有当编译器遇到了虚函数、纯虚函数后，才将该函数按照迟后联编处理。由于动态多态性是在程序运行时“认具体对象”，因此关于虚函数的应用有如下要求或建议。

(1) 只有类的成员函数才能被声明为虚函数，以实现动态多态性。可以考虑将所有可以设置成虚函数的成员函数都声明成虚函数是有好处的。

(2) 派生类直接继承基类的虚函数特性（派生类中可以省略保留字 `virtual`），虚函数可以被覆盖。但重载虚函数（形式参数不同）或返回类型不同的函数将不是虚函数，除非另行声明。虚函数在类体外定义时不能有保留字 `virtual`。

(3) 纯虚函数没有函数体。抽象类就是用来被继承的。不能创建抽象类的对象，但可以定义抽象类的指针变量、声明抽象类的引用。

(4) 构造函数、拷贝构造函数不能是虚函数，因为虚函数是认具体对象的，而执行构造函数时所处理的对象尚未构造完成，对象尚未成型。

(5) 静态成员函数不能是虚函数，因为静态成员函数不默认联系某对象。

(6) 析构函数可以是虚函数。而且最好将析构函数声明为虚函数，以便当对象生命周期结束时能够确实调用自己的析构函数完成善后工作。

(7) 为了使一些作为友元函数的运算符享受到动态多态性带来的便利（如插入运算符“`<<`”等），应该在基类中声明一个虚函数或纯虚函数以便在派生类中覆盖，并被重载运算符函数调用。

13.4 多重继承

前面所介绍的继承都只是单继承（包括多层派生——派生类的派生类）。其实客观事物有时是“一体多能”、“一身兼任数职”的。这意味着派生类从多个基类继承而来。

例如，在职攻读研究生学位的教师（既是教师又是研究生）；优秀学生兼任学生辅导员（既是学生又兼任教师工作）；专业技术人员兼任管理工作（既是员工又是管理者）等。又如“打印复印一体机”（既是打印机，又是复印机，还可能是电话机、传真机和扫描仪）。再如“手机”（是电话机、MP3 播放器、MP4 播放器、照相机、电视机、收音机、录音机、U 盘、闹钟和游戏机等）一机多能。

按继承的设计模式，对于这种“一身兼任数职”的事物应该有多个基类——这就是需要多重继承的原因。本书不打算详细介绍多重继承的有关内容，仅简单说明多重继承的语法、多重继承所带来的新问题（成员冲突和冗余）和解决新问题所引出的新语法规规定（虚拟继承）。

13.4.1 多重继承的一般形式

多重继承的语法格式如下。

```
class 派生类名 : 继承方式 基类类名1, 继承方式 基类类名2 ...
{
    // 新增加的属性和行为，或者基类成员函数的覆盖、重载。
};
```

派生类构造函数的参数应该可供所有基类数据成员的初始化使用，仍然通过冒号语法将参数传递给各个基类的构造函数或拷贝构造函数。

由于派生类继承了基类的所有成员（构造函数、拷贝构造函数和析构函数除外），在有多个基类时难免出现数据成员名重复的情况，此时将产生数据的冗余。例如：教师在职攻读研究生的对象中，同一个对象将有多个编号、姓名、性别和年龄等。其中多个编号是合理的，但多个姓名（即使是相同的），也未免不合适。另外，同一个对象将可能出现多个设置函数和输出函数。

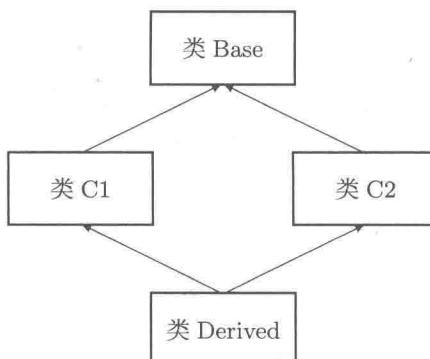


图 13-3 多重继承中可能的数据冗余

图 13-3 给出了由共同基类间接派生类的情形。派生类 `Derived` 通过分别继承类 `C1`, `C2` 两次间接地继承了它们的共同基类 `Base` 的数据成员, 这样不仅仅造成数据的冗余, 而且造成数据的二义性 (ambiguous), 参见下一小节“虚拟继承示例”程序的编译说明。解决这个问题需要引入新的机制——虚拟继承。

13.4.2 虚拟继承

虚拟继承能解决上一小节所提出的问题, 如图 13-4 所示。

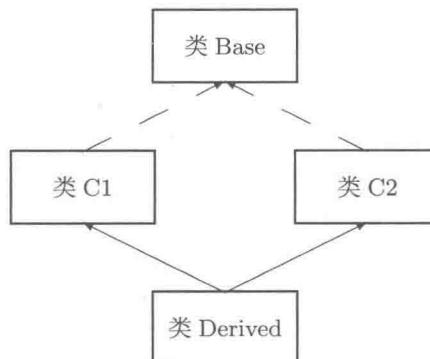


图 13-4 虚基类及虚拟继承

采用虚拟继承派生类 `C1` 及 `C2`, 则它们共同的基类被称为虚基类。虚拟继承仍然使用 C++ 保留字 `virtual`, 但与虚函数、纯虚函数概念不同。虚拟继承能有效地除去数据成员的冗余、消除成员函数的重复。虚拟继承的格式如下。

```

class 派生类名 : virtual 继承方式 基类类名
{
    // 新增加的属性和行为, 或者基类成员函数的覆盖、重载。
};
  
```

源代码 13.13 虚拟继承示例

```

1 // VirtualInheritance.cpp
2 #include <iostream>
3 using namespace std;
4
5 class Base
6 {
7 public:
8     Base(int x=0) { b = x; }
9     void Show() { cout << "b=" << b << endl; }
10
11 protected:
12     int b;
13 };
14
15 class C1 : virtual public Base           // 虚拟继承
16 {
17 public:
18     C1(int x=1) { c1 = x; }
19     void Set(int x, int y)
20     {
  
```

```
21         b = x, c1 = y;
22     }
23     void Show()
24     {
25         Base::Show();
26         cout << "c1=" << c1 << endl;
27     }
28
29 protected:
30     int c1;
31 };
32
33 class C2 : virtual public Base           // 虚拟继承
34 {
35 public:
36     C2(int x=2) { c2 = x; }
37     void Set(int x, int y)
38     {
39         b = x, c2 = y;
40     }
41     void Show()
42     {
43         Base::Show();
44         cout << "c2=" << c2 << endl;
45     }
46
47 protected:
48     int c2;
49 };
50
51 class Derived : public C1, public C2      // 多重继承
52 {
53 public:
54     Derived(int x=3) { d = x; }
55     void SetB(int x) { b = x; }
56     void SetC1(int x, int y) { C1::Set(x, y); }
57     void SetC2(int x, int y) { C2::Set(x, y); }
58     void Set(int x) { d = x; }
59     void Show()
60     {
61         C1::Show();
62         C2::Show();
63         Base::Show();
64         cout << "d=" << d << endl;
65     }
66
67 private:
68     int d;
69 };
70
71 int main()
72 {
73     Derived d;
74     d.SetC1(100, 200); d.Show(); cout << endl;
75     d.SetC2(500, 800); d.Show(); cout << endl;
76     return 0;
77 }
```

程序的运行结果：

```
b = 100
c1 = 200
b = 100
c2 = 2
b = 100
d = 3

b = 500
c1 = 200
b = 500
c2 = 800
b = 500
d = 3
```

从运行结果可见，当语句 `d.SetC1(100, 200);` 被执行后，`C2` 对应的 `b` 也发生了变化；当语句 `d.SetC2(500, 800);` 被执行后，`C1` 对应的 `b` 也发生了变化。这表面在派生类 `Derived` 的对象 `d` 中确实只有一份基类 `Base` 的数据成员 `b`。

在上述程序中，若去掉第 15 行及第 33 行中的保留字 `virtual`，则程序在用 GCC 编译器编译时给出如下错误信息。

```
-----Configuration: VirtualInheritance - Debug-----
Compiling source file(s)...
VirtualInheritance.cpp
VirtualInheritance.cpp: In member function 'void Derived::SetB(int)':
VirtualInheritance.cpp:55: error: request for member 'b' is ambiguous in
multiple inheritance lattice
VirtualInheritance.cpp:12: error: candidates are: int Base::b
VirtualInheritance.cpp:12: error:           int Base::b
VirtualInheritance.cpp: In member function 'void Derived::Show()':
VirtualInheritance.cpp:63: error: cannot call member function 'void
Base::Show()' without object
ConstructDestruct.exe - 4 error(s), 0 warning(s)
```

13.5 构造顺序

构造一个派生类对象时，可能涉及一系列构造函数的调用。它们的调用顺序是：

- (1) 所有虚基类的构造函数按照它们被继承的顺序构造。
- (2) 所有非虚基类的构造函数按照它们被继承的顺序构造。
- (3) 所有组合成员对象的构造函数按照它们声明的顺序构造。
- (4) 执行派生类构造函数体语句。

前三项均通过冒号语法执行。若定义派生类的构造函数未显式地使用冒号语法，则系统隐式地采用冒号语法调用相应的默认构造函数构造之。

析构的顺序与构造顺序相反：

- (1) 执行派生类析构函数体语句。
- (2) 所有组合成员对象的析构函数按照它们声明的相反顺序执行。
- (3) 所有非虚基类的析构函数按照它们被继承的相反顺序执行。

(4) 所有虚基类的析构函数按照它们被继承的相反顺序执行。

如下例所示。

源代码 13.14 继承与组合中派生类对象的构造与析构顺序

```
1 // ConstructDestruct.cpp
2 #include <iostream>
3 using namespace std;
4 class Obj1
5 {
6 public:
7     Obj1(){ cout << "Obj1" << endl; }
8     ~Obj1(){ cout << "~Obj1" << endl; }
9 };
10 class Obj2
11 {
12 public:
13     Obj2(){ cout << "Obj2" << endl; }
14     ~Obj2(){ cout << "~Obj2" << endl; }
15 };
16 class BASE1
17 {
18 public:
19     BASE1(){ cout << "BASE1" << endl; }
20     ~BASE1(){ cout << "~BASE1" << endl; }
21 };
22 class BASE2
23 {
24 public:
25     BASE2(){ cout << "BASE2" << endl; }
26     ~BASE2(){ cout << "~BASE2" << endl; }
27 };
28 class BASE3
29 {
30 public:
31     BASE3(){ cout << "BASE3" << endl; }
32     ~BASE3(){ cout << "~BASE3" << endl; }
33 };
34 class BASE4
35 {
36 public:
37     BASE4(){ cout << "BASE4" << endl; }
38     ~BASE4(){ cout << "~BASE4" << endl; }
39 };
40 class Derived : public BASE1, virtual public BASE2,
41                 public BASE3, virtual public BASE4
42 {
43 public:
44     Derived() : BASE4(), BASE3(), BASE2(), BASE1(), y(), x()
45     {
46         cout << "Derived" << endl;
47     }
48     ~Derived() { cout << "~Derived" << endl; }
49 private:
50     Obj1 x;
51     Obj2 y;
52 };
53
```

```

54 int main()
55 {
56     Derived d;
57     cout << "Return to Operator System." << endl;
58     return 0;
59 }

```

用 GCC 编译器编译此程序时，编译器给出如下温馨提示：

```

-----Configuration: ConstructDestruct - Debug-----
Compiling source file(s)...
ConstructDestruct.cpp
ConstructDestruct.cpp: In constructor 'Derived::Derived()':
ConstructDestruct.cpp:49: warning: base 'BASE3' will be initialized after
ConstructDestruct.cpp:49: warning: base 'BASE2'
ConstructDestruct.cpp:55: warning: 'Derived::y' will be initialized after
ConstructDestruct.cpp:54: warning: 'Obj1 Derived::x'
Linking...
ConstructDestruct.exe - 0 error(s), 4 warning(s)

```

程序的运行结果：

```

BASE2
BASE4
BASE1
BASE3
Obj1
Obj2
Derived
Return to Operator System.
~Derived
~Obj2
~Obj1
~BASE3
~BASE1
~BASE4
~BASE2

```

13.6 小结

继承和多态性的内容丰富。在大型软件设计中，首先要从抽象到具体逐步弄清事物之间的隶属层次关系、并列关系；还要认真研究各种对象的属性和行为的特异之处，尽量避免数据冗余，尽量避免行为冲突。这些方面的内容有待读者在今后的实际应用中逐步积累经验。

C++ 的单继承已经提供了足够强大的功能，C++ 语言同时还支持多重继承。我们建议，在可能的情况下应尽量避免使用多重继承，因为使用多重继承增加了程序的复杂度，使程序的编制、维护变得相对困难。

练习 13

- (1) 设计一个平面直角坐标系的点 (Point) 类，派生出一个圆 (Circle) 类；再由

圆类派生一个圆柱体（Cylinder）类。要求圆类能输出其对象周长和面积、圆柱体类能输出其对象的表面积和体积。

(2) 编写一个程序完成如下任务。有如下动物及其叫声（输出相应的文字）：

猫 (Cat)	miaow ...
狗 (Dog)	wang ...
鸡 (Rooster)	crow ...
牛 (Cattle)	moo ...
马 (Horse)	neigh ...

- ① 在基类 Animal 中设计一个成员函数 void Sound();。
 - ② 编写一个“动物声音播放器”函数 void Speaker(const Animal &a); (非成员函数) 根据实参动物类型播放其叫声 (输出相应的文字)。
 - ③ 增加一个重载函数, 其函数原型为 void Speaker(const Animal *a); 并测试之。
 - ④ 增加原型为 void Speaker1(Animal a); 的函数是否可行?
就函数 Animal::Sound() 为非虚函数、一般的虚函数和纯虚函数分别进行讨论。

(3) 给定一个头文件 Vec.h, 其中有抽象类模板 VECTOR 的设计。还有插入运算符重载、抽取运算符重载的普通 C++ 函数 (既不是成员函数, 又非友元函数)。

```

1 // Vec.h
2 #ifndef VEC_H
3 #define VEC_H
4 #include <iostream>
5 using namespace std;
6
7 template <typename T> class VECTOR
8 {
9 public:
10    VECTOR(int size=0, const T *x=NULL)
11    {
12        num = (size>0) ? size : 0;
13        p = NULL;
14        if(num>0)
15        {
16            p = new T[num];
17            for(int i=0; i<num; i++)
18                p[i] = (x==NULL)? 0 : x[i];
19        }
20    }
21    VECTOR(const VECTOR &v)
22    {
23        num = v.num;
24        p = NULL;
25        if(num>0)
26        {
27            p = new T[num];
28            for(int i=0; i<num; i++)
29                p[i] = v.p[i];
30        }
31    }

```

```
32     virtual ~VECTOR()
33     {
34         if(p!=NULL) delete [] p;
35     }
36     VECTOR & operator=(const VECTOR &v)
37     {
38         if(num!=v.num)
39         {
40             if(p!=NULL) delete [] p;
41             p = new T[num=v.num];
42         }
43         for(int i=0; i<num; i++)
44             p[i] = v.p[i];
45         return *this;
46     }
47     T & operator[](int index)
48     {
49         return p[index];
50     }
51     int getsize() const
52     {
53         return num;
54     }
55     void resize(int size)
56     {
57         if(size<0 || size==num) return;
58         else if(size==0)
59         {
60             if(p!=NULL) delete [] p;
61             num = 0;
62             p = NULL;
63         }
64         else
65         {
66             T *temp = p;
67             p = new T[size];
68             for(int i=0; i<size; i++)
69                 p[i] = (i<num) ? temp[i] : 0;
70             num = size;
71             delete [] temp;
72         }
73     }
74     virtual void Output(ostream &out) const = 0;
75     virtual void Input(istream &in) = 0;
76 protected:
77     int num;
78     T *p;
79 };
80
81 template <typename T>
82 ostream & operator<<(ostream &out, const VECTOR<T> &v)
83 {
84     v.Output(out);
85     return out;
86 }
87
88 template <typename T>
89 istream & operator>>(istream &in, VECTOR<T> &v)
90 {
```

```

91     v.Input(in);
92     return in;
93 }
94 #endif

```

① 将类模板 VECTOR 作为基类，通过公共继承派生一个新的类模板 Vector（向量类模板）。要求使如下测试程序能正确运行，并输出给定的结果。

```

1 // testVector.cpp
2 #include "Vector.h"
3 #include <fstream>
4 using namespace std;
5
6 int testVector()
7 {
8     int a[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
9     double x[8];
10    for(int i=0; i<8; i++)
11        x[i] = sqrt(double(i));
12    Vector<int> vi1(10, a), vi2(5, a+5);
13    Vector<double> vd1(8, x), vd2(3, x);
14    cout << "原始数据: " << endl;
15    cout << "vi1=" << vi1 << "\nvi2=" << vi2
16        << "\nvd1=" << vd1 << "\nvd2=" << vd2 << endl;
17    cout << "调整维数到 5: " << endl;
18    vi1.resize(5);
19    vi2.resize(5);
20    vd1.resize(5);
21    vd2.resize(5);
22    cout << "vi1=" << vi1 << "\nvi2=" << vi2
23        << "\nvd1=" << vd1 << "\nvd2=" << vd2 << endl;
24    cout << "\n将数据写入文件 vector.txt 中..." << endl;
25    ofstream outfile("vector.txt");
26    outfile << vi1 << '\n'
27        << vi2                                // 故意不换行
28        << vd1 << '\n' << vd2 << endl;
29    outfile.close();
30    cout << "\n清除对象的数据（即调整维数到 0）" << endl;
31    vi1.resize(0);
32    vi2.resize(0);
33    vd1.resize(0);
34    vd2.resize(0);
35    cout << "vi1=" << vi1 << "\nvi2=" << vi2
36        << "\nvd1=" << vd1 << "\nvd2=" << vd2 << endl;
37    cout << "\n从文件 vector.txt 中读取的数据: " << endl;
38    ifstream infile("vector.txt");
39    infile >> vi1 >> vi2 >> vd1 >> vd2;
40    infile.close();
41    cout << "vi1=" << vi1 << "\nvi2=" << vi2
42        << "\nvd1=" << vd1 << "\nvd2=" << vd2 << endl;
43    cout << "\nvi1+vi2=" << vi1 + vi2
44        << "\nvd1+vd2=" << vd1 + vd2 << endl;
45    return 0;
46 }

```

调用函数 testVector 的输出结果：

```
原始数据:
vi1 = (10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
vi2 = (5, 4, 3, 2, 1)
vd1 = (0, 1, 1.41421, 1.73205, 2, 2.23607, 2.44949, 2.64575)
vd2 = (0, 1, 1.41421)

调整维数到 5：
vi1 = (10, 9, 8, 7, 6)
vi2 = (5, 4, 3, 2, 1)
vd1 = (0, 1, 1.41421, 1.73205, 2)
vd2 = (0, 1, 1.41421, 0, 0)
```

将数据写入文件 vector.txt 中 ...

清除对象的数据（即调整维数到 0）

```
vi1 = ( )
vi2 = ( )
vd1 = ( )
vd2 = ( )
```

从文件 vector.txt 中读取的数据：

```
vi1 = (10, 9, 8, 7, 6)
vi2 = (5, 4, 3, 2, 1)
vd1 = (0, 1, 1.41421, 1.73205, 2)
vd2 = (0, 1, 1.41421, 0, 0)

vi1 + vi2 = (15, 13, 11, 9, 7)
vd1 + vd2 = (0, 2, 2.82842, 1.73205, 2)
```

② 将类模板 VECTOR 作为基类，通过公共继承派生一个新的自定义字符串类 String。要求不使用字符 '\0' 串结束标志（即字符串的长度由数据成员 num 记录），并且使如下测试程序能正确运行。

```
1 // testString.cpp
2 #include "MyString.h"
3 #include <iostream>
4 using namespace std;
5
6 int testString()
7 {
8     String str1 = "Hello", str2 = str1, str3;
9         // 转换构造    拷贝构造    默认构造
10    cout << "原始数据 (双引号是另外添加的)：" << endl;
11    cout << "str1=" << str1
12        << "\nstr2=" << str2
13        << "\nstr3=" << str3 << "" << endl;
14    str3 = str2;                                // 赋值运算
15    str1 = "C++_program.";
16    str2 = str3 + ",_world!";                   // 拼接运算
17    cout << "str1=" << str1
18        << "\nstr2=" << str2
19        << "\nstr3=" << str3 << "" << endl;
20    cout << "\n将数据写入文件 string.txt 中" << endl;
21    ofstream outfile("string.txt");
22    outfile << str1 << '\n'
23        << str2 << '\n'
```

```

24         << str3 << endl;
25     outfile.close();
26     cout << "\n清除对象的数据(即调整长度到0)" << endl;
27     str1.resize(0);
28     str2.resize(0);
29     str3.resize(0);
30     cout << "str1=" << str1 << "\nstr2=" << str2
31     << "\nstr3=" << str3 << endl;
32     cout << "\n从文件string.txt中读取的数据:" << endl;
33     ifstream infile("string.txt");
34     infile >> str1 >> str2 >> str3;
35     infile.close();
36     cout << "str1=" << str1 << "\nstr2=" << str2
37     << "\nstr3=" << str3 << endl;
38     return 0;
39 }

```

主函数所在的源程序文件如下：

```

1 // main.cpp
2 int testVector(), testString();
3
4 int main()
5 {
6     testVector();
7     testString();
8     return 0;
9 }

```

调用函数 `testString` 的输出结果：

原始数据(双引号是另外添加的)：

```

str1 = "Hello"
str2 = "Hello"
str3 = ""
str1 = "C++_program."
str2 = "Hello,_world!"
str3 = "Hello"

```

将数据写入文件 `string.txt` 中

清除对象的数据(即调整长度到0)

```

str1 = ""
str2 = ""
str3 = ""

```

从文件 `string.txt` 中读取的数据：

```

str1 = "C++_program."
str2 = "Hello,_world!"
str3 = "Hello"

```

③ 简要回答如下问题。

- 抽象向量类模板 VECTOR 中的成员函数 Input 和 Output 为什么设计成虚函数？程序中何处利用了上述虚函数的特性？
- 重载方括号运算符时为何采用引用返回？

第 14 章 I/O 流

本章介绍标准输入输出流、文件输入输出流、字符串流及其相关函数。学习本章后，读者应该理解和掌握输入输出操作中的一些常用方法；理解文本文件、二进制文件的概念，掌握文件的定位、读、写等操作；理解字符串流的概念及其使用方法。

14.1 标准 I/O 流

关于 I/O 流的格式控制已经在面向过程程序设计部分做了介绍。本节先介绍操作系统命令处理程序中有关 I/O 重新定向的概念和用法，然后介绍 C++ 中有关 I/O 流类的一些常用成员函数。

14.1.1 操作系统关于标准 I/O 及其重新定向

一般而言，输入是数据从计算机外部设备流向计算机内存的操作；输出是数据从计算机内存流向外部设备的操作。从操作系统的角度上看，计算机的输入输出设备都被看作文件。操作系统规定的标准输入设备特指键盘、标准输出设备特指显示器。在操作系统的命令处理程序（例如 Windows 操作系统中的“命令提示符”）中可以用 I/O 重新定向操作符“<，>，>>”及管道操作符“|”使标准输入与标准输出重新定向到其他设备（文件）。关于操作系统命令处理程序中使用 I/O 重新定向及管道操作参见下面程序运行的各种方法及其效果。

到目前为止，本书所使用的 C++ 标准 I/O 流操作基本上都是对对象 `cin`、`cout` 进行的。此外，C++ 还有几个标准 I/O 流对象如表 14-1 所示。

表 14-1 C++ 标准 I/O 流对象

对象名	所属类	设备名	说明
<code>cin</code>	<code>istream</code>	键盘	标准输入。可重新定向，有缓冲
<code>cout</code>	<code>ostream</code>	显示器	标准输出。可重新定向，有缓冲
<code>cerr</code>	<code>ostream</code>	显示器	常用于错误信息的标准输出。不可重新定向，无缓冲
<code>clog</code>	<code>ostream</code>	显示器	常用于错误信息的标准输出。不可重新定向，有缓冲

上述 I/O 流类的对象在标准名字空间 `std` 中。程序中只要包含了头文件 `iostream`，编译系统会保证在使用这些对象之前构造它们，将它们分别与标准输入设备、标准输出设备相联系。

例 14.1 标准 I/O 流及其重新定向。

本例程序中使用了四个标准 I/O 流类的对象。将程序编译、连接后生成可执行文件。假定可执行文件 `Hello.exe` 在文件夹 `D:\CPP\Hello\Debug` 中，我们将讨论多种不同的方式来运行该可执行文件。

源代码 14.1 标准 I/O 流示例程序

```

1 // Hello.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     char name[20];
8     int age;
9
10    cout << "What's your name?" << endl; // 此处单引号可不用转义字符
11    cin.getline(name, 20);
12    cerr << "How old are you?" << endl;
13    cin >> age;
14    clog << "姓名: " << name << ", 年龄: "
15        << age << ". 你好!" << endl;
16    return 0;
17 }

```

(1) 运行方式之一（标准输入输出）

这是以前一直使用的 I/O 操作，即从键盘输入数据、在显示器上输出数据。程序运行中用下画线表示键盘输入部分，其中第一行均为启动可执行程序的操作系统命令，命令前的字符为操作系统的提示符（包括当前目录的路径）。运行结果如下。

```

D:\CPP\Hello\Debug>Hello
What's your name?
Zhang San
How old are you?
20
姓名: Zhang San, 年龄: 20。你好!

```

(2) 运行方式之二（标准输入重新定向）

将标准输入设备（键盘）改变成一个指定的文件（即从文件中读取信息）。当然这个文件必须事先编辑好。例如，设其文件名为 Input.txt，内容为

```

Zhang San
20

```

启动程序执行的命令及运行结果如下。

```

D:\CPP\Hello\Debug>Hello < Input.txt
What's your name?
How old are you?
姓名: Zhang San, 年龄: 20。你好!

```

从运行结果可见，程序中的输入操作 `cin.getline(name, 20)` 和 `cin>>age` 直接从指定的文件中获取数据，不再需要操作者从键盘输入（也不会接受用户的键盘输入）。即标准输入（键盘）被重新定向。操作系统使用“<”作为输入重新定向操作符。这种重新定向在本次操作系统命令结束后结束。

(3) 运行方式之三（标准输出重新定向）

将标准输出设备（显示器）改变成一个指定的文件（即输出到文件）。启动程序执行的命令及运行结果如下。

```
D:\CPP\Hello\Debug>Hello > Output.txt↓  
Zhang San↓  
How old are you?  
20↓  
姓名: Zhang San, 年龄: 20。你好!
```

从运行结果可见，程序中插入到标准输出流对象 cout 的操作并不联系到显示器，其输出结果在文件 Output.txt 中（若该文件已经存在则覆盖其内容，否则新建该文件）。以至于程序运行时，在输入姓名前的提示信息“What's your name?”写入到文件中而不在显示器上输出。而插入到错误信息标准输出流对象 cerr, clog 的操作并不能被重新定向，依然在显示器上输出。操作系统使用“>”作为输出重新定向操作符。这种重新定向在本次操作系统命令结束后结束。

(4) 运行方式之四（标准输出重新定向）

将标准输出设备（显示器）改变成一个指定的文件（即输出到文件）。启动程序执行的命令及运行结果如下。

```
D:\CPP\Hello\Debug>Hello >> Output.txt↓  
Zhang San↓  
How old are you?  
20↓  
姓名: Zhang San, 年龄: 20。你好!
```

这里使用的操作系统重新定向操作符为“>>”。它将输出的内容追加到指定文件的尾部。若指定的文件不存在则与“>”的效果相同。

(5) 运行方式之五（标准输入输出重新定向）

结合上面的方式之二与方式之三，或者方式之二与方式之四。启动程序执行的命令及运行结果如下。

```
D:\CPP\Hello\Debug>Hello < Input.txt > Output.txt↓  
How old are you?  
姓名: Zhang San, 年龄: 20。你好!
```

请查看文件 Output.txt 的内容（仅有一行）。

```
D:\CPP\Hello\Debug>Hello < Input.txt >> Output.txt↓  
How old are you?  
姓名: Zhang San, 年龄: 20。你好!
```

请再查看文件 Output.txt 的内容（有两行）。

(6) 运行方式之六（管道操作）

操作系统使用的管道操作符号为“|”。管道操作将操作符号左侧命令的标准输出结果作为输入，传递给操作符右边的命令程序（可执行程序）。这一操作常常借助于一个临时文件实施，命令执行完毕后操作系统自动将临时文件删除。启动程序执行的命令及运行结果如下。

```
D:\CPP\Hello\Debug>type Input.txt | Hello+
What's your name?
How old are you?
姓名: Zhang San, 年龄: 20。你好!
```

还可以

```
D:\CPP\Hello\Debug>type Input.txt | Hello > Output.txt+
How old are you?
姓名: Zhang San, 年龄: 20。你好!
```

【说明】 `type` 是 Windows 操作系统的命令之一，其功能是显示指定文件的内容。Unix/Linux 操作系统中的相应命令名称为 `cat`。

14.1.2 常用输入流成员函数

1. 成员函数 `get` 及 `getline`

流成员函数 `get` 有多种形式，下面列出其中三个。

```
char istream::get(); // 返回从输入流中获取的一个字符
istream & istream::get(char &c); // 从输入流中获取一个字符
istream & istream::get(char *str, int n, char delim='\'n''); // 从输入流中获取一个 C- 字符串
istream & istream::getline(char *str, int n, char delim='\'n''); // 从输入流中获取一个 C- 字符串
```

其中第三种形式的 `get` 函数与 `getline` 函数类似。其功能是从输入流中获取 `n-1` 个字符，或者在不足 `n-1` 个字符时遇到指定的终止字符 `delim`（终止字符默认值为换行符 '`\n`'）或者读取到“文件结束标志”、流结束符后，按 C-字符串的形式（自动添加串结束标志字符 '`\0`'）存入以 `str` 为起始地址的内存中。

值得指出的是：使用此成员函数输入的数据为 C-字符串，其中可含空格 ''、制表符 '`\t`' 和换行符 '`\n`' 等，当然在这些字符不用于终止字符时（请参考第 12.2.1 小节）。

一种值得注意的情形：当我们用抽取运算符“`>>`”抽取数据时，位于输入流缓冲区中数据之前的空白符（''、'`\t`'、'`\n`'）会被自动过滤掉，因为系统默认 `std::skipws`（参见第 3.5 节）。而数据后面的空白符仍然留在输入流缓冲区中。若仍有数据分隔符留在输入流缓冲区，且接下来的 `getline` 调用恰好用该分隔符为终止字符时，则 `getline` 调用的结果将仅读到一些空白字符组成的串。例如：

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

```

3 int main()
4 {
5     int n;
6     char str1[80], str2[80];
7
8     cin >> n;
9     cin.getline(str1, 80);      // 此语句读取换行符后丢弃
10    // cin.sync();
11    for(int i=0; i<n; i++)
12    {
13        cin.getline(str1, 80);
14        cin.getline(str2, 80);
15        cout << str1 << "\n" << str2 << "\n" << endl;
16    }
17    return 0;
18 }
```

程序第 10 行是需要的，或者参考练习 5 的 3(2) 忽略其后的空白字符及换行字符。请读者用标准 I/O 及 I/O 重新定向的方式测试该程序。

2. 抽取数值不当的处理

`cin` 是 `istream` 类的对象，程序中的变量可以通过抽取运算符“`>>`”从输入流中抽取到各种类型的数据。用抽取运算符从输入流中抽取数据时，输入流中通常以一个或者多个空格 ('')、制表符 ('`\t`') 和换行符 ('`\n`') 等作为数据项的分隔符。当抽取到无效数据（例如：当需要一个整型数值或浮点型数值时，却抽取到一个字母或一个非数码字符串）或者遇到文件结束标志^[1]时，输入流 `cin` 就处于出错状态，`cin` 的值为 `NULL`。此时将无法继续正常抽取其他数据。需要清除出错状态（即设置成正确的状态）。

首先运行如下程序，观察抽取数值不当的情形，然后给出处理方法。

```

1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int n;
6     while(1)
7     {
8         cout << "请输入一个整数(0--退出):";
9         cin >> n;
10        if(n==0) break;
11        cout << "n=" << n << endl;
12    }
13    return 0;
14 }
```

若输入一个合法的数据（整数），则程序执行正常。若输入不当的数据，如输入字母、非数码字符串，则程序陷入无限循环，无法再正常地接收输入。

例 14.2 清理输入流的出错状态。下面的函数（写成函数模板）能即时清理输入流的出错状态，恢复到正常的输入状态。可以以容错的方式接收数值型（整型、浮点型）数据的输入。

[1] 从标准输入设备（即键盘）输入文件结束标志的方法：在 Windows 操作系统中，按 `Ctrl+Z` 组合键，记为 `^Z`；在 Unix/Linux 操作系统中，按 `Ctrl+D` 组合键，记为 `^D`。

源代码 14.2 输入数值不当的处理方法

```

1 // Input.cpp
2 #include <iostream>
3 using namespace std;
4
5 template <typename T> T &input(T &x)
6 {
7     while((cin >> x)==NULL)
8     {
9         cin.clear();           // 清理错误状态位
10        cin.sync();          // 刷新输入流缓冲区
11    }
12    return x;
13 }
14
15 int main()
16 {
17     int n;
18     while(1)
19     {
20         cout << "请输入一个整数(0--退出):";
21         input(n);           // 替换了语句 cin >> n;
22         if(n==0) break;
23         cout << "n=" << n << endl;
24     }
25     double x;
26     while(1)
27     {
28         cout << "请输入一个实数(0--退出):";
29         input(x);           // 替换了语句 cin >> x;
30         if(x==0) break;
31         cout << "x=" << x << endl;
32     }
33     return 0;
34 }
```

程序具有容错能力，程序运行时可以任意输入不合法的数据（如字母、非数码字符串）。若以有效数字打头输入时将抽取其中的有效数字，其后的不合法字符串将留在输入流缓冲区中。程序中的 `clear` 及 `sync` 是 `istream` 类的成员函数。

14.1.3 常用输出流成员函数

除了用插入运算符进行输出外，还可以用成员函数 `put` 输出一个字符。

```
ostream & ostream::put(char c); // 输出一个字符
```

因为 `cout` 为缓冲输出，当缓冲区未满时，位于缓冲区中的信息不一定立即在显示器上呈现。通过刷新缓冲区操作可使缓冲区中的信息立即在显示器上呈现出来。

```
cout << flush;           // 刷新输出流缓冲区
cout << '\n' << flush; // 输出换行符并刷新。等价于cout << endl;
```

14.2 文件 I/O 流

本节所介绍的文件是指存储在外部存储器（如磁盘、光盘或 U 盘）上的文件。C++ 程序可以对这些文件的内容进行操作（假定这些文件是可读或可写的）。文件 I/O 操作大致包括以下几个基本步骤：

(1) 根据对文件将要进行的操作，创建适当的文件流类的对象。

① 只读取文件内容。需创建一个输入文件流 `ifstream` 类 (`input-file-stream`) 的对象。

② 只将数据写入文件。需创建一个输出文件流 `ofstream` 类 (`output-file-stream`) 的对象。

③ 读/写文件内容。需创建一个输入/输出文件流 `fstream` 类 (`file-stream`) 的对象。

(2) 将创建的文件流对象与指定的文件进行关联（打开文件），需指定文件名，指定打开模式打开文件。

(3) 在文件内容中定位，读取文件内容或者将数据写入文件。

(4) 解除文件与文件流对象的关联（关闭文件）。

【说明】

① 利用文件流类的构造函数可将上述步骤 (1) 与 (2) 合并。

② 读/写文件内容后，文件内容的当前位置自动顺序地移动，也可以根据需要任意定位当前位置。

③ 关闭文件后，若文件流对象的生命期未终止，则该对象还可以与其他文件建立关联。

在 C++ 的 I/O 流类库中定义了几种文件流类：`ifstream`、`ofstream` 和 `fstream`。使用它们时需要编译预处理指令 `#include <fstream>` 包含头文件。这几个类分别有多个构造函数和一些成员函数。创建文件流对象的常用方法如下。

使用默认的构造函数创建对象，然后调用成员函数 `open` 关联到具体的文件如下。

```
文件流类名 对象名;  
对象名.open(文件名, 打开模式);
```

或使用带参数的构造函数，在创建文件流类对象的同时关联具体的文件，使用默认的打开模式如下。

```
文件流类名 对象名(文件名);
```

最常用的是在创建文件流对象的同时关联具体文件并且指定打开模式，如下。

```
文件流类名 对象名(文件名, 打开模式);
```

打开文件模式是在 `ios` 类中定义的，它们是一些枚举常量，有多种选择，如表 14-2 所示。

表 14-2 打开文件模式

模式	作用
<code>ios::in</code>	以输入方式打开文件
<code>ios::out</code>	以输出方式打开文件。若文件已经存在，则先清除其所有内容
<code>ios::app</code>	以输出方式打开文件。将写入的数据追加到原来数据的末尾
<code>ios::ate</code>	打开已存在的文件，文件内容指针指向文件末尾
<code>ios::trunc</code>	打开文件。若文件已经存在，则先清除其所有内容
<code>ios::binary</code>	以二进制方式打开文件

这些模式可以用二进制位或运算符“|”进行组合。例如：

```
ios::in | ios::out | ios::binary // 以二进制、可读、可写模式打开文件
```

如果打开文件成功，则文件流对象的值为非 0 值 (`true`)；如果打开文件失败，则返回 0 (`false`)。

解除文件流类对象与文件的关联应使用成员函数 `close`：

```
对象名.close();
```

14.2.1 文本文件

从不同的角度看文件，文件有多种不同的分类方式（如：系统文件/用户文件；程序文件/数据文件；图像文件/声音文件；…）。本节讨论数据文件，并根据文件中数据的存放格式将文件区分为文本文件和二进制文件。

从构成文件的各个字节看，若一个文件内容的各字节仅由可打印的 ASCII 字符（参见第 1.1.3 小节）、回车字符（'\r'）、换行字符（'\n'）和制表字符（'\t'）组成。则该文件便是一个文本文件。在 Windows 操作系统中，文本文件以回车、换行两个字符标注一行结束，有时还用字符 '\x1A' ((1A)₁₆ = 26) 作为文本文件的结束标志。在 Unix/Linux 操作系统中，文本文件仅以换行一个字符标注一行结束。常称文本文件为 ASCII 文件。

若将一个非文本文件当作文本文件打开，则可能遇到其中的 '\x1A' 字节即认为文件结束，可能导致其后的数据无法读取（Windows 操作系统中）。

C++ 源程序文件、头文件、Windows 批处理文件和 Linux 脚本文件等都是文本文件。文本文件可用文本编辑软件进行编辑。Windows 操作系统提供的“记事本”程序、C++ 集成开发环境中的编辑器均能新建、编辑文本文件。采用文本文件格式存储数据比较直观。可以用文本文件编辑器预先编辑好数据并存入磁盘文件中，在程序运行时直接读取其中的数据，特别适合有大量原始数据输入给程序的情形（这里不是指采用重新定向的方法）。

C++ 程序中，对文本文件内容进行操作的函数、运算符主要有插入运算符（<<）、抽取运算符（>>）、`put` 函数、`get` 函数和 `getline` 函数等。一般采用顺序方式读或写文件内容。文件中的数据之间应该添加特定的数据项分隔字符（如制表符、换行符、逗号或

双引号等），否则将难以正确地读取文件内容。当然，要求所选用的特定分隔字符不会出现在数据本身之中。

例 14.3 新建一个文本文件，将一些不同数据类型的数据写入其中，然后再从文件中将数据读取出来。如处理一本书的书名、作者、出版社（字符串）、页数（整数）和定价（浮点数）。

【分析】 字符串中可能出现空格字符。为了能正确地读取文件中的数据，必须在将数据写入文件时选定一种特殊的字符用于分隔各数据项。本例采用制表符 '\t' 作为数据之间的分隔字符。

源代码 14.3 文本文件的读写

```
1 // TextFile.cpp
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int SaveA(char *filename)      // 将数据写入文本文件
7 {
8     char book[80] = "Jane_Eyre",
9         author[20] = "Charlotte_Bronte",
10        press[20] = "*_*_*";
11    int pages = 300;
12    double price = 28.5;
13    ofstream outfile; // 创建输出文件流对象并关联指定文件
14    outfile << book << author << press
15    //       << pages << price << endl; // 输出数据项之间无分隔字符
16    outfile << book << '\t' << author << '\t' // 插入特定字符
17    //       << press << '\t' << pages << '\t' // 便于读取
18    //       << price << endl;
19    outfile.close();           // 关闭文件
20    return 0;
21 }
22
23 int LoadA(char *filename)
24 {
25     char book[80], author[20], press[20];
26     int pages;
27     double price;
28     ifstream infile; // 创建输入文件流对象并关联指定文件
29     infile.getline(book, 80, '\t');           // 读取文件内容
30     infile.getline(author, 20, '\t');
31     infile.getline(press, 20, '\t');
32     infile >> pages >> price;             // 读取文件内容
33     cout << book << ", " << author << ", " // 标准输出
34     //       << press << ", " << pages << ", "
35     //       << price << endl;
36     infile.close();           // 关闭文件
37     return 0;
38 }
39 int main()
40 {
41     SaveA("TextFile.txt");
42     LoadA("TextFile.txt");
43     return 0;
44 }
```

【说明】 若按照程序中第 15 及 16 行语句输出，由于输出的各数据项之间无分隔符，将导致无法正确地读取数据，故是不可取的。本程序运行后将产生一个名为 `TextFile.txt` 的文本文件，该文件可用“记事本”或任何文本文件编辑软件打开。

对于自定义的类类型只要按第 12 章重载了插入、抽取运算符，便可以直接用于文本文件的读写操作（参见第 12 章关于“评委评分”程序）。

14.2.2 二进制文件

从构成文件的各个字节看，任何文件都可以看成二进制文件，都可以按照二进制模式打开。准确地说，若数据按其在计算机内存中的存放格式存储在文件中，则称这种文件为**二进制文件**。二进制文件不便用文本文件的编辑软件打开。一般而言，二进制文件需要用特定的程序打开（例如，图像文件、声音文件等多媒体文件需要用专门的查看程序、播放程序打开）。

打开二进制文件后，可以根据需要定位文件内容的当前位置、读取文件内容或者将数据写入文件中。相关类的成员函数原型及功能如下。

```
istream & read(char *buffer, int len);
    // 读取文件中的 len 字节内容存放到以 buffer 为起始的地址处
ostream & write(const char *buffer, int len);
    // 将 buffer 为首地址的 len 字节内容写入到文件中
```

由于数据在内存中的存放格式与文件中的存放格式相同，因此只要将首地址、字节数传递给函数，便能够“原封不动”地由计算机内存到文件或由文件到计算机内存之间进行复制。因此上述两个函数的形式参数仅为 `char*` 型，其他数据类型指针需要强制转换。

文件打开后，自动建立一个指向文件内容当前位置的指针，对文件的操作将在该指针所指向的位置上进行。一般情况下，打开文件后指针指向文件头（与文件第一个字节的偏移量为 0）或指向文件尾（`ios::ate` 或 `ios::app` 模式下与文件的最后一个字节的偏移量为 0）。移动、获取文件当前位置内容的成员函数的原型及功能如下。

① 关于输入文件流对象（成员函数名中的字母 g 取自 getting）

```
istream & seekg(long);    // 将输入文件内容指针移至指定位置
istream & seekg(long, ios::seek_dir);
    // 根据 seek_dir 指定的参照位置，相对移动指定字节数，定位文件内容指针
long tellg();            // 返回输入文件内容指针的当前位置
int gcount();            // 最后一次所读入的字节数
```

② 关于输出文件流对象（成员函数名中的字母 p 取自 putting）

```
ostream & seekp(long);    // 将输出文件内容指针移至指定位置
ostream & seekp(long, ios::seek_dir);
    // 根据 seek_dir 指定的参照位置，相对移动指定字节数，定位文件内容指针
long tellp();            // 返回输出文件内容指针的当前位置
```

其中作为参照位置的量 `seek_dir` 取下面的三者之一。

<code>ios::beg</code>	文件起始处 (<code>begin</code> 的缩写)，这是默认值
<code>ios::cur</code>	文件内容指针的当前位置为参考点 (<code>current</code> 的缩写)
<code>ios::end</code>	文件末尾处

下面的例子将不同类型的数据分别按文本文件格式、二进制文件格式写入不同的文件中。并分析文件各字节的内容。

例 14.4 文本文件及二进制文件的读写。

源代码 14.4 文本文件及二进制文件的读写

```
1 // TextAndBinaryFile.cpp
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int SaveA(char *filename)      // 按文本文件格式写入指定文件
7 {
8     int n = 538610154;
9     double x = 3.1415926;
10    ofstream outfile(filename); // 创建输出文件流对象并关联指定文件
11    outfile.precision(8);
12    outfile << n << " " << x << endl;
13    outfile.close();
14    return 0;
15 }
16
17 int SaveB(char *filename)      // 按二进制文件格式写入指定文件
18 {
19     int n = 538610154;
20     double x = 3.1415926;
21     ofstream outfile(filename, ios::binary);
22         // 创建输出文件流对象并关联指定的二进制文件
23     outfile.write((char*)&n, sizeof(int));
24     outfile.write((char*)&x, sizeof(double));
25     outfile.close();
26     return 0;
27 }
28
29 int LoadA(char *filename)      // 从指定的文本文件中读取数据
30 {
31     int n=0;
32     double x=0;
33     ifstream infile(filename); // 创建输入文件流对象并关联指定文件
34     infile >> n >> x;
35     infile.close();
36     cout.precision(8);
37     cout << n << ", " << x << endl;
38     return 0;
39 }
40
41 int LoadB(char *filename)      // 从指定的二进制文件中读取数据
42 {
43     int n=0;
44     double x=0;
45     ifstream infile(filename, ios::binary);
46         // 创建输入文件流对象并关联指定的二进制文件
47     infile.read((char*)&n, sizeof(int));
48     infile.read((char*)&x, sizeof(double));
49     infile.close();
50     cout.precision(8);
51     cout << x << ", " << n << endl;
52     return 0;
53 }
```

```

54 int LoadAB(char *filename)      // 反例。将二进制文件按文本文件读取
55 {
56     char c;
57     int i=0;                      // 记录从文件中读出的字符个数
58     ifstream infile(filename);
59     while(infile.get(c))
60         cout << ++i << '\t' << (int)c
61         << "\t\" << c << "\n";
62     infile.close();
63     cout << endl;
64     return 0;
65 }
66
67
68 int main()
69 {
70     SaveA("Test.txt");           // 按文本文件格式写入
71     SaveB("Test.dat");           // 按二进制文件格式写入
72     LoadA("Test.txt");          // 按文本文件格式读取
73     LoadB("Test.dat");          // 按二进制文件格式读取
74     LoadAB("Test.dat");         // 将二进制文件按文本文件打开
75
76 }

```

运行程序，在显示器上输出的结果为

```

538605802, 3.1415926
3.1415926, 538605802
1      -22      '?'
2      120      'x'

```

此外，还产生了如下两个文件。

① `test.txt` 文件长度 21 字节，其十六进制值依次为（可借助第 14.4 节的程序获取如下信息）

```
35 33 38 36 30 35 38 30 32 20 33 2E 31 34 31 35 39 32 36 0D 0A
```

其中 '`\x35`' 为字符 '5'（ASCII 编码为 $(35)_{16} = (53)_{10}$ ，其他数码字符类推），'`\x20`' 为空格字符 ' '，'`\x2E`' 为圆点字符 '.'，'`\x0D`' 为回车字符 '\r'，'`\x0A`' 为换行字符 '\n'。

② `test.dat` 文件长度 12 字节，即 `sizeof(int) + sizeof(double)` 与数据在内存中占用的字节总数相等、存放格式相同。其 12 字节的十六进制依次为

```

EA 78 1A 20 (int 型数据的低位在前高位在后 (201A78EA)16 = (538605802)10)
4A D8 12 4D FB 21 09 40
(double 型数据 3.1415926 在内存中的具体存放形式)

```

可以看到，上述文件中的第三个字节恰好为 '`\x1A`'，因此将该文件按文本文件模式打开时，只能读取到前两个字节，这就是运行结果中最后只有两行的原因。其中十六进制数 '`\xEA`' 对应的 ASCII 字符是一个扩展 ASCII 字符，在中文操作系统下无法正常显示，该数表示成带符号的十进制整数为 -22（因为 $(EA)_{16} = 234$, $256 - 234 = 22$ ）。另一个十六进制表示的数 $(78)_{16} = 120$ ，为可打印字符 'x' 的 ASCII 码。

14.2.3 应用举例

例 14.5 变换文件内容。将文件内容的每个字节分别施行某种变换，相当于对文件内容进行一种加密。为简单起见，选用一种简单的处理方式：将文件的每个字节（8 位）的高 4 位与低 4 位交换。显然，这种变换的逆变换与变换相同，即两次变换为解密还原。用三种方式分别实现。

- ① 函数 ChangeFile1 将文件按可读、可写、二进制模式打开，边读边写。
- ② 函数 ChangeFile2 打开一个输入文件和一个输出文件，从输入文件读取数据，经处理后写入输出文件：。
- ③ 将输入文件的内容全部读入内存，经处理后写入文件（函数 ChangeFile3）。

源代码 14.5 变换文件内容

```

1 // ChangeFile.cpp
2 #include <iostream>
3 #include <fstream>
4 #include <cstring>
5 using namespace std;
6
7 int ChangeFile1(char *filename)
8 {
9     fstream file(filename, ios::in | ios::out | ios::binary);
10    if(file.fail())
11        return -1;      // 表示打开文件时出错
12
13    unsigned char c;
14    while(file.read((char*)&c, sizeof(char)))
15    {
16        c = (c & 0xF) << 4 | (c >> 4);      // 字符 c 的高、低 4 位互换
17        file.seekp(-1, ios::cur);            // 文件内容指针后退 1 字节
18        file.write((char*)&c, sizeof(char)); // 将变换结果写回文件
19    }
20    file.close();                      // 表示正常返回
21    return 0;                          // 表示正常返回
22 }
23
24 int ChangeFile2(char *filename1, char *filename2)
25 {
26     // 两个文件名不能相同，否则文件内容将被清除
27     ifstream infile(filename1, ios::binary);
28     if(infile.fail())
29         return -1;      // 表示打开文件时出错
30
31     ofstream outfile(filename2, ios::binary);
32     if(outfile.fail())
33         return -2;      // 表示创建文件时出错
34
35     unsigned char c;
36     while(infile.read((char*)&c, sizeof(char)))
37     {
38         c = (c & 0xF) << 4 | (c >> 4);
39         outfile.write((char*)&c, sizeof(char));
40     }
41     infile.close();
42     outfile.close();
43 }
```

```

43     return 0;           // 表示正常返回
44 }
45
46 int main(int argc, char *argv[])
47 {
48     int flag;
49
50     if(argc==1)
51     {
52         cout << "命令格式: " << argv[0]
53         << "<filename1> [<filename2>]" << endl;
54         return 1;
55     }
56     else if(argc==2 || argc>2 && strcmp(argv[1], argv[2])==0)
57         flag = ChangeFile1(argv[1]);
58     else
59         flag = ChangeFile2(argv[1], argv[2]);
60
61     switch(flag)
62     {
63     case -1: cout << "不能打开文件" << argv[1] << endl; break;
64     case -2: cout << "不能创建文件" << argv[2] << endl; break;
65     case 0: cout << "文件变换成功。" << endl; break;
66     }
67     return 0;
68 }
```

编译连接该程序生成可执行文件后，执行该程序的命令格式为（在操作系统的命令提示符下）

ChangeFile <文件名1> [文件名2]

其中命令格式的描述中采用了一般的通用记号：尖括号“<>”中的内容是必须提供的命令参数；方括号“[]”中的内容是可选的。括号本身只是记号不是命令中的一部分，不要输入。若文件名中包含空格字符，则需将整个文件名用双引号包围起来。关于命令行参数的内容参见第 5.8.1 小节。

第三种实现方法的函数 ChangeFile3 的特点是读全后再写。将该函数替换上面程序中的函数 ChangeFile2。此时，函数的两个实参文件名可以相同，但对于长度超长的文件可能由于分配内存出错而失败。下面的代码段中还给出了计算并返回文件长度的函数，并在函数 ChangeFile3 中调用以确定需要分配的堆空间的尺寸。函数定义如下。

```

1 int FileSize(ifstream &infile)
2 {
3     int cur_pos = infile.tellg();    // 获取文件内容当前位置
4     infile.seekg(0, ios::end);      // 位置指针移至文尾
5     int size = infile.tellg();      // 获取文件当前位置，即文件长度
6     infile.seekg(cur_pos, ios::beg); // 恢复文件当前位置
7     return size;                  // 返回文件长度（字节数）
8 }
9
10 int ChangeFile3(char *filename1, char *filename2)
11 {
12     ifstream infile(filename1, ios::binary);
13     if(infile.fail())
14         return -1;                // 表示打开文件时出错

```

```

15 int n = FileSize(infile);
16 if(n==0) return 0;
17 unsigned char *buf = new unsigned char[n];
18 if(buf==NULL)
19     return -3; // 表示没有足够的内存空间
20 infile.read((char*)buf, n); // 将文件内容全部读入内存
21 infile.close(); // 关闭输入文件
22
23 for(int i=0; i<n; i++) // 处理内存中的数据
24     buf[i] = (buf[i] & 0xF)<<4 | (buf[i]>>4);
25
26 ofstream outfile(filename2, ios::binary);
27 // filename2 可与 filename1 相同
28 if(outfile.fail())
29 {
30     delete [] buf; // 表示创建文件时出错
31     return -2;
32 }
33 outfile.write((char*)buf, n); // 将内存中的数据写入文件
34 outfile.close();
35 delete [] buf; // 释放堆内存资源
36 return 0; // 表示正常返回
37 }

```

14.3 字符串 I/O 流

14.3.1 C 语言中的字符串生成与解析

C 语言中的函数 `sprintf` 和函数 `sscanf` 的功能与字符串流有类似之处。先看一例：

```

1 // CStringIO.c                                C 语言程序
2 #include <stdio.h>
3
4 int main()
5 {
6     char str[1024];
7     int n = 100;
8     char c = 'A', s[20] = "HelloWorld.";
9     double x = 3.1415926;
10
11    sprintf(str, "%d,%c,%lf,%s\n",
12             n, c, x, s); // 根据多种数据生成文本字符串 str
13    x = n = s[1] = 0;
14    c = 'B'; // 改变数据的值不影响已经生成的 str
15
16    printf("%s", str); // 输出所生成的字符串 str
17    printf("%d,%c,%lf,%s\n", n, c, x, s); // 输出其他数据的值
18
19    sscanf(str, "%d,%c,%lf,%s\n",
20            &n, &c, &x, s); // 从字符串 str 中读取各种类型数据
21
22    printf("%d,%c,%lf,%s\n", n, c, x, s);
23
24    return 0;
25 }

```

程序的运行结果：

```
n = 100, c = A, x = 3.141593, s = Hello world.
0, B, 0.000000, H
100, A, 3.141593, Hello
```

【说明】 这个程序将字符数组 str 视为输入输出的“介质”，类似于标准输入和标准输出。运行结果的第一行是字符数组 str 的内容，它是各种数据写入到其中的结果（生成一个 C-字符串）。运行结果的第二行表明，这些不同类型的数据值的确被故意改变了（第三行的结果表明这些数据的改变不影响已经生成的字符串）。运行结果的第三行表明可以从字符串中读出各种类型的数据。第 19、20 行从字符串 str 中读取字符串到 s 中时，将空格作为数据项的分隔符，故仅能读取“Hello”，其后面的字符未被读取。

这里的字符数组 str 就宛如一个标准 I/O 设备，也好像一个文本文件。C 语言程序中常用这种方法生成字符串，或从字符串中解析出各种不同数据类型的数据。

14.3.2 C++ 字符串流类

C++ 提供字符串流 (string streams) 类，以其对象的组合成员 (string 类的对象) 视为读写数据的“介质”，数据均在计算机内存中流动并进行必要的 ASCII 格式与二进制格式转换。C++ 的字符串流类 **istringstream**、**ostringstream** 和 **stringstream** 分别是类 **istream**、**ostream** 和 **iostream** 的派生类。因而，对其对象的基本操作方法与标准 I/O 基本上相同。在这个意义上，我们也分别称为输入字符串流类，输出字符串流类，输入输出字符串流类。

使用字符串流类需包含头文件 **<sstream>**。

类 **istringstream**、**ostringstream** 和 **stringstream** 的对象都分别联系到其一个内部的组合成员 **string** 对象，该字符串成员可通过串流类的成员函数 **str** 访问。**istringstream** 类的对象在创建时可用 **string** 类的对象或 C-字符串（字符数组的首地址）做实参实现初始化。**ostringstream** 类的对象在创建时可采用其默认的初始化（使用其默认构造函数）而不必考虑其内部的 **string** 成员对象的尺寸，因为 **string** 类的对象能够自动扩展。

【说明】 作为存放数据的“介质”，字符串流类对象组合的内部字符串也类似标准 I/O 或文本文件，需要由程序员设定数据之间的分隔符、结束标志。

练习 5 的 3(1) 程序便是字符串流类的应用示例。下面再举一例。

例 14.6 C++ 字符串流类示例

源代码 14.6 C++ 字符串流类示例

```
1 #include <iostream>
2 #include <sstream>
3 #include <string>
4 using namespace std;
5
6 int main()
7 {
8     int n = 100, m;
9     double x = 3.1415926535897932, y;
```

```

10    ostringstream oss;
11    oss.precision(8); oss << fixed;
12    oss << "n=" << n << ",x=" << x;
13    cout << oss.str() << endl; // 输出所构造的字符串
14    istringstream iss(oss.str()); // 用所构造的字符串创建新对象
15    cout << iss.str() << endl; // 输出新对象的内部字符串
16    string str;
17    getline(iss, str, '='); // 从 iss 读字符串至遇到第一个 '='
18    cout << "|" << str << "|" << endl; // 标准输出
19    iss >> m;
20    iss.ignore(100, '='); // 直接忽略至 '='
21    iss >> y;
22    cout << "m=" << m // 标准输出
23    << ",y=" << y << endl;
24    return 0;
25 }

```

程序的运行结果：

```

n = 100, x = 3.14159265
n = 100, x = 3.14159265
|n |
m = 100, y = 3.14159

```

14.4 趣味程序——探究文件字节内容

将指定的文件按二进制模式打开，仿照 Windows 操作系统 `debug.exe` 的查看格式，以字节为单位按十六进制数及对应的字符依次输出文件的内容。利用这个程序可以探究文件的字节内容，进一步理解文本文件和二进制文件。

程序采用面向对象设计方法，设计了一个类 `Brow`。该类带内存堆资源，因而需要深拷贝构造函数、深赋值运算符函数和析构函数。在设计类 `Brow` 时，还提供了默认的构造函数和转换构造函数。使用类 `Brow` 时，可以先创建对象，然后使用成员函数 `Open` 打开指定的文件；也可以在创建对象时打开指定的文件（这一点是模仿创建文件流对象的做法。请读者仔细体会）。此外，还重载了类型转换运算符，可以通过判断对象的值了解对象的状态（这也是模仿 I/O 流类的做法^[1]。请读者体会）。

源代码 14.7 探究文件字节内容

```

1 // Brow.h
2 #ifndef BROW_H
3 #define BROW_H
4 #include <fstream>
5 #include <string>
6 using namespace std;
7
8 class Brow
9 {
10 public: // 公开的成员

```

^[1]例如，标准 I/O 流可以使用 `if(cin>x)`，或 `if(cin.getline(str, 80))`。文件流对象 `if(infile.fail())` 可以换成 `if(!infile)`。

```

11     Brow(char *filename=NULL);           // 构造函数
12     Brow(const Brow &b);             // 深复制构造函数
13     Brow & operator=(const Brow &b); // 深赋值运算符函数
14     operator int();                  // 类型转换运算符函数
15     virtual ~Brow();                // 析构函数
16     void Open(char *filename);        // 打开文件
17     void Show();                   // 输出函数
18 protected:                         // 内部成员函数 (受保护的)
19     int GetKey();                  // 接收键盘输入, 可为特殊键
20     int isPrint(unsigned char c);   // 判断是否为可打印字符
21     int FileSize(ifstream &infile); // 返回文件长度 (字节数)
22     void ShowPage();              // 显示一页
23 private:                           // 私有成员
24     string fname;                 // 文件名
25     unsigned char *buf;           // 内存堆资源首地址
26     int size;                    // 文件长度
27     int page, row, col;          // 内容当前位置
28 };
29 #endif

```

```

1 // Brow.cpp
2 #include <iostream>
3 #include <iomanip>
4 #include <sstream>
5 #include "Brow.h"
6 #include <conio.h>
7 using namespace std;
8
9 Brow::Brow(char *filename)
10 {
11     buf = NULL; size = 0;
12     if(filename==NULL)
13         fname = "";
14     else
15         Open(filename);
16 }
17 Brow::Brow(const Brow &b)
18 {
19     buf = NULL;      fname = b.fname;
20     size = b.size;   page = b.page;
21     row = b.row;    col = b.col;
22     if(size > 0)
23     {
24         buf = new unsigned char [size];
25         for(int i=0; i<size; i++)
26             buf[i] = b.buf[i];
27     }
28 }
29 Brow & Brow::operator=(const Brow &b)
30 {
31     if(&b == this) return *this;
32     if(buf!=NULL) delete [] buf;
33     buf = NULL;      fname = b.fname;
34     size = b.size;   page = b.page;
35     row = b.row;    col = b.col;
36     if(size > 0)
37     {
38         buf = new unsigned char [size];

```

```
39         for(int i=0; i<size; i++)
40             buf[i] = b.buf[i];
41     }
42     return *this;
43 }
44 Brow::operator int()
45 {
46     return size;                                // size 为 0 表示打开文件出错
47 }
48 Brow::~Brow()
49 {
50     if(buf!=NULL) delete [] buf;
51 }
52 void Brow::Open(char *filename)
53 {
54     if(buf!=NULL) delete [] buf;                // 释放原来可能有的资源
55     size = page = row = col = 0;
56     buf = NULL;
57     ifstream infile(filename, ios::binary);
58     if(infile.fail()) return;
59     size = FileSize(infile);
60     buf = new unsigned char[size];
61     infile.read((char*)buf, size);
62     infile.close();
63     fname = filename;
64 }
65 int Brow::GetKey()
66 {
67     int key = getch();
68     if(key==224 || key==0)
69         key = getch();
70     return key;
71 }
72 int Brow::isPrint(unsigned char c)           // 判断是否为可打印字符
73 {
74     c &= 0x7F;
75     return c>=' ' && c<0x7F;
76 }
77 int Brow::FileSize(ifstream &infile)          // 计算文件长度(字节数)
78 {
79     int cur_pos = infile.tellg();
80     infile.seekg(0, ios::end);
81     int size = infile.tellg();
82     infile.seekg(cur_pos, ios::beg);
83     return size;
84 }
85 void Brow::ShowPage()
86 {
87     system("cls");                            // 清屏。Windows 系统适用
88     char cstr[17];
89     unsigned char *p = buf + page * 0x100;
90     int i, j, len, m0=0x10*row + col;
91     len = (size-page*0x100 > 0x100) ? 0x100 : size-page*0x100;
92     cout << setfill('0') << hex << uppercase;
93     for(j=0; j<0x10 && m<len; j++)
94     {
95         cout << setw(8) << page*0x100 + j*0x10 << " " ;
96         ostringstream outstr;
97         outstr << hex << uppercase << setfill('0');
```

```

98         for(i=0; i<0x10 && m<len; i++,m++)
99         {
100             outstr << setw(2) << (unsigned short)p[m];
101             if(m == m0)
102                 outstr << "*";
103             else
104                 outstr << (i==7 ? '-' : '_');
105
106             cstr[i] = (isPrint(p[m]))? p[m] : '.';
107         }
108         for( ; i<0x10; i++)
109         {
110             outstr << "____";
111             cstr[i] = '\0';
112         }
113         cstr[16] = '\0';
114         cout << outstr.str() << ' ' << cstr << endl;
115     }
116     m = page*0x100 + row*0x10 + col;
117     cout << "\nFile:_" << fname
118         << "\t(0x" << m << "+1)/0x" << size << "_=_"
119         << dec << "(" << m << "+1)/" << size << "_=_"
120         << 100.0*(m+1)/size << "%" << endl;
121     cout << "\n<ESC>_<Home>_<End>_<PgUp>_<PgDn>_"
122         << "<Up>_<Down>_<Left>_<Right>" << endl;
123 }
124 void Brow::Show()
125 {
126     int m, key=0;
127     if(size==0)
128     {
129         cout << "\nFile:_" << fname << "_size:_0 byte." << endl;
130         return;
131     }
132     do
133     {
134         m = 0x100 * page;
135         ShowPage();
136         key = GetKey();
137         switch(key)
138         {
139             case 72: row--; break; // Up
140             case 75: col--; break; // Left
141             case 77: col++; break; // Right
142             case 80: row++; break; // Down
143             case 71: page = -1; break; // Home
144             case 79: page = size; break; // End
145             case 73: page--; break; // PgUp
146             case 81: page++; break; // PgDn
147         }
148         m = page * 0x100 + row * 0x10 + col;
149         if(m < 0) m = 0;
150         if(m >= size) m = size-1;
151         page = m / 0x100;
152         row = m / 0x10 % 0x10;
153         col = m % 0x10;
154     }while(key!=27);
155 }
```

```

1 // Main.cpp
2 #include <iostream>
3 #include "Brow.h"
4
5 int main(int argc, char *argv[])
6 {
7 // Brow bf;           // ** 可以先用默认的构造函数创建对象 **
8     char filename[80];
9     if(argc==1)
10    {
11         cout << "Please input filename:" << flush;
12         cin.getline(filename, 80);
13    }
14    else
15        strcpy(filename, argv[1]);
16 // bf.Open(filename); // ** 后打开文件 **
17 Brow bf(filename); // 创建对象同时打开文件
18 if(bf)             // 利用重载的类型转换运算符
19     bf.Show();
20 else
21     cout << "\nSorry. Can't open file " << filename << endl;
22 return 0;
23 }

```

程序经编译、连接生成可执行文件 Brow.exe。以探究文件 Main.cpp 文件为例，在操作系统命令提示符下，输入下列命令：

Brow Main.cpp

或者执行下面的命令，然后输入文件名 Main.cpp

Brow

程序的执行结果片段如下。

```

00000000 2F*2F 20 4D 61 69 6E 2E-63 70 70 0D 0A 23 69 6E // Main.cpp..#in
00000010 63 6C 75 64 65 20 3C 69-6F 73 74 72 65 61 6D 3E clude <iostream>
00000020 0D 0A 23 69 6E 63 6C 75-64 65 20 22 42 72 6F 77 ..#include "Brow
00000030 2E 68 22 0D 0A 0D 0A 69-6E 74 20 6D 61 69 6E 28 .h"....int main(
00000040 69 6E 74 20 61 72 67 63-2C 20 63 68 61 72 20 2A int argc, char *
00000050 61 72 67 76 5B 5D 29 0D-0A 7B 0D 0A 2F 2F 09 42 argv[])...{...//.B
00000060 72 6F 77 20 62 66 3B 09-09 09 2F 2F 20 2A 2A 20 row bf;...// **
00000070 BF C9 D2 D4 CF C8 D3 C3-C4 AC C8 CF B5 C4 B9 B9 可以先用默认的构
00000080 D4 EC BA AF CA FD B4 B4-BD A8 B6 D4 CF F3 20 2A 造函数创建对象 *
00000090 2A 0D 0A 09 63 68 61 72-20 66 69 6C 65 6E 61 6D *...char filenam
000000A0 65 5B 38 30 5D 3B 0D 0A-09 69 66 28 61 72 67 63 e[80];...if(argc
000000B0 3D 3D 31 29 0D 0A 09 7B-0D 0A 09 09 63 6F 75 74 ==1)...{....cout
000000C0 20 3C 3C 20 22 50 6C 65-61 73 65 20 69 6E 70 75 << "Please inpu
000000D0 74 20 66 69 6C 65 6E 61-6D 65 3A 20 22 20 3C 3C t filename: " <<
000000E0 20 66 6C 75 73 68 3B 0D-0A 09 09 63 69 6E 2E 67 flush;...cin.g
000000F0 65 74 6C 69 6E 65 28 66-69 6C 65 6E 61 6D 65 2C etline(filename,

```

File: main.cpp (0x0+1)/0x23F = (0+1)/575 = 0.173913%

<ESC> <Home> <End> <PgUp> <PgDn> <Up> <Down> <Left> <Right>

程序执行时可使用光标移动键查看文件的内容，按 ESC 键退出。其中左侧为文件中的对应字符距文件首字节的偏移量（十六进制表示）；中间用无符号的十六进制整数输出每个字符的 ASCII 码值，其中数字后面的星号“*”表示当前字符；右侧输出对应的字符，其中非可打印字符则显示成圆点“.”。

14.5 小 结

I/O 流是程序中常用的操作，标准 I/O 流的对象在程序启动执行时已经由系统自动打开，程序结束后由系统自动将其关闭。

文件流 I/O 则需要程序员创建文件流对象并与磁盘文件建立关联。文件能长期保存程序中的数据，特别有利于大量数据的处理。数据库就是由此而发展起来的专门处理大量数据的一项技术。

字符串流是将内存视为读写数据的介质，常用于生成字符串，或从字符串中解析数据。

本章的探究文件字节内容的程序令我们看到计算机中的所有信息皆是数字——皆是 0 或 1。

练习 14

(1) 编写一个程序 TypeF.cpp 模仿 Windows 命令 type 的功能显示指定文件的内容。要求做如下改进：

① 命令行的第一个参数为指定的文件名，不可缺省。若第一个参数为问号字符“?”，则输出本命令格式的说明（称命令本身为第 0 个参数）。

② 命令行的第二个参数为一个整数。用正整数表示仅显示文件中的前若干个字符数；用负整数表示仅显示文件尾部的若干个字符。第二个参数可以缺省，缺省时指显示整个文件内容。

③ 遇到文件中的非可打印字符时以输出圆点字符“.”代替。

(2) 编写一个程序 Win2Unix.cpp 将 Windows 系统中的文本文件转换成 Unix/Linux 系统中的文本文件。要求使用命令行参数，当第二个参数缺省时将输出结果覆盖第一个参数所指的文件（提示：按二进制格式打开文件，去掉其中的回车字符 '\r' 即可）。

(3) 编写一个程序 Unix2Win.cpp 将 Unix/Linux 系统中的文本文件转换成 Windows 系统中的文本文件。要求使用命令行参数，当第二个参数缺省时将输出结果覆盖第一个参数所指的文件（提示：按二进制格式打开文件，在所遇到的每一个换行字符 '\n' 前添加一个回车字符 '\r' 即可）。

第 15 章 异常处理

本章简要介绍程序中异常状况的概念，通过一个综合程序介绍一些典型的异常及其处理方法。学习本章后，读者应该初步掌握异常处理的语法和基本应用。

15.1 异常处理的概念

程序在编译、连接生成可执行程序文件后（表明程序无语法、连接错），便可以在操作系统下启动运行。程序在运行过程中可能出现一些可以预料、难以避免的状况。这些状况就是所谓的异常。异常不一定是错误。常见的异常状况有：

- (1) 用 0 做除数。
- (2) 打开输入文件错。
- (3) 内存空间分配错。
- (4) 输入数据时类型不匹配。

对于程序中所出现的异常，甚至是极端异常（错误）状况的处理，最好不要“一遇见便终止了事”。我们希望在所预料到的操作执行前中断它，转而进行适当的处理。本质上，异常处理是一种程序流程控制转移。这种控制转移不同于以前的函数逐个地嵌套调用（入栈）及逐层地返回（退栈），异常处理应该能够跨越函数。

C++ 提供了性能卓越的异常处理机制。使程序设计人员在庞大而复杂的程序设计中能够集中精力解决主要方面的问题，遇到那些可以预料的情况，只管将其向外抛掷而不必处理。程序设计的另一些人员则可以只管圈定、捕捉并处理异常即可。这表明发生异常和处理异常不在同一函数中。这一“不在现场处理”的机制使异常处理能轻松跨越复杂的函数嵌套调用，可以从最深层的异常抛掷现场自动地逐层传递，直到跳转到能处理这种异常的层次为止，处理完异常后，程序可以继续执行；若所抛掷的异常皆未被捕捉到，则系统会使用“强制捕捉器”`terminate` 函数进行捕捉。`terminate` 是系统函数，它的默认操作是调用系统的`abort` 函数而无条件地终止程序。

15.2 异常处理的方法

C++ 处理异常的机制是由两个部分组成的，包括：

- ① 在函数中预料的可能有异常之处抛掷（`throw`）异常。
- ② 圈定并尝试调用（`try`）那些可能抛掷异常的函数，再进行捕捉（`catch`）及处理。这里出现的`throw`、`try`、`catch` 都是 C++ 的保留字，其中`throw` 是运算符，其运算优先级为第 17 级，仅高于最低的逗号运算符。

15.2.1 抛掷异常

例 15.1 编写一个函数 `int Div(int a, int b);`；计算并返回实参的商（整数）。

```
int Div(int a, int b)
{
    return a/b;
}
```

显然可以预见，当调用上述函数的第二个实参为 0 时会出现除数为 0 的错误（将可能导致程序终止）。当然，我们不希望执行 0 为除数的除法。因此，在执行除法前进行判断是必要的。

遇到第二个实参为 0 时，如何处理呢？上述函数要求返回一个 int 型的数据，不能为其指定一个值返回了之，因为返回任何值都是不妥的。事实上，函数剩下的语句不能继续执行，必须当机立断跳转。C++ 的抛掷异常便具有这个功能。上述函数修改成具有抛掷异常的形式：

```
int Div(int a, int b) throw(char);           // 函数原型
int Div(int a, int b) throw(char)           // 函数定义
{
    if(b==0) throw (char)0;      // 抛掷一种异常，类型选定为 char
    return a/b;
}
```

在第 15.3 节的程序中，将该函数设计成如下抛掷多种异常的形式：

```
int Div(int a, int b) throw(char, double);
int Div(int a, int b) throw(char, double)
{
    if(b==0) throw (char)0;      // 视情况抛掷不同类型的异常
    if(a%b) throw (double)b;
    return a/b;
}
```

其中， $b==0$ 成立时为了避免错误而抛掷一个 char 型异常；而 $a \% b \neq 0$ 成立时则是为了回避一种特定的状态（不是错误），本程序中约定抛掷一个 double 型的异常。抛掷何种类型的异常及其值，由程序员自行设定。

要求函数自己处理异常的确是“强人所难”，是“为难”函数的事情。因为函数并不能擅自修改调用时所传入的参数值，而擅自变更任务内容。调用函数时所用的实际参数是由调用者或者更“高层”指定的，理应由他们进行处理。例如，设计一个打开文件的函数 void Open(char *filename);，当实参所指的文件不存在而无法打开时，显然，函数 Open 不能自作主张地随便打开另一个文件。当函数遇到自己无法完成的事情时，最好的处理方案是“说明原因，上交矛盾”，由下达任务者另行处理，这就是 C++ 异常处理的基本思想。

【说明】

(1) throw 是 C++ 的运算符。抛掷异常表达式为

```
throw 表达式
```

在其后加分号 “;” 便是抛掷异常语句。

(2) 函数在执行 `throw` 操作后将终止所在函数的执行，自动析构所在函数内已经创建，但尚未析构的局部自动对象，保证将可能的损失“降至最低”。程序控制转向本函数的调用者（其上一层函数）期待异常被捕捉及处理。

(3) 可以抛掷任何数据类型，包括用户自定义的类类型，任何指针类型。抛掷的类型可与函数类型相同或不同。可以分情况抛掷一种或多种异常类型。

(4) 可以在用于函数声明的函数原型上添加抛掷类型声明。这样便于使用该函数的程序员清楚地知道该函数可能有异常抛掷，利于圈定及捕捉处理。注意，在函数定义时也必须加上抛掷类型声明，否则编译系统认为抛掷类型不匹配而报错。若在函数原型中未列出可能抛掷的异常类型，则该函数允许抛掷任何类型的异常。

(5) 若要声明一个不能抛掷异常的函数，则需要在函数原型和定义中声明无参数的 `throw`，如：

```
int funcA(int a, double x) throw();
```

即使在函数执行过程中出现了 `throw` 操作，实际上也并不执行抛掷语句，不抛掷任何异常信息，但程序将非正常终止。

可以看到，若预料到了可能出现某种状况，其实只要“一抛了之”，将“矛盾上交”由“上级部门”酌情处理。

15.2.2 圈定及捕捉处理

执行圈定、捕捉并处理的语句是 `try-catch` 语句块。`try-catch` 语句块分为 `try` 子块和其后顺序排列的一系列 `catch` 子块构成。语法格式为

```
try
{
    被圈定的语句序列;          // 可能有多个语句抛掷多种异常
}
catch(数据类型1 形式参数) // 捕捉“数据类型1”的异常，形式参数可缺省
{
    具体的异常处理语句;
}
catch(数据类型2 形式参数) // 捕捉“数据类型2”的异常
{
    具体的异常处理语句;
}
catch(...)
{
    处理其他任何类型异常;
}
```

【说明】

(1) `try` 子块需与 `catch` 子块配合使用。一个 `try-catch` 语句块只能有一个 `try` 子块，可以有多个 `catch` 子块。当捕捉到一个异常时，将根据该异常类型（数据类型）依照 `catch` 子块顺序进行匹配；找到严格匹配的类型后即进入该 `catch` 子块执行其中的处理语句；处理语句执行完毕后，直接跳过其他的子块执行本 `try-catch` 语句块之后的语句。若没有严格的类型匹配则直接将异常继续上交（不进行类型兼容性匹配）。

(2) 在调用具有抛掷异常的函数时，最好用 `try` 子块将其圈定。否则一旦出现抛掷异常而又没有圈定（当然就无法处理），将导致强制捕捉和程序终止。一个 `try` 子块可以圈定多个可能抛掷异常的函数调用语句。若在执行调用函数时无异常抛掷，则不会执行其后面的任何 `catch` 子块语句。

(3) 捕捉异常时首先根据异常的类型——数据类型——分门别类地处理。`catch` 子块中可以缺省形式参数。`catch` 子块中的数据类型不能重复，类型名称可以用省略号（“...”，即三个句点号）表示任何数据类型。这种 `catch` 子块将捕捉任何类型的异常，因此应该将这种子块放置在所有 `catch` 子块之后，表示“其他”类型。这种 `catch` 子块可以缺省。

(4) `catch` 子块中指定的数据类型后可以带形式参数，形式参数可以是传值型、指针型、引用型的。此时在子块内可以访问所抛掷的异常变量或异常对象。

(5) `catch` 子块在处理异常时，还可以再次将所捕捉到的异常抛掷出去，期待上一层的函数进行进一步处理。若不再抛掷该异常，则表示该异常处理结束。

(6) 若调用函数所抛掷的异常未被其上一层的函数圈定，或圈定后未被捕捉到，则该异常自动地继续向更上一层调用函数传递。倘若最终无法捕捉并处理，则导致系统“强制捕捉”而终止程序。

例 15.2 简单的异常处理。

源代码 15.1 简单的异常处理示例程序

```

1 // 简单的异常处理.cpp
2 #include <iostream>
3 using namespace std;
4
5 int Div(int a, int b) throw(char, double);           // 函数原型
6
7 int main()
8 {
9     int x[5] = {10, 10, 8, 6, 4};
10    int y[5] = {5, 3, 0, 2, 0};
11    int n=0, i, z;
12
13    for(i=0; i<5; i++)
14    {
15        try                                // 圈定
16        {
17            cout << "No." << i+1 << "\t"
18            << x[i] << "/" << y[i];
19
20            z = Div(x[i], y[i]);           // 调用可能有异常抛掷的函数
21            cout << "=" << z << endl;
22            n++;
23        }
24        catch(char)                      // 捕捉及处理
25        {
26            cout << "\t除数为0。跳过此题。" << endl;
27        }
28        catch(double &x)                // 此处的 x 为引用型形式参数
29        {
30            cout << "\t除不尽，除数为" << x // 访问被抛掷的变量

```

```

31             << "。跳过此题。" << endl;
32     }
33 }
34 cout << "共" << n << "题。" << endl;
35 return 0;
36 }
37
38 int Div(int a, int b) throw(char, double) // 函数定义
39 {
40     if(b==0) throw (char)0;                  // 抛掷异常
41     if(a%b) throw (double)b;                // 抛掷异常
42     return a/b;
43 }
```

程序的运行结果：

```

No. 1 10/5 = 2
No. 2 10/3    除不尽，除数为 3。跳过此题。
No. 3 8/0      除数为 0。跳过此题。
No. 4 6/2 = 3
No. 5 4/0      除数为 0。跳过此题。
共 2 题。
```

【注意】 程序中第 22 行语句 `n++;` 的位置很巧妙。当第 20 行调用 `Div` 函数时，若有异常抛掷，则便不会执行其后的第 21、22 两行语句。变量 `n` 的值不会增加。在下一节的程序中有意改变了语句 `n++;` 的位置，是想说明异常处理时还可以做一些“有实际意义”的事情。

15.3 趣味程序——正整数算术运算测验程序

本节讨论一个“正整数算术运算测验”程序。所假想的用户是仅学了整数算术四则运算的小学生们，他们暂时还没有负数和小数的概念，出现负数、小数结果的题目便是异常的。

倘若测试数据（包括操作数和运算符）都随机产生，则出现“题目异常”是可以预料且难以避免的。为了简单起见，将题目算式按一定的格式编入文本文件中，由程序运行时通过指定文件名打开（可以预料可能出现“打开文件错”的异常发生）。文件中的第一行为该文件中的题目数量（实际题目数量取决于其后的题目行数，规定不超过该数量），接下来各行是具体题目的算式。下面是三个供测试程序使用的题目文件的具体内容。

文件 `test0.txt` 的内容：

```

-2
12-10
8/2
```

文件 `test1.txt` 的内容：

```

5
1 -2
10/3
10-20
```

4-10
3/0
6/3

文件 test2.txt 的内容:

20
10-12
8/0
7 * 6
9 + 3
4-5
2+8
7*2
6*0
3*10
5+0

① 文件 test0.txt 的第一行为负数将导致一种异常。即由于 `new int[-2]`; 操作导致动态分配内存错，这时系统将抛掷一个类型为 `bad_alloc` 的异常。

② 文件 test1.txt 的第一行为正整数 5，而其后有 6 个题目，此时，仅前 5 题被处理，我们不将此视为异常。然而前 5 道题皆为“题目异常”，则有效题数为 0，无法计算成绩视为异常。

③ 文件 test2.txt 的第一行为正整数 20，而其后仅有 10 个题目，则只处理这 10 道题目。并不将此视为异常。

程序中还处理了一种键盘输入异常，本程序按异常处理方法进行处理。另请参见第 14.1.2 小节中的内容“抽取数值不当的处理”，在第 15 章中采用的是现场即时处理方法（采用函数模板编写以适用于多种数值数据类型）。这两种处理方法皆是可取的。

程序中还将“输入用户名为空”作为一种异常，并利用这种异常跳出主函数中的主循环。这是利用异常处理机制实现控制转移的一种方法。

综上所述，本问题的异常类型指定及其处理方案如表 15-1 所示^[1]。

表 15-1 异常类型及其处理方案

序号	预料的异常现象	异常类型指定	抛掷异常函数	异常处理方案
1	被减数小于减数	<code>int</code>	<code>Sub, Testing</code>	该题不计分
2	除法运算除以 0	<code>char</code>	<code>Div, Testing</code>	该题不计分
3	除法运算除不尽	<code>double</code>	<code>Div, Testing</code>	该题不计分
4	键盘输入整数错	<code>float</code>	<code>InputInt</code>	重做该小题
5	打开题目文件错	<code>ifstream*</code>	构造函数, <code>Open</code>	重新开始
6	动态分配内存错	<code>bad_alloc</code>	构造,拷贝构造,赋值, <code>Open</code>	重新开始
7	输入空用户名称	<code>char*</code>	<code>GetName</code>	退出主循环

[1] 第 8 种异常是有效题目数为 0，即题目全部无效，此时无法计算成绩。这个异常的类型可能是 `int`, `char` 或 `double` 型的，它取决于最后一个无效的题目的异常类型。

为此，设计一个类 `class myTest;`，其中数据成员中有指向动态内存空间的指针。亦即该类的对象可能带有堆内存资源，故需要定义深拷贝构造函数、深赋值运算符函数和析构函数。此外，加减乘除函数、接收键盘输入整数的函数为友元。在函数（包括成员函数和友元函数）原型中明确地指出该函数可能抛掷的异常类型以便其他程序员使用。其中抛掷异常类型 `bad_alloc` 的真正源头是 `new` 操作符。下面是类 `class myTest;` 声明的头文件。

源代码 15.2 正整数算术运算测验程序·头文件（异常处理）

```

1 // TestException.h
2 #ifndef TEST_EXCEPTION_H
3 #define TEST_EXCEPTION_H
4 #include <string>
5 #include <iostream>
6 using namespace std;
7
8 class myTest
9 {
10 public:
11     myTest(char *username="NoName", char *filename=NULL)
12             throw(ifstream*, bad_alloc);
13     void Open(char *filename)    throw(ifstream*, bad_alloc);
14     // 可能抛掷文件指针异常，内存分配异常。采用指针类型以避免复制构造
15     myTest(const myTest &t)           throw(bad_alloc);
16     myTest & operator=(const myTest &t) throw(bad_alloc);
17     virtual ~myTest();
18     void Testing() throw(char, int, double);
19             // 可能抛掷三种类型异常
20     void Show();
21     friend int Add(int a, int b){ return a+b; }
22     friend int Mul(int a, int b){ return a*b; }
23     friend int Sub(int a, int b)      throw(int);
24             // 可能抛掷整型异常
25     friend int Div(int a, int b)      throw(char, double);
26             // 可能抛掷两种类型异常
27     friend int InputInt()           throw(float);
28
29 private:
30     string name;          // 用户（测试者）姓名
31     int num, *x, *y, *op; // 预定题数，题目数据，运算符号
32     double score;         // 测试成绩
33 };
34 #endif

```

成员函数和未定义的友元函数定义见下面的文件。其中成员函数 `void myTest::Testing() throw(char, int, double);` 是该类的一个主要函数，它是一些异常的圈定、捕捉及处理的现场：处理“题目异常”和“键盘输入整数错”。当出现“有效题数为 0”异常时，该函数无法计算成绩（`score`），它便将其捕捉到的异常再度抛掷出去，上交给主函数，期待主函数捕捉并处理。

源代码 15.3 正整数算术运算测验程序·源程序文件（异常处理）

```

1 // TestException.cpp
2 #include <iostream>
3 #include "TestException.h"

```

```

4  using namespace std;
5  // 友元函数定义
6  int Sub(int a, int b) throw(int)
7  {      // 本函数需要有返回值，出现异常时不能随便返回一个值了之
8      if(a<b)
9          throw b;           // 只管抛掷异常，由调用者处理
10     return a-b;
11 }
12 int Div(int a, int b) throw(char, double)
13 {      // 本函数需要有返回值，出现异常时不能随便返回一个值了之
14     if(b==0)           // 除数为零，抛掷字符型异常
15         throw (char)0;
16     if(a % b)          // 不能除尽，抛掷浮点型异常
17         throw (double)b;
18     return a/b;
19 }
20 int InputInt() throw(float)
21 {
22     int x;
23     if(!(cin>>x))
24         throw (float)x; // 如输入字符，Ctrl+Z 组合键等，抛掷异常
25     return x;
26 }
27
28 // 成员函数定义
29 myTest::myTest(char *username, char *filename)
30             throw(ifstream*, bad_alloc) : name(username)
31 {           // 此构造函数可能抛掷由 Open 函数所抛掷的任何异常
32     score = num = 0;
33     x = y = op = NULL;
34     if(filename) Open(filename);
35 }
36 myTest::myTest(const myTest &t) throw(bad_alloc) : name(t.name)
37 {           // 深复制构造函数。注意冒号语法的位置
38     score = t.score;
39     num = t.num;
40     x = new int[num];    // 此处可能由系统抛掷 bad_alloc 异常
41     y = new int[num];    // 同上
42     op = new int[num];   // 同上
43     for(int i=0; i<num; i++)
44     {
45         x[i] = t.x[i];
46         y[i] = t.y[i];
47         op[i] = t.op[i];
48     }
49 }
50 myTest::~myTest()
51 {           // 析构函数
52     if(x) delete [] x;
53     if(y) delete [] y;
54     if(op) delete [] op;
55 }
56 myTest & myTest::operator=(const myTest &t) throw(bad_alloc)
57 {           // 深赋值运算符重载
58     if(&t == this) return *this;
59     if(x) delete [] x;
60     if(y) delete [] y;
61     if(op) delete [] op;
62     score = t.score;

```

```

63     num = t.num;
64     x = new int[num];      // 此处可能由系统抛掷 bad_alloc 异常
65     y = new int[num];      // 同上
66     op = new int[num];     // 同上
67     for(int i=0; i<num; i++)
68     {
69         x[i] = t.x[i];
70         y[i] = t.y[i];
71         op[i] = t.op[i];
72     }
73     return *this;
74 }
75
76 void myTest::Open(char *filename) throw(ifstream*, bad_alloc)
77 {
78     // 创建对象后再打开文件，分配内存
79     if(x) delete [] x;
80     if(y) delete [] y;
81     if(op) delete [] op;
82     score = 0;
83     ifstream infile(filename);
84     if(infile.fail()) throw &infile; // 打开文件错，抛掷文件型异常
85
86     infile >> num;
87     x = new int[num];      // 此处可能由系统抛掷 bad_alloc 异常
88     y = new int[num];      // 同上
89     op = new int[num];     // 同上
90
91     char c;
92     int i;
93     for(i=0; i<num && infile >> x[i] >> c >> y[i]; i++)
94     {
95         if(c=='/') op[i] = 3;
96         else if(c=='*') op[i] = 2;
97         else if(c=='-') op[i] = 1;
98         else op[i] = 0;
99     }
100    num = i;           // 实际题目数量
101    infile.close();
102 }
103 void myTest::Testing() throw(char, int, double)
104 {
105     // 圈定、捕捉处理异常现场。可能继续抛掷异常
106     int i, n=0, z;
107     char ch_op[4] = {'+', '-', '*', '/'};
108     int (*f[4])(int, int) = {Add, Sub, Mul, Div};
109
110     score = 0;
111     for(i=0; i<num; i++)
112     {
113         cout << "No." << i+1 << "of" << num
114         << "\t" << x[i] << " " << ch_op[op[i]]
115         << " " << y[i] << " = ";
116         n++;
117         try
118         {
119             z = InputInt();          // 预料可能出现异常之处
120             if(z==f[op[i]](x[i], y[i])) // 预料可能出现异常之处
121                 score++;
122         }

```

```

122     catch(float)           // 按所捕获的异常类型，分情况处理
123     {
124         cout << "\t\t"
125             << "输入有误。此题重做。" << endl;
126         cin.clear();          // 清理错误状态位
127         cin.ignore(80, '\n'); // 跳过 80 个字符，或遇到换行符
128         n--;
129         i--;                // i--；的目的是重做此题
130     }
131     catch(int)
132     {
133         cout << "\t\t"
134             << "被减数小于减数。此题不计入成绩。" << endl;
135         n--;                  // 能处理的，尽量处理
136         if(i==num-1 && n==0) throw; // 不便处理，上交矛盾
137     }
138     catch(char)
139     {
140         cout << "\t\t"
141             << "除数为 0。此题不计入成绩。" << endl;
142         n--;                  // 能处理的，尽量处理
143         if(i==num-1 && n==0) throw; // 不便处理，上交矛盾
144     }
145     catch(double)
146     {
147         cout << "\t\t"
148             << "除不尽。此题不计入成绩。" << endl;
149         n--;                  // 能处理的，尽量处理
150         if(i==num-1 && n==0) throw; // 不便处理，上交矛盾
151     }
152 }
153 score *= 100.0/n;           // 此时的 n 不会为零
154 }
155 void myTest::Show()
156 {
157     cout << "Name:" << name << ", Score:" <<
158         score << "." << endl;
159 }
```

主函数是异常处理的最后一道屏障。一般应该使其具有处理所有异常的能力，不应有“漏网”者而导致程序异常结束。

源代码 15.4 正整数算术运算测验程序·主程序（异常处理）

```

1 // Main.cpp
2 #include <iostream>
3 #include "TestException.h"
4 using namespace std;
5
6 void GetName(char *name, char *filename) throw(char*)
7 {
8     cin.sync();           // 刷新标准输入缓冲区
9     // 特别是清除前面输入数据可能留下的换行符，便于下面的getline函数操作
10    cout << "请输入姓名: ";
11    cin.getline(name, 40);
12    if(name[0]=='\0') throw name; // 未输入而直接回车则抛掷异常
13    cout << "请输入题目文件名: ";
14    cin.getline(filename, 80);
```

```

15 }
16
17 int main()
18 {
19     char name[40], filename[80];
20     while(true)
21     {
22         try
23         {
24             GetName(name, filename);           // 预料可能有异常被抛掷
25             myTest user(name, filename);    // 预料可能有异常被抛掷
26             user.Testing();                // 预料可能有异常被抛掷
27             user.Show();
28         }
29         catch(ifstream*)
30         {
31             cout << "题目文件不存在。" << endl;
32         }
33         catch(bad_alloc)
34         {
35             cout << "内存分配错误。" << endl;
36         }
37         catch(int)
38         {
39             cout << "题目全部无效。" << endl; // 注意输出句号
40         }
41         catch(char)
42         {
43             cout << "题目全部无效！" << endl; // 注意输出感叹号
44         }
45         catch(double)
46         {
47             cout << "题目全部无效！！" << endl; // 注意输出双感叹号
48         }
49         catch(...)
50         {
51             cout << "其他异常。测试完毕。" << endl;
52             break;
53         }
54     }
55     cout << "返回操作系统。" << endl;
56     return 0;
57 }
```

程序的运行结果：

```

请输入姓名: 张三
请输入题目文件名: test0.txt
内存分配错误。
请输入姓名: 张三
请输入题目文件名: test1.txt
No. 1 of 5      1 - 2 = 0
被减数小于减数。此题不计入成绩。
No. 2 of 5      10 / 3 = 1
除不尽。此题不计入成绩。
No. 3 of 5      10 - 20 = 2
被减数小于减数。此题不计入成绩。
```

```

No. 4 of 5      4 - 10 = 3↓
被减数小于减数。此题不计入成绩。
No. 5 of 5      3 / 0 = 4↓
除数为 0。此题不计入成绩。
题目全部无效！
请输入姓名：李 四↓
请输入题目文件名： test2.txt↓
No. 1 of 10     10 - 12 = aaa↓
输入有误。此题重做。
No. 1 of 10     10 - 12 = 5↓
被减数小于减数。此题不计入成绩。
No. 2 of 10     8 / 0 = 6↓
除数为 0。此题不计入成绩。
No. 3 of 10     7 * 6 = bbb↓
输入有误。此题重做。
No. 3 of 10     7 * 6 = 42↓
No. 4 of 10     9 + 3 = 12↓
No. 5 of 10     4 - 5 = 10↓
被减数小于减数。此题不计入成绩。
No. 6 of 10     2 + 8 = 10↓
No. 7 of 10     7 * 2 = 14↓
No. 8 of 10     6 * 0 = 0↓
No. 9 of 10     3 * 10 = 30↓
No. 10 of 10    5 + 0 = 0↓
Name: 李 四, Score: 85.7143.
请输入姓名：王 五↓
请输入题目文件名： test3.txt↓
题目文件不存在。
请输入姓名：↓
其他异常。测试完毕。
返回操作系统。

```

程序执行时进行如下操作。

① 首先输入用户名“张 三”，输入文件名 `test0.txt`。该文件中的第一行题目数量为负数，无法分配堆内存空间。

② 输入用户名“张 三”和文件名 `test1.txt` 出现全部五次异常，由成员函数 `void myTest::Testing() throw(char, int, double);` 将异常“上交”给 `main` 函数进行处理。其中最后一道题目是 0 为除数，抛掷 `char` 型异常，主函数中相应的异常捕捉处理子块输出“题目全部无效！”（一个感叹号者）。

③ 输入用户名“李 四”和文件名 `test2.txt` 在十个题目中出现三次异常、输入时出现两次异常（分别输入字符串 `aaa`, `bbb`）被要求重做该小题。在有效的七题中最后一题做错，因此得分 $6/7 \times 100 \approx 85.7143$ 分。

④ 输入用户名“王 五”和文件名 `test3.txt`。由于该文件不存在，捕捉到所抛掷的异常。

⑤ 最后在输入用户名时直接回车，由函数 `void GetName(char *name, char *filename) throw(char*)` 抛掷一个 `char*` 型异常。程序中并没有专门捕捉这种类型的异常，这个异常是被 `catch(...)` 子块捕捉并处理的，执行其中的 `break;` 语句结束主函数中的循环，执行最后的语句输出“返回操作系统。”，程序结束。

15.4 小 结

函数调用是通过栈结构实现的。函数的嵌套调用是一个压栈过程，函数的返回是逐次退栈过程。异常的传播方向与函数退栈过程相同——使最近的能处理异常的层次处理（不一定蔓延到最高层，直至崩溃）。

异常不一定是错误。C++ 异常处理机制也给程序员提供了一种新的流程控制方法。

设计函数（包括类的成员函数）时，在预料出现异常处只管抛掷可能的异常，而不必考虑本函数将可能在何种情形下、何种深层次中被调用。

调用有抛掷异常的函数时，应该将其调用语句进行圈定。也就是将可能抛掷异常的函数调用语句放入 `try-catch` 语句块的 `try` 子块中，并根据异常类型分情况捕捉（`catch`）及进行处理。

练习 15

请分析第 15.3 节中的源程序。分别画出八种异常从其被抛掷出来到最后被圈定、捕捉及处理所经历的函数（异常传播路径）示意图。

参 考 文 献

- [1] 钱能. C++ 程序设计教程 (第 1 版). 北京: 清华大学出版社, 1999.
- [2] Bruce Eckel. C++ 编程思想 (第 1 卷) : 标准 C++ 导引. 刘宗田译. 北京: 机械工业出版社, 2002.
- [3] Stroustrup B. C++ 程序设计语言 (特别版). 裴宗燕译. 北京: 机械工业出版社, 2002.
- [4] Herb Sutter, Andrei Alexandrescu. C++ 编程规范. 刘基诚译. 北京: 人民邮电出版社, 2006.

附录 A ASCII 字符集

	0	1	2	3	4	5	6	7
0	NUL	DLE	空格	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	-	o	DEL

说明：上表列出了 ASCII 基本字符集（7位）。表中纵向 0~F 为低4位，横向 0~7 为高3位。例如，字符 A 的 ASCII 码的十六进制值为 0x41（十进制为 65）。

附录 B 常用库函数参考

B.1 C-字符串函数

函数原型在头文件 `cstring` 或 `string.h` 中。

函数原型	<code>int strlen(const char *str);</code>
入口参数	字符型地址值（指针）
返 回 值	整数
功 能	计算并返回字符串的长度，不包括串结束标志
函数原型	<code>char * strcpy(char *dest, const char *source);</code>
入口参数	<code>source</code> 为源字符串首地址， <code>dest</code> 为目标字符串首地址
返 回 值	目标字符串首地址
功 能	将源串的内容复制到目标字符串中
函数原型	<code>char * strncpy(char *dest, const char *source, size_t n);</code>
入口参数	<code>n</code> 为复制的字符数，其余的参数同 <code>strcpy</code> 函数
返 回 值	目标字符串首地址
功 能	将源串的前 <code>n</code> 个字符复制到目标字符串。若原串的长度小于 <code>n</code> ，则复制源串，不足 <code>n</code> 个字符时，用串结束标志补足。若字符串长度大于或等于 <code>n</code> ，则仅复制前 <code>n</code> 个字符。不复制、不添加串结束标志
函数原型	<code>void * memcpy(void *dest, const void *source, size_t n);</code>
入口参数	目标地址，源地址， <code>n</code> 为复制的字节数
返 回 值	目标地址值
功 能	将内存 <code>source</code> 起始处的 <code>n</code> 个字节复制到目标地址 <code>dest</code> 处
函数原型	<code>char * strcat(char *dest, const char *source);</code>
入口参数	源及目标字符串首地址
返 回 值	目标字符串首地址
功 能	将源字符串的内容拼接到目标字符串的尾部
函数原型	<code>char * strncat(char *dest, const char *source, size_t n);</code>
入口参数	源、目标字符串首地址，整数 <code>n</code>
返 回 值	目标字符串首地址
功 能	将源字符串中前 <code>n</code> 个字符拼接到目标字符串的尾部

函数原型	<code>int strcmp(const char *str1, const char *str2);</code>
入口参数	两个字符串首地址
返回值	整数
功能	将两个字符串的内容按字典序比较。根据 str1 的内容大于、小于、等于 str2 的内容，分别返回一个大于零、小于零、等于零的值
函数原型	<code>char *strupr(char *str);</code>
入口参数	字符串首地址
返回值	入口参数的地址值
功能	将字符串中的所有英文小写字母转换成对应的大写字母，其他字符不变
函数原型	<code>char *strlwr(char *str);</code>
入口参数	字符串首地址
返回值	入口参数的地址值
功能	将字符串中的所有英文大写字母转换成对应的小写字母，其他字符不变

B.2 转换函数

函数原型在 `cstdlib` 或 `stdlib.h` 中。

函数原型	<code>int atoi(const char *str);</code>
入口参数	字符串首地址
返回值	整数
功能	将数码字符串转换成相应的整数。若不能转换则返回整数 0
函数原型	<code>long atol(const char *str);</code>
入口参数	字符串首地址
返回值	长整型数
功能	将数码字符串转换成相应的长整型数。若不能转换则返回整数 0
函数原型	<code>double atof(const char *str);</code>
入口参数	字符串首地址
返回值	双精度浮点型数
功能	将数码字符串转换成相应的双精度浮点型数。若不能转换则返回 0.0

B.3 随机数发生器

函数原型	<code>void srand(unsigned int seed);</code>
入口参数	无符号整型数
返回值	无
功能	设置随机数发生器的种子

函数原型	<code>int rand();</code>
入口参数	无
返 回 值	<code>0~RAND_MAX</code> 之间的伪随机整数
功 能	产生伪随机数

B.4 数学函数

函数原型在头文件 `cmath` 或 `math.h` 中。

绝对值函数：

<code>int abs(int);</code>	// 整型数据的绝对值
<code>long labs(long);</code>	// 长整型数据的绝对值
<code>double fabs(double);</code>	// 浮点型数据的绝对值

三角函数及反三角函数：

<code>double sin(double x);</code>	// 正弦函数, x 为弧度
<code>double cos(double x);</code>	// 余弦函数, x 为弧度
<code>double tan(double x);</code>	// 正切函数, x 为弧度
<code>double asin(double x);</code>	// 反正弦函数, $x \in [-1, 1]$, 函数值为弧度
<code>double acos(double x);</code>	// 反余弦函数, $x \in [-1, 1]$, 函数值为弧度
<code>double atan(double x);</code>	// 反正切函数
<code>double atan2(double y, double x);</code>	// $x \neq 0$ 时, 为 <code>atan(y/x)</code> ; 否则返回 <code>sgn(y)π/2</code>

指数、对数、幂函数：

<code>double sqrt(double x);</code>	// 计算 \sqrt{x} , 要求 $x \geq 0$
<code>double exp(double x);</code>	// 计算 e^x
<code>double pow(double x, double y);</code>	// 计算 x^y , 要求 $x \geq 0$
<code>double log(double x);</code>	// 计算 $\ln(x)$, 要求 $x > 0$
<code>double log10(double x);</code>	// 计算 $\lg(x)$, 要求 $x > 0$

双曲函数：

<code>double sinh(double x);</code>	// 双曲正弦函数
<code>double cosh(double x);</code>	// 双曲余弦函数
<code>double tanh(double x);</code>	// 双曲正切函数

它们的定义为

$$\sinh(x) = \frac{e^x - e^{-x}}{2}, \quad \cosh(x) = \frac{e^x + e^{-x}}{2}, \quad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$