

C++程序设计经典 300 例

侯晓琴 编著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书针对 C++ 语言的各个应用方向,分为 3 篇共 16 章,一共收集了 300 个典型实例。第 1 篇涉及 C++ 语言的基础语法、数组、字符串、内存、指针、函数及类的应用等基础知识。第 2 篇涵盖泛型编程技术、输入/输出系统以及如何解决各类经典出错案例。第 3 篇重点针对各类应用展开介绍,如一些基础技术的复杂应用实例、Socket 网络及进程间通信、算法、多线程、动态链接库技术的应用等,最后还通过实例演示了数字图像处理技术和三维仿真技术的相关应用。

本书实例具有代表性,能直接应用于真实的开发实践中,可作为自学 C++ 语言和大中专院校师生提高编程实践能力的指导教材,也可作为在职 C++ 开发人员的参考书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

C++程序设计经典 300 例 / 侯晓琴编著. —北京: 电子工业出版社, 2014.6
ISBN 978-7-121-23065-3

I. ①C… II. ①侯… III. ①C 程序—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2014)第 081920 号

策划编辑: 牛 勇

责任编辑: 李利健

印 刷: 三河市双峰印刷装订有限公司

装 订: 三河市双峰印刷装订有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1092 1/16 印张: 24 字数: 669 千字

印 次: 2014 年 6 月第 1 次印刷

定 价: 59.00 元(含光盘 1 张)

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

前 言

对于正在学习 C++ 语言的读者来说，模仿大量的案例并学习其详细的解析是一个必不可少的过程。本书精心选择了 300 个实例，对 C++ 语言的常见应用进行了举例说明，其中不仅涵盖 C++ 语言的各类基础知识，还包括高级应用及结合第三方库综合应用。本书凝聚了笔者多年学习并实践 C++ 语言的开发经验和技巧。

本书的实例以最直观、最经典并紧密结合相关知识点的概念传授 C++ 语言的编程知识和开发思想。在学习这 300 个实例的过程中，读者可以慢慢体会 C++ 语言的编程套路，并形成自己的编程思维。书中的所有程序都已在 Visual Studio 开发平台下调试并通过，最后两章内容需要结合所介绍的相应版本库开发使用。

本书特点

本书选择的实例都是 C++ 语言在开发应用过程中经常会遇到的编程问题，从实践的角度阐述如何运用 C++ 语言的各个特性实现不同项目的开发。全书讲解由浅入深，内容全面，兼顾深度和广度，以期循序渐进地引导读者完成 C++ 语言的学习。全书在内容安排和实例讲解上的主要特点如下。

- 前两篇的实例涵盖了 C++ 语言的基本知识点，并且从实际应用出发使读者精通各个案例的发生背景。
- 讲解方式细致体贴，力求用准确的语言描述实例应用背景，用规范、精简的代码实现功能，用重点突出的方式提供注释介绍。
- 本书的实例代码长度适中，实现过程及思路有助于读者进一步领会新功能。
- 对于重要知识点，书中给出了详细的说明，以使读者做到触类旁通。
- 尽可能地体现 C++ 语言在当前的应用思想，比如多线程、网络和算法应用等。
- 除了基本的 C++ 语言应用，本书在最后两章介绍了使用 OpenCV 和 OSG 语言实现数字图像处理和三维仿真技术的开发，有助于读者了解这些专题的项目需求。

本书内容及知识体系

第 1 篇 C++ 入门案例（第 1 章～第 7 章），涵盖 C++ 语言的基本语法知识、典型的变量类型，学习本篇有助于读者更深入地了解 C++ 语言如何与计算机的内存进行“沟通”。

本篇包括 128 个实例，主要涉及 C++ 语言的基础语法、数组、字符串、内存、指针、函数及类的应用等基础知识。

第 2 篇 C++ 进阶案例（第 8 章～第 10 章），介绍了相对高级的 C++ 语言应用，以形成 C++ 编程的抽象思维。

本篇包括 68 个实例，主要涉及泛型编程技术、输入/输出系统及如何解决各类经典出错案例。读者通过对这些案例的学习，为后续的大型项目应用提供了关键性的承上启下作用。

第 3 篇 C++ 高级案例（第 11 章～第 16 章），主要介绍一些 C++ 语言的分支应用，通过对本篇的学习，使读者明确今后的发展方向。

本篇包括 104 个实例，主要涉及基础高级应用、Socket 网络及进程间通信、算法、多线程、动态链接库技术的应用，最后包含数字图像处理技术和三维仿真技术两个专题。



本书约定

- 【实例描述】对实例的应用背景给出说明，部分实例给出运行效果。
- 【实现过程】给出解决问题的步骤和代码。
- 【代码解析】对代码的实现原理和关键技术点进行解析。
- 【其他说明】给出有关实例操作需要特别注意的小技巧和小知识。

适合阅读本书的读者

- 对 C++语言应用开发感兴趣的自学读者
- C++语言刚刚入门，需进一步提高实战技术的开发人员
- 需要课后练习资料的大中专院校和培训学校的师生
- 需要相关参考资料的 C++语言初级、中级程序员
- 数字图像处理技术及三维仿真技术人员

阅读本书的建议

为了提高读者的学习效率，增强学习效果，笔者特别提出以下学习建议。

- 对于刚刚接触 C++语言的初学者，请严格按照章节顺序阅读本书，不可跳跃，以期打牢基础，尽快适应复杂知识点的学习。
- 语言的学习初期重在模仿，先不要着急独立写代码，多学习书中的套路。因此，对于本书的各个实例一定要明白其实现原理和过程。
- 熟悉完每个实例的实现思路和实现过程后，一定要亲自书写和调试代码，完成书中的所有实例，这样有利于加深对内容的理解和记忆。
- 学会利用网络资源，在遇到问题时，尽量独立思考解决问题。这样获取的知识更坚固，不容易忘记，更能打开自己的思路。
- 本书所有的代码都是在 Visual Studio 开发平台测试运行，建议读者使用该平台学习。另外，数字图像处理专题使用 OpenCV 2.0 版本开发，三维仿真专题使用 OSG 2.8.0 版本开发。

本书由侯晓琴组织编写，参与编写工作的还有张燕、杜海梅、孟春燕、吴金艳、鲍凯、庞雁豪、杨锐丽、鲍洁、王小龙、李亚杰、张彦梅、刘媛媛、李亚伟、张增强，在此一并表示感谢。

在本书的写作过程中，作者总结了学习及工作中遇到和解决各类问题，以期让读者站在实践的角度学习各个知识点。网络资源是一个好工具，它让更多志同道合的程序员通过网线连接在一起，感谢他们的慷慨分享。由于作者水平有限，不足之处甚至错误的地方在所难免，敬请广大读者批评、指正。

作 者

目 录

第 1 篇 C++入门案例

第 1 章 从最简单的案例讲述 C++	1
实例 001 在屏幕上输出内容	1
难度指数 ★☆☆☆☆ 占用时间 ②③③	
实例 002 规格不同的箱子（变量）	2
难度指数 ★★☆☆☆ 占用时间 ②③③	
实例 003 物品存放（变量）	3
难度指数 ★★☆☆☆ 占用时间 ②③③	
实例 004 交换物品（变量）	4
难度指数 ★★☆☆☆ 占用时间 ②③③	
实例 005 消失的重量（隐式类型转换）	4
难度指数 ★★☆☆☆ 占用时间 ②③③	
实例 006 游泳池的容量	5
难度指数 ★★☆☆☆ 占用时间 ②③③	
实例 007 显式转换（类型转换）	6
难度指数 ★★☆☆☆ 占用时间 ②③③	
实例 008 单片机应用（位操作）	7
难度指数 ★★☆☆☆ 占用时间 ②③③	
实例 009 房间面积（*）	8
难度指数 ★☆☆☆☆ 占用时间 ②③③	
实例 010 平分物品（/）	9
难度指数 ★☆☆☆☆ 占用时间 ②③③	
实例 011 取余数（%）	10
难度指数 ★☆☆☆☆ 占用时间 ②③③	
实例 012 公司组织看电影（综合）	11
难度指数 ★★☆☆☆ 占用时间 ②③③	
实例 013 称水果（比较）	12
难度指数 ★☆☆☆☆ 占用时间 ②③③	
实例 014 简单算术（优先级）	13
难度指数 ★★☆☆☆ 占用时间 ②③③	
实例 015 输出 Huffman 编码（结构+算法）	14
难度指数 ★★★★★ 占用时间 ②③③	
第 2 章 C++入门基础	19
实例 016 计算年份是否为闰年（各种运算符结合）	19
难度指数 ★★☆☆☆ 占用时间 ②③③	
实例 017 打印 ASCII 码表	20
难度指数 ★★☆☆☆ 占用时间 ②③③	



实例 018	求完数.....	21
	难度指数 ★★☆☆☆ 占用时间 ②②②	
实例 019	密码验证 (if...else)	23
	难度指数 ★☆☆☆☆ 占用时间 ②②②	
实例 020	图书管理 (if...else if...else)	24
	难度指数 ★★☆☆☆ 占用时间 ②②②	
实例 021	信号灯 (++--)	25
	难度指数 ★☆☆☆☆ 占用时间 ②②②	
实例 022	简单地获取变量的字节大小 (sizeof)	26
	难度指数 ★☆☆☆☆ 占用时间 ②②②	
实例 023	求余下的物品数 (%)	27
	难度指数 ★☆☆☆☆ 占用时间 ②②②	
实例 024	输入符合条件的整数 (跳出循环)	27
	难度指数 ★☆☆☆☆ 占用时间 ②②②	
实例 025	命令响应 (开关)	28
	难度指数 ★★☆☆☆ 占用时间 ②②②	
实例 026	买水果小游戏 (开关结构综合)	30
	难度指数 ★★★★★ 占用时间 ②②②	
实例 027	各类常量的使用示例 (各种常量)	32
	难度指数 ★★☆☆☆ 占用时间 ②②②	
实例 028	用循环计算 9 的 9 次方 (for)	33
	难度指数 ★★☆☆☆ 占用时间 ②②②	
实例 029	寻找出口小游戏 (do while)	34
	难度指数 ★★☆☆☆ 占用时间 ②②②	
实例 030	一个简单的数据统计系统 (综合)	35
	难度指数 ★★★★★ 占用时间 ②②②	
实例 031	投骰子游戏 (随机)	36
	难度指数 ★★★★★ 占用时间 ②②②	
第 3 章	数组.....	39
实例 032	推箱子 (数组元素移动)	39
	难度指数 ★★★★★ 占用时间 ②②②	
实例 033	数据复制 (复制一组数组到另一组数组)	40
	难度指数 ★★☆☆☆ 占用时间 ②②②	
实例 034	内存输出 (打印数据)	41
	难度指数 ★☆☆☆☆ 占用时间 ②②②	
实例 035	一维数组的应用.....	42
	难度指数 ★☆☆☆☆ 占用时间 ②②②	
实例 036	整数从大到小排序 (比较法)	42
	难度指数 ★★☆☆☆ 占用时间 ②②②	
实例 037	查找二维坐标点.....	43
	难度指数 ★★☆☆☆ 占用时间 ②②②	
实例 038	查找矩阵最大的元素.....	44
	难度指数 ★★☆☆☆ 占用时间 ②②②	



实例 039	二分法查找.....	45
	难度指数 ★★☆☆☆ 占用时间 ⑦①①	
实例 040	查找三维坐标点.....	46
	难度指数 ★★☆☆☆ 占用时间 ⑦①①	
实例 041	获取数组大小 (sizeof)	47
	难度指数 ★☆☆☆☆ 占用时间 ⑦①①	
实例 042	按位数排列.....	48
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 043	统计班上男生和女生的人数 (数组随机访问)	49
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 044	内存指令表 (数组+开关)	51
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 045	模拟栈空间 (数组+算法)	52
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 046	同学姓名册 (字符数组)	53
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 047	图书管理系统 (字符数组综合)	55
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 048	约瑟问题 (把异教徒投入海中排法)	58
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 049	数组转置.....	60
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
第 4 章	C++字符串.....	62
实例 050	输出字符串的每个字符 (for 访问数组)	62
	难度指数 ★☆☆☆☆ 占用时间 ⑦①①	
实例 051	循环改写一段字符串 (for 访问数组)	63
	难度指数 ★☆☆☆☆ 占用时间 ⑦①①	
实例 052	把一个字符串截断 (\0)	63
	难度指数 ★☆☆☆☆ 占用时间 ⑦①①	
实例 053	使用 getchar()函数吸收缓冲区垃圾.....	64
	难度指数 ★☆☆☆☆ 占用时间 ⑦①①	
实例 054	字符串输入 (getline())	65
	难度指数 ★☆☆☆☆ 占用时间 ⑦①①	
实例 055	复制一个字符串 (strcpy())	65
	难度指数 ★☆☆☆☆ 占用时间 ⑦①①	
实例 056	获得字符串长度 (strlen())	66
	难度指数 ★☆☆☆☆ 占用时间 ⑦①①	
实例 057	字符串的比较 (strcmp())	67
	难度指数 ★★☆☆☆ 占用时间 ⑦①①	
实例 058	连接两个字符串 (strcat())	68
	难度指数 ★☆☆☆☆ 占用时间 ⑦①①	
实例 059	将小写字母转换为大写 (strupr())	69
	难度指数 ★☆☆☆☆ 占用时间 ⑦①①	
实例 060	使用 C++字符串类 string 打印字符串.....	69
	难度指数 ★☆☆☆☆ 占用时间 ⑦①①	



实例 061	string 与 C 风格字符串转换	70
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 062	比较两个 string 字符串	71
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 063	查找 string 的某个元素	72
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 064	使用成员函数检测 string 字符串是否非空	73
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 065	获取 string 字符串的长度	74
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 066	提取 string 字符串的子串	74
	难度指数 ★☆☆☆☆ 占用时间 ②○○	
实例 067	把两个 string 字符串相加	75
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 068	string 字符串与 C 风格字符串相加	76
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 069	string 字符串与单字符相加	77
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 070	string 字符串替换	78
	难度指数 ★★☆☆☆ 占用时间 ②○○	
第 5 章	内存与指针	79
实例 071	坐标指针（数组+指针）	79
	难度指数 ★★☆☆☆ 占用时间 ②②○	
实例 072	强制修改常量的值	80
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 073	通信录（动态申请内存）	81
	难度指数 ★★☆☆☆ 占用时间 ②②○	
实例 074	万能箱子（void*）	83
	难度指数 ★★☆☆☆ 占用时间 ②②○	
实例 075	指向结构体变量的指针	84
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 076	打印内存数据（char 打印 1 字节）	85
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 077	错误地释放指针导致程序崩溃	86
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 078	防止野指针的代码	87
	难度指数 ★★☆☆☆ 占用时间 ②○○	
实例 079	统计数据（综合）	88
	难度指数 ★★☆☆☆ 占用时间 ②②○	
实例 080	指针应用常见问题（传送的是地址还是值）	89
	难度指数 ★★☆☆☆ 占用时间 ②②○	
实例 081	将 A 段内存复制到 B 段内存（指针内存复制）	90
	难度指数 ★★☆☆☆ 占用时间 ②②○	



实例 082	将内存的数据倒转过来（指针内存复制+算法）	91
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 083	将数据隐藏于内存（自定义数据访问规则）	92
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 084	输出本机内存数据排列顺序（高端先存还是低端先存）	93
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 085	寻找地址（指针加减法）	94
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 086	利用指针删除数组中的指定元素（指针移动）	95
	难度指数 ★★☆☆☆ 占用时间 ②②①	
第 6 章	函数	96
实例 087	格式打印（设计函数）	96
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 088	指令接收器（字符串形参）	98
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 089	汽车行驶里程（函数实现）	99
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 090	求班级男女生的人数	100
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 091	定义函数求 N 的 N 次方	101
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 092	内存整理（函数实现把 0 内存删除）	102
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 093	分水果（使函数一次性返回 N 个值）	103
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 094	图书名整理系统（按开头字母重新排列）	104
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 095	姓名测试（根据首字母开头+算法）	106
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 096	宝宝改名（函数参数直接引用变量（形参引用））	107
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 097	求最长字符串	108
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 098	补充代码并保证变量 A 的值等于 10	110
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 099	头文件重定义错误案例	110
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 100	更简便的命令解释器（函数重载）	111
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 101	函数重载陷阱案例	113
	难度指数 ★★☆☆☆ 占用时间 ②②①	
实例 102	main()后执行代码	114
	难度指数 ★☆☆☆☆ 占用时间 ②②①	



实例 103	阶乘计算 1 到 100 的积（递归）	115
	难度指数 ★☆☆☆☆ 占用时间 ⑦○○	
实例 104	验证码（函数实现）	116
	难度指数 ★★★★★ 占用时间 ⑦⑦○	
实例 105	DOS 命令解释器（使 main 函数接收参数）	118
	难度指数 ★☆☆☆☆ 占用时间 ⑦○○	
实例 106	补充代码使输出结果成立	119
	难度指数 ★☆☆☆☆ 占用时间 ⑦○○	
实例 107	互动式程序的基本框架	120
	难度指数 ★★★★★ 占用时间 ⑦⑦○	
实例 108	设计一个数据查询系统	121
	难度指数 ★★★★★ 占用时间 ⑦⑦○	
实例 109	学生成绩统计	124
	难度指数 ★★★★★ 占用时间 ⑦⑦○	
第 7 章 C++类基本应用		126
实例 110	产量统计（计算 A 村各类农作物的产量）	126
	难度指数 ★☆☆☆☆ 占用时间 ⑦⑦○	
实例 111	乡村生产总值（同类对象数据统计）	127
	难度指数 ★☆☆☆☆ 占用时间 ⑦⑦○	
实例 112	求圆的面积和周长	129
	难度指数 ★☆☆☆☆ 占用时间 ⑦⑦○	
实例 113	动物对象进化（继承）	130
	难度指数 ★★★★★ 占用时间 ⑦⑦○	
实例 114	员工月薪发放（多态）	131
	难度指数 ★★★★★ 占用时间 ⑦⑦○	
实例 115	家族性格遗传（纯虚函数）	132
	难度指数 ★★★★★ 占用时间 ⑦⑦○	
实例 116	比谁跑得快（类+算法）	134
	难度指数 ★★★★★ 占用时间 ⑦⑦○	
实例 117	错误的模糊引用（类继承问题）	135
	难度指数 ★★★★★ 占用时间 ⑦⑦○	
实例 118	实现类自动化管理内存	136
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 119	入学登记系统（类+算法+综合）	137
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 120	矩形范围（判断一个点是否超出矩形范围）	140
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 121	学生的假期生活（接口）	141
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 122	判断一个矩形是否成立	143
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 123	类的静态成员变量应用（对象间数据共享）	145
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 124	获取系统时间	147
	难度指数 ★★★★★ 占用时间 ⑦⑦○	



实例 125	内联函数应用于计算两点间的距离	148
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 126	this 指针的应用	150
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 127	复制构造函数的应用（复制矩阵）	151
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 128	走出迷宫（类+算法）	153
	难度指数 ★★★★★ 占用时间 ②③④⑤	

第 2 篇 C++进阶案例

第 8 章	泛型编程技术	156
实例 129	绕过形参限制（最简单的模板例程）	156
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 130	万能计算器（支持各类数据的加法函数）	157
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 131	输出浮点型数据和整型数据（隐式和显式实例化）	157
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 132	使用模板特化判断结构体的最大值	158
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 133	模板函数的重载例程	159
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 134	补充代码使输出结果成立	160
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 135	求 $N \times N$ 的值	161
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 136	判断参数为字符串类型就输出字符串	162
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 137	求 AB 对象的和（类参数）	163
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 138	输出内存区域的各种类型数据（void*）	164
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 139	变幻的对象——使用 template 定义一个类模板	165
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 140	分离类模板的声明和定义（求最大值）	166
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 141	类模板含有多个类型参数	167
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 142	类模板的静态成员变量	168
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 143	应用类模板的静态函数	169
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 144	类模板的友元应用	171
	难度指数 ★★☆☆☆ 占用时间 ②③④	



实例 145	类模板的继承.....	172
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 146	使用 STL 库创建容器.....	173
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 147	打印容器元素的值.....	175
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 148	队列镜像.....	176
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 149	获取队列头尾.....	177
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 150	插队（在容器中部插入元素）.....	178
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 151	裁员计划——获取容器元素的个数、删除和清空容器元素.....	179
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 152	图书印刷——复制元素并自动输出.....	180
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 153	利用容器适配器实现栈功能.....	181
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
第 9 章 C++输入/输出系统.....		183
实例 154	使用流类库输出一个文件.....	183
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 155	读写二进制文件.....	184
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 156	读写记事本.....	185
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 157	如何获得文件长度.....	186
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 158	移动文件指针在文件中写入数据.....	186
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 159	输出高精度浮点数（cout 高级应用案例）.....	187
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 160	使用 get 和 getline 函数读取 C 风格字符串.....	188
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 161	读取流状态.....	189
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 162	设置状态字.....	190
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 163	设置输出域宽.....	191
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 164	设计一个简单的学生数据库类.....	191
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 165	实现程序退出自动保存数据库内容到磁盘文件.....	194
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	



实例 166	实现程序启动时自动读取数据库.....	195
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 167	开发一个完整的学生数据管理系统 V1.0.....	196
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 168	开发一个完整的学生数据管理系统 V2.0.....	200
	难度指数 ★★★★★ 占用时间 ②③④	
第 10 章	各类经典案例与解决方法.....	204
实例 169	错误释放指针导致程序崩溃.....	204
	难度指数 ★★☆☆☆ 占用时间 ②④⑤	
实例 170	栈溢出的经典案例.....	205
	难度指数 ★★☆☆☆ 占用时间 ②④⑤	
实例 171	判断语句经典错误案例 (if).....	206
	难度指数 ★★☆☆☆ 占用时间 ②④⑤	
实例 172	使用指针引用问题.....	207
	难度指数 ★★☆☆☆ 占用时间 ②④⑤	
实例 173	显式调用析构函数案例.....	208
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 174	cin 输入队列错误案例.....	209
	难度指数 ★★☆☆☆ 占用时间 ②④⑤	
实例 175	数组越界访问案例.....	210
	难度指数 ★★☆☆☆ 占用时间 ②④⑤	
实例 176	sizeof 产生错误实例.....	211
	难度指数 ★★☆☆☆ 占用时间 ②④⑤	
实例 177	使用类自动管理指针.....	212
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 178	自定义 DLL 库导出函数.....	213
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 179	调用 DLL 导出函数.....	214
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 180	释放字符串常量内存错误案例.....	214
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 181	隐式转换错误案例.....	215
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 182	指示灯颜色 (static 变量).....	216
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 183	编写一个堆内存管理类.....	218
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 184	超出作用域错误案例.....	219
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 185	作用域的相互屏蔽例程.....	219
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 186	使用数组名作为函数参数.....	220
	难度指数 ★★☆☆☆ 占用时间 ②④⑤	
实例 187	让函数一次返回多个值.....	221
	难度指数 ★★☆☆☆ 占用时间 ②④⑤	



实例 188	数组错用 sizeof 案例	222
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 189	类型改名——使用 typedef 定义类型	223
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 190	错误检查——使用 assert 宏进行检测	223
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 191	使用 exit() 函数结束程序	224
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 192	程序异常退出（使用 abort() 函数进行异常退出）	225
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 193	自定义异常对象	226
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 194	使用 set_terminate() 函数设置 terminate() 函数指针	227
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 195	auto_ptr 类智能指针	228
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 196	auto_ptr 智能指针指向的内存类型	228
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	

第 3 篇 C++高级案例

第 11 章	C++高级应用例程	230
实例 197	用 C++实现定时器功能	230
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦〇	
实例 198	使用 atoi() 函数把字符串转换为整数	231
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 199	使用 itoa() 函数把整数转换为字符串	232
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 200	编写一个屏幕小时钟程序	232
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦〇	
实例 201	使用 system() 函数使屏幕停止	233
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 202	屏幕变色效果——使用 system() 函数改变屏幕颜色	234
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 203	清空屏幕——清屏的实现	234
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 204	七彩文字——改变文字色	235
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 205	屏幕背景闪动效果的实现	236
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦〇	
实例 206	文字闪动效果的实现	237
	难度指数 ★★☆☆☆ 占用时间 ⑦〇〇	
实例 207	定时关机	238
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦〇	



实例 208	设置 Win32 窗口	239
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 209	设计一个动态指令接收程序	241
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 210	编写指令响应程序	241
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 211	自定义函数生成一段随机数据	242
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 212	一个简单加密算法的实现	244
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 213	解密算法的实现	245
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 214	模拟打字软件	246
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 215	计算算法耗时	247
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 216	插入排序算法	248
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 217	冒泡排序	250
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 218	选择排序法	251
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 219	猜数字	252
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 220	数字小写变大写	254
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 221	计算三位数字的水仙花数	255
	难度指数 ★★☆☆☆ 占用时间 ⑦⑦⑦	
实例 222	杨辉三角形示例	255
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 223	剪刀石头布单机版小游戏	257
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 224	编写一个进制数转换器	261
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 225	建立链表	263
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 226	插入元素到链表	264
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
第 12 章	Socket 网络及进程间通信	266
实例 227	网络客户端开发 (TCP)	266
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 228	网络服务器端开发 (TCP)	267
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 229	网络服务器端开发 (UDP)	268
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	



实例 230	网络客户端开发 (UDP)	270
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 231	Windows 剪贴板通信之 A 端	271
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 232	Windows 剪贴板通信之 B 端	272
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 233	邮箱通信之 A 端	272
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 234	邮箱通信之 B 端	273
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 235	命名管道之客户端	274
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 236	命名管道之服务器	276
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 237	匿名管道通信之父进程	278
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 238	匿名管道通信之子进程	280
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 239	基于 TCP 的木马程序——服务器端	281
	难度指数 ★★★★★★ 占用时间 ⑦⑦⑦	
实例 240	基于 TCP 的木马程序——客户端	282
	难度指数 ★★★★★★ 占用时间 ⑦⑦⑦	
第 13 章	算法	284
实例 241	反转整数 (%)	284
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 242	古典问题——兔子繁殖	285
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 243	逆时针旋转方阵 90°	285
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 244	判断回文字符串	287
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 245	求最大公约和最小公倍数	288
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 246	图形输出算法	289
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 247	八皇后位置放置问题	290
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 248	百鸡百钱问题	291
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 249	求被 3 整除的数 (%+算法)	292
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	
实例 250	鸡兔同笼问题	293
	难度指数 ★★★★★☆ 占用时间 ⑦⑦⑦	



实例 251	求素数.....	294
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 252	0-1 背包问题（古老数学问题）.....	295
	难度指数 ★★★★★☆ 占用时间 ②③④	
实例 253	扫雷游戏 1.....	297
	难度指数 ★★★★★☆ 占用时间 ②③④	
实例 254	扫雷游戏 2.....	299
	难度指数 ★★★★★☆ 占用时间 ②③④	
实例 255	因式分解.....	301
	难度指数 ★★★★★☆ 占用时间 ②③④	
实例 256	爱因斯坦台阶问题.....	302
	难度指数 ★★★★★☆ 占用时间 ②③④	
实例 257	巧算 24 点问题.....	303
	难度指数 ★★★★★★ 占用时间 ②③④	
第 14 章	多线程、动态链接库.....	306
实例 258	创建多线程.....	306
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 259	设置线程的优先级.....	307
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 260	悬挂和恢复线程.....	308
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 261	利用临界区实现线程同步.....	309
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 262	预防单个线程霸占资源.....	311
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 263	利用事件实现线程同步.....	312
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 264	解析事件实现线程同步的原理.....	313
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 265	利用互斥量实现线程同步.....	314
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 266	利用信号量实现线程同步.....	315
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 267	自定义消息实现线程间通信.....	317
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 268	利用_declspec(dllexport)导出类.....	319
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 269	调用_declspec(dllexport)导出类.....	320
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 270	利用.def 文件导出函数.....	320
	难度指数 ★★☆☆☆ 占用时间 ②③④	
实例 271	隐式调用.def 导出的函数.....	321
	难度指数 ★★☆☆☆ 占用时间 ②③④	



实例 272	显式调用.def 导出函数问题.....	322
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 273	对应显式调用解决方法.....	323
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
第 15 章	数字图像处理专题.....	324
实例 274	载入并显示图像.....	324
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 275	图像灰度化.....	325
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 276	图像灰度均衡化.....	326
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 277	自适应化获取图像二值化阈值.....	328
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 278	二值化源图像.....	330
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 279	保存目标图像.....	331
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 280	去除图像噪声（形态学开运算）.....	332
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 281	去除图像噪声（形态学闭运算）.....	333
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 282	获取图像内物体轮廓（Canny 检测）.....	334
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 283	物体轮廓直线化（Hough 变换）.....	336
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 284	绘制图像灰度直方图.....	337
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 285	缩放图像.....	339
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 286	图像格式转换.....	340
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 287	播放视频.....	342
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
第 16 章	三维仿真技术专题.....	344
实例 288	OSG 语言应用的环境设置.....	344
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 289	加载和显示三维资源.....	345
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 290	绘制长方体.....	345
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	
实例 291	模型贴图.....	346
	难度指数 ★★★★★ 占用时间 ⑦⑦⑦	



实例 292	Shader 着色器.....	347
	难度指数 ★★★★★☆ 占用时间 ②③③	
实例 293	虚拟场景漫游.....	348
	难度指数 ★★★★★☆ 占用时间 ②②②	
实例 294	HUD 应用（显示二维文字）.....	351
	难度指数 ★★★★★☆ 占用时间 ②②②	
实例 295	显示三维文字.....	354
	难度指数 ★★★★★☆ 占用时间 ②②③	
实例 296	添加光源.....	355
	难度指数 ★★★★★☆ 占用时间 ②②③	
实例 297	缩放模型.....	356
	难度指数 ★★★★★☆ 占用时间 ②②③	
实例 298	利用粒子系统制作火焰.....	358
	难度指数 ★★★★★☆ 占用时间 ②②②	
实例 299	模拟雾效.....	360
	难度指数 ★★★★★☆ 占用时间 ②②②	
实例 300	响应回调事件.....	361
	难度指数 ★★★★★☆ 占用时间 ②②②	

第 1 篇 C++入门案例

第 1 章 从最简单的案例讲述 C++

C++发展自 C 语言，但更为成熟，并且是一种面向对象的编程语言。两者的最大区别在于，C 语言处理面向过程的开发应用，而 C++语言应用于面向对象的开发。

本章重点介绍 C++的基础知识，包括变量、类型转换、运算符、优先级。对于这些基础知识的应用，都各自给出实例代码及运行效果，有助于读者更直观地学习。在最后一个例子中，给出了对于结构和算法的一个基础性演示。



实例 001 在屏幕上输出内容

【实例描述】

在标识某件事物的好坏时，必须给出明确提示（以文字或图画的形式），C++语言的应用也不例外。学习 C++语言，首先要学习如何在屏幕上输出字符。C++语言的输出语句不同于 C 语言，它是由 `cout` 实现的。本实例在屏幕上显示不同数据类型的输出。数据类型一般可以分为常量和变量，该实例分别演示如何输出常量、变量以及转义字符，效果如图 1-1 所示。

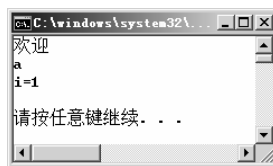


图 1-1 欢迎界面

【实现过程】

本着由浅入深的讲解原则，该实例先输出常量，再输出一个整型变量，最后输出一个转义字符 `\n`。其代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int i = 1;           //整型变量
07     cout<<"欢迎"<<endl;  //输出一个常量字符串
08     cout<<'a'<<endl;    //输出一个常量字符
09     cout<<"i="<<i<<endl; //输出一个整型变量
10     cout<<"\n";         //输出一个表示换行的转义字符
11     return 0;
12 }
```

【代码解析】

其中，`<<`与 `cout` 匹配使用。即 `cout` 输出语句后必须至少有一个 `<<`运算符出现，并且不能用 `>>`运算符代替。第 07 行的 `cout` 表示在输出界面输出常量字符串“欢迎”，需要用双引号。第 08 行是输出一个常量字符，所以只需要一对单引号。它们的区别由字符串与字符的对比知识可知。第 09 行是输出一个整型变量，可知输出变量不需要双引号或者单引号，直接跟在运算符 `<<`之后



即可。第 10 行是输出一个转义字符，因为属于字符，所以也是用单引号。'\n'的意思表示换行。



注意：如果没有第 02 行，第 07 行必须改为 `std::cout<<"欢迎"<<std::endl;`，否则会报错。



实例 002 规格不同的箱子（变量）

【实例描述】

C++语言的数据类型一般分为常量和变量，本实例主要讲解变量的类型。变量有很多种，如整型、浮点型、布尔型等。根据变量的特点，这些数据可以被比作箱子。变量有多种，箱子的规格也有不同类型，箱子也有被限制存放的数量。本实例主要介绍整型和浮点型变量的表示关键字，并给出它们各自的界值，运行效果如图 1-2 所示。

【实现过程】

整型数据类型分为基本整型（用 `int` 表示）、短整型（用 `short` 表示）、长整型（用 `long` 表示）。浮点型数据分为单精度型（用 `float` 表示）、双精度型（用 `double` 表示）、扩展双精度型（用 `long double` 表示）。在声明各种类型的变量后，用 `cout` 语句输出它们的表示范围，代码如下：

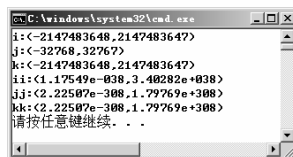


图 1-2 不同类型的变量界值

```
01 #include <iostream>
02 #include <climits>
03 #include <cstdio>
04 using namespace std;
05
06 int main()
07 {
08     int i; //基本整型
09     short j; //短整型
10     long k; //长整型
11     float ii; //单精度型
12     double jj; //双精度型
13     long double kk; //扩展双精度型
14
15     cout<<"i: ("<<INT_MIN<<","<<INT_MAX<<") "<<endl; //基本整型
16     cout<<"j: ("<<SHRT_MIN<<","<<SHRT_MAX<<") "<<endl; //短整型
17     cout<<"k: ("<<LONG_MIN<<","<<LONG_MAX<<") "<<endl; //长整型
18     cout<<"ii: ("<<FLT_MIN<<","<<FLT_MAX<<") "<<endl; //单精度型
19     cout<<"jj: ("<<DBL_MIN<<","<<DBL_MAX<<") "<<endl; //双精度型
20     cout<<"kk: ("<<LDBL_MIN<<","<<LDBL_MAX<<") "<<endl; //扩展双精度型
21
22     return 0;
23 }
```

【代码解析】

第 02 行是包含整型数据范围的头文件，必须有。第 03 行是包含浮点型数据范围的头文件，必须有。第 08~13 行用于声明不同类型的数据，第 15~20 行用于输出对应已声明数据类型变量的表示范围。其中，`INT_MIN` 表示基本整型数据的最小值，`INT_MAX` 表示基本整型数据的



最大值。其他值的表示意义依此类推，在此不再赘述。



注意：第 02~03 行的头文件必须包含，否则会报错。这两个头文件预定义了各种数据类型的最大值与最小值变量。



实例 003 物品存放（变量）

【实例描述】

由实例 002 可知，变量可以被比作箱子，箱子被用来存储物品。类似的，变量被用来存储数据。本例利用赋值运算符完成数据的存储，即物品的存放，实例运行效果如图 1-3 所示。

【实现过程】

物品需要被存放于箱子中，而在 C++ 语言中，常量数据需要被存放到内存中。计算机的内存按字节为单位被分为单个区域，这些区域可被不同的数据类型标识。以下代码是在内存中声明一个基本整型数据 i、一个短整型数据 j、一个长整型数据 k、一个单精度浮点型数据 ii、一个双精度浮点型数据 jj 和一个扩展双精度浮点型数据 kk。对这些数据进行初始化，即在这些箱子中存放限制范围内的物品，需要用到赋值运算符，具体代码如下：



图 1-3 赋值运算符应用

```
01  #include <iostream>
02  using namespace std;
03
04  int main()
05  {
06      int i = 1;                //基本整型
07      short j = 2;              //短整型
08      long k = 3;               //长整型
09      float ii = 1.1;           //单精度型
10      double jj = 1.234;        //双精度型
11      long double kk = 1.2342546; //扩展双精度型
12
13      cout<<"i="<<i<<endl;
14      cout<<"j="<<j<<endl;
15      cout<<"k="<<k<<endl;
16      cout<<"ii="<<ii<<endl;
17      cout<<"jj="<<jj<<endl;
18      cout<<"kk="<<kk<<endl;
19
20      return 0;
21  }
```

【代码解析】

第 06~11 行用来表示所声明的各类箱子，并对其进行初始化，即存放相应数量的物品。第 13~18 行用于输出各类箱子中所存放物品的数量值。



实例 004 交换物品（变量）

【实例描述】

对于相同类型的物品分别存放于不同编号的箱子，当需要交换不同箱子中的物品时，需要第三个箱子的参与方可完成。本实例要实现如何利用一个临时变量交换两个变量之间的数据。本实例的运行效果如图 1-4 所示。

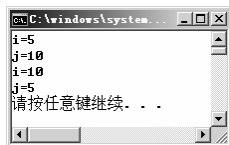


图 1-4 利用临时变量交换物品

【实现过程】

首先定义三个整型变量 `i`、`j`、`temp`。其中，变量 `i` 和 `j` 分别代表两个箱子，变量 `temp` 代表一个临时箱子。在实现交换的过程中，需要将其中一个变量（比如 `i`）的数值赋给临时变量 `temp`，然后将另一个变量 `j` 的数据赋给变量 `i`，最后将临时变量 `temp` 的数据赋给变量 `j`，完成物品的交换。具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int i;           //整型变量 i
07     int j;           //整型变量 j
08     int temp;        //临时变量
09
10     i=5;
11     j=10;
12     cout<<"i="<<i<<endl<<"j="<<j<<endl;    //交换前
13
14     //交换
15     temp = i;
16     i = j;
17     j = temp;
18
19     cout<<"i="<<i<<endl<<"j="<<j<<endl;    //交换后
20     return 0;
21 }
```

【代码解析】

第 08 行定义的临时变量 `temp` 起临时存放的作用，如果没有该变量，相当于没有第 15、17 行，结果只完成了赋值操作（即第 16 行），没有实现物品的交换。第 12 行与第 19 行的输出语句用于对比交换前后的变量数值，以验证是否完成交换。



注意：临时变量起一个缓冲性的作用，用于临时放置交换变量的其中一方。



实例 005 消失的重量（隐式类型转换）

【实例描述】

在进行简单的加减法运算时，较合理的编程规则是保持加减运算符两端的数据类型一致，



否则容易出现数据的缺失。以加法为例，变量 $k = \text{变量 } i + \text{变量 } j$ 。如果赋值运算符左边的数据类型不同于赋值运算符右边的，变量 k 的值会按照其数据类型确定。即将赋值运算符右边的数据类型转换为左边的，这个过程被称为隐式类型转换。本例以实际的加法运算给出具体的实现过程，运行效果如图 1-5 所示。



图 1-5 隐式类型转换

【实现过程】

该例旨在演示隐式数据类型转换，用到一个 `float` 型的变量 `i`（用于表示重量 1）、一个 `int` 型的变量 `j`（用于表示重量 2）。这两个变量的和赋给 `int` 型变量 `sum`（表示总重量）。该实例的实现代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     float i;           //浮点型数值-重量 1
07     int j;             //整型数值-重量 2
08     int sum;           //总重量
09
10     i = 0.002;
11     j = 2;
12     sum = i+j;         //计算总重量
13     cout<<"sum="<<sum<<endl; //输出
14
15     return 0;
16 }
```

【代码解析】

第 10 行给 `float` 型变量 `i` 赋值为 0.002，第 11 行赋值整型变量 `j` 为 2。第 12 行的赋值运算符左边是一个整型变量 `sum`，右边是一个加法等式，等式的两个变量类型不同，但是计算结果遵循精度较高的数据类型，所以 `i+j` 的值为 2.002。将 2.002 赋给整型变量 `sum`，必须将小数点后的位数舍去，才能被存入整型变量。所以 `sum` 的值为 2。

由此可见，总的重量减少了，有些重量消失了。在实际应用中，可以输入以下代码检测 `i+j` 的值是否为 2.002。

```
cout<<i+j<<endl;
```



注意：被赋值的变量值必须符合其数据类型的属性特征。如果一个 `float` 型变量小数点后值为 0，也需要用 0 补齐，否则被视为整型数据。



实例 006 游泳池的容量

【实例描述】

由实例 002 可知，不同数据类型都有各自的表示范围。如果超出范围，被视为越界，程序在运行的过程中会出现问题。产生问题的原因有多种，这里要讲的是原先的数据类型不再适用，需要更换表示范围更大的数据类型。如同游泳池已被加满水，如果继续添加，水会溢出。该例



的运行效果如图 1-6 所示。

【实现过程】

为了简单直观地表示应用概念，此处结合运算符<完成例子的实现。设定游泳池的容量为 100 升，整型变量 Pool_volume 表示当前给游泳池加水的容量。如果 Pool_volume 小于 100 升，可以继续加水。否则，程序显示游泳池的水已满，发出停止加水的信号。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int Pool_volume;                //游泳池容量
07     int count = 10;                //循环次数
08     while(count!=0)
09     {
10         cin>>Pool_volume;          //输入数据
11         if(Pool_volume < 100)      //如果小于 100 升，池中水没溢出
12             cout<<"游泳池中水没溢出，可以继续加水"<<endl;
13         else
14             cout<<"游泳池中水已满，必须停止加水或者加入另一个游泳池"<<endl;
15         count --;
16     }
17
18     return 0;
19 }
```

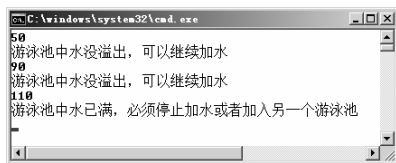


图 1-6 游泳池的容量

【代码解析】

第 07 行的变量 count 表示加水次数的限制，用 while 循环判断（如第 08 行），进入下一步的条件表达式由第 15 行给出。每循环一次，用户在界面上输入当前游泳池的水容量。第 11~14 行的 if...else 结构判断当前游泳池的水是否溢出。对应各自的状况，在屏幕上输出不同的结果。



实例 007 显式转换（类型转换）

【实例描述】

在实例 005 中提到隐式类型转换，此处讲解数据类型的显式转换。显式类型转换的一般形式如下：

（转换后的类型）变量；

其中，变量可以是一个表达式计算后的结果。此时，就需要用括号将表达式括起来。本例以被赋值后的变量结果演示显式类型转换的效果，其运行效果如图 1-7 所示。

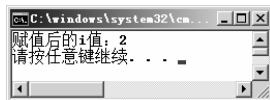


图 1-7 显式类型转换效果

【实现过程】

程序显示数据类型的显式转换，首先定义两个数据变量，分别为整型 i 和单精度浮点型 j。按照之前介绍的显式类型转换编码格式书写代码如下：



```

01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int i = 1;                //整型
07     float j = 2.234;         //浮点型
08
09     i = (int)j;               //显式类型转换
10     cout<<"赋值后的 i 值: "<<i<<endl;
11
12     return 0;
13 }

```

【代码解析】

本例的重点是第 09 行, 对 float 型变量 j 做显式数据变换为 int 型数据类型, 因此在赋值后, 变量 i 的值变为 2。



注意: 相比隐式类型转换, 显式类型转换对于代码的理解更有帮助, 并且在程序出现 bug 的情况下, 更易找出代码中存在的错误。



实例 008 单片机应用（位操作）

【实例描述】

在单片机应用中, 位操作符经常被用到。C++语言中的位操作符有很多, 包括&（按位与）、|（按位或）、^（按位异或）、~（按位求反）、>>（右移）、<<（左移）。本例以单片机应用为例, 介绍位操作符的实际操作, 效果如图 1-8 所示。

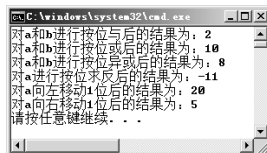


图 1-8 单片机应用位操作

【实现过程】

首先需要定义 3 个整型变量, 分别命名为 a、b、c。变量 c 用于存储计算结果, 变量 a 和 b 用于位操作的操作数。其中按位与、或、异或都需要两个变量参与, 而按位求反、左移和右移只作用于 1 个操作数。当计算完成后, 用 cout 语句输出计算结果, 该实例的具体代码如下:

```

01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int a = 10;
07     int b = 2;
08     int c;
09     c = a & b;                //按位与
10     cout<<"对 a 和 b 进行按位与后的结果为: "<<c<<endl;
11     c = a | b;                //按位或
12     cout<<"对 a 和 b 进行按位或后的结果为: "<<c<<endl;
13     c = a ^ b;                //按位异或
14     cout<<"对 a 和 b 进行按位异或后的结果为: "<<c<<endl;
15     c = ~a;                   //按位求反

```



```

16      cout<<"对 a 进行按位求反后的结果为: "<<c<<endl;
17      c = a<<1;                      //左移 1 位
18      cout<<"对 a 向左移动 1 位后的结果为: "<<c<<endl;
19      c = a>>1;                      //右移 1 位
20      cout<<"对 a 向右移动 1 位后的结果为: "<<c<<endl;
21
22      return 0;
23  }

```

【代码解析】

对变量进行位操作时需要还原变量的二进制码，即要将十进制数转换为二进制数表示。另外，在转换十进制数为二进制数前，需要了解当前使用计算机的整型变量占用字节数。此处的计算机占有 4 字节，1 字节占 8 位，所以一共有 32 位，最高位表示符号位（0 表示正，1 表示负）。因此第 06 行表示变量 a 的值为 10，它的二进制数表示为：0000...1010。第 07 行表示变量 b 值为 2，它的二进制数表示为：0000...0010。

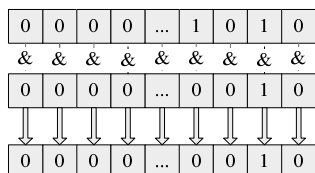


图 1-9 按位与操作原理

第 09 行对变量进行按位与操作的原理是：对两个操作数的对应二进制位进行与操作，即 a 的第 1 位与 b 的第 1 位进行与计算，其他位依此类推。按位与操作的计算过程如图 1-9 所示。

由图 1-9 可知，对变量 a 和 b 进行按位与操作的二进制结果为 0000...0010。之后换算为十进制数表示为 2。同理，第 11 行变量间的按位或操作也是对应位的数据进行或运算，二进制结果表示为 0000...1010，十进制数表示为 10。第 13 行的按位异或操作的二进制数表示为 0000...1000，它的十进制数为 8。

第 15 行对变量 a 进行按位求反操作，它的二进制数结果为 1111...0101。它的最高位为 1，所以结果应该为负数，它的十进制数为-11。负数的十进制数转换为二进制数涉及原码、反码和补码的知识，此处简单介绍一下。比如，-11 的二进制原码是 1000...1011，它的反码为 1111...0100，而它的补码是在反码的基础上加 1，结果为 1111...0101。因此，对于负数的二进制数换算为十进制数可以在上述原理的基础上反推。

第 17 行对变量 a 左移 1 位，低位不足的全部补 0，所以它的二进制数表示为 0000...10100，换算为十进制数为 20。第 19 行对变量 a 右移 1 位，移动后的二进制数表示为 0000...0101，换算为十进制数为 5。

右移的计算原理分为两种：有符号数据和无符号数据。如果有符号数据类型，在右移的过程中，符号位也随之一起移动。在右移前，如果是正数，移动后的最高位补 0。如果是负数，移动后的最高位可以补 0，也可以补 1，其由编译系统决定。



注意：对于有符号的数据，在进行右移操作时，可以先使用变量试验并了解当前的编译系统对于负数的右移操作规律。



实例 009 房间面积 (*)

【实例描述】

运算符*属于 C++众多运算符中的一种，它的作用是将运算符两边的变量相乘，计算规则



相同于数学的乘法运算。该例利用运算符*计算房间的面积，运行效果如图 1-10 所示。

【实现过程】

对于该实例的实现，首先要声明 3 个单精度浮点型变量 width、length 和 area，分别表示房间的宽度、长度和面积。房间的面积计算公式为：

面积=宽度*长度；

该实例的具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     float width;           //房间宽度
07     float length;          //房间长度
08     float area;            //房间面积
09
10     cin>>width>>length;    //依次输入宽度和长度
11     area = width * length;  //计算面积
12
13     cout<<"房间面积为: "<<area<<endl;
14     return 0;
15 }
```

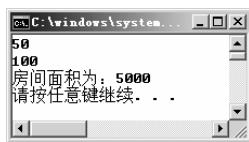


图 1-10 应用*运算符计算房间面积

【代码解析】

第 10 行表示需要用户自行输入房间的宽度和长度值，第 11 行利用乘法运算符*表示面积的计算公式，最后由最第 13 行输出房间的面积值。



实例 010 平分物品 (/)

【实例描述】

在幼儿园里，老师经常会将各种物品平分给小朋友们，旨在教给小朋友们一个与他人分享的良好意识。本实例用于实现如何让幼儿园的老师平分物品，在 C++ 编程中需要用到运算符/，运行效果如图 1-11 所示。

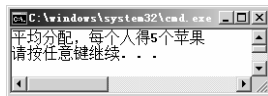


图 1-11 平分物品 (/)

【实现过程】

现有 60 个苹果需要平均分配，而幼儿园中一共有 12 个小朋友，每个小朋友能分多少个苹果？本实例的计算公式如下：

每个人所得的苹果数=共有的苹果数/总共的人数；

代码如下：

```
01 #include <iostream>
02 using namespace std;
03
```



```

04  int main()
05  {
06      int apple_number = 60;                //60 个苹果
07      int people = 12; //12 个人
08      int apple_per_people;                //每个人分多少个苹果
09      if(people != 0)                       //判断除数是否为 0
10      {
11          apple_per_people = apple_number/people; //计算平均数
12          cout<<"平均分配, 每个人得"<<apple_per_people<<"个苹果"<<endl; //输出
13      }
14      else
15          cout<<"除数为 0, 出现异常"<<endl;
16
17      return 0;
18  }

```

【代码解析】

第 06~08 行用于声明除法公式所需的 3 个变量, 分别为被除数、除数、商。第 09 行用于判断除数是否为 0。如果除数不为 0, 第 11 行表示本实例所需的计算公式, 第 12 行输出最后结果。如果除数为 0, 第 15 行输出警告。



注意: 一旦遇到除法运算, 一定要先判断除数是否为 0。在没有判断的情况下就进行运算, 可能会导致程序出错。



实例 011 取余数 (%)

【实例描述】

接着实例 010, 幼儿园给小朋友提供的苹果并不可能全部分配出去, 有的小朋友可能分到的苹果比其他人少。为避免出现如此尴尬的情况, 在分配之前需要做运算, 即本实例涉及的取余运算, 实例效果如图 1-12 所示。

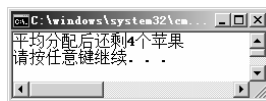


图 1-12 取余数

【实现过程】

本实例模拟的情形有 60 个苹果, 7 个小朋友, 如果 7 个小朋友收到的苹果一样多, 请问还剩几个苹果? 其计算公式如下:

剩余苹果数 = 总共苹果数 % 总人数;

代码如下:

```

01  #include <iostream>
02  using namespace std;
03
04  int main()
05  {
06      int apple_number = 60;                //一共 60 个苹果
07      int people = 7;                       //有 7 个小朋友
08      int apple_remain;                    //平均分后还剩几个苹果

```



```

09     if(people == 0)                                //判断取余数是否为0
10         cout<<"错误"<<endl;
11     else
12     {
13         apple_remain = apple_number % people;        //取余
14         cout<<"平均分配后还剩"<<apple_remain<<"个苹果"<<endl;
15     }
16
17     return 0;
18 }

```

【代码解析】

第 06~08 行声明了 3 个变量，总共的苹果数、总共的人数、剩余苹果数（平均分配后）。第 13 行给出了对应计算公式的编码，第 14 行输出平均分配后还剩多少个苹果。第 09~15 行判断取余数是否为 0，并在对应的情形下做出相应的操作。



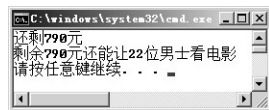
注意：取余运算时也必须判断取余数是否为 0，原理与除法运算一样。



实例 012 公司组织看电影（综合）

【实例描述】

公司预支 1000 元给 8 名员工去看电影，其中 4 男 4 女。电影院实行优惠政策，男士票为全价，而女士票为半价。现已知一张电影票的全价为 35 元，则最后还剩多少钱？所剩的钱最多能让几位男士看电影？本实例运行效果如图 1-13 所示。



【实现过程】

图 1-13 公司组织看电影（综合）

首先声明 8 个变量，sum 用于表示资金总额，male 和 female 分别表示男女员工的人数，ticket_value_male 和 ticket_value_female 分别表示每个男女员工看电影需花多少钱，remain 表示还剩多少钱，male_remain 表示剩余的钱还够几个男员工观看电影，其代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     float sum = 1000.0;                //总额为1000元
07     int male = 4;                      //4名男性员工
08     int female = 4;                   //4名女性员工
09     float ticket_value_male = 35.0;
10     float ticket_value_female;
11     ticket_value_female = ticket_value_male/2;
12     float remain;                      //剩余多少钱
13     int male_remain;
14     remain = sum - (male * ticket_value_male + female * ticket_value_female);
                                           //计算剩余
15     cout<<"还剩"<<remain<<"元"<<endl;
16     male_remain = (int)(remain / ticket_value_male);
                                           //计算剩余钱数可供几个男士看

```



```

17         cout<<"剩余"<<remain<<"元还能让"<<male_remain<<"位男士看电影"<<endl;
18
19         return 0;
20     }

```

【代码解析】

第 06~13 行用于声明本实例所需的 8 个变量，第 11 行给出了每个女员工需付的票价。第 14 行给出了 8 个员工付钱后还剩余多少钱的计算公式。第 16 行给出了还有多少个男士可以看电影的计算方法。因为结果可能不是整数，但人数不能出现小数，所以需要显式类型转换。



注意：对于第 16 行的显式类型转换是不需要的，但为了代码的可读性，显式类型转换比隐式转换具有较高的可读性。



实例 013 称水果（比较）

【实例描述】

平常大家去商店买水果的时候，为了方便计算水果的价格，商家都喜欢将水果的重量四舍五入以凑整数。本实例旨在用比较的方法得到当前水果的重量应该是四舍还是五入，运行效果如图 1-14 所示。

【实现过程】

本实例的实现原理很简单，如果重量的小数部分小于 0.5，需要舍去小数部分；反之，向大的方向凑成整数。比如，水果重量为 23.6 斤，它的小数部分为 0.6，因为大于 0.5，所以最后计算的重量应该为 24 斤，其代码如下：

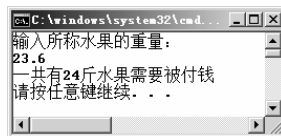


图 1-14 称水果（比较）

```

01 #include <iostream>
02 using namespace std;
03
04 //称水果，四舍五入
05 int main()
06 {
07     float fruit_weight;           //水果的重量
08     int fruit_weight_cal;         //被计算的水果重量
09     cout<<"输入所称水果的重量: "<<endl;
10     cin>>fruit_weight;
11     if((fruit_weight - (int)fruit_weight) < 0.5) //舍去
12         fruit_weight_cal = (int)fruit_weight;
13     else //五入
14         fruit_weight_cal = (int)fruit_weight + 1;
15     cout<<"一共有"<<fruit_weight_cal<<"斤水果需要被付钱"<<endl;
16
17     return 0;
18 }

```

【代码解析】

第 07 行表示实际的水果重量，应该被声明为 float 型。第 08 行表示参与计算的水果重量，



必须为整型。第 10 行输入当前水果的重量。本实例的重点是如何计算重量的小数部分，由第 11 行得出。由显式类型转换得到水果重量的整数部分，然后用水果的实际重量减去其整数部分即得小数部分。第 11~14 行实现了水果重量的四舍五入过程。



实例 014 简单算术（优先级）

【实例描述】

C++中有很多运算符，包括赋值运算符、算术运算符、逻辑运算符、关系运算符。对于这些运算符在参与运算时，并不是按照从左到右开始运算，每个运算符都有各自的优先级。优先级较高的先于优先级低的进入运算。本实例对此进行演示，具体运行效果如图 1-15 所示。

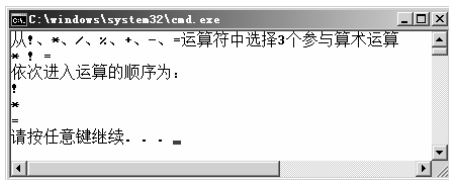


图 1-15 简单算术（优先级）

【实现过程】

本实例利用函数 `int level(char ch)` 判断输入运算符的级别，它的返回值为输入运算符的级别数值。`char symbol[3]` 用于存储输入的运算符，`int value[3]` 用于存储输入运算符的优先级。具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int level(char ch)
05 {
06     switch(ch)
07     {
08         case '!':
09             return 2;
10             break;
11         case '*':
12             return 3;
13             break;
14         case '/':
15             return 3;
16             break;
17         case '%':
18             return 3;
19             break;
20         case '+':
21             return 4;
22             break;
23         case '-':
24             return 4;
25             break;
26         case '=':
27             return 5;
28             break;
29         default:
30             return -1;
31             break;
32     }
33 }
34
```



```
35 int main()
36 {
37     char symbol[3];
38     int value[3];
39     cout<<"从!、*、/、%、+、-、=运算符中选择 3 个参与算术运算"<<endl;
40     cin>>symbol[0]>>symbol[1]>>symbol[2];
41
42     value[0]=level(symbol[0]);
43     value[1]=level(symbol[1]);
44     value[2]=level(symbol[2]);
45
46     if((value[0]==-1) || (value[1]==-1) || (value[2]==-1))//只要有一个返回负数
47         cout<<"库中没有输入的运算符"<<endl;
48     else
49     {
50         //从大到小排序
51         for (int i = 0;i < 3; i++)
52         {
53             for (int j = i; j<3;j++)
54             {
55                 if(value[i]>value[j])                //如果前一个大于后一个，开始交换
56                 {
57                     //交换
58                     char temp;
59                     temp=symbol[i];
60                     symbol[i]=symbol[j];
61                     symbol[j]=temp;
62                 }
63             }
64         }
65         cout<<"依次进入运算的顺序为: "<<endl;
66         for(int k=0;k<3;k++)
67             cout<<symbol[k]<<endl;
68     }
69
70     return 0;
71 }
```

【代码解析】

第 46~68 行用于判断输入的运算符是否为数据库中的元素，其中，第 51~64 行为输入运算符的优先级按从小到大排序，第 66、67 行输出排序后的运算符参与运算先后顺序。



注意：本实例所用的排序方法可以升级为冒泡排序法，可以节约算法时间。



实例 015 输出 Huffman 编码（结构+算法）

【实例描述】

本实例演示如何输出 Huffman 编码。现有一个字符串为 agdfaghdbabsba，需要利用 C++ 编码输出该字符串的 Huffman 编码，实例效果如图 1-16 所示。



图 1-16 输出字符串的 Huffman 编码

【实现过程】

对于本实例的实现，需要结构体的参与。另外，将该功能的实现形成一种现成的算法，以简单地实现 Huffman 编码的获取。在解决本实例前，需要统计字符串中出现字符的频率，现统计如表 1-1 所示。然后，列出字符串的 Huffman 树，如图 1-17 所示。

表 1-1 字符串agdfaghdabsba中各字符出现频率

出现的字符	字符出现的次数
a	4
b	2
d	2
f	1
g	2
h	1
s	1
合计	13

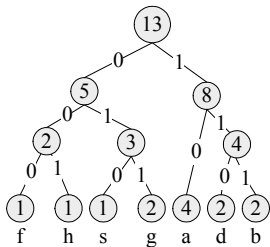


图 1-17 字符串的 Huffman 树

由图 1-17 可知，各字符的编码如表 1-2 所示。

表 1-2 对应字符的编码

字 符	编 码
a	10
b	111
d	110
f	000
g	011
h	001
s	010

所以，整个字符串的编码为：1001111000010011001110101110101110。实现代码分别由 3 部分给出，第一部分是头文件的包含、结构体的定义、函数 coding()的实现，具体如下：

```
01 #include<iostream>
02 #include<string>
03 using namespace std;
04
05 struct huffTree
```



```
06 {
07     int parent;           //父亲
08     int lchild;           //左孩子
09     int rchild;           //右孩子
10     int weight;           //权重
11     string flag;          //标志
12 };
13 struct Lowest_Node        //第 0 级节点的字符与频度
14 {
15     char ch;
16     int ch_num;
17 };
18 void coding(int length, huffTree tree[],int n,int &a,int &b)
19 {
20     int i;
21     int r,s;
22     r=s=length;           //节点个数最大不会超过字符串的长度
23     for(i=0;i<n;i++)
24     {
25         if((tree[i].weight<r)&&(tree[i].parent== -1))
26         {
27             r=tree[i].weight;
28             a=i;
29         }
30     }
31     for(i=0;i<n;i++)
32     {
33         if((tree[i].weight<s)&&(i!=a)&&(tree[i].parent== -1))
34         {
35             s=tree[i].weight;
36             b=i;
37         }
38     }
39 }
```

第二部分代码是本实例的核心函数，`frequency()`的重点是计算每个字符出现的频度，并从大到小排序其频度，具体代码如下：

```
01 void frequency(string str)
02 {
03     int length=str.length();           //长度
04     Lowest_Node *node=new Lowest_Node[length]; //声明最 0 级节点
05
06     int i,j;                           //循环因子
07     for(i=0;i<length;i++)
08         node[i].ch_num=0;               //初始化频度
09
10     int char_type_num=0;                 //初始为 0 种字符
11     for(i=0;i<length;i++)               //循环整个字符串
12     {
13         for(j=0;j<char_type_num;j++)
14             if(str[i]==node[j].ch||(node[j].ch>='a'&&node[j].ch<='z'&&str[i]+32==node[j].ch))
15                 break;                   //该字符没有出现过，跳出循环
16
17         if(j<char_type_num)               //字符重复出现对应计数器加 1
18             node[j].ch_num++;
19         else                               //新出现的字符，记录到 ch[j] 中，对应计数器加 1
20         {
21             if(str[i]>='A'&&str[i]<='Z')
22                 node[j].ch=str[i]+32;
23             else
```



```

24         node[j].ch=str[i];
25         node[j].ch_num++;
26         char_type_num++;           //字符的种类数加1
27     }
28 }
29
30 //按频度从大到小排序
31 for(i=0;i<char_type_num;i++)
32 {
33     for(j=i;j<char_type_num;j++)
34     {
35         if(node[j].ch_num<node[j+1].ch_num)//如果前一个小于后一个, 交换
36         {
37             int temp;                //临时频度
38             char ch_temp;            //临时字符
39             temp=node[j].ch_num;
40             ch_temp=node[j].ch;
41             node[j].ch_num=node[j+1].ch_num;
42             node[j].ch=node[j+1].ch;
43             node[j+1].ch_num=temp;
44             node[j+1].ch=ch_temp;
45         }
46     }
47 }
48
49 for(i=0;i<char_type_num;i++)        //打印字符频度
50     cout<<"字符"<<node[i].ch<<"出现了"<<node[i].ch_num<<"次"<<endl;
51
52 huffTree *huff=new huffTree[2*char_type_num-1];
53                                     //此变量的声明需位于确定 char_type_num 值后
54 huffTree temp;
55 string *code=new string[2*char_type_num-1]; //存放各个字符的编码
56
57 for(i=0;i<2*char_type_num-1;i++)    //节点初始化
58 {
59     huff[i].lchild=-1;
60     huff[i].parent=-1;
61     huff[i].rchild=-1;
62     huff[i].flag=-1;
63 }
64 for(j=0;j<char_type_num;j++)        //将排序后的第0级节点权重赋给树节点
65 {
66     huff[j].weight=node[j].ch_num;
67 }
68 int min1,min2;
69 for(int k=char_type_num;k<2*char_type_num-1;k++) //赋值0级之上的节点
70 {
71     coding(length,huff,k,min1,min2);
72     huff[min1].parent=k;
73     huff[min2].parent=k;
74     huff[min1].flag="0";
75     huff[min2].flag="1";
76     huff[k].lchild=min1;
77     huff[k].rchild=min2;
78     huff[k].weight=huff[min1].weight+huff[min2].weight;
79 }
80 for(i=0;i<char_type_num;i++)
81 {
82     temp=huff[i];
83     while(1)
84     {
85         code[i]=temp.flag+code[i];
86         temp=huff[temp.parent];

```



```
87             if(temp.parent==-1)
88                 break;
89         }
90     }
91     cout<<"字符串的每个字符 huffman 编码为:"<<endl;
92     for(i=0;i<char_type_num;i++)
93         cout<<node[i].ch<<" "<<code[i]<<endl;
94
95     cout<<"整个字符串的 huffman 编码为: "<<endl;
96     for(i=0;i<length;i++)
97     {S
98         for(j=0;j<char_type_num;j++)
99         {
100             if(str[i]==node[j].ch)
101                 cout<<code[j];
102         }
103     }
104
105     //释放内存
106     delete[] node;
107     node=NULL;
108     delete[] huff;
109     huff=NULL;
110     delete[] code;
111     code=NULL;
112 }
```

第三部分代码是本实例的 main()函数实现，代码如下：

```
01 int main()
02 {
03     int length=0;           //字符串长度
04     string str;             //目标字符串
05     cout<<"请输入一个字符串:";
06     cin>>str;
07     frequency(str);         //求各个元素的频度
08
09     return 0;
10 }
```

【代码解析】

在第一部分代码中，第 05~12 行是 Huffman 树的结构体定义，元素包括父节点（parent）、左右子节点（lchild、rchild）、每个节点的权重（weight）和标志量（因为之后要定义该结构的数组，所以有的节点在当前操作下无效）。第 13~17 行定义的结构体 Lowest_Node 用于表示第 0 级节点的字符与频度。函数 coding()（第 18~39 行）用于确定每个字符的 Huffman 编码，输出参数为 a、b。

在第二部分代码中，函数 frequency()的实现标有最详细的注释，此处只给出函数中各个小模块的实现。第 03~28 行是统计字符串中出现字符的种类及其频度，第 31~47 行给第 0 级节点按频度从大到小排序，并由第 49、50 输出结果。第 52~90 行给每个字符做 Huffman 编码，并由第 91~93 行输出。第 95~103 行输出该字符串的 Huffman 编码。



注意：利用 new 为数组申请，必须在最后用 delete 释放。

第 2 章 C++入门基础

本章主要讲解如何实现逻辑判断（if、if...else if...else 等结构）、循环（for、do while 等结构）、如何跳出循环等。本章的例子均采用生活实例作为背景，以期更切实地讲解知识，并且在最后一个实例中，给出了投骰子的小游戏，使学习更有趣味性。



实例 016 计算年份是否为闰年（各种运算符结合）

【实例描述】

众所周知，每年二月份的天数不是固定不变的。如果这一年为闰年，则该年二月的天数是 29 天；如果不是闰年，则为 28 天。因此，如何判断当年是否为闰年，有一个判断标准，即只要符合以下两个条件之一就被判定为闰年。

- 年份可以被 4 整除，但不能被 100 整除。
- 年份能被 400 整除。

本实例的效果如图 2-1 所示。

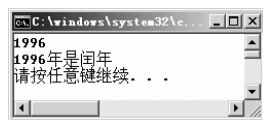


图 2-1 欢迎界面

【实现过程】

被判断的年份值由用户在界面中输入，变量被声明为 `year`。首先判断年份是否能被 400 整除，如果成立，则为闰年；反之则不是。如果不能被 400 整除，再判断上述第 1 个条件。代码如下：

```
01  #include <iostream>
02  using namespace std;
03
04  int main()
05  {
06      int year;                //年份
07      cin>>year;              //输入年份
08      if(year % 400 == 0)      //如果能被 400 整除，则为闰年
09          cout<<year<<"年是闰年"<<endl;
10      else                    //不能被 400 整除
11      {
12          if((year % 4 == 0) && (year % 100 != 0))    //判断闰年的另一个条件
13              cout<<year<<"年是闰年"<<endl;
14          else
15              cout<<year<<"年不是闰年"<<endl;
16      }
17
18      return 0;
19  }
```

【代码解析】

第 07 行表示用户输入被判断的年份值，第 08~16 行表示判断所输的值能否满足上述两个



条件之一。第 08、09 行表示是否满足第 2 个条件，第 10~16 行表示是否满足第 1 个条件。对于两个条件的判断，只要满足其中一种就可以被认为是闰年。



注意：上述两个条件并不是呈互相排斥关系，所以在 else 结构中必须还要有 if 条件的判断。



实例 017 打印 ASCII 码表

【实例描述】

在 C++ 编程中，ASCII 码是经常被使用的，它的作用是将字符、数字、标点转换为计算机可以识别的二进制形式。一般来讲，ASCII 码分为非打印控制字符和打印字符两种。每个 ASCII 码都由 8 位二进制数表示，本实例为了便于观察，用十进制数代替二进制数显示。ASCII 共有 128 个，从 0 到 27。其中，0~31 表示非打印控制字符，48~57 表示 10 个阿拉伯数字，65~90 表示 26 个大写英文字母，97~122 表示 26 个小写英文字母，其他表示标点符号，程序运行结果如图 2-2 所示。

```
C:\windows\system32\cmd.exe
0= : 16=> : 32= : 48=0 : 64=@ : 80=P : 96=' : 112=p :
1=@ : 17=< : 33=! : 49=1 : 65=A : 81=Q : 97=a : 113=q :
2=@ : 18=& : 34=" : 50=2 : 66=B : 82=R : 98=b : 114=r :
3=@ : 19=!! : 35=# : 51=3 : 67=C : 83=S : 99=c : 115=s :
4=@ : 20=@ : 36=$ : 52=4 : 68=D : 84=T : 100=d : 116=t :
5=@ : 21=$ : 37=% : 53=5 : 69=E : 85=U : 101=e : 117=u :
6=@ : 22=_ : 38=& : 54=6 : 70=F : 86=V : 102=f : 118=v :
7=@ : 23=@ : 39=' : 55=? : 71=G : 87=W : 103=g : 119=w :
8=@ : 24=@ : 40=< : 56=8 : 72=H : 88=X : 104=h : 120=x :
9=@ : 25=@ : 41=> : 57=9 : 73=I : 89=Y : 105=i : 121=y :
10=@ : 26=@ : 42=* : 58=: : 74=J : 90=Z : 106=j : 122=z :
11=@ : 27=@ : 43+= : 59=: : 75=K : 91=[ : 107=k : 123=[ :
12=@ : 28=@ : 44=, : 60=< : 76=L : 92=\ : 108=l : 124=[ :
13=@ : 29=@ : 45=- : 61== : 77=M : 93=] : 109=m : 125=[ :
14=@ : 30=@ : 46=. : 62=> : 78=N : 94=^ : 110=n : 126=[ :
15=@ : 31=@ : 47=/ : 63=? : 79=0 : 95=_ : 111=o : 127=@ :
请按任意键继续. . .
```

图 2-2 打印 ASCII 码表

【实现过程】

本实例实现每列打印 16 个 ASCII 码，打印格式如下：

十进制数字=对应的 ASCII 码 | ;

在这 128 个 ASCII 码中，有 7 个可以用转义字符代替输出，如表 2-1 所示为 ASCII 功能及转义字符对应关系。

表 2-1 ASCII 码—转义字符对应表

ASCII 码（十进制数）	功 能	转义字符
7	振铃	\a
8	退格	\b
9	水平制表符	\t
10	换行	\n
11	竖直制表符	\v
12	换页	\f
13	回车	\r



具体代码如下：

```

01  #include <iostream>
02  using namespace std;
03
04  int main()
05  {
06      int i = 0;
07      for(int rows = 0; rows < 16; rows++)
08      {
09          i = rows;
10          while(i <= 127)
11          {
12              switch(i)
13              {
14                  case 7:                                //震铃
15                      cout<<i<<"="<<"\a"<<" | ";
16                      break;
17                  case 8:                                //退格
18                      cout<<i<<"="<<"\b"<<" | ";
19                      break;
20                  case 9:                                //水平制表符
21                      cout<<i<<"="<<"\t"<<" | ";
22                      break;
23                  case 10:                               //换行
24                      cout<<i<<"="<<"\n"<<" | ";
25                      break;
26                  case 11:                               //竖直制表符
27                      cout<<i<<"="<<"\v"<<" | ";
28                      break;
29                  case 12:                               //换页
30                      cout<<i<<"="<<"\f"<<" | ";
31                      break;
32                  case 13:                               //回车
33                      cout<<i<<"="<<"\r"<<" | ";
34                      break;
35                  default:
36                      cout<<i<<"="<<char(i)<<" | ";
37                      break;
38              }
39
40              i+=16;                                     //每隔 16 个另起一列
41          }
42          cout<<endl;
43      }
44      return 0;
45  }

```

【代码解析】

第 07 行表示每列打印 16 个 ASCII 码，for 循环中嵌套 while 循环，用于打印所有的 ASCII 码。对于 7 个转义字符的打印，使用 switch 结构选择打印，如第 12~38 行。第 40 行表示在当前行中，前一个元素比后一个元素相差 16。



实例 018 求完数

【实例描述】

本实例模拟如何判断一个自然数是否是完数，首先要了解什么是完数。下面列出完数的判



断标准。

- 完数的判断对象必须是自然数。
- 一个自然数的真因子之和等于这个自然数。

不同于实例 016，本实例的判断标准必须同时满足上述两个条件，这个自然数才可以被认为是完数。本例运行效果如图 2-3 所示。

【实现过程】

首先声明一个整型变量 `shu` 用于存放被判断的数字。然后声明该数的所有真因子变量，因为事先不知道有多个真因子，所以变量类型为整型指针，变量名为 `zhen_yinzi`，大小为 `shu`（真因子的个数一定不会大于该数的值）。

另外，每个自然数都有一个值为 1 的真因子，存储于变量 `zhen_yinzi[0]` 中。整型变量 `sum` 为所有真因子的和，`index` 为真因子个数的索引值，具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int shu; //变量数字
07     cout<<"请输入一个任意的自然数: "<<endl;
08     cin>>shu; //输入数字
09     //求输入数字的所有真因子
10     int *zhen_yinzi = new int[shu]; //不知道真因子有多少个，所以用指针指向
11     int index = 0; //个数索引变量，初始化为 0
12     int sum = 0; //所有真因子的总和
13     zhen_yinzi[0] = 1; //所有数字都有一个真因子为 1
14     for (int i = 2; i < shu; i++) //循环查找所有的因子
15     {
16         if(shu % i == 0) //表示该因子为真因子
17         {
18             index ++;
19             zhen_yinzi[index] = i;
20         }
21     }
22
23     for(int j = 0; j <= index; j++) //将所有真因子相加
24     {
25         sum += zhen_yinzi[j];
26     }
27     delete[] zhen_yinzi; //释放内存
28     zhen_yinzi = NULL;
29
30     if(sum == shu) //如果真因子之和等于数字本身，即为完数
31         cout<<"数字"<<shu<<"是完数"<<endl;
32     else
33         cout<<"数字"<<shu<<"不是完数"<<endl;
34
35     return 0;
36 }
```

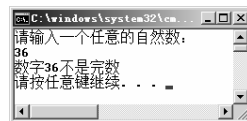


图 2-3 求完数应用

【代码解析】

第 06~13 行用来声明各种变量，并将真因子 `zhen_yinzi` 的第 1 个元素赋值为 1。第 04~21



行用于计算 shu 的所有真因子。一个数一定会被因子整除，所以 for 循环的判断范围为 2 到 shu，判断条件为 shu 被当前值 i 整除（如第 16 行）。第 23~26 行用于把所有的真因子相加，第 30 行判断相加之和是否等于 shu 的值。



注意：第 27、28 行是释放指针内存，在用 new 申请内存时，必须在程序结尾包含这句代码，否则会出现内存泄漏或野指针。



实例 019 密码验证（if...else）

【实例描述】

在设计一个登录系统时，一定要判断登录者的密码是否正确，此时就涉及密码验证功能。本实例利用 if...else 结构实现该功能，运行效果如图 2-4 所示。

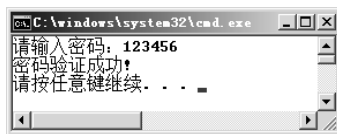


图 2-4 密码验证（if...else）

【实现过程】

首先定义被判断的输入密码 input_code，变量类型为 string。然后与事先定义的常量 Code 进行比较，如果相同，则验证成功；反之，失败。具体代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04 #define Code "123456"
05
06 int main()
07 {
08     cout<<"请输入密码: ";
09     string input_code;           //被验证的密码
10     cin>>input_code;
11
12     if(input_code == Code)       //如果相同，则密码为真
13         cout<<"密码验证成功!"<<endl;
14     else //为假
15         cout<<"密码验证失败!"<<endl;
16
17     return 0;
18 }
```

【代码解析】

由于密码的变量类型为 string，所以在程序前面要包含头文件 string.h，格式如第 02 行。第 04 行使用宏定义常量 Code 用于表示字符“123456”。第 09、10 行用于获取被判断的密码，第 12~15 行用于实现密码的验证功能。



注意：#define 定义的常量语句不能加分号。



实例 020 图书管理 (if...else if...else)

【实例描述】

图书馆中存有很多书籍，这些书籍又有各自的分类及名称，所以涉及如何快速查询想要借阅的图书。另外，借书和还书是图书馆的两大基本功能。针对此问题，当前图书的状态信息是一项很重要的功能实现。本实例基于控制台程序实现图书管理的基本功能演示，运行效果如图 2-5 所示。

【实现过程】

本实例实现的图书管理功能包括以下几大模块：

- 借书功能管理。
- 还书功能管理。
- 图书分类管理。
- 图书基本信息管理。
- 当前图书状态查询功能。
- 退出。

针对用户输入的功能编号，界面会弹出进入对应模块的提示语句，具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     cout<<"图书管理系统功能: "<<endl;
07     cout<<"1-借书功能管理"<<endl;
08     cout<<"2-还书功能管理"<<endl;
09     cout<<"3-图书分类管理"<<endl;
10     cout<<"4-图书基本信息管理"<<endl;
11     cout<<"5-当前图书状态查询功能"<<endl;
12     cout<<"0-退出"<<endl;
13     int status;
14     int count = 10;
15
16     while(count!=0)
17     {
18         cout<<"请输入您当前的选择: ";
19         cin>>status;
20
21         if(status == 0) //退出
22         {
23             cout<<"退出成功!"<<endl;
24             break;
25         }
26         else if(status == 1) //借书
27             cout<<"进入借书功能管理模块!"<<endl;
28         else if(status == 2) //还书
29             cout<<"进入还书功能管理模块!"<<endl;
```

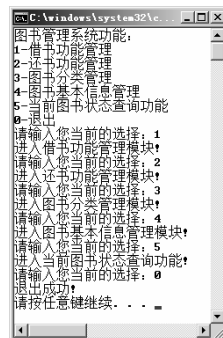


图 2-5 图书管理



```

30         else if(status == 3)                //图书分类管理
31             cout<<"进入图书分类管理模块!"<<endl;
32         else if(status == 4)                //图书基本信息
33             cout<<"进入图书基本信息管理模块!"<<endl;
34         else                                //查询当前图书状态
35             cout<<"进入当前图书状态查询功能!"<<endl;
36         count--;
37     }
38
39     return 0;
40 }

```

【代码解析】

第 06~12 行给出了图书管理系统的提示语句，整型变量 `status` 表示当前用户的选择，整型变量 `count` 表示用户可以重复多少次做出选择。第 16~37 行为 `while` 循环结构，第 21~35 行使用 `if...else if...else` 结构判断用户当前的选择，并进入相应的模块。其中，第 21~25 行给出当选择为 5 时，需要退出系统。如果循环次数没有达到 `count` 次，系统会强制退出，如第 24 行。



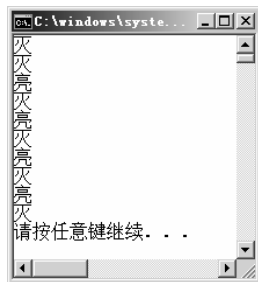
注意：本实例使用的 `if...else if...else` 结构可以用 `switch` 结构代替。



实例 021 信号灯 (++)

【实例描述】

信号灯的应用目的是向指定的人发出警报，如在大海上航行的船只、在天上飞行的飞机等，当这些物体在移动的过程中遇到危险时，地面的工作者会提前收到提醒，比如使用信号灯做提示。信号灯在工作的时候都会一灭一亮的，现规定如果计数次数为偶，则呈亮的状态，如果计数次数为奇，则呈灭的状态。本实例的运行效果如图 2-6 所示。



【实现过程】

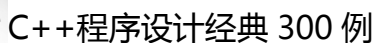
本实例演示时使用运算符 `++` 做辅助，首先定义一个整型变量 `count` 作为计数次数，然后用 `for` 循环判断当前 `count` 是偶还是奇，并且信号灯的最初状态为“灭”，具体代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int count=1;
07     cout<<"灭"<<endl;
08     for(;count<10;)                //循环
09     {
10         int a;
11         a = count++;
12         if(a % 2 == 0)                //偶数
13             cout<<"亮"<<endl;

```

图 2-6 ++位于变量后的信号灯状态



【代码解析】

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\windows\system...'. The command prompt contains a series of 'echo 亮' commands followed by a 'pause' command. The output shows the character '亮' being printed repeatedly on each line. At the bottom, the prompt '请按任意键继续...' is visible.

```
a = ++count;
```

注意：同理，运算符--的运算规则也一样。



实例 022 简单地获取变量的字节大小 (sizeof)

【实例描述】



图 2-8 获取变量的字节大小

【实现过程】

字节大小 = sizeof(变量);

具体代码如下:

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int a = 6;                                //一个整型变量
07     int int_size;                              //整型变量的字节大小
08     int_size = sizeof(a);                      //取变量字节大小
09
10     cout<<"变量 a 所占的字节大小为: "<<int_size<<endl;
11
12     return 0;
13 }
```



【代码解析】

第 06、07 行声明两个变量 `a` 和 `int_size`，第 08 行根据上述格式获取变量 `a` 的字节大小，第 10 行输出变量 `a` 的字节大小。



注意：本实例的 `sizeof(a)` 相当于获取整型变量的字节大小。



实例 023 求余下的物品数 (%)

【实例描述】

某公司准备在节假日给每位员工发相同数量的日用品。已知日用品的件数一共有 10000 件，员工有 189 名，求平均分配后还剩多少件日用品。本实例的运行效果如图 2-9 所示。

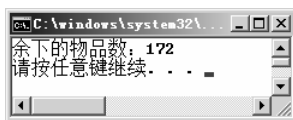


图 2-9 求余下的物品数 (%)

【实现过程】

首先需要定义 3 个整型变量，分别为总的物品数 `quantity_goods`、员工数目 `number_staff`、剩余物品数 `goods_remain`。现使用取余运算符计算机平均分配后还剩多少件日用品，具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int quantity_goods=10000;           //总的物品数
07     int number_staff=189;               //员工数目
08     int goods_remain;                   //剩余物品
09
10     goods_remain = quantity_goods % number_staff; //计算剩余物品
11     cout<<"余下的物品数: "<<goods_remain<<endl;
12
13     return 0;
14 }
```

【代码解析】

第 06~08 行用于声明 3 个变量，第 10 行表示计算所剩日用品件数的公式，第 11 行输出最后结果。



实例 024 输入符合条件的整数（跳出循环）

【实例描述】

本实例旨在模拟如何跳出循环，即在循环中如何打断它的下一步条件。现需要用户输入 1 个介于 1 和 50 的整数，如果不满足条件，重新输入；如果满足条件，则跳出循环。可以使用 `break` 跳出循环，运行效果如图 2-10 所示。

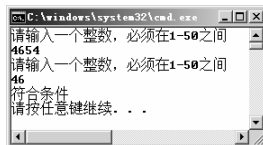
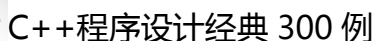


图 2-10 输入符合条件的整数



定义整型变量 `number` 用于存储输入的整数值，使用 `while(1)` 做无限循环，直到整数条件满足，则跳出循环。本实例的具体代码如下：

【代码解析】



实例 025 命令响应（开关）

布尔变量一般用于判断使用状态，比如开关的状态、命令是否被响应等。本实例实现是否响应命令、是否继续响应、是否暂停响应、是否继续等待，效果如图 2-11 所示。

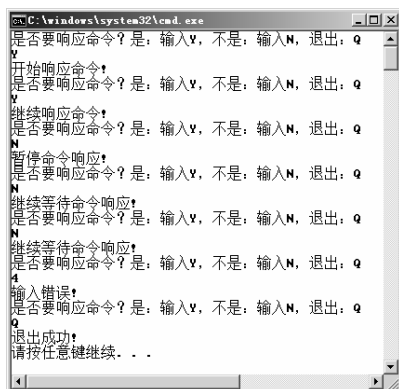


图 2-11 命令响应 (开关)



【实现过程】

基于上述的描述需要有两个布尔变量，`switcher` 用于存储开关的当前状态、`switcher_ex` 用于存储开关前一时刻的状态。布尔变量只有两个值，即 `true` 和 `false`。在变量初始时，都赋值 `false`。

因为需要获取用户输入值，需要定义一个 `char` 型变量 `ch` 用于存储该值。该程序只响应 `ch` 的 3 个类型值，即 Y、N、Q，具体代码如下：

```
01  #include <iostream>
02  using namespace std;
03
04  int main()
05  {
06      bool switcher=false;           //开关状态
07      bool switcher_ex=false;       //前一时刻的开关状态
08      char ch;
09
10      while(1)
11      {
12          cout<<"是否要响应命令? 是: 输入 Y, 不是: 输入 N, 退出: Q"<<endl;
13          cin>>ch;
14          if(ch=='Y')                //响应命令
15          {
16              switcher_ex=switcher; //前一状态
17              switcher=true;         //当前状态
18              if(switcher_ex)
19                  cout<<"继续响应命令!"<<endl;
20              else
21                  cout<<"开始响应命令!"<<endl;
22          }
23          else if(ch=='N')            //不响应
24          {
25              switcher_ex=switcher;
26              switcher=false;
27              if(switcher_ex)
28                  cout<<"暂停命令响应!"<<endl;
29              else
30                  cout<<"继续等待命令响应!"<<endl;
31          }
32          else if(ch=='Q')            //退出
33          {
34              cout<<"退出成功!"<<endl;
35              break;
36          }
37          else                        //输入其他值
38          {
39              cout<<"输入错误!"<<endl;
40          }
41      }
42
43      return 0;
44  }
```

【代码解析】

第 10~41 行为整个 `while` 循环，该循环为无限，只有满足退出条件才会跳出循环，如第 32~36 行所示。`while` 循环中嵌套 `if...else if...else` 判断结构。如果输入 `ch` 值不是 Y、N、Q 之一，则提醒输入错误，如第 37~40 行所示。



本实例的重点是判断用户输入的是 Y 还是 N, 实现判断的代码分别为第 14~22 行和第 23~31 行所示。它们的结构都是先将当前的开关状态 `switcher` 赋给前一个状态变量 `switcher_ex`, 然后根据 `ch` 值赋不同的值给当前状态变量 `switcher`, 最后根据前一个状态变量 `switcher_ex` 判断是否继续执行前一个状态。



注意: 如果 `while` 循环为无限循环, 即平常所说的死循环, 那么内部必须有跳出循环的判断条件。



实例 026 买水果小游戏（开关结构综合）

【实例描述】

本实例实现一个简单的买水果小游戏, 首先系统提示用户店里提供了哪些水果, 单价为多少。最后水果店根据用户的购买品种数量、所买品种各自的数量结算。本例运行效果如图 2-12 所示。

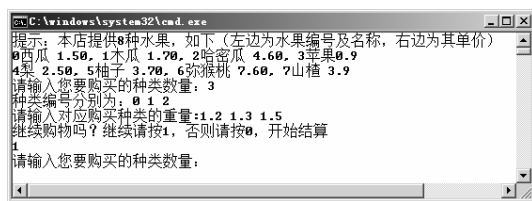


图 2-12 买水果小游戏（开关结构综合）

【实现过程】

本实例使用循环 (`while`、`for`) 和判断 (`switch`) 结构完成编码, 提示输入水果重量, 然后取得重量, 输入水果品种编号, 取得品种编号对应的价格, 计算应付的钱。在运行过程中, 可能会输入不符合条件的数值, 系统会给出相应的提示。具体代码如下:

```
01 #include <iostream>
02 using namespace std;
03
04 double capital=0.0; //金额
05
06 double cal(int num, double quantity)
07 {
08     switch(num)
09     {
10     case 0:
11         return quantity*1.50;
12     case 1:
13         return quantity*1.70;
14     case 2:
15         return quantity*4.60;
16     case 3:
17         return quantity*0.90;
18     case 4:
19         return quantity*2.50;
20     case 5:
21         return quantity*3.70;
```



```
22     case 6:
23         return quantity*7.60;
24     case 7:
25         return quantity*3.90;
26     }
27 }
28
29 int main()
30 {
31     cout<<"提示: 本店提供 8 种水果, 如下 (左边为水果编号及名称, 右边为其单价) "<<endl;
32     cout<<"0 西瓜 1.50, 1 木瓜 1.70, 2 哈密瓜 4.60, 3 苹果 0.9\n"
33         <<"4 梨 2.50, 5 柚子 3.70, 6 猕猴桃 7.60, 7 山楂 3.9"<<endl;
34     int num_choice=0; //几种选择
35     while(1)
36     {
37         int Y_N;
38         cout<<"请输入您要购买的种类数量: ";
39         cin>>num_choice;
40
41         if(num_choice==0) //不买东西
42         {
43             cout<<"您确定只是看看吗? 要不买点吧? 继续请按 1, 否则请按 0, 直接退出
44             "<<endl;
45             cin>>Y_N;
46             if(Y_N==0)
47             {
48                 cout<<"退出成功! "<<endl;
49                 break;
50             }
51             else if(Y_N==1)
52             {}
53             else
54                 cout<<"不要捣乱, 按规定输, 亲"<<endl;
55         }
56         else
57         {
58             cout<<"种类编号分别为: ";
59             int *choice=new int[num_choice];
60             double *quantity_choice=new double[num_choice];
61             for(int i=0;i<num_choice;i++)
62                 cin>>choice[i]; //种类编号
63             cout<<"请输入对应购买种类的重量:";
64             for(int i=0;i<num_choice;i++)
65                 cin>>quantity_choice[i];
66
67             for(int j=0;j<num_choice;j++) //计算金额
68             {
69                 capital += cal(choice[j], quantity_choice[j]);
70             }
71
72             cout<<"继续购物吗? 继续请按 1, 否则请按 0, 开始结算"<<endl;
73             cin>>Y_N;
74             if(Y_N==0)
75             {
76                 cout<<"开始结算, 请付"<<capital<<"元"<<endl;
77                 break;
78             }
79             else if(Y_N==1)
80             {}
81             else
82                 cout<<"不要捣乱, 按规定输, 亲"<<endl;
```



```

83         }
84     }
85     delete[] choice;
86     choice=NULL;
87     delete[] quantity_choice;
88     quantity_choice=NULL;
89     return 0;
90 }

```

【代码解析】

(1) 第 04 行定义一个全局变量 `capital` 用于存储顾客所需付钱的总额。

(2) 第 06~27 行为 `cal` 函数，作用是计算顾客购买水果的最后价格。输入参数为某种类型水果店的数量和重量，返回值为该种类型水果的总价。`cal()` 函数使用 `switch` 结构实现。其中每个 `case` 中有 `return` 语句，所以不需要有 `break` 语句。

(3) 第 29~90 行为入口函数 `main()`，第 34 行为顾客需要购买多少种水果 (`num_choice`)。第 35~84 行是一个死循环，当满足条件，则跳出该循环。如果 `num_choice` 为 0 时，系统会提示顾客，如第 43 行所示。反之，系统需要顾客输入水果类型各自的编号，如第 61~62 行的 `for` 循环所示。

(4) 第 59 行的 `choice` 为水果编号，第 60 行的 `quantity_choice` 为所选水果各自的重量。因为事先不确定购买多少种水果，所以这两个变量需要用 `new` 方式申请内存。第 67~70 行的 `for` 循环目的是实现金额的计算。当完成该次购物后，系统会提示是否继续购买，如第 72 行所示。



注意：将金额 `capital` 设置为全局变量，目的是在多次购买时，下一次的金额不会覆盖上一次的金额值。



实例 027 各类常量的使用示例（各种常量）

【实例描述】

C++ 中常量的调用方法有很多种，如直接调用 (`1`、`1.2`、`true`)、定义为常量以被调用（使用宏定义 `#define`、常量定义关键字 `const`）。本实例针对上述两种方法详细列出它们的调用方式，效果如图 2-13 所示。

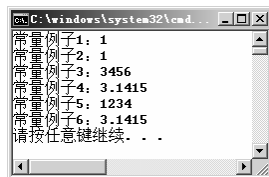


图 2-13 各类常量的使用例子
(各种常量)

【实现过程】

使用 `#define` 宏定义浮点型变量 `Pi` 和字符串变量 “1234”。利用 `const` 关键字定义 `double` 型常量 `Pi1`。其他常量的应用直接用 `cout` 语句输出，其代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 #define Pi 3.1415
05 #define code "1234"
06
07 int main()
08 {
09     cout<<"常量例子 1: "<<1<<endl;           //各类常量的输出方式
10     cout<<"常量例子 2: "<<true<<endl;

```



```

11      cout<<"常量例子 3: "<<"3456"<<endl;
12      cout<<"常量例子 4: "<<Pi<<endl;           //输出 define 宏定义的常量
13      cout<<"常量例子 5: "<<code<<endl;
14      const double Pi1 = 3.1415;
15      cout<<"常量例子 6: "<<Pi1<<endl;         //输出 const 定义的常量
16
17      return 0;
18  }

```

【代码解析】

任何基本类型的常量都可以直接调用，如第 09~11 行，分别为整型、布尔型、字符串。第 12、13 行调用#define 定义的常量 Pi 和 code，第 15 行调用 const 定义的常量 Pi1。



注意：使用#define 和 const 方式定义的常量都必须初始化，之后不可被再赋值。



实例 028 用循环计算 9 的 9 次方 (for)

【实例描述】

对于数学中的幂次运算，需要与自身重复相乘许多次。本实例需要实现计算 9 的 9 次方运算，运行效果如图 2-14 所示。

【实现过程】

本实例利用 for 循环实现 9 的 9 次方计算，首先声明两个变量：a 和 result。result 用于存储每次乘法之后的结果，可以使用*=运算符实现。每次的计算结果为：

```
result *= a;
```

因为每次相乘的乘法都是一样的，所以重复该计算公式 9 次即可，代码如下：

```

01  #include <iostream>
02  using namespace std;
03
04  int main()
05  {
06      int a = 9;
07      int result = 1;           //结果
08      for(int i = 1; i <= 9; i++) //循环
09      {
10          result *= a;
11      }
12      cout<<"9 的 9 次方等于"<<result<<endl; //输出
13
14      return 0;
15  }

```



图 2-14 用循环计算 9 的 9 次方 (for)

【代码解析】

此处用 for 循环实现幂次运算，第 10 行给出了程序的核心公式。只要将此公式的运算重复 9 次，即可完成计算。第 08 行按照 for 循环的书写格式给出起始条件 (int i=1)、终止条件 (i<=9)、



进入下一步条件 (i++)。



注意：在 C++ 的 API 函数中有 pow() 函数可以完成上述计算，需要包含头文件 math.h。



实例 029 寻找出口小游戏 (do while)

【实例描述】

走迷宫这款游戏也被称为寻找出口小游戏，本实例模拟该游戏实现对循环结构 do...while 的练习。在判断当前元素是否为出口时，使用 switch 结构完成，运行效果如图 2-15 所示。



图 2-15 寻找出口小游戏 (do while)

【实现过程】

因为涉及行列循环，所以本实例使用两个 do...while 结构实现，即一个 do...while 结构中嵌套另一个 do...while，各自进行的下一步条件为行列加 1。首先定义一个二维数组变量 migong，用于存放迷宫数据。当元素为 0 时，可以通过；如果是 1，不可通过；如果是 2，则为迷宫出口。其代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int migong[5][5]={0,0,1,1,1},{1,0,0,1,1},{1,1,0,1,1},{1,1,0,0,1},{1,1,1,0,2}};
07                                     //迷宫
08     int row,column;                 //行列
09     int path_row[25];               //通行路径的行
10     int path_column[25];            //通行路径的列
11     for(int i=0;i<25;i++)           //初始化
12         path_row[i]=path_column[i]=0;
13     row=0;
14     column=0;
15     int count=0;                   //次数
```



```

16
17     do                                     //按行循环，先处理，后判断
18     {
19         column = 0;
20         do                                 //按列循环，先处理，后判断
21         {
22             switch(migong[row][column])
23             {
24                 case 0:                     //可以通行
25                     path_row[count]=row;
26                     path_column[count]=column;
27                     cout<<"加油，快要找到出口了"<<endl;
28                     count++;
29                     break;
30                 case 1:                     //不可通行
31                     cout<<"不可通行"<<endl;
32                     break;
33                 case 2:                     //到达出口
34                     path_row[count]=row;
35                     path_column[count]=column;
36                     cout<<"到达出口"<<endl;
37                     count++;
38                     break;
39             }
40             column += 1;
41         }while(column<5);
42
43         row += 1;
44     }while(row<5);
45
46     cout<<"到达出口的路径为:"<<endl;
47     for(int j=0;j<count;j++)               //输出口路径
48     {
49         cout<<"("<<path_row[j]<<","<<path_column[j]<<")->";
50     }
51     cout<<endl;
52
53     return 0;
54 }

```

【代码解析】

由第 06 行可知，该迷宫的出口位于第 4 行第 4 列，第 08、09 行用于存放到达出口的路径各行列值，第 17~41 行为按行循环的 do...while 结构，第 20~39 行为按列循环的 do...while 结构，第 47~50 行输出路径链。



注意：第 19 行将列值赋为 0，该句必须有，必须回到数组列数的第 0 列，否则会越界。



实例 030 一个简单的数据统计系统（综合）

【实例描述】

本实例实现一个简单的数据统计系统，即在一串输入的字符中找到目标字符出现过多少



次，可被分为两个功能：搜索和统计。运行效果如图 2-16 所示。

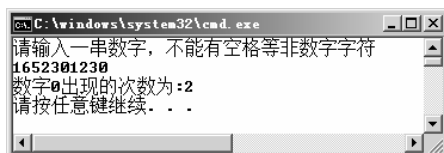


图 2-16 一个简单的数据统计系统 (综合)

【实现过程】

定义一个字符串 `str`，用于获取输入的源数据。`count_0` 统计数据中出现 0 的次数，代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     cout<<"请输入一串数字，不能有空格等非数字字符"<<endl;
08     string str;
09     int count_0=0;                //0 出现的次数
10     cin>>str;                    //输入
11     cout<<"数字 0 出现的次数为:";
12     for(int i=0;i<str.length();i++) //判断
13     {
14         if(str[i]=='0')            //如果等于 0
15             count_0++;
16     }
17     else
18     {}
19
20     cout<<count_0<<endl;
21
22     return 0;
23 }
```

【代码解析】

本实例的核心内容是第 14、15 行，先判断当前元素是否为 0。如果为 0，`count_0` 加 1，否则什么都不做，直到搜索到达数据的末位（结束搜索）。



注意：字符串的长度可以用 `string` 的成员函数 `length()` 获取。



实例 031 投骰子游戏（随机）

【实例描述】

在玩牌或麻将等游戏时总少不了投骰子，本实例模拟投骰子游戏。投骰子的结果可由概率统计得到，本实例演示掷一枚 6 点骰子（即一个骰子有 6 面，包括 1、2、3、4、5、6 这 6 个点），得到其结果点数。由概率论可知，出现其中一个点数的概率是 1/6，可被看作是随机现象。本例运行效果如图 2-17 所示。

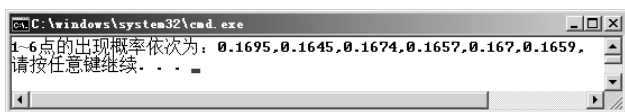


图 2-17 投骰子游戏（随机）

【实现过程】

实现本实例需要两个函数 `srand()` 和 `rand()`，用于模拟当前时刻得到位于 1 到 6 之间的整数。函数 `srand()` 和 `rand()` 一起使用可以产生伪随机数。本实例循环 10000 次，统计结果并打印每个点数出现的概率，其代码如下：

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <time.h>
04 #include <iostream>
05 using namespace std;
06
07 int main()
08 {
09     int i,result;
10     srand((int)time(0));           //利用系统时间产生随机序列的种子值
11     int count[6]={0};             //1~6 的统计个数
12     for(i=0;i<10000;i++)
13     {
14         result=1+(int)(6.0*rand()/(RAND_MAX+1.0)); //设置出现 1~6 之间的整数
15         switch(result)
16         {
17             case 1:
18                 count[0]++;
19                 break;
20             case 2:
21                 count[1]++;
22                 break;
23             case 3:
24                 count[2]++;
25                 break;
26             case 4:
27                 count[3]++;
28                 break;
29             case 5:
30                 count[4]++;
31                 break;
32             case 6:
33                 count[5]++;
34                 break;
35         }
36     }
37
38     cout<<"1~6 点的出现概率依次为: ";
39     for(int k =0;k<6;k++)           //打印每点出现的概率
40         cout<<count[k]/10000.0<<" ";
41
42     cout<<endl;
43
44     return 0;
45 }
```



【代码解析】

其中，第 09 行定义变量 `i`（表示循环次数）、`result`（当前产生的随机数）。第 10 行利用系统时间产生随机序列的种子值，这样可以保证每次运行程序时产生的随机值都不同。第 11 行定义的数组 `count` 用于计数 1~6 产生的次数，由第 15~35 行的 `switch` 结构。第 12~36 行循环 10000 次，获取 10000 个投骰子后结果。第 14 行利用 `rand()` 函数产生介于 1~6 之间的随机数。第 39~40 行输出每点出现的概率，实验证明结果接近于 1/6。



注意：使用函数 `srand()`、`rand()`、`time()` 必须包含第 01~03 行的头文件。

第 3 章 数 组

本章介绍的数组是继 C++ 基础变量后的一个重点内容，它可以将多个同一类型的变量集合在一个数组中使用。数组的具体定义格式是将一系列同类型的变量以一定的顺序排列，以组成变量集合。这些变量有各自的编号以示区分，编号被称为下标。一般情况下，常用的数组为一维和二维，同样也有多维数组。本章重点介绍一维、二维和字符数组的应用。



实例 032 推箱子（数组元素移动）

【实例描述】

本实例旨在实现移动数组元素，借助推箱子游戏的思想进行模拟。现假设一个整型数组只有 4 个整数值，分别为 0、1、2、3。其中，元素 0 表示可以通过；元素 1 表示不可以通过；元素 2 表示箱子所在处；元素 3 表示目的地。本实例实现将箱子推向目的地，效果如图 3-1 所示。

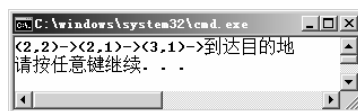


图 3-1 模拟数组元素移动——推箱子

1	1	0	1
1	1	2	1
1	0	0	1
3	0	1	1

图 3-2 array_032 数组内容

【实现过程】

现定义一个整型数组 array_032（4 行 4 列），初始化为如图 3-2 所示的矩阵形式。

现设定入口是在第 0 行第 2 列处，箱子位于第 1 行第 2 列处，出口位于第 3 行第 0 列处。本实例模拟从上到下，从右到左搜索元素为 0 的位置，移动数组元素 2 到出口处（第 3 行第 0 列），代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 struct enter
05 {
06     int row;
07     int col;
08 };
09 int main()
10 {
11     int array_032[4][4]={{1,1,0,1},{1,1,2,1},{1,0,0,1},{3,0,1,1}}; //数组元素
12     enter enter_box; //箱子的位置
13     enter_box.row=1; //箱子行
14     enter_box.col=2; //箱子列
15
16     for(int i=enter_box.row;i<4;i++) //从上到下，从右到左搜索元素为 0 的位置
17     {
18         for(int j=enter_box.col;j>=0;j--)
19         {
20             if(array_032[i][j]==0) //继续搜索
```



```

21         {
22             enter_box.row=i;
23             enter_box.col=j;
24             cout<<" "<<enter_box.row<<" "<<enter_box.col<<"->";
25         }
26         else if(array_032[i][j]==1)                //不能移动
27         {}
28         else if(array_032[i][j]==3)
29         {cout<<"到达目的地"<<endl;}
30         else
31         {}
32     }
33 }
34     return 0;
35 }

```

【代码解析】

第 04~08 行声明结构体 `enter`，它的两个元素包括行和列。第 11 行定义数组变量 `array_032`，第 12 行定义箱子的位置，第 13、14 行给箱子的初始位置赋值。第 16~33 行为嵌套 `for` 循环，从上到下，从右到左搜索元素为 0 的位置。

其中，第 20~25 行为当前元素为 0 时，记录当前数组移动的位置。第 26、27 行表示遇到元素为 1 时，不能移动。第 28、29 行表示遇到元素为 3 时，已到达目的地。



注意：本实例基于对数组元素的规律观察，得到从上到下、从左到右搜索组内元素可以到达目的地。



实例 033 数据复制（复制一组数组到另一组数组）

【实例描述】

在数组的应用中，最常见的是数据的复制，即将一组数组的变量值复制到另一组数组中，该操作被称为数组间数据复制。数组间数据复制需要它们的变量类型相同，如果不相同，可以使用强制类型转换实现。但缺点是可能会有损有效数据的信息。例如，`double` 型数据复制到 `float` 型变量时，由于 `double` 的精度大于 `float` 数据，在强制转换后会丢失有效位数后的信息，即

```

double a=0.1234567895;
float b;
b=a;

```

此时，`b` 内存的值是 0.123457，这样造成了数据信息的丢失，运行效果如图 3-3 所示。

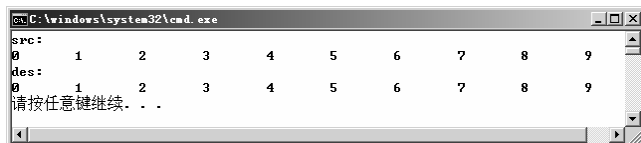


图 3-3 数组间信息的复制

【实现过程】

定义两组数组变量，分别为 `src`（源数组）和 `des`（目标数组）。它们的长度都为 10，在最



初给 `src` 数组初始化, 经过数据的复制后, 分别输出两个数组的值。代码如下:

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int src[10];
07     int des[10];
08     for(int i=0;i<10;i++)          //src 的初始化, 并复制到 des 数组中
09     {
10         src[i]=i;
11         des[i]=src[i];
12     }
13     cout<<"src:"<<endl;
14     for(int j=0;j<10;j++)          //输出 src 数组
15         cout<<src[j]<<" ";
16     cout<<"des:"<<endl;
17     for(int k=0;k<10;k++)          //输出 des 数组
18         cout<<des[k]<<" ";
19     return 0;
20 }
```

【代码解析】

第 06、07 行定义两个数组, 第 08~12 行使用 `for` 循环初始化数组 `src`, 并同时每个元素值复制到 `des` 数组对应的元素。第 13~15 行输出 `src` 数组各元素的值, 第 16~18 行输出 `des` 数组每个元素的值。



注意: 源数组和目标数组的长度也可以不相等, 此时需要改变 `for` 循环的起始与终止条件, 但是需要注意不能超过两个数组的界限。



实例 034 内存输出 (打印数据)

【实例描述】

数组在初始化后, 其内存的某个区域保存这些变量的值。本实例演示如何将内存中这些的数据打印出来, 运行效果如图 3-4 所示。

【实现过程】

对于本实例的实现, 需要定义一个数组 `array_034` (此处以 `int` 型数据为例), 并且初始化。通过 `for` 循环把内存区的数据打印出来, 具体代码如下:

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int array_034[10]={1,1,2,3,4};    //定义数组
07     for(int i=0;i<10;i++)
08         cout<<array_034[i]<<endl;    //输出数组中的各个元素
09     return 0;
10 }
```

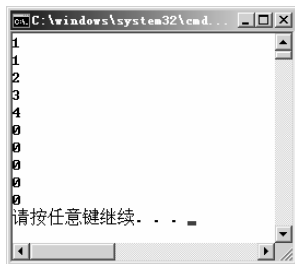


图 3-4 内存输出



【代码解析】

第 06 行定义整型数组 `array_034`，并进行初始化。其中只给前 5 位变量赋值，后 5 位自动赋为 0。第 07、08 行输出内存变量值。



实例 035 一维数组的应用

【实例描述】

本实例实现一维数组的应用，一维数组的每个元素只有一个下标值，即位于数组的第几位。本实例演示找到某下标元素的值，运行效果如图 3-5 所示。

【实现过程】

定义一个整型数组 `array_035`，长度为 10。另外，整型变量 `temp` 用于存放需要查找的元素下标值，目的是查找该下标元素的数据为多少，具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     cout<<"请输入大小为 10 的整型数组 array_035 的各元素值"<<endl;
07     int array_035[10];           //定义数组
08     int temp;                     //定义临时变量
09     for(int i=0;i<10;i++)
10         cin>>array_035[i];       //初始化数组元素
11     cout<<"请输入您要查找的某位元素下标: ";
12     cin>>temp;                   //输出要查找的元素下标
13     cout<<"经搜索，第"<<temp<<"位元素的值为:"<<array_035[temp]<<endl;
14
15     return 0;
16 }
```

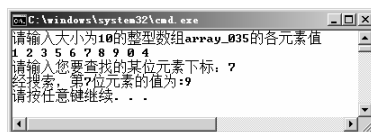


图 3-5 一维数组的某位数据检索

【代码解析】

第 07、08 行定义数组 `array_035` 和临时变量 `temp`，第 09、10 行初始化数组元素，第 12 行获取需要搜索的元素下标值，第 13 行输出该下标元素的数据。



注意：该程序的改进之处为需要判断 `temp` 的值是否越界，否则会造成程序崩溃。



实例 036 整数从大到小排序（比较法）

【实例描述】

本实例实现利用比较法对整数进行从大到小的排序，将一组需要排序的数据存放于数组中，然后利用 `for` 循环比较两个元素。大的元素往前排，小的元素往后排，结果数组是一个按从大到小顺序排列的元素变量。运行效果如图 3-6 所示。



【实现过程】

定义整型数组 `src`，长度为 10，初始化为{11,12,47,24,49,69,90,89,18,39}。之后用嵌套 `for` 循环比较相邻两个元素的大小，如果前一个元素大于后一个，不做任何操作；反之，互相交换。在交换的过程中需要临时变量暂时存放第一个元素的值，命名为 `temp`。本实例的实现代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int src[10] = {11,12,47,24,49,69,90,89,18,39}; //一维数组中包含 10 个整数
07     //从大到小排序
08     for(int i = 0; i < 10; i++)
09     {
10         for(int j = i+1; j < 10; j++)
11         {
12             if(src[i]<src[j]) //如果前一个元素小于后一个元素
13             {
14                 int temp; //临时变量
15                 temp = src[i];
16                 src[i] = src[j]; //大的元素到前一个位置
17                 src[j] = temp; //小的元素到后一个位置
18             }
19         }
20     }
21     for(int k = 0; k < 10; k++)
22         cout<<src[k]<<endl;
23     return 0;
24 }
```



图 3-6 数组元素从大到小排序

【代码解析】

第 06 行定义包含 10 个整型变量的数组 `src`，并进行初始化。第 08~20 行完成元素的从大到小排序操作，最后由第 21、22 行输出各元素。



注意：第 10 行 `j` 的初始值为 `i+1`，即完成前一个元素与后续元素的一一比较。



实例 037 查找二维坐标点

【实例描述】

二维数组含有多个元素，对于目标元素的查找可以通过嵌套 `for` 循环实现，它的返回值包括行和列值（也被称为二维坐标点）。本实例实现二维坐标点的查找，运行效果如图 3-7 所示。

【实现过程】

定义整型二维数组 `array_037`，大小为两行三列，并初始化为{1,1,3,4,1,6}。本实例的目的是查找元素为 1 的二维

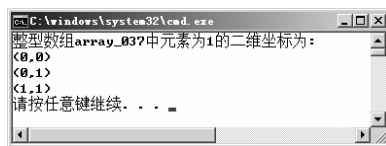


图 3-7 二维坐标点的查找



坐标点（也即其行列值，结果以二维坐标点的形式输出），代码如下：

```
01  #include <iostream>
02  using namespace std;
03
04  void main()
05  {
06      int array_037[2][3]={1,1,3,4,1,6};           //定义二维数组
07      cout<<"整型数组 array_037 中元素为 1 的二维坐标为:"<<endl;
08      for(int i=0;i<2;i++)
09      {
10          for(int j=0;j<3;j++)
11          {
12              if(array_037[i][j]==1)               //记录下标
13                  cout<<"("<<i<<","<<j<<")"<<endl;    //输出
14          }
15      }
16  }
```

【代码解析】

第 06 行定义数组 array_037，并进行初始化，第 08~15 行使用嵌套 for 循环实现目标元素的查找，并且由第 13 行输出该元素的所有下标。



实例 038 查找矩阵最大的元素

【实例描述】

基于实例 037 输出目标元素的下标值，也可以通过嵌套 for 循环查找矩阵中最大的元素。此处介绍嵌套 for 循环的格式，即一个 for 循环存在于另一个 for 循环中，格式如下：

```
for(起始条件;终止条件;下一步条件)
{
    for(起始条件;终止条件;下一步条件)
    {...}
}
```

结果输出矩阵中最大元素的值和其下标值，本实例运行效果如图 3-8 所示。

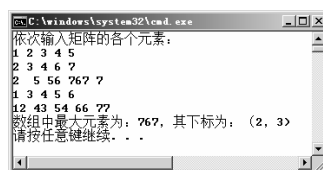


图 3-8 查找矩阵的最大元素

【实现过程】

定义二维数组 juzhen 为 M 行 N 列变量，其中 M 和 N 值由宏预先定义。定义整型变量 max、row 和 column，分别存放最大元素的值、最大元素的行列下标，代码如下：

```
01  #include <iostream>
02  using namespace std;
03
04  #define M 5
05  #define N 5
06
07  int main()
08  {
09      int juzhen[M][N];
10      int max;           //最大值
11      int row,column;    //行列下标
```



```

12      cout<<"依次输入矩阵的各个元素: "<<endl;
13      for(int i=0;i<M;i++)                //初始化
14      {
15          for(int j=0;j<N;j++)
16          {
17              cin>>juzhen[i][j];
18          }
19      }
20      max=juzhen[0][0];
21      for(int i=0;i<M;i++)                //查找最大元素
22      {
23          for(int j=0;j<N;j++)
24          {
25              if(max<juzhen[i][j])
26              {
27                  max=juzhen[i][j];        //最大元素赋值
28                  row = i+1;
29                  column = j+1;
30              }
31          }
32      }
33      cout<<"数组中最大元素为:"<<max<<"，其下标为: ("<<row<<","<<column<<")"<<endl;
34      return 0;
35  }

```

【代码解析】

第 04、05 行宏定义 M 和 N 的值（此为常量），用于定义数组 juzhen 的大小（第 09 行）。第 10、11 行定义变量 max、row 和 column。第 13~19 行初始化数组 juzhen，第 20 行初始化 max 的值为 juzhen 的第 0 行第 0 列元素。第 21~30 行查找数组的最大元素，并做记录，第 33 行输出最大元素的值，及其行列下标值。



注意：在 C++ 语言中，数组也被称为矩阵。



实例 039 二分法查找

【实例描述】

实例 038 中，查找 M 行 N 列数组中最大的元素需要比较 $M \times N$ 次，当 M 和 N 的值较大时，比较次数也较多，将会耗费大量的时间。使用二分法查找目标元素的时间比较短，但是它的适用前提是数组数据已经被排序。二分法的基本思想为：对于给定值的查找，如果大于该数组的中间元素，下一步在元素值大的区域继续与其中间元素比较；否则，下一步在元素值小的区域内继续查找，直到找到目标元素。如果到最后还没有找到，则输出“数组中没有该元素”。本例运行效果如图 3-9 所示。

【实现过程】

定义一个一维数组 array_039，并进行初始化。此处为减少排序环节，每个元素的值呈从小到大趋势。另外，定义整型变量 temp 获取需要查找的元素值，具体代码如下：

```
01 #include <iostream>
```

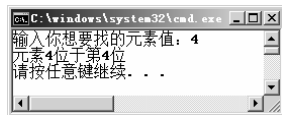


图 3-9 二分法查找应用于一维数组



```

02  using namespace std;
03
04  int array_039[10]={0,1,2,3,4,5,6,7,8,9};
05
06  void binary_search(int left,int right,int value)
07  {
08      int search_index;
09      search_index=(left+right)/2;           //二分
10      if(array_039[search_index]==value)    //如果等于二分点值
11          cout<<"元素"<<value<<"位于第"<<search_index<<"位"<<endl;
12      else if(value>array_039[search_index]) //大于，往右边移
13          binary_search(search_index+1,right,value);
14      else if(value<array_039[search_index]) //小于，往左边移
15          binary_search(left,search_index-1,value);
16      else
17          return;
18  }
19
20  int main()
21  {
22      int temp;
23      cout<<"输入你要查找的元素值: ";
24      cin>>temp;
25      binary_search(0,9,temp);
26
27      return 0;
28  }

```

【代码解析】

第 04 行定义一维数组 `array_039`，并初始化为{0,1,2,3,4,5,6,7,8,9}。第 06~18 行为函数 `binary_search()`的定义体，第 20~28 行为主函数 `main()`。在 `main()`函数中，第 22 行定义查找的元素变量 `temp`，第 24 行获取其值，第 25 行调用 `binary_search()`函数。

在 `binary_search()`函数中，第 1 个参数为二分搜索范围的左边坐标值，第 2 个参数为其右边坐标值，第 3 个参数为需要查找的元素。在 `binary_search()`函数中，第 08 行的 `search_index`为要二分查找的当前下标值，第 10~17 行的 `if...else if...else` 结构判断当前下标的元素是否等于 `temp`，并做出相应的动作。



注意：在计算二分查找的当前下标值 `search_index` 时，因为 `left+right` 可能为奇数，此时的 `search_index` 为其 `(left+right)/2` 结果的四舍值。



实例 040 查找三维坐标点

【实例描述】

实例 037 演示了查找二维坐标点的实现过程，本实例实现查找三维坐标点。给定一个常量值，在三维数组中查找该元素位于数组的哪个位置，运行效果如图 3-10 所示。

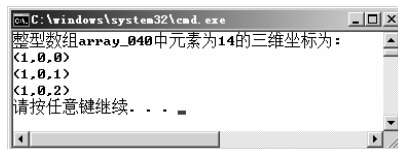


图 3-10 查找三维坐标点



【实现过程】

定义一个三维数组 `array_040[2][3][4]`，并进行初始化。本实例查找元素值 14 位于该数组的哪个位置，需要应用 3 个 `for` 循环完成查找。这 3 个 `for` 循环的终止条件分别为行、列、面的长度，具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     int array_040[2][3][4]={1,2,3,4,5,6,7,8,9,11,12,13,14,14,14,3245,56};
07     cout<<"整型数组 array_040 中元素为 14 的三维坐标为:"<<endl;
08     for(int i=0;i<2;i++)                //扫描第一维
09     {
10         for(int j=0;j<3;j++)            //扫描第二维
11         {
12             for(int k=0;k<4;k++)        //扫描第三维
13             {
14                 if(array_040[i][j][k]==14) //记录下标
15                 {
16                     cout<<"("<<i<<","<<j<<","<<k<<")"<<endl;
17                 }
18             }
19         }
20     }
21 }
```

【代码解析】

第 06 行定义三维数组 `array_040`，并进行初始化。第 08~20 行实现 3 个 `for` 循环嵌套以查找值为 14 的元素坐标，然后由第 16 行输出。



实例 041 获取数组大小（sizeof）

【实例描述】

利用 `sizeof()` 函数可以获取变量在内存中所占字节的大小，如 `char` 型变量占 1 字节，`int` 型变量占 4 字节，那么一组数组在内存占多少字节呢？它也可以由 `sizeof()` 获取，返回值为数组在内存中所占的字节数。而数组的大小还需进一步计算，公式如下：

数组大小=数组所占字节数/数组类型所占字节数；

本实例的演示效果如图 3-11 所示。



图 3-11 获取数组大小

【实现过程】

首先定义一维数组 `array_041[4]`，然后利用 3 个 `for` 循环嵌套以记录被查找的目标元素下标，其代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
```



```

06      int array_041[4]={1,1,2,4};                //定义一维数组
07      cout<<"整型数组 array_041 的大小: "<<sizeof(array_041)/sizeof(int)<<endl;
                                                //数组大小

08      return 0;
09  }
```

【代码解析】

第 06 行定义一维数组 `array_041`，第 07 行获取其大小并输出。其中，`sizeof(array_041)` 的返回值为该数组在内存中所占字节数，`sizeof(int)` 为整型数据在内存中所占字节数。因为 `array_041` 属于整型数组，其中有 4 个整型变量，所以它的大小必须由数组所占字节除以所属类型的字节。



注意：一个数组的字节数是其类型字节数的整数倍。



实例 042 按位数排列

【实例描述】

本实例要实现对一维数组的元素按位数进行排列，即位数越多，越往后排（十位数排在个位数后面，百位数排在十位数后，等等），运行效果如图 3-12 所示。



图 3-12 按位数排列一维数组各元素

【实现过程】

模拟排列一维数组 `array_042`，元素个数为 10。首先计算数组中各个元素的位数，定义为函数 `weishu()`，代码如下：

```

01  int weishu(int chushu, int shu)
02  {
03      double result;
04      result=double(shu/chushu);
05      int new_chushu=chushu;
06      if((result>=0)&&(result<=9))                //判断当前位数是否只有一位
07          return new_chushu;
08      else                                        //如果不是一位，继续除以 10
09      {
10          new_chushu=chushu*10;
11          weishu(new_chushu, shu);                //调用自身
12      }
13  }
```

函数 `weishu()` 使用嵌套调用获取每个元素的位数，返回值为数的位数。`main()` 函数中定义 10 个数的位数 `beishu`，初始化为 1。在两个 `for` 循环中，如果前一位数小于或等于后一个，则不做改变，否则相互交换，代码如下：

```

01  int main()
02  {
03      int array_042[10]={1,21,3231,423,1,11,14334,15466}; //定义数组元素
04      int beishu[10]={1};                                   //每个元素的位数初始化为 1
05      for(int i=0;i<10;i++)
06          beishu[i]=weishu(1,array_042[i]);                //获取每个元素的位数
07      for(int i=0;i<10;i++)                                //对 beishu 按从小到大排序
08      {
```



```

09         for(int j=i+1;j<10;j++)
10         {
11
12             if(beishu[i]<=beishu[j])           //不改变位置
13             {}
14             else                               //调换位置
15             {
16                 int temp;
17                 temp=array_042[i];
18                 array_042[i]=array_042[j];
19                 array_042[j]=temp;
20                 temp=beishu[i];
21                 beishu[i]=beishu[j];
22                 beishu[j]=temp;
23             }
24         }
25     }
26     cout<<"按位数排列后的 array_042 为: "<<endl;
27     for(int i=0;i<10;i++)
28         cout<<array_042[i]<<endl;           //按位数从小到大输出
29     return 0;
30 }

```

【代码解析】

(1) 在 `weishu()` 函数中, 如果除法结果在 0 和 9 之间, 则返回位数; 否则将除数的值再乘以 10, 继续判断, 直到满足条件 0 和 9 之间为止 (第 06~12 行)。

(2) 在 `main()` 函数中, 第 05、06 行获取每个元素值的位数, 并存储到数组 `beishu` 中。第 07~25 行判断前一个元素的位数和后一个元素的位数。第 27、28 行输出按位数排列后的数组元素。



注意: 在判断元素位数时, 当满足条件 (第 14 行) 时, 不但要交换数组 `array_042` 元素的位置, 还要交换 `beishu` 的两元素位置。



实例 043 统计班上男生和女生的人数 (数组随机访问)

【实例描述】

本实例模拟数组的随机访问, 以此来统计班上男生和女生的人数。比如, 班上一共有 N 个学生, 但不知道男生和女生各有多少人, 可以随机访问其中一位学生, 让其说明性别, 然后在相应的性别属性中进行统计, 直到访问完所有的学生。实例运行效果如图 3-13 所示。

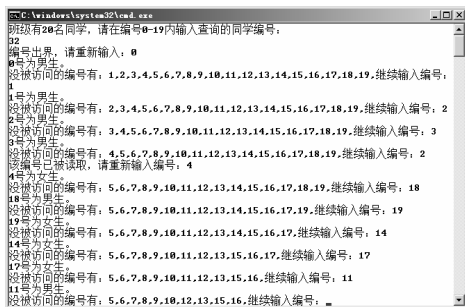


图 3-13 数组随机访问



【实现过程】

定义整型数组 `array_043[20]`，初始化其各个元素的值。值为 0 表示女生，值为 1 表示男生。布尔型数组变量 `index[20]` 标识 `array_043[]` 的对应元素是否已被读取，`false` 表示未被读取，`true` 表示已被读取。`temp` 接收键盘输入随机值，`female_num` 和 `male_num` 分别表示班级中男女生的人数。变量 `random_count` 标识正确访问的次数统计。

当获取由键盘输入的随机值时，先判断该随机值是否在总人数范围内，如果不在，重新输入值，否则进行接下来的判断，即判断是否被读取，如果没有被读取，则判断是男还是女。如果已被读取，继续获取键盘随机值。在判断是否已访问完所有的人数时，通过正确访问次数 `random_count` 标识。本实例程序的逻辑性很强，可以归结为如图 3-14 所示的流程图。

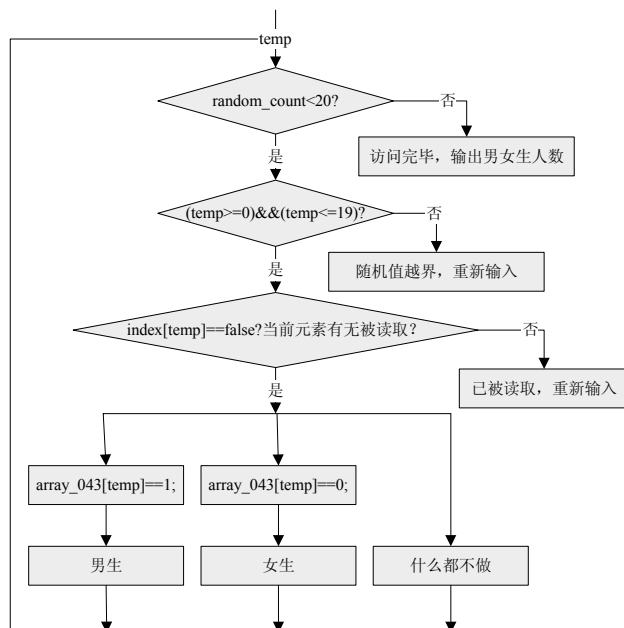


图 3-14 数组随机访问流程

代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int array_043[20]={1,1,1,1,0,1,0,0,0,0,1,1,1,0,0,0,1,0,1,0}; //学生的属性
07     bool index[20]; //表示在随机访问时是否已被读取, false 为未被读取, true 为已被读取
08     int temp; //随机值
09     int female_num=0;
10     int male_num=0;
11     int random_count=0; //随机访问的次数
12     for(int i=0;i<20;i++)
13         index[i]=false; //初始化各自的读取状态
14     cout<<"班级有 20 名同学, 请在编号 0~19 内输入查询的同学编号: "<<endl;
15
16     while(random_count<20) //未访问完
17     {
18         cin>>temp;
```



```

19         if ((temp>=0) && (temp<=19))
20         {
21             if (index[temp]==false)                //未被读取, 继续
22             {
23                 if (array_043[temp]==1)            //男生
24                 {
25                     male_num++;
26                     cout<<temp<<"号为男生。";
27                 }
28                 else if (array_043[temp]==0)        //女生
29                 {
30                     female_num++;
31                     cout<<temp<<"号为女生。";
32                 }
33                 else                                //什么都不做
34                 {
35                     index[temp]=true;
36                     random_count++;
37                     cout<<endl;
38                     cout<<"未被访问的编号有: ";
39                     for (int i=0; i<20; i++)
40                     {
41                         if (index[i]==false)
42                             cout<<i<<" ";
43                     }
44                     cout<<"继续输入编号: ";
45                 }
46             } else                                //该随机值已出现过
47             {
48                 cout<<"该编号已被读取, 请重新输入编号: ";
49             }
50         } else                                    //编号已出现过
51         {
52             cout<<"编号出界, 请重新输入: ";
53         }
54     }
55     cout<<"访问完毕, 男女生人数分别为: "<<male_num<<" "<<female_num<<endl;
56     return 0;
57 }
```

【代码解析】

第 06~11 行定义上述各种变量, 第 12、13 行初始化 `index[]`, 都为未被读取。第 16~51 行为 `while()` 循环结构体, 表示访问完全班人数的属性, 第 52 行输出男女人数。第 19~50 行的 `if...else` 结构表示输入随机值是否属于学生的编号。如果属于学生编号, 由第 21~47 行的 `if...else` 结构判断该编号的状态是否被读取。如果没被读取, 由第 23~44 行构成其内容。

其中, 第 23~34 行判断当前编号是男生还是女生。第 35 行将当前读取的编号状态置为 `true`, 并且 `random_count` 加 1。第 39~43 的 `for` 循环输出没有被读取的学生编号, 以做提示。



注意: 本实例的随机值可以通过 `rand()` 函数实现。



实例 044 内存指令表 (数组+开关)

【实例描述】

在计算机的内存中, 每块内存都存放数据, 其中有系统数据, 也有用户临时存储的。内存



是一块块的单元格，有的内存存放重要的数据（不可读写，为保护系统），而有的内存可以随时读写，就像编写 C++ 程序所用的内存。本实例利用数组和开关变量模拟内存指令表，指示哪块内存可以读写，哪块不可以，效果如图 3-15 所示。

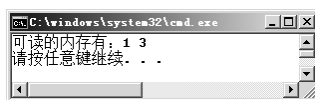


图 3-15 内存指令表

【实现过程】

定义布尔型数组 memo[5]，初始化第 0 个变量为 false（即不可读写），剩下的 4 个变量值都取前一个变量的反，最后输出可以读写的内存编号，代码如下：

```
01 #include <iostream>
02 using namespace std;
03 int main()
04 {
05     bool memo[5];
06     memo[0]=false;           //第 0 块内存不可读
07     for(int i=1;i<5;i++)     //内存可读写初始化
08     {
09         memo[i]=!memo[i-1]; //取上一位的反
10     }
11     cout<<"可读的内存有: ";
12     for(int i=0;i<5;i++)     //输出可读内存的值
13     {
14         if(memo[i])           //可读
15             cout<<i<<" ";
16     }
17     cout<<endl;
18     return 0;
19 }
```

【代码解析】

第 05 行定义 memo[5]，第 06 行初始化 memo[0]，第 07~10 行初始化剩余的 4 个变量。如果值为 false，表示不可读写；反之，可以读写。第 12~16 行输出可以读写的内存编号。



注意：在程序编码平台上，系统会自动分配内存，此时不会触及对整个系统重要的内存块。但有的程序需要另外定义不可读写的内存区，此时就需要事先判断。



实例 045 模拟栈空间（数组+算法）

【实例描述】

本实例利用数组模拟栈空间，栈是数据结构的一种，它的存储规则是先进后出，当栈满时，就不再写入，只能读出。本实例在数组的基础上形成算法以实现先进后出的功能及判定栈是否溢出，效果如图 3-16 所示。

【实现过程】

使用整型数组模拟进栈和出栈运算，定义数组 aa[M]，M 的值利



图 3-16 数组模拟栈空间



用宏定义设为 10。定义两个索引值 i 和 j ，分别初始化为 0 和 M ，作为进栈和出栈计数所用，其代码如下：

```

01  #include <iostream>
02  using namespace std;
03
04  #define M 10
05
06  int main()
07  {
08      int aa[M];                //定义长度为 M 的数组
09      bool flag=false;          //初始时 false 为可以读写
10      int i=0;                  //进栈索引变量 i
11      int j=M;                  //出栈索引变量 j
12      while(flag==false)        //如果可以读写继续循环
13      {
14          if(i>M-1)              //进栈已满，开始出栈
15          {
16              j--;                //出栈索引减 1
17              if(j==0)            //到栈底
18                  flag=true;      //不能再读写
19              cout<<aa[j]<<" ";    //出栈
20          }
21          else                    //元素进栈
22          {
23              cin>>aa[i];          //进栈
24              i++;
25          }
26      }
27      cout<<endl;
28      return 0;
29  }
```

【代码解析】

第 04 行宏定义 M 的值，第 08、09 行定义变量 aa 和布尔型 $flag$ ，用以标识该栈还能否读写。当为 $true$ 时，不能读写；反之，可以读写。第 12~26 行为 $while$ 循环，如果可以读写，进入循环；反之，则退出。在进入循环后， $if\cdots else$ 结构判断是进栈还是出栈，判断条件是进栈后是否溢出 ($i>M-1$)。如果没有溢出，继续写入，并且进行 $i++$ 操作。如果溢出，先进行 $j--$ 操作，再判断出栈操作是否到达栈底。如果到栈底，不可读写， $flag$ 赋值为 $true$ ，然后输出元素值。



注意：因为在 $if(i>M-1)$ 结构中，先进行 $j--$ ，所以先到达栈底，再输出。此时不需要将第 19 行包含到 $if(j==0)$ 结构中。



实例 046 同学姓名册（字符数组）

【实例描述】

字符数组是由一串字符组成的数组，它可以应用于同学的姓名册，其一般按姓的首字母大小排序。字符数组可以分为一维、二维及其多维，本实例应用二维数组模拟同学姓名册，效果如图 3-17 所示。

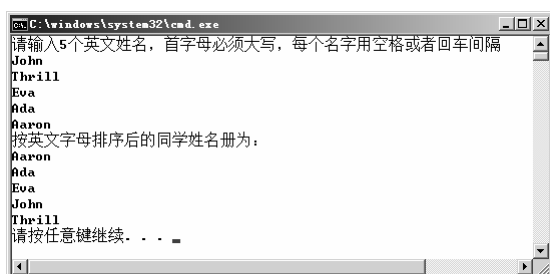


图 3-17 字符数组模拟同学姓名册

【实现过程】

定义字符数组 `student[5][10]`，此处行表示第几个同学，列表示姓名的长度（假设学生的姓名长度不超过 9 个字符）。输入 5 个同学的姓名，并按首字母顺序排序，然后输出，代码如下：

```
01  #include <iostream>
02  using namespace std;
03
04  int main()
05  {
06      char student[5][10];                //5 个学生姓名
07      cout<<"请输入 5 个英文姓名，首字母必须大写，每个名字用空格或者回车间隔"<<endl;
08      for(int i=0;i<5;i++)
09          cin>>student[i];
10      for(int i=0;i<5;i++)                //循环第一维
11      {
12          for(int j=i+1;j<5;j++)          //循环第二维
13          {
14              int index=0;                //英文姓名的第一位
15              do
16              {
17                  if(student[i][index]>student[j][index])//当前位前大于后，交换
18                  {
19                      char temp[10];
20                      for(int k=0;k<10;k++)    //交换两者的英文姓名
21                      {
22                          temp[k]=student[i][k];
23                          student[i][k]=student[j][k];
24                          student[j][k]=temp[k];
25                      }
26                      index=10;
27                  }
28                  else if(student[i][index]==student[j][index])
29                      //当前位相等，比较下一个
30                      {index++;}
31                  else
32                      //不交换，也不比较
33                      {index=10;}
34              }while(index<10);
35          }
36      }
37      cout<<"按英文字母排序后的同学姓名册为："<<endl;
38      for(int i=0;i<5;i++)
39          cout<<student[i]<<endl;
40      return 0;
41  }
```



【代码解析】

第 06 行定义字符数组 `student[5][10]`，第 08、09 行表示 `for` 循环获取每个学生的姓名。第 10~34 行将这 5 个姓名按字母顺序排列。如果姓名的第 0 个字符相等（第 28 行），则比较姓名的下一个字符，直到 10 个字符都比较完为止。

其中，第 17~30 行使用 `if...else if...else` 结构判断当前元素的大小关系。第 15~32 的 `do...while` 结构循环判断对应的字符，直到到达每个姓名的最后位置。



注意：由于中文姓名的获取涉及编码的方式，所以本实例只介绍英文名字。在内存中，一个汉字占两字节。



实例 047 图书管理系统（字符数组综合）

【实例描述】

本实例模拟利用字符数组实现图书管理系统，比如，图书编号、书名、作者、出版社等信息。基于上述图书信息，本实例提供功能有添加、删除、修改、查询、退出。功能的具体解释如下。

- (1) 添加：添加新的图书到数据库管理系统。
- (2) 删除：删除某编号书的信息。
- (3) 修改：修改某编号书的某项信息。
- (4) 查询：查询某编号的所有信息。
- (5) 退出：退出该系统。

效果如图 3-18 所示。

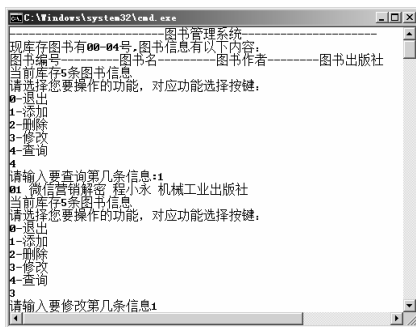


图 3-18 字符数组模拟图书管理系统

【实现过程】

定义结构体 `book`，用于存放图书管理系统对于图书信息的编辑，有 `ch_num[3]`（图书编号）、`ch_name[100]`（图书名称）、`ch_author[10]`（图书作者）、`ch_publisher[50]`（图书出版社），代码如下：

```
01 struct book
02 {
03     char ch_num[3];           //2 字节，最多为 99 字节
04     char ch_name[100];        //100 字节，最多为 50 个汉字
05     char ch_author[10];       //10 字节，最多为 5 个汉字
06     char ch_publisher[50];    //50 字节，最多为 25 个汉字
07 };
```

在执行函数 `main()` 中定义结构体变量 `Book[100]`（最多存放 100 本图书信息）、`number`（初始有几本图书）、`choice`（当前选择项目功能）、`del_num`（删除第几条信息）、`modify_num`（修改第几条信息）、`inspect_num`（查询第几条信息）、`xunhuan`（是否退出系统），功能代码如下：

```
01 int main()
02 {
03     book Book[100];
```

//数据库只能存 100 条项目



```
04      int number=5;                                //初始化为 5 条
05      strcpy(Book[0].ch_num, "00");                //数据库中现有 5 条记录
06      strcpy(Book[0].ch_name, "微信, 这么玩才赚钱");
07      strcpy(Book[0].ch_author, "王易");
08      strcpy(Book[0].ch_publisher, "机械工业出版社");
09      strcpy(Book[1].ch_num, "01");
10      strcpy(Book[1].ch_name, "微信营销解密");
11      strcpy(Book[1].ch_author, "程小永");
12      strcpy(Book[1].ch_publisher, "机械工业出版社");
13      strcpy(Book[2].ch_num, "02");
14      strcpy(Book[2].ch_name, "C++ Primer 中文版 (第 5 版)");
15      strcpy(Book[2].ch_author, "(美) 李普曼、拉乔伊");
16      strcpy(Book[2].ch_publisher, "电子工业出版社");
17      strcpy(Book[3].ch_num, "03");
18      strcpy(Book[3].ch_name, "疯狂 Android 讲义");
19      strcpy(Book[3].ch_author, "李刚");
20      strcpy(Book[3].ch_publisher, "电子工业出版社");
21      strcpy(Book[4].ch_num, "04");
22      strcpy(Book[4].ch_name, "C 程序设计 第四版");
23      strcpy(Book[4].ch_author, "谭浩强");
24      strcpy(Book[4].ch_publisher, "清华大学出版社");
25      cout<<"-----图书管理系统-----"<<endl;
26      cout<<"现库存图书有 00-04 号, 图书信息有以下内容: "<<endl;
27      cout<<"图书编号-----图书名-----图书作者-----图书出版社"<<endl;
28      int choice;                                    //选择功能
29      int del_num;                                    //删除信息
30      int modify_num;                                //修改信息
31      int inspect_num;                                //查询信息
32      bool xunhuan=true;
33      while(xunhuan)
34      {
35          cout<<"当前库存"<<number<<"条图书信息"<<endl;
36          cout<<"请选择您要操作的功能, 对应功能选择按键: "<<endl;
37          cout<<"0-退出"<<endl;
38          cout<<"1-添加"<<endl;
39          cout<<"2-删除"<<endl;
40          cout<<"3-修改"<<endl;
41          cout<<"4-查询"<<endl;
42          cin>>choice;
43          switch(choice)
44          {
45              case 0:                                    //退出
46                  cout<<"退出成功"<<endl;
47                  xunhuan=false;
48                  break;
49              case 1:                                    //添加
50                  cout<<"请遵照上面信息排列输入新图书的信息: "<<endl;
51                  number++;
52
53                  cin>>Book[number].ch_num>>Book[number].ch_name>>Book[number].ch_author
54                  >>Book[number].ch_publisher;
55                  break;
56              case 2:                                    //删除
57                  cout<<"要删除的第几条信息";
58                  cin>>del_num;
59                  number--;
60                  if (del_num>number)
```



```

61         cout<<"没有该条信息"<<endl;
62     else if(del_num==number)           //删除最后一条
63     {
64         strcpy(Book[del_num-1].ch_author,"");
65         strcpy(Book[del_num-1].ch_name,"");
66         strcpy(Book[del_num-1].ch_publisher,"");
67         strcpy(Book[del_num-1].ch_num,"");
68     }
69     else                               //删除其余项目,后面的需全部向前移
70     {
71         for(int i=del_num-1;i<number;i++)
72         {
73             strcpy(Book[i].ch_author,Book[i+1].ch_author);
74             strcpy(Book[i].ch_name,Book[i+1].ch_name);
75             strcpy(Book[i].ch_publisher,Book[i+1].ch_publisher);
76         }
77     }
78     break;
79     case 3:                             //修改
80         cout<<"请输入要修改第几条信息";
81         cin>>modify_num;
82         if(modify_num>number)
83             cout<<"没有该条信息"<<endl;
84         else
85         {
86             cout<<"按次序输入新内容,如果部分修改,可将原信息原样输入"<<endl;
87
88             cin>>Book[modify_num-1].ch_num>>Book[modify_num-1].ch_name>>Book[modi
89 fy_num-1].c h_author>>Book[modify_num-1].ch_publisher;
90         }
91         break;
92     case 4:                             //查询
93         cout<<"请输入要查询第几条信息:";
94         cin>>inspect_num;
95         cout<<Book[inspect_num].ch_num<<" "<<Book[inspect_num].ch_name<<"
96 "<<Book[inspect_num].ch_author<<" "<<Book[inspect_num].ch_publisher<<endl;
97         break;
98     default:
99         break;
00     }
01 }
02 return 0;
03 }

```

【代码解析】

第 03~32 行定义各种变量,并初始化现有库存的 5 种图书信息。第 32~99 行为 while() 循环,用于用户当前选用哪种功能。在 switch() 结构中,第 45~48 行为退出功能,第 49~55 行为添加功能,第 56~78 行为删除功能。删除功能又以 if...else if...else 结构判断当前删除的条目超出条目总数、最后一条及其他情况。第 79~90 行为修改功能,对于不需要修改的字段信息,需要原样输入。第 92~97 行为查询功能。



注意: 在删除功能中,如果不判断是否为最后一条,就不会真正从内存中删除。



实例 048 约瑟问题（把异教徒投入海中排法）

【实例描述】

本实例演示约瑟问题的一个应用，即如何把所有的异教徒投入海中。该应用的背景为，有 30 名教徒一起乘船航行，其中基督徒和异教徒各 15 名，但因为遇到危险需要将一半人投入海中，方可得救。解决方法是 30 个人围成一圈，由第 1 个人报数 1，接着第 2 个人报数 2，依此类推，当报数为 9 的那个人被投入海。之后进行下一轮，下一个人接着从 1 开始报数，第 9 个人又被投入海中。按此规则循环，直到船上只剩下 15 个人为止，怎样排列这 30 个人才能将所有的异教徒投入海中？程序运行效果如图 3-19 所示。

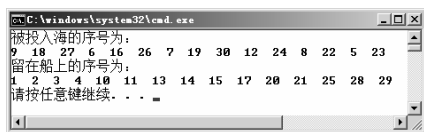


图 3-19 模拟约瑟问题

【实现过程】

定义整型数组 `all[30]`、`yijiao[15]`、`jidu[15]` 分别存放所有人的编号、异教徒编号和基督徒编号。在循环查找异教徒的编号时，需要用到控制变量。此处定义整型变量 `yijiao_count`（被投入海的人数计数）、`yijiao_index`（被投入海的下标）、`jidu_index`（留在船上的下标）。该算法的实现流程如图 3-20 所示。

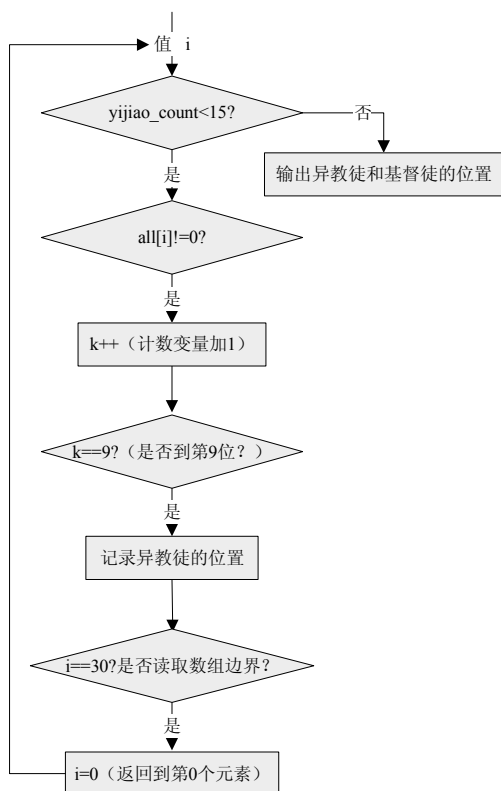


图 3-20 模拟约瑟问题



具体代码如下:

```
01 int main()
02 {
03     int all[30];                //所有人编号
04     int yijiao[15];            //异教徒编号
05     int jidu[15];              //基督徒编号
06     int i,k,yijiao_count,yijiao_index,jidu_count;
07     for (i=0;i<30;i++)
08         all[i]=i+1;            //每人编号
09     i=0;                        //i 为每次循环时计数变量
10     k=0;                        //k 为按 1, 2, ..., 9 报数时计数变量
11     yijiao_count=0;            //投入海人数
12     yijiao_index=0;            //存被投入海者数组的下标
13     jidu_count=0;              //存在船上人编号数组的下标
14     while (yijiao_count<15)    //有 15 个投入海
15     {
16         if (all[i]!=0)          //没被丢入海
17             k++;
18         if (k==9)
19         {
20             yijiao[yijiao_index]=all[i];
21             yijiao_index++;
22             all[i]=0;            //被丢入海标志
23             k=0;
24             yijiao_count++;
25         }
26         i++;
27         if (i==30)              //到边界
28             i=0;
29     }
30     for(i=0;i<30;i++)
31     {
32         if (all[i]!=0)
33         {
34             jidu[jidu_count]=all[i];
35             jidu_count++;
36         }
37     }
38     cout<<"异教徒的序号为: "<<endl;
39     for(i=0;i<15;i++)
40     {
41         cout<<yijiao[i]<<" ";
42     }
43     cout<<endl<<"基督徒的序号为: "<<endl;
44     for(i=0;i<15;i++)
45     {
46         cout<<jidu[i]<<" ";
47     }
48     cout<<endl;
49
50     return 0;
51 }
```

【代码解析】

第 03~13 行定义各种所需变量, 并进行初始化。第 14~29 行是 while() 循环, 用以获取 15 个异教徒都被丢入海的下标。第 30~37 行的 for 循环用于获取 15 个基督徒的下标值。第 39~47 行分别输出异教徒和基督徒的序号。



注意：一定要判断是否到达数组 `all[]` 的边界（如第 27 行），并且返回到第一个元素。



实例 049 数组转置

【实例描述】

在学习矩阵或线性代数时会学到矩阵，经常会听这样一个名词：转置。在学习 C++ 之后，有一个名词叫数组。事实上，二维数组的表现形式与矩阵相差无几，如何实现数组的转置？本实例就是利用代码实现二维数组的转置功能，效果如图 3-21 所示。

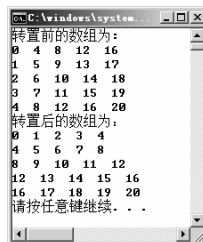


图 3-21 数组转置

【实现过程】

首先定义一个 5×5 的二维数组 `array_2`，然后进行数组初始化。先输出转置前的二维数组中各个元素的值，之后进行转置运算，最后输出转置后的二维数组各元素的值。本实例使用了 3 段 `for` 循环：初始化、转置、转置数组输出，其代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     //一维数组的转置
07     int array_2[5][5];           //定义一个二维数组
08
09     cout<<"转置前的数组为:"<<endl;
10     for(int i=0;i<5;i++)        //数组初始化
11     {
12         for(int j=0;j<5;j++)
13         {
14             array_2[i][j]=i+j*4;
15             cout<<array_2[i][j]<<" ";
16         }
17         cout<<endl;
18     }
19
20     //转置
21     for(int i=0;i<5;i++)
22     {
23         for(int j=i+1;j<5;j++)
24         {
25             //以对角线为对称轴互换元素
26             int temp;
27             temp=array_2[i][j];
28             array_2[i][j]=array_2[j][i];
29             array_2[j][i]=temp;
30         }
31     }
32
33     cout<<"转置后的数组为: "<<endl;
34     for(int i=0;i<5;i++)
35     {
36         for(int j=0;j<5;j++)
```



```
37         {
38             cout<<array_2[i][j]<<" ";    //输出转置后元素值
39         }
40         cout<<endl;
41     }
42
43     return 0;
44 }
```

【代码解析】

第 10~18 行用于二维数组的初始化及转置前的输出。为了避免数组中以对角线对称位置的元素相同的状况，利用第 14 行为它们赋予不同的值。在初始化的同时，输出各元素的值。第 21~31 行用于对数组的转置，为了减少循环次数，所以只需要处理对角线一侧的元素即可（如第 23 行所示的 for 循环起始条件）。根据二维数组元素下标的对称性即可完成转置。

由于转置后的数组元素全部输出，所以代码如第 34~41 行。另外，为了更直观地看到转置前后的对比，在每行元素都输出后，都会输出一个回车符，如第 17、40 行。

第 4 章 C++字符串

本章内容围绕字符串展开阐述，C++语言中处理字符串的方式有两种，一种是字符数组，另一种是字符串类 `string`。虽然都处理字符串，但是两者的操作原理有所区别。为了了解各自的基本原理，本章从输入、输出、复制、比较及两者之间的相互转化，给出了对应的实例，以期从最基本的概念让读者学会如何使用 C++字符串。此处声明在使用 `string` 类做字符串操作时，必须包含头文件 `string.h`。



实例 050 输出字符串的每个字符（for 访问数组）

【实例描述】

本实例利用字符数组形成字符串，然后用 `for` 循环访问字符数组的每个元素并输出，效果如图 4-1 所示。

【实现过程】

首先定义由 10 个元素组成的字符数组 `ch`，然后用 `for` 循环访问每个元素并输出，其代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 //输出字符串的每个字符
05 int main()
06 {
07     char ch[10]={'a','b','s','c','d','w','j','e','o','w'}; //字符数组
08     for(int i =0;i<10;i++) //循环访问
09         cout<<ch[i]<<endl; //输出
10     return 0;
11 }
```



图 4-1 for 循环访问字符数组并输出每个字符

【代码解析】

其中，第 07 行定义字符数组 `ch`，元素个数为 10。第 08、09 行使用 `for` 循环访问该数组各元素并输出。



注意：第 07 行是显式定义字符数组的大小，可以不写数组的元素个数，书写格式为 `char ch[]`。此时，后面赋值元素有几个，该字符数组的大小就是多少。



实例 051 循环改写一段字符串（for 访问数组）

【实例描述】

字符数组被声明后，必须进行初始化。但是在使用的过程中，需要对字符串的某个元素或者全部元素都进行改写，此时也需要使用 for 循环先访问数组，然后重新赋值。运行效果如图 4-2 所示。



图 4-2 for 循环访问数组以改写字符串

【实现过程】

本实例利用字符串类 string 实现字符串的改写，目标字符串为 str。使用 cin 输入进行改写字符串中操作，具体代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     string str="abscdwjeowifj";           //字符串变量
08     for(int i =0;i<str.length();i++)      //循环访问
09         cin>>str[i];                     //改写
10     cout<<str<<endl;                    //输出
11
12     return 0;
13 }
```

【代码解析】

第 07 行定义被改写的字符串 str，并进行初始化。第 08、09 行循环访问并改写，第 10 行输出改写后的字符串 str 值。



注意：由运行结果可以得出，cin 只获取字符串长度的元素，超出范围后不予考虑。



实例 052 把一个字符串截断（\0）

【实例描述】

本实例介绍如何截断一个字符串，可以通过判断字符串的大小来完成。同时，也可以通过截断标志符实现，即字符\0。通过该方法可以在不知道字符串大小的情况下，通过字符\0 输出真正的字符串内容。而字符\0 之后的内容被看作冗余。运行效果如图 4-3 所示。

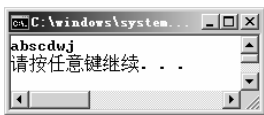
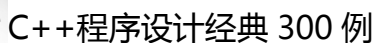


图 4-3 字符串截断标志符\0

【实现过程】

字符串截断标志符\0 同时适用于字符数组和字符串类 string。当字符串在声明或初始化的时候不清楚其大小，可以搜索截断标志符\0 以判定在哪个位置应该结束字符串的赋值。本实例以字符串 string 为例，对于字符数组的应用同理，此处不再赘述，

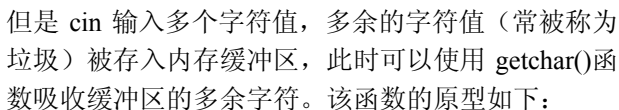


```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     string str="abscdwj\0eowifj";           //字符串变量
08     cout<<str<<endl;                         //输出
09
10     return 0;
11 }
```

第 07 行定义字符串 `str`，并在其初始化的过程中，包含元素 0，以标志 `str` 的赋值结束。第 08 行输出 `str` 的值。



在编程过程中，需要定义多个变量并进行初始化，尤其是在使用指针的过程中，输入的变量个数会大于需求数。此处以字符型变量为例展开介绍，定义一个字符型变量，由 `cin` 赋值。



`getchar()`函数的返回值类型为 `int`，当该函数被调用时，程序等待用户从键盘输入值。利用键盘输入的字符先被存入键盘缓冲区，一旦按键为回车，`getchar()`函数从缓冲区中读入一个字符，返回值为当前从缓冲区读入字符的 ASCII 码值，之后又回显于屏幕区。每调用一次 `getchar()`函数，会从缓冲区读入一个字符。

本实例的运行效果如图 4-4 所示。

本实例的实现过程较简单，不需要定义变量，只需无限循环调用 `getchar()` 函数，即可完成吸收缓冲区垃圾，代码如下：

64



【代码解析】

第 06 行表示死循环，第 07 行表示输出键盘缓冲区的字符。考虑到 `getchar()` 函数返回字符的 ASCII 码值，此处强制转换为 `char` 型，以输出其字符格式。



实例 054 字符串输入（`getline()`）

【实例描述】

在处理字符串的赋值时，可以通过各种方式实现，如运算符操作和函数应用等，分别为赋值运算符=、操作符 `cin>>`、`getline()` 函数等。本实例介绍如何使用函数 `getline()` 完成字符串的输入，其函数原型如下：

```
istream& getline(istream &is, string &str, char delim);
```

第 1 个参数指进行读入操作的输入流，可以是 `cin`、`ifstream` 等，它属于输入参数。第 2 个参数是 `string` 类型，属于输出参数，指输入的字符串存储于 `str` 中。第 3 个参数表示终结标志符，即在字符串输入的过程中，一旦遇到该终结符，即可终止字符串的输入。运行效果如图 4-5 所示。



【实现过程】

本实例用的 `getline()` 函数必须作用于字符串 `string` 类型，所以先定义一个 `string` 类型变量 `str`。在输入的过程中，如果遇到字符 `$`，终止字符串的输入。本实例的实现代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     string str;                //字符串变量 str
08     getline(cin, str, '$');    //读取字符串数据，直到出现字符$
09     cout<<str<<endl;         //输出
10     return 0;
11 }
```

【代码解析】

第 08 行运用 `getline()` 函数完成对字符串 `str` 的输入，第 09 行输出 `str` 的值。



注意：`getline()` 函数只能作用于 `string` 类型的字符串，不可以使用字符数组。



实例 055 复制一个字符串（`strcpy()`）

【实例描述】

本实例利用函数 `strcpy()` 完成字符串的复制，它的适用对象是字符数组。先来看其函数原型，如下：



```
char *strcpy( char *strDestination, const char *strSource );
```

第 1 参数为输出参数，第 2 参数为输入参数。`strcpy()`属于 C 语言标准库函数，其作用是将首地址为 `src` 直到遇到结束符的字符串复制到以首地址为 `dest` 开始的地址空间。本实例的运行效果如图 4-6 所示。



图 4-6 利用 `strcpy()`复制一个字符串

【实现过程】

对于本实例的实现，需要定义两个字符串 `str1` 和 `str2`，并且必须是字符数组类型，不能是 `string` 类型。变量 `str1` 的值为 `Hello World`，通过 `strcpy()`复制到 `str2`。代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     char str1[]="Hello World";    //字符数组 1
08     char str2[30];                //字符数组 2
09     strcpy(str2, str1);           //复制
10     cout<<str1<<endl;            //输出被复制的字符串
11     return 0;
12 }
```

【代码解析】

第 07 行定义变量 `str1`，没有标明数组大小，可以由初始化值的长度决定。为免于被复制字符串区的大小不满足 `str1` 的长度要求，`str2` 的大小需要大于 `str1` 的长度，如第 08 行所示。第 09 行完成两个字符数组的复制，并由第 10 行输出目标字符串 `str2`。



注意： `str2` 中其他未复制的值为空格字符。



实例 056 获得字符串长度 (`strlen()`)

【实例描述】

在定义字符数组时，可能不写明它的长度。对于此种情况，可以通过函数 `strlen()`获取，其函数原型如下：

```
size_t strlen( const char *string );
```

参数 `string` 的类型为 `char` 型指针，它的目标是将获取以 `string` 为首地址的字符串长度并返回。该函数的适用对象也是字符数组，其运行效果如图 4-7 所示。



图 4-7 `strlen()`函数获取字符串长度



【实现过程】

本实例较为简单，主要演示如何利用 `strlen()` 函数获取字符串 `str` 的长度，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     char str[]="HelloWorld";           //已知字符数组
07     cout<<"str 的长度为:"<<strlen(str)<<endl;   //输出字符数组的长度
08     return 0;
09 }
```

【代码解析】

在第 06 行定义字符串 `str` 时没有设定其长度，在第 07 行通过 `strlen()` 函数获得，并输出。



实例 057 字符串的比较 (`strcmp()`)

【实例描述】

本实例旨在实现如何完成 C 风格的字符串比较，可以使用 `strcmp()` 函数，它的原型如下：

```
int strcmp( const char *string1, const char *string2 );
```

其中，两个参数是需要比较的两个字符串。这两个字符串必须是 C 风格形式的，即字符数组。返回值为比较结果，如果返回 0，表示 `string1` 等于 `string2`；如果返回值大于 0，表示 `string1` 大于 `string2`；如果返回值小于 0，表示 `string1` 小于 `string2`。本实例的效果如图 4-8 所示。

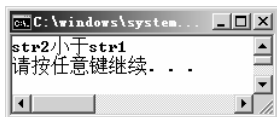


图 4-8 利用 `strcmp()` 函数比较两个 C 风格字符串

【实现过程】

首先定义两个 `char` 型字符数组 `str1` 和 `str2`，并分别进行初始化。整型变量 `result` 用于存放比较结果。本实例的具体代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     char str1[]="HelloWorld";           //字符数组 1
08     char str2[]="Hello World";          //字符数组 2
09     int result;                          //比较结果
10     result=strcmp(str2, str1);           //比较两个字符串
11
12     if(result<0)                          //str2 小于 str1
13         cout<<"str2 小于 str1"<<endl;
14     else if(result>0)                     //str2 大于 str1
```



```

15         cout<<"str2 大于 str1"<<endl;
16     else                                     //两者相等
17         cout<<"str2 等于 str1"<<endl;
18
19     return 0;
20 }

```

【代码解析】

第 07~09 行为两个比较字符串 str1、str2 和比较结果 result。第 10 行得到比较结果，第 12~17 行判断比较结果 result，并输出对应的比较大小。



注意：在 C++编程语言中，即使两个字符串的大小及内容都相同，但是有大小写区分，结果也是不相同的，并且大写字母小于小写字母。



实例 058 连接两个字符串（strcat()）

【实例描述】

对于 C 风格的字符串，可以利用函数 strcat()实现两个字符串的连接，该函数的原型如下：

```
char *strcat( char *strDestination, const char *strSource );
```

它的作用是把字符串 strDestination 和 strSource 进行连接，并将连接结果存储到字符串 strDestination 中，效果如图 4-9 所示。



【实现过程】

定义两个字符串 str1 和 str2，其中 str1 为目标字符串，即

图 4-9 用 strcat()连接两个字符串 第 1 个参数，str2 为第 2 个参数。本实例的具体代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     char str1[100]="Hello";    //一定要给 str1 的大小赋值，否则会越界
07     char str2[]=" World";     //另一个字符串
08
09     strcat(str1,str2);         //连接字符串 str2 到 str1 上
10     cout<<str1<<endl;        //输出 str1
11
12     return 0;
13 }

```

【代码解析】

第 06、07 行定义两个字符串，第 09 行实现字符串的连接，第 10 行输出。



注意：因为 str1 为目标字符串，所以它的大小必须大于自身，并且要足够大到可以容纳要连接的字符串 str2。



实例 059 将小写字符转换为大写 (strupr())

【实例描述】

平常在输入验证码的时候, 可以不考虑字母的大小写, 实际上是后台对输入字符进行了大小写处理, 实现函数为 `strupr()`, 原型如下:

```
extern char *strupr(char *s);
```

由上可知, 它的适用对象也是字符数组。输入参数为要转为大写字母的字符串首地址, 返回值为变换为大写字母的字符串首地址, 效果如图 4-10 所示。



图 4-10 strupr()实现小写转为大写

【实现过程】

定义 `char` 型数组变量 `str` 并进行初始化, 然后用 `cout` 输出经函数 `strupr()` 处理后的返回值, 其代码如下:

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     char str[]="Hello World";           //目标字符串
07     cout<<strupr(str)<<endl;           //将 str 所有的元素变为大写
08
09     return 0;
10 }
```

【代码解析】

第 06 行定义并初始化 `str`, 第 07 行输出 `strupr()` 函数的返回值。此处要重复一个知识点, 即 `char` 型数组可以当作 `char` 指针使用, 这就是为什么该函数及前面的所有函数在使用 `char` 型指针时可以用 `char` 型数组代替。如果还有不明白, 请返回实例 046 查看。



实例 060 使用 C++字符串类 string 打印字符串

【实例描述】

实例 050 介绍了如何输出字符数组, 本实例介绍如何打印 C++字符串类 `string` 的值, 效果如图 4-11 所示。

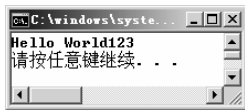


图 4-11 打印 C++字符串类 string



【实现过程】

定义 `string` 类型变量 `str` 并进行初始化，然后利用 `cout` 输出，具体代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     string str="Hello World123";           //string 变量
08     cout<<str<<endl;                       //string 型变量的输出
09
10     return 0;
11 }
```

【代码解析】

在 `string` 类中有成员函数重载了运算符`=`，所以可以使用第 07 行所示的格式进行赋值。同理，`string` 类也重载了运算符`<<`进行输出，格式如第 08 行所示。



注意：对于 `string` 类定义的变量初始化，还可以使用构造函数实现，格式如下：

```
string str("HelloWorld123");
```



实例 061 string 与 C 风格字符串转换

【实例描述】

字符串的表达方式有两种，即 `string` 类和 C 风格的字符数组。在不同的应用场合，所需要的字符串类型也不同，因此也就有场合需要两种字符串类型一起参与，那么如何进行 `string` 与 C 风格字符串的转换？本实例旨在实现此功能，效果如图 4-12 所示。



图 4-12 string 与 C 风格字符串的相互转换

【实现过程】

`string` 字符串转换为 C 风格字符串需要利用 `string` 类成员函数 `c_str()`。而 C 风格字符串转换为 `string` 字符串可以直接利用运算符`=`。首先介绍 `c_str()`函数的原型：

```
const value_type *c_str( ) const;
```

它的返回值类型为 `const char*`，所以定义的 C 风格字符串需要用 `const char*`指针指向，变量名为 `ch`。`string` 类型变量为 `str`，值为 `Hello`，`ch` 指向的字符串内容为 `World`。代码实现如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
```



```

07     string str="Hello";
08     const char *ch;
09     ch=str.c_str();           //string 转为 C 风格字符串
10     cout<<ch<<endl;
11
12     ch="World";
13     str=ch;                   //C 风格字符串转为 string
14     cout<<str<<endl;
15     return 0;
16 }

```

【代码解析】

第 07、08、12 行定义并初始化变量 `str` 和 `ch`。第 09、10 行将 `string` 类型转换为 C 风格字符串 `ch`，并输出。第 13、14 行将 `ch` 转换为 `str`，并输出。



实例 062 比较两个 string 字符串

【实例描述】

实例 057 利用函数 `strcmp()` 完成比较两个 C 风格字符串的操作。本实例演示如何比较两个 `string` 字符串，有两种方式，一种是直接利用运算符 `!=`、`>=`、`<=`，另一种方式是利用 `string` 类的成员函数 `compare()`，运行效果如图 4-13 所示。



图 4-13 比较两个 string 字符串

【实现过程】

定义 3 个字符串 `str1`、`str2`、`str3`，分别初始化为 `Hello`、`World`、`HELLO`。整型变量 `compare_result` 用于存放比较结果，代码如下：

```

01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     string str1="Hello";
08     string str2="World";
09     string str3="HELLO";
10     int compare_result;
11     if(str1!=str2)           //方式 1
12         cout<<"str1 不等于 str2"<<endl;
13
14     if(bool(compare_result=str3.compare(str1))) //不相等（方式 2）
15         cout<<"str1 不等于 str3"<<endl;
16     return 0;
17 }

```

【代码解析】

第 07~10 行定义并初始化各类变量，第 11、12 行使用第一种方式比较 `str1` 和 `str2`，并输出结果。第 14、15 行使用第 2 种方式比较 `str1` 和 `str3`，并输出结果。

其中，第 14 行是 `compare()` 为 `string` 类的成员函数，它的应用格式如下：

输出结果=字符串 1.compare(字符串 2)；



注意：C++字符串中的内容区分大小写，大写字母不等于其小写字母。



实例 063 查找 string 的某个元素

【实例描述】

string 类中有多个成员函数可以实现在目标字符串中查找某个元素，分别有 find()、find_first_not_of()、find_first_of()、find_last_not_of()、find_last_of()和 rfind()函数。对于不同的函数，有不同的查找规则，另外，同样名字的查找函数还被重载。下面只介绍每个查找函数的其中一种形式，它们的具体功能描述如下。

(1) find()

```
int find(char c, int pos=0) const;
```

从 pos 位置开始从前向后查找字符 c 在当前字符串中的位置，默认 pos 的值为 0。

(2) find_first_not_of()

```
int find_first_not_of(char c, int pos=0) const;
```

从 pos 位置开始从前向后查找字符 c 在当前字符串中第一个不匹配的字符位置。

(3) find_first_of()

```
int find_first_of(char c, int pos = 0) const;
```

从 pos 位置开始从前向后查找字符 c 在当前字符串中第一次出现的位置。

(4) find_last_not_of()

```
int find_last_not_of(char c, int pos = npos) const;
```

从 pos 位置开始从后向前查找字符 c 在当前字符串中第一个不匹配的字符位置。

(5) find_last_of()

```
int find_last_of(char c, int pos = npos) const;
```

从 pos 位置开始从后向前查找字符 c 在当前字符串中第一次出现的位置。

(6) rfind()

```
int rfind(char c, int pos = npos) const;
```

从 pos 位置开始从后向前查找字符 c 在当前字符串中的位置。

效果如图 4-14 所示。

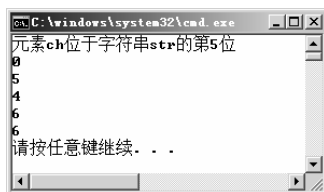


图 4-14 利用多种方式查找 string 中的某个元素

【实现过程】

定义 string 字符串 str，并初始化为 Hello;;，定义字符变量 ch，初始化为;。之后输出上述



各种查找字符 ch 的位置函数，其代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     string str="Hello;;";
08     char ch=';';
09     cout<<"元素 ch 位于字符串 str 的第"<<str.find(ch)<<"位"<<endl;
10     cout<<str.find_first_not_of(ch)<<endl; //向后找不是 ch 字符的第一个位置
11     cout<<str.find_first_of(ch)<<endl;      //第一个为 ch 的位置
12     cout<<str.find_last_not_of(ch)<<endl;   //从后向前找不是 ch 字符的位置
13     cout<<str.find_last_of(ch)<<endl;       //从后向前找是 ch 字符的位置
14     cout<<str.rfind(ch)<<endl;             //从后向前找 ch 的首个位置
15
16     return 0;
17 }
```

【代码解析】

第 07、08 行定义程序所需变量 str 和 ch。第 09~14 行输出多种位置查找函数返回结果。



注意：string 类的查找函数还有其他格式，此处不一一列举，可以查看 string.h 头文件。



实例 064 使用成员函数检测 string 字符串是否非空

【实例描述】

本实例检测字符串是否为空，可以利用成员函数 empty()实现。如果为空，返回布尔值 true；反之，返回 false。该函数的格式如下：

string 变量名.empty();

效果如图 4-15 所示。



图 4-15 empty()函数检测字符串是否非空

【实现过程】

定义 string 变量 str，但不被初始化，所以 str 应该为空，判断代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     string str;
```



```

08         if(str.empty())                //为空
09             cout<<"str 为空"<<endl;
10         else                            //不为空
11             cout<<"str 不为空"<<endl;
12         return 0;
13     }

```

【代码解析】

第 07 行定义 `string` 变量 `str`，第 08~11 行根据 `str` 的内容进行判断，并做相应的输出。

**实例 065 获取 string 字符串的长度****【实例描述】**

C 风格字符串利用函数 `strlen()` 获取其长度，相应的 `string` 字符串运用成员函数 `length()` 获取其长度。它的格式如下：

长度变量=字符串.length();

效果如图 4-16 所示。

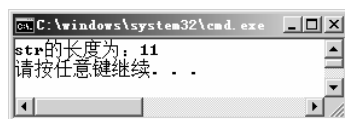


图 4-16 运用 `length()` 获取字符串长度

【实现过程】

定义字符串 `str`，赋值为 `Hello World`；定义长度变量 `length`，初始化为 0，代码如下：

```

01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     string str="Hello World";
08     int length=0;                //长度
09     cout<<"str 的长度为: "<<str.length()<<endl;    //string 型字符串的长度输出
10     return 0;
11 }

```

【代码解析】

第 07、08 行定义并初始化 `str` 和 `length`，第 09 行获取 `str` 的长度并输出。



注意：不同形式的字符串变量，它们的长度获取函数也不同。

**实例 066 提取 string 字符串的子串****【实例描述】**

在字符串的应用实例中，有时候需要提取它们的子串。本实例针对 `string` 型字符串演示子串的提示函数 `substr()`，它的原型如下：

```
basic_string substr(size_type _Off = 0, size_type _Count = npos) const;
```

其中，第 1 个参数是子串的起始位置，第 2 个参数是子串的长度。它的格式如下：



子串变量=源字符串.sub_str(起始位置, 子串长度);

效果如图 4-17 所示。



图 4-17 提取字符串的子串

【实现过程】

定义字符串 str 和子串 sub_str, 并进行初始化。提取 str 从位置 0 开始、长度为 5 的子串赋给 sub_str, 代码如下:

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 //提取 string 字符串的子串
06 int main()
07 {
08     string str="Hello World";    //源串
09     string sub_str="";           //子串
10     sub_str=str.substr(0,5);      //string 型的提取子串方法
11     cout<<sub_str<<endl;        //输出子串
12     return 0;
13 }
```

【代码解析】

第 08、09 行定义 str 和 sub_str, 第 10 行运用函数 substr()提取子串, 并由第 11 行输出子串。



实例 067 把两个 string 字符串相加

【实例描述】

C 风格字符串相加可利用 strcat()函数实现, 在 string 字符串中可以利用运算符(+)得到相加结果, 格式如下:

string 类型字符串 1= string 类型字符串 2+ string 类型字符串 3;

运行效果如图 4-18 所示。

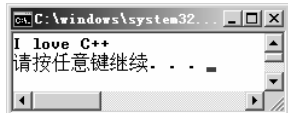


图 4-18 两个 string 字符串相加

【实现过程】

定义 3 个 string 字符串 str1、str2 和 str3, 并进行初始化。将 str1 和 str2 相加, 结果赋给 str3。



代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     string str1="I love";
08     string str2=" C++";
09     string str3="";           //相加后的字符串
10     str3=str1+str2;          //字符串相加
11     cout<<str3<<endl;       //输出
12     return 0;
13 }
```

【代码解析】

第 07~09 行定义并初始化 3 个 string 变量，第 10 行实现两个字符串的相加，由第 11 行输出。



注意：为了更好地观察相加结果，特意将 str3 初始化为空。



实例 068 string 字符串与 C 风格字符串相加

【实例描述】

实例 067 是处理两个 string 字符串的相加，本实例针对 string 字符串与 C 风格字符串的相加操作。对于本实例的实现，可以采用将其中一方转化为另一方，再完成相加。事实上，两个字符串是可以使用运算符 (+) 直接相加的。它的目的是实现两者的连接，返回值类型为 string。其格式如下：

string 型变量 1=string 型变量 2+C 风格字符串 3;

还可以利用成员函数 `append()` 完成相加，该函数的重载函数有很多，此处只介绍最简单的一种，原型如下：

```
string &append(const char *s);
```

它的作用是将 s 连接到目标 string 变量中，运用上述两种方式实现不同类型的字符串相加，效果如图 4-19 所示。



图 4-19 string 字符串与 C 风格字符串相加

【实现过程】

定义 4 个字符串变量 str 和 str1 (string 类型)、ch 和 ch1 (C 风格字符串)，并进行初始化。将 str 和 ch 相加赋值给 str1，之后 ch1 被连接到 str1 中，其代码如下：

```
01 #include <iostream>
02 #include <string>
```



```

03 using namespace std;
04
05 int main()
06 {
07     string str="Hello";           //string 型字符串
08     char *ch="World";             //C 风格字符串 1
09     char *ch1="C++";              //C 风格字符串 2
10     string str1="";
11     str1=str+ch;                   //两种字符串的相加方式 1
12     cout<<str1<<endl;
13     str1.append(ch1);              //相加方式 2
14     cout<<str1<<endl;
15     return 0;
16 }

```

【代码解析】

第 07~10 行定义并初始化 4 个所需变量，第 11 行利用运算符(+)实现两种字符串的相加，第 13 行利用 append()函数实现两种风格字符串的连接。



注意：string 字符串与 C 风格字符串运用运算符(+)相加后的返回值类型为 string。



实例 069 string 字符串与单字符相加

【实例描述】

既然可以实现 string 字符串与 C 风格字符串的相加，同样，也可以利用运算符(+)完成 string 字符串与单字符的相加，效果如图 4-20 所示。

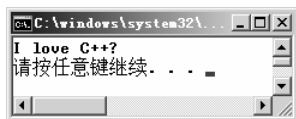


图 4-20 string 字符串与单字符相加

【实现过程】

定义 string 字符串 str 和字符变量 ch，实现两者的相加，并赋给 string 字符串 str1，然后输出，代码如下：

```

01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     string str="I love C++";       //字符串
08     char ch='?';                   //字符变量
09     string str1="";                //相加结果
10     str1=str+ch;                   //加法
11     cout<<str1<<endl;
12     return 0;
13 }

```



【代码解析】

第 07~09 行定义上述 3 个变量, 并进行初始化, 第 10 行实现 string 字符串 str1 与字符 ch 的相加, 第 11 行输出结果字符串 str1。



实例 070 string 字符串替换

【实例描述】

字符串不仅要被连接, 有时候还需要被替换内容。本实例针对 string 字符串的替换, 使用成员函数 replace()。该函数的重载函数也有很多, 此处只介绍最简单的, 格式如下:

```
string &replace(int p0, int n0, const string &s);
```

目的是删除从 p0 从前向后数的 n0 个字符, 然后在 p0 处插入字符串 s。

运行效果如图 4-21 所示。

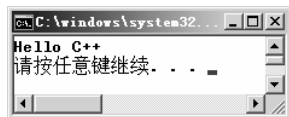


图 4-21 string 字符串替换

【实现过程】

定义两个字符串 str 和 str2, 并且进行初始化。从 str 的第 6 位开始删除其 5 位字符, 并在 str 的第 6 位插入字符串 str2。代码如下:

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 int main()
06 {
07     string str="Hello World";           //被替换字符串
08     string str2="C++";                 //替换字符串
09     str.replace(6,5,str2);              //替换
10     cout<<str<<endl;                  //输出结果
11     return 0;
12 }
```

【代码解析】

第 07、08 行定义并初始化 string 字符串 str 和 str2, 第 09 行实现字符串的替换, 结果由第 10 行输出。



注意: string 字符串的替换子串还可以是 C 风格字符串, 因为 replace() 的重载函数有一个是如下格式:

```
string &replace(int p0, int n0, const char *s);
```

它的第 3 个参数为常类型字符指针。

第 5 章 内存与指针

指针源于 C 语言，并由 C++ 语言继承。指针可谓 C 语言编程的一大难点，也是重点。由指针定义的变量被称为指针变量，它指内存的某个存储单元（以字节为单位），有指针出现的地方都会涉及内存的概念。本章的实例教会大家如何将数组和指针一起应用、动态申请内存、如何正确申请并释放指针及指针的基本加减法。由于涉及计算机的内存，所以指针的应用一定要慎重。由此可见，本章的学习也相当重要，稍有不慎就可能造成计算机系统的崩溃。



实例 071 坐标指针（数组+指针）

【实例描述】

将指针与数组结合在一起使用，可以形成两种变量，即数组指针和指针数组。这两种变量的定义格式如下：

```
类型标识符 (*变量名)[个数];           // 数组指针
类型标识符 *变量名[个数];             // 指针数组
```

二维数组的行列下标与直角坐标系中的坐标类似。本实例模拟坐标指针指向二维数组的某个元素（也即某块内存），并输出该内存所存储的值，效果如图 5-1 所示。

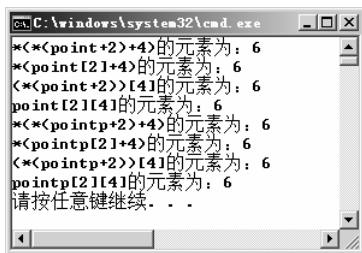


图 5-1 坐标指针

【实现过程】

定义二维数组 `array_071[5][6]`、数组指针 `point[6]`（该数组首元素指向 `array_071`，本质为指针）和指针数组 `pointp[5]`（表示每个指针指向 `array_071` 的一行，本质是数组）。利用变量 `point` 和 `pointp` 使用不同的方式输出 `array_071` 中第 2 行第 4 列的元素值。其代码如下：

```
01 #include <iostream>
02 using namespace std;
03 #include <stdio.h>
04
05 int main()
06 {
07     int array_071[5][6];           // 定义二维数组
08     int (*point)[6]=array_071;    // 数组指针
09     int *pointp[5];               // 指针数组
10     for(int i=0;i<5;i++)
11     {
12         for(int j=0;j<6;j++)
13             array_071[i][j]=i+j;  // 初始化
14     }
15     cout<<"* (* (point+2)+4) 的元素为: "<<* (* (point+2)+4)<<endl;
16                                     // 数组指针访问内存
17     cout<<"* (point[2]+4) 的元素为: "<<*(point[2]+4)<<endl;
```



```

17      cout<<" (* (point+2)) [4]的元素为: "<<" (* (point+2)) [4]<<endl;
18      cout<<"point[2][4]的元素为: "<<"point[2][4]<<endl;
19
20      for(int i=0;i<5;i++)
21          pointp[i]=array_071[i];                      //指针数组赋值
22      cout<<"* (* (pointp+2)+4)的元素为: "<<"* (* (pointp+2)+4)<<endl;
                                           //指针数组访问内存
23      cout<<"* (pointp[2]+4)的元素为: "<<"* (pointp[2]+4)<<endl;
24      cout<<" (* (pointp+2)) [4]的元素为: "<<" (* (pointp+2)) [4]<<endl;
25      cout<<"pointp[2][4]的元素为: "<<"pointp[2][4]<<endl;
26      return 0;
27  }

```

【代码解析】

第 07 行定义数组 `array_071[5][6]`，第 08 行将 `array_071` 的首元素赋给 `point` 数组的首元素。第 10~14 行初始化 `array_071`，第 15~18 行使用数组指针以不同的方式输出第 2 行第 4 列元素的值。第 20、21 行将 `array_071` 每一行的首地址赋给 `pointp` 的每个元素，第 22~25 行用指针数组以不同的访问方式输出第 2 行第 4 列元素的值。

由上可知，数组指针和指针数组用于表示数组的某个坐标的方式有如下几种方式（以 `p` 代表上述的 `point` 和 `pointp`，`i` 和 `j` 分别表示行和列）：

```

* (* (point+i)+j)
* (point[i]+j)
(* (point+i)) [j]
point[i][j]

```



注意：指针数组和数组指针的区别在于，指针数组的每个元素都是指针，而数组指针只有整个数组为指针。



实例 072 强制修改常量的值

【实例描述】

在第 2 章的介绍中，知道常量的值是不可以被修改的。但是由于指针的使用，使得常量值的改变成为可能。本实例实现如何采用强制手段修改常量的值，它所修改的常量也只是由 `const` 定义的。通过取地址运算符指向该常量的内存，然后改变该内存的内容，实现常量值的强制修改。运行效果如图 5-2 所示。

编译器是从来不考虑这个常量从地址被修改的情况的，即一开始定义后，就不会被重新从地址取值。

【实现过程】

本实例以整型常量为例，定义整型指针 `point` 及整型数据 `changliang`（由 `const` 定义的常量）。将 `point` 指向 `changliang` 所在的内存，通过改变内存的内容，实现强制性的改变常量值。代码如下：

```

01  #include <iostream>
02  using namespace std;
03
04  int main()

```

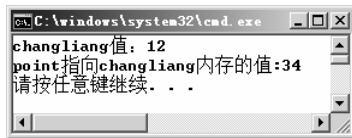


图 5-2 不同类型的变量界面



```

05 {
06     int *point;                //指针
07     const int changliang=12;    //常量
08     point=(int*)&changliang;    //取 changliang 的地址
09     *point=34;                 //改变指针指向内存的值
10     cout<<"changliang 值: "<<changliang<<endl;
11     cout<<"point 指向 changliang 内存的值:"<<*point<<endl;
12     return 0;
13 }

```

【代码解析】

第06、07行定义整型指针变量 `point` 和整型常量 `changliang`。第08行将 `point` 指向 `changliang` 所在内存地址，此处需要用 `int*` 做类型强制转换（因为 `&changliang` 的类型为 `const int*`，而 `point` 的类型为 `int*`，所以需要强制转换类型）。第09行改变指针指向内存 `changliang` 的值，第10、11行分别输出 `changliang` 和 `point` 指向内存的值。



注意：由运行结果可以看出，强制改变常量的值并没有真正实现。它只是改变了 `changliang` 所占内存的值，并没有改变 `changliang` 的值。



实例 073 通信录（动态申请内存）

【实例描述】

当用指针指向一段内存时，事先并不知道需要申请多少，此时动态申请内存就显得很必要。本实例以通信录的方式介绍动态申请内存的知识。C++语言中有多个动态申请内存的方式，下面分别介绍。

（1）malloc 方式

当申请一维内存时，它的格式如下：

类型标识符 *变量名；

变量名=(类型标识符*)malloc(sizeof(类型标识符)*数组大小)；

在使用完该方式申请的内存后，必须用 `free()` 函数及时释放，格式如下：

`free(变量名)；`

变量名=NULL；

当申请二维内存时，它的格式如下：

类型标识符 **变量名；

变量名=(类型标识符**)malloc(sizeof(类型标识符)*数组行大小)；

for(int i=0;i<数组行大小;i++)

变量名[i]=(类型标识符*)malloc(sizeof(类型标识符)*数组列大小)；

它的释放格式如下：

`free(变量名)；`

变量名=NULL；

（2）new 方式

当申请一维内存时，它的格式如下：

类型标识符 *变量名；

变量名=new 类型标识符[数组大小]；



在使用完该方式申请的内存后，必须用 `delete()` 函数及时释放，格式如下：

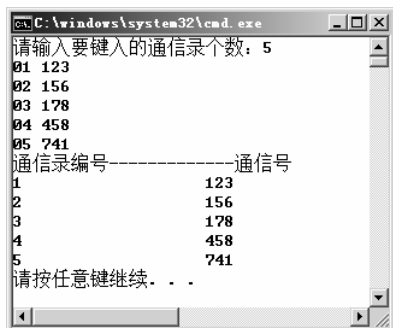


图 5-3 动态申请内存

```
delete[] 变量名;
```

```
变量名=NULL;
```

当申请二维内存时，它的格式如下：

```
类型标识符 **变量名;
```

```
变量名=new 类型标识符*[数组行大小];
```

```
for (int i=0;i<数组行大小;i++)
```

```
    变量名[i]= new 类型标识符[数组列大小];
```

它的释放格式如下：

```
delete[] 变量名;
```

```
变量名=NULL;
```

本实例效果如图 5-3 所示。

【实现过程】

定义变量 `n` 和 `m`，分别表示通信录中的联系人编号及其对应的通信号。此处预先定义 `m` 的值为 2，`n` 的值从屏幕输入。定义二维内存 `memo`，用以存放输入的通信录信息。最后输出该通信录内容，具体代码如下：

```
01 #include <malloc.h>
02 #include <iostream>
03 using namespace std;
04
05 int main()
06 {
07     int n; //行，通信录中有几个联系人
08     int m=2; //列，名字和其对应的通信号
09     cout<<"请输入要键入的通信录个数: ";
10     cin>>n;
11     int **memo; //通信录
12     memo=(int**)malloc(sizeof(int *)*n); //申请内存
13     for (int k=0;k<n;k++)
14         memo[k]=(int *)malloc(sizeof(int)*m);
15
16     for (int i=0;i<n;i++)
17     {
18         for (int j=0;j<m;j++)
19             cin>>memo[i][j]; //输入通信录内容
20     }
21     cout<<"通信录编号-----通信号"<<endl;
22     for (int i=0;i<n;i++)
23     {
24         for (int j=0;j<m;j++)
25             cout<<memo[i][j]<<" "; //输出
26         cout<<endl;
27     }
28     free(memo); //释放内存
29     memo=NULL;
30     return 0;
31 }
```

【代码解析】

第 07、08 行定义通信录 `memo` 的行和列值，第 10 行键入 `n` 的值。第 11 行定义 `memo`，并由第 12~14 行动态申请内存 `memo`。第 16~20 行获取通信录信息，第 22~27 行输出通信录信



息。第 28 行释放内存 memo。



注意：在使用 malloc()动态申请内存时，必须包含头文件 malloc.h。



实例 074 万能箱子 (void*)

【实例描述】

指针有一个变量类型为 void*, 它可以指向任意类型的数据, 即任意类型的指针都可以赋值给 void*。但反过来, void*不可以赋值给其他类型的指针, 除非采用强制类型转换。本实例以万能的箱子演示 void*指针如何被任意类型的指针变量赋值, 效果如图 5-4 所示。

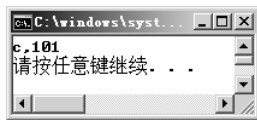


图 5-4 万能的箱子 (void*)

【实现过程】

定义变量 a (char 型)、b (整型), 并且进行初始化。函数 decrease()用于将变量的值减 1, 该函数的输入参数是 void*类型, 因此可以实现任意类型变量的减 1 功能 (本实例只实现对 char 型和 int 型变量的功能操作), 所有的代码如下:

```
01 #include <iostream>
02 using namespace std;
03
04 void decrease(void* data, int psize)           //减法函数
05 {
06     switch(psize)
07     {
08         case 1:                               //字符
09             char* pchar;
10             pchar=(char*)data;                //void*转换为 char 型
11             --(*pchar);                       //减法
12             break;
13         case sizeof(int):                     //整型
14             int *pint;
15             pint=(int*)data;                  //void*转换为 int 型
16             --(*pint);                       //减法
17     }
18 }
19 int main()
20 {
21     char a='d';
22     int b=102;
23     decrease(&a,sizeof(a));                  //char 型减法
24     decrease(&b,sizeof(b));                  //int 型减法
25     cout<<a<<" "<<b<<endl;
26
27     return 0;
28 }
```

【代码解析】

第 04~18 行是函数 decrease()的实现, 第 1 个参数是实现减 1 的变量, 第 2 个参数是该变量的字节大小。第 10、15 行对 void*指针类型变换为对应的指针类型。第 11、16 行实现指针指向内容减 1。



第 19~28 行是 main() 函数的实现, 第 21、22 行定义变量 a 和 b, 并进行初始化。第 23、24 行调用 decrease() 函数, 最后由第 25 行输出结果。



注意: 经检验, void* 指针赋值给其他类型指针也需要类型转换, 如第 10、15 行。



实例 075 指向结构体变量的指针

【实例描述】

实例 074 演示了指向单个变量的指针应用, 本实例讲解指向结构体变量的指针使用。当指针指向结构体变量时, 如何获取结构体中的各个元素? 如何让指针变量指向结构体的单个成员? 这两个问题将在本实例展开介绍, 运行效果如图 5-5 所示。

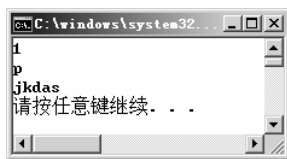


图 5-5 指向结构体变量的指针

【实现过程】

定义名为 newtype 的结构体, 成员变量有整型变量 a、字符变量 ch 和 string 型变量 str, 代码如下:

```
01 struct newtype
02 {
03     int a;
04     char ch;
05     string str;
06 };
```

在实例的 main() 函数中实现指针指向结构体变量, 需要定义 newtype 变量 new_type 和其类型的指针 new_type_point, 并初始化变量 new_type 中的各个成员变量值, 之后, 通过 cout 计算出前两个成员变量的值。最后为了明确单个成员变量指针的使用, 再定义 string 型指针 str_point, 以指向结构体变量 new_type 的 str 成员, 并输出其值, 代码如下:

```
01 int main()
02 {
03     newtype *new_type_point, new_type;
04     new_type_point=&new_type;
05     new_type.a=1;
06     new_type.ch='p';
07     new_type.str="jkdas";
08     cout<<(*new_type_point).a<<endl;           //指向结构体成员 a
09     cout<<new_type_point->ch<<endl;           //指向结构体成员 a
10     //new_type_point.a;                        //错误形式
11     string *str_point;
12     str_point=&new_type.str;                   //指向结构体中元素 ch 的指针
13     cout<<*str_point<<endl;
14     return 0;
15 }
```

【代码解析】

第 03 行定义结构体变量 new_type 和其指针变量 new_type_point, 第 04 行使 new_type_point 指向 new_type 变量。第 05~07 行初始化变量 new_type 各个成员值。第 08、09 行分别以两种方式获取变量 new_type 中前两个成员的值, 第 10 行读取成员 a 的方式是错误的。第 11 行定义



string 型指针 `str_point`，在第 12 行指向 `new_type` 的第 3 个成员 `str`，并由第 13 行输出指针指向的内容。



注意：在指针应用场合，&是取地址运算符，*是取该地址指向内存的内容，->是由指针直接取内存内容。



实例 076 打印内存数据（char 打印 1 字节）

【实例描述】

对于不同的数据类型，在内存中所占的字节数也是不同的。`char` 型占 1 字节，`int` 型占 4 字节（对于不同的计算机，也有占 2 字节的情况）。当打印变量时，系统会根据变量的类型，每隔 `sizeof(数据类型)` 字节就打印。比如，打印占 4 字节的 `int` 型变量，它在内存的存储情况如图 5-6 所示。

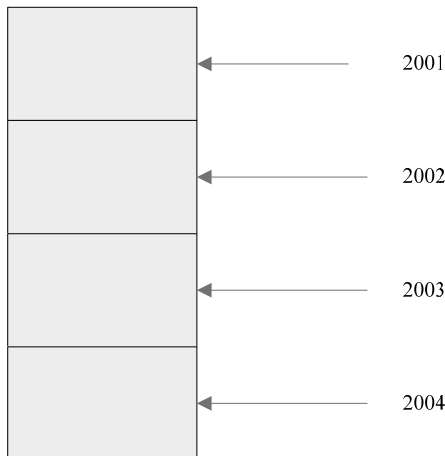


图 5-6 一字节的打印内存数据

`int` 型变量存于地址 2001 到 2004 这 4 字节中。当打印时，只会一次性输出该变量保存于 2001~2004 内存的值，却不会分别打印这 4 字节各自的数据。本实例要实现一字节一字节地打印内存数据，效果如图 5-7 所示。

【实现过程】

定义 `int` 型变量 `aa`，赋值为 1234。定义字符数组 `ch[4]`，用于存储 `aa` 的每字节的数据，并且以 `char` 类型打印 `aa` 每字节的数据，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int aa=1234;           //整型
07     char ch[4];           //char 型数组
```

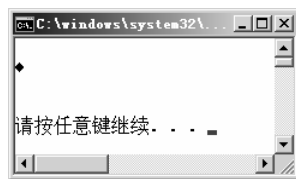


图 5-7 逐字节打印内存数据



```

08     for(int i=0;i<sizeof(int);i++)
09         cout<<((char*)&aa)[i]<<endl;    //一字节一字节地输出数据
10     return 0;
11 }

```

【代码解析】

第 06、07 行定义变量 `aa` 和数组 `ch[4]`，第 08、09 行分 4 次访问 `aa` 的 4 字节中的数据，并输出。



实例 077 错误地释放指针导致程序崩溃

【实例描述】

当用 `new` 和 `malloc` 函数动态申请内存时，最后需要释放指针和内存。但在释放指针前必须先判断当前指针是否指向内存的首地址，如果没有，将会导致程序崩溃。本实例即模拟如何正确释放指针，实例运行效果如图 5-8 所示。

【实现过程】

动态申请内存 `aa`，定义整型指针 `copy_aa`。`copy_aa` 保存 `aa` 内在的首地址，利用移动内存指针实现内存中各元素的赋值，并且输出，代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     int *aa=new int[10];
07     int *copy_aa=aa;
08     cout<<hex;    //十六进制输出
09     for(int i=0;i<10;i++)
10     {
11         *aa=i;
12         cout<<aa<<" "<<*aa<<endl;    //输出指针指向内存值
13         aa++;
14     }
15     cout<<endl<<"当前指针指向内存的地址为："<<aa<<endl;
16     aa=copy_aa;
17     delete[] aa;
18     aa=NULL;
19 }

```

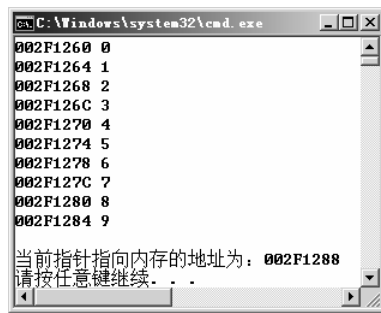


图 5-8 错误释放指针后果

【代码解析】

第 06 行动态申请内存 `aa`，长度为 10。第 07 行定义指针 `copy_aa`，其指向 `aa` 的首地址。第 08 行采用十六进制输出结果。第 09~14 行的 `for` 循环完成每个元素的赋值，先赋值和输出，再将指针移向下一个元素（第 13 行）。第 15 行输出当前指针指向哪里，如图 5-8 所示，指针已超出了内存块 `aa` 的边界。

此时，如果没有第 16 行将当前指针返回到内存块的首地址，运行程序后会出现如图 5-9 所示的错误。

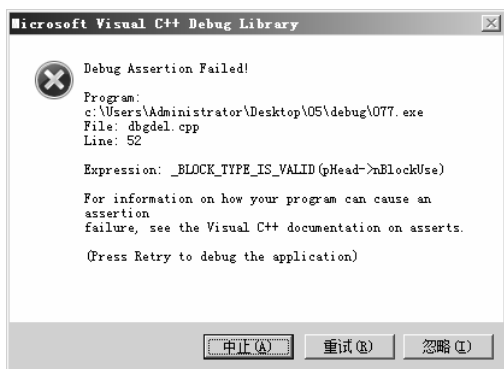


图 5-9 错误释放指针的提示



注意：当指向内存块的指针已超出其边界，一定要将指针指回首地址。



实例 078 防止野指针的代码

【实例描述】

本实例实现如何防止野指针。首先了解野指针的成因有以下 3 种。

(1) 指针创建后没有初始化

解决方法是初始化为 NULL 或者指向合法的内存。

(2) 指针在使用 free() 和 delete() 函数释放指向的内存时，但没有释放指针

当程序再一次调用该指针后，该野指针会乱指地址。但有些内存是受保护并且不可读写的，严重时会导致程序崩溃。解决方法是在释放内存后，也将指针设为 NULL。

(3) 指针的创建和操作不在同一个作用域中。

本实例针对上述 3 种情况展开，效果如图 5-10 所示。

【实现过程】

定义整型指针 a，动态申请内存 b[2]，整型变量 aa，指针 a 指向内存 aa。之后输出 a 和 b 的地址值和指向内存值。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     int *a;
07     int *b=new int[2];           //new 方式申请内存
08     int aa=10;
09     a=&aa;                       //取地址
10     *b=2;
11     cout<<a<<" "<<*a<<endl;
12     cout<<b<<" "<<*b<<endl;
13     delete[] b;                 //释放内存
14     if(b!=NULL)
15         b=NULL;                //释放指针
16 }
```

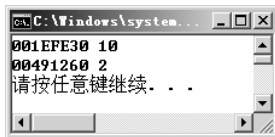


图 5-10 防止野指针问题



【代码解析】

本实例代码最重要的是第 9 行、第 13~15 行。在初始化指针时，将 a 指向合法内存 aa。当用 delete 释放指针时，必须判断指针 b 是否为 NULL。如果不为 NULL，设为 NULL。



注意：在今后使用指针的过程中，一定要严格按照上述格式书写，可以在遇到 bug 时优先排除野指针的可能性。



实例 079 统计数据（综合）

【实例描述】

本实例实现统计每个国家的男女人数，一共有 20 个人，但来自不同的国家。其中男性用 0 表示，女性用 1 表示。来自的国家分为 A 国、B 国和 C 国，各自用编号 0、1、2 表示，结果统计来自 3 个国家的男女性各多少人，一共多少人？效果如图 5-11 所示。

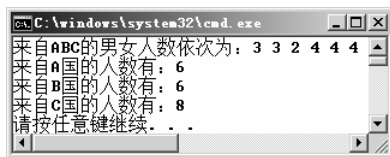


图 5-11 统计数据

【实现过程】

对于本实例的实现，定义数组 people[ROW][COL]，ROW 和 COL 使用宏定义设置为 20 和 2。动态申请内存 num 存储来自 A、B、C 的男女人数，将其初始化为 0。具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 #define ROW 20
05 #define COL 2
06
07 void main()
08 {
09     int people[ROW][COL]={0,0},{1,1},{1,2},{0,2},{0,1},{0,0},
10     {1,0},{1,2},{1,1},{1,0},{0,2},{0,1},{0,0},
11     {0,2},{1,2},{1,1},{1,0},{1,2},{0,2},{1,1}};
12     int *num=new int[6];           //分别为 A、B、C 的男女
13     for(int i=0;i<6;i++)
14         num[i]=0;
15     for(int i=0;i<ROW;i++)
16     {
17         switch(*( *(people+i)+1))
18         {
19             case 0:                //A 国
20                 if(*( *(people+i)+0)==0) //男
21                     (*num)++;
22                 else
23                     (*(num+1))++;
24                 break;
25             case 1:                //B 国
26                 if(*( *(people+i)+0)==0) //男
27                     (*(num+2))++;
28                 else
29                     (*(num+3))++;
30                 break;
```



```

31         case 2:                                     //C 国
32             if (*(people+i)+0)==0)                 //男
33                 (*(num+4))++;
34             else
35                 (*(num+5))++;
36             break;
37         default:
38             break;
39     }
40 }
41 cout<<"来自 ABC 的男女人数依次为: ";
42 for(int i=0;i<6;i++)
43     cout<<*(num+i)<<" ";
44 cout<<endl;
45 cout<<"来自 A 国的人数有: "<<(*num)+(* (num+1))<<endl;
46 cout<<"来自 B 国的人数有: "<<(* (num+2))+(* (num+3))<<endl;
47 cout<<"来自 C 国的人数有: "<<(* (num+4))+(* (num+5))<<endl;
48 }

```

【代码解析】

第 09~14 行定义 `people` 和 `num` 内存并进行初始化,第 15~40 行判断当前指针指向的内存值来自哪个国家,并且判断是男性还是女性。最后由第 42~47 行输出来自 3 个国家的男女人数及总的人数。



实例 080 指针应用常见问题(传送的是地址还是值)

【实例描述】

本实例利用交换两个变量的值讨论指针应用时常见的问题——传送值还是地址。它的应用范围是通过定义函数 `change()` 交换两个变量的值。传值的函数形式为:

```
void change(int a, int b);
```

传送地址的函数形式为:

```
void change(int *a, int *b);
```

本例运行效果如图 5-12 所示。

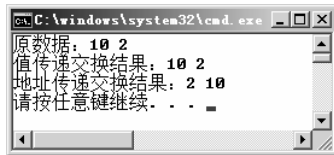


图 5-12 函数传送值或地址的交换结果

【实现过程】

定义两个函数(值传递和地址传递),都用于交换两个变量的值。值传递函数 `change(int a, int b)` 的两个参数都是值变量,地址传递函数 `change(int *a, int *b)` 的参数都是指针。在执行函数的过程中,定义两个变量 `aa` 和 `bb`,两个指针 `point1` 和 `point2`,分别指向 `aa` 和 `bb`。分别调用两个 `change()` 函数,最后输出 `aa` 和 `bb` 的值,代码如下:

```

01 #include <iostream>
02 using namespace std;
03
04 void change(int a, int b)                //值传递
05 {
06     int temp;
07     temp=a;
08     a=b;
09     b=temp;
10 }
11 void change(int *a, int *b)             //地址传递

```



```

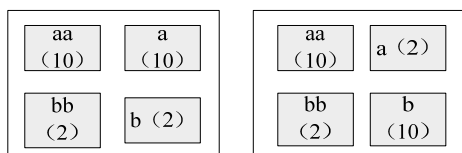
12 {
13     int temp;
14     temp=*a;
15     *a=*b;
16     *b=temp;
17 }
18 int main()
19 {
20     int aa=10,bb=2;
21     int *point1=&aa;
22     int *point2=&bb;
23     cout<<"原数据: "<<aa<<" "<<bb<<endl;
24     change(aa,bb);
25     cout<<"值传递交换结果: "<<aa<<" "<<bb<<endl;
26     change(point1,point2);
27     cout<<"地址传递交换结果: "<<aa<<" "<<bb<<endl;
28     return 0;
29 }

```

【代码解析】

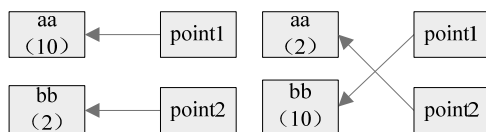
第 04~10 行是值传递函数的定义体，第 11~17 行是地址传递函数的定义体。第 18~29 行是执行函数 main() 的定义体，第 20 行定义变量 aa 和 bb。第 21~22 行定义指针变量 point1（指向 aa），point2（指向 bb）。

第 24、26 行分别调用这两个 change() 函数，最后输出结果。如图 5-13 所示是值传递函数过程实现，如图 5-14 所示是地址传递函数的过程实现。



变换前

变换后



变换前

变换后

图 5-13 值传递实现原理

图 5-14 地址传递实现原理

由上可见，值传递只是将函数参数 a 和 b 的值做了交换，并没有实现 aa 和 bb 的交换。地址传递是在传地址，所以最后可以实现 aa 和 bb 的交换。



注意：虽然这两个函数名相同，但是因为参数类型不一样，也会根据不同的要求调用不同的函数体，这叫函数重载，相关内容将在第 6 章介绍。



实例 081 将 A 段内存复制到 B 段内存(指针内存复制)

【实例描述】

使用指针处理内存较为灵活，对于计算机本身的内存比较零散，不能大段大段地被申请的问题，可以使用指针复制内存的方式实现此功能。本实例实现将 A 段内存复制到 B 段内存，效果如图 5-15 所示。



【实现过程】

定义结构体 `person`，包含两个数据：`int` 型 `age` 和 `char` 型数据 `name[10]`。申请两块内存用于存放变量 `p1` 和 `p2` 的值，`person` 型指针 `pp` 用于复制内存。本实例的目的是将内存中存放 `p1` 变量的数据复制到存放 `p2` 变量的内存段，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 struct person           //结构体
05 {
06     int age;             //年龄
07     char name[10];       //姓名
08 };
09 int main()
10 {
11     person p1,p2;        //结构体对象
12     person *pp;          //结构体指针
13     pp=&p1;              //取地址
14     p1.age=10;           //赋值
15     strcpy(p1.name,"John"); //复制
16     p2.age=pp->age;       //指针读取元素 age 值
17     strcpy(p2.name,pp->name); //复制
18     cout<<"pp2 的 age:"<<p2.age<<endl;
19     cout<<"pp2 的 name:"<<p2.name<<endl;
20     return 0;
21 }
```



图 5-15 指针内存复制

【代码解析】

第 04~08 行定义结构体 `person`，第 11、12 行定义结构体变量 `p1`、`p2` 和指针 `pp`。其中，`pp` 取址于 `p1`（第 13 行）。第 14、15 行初始化 `p1` 的值。第 16、17 行使用指针 `pp` 实现内存的复制，最后由第 18、19 行输出结果以验证。



注意：第 16 行的 `pp->age` 还可以由 `(*pp).age` 实现。同理，第 17 行的 `pp->name` 由 `(*pp).name` 实现。



实例 082 将内存的数据倒转过来（指针内存复制+算法）

【实例描述】

本实例用于实现如何将内存中的数据倒转并显示结果，该功能的实现需要用到实例 081 的指针内存复制原理，并实现功能算法。此外，本实例的算法只用倒转整型数据内存，效果如图 5-16 所示。



图 5-16 内存数据的倒转示例

【实现过程】

定义整型数组 `memo[10]`，初始值为 {3,4,5,6,7,8,9,9,0,0}（其中不足的补为 0）。反转后的结

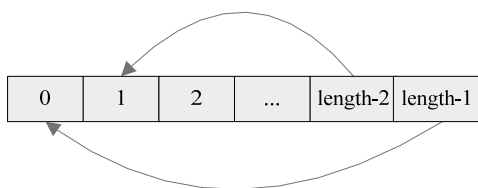


图 5-17 内存倒转原理

```

06     int temp;
07     int i,j;
08     for(i=0;i<=(length-1)/2;i++)           //二分反转
09     {
10         j=length-1-i;
11         temp=a[i];
12         a[i]=a[j];
13         a[j]=temp;
14     }
15 }
16 int main()
17 {
18     int memo[10]={3,4,5,6,7,8,9,9};
19     cout<<"原内存数据: ";
20     for(int i=0;i<10;i++)
21         cout<<memo[i];
22     cout<<endl;
23     inv(memo,10);                           //反转
24     cout<<"反转后的内存数据: ";
25     for(int i=0;i<10;i++)
26         cout<<memo[i];
27     cout<<endl;
28     return 0;
29 }

```

【代码解析】

第 04~15 行定义函数 `inv()`，第 1 个参数是所要反转的内存首地址，第 2 个参数是内存大小。第 16~29 行是执行函数 `main()` 定义体，第 18 行定义变量 `memo[10]`，第 20、21 行输出原内存数据。第 23 行调用函数 `inv()`，第 25、26 行输出反转后 `memo[10]` 内存的数据。



注意：函数 `inv()` 的第一个参数还可以写为 `int a[]`，也能实现反转效果，此时数组相当于指针。



实例 083 将数据隐藏于内存(自定义数据访问规则)

【实例描述】

本实例实现将数据隐藏于内存中，模拟只读写位于内存中偶数位置的数据。比如，内存中存有数据 `HIellolvooe,,Wco+r+ld`，但读到的数据只是 `Hello,World`，效果如图 5-18 所示。

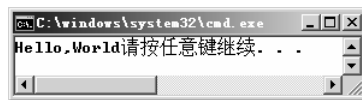


图 5-18 将数据隐藏于内存



【实现过程】

定义字符数组 `ch`，初始化为 `HIellolvoe,,Wco+r+l d`。在字符数组的长度内实现 `while` 循环以读出位于偶数位的数据，其代码如下：

```
01 #include <iostream>
02 using namespace std;
03 void main()
04 {
05     char ch[]="HIellolvoe,,Wco+r+l d";
06     int i=0;
07     while(i<strlen(ch))           //没有到字符数组末尾
08     {
09         if(i%2==0)                //读出位于偶数位的数据
10             cout<<ch[i];
11         else
12             { }
13         i++;                       //读下一位
14     }
15 }
```

【代码解析】

第 07~14 行的 `while` 循环实现读出偶数位的内存数据，条件如第 09 行所示。第 07 行的 `while` 循环终止条件是读到字符数组的最后一位。



注意：自定义数据访问规则可以将不可读写的数据隐藏，一般用于加密。



实例 084 输出本机内存数据排列顺序（高端先存还是低端先存）

【实例描述】

在不同的开发平台上编写代码，其内存数据的存储顺序也不同。本实例以整型数据为例，`int` 型数据占 4 字节，如 `int` 型变量 `aa=0x00000001`，编写代码以验证它在内存中的存储顺序是按如图 5-19 所示，还是按如图 5-20 所示的形式。

测试效果如图 5-21 所示。



图 5-19 高端先存

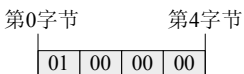


图 5-20 低端先存

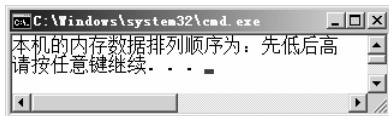


图 5-21 测试内存数据排列顺序

【实现过程】

定义组合体 `test`，其中有两个元素：`int` 型 `aa` 和 `char` 型 `ch`。先解释为什么用组合体，因为组合体变量占用内存的大小只是元素中最大的。所以，`test` 所占内存大小只是 `int` 型的大小。此外，其他元素从组合体的内存首地址开始存储，所以 `ch` 位于 `aa` 的第 1 字节处，测试代码如下：



```

01  #include <iostream>
02  using namespace std;
03
04  union test
05  {
06      int aa;
07      char ch;
08  };
09  void main()
10  {
11      test _test;                                //组合
12      _test.aa=0x00000001;
13      cout<<"本机的内存数据排列顺序为: ";
14      if(_test.ch==1)                            //如果读出数据为 1, 则为低端先存
15          cout<<"先低后高"<<endl;
16      else if(_test.ch==0)                        //如果读出数据为 0, 则为高端先存
17          cout<<"先高后低"<<endl;
18  }

```

【代码解析】

第 04~08 行是联合体类型 `test` 的定义, 第 12 行初始化 `_test` 的元素 `aa` 的值, 其值用十六进制数表示。第 14~17 行判断内存数据是高端先存还是低端先存, 即获取 `ch` 的值即可。



注意: 一个十六进制代表 4 位二进制, 所以表示一个 `int` 型数据需包含 8 个十六进制。



实例 085 寻找地址（指针加减法）

【实例描述】

本实例通过指针的加减法实现地址的寻找, 效果如图 5-22 所示。

【实现过程】

定义整型数组 `num[]`, 并进行初始化, 定义整型指针 `ptr`, 并将 `num` 的第 0 个元素地址赋给 `ptr`。最后寻找第 0、5 个元素的地址及内存值, 代码如下:

```

01  #include <iostream>
02  using namespace std;
03
04  int main()
05  {
06      int num[]={0,1,2,3,4,5,6};                //数组
07      int *ptr;                                   //指针
08      ptr=&num[0];                                //取址
09      cout<<ptr<<" "<<*(ptr++)<<endl;
10      cout<<ptr+5<<" "<<*(ptr+5)<<endl;
11      return 0;
12  }

```

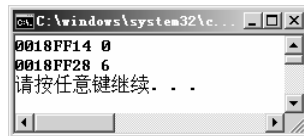


图 5-22 指针加减法



【代码解析】

第 06、07 行定义整型数组 num 和整型指针 ptr, 第 08 行将 num 第 0 个元素的地址赋给 ptr。第 09 行输出当前 ptr 的地址值和该地址的内存值, 第 10 行输出 num 第 6 个元素的地址值和其内存值。



注意: 因为第 09 行运用变量自加, 因此当前参与运算时它的地址并没有改变, 当第 09 行执行完后, ptr 再加 1。如有不懂, 请查看位于不同的自增运算符。



实例 086 利用指针删除数组中的指定元素(指针移动)

【实例描述】

本实例演示如何利用指针删除数组中指定的元素, 刚开始指针获取地址时都会被赋值数组的第 0 位元素地址, 通过移动指针可以达到上述目的, 效果如图 5-23 所示。

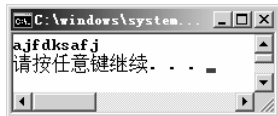


图 5-23 移动指针删除数组中某元素

【实现过程】

定义字符数组 ch[] 和整型指针变量 ptr, 并将字符数组的起始地址值赋给 ptr, 然后删除 ch 中中间的那位元素, 代码如下:

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     char ch[]="ajfdklsafj";
07     char *ptr=ch;                //获取 ch 的首地址
08     ptr += strlen(ch)/2;          //向后移动 strlen(ch)/2 位
09     for(int i=strlen(ch)/2;i<strlen(ch);i++) //删除第 strlen(ch)/2 个元素
10         ch[i]=ch[i+1];
11     cout<<ch<<endl;
12     return 0;
13 }
```

【代码解析】

第 06 行定义字符数组 ch, 并没有设定 ch 的长度。第 07 行定义字符指针 ptr, 并将 ch 的首地址赋给 ptr。本实例删除字符数组的中间位置元素, 如第 08 行所示, 将 ptr 指针移到 strlen(ch)/2 处。第 09、10 行实现将 strlen(ch)/2 之后的元素赋给前一位的内存中, 以实现元素的删除。



注意: 由于数组的长度不一定是偶数, 所以所删除元素的位置是 strlen(ch)/2 的舍位置, 即如果是 15/2, 则删除第 7 位。

第 6 章 函 数

在 C++ 程序设计中，函数是将多个语句组织在一起，形成具有特定功能的程序。函数的应用使程序的设计更具模块化，可读性也更高。本章包含 23 个实例，涵盖 C++ 语言对于函数从基础到提高的应用。比如，如何设计函数、函数重载、函数中形参的引用等。最后 3 个实例以简单的函数应用给出各个程序的实现框架。



实例 087 格式打印（设计函数）

【实例描述】

在控制台程序中，打印数据到屏幕使用 `cout<<` 语法，但是打印到屏幕上的格式却是五花八门。如何按格式打印数据到屏幕？本实例设计的函数即可实现该目的。分各种进制打印整型数据，对于浮点数据的小数位显示也可以控制，对于字符串的输出，采用对齐方式以不同的宽度显示。效果如图 6-1 所示。



图 6-1 设计函数按格式打印

【实现过程】

定义 3 个函数，分别对 `int`、`float`、`string` 类型进行格式输出：`print_int()`、`print_float()`、`print_string`，代码如下：

```
01 #include <iostream>
02 #include <iomanip>
03 #include <string>
04 using namespace std;
05
06 void print_int(int x,int y)
07 {
08     cout<<x<<' '<<y<<endl;           //按十进制数输出
09     cout<<oct<<x<<' '<<y<<endl;       //按八进制数输出
10     cout<<hex<<x<<' '<<y<<endl;       //按十六进制数输出
```



```

11 }
12 void print_float(float a, float b)
13 {
14     cout<<a<<' '<<b<<endl;           //无格式输出
15     cout.setf(ios::showpos);          //强制在正数前加+号
16     cout<<a<<' '<<b<<endl;
17     cout.unsetf(ios::showpos);         //取消正数前加+号
18     cout.setf(ios::showpoint);         //强制显示小数点后的无效0
19     cout<<a<<' '<<b<<endl;
20     cout.unsetf(ios::showpoint);       //取消显示小数点后的无效0
21     cout.setf(ios::scientific);        //科学记数法
22     cout<<a<<' '<<b<<endl;
23     cout.unsetf(ios::scientific);      //取消科学记数法
24     cout.setf(ios::fixed);              //按点输出显示
25     cout<<a<<' '<<b<<endl;
26     cout.unsetf(ios::fixed);           //取消按点输出显示
27     cout.precision(18);                //精度为18, 正常为6
28     cout<<a<<' '<<b<<endl;
29     cout.precision(6);                 //精度恢复为6
30 }
31 void print_string(string a,string b,string *c,int length)
32 {
33     cout<<a<<endl;
34     cout<<" ";
35     cout<<setiosflags(ios::left)<<setw(10); //设置宽度为10, left 对齐
36     cout<<b<<endl;
37     cout<<resetiosflags(ios::left);       //取消对齐方式
38     for(int i=0;i<length;i++)
39     {
40         cout<<" ";
41         cout<<setw(7)<<setiosflags(ios::left); //设置宽度为10, left 对齐
42         cout<<c[i];
43         cout<<resetiosflags(ios::left);       //取消对齐方式
44         cout<<setfill('.')<<setw(30)<<i+1<<endl; //宽度为30, 填充为'.'输出
45     }
46 }
47 void main()
48 {
49     int xx=100, yy=200;
50     print_int(xx,yy);                     //按进制输出
51
52     float f1=0.000000001,f2=-0.6;
53     print_float(f1,f2);                   //浮点数
54
55     string str_caption="第6章";
56     string str1="实例 087 格式打印(设计函数)";
57     string str_sub[3]={"实例描述","实现过程","代码解析"};
58     print_string(str_caption,str1,str_sub,3); //字符串输出
59 }

```

【代码解析】

第06~11行是格式输出整型变量,在main()函数中对应的调用语句为第49、50行,分别按十进制数、八进制数和十六进制数输出整数。第12~30行是按格式输出浮点型变量,本实例以float型为例,格式分别为:在正数前加+号输出,强制显示小数点后无效的0,科学计数法输出float型变量,按点输出变量,设定精度输出。float型数据的精度为6,此时精度18输出,所以不足位的用内存数据填补。



注意：在用其他进制输出整数后，需再设置用十进制数输出，否则本程序在执行完第 10 行后都是以十六进制数输出整型数据，格式为：`cout<<dec;`或者为 `cout.unsetf(ios::hex);`。



实例 088 指令接收器（字符串形参）

【实例描述】

本实例利用函数的形参实现指令接收器，当用户发出某种指令后，系统做出相应的操作。函数的参数为字符串，运行效果如图 6-2 所示。

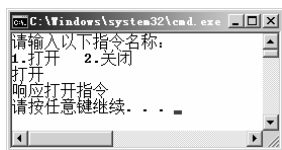


图 6-2 指令接收器

【实现过程】

定义字符串变量 `receive_str` 存储用户输入的指令，函数 `receive_code()` 用于实现指令接收器功能（对应两种指令：打开和关闭），函数代码如下：

```
01 void receive_code(string str)           //接收指令
02 {
03     if(str=="打开")
04         cout<<"响应打开指令"<<endl;
05     else if(str=="关闭")
06         cout<<"响应关闭指令"<<endl;
07     else
08         cout<<"该指令无效"<<endl;
09 }
main()代码如下:
01 void main()
02 {
03     string receive_str;                 //指令字符串
04     cout<<"请输入以下指令名称: "<<'\n';
05     cout<<"1.打开 2.关闭"<<endl;
06     cin>>str;
07     receive_code(receive_str);         //调用函数
08 }
```

【代码解析】

- （1）在函数 `receive_code()` 中，第 03~08 行对于参数 `str` 的值输出不同指令的接收器状态。
- （2）执行函数 `main()` 中，第 03 行定义 `string` 型变量 `receive_str`，第 04、05 行输出说明性语句，第 06 行获取指令，第 07 行调用函数 `receive_code()`。



注意：由于形参是值传递，所以只能接收指令，并不能将该指令返回到 `main()` 函数。



实例 089 汽车行驶里程（函数实现）

【实例描述】

本实例实现函数用于计算汽车行驶里程，假设一辆汽车每小时可行驶 90 公里，每连续行驶两小时休息半小时，再继续行驶。问汽车工作 8 小时行驶的里程为多少？假设汽车一直按照此规律运行，没有意外发生，运行效果如图 6-3 所示。

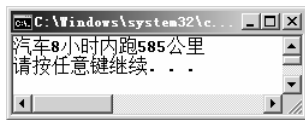


图 6-3 汽车行驶里程

【实现过程】

定义函数 `mile()` 用于计算汽车一天的行驶里程，`mile()` 函数有 4 个参数，依次为 `float a`（一天工作时间长度）、`float b`（每小时行驶里程）、`float c`（间隔相邻工作段的休息时间）、`int d`（每个工作段的时间）。返回值类型为 `float`，即一天行驶的里程。具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 float mile(float a,float b,float c,int d)
05 {
06     float result=0.0;           //总里程
07     bool flag=true;
08     while(flag==true)           //时间没用完
09     {
10         if(a>=d)
11             result+=d*b;
12         else
13         {
14             flag=false;
15             if(a<=d)
16                 result+=a*b;
17             else
18                 result+=b;
19         }
20         a-= (c+d);               //先减
21     }
22     return result;
23 }
24 void main()
25 {
26     float whole_hour=8;          //总时间
27     float mile_per_hour=90;      //每小时里程
28     float relax=0.5;             //休息时间
29     int diff=2;                  //工作两小时休息
30     cout<<"汽车 8 小时内跑"<<mile(whole_hour,mile_per_hour,relax,diff)<<"公
    里"<<endl;
31 }
```

【代码解析】

第 04~23 行为函数 `mile()` 的定义体，第 06 行为一天行驶的总里程变量 `result` 定义，第 07 行为标志量 `flag`，用于标识一天的工作时间是否全部使用完。第 08~21 行为 `while()` 循环，当时间没有用完时，继续计算汽车还可以走多少里。它的实现流程如图 6-4 所示。

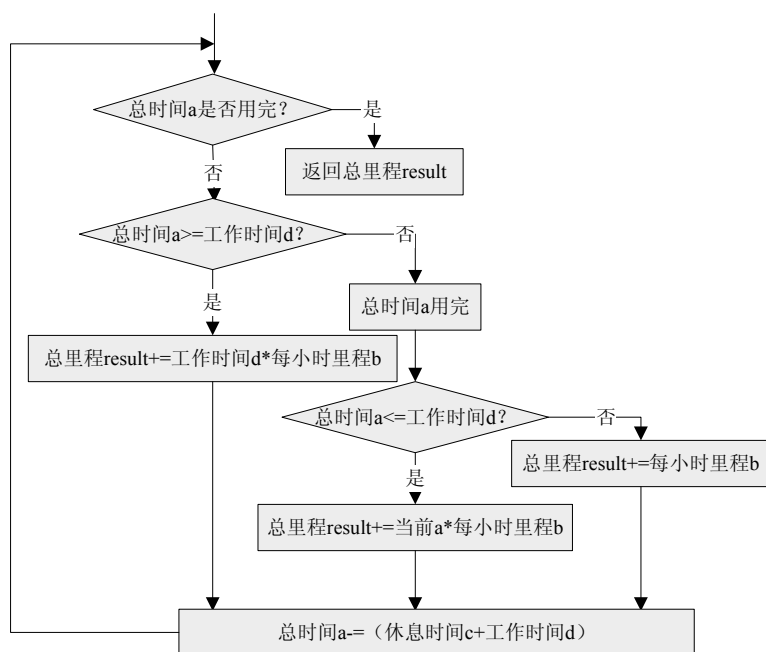


图 6-4 汽车行驶里程函数 mile()实现流程



实例 090 求班级男女生的人数

【实例描述】

对于求班级中男女的人数有很多种方法，本实例将其中一种方法封装为函数实现。它的形式为：

```
void fenlei(int *a, int * b, int length);
```

其中，第 1、3 个参数为输入参数，第 2 个为输出参数，用以表征男女性别分别为 0 和 1 的区别，效果如图 6-5 所示。

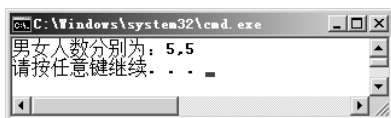


图 6-5 求班级男女人数

【实现过程】

定义整型数组 sex_banji[10]表示一个班有 10 位学生，分别用 0 和 1 表示各自的性别。数组 num[2]表示男女人数，num[0]表示男生人数，num[1]表示女生人数。具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void fenlei(int* a, int* b, int length)    //分类
05 {
```



```

06     for(int i=0;i<length;i++)
07     {
08         if(a[i]==0)                                //元素为0，为男
09             b[1]++;
10         else if(a[i]==1)                            //为1，为女
11             b[0]++;
12     }
13 }
14 void main()
15 {
16     int sex_banji[10]={1,1,0,0,0,1,0,0,1,1}; //0-female, 1-male
17     int num[2]={0};                             //男女人数，元素0为男，元素1为女
18     fenlei(sex_banji,num,10);
19     cout<<"男女人数分别为: "<<num[0]<<","<<num[1]<<endl;
20 }

```

【代码解析】

(1) 第 04~13 行为函数 `fenlei()` 的定义体，第 1 个参数为班级所有人的性别，第 2 个参数为输出参数，表征男女人数。第 3 个参数为班级有多少人。总共循环 `length` 次，如第 06 行，如果 `a[i]` 等于 0，女生人数加 1；反之，男生人数加 1。

(2) 第 14~20 行为 `main()` 函数定义体，第 16、17 行定义数组 `sex_banji[10]` 和 `num[2]`，之后在第 18 行调用函数 `fenlei`，然后输出结果。



注意：`num[2]` 数组表示男女人数必须初始化为 0。



实例 091 定义函数求 N 的 N 次方

【实例描述】

在计算器中有一种运算叫幂运算，即 N 个 N 相乘。本实例的函数实现求 N 的 N 次方，定义函数形式为：

```
double mi(int n);
```

函数输入参数为 N 的值，返回类型为 `double`，值为 N 的 N 次方结果，运行效果如图 6-6 所示。

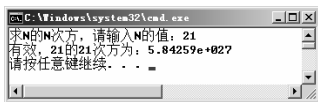


图 6-6 定义函数求 N 的 N 次方

【实现过程】

函数 `mi()` 中需定义 `double` 型变量 `ji` 以存储结果，循环 N 次实现 N 个 N 相乘，最后返回 `ji` 的值。`mi()` 和 `main()` 的代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 double mi(int n)                //求 n 的 n 次方
05 {
06     double ji=1;

```



```

07     for(int i=n;i>0;i--)           //循环 n 次
08         ji*=n;
09     return ji;                     //返回结果
10 }
11 void main()
12 {
13     int N;
14     cout<<"求 N 的 N 次方, 请输入 N 的值: ";
15     cin>>N;
16     if (N>INT_MAX)                 //输入值是否超出 int 型的最大值
17         cout<<"无效, N 的值超出 int 类型的界限 ("<<INT_MAX<<") ";
18     else
19         cout<<"有效, "<<N<<"的"<<N<<"次方为: "<<mi(N)<<endl;
20 }

```

【代码解析】

(1) 第 04~10 行为函数 `mi()` 实现体, 其中第 07 行的 `for` 循环初始表示式为 `N` 的值, 终止表达式为 `i>0`, 下一步条件为 `i--`。

(2) 第 11~20 行为 `main()` 函数实现, 第 14 行输出函数 `mi()` 的功能, 第 16~19 行的 `if...else` 结构判断输入的 `N` 值是否在 `int` 型的表示范围内。如果越界, 则提示; 反之, 则输出结果。



注意: 由于在计算幂次运算时, 若 `N` 的值相当大, 结果也会很大, 可能超出 `int` 数据的界限, 所以结果用 `double` 型代替。



实例 092 内存整理（函数实现把 0 内存删除）

【实例描述】

当用字符数组表征字符串, 事先并不知道所定义的字符数组应该多大, 所以其大小都必须以大为主, 以免在赋值的过程中越界。但是, 如果赋值的字符串大小小于字符数组的长度, 会有 0 内存, 造成内存的浪费, 此时就需将 0 内存删除。本实例即实现该功能, 运行效果如图 6-7 所示。

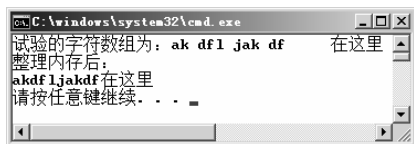


图 6-7 删除 0 内存

【实现过程】

定义函数 `delZero()` 去除 0 内存, 当字符为空格时, 就被视为零内存。在 `main()` 函数中定义变量 `ch_test`, 并动态分配内存, 大小为 `M`, 代码如下:

```

01 #include <iostream>
02 #include <stdlib.h>
03 using namespace std;
04
05 #define M 30
06
07 void delZero(char *ch, int size)           //删除 0 内存

```



```

08  {
09      int length=size;
10      for(int i=0;i<size;i++)                //循环目标内存
11      {
12          if(ch[i]==' ')                    //如果为 0
13          {
14              for(int j=i;j<size;j++)        //0 后的元素移前
15                  ch[j]=ch[j+1];
16              length--;                      //内存长度减 1
17              i--;
18          }
19      }
20      ch=new char[length];
21  }
22  void main()
23  {
24      char *ch_test;
25      ch_test=new char[M];                  //试验数组
26      strcpy(ch_test,"ak dfl jak df ");    //赋值
27      cout<<"试验的字符数组为: "<<ch_test<<"在这里"<<endl;
28      delZero(ch_test,M);                  //调用除 0 函数
29      cout<<"整理内存后: "<<endl;
30      cout<<ch_test<<"在这里"<<endl;
31      free(ch_test);                      //释放内存
32      ch_test=NULL;
33  }

```

【代码解析】

第 05 行宏定义 `ch_test` 的初始大小为 30，第 07~21 行是函数 `delZero()` 的定义体，第 1 个参数为内存，第 2 个为内存大小。第 10~19 行使用 `for` 嵌套实现 0 内存的去除。当遇到空格时，将之后的元素全部向前移 1 个单位，内存的大小减 1。继续判断，直到内存全部判断完。为了试验位于最后的空格是否被删除，特用“在这里”标识，如第 27、30 行。



实例 093 分水果（使函数一次性返回 N 个值）

【实例描述】

一般情况下，定义的函数不返回值或者返回一个值，本实例实现使函数一次性返回 N 个值，其格式如下：

```
int* fenpei(int* a, int* b, int num);
```

返回值为数组的首地址，第 1、2 个参数为两个数组变量的首地址，第 3 个参数为数组的长度。本实例演示分水果过程，多种水果分别分配给对应的人数，并输出平均分配后还剩多少个，效果如图 6-8 所示。

【实现过程】

定义数组 `fruit[10]`（表示 10 种水果各自的个数）、`num_people[10]`（每种水果对应分配的人数）、`remain[10]`（对应每种水果分配后的剩余个数）。函数 `fenpei()` 和 `main()` 的代码如下：

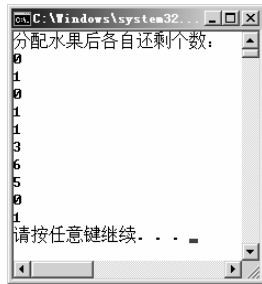


图 6-8 分水果（函数一次返回 N 个值）



```

01  #include <iostream>
02  using namespace std;
03
04  int* fenpei(int* a, int* b, int num)                //分配函数
05  {
06      int *remain=new int[num];
07      for(int i=0;i<num;i++)
08          remain[i]=a[i]%b[i];                      //取余运算
09      return remain;
10  }
11  void main()
12  {
13      int fruit[10];                                //10 种水果各自的个数
14      int num_people[10];                            //每种水果对应分配人数
15      int *premain=NULL;
16      for(int i=0;i<10;i++)                          //初始化数组
17      {
18          fruit[i]=100*i+1;
19          num_people[i]=i+1;
20      }
21      premain=fenpei(fruit,num_people,10);            //调用分配函数
22      cout<<"分配水果后各自还剩个数: "<<endl;
23      for(int i=0;i<10;i++)
24          cout<<premain[i]<<endl;
25      delete[] premain;
26      premain=NULL;
27  }

```

【代码解析】

第 04~10 行为函数 `fenpei()` 定义体, `remain` 是动态分配内存, 第 08 行使用 `%` 运算符获得剩余个数, 由第 09 行返回 `remain` 的首地址。第 11~27 行是 `main()` 定义体, 第 15 行定义指针 `premain` 指向函数 `fenpei()` 的返回首地址, 并由第 23、24 行输出各种水果在平均分配后的剩余量。最后在第 25、26 行释放指针和内存。



注意: 第 06 行用 `new` 申请的内存不能在该函数体内释放, 否则在第 21 行将获取不到返回首地址。



实例 094 图书名整理系统（按开头字母重新排列）

【实例描述】

本实例的图书名整理系统是将图书顺序排列, 规则是按书名的开头字母在字母表中的顺序。涉及的图书名都是英文名, 如果是中文名, 还需获取该首字的汉语拼音, 较为麻烦, 所以本实例只以英文书名为例进行介绍, 效果如图 6-9 所示。

【实现过程】

定义整型变量 `m` 和 `n`, 分别表示二维字符数组的行和列。动态申请二维内存存储图书名字, 变量名为 `ch`。

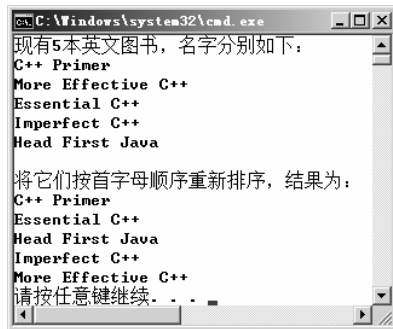


图 6-9 按开头字母排列图书名



初始化 5 本图书名，分别为 C++ Primer、More Effective C++、Essential C++、Imperfect C++ 和 Head First Java。然后调用字母排序函数 `sequence()`，完成排序，且输出结果，代码如下：

```

01  #include <iostream>
02  using namespace std;
03
04  void sequence(char **ch, int row, int col)           //图书排序
05  {
06      for(int i=0;i<row;i++)                          //行
07      {
08          for(int j=i+1;j<row;j++)                    //列
09          {
10              if(ch[i][0]>ch[j][0])                    //图书首元素比较
11              {
12                  char *ch1=new char[col];
13                  strcpy(ch1,ch[i]);                  //交换
14                  strcpy(ch[i],ch[j]);
15                  strcpy(ch[j],ch1);
16              }
17          }
18      }
19  }
20
21  void main()
22  {
23      int m=5,n=50;                                    //行列值定义
24      cout<<"现有 5 本英文图书，名字分别如下："<<endl;
25      char **ch=new char*[m];                          //二维内存
26      for(int i=0;i<m;i++)
27          ch[i]=new char[n];
28
29      strcpy(ch[0],"C++ Primer");                      //初始化 5 本书
30      strcpy(ch[1],"More Effective C++");
31      strcpy(ch[2],"Essential C++");
32      strcpy(ch[3],"Imperfect C++");
33      strcpy(ch[4],"Head First Java");
34      for(int i=0;i<5;i++)
35          cout<<ch[i]<<endl;
36
37      cout<<endl<<"将它们按首字母顺序重新排序，结果为："<<endl;
38      sequence(ch,m,n);                                //调用排序函数
39      for(int i=0;i<5;i++)
40          cout<<ch[i]<<endl;
41      delete[] ch;
42      ch=NULL;
43  }
```

【代码解析】

(1) `sequence()`

第 04~19 行为其定义体，输入参数为图书名二维内存、二维内存的大小（行和列）。使用两个 `for` 循环完成前后一维内存的首元素比较，如第 10 行。如果前一个大于后一个，就交换。

(2) `main()`

第 23 行定义二维内存的行、列值。第 25~27 行定义二维内存 `ch`。第 29~33 行初始化内存值，并输出（第 34、35 行）。第 38 行调用排序函数 `sequence()`，并且输出结果。最后，在第 41、42 行释放内存和指针。



注意：在交换数组时，声明的临时变量要与赋值元素的大小相同，如第 12 行。



实例 095 姓名测试（根据首字母开头+算法）

【实例描述】

本实例模拟最近网上流行的姓名测试小游戏，即根据个人的姓名 3 个首字母测算您在古代是何许人。规则是从姓氏开始找，如果是复姓，只取第 1 个字的首字母。接下来取中间名的首字母，再取最后一个名的首字母。本实例只针对名为两个字的，现测试姓名为：秦小猴的结果，效果如图 6-10 所示。

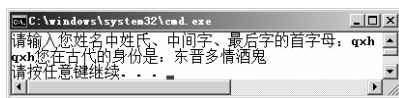


图 6-10 根据首字母测试姓名

【实现过程】

定义函数 `name_test()` 用于测试姓名，共有两个参数，现解释如下：

`name_test(char *ch, string &str);`

(1) `char *ch`

输入的姓名首字母数组的首地址

(2) `string &str`

输出的表示何许人结果。

定义 3 个字符串数组，用于存储对应姓、中间名、最后名的 26 个首字母的代表身份，分别为 `str_first[26]`、`str_mid[26]`、`str_late[26]`。如果获取的首字母有小写，则先转化为大写，再取与字母 A 的差值，以期用数组索引值找到对应的身份。在 `main()` 中定义 `ch[3]` 获取 3 个首字母，`str_result` 输出最后结果，代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 void name_test(char *ch, string &str) //姓名测算
06 {
07     string str_first[26]={"曹魏","东吴","蜀汉","吐蕃","秦宋","大金","明朝","鲜卑",
08                          "","五代","\n",
09                          "晚清","西周","赵国","大辽","夏商","战国","春秋","东晋","羌族","\n",
10                          "民国","齐国","闽国","南燕","大理","西凉","南唐","蒙古"};
11     string str_mid[26]={"阴毒","疯骚","猥琐","豪门","白痴","天才","自卑","美貌",
12                        "花痴","\n",
13                        "英勇","臭屁","卖国","饭桶","变态","热血","嗜血","旷世","守财","\n",
14                        "血手","平庸","爱国","亡命","暴力","多情","自恋","逍遥"};
15     string str_late[26]={"道士","和尚","财主","屠夫","马夫","嫖妃","兵士","酒鬼",
16                        "","地主","\n",
17                        "医师","农夫","稳婆","更夫","书生","乞丐","帝或后","臣相","流氓","\n",
18                        "戏子","土匪","守墓人","媒婆","庖丁","诗人","歌妓","讼师"};
```



```

16     for(int i=0;i<3;i++)
17     {
18         if(ch[i]<='z'&&ch[i]>='a')
19             ch[i]==(32+65);           //变成大写，再取差值
20         else
21             ch[i]==65;
22         switch(i)
23         {
24             case 0:                     //第一个字
25                 str+=str_first[ch[i]];
26                 break;
27             case 1:                     //第二个字
28                 str+=str_mid[ch[i]];
29                 break;
30             case 2:                     //第三个字
31                 str+=str_late[ch[i]];
32                 break;
33             default:
34                 break;
35         }
36     }
37 }
38 void main()
39 {
40     char ch[3]={" "};
41     string str_result;
42     cout<<"请输入您姓名中姓氏、中间字、最后字的首字母: ";
43     cin>>ch[0]>>ch[1]>>ch[2];         //输入姓名首字母
44     cout<<ch[0]<<ch[1]<<ch[2];
45     cout<<"您在古代的身份是: ";
46     name_test(ch,str_result);         //调用测算函数
47     cout<<str_result<<endl;
48 }

```

【代码解析】

第 05~61 行为函数 `name_test()` 的定义体，第 07~15 行定义三个字符串数组，第 16~60 行判断 3 个字符分别代表什么。第 18~21 行的 `if...else` 结构判断首字母是大写还是小写。第 46~59 行的 `switch` 输入 3 个差值，并赋值 `str_result`。



实例 096 宝宝改名（函数参数直接引用变量（形参引用））

【实例描述】

当形参采用值传递时，并没有改变输入变量的值。而使用地址传递时却可以，但是使用指针对于内存的安全性并没有保障。除了指针传递外，还有一种更安全的方法，即形参引用同样可以达到目的。本实例模拟宝宝改名演示如何进行形参引用，效果如图 6-11 所示。

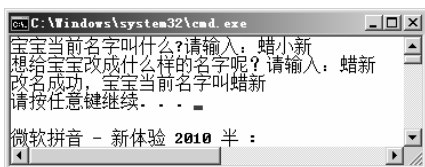


图 6-11 形参引用



【实现过程】

定义 string 型变量 cur_name 和 exp_name，分别表示当前名字和期望名字。通过调用函数 change_name()完成宝宝改名，代码如下：

```
01 void change_name(string &name,string later)
02 {
03     name=later;
04 }
05 void main()
06 {
07     string cur_name;           //当前名字
08     string exp_name;          //期望名字
09     cout<<"宝宝当前名字叫什么?请输入: ";
10     cin>>cur_name;            //输入当前姓名
11     cout<<"想给宝宝改成什么样的名字呢?请输入: ";
12     cin>>exp_name;            //输入期望姓名
13     change_name(cur_name,exp_name); //改名
14     cout<<"改名成功,宝宝当前名字叫"<<cur_name<<endl;
15 }
```

【代码解析】

第 01~04 行为改名函数 change_name()定义体，其中，第 1 个参数为引用，表示宝宝当前姓名。第 2 个参数为期望姓名（值传递）。在 main()函数中由用户输入 cur_name 和 exp_name 值，调用函数（第 13 行）实现改名。最后在第 14 行输出当前姓名，以验证结果正确。



注意：参数引用相对于指针传递而言，更安全，因为它不直接接触内存。



实例 097 求最长字符串

【实例描述】

在目标字符数组中并不是所有的元素都是字符，可能会有数字和标点符号等非字符元素。本实例要实现寻找最长字符，即所有连续的元素都是字符。如果遇到两个字符串的连续元素都为字符，并且长度也相等，则输出位置靠前的字符串。比如，字符串 ab="143adfwe23;adfik"。



图 6-12 求最长字符串

其中字符串子串“adfwe”和“adfik”都是连续字符，并且长度也相同，则输出字符串“adfwe”，效果如图 6-12 所示。

【实现过程】

申请一维内存存储目标字符串数据，变量为 ch。定义 4 个整型变量，分别为 start、left、length 和 length_max，它们分别表示最长字符串的开头位置、当前字符串的最左边位置、当前字符串的长度、最长字符串的长度。循环访问字符数组的每个元素，如果当前元素为字母，当前的字符串长度加 1。如果不是字母，再判断当前的字符串长度是否大于最长字符串。如果大于最大字符串的长度，开始更新 start 和 length_max 的值。接下来，更新 left 和 length 的值。最后输出从 start 位置开始，长度为 length_max 的字符串。代码如下：

```
01 #include<cctype>
```



```

02 #include<iostream>
03 using namespace std;
04 #define M 1000
05 const char *extra="$";
06 void main()
07 {
08     char *ch=new char[M];
09     int start = 0;           //最长字符的开始
10     int left = 0;           //每段字符的左边值
11     int length = 0;         //当前字符串的长度
12     int length_max = 0;     //最长字符的长度
13     cin>>ch;               //输入字符串
14     if(isalpha(ch[strlen(ch)-1])) //是字母
15     {
16         //加一个不是字母的元素，以找到最后一个字符子串
17         strcat(ch,extra);
18     }
19     for(int i=0; i<strlen(ch); i++)
20     {
21         if(isalpha(ch[i])) //是字母
22         {
23             length++;      //当前字符串长度加 1
24         }
25         else               //不是字母
26         {
27             if(length>length_max) //如果当前字符串的长度大于最大字符的长度
28             {                  //更新开始
29                 start=left;
30                 length_max=length;
31             }
32             left=i+1;        //新的左边值
33             length=0;        //新的字符串长度从 0 开始增加
34         }
35     }
36     cout<<"最长字符串: ";
37     for(int i=start;i<start+length_max;i++)
38         cout<<ch[i];
39     cout<<endl;
40     delete ch;
41     ch=NULL;
42 }

```

【代码解析】

第 04、05 行定义常量 `M` 和 `extra`，分别代表一维内存的大小和字符数组的结尾元素。第 08~12 行定义各类变量，第 13 行获取字符数组的值。第 14 行判断最后一个元素是否为字母，如果是字母，则必须在末尾加上非字母的元素，此处用 `extra` 的值，它的作用是当检测最后一个元素时，如果仍为字母，将不会更新信息。如果最后一个元素不是字母，信息就会更新。

第 19~35 行循环字符数组中的每个元素，判断是否为字母，并做相应的更新信息操作。第 37、38 行输出字符数组中最长的字符子串。第 40、41 行释放指针和内存。



注意：第 05 行定义的 `extra` 常量值可以是任意非字母值。



实例 098 补充代码并保证变量 A 的值等于 10

【实例描述】

接着实例 097, 本实例实现判断字符串中是否有长度为 10 的字符子串, 效果如图 6-13 所示。

【实现过程】

在实例 097 中, 第 35 行代码后判断 `length_max` 的值是否小于 10。如果小于 10, 表示该字符串中没有最长为 10 的字符子串, 反之, 输出此字符子串。补充代码如下:

```
01 #include<cctype>
02 #include<iostream>
03 using namespace std;
04 #define M 1000
05 const char *extra="$";
06 void main()
07 {
08     ...
09     for(int i=0; i<strlen(ch); i++)
10     {
11         ...
12     }
13     if(length_max<10) //没有最长为 10 的字符
14         cout<<"该字符串中没有最长为 10 的字符子串"<<endl;
15     else
16     {
17         length_max=10; //最长长度为 10
18         cout<<"长度为 10 的字符子串: ";
19         for(int i=start; i<start+length_max; i++)
20             cout<<ch[i];
21     }
22     ...
23 }
```

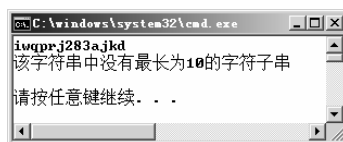


图 6-13 求最长为 10 的字符子串

【代码解析】

第 13~21 行的 `if...else` 结构判断 `length_max` 的值是否小于 10, 并做相应的输出。



注意: 第 01 行包含的头文件 `cctype.h` 用于函数 `isalpha()` 的使用, 它可以判断该元素是否为字母。



实例 099 头文件重定义错误案例

【实例描述】

在编程时除了 `.cpp` 文件, 还需要 `.h` 文件。良好的编程习惯是 `.h` 文件只是函数和类的声明, 其定义体出现在对应的 `.cpp` 文件中。如果定义体也书写在 `.h` 文件中, 并且由多个 `.cpp` 文件包含,



就会出现头文件重定义的错误，效果如图 6-14 所示。

```
1>----- Build started: Project: 099, Configuration: Debug Win32 -----
1>Compiling...
1>099(1).cpp
1>099.cpp
1>Generating Code...
1>Compiling manifest to resources...
1>Linking...
1>099.obj : error LNK2005: "int __cdecl add(int,int)" (?add@099YA000027) already defined in 099(1).obj
1>099.obj : error LNK2005: "int a" (?a@093HA) already defined in 099(1).obj
1>099.obj : error LNK2005: "float b" (?b@03MA) already defined in 099(1).obj
Go to Previous Message
```

图 6-14 头文件重定义错误

【实现过程】

新建 3 个文件，分别为 099.h、099.cpp、099(1).cpp，后两个源文件都包含头文件 099.h。在 099.h 中定义变量 a 和 b、函数 add()，头文件 099.h 的代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int a=0; //变量定义
05 float b=0;
06 int add(int a,int b) //函数定义
07 {
08     return a+b;
09 }
```

源文件 099.cpp 的代码如下：

```
01 #include "099.h"
02 void main()
03 {}
```

源文件 099(1).cpp 的代码如下：

```
#include "099.h"
```

【代码解析】

程序在编译 099.cpp 时，已经将其所包含的头文件 099.h 中的变量 a、b 和函数 add() 做了编码定义。但是在编译 099(1).cpp 文件时，由于也包含该头文件，也将其内容重新定义，所以出现如图 6-13 所示的错误。解决方法是在头文件中删除变量的定义和函数 add() 的定义体，只做声明，代码如下：

```
01 #include <iostream>
02 using namespace std;
03 int add(int a,int b);
```

并将两个变量的定义和函数的定义体放到对应的源文件 099.cpp 中即可，此时重新编译链接没有错误。



注意：一定要有良好的编程习惯，否则遇到此类错误时很难找到它。



实例 100 更简便的命令解释器（函数重载）

【实例描述】

在 Windows 的仿 DOS 界面可以输入如表 6-1 所示的 DOS 命令。



表 6-1 DOS命令

序 号	DOS 命令	说 明
00	cd	改变当前目录
01	copy	复制文件
02	format	格式化磁盘
03	mem	查看内存状况
04	rd	删除目录

本实例通过函数重载，根据不同的输入方式，输出相同的结果。比如，输入序号 00 与输入 cd 命令，结果都能解释出改变当前目录的意思，效果如图 6-15 所示。



图 6-15 函数重载——实现命令解释器

【实现过程】

定义函数 decode()并重载，区别在于输入参数不同。用户根据不同的方式解释命令，输入不同的命令格式，此时系统调用不同的 decode()函数，代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 string str_dos[5]={"cd","copy","format","mem","rd"};
06 string str_meaning[5]={"改变当前目录","复制文件","格式化磁盘","查看内存状况","删除目录"};
07
08 void decode(int a) //解码，参数为 int
09 {
10     cout<<"解释为: "<<str_meaning[a]<<endl;
11 }
12 void decode(string str) //解码，参数为 string
13 {
14     for(int i=0;i<5;i++)
15     {
16         if(str_dos[i]==str)
17         {
18             cout<<"解释为: "<<str_meaning[i]<<endl;
19             i=5;
20         }
21     }
22 }
23 void main()
24 {
25     int index; //命令序号
26     string str_code; //命令
27     int choice; //选择
28     while(1)
```



```

29     {
30         cout<<"0-按序号解释, 1-按命令解释, 请选择: ";
31         cin>>choice;
32         switch(choice)
33         {
34             case 0:                                //命令序号
35                 cout<<"请输入序号 0~4 的一种: ";
36                 cin>>index;
37                 decode(index);
38                 break;
39             case 1:                                //命令名称
40                 cout<<"请输入命令为 cd、copy、format、mem、rd 的一种: ";
41                 cin>>str_code;
42                 decode(str_code);
43                 break;
44             default:
45                 break;
46         }
47         cout<<"0-继续, 1-退出, 是否继续?";
48         int x;
49         cin>>x;
50         if (x==0)                                //继续
51         {}
52         else if (x==1)                            //退出
53         {
54             cout<<"退出成功"<<endl;
55             break;
56         }
57     }
58 }

```

【代码解析】

第 05、06 行定义全局变量 `str_dos[5]` 和 `str_meaning[5]`, 分别表示 5 个 DOS 命令和对其对应的解释。第 08~11 行是 `decode()` 函数, 它以 DOS 命令的序号做解释。第 12~22 行是被重载的 `decode()` 函数, 它以 DOS 命令做解释。



注意: 本实例的重载函数 `dcode()` 参数类型不同, 一个为整型, 另一个为 `string` 型。



实例 101 函数重载陷阱案例

【实例描述】

函数重载体现在多个函数的名字必须相同, 但是参数列表的内容必须不同。参数的不同可以体现在个数及类型上, 比如, 以下例子:

```

int add(int a, int b);
int add(int a, int b, int c);
double add(double a, double b);

```

上述写法都是合法的函数重载, 但如果不改变参数的个数及类型, 只改变返回值类型, 就不合法。此外, 如果形参有了默认值, 可能会产生二义性, 格式如下:

```

int add(int a, int b);
int add(int a, int b, int c=10);

```



在计算 `add(2,3)` 时, 系统不知道该调用上述哪个函数, 程序弹出如图 6-16 所示的错误。

```
1>----- Build started: Project: 101, Configuration: Debug Win32 -----
1>Compiling...
1>101.cpp
1>c:\users\administrator\desktop\06\101\101.cpp(15) : error C2668: 'add' : ambiguous call to overloaded function
1>      c:\users\administrator\desktop\06\101\101.cpp(7): could be 'int add(int,int,int)'
1>      c:\users\administrator\desktop\06\101\101.cpp(4): or 'int add(int,int)'
1>      while trying to match the argument list '(int, int)'
1>Build log was saved at "file://c:\Users\Administrator\Desktop\06\101\Debug\BuildLog.htm"
1>101 - 1 error(s), 0 warning(s)
===== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =====
```

图 6-16 函数重载的二义性

【实现过程】

定义 3 个重载函数 `add()`, 并分别被调用计算加法结果, 其代码如下:

```
01 #include <iostream>
02 using namespace std;
03
04 int add(int a,int b)                //int 型加法, 2 个参数
05 {return a+b;}
06
07 int add(int a,int b,int c=10)       //int 型加法, 3 个参数
08 {return a+b+c;}
09
10 double add(double a,double b)      //double 型加法
11 {return a+b;}
12
13 void main()
14 {
15     cout<<"2+3="<<add(2,3)<<endl;
16     cout<<"2+3+4="<<add(2,3,4)<<endl;
17     cout<<"2.2+3.3="<<add(2.2,3.3)<<endl;
18 }
```

【代码解析】

第 04~11 行是 3 个重载函数 `add()`, 第 1 个函数 `add()` 的参数有两个, 都是 `int` 型。第 2 个 `add()` 函数的参数有 3 个, 也都是 `int` 型, 但是第 3 个参数有默认值。第 3 个 `add()` 函数的参数有两个, 都是 `double` 型。在第 15~17 行对其分别调用时, 由于第 2 个 `add()` 函数的第 3 个参数已有默认值, 系统对其与前一个函数的解释一样, 因此在执行 15 行代码时, 不清楚到底调用哪个函数。



注意: 函数重载的二义性发生在当参数类型相同、个数不一样时, 由于参数默认值造成它们的参数个数实质一样。对于此种情况, 一定要小心。



实例 102 main()后执行代码

【实例描述】

`main()` 是所有程序的入口函数, 但是在 `main()` 函数结束后还能再执行代码, 此时需在 `main()` 中注册函数。在 `main()` 的最后一行代码被执行完后, 还能执行新的代码, 效果如图 6-17 所示。

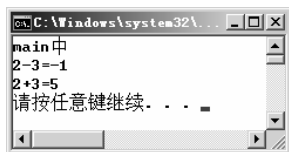


图 6-17 main()后执行代码

【实现过程】

利用 `atexit()` 注册的函数返回类型必须是 `void`，其代码如下：

```
01 void add() //加法
02 {
03     cout<<"2+3="<<2+3<<endl;
04 }
05 void subtract() //减法
06 {
07     cout<<"2-3="<<2-3<<endl;
08 }
09 void main()
10 {
11     atexit(add); //main 后执行加法
12     atexit(subtract); //main 后执行减法
13     cout<<"main 中"<<endl;
14 }
```

【代码解析】

第 01~08 行的 `add()` 和 `subtract()` 是在 `main()` 中注册的后执行函数，第 11、12 行的格式为注册函数，总结如下：

`atexit(无返回值类型函数名);`

虽然第 13 行位于最后，却是第一个执行。另外，先被注册的函数后被执行。



注意：`_onexit()` 也可以注册函数，但不能是 `void` 返回类型，返回类型可以为 `int` 型。



实例 103 阶乘计算 1 到 100 的积（递归）

【实例描述】

阶乘运算可以通过递归函数实现，本实例实现计算 1 到 100 的积，即 100 的阶乘。实现函数形式如下：

```
double jiecheng(int n);
```

效果如图 6-18 所示。

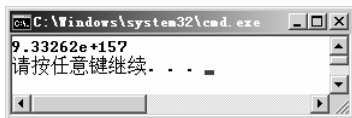


图 6-18 递归函数实现阶乘



【实现过程】

定义变量 $m=100$ ，递归函数的实现是函数调用自身，jiecheng()和 main()函数的代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 double jiecheng(int n)
05 {
06     if(n==0)                //0 的阶乘
07         return 1;
08     else
09         return n*jiecheng(n-1);    //递归调用
10 }
11
12 int main()
13 {
14     int m=100;
15     cout<<jiecheng(m)<<endl;
16     return 0;
17 }
```

【代码解析】

第 04~10 行是函数 jiecheng()的实现，第 12~17 行是 main()的实现，第 15 行调用 jiecheng()函数。在函数 jiecheng()中，首先判断输入参数 n 是否为 0。如果输入参数为 0，返回 1；如果不为 0，返回 n 乘以 jiecheng($n-1$)（再一次调用自身，但输入参数的值为 $n-1$ ）。

以数值 100 为例，首次调用函数 jiecheng()，因为不等于 0，返回 $100*jiecheng(99)$ 。此时 jiecheng(99)继续执行代码，由于还不为 0，返回 $99*jiecheng(98)$ ，直到输入参数为 0。由上所述可得，结果为 $100 \times 99 \times 98 \times 97 \times \dots \times 1$ ，即 100 的阶乘。



注意：递归函数是一直返回，直到不再调用自身，即终止。



实例 104 验证码（函数实现）

【实例描述】

现今，验证码的使用随处可见，比如网上营业厅登录手机号码等。本实例实现简单的验证码，由大写字母、小写字母和数字组成。本实例效果如图 6-19 所示。

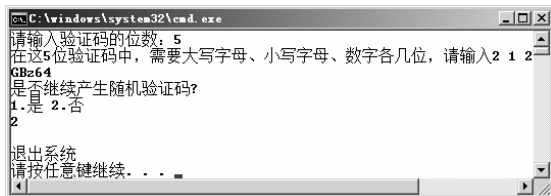


图 6-19 函数实现验证码

【实现过程】

定义 3 个函数 generate_lower()、generate_upper()和 generate_number()，分别表示生成小写



字母、大写字母和数字。输出规则为：先输出大写字母、小写字母，再输出数字。对于生成验证码的位数，由用户从屏幕输入，对应每种生成个数也从屏幕输入，分别由变量 num、upper、lower 和 number 存储，代码如下：

```

01  #include <iostream>
02  #include <stdlib.h>
03  #include <time.h>
04  using namespace std;
05
06  void generate_lower()                //生成小写
07  {
08      char lower;
09      int lower_index;
10      lower_index=1+(int) (26.0*rand() / (RAND_MAX+1.0)); //生成随机数
11      lower=char(lower_index+96);      //生成小写字母
12      cout<<lower;
13  }
14  void generate_upper()                //生成大写
15  {
16      char upper;
17      int upper_index;
18      upper_index=1+(int) (26.0*rand() / (RAND_MAX+1.0)); //生成随机数
19      upper=char(64+upper_index);      //生成大写字母
20      cout<<upper;
21  }
22  void generate_number()               //生成数字
23  {
24      int number;
25      number=0+(int) (9.0*rand() / (RAND_MAX+0.0));      //生成 0~9 的随机数
26      cout<<number;
27  }
28  int main()
29  {
30      srand((int)time(0));              //种子值
31      int num;                          //验证码位数
32      int upper;                        //大写个数
33      int lower;                        //小写个数
34      int number;                      //数字个数
35      while(1)
36      {
37          cout<<"请输入验证码的位数: ";
38          cin>>num;
39          cout<<"在这"<<num<<"位验证码中，需要大写字母、小写字母、数字各几位，请输入 40 入";
40          cin>>upper>>lower>>number;
41          while(upper)                  //生成大写字母
42          {
43              generate_upper();          //调用大写
44              upper--;                    //减 1
45          }
46          while(lower)                  //生成小写字母
47          {
48              generate_lower();          //调用小写
49              lower--;                    //减 1
50          }
51          while(number)                 //生成数字
52          {
53              generate_number();          //调用数字
54              number--;                  //减 1
55          }
56      }
57  }

```



```
56         }
57         cout<<endl<<"是否继续产生随即验证码?"<<endl;
58         cout<<"1.是 2.否"<<endl;
59         int y;
60         cin>>y;
61         cout<<endl;
62         if(y==1)                                //继续
63         {}
64         if(y==2)                                //退出
65         {
66             cout<<"退出系统"<<endl;
67             break;
68         }
69     }
70     return 0;
71 }
```

【代码解析】

第 06~13 行是 generate_lower()生成小写字母,第 14~21 行是 generate_upper()生成大写字母,第 22~27 是 generate_number()生成数字。在第 28~70 行的 main()函数中有一个 while(1) 循环,如果不继续产生随机验证码,则退出系统。



注意: 验证码中大写字母、小写字母和数字的排序也可以随机,此为程序改进方向。



实例 105 DOS 命令解释器(使 main 函数接收参数)

【实例描述】

main 函数可以接收参数解释 DOS 命令,它是在启动运行时传递参数的,格式有两种,分别如下:

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

带参数的 main 函数形成的.exe 文件需配合命令提示符使用,在命令提示符输入环境下,一条完整的命令包括两部分:命令和参数。其中,命令是.exe 文件全称,参数是用来传入 main 函数的。此外,参数可以是多个,命令居于首位。命令和参数间、参数和参数间都以空格隔开,格式如下:

命令 参数 1 参数 2 参数 3...

效果如图 6-20 所示。

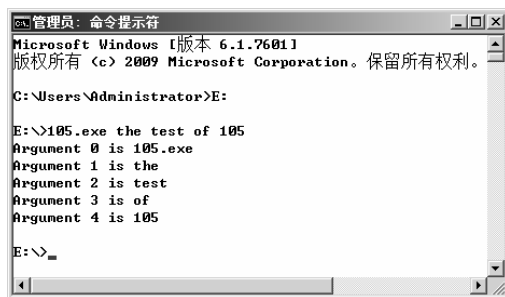


图 6-20 main 函数接收参数



【实现过程】

使 main 函数接收参数，并逐个输出所接收的命令和参数值，代码如下：

```
01 #include<iostream>
02 using namespace std;
03
04 int main(int argc,char **argv)           //带参数的main()函数
05 {
06     for(int i=0;i<argc;i++)
07         cout<<"Argument "<<i<<" is "<<argv[i]<<endl;    //输出参数值
08     return 0;
09 }
```

【代码解析】

其中，第 1 个参数是在命令队列中有几条命令，第 2 个参数是命令的首地址。在弹出的命令提示符环境下随便输入字符，以空格隔开，如图 6-20 所示。首先将当前命令路径改为 105.exe 的当前目录下，本实例在 E:\下。之后键入 105.exe the test of 105，输出 5 个 Argument。



注意：main 函数的第 2 个参数表示二维内存，因为接收的命令不是单个字符。



实例 106 补充代码使输出结果成立

【实例描述】

本实例继续实例 105 介绍，实现接收命令并响应。具体效果需要命令提示符的配合，命令提示符窗口的打开可通过单击“系统”→“运行”，在弹出的对话框中输入 cmd 命令实现。如果命令有 4 个，则输出一共有 4 个命令，否则输出没有 4 个命令，效果如图 6-21 所示。

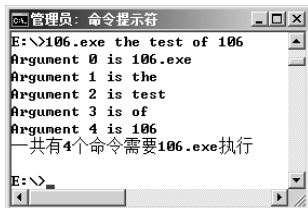


图 6-21 命令提示符下调用 106.exe 可执行文件

【实现过程】

判断接收的参数是否为 4 个，代码如下：

```
01 #include<iostream>
02 using namespace std;
03
04 int main(int argc,char **argv)           //带参数的main()函数
05 {
06     for(int i=0;i<argc;i++)
07         cout<<"Argument "<<i<<" is "<<argv[i]<<endl;    //输出参数值
08     if(argc==5)
09         cout<<"一共有 4 个命令需要 106.exe 执行"<<endl;
10     else
```



```

11         cout<<"不是 4 个命令"<<endl;
12         return 0;
13     }

```

【代码解析】

继续实例 105，新加入的代码为第 08~11 行。当判断有几个命令时，需要加 1（包含可执行文件的全名），如第 08 行。



注意：还可以使用第 2 种 main 函数接收参数的格式，即第 2 个参数用 char* argv[] 代替。



实例 107 互动式程序的基本框架

【实例描述】

综合上述各例可知，大多数程序与用户进行互动。首先注明该互动程序的功能，需要用户输入什么内容。当系统获取用户输入的数据后，调用相应的函数实现功能，并输出结果。之后由用户选择是继续实例的演示，还是退出程序。本实例对此类型的互动式程序编写基本框架，效果如图 6-22 所示。



图 6-22 互动式程序的基本功能

【实现过程】

定义函数 fun1()、fun2()、fun3()和 function()，分别实现不同的功能。在 function()中根据输入参数从 fun1()到 fun3()中选择不同的函数实现相应的功能。在 main()函数中定义整型变量 choice，获取用户想要实现哪种功能，代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 void fun1 ()                                //功能 1
05 {cout<<"功能 1 实现"<<endl;}
06 void fun2 ()                                //功能 2
07 {cout<<"功能 2 实现"<<endl;}
08 void fun3 ()                                //功能 3
09 {cout<<"功能 3 实现"<<endl;}
10 void function(int a)                        //总功能
11 {
12     switch(a)

```



```

13     {
14     case 1:                                //功能 1
15         fun1();
16         break;
17     case 2:                                //功能 2
18         fun2();
19         break;
20     case 3:                                //功能 3
21         fun3();
22         break;
23     default:
24         break;
25     }
26 }
27 void main()
28 {
29     int choice;                            //选择
30     cout<<"本实例实现最基本的互动式程序, "<<"\n";
31     cout<<"可实现的功能有 3 种: 1-功能 1, 2-功能 2, 3-功能 3, \n";
32     bool flag=true;                        //标志量
33     while(flag==true)
34     {
35         cout<<"请输入功能编号: "<<endl;
36         cin>>choice;
37         function(choice);                  //选择功能
38         cout<<"是否继续? 0-是, 1-否";
39         int x;
40         cin>>x;
41         if(x==0)                          //继续
42         {}
43         else if(x==1)                      //退出
44         {
45             flag=false;
46             cout<<"退出成功"<<endl;
47         }
48     }
49 }

```

【代码解析】

第 04~09 行是 fun1()、fun2()和 fun3()三种功能的实现,第 10~26 行根据用户的选择,响应对应的功能函数,用 switch 结构实现。第 27~49 行是 main()函数的定义体,第 32 行定义的布尔变量 flag 用以标识当前是否继续执行功能的实现。它的状态改变由第 40 行输入 x 值在第 41~47 行的 if…else if 判断中决定。



注意: 对于本实例实现的互动式程序框架,根据不同的功能实现改变相应的代码,即可形成新的实例。



实例 108 设计一个数据查询系统

【实例描述】

本实例实现一个数据查询系统,现已知数据库中包含的数据,用户根据自己的意愿查询该



数据是否存在。本实例模拟火车票当日的售卖情况，效果如图 6-23 所示。

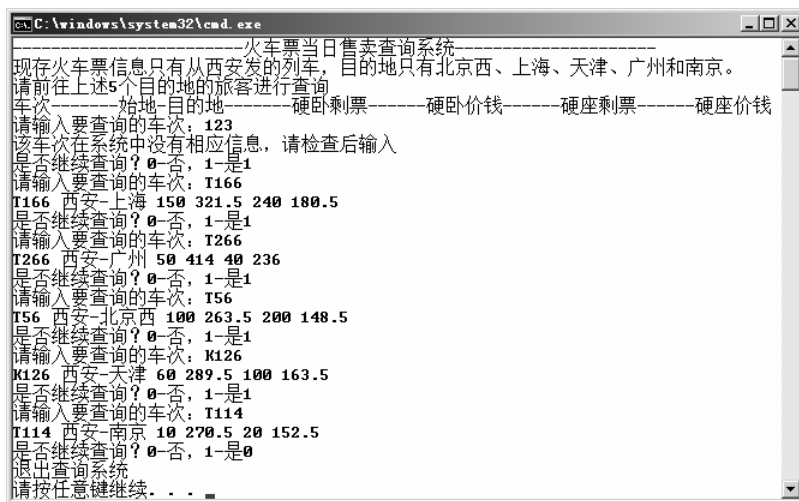


图 6-23 火车票数据查询系统

【实现过程】

定义两个函数 `init()` 和 `query()`，分别初始化系统信息和查询信息之用。定义结构体 `ticket`，其中包括车次、起始一目的地、硬卧票数、硬卧价钱、硬座票数和其价钱等信息。初始化状态下，系统只含有 5 条火车票信息。系统根据用户的意愿输出相应的车次信息，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 struct ticket
05 {
06     char train_number[5];           //车次
07     char train_name[50];           //从起点到终点
08     int yingwo_num;                 //硬卧票数
09     float yingwo_cash;              //硬卧钱
10     int yingzuo_num;                //硬座票数
11     float yingzuo_cash;             //硬座钱
12 };
13 ticket Ticket[5];                  //数据库目前只存 5 条项目
14 void init()                         //车票信息初始化
15 {
16     strcpy(Ticket[0].train_number, "T56");
17     strcpy(Ticket[0].train_name, "西安—北京西");
18     Ticket[0].yingwo_num=100;
19     Ticket[0].yingwo_cash=263.5;
20     Ticket[0].yingzuo_num=200;
21     Ticket[0].yingzuo_cash=148.5;
22
23     strcpy(Ticket[1].train_number, "K126");
24     strcpy(Ticket[1].train_name, "西安—天津");
25     Ticket[1].yingwo_num=60;
26     Ticket[1].yingwo_cash=289.5;
27     Ticket[1].yingzuo_num=100;
28     Ticket[1].yingzuo_cash=163.5;
29
30     strcpy(Ticket[2].train_number, "T166");
```



```

31     strcpy(Ticket[2].train_name, "西安—上海");
32     Ticket[2].yingwo_num=150;
33     Ticket[2].yingwo_cash=321.5;
34     Ticket[2].yingzuo_num=240;
35     Ticket[2].yingzuo_cash=180.5;
36
37     strcpy(Ticket[3].train_number, "T266");
38     strcpy(Ticket[3].train_name, "西安—广州");
39     Ticket[3].yingwo_num=50;
40     Ticket[3].yingwo_cash=414;
41     Ticket[3].yingzuo_num=40;
42     Ticket[3].yingzuo_cash=236;
43
44     strcpy(Ticket[4].train_number, "T114");
45     strcpy(Ticket[4].train_name, "西安—南京");
46     Ticket[4].yingwo_num=10;
47     Ticket[4].yingwo_cash=270.5;
48     Ticket[4].yingzuo_num=20;
49     Ticket[4].yingzuo_cash=152.5;
50 }
51 void query()
52 {
53     char checi[5];
54     bool flag=true;
55     bool whether=false;           //是否有
56     while(flag==true)
57     {
58         cout<<"请输入要查询的车次: ";
59         cin>>checi;
60         for(int i=0;i<5;i++)
61         {
62             if(strcmp(Ticket[i].train_number, checi)==0)
63             {
64                 whether=true;
65                 cout<<Ticket[i].train_number<<" "<<Ticket[i].train_name;
66                 cout<<" "<<Ticket[i].yingwo_num<<" "<<Ticket[i].yingwo_cash;
67                 cout<<" "<<Ticket[i].yingzuo_num<<" "<<Ticket[i].yingzuo_
68                     cash<<endl;
69                 i=5;
70             }
71             else
72                 whether=false;
73         }
74         if(whether==true)           //有
75         {}
76         else
77         {cout<<"该车次在系统中没有相应信息, 请检查后输入"<<endl;}
78         cout<<"是否继续查询? 0-否, 1-是";
79         int x;
80         cin>>x;
81         if(x==0)
82         {
83             flag=false;
84             cout<<"退出查询系统"<<endl;
85         }
86         else if(x==1)
87         {}
88     }
89 int main()
90 {
91     cout<<"-----火车票当日售卖查询系统-----"

```



```

    "<<endl;
92    cout<<"现存火车票信息只有从西安发的列车，";
93    cout<<"目的地只有北京西、上海、天津、广州和南京。"<<endl;
94    cout<<"请前往上述 5 个目的地的旅客进行查询"<<endl;
95    cout<<"车次-----始地—目的地-----硬卧剩票";
96    cout<<"-----硬卧价钱-----硬座剩票-----硬座价钱"<<endl;
97    init(); //初始化信息
98    query(); //查询
99    return 0;
00 }

```

【代码解析】

第 04~12 行定义车票信息结构体 ticket，第 13 行定义全局变量 Ticket[5]，存储现有的 5 条火车票信息。第 14~50 行为函数 init() 的定义体，初始化 5 条火车票信息。第 51~86 行为具有查询功能的函数 query()。如果所要查询的车次在当前系统中没有记录，系统输出信息无提示，如第 76 行。

只要有一条信息与查询的车次相匹配，则将标志量 whether（系统中是否有该信息）置为 true，并且跳出 for 循环（第 64~68 行）。完成第 1 次信息查询后，系统提示用户是否进行下一次查询，可以输入 0 或 1，以表示不需要或需要继续查询。



注意：在第 60~72 行的 for 循环中，一定要在相应的条件置 whether 值为 true 和 false，一个都不能少，否则第 73~76 行的 if...else 只能成立一次。



实例 109 学生成绩统计

【实例描述】

假设数学功课的总分为 100 分，现设计程序统计一个班级中学生数学成绩的分布情况。成绩分为 5 个级别，分别为：<60（不及格）、60~70（含 60 不含 70）、70~80（含 70 不含 80）、80~90（含 80 不含 90）、>90（优秀），效果如图 6-24 所示。

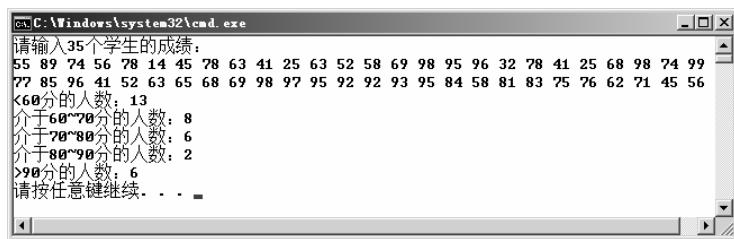


图 6-24 统计学生成绩

【实现过程】

定义 double 型数组 math_score[35] 表示 35 个学生的数学成绩，int 型数组 num[5] 表示 5 个阶段的人数，从键盘输入 35 个学生的成绩，然后调用函数 cal_num() 统计各个阶段的人数，最后输出各自的人数，代码如下：

```
01 #include <iostream>
```



```
02 using namespace std;
03
04 void cal_num(double *a, int *b, int number)
05 {
06     for(int i=0;i<number;i++)
07     {
08         if(a[i]<60)                                //不及格
09             b[0]++;
10         else if(a[i]>=60 && a[i]<70)
11             b[1]++;
12         else if(a[i]>=70 && a[i]<80)
13             b[2]++;
14         else if(a[i]>=80 && a[i]<90)
15             b[3]++;
16         else                                        //优秀
17             b[4]++;
18     }
19 }
20 void main()
21 {
22     double math_score[35];                        //班级有 35 个学生
23     int num[5]={0};                                //5 个等级的人数
24     cout<<"请输入 35 个学生的成绩: "<<endl;
25     for(int i=0;i<35;i++)
26         cin>>math_score[i];
27     cal_num(math_score,num,35);
28     cout<<"<60 分的人数: "<<num[0]<<endl;
29     cout<<"介于 60~70 分的人数: "<<num[1]<<endl;
30     cout<<"介于 70~80 分的人数: "<<num[2]<<endl;
31     cout<<"介于 80~90 分的人数: "<<num[3]<<endl;
32     cout<<">90 分的人数: "<<num[4]<<endl;
33 }
```

【代码解析】

第 04~19 行定义函数 `cal_num()`，利用 `if...else if...else` 结构判断当前 `a[i]` 的值介于哪个阶段，对应的数组 `b` 元素值加 1（第 08~16 行），最后由第 28~32 行输出结果。



注意：从图 6-24 可知，即使输入的数值个数大于 35，程序也只取前 35 个数。

第 7 章 C++类基本应用

类是 C++ 语言面向对象开发的必要条件，本章着重介绍类的基本应用，包括对象的创建、继承、多态等应用。此外，还涉及类接口的应用。对于类的基本知识，本章以应用实例的形式给出演示，以便更有效地掌握类的应用。



实例 110 产量统计（计算 A 村各类农作物的产量）

【实例描述】

本实例利用类的应用实现统计 A 村各类农作物的产量。已知 A 村现有 3 类农作物：蔬菜、水果和稻谷。输入 3 类农作物的占地亩数，以及每类农作物每亩地可产出的数量，最后输出各类农作物的产量，效果如图 7-1 所示。

【实现过程】

定义类 Cun_A，由其带参数的构造函数获取 3 类农作物占地多少亩。成员函数 SetPerProduct() 获取 3 类农作物在每亩地分别能产出多少。veget()、fruit() 和 grain() 函数计算这 3 类农作物各自的产量。代码如下：

```
01  #include <iostream>
02  using namespace std;
03
04  class Cun_A
05  {
06  public:
07      Cun_A(double x,double y,double z)           //带参数的构造函数
08      {
09          acer_veget=x;
10          acer_fruit=y;
11          acer_grain=z;
12      }
13      ~Cun_A(){}                                   //析构函数
14      void SetPerProduct(double a,double b,double c)
15      {
16          output_per_acerV=a;
17          output_per_acerF=b;
18          output_per_acerG=c;
19      }
20      void veget()                                   //蔬菜
21      {
22          double result=acer_veget*output_per_acerV;
23          cout<<"A 村蔬菜的产量为: "<<result<<endl;
24      }
```

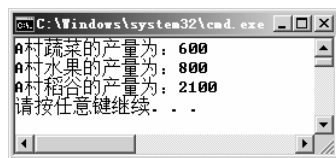


图 7-1 产量统计



```

25     void fruit()                                //水果
26     {
27         double result=acer_fruit*output_per_acerF;
28         cout<<"A 村水果的产量为: "<<result<<endl;
29     }
30     void grain()                                //稻谷
31     {
32         double result=acer_grain*output_per_acerG;
33         cout<<"A 村稻谷的产量为: "<<result<<endl;
34     }
35 private:
36     double acer_veget;                          //蔬菜作物的占地
37     double output_per_acerV;                    //每亩地产蔬菜多少
38     double acer_fruit;                          //水果作物的占地
39     double output_per_acerF;                    //每亩地产水果多少
40     double acer_grain;                          //稻谷作物的占地
41     double output_per_acerG;                    //每亩地产稻谷多少
42 };
43 void main()
44 {
45     Cun_A A(1.2,2,3);                          //类对象
46     A.SetPerProduct(500,400,700);               //设置产量
47     A.veget();                                  //计算蔬菜产量
48     A.fruit();                                  //计算水果产量
49     A.grain();                                  //计算稻谷产量
50 }

```

【代码解析】

第 04~44 行是类 Cun_A 的定义，其声明和定义在一处。第 07~12 行是其带参数的构造函数，输入参数赋值给类的私有成员变量。第 13 行是析构函数的定义，第 14~19 行定义成员函数 SetPerProduct()将输入参数赋值私有成员变量。第 20~34 行分别是计算蔬菜、水果、稻谷农作物的产量，第 35~41 行是私有成员变量的定义。第 45 行定义类对象 A，利用带参数的构造函数创建，之后调用其成员函数。



注意：类的定义必须是方括号后加英文状态下的分号。



实例 111 乡村生产总值（同类对象数据统计）

【实例描述】

本实例在实例 110 的基础上，统计 3 个农村中 3 类农作物的生产总值，即定义 A、B、C 三个对象，之后相加得到结果，运行效果如图 7-2 所示。



图 7-2 同类对象数据统计



【实现过程】

定义类 `Cun_product`，其中成员变量 `veget_result`、`fruit_result` 和 `grain_result` 分别表示蔬菜、水果和稻谷的总产量，它们为 `private` 访问类型。为了使类对象相加，需要重载运算符 (+)。因为需要访问私有成员变量，所以定义该重载运算符函数为友元函数，具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 class Cun_product
05 {
06 public:
07     Cun_product() {veget_result=0;fruit_result=0;grain_result=0;} //构造函数
08     Cun_product(double x,double y,double z) //有参数构造函数
09     {
10         veget_result=x;
11         fruit_result=y;
12         grain_result=z;
13     }
14     ~Cun_product() {}
15     friend Cun_product operator+(Cun_product cun1,Cun_product cun2); //重载友元运算符函数
16
17     void output() //输出
18     {
19         cout<<veget_result<<","<<fruit_result<<","<<grain_result<<endl;
20     }
21 private: //私有成员变量
22     double veget_result;
23     double fruit_result;
24     double grain_result;
25 };
26 Cun_product operator+(Cun_product cun1,Cun_product cun2) //运算符函数定义
27 {
28     Cun_product add;
29     add.veget_result=cun1.veget_result+cun2.veget_result;
30     add.fruit_result=cun1.fruit_result+cun2.fruit_result;
31     add.grain_result=cun1.grain_result+cun2.grain_result;
32     return add;
33 }
34 void main()
35 {
36     Cun_product cun_A(1.2,3,4),cun_B(2.3,5,1),cun_C(1.4,3,2),cun;
37     cun=cun_A+cun_B+cun_C; //加法运算
38     cout<<"ABC 村总共的蔬菜、水果和稻谷产量分别为: ";
39     cun.output();
40 }
```

【代码解析】

第 04~25 行是类 `Cun_product` 的定义，其有两个构造函数：带参数的和不带参数的。第 15 行定义运算符 (+) 重载函数为类 `Cun_product` 的友元函数，用关键字 `friend` 标识。第 17~20 行定义的成员函数 `output()` 输出 3 个私有变量的值。

第 26~33 行是友元函数的定义体，返回值类型为类。第 37 行为 3 个类对象相加，返回值类型也是 `Cun_product`。最后由它的成员函数输出结果，如第 39 行。



注意：如果不将运算符重载函数定义为类的友元函数，将不能访问其私有变量。



实例 112 求圆的面积和周长

【实例描述】

本实例利用类实现求圆的面积和周长，面积和周长的计算都会用到圆的半径，它们的求解公式如下：

面积=圆周率*半径*半径；

周长=2*圆周率*半径；

求解效果如图 7-3 所示。

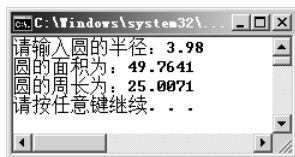


图 7-3 求圆的面积和周长

【实现过程】

定义类 `circle`，包括两个公共访问类型的函数 `area()` 和 `circumfer()`，分别用于求解面积和周长。还有一个公共访问成员变量 `radius`，表示圆的半径。宏定义圆周率 `Pi`，在 `main()` 函数中定义类对象 `_circle`，由用户从控制台界面输入半径的值，然后输出面积和周长的值，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 #define Pi 3.14159265
05 class circle
06 {
07 public:
08     double area(double x)                //面积
09     {
10         return Pi*x*x;
11     }
12     double circumfer(double y)           //周长
13     {
14         return 2*Pi*y;
15     }
16 public:
17     double radius;                       //半径
18 };
19 void main()
20 {
21     circle _circle;                      //类对象
22     cout<<"请输入圆的半径: ";
23     cin>>_circle.radius;                 //输入半径
24     cout<<"圆的面积为: "<<_circle.area(_circle.radius)<<endl; //输出面积
25     cout<<"圆的周长为: "<<_circle.circumfer(_circle.radius)<<endl; //输出周长
26 }
```

【代码解析】

第 04 行宏定义圆周率的值，第 05~18 行定义类 `circle`，包括成员函数 `area()` 和 `circumfer()`，以及成员变量 `radius`。类只有在声明类对象才可以被使用，如第 21 行。由该对象 `_circle` 调用其内部函数和变量，如第 23~25 行。



注意：类中只有访问类型为 `public` 的成员才可以被外部访问。



实例 113 动物对象进化（继承）

【实例描述】

人类属于高等动物，且学术界有些观点指出人是由猿进化而来。因此可以理解为动物的祖先进化为猿，然后进化为人类。本实例通过类的继承演示人类从动物的祖先和猿继承了哪些特质，运行效果如图 7-4 所示。

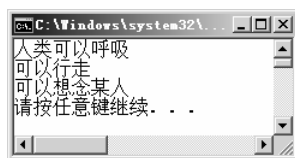


图 7-4 类继承示例

【实现过程】

定义 3 个类，分别为 `animal_ancestor`（动物的祖先）、`ape`（猿）和 `human`（人类）。其中，`ape` 继承自 `animal_ancestor`，`human` 继承自 `ape`，每个进化阶段动物都有各自的功能。比如，`animal_ancestor` 可以呼吸，`ape` 既可以呼吸，还可以行走。而 `human` 除了具有上述两个功能外，还可以想念某人。本实例由这 3 个类实现输出 `human` 可以做到哪些事情，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 class animal_ancestor           //基类
05 {
06 public:
07     animal_ancestor() {}
08     ~animal_ancestor() {}
09     void respire()               //呼吸
10     {
11         cout<<"可以呼吸"<<endl;
12     }
13 };
14 class ape:public animal_ancestor //由基类派生的子类
15 {
16 public:
17     ape() {}
18     ~ape() {}
19     void move()                  //行走
20     {
21         cout<<"可以行走"<<endl;
22     }
23 };
24 class human:public ape           //由子类派生的子类
25 {
26 public:
27     human() {}
28     ~human() {}
29     void miss()
30     {
31         cout<<"可以想念某人"<<endl;
32     }
33 };
34 void main()
35 {
36     human _human;                //子类 human
37     cout<<"人类";
38     _human.respire();
```



```

39         _human.move();
40         _human.miss();
41     }

```

【代码解析】

第 04~13 是类 `animal_ancestor` 的定义体，它具有呼吸的功能，由成员函数 `respire()` 完成。第 15~24 行是类 `ape` 的定义体，它只有一个成员函数 `move()`（第 19~22 行），完成行走功能。此外，由于它继承自 `animal_ancestor`，所以它也有呼吸的功能。同理，`human` 除了自身的 `miss()` 功能（第 29~32 行），还继承有呼吸和行走功能。现总结上述 3 个类的继承关系，如图 7-5 所示。

其中，箭头表示从哪个类继承的，`ape` 从 `animal_ancestor` 继承来。

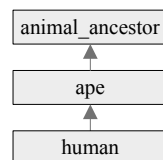


图 7-5 `animal_ancestor`、`ape` 和 `human` 类的继承关系



注意：类继承的实现可以节省很多代码量。



实例 114 员工月薪发放（多态）

【实例描述】

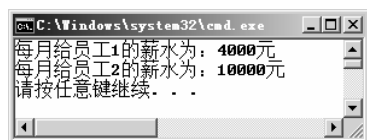


图 7-6 类的多态性实现

类的多态性体现在当响应同一条消息时，结果却不一样，也即结果呈多样性展示。类多态的应用存在于基类的指针指向子类的对象。本实例演示员工月薪的发放，现有基类 `salary`，子类 `member1` 和 `member2` 都继承自类 `salary`。目的是实现当利用基类给员工 `member1` 和 `member2` 发月薪时，需要按各自的情况执行，运行效果如图 7-6 所示。

【实现过程】

基类 `salary` 中定义一个虚成员函数 `pay()`（用关键字 `virtual` 标识）。子类 `member1` 和 `member2` 也有一个同名的成员函数 `pay()`。但是基类 `salary`、子类 `member1` 和 `member2` 的 `pay()` 函数执行不一样的代码。之后定义基类 `salary` 的两个指针 `sa1` 和 `sa2`，分别指向子类 `member1` 和 `member2` 的对象 `m1` 和 `m2`。当每个指针执行 `pay()` 函数时，都执行指向对象的 `pay()` 函数。代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 class salary //基类
05 {
06 public:
07     salary(){}
08     ~salary(){}
09     virtual void pay(){}
10 private:
11 };
12 class member1:public salary //子类1
13 {
14 public:
15     member1(double a){_total=a;}

```



```

16     ~member1() {}
17     void pay()
18     {
19         cout<<"每月给员工 1 的薪水为: "<<_total<<"元"<<endl;
20     }
21 private:
22     double _total;
23 };
24 class member2:public salary          //子类 2
25 {
26 public:
27     member2(double a){_total=a;}
28     ~member2() {}
29     void pay()
30     {
31         cout<<"每月给员工 2 的薪水为: "<<_total<<"元"<<endl;
32     }
33 private:
34     double _total;
35 };
36 void main()
37 {
38     member1 m1(4000.0);              //子类 1 对象
39     member2 m2(10000.0);             //子类 2 对象
40     salary *sa1=&m1;                  //基类对象指向 m1
41     salary *sa2=&m2;                  //基类对象指向 m2
42     sa1->pay();                       //执行各自的成员函数
43     sa2->pay();
44 }

```

【代码解析】

第 04~11 行是基类 salary 的定义体，其中，第 09 行是虚函数 pay()，用于类多态的实现。第 12~23 行是子类 member1 的定义体，第 24~35 行是子类 member2 的定义体。第 38、39 行分别定义子类的对象 m1 和 m2。第 40、41 行定义基类指针，分别指向子类对象。第 42、43 行由基类指针分别执行 pay() 函数，实现多态。



注意：类多态的发生需要多个条件，一是类指针和引用的操作，二是基类和子类中有相同的函数，并且基类的该函数用 virtual 标识。



实例 115 家族性格遗传（纯虚函数）

【实例描述】

实例 114 提到使用虚函数实现多态，本实例演示如何使用纯虚函数实现家族性格遗传。下一代的性格是从上一代继承而来的，现定义基类 xingge 作为一个抽象类，子类 child1、child2 分别继承自抽象类 xingge。实例效果如图 7-7 所示。

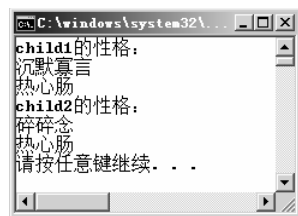


图 7-7 纯虚函数应用

【实现过程】

定义抽象类 xingge、子类 child1 和 child2。定义基类有两个纯虚函数，为 talk() 和 kind_



hearted()。实例化 child1 和 child2，由基类的指针取址，输出结果。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 class xingge //抽象类
05 {
06 public:
07     xingge(){} //构造函数
08     virtual void talk()=0; //纯虚函数
09     virtual void kind_hearted()=0; //纯虚函数
10 };
11 class child1:public xingge //子类 child1
12 {
13 public:
14     child1(){}
15     void talk()
16     {cout<<"沉默寡言"<<endl;}
17     void kind_hearted()
18     {cout<<"热心肠"<<endl;}
19 };
20 class child2:public xingge //子类 2
21 {
22 public:
23     child2(){}
24     void talk()
25     {cout<<"碎碎念"<<endl;}
26     void kind_hearted()
27     {cout<<"热心肠"<<endl;}
28 };
29 void main()
30 {
31     xingge *_xg; //抽象类指针
32     child1 _c1; //child1 实例化
33     child2 _c2; //child2 实例化
34     _xg=&_c1; //取址 child1
35     cout<<"child1 的性格: "<<endl;
36     _xg->talk();
37     _xg->kind_hearted();
38     _xg=&_c2; //取址 child2
39     cout<<"child2 的性格: "<<endl;
40     _xg->talk();
41     _xg->kind_hearted();
42 }
```

【代码解析】

第 04~10 行为抽象类 xingge 的定义，其中，第 08、09 行为纯虚函数的书写格式。第 11~19 行为子类 child1 的定义体，第 20~28 行为子类 child2 的定义体，分别对纯虚函数依照各自的情况定义。

在 main() 函数中，第 31~33 行定义基类指针，及实例化子类对象。第 34、38 行分别为子类对象取址，然后调用基类的纯虚函数，响应不同的指令。



注意：基类 xingge 属于抽象类，不能实例化，只能定义其指针和引用，如第 31 行。



实例 116 比谁跑得快（类+算法）

【实例描述】

本实例实现判断两个人谁跑得快，判断标准为各自的平均速度，如果平均速度大，则视为跑得快。运行效果如图 7-8 所示。

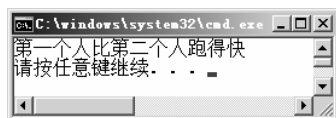


图 7-8 比谁跑得快

【实现过程】

定义类 `speed`，以计算每个人的平均速度。私有变量 `_time` 表示所用时间，`_distance` 表示跑了多远。在成员函数 `cal_speed()` 中返回平均速度 `_distance/_time`，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 class speed
05 {
06 public:
07     speed(double a,double b)                //构造函数
08     {
09         _time=a;                            //初始化私有变量
10         _distance=b;
11     }
12     ~speed(){}
13     double cal_speed()                      //计算平均速度
14     {
15         return _distance/_time;
16     }
17 private:
18     double _time;                          //时间
19     double _distance;                      //距离
20 };
21 void main()
22 {
23     speed peop1(1.5,5000);                 //花 1.5 小时跑了 5 千米
24     speed peop2(1,3000);                   //花 1 小时跑了 3 千米
25     if(peop1.cal_speed()>peop2.cal_speed()) //如果 1 大于 2
26         cout<<"第一个人比第二个人跑得快"<<endl;
27     else if(peop1.cal_speed()<peop2.cal_speed()) //如果 2 大于 1
28         cout<<"第二个人比第一个人跑得快"<<endl;
29     else
30         cout<<"两个人跑得一样快"<<endl;
31 }
```

【代码解析】

第 04~20 行为类 `speed` 的定义体，利用带参数的构造初始化对象的私有变量，如第 07~11 行。第 13~16 行的 `cal_speed()` 函数返回平均速度。在 `main()` 函数中，第 23、24 行定义两个 `speed` 对象，并初始化各自的距离和时间，第 25~30 行判断各自的大小，并输出不同的结果。



实例 117 错误的模糊引用（类继承问题）

【实例描述】

当类继承出现如图 7-9 所示的关系时，可能出现二义性问题。

上述继承关系被称为菱形继承。当类 `me` 访问基类 `human` 的公共成员变量时，系统将不知道通过哪个上一层类访问该成员变量。因为 `Chinese` 和 `woman` 都从 `human` 继承了成员变量，此二义性错误如图 7-10 所示。

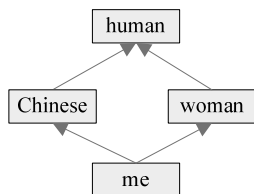


图 7-9 显式类型转换效果

```

1>----- Build started: Project: 116, Configuration: Debug Win32 -----
1>Compiling...
1>116.cpp
1>c:\users\administrator\desktop\07\116\116.cpp(26) : error C2385: ambiguous access of 'm_beauty'
1>    could be the 'm_beauty' in base 'human'
1>    or could be the 'm_beauty' in base 'human'
1>c:\users\administrator\desktop\07\116\116.cpp(32) : error C2385: ambiguous access of 'getBeauty'
1>    could be the 'getBeauty' in base 'human'
1>    or could be the 'getBeauty' in base 'human'
1>c:\users\administrator\desktop\07\116\116.cpp(32) : error C3881: 'getBeauty': identifier not found
1>Build log was saved at "file:///c:/Users/Administrator/Desktop/07/116/Debug/BuildLog.htm"
1>116 - 3 error(s), 0 warning(s)
===== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =====
  
```

图 7-10 多继承产生的二义性错误

【实现过程】

上述二义性现象可以通过虚继承避免，它的格式如下：

```

class A
{
};
class B: virtual public A
{
};
  
```

本实例实现的具体代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 class human //人类
05 {
06 public:
07     bool getBeauty() //成员函数
08     {
09         return m_beauty;
10     }
11     bool m_beauty;
12 };
13 class Chinese:virtual public human //中国人
14 {
15 };
16 class woman:virtual public human //女人
17 {
18 };
19 class me:public Chinese, public woman //多重继承
20 {
21 public:
22     me(bool a)
23     {
24         m_beauty=a;
  
```



```

25     }
26 };
27 void main()
28 {
29     me _me(true);           //多重继承类对蠊
30     cout<<_me.getBeauty()<<endl; //输出
31 }

```

【代码解析】

第 04~12 行是基类 `human` 的定义体，其中有成员变量 `m_beauty` 和成员函数 `getBeauty()`。第 13~18 行是子类 `Chinese` 和 `woman`，虚继承自基类 `human`。第 19~26 行的类 `me` 多继承于 `Chinese` 和 `woman`。由于虚继承，所以当类 `me` 的对象 `_me` 访问基类的成员变量时无二义性错误。



注意：现实编程中，很少用到多继承。



实例 118 实现类自动化管理内存

【实例描述】

本实例实现类自动化管理内存。所谓自动化管理内存，即相对于程序员手动分配和释放内存块来说，系统自动完成这些任务。本实例以一维整型内存为例演示自动化的内存管理，运行效果如图 7-11 所示。

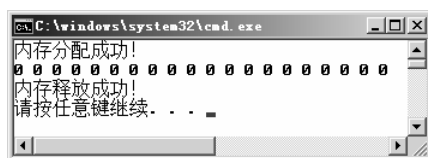


图 7-11 类自动化管理内存

【实现过程】

定义类 `memory` 用于模拟自动化内存管理，自动分配由构造函数实现，自动释放内存由析构函数实现。此外，还定义成员函数 `output()`。成员变量有一维内存 `memo` 和内存长度 `_length`，其作用是定义类 `memory` 对象 `_mem`，并输出其初始化值，代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 class memory
05 {
06 public:
07     memory(int length)
08     {
09         _length=length;
10         memo=new int[_length];           //申请一维内存
11         for(int i=0;i<_length;i++)       //初始化
12             memo[i]=0;
13         cout<<"内存分配成功!"<<endl;
14     }
15     ~memory()
16     {
17         delete []memo;                   //释放内存
18         if (memo!=NULL)
19             memo=NULL;                   //释放指针
20         cout<<"内存释放成功!"<<endl;
21     }
22     void output()                         //输出初始值

```



```

23     {
24         for(int i=0;i<_length;i++)
25             cout<<memo[i]<<" ";
26         cout<<endl;
27     }
28 private:
29     int *memo;                //一维内存
30     int _length;              //内存长度
31 };
32 void main()
33 {
34     memory _mem(20);           //申请内存长度为 20
35     _mem.output();             //调用输出函数
36 }

```

【代码解析】

第 07~14 行为构造函数，自动分配一维整型内存，申请内存方式为 new。第 15~21 行为析构函数，自动释放一维整型内存，对应的 delete 释放方式。第 22~27 行为成员函数 output() 定义体，输出内存 memo 的初始化值。第 29、30 行是类 memory 的两个成员变量。



注意：自动化管理内存体现的自动分配和释放分别由构造和析构函数实现。



实例 119 入学登记系统（类+算法+综合）

【实例描述】

本实例实现入学登记系统，对学生的各类信息进行记录，如姓名、性别、年龄、籍贯、手机号、专业及班级号等。如果当前输入的信息已知存在，则输出提示信息。本实例最大只记录 100 个学生的信息，效果如图 7-12 所示。

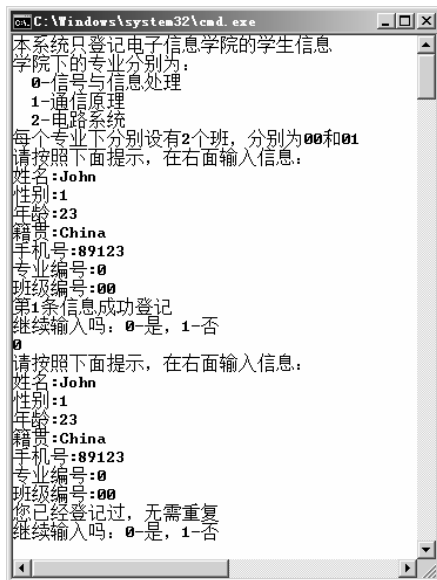


图 7-12 入学登记系统



【实现过程】

定义结构体 `stu_info` 用于保存学生的信息，其内包含 7 个变量。定义类 `student` 实现学生的登记，其内有成员变量 `stu_info` 型变量 `_stu[100]`（存储 100 个学生的信息）和 `temp`（存储当前输入的学生信息），变量 `num` 表示当前系统有几个学生信息，代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 struct stu_info           //学生信息
06 {
07     string name;           //姓名
08     int sex;               //性别，0-女，1-男
09     int age;               //年龄
10     string hometown;      //籍贯
11     string mobile_number;  //手机号
12     int major;             //专业
13     int class_num;        //班级号
14 };
15 class student
16 {
17 public:
18     student();             //构造函数
19     void Print();          //打印学院情况
20     void Input();          //输入信息
21     bool Certain();        //该条信息是否已存在
22     void Register();       //登记信息
23 private:
24     stu_info _stu[100];    //系统只能包括 100 人的信息
25     stu_info temp;         //临时信息
26     int num;               //总共几个人
27 };
28 student::student()
29 {
30     num=0;                 //初始只有 0 个人
31 }
32 void student::Print()      //打印信息
33 {
34     cout<<"本系统只登记电子信息学院的学生信息"<<endl;
35     cout<<"学院下的专业分别为："<<endl;
36     cout<<" 0-信号与信息处理"<<endl;
37     cout<<" 1-通信原理"<<endl;
38     cout<<" 2-电路系统"<<endl;
39     cout<<"每个专业下分别设有 2 个班，分别为 00 和 01"<<endl;
40 }
41 void student::Input()      //输入信息
42 {
43     cout<<"请按照下面提示，在右面输入信息："<<endl;
44     cout<<"姓名:"; cin>>temp.name;
45     cout<<"性别:"; cin>>temp.sex;
46     cout<<"年龄:"; cin>>temp.age;
47     cout<<"籍贯:"; cin>>temp.hometown;
48     cout<<"手机号:"; cin>>temp.mobile_number;
49     cout<<"专业编号:"; cin>>temp.major;        //输入专业
50     cout<<"班级编号:"; cin>>temp.class_num;    //输入班级号
```



```
51 }
52 void student::Register() //登记信息
53 {
54     if(Certain()==true) //当前信息还没有
55     {
56         if(num>99)
57             cout<<"登记系统已满，下午再来，谢谢"<<endl;
58         else
59         {
60             _stu[num]=temp; //赋值
61             cout<<"第"<<num+1<<"条信息成功登记"<<endl;
62         }
63         num++; //人数加1
64     }
65     else
66         cout<<"您已经登记过，无需重复"<<endl;
67 }
68 bool student::Certain() //是否
69 {
70     for(int i=0;i<num;i++) //与所有有效信息比较
71     {
72         if(_stu[i].age==temp.age && _stu[i].class_num==temp.class_num
73             && _stu[i].hometown==temp.hometown
74             && _stu[i].major==temp.major
75             && _stu[i].mobile_number==temp.mobile_number
76             && _stu[i].name==temp.name && _stu[i].sex==temp.sex) //已存在
77             return false;
78     }
79     return true;
80 }
81 void main()
82 {
83     student a; //学生类对象
84     int ch; //选择
85     a.Print(); //打印注意事项
86     do
87     {
88         a.Input(); //输入信息
89         a.Register(); //登记
90         cout<<"继续输入吗：0-是，1-否"<<endl;
91         cin>>ch;
92     }while(ch==0);
93 }
```

【代码解析】

第 05~14 行为结构体 `stu_info` 的定义，并定义姓名、性别、年龄、籍贯、手机号、专业及班级号等学生信息。第 15~27 行为类 `student` 的声明，它的构造函数对成员变量 `num` 进行初始化，如第 28~31 行。第 32~40 行定义成员函数 `Print()`，打印该系统的提示信息。第 41~51 行定义函数 `Input()` 以获取被登记学生的信息。

第 52~67 行为函数 `Register()` 的定义体，它调用函数 `Certain()`，由此判断当前输入的学生信息是否重复（如第 70~78 行）。如果重复，输出第 66 行，否则进一步判断，是否已经有 100 条信息。如果已经有 100 条信息，执行第 57 行；否则执行第 58~62 行。



实例 120 矩形范围(判断一个点是否超出矩形范围)

【实例描述】

本实例实现最简单的判断点是否在矩形范围内，现定义在直角坐标系中，以原点为起始点，X 轴正方向为矩形的长，Y 轴正方向为矩形的宽。然后给出点的坐标，判断该点是否超出矩形范围，效果如图 7-13 所示。

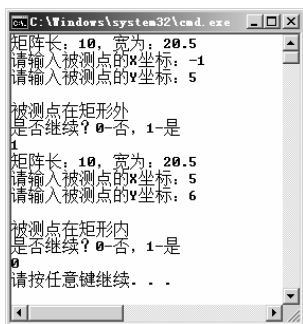


图 7-13 矩形范围

【实现过程】

定义类 `Point_And_Rectangle`，其有成员变量 `length` 和 `width`，分别表示矩形的长和宽。X 和 Y 分别表示被测点的坐标值。构造函数对成员变量初始化，`init()` 函数获取被测点的坐标值，`output()` 函数输出被测点的结果，其调用函数 `PointInRectangle()`（返回值为 `bool` 型）。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 class Point_And_Rectangle
05 {
06 public:
07     Point_And_Rectangle(float x, float y)           //构造函数
08     {
09         length=x;                                   //长度
10         width=y;                                    //宽度
11     }
12     void init()
13     {
14         cout<<"矩阵长: "<<length<<"，宽为: "<<width<<endl;
15         cout<<"请输入被测点的 X 坐标: ";
16         cin>>X;
17         cout<<"请输入被测点的 Y 坐标: ";
18         cin>>Y;
19         cout<<endl;
20     }
21     void output ()
22     {
23         if (PointInRectangle ()==false)             //矩形外
24             cout<<"被测点在矩形外\n";
25         else                                          //内
26             cout<<"被测点在矩形内\n";
```



```

27     }
28     bool PointInRectangle()
29     {
30         if(X<0 || X>length || Y<0 || Y> width)    //在矩形外
31             return false;
32         else    //矩形内
33             return true;
34     }
35 private:
36     float length;    //长度
37     float width;    //宽度
38     float X;    //被测点 x 坐标值
39     float Y;    //被测点 y 坐标值
40 };
41 void main()
42 {
43     int goon;    //是否继续
44     Point_And_Rectangle m(10,20.5);
45     do
46     {
47         m.init();    //初始化
48         m.output();    //输出结果
49         cout<<"是否继续? 0-否, 1-是"<<endl;
50         cin>>goon;
51     }while(goon==1);
52 }

```

【代码解析】

第 04~40 行为类 `Point_And_Rectangle` 的定义体，其中第 07~11 行为构造函数。第 12~20 行为 `init()` 函数，获取被测点的坐标值，如第 16、18 行。第 21~27 行为 `output()` 函数，第 23 行调用成员函数 `PointInRectangle()`，判断条件如第 30 行。在 `main()` 函数中，定义变量 `goon` 表示是否继续判断（第 43 行），第 44 行定义类对象 `m`，首先调用 `init()` 函数，初始化被测点，然后输出结果（第 47、48 行）。



注意：在判断点是否在矩形内时，列举不在矩形内的条件较简单，如第 30 行。



实例 121 学生的假期生活（接口）

【实例描述】

现有一学生的假期安排是吃饭、睡觉、打豆豆。学生可以依据当天的心情和前一天的时间安排当天做每件事的时间段（以小时为单位），效果如图 7-14 所示。

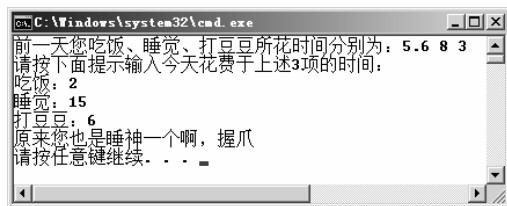


图 7-14 学生的假期生活



【实现过程】

定义类 `vocation`，其有 `double` 型成员变量 `_dinner`、`_sleep`、`_play`，分别表示吃饭、睡觉、打豆豆花费的时间。成员函数 `Setdinner()`、`Setsleep()`、`Setplay()` 设置 3 项时间，`Getdinner()`、`Getsleep()`、`Getplay()` 分别获取 3 项当前时间。在 `main()` 函数中，提示输入 3 项花费时间，并判断输出。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 class vocation
05 {
06 public:
07     vocation(){} //默认构造函数
08     vocation(double x,double y,double z); //带参数构造函数
09     ~vocation(){} //析构函数
10     void Setdinner(double a); //设置吃饭时间
11     double Getdinner(); //接口 1（获取吃饭时间）
12     void Setsleep(double a);
13     double Getsleep(); //接口 2（获取睡觉时间）
14     void Setplay(double a); //设置打豆豆时间
15     double Getplay(); //接口 3（获取打豆豆时间）
16 private:
17     double _dinner; //吃饭时间
18     double _sleep; //睡觉时间
19     double _play; //打豆豆时间
20 };
21 vocation::vocation(double x,double y,double z)
22 {
23     _dinner=x;
24     _sleep=y;
25     _play=z; //初始化
26 }
27 void vocation::Setdinner(double a)
28 {
29     _dinner=a;
30 }
31 double vocation::Getdinner()
32 {
33     return _dinner;
34 }
35 void vocation::Setsleep(double a)
36 {
37     _sleep=a;
38 }
39 double vocation::Getsleep()
40 {
41     return _sleep;
42 }
43 void vocation::Setplay(double a)
44 {
45     _play=a;
46 }
47 double vocation::Getplay()
48 {
49     return _play;
50 }
51 void main()
52 {
```



```

53     double a,b,c;                                //时间
54     vocation va(5.6,8.0,3.0);
55     cout<<"前一天您吃饭、睡觉、打豆豆所花时间分别为: ";
56     cout<<va.Getdinner()<<" "<<va.Getsleep()<<" "<<va.Getplay()<<endl;
57     cout<<"请按下面提示输入今天花费于上述3项的时间: "<<endl;
58     cout<<"吃饭: ";
59     cin>>a;
60     cout<<"睡觉: ";
61     cin>>b;
62     cout<<"打豆豆: ";
63     cin>>c;
64     if((a+b+c)>24)                                //大于一天
65         cout<<"人才啊,一天当"<<a+b+c<<"小时用"<<endl;
66     else
67     {
68         if(a>4.5)                                //大于吃饭限值
69             cout<<"真是饭桶! "<<endl;
70         va.Setdinner(a);                          //设置吃饭
71         if(b>12)                                  //大于睡觉限值
72             cout<<"原来您也是睡神一个啊,握爪"<<endl;
73         va.Setsleep(b);                          //设置睡觉
74         if(c>6)                                   //大于打豆豆限值
75             cout<<"亲,打这么长时间,手不疼吗? "<<endl;
76         va.Setplay(c);                            //设置打豆豆
77     }
78 }

```

【代码解析】

第04~20行为类 `vocation` 的声明,第21~50行为类成员函数的定义。第53行定义变量 `a`、`b`、`c` 保存由 `cin` 输入的值,第54行定义类对象,并初始化成员变量。第56行输出前一天的时间分布,第59、61、63行获取3项时间。第64行判断3项时间总和是否大于24小时,如果大于,则输出第65行,否则,执行 `else` (第67~77行)。在 `else` 中,继续判断 `a`、`b` 和 `c` 各自是否大于各项限值,如第68、71、74行,并输出条件语句和设置对象 `va` 关于3项的值(第70、73、76行)。



注意: 写程序一定要细化自己的思考方法。



实例 122 判断一个矩形是否成立

【实例描述】

本实例实现判断一个矩形是否成立,即定义4个坐标点,分别为 `A`、`B`、`C` 和 `D`。判断四边形 `ABCD` 是否为矩形。根据判定矩形的原理,判断四边形的4个角中如果有3个都为直角,则为矩形。效果如图7-15所示。

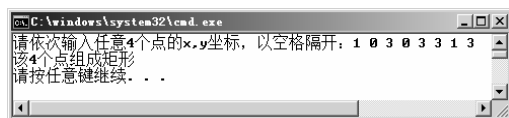


图 7-15 判断一个矩形是否成立



【实现过程】

定义结构体 `zuobiao`，其中包括两个 `double` 型变量，分别表示坐标点的 `x` 和 `y` 值。定义类 `Quadrilateral` 判断四边形是否为矩形，其中带参数的构造函数获取四边形的 4 个点坐标。成员函数 `Is_rectangle()` 判断该四边形是否为矩形，判定原理是四边形构成的 4 个向量相互乘积如果为 0，则两个向量的夹角为直角。如果有 3 个直角，则该四边形为矩形。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 struct zuobiao
05 {
06     double x;           //坐标 x
07     double y;           //坐标 y
08 };
09 class Quadrilateral
10 {
11 public:
12     Quadrilateral(zuobiao zb[])           //带参数构造函数
13     {
14         zhijiao=0;                         //初始为没有直角
15         for(int j=0;j<4;j++)
16         {
17             m_zb[j].x=zb[j].x;
18             m_zb[j].y=zb[j].y;
19         }
20     }
21     ~Quadrilateral(){}
22     bool Is_rectangle()
23     {
24         int k=0;
25         for(int i=1;i<4;i++)
26         {
27             if(i==3)                       //返回到起始那个点
28             {
29                 k=0;
30             }
31             else
32             {
33                 k=i+1;                     //判断下一条边
34             }
35
36             if((m_zb[i].x-m_zb[k].x)*(m_zb[i].x-m_zb[i-1].x)+(m_zb[i].y-m_zb[k].y)*
37             (m_zb[i].y-m_zb[i-1].y))==0)    //垂直
38             {
39                 zhijiao++;                 //直角个数加 1
40             }
41         }
42         if(zhijiao==3)
43             return true;
44         else
45             return false;
46     }
47 private:
48     zuobiao m_zb[4];
49     int zhijiao;
50 };
51 void main()
52 {
```



```

53     zuobiao _zb[4];                                //4 个点坐标
54     cout<<"请依次输入任意 4 个点的 x,y 坐标, 以空格隔开: ";
55     for(int i=0;i<4;i++)
56     {
57         cin>>_zb[i].x>>_zb[i].y;
58     }
59     Quadrilateral m_qu(_zb);                        //类对象
60     if(m_qu.Is_rectangle())                          //组成矩形
61         cout<<"该 4 个点组成矩形"<<endl;
62     else
63         cout<<"该 4 个点不能组成矩形"<<endl;
64 }

```

【代码解析】

第 04~08 行为坐标点结构体 `zuobiao` 的定义,第 09~50 行为类 `Quadrilateral` 的定义。其中,第 12~20 行为其带参数的构造函数定义体,初始化其成员变量 `zhijiao` 和 `m_zb`。第 22~46 行的函数 `Is_rectangle()` 判断四边形是否为矩形。其中,第 25 行的循环条件是判断 3 次,即只要有 3 个直角存在,就是矩形。第 27~34 行的 `if...else` 结构判断向量是否需要返回到原始处。



注意: 矩形有 4 条边,即边 0、边 1、边 2 和边 3。其中,边 0 和边 1 构成角,边 1 和边 2 构成角,边 2 和边 3 构成角,边 3 和边 0 构成角,所以最后一定要返回最起始的元素。



实例 123 类的静态成员变量应用(对象间数据共享)

【实例描述】

在非类的编程中,可以声明全局变量实现数据的共享。然而在类中,全局变量的使用具有局限性,此时可以定义为静态数据成员实现多个对象间的数据共享,效果如图 7-16 所示。



图 7-16 静态数据成员实现数据共享

【实现过程】

定义类 `Apple`, 其有成员变量 `apples` 为静态数据, `apple` 为非静态数据。成员函数 `setApples()` 给成员变量赋值, `getApples()` 获取静态变量的值, `getApple()` 获取非静态变量的值。非静态变量的初始化由构造函数实现, 然后赋值前后两种变量的值。代码如下:



```
01 #include <iostream>
02 using namespace std;
03
04 class Apple
05 {
06 private:
07     static int apples;           //静态成员变量，表示苹果个数
08     int apple;                  //非静态变量
09 public:
10     Apple() {apple=5;}          //构造函数
11     void setApples(int a,int b) //设置苹果个数
12     {
13         apples=a;               //静态
14         apple=b;                //非静态
15     }
16     static int getApples()       //获取静态苹果个数
17     {
18         return apples;          //返回苹果数目
19     }
20     int getApple()               //获取非静态变量值
21     {
22         return apple;
23     }
24 };
25 int Apple::apples=10;           /*初始化静态变量*/
26
27 void main()
28 {
29     cout<<"未定义 Apple 类对象前，其静态成员 apples="<<Apple::getApples()<<endl;
30     Apple a,b;                  //两个对象
31     cout<<"未赋值前，两个对象的各自变量值如下："<<endl;
32     cout<<"a.apples="<<a.getApples();
33     cout<<"\na.apple="<<a.getApple()<<endl;
34     cout<<"b.apples="<<b.getApples();
35     cout<<"\nb.apple="<<b.getApple()<<endl;
36     a.setApples(23,50);         //对象 a 赋值
37     cout<<"赋值后，两个对象的各自变量值如下："<<endl;
38     cout<<"a.apples="<<a.getApples();
39     cout<<"\na.apple="<<a.getApple()<<endl;
40     cout<<"b.apples="<<b.getApples();
41     cout<<"\nb.apple="<<b.getApple()<<endl;
42 }
```

【代码解析】

第 07、08 行分别为静态数据和非静态数据，第 10 行的构造函数对非静态数据进行初始化，而静态数据的初始化由第 25 行实现。第 11~15 行为函数 setApples()，设置成员变量值。第 16~19 行的静态函数返回静态数据的值，第 20~23 行返回非静态数据的值。

在未创建对象前，静态数据可以被访问，如第 29 行，但非静态数据不可以。第 30 行声明两个对象，首先输出未调用 setApples()函数时成员变量的值，如第 32~35 行。第 36 行给对象赋值，再由第 38~41 行输出其值。由效果图可知，静态变量可以实现不同对象间的数据共享，但非静态变量不可以。



注意：非静态变量的初始化需位于类体外。



实例 124 获取系统时间

【实例描述】

本实例演示如何获取系统时间，然后输出达到毫秒级的当前总时间、时、分、秒等计时，效果如图 7-17 所示。

【实现过程】

定义类 TIME，其有私有变量 timeM（标识总时间）、time_H（标识时）、time_M（标识分）、time_S（标识秒）和 millisecond（标识毫秒）。成员函数有 cal()（计算当前时间）、getSysime()（获取总时间）、getH()、getM()、getS()（获取当前时刻的时、分、秒）。代码如下：

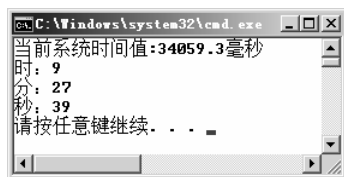


图 7-17 获取系统时间

```

01 #include <sys/timeb.h>
02 #include <time.h>
03 #include <iostream>
04 using namespace std;
05
06 class TIME
07 {
08 private:
09     double timeM;           //总时间，换算为毫秒
10     int time_H;             //时
11     int time_M;             //分
12     int time_S;             //秒
13     double millisecond;     //毫秒
14 public:
15     void cal();              //计算当前时间
16     double getSysime();      //获取总时间
17     int getH();              //获取时
18     int getM();              //获取分
19     int getS();              //获取秒
20 };
21 void TIME::cal()
22 {
23     struct _timeb timebuffer; //时间缓冲变量
24     char *timeline;
25     unsigned short millitml;
26     char temp_H[2];           //字符变换量
27     char temp_M[2];
28     char temp_S[2];
29     _ftime(&timebuffer);      //获取缓冲量
30     timeline = ctime(&(timebuffer.time)); //转换为 char*型
31     millitml = timebuffer.millitm; //获取毫秒值
32     millisecond = (double)millitml/1000; //变换为真正的毫秒
33
34     for(int i = 0; i < 2; i++)
35     {
36         temp_H[i] = timeline[i+11]; //从第 11 字节开始为时的字符值，下同
37     }
38     for(int j = 0; j < 2; j++)
39     {

```



```

40         temp_M[j] = timeline[j+14];           //分的字符值
41     }
42     for(int k = 0; k < 2; k++)
43     {
44         temp_S[k] = timeline[k+17];           //秒的字符值
45     }
46     time_H = atoi(temp_H);                     //时变换为整型
47     time_M = atoi(temp_M);                     //分变换为整型
48     time_S = atoi(temp_S);                     //秒变换为整型
49     timeM = double(time_H) * 3600 + double(time_M) * 60 + double(time_S) +
50         millisecond;                           //计算总时间
51 }
52 double TIME::getSystime()
53 {
54     return timeM;                             //返回总时间
55 }
56 int TIME::getH()
57 {
58     return time_H;                           //返回时
59 }
60 int TIME::getM()
61 {
62     return time_M;                           //返回分
63 }
64 int TIME::getS()
65 {
66     return time_S;                           //返回秒
67 }
68 void main()
69 {
70     TIME t;                                   //时间对象
71     t.cal();
72     cout<<"当前系统时间值:"<<t.getSystime()<<"毫秒"<<endl;
73     cout<<"时: "<<t.getH()<<endl;
74     cout<<"分: "<<t.getM()<<endl;
75     cout<<"秒: "<<t.getS()<<endl;
76 }

```

【代码解析】

第 01、02 行为计算时间必须包含的头文件，第 06~20 行为类 TIME 的声明，第 09~13 为成员变量声明，第 15~19 行为成员函数声明。第 21~51 行为成员函数 cal() 定义体，第 52~67 行分别为其他成员函数定义体。由于外部变量不能访问类的私有变量，所以利用成员函数返回。第 70 行定义类对象 t，首先调用 cal()，计算当前时间，然后输出当前时间、时、分、秒。



注意：第 46~48 行的 atoi() 函数是将字符串转换为整数，具体由后续章节介绍。另外，变量 timeline 包含整个时间，即年、月、日，所以应从某个位置获取。



实例 125 内联函数应用于计算两点间的距离

【实例描述】

在类内部被声明又被定义的函数称为内联函数，它的应用是解决函数调用的效率问题。一



般的调用函数，在进入被调用函数的地址前，先保存现场，再转入函数地址。待被调用函数执行完后，转回先前的地址，恢复现场，继续执行。而内联函数的使用是将该函数体直接放到当前执行地址，因此提高了效率。本实例利用内联函数计算两点间的距离，效果如图 7-18 所示。

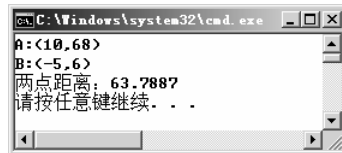


图 7-18 内联函数计算两点间的距离

【实现过程】

定义类 Line，其有成员变量 x1 和 y1（分别表示一个点的 x 轴和 y 轴的坐标值）。同理，x2 和 y2 也是如此。成员函数为 printPoint()（打印两点坐标）、getDis()（计算两点距离）。另外，其有两个构造函数，即默认的和带参数的。代码如下：

```

01  #include <iostream>
02  using namespace std;
03
04  class Line
05  {
06  private:
07      int x1,y1,x2,y2;
08  public:
09      Line();                                //默认构造函数
10      Line(int a=0,int b=0,int c=0,int d=0); //带参数的构造函数
11      void printPoint();                     //打印两点坐标
12      double getDis();                       //计算两点距离
13  };
14  inline Line::Line(int a,int b,int c,int d) //初始化点
15  {
16      x1=a;
17      y1=b;
18      x2=c;
19      y2=d;
20  }
21  inline void Line::printPoint()
22  {
23      cout<<"A: ("<<x1<<","<<y1<<") "<<endl;
24      cout<<"B: ("<<x2<<","<<y2<<") "<<endl;
25  }
26  inline double Line::getDis()
27  {
28      double dis;
29      dis=sqrt(double(((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))));
30      return dis;
31  }
32  void main()
33  {
34      Line line(10,68,-5,6);
35      line.printPoint();                     //输出两点坐标
36      cout<<"两点距离: "<<line.getDis()<<endl; //输出两点距离
37  }

```

【代码解析】

第 07 行为类 Line 的成员变量，第 09、10 为其两个构造函数，第 11、12 行为其成员函数。第 14~20 行为带参数的构造函数定义体，第 21~31 行分别为成员函数的定义体，由关键字 inline 判断其为内联函数。

在 main() 中，先定义类对象 line（如第 34 行），接下来调用 printPoint() 和 getDis()（如第 35、36 行）。



注意：内联函数的行数不能太多，1~5 行为宜，否则会有反效果。



实例 126 this 指针的应用

【实例描述】

每个类对象在实例化时都有一个 `this` 指针指向其数据的首地址。当类的非静态成员函数在访问非静态成员变量时，若遇到形参变量和成员变量相同，可以使用 `this` 指针指向成员变量，以示区别，效果如图 7-19 所示。

【实现过程】

定义类 A，其有成员变量 `num`，成员函数 `output()`，以输出 `num` 的值。利用构造函数初始化成员变量，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 class A
05 {
06 private:
07     int num;                //成员变量
08 public:
09     A(int num)              //构造函数
10     {
11         //num=num;
12         this->num=num;      //初始化
13     }
14     void output()
15     {
16         cout<<"私有成员变量 num="<<num<<endl;
17     }
18 };
19 void main()
20 {
21     A a(10);                //定义类对象
22     a.output();             //输出
23 }
```

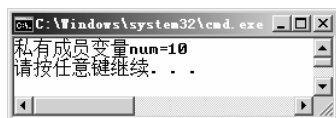


图 7-19 this 指针应用

【代码解析】

第 04~18 行为类 A 的定义，第 07 行为私有变量 `num`，其初始化由第 09~13 行的带参构造函数实现。如果将第 12 行用第 11 行代替，程序结果将如图 7-20 所示。

可见，第 11 行并没有类成员变量的初始化。第 14~17 行为成员函数 `output()`，在 `main()` 中，先定义类对象，再输出成员变量值（如第 21、22 行）。

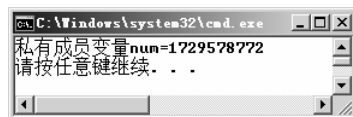


图 7-20 相同变量赋值不用 this 指针



注意：当类中的非静态函数对与成员变量同名的变量进行操作时，切记用 `this` 指针标明该类的成员变量。



实例 127 复制构造函数的应用（复制矩阵）

【实例描述】

本实例实现使用复制构造函数从一个类对象复制另一个对象，可从屏幕输入矩阵元素，调用拷贝构造函数后，创建另一个对象，并初始化其成员变量。效果如图 7-21 所示。

【实现过程】

本实例定义类 CMatrix，其有成员变量 m_nRow、m_nCol 和 **m_pData，分别表示矩阵行、列和矩阵内存。成员函数 output() 输出矩阵元素，Row() 和 Col() 获取矩阵的行列。本实例将类 CMatrix 的声明和定义分开，首先列出 matrix.h 文件的代码，如下：

```
01 #include <iostream>
02 using namespace std;
03
04 class CMatrix
05 {
06 public:
07     CMatrix();                //默认
08     CMatrix(int nRow, int nCol, int* pAry);    //带参数的
09     CMatrix(CMatrix & src);    //复制构造函数
10     ~CMatrix();              //析构函数
11     void output();           //输出函数
12     int Row() const { return m_nRow; }         //获取行
13     int Col() const { return m_nCol; }         //获取列
14 private:
15     int m_nRow;              //行数
16     int m_nCol;              //列数
17     int **m_pData;           //矩阵
18 };
```

源文件 matrix.cpp 的代码如下：

```
01 #include "matrix.h"
02 #include <math.h>
03 #include <assert.h>
04
05 CMatrix::CMatrix()            //构造函数
06 {
07     m_nRow = 0;
08     m_nCol = 0;
09     m_pData = NULL;           //空值
10 }
11 CMatrix::CMatrix(int nRow, int nCol, int* pAry)    //带参构造函数
12 {
13     m_nRow = nRow;
14     m_nCol = nCol;
15     m_pData = new int*[m_nRow];    //二维内存初始化
16     for (int i = 0; i < m_nRow; i++)
17     {
```

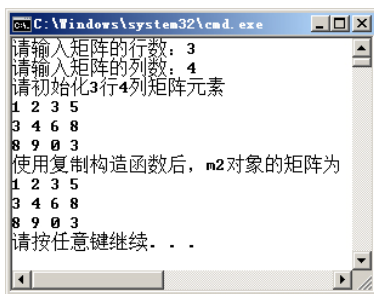


图 7-21 复制构造函数应用



```
18         m_pData[i] = new int[nCol];
19         for (int j = 0; j < m_nCol; j++)
20         {
21             m_pData[i][j] = pAry[i*m_nCol+j];    //赋值
22         }
23     }
24 }
25 CMatrix::CMatrix(CMatrix & src)                //复制构造函数
26 {
27     m_nRow = src.Row();                        //获取行
28     m_nCol = src.Col();                        //获取列
29     m_pData = new int*[src.Row()];
30     for(int i = 0; i < src.Row(); i++)          //行
31     {
32         m_pData[i] = new int[src.Col()];
33         for(int j = 0; j < src.Col(); j++)      //列
34         {
35             m_pData[i][j] = src.m_pData[i][j]; //赋值
36         }
37     }
38 }
39 void CMatrix::output()
40 {
41     for(int i=0;i<m_nRow;i++)//行
42     {
43         for(int j=0;j<m_nCol;j++)//列
44             cout<<m_pData[i][j]<<" ";          //输个元素
45         cout<<endl;
46     }
47 }
48 CMatrix::~CMatrix()                            //析构函数
49 {
50     if (m_pData!=NULL)                        //如果不为空，释放
51     {
52         for (int i = 0; i < m_nRow; i++)        //释放每行
53         {
54             delete[] m_pData[i];
55         }
56         delete[] m_pData;
57         m_pData = NULL;
58     }
59     m_nRow = 0;                                //行列值回 0
60     m_nCol = 0;
61 }
```

执行函数 main()所在文件的代码如下：

```
01 #include "matrix.h"
02
03 void main()
04 {
05     int n,m;                                    //行列
06     int *data;                                  //矩阵内存
07     cout<<"请输入矩阵的行数: ";
08     cin>>n;                                    //行
09     cout<<"请输入矩阵的列数: ";
10     cin>>m;                                    //列
11     data = new int[n*m];                       //申请内存
12     cout<<"请初始化"<<n<<"行"<<m<<"列矩阵元素"<<endl;
13     for (int i=0; i<n; i++)
```



```

14      {
15          for (int j=0; j<m; j++)
16              cin>>data[i*m+j];          //初始化
17      }
18      CMatrix m1(n,m,data);              //声明类对象 1
19      CMatrix m2(m1);                    //声明类对象 2
20      cout<<"使用复制构造函数后, m2 对象的矩阵为"<<endl;
21      m2.output();                       //输出
22  }

```

【代码解析】

(1) 第 07~09 行为构造函数, 其中第 09 行为复制构造函数。第 11~13 行为成员函数, 其中第 12、13 行为 const 函数, 它的意义在于不改变成员变量的值。第 15~17 行为私有成员变量。

(2) 第 05~10 行为默认构造函数, 第 11~24 行为带参数的构造函数, 它的意义是初始化成员变量。第 25~38 行为复制构造函数, 它的应用意义是复制一个类对象到另一个。第 39~47 行的 output() 函数输出成员变量 m_pData 的各元素值, 第 48~61 行的析构函数用于释放内存、删除指针, 同时重置成员变量 m_nRow 和 m_ncol 的值。

第 05 行定义行列变量 n 和 m, 第 06 行定义一维内存 data, 它的大小为 n*m, 为矩阵元素。第 08、10 行获取行列的值, 第 11~17 行申请一维内存 data, 并对其初始化。第 18、19 行定义类 CMatrix 的两个对象, 先对 m1 调用带参数的构造函数创建, 而对象 m2 利用复制构造函数创建。最后由第 21 行输出 m2 的矩阵元素值。



注意: 变量被声明为 const 型, 可以构造常量。函数被声明为 const 型, 其代码不能对类成员变量的值做改变。



实例 128 走出迷宫 (类+算法)

【实例描述】

本实例模拟走迷宫原理, 实现该算法。已知迷宫布局, 其中, 1 表示不可通行, 0 为可通行。由用户输入出入口位置, 经过算法处理, 输出路径。效果如图 7-22 所示。

【实现过程】

定义类 Maze, 处理走出迷宫算法。首先宏定义迷宫的行列组成, 分别为 M 和 N 表示。类 Maze 的成员变量为 maze[M][N], 表示迷宫, 被初始化于构造函数。变量 start_row、start_col、end_row 和 end_col 表示出入口的行列数, succeed 表示是否找到出口。成员函数有 PrintMaze()、GetPos()、SearchMaze() 和 EvaluateMaze(), 用于打印初始化迷宫、获取出入口位置、搜索走出迷宫路径。首先列出类 Maze 的声明代码, 如下:

```

01  #include <iostream>
02  using namespace std;
03
04  #define M 10

```

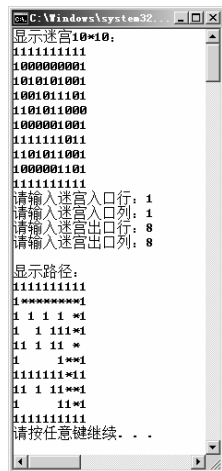


图 7-22 用类实现走出迷宫



```
05 #define N 10
06 class Maze
07 {
08 public:
09     Maze(); //构造函数
10     ~Maze(){} //析构函数
11     void PrintMaze(); //打印迷宫
12     void GetPos(); //获取出入口位置
13     void SearchMaze(); //获取迷宫路径
14     bool EvaluateMaze(int i,int j); //判断当前点是否可行
15 private:
16     int maze[M][N]; //迷宫
17     int start_row; //入口行
18     int start_col; //入口列
19     int end_row; //出口行
20     int end_col; //出口列
21     bool succeed; //是否找到出口
22 };
```

接下来，列出类成员函数的定义，代码如下：

```
01 Maze::Maze()
02 {
03     int copy[M][N]={1,1,1,1,1,1,1,1,1,1},
04     {1,0,0,0,0,0,0,0,0,1},
05     {1,0,1,0,1,0,1,0,0,1},
06     {1,0,0,1,0,1,1,1,0,1},
07     {1,1,0,1,0,1,1,0,0,0},
08     {1,0,0,0,0,0,1,0,0,1},
09     {1,1,1,1,1,1,1,0,1,1},
10     {1,1,0,1,0,1,1,0,0,1},
11     {1,0,0,0,0,0,1,1,0,1},
12     {1,1,1,1,1,1,1,1,1,1}};
13     memcpy(maze,copy,sizeof(maze)); //迷宫初始化，0 为可行，1 为不可行
14     succeed=false; //初始化为没有
15 }
16 void Maze::PrintMaze()
17 {
18     cout<<"显示迷宫 10*10: "<<endl;
19     for(int i=0;i<M;i++) //行
20     {
21         for(int j=0;j<N;j++) //列
22         {
23             if(maze[i][j]==1) //不可行
24                 cout<<"1";
25             else //可行
26                 cout<<"0";
27         }
28         cout<<endl; //下一行
29     }
30 }
31 void Maze::GetPos()
32 {
33     cout<<"请输入迷宫入口行: ";
34     cin>>start_row;
35     cout<<"请输入迷宫入口列: ";
36     cin>>start_col;
37     cout<<"请输入迷宫出口行: ";
38     cin>>end_row;
39     cout<<"请输入迷宫出口列: ";
```



```

40     cin>>end_col;
41 }
42 void Maze::SearchMaze()
43 {
44     if(EvaluateMaze(start_row,start_col)==false)    //没有找到出口
45         cout<<endl<<"没有找到出口\n";
46     else
47     {
48         cout<<endl<<"显示路径: "<<endl;
49         for(int i=0;i<M;i++)    //行
50         {
51             for(int j=0;j<N;j++)    //列
52             {
53                 if(maze[i][j]==1)    //打印不可通行位置
54                     cout<<"1";
55                 else if(maze[i][j]==2)    //打印路径
56                     cout<<"*";
57                 else
58                     cout<<" ";
59             }
60             cout<<endl;
61         }
62     }
63 }
64 bool Maze::EvaluateMaze(int i,int j)
65 {
66     maze[i][j]=2;    //与迷宫已被判断的元素区别
67     if(i==end_row&&j==end_col)    //搜索到出口
68         succeed=true;
69     if(succeed!=true&&maze[i][j+1]==0)    //向后一列可通行的
70         EvaluateMaze(i,j+1);
71     if(succeed!=true&&maze[i+1][j]==0)    //向下一行可通行的
72         EvaluateMaze(i+1,j);
73     if(succeed!=true&&maze[i][j-1]==0)    //向前一列可通行的
74         EvaluateMaze(i,j-1);
75     if(succeed!=true&&maze[i-1][j]==0)    //向前一行可通行的
76         EvaluateMaze(i-1,j);
77
78     if(succeed!=true)    //重新赋为可通行的
79         maze[i][j]=0;
80     return succeed;
81 }

```

【代码解析】

在第一段代码中,第04、05行为迷宫行列数的宏定义,第11~14行为其成员函数,第16~21行为成员变量定义。

在第二段代码中,第01~15行为类构造函数,初始化迷宫。第16~30行打印迷宫,第31~41行获取出入口的行列值。第42~63行的SearchMaze()函数判断是否搜索到出口,并调用函数EvaluateMaze()。第64~81行为函数EvaluateMaze()的定义体,它为递归函数,调用自身,如第70、72、74、76行所示。



注意:构造函数在对迷宫进行初始化时,并不是直接赋值,而是利用函数memcpy()实现,因为在类中初始化数组并不能像在普通场合进行整体赋值,必须单独赋值。

第 2 篇 C++进阶案例

第 8 章 泛型编程技术

泛型编程技术代表高效、可重复利用。它的首次提出，即要实现一种语言机制，可以交互性地使用组件，比如容器及算法。本章从 STL 中涉及的容器、迭代器及算法思想讲述泛型编程技术的应用。



实例 129 绕过形参限制（最简单的模板例程）

【实例描述】

本实例简单介绍什么是模板，它可以不理睬形参的类型。不同的数据类型都可以调用该函数，本实例通过函数模板输出不同类型的变量值，效果如图 8-1 所示。



图 8-1 输出不同类型的变量值

【实现过程】

定义函数模板的一般形式如下：

```
template <class T> 返回类型 函数名（参数列表）；
```

本实例输出 1 和 3.4，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 template<class T> void output(T a)           //模板输出函数
05 {
06     cout<<a<<endl;
07 }
08 void main()
09 {
10     output(1);                               //输出整数变量
11     output(3.4);                             //输出浮点变量
12 }
```

【代码解析】

第 04~07 行定义函数模板 output()，它的输入参数类型为 T，返回值类型为 void。第 10、11 行输出 1 和 3.4。



注意：函数模板的参数类型至少有一个是模板类型 T，返回值类型可以是 void 及其他。



实例 130 万能计算器（支持各类数据的加法函数）

【实例描述】

本实例实现万能加法计算器，函数模板可以适用于不同类型的数据进行加法运算，再也不用重载函数。本实例模拟整型及浮点型变量的加法运算，效果如图 8-2 所示。

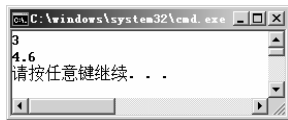


图 8-2 各类数据的加法运算

【实现过程】

定义模板加法函数 `add()`，其有两个参数，返回参数之和。输出 1 加 2、1.2 加 3.4 的结果，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 template<class T> T add(T a, T b)          //模板加法函数
05 {
06     return a+b;
07 }
08 void main()
09 {
10     cout<<add(1,2)<<endl;                  //整型加法
11     cout<<add(1.2,3.4)<<endl;              //浮点加法
12 }
```

【代码解析】

第 04~07 行为模板加法函数 `add()`，它的参数及返回值类型都是 `T`。第 10、11 行针对不同的数据类型调用函数 `add()`。



注意：由于 `add()` 函数的两个参数类型相同，所以不能实现不同类型数据的相加。



实例 131 输出浮点型数据和整型数据

（隐式和显式实例化）

【实例描述】

函数模板实现定义一系列功能相同的函数，也即重载。为了对输入的特定参数类型进行操作，首先要对函数模板进行实例化。一般情况下，函数模板的实例化都是隐式的。但当程序不能根据被传入的实参类型对模板函数进行实例化时，需要显式实例化模板函数。本实例使用隐式实例化及显式实例化输出浮点型和整型数据，效果如图 8-3 所示。

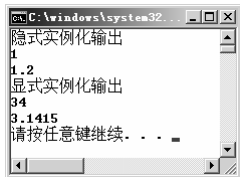


图 8-3 隐式和显式实例化



【实现过程】

定义函数模板 `output()`，然后在 `main()` 中分隐式实例化和显式实例化输出整型和浮点型数据，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 template<class T> void output(T a)      //函数模板
05 {cout<<a<<endl;}
06
07 void main()
08 {
09     //隐式实例化
10     cout<<"隐式实例化输出"<<endl;
11     output(1);                          //整型
12     output(1.2);                        //浮点型
13     cout<<"显式实例化输出"<<endl;
14     //显式实例化
15     output<int>(34);                    //整型
16     output<double>(3.1415);             //浮点型
17 }
```

【代码解析】

第 04、05 行为函数模板 `output()`，第 11、12 行以隐式实例化方式调用 `output()`，第 15、16 行以显式实例化方式调用 `output()`。



实例 132 使用模板特化判断结构体的最大值

【实例描述】

模板的使用一方面实现了代码的重用，减少了代码量。另一方面，使参数类型可变化，比如，利用函数模板实现最大数的计算。但是结构体中包含多个参数，使用函数模板并不能被实现，此时可考虑模板特化。效果如图 8-4 所示。

【实现过程】

首先定义结构体类型 `data`，其内包含两个参数：`int a` 和 `char ch`。定义函数模板 `T mymax()` 获取两个的最大值，对结构体变量的最大值计算使用特例化。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 struct data
05 {
06     int a;                          //整型
07     char ch;                         //字符型
08 };
09 template<class T> T mymax(const T t1,const T t2)
10 {
11     return t1<t2?t2:t1;              //返回最大值
12 }
```



图 8-4 模板特化



```

13  const struct data mymax(const struct data t1,const struct data t2)
14  {
15      if(t1.a>t2.a)                //第1个的a大
16          return t1;              //返回第1个
17      else if(t1.a==t2.a)          //a相等
18      {
19          if(t1.ch>=t2.ch)          //判断第2个, t1 不小于 t2
20              return t1;          //返回第1个
21          else                      //t1 小于 t2
22              return t2;          //返回第2个
23      }
24      else                          //第2个的a大
25          return t2;              //返回第2个
26  }
27  void main()
28  {
29      data a,b;                    //data 型变量
30      a.a=4;a.ch='c';              //初始化
31      b.a=59;b.ch='t';
32      cout<<mymax(2,15)<<" ";
33      cout<<mymax('b','x')<<" ";
34      cout<<mymax(true,false)<<" ";
35      cout<<"<<mymax(a,b).a<<" "<<mymax(a,b).ch<<" "<<endl;
36  }

```

【代码解析】

第 04~08 行为结构体 `data`，第 09~12 行为函数模板 `mymax()`，第 13~26 行为该模板的特化（针对结构体 `data`）。第 29~31 行定义两个 `data` 型变量 `a` 和 `b`，并初始化，最后由第 32~35 行输出 `int` 型、`char` 型、`bool` 型及 `data` 型的最大值。



注意：模板的特化是对所用参数类型进行显式操作，而偏特化是对部分参数进行显式操作。



实例 133 模板函数的重载例程

【实例描述】

模板函数是一种函数，当然也可以被重载。本实例即要实现模板函数的重载，实例 130 是计算两个参数的加法，本实例计算 3 个参数的加法，运行效果如图 8-5 所示。

【实现过程】

接着实例 130，重载加法模板函数 `add()`，它的参数为 3 个。调用这两个函数，分别输出 1 加 2 的值、1.2 加 3.4，再加 4.5 的值，代码如下：

```

01  #include <iostream>
02  using namespace std;
03
04  template<class T> T add(T a, T b)          //模板加法函数
05  {
06      return a+b;

```



图 8-5 模板函数的重载



```

07  }
08  template<class T> T add(T a, T b, T c)    //重载模板加法函数
09  {
10      return a+b+c;
11  }
12  void main()
13  {
14      cout<<add(1,2)<<endl;                //整型加法
15      cout<<add(1.2,3.4,4.5)<<endl;        //浮点加法
16  }

```

【代码解析】

第 04~07 行是原 add()函数定义体,第 08~11 行是重载模板 add()函数,它的不同之处在于有 3 个参数。第 14、15 行分别调用两个函数并输出结果。



注意：模板函数的重载注意事项与一般函数相同。



实例 134 补充代码使输出结果成立

【实例描述】

继续实例 133,本实例补充代码输出结构体变量的加法,运行效果如图 8-6 所示。

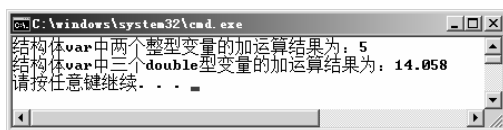


图 8-6 调用重载模板函数并输出

【实现过程】

定义结构体 var,包含 5 个变量,即 2 个整型、3 个 double 型。在 main()中分别初始化结构体变量 _var 的各个变量值,并输出 2 个整型及 3 个 double 型变量的加法结果。代码如下:

```

01  #include <iostream>
02  using namespace std;
03
04  struct var                                //结构体 var
05  {
06      int var_int1;
07      int var_int2;
08      double var_doul;
09      double var_dou2;
10      double var_dou3;
11  };
12  ...
13  void main()
14  {
15      var _var;                              //结构体变量
16      _var.var_int1=3;
17      _var.var_int2=2;
18      _var.var_doul=1.345;
19      _var.var_dou2=3.145;

```



```

20     _var.var_dou3=9.568;
21     cout<<"结构体 var 中两个整型变量的加运算结果为:
22     "<<add(_var.var_int1,_var.var_int2)<<endl;           //整型加法
23     cout<<"结构体 var 中三个 double 型变量的加运算结果为:
24     "<<add(_var.var_dou1,_var.var_dou2,_var.var_dou3)<<endl; //浮点加法
25 }

```

【代码解析】

第 04~11 行定义结构体 var，第 15 行定义结构体变量 _var，第 16~20 行初始化 _var 中各个变量的值，第 21~24 行调用不同的函数输出结果。



实例 135 求 $N \times N$ 的值

【实例描述】

若计算 $N \times N$ 时并不知道 N 的类型，此时可以使用模板函数实现，即可以计算不同类型变量的 $N \times N$ 结果，运行效果如图 8-7 所示。

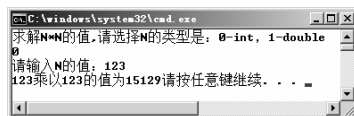


图 8-7 求 $N \times N$ 的值

【实现过程】

定义模板函数 multiply，参数只有一个为 N ，返回 $N \times N$ 的值。判断用户是执行整型还是 double 型的相乘，并调用 multiply() 函数，代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 template <class T> T multiply(T N)           //相乘函数
05 {
06     return N*N;                             //返回结果
07 }
08
09 void main()
10 {
11     int n_int=0;                             //整型
12     double n_double=.0;                     //double 型
13     cout<<"求解 N*N 的值, 请选择 N 的类型是: 0-int, 1-double"<<endl;
14     int i;                                  //选择
15     cin>>i;                                //输入选择
16     cout<<"请输入 N 的值: ";
17     if(i==0)                                //int
18     {
19         cin>>n_int;
20         cout<<n_int<<"乘以"<<n_int<<"的值为"<<multiply(n_int);
21     }
22     else if(i==1)                            //double
23     {
24         cin>>n_double;
25         cout<<n_double<<"乘以"<<n_double<<"的值为"<<multiply(n_double);
26     }
27 }

```

【代码解析】

第 04~07 行是相乘函数的定义体，第 11、12 行是不同类型变量的定义。第 14、15 定义选



择变量类型，并接收选择类型。第 17~26 行针对不同类型的变量调用模板函数，并输出结果。



注意：模板函数使代码更为简洁，易于维护。



实例 136 判断参数为字符串类型就输出字符串

【实例描述】

由于模板函数的参数类型可以是任意的，本实例判断如果参数类型为字符串，就输出该字符串，否则不做处理，效果如图 8-8 所示。

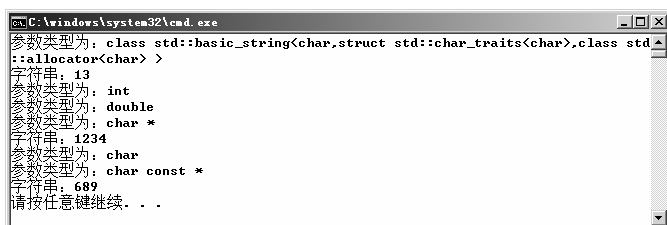


图 8-8 判断参数类型是否为字符串类型

【实现过程】

定义模板函数 `output()`，使用 `typeid()` 获取参数的类型。由于该函数的返回值为 `char*` 型，所以使用 C 风格字符串的函数 `strstr` 查找是否有子串 `char`。如果有，再判断是否为字符。如果两个条件都合格，输出字符串，代码如下：

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 template <class T> void output(T a)
06 {
07     cout<<"参数类型为: "<<typeid(a).name()<<endl;
08     if(strstr(typeid(a).name(),"char")!=NULL) //有子串 char
09     {
10         if(strlen(typeid(a).name())==4) //字符
11         {}
12         else //字符串
13             cout<<"字符串: "<<a<<endl;
14     }
15 }
16 void main()
17 {
18     string str="13"; //字符串
19     output(str); //输出
20     int var1=1; //整型
21     output(var1); //输出
22     double var2=2.4545; //浮点型
23     output(var2); //输出
24     char ch[]="1234"; //C 风格字符串
25     output(ch);
```



```

26         output('d');           //字符
27         output("689");         //字符串常量
28     }

```

【代码解析】

第 05~15 行为模板函数 `output()`，参数为 `T a`，返回值为 `void` 型。首先输出参数的类型，如第 07 行，第 08 行判断该类型字符串中是否包含 `char` 子串，如果包含，再判断是否为字符型，如第 10 行。如果不是字符型，输出字符串。在 `main()` 中，第 19、21、23 行，以及第 25~27 行分别对 `string` 型字符、整型、浮点型、C 风格字符串、字符及字符串常量进行判断输出。



注意：本实例在第 11 行先判断类型中是否包含 `char` 子串，再在第 10 行排除字符型变量的干扰。

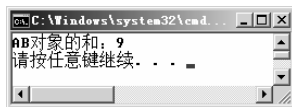


实例 137 求 AB 对象的和（类参数）

【实例描述】

之前列的函数模板只有一个参数类型 `T`，本实例通过定义多个参数类型实现求不同类参数的和，运行效果如图 8-9 所示。

【实现过程】



定义类 `A` 和类 `B`，都有私有成员变量 `a`，都利用成员函数 `getVar()` 返回成员变量 `a` 的值。定义模板函数计算不同类对象的成员变量之和。代码如下：

```

01 #include <iostream>
02 using namespace std;
03 class A           //类 A
04 {
05 private:
06     int a;         //成员变量
07 public:
08     A(int x){a=x;} //构造函数
09     int getVar() const{return a;} //返回私有变量值
10 };
11 class B           //类 B
12 {
13 private:
14     int a;         //成员变量
15 public:
16     B(int x){a=x;} //构造函数
17     int getVar() const{return a;} //返回私有变量值
18 };
19 template<class T1,class T2> void add(T1 t1,T2 t2) //函数模板
20 {
21     cout<<"AB 对象的和: "<<t1+t2<<endl;
22 }
23 void main()
24 {
25     A _a(5);       //类 A 对象
26     B _b(4);       //类 B 对象

```



```

27         add(_a.getVar(),_b.getVar());           //AB 对象和
28     }

```

【代码解析】

第 03~10 行、第 11~18 行分别为类 A 和类 B 的定义体。第 19~22 行的模板函数定义两种参数类型 T1 和 T2，并输出和值（如第 21 行）。第 25、26 行定义类 A 和类 B 对象，由第 27 行输出两者之和。



实例 138 输出内存区域各类型数据（void*）

【实例描述】

void* 也是泛型编程的一种，它可以接受任意类型的指针。本实例实现利用 void* 输出内存区域各类型数据。效果如图 8-10 所示。



图 8-10 输出内存区域各类型数据

【实现过程】

定义函数 output()，第 1 个参数为内存地址，类型为 void*；第 2 个参数为强制转换时使用，输出 int、char、double、string 型数据，代码如下：

```

01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 void output(void* p,char ch)
06 {
07     switch(ch)
08     {
09         case 'a':
10             {
11                 int *ap=(int*)p;           //强制转换
12                 cout<<*ap<<" ";
13             }
14             break;
15         case 'b':
16             {
17                 char *chp=(char*)p;       //强制转换
18                 cout<<*chp<<" ";
19             }
20             break;
21         case 'c':
22             {
23                 double *dp=(double*)p;   //强制转换
24                 cout<<*dp<<" ";
25             }
26             break;
27         case 'd':
28             {
29                 string *strp=(string*)p; //强制转换
30                 cout<<*strp<<" ";
31             }
32             break;
33     }
34 }

```



```

35 void main()
36 {
37     int a=4;                //int
38     char ch='a';            //char
39     double d=0.2145;        //double
40     string str="Hello";     //string
41     output(&a, 'a');         //输出整型
42     output(&ch, 'b');        //char 型
43     output(&d, 'c');         //double 型
44     output(&str, 'd');       //字符串
45     cout<<endl;
46 }

```

【代码解析】

第 05~34 行为函数 `output()` 定义体, 依据第 2 个参数判断强制转换时的类型, 如第 11、17、23、29 行。第 37~40 行定义 4 种变量, 第 41~44 行输出内存数据。



注意: `switch` 结构在 `case` 中不能定义新变量, 除非用 `{}` 括起来。



实例 139 变幻的对象——使用 `template` 定义一个类模板

【实例描述】

使用 `template` 可以定义函数模板, 也可以定义类模板。它可以处理不同类型的对象, 其定义的最简单格式如下:

```

template <class T> class 类名
{
    //类模板内容
};

```

本实例定义类模板 `output`, 用于处理不同类型的数据, 并输出, 效果如图 8-11 所示。

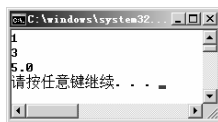


图 8-11 类模板

【实现过程】

定义类模板 `output`, 有成员变量 `_var`, 变量为 `T`。带参数构造函数用于获取变量值, 成员函数 `var()` 用于返回其成员变量值, 其代码如下:

```

01 #include <iostream>
02 using namespace std;
03
04 template <class T> class output          //模板类
05 {
06     private:
07         T _var;                          //变量
08     public:
09         output(const T& v):_var(v)        //构造函数
10         {}
11         T var()                          //获取成员变量值, 并返回
12         {return _var;}

```



```

13     };
14     void main()
15     {
16         output<double> x(1.0);           //double 型对象
17         output<int> j(3);                //int 型对象
18         output<char*> str("5.0");        //字符串对象
19         cout<<x.var()<<endl;
20         cout<<j.var()<<endl;
21         cout<<str.var()<<endl;
22     }

```

【代码解析】

第 04 行为定义类模板的简单格式，第 07 行为其私有成员变量 `_var`。第 09 行为带参数的构造函数，第 11、12 行为成员函数 `var()`，返回其变量值，作用为在第 19~21 行输出其值。第 16~18 行定义 3 种类型的对象，分别为 `double`、`int` 和 `char*` 型。



注意：类模板和模板函数的使用可以节省很多重复的代码。



实例 140 分离类模板的声明和定义（求最大值）

【实例描述】

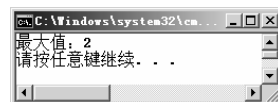
本实例演示如何将一个类模板的声明和定义分离。成员函数在体外的定义格式为：

```

template <class T>
返回值类型 类名<T>::成员函数名(参数列表)
{
    //类模板内容
};

```

程序运行效果如图 8-12 所示。



【实现过程】

图 8-12 分离类模板的声明和定义

简单定义类模板 `MAX`，成员变量 `x` 和 `y`，成员函数 `getmax()` 用于获取最大值，代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 template <class T>
05 class MAX                                //类模板
06 {
07     private:
08         T x,y;                            //变量
09     public:
10         MAX(T v1,T v2);                  //构造函数
11         T getmax(MAX &a);                //获取最大值
12 };
13 template <class T>
14 MAX<T>::MAX(T v1,T v2)                    //定义体
15 {
16     x=v1;
17     y=v2;
18 }

```



```

19  template <class T>
20  T MAX<T>::getmax(MAX &a)           //定义体
21  {
22      return a.x<a.y?a.y:a.x;
23  }
24  void main()
25  {
26      MAX<int> x(1,2);               //对象
27      cout<<"最大值: "<<x.getmax(x)<<"\n";
28  }

```

【代码解析】

第 04~12 行为类模板 MAX 的声明，第 13~18、第 19~23 行为类模板 MAX 的构造函数和成员函数 getmax() 的定义体。



注意：在以后的应用中，经常要分享类的声明和定义。当定义类模板时，请严格按照上述格式书写。



实例 141 类模板含有多个类型参数

【实例描述】

类似于函数模板，类模板也可以含有多个类型参数。本实例模拟类模板的两个成员变量分属不同的类型，并初始化后输出其值，效果如图 8-13 所示。



图 8-13 类模板含多个类型参数

【实现过程】

定义类模板 multitype，包含两个类型参数 T1 和 T2。构造函数对其成员变量进行初始化，成员函数 show() 输出其值。其代码如下：

```

01  #include <iostream>
02  using namespace std;
03
04  template <class T1,class T2>           //类模板有两个类型参数
05  class multitype
06  {
07  private:                               //私有变量
08      T1 x;
09      T2 y;
10  public:
11      multitype(T1 t1,T2 t2);           //构造函数
12      void show();                       //显示变量值
13  };
14  template <class T1,class T2>
15  multitype<T1,T2>::multitype(T1 t1,T2 t2)
16  {
17      x=t1;                             //赋值
18      y=t2;
19  }
20  template <class T1,class T2>
21  void multitype<T1,T2>::show()
22  {

```



```

23         cout<<x<<" "<<y<<"\n";
24     }
25     void main()
26     {
27         multitype<int,double> m(1,2.1);           //定义对象
28         m.show();                                 //显示
29     }

```

【代码解析】

第 04~13 行声明类模板 `multitype`, 第 08、09 行为其私有变量, 第 12 行为其成员函数 `show()`。第 14~19 行是带参构造函数的定义, 第 20~24 行是成员函数 `show()` 的定义。第 27 行定义类模板对象 `m`, 并初始化其成员变量值分别为 1 和 2.1。第 28 行输出两个变量的值。



注意: 在类体外定义成员函数时, 它的格式如第 14、15 行或第 20~24 行所示。



实例 142 类模板的静态成员变量

【实例描述】

通过类模板, 在操作静态成员变量时与普通类一样, 本实例演示如何在不同的 .cpp 文件中调用类模板的静态成员变量。在缩写大型程序时, 不可能只在一个源文件下实现。类模板的声明、定义和调用分列于不同的文件。针对静态成员的共享机制, 本实例运行效果如图 8-14 所示。

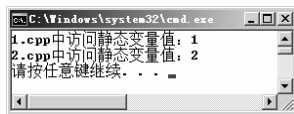


图 8-14 应用类模板的静态成员变量

【实现过程】

定义头文件 `1.h`, 以及源文件 `1.cpp`、`2.cpp` 和 `142.cpp`。在头文件 `1.h` 中声明类模板 `A`, 其有私有非静态变量 `x` 和公共静态成员变量 `a`。代码如下:

```

01 // #pragma once
02 #include <iostream>
03 using namespace std;
04
05 template<class T>
06 class A
07 {
08     private:
09         int x;                               //非静态成员变量
10     public:
11         static int a;                         //静态成员变量
12         A(int xx);                           //构造函数
13 };

```

源文件 `1.cpp` 中包含头文件 `1.h`, 其有类模板 `A` 的定义, 及访问函数 `show1()` 的定义, 代码如下:

```

01 #include "1.h"
02
03 template<class T>int A<T>::a=0;             //静态成员变量初始化
04 template <class T>
05 A<T>::A(int xx)

```



```
06 {  
07     x=xx; //非静态成员变量初始化  
08     a++;  
09 }  
10 void show1()  
11 {  
12     A<int> c(23);  
13     cout<<"1.cpp 中访问静态变量值: "<<c.a<<endl;  
14 }
```

源文件 2.cpp 中也包含头文件 1.h, 只有访问函数 show2() 的定义, 代码如下:

```
01 #include "1.h"  
02  
03 void show2()  
04 {  
05     A<int> c(33);  
06     cout<<"2.cpp 中访问静态变量值: "<<c.a<<endl;  
07 }
```

最后, 源文件 142.cpp 也包含头文件 1.h, 在此文件中 main() 函数调用 show1() 和 show2(), 所以需要列出函数的声明, 代码如下:

```
01 void show1(); //函数声明  
02 void show2();  
03  
04 void main()  
05 {  
06     show1(); //调用  
07     show2();  
08 }
```

【代码解析】

(1) 1.h

第 05、06 行为类模板的书写格式, 第 09 行为私有成员变量 x, 第 11 行为公共静态成员变量, 第 12 行为构造函数。

(2) 1.cpp

第 03 行为静态成员变量 a 的初始化, 第 04~09 行为构造函数对变量 x 的初始化及对变量 a 的加 1 计算。第 10~14 行为外部函数 show1() 访问静态变量 a。

(3) 2.cpp

第 05 行定义类模板 A 对象 c, 第 06 行访问其静态成员变量 a。

(4) 142.cpp

第 01、02 行列出两个函数的声明, 第 06、07 行调用。



注意: 在 1.h 头文件中千万不能有类模板的任何定义及初始化, 即使是在第 01 行列出 #pragma once, 也会出现重定义的错误。



实例 143 应用类模板的静态函数

【实例描述】

普通函数有静态函数, 用关键字 static 标识。类模板也可以定义静态函数, 它只能访问其



静态变量。本实例应用静态函数效果如图 8-15 所示。

【实现过程】

在实例 142 的基础上增加静态成员函数 fun(), 它的作用是在初始化静态变量后, 输出其值。文件包含 1.h、1.cpp、2.cpp 和 143.cpp。代码分别如下:

1.h:

```
01 ...
02 template<class T>class A
03 {
04 ...
05 public:
06 ...
07     static void fun();           //静态成员函数
08 };
```

1.cpp:

```
01 ...
02 template<class T>void A<T>::fun()
03 {
04     // x++;                      //不可对非静态变量进行操作
05     a=a+2;                       //错误, 不能访问非静态成员
06     //cout<<x<<endl;
07     cout<<a<<endl;
08 }
09 void show1()
10 {
11 ...
12     c.fun();                     //调用静态函数
13 }
```

2.cpp: (143.cpp 与文件 142.cpp 的内容相同, 在此不再赘述。)

```
01 ...
02 void show2()
03 {
04 ...
05     c.fun();                     //调用静态函数
06 }
```

【代码解析】

(1) 1.h

第 07 行为静态函数 fun()。

(2) 1.cpp

第 02~08 行为静态成员函数 fun()定义体, 第 04、06 行为访问非静态成员变量, 都是错误形式, 其后有注释。第 05 行访问静态变量 a, 正确。第 12 行访问静态函数。

(3) 2.cpp

第 05 行函数 show2()访问静态函数 fun()。



图 8-15 应用类模板的静态函数



注意: 静态变量不仅可以被静态成员函数调用, 也可以被非静态函数访问。



实例 144 类模板的友元应用

【实例描述】

本实例演示如何实现类模板的友元函数，该友元函数实现运算符+=的重载，完成两个对象对于静态成员变量的相加运算，效果如图 8-16 所示。

【实现过程】

在类模板 A 中添加友元运算符重载函数，它的功能是实现两个对象静态变量的相加。本实例包含 3 个文件，即 1.h、1.cpp 和 144.cpp。

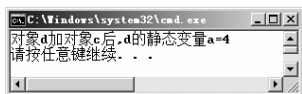


图 8-16 类模板的友元函数

头文件 1.h 的代码如下：

```
01  ...
02  class A
03  {
04  ...
05  public:
06  ...
07      friend A<T>&operator += (A<T>& a,A<T>& b)    //友元函数
08      {
09          a.a+=b.a;                                //变量相加
10          return a;                                //返回对象
11      }
12  };
```

源文件 1.cpp 新添代码如下：

```
01  ...
02  void show1()
03  {
04      A<int> c(23);                                //对象 1
05      A<int> d(34);                                //对象 2
06      d+=c;                                          //相加
07      cout<<"对象 d 加对象 c 后,d 的静态变量 a="<<d.a<<endl;
08  }
```

最后，源文件 144.cpp 代码如下：

```
01  void show1();                                    //函数声明
02
03  void main()
04  {
05      show1();                                       //函数调用
06  }
```

【代码解析】

(1) 1.h

第 07~11 行为友元函数的定义体，其中第 09 行实现两对象变量的+=运算，第 10 行返回对象 a。

(2) 1.cpp

第 02~08 行为函数 show1()的定义体，第 04、05 行为类模板 A 对象 c 和 d，第 06 行实现



运算符+=运算，第 07 行输出相加后对象 d 的静态变量 a 的值。

(3) 144.cpp

第 01 行声明函数 show1(), 第 05 行调用该函数。



注意：在类模板中定义友元函数，最好在类体内定义。如果将声明和定义分开，会出现错误。



实例 145 类模板的继承

【实例描述】

本实例演示类模板的继承，包括其书写格式、如何在派生类中初始化基类成员变量、如何用 new 方式创建一个子类对象以及访问子类和基类的成员函数。本例效果如图 8-17 所示。

【实现过程】

定义基类模板 A，派生类模板 B。基类有成员变量 X 和 Y、成员函数 show(), 子类有成员变量 W 和 H，成员函数 display()。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 template<class T1,class T2>
05 class A                                //基类 A
06 {
07 private:
08     T1 X;                              //成员变量 T1 型
09     T2 Y;                              //成员变量 T2 型
10 public:
11     A() {}                             //默认构造函数
12     A(T1 x,T2 y);                      //带参构造函数
13     void show();
14     ~A() {}                            //析构函数
15 };
16 template <class T1,class T2>
17 A<T1,T2>::A(T1 x,T2 y)                 //构造函数定义
18 {
19     X=x;
20     Y=y;
21 }
22 template<class T1,class T2>
23 void A<T1,T2>::show()                  //成员函数定义
24 {
25     cout<<"X="<<X<<"\n"<<"Y="<<Y<<endl;
26 }
27
28 template<class T1,class T2>
29 class B:public A<T1,T2>                //派生类 B
30 {
31 public:
32     B() {}                             //派生类默认构造函数
```

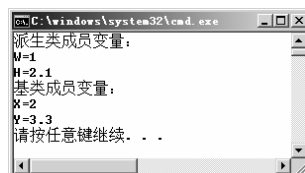


图 8-17 类模板的继承



```

33         B(T1 x,T2 y,T1 w,T2 l);           //带参构造函数
34         void display();
35         ~B(){}                             //派生类析构函数
36     private:                               //私有成员变量
37         T1 W;
38         T2 H;
39     };
40     template<class T1,class T2>
41     B<T1,T2>::B(T1 x,T2 y,T1 w,T2 h):A(x,y)
42     {
43         W=w;
44         H=h;
45     }
46     template<class T1,class T2>
47     void B<T1,T2>::display()
48     {
49         cout<<"W="<<W<<"\n"<<"H="<<H<<endl;
50     }
51     void main()
52     {
53         B<int,double> *q=new B<int,double>(2,3.3,1,2.1);    //子类指针
54         cout<<"派生类成员变量: "<<endl;
55         q->display();                                         //子类函数
56         cout<<"基类成员变量: "<<endl;
57         q->show();                                           //基类函数
58         delete q;                                           //释放指针
59     }

```

【代码解析】

第 04~15 行为基类 A 的声明，其中第 08、09 行为成员变量 X 和 Y，第 11、12 行为基类 A 的默认和带参构造函数，第 13、14 行为其成员函数 show() 和析构函数，第 16~21 行为带参构造函数的定义体，第 22~26 行为成员函数 show() 的定义体。

第 28~39 行为派生类 B 的声明，其中，第 32、33 行为默认和带参构造函数，第 34、35 行为成员函数的析构函数，第 37、38 行为子类 B 的成员变量。总结类模板的继承书写格式如下：

```

template<参数类型列表>
class 子类名: public 基类名<基类参数类型列表>
{
};

```

第 40~45 行为子类的带参构造函数定义，第 46~50 行为子类的成员函数 display() 定义体。第 53 行用 new 方式创建子类 B 指针，第 55、57 行分别调用子类和基类的成员函数 display() 和 show()。



注意：在子类中给基类成员变量初始化时请参看第 41 行。



实例 146 使用 STL 库创建容器

【实例描述】

STL 是标准模板库的英文缩写，内容包含容器、迭代器和算法。STL 可被称为 C++ 语言编程天才之作，容器是其中一种模板类。使用容器可以更方便地处理各种类型的数据。本实例实



现如何使用 STL 创建容器，并对容器的各元素进行初始化。最后输出各个容器的大小，效果如图 8-18 所示。

【实现过程】

容器分为 3 类，分别为顺序性容器（vector、deque、list）、关联式容器（set、multiset、map、multimap）和容器适配器（stack、queue、priority_queue）。下列代码演示如何创建容器及初始化容器。



图 8-18 使用 STL 库创建容器

```
01 #include <iostream>
02 #include <vector>
03 #include <deque>
04 #include <list>
05 #include <set>
06 #include <map>
07 #include <string>
08 using namespace std;
09
10 void main()
11 {
12     vector<int> v(4,1);           //创建 vector 容器对象
13     deque<float> deq;             //创建 deque 容器对象
14     deq.push_back(1);             //加入一个元素到队列
15     deq.push_back(2);             //加入另一个元素
16     deq.pop_back();               //删除最后一个元素
17     deq.push_front(3);            //加到第一个元素位置
18     list<int> l;                  //创建 list 容器对象
19     l.push_back(1);
20     set<short> _set;              //创建 set 容器对象
21     _set.insert(1);
22     _set.insert(1);
23     _set.insert(2);
24     multiset<short> _mset;        //创建 multiset 容器对象
25     _mset.insert(1);
26     _mset.insert(1);
27     _mset.insert(2);
28     map<string,string> _map;      //创建 map 容器对象
29     _map.insert(pair<string,string>("1","1"));
30     _map.insert(pair<string,string>("1","2"));
31     multimap<string,string> _mmap; //创建
32     _mmap.insert(pair<string,string>("1","1"));
33     _mmap.insert(pair<string,string>("1","2"));
34     cout<<"此 vector 的大小: "<<v.size()<<endl;
35     cout<<"此 deque 的大小: "<<deq.size()<<endl;
36     cout<<"此 list 的大小: "<<l.size()<<endl;
37     cout<<"此 set 的大小: "<<_set.size()<<endl;
38     cout<<"此 multiset 的大小: "<<_mset.size()<<endl;
39     cout<<"此 map 的大小: "<<_map.size()<<endl;
40     cout<<"此 multimap 的大小: "<<_mmap.size()<<endl;
41 }
```

【代码解析】

第 12、13、18、20、24、28、31 行分别创建各类容器，容器中的数据类型由其尖括号内的内容决定，下面列出其创建格式：

vector<数据类型>变量名(大小,变量值);



```
deque<数据类型>变量名;
list<数据类型>变量名;
set<数据类型>变量名;
multiset<数据类型>变量名;
map<数据类型, 数据类型>变量名;
multimap<数据类型, 数据类型>变量名;
```

vector 的初始化可利用构造函数, deque、list 的初始化使用各自的 push_back() 函数。set、multiset、map 和 multimap 使用 insert() 函数。



注意: 容器适配器并不直接存储数据, 而是容器的容器, 在此先不介绍。



实例 147 打印容器元素的值

【实例描述】

继续实例 146, 本实例实现如何打印容器中元素的值, 对于每个容器的值, 其输出也有区别。打印效果如图 8-19 所示。

【实现过程】

对于 vector 和 deque, 使用模板类成员函数 at() 即可输出第 i 条元素。而其他容器必须由其类型的迭代器才可被访问, 代码如下:

```
01 #include <iostream>
02 #include <vector>
03 #include <deque>
04 #include <list>
05 #include <set>
06 #include <map>
07 #include <string>
08 using namespace std;
09
10 void main()
11 {
12     ...
13     list<int>::iterator lit;           //list 的迭代器, 用于读取元素
14     set<short>::iterator iset;        //创建 set 的迭代器
15     multiset<short>::iterator mul;
16     map<string, string>::iterator mit; //创建 map 的迭代器
17     multimap<string, string>::iterator mulm;
18     cout<<"vector 容器中各元素的值: ";
19     for(int i=0; i<v.size(); i++)
20         cout<<v.at(i)<<" ";
21     cout<<endl;
22     cout<<"deque 容器中各元素的值: ";
23     for(int i=0; i<deq.size(); i++)
24         cout<<deq.at(i)<<" ";
25     cout<<endl;
26     cout<<"list 容器中各元素的值: ";
27     for(lit=l.begin(); lit!=l.end(); lit++)
28         cout<<*lit<<" ";
29     cout<<endl;
30     cout<<"set 容器中各元素的值: ";
```

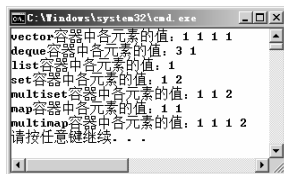


图 8-19 打印容器元素的值



```

31     for(iset=_set.begin();iset!=_set.end();iset++)
32         cout<<*iset<<" ";
33     cout<<endl;
34     cout<<"multiset 容器中各元素的值: ";
35     for(mul=_mset.begin();mul!=_mset.end();mul++)
36         cout<<*mul<<" ";
37     cout<<endl;
38     cout<<"map 容器中各元素的值: ";
39     for(mit=_map.begin();mit!=_map.end();mit++)
40         cout<<mit->first<<" "<<mit->second<<" ";
41     cout<<endl;
42     cout<<"multimap 容器中各元素的值: ";
43     for(mulm=_mmmap.begin();mulm!=_mmmap.end();mulm++)
44         cout<<mulm->first<<" "<<mulm->second<<" ";
45     cout<<endl;
46 }

```

【代码解析】

第 13~17 行定义各容器的迭代器，第 19~24 行利用函数 `at()` 访问 `vector` 和 `deque` 的各元素。第 26~45 行利用各容器的迭代器访问各元素，其中，对于 `multiset` 和 `multimap` 容器，因为一个位置处包含两个元素，它们的访问格式如第 40、44 行所示。



注意：容器 `vector` 和 `deque` 也可以利用其对应的迭代器访问。



实例 148 队列镜像

【实例描述】

本实例演示输出一个队列的镜像，它的实现原理很简单，即两个队列的元素呈镜面对称。实现过程是将原队列的元素反转赋给镜像队列，效果如图 8-20 所示。

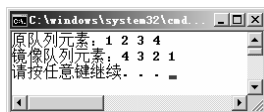


图 8-20 队列镜像

【实现过程】

创建两个队列 `qu` 和 `qu_copy`，再定义一个链表 `l` 和其对应的迭代器 `lit`。往 `qu` 中压入 4 个元素，首先输出 `qu` 的元素值。然后将 `qu` 的元素写进链表 `l` 中，再由链表 `l` 写入队列 `qu_copy` 中并输出。代码如下：

```

01 #include <iostream>
02 #include <queue>
03 #include <list>
04 using namespace std;
05
06 void main()
07 {
08     queue<int> qu;                //创建队列
09     queue<int> qu_copy;          //复件，用于存储镜像队列
10     list<int> l;                 //链表
11     list<int>::iterator lit;     //链表迭代器
12
13     qu.push(1);                  //原队列压入元素
14     qu.push(2);

```



```

15     qu.push(3);
16     qu.push(4);
17
18     cout<<"原队列元素: ";
19     while(qu.empty()!=true)           //如果不为空
20     {
21         l.push_front(qu.front());      //首元素读出
22         cout<<qu.front()<<" ";        //输出
23         qu.pop();                      //首元素出队列
24     }
25     cout<<endl;
26     for(lit=l.begin();lit!=l.end();lit++) //遍历链表元素
27     {
28         qu_copy.push(*lit);            //压入镜像队列
29     }
30     cout<<"镜像队列元素: ";
31     while(qu_copy.empty()!=true)       //如果不为空
32     {
33         cout<<qu_copy.front()<<" ";    //输出镜像队列元素
34         qu_copy.pop();
35     }
36     cout<<endl;
37 }

```

【代码解析】

第 08~11 行定义队列 `qu` 和 `qu_copy`，链表 `l` 和其类型的迭代器 `lit`。第 13~16 行压入元素到队列 `qu` 中，第 19~24 行的 `while` 循环边写 `qu` 的各元素到链表 `l` 中，边输出。第 26~29 行的 `for` 循环遍历链表各元素，并写入队列 `qu_copy` 中。第 31~35 行输出各元素。



注意：如果想读出队列中的某个元素值，需要用 `pop` 输出前几个元素，并且队列的读写规则是先进先出。



实例 149 获取队列头尾

【实例描述】

本实例模拟获取队列头尾，它使用队列类 `queue` 的成员函数 `front()` 和 `back()`。本实例在获取队列头尾后，输出其元素值，效果如图 8-21 所示。



图 8-21 获取队列头尾

【实现过程】

定义队列对象 `qu`，压入 4 个元素，分别为 1、2、3、4。输出队列头和尾元素的值，代码如下：

```

01 #include <iostream>
02 #include <queue>
03 using namespace std;
04
05 void main()
06 {
07     queue<int> qu;                //创建队列

```



```

08     qu.push(1);
09     qu.push(2);
10     qu.push(3);
11     qu.push(4);
12
13     cout<<"队列头元素: "<<qu.front()<<endl;           //队列头
14     cout<<"队列尾元素: "<<qu.back()<<endl;           //队列尾
15 }

```

【代码解析】

第 07~11 行创建队列 `qu`，并初始化 4 个元素。第 13、14 行输出其头和尾的元素值。



注意：使用队列容器时，需要包含头文件 `queue.h`。



实例 150 插队（在容器中部插入元素）

【实例描述】

本实例演示如何在队列容器中部插入元素，在队列中部某位置插入元素，需先将该位置后的元素全部用 `pop` 输出，再插入新元素和用 `pop` 输出的所有元素即可。效果如图 8-22 所示。

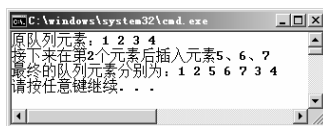


图 8-22 在队列中部插入元素

【实现过程】

定义 `i` 表示插入的第 1 个元素值，`count` 表示在队列的哪个位置后插入新元素。`qu` 为队列，`l` 为链表，`lit` 为链表迭代器。首先对 `qu` 初始化为 1、2、3、4，然后在第 2 个元素后插入元素 5、6、7，并输出结果，代码如下：

```

01 #include <iostream>
02 #include <queue>
03 #include <list>
04 using namespace std;
05
06 void main()
07 {
08     int i=5;                                //插入的第一个元素
09     int count=2;                            //在中部第 2 个元素后插入
10     queue<int> qu;                          //创建队列
11     list<int> l;                            //链表
12     list<int>::iterator lit;               //链表迭代器
13     qu.push(1);                            //原队列压入元素
14     qu.push(2);
15     qu.push(3);
16     qu.push(4);
17
18     cout<<"原队列元素: ";
19     while (qu.empty() != true)              //如果不为空
20     {
21         l.push_back(qu.front());           //读出首元素，然后压入 list 中
22         cout<<qu.front()<<" ";           //输出
23         qu.pop();                          //首元素出队列
24     }
25     cout<<endl;

```



```

26      cout<<"接下来在第 2 个元素后插入元素 5、6、7"<<endl;
27      cout<<"最终的队列元素分别为: ";
28      lit=l.begin();           //指向链表的首元素
29      while(count>0)           //遍历链表元素
30      {
31          cout<<*lit<<" ";     //输出队列元素
32          qu.push(*lit);       //压入队列
33          lit++;               //遍历链表元素
34          count--;             //减 1
35      }
36      while(i<8)
37      {
38          cout<<i<<" ";
39          qu.push(i);
40          i++;                 //元素值加 1
41      }
42      for(;lit!=l.end();lit++)
43      {
44          cout<<*lit<<" ";
45          qu.push(*lit);       //压入队列
46      }
47      cout<<endl;
48  }

```

【代码解析】

第 08~12 行为对各变量的定义,第 13~16 行为对队列元素的初始化。首先输出原队列元素的值,如第 18~24 行。接下来先赋值前 `count` 个队列元素的值,如第 29~35 行。然后在队列中间插入 5、6、7 这 3 个元素,如第 36~41 行。最后在向队列中压入原位置 `count` 后的元素,如第 42~46 行。



注意: 如果对 `list` 容器中部插入元素,可以使用其成员函数 `insert()` 简单完成,代码如下:

```

list<int> c(3);           //由 3 个 0 组成
c.insert(++c.begin(),100); //在第 2 个元素处插入 100
list<int>::iterator lit;  //链表迭代器
for(lit=c.begin();lit!=c.end();lit++)
cout<<*lit<<" ";        //输出各元素

```



实例 151 裁员计划——获取容器元素的个数、删除和清空容器元素

【实例描述】

本实例以裁员计划演示如何获取容器元素的个数、如何删除容器元素,以及如何清空容器元素,效果如图 8-23 所示。

【实现过程】

本实例以链表容器为例进行介绍,其他容器的应用类似,在此

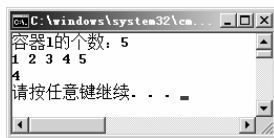


图 8-23 容器的裁员计划



不再赘述。定义链表 l，元素类型为 int，其对应的迭代器为 lit，整型变量 num 存储链表元素的个数。为了对比发现是否如期删除某个元素，先输出原链表中所有的元素值。然后用 erase() 函数删除特定位置或值的元素，最后用 clear() 函数清空容器元素。代码如下：

```

01  #include <iostream>
02  #include <list>
03  using namespace std;
04
05  void main()
06  {
07      list<int> l;                //创建 list 容器对象
08      list<int>::iterator lit;
09      int num=0;
10      l.push_back(1);
11      l.push_back(2);
12      l.push_back(3);
13      l.push_back(4);
14      l.push_back(5);
15      num=l.size();
16      cout<<"容器 l 的个数: "<<num<<endl;
17      for(lit=l.begin();lit!=l.end();lit++)
18          cout<<*lit<<" ";
19      cout<<endl;
20      l.erase(l.begin());        //删除首元素
21      l.pop_back();              //删除最后一个
22      l.pop_front();             //删除首元素
23      l.remove(3);               //删除值为 3 的元素
24      for(lit=l.begin();lit!=l.end();lit++)
25          cout<<*lit<<" ";
26      cout<<endl;
27      l.clear();
28  }
```

【代码解析】

第 07~14 行定义各种变量，并且初始化链表的元素，第 15 行获取链表容器的元素个数，并输出。第 17、18 行输出链表各元素的值。第 20~23 行都是删除链表中某个元素，其中，第 20 行删除链表的首元素，第 21 行运用函数 pop_back() 删除最后一个元素，第 22 行利用函数删除首元素。第 23 行使用 remove() 函数删除值为 3 的元素。第 27 行的 clear() 函数清空容器元素，



注意：其他的容器类似裁员计划，请查相关资料，书写格式都类似。



实例 152 图书印刷——复制元素并自动输出

【实例描述】

在标准模板库的算法中，有一个专门的函数为 copy()，它以前向遍历的方式访问源序列的元素，同时赋值给目标序列。本实例演示复制元素，并自动输出图书的字符，效果如图 8-24 所示。



图 8-24 复制元素并自动输出

【实现过程】

定义字符数组 ch[12]，并赋值为 Hello World，作为图书内容。定义两个向量 v_src 和 v_dst，



首先将 `ch` 值初始化赋给 `vzsrc`，并输出图书样本内容。再调用函数 `copy()`，复制元素到 `v_dst`，并输出 `v_dst` 的值。代码如下：

```
01 #include <iostream>
02 #include <vector>
03 #include <algorithm>
04 using namespace std;
05
06 void main()
07 {
08     char ch[12]="Hello World";
09     vector<char> v_src(12,' '),v_dst(12,' ');//源向量和目标向量
10     cout<<"图书样本内容: ";
11     for(int i=0;i<12;i++)
12     {
13         v_src.at(i)=ch[i];    //赋值
14         cout<<ch[i];        //输出源内容
15     }
16     cout<<endl;
17     copy(v_src.begin(),v_src.end(),v_dst.begin());//把 v_src 复制到 v_dst
18     cout<<"经过印刷, 图书内容为: ";
19     for(int i=0;i<12;i++)
20         cout<<v_dst.at(i);    //输出目标内容
21     cout<<endl;
22 }
```

【代码解析】

第 08、09 行定义变量 `ch`、`v_src` 和 `v_dst`，第 11~15 行初始化 `v_src` 向量并输出源内容。第 17 行用 `copy()` 实现元素复制，第 19、20 行输出目标 `v_dst` 的内容。



注意：使用 STL 算法函数，包含头文件 `algorithm.h`。



实例 153 利用容器适配器实现栈功能

【实例描述】

栈功能表现为先进后出，即先进入栈的元素最后被读出。本实例读出所定义栈对象的所有元素值，效果如图 8-25 所示。



图 8-25 利用容器适配器 `stack` 实现栈功能

【实现过程】

创建栈对象 `st`，压入 4 个元素，值分别为 1、2、3、4。利用其成员函数 `top()` 读出每个元素，代码如下：

```
01 #include <iostream>
02 #include <stack>
03 using namespace std;
04
05 void main()
06 {
07     stack<int> st;
08 }
```



```
09      st.push(1);                //向栈内压入元素
10      st.push(2);
11      st.push(3);
12      st.push(4);
13      while(!st.empty())         //没到末尾
14      {
15          cout<<st.top()<<" ";   //输出
16          st.pop();              //用 pop 输出
17      }
18      cout<<endl;
19  }
```

【代码解析】

第 07 行定义 `stack<int>` 类型数据 `st`，第 09~12 行先后压入值为 1、2、3、4 的元素入栈。第 13~17 行的 `while` 循环读出栈顶元素值，然后将该栈顶元素用 `pop` 输出，如第 16 行。第 13 行的 `empty()` 函数用于判断是否到栈底。



注意：容器内数据的类型可以有多种，此处以 `int` 型为例展开介绍。

第 9 章 C++输入/输出系统

本章介绍 C++输入/输出系统，它的处理对象是外部文件和显示器。C++输入/输出功能非常强大，它可以保存重要的数据，以备将来查看，或者输出显示器以直观地观察。在控制台开发环境下，常用的输入/输出文件工具是流库 `ifstream` 和 `ofstream`，输入/输出显示器工具为 `cin` 和 `cout`。本章的内容主要围绕这 4 个类展开。



实例 154 使用流类库输出一个文件

【实例描述】

本实例演示如何使用流类库输出一个文本文件，该文件名叫 `out.txt`，输出内容为 `Hello World`。`out.txt` 文件的输出效果如图 9-1 所示。



图 9-1 使用流类库输出一个文本文件

【实现过程】

定义输出流对象 `fout`，先打开文件 `out.txt`，再判断是否成功打开。如果成功打开，输出 `Hello World`。代码如下：

```
01 #include <iostream>
02 #include <fstream>
03 using namespace std;
04
05 void main()
06 {
07     ofstream fout;
08     fout.open("out.txt");           //打开文件
09     if(fout.fail())                 //如果打开失败
10         cout<<"out.txt 文件打开失败"<<endl;
11     else
12         fout<<"Hello World"<<"\n"; //输出
13 }
```

【代码解析】

第 07 行定义输出流对象 `fout`，它的默认构造函数可以创建文件。如果打开的文件不存在，先创建。第 08 行使用其成员函数 `open` 打开文件 `out.txt`。第 09~12 行判断是否可以成功打开文件，并做出相应的输出。



注意：应用流类库，包含头文件 `fstream.h`，如第 02 行所示。



实例 155 读写二进制文件

【实例描述】

本实例演示使用二进制数据格式读写文件，它与打开文件的方式有关，关键项是 `ios::binary`，即以二进制数据格式打开文件。如图 9-2(a)所示为读入二进制文件效果，如图 9-2(b)所示为写出二进制文件效果。



图 9-2(a) 读入二进制文件



图 9-2(b) 写出二进制文件

【实现过程】

本实例的读写二进制文件需要先写出二进制文件，再读入，文件名为 `out.me`，二进制文件的后缀后可以随便命名。写出内容为 `IloveC++`，当读入 `out.me` 文件内容后，再输出到屏幕。代码如下：

```
01 #include <iostream>
02 #include <fstream>
03 #include <string>
04 using namespace std;
05
06 void main()
07 {
08     ofstream fout; //输出流对象
09     fout.open("out.me",ios::binary); //打开
10     fout<<"IloveC++"<<"\n"; //写出
11     fout.close(); //关闭
12
13     ifstream fin; //输入流对象
14     fin.open("out.me",ios::binary); //打开
15     string str;
16     fin>>str; //读入
17     cout<<str<<endl; //输出屏幕
18     fin.close(); //关闭
19 }
```

【代码解析】

第 08、13 行分别定义输出流及输入流对象 `fout` 和 `fin`。第 09、14 行打开各自的读写二进制文件 `out.me`。第 10 行写出二进制文件内容，第 15 行定义读入内容所存储的变量 `str`，第 16 行读入文件内容，第 17 行输出 `str` 内容于屏幕。第 11、18 行关闭各自的流对象。



注意：读写二进制文件即是在打开读写文件时以二进制的形式打开。



实例 156 读写记事本

【实例描述】

本实例演示的读写记事本即读写文本文件，实例 154 利用流输入/输出操作符写字符串到文件，此处读写其他类型的数据，读记事本运行效果如图 9-3(a)所示，写记事本运行效果如图 9-3(b)所示。



图 9-3(a) 读记事本

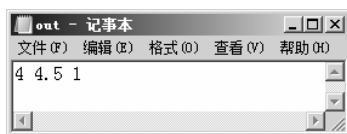


图 9-3(b) 写记事本

【实现过程】

定义变量 data1、data2 和 flag，用于初始化变量，并写到记事本 out.txt。定义变量 data1_copy、data2_copy 和 flag_copy，用于存储读入记事本 out.txt 的数据，并输出屏幕。具体代码如下：

```
01 #include <iostream>
02 #include <fstream>
03 using namespace std;
04
05 void main()
06 {
07     ofstream fout;                //输出流
08     fout.open("out.txt");         //打开文件，如果没有，则先创建
09     int data1=4,data1_copy=0;     //6个写出/读入变量
10     float data2=4.5,data2_copy=.0;
11     bool flag=true,flag_copy=false;
12     fout<<data1<<" "<<data2<<" "<<flag; //写出
13     fout.close();                //关闭
14
15     ifstream fin;                //输入流
16     fin.open("out.txt");         //打开记事本
17     fin>>data1_copy>>data2_copy>>flag_copy; //读入
18     cout<<data1_copy<<'\\n'<<data2_copy<<'\\n'<<flag_copy<<'\\n'; //输出到屏幕
19     fin.close();                //关闭
20 }
```

【代码解析】

第 07、08 行和第 15、16 行是输入/输出流的标准语法格式，第 09~11 行定义 6 个读写变量，并初始化。第 12、17 行写出/读入数据，第 18 行将读入的数据输出到屏幕。



注意：如果第 12 行写出的数据之间没有空格或回车符，在第 17 行读入数据时不是原先写出的值。因为在读记事本数据时，是以空格和回车符为标识区分两个数据的。



实例 157 如何获得文件长度

【实例描述】

如果一个文件中的数据比较大，那么如何知道文件的长度呢？可以通过移动文件指针至文件末尾，获取其当前位置得到文件长度，运行效果如图 9-4 所示。

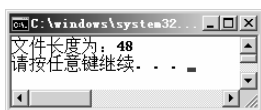


图 9-4 获得文件长度

【实现过程】

定义输入文件流 `fin`，打开文件 `in.txt`，然后移动文件指针到尾部，获取其当前位置，保存到变量 `ps` 中，此时 `ps` 的值即为文件的长度，代码如下：

```
01 #include <iostream>
02 #include <fstream>
03 using namespace std;
04
05 void main()
06 {
07     ifstream fin("in.txt");
08     fin.seekg(0,ios::end);           //设置文件指针到文件流尾部
09     streampos ps=fin.tellg();       //读取文件指针的当前位置
10     cout<<"文件长度为: "<<ps<<endl;
11     fin.close();                   //关闭文件流
12 }
```

【代码解析】

第 07 行定义输入文件流 `fin`，打开文件 `in.txt`。第 08 行移动文件指针至尾部，第 09 行获取文件指针当前位置，并保存到 `streampos` 类型变量 `ps` 中，第 10 行输出文件长度。



注意：文件的长度标识文件的大小。



实例 158 移动文件指针在文件中部写入数据

【实例描述】

在每一个文件中都有文件指针，当文件指针指向文件的哪个位置，下一个被输入的数据会从当前位置开始写出。本实例实现如何移动文件指针在文件中部写入数据，源文件夹效果如图 9-5(a)所示，现要求在 `Hello` 后写入“`World`”数据，写入后的效果如图 9-5(b)所示。



图 9-5(a) 写入前



图 9-5(b) 写入后



【实现过程】

该例旨在演示隐式数据类型转换，用到一个 `float` 型的变量 `i`（用于表示重量 1）、一个 `int` 型的变量 `j`（用于表示重量 2）。这两个变量的和赋给 `int` 型变量 `sum`（表示总重量）。该实例的实现代码如下：

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04 int main()
05 {
06     ofstream out("out.txt", ios::out | ios::in);           //打开文件 out.txt
07     if(out.is_open())                                       //打开成功
08     {
09         out.seekp(21, ios::beg);                             //移动文件指针
10         out.write(" World", 6);                             //写入
11     }
12     else
13         cout<<"out.txt 不存在"<<endl;
14     out.close();                                           //关闭
15 }
```

【代码解析】

第 06 行以输入/输出的形式打开文件 `out.txt`，第 07 行判断是否成功打开 `out.txt`。如果成功打开，移动文件指针到第 21 字节处（第 09 行）。第 10 行使用 `write()` 函数写入数据，最后在第 14 行关闭输出文件流 `out`。



注意：在文件中写入数据能够覆盖当前位置的字符。



实例 159 输出高精度浮点数（cout 高级应用案例）

【实例描述】

在 `cout` 高级应用案例中，可以使用函数 `setprecision()` 输出高精度浮点数。本实例模拟输出 `double` 型变量的高精度结果，运行效果如图 9-6 所示。

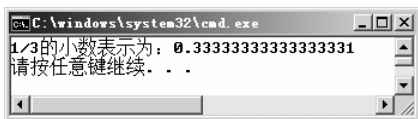


图 9-6 输出高精度浮点数

【实现过程】

定义 `double` 型变量 `a`，赋值为 1.0 除以 3。设置精度 17，并输出其小数表示，代码如下：

```
01 #include <iostream>
02 #include <iomanip>
03 using namespace std;
04
05 void main()
06 {
```



```
07      double a=1.0/3;                                //浮点数
08      cout<<"1/3 的小数表示为: "<<setprecision(17)<<a<<endl;    //设置精度
09  }
```

【代码解析】

第 07 行定义变量 `a`，其值为 1.0 除以 3，它的小数表示由第 08 行输出，精度为 17。



实例 160 使用 `get` 和 `getline` 函数读取 C 风格字符串

【实例描述】

本实例使用 `get()` 和 `getline()` 函数读取记事本中 C 风格的字符串，利用函数 `get()` 逐个字符地读取，而函数 `getline()` 逐行地读取（以回车为终止符）的特征。在记事本中第一行内容为 Hello World，第二行内容为 I love C++，读取效果如图 9-7 所示。



图 9-7 使用 `get` 和 `getline` 读取 C 风格的字符串

【实现过程】

定义输入流对象 `fin`，打开记事本 `out.txt`。定义字符数组 `ch[20]` 和 `ch1[20]`，利用 `get()` 读取 12 个字符，然后利用 `getline()` 读取下一行内容，代码如下：

```
01  #include <iostream>
02  #include <fstream>
03  using namespace std;
04
05  void main()
06  {
07      ifstream fin;                                //输入流
08      fin.open("out.txt");                          //打开
09      char ch[20]={'a'};                            //C 风格的字符串
10      char ch1[20]={'a'};
11      for(int i=0;i<12;i++)
12      {
13          fin.get(ch[i]);                          //逐个读取，共读 12 个
14          cout<<ch[i];
15      }
16      cout<<endl;
17      fin.getline(ch1,20);                          //读取第 2 行
18      cout<<ch1<<endl;
19      fin.close(); //关闭
20  }
```

【代码解析】

第 07~10 行定义输入流对象，并打开记事本 `out.txt`，接着定义两个字符数组。第 11~15 行利用 `get()` 一个个地读取 12 个字符，最后一个字符为回车，所以输出一个回车。第 17 行使用 `getline()` 函数读取下一行。



注意：在 cin 流对象中对于 get() 和 getline() 的应用也是如此。



实例 161 读取流状态

【实例描述】

本实例模拟文件流状态的判定，可以使用以下函数判定文件流状态。

(1) is_open()

判断打开文件是否成功，成功则返回 true，否则返回 false。

(2) good()

操作是否成功，成功则返回 true，否则返回 false。

(3) fail()

与上相反，操作失败则返回 true，否则返回 false。

(4) bad()

如果操作为非法，则返回 true，否则返回 false。

(5) eof()

判断是否到达文件尾，是则返回 true，否则返回 false。

本实例打开文件 in.txt，并读取内容，效果如图 9-8 所示。

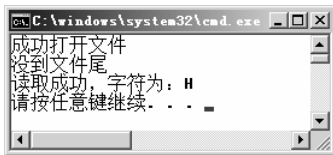


图 9-8 读取文件流状态

【实现过程】

定义输入流对象 fin，打开文件 in.txt，读取第一个字符，并存储于字符 ch 中。之后判断是否成功打开文件、是否到文件尾及是否读取成功等，代码如下：

```
01 #include <iostream>
02 #include <fstream>
03 using namespace std;
04
05 void main()
06 {
07     ifstream fin;                //输入流对象
08     char ch;                     //字符变量
09     fin.open("in.txt");          //打开文件
10     if(fin.is_open())            //成功打开
11     {
12         cout<<"成功打开文件"<<endl;
13         if(fin.eof())             //到达文件尾
14             cout<<"已到文件尾"<<endl;
15         else                     //没到尾
16             {
17                 cout<<"没到文件尾"<<endl;
```



```

18             fin>>ch;                                //读取一个字符
19             if(fin.good())                            //读取成功
20                 cout<<"读取成功, 字符为: "<<ch<<endl;
21             else                                        //读取失败
22                 cout<<"读取失败"<<endl;
23         }
24     }
25     else                                                //打开失败
26         cout<<"文件没被打开"<<endl;
27 }

```

【代码解析】

第 07 行定义输入流对象, 第 08 行定义字符变量, 第 09 行打开文件。第 10~26 行判断是否成功打开文件, 第 13~23 行判断是否到达文件尾, 第 19~22 行判断是否读取成功。



实例 162 设置状态字

【实例描述】

使用 cin/cout (输入/输出) 可以设置其状态字, 比如进制输出、大小写输出等。本实例模拟上述两种状态字的设置, 效果如图 9-9 所示。

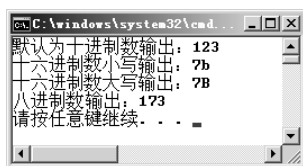


图 9-9 设置状态字

【实现过程】

定义整型变量 a, 初始化为 123。使用十进制、十六进制及八进制数格式输出变量 a。其中, 十六进制数的表示含有字母, 分为小写和大写两种格式输出。代码如下:

```

01 #include <iostream>
02 #include <iomanip>
03 using namespace std;
04
05 void main()
06 {
07     int a=123;                                //整数 123
08     cout<<"默认为十进制数输出: "<<a<<endl;
09     cout<<hex<<"十六进制数小写输出: "<<a<<endl;
10     cout<<setiosflags(ios::uppercase)<<hex<<"十六进制数大写输出: "<<a<<endl;
11     cout<<oct<<"八进制数输出: "<<a<<endl;
12 }

```

【代码解析】

第 07 行定义变量 a, 第 08 行以默认格式输出十进制数 (关键字为 dec), 第 09 行以小写的十六进制数格式输出 (关键字为 hex), 第 10 行以大写的十六进制数格式输出 (关键代码为 ios::uppercase),



第 11 行以八进制数格式输出（关键字为 oct）。



注意：cin/cout（输入/输出）适用于文件流对象。



实例 163 设置输出域宽

【实例描述】

本实例模拟的输出域宽设置只对字符串有效，使用 setw() 函数完成设置。如果设置宽度大于字符串宽度，按设置宽度输出。反之，按实际宽度输出。效果如图 9-10 所示。

【实现过程】

定义两个字符串 str1 和 str2，以左对齐方式输出字符串，一个宽度为 20，一个宽度为 10，代码如下：

```
01 #include <iostream>
02 #include <iomanip>
03 #include <string>
04 using namespace std;
05
06 void main()
07 {
08     string str1="Hello World";           //字符串 1
09     string str2("I love C++");          //字符串 2
10     cout<<setw(20)<<str1;                //宽度 20
11     cout<<"在这里"<<endl;                //输出标志
12     cout<<setw(10)<<str2;                //宽度 10
13     cout<<"在这里"<<endl;                //输出标志
14 }
```

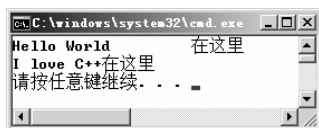


图 9-10 设置输出域宽

【代码解析】

第 02 行为 setw() 和 setiosflags() 函数的头文件，第 08、09 行定义两个字符串。第 10、12 行输出，第 11、13 行输出标志量表示输出的宽度。



注意：setw() 设置的宽度对于 C 风格字符串的应用还涉及其结束符 \0，因此，在 cin 操作中一定要注意。



实例 164 设计一个简单的学生数据库类

【实例描述】

本实例设计学生数据库类 student 及其简单应用，功能有获取新记录值及罗列最新记录，效果如图 9-11 所示。

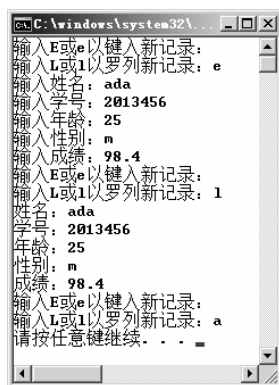


图 9-11 学生数据库类简单应用

【实现过程】

定义学生数据库类 `student`，其成员变量有姓名 `name`、学号 `num`、年龄 `age`、性别 `sex` 及成绩 `score`。此外，定义变量 `count` 计数当前有几个记录。成员函数 `new_record()` 键入新记录、`listnew()` 函数罗列新记录的各项，代码如下：

```
01 class student
02 {
03 private:
04     string name;           //姓名
05     long num;              //学号
06     int age;               //年龄
07     char sex;              //性别
08     float score;           //成绩
09     int count;             //计数
10 public:
11     student();
12     void new_record();      //新记录
13     void listnew();        //罗列新记录
14 };
15 student::student()         //构造函数
16 {
17     name="";
18     num=0;
19     age=0;
20     sex=' ';
21     score=.0;
22     count=0;
23 }
24 void student::new_record()  //新记录
25 {
26     cout<<"输入姓名: ";
27     cin>>name;
28     cout<<"输入学号: ";
29     cin>>num;
30     cout<<"输入年龄: ";
31     cin>>age;
32     cout<<"输入性别: ";
33     cin>>sex;
34     cout<<"输入成绩: ";
35     cin>>score;
```



```
36     count++; //记录个数加 1
37 }
38 void student::listnew() //罗列新记录
39 {
40     if(count<=0) //没有记录
41         cout<<"数据库中没有记录"<<endl;
42     else
43     {
44         cout<<"姓名: "<<name<<endl;
45         cout<<"学号: "<<num<<endl;
46         cout<<"年龄: "<<age<<endl;
47         cout<<"性别: "<<sex<<endl;
48         cout<<"成绩: "<<score<<endl;
49     }
50 }
51 void main()
52 {
53     student _stu; //学生数据库对象
54     char ch; //是否键入新记录
55     bool flag=true; //继续输入
56     while(flag)
57     {
58         cout<<"输入 E 或 e 以键入新记录: ";
59         cout<<endl<<"输入 L 或 l 以罗列新记录: ";
60         cin>>ch; //获取
61         switch(ch) //判断
62         {
63             case 'e':
64             case 'E':
65                 _stu.new_record(); //新记录
66                 break;
67             case 'l':
68             case 'L':
69                 _stu.listnew(); //罗列新记录
70                 break;
71             default:
72                 flag=false; //跳出循环
73                 break;
74         }
75     }
76 }
```

【代码解析】

第 04~09 行为类 `student` 的私有成员变量，以存储各个值。第 09 行的 `count` 表示当前数据库中有几条记录，第 11 行为构造函数，对私有变量进行初始化。第 12、13 行为成员函数的声明，功能分别为键入新记录和罗列新记录。

第 15~23 行为构造函数定义，对成员变量进行初始化。第 24~37 行为获取新记录的值，第 38~54 行罗列新记录的值，判断当前数据库记录是否为 0，如第 40 行。如果为 0，表示数据库当前没有记录，反之，则输出当前记录。

`main()` 函数中，第 53 行定义数据库类对象，第 54 行定义是否键入新记录，第 55 行定义是否继续输入变量 `flag`。第 56~75 行为 `while` 循环，该循环中包含 `switch` 判断结构，根据选择不同的 `ch` 值，调用不同的成员函数。当 `ch` 不是 `e`、`E`、`l` 和 `L` 的其中一个时，退出循环，完成 `main()` 函数的执行。



注意：本实例只是定义一个简单的学生数据库类，具体应用请看后面的例子。



实例 165 实现程序退出自动保存数据库内容到磁盘文件

【实例描述】

本实例继续实例 164 实现的学生数据库类 `student`，实现在程序退出时自动将数据库当前的内容保存到磁盘文件，如图 9-12 所示为磁盘文件内容。



图 9-12 自动保存数据库内容到磁盘文件

【实现过程】

在类 `student` 中添加成员变量 `fout`，以打开磁盘文件 `out.txt`。添加成员函数 `init()`（初始化数据库字段）和 `quit()` 函数（退出程序，写出当前变量值），代码如下：

```
01  ...
02  using namespace std;
03
04  class student
05  {
06  private:
07      ...
08      ofstream fout; //输出流对象
09  public:
10      ...
11      void init(); //初始化文件
12      void quit(); //退出
13  };
14  void student::init()
15  {
16      fout.open("out.txt", ios::out | ios::ate); //打开输出文件
17      if(fout.good()) //被打开
18      {
19          fout<<setiosflags(ios::left)<<setw(10)<<"姓名"; //初始化数据库字段
20          fout<<setiosflags(ios::left)<<setw(10)<<"学号";
21          fout<<setiosflags(ios::left)<<setw(10)<<"年龄";
22          fout<<setiosflags(ios::left)<<setw(10)<<"性别";
23          fout<<setiosflags(ios::left)<<setw(10)<<"成绩\n";
24      }
25  }
26  void student::quit()
27  {
28      fout<<setiosflags(ios::left)<<setw(10); //设置输出格式
29      fout<<"name<<"<<num<<" " <<age<<" " <<sex<<" "<<score<<"\n";
30      //输出当前数据库内容
31      fout.close(); //关闭
32  }
```



```

33 void main()
34 {
35     ...
36     _stu.init(); //初始化文件
37     while(flag)
38     {
39         ...
40         cout<<endl<<"输入 Q 或 q 退出程序: ";
41         cin>>ch; //获取
42         switch(ch) //判断
43         {
44             ...
45             case 'q':
46             case 'Q':
47                 _stu.quit(); //退出
48                 flag=false;
49                 break;
50         }
51     }
52 }

```

【代码解析】

第 08 行为输出流对象,第 11、12 行为成员函数 `init()` 和 `quit()`,它们的定义体分别为第 14~25 行和第 26~32 行。在初始化 `init()` 函数中,先打开磁盘文件 `out.txt`,再输出数据库的字段名。在 `quit()` 函数中,按格式输出当前记录的各值。回到 `main()` 函数的第 36 行,先调用初始化函数,即先打开磁盘文件。在 `while` 循环中,当输入 `q` 或 `Q` 值时,表示退出程序,此时调用 `quit()` 函数,执行写出数据功能。按格式输出当前记录值,如第 28、29 行。



注意: 第 16 行的 `ios::ate` 参数是在打开文件时,将文件指针指向文件的末尾。



实例 166 实现程序启动时自动读取数据库

【实例描述】

继续实例 165,除了可以实现程序退出时自动保存数据,我们还可以在程序启动时读取数据库,实例运行效果如图 9-13 所示。

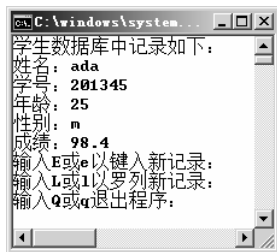


图 9-13 程序启动时自动读取数据库

【实现过程】

在实例 165 的基础上,添加成员变量 `fin`,打开 `out.txt`,读取内容。再改变其成员函数 `init()`



的内容，即可完成当程序启动时读取数据库，代码如下：

```
01  ...
02  class student
03  {
04  private:
05      ...
06      ifstream fin;                //输入流对象
07      ...
08  };
09  void student::init()
10  {
11      fin.open("out.txt",ios::in);    //打开输入文件
12      if(fin.good())                //被打开
13      {
14          cout<<"学生数据库中记录如下: "<<endl;
15          string temp;                //临时字符串，读取字段
16          for(int i=0;i<5;i++)
17              fin>>temp;              //读取字段
18          fin>>name>>num>>age>>sex>>score; //读取记录
19      }
20      cout<<"姓名: "<<name<<endl;      //输出记录值
21      cout<<"学号: "<<num<<endl;
22      cout<<"年龄: "<<age<<endl;
23      cout<<"性别: "<<sex<<endl;
24      cout<<"成绩: "<<score<<endl;
25  }
26  void main()
27  {
28      ...
29      _stu.init();                  //初始化文件
30      while(flag)
31      {
32          ...
33      }
34  }
```

【代码解析】

第 06 行为输入流对象的定义，第 09~25 行为函数 `init()` 的新定义。其中，第 11 行打开输入文件 `out.txt`。第 12 行标识打开成功，由于数据库的第 1 行为其字段标识符，因此需要先读取到 `temp` 临时字符串中，以使文件指针指向第 2 行（共循环 5 次，因为有 5 个字段）。接下来读取每个字段下的值，如第 18 行。最后由第 20~24 行输出。



注意：事实上，第 16、17 行的移动文件指针可以直接由函数 `seekp()` 完成。但此实例不清楚第 1 行由多少字节构成，却只知道有几个字段，所以第 1 种方法可取。



实例 167 开发一个完整的学生数据管理系统 V1.0

【实例描述】

本实例开发一个完整的学生数据管理系统，每次登记学生信息时，将该信息写入文件 `stu_info.txt` 中。读取信息时，打开该文件，显示到屏幕，效果如图 9-14 所示。

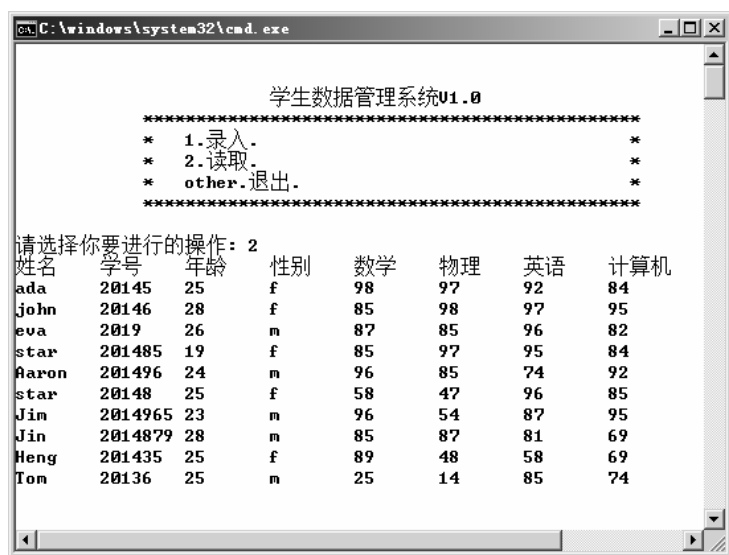


图 9-14 学生数据管理系统 V1.0

【实现过程】

结合前面的例子所用类 student，添加成员变量 score 型 s，表示学生的各科成绩。成员函数 mainpage() 为显示主页面，readfile() 和 writefile() 用于读写文件 stu_info.txt，insert() 用于输入学生信息，display() 读取学生信息，并显示于屏幕。代码如下：

```

01 #include <iostream>
02 #include <fstream>
03 #include <iomanip>
04 #include <string>
05 #include <assert.h>
06 using namespace std;
07
08 struct score                                //成绩结构体
09 {
10     int Math;                                //数学
11     int Physics;                            //物理
12     int English;                            //英语
13     int Computer;                          //计算机
14 };
15 class student
16 {
17 private:
18     ...
19     score s;                                //成绩
20     ...
21 public:
22     ...
23     void mainpage();                        //主页面
24     void readfile();                       //读入文件
25     void writefile();                      //写出文件
26     void insert();                         //输入学生信息
27     void display();                        //读取
28 };
29 void student::readfile()

```



```
30 {
31     cout<<"姓名  学号  年龄  性别  数学  物理  英语  计算机"<<endl;
32     fin.open("stu_info.txt",ios::in|ios::out);           //打开输入文件
33     if(fin.good())                                         //被打开
34     {
35         while(!fin.eof())                                  //没到文件末尾
36         {
37             fin>>name>>num>>age>>sex>>s.Math>>s.Physics>>s.English>>s.Computer;

38                                                     //读取记录
39             cout<<name<<" "<<num<<" "<<age<<" "<<sex<<" "; //输出屏幕
40             cout<<s.Math<<" "<<s.Physics<<" "<<s.English<<"
41             "<<s.Computer<<endl;
42         }
43     }
44     fin.close();                                           //关闭
45 }
46 void student::writefile()
47 {
48     fout.open("stu_info.txt",ios::out|ios::in|ios::ate);
49     if(fout.good())
50     {
51         fout<<"\n"<<name<<" "<<num<<" "<<age<<" "<<sex<<" ";
52         fout<<s.Math<<" "<<s.Physics<<" "<<s.English<<"
53         "<<s.Computer;                                     //写出
54     }
55     cout<<"写入成功"<<endl;
56     fout.close();                                          //关闭
57 }
58 ...
59 void student::insert()                                   //新记录
60 {
61     char n;
62     do
63     {
64         cout<<"输入姓名: ";
65         cin>>name;
66         cout<<"输入学号: ";
67         cin>>num;
68         cout<<"输入年龄: ";
69         cin>>age;
70         cout<<"输入性别: ";
71         cin>>sex;
72         cout<<"输入数学成绩: ";
73         cin>>s.Math;
74         cout<<"输入物理成绩: ";
75         cin>>s.Physics;
76         cout<<"输入英语成绩: ";
77         cin>>s.English;
78         cout<<"输入计算机成绩: ";
79         cin>>s.Computer;
80         writefile();                                       //写出学生信息
81         count++;
82         cout<<"继续输入? (Y/N) ";
83         cin>>n;
84     }while(n!='Y' || n!='y');
85     system("cls");                                        //清屏
86     mainpage();                                          //主页面
87 }
```



下列代码列出函数 `display()`、`mainpage()` 和 `main()` 的定义体。

```

01 void student::display()
02 {
03     readfile();
04     mainpage(); //主页面
05 }
06 void student::mainpage()
07 {
08     int choose; //选择
09     cout<<"\n\n";
10     cout<<"\t\t\t 学生数据管理系统 v1.0\n";
11     cout<<"\t ";
12     for(int i=0;i<47;i++)
13         cout<<"*";
14     cout<<"\n";
15     cout<<"\t * 1.录入.\t\t\t\t\t * \n";
16     cout<<"\t * 2.读取.\t\t\t\t\t * \n";
17     cout<<"\t * other.退出.\t\t\t\t\t * \n";
18     cout<<"\t ";
19     for(int i=0;i<47;i++)
20         cout<<"*";
21     cout<<"\n\n";
22     cout<<"请选择你要进行的操作: ";
23     cin>>choose; //输入选择
24     switch(choose)
25     {
26     case 1: insert(); //录入
27         break;
28     case 2: display(); //读取
29         break;
30     default: exit(-1);
31     }
32 }
33 void main()
34 {
35     student _stu; //学生数据库对象
36     _stu.mainpage(); //显示主页面
37 }

```

【代码解析】

(1) 第 08~14 行为结构体 `score`，表示学生各科成绩。第 15~28 行为类 `student`，第 19 行为成员变量 `s` 的声明，第 23~27 行为其成员函数。第 29~45 行为 `readfile()` 的定义体，当没有到达文件末尾时，继续读取文件，并显示结果于屏幕。第 46~57 行为 `writefile()` 的定义体，当产生新记录时，写入文件。`readfile()` 被 `display()` 调用，`writefile()` 被 `insert()` 函数调用。第 59~87 行为 `insert()` 函数定义体，主要用于插入新信息。

(2) 第 01~05 行为 `display()` 函数，先调用 `readfile()`，再调用 `mainpage()` 显示主页面。第 06~32 行为 `mainpage()` 函数，在 `main()` 中，定义类对象 `_stu`，调用成员函数 `mainpage()`。



注意：在 `readfile()` 函数中，第 35 行判断是否到达文件末尾。如果文件末尾有回车符，会继续输出当前变量的值，即最后一条信息会输出两次。所以，在操作文件时，需要特别注意文件末尾的回车符。



实例 168 开发一个完整的学生数据管理系统 V2.0

【实例描述】

本实例是在实例 167 的基础上，加入两个新的功能，即更新数据和删除数据。所以更新为系统 V2.0，效果如图 9-15 所示，是删除学号为 20148 的学生信息。本实例的学生数据管理系统有严格的格式要求，即对应每项的数据字节大小需要一致，否则在进行读取操作时会进入无限循环。

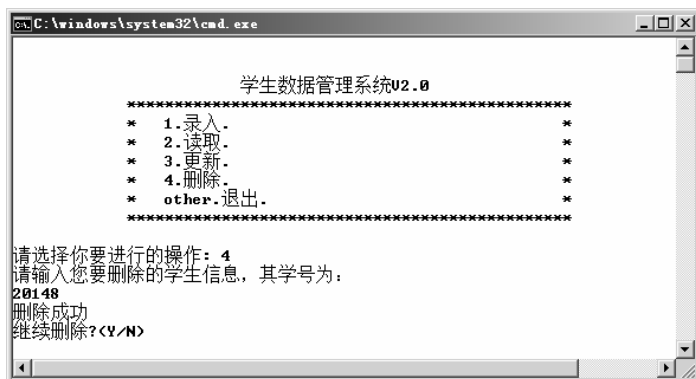


图 9-15 学生数据管理系统 V2.0

【实现过程】

首先，定义成员函数 `update()` 和 `del()`，分别用于更新和删除数据。其次，`stu_info.txt` 文件的格式需要按照如图 9-16 所示的格式书写。

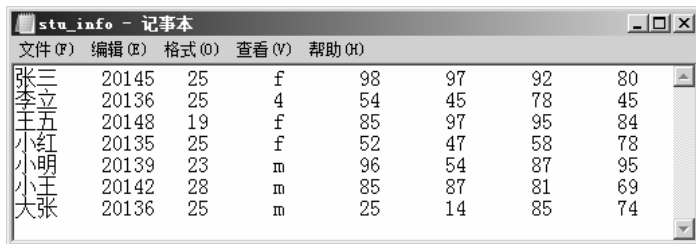


图 9-16 学生数据管理系统 V2.0 文本文件

即姓名只占两字节，学号为 5 位，年龄为两位，性别占 1 字节，剩余的成绩都只是两位整型数据。本实例的重点是成员函数 `update()`，其代码如下：

```
01 void student::update()
02 {
03     ifstream fin;
04     ofstream fout;
05     bool flag=false;                                //是否找到对应项
06     streampos ps_ex=0;                              //前一行末尾位置
07     streampos ps_every=0;                          //每一行有几个位置单位
08     fin.open("stu_info.txt",ios::in|ios::out);      //打开输入文件
09     char ch;
10     if(!fin.good())
```



```

11         return;
12     do
13     {
14         int n;
15         cout<<"请输入您要更新的学生信息, 其学号为: "<<endl;
16         cin>>num;
17         while(!fin.eof())
18         {
19             long temp;                                //临时变量
20             fin>>name>>temp>>age>>sex>>s.Math>>s.Physics>>s.English>>s.Computer;
21                                                     //读取记录
22             ps_every=fin.tellg()-ps_ex;                //计算每行的位置单位个数
23             ps_ex=fin.tellg();                          //获取前一行的末尾位置
24             if(temp==num)                              //相同
25             {
26                 flag=true;                            //找到
27                 fin.close();
28                 fout.open("stu_info.txt",ios::out|ios::in);
29                 if(!fout.good())
30                     return;
31                 fout.seekp(ps_ex-ps_every,ios::beg);    //移动文件指针
32                 cout<<"请依次输入下列需更新的项目,不需要更新的请原样输入: "<<endl;
33                 cout<<"输入姓名: ";
34                 cin>>name;
35                 cout<<"输入学号: ";
36                 cin>>num;
37                 cout<<"输入年龄: ";
38                 cin>>age;
39                 cout<<"输入性别: ";
40                 cin>>sex;
41                 cout<<"输入数学成绩: ";
42                 cin>>s.Math;
43                 cout<<"输入物理成绩: ";
44                 cin>>s.Physics;
45                 cout<<"输入英语成绩: ";
46                 cin>>s.English;
47                 cout<<"输入计算机成绩: ";
48                 cin>>s.Computer;
49                 fout<<"\n"<<name<<" "<<num<<" "<<age<<" "<<sex<<" ";
50                 fout<<s.Math<<" "<<s.Physics<<" "<<s.English<<"
51                     "<<s.Computer;                    //写出
52                 cout<<"更新成功"<<endl;
53                 break;
54             }
55         }
56         if(flag==false)
57             cout<<"没有找到符合选项"<<endl;
58         cout<<"继续更新?(Y/N) ";
59         cin>>ch;
60     }while(ch=='Y' || ch=='y');
61     fout.close();                                //关闭
62     system("cls");                                //清屏
63     mainpage();                                  //主页面
64 }

```

成员函数 del()的代码如下(它的实现原理是将该行数据全部用空格表示)。

```

01 void student::del()
02 {
03     ifstream fin;

```



```
04      ofstream fout;
05      bool flag=false;
06      streampos ps_ex=0;                                //前一行末尾位置
07      streampos ps_every=0;                             //每一行有几个位置单位
08      fin.open("stu_info.txt",ios::in|ios::out);        //打开输入文件
09      char ch;
10      if(!fin.good())
11          return;
12      do
13      {
14          int n;
15          cout<<"请输入您要删除的学生信息,其学号为: "<<endl;
16          cin>>num;
17          while(!fin.eof())
18          {
19              long temp;                                //临时变量
20              fin>>name>>temp>>age>>sex>>s.Math>>s.Physics>>s.English>>s.Computer;
21              //读取记录
22              ps_every=fin.tellg()-ps_ex;               //计算每行的位置单位个数
23              ps_ex=fin.tellg();                       //获取前一行的末尾位置
24              if(temp==num)                             //相同
25              {
26                  flag=true;
27                  fin.close();
28                  fout.open("stu_info.txt",ios::out|ios::in);
29                  if(!fout.good())
30                      return;
31                  fout.seekp(ps_ex-ps_every,ios::beg); //移动文件指针
32                  streampos ps=fout.tellp();           //获取当前位置
33                  while(ps!=ps_ex)
34                  {
35                      fout<<" ";
36                      ps=fout.tellp();                 //再次获取
37                  }
38                  cout<<"删除成功"<<endl;
39                  break;
40              }
41          }
42          if(flag==false)
43              cout<<"没有找到对应项"<<endl;
44          cout<<"继续删除?(Y/N) ";
45          cin>>ch;
46          }while(ch=='Y' || ch=='y');
47          fout.close();                                //关闭
48          system("cls");                                //清屏
49          mainpage();                                  //主页面
50      }
```

【代码解析】

(1) update()

第 03、04 行定义输入/输出流对象 `fin` 和 `fout`, 第 05 行定义布尔型变量 `flag`, 标识所更新的对象是否存在。第 06、07 行的 `streampos` 型变量分别记录前一行的末尾位置和每一行有几个位置单位。`ps_every` 的应用是在 `fout` 输出更新内容时, 先将文件指针指向更新行的首部。

第 08~11 行打开输入文件, 并判断是否成功打开。第 12~60 行的 `do...while` 循环结构判断是否还需更新, 其内有 `while` 循环, 寻找目标学号。第 17 行表示还没到达文件末尾, 第 22、23 行记录当前位置的前一行位置及当前行的位置个数。第 24 行表示找到目标学号, 其后的第



25~54 行对该行数据进行更新。在更新前，先将 fout 对象的文件指针指回目标行的首部，如第 31 行操作。

当不需要继续更新时，关闭流对象，清屏并返回主页面。另外，输入流对象必须在打开输出流对象前关闭，如第 27 行。

(2) del()

第 03、04 行定义输入/输出流对象 fin 和 fout，第 05 行定义布尔型变量 flag，标识所删除的对象是否存在。第 06、07 行的 streampos 型变量分别记录前一行的末尾位置和每一行有几个位置单位。ps_every 的应用是在 fout 删除内容时，先将文件指针指向更新行的首部。

第 08~11 行打开输入文件，并判断是否成功打开。第 12~46 行的 do...while 循环结构判断是否还需更新，其内有 while 循环，寻找目标学号。第 17 行表示还没到达文件末尾，第 22、23 行记录当前位置的前一行位置及当前行的位置个数。第 24 行表示找到目标学号，其后的第 25~40 行删除该行数据。在删除前，先将 fout 对象的文件指针指回目标行的首部，如第 31 行操作。

当不需要继续更新时，关闭流对象，清屏并返回主页面。另外，输入流对象必须在打开输出流对象前关闭，如第 27 行。第 33~37 行是将所删除目标行的目标全部用空格代替，它的判断条件是当前文件指针位置是否等于 ps_ex，如第 33 行。



注意：从实例 165 开始定义成员变量 fin 和 fout，本实例在类 student 的基础上，删除这两个成员变量。原因是在读取、更新和删除功能中，各个函数的输入/输出流互相影响，文件指针时常发生变化。所以将文件输入/输出流定义为局部变量。

第 10 章 各类经典案例与解决方法

本章的各个实例皆是经典型、较易出错的概念总结。经典案例大都发生在错误使用指针、错误释放内存、数组访问越界、程序异常退出、栈溢出等方面。在描述案例的同时，还给出了相应的解决方法。



实例 169 错误释放指针导致程序崩溃

【实例描述】

本实例模拟错误地释放指针时导致程序崩溃，它表现在指针超出合法内存的边界。当用指针读取合法内存块中的数据时，如果读到内存的最后一个元素时，再重复执行一次相同的指令后，指针的值加 1 会越界，此时释放指针就会导致程序崩溃，解决方法是及时将指针指回首地址。本实例利用指针读取内存的值，效果如图 10-1 所示。



图 10-1 错误释放指针导致程序崩溃

【实现过程】

定义 `double` 型一维内存 `aa` 及相同类型的指针 `copy_aa`，将 `aa` 的首地址赋给 `copy_aa`，初始化内存 `aa`，并输出各内存值，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     double *aa=new double[10];           //double 型一维内存
07     double *copy_aa=aa;                  //首地址
08     cout<<hex;                            //十六进制数输出
09     for(int i=0;i<10;i++)                //初始化并输出内存值
10     {
```



```
11         *aa=i;                                //初始化
12         cout<<aa<<" "<<*aa<<endl;            //输出地址及内存值
13         aa++;                                  //地址加 1
14     }
15     cout<<endl<<"当前指针指向内存的地址为: "<<aa<<endl;
16     aa=copy_aa;                                //指回首地址
17     cout<<endl<<"当前指针指回首地址为: "<<aa<<endl;
18     delete[] aa;                                //释放内存
19     aa=NULL;                                    //释放指针
20 }
```

【代码解析】

第 06、07 行定义一维内存 `aa` 及指针 `copy_aa`。第 08 行表示使用十六进制数输出内存值及其地址。第 09~14 行初始化内存，并输出各内存值。第 18、19 行分别释放内存及指针。



注意：该方法使用地址获取内存值，而不是下标法，所以一定要注意在最后指回首地址。



实例 170 栈溢出的经典案例

【实例描述】

每块被申请的内存都被称为栈变量，如果所申请的整块内存大小大于栈空间的限定时，就被称为栈溢出。本实例演示怎样定义内存就被视为栈溢出，及如何解决。如图 10-2 所示的效果为栈溢出的提示。



图 10-2 栈溢出提示

【实现过程】

定义一个 1000×1000 的整型数组 `a`，并为其初始化。内存 `a` 是被整块申请，因为其大小已超出栈空间的大小，所以会被认为栈溢出。解决方法是采用动态申请内存，即使申请大小超出限定值，但因为不是连续内存块，而是由零碎内存块所组成的，所以不被认为栈溢出。这就是动态申请内存的优点，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     //int a[1000][1000];          //静态申请
07     int **a=new int*[1000];      //动态申请
```



```

08     for(int i=0;i<1000;i++)
09         a[i]=new int[1000];
10     for(int i=0;i<1000;i++)           //初始化
11     {
12         for(int j=0;j<1000;j++)
13             a[i][j]=i+j;
14     }
15     delete []a;                       //释放
16     a=NULL;
17 }

```

【代码解析】

第 06 行为静态申请内存，但因其内存大于 1000×1000 ，超出限定值，被视为栈溢出。所以改为第 07~09 行的动态申请内存方式。第 10~14 行对内存 `a` 进行初始化。第 15、16 行释放内存。



注意： 动态申请的内存并不是整体的。



实例 171 判断语句经典错误案例（if）

【实例描述】

本实例针对 if 判断结构的判断条件做介绍，如果 if 判断条件为真，执行 if 语句。当判断两个值相等时，经常会少写一个 `=` 运算符，此时程序的执行效果并不如期望一样，效果如图 10-3 所示。

【实现过程】

定义变量 `num`，并初始化为 0。先输出 `num` 的值，然后判断 `num` 是否等于 20。最后再次输出 `num` 的值，代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     int num=0;                       //定义变量 num
07     cout<<"num="<<num<<endl;       //输出变量的值
08     if(num=20)                       //判断变量 num 等于 20
09         cout<<"num 等于 20"<<endl;
10     else                             //不等于 20
11         cout<<"num 不等于 20"<<endl;
12     cout<<"num="<<num<<endl;       //再次输出 num 的值
13 }

```

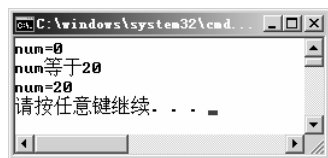


图 10-3 判断语句错误案例

【代码解析】

第 06、07 行定义变量 `num`，并初始化为 0。第 08~11 行判断 `num` 是否等于 20，但是第 08 行的判断条件的 `=` 运算符变成了赋值运算符 `=`，所以表达式的返回值是 `num` 的当前值，此时为真，所以执行 if 的语句，输出 `num 等于 20`。第 12 行再次输出 `num` 值，已被改为 20。



注意：判断条件中如果有==运算符，一定不要少了一个等号。



实例 172 使用指针引用问题

【实例描述】

在前面的例子中介绍过指针及引用的使用，那么如何使用指针引用呢？本实例将详细介绍。指针引用不仅可以改变指针所指对象，还能改变指针本身，下面分别应用指针及指针引用，并输出各自的值，效果如图 10-4 所示。

【实现过程】

定义两个函数 fun1()和 fun2()，分别定义两个整型变量 a 和 b，并初始化为 10 和 20。各自的整型指针 temp 指向变量 a。函数 fun1()中指针 b 先指向 a，然后指向变量 b，最后改变变量 b 的值为 5。在函数 fun2()中定义指针引用 p，最初被赋值为 a 的地址，然后又指回 b，进而改变 b 的值。代码如下：

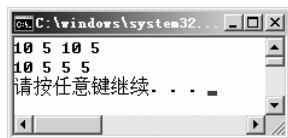


图 10-4 使用指针引用

```
01 #include <iostream>
02 using namespace std;
03
04 void fun1 ()
05 {
06     int a=10;
07     int b=20;
08     int *temp=&a;//temp 取址于 a
09     int *p=temp;//p 也取址于 a
10     p=&b;//又指向 b
11     *p=5;//改变 b 的值
12     cout<<a<<" "<<b<<" "<<*temp<<" "<<*p<<endl;
13 }
14 void fun2 ()
15 {
16     int a=10;
17     int b=20;
18     int *temp=&a;//temp 取址于 a
19     int *&p=temp;//引用 p 也取址于 a
20     p=&b;//又指向 b
21     *p=5;//改变 b 的值
22     cout<<a<<" "<<b<<" "<<*temp<<" "<<*p<<endl;
23 }
24 void main()
25 {
26     fun1 ();
27     fun2 ();
28 }
```

【代码解析】

函数 fun1()中应用指针变量，函数 fun2()中应用指针引用。在函数 fun1()中，它的内存指向改变如图 10-5 所示。

在函数 fun2()中，它的内存指向改变如图 10-6 所示。

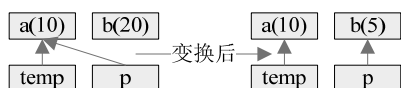


图 10-5 指针在内存中的指向改变

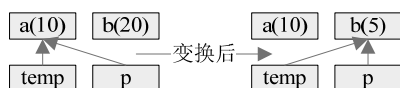


图 10-6 指针引用在内存中的指向改变

当使用指针引用后，它是指针 `temp` 的另一个名字。所以，如果 `p` 的指向改变，本质上就是改变 `temp` 的指向。



注意：引用实质上是操作同一个变量，只不过有一个曾用名。



实例 173 显式调用析构函数案例

【实例描述】

显式调用的另一面是隐式调用，一般情况下不需要显式调用析构函数，该工作系统会自动完成。如果系统已有析构对象，程序员再次调用析构函数时，可能会重复释放内存，清除已经不存在的数据，将会导致程序运行错误。另一种情况是，显式调用析构函数并不能起到隐式调用析构函数的作用，不能彻底销毁对象。针对此隐患，本实例给出演示，效果如图 10-7 所示。

【实现过程】

定义类 `data` 用于模拟显式调用析构函数，其中有成员变量 `_var`、指针 `temp_point` 和标志量 `heap_deleted`（表示堆内存是否被释放）。在构造函数中初始化成员变量并构造对象，在析构函数中释放堆内存，成员函数 `change()` 改变当前成员变量 `_var` 的值，`output()` 输出成员变量 `_var` 的值，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 class data
05 {
06 private:
07     int _var; //内存数据
08     int *temp_point; //临时指针
09     bool heap_deleted; //堆内存被释放标志
10 public:
11     data(int var)
12     {
13         heap_deleted=false; //初始化堆内存标志量
14         temp_point=new int; //申请内存
15         *temp_point=var; //取址并初始化
16         _var=*temp_point; //初始化_var
17         cout<<"构造对象"<<endl;
18     }
19     ~data() //析构函数
20     {
21         if(heap_deleted==false) //堆内存没被释放
22         {
23             cout<<"地址为: "<<temp_point<<endl;
```

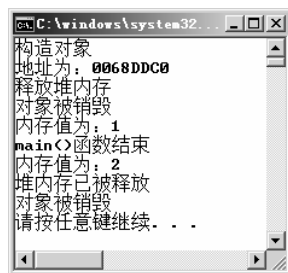


图 10-7 显式调用析构函数



```

24         delete temp_point;           //释放内存
25         heap_deleted=true;          //标志已被释放
26         cout<<"释放堆内存"<<endl;
27     }
28     else                               //已被释放
29         cout<<"堆内存已被释放"<<endl;
30     cout<<"对象被销毁"<<endl;
31 }
32 void change(int x)                     //改变成员变量值
33 {
34     _var=x;
35 }
36 void output()                         //输出成员变量值
37 {
38     cout<<"内存值为: "<<_var<<endl;
39 }
40 };
41 void main()
42 {
43     data a(1);                         //定义类对象
44     a~data();                          //显式调用构造函数
45     a.output();                        //输出值
46     cout<<"main() 函数结束"<<endl;
47     a.change(2);                       //改变变量值
48     a.output();                       //再次输出变量值
49 }

```

【代码解析】

第 04~40 行为类 `data` 的定义体，其中，第 07~09 行为私有成员变量，第 11~18 行为构造函数，第 19~31 行为析构函数。第 32~39 行分别为函数 `change()` 和 `output()` 的定义体。在第 41~49 行的 `main()` 函数定义体中，定义类对象 `a`，接着调用其析构函数（第 44 行），此时释放堆内存，然后在第 45 行输出 `_var` 的值。在之后的第 47、48 行改变变量及输出其值。由效果图可知，即使第 47、48 行没有调用析构函数，系统在最后也会自动执行。



注意：一般情况下，类对象的销毁都是由系统处理的，不需要显式调用析构函数。



实例 174 cin 输入队列错误案例

【实例描述】

当使用 `cin` 输入字符串时，会因为字符串中含有 `cin` 的结束标志量而终止读取。`cin` 的结束标志量有回车、空格及 Tab 键。当字符串的值为 `jfak ajfkdl`，其中含有空格，但 `cin` 获取的字符串值只是 `jfak`。如果要将空格也读入，此时需要调用其他成员函数 `get()` 读取一定长度的字符串，效果如图 10-8 所示。

【实现过程】

定义字符数组 `ch`，大小为 20。首先用 `get()` 函数读取



图 10-8 cin 输入队列错误



缓冲区中的 20 个字符，然后使用操作符>>读取以空格结束的字符串，代码如下：

```
01  #include <iostream>
02  using namespace std;
03
04  void main()
05  {
06      char ch[20];                //字符串
07      cin.get(ch,20);             //函数 get () 读入
08      cout<<"读满字符串: "<<ch<<endl;
09      cin>>ch;                   //操作符>>读入
10      cout<<"遇到空格结束的字符串队列读入: "<<ch<<endl;
11  }
```

【代码解析】

第 06 行定义字符数组 ch，第 07 行使用成员函数 get() 获取 20 个字符，并由第 08 行输出。第 09、10 行获取操作符>>读入的字符串，并输出。



注意：除了 get() 函数，还可以使用 getline() 函数读取一定长度的字符串。但是当字符串长度太长时，getline() 函数的使用会影响之后的 cin 操作。



实例 175 数组越界访问案例

【实例描述】

在访问数据元素时，可能会存在访问越界。本实例模拟当数组访问超界时，会出现什么错误。如图 10-9 所示为其错误提示。



图 10-9 数组超界访问

【实现过程】

定义整型数组 a，长度为 10。访问下标值为 0 到 10 的元素值，并赋值，代码如下：

```
01  #include <iostream>
02  using namespace std;
03
04  void main()
05  {
06      int a[10];
07      for(int i=0;i<=10;i++)        //当访问下标为 10 的元素时超界
```



```
08         a[i]=i;
09     }
```

【代码解析】

在第 07 行 `i` 的值可以为 10，此时访问数组的元素下标为 10，已经超出其界限。



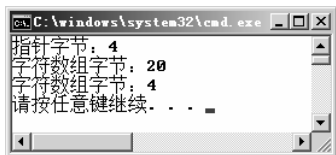
注意：在操作数组时一定不能越界，严重时可能会改变重要内存的值，将会带来不可逆转的后果。



实例 176 sizeof 产生错误实例

【实例描述】

在指针应用的实例中，对于 `sizeof` 的使用可能会出现非预期效果。本实例重点介绍指针所占内存字节的大小，及字符数组作用函数参数时，它的字节变化，效果如图 10-10 所示。



【实现过程】

定义长度为 20 的字符数组 `ch`，并初始化为 `Hello World`。定义 `char` 型指针 `point`，将其指向 `ch` 的首地址。另外，定义函数 `length()`，它的参数为字符数组，作用输出字符数组的字节大小。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void length(char ch[])
05 {
06     cout<<"字符数组字节: "<<sizeof ch<<endl;    //输出输入参数的字节大小
07 }
08 void main()
09 {
10     char ch[20]="Hello World";    //字符数组
11     char *point;    //char 型指针
12     point=ch;    //赋值
13     cout<<"指针字节: "<<sizeof point<<endl;    //输出指针的字节大小
14     cout<<"字符数组字节: "<<sizeof ch<<endl;    //输出字符数组所占字节
15     length(ch);    //调用 length() 函数
16 }
```

图 10-10 sizeof 产生错误

【代码解析】

第 04~07 行为函数 `length()` 的定义体，第 10~12 行定义变量 `ch` 和 `point`，并将 `point` 指向 `ch` 的首地址。第 13、14 行输出指针的字节及数组的字节，结果分别为 4 和 20。因为指针用于存放地址，它是一个 `unsigned long` 型的变量，所以字节大小为 4。另外，字符数组的字节数应该为数组的大小乘以其类型所占字节的大小，所以为 20。第 15 行调用 `length()` 函数，它的输入参数为字符数组，但它是首地址传入的函数，因此它的输出值只是其地址的字节大小，故为 4。



注意：当数组作为参数时，它传入的是其首地址，而不是整个数组。



实例 177 使用类自动管理指针

【实例描述】

在使用指针时必须非常谨慎，及时初始化并且释放它。本实例定义一个自动管理指针的类模板，程序运行效果如图 10-11 所示。

【实现过程】

定义类模板 `ptr`，在其构造函数中对成员指针变量 `ip` 进行初始化，并在析构函数中释放它，具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 template <class T>
05 class ptr
06 {
07 private:
08     T *ip;                                //指针
09 public:
10     ptr(T *p);                            //构造函数
11     ~ptr();                               //析构函数
12 };
13 template <class T>
14 ptr<T>::ptr(T *p)
15 {
16     ip=p;                                //初始化
17     cout<<"调用指针管理类构造函数"<<endl;
18 }
19 template <class T>
20 ptr<T>::~~ptr()
21 {
22     ip=NULL;                             //释放
23     cout<<"指针管理类析构函数被调用"<<endl;
24 }
25 void main()
26 {
27     int a=10;                            //自定义变量 a
28     ptr<int> p(&a);                      //自定义指针类对象
29 }
```

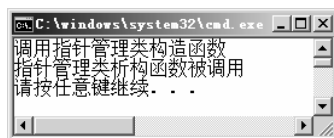


图 10-11 使用类自动管理指针

【代码解析】

第 04~12 行为类模板 `ptr` 的声明，其中第 08 行为成员变量指针 `ip`，第 10、11 行分别为构造和析构函数。第 13~18 行是构造函数的定义体，其中第 16 行对成员变量初始化。第 19~24 行是析构函数的定义体，第 22 行释放指针，并赋值为 `NULL`。在 `main()` 函数中，第 27 行定义整型变量 `a`，用于初始化第 28 行定义的指针变量。



实例 178 自定义 DLL 库导出函数

【实例描述】

本实例的实现平台不是控制台程序，而是 DLL 程序。该动态库导出函数 `Add()` 实现两数相加。编译后，在 Debug 文件下生成 3 个文件，分别为 `178.dll`、`178.lib` 和 `178.exp`。如果编译链接成功，会生成如图 10-12 所示的窗口。

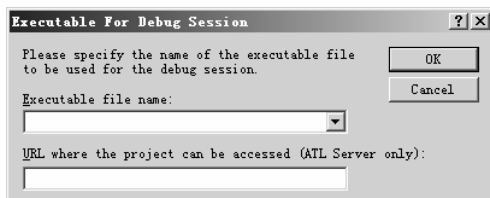


图 10-12 自定义 DLL 库

【实现过程】

定义头文件 `178.h` 和源文件 `178.cpp`，头文件定义 DLL 导出函数的声明，源文件着重包含导出函数的定义体。下面分别列出这两个文件的代码。

(1) 178.h

```
01 #ifdef ADD_API
02 #else
03 #define ADD_API __declspec(dllimport)           //导入
04 #endif
05
06 ADD_API double Add(double a,double b);        //导出加法函数
```

(2) 178.cpp

```
01 #define ADD_API __declspec(dllexport)          //导出
02 #include "178.h"
03
04 double Add(double a,double b)
05 {
06     return a+b;                                //返回和
07 }
```

【代码解析】

(1) 178.h

第 01~04 行是预编译指令，它的格式如下：

```
#ifdef 标识符
命令 1
#else
命令 2
#endif
```

第 06 行把 `Add()` 函数导出。

(2) 178.cpp

第 01 行定义导出命令，第 02 行包含头文件。第 04~07 行为函数 `Add()` 的定义体。



注意:在效果图的 Executable file name 中调用导出函数 Add()的可执行文件路径输入,即可运行。



实例 179 调用 DLL 导出函数

【实例描述】

在有些程序应用中,需要调用外部 lib 文件。本实例以调用实例 178 的导出函数 Add()为例说明问题。先将 178.lib 和 178.dll 文件复制到当前路径下,然后调用函数编写程序,运行效果如图 10-13 所示。

【实现过程】

定义 double 型变量 res 用以保存加法结果,然后输出加法结果,代码如下:

```
01 #pragma comment(lib,"178.lib")           //包含 lib 文件
02 #include "178.h"
03 #include<iostream>
04 using namespace std;
05
06 void main()
07 {
08     double res;                           //结果变量
09     res=Add(10,2.3);                      //调用导出函数
10     cout<<"调用动态链接库,两数和:"<<res<<endl; //输出
11 }
```

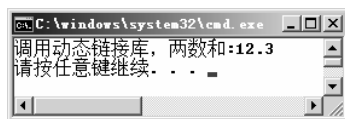


图 10-13 调用 DLL 导出函数

【代码解析】

第 01 行包含 lib 文件 178.lib,第 02 行包含头文件 178.h,第 08 行定义变量 res,第 09 行调用 DLL 导出函数 Add(),第 10 行输出加法结果。



注意:如果不在工程属性中设置,必须包含代码第 01 行。另外,由于编译类型有 debug 和 release 之分。所以,lib 文件包含写法规范如下:

```
#ifdef _DEBUG
#pragma comment(lib,"178.lib")
#else
#pragma comment(lib,"178.lib")
#endif
```



实例 180 释放字符串常量内存错误案例

【实例描述】

对于常量和不同类型的变量,它们存储在不同的内存区中。局部变量存储在栈中,它随着作用域的结束而被自动释放。全局变量和静态变量存储在静态存储区中,一经创建,则一直存



在，直到程序结束。动态申请的内存放在堆中，需要手动释放。对于字符串常量也被存放于静态存储区中，本实例模拟释放字符串常量内存导致的错误，效果如图 10-14 所示。

【实现过程】

定义 3 个函数 `returnstr1()`、`returnstr2()` 和 `returnstr3()`，分别返回指针内容、栈内容和静态存储区内容。字符串常量的表示为 `Hello World`，将其赋值给上述 3 个内存区。在超出作用域时输出其内容，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 char *returnstr1()
05 {
06     char *ch="hello world";           //指针
07     return ch;
08 }
09 char *returnstr2()
10 {
11     char ch[]="hello world";           //字符数组，存于栈中
12     return ch;
13 }
14 char *returnstr3()
15 {
16     static char ch[]="hello world";    //静态变量，存于静态存储区
17     return ch;
18 }
19 void main()
20 {
21     char *str=NULL;                    //初始化为 NULL
22     str=returnstr1();                  //指针
23     cout<<"指针指向内存内容: "<<str<<endl;
24     str=returnstr2();                  //栈内容
25     cout<<"栈内容: "<<str<<endl;
26     str=returnstr3();                  //静态存储
27     cout<<"静态存储区内容: "<<str<<endl;
28 }
```

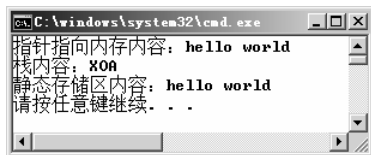


图 10-14 释放字符串常量内存导致的错误

【代码解析】

第 04~08 行返回指针指向内存的首地址，由于静态存储区一直存在，所以它在第 23 行输出的值不变。第 09~13 行返回栈中内容，它是将静态存储区的内容复制一份到栈中。但因为栈内存存在超出作用域时被自动释放，所以在第 25 行读到的栈内容为乱码。第 14~18 行返回静态存储区内容，所以内容不变。



注意：在使用内存的过程中，一定要明确当前的内存属于什么类型。



实例 181 隐式转换错误案例

【实例描述】

在不同类型的变量之间可以进行隐式转换，比如，整型转为浮点型等。但是类型转换最好



采用显式转换，隐式转换可能会给程序带来相当大的损害。比如，将浮点型隐式转换为整型，可能会使除数为 0。程序运行效果如图 10-15 所示。

【实现过程】

本实例先计算 1 除以 3 的商，在程序中，1 和 3 都是浮点型变量。然后将该商赋值给整型变量。最后用整型变量值 10 除以该商，获得最终结果。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     cout<<"现计算 10/(1/3)的结果"<<endl;
07     double i=1,j=3;                //被除数与除数
08     int res1;                       //商 1
09     if(j==0)
10     {
11         cout<<"除数为 0"<<endl;
12         return;
13     }
14     res1=i/j;                       //商 1（隐式转换）
15     int k=10;                       //被除数
16     int res2;                       //商 2
17     if(res1==0)
18     {
19         cout<<"除数为 0"<<endl;
20         return;
21     }
22     res2=k/res1;                    //商 2
23     cout<<res2<<endl;              //输出商 2 结果
24 }
```

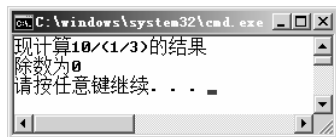


图 10-15 隐式转换错误

【代码解析】

第 07 行定义变量 *i* 和 *j*，它们的值分别为 1 和 3。第 08 行定义 1 除 3 的商变量 *res1*，第 09 行判断除数 *j* 是否为 0。如果为 0，则返回。第 14 行隐式将浮点型商赋给整型变量 *res1*，第 15、16 行分别定义变量 *k* 和变量 *res2*。第 17 行再一次判断除数是否为 0，第 22 行计算最后的结果，第 23 行输出结果。



注意：在涉及求商的程序中，要注意两点，一是除数是否为 0，二是在不同类型的变量相互赋值时，一定要采用显式类型转换。



实例 182 指示灯颜色（static 变量）

【实例描述】

static 变量具有记忆性，即使是局部变量，初始化也只有一次。本实例以交通指示灯的颜色变化（红→黄→绿）规则，演示 *static* 变量的性质，运行效果如图 10-16 所示。

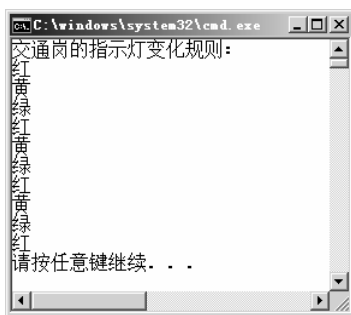


图 10-16 指示灯颜色

【实现过程】

定义函数 `indicator()` 先计数，然后取 3 的余数，判断当前指示的颜色，总共循环 10 次，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void indicator()           //指示灯
05 {
06     static int num=0;      //计数
07     int a=num%3;           //取余
08     switch(a)
09     {
10     case 0:
11         cout<<"红\n";
12         break;
13     case 1:
14         cout<<"黄\n";
15         break;
16     case 2:
17         cout<<"绿\n";
18         break;
19     }
20     num++;                 //计数加 1
21 }
22 void main()
23 {
24     cout<<"交通岗的指示灯变化规则:"<<endl;
25     for(int i=0;i<10;i++)  //循环 10 次
26         indicator();       //指示灯函数
27 }
```

【代码解析】

第 04~21 行为函数 `indicator()` 的定义，其中，第 06 行的计数变量是 `static` 型。它只初始化一次，即当前进入函数 `indicator()` 的值是上一次退出时的值。第 07 行对计数变量 `num` 取 3 的余数，第 08~19 行是 `switch` 判断结构。第 20 行对计数变量做加 1 运算。第 25 行循环调用 `indicator()` 函数 10 次。



注意：如果第 06 行变量 `num` 不是 `static` 型，运行结果显示指示灯颜色全是红色。



实例 183 编写一个堆内存管理类

【实例描述】

使用 new 和 malloc 方法申请的内存被称为堆内存，本实例编写类管理堆内存，效果如图 10-17 所示。

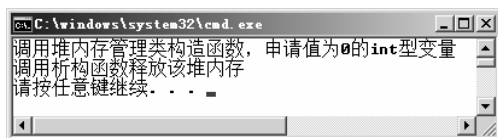


图 10-17 堆内存管理类

【实现过程】

定义类 heap 用于管理堆内存，在构造函数中申请堆内存，在析构函数中释放堆内存，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 class heap
05 {
06 private:
07     int *ip;                                //指针
08 public:
09     heap()
10     {
11         ip=new int(0);                      //初始化
12         cout<<"调用堆内存管理类构造函数, 申请值为 0 的 int 型变量"<<endl;
13     }
14     ~heap()
15     {
16         if(ip!=NULL)
17         {
18             delete ip;                      //释放
19             ip=NULL;
20         }
21         cout<<"调用析构函数释放该堆内存"<<endl;
22     }
23 };
24 void main()
25 {
26     int a=10;                                //自定义变量 a
27     heap p;                                  //自定义指针类对象
28 }
```

【代码解析】

第 07 行为私有成员变量，利用 new 申请一块堆内存。在第 09~13 行的构造函数中，初始化该堆内存。在第 14~22 行的析构函数中，释放该堆内存。



注意：堆内存的使用需要先申请，再释放，申请后一定要初始化。当生命周期结束时，一定要释放。



实例 184 超出作用域错误案例

【实例描述】

作用域限定变量的有效范围，即变量在作用域内被创建和销毁。变量的作用域由其所在最近的一对括号确定，进入该作用域后，变量被创建，当离开时，变量被销毁。如果超出作用域时，依然访问该变量，则会出现如图 10-18 所示的错误。



图 10-18 超出作用域错误使用变量

【实现过程】

定义整型数组 `aa[4]`，对其进行初始化时利用 `for` 循环，其中的索引变量使用局部变量 `i`，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     int aa[4];                //整型数组
07     for(int i=0;i<4;i++)      //变量 i 的作用域
08     {
09         aa[i]=i;
10     }
11     cout<<i<<endl;          //输出当前 i 的值
12 }
```

【代码解析】

第 06 行定义整型数组 `aa`，第 07~10 行使用 `for` 循环对其进行初始化，第 11 行输出当前 `i` 的值。但是变量 `i` 有自己的作用范围，第 11 行已超出作用域，因此，`i` 被认为是没有定义的标识符。所以，将变量 `i` 定义为 `main` 函数全局作用范围内，写法改为如下形式：

```
int i;
for(i=0;i<4;i++)    //改变写法
```



注意：除了变量有作用域，函数也有其作用域。如果一个函数只是在某个作用域中使用，可以被定义为 `static` 函数，它的好处是防止与同名的函数发生冲突。



实例 185 作用域的相互屏蔽例程

【实例描述】

在变量的作用域中分有全局变量和局部变量，使用局部变量时可以屏蔽全局变量。本实例



定义变量名相同的两种变量，经过减 1 运算输出其结果，效果如图 10-19 所示。

【实现过程】

定义命名相同的全局变量和局部变量 a，并分别赋值为 10 和 9.0，然后分别对其做减 1 操作，最后输出各自的当前值，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int a=10;                //全局变量 a
05
06 void main()
07 {
08     float a=9.0;         //局部变量 a
09     a=a-1;               //局部
10     cout<<a<<endl;       //输出局部变量值
11     ::a=:a-1;            //全局变量减 1
12     cout<<::a<<endl;    //输出全局变量值
13 }
```



图 10-19 作用域的相互屏蔽

【代码解析】

第 04 行定义全局变量 a。第 08 行定义局部变量 a。第 09 行对 a 进行减 1 运算，由于作用域的屏蔽作用，只对局部变量有效，所以第 10 行输出 float 型变量 a 的结果。第 11 行明确表明对全局变量进行操作，因此在第 12 行输出全局变量 a 的结果。



注意：作用运算符::可以取消局部变量对全局变量的屏蔽。



实例 186 使用数组名作为函数参数

【实例描述】

数组名也即数组的首地址，本实例模拟使用数组名作为函数的参数。数组包含多个元素，返回最大值，效果如图 10-20 所示。

【实现过程】

定义函数 max()，第 1 个参数为数组，第 2 个参数为数组的大小，返回数组的最大元素值。定义整型数组 aa[10]，并进行初始化，最后输出最大的元素值，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int max(int a[],int num)
05 {
06     int max=0;                //最大值
07     for(int i=0;i<num;i++)
08         for(int j=i+1;j<num;j++)
09             if(a[i]<a[j])
```

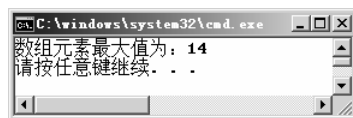


图 10-20 使用数组名作为函数参数



```

10             max=a[j];           //最大值
11     return max;                 //返回
12 }
13 void main()
14 {
15     int aa[10];                 //数组元素
16     for(int i=0;i<10;i++)
17         aa[i]=i+5;             //初始化
18     cout<<"数组元素最大值为: "<<max(aa,10)<<endl;
19 }

```

【代码解析】

第 04~12 行为函数 max() 的定义体, 第 06 行定义变量 max 表示最大值, 第 07~10 行获取数组的最大值元素。它的第 1 个参数为数组名, 可作为首地址使用。



注意: 数组名可作为指针使用, 所以第 04 行还可以写为: int max(int *a, int num)。



实例 187 让函数一次返回多个值

【实例描述】

通常情况下, 函数都只返回一个值, 本实例实现让函数一次性返回多个值, 可以由返回地址实现, 效果如图 10-21 所示。

【实现过程】

定义有多个返回值的函数 max_min(), 它的返回值为数组的首地址, 包含数组的最大值和最小值。在 main() 函数中定义数组 aa[10], 并进行初始化, 最后输出它的最大值和最小值, 代码如下:

```

01 #include <iostream>
02 using namespace std;
03
04 int* max_min(int *a,int num)
05 {
06     int min_max[2];           //最大值和最小值
07     min_max[0]=min_max[1]=a[0]; //赋值第一个元素
08     for(int i=0;i<num;i++)
09     {
10         if(a[i]<min_max[1])      //如果小于最小值
11         {
12             min_max[1]=a[i];    //最小值
13         }
14         if(a[i]>min_max[0])      //大于最大值
15         {
16             min_max[0]=a[i];    //最大值
17         }
18     }
19     return min_max;           //返回
20 }
21 void main()

```

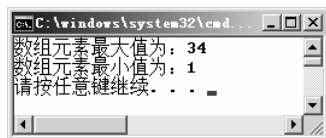


图 10-21 函数一次性返回多个值



```

22 {
23     int aa[10]={34,12,24,1,1,4,6,7,8,9};    //数组元素
24     cout<<"数组元素最大值为: "<<max_min(aa,10)[0]<<endl;
25     cout<<"数组元素最小值为: "<<max_min(aa,10)[1]<<endl;
26 }

```

【代码解析】

第 04~20 行为函数 max_min 的定义体,数组的 10 个元素在循环过程中,获取其最大与最小值,如第 10~17 行的判断。其中,数组 min_max 的第 0 个元素为最大值,第 1 个元素为最小值。



注意: 运用数组除了可以返回多个值,还可以返回结构体。



实例 188 数组错用 sizeof 案例

【实例描述】

对于单个变量运用 sizeof 可以获取其所占内存字节的大小,但用在数组上,返回值是所有的数组元素所占内存的字节大小,即数组的大小×每个元素所占内存的字节数。本实例利用 sizeof 访问数组元素发生越界,效果如图 10-22 所示。

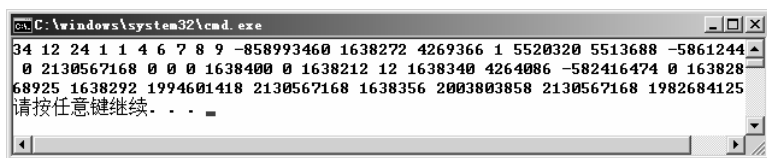


图 10-22 数组错用 sizeof 访问元素

【实现过程】

定义整型数组 aa[10],并进行初始化。利用 sizeof 访问数组的各个元素,代码如下:

```

01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     int aa[10]={34,12,24,1,1,4,6,7,8,9};    //数组元素
07     for(int i=0;i<sizeof(aa);i++)
08         cout<<aa[i]<<" ";                //输出每个元素值
09     cout<<endl;
10 }

```

【代码解析】

第 06 行定义数组 aa[10],第 07、08 行访问各元素,并输出。由效果图可以知,访问越界,原因在于 sizeof(aa)并不是数组的大小。如果要表示大小,必须表达为下列代码:

```
for(int i=0;i<sizeof(aa)/sizeof(int);i++)
```



注意: 数组的字节数是其单个元素字节数的整数倍。



实例 189 类型改名——使用 typedef 定义类型

【实例描述】

在 C++ 语言中, 每个变量都有各自的关键字表示, 比如, 基本整型变量的类型为 int。事实上, 可以使用 typedef 重命名 int。本实例定义重命名的 int 型变量, 并定义实例 188 的数组, 然后输出各值, 效果如图 10-23 所示。

【实现过程】

重命名 int 类型为 ZHENG, 并输出类型为 ZHENG 的数组 aa 的各元素值, 代码如下:

```
01 #include <iostream>
02 using namespace std;
03
04 typedef int ZHENG;           //重命名
05
06 void main()
07 {
08     ZHENG aa[10]={34,12,24,1,1,4,6,7,8,9}; //数组元素
09     for(int i=0;i<sizeof(aa)/sizeof(int);i++)
10         cout<<aa[i]<<" ";           //输出每个元素值
11     cout<<endl;
12 }
```

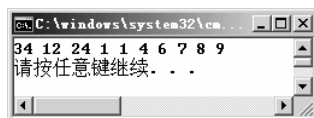


图 10-23 使用 typedef 改类型

【代码解析】

本实例重点的内容为第 04 行, 对 int 类型重命名。第 08 行定义 ZHENG 型数组 aa, 第 09、10 行输出各元素值。



注意: typedef 可以对任意类型的数据进行改名, 在使用过程中, 其不能与 auto、extern、mutable 等关键字出现在同一个表达式中。具体情况请查看相关资料。



实例 190 错误检查——使用 assert 宏进行检测

【实例描述】

assert 宏用于调试程序时, 若出现不应该发生的情况, 程序就会被中止。assert 宏中有一个条件, 当条件为假时, 程序出现错误, 并立即中止。本实例模拟用 assert 宏检测除数是否为 0, 当除数为 0 时, 它输出如图 10-24 所示的提示。



图 10-24 使用 assert 宏检查错误



【实现过程】

定义整型变量 `a` 和 `b`，分别代表被除数和除数。利用 `assert` 宏检查除数是否为 0，代码如下：

```
01 #include <iostream>
02 #include <assert.h>
03 using namespace std;
04
05 void main()
06 {
07     int a,b; //被除数和除数
08     cout<<"请输入被除数: ";
09     cin>>a; //被除数
10     cout<<"请输入除数: ";
11     cin>>b; //除数
12     assert(b!=0); //检测除数是否为 0，如果为 0，则中止程序
13     cout<<"商为: "<<a/b<<endl;
14 }
```

【代码解析】

第 07~11 行定义变量及获取其值，第 12 行用 `assert` 宏检查除数是否为 0。



注意：在使用 `assert` 宏时，需要包含头文件 `assert.h`。



实例 191 使用 `exit()` 函数结束程序

【实例描述】

使用 `exit()` 函数可以直接退出程序。本实例模拟三局两胜的游戏，如果有一方在前两局都输了或者都赢了，就没必要再进行第三局，直接退出即可，效果如图 10-25 所示。

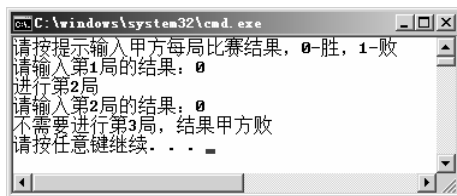


图 10-25 使用 `exit()` 函数结束程序

【实现过程】

定义 3 个整型变量 `flag1`、`flag2` 和 `flag3`，分别表示三局比赛的甲方结果，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     int flag1,flag2,flag3; //败或胜
07     cout<<"请按提示输入甲方每局比赛结果，0-胜，1-败"<<endl;
08     cout<<"请输入第 1 局的结果: ";
09     cin>>flag1;
```



```

10      cout<<"进行第 2 局"<<endl;
11      cout<<"请输入第 2 局的结果: ";
12      cin>>flag2;
13      if(flag1==flag2)                                //两局比赛结果一样
14      {
15          if(flag1=0)                                    //胜
16              cout<<"不需要进行第 3 局, 结果甲方胜"<<endl;
17          else                                           //败
18              cout<<"不需要进行第 3 局, 结果甲方败"<<endl;
19          exit(0);                                       //直接退出程序
20      }
21      else
22      {
23          cout<<"进行第 3 局"<<endl;
24          cout<<"请输入第 3 局的结果: ";
25          cin>>flag3;                                    //输入第 3 局结果
26      }
27  }

```

【代码解析】

第 06 行定义 3 局比赛的甲方胜败结果, 第 09、12、25 行获取 3 局结果。第 13 行判断前两局的结果是否一样, 如果一样, 表示不需要进行最后一局, 直接依照前面的结果判断胜或败, 然后退出程序, 如第 19 行所示, 否则还需要进行下一局。



注意: exit()函数的使用与循环中 break 的使用一样, 当条件满足时, 跳出该代码区。



实例 192 程序异常退出(使用 abort()函数进行异常退出)

【实例描述】

在执行程序的过程中, 可能会出现异常错误, 此时可调用 abort()函数进行异常退出。但是它的调用不同于 exit()函数, 它只是使程序异常终止, 并不会对程序使用的内存块做清除。本实例在实例 191 的基础上替换其函数名, 当运行到 abort()函数时, 弹出如图 10-26 所示的 Bug。

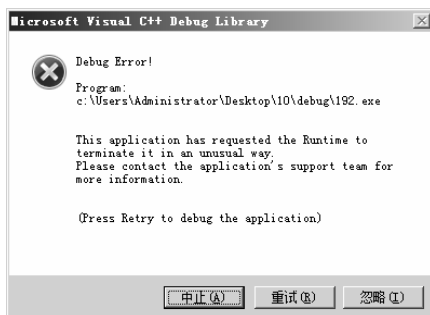


图 10-25 使用 abort()函数异常退出

【实现过程】

在实例 191 代码的基础上, 替换其第 19 行的函数为 abort(), 代码如下:



```

01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     ...
07     abort();    //异常退出程序
08     ...
09 }

```

【代码解析】

第 07 行为 abort()函数的调用。



注意：与 abort()函数相反，exit()函数是在完成清理工作后，再退出程序。



实例 193 自定义异常对象

【实例描述】

C++标准库定义了 8 个异常类，包括对语法错误及运行错误处理的异常。本实例针对除数为 0 由 throw 抛出异常，再由 catch 捕获，效果如图 10-27 所示。

【实现过程】

定义函数 divi()，用来计算两个数的除数。其中，第 1 个参数是被除数，第 2 个参数是除数。如果除数为 0，抛出异常。在 main()函数中，用 try...catch() 结构定义并捕获异常。代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 double divi(double a,double b)
05 {
06     if(b==0)                //除数为 0
07         throw b;            //抛出异常
08     return a/b;              //返回值
09 }
10 void main()
11 {
12     try                      //定义异常
13     {
14         cout<<"2 除以 3 的值为: "<<divi (2,3)<<endl;    //除数不为 0
15         cout<<"2 除以 0 的值为: ";                    //除数为 0
16         cout<<divi (2,0)<<endl;
17     }
18     catch(double)            //捕获异常
19     {
20         cerr<<"错误: 除数为 0"<<endl;                //输出错误提示
21         exit(-1);                                         //强制退出
22     }
23 }

```



图 10-27 自定义异常对象



【代码解析】

第 04~09 行为函数 `divi()` 的定义体, 其中第 06 行判断如果除数为 0, 抛出异常 (第 07 行), 否则, 将除法结果返回。在 `main()` 中, 第 12 行定义异常。在其结构中 (第 13~17 行), 定义可能出现异常的代码。第 18~22 行的 `catch()` 结构捕获针对 `double` 类型的异常, 输出异常信息 (第 20 行) 和强制退出程序 (第 21 行)。



注意: 在 `catch()` 的圆括号中定义捕获异常的类型, 如果括号内是 3 个点 (如 `catch(...)`), 表示可以捕获所有类型的异常。



实例 194 使用 `set_terminate()` 函数设置 `terminate()` 函数指针

【实例描述】

在 C++ 语言中, 当出现异常时, 可以利用 `set_terminate()` 函数设置 `terminate()` 函数指针退出程序。本实例在实例 193 的基础上, 当捕获到异常时, 调用 `set_terminate()` 函数, 效果如图 10-28 所示。



【实现过程】

定义函数 `term_func()`, 输出该函数被 `terminate()` 调用。当除法为 0 时, 响应该函数。代码如下:

图 10-28 设置 `terminate()` 函数指针

```
01 #include <iostream>
02 using namespace std;
03
04 double divi(double a, double b)
05 {
06     ...
07 }
08 void term_func()
09 {
10     cout << "函数 term_func() 被 terminate() 调用.\n";
11     exit(-1); // 强制退出
12 }
13 void main()
14 {
15     set_terminate(term_func); // 设置函数指针
16     divi(10, 0); // 调用除法函数
17 }
```

【代码解析】

第 04~07 行是函数 `divi()` 的定义体, 第 08~12 行是函数 `term_func()` 的定义体, 第 15 行设置 `terminate()` 函数指针, 第 16 行使 `divi()` 函数的除数为 0。由此抛出异常, 调用 `terminate()`。



注意: 如果第 11 行不强制退出程序, 将会调用 `abort()` 函数异常退出。



实例 195 auto_ptr 类智能指针

【实例描述】

auto_ptr 属于智能指针的一种,它可以通过析构函数释放其管理的内存。与其他指针不同,auto_ptr 指针在被复制或者赋值后,复制或赋值对象就成为一个 NULL 指针,运行效果如图 10-29 所示。

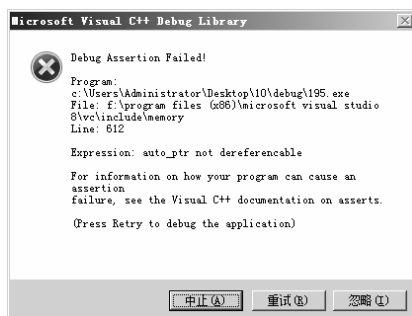


图 10-29 auto_ptr 类智能指针

【实现过程】

定义使用 new 方式申请的 double 型堆内存 p, 初始化为 0.5。定义智能指针 ap1 和 ap2, ap1 初始化为 p, ap2 被 ap1 赋值。代码如下:

```
01 #include <iostream>
02 #include <memory>
03 using namespace std;
04
05 void main()
06 {
07     double *p=new double(0.5);    //申请堆内存
08     auto_ptr<double> ap1(p);       //指针 1
09     auto_ptr<double> ap2=ap1;      //指针 2
10     cout<<*ap2<<" "<<*ap1<<endl; //输出结果
11 }
```

【代码解析】

第 07 行用 new 方式申请堆内存 p, 第 08、09 行定义智能指针对象 ap1 和 ap2。第 10 行输出指针对象的内容。由于 ap2 被 ap1 赋值,造成 ap1 对象已被释放。所以程序运行有 bug,弹出如图 10-29 所示的对话框。



注意: 使用智能指针必须包含头文件 memory.h。



实例 196 auto_ptr 智能指针指向的内存类型

【实例描述】

由于 auto_ptr 智能指针指向的内存类型是由 new 创建的堆内存,所以当指向其他类型的内



存时，将会出现如图 10-30 所示的错误。

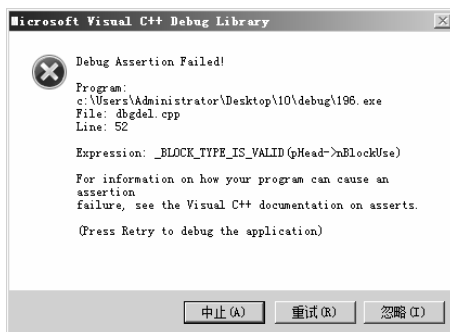


图 10-30 auto_ptr 指针指向内存类型错误

【实现过程】

定义 double 型变量 a，初始化为 0.5。double 型指针变量 p，取址于 a。智能型指针 ap，初始化为 p。最后输出 ap 指向内存的值，代码如下：

```
01 #include <iostream>
02 #include <memory>
03 using namespace std;
04
05 void main()
06 {
07     double a=0.5;           //定义变量
08     double *p=&a;           //申请指针
09     auto_ptr<double> ap(p);  //智能指针
10     cout<<*ap<<endl;       //输出结果
11 }
```

【代码解析】

第 07~09 行分别定义变量 a、对应类型指针 p 和智能指针 ap。第 10 行输出智能指针 ap 指向内存的内容。



注意：auto_ptr 智能指针具有对象所有权独有的性质，所以一块堆内存由一个 auto_ptr 指针所有。

第 3 篇 C++高级案例

第 11 章 C++高级应用例程

本章属于 C++高级案例的第一章，利用 C++高级 API 函数实现较复杂的功能。本章所实现的案例都是在控制台环境下实现的，所实现的功能有定时器的应用、屏幕时钟、七彩文字、数据加密和解密等。



实例 197 用 C++实现定时器功能

【实例描述】

本实例依然在控制台环境下实现，定时器可以使用 windows.h 文件中的 SetTimer()函数，实现的功能是在一定时间内发送信息，输出字符串 Hello World，效果如图 11-1 所示。

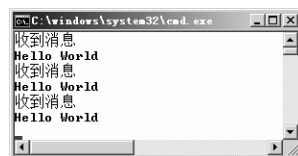


图 11-1 定时器

【实现过程】

定义回调函数 TimerProc()，即当响应时，回调函数 TimerProc()，执行函数内的代码。定义定时器 ID（初始化为 1），消息响应间隔变量 elapse（初始化为 1000ms），调用函数 SetTimer()，代码如下：

```
01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 VOID CALLBACK TimerProc(HWND hwnd, UINT uMsg, UINT idEvent, DWORD dwTime)
                                //回调函数
06
07 {
08     cout<<"Hello World"<<endl;
09 }
10 void main()
11 {
12     int ID=1;                  //定时器 ID
13     int elapse=1000;           //每隔多长时间响应，单位为毫秒
14     MSG msg;                   //消息
15     SetTimer(NULL, ID, elapse, TimerProc); //定时器函数
16     while (GetMessage(&msg, NULL, NULL, NULL) !=0 &&
17           GetMessage(&msg, NULL, NULL, NULL) !=-1) //获取的消息不是退出，也不是错误
18     {
19         if (msg.message==WM_TIMER) //如果收到的信息为定时器
20         {
21             cout<<"收到消息"<<endl; //打印结果
22             TranslateMessage(&msg); //寄送消息到消息队列
23             DispatchMessage(&msg); //发送消息到窗口程序
```



```

24         }
25     }
26 }

```

【代码解析】

第 05~09 行为回调函数 TimerProc() 的定义体, 其中只有函数名和函数内容可以改变。第 12、13 行是定时器 ID 和定时器响应间隔变量的定义及初始化。第 14 行定义消息变量, 第 16、17 行判断所收到的消息是否为退出或错误类型。如果不是, 继续判断消息是否为定时器消息。如果是定时器消息, 则执行第 21~23 行, 即打印结果、寄送消息到消息队列、发送消息到窗口程序。



注意: 如果只有第 22 行, 程序只输出第 21 行, 并不调用函数 TimerProc(), 因为它只是将消息放到了消息队列。



实例 198 使用 atoi() 函数把字符串转换为整数

【实例描述】

在 C++ 高级编程案例中, 通常要将字符串转换为整数, 此时可以调用函数 atoi(), 其调用格式如下:

```
int 型变量=atoi(char *变量名);
```

另外, 如果将字符串变为整型, 则更方便数据的处理, 转换效果如图 11-2 所示。

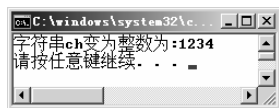


图 11-2 atoi() 函数变字符串为整数

【实现过程】

定义整型变量 a, 初始化为 0; 定义字符串 ch, 初始化为 1234。调用 atoi() 函数将 ch 变换为 a 类型, 代码如下:

```

01 #include <iostream>
02 #include <stdlib.h>
03 using namespace std;
04
05 void main()
06 {
07     int a=0;                //整型变量
08     char *ch="1234";        //字符串
09     a=atoi(ch);           //字符串转换为整型
10     cout<<"字符串 ch 变为整数为:"<<a<<endl;    //输出整数
11 }

```

【代码解析】

第 07、08 行定义变量 a 和 ch, 第 09 行调用函数 atoi(), 并由第 10 行输出整数结果。



注意: atoi() 函数与之后使用的 itoa() 函数在 C++ 高级编程中经常被用到, 一定要记住。



实例 199 使用 itoa()函数把整数转换为字符串

【实例描述】

整数转换为字符串时，可以调用 itoa()函数。本实例模拟整数转换为字符串，运行效果如图 11-3 所示。

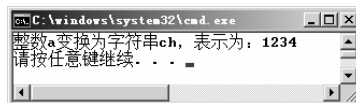


图 11-3 itoa()函数变换整数为字符串

【实现过程】

定义 int 型变量 a，并初始化为 1234；定义 char 型数组 ch，对其转换代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     int a=1234;                //整数
07     char ch[10];              //字符数组
08     itoa(a,ch,10);            //转换整数为字符串
09     cout<<"整数 a 变换为字符串 ch, 表示为: "<<ch<<endl;
10 }
```

【代码解析】

第 06、07 行定义 int 型变量 a 和 char 型数组 ch，第 08 行将整数转换为字符串，第 09 行输出结果。



实例 200 编写一个屏幕小时钟程序

【实例描述】

本实例在实例 124 的基础上，实现屏幕显示时钟功能。即每隔 0.5 秒更新屏幕时间，依然用到回调函数、定时器函数及系统时间获取函数。运行效果如图 11-4 所示。

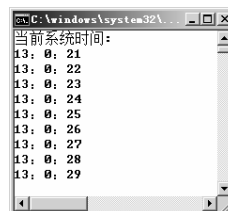


图 11-4 屏幕小时钟

【实现过程】

利用实例 124 编写的获取系统时间类 TIME，定义回调函数 TimerProc()，设置定时器每隔 0.5 秒发送消息执行函数，代码如下：

```
01 #include <sys/timeb.h>
02 #include <time.h>
03 #include <windows.h>
04 #include <iostream>
05 using namespace std;
06
07 class TIME
08 {
09 ...
10 };
11 VOID CALLBACK TimerProc(HWND hwnd, UINT uMsg, UINT idEvent,DWORD dwTime)
//回调函数
```



```

12
13 {
14     TIME t;                                //时间对象
15     t.cal();                               //计算当时时间
16     cout<<t.getH()<<"<<t.getM()<<"<<t.getS()<<endl;    //输出
17 }
18 void main()
19 {
20     ...
21     int elapse=500;                        //每隔多长时间，单位为毫秒
22     ...
23     cout<<"当前系统时间:"<<endl;
24     while(...)                             //获取消息非退出非错误
25     {
26         if(msg.message==WM_TIMER)          //如果收到的信息为定时
27         {
28             TranslateMessage(&msg);        //寄送消息到消息队列
29             DispatchMessage(&msg);        //发送消息到窗口程序
30         }
31     }
32 }

```

【代码解析】

第 07~10 行为类 TIME，第 11~17 行为回调函数 TimerProc()。其中，在第 14 行定义 TIME 类对象 t，第 15 行计算当时的时间，第 16 行输出当前时间。第 21 行设置时间间隔为 0.5 秒，第 23 行输出说明语，第 28、29 行发送消息。



注意：理论上应该每隔 1 秒输出时间，定时器的间隔时间也是 1 秒。但是代码执行过程中也需要时间，所以试验表明 0.5 秒最好。



实例 201 使用 system()函数使屏幕停止

【实例描述】

本实例模拟如何在调试的情况下让屏幕停止，可以使用 system()函数实现。在调试的情况下，如果程序中没有断点，程序只执行，但不给用户时间观察。system()函数中传入参数 pause 可以冻结屏幕，使其只显示调用该函数之前的内容，运行效果如图 11-5 所示。



图 11-5 使用 system()函数使屏幕停止

【实现过程】

在调用 system()函数之前，输出“Hi,你好”，在其后输出 Hello，代码如下：

```

01 #include <iostream>
02 using namespace std;
03
04 void main()

```



```
05 {  
06     cout<<"Hi,你好"<<endl;  
07     system("pause");           //让屏幕停止  
08     cout<<"hello"<<endl;  
09 }
```

【代码解析】

由效果图可看到，它只显示 `system()` 函数之前的 `cout` 语句。



实例 202 屏幕变色效果——使用 `system()` 函数 改变屏幕颜色

【实例描述】

使用 `system()` 函数，当输入参数不同时，可以改变屏幕颜色，其格式如下：

`system("color 背景色代号前景色代号")`；

本实例模拟将屏幕背景色变为黑色，前景色变为淡紫色，其运行效果如图 11-6 所示。



图 11-6 使用 `system()` 函数
改变屏幕颜色

【实现过程】

输出文字 `Hi,你好`，并改变屏幕颜色，代码如下：

```
01 #include <iostream>  
02 using namespace std;  
03  
04 void main()  
05 {  
06     cout<<"Hi,你好"<<endl;  
07     system("color 0D");           //改变屏幕颜色  
08 }
```

【代码解析】

本实例的重点代码为第 07 行，即改变屏幕颜色，其中 0 代表黑色，D 代表淡紫色。



注意：`system()` 函数因输入参数的不同，具有不同的功能。



实例 203 清空屏幕——清屏的实现

【实例描述】

使用 `system()` 函数还可以实现清屏功能，即输入参数为 `cls`。首先输出 `Hello World`，然后清空屏幕文字，接着输出 `I love C++`，效果如图 11-7 所示。



图 11-7 清屏实现

【实现过程】

定义整型变量 `clear`，表示是否清屏。如果值为 0，表示清屏，如果为 1，不清屏。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     int clear;                //是否清屏
07     cout<<"Hello World"<<endl;
08     cout<<"是否清屏? 0-是, 1-否"<<endl;
09     cin>>clear;
10     if(clear==0)              //清屏
11         system("cls");
12     else if(clear==1)         //不清屏
13     {}
14     cout<<"I love C++"<<endl;
15 }
```

【代码解析】

第 06 行定义变量 `clear`，判断是否清屏。第 09 行获取清屏变量的值，第 10~13 行的 `if...else if` 结构判断是否进行清屏操作，第 14 行输出另一条语句。



注意：`cls` 的使用相当于 `matlab` 中的 `clr` 应用，主要是清除屏幕中太多的内容。



实例 204 七彩文字——改变文字色

【实例描述】

本实例演示如何改变文字颜色，屏幕颜色为代号 7，输出文字“你好”，用淡红色显示，运行效果如图 11-8 所示。

【实现过程】

定义字符串 `a`，赋值为“你好”。调用函数 `system()`，传入参数 `color 7C`，然后输出字符串 `a`。代码如下：

```
01 #include <iostream>
02 #include <string>
03 #include <windows.h>
04 using namespace std;
05
06 void main()
```



图 11-8 七彩文字



```
07 {  
08     string a("你好");           //文字  
09     system("color 7C");          //设置文字颜色  
10     cout<<a<<endl;             //输出  
11 }
```

【代码解析】

本实例代码较简单，其中第 08 行定义字符串 a，第 09 行执行函数 system()，第 10 行输出字符串 a。



实例 205 屏幕背景闪动效果的实现

【实例描述】

本实例在改变屏幕文字颜色的基础上，实现屏幕背景的闪动效果。实现原理是每隔一段时间改变屏幕的背景色，效果如图 11-9 所示。

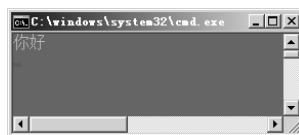


图 11-9 屏幕背景闪动效果

【实现过程】

在无限循环中，先定义 string 型变量 str，初始化为 color 。static 型整型变量 n，用于存储背景代号。char 型数组（长度为 1）存储转换后的背景代号，每隔 0.8 秒更换一次背景色。代码如下：

```
01 #include <iostream>  
02 #include <string>  
03 #include <windows.h>  
04 using namespace std;  
05  
06 void main()  
07 {  
08     cout<<"你好\n";  
09     while(1)                               //无限循环  
10     {  
11         string str="color ";              //母串  
12         static int n=0;                    //背景代号  
13         char s[1];                        //字符  
14         if(n>9)                           //如果超出背景代号 9  
15             n=0;                          //返回最初的 0 值  
16         itoa(n,s,10);                     //转换为字符  
17         str+=s;                            //连接  
18         str+='A';                          //连接前景代号  
19         system(str.c_str());               //转换类型并执行  
20         Sleep(800);                        //睡觉  
21         n++;                              //背景代号加 1  
22     }  
23 }
```

【代码解析】

第 08 行输出文字，第 09~22 行为无限循环。其中，第 11~13 行声明所需变量，第 14 行



判断背景代号是否大于 9。如果大于 9，返回最初的 0 值。然后将整型数据转换为 char 型变量，如第 16 行。第 17、18 行完成字符串的连接，作为参数传入函数 system()，如第 19 行。第 20 行睡觉 0.8 秒，第 21 行使背景代号加 1，实现屏幕的闪动效果。



注意：本实例只实现背景色从 0 到 9。此外，还可以扩展到代号 F。



实例 206 文字闪动效果的实现

【实例描述】

本实例在改变屏幕文字颜色的基础上，实现文字闪动效果。通过每隔 0.8 秒改变文字的颜色达到该效果，效果如图 11-10 所示。

【实现过程】

构建无限循环，在此循环中定义 system() 函数接收的参数字符串。通过改变变量 s 的值，实现文字颜色的改变，代码如下：

```
01 #include <iostream>
02 #include <string>
03 #include <windows.h>
04 using namespace std;
05
06 void main()
07 {
08     cout<<"你好\n";
09     while(1)                //无限循环
10     {
11         string str="color 7"; //母串
12         static char s='A';    //最后的字符
13         if(s>'F')             //如果超出文字色
14             s='A';            //返回最初值
15         str+=s;                //连接
16         system(str.c_str());   //转换类型并执行
17         Sleep(800);           //睡觉
18         s++;                   //文字编号加 1
19     }
20 }
```

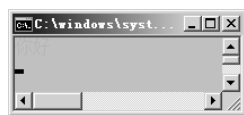


图 11-10 文字闪动效果

【代码解析】

第 09 行为无限循环，第 11、12 行为 system() 函数接收参数的母串和最后的字符。第 13、14 行判断如果字符超过文字颜色代号的值，则返回最小值。第 15 行实现 str 和 s 的连接，第 16 行调用函数 system()，第 17 行执行睡觉指令。然后在第 18 行对字符进行加 1 操作，继续执行循环体内的代码。



注意：第 12 行要将 s 定义为 static 类型，为使下一次执行不被初始化。



实例 207 定时关机

【实例描述】

调用 `system()`除了可以改变屏幕的前景色和背景色，还可以设定时间实现关机，它的实现原理是调用 DOS 命令。本实例在关机前 1 分钟给出提示，效果如图 11-11 所示。

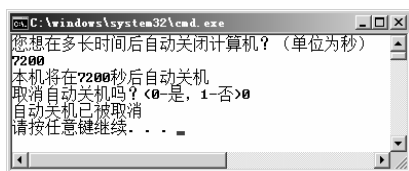


图 11-11 定时关机提示

【实现过程】

定义关机母串变量 `str`，赋值为 “`shutdown -s -t` ”。变量 `n` 表示关机时间，单位为秒。变量 `s` 用于存储关机时间变量 `n` 的转换值。变量 `no` 用于标识是否取消关机指令。代码如下：

```
01 #include <iostream>
02 #include <string>
03 #include <windows.h>
04 using namespace std;
05
06 void main()
07 {
08     string str="shutdown -s -t ";           //母串
09     int n=0;                               //关机时间
10     char s[10];                            //字符串
11     int no=0;                              //取消关机否
12     cout<<"您想在多长时间后自动关闭计算机？（单位为秒）"<<endl;
13     cin>>n;                                //输入
14     itoa(n,s,10);                          //转换为字符串
15     str+=s;                                //连接
16     system(str.c_str());                   //关机指令
17     cout<<"本机将在"<<n<<"秒后自动关机"<<endl;
18     cout<<"取消自动关机吗？(0-是，1-否) ";
19     cin>>no;
20     if(no==0)                              //取消
21     {
22         system("shutdown -a");
23         cout<<"自动关机已被取消"<<endl;
24     }
25     system("pause");
26 }
```

【代码解析】

第 08~11 行是本程序所需的各类变量，第 13 行输入关机时间值。第 14 行将整型变量 `n` 值转换为字符串 `s`，第 15 行将字符串 `s` 连接到母串 `str` 中。第 16 行执行关机指令，第 17 行提示多长时间关机，是否取消关机。第 19 行获取取消关机选择，如果取消关机，则执行第 21~24 行代码。第 22 行的指令为取消关机状态。



注意：除上述指令可以完成关机外，还有指令 shutdown -p 实现立即关机，指令 shutdown -l 实现注销计算机。



实例 208 设置 Win32 窗口

【实例描述】

C++语言除了可以应用于控制台程序，还可以在 Windows 应用程序中发挥作用。控制台程序是传统的 DOS 窗口，Win32 应用程序可以自由设置其显示窗口。本实例提示代码设置方式，效果如图 11-12 所示。



图 11-12 设置 Win32 窗口

【实现过程】

Win32 应用程序的入口函数是 WinMain()，在该函数的入口处先创建窗口类。利用类 WNDCLASS，其中可以设置窗口的背景色、窗口箭头光标类型、窗口响应函数、窗口类名等。代码如下：

```
01 #include <windows.h>
02 #include <stdio.h>
03
04 //回调函数声明
05 LRESULT CALLBACK WinHouProc(
06     HWND hwnd,                //窗口句柄
07     UINT uMsg,                //消息标识符
08     WPARAM wParam,            //第一个消息参数
09     LPARAM lParam              //第二个消息参数
10 );
11 int WINAPI WinMain(
12     HINSTANCE hInstance,        //指向当前实例的句柄
13     HINSTANCE hPrevInstance,    //指向先前实例的句柄
14     LPSTR lpCmdLine,           //命令行
15     int nCmdShow               //显示状态
16 )
17 {
18     WNDCLASS wndclass;          //创建一个窗口类
19     wndclass.cbClsExtra=0;      //窗口类无扩展
20     wndclass.cbWndExtra=0;      //窗口实例无扩展
21     wndclass.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH); //窗口白色
22
23     wndclass.hCursor=LoadCursor(NULL, IDC_HAND); //手型
24     wndclass.hIcon=LoadIcon(NULL, IDI_APPLICATION); //最小化图标为默认图标
25     wndclass.hInstance=hInstance; //当前实例句柄
26     wndclass.lpfnWndProc=WinHouProc; //定义窗口处理函数
27     wndclass.lpszClassName="Win32"; //窗口类名
28     wndclass.lpszMenuName=NULL; //窗口无菜单
29     wndclass.style=CS_HREDRAW | CS_PARENTDC; //设置窗口类型
30     RegisterClass(&wndclass); //注册窗口类
31
32     HWND hwnd;
33     hwnd=CreateWindow("Win32", "Win32 窗口", WS_OVERLAPPEDWINDOW,
```



```

34         100, 50, 600, 400, NULL, NULL, hInstance, NULL); //创建窗口
35
36     ShowWindow(hwnd, SW_SHOWNORMAL); //显示窗口
37     UpdateWindow(hwnd); //更新窗口
38
39     MSG msg;
40     while (GetMessage(&msg, NULL, 0, 0)) //获取消息
41     {
42         TranslateMessage(&msg); //发送消息到消息队列
43         DispatchMessage(&msg); //发送消息到窗口程序
44     }
45     return 0;
46 }
47 LRESULT CALLBACK WinHouProc(
48     HWND hwnd, //窗口句柄
49     UINT uMsg, //消息标识符
50     WPARAM wParam, //第一个消息参数
51     LPARAM lParam //第二个消息参数
52 )
53 {
54     switch(uMsg)
55     {
56     case WM_PAINT: //绘图
57         HDC hDC;
58         PAINTSTRUCT ps;
59         hDC=BeginPaint(hwnd, &ps); //获取绘图所用句柄
60         TextOut(hDC, 20, 10, "Win32 窗口程序", strlen("Win32 窗口程序")); //输出文本
61         EndPaint(hwnd, &ps);
62         break;
63     default:
64         return DefWindowProc(hwnd, uMsg, wParam, lParam); //返回窗口
65     }
66     return 0;
67 }

```

【代码解析】

第 04~10 行是窗口所用回调函数的声明,其定义体在第 47~67 行。第 11~46 行是 WinMain() 函数定义体,其中,第 18 行创建窗口类对象 wndclass。第 19~29 行设置该对象的各个成员变量值。第 21 行设置窗口背景为白色,第 23 行将窗口的光标设置为手型,第 24 行载入缺少图标。第 26 行定义窗口函数为 WinHouProc(),第 27 行给窗口类命名为 Win32,第 29 行设置窗口类型。

第 30 行注册该窗口类,之后创建窗口(如第 32~34 行),设置窗口的位置和大小。第 36、37 行显示并更新窗口。第 39~44 行处理消息,其中第 40 行循环获取消息,第 42 行将消息发送到消息队列,第 43 行发送消息到窗口程序。

在窗口响应回调函数 WinHouProc()中,第 54~65 行的 switch 结构处理消息类型。其中,第 56~62 行处理绘图消息,输出文本“Win32 窗口程序”。第 63、64 行处理 default 消息类型,并返回窗口过程继续执行。



注意: 第 42 行是将消息发送到消息队列等待执行,第 43 行是发送消息到窗口程序执行,并显示结果。



实例 209 设计一个动态指令接收程序

【实例描述】

继续实例 208，本实例设计一个动态指令接收程序，其目标是动态接收按键指令和关闭窗口指令，运行效果如图 11-13 所示。



图 11-13 动态指令接收程序

【实现过程】

本实例的改动主要发生在回调函数 WinHouProc() 中，在 switch 中添加按键消息接收和关闭窗口指令接收，其代码如下：

```

01  ...
02  LRESULT CALLBACK WinHouProc(
03      HWND hwnd,                //窗口句柄
04      UINT uMsg,                //消息标识符
05      WPARAM wParam,            //第一个消息参数
06      LPARAM lParam              //第二个消息参数
07  )
08  {
09      switch(uMsg)
10      {
11          case WM_CHAR:           //按键
12              MessageBox(hwnd, "按键指令", "Win32", 0);
13              break;
14          case WM_PAINT:          //绘图
15              ...
16              break;
17          case WM_CLOSE:          //关闭窗口
18              MessageBox(hwnd, "关闭窗口指令", "Win32", 0);
19              DestroyWindow(hwnd); //销毁窗口
20              break;
21          default:
22              return DefWindowProc(hwnd, uMsg, wParam, lParam); //返回窗口过程
23      }
24      return 0;
25  }

```

【代码解析】

在 WinHouProc() 函数中，第 11~13 行为接收按键指令，输出按键消息框（如第 12 行）。第 17~20 行为关闭窗口指令接收，其中第 18 行输出关闭窗口提示，第 19 行销毁当前窗口。



注意：实例 208 虽然将窗口关闭，但并没有将其进程退出。本实例的关闭窗口指令即可将该进程退出。读者可进入 Windows 进程查看。



实例 210 编写指令响应程序

【实例描述】

实例 209 在接收按键指令后并没有明确所按的键值。本实例响应该按键指令，然后输出所按的键值，效果如图 11-14 所示。

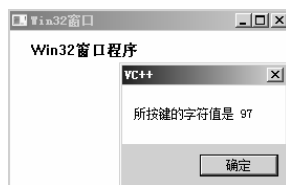


图 11-14 指令响应

【实现过程】

在回调函数 `WinHouProc()` 的 `WM_CHAR` 类型消息下先将 `wParam` 转换为 `char` 型变量，再用消息盒子输出所按键值，代码如下：

```
01  ...
02  LRESULT CALLBACK WinHouProc(
03      HWND hwnd,                      //窗口句柄
04      UINT uMsg,                      //消息标识符
05      WPARAM wParam,                  //第一个消息参数
06      LPARAM lParam                    //第二个消息参数
07  )
08  {
09      switch(uMsg)
10      {
11          case WM_CHAR:                //当按键时，显示所按的键值
12              MessageBox(hwnd, "按键指令", "Win32", 0);
13              char szChar[100];
14              sprintf(szChar, "所按键的字符值是 %d", wParam); //转换按键值类型
15              MessageBox(hwnd, szChar, "VC++", 0);           //显示按键的值
16              break;
17          ...
18      }
19      return 0;
20  }
```

【代码解析】

按键指令响应代码为第 12~16 行，其中第 12 行输出按键类型指令提示盒子，第 13 行声明字符串变量 `szChar`。第 14 行利用函数 `sprintf()` 将 `WPARAM` 类型变量转换为字符串变量 `szChar`，第 15 行以消息盒子显示按键值。



注意：实例 208~实例 210 需要在方案设置中将字符集格式改为多字节，该项位于工程的 `Property`→`General`→`Character Set` 项目中。



实例 211 自定义函数生成一段随机数据

【实例描述】

本实例自定义函数生成一段 0~100 的随机数据，利用以下公式实现：

随机数=最小值+(最大值*`rand()`)/(`RAND_MAX`+最小值)；

实例运行效果如图 11-15 所示。

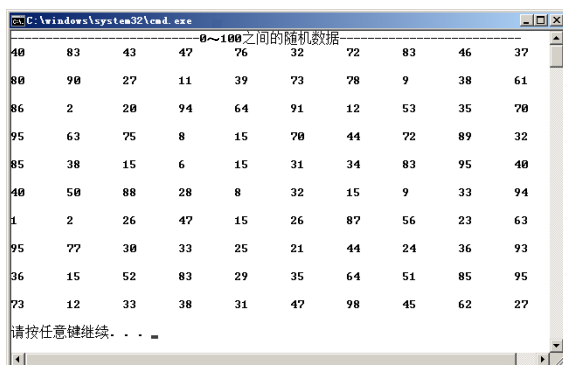


图 11-15 生成随机数据

【实现过程】

定义函数 `gene_rand()` 实现随机数据的生成，具体代码如下：

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <time.h>
04 #include <iostream>
05 using namespace std;
06
07 #define MAX 100 //最大值
08 #define MIN 0 //最小值
09
10 void gene_rand()
11 {
12     int data; //随机数据
13     for(int i=0;i<10;i++)
14     {
15         for(int j=0;j<10;j++)
16         {
17             data=MIN+(int) (MAX*rand() / (RAND_MAX+MIN)); //生成随机数
18             cout<<data<<" ";
19         }
20         cout<<endl;
21     }
22 }
23 void main()
24 {
25     cout<<"-----0~100 之间的随机数据-----"<<endl;
26     srand((int)time(0)); //利用系统时间产生随机序列的种子值
27     gene_rand();
28 }
```

【代码解析】

第 07、08 行是随机数的生成范围 (0~100)，第 10~22 行是生成随机数的函数 `gene_rand()` 的定义体，其中分 10 行 10 列生成随机数。第 17 行是随机数的生成公式，第 18 行输出随机数，第 26 行是产生随机数的种子值。



注意：由该公式生成的随机数比用公式 `rand()%100+1` 生成的数据更随机。



实例 212 一个简单加密算法的实现

【实例描述】

本实例演示一个简单的加密算法，输入一个字符串明文和整型密钥。然后经过加法运算转换为密文字符串输出，运行效果如图 11-16 所示。

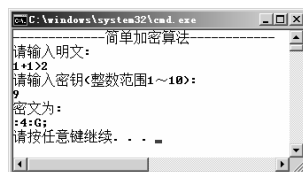


图 11-16 简单加密算法

【实现过程】

定义 `string` 型字符串变量 `src`，整型变量 `key` 和 `num` 分别表示明文、密钥和字符串长度。算法的具体代码如下：

```
01  #include<iostream>
02  #include <string>
03  using namespace std;
04
05  void main()
06  {
07      cout<<"-----简单加密算法-----"<<endl;
08      string src;                //明文
09      int key=0;                 //密钥
10      int num=0;                 //长度
11      cout<<"请输入明文:\n";
12      cin>>src;                  //明文
13      cout<<"请输入密钥(整数范围 1~10):\n";
14      cin>>key;                  //密钥
15
16      num=src.length();          //取字符串长度
17      const char *src_c=new char[num]; //C 风格的字符串
18      char *code=new char[num];      //密文
19
20      src_c=src.c_str();          //转换风格
21      for(int i=0;i<num;i++)      //加密
22          code[i]=src_c[i]+key;
23
24      cout<<"密文为:\n";
25      for(int i=0;i<num;i++)      //输出密文
26          cout<<code[i];
27      cout<<endl;
28  }
```

【代码解析】

第 07 行输出提示语句，第 08~10 行定义所需的 3 个变量。第 12 行获取明文，第 14 行获取密钥。获取字符串长度的目标是动态申请 C 风格字符串的大小和密文大小，如第 17、18 行。第 20 行将 `string` 字符串转换为 C 风格的字符串，第 21、22 行实现加密，第 25、26 行输出密文。



注意：第 14 行所示的整数范围列出 1~10，是一个相对安全的加密范围（具体原因请查看 ASCII 码表）。



实例 213 解密算法的实现

【实例描述】

继续实例 212，加密后的密文是将明文的各字符加了一个整数。解密算法的实现原理是做逆向运算，将密文的每个字符减去密钥。运行效果如图 11-17 所示。

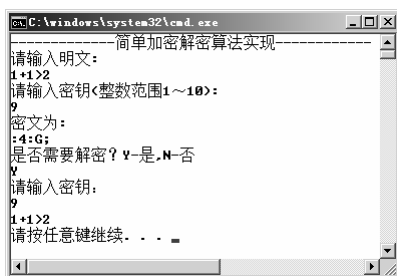


图 11-17 解密算法实现

【实现过程】

在实例 212 的基础上，定义字符型变量 `decode` 和整型变量 `decode_key`。判断 `decode_key` 是否与密钥相同，输出各自的结果，代码如下：

```
01 #include<iostream>
02 #include <string>
03 using namespace std;
04
05 void main()
06 {
07     cout<<"-----简单加密解密算法实现-----"<<endl;
08     ...
09     char decode;                //解密否
10     int decode_key;             //解密密钥
11     cout<<"是否需要解密? Y-是,N-否\n";
12     cin>>decode;
13     if(decode=='y' || decode=='Y')
14     {
15         cout<<"请输入密钥: \n";
16         cin>>decode_key;
17         if(decode_key==key)     //相同
18             cout<<src_c;       //输出明文
19         else                     //密钥不同
20             cout<<"密钥错误!";
21     }
22     cout<<endl;
23 }
```

【代码解析】

第 09~10 行定义各类变量，第 12 行获取 `decode` 的值。第 13 行判断是否进行解密，第 16 行输入解密密钥 `decode_key` 的值。第 17 行判断密钥是否正确，如果正确，则输出第 18 行，否则输出第 20 行。



注意：复杂的加密算法是对明文进行各类复杂的运算形成的，如加、减、乘、除、幂次等。



实例 214 模拟打字软件

【实例描述】

本实例利用实例 211 的随机函数生成一段字符，模拟打字软件检测用户的打字准确率和大小写，并进行统计、打印字符串等。本例效果如图 11-18 所示。

【实现过程】

定义函数 `gene_rand()`，其两个输入参数分别为随机数的最大和最小值。定义字符串的随机个数及随机内容，通过对比输入字符串值和打印机字符串，计算其打印准确率。代码如下：

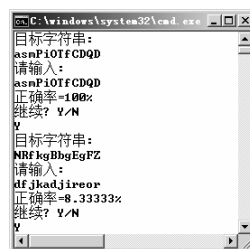


图 11-18 模拟打字软件

```

01 #include <iostream>
02 #include <cstdlib>
03 #include <ctime>
04 using namespace std;
05
06 int gene_rand(int max,int min)
07 {
08     int data; //随机数据
09     data=min+(int) (max*rand()/(RAND_MAX+min)); //生成随机数
10     return data;
11 }
12 void main()
13 {
14     char src[20]; //被打印字符串
15     char dst[20]; //打印字符串
16     int num_ch; //字符个数
17     char contiue; //是否继续
18     do
19     {
20         int right=0; //正确个数
21         float precision=.0; //准确率
22         cout<<"目标字符串:\n";
23         srand((unsigned)time(NULL)); //种子值
24         num_ch=gene_rand(20,1); //字符个数
25         for(int i=0; i<num_ch; i++)
26         {
27             char c;
28             c='A'+gene_rand(52,0); //52 表示共有 52 个大小写字母
29             while(c>=('A'+26) && c<('A'+32)) //不是字母则重新选择
30             {
31                 c='A'+gene_rand(52,0);
32             }
33             cout<<c; //循环输出字母
34             src[i]=c;
35         }

```



```

36         cout<<"\n 请输入:\n";
37         for(int j=0;j<num_ch;j++)           //输入字符
38             cin>>dst[j];
39         for(int j=0;j<num_ch;j++)           //判断正误
40         {
41             if(src[j]==dst[j])               //相同
42                 right++;                     //正确数加 1
43         }
44         precision=(float)right/num_ch*100;
45         cout<<"正确率="<<precision<<"%"<<endl;
46         cout<<"继续? Y/N\n";
47         cin>>conti;
48         }while(conti=='y' || conti=='Y');
49     }

```

【代码解析】

第 06~11 行是函数 `gene_rand()` 的定义体，返回随机值。第 14~17 行定义被打印的字符串等变量，第 18~48 行是一个 `do...while` 循环结构，输出打印字符串等准确率统计处理。其中，第 20、21 行定义正确个数和准确率变量 `right` 和 `precision`。第 23 行获取种子值，第 24 行获取随机字符个数 `num_ch`。第 25~35 行输出 `num_ch` 个随机字符，第 37、38 行输入打印字符内容，第 39~43 行判断正误，第 44 行计算准确率。



注意：如果第 20、21 行的变量不位于 `do...while` 结构中，必须在准确率计算完后立即将其置为初始值。



实例 215 计算算法耗时

【实例描述】

验证某个算法的效率是否高，需要计算其耗费多少时间。本实例实现如何计算完成一个减法耗费的时间，以实例 211 的随机数据生成算法为例，获取其耗时如图 11-19 所示。

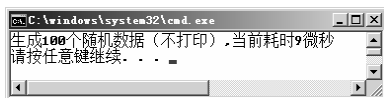


图 11-19 算法耗时计算

【实现过程】

定义函数 `getTime()` 获取 CPU 当前的精确时间值，在调用随机数据生成算法前后都分别获取时间值，然后相减就得算法耗时，代码如下：

```

01     ...
02     void gene_rand()
03     {
04         ...
05         {
06             data=MIN+(int) (MAX*rand() / (RAND_MAX+MIN)); //生成随机数
07         }
08     }

```



```

09  }
10  LONG64 getTime()                                //获取 CPU 当前精确时间
11  {
12      LARGE_INTEGER litmp;
13      LONG64 QPart;                                //时间类型变量
14      QueryPerformanceCounter(&litmp);             //获取当前时间
15      QPart=litmp.QuadPart;                         //获取 longlong 型数据
16      return QPart;
17  }
18  void main()
19  {
20      srand((int)time(0));                          //种子值
21      LONG64 QPart1,QPart2;
22      double d=.0;                                  //算法时间
23      QPart1=getTime();                             //执行算法前时间
24      gene_rand();                                   //随机数据生成算法
25      QPart2=getTime();                             //执行算法后时间
26      d=(double)(QPart2-QPart1);                   //算法耗时
27      cout<<"生成 100 个随机数据（不打印），当前耗时"<<d<<"微秒"<<endl;
28  }

```

【代码解析】

第 02~09 行是函数 `gene_rand()` 的定义体，其中将数据打印去除。第 10~17 行获取 CPU 当前时间，其中第 12、13 行定义所需变量，第 14 行获取当前时间，第 15 行获取时间结构体中 `LONGLONG` 型数据。

第 21 行定义算法前后的时间值，第 22 行定义算法耗时变量。第 23、25 行调用函数 `getTime()` 获取时间，第 26 行计算算法耗时，第 27 行输出耗时。



注意：由于算法运行时，系统资源周边环境不同，所以相同的算法在不同的时刻运行，耗时也不同。



实例 216 插入排序算法

【实例描述】

本实例演示插入排序算法的实现，其原理为对数组的前某个数量元素进行比较，将其最大元素置于最后，最小元素置于最前，其实现过程如图 11-20 所示。



图 11-20 插入排序算法



【实现过程】

定义函数 InsertSort()实现直接插入排序算法,定义 int 型数组变量 a 作为目标排序数组,代码如下:

```
01  #include <iostream>
02  using namespace std;
03
04  #define M 11
05
06  void InsertSort(int a[])
07  {
08      cout<<"排序过程:"<<endl;
09      int temp;                      //临时变量
10      int i,j;                      //循环变量
11      for(i=1;i<M;i++)
12      {
13          temp=a[i];                //获取比较值
14          for(j=i;j>0&& a[j-1]>temp;j--) //前 i 个元素,如果有大元素交换
15              a[j]=a[j-1];          //移到当前位置
16          a[j]=temp;                //将最后一个交换的 j 位置元素赋值 temp
17          for(int k=0;k<M;k++)
18              cout<<a[k]<<" ";
19          cout<<endl;
20      }
21  }
22  void main()
23  {
24      cout<<"-----插入排序-----"<<endl;
25      int a[M]={0, 209, 386, 768, 185, 247, 606, 230, 834, 54, 12};
26      cout<<"排序之前的元素为:\n";
27      for(int i=0; i<M; i++)          //循环排序前数组
28          cout<<a[i]<<" ";
29      cout<<endl;
30      InsertSort(a);                  //插入排序法
31      cout<<"排序结果为:\n";
32      for(int i=0; i<M; i++)          //循环排序后数组
33          cout<<a[i]<<" ";
34      cout<<endl;
35  }
```

【代码解析】

第 04 行定义数组 M 的大小,第 06~21 行为插入排序算法的定义体。其中,第 11~20 行实现对其 i 个元素进行比较大小后排序,直到对 M-1 个元素进行排序。第 17、18 行输出排序过程中数组元素的变化。第 25 行定义数组变量 a,第 27、28 行输出排序前数组,第 30 行调用插入排序函数,第 32、33 行输出排序后的数组。



注意: 第 14 行的 for 循环终止条件有两个,一是 j 大于 0,二是元素 a[j-1]大于比较值 temp。如果这两个条件不同时满足,不能交换相邻元素值。所以第 16 行的比较值只能赋给最后参与交换的 a[j]元素。



实例 217 冒泡排序

【实例描述】

本实例模拟冒泡算法对目标元素进行排序。当进行从小到大的排序时，是将大的元素以冒泡的方式浮到最后位置。该算法排序的实现过程如图 11-21 所示。



图 11-21 冒泡排序

【实现过程】

定义函数 Bubble() 做冒泡排序算法实现，定义 int 型数组 a[M]，大小为 11，分别输出排序前、排序过程及排序后的数组元素。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 #define M 11
05
06 void Bubble(int a[]) //冒泡排序
07 {
08     cout<<"排序过程:"<<endl;
09     int temp; //临时变量
10     for(int i=0;i<M-1;i++) //元素比较界限
11     {
12         for(int j=0;j<M-1-i;j++) //该元素与加 1 位置元素进行比较
13         {
14             if(a[j]>a[j+1]) //把大元素放到位置右
15             {
16                 temp = a[j]; //元素交换
17                 a[j] = a[j+1];
18                 a[j+1] = temp;
19             }
20         }
21         for(int k=0;k<M;k++)
22             cout<<a[k]<<" "; //输出每次排序的结果
23         cout<<endl;
24     }
25 }
26 void main()
27 {
28     cout<<"-----冒泡排序-----"<<endl;
29     int a[M]={0, 209, 386, 768, 185, 247, 606, 230, 834, 54, 12};
```



```

30     cout<<"排序之前的元素为:\n";
31     for(int i=0; i<M; i++)                //循环排序前数组
32         cout<<a[i]<<" ";
33     cout<<endl;
34     Bubble(a);                            //冒泡法
35     cout<<"排序结果为:\n";
36     for(int i=0; i<M; i++)                //循环排序后数组
37         cout<<a[i]<<" ";
38     cout<<endl;
39 }

```

【代码解析】

第 04 行定义数组 a 的大小, 第 06~25 行是函数 Bubble() 的定义体。其中第 10~24 行实现冒泡算法, 第 14~19 行交换符合条件的相邻元素, 第 21、22 行输出排序过程中的排序结果。第 29 行定义目标数组, 第 31、32 行输出排序前的元素, 第 34 行调用函数 Bubble(), 第 36、37 行输出排序后的数组。



注意: 冒泡排序的算法平均时间复杂度为 $O(n^2)$ 。



实例 218 选择排序法

【实例描述】

本实例实现选择排序算法, 核心思想是首先选取当前最小值的位置及值, 与其后各个元素相比, 如果还有最小元素, 则交换位置, 直到到达数组最末尾, 其实现过程如图 11-22 所示。

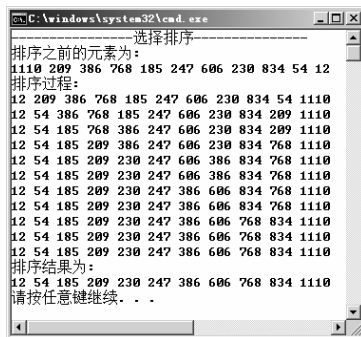


图 11-22 选择排序法

【实现过程】

定义数组 a[M], 定义函数 SelectSort() 用于选择排序算法。依据上述算法的实现原理, 代码如下:

```

01 #include <iostream>
02 using namespace std;
03
04 #define M 11
05
06 void SelectSort(int a[])
07 {

```



```

08      cout<<"排序过程:"<<endl;
09      int pos;                      //目前最小的数字的位置
10      int temp;                     //temp 存最小数字
11      for(int i=0;i<M;i++)
12      {
13          pos=i;                    //最小值位置
14          temp=a[i];                //最小值
15          for(int j=i+1;j<M;j++)    //查找最小的字符
16          {
17              if(a[j]<temp)           //新的最小值出现
18              {
19                  pos=j;              //新的最小字符的位置
20                  temp=a[j];
21              }
22          }
23          a[pos]=a[i];               //交换元素
24          a[i]=temp;                 //最小值置于最低位
25          for(int k=0;k<M;k++)
26              cout<<a[k]<<" ";
27          cout<<endl;
28      }
29  }
30  void main()
31  {
32      cout<<"-----选择排序-----"<<endl;
33      int a[M]={1110, 209, 386, 768, 185, 247, 606, 230, 834, 54, 12};
34      cout<<"排序之前的元素为:\n";
35      for(int i=0; i<M; i++)          //循环排序前数组
36          cout<<a[i]<<" ";
37      cout<<endl;
38      SelectSort(a);                  //选择排序法
39      cout<<"排序结果为:\n";
40      for(int i=0; i<M; i++)          //循环排序后数组
41          cout<<a[i]<<" ";
42      cout<<endl;
43  }

```

【代码解析】

第 06~29 行为函数 `SelectSort()` 的定义体, 其中第 09、10 行定义最小值的位置和内容变量。第 11~28 行为算法实现, 其中首先获取当前最小值的位置和值 (第 13、14 行)。然后在后面元素进行比较找寻新的最小值, 更新变量 `pos` 和 `temp` 的值 (第 15~22 行)。第 23、24 行实现元素的交换, 第 25、26 行输出算法过程中元素的变化。第 38 行调用插入排序函数 `SelectSort()`。



注意: 当元素个数较小时, 选择排序比冒泡排序快。



实例 219 猜数字

【实例描述】

本实例实现大家熟悉的猜数字游戏, 程序随机生成 0~100 之间的一个整数, 由游戏者输入答案, 程序根据答案及数字的大小输出提示语, 游戏者根据提示继续输入, 直到猜中为止。运行效果如图 11-23 所示。

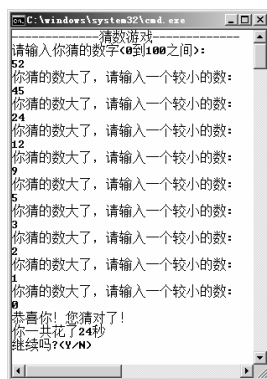


图 11-23 猜数字

【实现过程】

在算法实现前后获取当前时间值，最后输出游戏者共用多长时间猜中数字，其代码如下：

```
01 #include <time.h>
02 #include <stdlib.h>
03 #include <iostream>
04 using namespace std;
05
06 void main()
07 {
08     char c;
09     time_t begin,end;                //所用时间
10     int i;                          //数字
11     int guess;                      //用户所猜数字
12     srand((unsigned)time(NULL));    //种子值
13     cout<<"-----猜数游戏-----\n";
14     do
15     {
16         i=0+(int)(100*rand()/(RAND_MAX+0)); //被猜数字
17         cout<<"请输入你猜的数字(0 到 100 之间):\n";
18         begin=time(NULL);             //起始时间
19         cin>>guess;
20         while(guess!=i)
21         {
22             if(guess>i)
23                 cout<<"你猜的数大了, 请输入一个较小的数:\n";
24             else
25                 cout<<"你猜的数小了, 请输入一个较大的数:\n";
26             cin>>guess;
27         }
28         cout<<"恭喜你! 您猜对了! \n";
29         end=time(NULL);              //终止时间
30         cout<<"你一共花了"<<difftime(end,begin)<<"秒\n";
31         cout<<"继续吗?(Y/N) \n";
32         cin>>c;                      //输入选择
33     }while(c=='Y' || c=='y');
34 }
```

【代码解析】

第 08~11 行定义时间变量、数字及用户所猜数字，第 12 行为随机数字生成种子值。第 14~



33 行的 do...while 实现是否继续进行游戏, 第 16 行生成 0~100 的随机数字。第 18、29 行分别获取起始和终止时间, 第 19 行输入所猜数字。如果数字不是正确答案, 执行第 22~26 行。



注意: 第 18、29、30 行是另一种获取时间和计算时间差的方法。



实例 220 数字小写变大写

【实例描述】

本实例意在实现输入小写数字后, 通过处理变为大写输出。程序运行效果如图 11-24 所示。

【实现过程】

定义字符串数组 str, 大小为 19, 存储大写数字。长整型变量 num 和 i 分别标定目标数字和索引值。只对 20 位以内的数字进行转换, 代码如下:

```
01 #include<iostream>
02 #include <string>
03 using namespace std;
04
05 #define BIAO 11
06
07 void main()
08 {
09     string str[19]={" ", "壹", "贰", "叁", "肆", "伍", "陆", "柒",
10         "捌", "玖", " ", "拾", "佰", "仟", "万", "拾", "佰", "仟", "亿"};
11     long int num=0, i=0; //目标数字和索引值
12     int temp[20]={0}; //临时内存
13     cout<<"请输入要大写的正整数值(不可超过 20 位):"<<endl;
14     cin>>num;
15     while(num) //数字不为 0
16     {
17         temp[i]=num%10; //取最低位
18         num=num/10; //去掉最低位
19         temp[i+1]=BIAO+i/2; //上一位的单位
20         i+=2; //进入当前的后两位
21     }
22     int j=i-2; //返回前两位, i-1 位是单位
23     for(;j>=0;j--)
24         cout<<str[temp[j]];
25     cout<<endl;
26 }
```

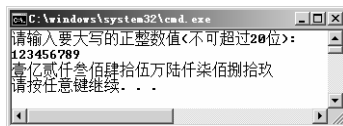


图 11-24 数字小写变大写

【代码解析】

第 09、10 行定义字符串变量 str, 第 11、12 行定义变量数字和临时内存。第 14 行获取数字值, 第 15~21 行的 while 循环将数字 num 的每位及该位代表的单位存入 temp 中。为了输出大写变量, 第 22 行将数组指针返回 2 位 (第 22 位)。第 23、24 行输出数字大写内容。



注意: 因为在 str 中元素“拾”的位置是 11, 所以在第 19 行的变量 BIAO 应该定义为 11。



实例 221 计算三位数字的水仙花数

【实例描述】

水仙花数指的是一个位数(N 次)不小于 3 的数字, 如果其每位数字的 N 次幂之和等于它本身, 就被称为水仙花数。本实例判定三位数中属于水仙花数的数字, 运行效果如图 11-25 所示。

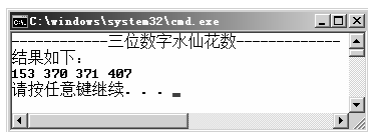


图 11-25 判定三位数的水仙花数

【实现过程】

定义变量 num 表示一个三位数, 变量 i、j 和 k 表示三位数的个、十、百位, 根据判定原理输出三位数的水仙花数, 代码如下:

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     cout<<"-----三位数字水仙花数-----"<<endl;
07     cout<<"结果如下: "<<endl;
08     int num; //三位数
09     int i,j,k; //每位数字
10     for(num=100;num<1000;num++) //三位的水仙花数
11     {
12         i=num/100; //百位
13         j=num/10%10; //十位
14         k=num%10; //个位
15         if (num==i*i*i+j*j*j+k*k*k) //是否符合条件
16             cout<<num<<" ";
17     }
18     cout<<endl;
19 }
```

【代码解析】

第 08、09 行定义变量以标识三位数及其各位数, 第 10~17 行的 for 循环将符合条件的水仙花数字输出。其中第 12~14 行获取个、十、百位, 第 15 行是判定条件。



注意: 利用该程序的原理, 还可以判定更多位数字是否是水仙花数。



实例 222 杨辉三角形示例

【实例描述】

数学中有一个杨辉三角形, 本实例利用 C++ 语言对其进行简单实现, 它的两个等腰边元素值为 1, 其内部元素值是其两肩元素之和, 运行效果如图 11-26 所示。

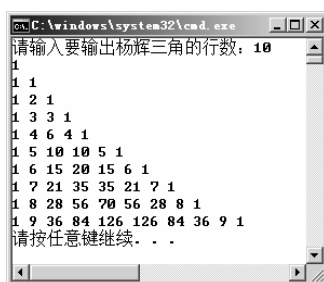


图 11-26 杨辉三角

【实现过程】

定义变量 N 作为杨辉三角显示的行数，为了方便计算，申请 $(N+1)*(N+1)$ 个二维内存。首先确定外边元素都为 1，然后计算内部元素值，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     int N; //行数
07     cout<<"请输入要输出杨辉三角的行数: ";
08     cin>>N;
09     N+=1; //加一圈外墙
10     int **a=new int*[N];
11     for(int i=0;i<N;i++)
12         a[i]=new int[N];
13     for(int i=1;i<N;i++) //确定元素 1
14     {
15         a[i][1]=1;
16         a[i][i]=1;
17     }
18     for(int i=3;i<N;i++) //确定其他元素
19     {
20         for(int j=2;j<=i-1;j++)
21             a[i][j]=a[i-1][j-1]+a[i-1][j]; //当前元素是肩上两个元素之和
22     }
23     for(int i=1;i<N;i++) //输出杨辉三角
24     {
25         for(int j=1;j<=i;j++)
26             cout<<a[i][j]<<" ";
27         cout<<"\n";
28     }
29 }
```

【代码解析】

第 06 行定义变量行数，第 09 行使数组内存加 1。第 10~12 行申请二维内存，第 13~17 行确定值为 1 的元素。第 18~22 行确定其内部元素的值，第 23~28 行输出行数为 N 的杨辉三角。



注意：细心一点可以看出，本程序最后没有用 delete 命令删除二维内存。如果在大型程序中，这会导致程序运行出现错误。



实例 223 剪刀石头布单机版小游戏

【实例描述】

本实例实现单机版的剪刀石头布游戏，规则为三局两胜，人机对战。每一局由玩家输入自己的选择，再输出电脑的选择。当满足条件时，输出玩家当前的输赢状态。当三局完成后，提醒玩家是否再来一局。效果如图 11-27 所示。

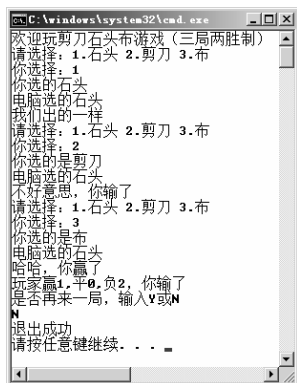


图 11-27 一个简单的数据统计系统（综合）

【实现过程】

定义类 Game 实现单机版的剪刀石头布游戏，包含成员变量 `_player`（玩家的选择）、`_computer`（电脑的选择）、`_win`（玩家赢的次数）、`_draw`（玩家平局次数）、`_count`（规定三局几胜制）。成员函数 `player()`（获取玩家当前的选择）、`computer()`（获取电脑当前的选择）、`compare()`（比较两者的输入）、`display()`（输出当前比赛结果）。代码如下：

```
01 #include<iostream>
02 #include<time.h>
03 #include<stdlib.h>
04 #include<cmath>
05 using namespace std;
06
07 class Game
08 {
09 private:
10     int _player;           //用来存放玩家的选择
11     int _computer;        //用来存放电脑的选择
12     int _win;             //计数玩家赢的局数
13     int _draw;           //平局次数
14     int _count;          //几局胜
15 public:
16     Game();
17     void player();        //用来获取玩家的合法输入，其值保存在_player中
18     void computer();      //用来获取计算机的输入，其值保存在_computer中
19     void compare();       //比较
20     void display();       //显示比赛结果
21 };
```

然后定义类 Game 的构造函数，以及成员函数 `display()`、`player()` 和 `computer()` 的定义，代码



如下:

```
01 Game::Game()
02 {
03     _player=0;                //玩家
04     _computer=0;              //电脑
05     _win=0;                   //赢了几次
06     _draw=0;                  //平局次数
07     _count=2;                 //三局两胜
08 }
09 void Game::display()
10 {
11     if(_win>=_count)           //玩家赢的次数不小于参与次数
12         cout<<"玩家赢"<<_win<<"，平"<<_draw<<"，负"<<3-_win-_draw<<"，你赢了"
13         <<endl;
14     else
15     {
16         if(_draw==0)           //没有平局
17             cout<<"玩家赢"<<_win<<"，平"<<_draw<<"，负"<<3-_win-_draw<<"，你输了"
18             <<endl;
19         else                    //有平局
20             cout<<"本局结果无效"<<endl;
21     }
22 void Game::computer()
23 {
24     srand((unsigned)time(NULL)); //种子值
25     _computer=1+rand()%(1-3+1); //生成1~3 内的整数
26     switch(_computer)
27     {
28     case 1:                     //石头
29         cout<<"电脑选的石头 "<<endl;
30         break;
31     case 2:                     //剪刀
32         cout<<"电脑选的剪刀"<<endl;
33         break;
34     case 3:                     //布
35         cout<<"电脑选的布"<<endl;
36         break;
37     }
38 }
39 void Game::player()
40 {
41     do                          // 获取玩家的有效输入
42     {
43         cout<<"请选择: 1.石头 2.剪刀 3.布"<<endl;
44         cout<<"你选择: ";
45         cin>>_player;           //输入选择
46         if(_player<1||_player>3) //选择无效
47         {
48             cout<<"你的选择无效"<<endl;
49         }
50     }while(_player<1||_player>3); //跑出循环条件
51
52     switch(_player)             //显示玩家的选择
53     {
54     case 1:
55         cout<<"你选的石头"<<endl;
56         break;
```



```
57     case 2:
58         cout<<"你选的是剪刀"<<endl;
59         break;
60     case 3:
61         cout<<"你选的是布"<<endl;
62         break;
63     }
64 }
```

在获取玩家和电脑的选择后，下一步对两者进行比较，并保存结果，由函数 `compare()` 执行，代码如下：

```
01 void Game::compare()
02 {
03     if(_computer==_player)                //所出一样
04         cout<<"我们出的一样"<<endl;
05     else                                    //所出不一样
06     {
07         if(_computer==1)                  //电脑石头
08         {
09             if(_player==2)                //玩家剪刀
10                 cout<<"不好意思，你输了"<<endl;
11             else if(_player==3)           //玩家布
12             {
13                 cout<<"哈哈，你赢了"<<endl;
14                 _win++;                    //玩家赢次数加 1
15             }
16             else
17             {
18                 cout<<"平局"<<endl;
19                 _draw++;                    //平局加 1
20             }
21         }
22     else if(_computer==2)                  //电脑剪刀
23     {
24         if(_player==1)                    //玩家石头
25         {
26             cout<<"哈哈，你赢了"<<endl;
27             _win++;                        //玩家赢次数加 1
28         }
29         else if(_player==3)               //玩家布
30             cout<<"不好意思，你输了"<<endl;
31         else
32         {
33             cout<<"平局"<<endl;
34             _draw++;                        //平局加 1
35         }
36     }
37     else if(_computer==3)                  //电脑布
38     {
39         if(_player==1)                    //玩家石头
40             cout<<"不好意思，你输了"<<endl;
41         else if(_player==2)               //玩家剪刀
42         {
43             cout<<"哈哈，你赢了"<<endl;
44             _win++;                        //玩家赢次数加 1
45         }
46         else
47         {
```



```
48             cout<<"平局"<<endl;
49             _draw++;                      //平局加 1
50         }
51     }
52 }
53 }
```

最后，列出 main()函数的定义体，代码如下：

```
01 void main()
02 {
03     int j=3;                             //3 局
04     char ch;                             //是否再来一局
05     cout<<"欢迎玩剪刀石头布游戏（三局两胜制）"<<endl;
06     Game game;                           //游戏类对象
07     while(j>0)
08     {
09         game.player();                   //玩家选择
10         game.computer();                 //电脑选择
11         game.compare();                  //比较
12
13         if(j==1)
14         {
15             game.display();              //显示结果
16             cout<<"是否再来一局，输入 Y 或 N"<<endl;
17             cin>>ch;
18             if(ch=='Y')
19                 j=4;                     //因为之后有 j 减 1，所以赋值为 4
20             else
21                 cout<<"退出成功"<<endl;
22         }
23         j--;                             //可再进行的局数减 1
24     }
25 }
```

【代码解析】

第 1 段代码中，第 07~21 行为类 Game 的定义体，在实现过程中已对各类变量及函数做解释，此处不再赘述。

第 2 段代码中，第 01~08 行为构造函数对成员变量的初始化。第 09~21 行为成员函数 display()的定义体，主要是 if...else 结构判断玩家赢的次数 _win 与 _count 的大小，并输出玩家赢平负的次数。第 22~38 行为 computer()的定义体，第 39~64 行为 player()函数的定义体。在生成电脑的选择时采用随机值，所以第 24 行生成种子值，第 25 行生成 1 到 3 内的整数。玩家的选择从屏幕输入，如第 45 行。如果没有输入有效值，继续进入 do...while 循环体，如第 41~50 行。

第 3 段代码中，首先判断两者的值是否相同，如果相同，则输出结果一样（第 03、04 行）。如果不相同，根据各自的值，分类判断，结果对应的变量 _win 和 _draw 加 1。

第 4 段代码中，第 03 行定义的变量 j 标识一个回合进行几局，第 04 行的 ch 标识是否再来一个回合。第 06 行定义类 Game 对象，第 07~24 行的 while 循环判断一个回合是否已完成。第 09~11 行调用成员函数，第 13 行判断当进入最后一局时，显示结果，并询问是否再来一局。当输入 Y 时，将 j 的值赋为 4，进行下一个回合。



注意：如果第 24 行在第 19 行之前，j 的第二次赋值应该为 3，本实例采用先操作，后减 1。



实例 224 编写一个进制数转换器

【实例描述】

本实例实现一个进制转换器，可输入二进制、八进制、十进制或十六进制数，经过选择，转换为目标进制数。运行效果如图 11-28 所示。

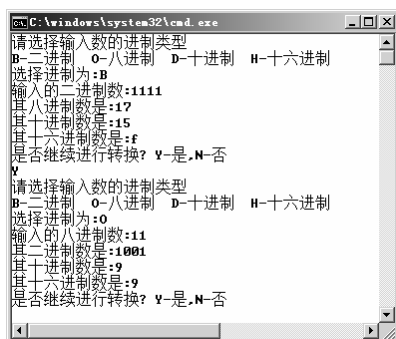


图 11-28 进制转换器

【实现过程】

定义函数 BtoD()实现二进制数转换为十进制数，函数 itoa()用于将其他进制数转换为二进制数。其他进制数的转换由 cout 输出流的输出格式变换，代码如下：

```
01  #include <iostream>
02  #include<stdlib.h>
03  #include<cmath>
04  using namespace std;
05
06  void BtoD(int B,int &D,int &W)           //二转十
07  {
08      int bit;
09      if(B>0)
10      {
11          bit=B%10;                       //取余，最低位是否是 1
12          D=D+bit*(int) (pow(2,float(W)));
13          B=B/10;                          //去掉最低位
14          W++;                             //二进制码每位对应权重加 1
15          BtoD(B,D,W);
16      }
17  }
18
19  void main()
20  {
21      char bi[255];                       //二进制数内存
22      int input;                          //输入进制数
23      char type;                          //输入进制类型
24      char contin;                        //标志是否继续选择
25      int de=0;                          //十进制数
26      int weight=0;                      //二进制码最低位权重
27      do
28      {
29          cout<<"请选择输入数的进制类型"<<endl;
```



```
30         cout<<"B-二进制 O-八进制 D-十进制 H-十六进制"<<endl;
31         cout<<"选择进制为:";
32         cin>>type;
33         switch(type)
34         {
35             case 'B':
36                 cout<<"输入的二进制数:";
37                 cin>>input;
38                 BtoD(input,de,weight); //二转为十进制
39                 cout<<"其八进制数是:"<<oct<<de<<endl;
40                 cout<<"其十进制数是:"<<dec<<de<<endl;
41                 cout<<"其十六进制数是:"<<hex<<de<<endl;
42                 break;
43             case 'O':
44                 cout<<"输入的八进制数:";
45                 cin>>oct>>input;
46                 cout<<"其二进制数是:";
47                 itoa(input,bi,2); //八进制转换为二进制数
48                 cout<<bi<<endl; //输出二进制
49                 cout<<"其十进制数是:"<<dec<<input<<endl;
50                 cout<<"其十六进制数是:"<<hex<<input<<endl;
51                 break;
52             case 'D':
53                 cout<<"输入的十进制数:";
54                 cin>>dec>>input;
55                 cout<<"其二进制数是:";
56                 itoa(input,bi,2); //十转为二进制
57                 cout<<bi<<endl;
58                 cout<<"其八进制数是:"<<oct<<input<<endl;
59                 cout<<"其十六进制数是:"<<hex<<input<<endl;
60                 break;
61             case 'H':
62                 cout<<"输入的十六进制数:";
63                 cin>>hex>>input;
64                 cout<<"其二进制数是:";
65                 itoa(input,bi,2); //十六进制转为二进制
66                 cout<<bi<<endl;
67                 cout<<"其八进制数是:"<<oct<<input<<endl;
68                 cout<<"其十进制数是:"<<dec<<input<<endl;
69                 break;
70             default:
71                 cout<<"选择进制类型错误\n";
72                 exit(-1);
73         }
74         cout<<"是否继续进行转换? Y-是,N-否"<<endl;
75         cin>>contin;
76         }while(contin=='Y');
77     }
```

【代码解析】

第 06~17 行是二进制数转换为十进制数的函数 BtoD() 定义体, 其中第 08 行的变量 bit 表示二进制当前位是 1 还是 0。如果二进制数不为 0, 进入 if 结构 (第 10~15 行)。首先取当前位变量 bit, 然后计算当前位变换为十进制数结果 (第 12 行), 第 13 行去掉已被计算的当前最低位。随着最低位逐渐被去掉, 当前最低位权重加 1 (第 14 行), 然后递归调用 (第 15 行), 直到变量 B 已变为 0 (第 09 行)。



第 21 行为二进制数转换内存, 第 22 行为输入进制数值, 第 23 行的变量 `type` 是输入进制类型, 第 24 行的变量代表是否继续进行进制转换, 第 25、26 行是二进制转换为十进制的数及二进制码最低位权重初始化。

第 29~31 行输出提示性语句, 第 32 行输入进制类型。第 33~73 行的 `switch` 结构依据输入类型 `type` 值对不同进制的数进行转换运算输出。在二进制运算中, 第 38 行将二进制数转换为十进制数, 第 39~41 行以不同的进制输出。在八进制运算中, 第 47 行将八进制转换为二进制数。同理, 第 56、65 行是将十进制数、十六进制数转换为二进制数。



注意: 利用 `cin` 输入流获取进制数时, 已经以不同进制格式将数据存入变量 `input` 中, 如第 37、45、54 和 63 行。



实例 225 建立链表

【实例描述】

根据数据结构中对链表操作的思想, 本实例简单实现一个链表以记录人口信息, 包括姓名、性别及年龄, 效果如图 11-29 所示。

【实现过程】

定义结构体 `human` 表示人口信息, 其有成员 `sex`、`name`、`age` 和 `next` 分别表示性别、姓名、年龄及下个节点。链表中现有 3 个节点, 对其进行初始化, 然后输出, 代码如下:

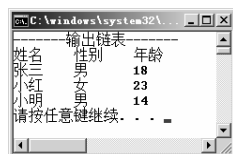


图 11-29 链表记录人口信息

```
01 #include <iostream>
02 using namespace std;
03
04 struct human
05 {
06     char *sex;           //性别
07     char *name;         //姓名
08     int age;            //年龄
09     struct human *next;  //下一个
10 };
11 void main()
12 {
13     struct human a,b,c; //3 个已有信息
14     struct human *head,*temp; //头指针, 临时指针
15     a.sex="男";          //初始化节点
16     a.name="张三";
17     a.age=18;
18     b.sex="女";
19     b.name="小红";
20     b.age=23;
21     c.sex="男";
22     c.name="小明";
23     c.age=14;
24     head=&a;             //头节点指向 a
25     a.next=&b;           //下个节点为 b
26     b.next=&c;           //节点 c
```



```

27      c.next=NULL;                //其后无节点
28      temp=head;
29      cout<<"-----输出链表-----\n 姓名\t 性别\t 年龄\n";
30      while (temp!=NULL)
31      {
32          cout<<temp->name<<"\t"<<temp->sex<<"\t"<<temp->age;
33          cout<<endl;
34          temp=temp->next;
35      }
36  }

```

【代码解析】

第 04~10 行表示结构体变量 `human`，其中，第 09 行是下一个节点。第 13、14 行分别定义结构体变量及链表节点，第 15~23 行对链表中各节点进行初始化。第 24 行将链表头节点赋值为 `a`，以后节点依次为 `b` 和 `c`。第 30~35 行输出链表中所有的元素值。



注意：链表的应用使得元素的插入在一定程度上变得简单易行。



实例 226 插入元素到链表

【实例描述】

在实例 225 的基础上，本实例实现如何在链表中插入一个元素。已知在某条信息后插入新信息，并更新链表信息。运行效果如图 11-30 所示。

【实现过程】

定义变量 `num` 表示新信息插入位置，`struct human` 型变量 `d` 表示被插入信息。`string` 型变量 `str` 以做转换变量类型之用。由 `cin` 获取 `num` 及信息等内容，完成插入后更新链表信息。代码如下：

```

01  ...
02  void main()
03  {
04      ...
05      int num;                //信息号
06      struct human d;         //临时信息
07      string str;              //string 型字符串
08      d.name=" ";              //初始化
09      d.sex=" ";
10      d.age=0;
11      d.next=NULL;
12      cout<<"需要在第几条后插入信息? ";
13      cin>>num;                //某条信息后
14      cout<<"请输入信息内容: "<<endl;
15      cout<<"姓名: ";cin>>str; //姓名
16      d.name=(char*)str.c_str(); //转换为 char* 型
17      cout<<"性别: ";cin>>str; //性别
18      d.sex=(char*)str.c_str(); //强制转换
19      cout<<"年龄: ";cin>>d.age; //年龄

```

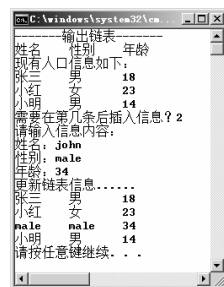


图 11-30 插入元素到链表



```
20     switch(num)                                //改变节点指向
21     {
22     case 1:
23         a.next=&d;
24         d.next=&b;
25         break;
26     case 2:
27         b.next=&d;
28         d.next=&c;
29         break;
30     case 3:
31         c.next=&d;
32         d.next=NULL;
33         break;
34     }
35     cout<<"更新链表信息....."<<endl;
36     temp=head;                                //指向头节点
37     while(temp!=NULL)                          //输出
38     {
39         cout<<temp->name<<"\t"<<temp->sex<<"\t"<<temp->age;
40         cout<<endl;
41         temp=temp->next;
42     }
43 }
```

【代码解析】

第 05~07 行定义变量，第 08~11 行初始化结构体变量 d。第 13 行获取插入信息的位置，第 15~19 行对新信息各元素赋值。第 20~34 行根据变量 num 的值改变节点的 next 指向，第 36 行使变量 temp 重新指向头节点，第 37~42 行输出新的链表信息。



注意：第 16、18 行是指 const char* 型强制转换为 char* 型变量。

第 12 章 Socket 网络及进程间通信

本章的实例围绕 Socket 网络编程及进程间通信展开介绍。对于 Socket 网络编程，围绕 TCP 和 UDP 的客户和服务端端间的数据通信介绍知识点。而进程间通信围绕各类手段展开模拟测试。这些手段包括 Windows 剪贴板、邮槽、匿名管道和命名管道等。



实例 227 网络客户端开发（TCP）

【实例描述】

套接字编程服务器/客户端模型有两个，分别是面向连接的 TCP 服务和面向无连接的 UDP 服务。本实例演示基于 TCP 服务类型的客户端开发，运行效果如图 12-1 所示。



图 12-1 网络客户端开发
(TCP/UDP)

【实现过程】

本例包含套接字库文件，然后定义套接字变量初始化 DLL 和地址结构信息等，等待服务器发来的数据。具体代码如下：

```
01 #pragma comment(lib, "Ws2_32.lib")
02 #include <Winsock2.h>
03 #include <iostream>
04 using namespace std;
05
06 void main()
07 {
08     cout<<"-----客户端开发-----"<<endl;
09     SOCKET sock; //套接字
10     struct sockaddr_in add; //地址信息
11     int servport = 5000; //端口号
12     char buff[256]; //缓冲区
13     int len; //缓冲区长度
14     WSADATA ws; //DLL 信息
15     WORD wVersionRequested; //版本号
16     wVersionRequested=MAKEWORD(2,2); //计算版本号
17     if(WSAStartup(wVersionRequested, &ws) !=0) //初始 DLL 失败
18     {
19         cout<<"初始化 DLL 失败"<<endl;
20         return;
21     }
22     sock=socket(AF_INET, SOCK_STREAM, 0); //创建套接字
23     memset(&add, 0, sizeof(add)); //初始化地址结构为 0
24     add.sin_family=AF_INET; //赋值
25     add.sin_port=htons(servport); //赋值端口信息
26     add.sin_addr.s_addr = inet_addr("192.168.1.101"); //IP 地址
27     connect(sock, (const sockaddr*)&add, sizeof(add)); //连接服务器
```



```

28     memset(buff,0,sizeof(buff));           //初始缓冲区
29     len=recv(sock,buff,sizeof(buff),0);     //接收数据
30     cout<<buff<<endl;                     //输出数据
31     closesocket(sock);                     //关闭套接字
32     WSACleanup();                          //清理资源
33 }

```

【代码解析】

第 01、02 行是使用套接字必须包含的库文件和头文件，第 09~11 行定义套接字变量，第 12、13 行定义获取信息的变量。第 14~21 行初始化套接字 DLL，第 22~27 行创建套接字，并初始化其有关信息。第 28~30 行初始化获取数据变量，并输出数据。第 31、32 行关闭套接字并清理资源。



注意：第 26 行的 IP 地址必须是服务器端运行的 IP 地址。



实例 228 网络服务器端开发（TCP）

【实例描述】

本实例与实例 227 在进行网络编程时配对使用，客户端接收的数据是服务器端发来的。本实例使用套接字完成服务器端的开发，运行效果如图 12-2 所示。



图 12-2 网络服务器端开发

【实现过程】

使用套接字开发网络服务器端，需要先创建套接字、绑定套接字、监听客户端和等待连接请求等准备工作。本实例在最后输出服务器端发出数据及本地地址信息。代码如下：

```

01 #pragma comment(lib,"Ws2_32.lib")
02 #include <Winsock2.h>
03 #include <iostream>
04 using namespace std;
05
06 void main()
07 {
08     cout<<"-----服务器端开发-----"<<endl;
09     SOCKET sersock,clisock;           //服务器与客户端套接字
10     struct sockaddr_in add;           //服务器信息
11     struct sockaddr_in cliaddr;       //客户端信息
12     int serport = 5000;               //端口号
13     char buff[256];                  //缓冲区
14     WSADATA ws;                      //记录 WinSock DLL 信息
15     int len;                          //缓冲区长度
16     WORD wVersionRequested;          //版本号
17     wVersionRequested=MAKEWORD(2,2); //计算版本号
18     if(WSAStartup(wVersionRequested,&ws)!=0) //初始化 Winsock
19     {
20         cout<<"初始化 DLL 失败!"<<endl;
21         return;

```



```

22     }
23     cout<<"创建套接字中..."<<endl;
24     sersock=socket(AF_INET, SOCK_STREAM, 0);           //创建套接字
25     memset(&add, 0, sizeof(add));                     //初始化地址信息
26     add.sin_family=AF_INET;
27     add.sin_port=htons(serport);
28     add.sin_addr.s_addr = INADDR_ANY;                 //对任意 IP 通信
29
30     cout<<"正在绑定中..."<<endl;
31     if(bind(sersock, (const sockaddr*)&add, sizeof(add))!=0) //绑定套接字
32     {
33         cout<<"绑定失败"<<WSAGetLastError()<<endl;
34         return;
35     }
36     cout<<"正在监听中..."<<endl;
37     if(listen(sersock, 5)!=0)                          //监听有错误
38     {
39         cout<<"监听失败"<<WSAGetLastError()<<endl;
40         return;
41     }
42     cout<<"等待连接请求中..."<<endl;                //等待连接请求
43     len=sizeof(cliaddr);
44     clisock=accept(sersock, (struct sockaddr*)&cliaddr, &len); //接受连接请求
45     cout<<"接受客户端连接请求"<<inet_ntoa(cliaddr.sin_addr)
46         <<ntohs(cliaddr.sin_port)<<endl;
47     sprintf(buff, "大家好%s", inet_ntoa(cliaddr.sin_addr)); //附加 IP 地址信息
48     send(clisock, buff, strlen(buff), 0);             //发送数据
49     closesocket(clisock);                             //关闭客户端套接字
50     closesocket(sersock);                             //关闭服务器套接字
51     WSACleanup();                                     //清理资源
52 }

```

【代码解析】

同开发客户端程序一样，第 09~17 行定义各类变量，第 18~22 行初始化 Winsock。第 23~28 行初始化服务器端的套接字，第 30~35 行对套接字进行绑定，第 36~41 行监听客户端信息，第 42~46 行等待客户端的连接请求。第 47 行输出服务器端发送的信息，第 48 行发送数据，第 49、50 行关闭服务器端和客户端的套接字，第 51 行清理资源。



注意：第 28 行设置为对任意地址进行通信，如果不想这样做，则需要设置为客户端的 IP 地址。



实例 229 网络服务器端开发（UDP）

【实例描述】

本实例结合实例 230 实现面向无连接的 UDP 服务网络编程，实现服务器端开发。这两个实例可以实现网络聊天，运行效果如图 12-3 所示。

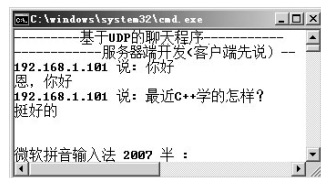


图 12-3 基于 UDP 网络聊天之服务器端



【实现过程】

同基于 TCP 的服务器端一样，需要事先包含 lib 和头文件。然后初始化 DLL 和地址信息，循环获取数据、发送数据、获取客户端发来数据等。具体代码如下：

```

01 #pragma comment(lib, "Ws2_32.lib")
02 #include <Winsock2.h>
03 #include <iostream>
04 using namespace std;
05
06 void main()
07 {
08     cout<<"-----基于 UDP 的聊天程序-----"<<endl;
09     cout<<"-----服务器端开发(客户端先说) --"<<endl;
10     WORD wVersionRequested;                //版本号
11     WSADATA wsaData;                      //记录 WinSockDLL 信息
12     wVersionRequested=MAKEWORD(2,2);      //计算版本号
13     SOCKET ss;                            //服务器端套接字
14     SOCKADDR_IN addrS;                   //地址
15     int len=sizeof(sockaddr);             //长度
16     SOCKADDR_IN addrC;                   //客户端地址
17     char recvBuf[128];                   //接收
18     char sendBuf[128];                   //发送
19     char tempBuf[128];                   //临时
20     if(WSAStartup(wVersionRequested, &wsaData) !=0)
21     {
22         cout<<"初始化 DLL 失败"<<endl;
23         return;
24     }
25     ss=socket(AF_INET, SOCK_DGRAM, 0);    //创建套接字
26     addrS.sin_family=AF_INET;
27     addrS.sin_port=htons(5000);          //端口
28     addrS.sin_addr.s_addr=INADDR_ANY;    //对任意 IP 通信
29     bind(ss, (sockaddr*)&addrS, len);    //绑定套接字
30     while(1)
31     {
32         recvfrom(ss, recvBuf, 128, 0, (sockaddr*)&addrC, &len); //接收
33         char *c_ip=inet_ntoa(addrC.sin_addr);
34         sprintf(tempBuf, "%s 说: %s\n", c_ip, recvBuf); //附加信息
35         cout<<tempBuf; //输出
36         gets(sendBuf); //获取服务器端发送数据
37         sendto(ss, sendBuf, strlen(sendBuf)+1, 0, (sockaddr*)&addrC, len); //发送
38     }
39     closesocket(ss); //关闭
40     WSACleanup(); //清理
41 }

```

【代码解析】

第 10~19 行定义各类变量用于创建套接字、接收和发送的数据，第 20~24 行判断是否初始化 DLL 成功。第 25~28 行创建套接字和初始化服务器端的地址信息，第 29 行绑定套接字。第 30~38 行无限循环地接收客户端数据、输出数据及发送数据。



实例 230 网络客户端开发 (UDP)

【实例描述】

本实例实现基于无连接型的 UDP 网络编程之客户端开发,服务器端对任意地址通信,而客户端针对服务器端地址通信。运行效果如图 12-4 所示。

【实现过程】

如同服务器端的实现步骤一样,获取屏幕数据、发送数据 and 获取服务器端数据。具体代码如下:

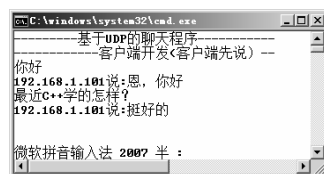


图 12-4 基于 UDP 网络聊天之客户端

```

01 #pragma comment(lib, "Ws2_32.lib")
02 #include <Winsock2.h>
03 #include <iostream>
04 using namespace std;
05
06 void main()
07 {
08     cout<<"-----基于 UDP 的聊天程序-----"<<endl;
09     cout<<"-----客户端开发(客户端先说) --"<<endl;
10     WORD wVersionRequested; //版本号
11     WSADATA wsaData; //记录 WinSockDLL 信息
12     wVersionRequested=MAKEWORD(2,2); //计算版本号
13     SOCKET cs; //客户端套接字
14     SOCKADDR_IN addr; //客户端地址
15     char recvBuf[128]; //接收
16     char sendBuf[128]; //发送
17     int len=sizeof(sockaddr); //长度
18     if(WSAStartup(wVersionRequested, &wsaData) !=0)
19     {
20         cout<<"初始化 DLL 失败"<<endl;
21         return;
22     }
23     cs=socket(AF_INET, SOCK_DGRAM, 0);
24     addr.sin_family=AF_INET;
25     addr.sin_port=htons(5000); //端口
26     addr.sin_addr.s_addr=inet_addr("192.168.1.101"); //服务器端 IP
27     while(1)
28     {
29         gets(sendBuf); //获取数据
30         sendto(cs, sendBuf, strlen(sendBuf)+1, 0, (sockaddr*)&addr, len); //发送
31         recvfrom(cs, recvBuf, 128, 0, (sockaddr*)&addr, &len); //获取
32         char *s_ip=inet_ntoa(addr.sin_addr); //转换
33         cout<<s_ip<<"说:"<<recvBuf<<endl; //输出
34     }
35     closesocket(cs);
36     WSACleanup();
37 }

```

【代码解析】

第 10~17 行定义各类变量对于客户端编程,第 18~22 行判断是否成功初始化 DLL。第 23~



26 行初始化客户端的地址信息，第 27~34 行无限循环地进行与服务器端的数据通信。



注意：基于 UDP 的服务器端不同于 TCP，它不需要监听。



实例 231 Windows 剪贴板通信之 A 端

【实例描述】

本实例结合实例 232 完成通过 Windows 剪贴板通信，它的实现原理是 A、B 端访问全局共享内存。本实例是 A 端实现，运行效果如图 12-5 所示。

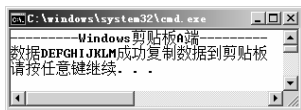


图 12-5 Windows 剪贴板通信之 A 端

【实现过程】

A 端发送数据到 Windows 剪贴板，在此之前需要打开剪贴板。然后发送文本信息，具体代码如下：

```
01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 void main()
06 {
07     cout<<"-----Windows 剪贴板 A 端-----"<<endl;
08     HWND hWnd = GetClipboardOwner();           //获取剪贴板的窗口句柄
09     DWORD dwLength=10;
10     HGLOBAL data;                               //全局内存
11     data = GlobalAlloc(GMEM_DDESHARE,dwLength+1); //分配内存
12     char *pdata;                                //先定义再初始化
13     pdata=(char*)GlobalLock(data);              //加锁
14     for(int i=0;i<dwLength;i++)
15         pdata[i]=char(i+68);
16     GlobalUnlock(data);                          //解锁
17     if(!OpenClipboard(hWnd))                    //没打开剪贴板
18     {
19         cout<<"剪贴板打开失败"<<endl;
20         return;
21     }
22     EmptyClipboard();                            //清空原有剪贴板内容
23     SetClipboardData(CF_TEXT,data);              //发送数据到剪贴板
24     CloseClipboard();                            //关闭剪贴板
25     cout<<"数据"<<pdata<<"成功复制数据到剪贴板"<<endl;
26 }
```

【代码解析】

第 08 行获取剪贴板窗口句柄，第 11 行分配全局内存，第 12 行定义数据变量 pdata。第 13 行给该块数据加锁，第 14、15 行初始化数据变量 pdata。第 16 行对其进行解锁，第 17 行判断是否成功打开剪贴板，第 22 行清空剪贴板原有的内容，第 23 行发送数据到剪贴板，第 24 行关闭剪贴板，第 25 行输出提示语句。



实例 232 Windows 剪贴板通信之 B 端

【实例描述】

本实例模拟利用 Windows 剪贴板实现进程间通信的另一端编程，该例先访问剪贴板，再获取其中的数据，然后输出，运行效果如图 12-6 所示。

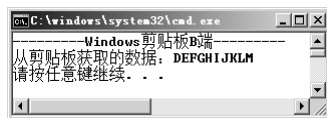


图 12-6 Windows 剪贴板通信之 B 端

【实现过程】

同 A 端编程一样，先获取窗口句柄，再打开剪贴板，然后获取并输出内容，具体代码如下：

```
01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 void main()
06 {
07     cout<<"-----Windows 剪贴板 B 端-----"<<endl;
08     HWND hWnd = GetClipboardOwner();           //获取剪贴板的窗口句柄
09     OpenClipboard(hWnd);                       //打开剪贴板
10     if(IsClipboardFormatAvailable(CF_TEXT))    //数据格式是否为 CF_TEXT
11     {
12         HANDLE data = GetClipboardData(CF_TEXT); //获取剪贴板数据句柄
13         char *pdata;
14         pdata=(char*)GlobalLock(data);         //锁定内存
15         cout<<"从剪贴板获取的数据: "<<pdata<<endl; //输出剪贴板中的内容
16         GlobalUnlock(data);                   //内存解锁
17     }
18     CloseClipboard();                          //关闭剪贴板
19 }
```

【代码解析】

第 08 行获取剪贴板窗口句柄，第 09 行打开剪贴板，第 10 行判断剪贴板的数据格式是否为文本。如果是文本，获取数据、锁定内存、输出数据。



注意：本实例在第 10 行判断数据格式是否为文本，是因为在 A 端程序的第 23 行发送的数据格式是文本。



实例 233 邮槽通信之 A 端

【实例描述】

本实例和实例 234 实现通过邮槽进行通信，邮槽端口分为 A 端和 B 端，A 端创建邮槽，并读取 B 端发送的数据，然后输出。运行效果如图 12-7 所示。

【实现过程】

先创建邮槽，当创建成功后读取另一端发来的数据。如

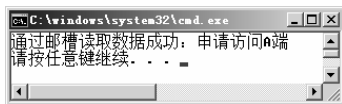


图 12-7 邮槽 A 端——读数据



果读取成功，输出获取的数据，代码如下：

```

01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 HANDLE hMailslot;           //邮箱句柄
06 void main()
07 {
08     hMailslot = CreateMailslot(TEXT("\\\\.\\mailslot\\slot"),0,
09         MAILSLLOT_WAIT_FOREVER,NULL);           //创建邮箱
10     if(hMailslot==INVALID_HANDLE_VALUE)
11     {
12         cout<<"创建邮箱失败。。。"<<endl;
13         return;
14     }
15     char buffer[100];        //缓冲区
16     DWORD dwRead;
17     if(!ReadFile(hMailslot,buffer,100,&dwRead,NULL)) //读取数据
18     {
19         cout<<"通过邮箱读取数据失败。。。"<<endl;
20         return;
21     }
22     cout<<"通过邮箱读取数据成功: "<<buffer<<endl; //输出
23     CloseHandle(hMailslot); //关闭句柄
24 }

```

【代码解析】

第 05 行定义邮箱句柄变量 hMailslot。第 08、09 行创建邮箱。第 10 行判断邮箱是否被创建成功，如果不成功则返回，反之，则继续执行代码。第 17 行判断读取数据是否成功，如果读取成功，输出获取的数据，如第 22 行。



注意：邮箱通信的一端只能读，另一端只能写，但邮箱可以实现跨网络通信。



实例 234 邮箱通信之 B 端

【实例描述】

本实例模拟邮箱通信的 B 端，其只能写数据，从 B 端发送字符串到 A 端，运行效果如图 12-8 所示。

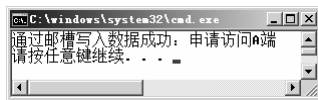


图 12-8 邮箱 B 端——写数据

【实现过程】

本实例执行 3 个操作，分别定义邮箱句柄、打开邮箱、写入数据，具体代码如下：

```

01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 HANDLE hMailslot;           //邮箱句柄
06 void main()
07 {
08     hMailslot = CreateFile(TEXT("\\\\.\\mailslot\\slot"),GENERIC_WRITE,
09         FILE_SHARE_READ,NULL,OPEN_EXISTING,

```



```
10         FILE_ATTRIBUTE_NORMAL, NULL); //打开邮箱
11     if(hMailslot==INVALID_HANDLE_VALUE)
12     {
13         cout<<"打开邮箱失败。。。"<<endl;
14         return; //返回
15     }
16     DWORD dwWrite;
17     char* buffer = "申请访问 A 端";
18     if(!WriteFile(hMailslot, buffer, strlen(buffer)+1, &dwWrite, NULL)) //写入数据
19     {
20         cout<<"通过邮箱写入数据失败。。。"<<endl;
21         return;
22     }
23     cout<<"通过邮箱写入数据成功: "<<buffer<<endl;
24 }
```

【代码解析】

第 05 行定义 B 端的邮箱句柄。第 08~10 行打开邮箱。第 11 行判断是否成功打开邮箱，如果成功打开，则写数据到邮箱。第 17 行定义数据 buffer。第 18 行判断是否成功写入。



实例 235 命名管道之客户端

【实例描述】

本实例和实例 234 模拟利用命名管道实现客户端及服务器端的通信。本实例给出客户端的模拟，运行效果如图 12-9 所示。

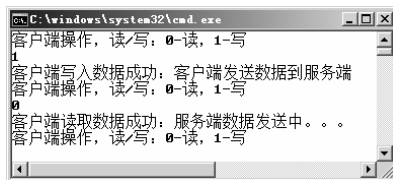


图 12-9 命名管道之客户端

【实现过程】

在客户端可以单独执行读写操作，定义函数 ClientR()和 ClientW()分别实现读写功能。具体代码如下：

```
01 #include <Windows.h>
02 #include <iostream>
03 using namespace std;
04
05 HANDLE hNamedPipe; //命名管道句柄
06
07 void ClientR() //读
08 {
09     char buffer[100];
10     DWORD dwRead;
11     if(!ReadFile(hNamedPipe, buffer, 100, &dwRead, NULL)) //读数据失败
12     {
```



```

13         cout<<"客户端读取数据失败。。。"<<endl;
14         return;
15     }
16     cout<<"客户端读取数据成功: "<<buffer<<endl;           //输出
17 }
18 void ClientW()                                           //写
19 {
20     DWORD dwWrite;
21     char* buffer = "客户端发送数据到服务器端";
22     if(!WriteFile(hNamedPipe, buffer, strlen(buffer)+1, &dwWrite, NULL))
23         //写入数据
24     {
25         cout<<"客户端写入数据失败。。。"<<endl;
26         return;
27     }
28     cout<<"客户端写入数据成功: "<<buffer<<endl;
29 }
30 void main()
31 {
32     if(!WaitNamedPipe(TEXT("\\\\.\\pipe\\namedpipe"), NMPWAIT_WAIT_FOREVER))
33         //等待连接命名管道
34     {
35         cout<<"命名管道实例不存在。。。"<<endl;
36         return;
37     }
38     hNamedPipe = CreateFile(TEXT("\\\\.\\pipe\\namedpipe"), GENERIC_READ |
39     GENERIC_WRITE,
40     0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL); //打开命名管道
41     if(INVALID_HANDLE_VALUE == hNamedPipe)
42     {
43         cout<<"打开命名管道失败。。。"<<endl<<endl;
44         return;
45     }
46     while(1)
47     {
48         cout<<"客户端操作, 读/写: 0-读, 1-写"<<endl;
49         int choose;
50         cin>>choose;
51         switch(choose)
52         {
53             case 0:
54                 ClientR();           //读
55                 break;
56             case 1:
57                 ClientW();           //写
58                 break;
59             default:
60                 break;
61         }
62     }
63 }

```

【代码解析】

第 05 行定义命名管道句柄 hNamedPipe, 第 07~17 行是函数 ClientR() 的定义, 其中第 11 行判断读数据是否成功, 所读取的数据存于字符数组 buffer。第 18~28 行是函数 ClientW() 的定义, 其中第 21 行初始化写数据变量, 第 22 行判断写入数据是否成功。在 main() 函数中, 先判断命名管道实例是否存在 (第 31 行), 如果存在, 则打开命名管道 (第 36、37 行)。打开命名



管道成功后,无限循环选择读写操作,如第 43~59 行。



注意: 命名管道不同于匿名管道,两个进程的地位处于同一级,即服务器端和客户端都可以主动读写数据。



实例 236 命名管道之服务器

【实例描述】

本实例模拟命名管道实现进程间通信的服务器应用,实例运行效果如图 12-10 所示。

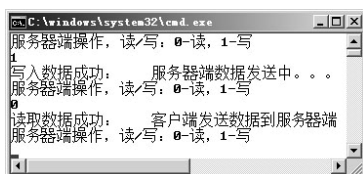


图 12-10 命名管道之服务器

【实现过程】

在命名管道的服务器端执行命令时,必须先创建命名管道,才能由客户端判断是否有实例存在。然后创建事件对象,具体代码如下:

```
01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 HANDLE hNamedPipe; //命名管道句柄
06 void ServerR()
07 {
08     char buffer[100];
09     DWORD dwRead;
10     if(!ReadFile(hNamedPipe,buffer,100,&dwRead,NULL))
11     {
12         cout<<"读取数据失败..."<<endl;
13         return;
14     }
15     cout<<"读取数据成功:    "<<buffer<<endl; //输出
16 }
17 void ServerW()
18 {
19     DWORD dwWrite;
20     char* buffer = "服务器端数据发送中。。。";
21     if(!WriteFile(hNamedPipe,buffer,strlen(buffer)+1,&dwWrite,NULL)) //写入数据
22     {
23         cout<<"写入数据失败 ..."<<endl;
24         return;
25     }
26     cout<<"写入数据成功:    "<<buffer<<endl;
27 }
28 void main()
29 {
```



```

30     HANDLE hEvent;
31     OVERLAPPED ovlpd;
32     hNamedPipe = CreateNamedPipe(TEXT("\\\\.\\pipe\\namedpipe"),
                                   //创建命名管道
33     PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED, //双向重叠模式
34     0, 1, 1024, 1024, 0, NULL);
35     if(INVALID_HANDLE_VALUE == hNamedPipe)
36     {
37         hNamedPipe = NULL;
38         cout<<"创建命名管道失败。。。"<<endl;
39         return;
40     }
41     hEvent = CreateEvent(NULL, TRUE, FALSE, NULL); //创建事件对象
42     if(!hEvent)
43     {
44         cout<<"创建事件失败。。。"<<endl;
45         return;
46     }
47     memset(&ovlpd, 0, sizeof(OVERLAPPED)); //初始化为 0
48     ovlpd.hEvent = hEvent; //赋值
49     if(!ConnectNamedPipe(hNamedPipe, &ovlpd)) //连接客户端
50     {
51         if(ERROR_IO_PENDING != GetLastError())
52         {
53             CloseHandle(hNamedPipe);
54             CloseHandle(hEvent);
55             cout<<"等待客户端连接失败 ..."<<endl;
56             return;
57         }
58     }
59     if(WAIT_FAILED == WaitForSingleObject(hEvent, INFINITE)) //等待事件状态
60     {
61         CloseHandle(hNamedPipe); //关闭命名管道
62         CloseHandle(hEvent); //关闭事件
63         cout<<"等待对象失败 ..."<<endl;
64         return;
65     }
66     CloseHandle(hEvent); //关闭事件
67     while(1)
68     {
69         cout<<"服务器端操作, 读/写: 0-读, 1-写"<<endl;
70         int choose;
71         cin>>choose;
72         switch(choose)
73         {
74             case 0:
75                 ServerR(); //读
76                 break;
77             case 1:
78                 ServerW(); //写
79                 break;
80             default:
81                 break;
82         }
83     }
84 }

```

【代码解析】

第 05 行创建命名管道句柄变量 hNamedPipe, 第 06~16 行定义服务器端读函数 ServerR(),



第 17~27 行定义服务器端写函数 ServerW()函数。第 32~34 行创建双向模式的命名管道，第 41 行创建事件对象。第 49 行连接客户端，第 59 行判断当前事件对象状态。第 67~83 行的 while 循环执行服务器端的读写操作。



注意：命名管道的服务器端需要结合事件对象一起使用。



实例 237 匿名管道通信之父进程

【实例描述】

匿名管道通信与命名管道相比，其功能要小很多，表现在不能进行跨网络通信。本实例和实例 238 实现利用匿名管道进行通信，父进程的运行效果如图 12-11 所示。

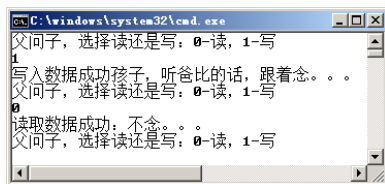


图 12-11 匿名管道之父进程

【实现过程】

定义函数 WritePipe()和 ReadPipe()分别作为父进程的写命令和读命令，当创建子进程成功后，循环读写匿名管道，其代码如下：

```
01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 void WritePipe(HANDLE hwrite)
06 {
07     char *buffer="孩子，听爸比的话，跟着念。。。"; //写入数据
08     DWORD dwWrite;
09     if(!WriteFile(hwrite,buffer,strlen(buffer)+1,&dwWrite,NULL)) //写入失败
10     {
11         cout<<"写入数据失败!"<<endl;
12         return;
13     }
14     cout<<"写入数据成功"<<buffer<<endl;
15 }
16 void ReadPipe(HANDLE hread)
17 {
18     char buffer[100]; //读出数据缓冲内存
19     DWORD dwRead;
20     if(!ReadFile(hread,buffer,100,&dwRead,NULL)) //读取失败
21     {
22         cout<<"读取数据失败!"<<endl;
23         return;
24     }
25     cout<<"读取数据成功："<<buffer<<endl; //输出
26 }
27 void main()
```



```

28  {
29      SECURITY_ATTRIBUTES sa;                //安全属性变量
30      PROCESS_INFORMATION process_info;        //进程信息
31      STARTUPINFO startup_info;              //开始信息
32      HANDLE hRead;                          //读
33      HANDLE hWrite;                         //写
34      sa.bInheritHandle=true;                //可继承句柄
35      sa.lpSecurityDescriptor=NULL;           //安全描述符
36      sa.nLength=sizeof(SECURITY_ATTRIBUTES); //长度
37      if(!CreatePipe(&hRead,&hWrite,&sa,0))    //创建匿名管道失败
38      {
39          cout<<"创建匿名管道失败!"<<endl;
40          return;
41      }
42      memset(&startup_info,0,sizeof(STARTUPINFO)); //初始化变量为0
43      startup_info.cb=sizeof(STARTUPINFO);
44      startup_info.dwFlags=STARTF_USESTDHANDLES; //标志
45      startup_info.hStdInput=hRead;            //输入
46      startup_info.hStdOutput=hWrite;          //输出
47      startup_info.hStdError=GetStdHandle(STD_ERROR_HANDLE); //错误处理句柄
48      if(!CreateProcess(TEXT("../debug/239.exe"),NULL,NULL,
49          NULL,true,CREATE_NEW_CONSOLE,NULL,NULL,
50          &startup_info,&process_info))        //创建子进程失败
51      {
52          CloseHandle(hRead);                  //关闭读句柄
53          CloseHandle(hWrite);                 //关闭写句柄
54          hRead=NULL;                          //置0
55          hWrite=NULL;                         //置0
56          cout<<"创建子进程失败!"<<endl;
57          return;
58      }
59      else                                     //子进程创建成功
60      {
61          CloseHandle(process_info.hProcess); //关闭进程
62          CloseHandle(process_info.hThread);  //释放资源
63          while(1)
64          {
65              cout<<"父问子, 选择读还是写: 0-读, 1-写"<<endl;
66              int choose;                      //选择
67              cin>>choose;
68              switch(choose)
69              {
70              case 0:
71                  ReadPipe(hRead);              //读
72                  break;
73              case 1:
74                  WritePipe(hWrite);            //写
75                  break;
76              default:
77                  break;
78              }
79          }
80      }
81  }

```

【代码解析】

第05~15行是函数 WritePipe() 的定义, 其中第07行是写入数据, 第09行判断写入数据是



否成功。第 16~26 行是函数 ReadPipe() 的定义，其中第 18 行存储读出数据，第 20 行判断读取数据是否成功。

第 29~36 行定义各类变量，并且为变量 sa 初始化。第 37~41 行创建匿名管道。第 42~47 行初始化变量 startup_info，第 48~58 行创建子进程，如果失败，则关闭读写句柄。如果子进程创建成功，则循环读写数据（如第 61~79 行）。



注意：第 48 行函数 CreateProcess() 的第 1 个参数是子进程 EXE 所在的目录。



实例 238 匿名管道通信之子进程

【实例描述】

本实例实现子进程，它的实质是父进程的附属。实例运行效果如图 12-12 所示，当父进程执行写操作时，子进程以消息盒子的形式收到数据。

【实现过程】

当父进程写入数据时，子进程接收数据。当父进程读取数据时，子进程写入数据。具体代码如下：

```
01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 void main()
06 {
07     HWND hw=FindWindow(L"ConsoleWindowClass",NULL); //控制台窗口
08     HANDLE hChildRead; //子进程读
09     HANDLE hChildWrite; //子进程写
10     hChildRead = GetStdHandle(STD_INPUT_HANDLE); //继承读句柄于父
11     hChildWrite= GetStdHandle(STD_OUTPUT_HANDLE); //继承写句柄于父
12     DWORD dwRead;
13     char Readbuf[100]; //缓冲内存
14     if(!ReadFile(hChildRead,Readbuf,datalength,&dwRead,NULL)) //读取数据
15     {
16         cout<<"通过匿名管道接收父进程数据失败!"<<endl;
17         return;
18     }
19     ::MessageBox(hw,L"通过匿名管道接收父进程数据成功",L"提示",MB_OK);
20     char* pSendbuf="不念。。。"; //发送内容
21     DWORD dwWrite;
22     if(!WriteFile(hChildWrite,pSendbuf,datalength,&dwWrite,NULL)) //写入数据
23     {
24         cout<<"给匿名管道发送数据失败!"<<endl;
25         return;
26     }
27     Sleep(10000); //睡一会儿
28 }
```

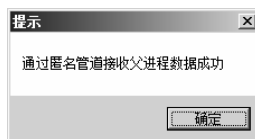


图 12-12 匿名管道之子进程



【代码解析】

第 07 行获取控制台窗口。第 08、09 行定义子进程读写句柄。第 10、11 行继承父进程的读写句柄。第 14 行判断通过匿名管道是否成功读取数据，第 22 行判断通过匿名管道是否成功发送数据。



注意：子进程的读写句柄需继承自父进程，不可自行创建。



实例 239 基于 TCP 的木马程序——服务器端

【实例描述】

本实例与实例 240 一起完成一个简单的木马程序，本实例实现服务器端的编程，主要可实现的功能有打开光驱、切换鼠标左右键、重置鼠标功能和退出。运行效果如图 12-13 所示。

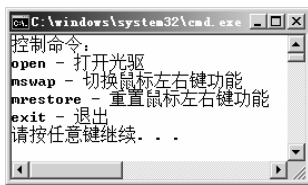


图 12-13 基于 TCP 的木马程序——服务器端

【实现过程】

定义函数 Dispatch()用以处理客户端发来的指令，打开光驱由 Winmm.lib 处理。本实例显示客户端可以发送指令，并接收客户端发来信息。具体代码如下：

```
01 #include <WINSOCK2.H>
02 #include <windows.h>
03 #include <iostream>
04 using namespace std;
05 #pragma comment (lib,"ws2_32")
06 #pragma comment(lib, "Winmm.lib ")           //用以处理光驱
07
08
09 BOOL Dispatch(SOCKET sock,char* szCmd)       //消息处理
10 {
11     BOOL bRet=false;
12     if(!strcmp(szCmd,"open"))                //打开光驱
13     {
14         mciSendString(L"set cdaudio door open",NULL,NULL,NULL);
15         send(sock,"打开光驱",strlen("打开光驱")+sizeof(char),0);
16         bRet=true;
17     }
18     else if(!strcmp(szCmd,"mswap"))           //调换鼠标左右键
19     {
20         SwapMouseButton(TRUE);
21         send(sock,"切换鼠标左右键功能",strlen("切换鼠标左右键功能")+sizeof(char),0);
22         bRet=true;
23     }
24     else if(!strcmp(szCmd,"mrestore"))        //恢复左右键
25     {
26         SwapMouseButton(FALSE);
27         send(sock,"重置鼠标左右键功能",strlen("重置鼠标左右键功能")+sizeof(char),0);
28         bRet=true;
29     }
30     else
```



```

31         send(sock, "无命令", strlen("无命令")+sizeof(char), 0);
32     return bRet;
33 }
34 void main()
35 {
36     cout<<"控制命令: \n";
37     cout<<"open - 打开光驱 \r\n"\
38     "mswap - 切换鼠标左右键功能 \r\n"\
39     "mrestore - 重置鼠标左右键功能 \r\n"\
40     "exit - 退出"<<endl;
41     WSADATA wsaData; //DLL 信息
42     WSStartup(MAKEWORD(2,2), &wsaData);
43     SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP); //套接字(TCP)
44     sockaddr_in addr; //地址
45     addr.sin_family=PF_INET;
46     addr.sin_addr.s_addr=inet_addr("192.168.1.101"); //IP
47     addr.sin_port=htons(5000);
48     bind(sock, (SOCKADDR*)&addr, sizeof(SOCKADDR)); //绑定
49     listen(sock, 5); //监听
50     SOCKADDR addr_c; //客户端地址
51     int nsize=sizeof(SOCKADDR);
52     SOCKET sock_c; //客户端套接字
53     sock_c=accept(sock, (SOCKADDR*)&addr_c, &nsize); //接收客户端请求
54     while(true)
55     {
56         send(sock_c, "指令>>", strlen("指令>>")+sizeof(char), 0);
57         char buff[256]={0};
58         recv(sock_c, buff, 256, 0); //接收控制指令
59         if(!strcmp(buff, "exit"))
60             break; //退出
61         BOOL bRet=Dispatch(sock_c, buff); //处理消息
62         if(bRet==false)
63             send(sock_c, "执行指令失败", strlen("执行指令失败")+sizeof(char), 0);
64     }
65     closesocket(sock_c); //关闭客户端套接字
66     closesocket(sock); //关闭服务器端套接字
67     WSACleanup();
68 }

```

【代码解析】

第 01、05 行包含套接字的头文件及 .lib 文件，第 06 行的 .lib 文件用以处理光驱指令。第 09～33 行是消息处理函数 Dispatch()，判断接收的字符串是否同目标字符串一样，以执行不同的指令。第 36～40 行是服务器端可以执行指令说明，第 41～53 行定义套接字，绑定、监听客户端并接收客户端发来的请求。第 54～64 行循环接收客户端的信息，第 65～67 行关闭套接字，清理资源。



注意：第 46 行的 IP 地址也可以设为任意，即可以被任意 IP 侵入。



实例 240 基于 TCP 的木马程序——客户端

【实例描述】

本实例用以实现客户端的操作，客户端可以接受服务器端发来的信息，比如提示和执行结



果。然后获取输入指令，发送服务器端执行。运行效果如图 12-14 所示。

【实现过程】

在获取消息循环中，首先接收“指令>>”字符串，然后获取输入指令，经过执行后获取执行结果，具体代码如下：

```
01 #include <WINSOCK2.H>
02 #include <windows.h>
03 #include <iostream>
04 using namespace std;
05 #pragma comment (lib, "ws2_32")
06
07 void main()
08 {
09     WSADATA wsaData;
10     WSStartup(MAKEWORD(2,2), &wsaData);
11     SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
12     sockaddr_in addr;
13     addr.sin_family=PF_INET;
14     addr.sin_addr.s_addr=inet_addr("192.168.1.101"); //服务器端 IP
15     addr.sin_port=htons(5000); //端口
16     connect(sock, (SOCKADDR*)&addr, sizeof(SOCKADDR)); //连接服务器端
17     while(true)
18     {
19         char buff[256]={0};
20         char Cmd[256]={0};
21         recv(sock, buff, 256, 0); //接收
22         cout<<buff; //指令
23         cin>>Cmd; //输入命令
24         send(sock, Cmd, 256, 0); //发送
25         if(!strcmp(Cmd, "exit"))
26         {
27             cout<<"退出!\r\n";
28             break;
29         }
30         memset(buff, 0, 256);
31         recv(sock, buff, 256, 0); //接受内容
32         cout<<buff<<"\r\n";
33     }
34     getchar();
35     WSACleanup(); //清除资源
36 }
```



图 12-14 基于 TCP 的木马程序——客户端

【代码解析】

本实例最重要的是第 17~33 行的 while 循环，其中，第 19、20 行定义接收字符和指令字符。第 21 行首先接收服务器端发来的提示语句，然后由第 22 行输出。客户端获取输入指令，如第 23 行。然后由第 24 行发送到服务器端，第 25~29 行判断指令是否为 exit。如果为真，退出程序。反之，继续接收服务器端发送的信息，如第 31 行，并由第 32 行输出。



注意：在第 30 行必须使用 buff 清 0，否则在执行第 31 行后，buff 的内容会出现上一次 buff 的内容，除非当前 buff 接收的内容长度能完全覆盖上一次 buff 的内容。比如，上一次 buff 的内容是 abcdefg，而当前 buff 接收的内容是 efd，则在执行第 32 行时，输出内容是 efddefg。

第 13 章 算 法

算法是利用 C++ 语言编写实现某种功能的代码组成，本章的算法有难有易，但都是一些典型示例。比如，兔子繁殖和鸡兔同笼都是数学问题，反转整数、因式分解和被 3 整除的数是数字问题，等等。



实例 241 反转整数 (%)

【实例描述】

本实例模拟如何反转整数，当输入整数为 234456 时，输出结果应为 654432，实现原理为对 10 取余，运行效果如图 13-1 所示。

【实现过程】



图 13-1 反转整数

定义整型变量 `aa`（表示输入整数）及 `right_digit`（表示当前结果的最右位数）。如果没有全部反转，继续取余，待输出当前结果的最右位后，去掉该位，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main(void)
05 {
06     int aa;                // 整数
07     int right_digit;        // 当前结果数的最右位
08     cout<<"输入一个整数: ";
09     cin>>aa;
10     cout<<"反转整数后结果: ";
11     while(aa!=0)           // 最高位还没被输出
12     {
13         right_digit=aa%10;  // 计算最右位数字
14         cout<<right_digit;  // 输出
15         aa/=10;             // 去掉最右位
16     }
17     cout<<endl;
18 }
```

【代码解析】

第 06、07 行定义变量 `aa` 和 `right_digit`，第 09 行获取整数 `aa` 的值。第 11~16 行的 `while` 循环判断是否已全部反转，如果没有，则继续取余（第 13 行），输出最右位（第 14 行）。然后去掉最右位（第 15 行）。



注意：本算法的核心即对当前结果取 10 的余数。



实例 242 古典问题——兔子繁殖

【实例描述】

兔子繁殖问题是指刚出生的兔子从第 3 个月开始，每个月生一对兔子。现假设兔子都不死，问每个月的兔子总数为多少？总结每个月兔子总数为 1, 1, 2, 3, 5, 8, 13, 21, …，其递推公式是从第 3 个月开始，当前月的兔子总数是前两个月兔子数之和。本实例输出 30 个月中每个月兔子的总数，效果如图 13-2 所示。



1	1	2	3	5
8	13	21	34	55
89	144	233	377	610
987	1597	2584	4181	6765
10946	17711	28657	46368	75025
121393	196418	317811	514229	832040

请按任意键继续. . .

图 13-2 兔子繁殖问题

【实现过程】

定义变量 *i* 表示每个月的索引，数组 *F*[30] 表示 30 个月中每个月的兔子数，每隔 5 个月换行输出兔子数，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     cout<<"-----古典兔子问题-----\n";
07     int i;                //月数
08     int F[30]={1,1};      //初始化前两个月兔子数
09     for(i=2;i<30;i++)
10         F[i]=F[i-2]+F[i-1]; //计算后 28 个月兔子数
11     for(i=0;i<30;i++)    //输出兔子数
12     {
13         if(i%5==0)       //每隔 5 个月换行
14             cout<<"\n";
15         cout<<F[i]<<"\t";
16     }
17     cout<<"\n";
18 }
```

【代码解析】

第 07、08 行定义变量 *i* 和 *F*[30]。第 09、10 行计算每个月的兔子总数。第 11~16 行输出 30 个月中每个月的兔子总数。



注意：本实例所研究的数列又被称为 Fibonacci 数列。



实例 243 逆时针旋转方阵 90°

【实例描述】

本实例的算法实现输入一个方阵（即行和列的值相同），按逆时针方向旋转方阵 90°，并输出，运行效果如图 13-3 所示。



图 13-3 逆时针旋转方阵 90°

【实现过程】

定义变量 `row` 存储方阵的行列值，二维内存 `src` 和 `dst` 分别存储旋转前及旋转后的方阵。通过计算公式实现方阵的逆时针旋转，具体实现代码如下：

```
01  #include <iostream>
02  using namespace std;
03
04  void main()
05  {
06      int row;                                // 目标方阵行数
07      cout<<"-----逆时针旋转方阵 90 度-----"<<endl;
08      cout<<"请输入目标方阵的行数: ";
09      cin>>row;
10      int **src=new int*[row];                // 原方阵
11      int **dst=new int*[row];                // 旋转后
12      for(int i=0;i<row;i++)                  // 申请二维内存
13      {
14          src[i]=new int[row];
15          dst[i]=new int[row];
16      }
17      cout<<"请输入"<<row*row<<"个元素: "<<endl;
18      for(int i=0;i<row;i++)                  // 初始化
19      {
20          for(int j=0;j<row;j++)
21          {
22              cin>>src[i][j];
23              dst[row-1-j][i]=src[i][j];      // 旋转
24          }
25      }
26      cout<<"被旋转后的方阵: "<<endl;
27      for(int i=0;i<row;i++)                  // 输出旋转后的方阵
28      {
29          for(int j=0;j<row;j++)
30              cout<<dst[i][j]<<" ";
31          cout<<endl;
32      }
33      delete []src;
34      src=NULL;
35      delete []dst;
36      dst=NULL;
37  }
```

【代码解析】

第 06 行定义方阵的行数，第 09 行获取其值。第 10~16 行动态申请二维内存，以存储原



方阵和旋转后的方阵变量。第 18~25 行对原方阵元素进行初始化,并对其进行旋转运算,存储元素在目标方阵 dst 中(关键代码为第 23 行)。



注意: 可以利用第 23 行实现对原方阵的顺/逆时针 90°、180°、270° 及 360° 旋转。



实例 244 判断回文字符串

【实例描述】

本实例算法实现回文字符串的判断,回文字符串是指不管从左到右读还是从右到左读,该字符串内容都不变。比如,字符串 aabaa 就是回文字符串。运行效果如图 13-4 所示。

【实现过程】

定义变量 str 和 num 分别存储字符串及其长度,布尔型变量 flag 标志其是否为回文字符串。利用 string 类成员函数 substr() 获取其在某个位置的字符,判断对称位置的元素是否相同,代码如下:



图 13-4 判断回文字符串

```
01 #include <string>
02 #include <iostream>
03 using namespace std;
04
05 void main()
06 {
07     cout<<"-----回文字符串判断-----"<<endl;
08     string str;
09     cout<<"输入字符串: ";
10     cin>>str;
11     int num=str.length();           //字符串长度
12     bool flag=true;                 //是回文字符串
13     for(int i=0;i<num;i++)
14     {
15         if(str.substr(num-1-i,1)!=str.substr(i,1)) //对应位置不同
16         {
17             flag=false;           //不是
18             break;
19         }
20     }
21     if(flag==true)
22         cout<<"是回文字符串"<<endl;
23     else
24         cout<<"不是回文字符串"<<endl;
25 }
```

【代码解析】

第 08 行定义字符串变量 str,第 10 行获取其值。第 11、12 行定义字符串长度及其标志量 flag,第 13~20 行的嵌套 for 循环判断是否为回文字符串。当碰到字符不相同(第 15 行)时,立即退出循环,判定不是回文串。第 21~24 行根据 flag 的值输出其是否为回文字符串。



注意：本实例利用 string 类字符串是因为在用 cin 获取变量值时，不用刻意考虑其字符串的长度。



实例 245 求最大公约数和最小公倍数

【实例描述】

本实例模拟数字问题：如何求两数的最大公约数和最小公倍数。现输出两个正整数，经过运算输出其最大公约数和最小公倍数。运行效果如图 13-5 所示。

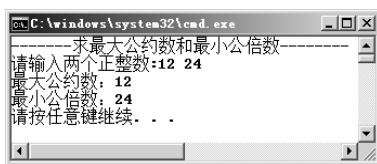


图 13-5 最大公约数和最小公倍数

【实现过程】

定义函数 MAX_CF()求最大公约数，函数 MIN_CD()求最小公倍数，变量 num1 和 num2 存储两个正整数，变量 cf 和 cd 表示计算后的最大公约数和最小公倍数，具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 int MAX_CF(int a,int b)           //最大公约数
05 {
06     int temp=0;
07     while(b!=0)
08     {
09         temp=a%b;                 //取余
10         a=b;                     //交换
11         b=temp;
12     }
13     return a;                    //返回目标值
14 }
15 int MIN_CD(int u,int v,int h)     //最小公倍数
16 {
17     return(u*v/h);
18 }
19 void main()
20 {
21     int num1,num2;                //两个数
22     int cf,cd;                   //公约和公倍数
23     cout<<"-----求最大公约数和最小公倍数-----"<<endl;
24     cout<<"请输入两个正整数:";
25     cin>>num1>>num2;
26     cf=MAX_CF(num1,num2);
27     cout<<"最大公约数: "<<cf<<endl;
28     cd=MIN_CD(num1,num2,cf);
29     cout<<"最小公倍数: "<<cd<<endl;
30 }
```



【代码解析】

第 04~14 行是函数 MAX_CF() 的定义体, 其中定义一个临时变量用于当 b 不为 0 时交换两数。当 b 为 0 时, 此时的 a 值为最大公约数, 由第 13 行返回。第 15~18 行为函数 MIN_CD() 的定义体, 其由第 17 行返回两数的最小公倍数, 它的第 3 个参数是已求得的最大公约数。第 21~22 行定义所需的 4 个变量, 第 25 行获取两数值, 第 26 行求取最大公约数, 第 28 行求取最小公倍数。



实例 246 图形输出算法

【实例描述】

本实例编写算法输出特定形状的图形, 现需控制台程序输出等边三角形, 即三角形的每条边都由相同数量的星号 (*) 组成。运行效果如图 13-6 所示。

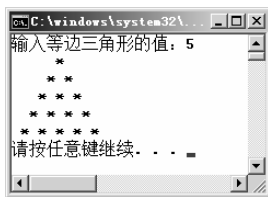


图 13-6 输出等边三角形

【实现过程】

定义函数 output() 输出图形, 该函数接收参数 n, 意指三角形的边由多少个星号组成。在编写代码前, 先明确该图形的组成, 现以每条边由 5 个星号组成为例进行介绍, 如图 13-7 所示为其星号位置。

	0	1	2	3	4	5	6	7	8
0					*				
1				*		*			
2			*		*		*		
3		*		*		*		*	
4	*		*		*		*		*

图 13-7 等边三角形图形图示

```
01 #include <iostream>
02 using namespace std;
03
04 void output(int n)
05 {
06     for(int i=0;i<n;i++)           //循环 n 次
07     {
08         int temp=i;               //临时值
09         for(int j=0;j<=n-(i+1);j++)
10             cout<<" ";           //先输出第一个星号前的空格
11         while((temp+1)>0)          //输出 i+1 个星号
12         {
13             cout<<"* ";           //再输出星号
14             temp--;               //星号个数减 1
15         }
16         cout<<endl;              //换行
17     }
18 }
19 void main()
20 {
21     int a;                        //三角形边长度
22     cout<<"输入等边三角形的值: ";
23     cin>>a;
24     output(a);                   //调用函数
25 }
```

【代码解析】

第 04~18 行是函数 output() 的定义体, 第 21 行定义三角形长度变量 a, 第 23 行获取其值,



第 24 行调用函数 `output()`。回到第 06 行循环 n 次，输出图形。对于每一行的第一个星号前空格的输出，请看第 09、10 行。每行第一个星号前的空格数量计算是 $n-(i+1)$ （其中， n 为边的长度， i 为当前行号）。第 11~15 行输出每行的星号加一个空格，每行的星号个数表达式为 $i+1$ 。每行输出完毕后输出一个回车符，如第 16 行。



注意：第 08 行一定要声明 `temp` 变量，用于赋值 i ，不能在 `while()` 循环中直接用 i 。



实例 247 八皇后位置放置问题

【实例描述】

本实例是八皇后位置放置问题，即在 8×8 格的国际象棋盘上放置 8 个皇后，使其不能互相攻击。因此，任意两个皇后不能放在同一行、同一列或同一条斜线上。本实例计算其有多少种摆法，并输出其中一种摆法。运行效果如图 13-8 所示。

【实现过程】

定义函数 `backtrack()` 和 `solution()`，分别表示皇后放置位置求解及输出解决方案。宏定义 N 表示皇后个数，数组 `column`、`rup` 和 `lup` 分别表示同列/行、副对角线、主对角线是否有皇后，数组 `queen` 表示皇后位置。代码如下：

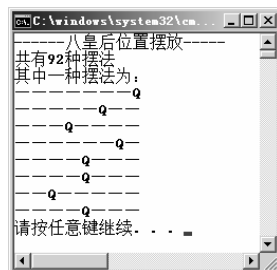


图 13-8 八皇后位置放置问题

```
01 #include <iostream>
02 using namespace std;
03
04 #define N 8
05 int column[N+1];           //同栏是否有皇后，1 表示有
06 int rup[2*N+1];           //副对角线是否有皇后
07 int lup[2*N+1];           //主对角线是否有皇后
08 int queen[N+1]={0};       //八皇后位置
09 int num=0;                 //摆法
10
11 void backtrack(int);        //求解
12 void solution();           //显示解决方案
13
14 void main()
15 {
16     cout<<"-----八皇后位置摆放-----"<<endl;
17     int i;
18     for(i=1;i<=N;i++)
19         column[i]=1;
20     for(i=1;i<=2*N;i++)
21         rup[i]=lup[i]=1;
22     backtrack(1);           //回溯法
23     cout<<"共有"<<num<<"种摆法"<<endl;
24     cout<<"其中一种摆法为："<<endl;
25     solution();             //显示当前摆法
26 }
27 void backtrack(int i)
28 {
29     int j;                 //循环量
30     if(i>N)                 //八个皇后位置都确定
```



```

31         num++;                                //解决方法加 1
32     else                                        //继续确定剩余皇后的位置
33     {
34         for (j=1; j<=N; j++)                    //循环
35         {
36             if (column[j]==1 && rup[i+j]==1 && lup[i-j+N]==1)    //都有皇后
37             {
38                 queen[i]=j;                        //确定皇后位置
39                 column[j]=rup[i+j]=lup[i-j+N]=0;        //去掉皇后
40                 backtrack(i+1);
41                 column[j]=rup[i+j]=lup[i-j+N]=1;        //恢复
42             }
43         }
44     }
45 }
46 void solution()
47 {
48     int i,j;
49     for (j=1; j<=N; j++)
50     {
51         for (i=1; i<=N; i++)
52         {
53             if (queen[j]==i)                        //皇后位置
54                 cout<<"Q";
55             else//否
56                 cout<<"-";
57         }
58         cout<<endl;
59     }
60 }

```

【代码解析】

第 04~09 行定义各类变量，并初始化有关项，第 11、12 行是两个函数的声明。第 18~21 行是对数组 column、rup 和 lup 的初始化，第 22 行调用函数 backtrack()，最后调用 solution() 显示当前的摆法。

在 backtrack() 中，当 8 个皇后的位置都已确定，解决方法是变量 num 加 1（第 30、31 行），否则继续确定剩余皇后的位置，判断是否都有皇后（第 36 行），先确定皇后位置，然后清除相应位置的皇后（第 38、39 行），接下来执行第 40 行。在 solution() 中，输出当前八个皇后的位置。皇后位置用 Q 代替，其他由一代替。



注意：本实例用一维数组即可实现八皇后位置的摆法，但却使用了较好的算法。



实例 248 百鸡百钱问题

【实例描述】

本实例演示“百鸡百钱问题”，已知用 100 百元人民币买 100 只鸡，鸡的种类共有 3 种，分别为公鸡（每只 5 元）、母鸡（每只 3 元）和小鸡（每只 1/3 元）。本实例给出了解决方案，运行效果如图 13-9 所示。

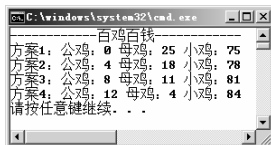


图 13-9 百鸡百钱问题



【实现过程】

本实例在公鸡数量的条件限制下判断最后的总数是否为 100。另外，因为小鸡的单价不是整数，所以它的数量是取 3 的余数必须为 0。代码如下：

```

01  #include <iostream>
02  using namespace std;
03
04  void main()
05  {
06      cout<<"-----百鸡百钱-----"<<endl;
07      int num=100;                                //100 只鸡
08      int money=100;                              //100 元
09      int gong,mu,xiao;                          //公、母、小鸡数量
10      int index=0;                                //方案编号
11      for(gong=0;gong<=20;gong++)                //公鸡界限
12      {
13          for(mu=0;mu<=33;mu++)                  //母鸡数量
14          {
15              xiao=100-gong-mu;                  //小鸡数量
16              if(xiao%3==0&&mu>0&&5*gong+3*mu+xiao/3==100) //满足条件
17              {
18                  cout<<"方案"<<++index<<": ";
19                  cout<<"公鸡: "<<gong<<" 母鸡: "<<mu<<" 小鸡: "<<xiao<<endl;
20              }
21          }
22      }
23  }
```

【代码解析】

第 07~10 行定义各类变量用于计算 3 种鸡的数量，第 11 行为母鸡数量的范围，第 13 行是母鸡数量的界限，第 15 行是小鸡的数量计算，第 16 行是形成解决方案的满足条件。



实例 249 求被 3 整除的数（%+算法）

【实例描述】

本实例实现求被 3 整除的数，它的核心内容是对 3 取余，判断结果是否为 0。如果为 0，可以被整除，反之，不能被整除。本实例提示输入一个整数，然后输出其是否可以被整除，运行效果如图 13-10 所示。

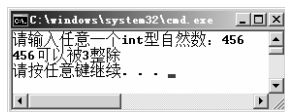


图 13-10 被 3 整除的数

【实现过程】

定义 unsigned int 型变量 aa，然后获取其值，判断其对 3 取余，结果是否为 0，代码如下：

```

01  #include <iostream>
02  using namespace std;
03
04  void main()
05  {
06      unsigned int aa;
07      cout<<"请输入任意一个 int 型自然数: ";
08      cin>>aa;
09      if(aa%3==0)                                //被 3 整除
```



```

10         cout<<aa<<"可以被 3 整除"<<endl;
11     else                                     //不能被 3 整除
12         cout<<aa<<"不可以被 3 整除"<<endl;
13 }

```

【代码解析】

第 06 行定义变量 aa, 第 08 行获取其值, 第 09~12 行判断取余结果是否为 0, 并输出相应的结果。



注意：取余运算符右边的变量值一定不能为 0。



实例 250 鸡兔同笼问题

【实例描述】

本实例模拟古老的鸡兔同笼问题, 即已知一个笼子中头和脚的数量, 求鸡兔各有几只, 运行效果如图 13-11 所示。

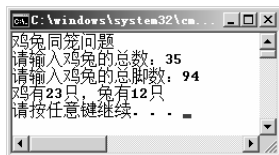


图 13-11 鸡兔同笼问题

【实现过程】

定义 unsigned int 型变量 _tou (表示总头数)、_jiao (表示总脚数), 以及变量 _ji 和 _tu, 分别表示鸡兔各自的数量, 其计算公式如下:

兔的只数 = (总脚数 - 总头数 * 2) / (每只兔的脚数 - 每只鸡的脚数);
鸡的只数 = 总头数 - 兔的只数;

其代码如下:

```

01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     unsigned int _tou, _jiao;           //总的头数和脚数
07     unsigned int _ji, _tu;             //鸡兔各自的数量
08     cout<<"鸡兔同笼问题"<<endl;
09     cout<<"请输入鸡兔的总数: ";
10     cin>>_tou;                         //头数
11     cout<<"请输入鸡兔的总脚数: ";
12     cin>>_jiao;                        //脚数
13     _tu=(_jiao-_tou*2)/(4-2);
14     _ji=_tou-_tu;
15     if((_ji*2+_tu*4)==_jiao)           //如果总数相等
16         cout<<"鸡有"<<_ji<<"只, 兔有"<<_tu<<"只"<<endl;
17     else
18         cout<<"给定总数和总脚数不合逻辑, 请检查."<<endl;
19 }

```

【代码解析】

第 06、07 行定义 4 个变量。第 10、12 行获取总头数和总脚数。第 13、14 行计算鸡兔各自的数量。第 15~18 行判断计算结果是否正确。



注意：本算法的计算都是针对理想情况的，残缺不全的情况不予考虑。



实例 251 求素数

【实例描述】

本实例演示求解素数，即输入一个正整数，判断其是否为素数。素数是指其因子只是 1 和其自身，运行效果如图 13-12 所示。

【实现过程】

定义变量 `range_max` 获取所求素数的最大界限，一维内存变量表示界限内的所有数字。由大于 2 的两个因子相乘作为另一个数，此时该数作为索引值，其数组元素被赋值为 0。最后输出 `range_max` 界限内的所有素数。代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void main()
05 {
06     int range_max; //最大值
07     cout<<"-----判断素数-----"<<endl;
08     cout<<"请输入最大素数检测界限值: ";
09     cin>>range_max;
10     int *num=new int[range_max+1]; //申请内存
11     for(int i=0;i<=range_max;i++) //赋值
12         num[i]=i;
13     for(int j=2;j<=range_max;j++) //由大于 2 的值为一个因子
14     {
15         if(num[j]!=0) //没确定为合数
16         {
17             for(int k=2;k*j<=range_max;k++) //由大于 2 的值为另一个因子
18                 num[k*j]=0; //此值重新赋为 0
19         }
20     }
21     for(int n=2;n<=range_max;n++) //输出范围内的所有素数
22     {
23         if(num[n]!=0) //不为 0 即为素数
24         {
25             static int count=0; //计数
26             cout<<num[n]<<" "; //输出素数
27             count++;
28             if(count%5==0) //5 个数为为一行
29                 cout<<endl;
30         }
31     }
32     cout<<endl;
33     delete []num;
34     num=NULL;
35 }
```



图 13-12 判断素数



【代码解析】

第 06 行定义素数并判定最大界限值,第 09 行获取其值。第 10~12 行申请一维内存,并初始化。第 13~20 行利用两个大于或等于 2 的因子相乘得到一个合数,使其值为 0。第 21~31 行输出范围内的所有素数,其中第 28、29 行使素数的输出以 5 个为一行,最后在第 33、34 行释放一维内存 num。



注意: 1 既不是素数,也不是合数,素数是指大于或等于 2 的正整数。



实例 252 0-1 背包问题（古老数学问题）

【实例描述】

本实例模拟算法古老的数学问题,即 0-1 背包问题。现有 M 件物品,每件物品的重量分别为 W1、W2、…、WM,每件物品代表不同的价值,分别为 V1、V2、…、VM。从中选若干件物品放入可承重为 W 的背包中,现在求出该背包能达到的最大价值。本实例运行效果如图 13-13 所示。

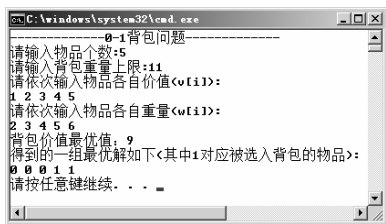


图 13-13 0-1 背包问题

【实现过程】

定义函数 best()求 0-1 背包的最大价值,函数 traceback()求解最大价值的最优解。在 main()函数中定义变量 num、weight_max、value 和 weight,分别表示现有物品个数、背包的最大重量、每件物品代表的价值和重量。代码如下:

```
01 #include<iomanip>
02 #include <iostream>
03 using namespace std;
04
05 void best(int v[],int w[],int c,int n,int**m)           //求最优值
06 {
07     int jmax=min(w[n]-1,c);                             //求最小值
08     for(int j=0;j<=jmax;j++)
09         m[n][j]=0;
10     for(int jj=w[n];jj<=c;jj++)
11         m[n][jj]=v[n];
12     for(int i=n-1;i>1;i--)
13     {
14         jmax=min(w[i]-1,c);
15         for(int j=0;j<=jmax;j++)
16             m[i][j]=m[i+1][j];
17         for(int jj=w[i];jj<=c;jj++)                     //求当前的背包价值
18             m[i][jj]=max(m[i+1][jj],m[i+1][jj-w[i]]+v[i]);
```



```
19     }
20     m[1][c]=m[2][c];
21     if(c>=w[1])
22         m[1][c]=max(m[1][c],m[2][c-w[1]]+v[1]);
23     cout<<"背包价值最优值: "<<m[1][c]<<endl;
24 }
25 void traceback(int **m,int w[],int c,int n,int x[])    //回代, 求最优解
26 {
27     cout<<"得到的一组最优解如下 (其中 1 对应被选入背包的物品):"<<endl;
28     for(int i=1;i<n;i++)    //确定是否入包
29     {
30         if(m[i][c]==m[i+1][c])
31             x[i]=0;
32         else
33         {
34             x[i]=1;
35             c-=w[i];
36         }
37     }
38     x[n]=(m[n][c])?1:0;    //确定是否入包
39     for(int y=1;y<=n;y++)    //输出最优解
40         cout<< x[y]<< " ";
41 }
42 void main()
43 {
44     int num;    //物品个数
45     int weight_max;    //背包重量上限
46     int **m;    //二维内存
47     cout<<"-----0-1 背包问题-----"<<endl;
48     cout<<"请输入物品个数:";
49     cin>>num;
50     cout<<"请输入背包重量上限:";
51     cin>>weight_max;
52     int *value=new int[num+1];    //物品价值
53     cout<<"请依次输入物品各自价值(v[i]):"<<endl;
54     for(int i=1;i<=num;i++)    //从 1 开始到 n
55         cin>>value[i];
56     int *weight=new int[num+1];    //物品各自的重量
57     cout<<"请依次输入物品各自重量(w[i]):"<<endl;
58     for(int j=1;j<=num;j++)    //从 1 开始到 n
59         cin>>weight[j];
60     int *x=new int[num+1];    //一维内存
61     m=new int*[num+1];    //动态的分配二维数组
62     for(int i=0;i<num+1;i++)
63     {
64         m[i]=new int[weight_max+1];
65     }
66     best(value,weight,weight_max,num,m);    //求最优值
67     traceback(m,weight,weight_max,num,x);    //求最优解
68     cout<<endl;
69 }
```

【代码解析】

第 05~24 行是函数 `best()` 的定义体, 它的目的是求解背包的最大价值。第 07 行获取最大价值与物品价值的最小值, 第 08~11 行初始化二维内存 `m` 的最后一列。第 12~19 行求当前背包的价值, 第 21~22 行获取背包能达到的最大价值。第 25~42 行是函数 `traceback()` 的定义体, 第 28~38 行确定每件是否可以入包以达到最优解, 第 39、40 行输出最优解。



第 44~46 行定义变量,第 49 行获取物品个数,第 51 行获取背包重量上限,第 52~59 行定义每个物品的价值及重量,并对其赋值。第 60~65 行定义一维内存 x ,并对二维内存 m 进行初始化。第 66 行调用函数 `best()` 求解背包的最大价值,第 67 行求解进入背包物品的最优解。



注意: 0-1 背包问题是假设一件物品进入背包,判断其当前最大价值及重要程度是否满足条件,如果不满足,则去掉当前物品,再放入另一件物品。如果满足,则继续放入物品,再判断。



实例 253 扫雷游戏 1

【实例描述】

本实例和实例 254 共同实现扫雷游戏,本实例实现扫雷游戏类的声明及雷盘输出函数的定义。设置雷盘的行数为 9,列数为 9,雷的个数为 9,其输出效果如图 13-14 所示。



图 13-14 雷盘输出

【实现过程】

定义类 `Game` 实现扫雷游戏,其有成员函数 `CreatePane()` 和 `PrintPane()`,分别用于创建雷盘和绘制雷盘功能。首先列出类 `Game` 的声明,代码如下:

```
01 #include <iostream>
02 #include <ctime>
03 using namespace std;
04
05 class Game
06 {
07 public:
08     Game();
09     ~Game();
10     void CreatePane();           //布局雷区
11     void PrintPane();           //画雷区
12 private:
13     int _row;                   //行
14     int _col;                   //列
15     int _lei;                   //雷个数
```



```
16     int **zone;                //布局
17 };
```

接下来, 列出 `main()` 函数代码, 如下:

```
01 void main()
02 {
03     cout<<"-----扫雷游戏-----"<<endl;
04     Game game;                  //游戏对象
05     srand((unsigned)time(NULL)); //种子值
06     game.CreatePane();          //创建扫雷布局
07     game.PrintPane();           //画雷盘
08 }
```

由于在成员变量中有动态申请的二维内存, 所以在析构函数中需要释放, 代码如下:

```
01 Game::Game()
02 {
03     _row=9;                      //行列的初始化
04     _col=9;
05     _lei=10;                     //雷的个数
06     zone=new int *[_row];        //二维内存申请
07     for(int i=0;i<_row;i++)
08         zone[i]=new int[_col];
09 }
10 Game::~Game()
11 {
12     if(zone!=NULL)
13     {
14         delete[] zone;
15         zone=NULL;
16     }
17 }
```

函数 `CreatePane()` 和 `PrintPane()` 的代码如下:

```
01 void Game::CreatePane()
02 {
03     int rtemp, ctemp;            //雷区的行列
04     while(_lei>0)                //雷的个数不为 0
05     {
06         rtemp=0+(int) (_row*rand()/(RAND_MAX+0)); //求雷区的行
07         ctemp=0+(int) (_col*rand()/(RAND_MAX+0)); //求雷区的列
08         if(zone[rtemp][ctemp]!=-1)                //不是雷区
09         {
10             zone[rtemp][ctemp]=-1;                //设为雷区
11             _lei--;                                //雷个数减 1
12         }
13     }
14 }
15 void Game::PrintPane()
16 {
17     cout<<" ";
18     for(int i=0;i<_col;i++)
19         cout<<i<<" "; //画出列数标记
20     cout<<endl;
21     for(int i=0;i<_row;i++)
22     {
23         cout<<" -----\n";
24         cout<<i<<" | "; //输出行标
25         for(int j=0;j<_col;j++)
```



```

26         {
27             cout.width(2);                                //输出宽度设置
28             if(zone[i][j]>-1)
29             {
30                 if(zone[i][j]==0)                        //周围无雷
31                     cout<<" ";
32                 else                                     //输出周围雷的个数
33                     cout<<zone[i][j];
34             }
35             else if(zone[i][j]==-1)                       //雷区
36                 cout<<"XX";
37             else                                         //未开发区
38                 cout<<"XX";
39             cout<<" | ";                                //输出分隔线
40         }
41         cout<<endl;
42     }
43     cout<<" -----\n";
44 }

```

【代码解析】

第 1 段代码中，第 08~11 行是其成员函数，第 13~16 行是其私有成员变量。

第 2 段代码中，第 04 行定义游戏类对象 game，第 05 行为雷盘中雷位置的随机性创建种子。第 06、07 行创建雷盘和绘制雷盘。

第 3 段代码中，第 01~09 行是构造函数，是对成员变量的初始化。第 10~17 行是析构函数，释放二维内存。

第 4 段代码中，第 01~14 行是创建雷盘函数，根据雷的个数设置雷所在行列的随机值，直到未配置雷为 0。第 15~44 行输出每次扫雷后当前雷盘的状况，其具体内容请看注释，在此不赘述。



注意：对于二维内存，在构造函数中没有对其进行初始化，是为了在玩游戏的过程中可以根据当前元素的值不断绘制当前的雷盘。



实例 254 扫雷游戏 2

【实例描述】

本实例接实例 253，实现雷盘中雷的排除操作。用户输入行列号，由计算机判断该位置是否有雷。如果不是雷且周围也没雷，输出空格。如果不是雷，但周围有雷，输出雷的个数。如果是雷，输出游戏的输赢。运行效果如图 13-15 所示。

【实现过程】

在实例 253 中，main()函数的最后调用 Play()成员函数，开始扫雷游戏。其添加成员函数 Play()和 CheckWin()的代码如下：

```

02     {
03     public:
04         ...
05         void Play();

```

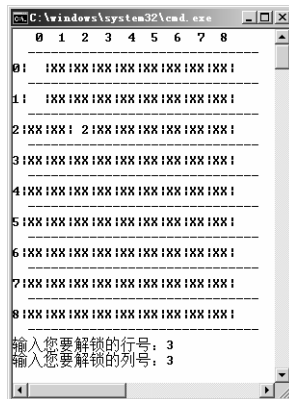


图 13-15 扫雷游戏的开始

//开始游戏



```
06         bool CheckWin();                                //是否成功
07     ...
08 };
09 ...
10 void Game::Play()
11 {
12     int win=0;                                           //标志是否赢
13     while(!win)                                         //没赢
14     {
15         int r,c,br,bc,er,ec,mines=0;                  //各种变量
16         cout<<"输入您要解锁的行号: ";
17         cin>>r;                                          //输入行
18         cout<<"输入您要解锁的列号: ";
19         cin>>c;                                          //输入列
20         if(zone[r][c]==-1)                             //扫到雷
21         {
22             cout<<"扫到雷, 你输了!\n";
23             break;                                       //退出, 游戏结束
24         }
25         else
26         {
27             if(r<2)                                     //第0/1行
28             {
29                 br=0;
30                 er=r+1;
31             }
32             else if(r>7)                                //最后两行
33             {
34                 br=r-1;
35                 er=_row-1;
36             }
37             else                                        //中间行
38             {
39                 br=r-1;
40                 er=r+1;
41             }
42             if(c<2)                                     //第0/1列
43             {
44                 bc=0;
45                 ec=c+1;
46             }
47             else if(c>7)                                //最后两列
48             {
49                 bc=c-1;
50                 ec=_col-1;
51             }
52             else                                        //中间列
53             {
54                 bc=c-1;
55                 ec=c+1;
56             }
57         }
58         mines=0;                                         //当前位置周围有雷的个数
59         for(;br<=er;br++)
60         {
61             for(int x=bc;x<=ec;x++)
62             {
63                 cout<<br<<" "<<x<<endl;
64                 if(zone[br][x]==-1)
65                     mines++;                             //该位置周围有雷的个数
```



```

66         }
67     }
68     zone[r][c]=mines;           //赋值
69     win=CheckWin();             //判断是否还有雷
70     system("cls");              //清屏
71     PrintPane();                //输出
72     if(win)                     //没雷
73         cout<<"\n\n 你赢了!\n";
74 }
75 }
76 bool Game::CheckWin()           //检测是否成功
77 {
78     for(int r=0;r<_row;r++)
79     {
80         for(int c=0;c<_col;c++)
81         {
82             if(zone[r][c]<-1)     //还有雷
83                 return false;
84         }
85     }
86     return true;                //无雷
87 }
88 void main()
89 {
90     ...
91     game.Play();                 //游戏开始
92 }

```

【代码解析】

第 05、06 行是新添加的成员函数，第 10~76 行是函数 Play() 的定义体。其中第 12 行变量 win 标志是否赢得游戏，如果没有赢，则继续执行 while 循环内的代码。第 17、19 行输入所查行列数，第 20 行判断是否扫到雷。如果没有扫到雷，则执行 else 体内代码，根据输入行列值，对变量 br、er、bc 和 ec 进行赋值，它的意义是用于判断该点周围有多少个雷。

第 59~67 行计算周围雷数，并赋值给当前点（第 68 行），第 69 行调用函数 CheckWin() 判断是否还有雷没有排查，第 70、71 行先清屏，再输出雷盘。如果没有雷，输出第 73 行。第 76~87 行是函数 CheckWin() 的定义体，由第 78~85 行的双重 for 循环检测当前雷盘内是否还有雷。



注意：第 27~56 行是统计本元素附近雷的个数，确定行列的检测范围。



实例 255 因式分解

【实例描述】

本实例实现因式分解问题，输入一个正整数，通过从 2 到当前最大因数为止，对正整数做除法运算，其运行效果如图 13-16 所示。

【实现过程】

定义整型变量 num 和 i，分别表示被分解数和因数，利用 for 循环对变量 num 在 2~num 范围内做除法运算。具体代码如下：

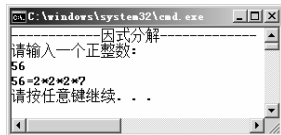


图 13-16 因式分解问题



```

01  #include <iostream>
02  using namespace std;
03
04  void main()
05  {
06      cout<<"-----因式分解-----\n";
07      int num,i;                                //被分解数, 因数
08      cout<<"请输入一个正整数:\n";
09      cin>>num;                                //输入
10      cout<<num<<"=";
11      for (i=2;i<=num;i++)                    //从 2 开始到 num 为止
12      {
13          while(num!=i)                        //没截止
14          {
15              if(num%i==0)                    //可被整除
16              {
17                  cout<<i<<"*";
18                  num /= i;                    //除
19              }
20              else                             //不可整除
21                  break;
22          }
23      }
24      cout<<num<<"\n";
25  }

```

【代码解析】

第 07 行定义变量 `num` 和 `i`, 第 09 行获取 `num` 的值。第 11~23 行的 `for` 循环用 `2~num` 的因子对当前 `num` 值做除法。其中, 第 13 行 `while` 循环判断因子是否与当前 `num` 值相同, 在 `while` 循环中, 如果 `num` 对 `i` 取余为 0, 则 `i` 视为因子 (第 17 行); 否则退出循环 (第 21 行)。



注意: 每做一个除法运算时, 当前 `num` 值除以当前的 `i` 值, 如第 18 行。



实例 256 爱因斯坦台阶问题

【实例描述】

爱因斯坦问题是假设某人走一个台阶, 如果每步走两级, 最后只剩 1 级, 如果每步走 3 级, 最后剩两级, 如果每步走 4 级, 最后剩 3 级, 如果每步走 5 级, 最后剩 4 级, 如果每步走 6 级, 最后剩 5 级, 如果每步走 7 级, 则一个也不剩。本实例求出 1000 以内符合该条件的数字, 实例运行效果如图 13-17 所示。

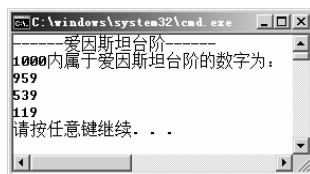


图 13-17 爱因斯坦台阶问题

【实现过程】

本实例抽象为数学问题是: 总的台阶数分别对 2、3、4、5、6、7 取余的结果是 1、2、3、4、5、0。具体代码如下:

```

01  #include<iostream>
02  using namespace std;
03
04  bool Is_right(int n)

```



```

05  {
06      for(int i=1;i<7;i++)          //循环 6 次
07      {
08          if(i==6)                  //7
09          {
10              if(n%(i+1)!=0)
11                  return false;
12          }
13          else                      //2~6
14          {
15              if(n%(i+1)!=i)        //不符合
16                  return false;
17          }
18      }
19      return true;                  //成立
20  }
21  void main()
22  {
23      cout<<"-----爱因斯坦台阶-----"<<endl;
24      cout<<"1000 内属于爱因斯坦台阶的数字为: "<<endl;
25      int max=1000;                  //最大值
26      while(max)                    //1000 个数循环
27      {
28          if(!Is_right(max))        //不符合
29          {}
30          else                      //符合
31              cout<<max<<endl;      //输出
32          max--;                    //减 1
33      }
34  }

```

【代码解析】

第 04~20 行定义函数 `Is_right()`，用于判断输入参数是否符合上述条件，第 08 行判断对 7 取余，第 13 行判断对其他数取余。第 25 行设置对 1000 以内的数进行判断，第 28 行判断不符合要求，第 30 行判断符合要求，输出该数（第 31 行），然后对 `max` 值减 1。



注意：第 08~17 行的判断可以利用三相运算符实现，减少代码量。



实例 257 巧算 24 点问题

【实例描述】

算 24 点是从一副牌中任意取 4 张牌（除大小王外），用加、减、乘、除和括号运算符使这 4 个数经过运算后成为 24，且每个数只能被使用 1 次。本实例输入 4 个数，求其是否能经过运算后形成 24，并输出运算式，运行效果如图 13-18 所示。

【实现过程】

宏定义数字个数、结果值和精度要求，变量名为 `NUM`、`RESULT` 和 `PRECISION`。定义函数 `Cal()` 用于求解构成 24 的计算公式，具体代码如下：

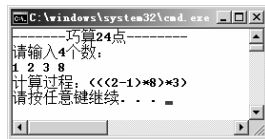


图 13-18 巧算 24 点问题



```
01 #include <iostream>
02 #include <string>
03 #include <cmath>
04 using namespace std;
05
06 #define NUM 4 //4 个数
07 #define RESULT 24 //结果 24
08 #define PRECISION 1E-6 //精度
09
10 bool Cal(int n,double* number,string* equation)
11 {
12     if(n==1) //计算结束
13     {
14         if(fabs(number[0]-RESULT)<PRECISION) //小于精度
15             return true;
16         else
17             return false;
18     }
19     for(int i=0;i<n-1;i++) //没结束，继续循环
20     {
21         for(int j=i+1;j<n;j++)
22         {
23             double a, b;
24             string expa, expb;
25             a = number[i];
26             b = number[j];
27             //挪动后面的有效数字
28             number[j] = number[n-1];
29             expa = equation[i];
30             expb = equation[j];
31             equation[j] = equation[n-1];
32             //a+b
33             equation[i] = '(' + expa + '+' + expb + ')';
34             number[i] = a+b;
35             if(Cal(n-1,number,equation))
36                 return true;
37             //a-b
38             equation[i] = '(' + expa + '-' + expb + ')';
39             number[i] = a-b;
40             if(Cal(n-1,number,equation))
41                 return true;
42             //b-a
43             equation[i] = '(' + expb + '-' + expa + ')';
44             number[i] = b-a;
45             if(Cal(n-1,number,equation))
46                 return true;
47             //(a*b)
48             equation[i] = '(' + expa + '*' + expb + ')';
49             number[i] = a*b;
50             if(Cal(n-1,number,equation))
51                 return true;
52             //(a/b)
53             if (b!=0) //除数不为 0
54             {
55                 equation[i] = '(' + expa + '/' + expb + ')';
56                 number[i] = a/b;
57                 if(Cal(n-1,number,equation))
58                     return true;
59             }
60             //(b/a)
61             if (a!=0) //除数不为 0
62             {
```



```

63         equation[i] = '(' + expb + '/' + expa + ')';
64         number[i] = b/a;
65         if(Cal(n-1,number,equation))
66             return true;
67     }
68     //恢复
69     number[i] = a;
70     number[j] = b;
71     equation[i] = expa;
72     equation[j] = expb;
73 }
74 }
75 return false;
76 }
77 void main()
78 {
79     double a[NUM];                //4个数
80     string eq[NUM];               //公式
81     cout<<"-----巧算 24 点-----"<<endl;
82     cout<<"请输入 4 个数: "<<endl;
83     for(int i=0; i<NUM; i++)      //输入 4 个数
84     {
85         char buffer[20];
86         int x;
87         cin >> x;
88         a[i] = x;
89         itoa(x, buffer, 10);      //整数变字符串
90         eq[i] = buffer;           //string
91     }
92     if(Cal(NUM,a,eq))             //运算成功
93         cout<<"计算过程: "<<eq[0]<<endl;    //输出结果
94     else
95         cout<<"该 4 个数构不成 24"<<endl;
96 }

```

【代码解析】

第 06~08 行宏定义 3 个变量，第 10~76 行定义函数 Cal()，其第 1 个参数是计入计算的数字个数，第 2 个参数是 4 个数字的值，第 3 个参数是 4 个公式字符串。第 12~18 行是当 n 值为 1 时，已经进入最后运算阶段，当小于精度时（第 14 行），返回真。反之，返回假，即经过运算不能形成 24。第 19~74 行进行循环运算，其中，分别进行加、减、乘、除、加括号运算，并实时更新计入运算的数字。

在 main() 函数中，第 79~80 行定义数组 a 和 eq，分别表示 4 个数和公式。第 83~91 行获取 4 个数字，并将整数变为字符串。第 92~95 行判断是否成功计算为 24，并输出对应的结果。



注意：在 cmath.h 的 fabs() 函数中，没有对 int 型变量进行运算，所以在声明数字数组时，定义为 double 型，为免于强制转换。但在数字转换为字符串时，采用 int 型，如第 86~90 行。

第 14 章 多线程、动态链接库

本章介绍多线程及动态链接库的创建及应用，多线程内容包括设置线程的优先级、悬挂和恢复。在介绍上述内容后，继续演示如何实现线程间的同步。对于动态链接库，本章演示如何使用_declspec(dllexport)和.def 方式导出函数和类，以及隐式和显式调用导出类及函数。最后，针对显式调用动态链接的错误，给出了解决方式。



实例 258 创建多线程

【实例描述】

在一个进程中有多个线程可以并行运行，当只有一个线程时被称为单线程。本实例模拟两个线程的应用，运行效果如图 14-1 所示。

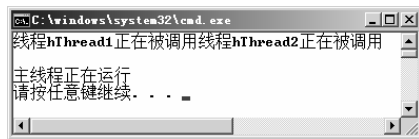


图 14-1 多线程应用

【实现过程】

定义函数 ThreadProc1()和 ThreadProc2()，分别作为两个线程的执行函数，并输出各自线程正被调用，具体代码如下：

```
01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 HANDLE hThread1; //线程 1 句柄
06 HANDLE hThread2; //线程 2 句柄
07
08 DWORD WINAPI ThreadProc1(LPVOID lpParameter)
09 {
10     cout<<"线程 hThread1 正在被调用"<<endl;
11     return 0;
12 }
13 DWORD WINAPI ThreadProc2(LPVOID lpParameter)
14 {
15     cout<<"线程 hThread2 正在被调用"<<endl;
16     return 0;
17 }
18 void main()
19 {
20     hThread1=CreateThread(NULL,0,ThreadProc1,NULL,0,NULL); //创建线程 1
21     hThread2=CreateThread(NULL,0,ThreadProc2,NULL,0,NULL); //创建线程 2
22     Sleep(1000); //睡一会儿
23     CloseHandle(hThread1); //关闭线程 1
24     CloseHandle(hThread2); //关闭线程 2
25     cout<<"主线程正在运行"<<endl;
26 }
```




```

33      CloseHandle(hThread2);                      //关闭线程句柄
34      cout<<"主线程正在运行"<<endl;
35  }

```

【代码解析】

第 05、06 行定义两个线程的句柄，第 07~15 行是线程 1 的执行函数，第 16~24 行是线程 2 的执行函数，第 28 行设置线程 1 的优先级为空闲，第 30 行设置线程 2 的优先级为最高。



注意：设置线程的优先级可用函数 `SetThreadPriority()`，它可以被应用于任何地方。



实例 260 悬挂和恢复线程

【实例描述】

本实例模拟如何悬挂和恢复线程，当线程 1 遇全局变量对 2 取余为 0 时，悬挂自己。当线程 2 遇全局变量对 9 取余为 0 时，恢复线程 1。实例运行效果如图 14-3 所示。

【实现过程】

定义两个可悬挂的线程，以及其执行函数 `ThreadProc1()` 和 `ThreadProc2()`，具体代码如下：

```

01  #include <windows.h>
02  #include <iostream>
03  using namespace std;
04
05  HANDLE hThread1;                      //定义句柄 1
06  HANDLE hThread2;                      //定义句柄 2
07  int i=1;                              //全局变量
08  bool flag1=true;
09  bool flag2=true;
10  DWORD WINAPI ThreadProc1(LPVOID lpParameter)  //线程 1
11  {
12      while(flag1)
13      {
14          if(i<50)                      //界值 50
15          {
16              cout<<"进入线程 1。。 " <<endl;
17              cout<<"线程 hThread1 当前访问的 i 值: " <<i <<endl;
18              if(i%2==0)
19              {
20                  SuspendThread(hThread1);    //挂起自己
21                  cout<<"挂起线程 1" <<endl;
22              }
23              i*=2;                        //每次乘以 2
24          }
25          else
26              flag1=false;
27      }
28      return 0;
29  }

```



图 14-3 悬挂和恢复线程



```

30  DWORD WINAPI ThreadProc2(LPVOID lpParameter)           //线程 2
31  {
32      while(flag2)
33      {
34          if(i<100)                                       //界值 100
35          {
36              cout<<"进入线程 2。。。"<<endl;
37              cout<<"线程 hThread2 当前访问的 i 值: "<<i<<endl;
38              if(i%9==0)
39              {
40                  ResumeThread(hThread1);               //恢复线程 hThread1
41                  cout<<"恢复线程 1"<<endl;
42              }
43              i*=3;                                       //每次乘以 3
44          }
45          else
46              flag2=false;
47      }
48      return 0;
49  }
50  void main()
51  {
52      hThread1 = CreateThread(NULL,0,ThreadProc1,NULL,CREATE_SUSPENDED,NULL);
53      ResumeThread(hThread1);                             //恢复线程
54      hThread2 = CreateThread(NULL,0,ThreadProc2,NULL,CREATE_SUSPENDED,NULL);
55      ResumeThread(hThread2);
56      Sleep(5000);                                         //睡一会儿
57      CloseHandle(hThread1);                             //关闭线程句柄
58      CloseHandle(hThread2);                             //关闭线程句柄
59      cout<<"主线程正在运行"<<endl;
60  }

```

【代码解析】

第 05、06 行定义两个线程句柄，第 07 行定义全局变量 *i*，第 08、09 行定义线程的标志量。第 10~29 行定义线程 1 的执行函数，其中，第 18 行为真时挂起线程 1。第 30~49 行定义线程 2 的执行函数，其中，第 38 行为真时恢复线程 1。第 52~55 行定义可悬挂的线程 1 和线程 2。



注意：由运行效果图可以知，悬挂线程输出在恢复线程之后。这样，线程之间并没有达到同步，因此，后几例给出线程同步的不同实现。



实例 261 利用临界区实现线程同步

【实例描述】

运用临界区可以实现线程同步，即线程访问公共资源必须一个一个地进行。本例定义两个线程，轮番输出全局变量的值，运行效果如图 14-4 所示。

【实现过程】

定义两个线程和全局变量 *i*，线程 1 每次对 *i* 进行乘 2 运算，而线程 2 每次对 *i* 进行乘 3 运



图 14-4 临界区应用



算。具体代码如下：

```
01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 CRITICAL_SECTION cs;           //临界区结构对象
06 int i = 1;                     //共享资源
07
08 DWORD WINAPI ThreadProc1(LPVOID lpParameter)    //线程 1
09 {
10     while(true)
11     {
12         EnterCriticalSection(&cs);             //进入临界区
13         if(i<50)
14         {
15             cout<<"线程 hThread1 当前访问的 i 值: "<<i<<endl;
16             i*=2;                                //每次乘以 2
17         }
18         else
19             break;                               //退出
20         LeaveCriticalSection(&cs);              //离开临界区
21     }
22     return 0;
23 }
24
25 DWORD WINAPI ThreadProc2(LPVOID lpParameter)    //线程 1
26 {
27     while(true)
28     {
29         EnterCriticalSection(&cs);             //进入临界区
30         if(i<50)
31         {
32             cout<<"线程 hThread2 当前访问的 i 值: "<<i<<endl;
33             i*=3;                                //每次乘以 3
34         }
35         else
36             break;                               //退出
37         LeaveCriticalSection(&cs);              //离开临界区
38     }
39     return 0;
40 }
41
42 void main()
43 {
44     cout<<"-----临界区应用-----"<<endl;
45     InitializeCriticalSection(&cs);             //初始临界区
46     HANDLE hThread1;                            //定义线程 1
47     HANDLE hThread2;                            //定义线程 2
48     hThread1=CreateThread(NULL,0,ThreadProc1,NULL,0,NULL); //创建线程 1
49     hThread2=CreateThread(NULL,0,ThreadProc2,NULL,0,NULL); //创建线程 2
50     CloseHandle(hThread1);                       //关闭线程 1
51     CloseHandle(hThread2);                       //关闭线程 2
52     Sleep(5000);                                 //主线程睡一会儿
53     cout<<"主线程正在运行"<<endl;
54     DeleteCriticalSection(&cs);                 //释放临界区
55 }
```

【代码解析】

第 05 行定义临界区结构对象 cs，第 06 行定义全局变量 i（初始化为 1）。第 08~23 行定义



线程 1 执行函数 ThreadProc1(), 其第 12 行进入临界区, 第 13 行判断变量 i 是否符合条件。第 16 行进行乘 2 运算, 然后在第 20 行离开临界区。

第 24~39 行是线程 2 的执行函数 ThreadProc2(), 它对 i 进行乘 3 运算 (如第 32 行)。第 43 行初始临界区, 第 44~47 行初始化两个线程, 第 52 行释放临界区。



注意: 由效果图可知, 本程序只是由一个线程访问公共资源, 另一个线程没有机会进入临界区。具体原因在实例 262 中解释。



实例 262 预防单个线程霸占资源

【实例描述】

使用临界区执行线程同步的速度非常快, 一个线程在退出临界区后还没等另一个线程反应过来, 就已经重复抢夺到资源。因此造成了只有一个线程在运行, 本实例在每个线程退出临界区后都睡一会儿觉, 再开始抢夺资源。运行效果如图 14-5 所示。



图 14-5 预防单个线程霸占资源

【实现过程】

在每个线程退出临界区后添加睡觉函数 Sleep(), 具体代码如下:

```
01  ...
02  DWORD WINAPI ThreadProc1(LPVOID lpParameter)           //线程1
03  {
04      while(true)
05      {
06          ...
07          LeaveCriticalSection(&cs);                     //离开临界区
08          Sleep(1000);                                    //睡一会儿
09      }
10      return 0;
11  }
12  DWORD WINAPI ThreadProc2(LPVOID lpParameter)           //线程1
13  {
14      while(true)
15      {
16          ...
17          LeaveCriticalSection(&cs);                     //离开临界区
18          Sleep(1000);                                    //睡一会儿
19      }
20      return 0;
21  }
22  void main()
23  {
24      ...
25  }
```

【代码解析】

本实例的重点代码如第 08、18 行的睡觉 1 秒。



注意：事实上，不同的计算机在不同的时刻运行该程序的结果也不同。



实例 263 利用事件实现线程同步

【实例描述】

本实例利用事件使线程 1 和线程 2 同步访问公共资源，以使每个线程都能得到更新后的公共资源状态。本实例利用两个线程依次获取当前全局变量的值，运行效果如图 14-6 所示。



图 14-6 事件实现线程同步

【实现过程】

定义两个线程执行函数，访问全局变量 *i* 的当前值。在线程 1 中，当 *i* 值不小于 5 时停止输出，在线程 2 中，当 *i* 值不小于 20 时停止输出。代码如下：

```
01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 int i = 1; //全局变量
06 HANDLE hEvent; //事件
07
08 DWORD WINAPI ThreadProc1(LPVOID lpParameter) //线程 1
09 {
10     while(true)
11     {
12         WaitForSingleObject(hEvent, INFINITE);
13         if(i<5) //界值 5
14         {
15             cout<<"线程 hThread1 当前访问的 i 值: "<<i<<endl;
16             i*=2; //每次乘以 2
17         }
18         else
19             break; //退出
20         SetEvent(hEvent);
21     }
22     return 0;
23 }
24 DWORD WINAPI ThreadProc2(LPVOID lpParameter) //线程 2
25 {
26     while(true)
27     {
28         WaitForSingleObject(hEvent, INFINITE);
29         if(i<20) //界值 20
30         {
31             cout<<"线程 hThread2 当前访问的 i 值: "<<i<<endl;
32             i*=2; //每次乘以 2
33         }
34         else
35             break; //退出
36         SetEvent(hEvent);
37     }
38     return 0;
39 }
```



```

40
41 void main()
42 {
43     HANDLE hThread1;                //定义句柄
44     HANDLE hThread2;                //定义句柄
45     hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, NULL);
46     hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, NULL); //创建线程
47     CloseHandle(hThread1);          //关闭线程 1
48     CloseHandle(hThread2);          //关闭线程 2
49     hEvent = CreateEvent(NULL, false, false, NULL); //创建事件
50     if(hEvent)
51     {
52         if(GetLastError() == ERROR_ALREADY_EXISTS) //创建事件有错
53         {
54             cout<<"该事件已经存在!"<<endl;
55             return;
56         }
57     }
58     SetEvent(hEvent);                //设置事件
59     Sleep(4000);
60     CloseHandle(hEvent);            //关闭事件句柄
61 }

```

【代码解析】

第 05、06 行定义全局变量 `i` 和事件变量 `hEvent`，第 08~23 行和第 24~39 行分别是执行函数 `ThreadProc1()` 和 `ThreadProc2()` 的定义体。其中，第 12、28 行是等待事件对象的信号状态，只有存在信号时才访问资源。第 49 行创建事件对象 `hEvent`，第 58 行设置事件信号，第 60 行关闭事件句柄。



实例 264 解析事件实现线程同步的原理

【实例描述】

由实例 263 的效果图可知，当线程 1 已满足继续访问 `i` 值时，线程 2 不能访问 `i`。原因是线程 1 在 `break` 后会处于无限循环，线程 2 根本没有机会访问变量 `i`。解析过程如图 14-7 所示。

【实现过程】

当不满足线程执行条件时，将 `while` 循环的条件置为假，并输出每个线程的访问情况，代码如下：

```

01 ...
02 bool th1_flag=true;                //线程 1
03 bool th2_flag=true;                //线程 2
04 DWORD WINAPI ThreadProc1(LPVOID lpParameter) //线程 1
05 {
06     while(th1_flag)
07     {
08         WaitForSingleObject(hEvent, INFINITE);
09         cout<<"进入线程 1....."<<endl;
10         ...
11     }
12     else

```

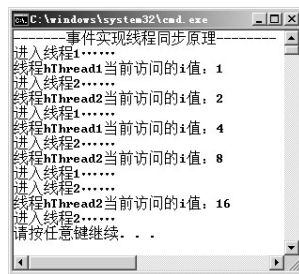


图 14-7 线程同步事件原理



```

12         th1_flag=false;                //置假
13         SetEvent(hEvent);
14     }
15     return 0;
16 }
17 DWORD WINAPI ThreadProc2(LPVOID lpParameter)    //线程 2
18 {
19     while(th2_flag)
20     {
21         WaitForSingleObject(hEvent, INFINITE);
22         cout<<"进入线程 2....."<<endl;
23         ...
24         else
25             th2_flag=false;            //置假
26         SetEvent(hEvent);
27     }
28     return 0;
29 }
30 void main()
31 {
32     ...
33 }

```

【代码解析】

第 02、03 行定义有效进入线程执行函数的标志量，初始为真。第 06、19 行表示当为真时，继续执行代码。第 12、25 行表示当条件不满足时，置标志量的值为假。此时，线程 1 不会进入无限循环。



注意：如果循环条件一直为 true，当条件不满足时，线程执行永远不会执行代码 SetEvent(hEvent)。



实例 265 利用互斥量实现线程同步

【实例描述】

顾名思义，互斥量是指线程 1 进行公共资源后不允许其他线程进入。本实例利用互斥量实现线程间的同步，具体效果与实例 264 相同，如图 14-8 所示。



图 14-8 互斥量实现线程同步

【实现过程】

在实例 264 的基础上，改变线程同步实现变量。本实例定义互斥量 hMutex，在使用完当前所有权时，使用函数 ReleaseMutex()释放，具体代码如下：

```

01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 HANDLE hMutex;                //互斥量
06 int i = 1;                    //公共资源
07
08 DWORD WINAPI ThreadProc1(LPVOID lpParameter)    //线程 1
09 {
10     while(true)

```



```

11     {
12         WaitForSingleObject(hMutex, INFINITE);
13         if(i<5)
14         {
15             cout<<"线程 hThread1 当前访问的 i 值: "<<i<<endl;
16             i*=2;                                //每次乘以 2
17         }
18         else
19             break;                                //退出
20         ReleaseMutex(hMutex);                    //释放互斥对象的所有权
21     }
22     return 0;
23 }
24 DWORD WINAPI ThreadProc2(LPVOID lpParameter)    //线程 2
25 {
26     while(true)
27     {
28         WaitForSingleObject(hMutex, INFINITE);
29         if(i<50)
30         {
31             cout<<"线程 hThread2 当前访问的 i 值: "<<i<<endl;
32             i*=2;                                //每次乘以 2
33         }
34         else
35             break;                                //退出
36         ReleaseMutex(hMutex);                    //释放互斥对象的所有权
37     }
38     return 0;
39 }
40 void main()
41 {
42     cout<<"-----互斥量实现线程同步-----"<<endl;
43     hMutex = CreateMutex(NULL, false, NULL);      //创建互斥量
44     ...
45     WaitForSingleObject(hMutex, INFINITE);        //等待互斥量状态
46     ReleaseMutex(hMutex);                        //释放互斥量
47     Sleep(4000);
48     cout<<"主线程正在运行"<<endl;
49 }

```

【代码解析】

第 05、06 行定义互斥量及公共资源 i，第 08~23 行、第 24~39 行分别定义线程的执行函数，第 43 行创建互斥对象，第 45 行等待互斥量的状态，第 46、47 行释放两个线程当前拥有互斥对象。



注意：可以看出，即使是无限循环，互斥量也不会陷入死循环。因为互斥量在最后会释放所有权，而不是置某个事件有效。所以，另一个线程仍可以继续访问。



实例 266 利用信号量实现线程同步

【实例描述】

本实例利用信号量实现线程同步，在每次进出线程执行函数时拥有资源所有权、释放资源



所有权。运行效果如图 14-9 所示。

【实现过程】

信号量的应用与事件对象相似，因此，while 用于循环判断标志量，以决定可否继续执行。限定线程 1 的条件是 i 小于 5，线程 2 的条件是 i 小于 20，具体代码如下：

```

01  #include <windows.h>
02  #include <iostream>
03  using namespace std;
04
05  bool idle = true;           //标志资源是否可被访问
06  int i=1;                   //初始值
07  HANDLE hDemaphore;         //信号量
08  bool th1_flag=true;        //线程 1 标志
09  bool th2_flag=true;        //线程 2 标志
10  DWORD WINAPI ThreadProc1(LPVOID lpParameter) //线程 1
11  {
12      while(th1_flag)
13      {
14          WaitForSingleObject(hDemaphore, INFINITE);
15          cout<<"访问线程 1... "<<endl;
16          if(i<5)
17          {
18              if(idle)           //空闲
19              {
20                  cout<<"线程 hThread1 当前访问的 i 值: "<<i<<endl;
21                  idle=false;    //不可访问
22                  Sleep(1000);   //使用期 1 秒
23                  idle=true;     //可被访问
24              }
25              i*=2;
26          }
27          else
28              th1_flag=false;
29          ReleaseSemaphore(hDemaphore, 1, NULL); //释放信号
30      }
31      return 0;
32  }
33  DWORD WINAPI ThreadProc2(LPVOID lpParameter) //线程 2
34  {
35      while(th2_flag)
36      {
37          WaitForSingleObject(hDemaphore, INFINITE);
38          if(i<20)
39          {
40              if(idle)           //空闲
41              {
42                  cout<<"线程 hThread2 当前访问的 i 值: "<<i<<endl;
43                  idle=false;    //不可访问
44                  Sleep(1000);   //使用期 1 秒
45                  idle=true;     //可被访问
46              }
47              i*=2;
48          }
49          else
50              th2_flag=false;

```

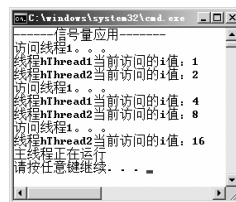


图 14-9 信号量实现线程同步



```

51         ReleaseSemaphore(hDemaphore, 1, NULL);           //释放信号
52     }
53     return 0;
54 }
55 void main()
56 {
57     cout<<"-----信号量应用-----"<<endl;
58     hDemaphore = CreateSemaphore(NULL, 1, 1, NULL);       //信号量对象
59     //hDemaphore = CreateSemaphore(NULL, 2, 3, NULL);
60     //最大信号量为 3, 但只能开通 2 个
61     HANDLE hThread1;                                     //定义句柄
62     HANDLE hThread2;                                     //定义句柄
63     hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, NULL);
64     hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, NULL); //创建线程
65     CloseHandle(hThread1);                               //关闭线程句柄
66     CloseHandle(hThread2);
67     WaitForSingleObject(hDemaphore, INFINITE);
68     ReleaseSemaphore(hDemaphore, 1, NULL);
69     Sleep(5000);
70     cout<<"主线程正在运行"<<endl;
71 }

```

【代码解析】

第 05~07 行定义 3 个全局变量, 第 08、09 定义每个线程的标志量。第 10~32 行、第 33~54 行分别为线程 1 和线程 2 执行函数的定义体。以其中一个函数为例解释, 第 16 行标定资源小于 5 次可被线程访问, 第 18 行表示当前资源处于空闲可被访问。第 20 行输出当前资源的值, 第 21 行表示资源不可被访问, 第 22 行睡一会儿后再标识资源可被访问 (如第 44 行)。第 25 行标识变量 i 的乘法, 第 29 行释放信号量。第 58 行定义信号量对象, 第 60~63 行定义两个线程, 第 66 行等待信号处于有效。



注意: 第 58 行表示只有 1 个资源, 且同一时刻只允许 1 个线程访问。第 59 行表示有 3 个资源, 但同一时刻只允许两个线程访问。



实例 267 自定义消息实现线程间通信

【实例描述】

线程间的通信可以通过多种手段实现, 比如全局变量、消息传送和事件对象。本实例利用自定义消息实现线程间通信, 线程 1 每隔 1 秒发送空闲消息, 线程 2 每隔 1 秒发送繁忙信息。运行效果如图 14-10 所示。

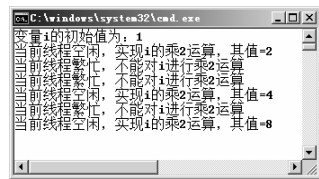


图 14-10 应用自定义消息

【实现过程】

本实例定义两个线程, 利用宏定义空闲和繁忙消息。在主线程中使用 while 循环实时获取当前消息, 具体代码如下:

```

01 #include <windows.h>
02 #include <iostream>
03 using namespace std;

```



```
04
05 #define WC_IDLE WM_USER+1 //空闲
06 #define WC_BUSY WM_USER+2 //繁忙
07
08 DWORD WINAPI ThreadProc1(LPVOID lpParameter)
09 {
10     DWORD dwThreadId = *(DWORD*)lpParameter; //线程 ID
11     while(true)
12     {
13         Sleep(1000);
14         PostThreadMessage(dwThreadId, WC_IDLE, 0, 0); //每隔 1 秒发送空闲消息
15     }
16     return 0;
17 }
18 DWORD WINAPI ThreadProc2(LPVOID lpParameter)
19 {
20     DWORD dwThreadId = *(DWORD*)lpParameter; //线程 ID
21     while(true)
22     {
23         Sleep(1000);
24         PostThreadMessage(dwThreadId, WC_BUSY, 0, 0); //每隔 1 秒发送繁忙消息
25     }
26     return 0;
27 }
28 void main()
29 {
30     int i=1; //变量
31     cout<<"变量 i 的初始值为: "<<i<<endl;
32     DWORD dwValue = GetCurrentThreadId(); //获取当前线程 ID
33     HANDLE hThread1; //定义句柄
34     HANDLE hThread2; //定义句柄
35     hThread1=CreateThread(NULL, 0, ThreadProc1, &dwValue, 0, NULL);
36     hThread2=CreateThread(NULL, 0, ThreadProc2, &dwValue, 0, NULL); //创建线程
37     CloseHandle(hThread1); //关闭线程句柄
38     CloseHandle(hThread2); //关闭线程句柄
39     MSG msg; //消息变量
40     while(GetMessage(&msg, NULL, 0, 0)) //获取消息循环
41     {
42         switch(msg.message)
43         {
44             case WC_IDLE:
45                 i*=2;
46                 cout<<"当前线程空闲, 实现 i 的乘 2 运算, 其值="<<i<<endl;
47                 Sleep(5000); //允许用 5 分钟
48                 break;
49             case WC_BUSY:
50                 cout<<"当前线程繁忙, 不能对 i 进行乘 2 运算"<<endl;
51                 Sleep(1000); //再等待 1 秒
52                 break;
53             default:
54                 cout<<"获取不知名的消息"<<msg.message<<endl;
55                 break;
56         }
57         Sleep(1000); //每隔 1 秒获取一次消息
58     }
59     cout<<"主线程完成"<<endl;
60 }
```



【代码解析】

第 05、06 行自定义消息 WC_IDLE 和 WC_BUSY，第 08~17 行、第 18~27 行分别定义两个线程的执行函数。其中，第 14、24 行每隔 1 秒发送消息，第 30 行定义主线程的局部变量 i。第 32 行获取当前线程 ID，第 33~38 行定义两个线程并进行相应的操作。第 39 行定义消息变量，第 40~58 行的 while 循环实时获取当前消息类型。如果是 WC_IDLE 的消息，则打印当前 i 的值，如果是繁忙消息，则不打印。



实例 268 利用_declspec(dllexport)导出类

【实例描述】

_declspec(dllexport)导出函数在第 10 章已有介绍，本实例给出利用该方式导出整个类。虽然同为控制台工程，但它的配置类型不是 EXE 型，而是 DLL 型，这可以在属性的配置方式中选择，配置选择如图 14-11 所示。

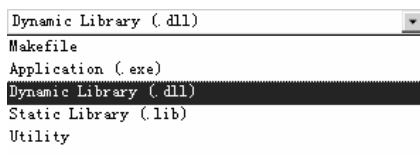


图 14-11 配置 DLL 工程

【实现过程】

本实例导出第 7 章的走迷宫类 Maze，具体代码请查看实例 128。下面给出基本框架，首先为头文件：

```
01 #ifndef MAZE_API
02 #else
03 #define MAZE_API _declspec(dllexport) //定义导出标识符
04 #endif
05
06 #include <iostream>
07 using namespace std;
08 ...
09 class MAZE_API Maze //导出类声明
10 {
11 ...
12 };
```

接下来，给出源文件定义类中各个函数的代码，如下：

```
01 #define MAZE_API _declspec(dllexport) //导出
02 #include "268.h" //头文件
03 ...
```

【代码解析】

第 1 段代码中，第 01~04 行定义类导出标识符 MAZE_API，第 09~12 行将导出整个类。第 2 段代码中，第 01 行导出以标识符 MAZE_API 标志的所有函数，第 02 行包含头文件。



注意：也可以只导出类中的任意一个函数，其操作规则同普通函数一样。



实例 269 调用_declspec(dllexport)导出类

【实例描述】

本实例给出调用_declspec(dllexport)导出的类，定义 main() 函数，创建迷宫对象调用类函数。运行效果如图 14-12 所示。

【实现过程】

首先复制 268.h、268.lib 和 268.dll 到该工程的当前目录下，然后输入以下代码：

```
01 #pragma comment(lib, "268.lib")    //包含 lib 文件
02 #include "268.h"
03
04 void main()
05 {
06     Maze _maze;                    //迷宫对象
07     _maze.PrintMaze();             //打印迷宫
08     _maze.GetPos();               //获取出入口位置
09     _maze.SearchMaze();           //寻找出口路径
10 }
```

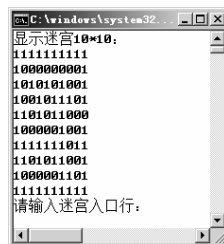


图 14-12 调用导出类——走出迷宫

【代码解析】

第 01 行包含 lib 文件。第 02 行包含头文件 263.h。第 06 行定义迷宫对象 _maze。第 07~09 行调用导出类的成员函数，以实现具体功能。



注意：对于包含 lib 文件的方式，除了第 01 行所示的方式外，还可以直接在属性中进行设置。



实例 270 利用.def 文件导出函数

【实例描述】

本实例演示如何使用.def 文件导出函数，使用该方式导出函数需要添加一个.def 文件，用于编辑被导出的函数。最值得注意的是，要在属性→链接器→输入→模块定义文件的选项中输入.def 文件的全称。本实例的操作如图 14-13 所示。

Module Definition File export.def

【实现过程】

添加一个源文件，名为 270.cpp，其定义被导出函数。

本实例导出的是实例 251 用于求素数，代码如下：

```
01 #include <iostream>
02 using namespace std;
03
04 void _stdcall find_sushu(int range_max)
05 {
06     int *num=new int[range_max+1];    //申请内存
07     ...
08 }
```

图 14-13 .def 属性设置



然后定义文件 `export.def` 用于导出目标函数，代码如下：

```
01 ;270Def.lib: 导出 DLL 函数
02 LIBRARY 270
03 EXPORTS
04 find_sushu @ 1
```

【代码解析】

在第 1 段代码的源文件中，最重要的是关键字 `_stdcall`，它是指调用 DLL 中函数的约定类型。除此之外，还有其他类型，比如 `_cdecl`、`_fastcall`、`_thiscall` 等。

第 2 段代码中，第 01 行为注释语句，即表明该文件的作用。第 02 行是导出 DLL 和 LIB 名称，第 03 行表示导出作用。第 04 行表示导出的第 1 个函数 `find_sushu()`，若想导出第 2 个函数，可以依次增加最后编号的值。



注意：切记，一定要做如图 14-13 所示的属性设置，否则将不会导出 .lib 文件。



实例 271 隐式调用 .def 导出的函数

【实例描述】

本实例将隐式地调用 .def 导出的函数 `find_sushu()`，实例运行效果如图 14-14 所示。



【实现过程】

复制 `270.lib` 和 `270.dll` 文件到当前工程的目录下，并输入以下代码到源文件 `266.cpp` 中。

```
01 #pragma comment(lib, "270.lib")
02 #include <iostream>
03 using namespace std;
04
05 extern void _stdcall find_sushu(int range_max); //声明该函数
06
07 void main()
08 {
09     int n;
10     cout<<"-----判断素数-----"<<endl;
11     cout<<"请输入最大素数检测界限值: ";
12     cin>>n; //输入素数
13     find_sushu(n); //调用函数
14 }
```

【代码解析】

第 01 行表示包含 .lib 文件，第 05 行表示声明 .def 方式导出函数 `find_sushu()`，第 13 行调用函数 `find_sushu()`。



注意：关键词 `extern` 提示程序链接到其他模块中寻找函数定义。



实例 272 显式调用.def 导出函数问题

【实例描述】

本实例实现利用显式方法调用.def 导出的函数，即只用导出函数的.dll 文件。运行代码后弹出窗口如图 14-15 所示。

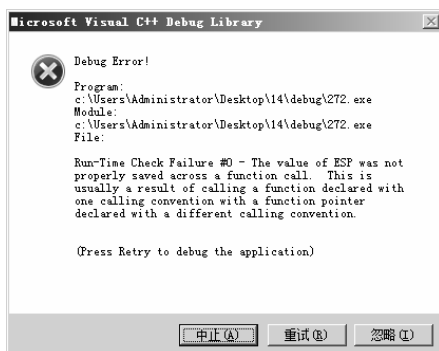


图 14-15 显式调用.def 方式导出函数问题

【实现过程】

首先定义指针函数的类型及函数指针 find_sushu，再用函数 LoadLibrary()加载动态库，当调用 DLL 失败或获取函数地址失败时，都退出。具体代码如下：

```
01 #include <iostream>
02 using namespace std;
03 #include <windows.h>
04
05 void main()
06 {
07     typedef void(*fff)(int); //定义指针函数
08     fff find_sushu; //定义函数指针
09     HINSTANCE hInst; //实例
10     hInst=LoadLibrary(L"270.dll"); //显式加载
11     if(hInst==NULL)
12     {
13         cout<<"调用 DLL 失败"<<endl;
14         return;
15     }
16     find_sushu=(fff)GetProcAddress(hInst,"find_sushu"); //获取函数地址
17     if(!find_sushu) //地址为空
18     {
19         cout<<"获取函数地址失败"<<endl;
20         return;
21     }
22     int n;
23     cout<<"-----判断素数-----"<<endl;
24     cout<<"请输入最大素数检测界限值: ";
25     cin>>n; //输入素数
26     find_sushu(n); //调用函数
27     FreeLibrary(hInst);
28 }
```



【代码解析】

第 07~09 行定义函数指针和实例句柄,第 10 行显式加载动态库,第 11 行判断是否成功调用 DLL,第 16 行获取目标函数的地址。如果获取失败,则执行第 19、20 行。反之,继续执行。第 22~26 行调用函数 find_sushu(),第 27 行释放动态库。



注意: 第 07 行的定义格式为:

```
typedef 目标函数返回类型 (*指针函数类型) (目标函数参数表)
```



实例 273 对应显式调用解决方法

【实例描述】

实例 272 在执行调用 DLL 的导出函数时,由于转换类型不匹配,就会出现如图 14-15 所示的错误。因此,只要双方的转换类型匹配,即可正确调用函数,运行效果如图 14-16 所示。

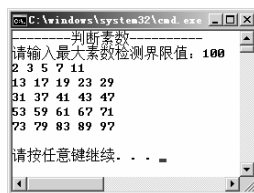


图 14-16 成功显式调用.def 方式导出函数

【实现过程】

本实例在定义指针函数类型时添加标志符 WINAPI,即可实现匹配,具体代码如下:

```
01  ...
02  void main()
03  {
04      typedef void(WINAPI*fff) (int);
05      ...
06  }
```

【代码解析】

第 04 行是唯一与实例 272 的代码不同的地方,WINAPI 用于 Win32 DLL 的变换。对于其他类型的 DLL,还有_stdcall 和_cedcall 方式。



注意: 利用.def 导出的函数并没有改变名字。当显式调用_declspec(dllexport)方式导出的函数时,需要在函数 GetProcAddress()的第 2 个参数处输入被改后的函数名。改后的函数名可在其.map 文件中查看。

第 15 章 数字图像处理专题

本章介绍利用 OpenCV 库实现数字图像处理，利用第三方库的优点是减少代码量。本章涉及数字图像处理最基本的算法，比如打开、保存、缩放、边缘检测图像等，采用 OpenCV2.0 版本进行编译，然后应用。



实例 274 载入并显示图像

【实例描述】

使用 OpenCV 库可以处理多种格式的图像，比如 BMP 和 JPEG 等。在编写程序前，先设置路径。本章将编译后的 OpenCV 库放在路径 C:\OpenCV2.0 下，在 include 下添加路径如下：

C:\OpenCV2.0\include\opencv

在 lib 下添加路径如下：

C:\OpenCV2.0\bZuild\lib\release

在 Executable files 下添加路径如下：

C:\OpenCV2.0\bin

然后，新建库文件编译工程，取名为 Common，头文件为 Common.h，源文件为 Common.cpp。在此只注重其 Common.h，代码如下：

```
#ifndef _COMMON_H
#define _COMMON_H
#ifdef _DEBUG
#pragma comment(lib, "cv200d.lib")
#pragma comment(lib, "cvaux200d.lib")
#pragma comment(lib, "cxcore200d.lib")
#pragma comment(lib, "highgui200d.lib")
#pragma comment(lib, "ml200d.lib")
#else
#pragma comment(lib, "cv200.lib")
#pragma comment(lib, "cvaux200.lib")
#pragma comment(lib, "cxcore200.lib")
#pragma comment(lib, "highgui200.lib")
#pragma comment(lib, "ml200.lib")
#endif
#endif
```

本实例载入并显示图像效果如图 15-1 所示。

【实现过程】

首先，定义指向源图像的指针 src，使用 cvLoadImage() 函数载入图像 005.jpg 到内存。然后，创建窗口显示图像，代码如下：



图 15-1 载入并显示图像



```
01 #include "../Common/Common.h"
02 #include "highgui.h"
03 #include <iostream>
04 using namespace std;
05
06 void main()
07 {
08     IplImage* src; //源图像指针
09     src = cvLoadImage("../Common/005.jpg ",3); //加载源图像
10     if(&src == NULL)
11         cout<<"加载源图像失败!"<<endl;
12     else
13     {
14         cvNamedWindow("src"); //创建图像窗口
15         cvShowImage("src",src); //显示图像函数
16         cvWaitKey(-1); //等待用户响应
17         cvDestroyWindow("src"); //销毁窗口
18     }
19     cvReleaseImage(&src); //释放源图像指针
20 }
```

【代码解析】

第 01 行为包含库定义文件。第 02 行为 OpenCV 的库文件。第 08 行定义 IplImage 型指针变量 src。第 09 行加载源图像。第 10 行判断图像是否加载成功，如果成功，由第 14 行创建图像显示窗口。第 15 行显示图像。第 16 行等待用户响应。第 17 行销毁窗口。最后由第 19 行释放指针 src。



注意：在运行程序时，如果提醒缺少某个 dll 文件，那么在环境变量的 Path 下添加 bin 路径（bin 下存储 dll 文件）即可。



实例 275 图像灰度化

【实例描述】

在对图像进行进一步处理前，都必须先灰度化处理。本实例对源图像进行灰度处理并显示，功能函数为 cvCvtColor()，效果如图 15-2 所示。



图 15-2 图像灰度化



【实现过程】

定义源图像指针 `src` 和灰度图像指针 `gray`，先加载源图像，再灰度化，最后显示灰度图像。代码如下：

```
01 #include "../Common/Common.h"
02 #include "highgui.h"
03 #include "cv.h"
04 #include <iostream>
05 using namespace std;
06
07 void main()
08 {
09     IplImage* src;                //源图像指针
10     IplImage* gray;              //灰度化后图像内存
11     src = cvLoadImage("../Common/005.jpg", 3);    //加载源图像
12     if(&src == NULL)
13         cout<<"加载源图像失败!"<<endl;
14     else
15     {
16         cvNamedWindow("gray");    //创建图像窗口
17         gray = cvCreateImage(cvGetSize(src), src->depth, 1);
18                                     //创建 gray 的内存大小等初始化
19         cvCvtColor(src, gray, CV_BGR2GRAY);    //灰度化
20         cvShowImage("gray", gray);    //显示图像函数
21         cvWaitKey(-1);    //等待用户响应
22         cvDestroyWindow("gray");    //销毁窗口
23     }
24     cvReleaseImage(&src);    //释放源图像指针
25     cvReleaseImage(&gray);    //释放灰度图像指针
26 }
```

【代码解析】

第 09、10 行定义源图像和灰度图像指针 `src` 和 `gray`。第 11 行加载源图像。第 17 行初始化灰度图像指针。第 19 行对 `src` 指向的图像灰度化，并保存到指针 `gray` 指向的内存，最后由第 20 行显示。



注意：函数 `cvCvtColor()` 的使用需要包含头文件 `cv.h`。



实例 276 图像灰度均衡化

【实例描述】

对图像进行均衡化是为了突出有效信息，处理对象是图像的各像素差别不是很大。为了使亮色更亮，暗色更暗，均衡化是一个很好的处理方法，效果如图 15-3 所示。



图 15-3 图像灰度均衡化

【实现过程】

因为均衡化是在直方图的基础上完成的，所以本实例定义 3 个 `IplImage*` 类型变量 `src`、`gray` 和 `hist`，分别表示为源图像、灰度图和均衡图，代码如下：

```
01 void main()
02 {
03     IplImage* src;                //源图像指针
04     IplImage* gray;              //灰度化后图像内存
05     IplImage* hist;              //均衡化图像
06     src = cvLoadImage("../Common/005.jpg", 3); //加载源图像
07     if(&src == NULL)
08         cout<<"加载源图像失败!"<<endl;
09     else
10     {
11         cvNamedWindow("hist");    //创建图像窗口
12         gray = cvCreateImage(cvGetSize(src), src->depth, 1); //创建 gray 并初始化
13         cvCvtColor(src, gray, CV_BGR2GRAY); //灰度化
14         hist = cvCreateImage(cvGetSize(src), src->depth, 1); //创建 hist 并初始化
15         hist = HistEqualization(gray); //均衡化
16         cvShowImage("hist", hist); //显示图像
17         cvWaitKey(-1);             //等待用户响应
18         cvDestroyWindow("hist");  //销毁窗口
19     }
20     cvReleaseImage(&src);          //释放源图像指针
21     cvReleaseImage(&gray);         //释放灰度图像指针
22     cvReleaseImage(&hist);        //释放均衡图像指针
23 }
```

函数 `HistEqualization()` 为自定义函数，代码如下：

```
01 IplImage * HistEqualization(IplImage *src)
02 {
03     IplImage* dst=0;              //定义目标图像并初始化
04     CvHistogram* hist=0;          //定义直方图
05     int n=256;                   //256 个阶度
06     double nn[256];              //直方图数组
07     uchar T[256];                //归一化后的直方图数组
08     CvMat* T_mat;                //定义矩阵
```



```

09         int x;                                //计数器
10         int sum=0;                             //图像中像素点的总和
11         double val=0;
12         hist=cvCreateHist(1,&n,CV_HIST_ARRAY,0,1); //创建直方图
13         cvCalcHist(&src,hist,0,0);              //计算直方图
14         val=0;                                  //累加计数器
15         for (x=0;x<n;x++)                       //计算源图像直方图某个值的个数
16         {
17             val=val+cvGetReal1D(hist->bins,x);
18             nn[x]=val;
19         }
20         //归一化直方图
21         sum=src->height*src->width;
22         for (x=0;x<n;x++)
23         {
24             T[x]=(uchar) (255*nn[x]/sum);
25         }
26         //创建均衡化处理后的新图
27         dst=cvCloneImage(src);                  //复制图像
28         T_mat=cvCreateMatHeader(1,256,CV_8UC1); //创建矩阵头
29         cvSetData(T_mat,T,0);                  //设置新的 data 数据
30         cvLUT(src,dst,T_mat);                  //填充数据
31         //释放内存
32         cvReleaseImage(&src);
33         cvReleaseHist(&hist);
34         src=NULL ;
35         return dst;                             //返回均衡化后图像
36     }

```

【代码解析】

(1) main()

第 03~05 行定义 3 个变量，第 06 行是加载源图像。第 10~18 行中，先灰度化（第 12、13 行），再调用函数 `HistEqualization()` 实现均衡化。最后显示均衡化后的图像，如第 16 行。

(2) HistEqualization()

第 03~11 行定义计算均衡化图像所需变量，请参看后面的注释。第 12 行创建直方图，第 13 行计算源图像直方图，第 15~19 行计算直方图中某个值的个数。第 21~25 行归一化直方图，第 27~30 行将均衡化的直方图数据写入变量 `dst` 中。第 32~34 行释放相应的内存，第 35 行返回均衡化后的图像。



注意：均衡化是在灰度图像的基础上进行的操作，所以需先对图像灰度化。



实例 277 自适应化获取图像二值化阈值

【实例描述】

图像处理的二值化原理是将某个阈值左右范围赋值不同的数值，即大于或等于阈值时，像素值赋为 0 或 1；小于阈值时，像素值被赋为 1 或 0。本实例实现如何自适应化获取图像的二值化阈值，效果如图 15-4 所示。

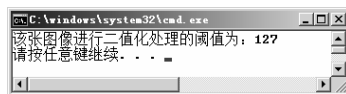


图 15-4 自适应获取图像二值化阈值



【实现过程】

二值化处理是在灰度图像的基础上进行的，所以还需先进行灰度化处理。二值化阈值的自适应获取基于图像的像素比例，然后遍历灰度级，计算出源图像阈值。代码如下：

```

01  #include "../Common/Common.h"
02  #include "highgui.h"
03  #include "cv.h"
04  #include <iostream>
05  using namespace std;
06
07  int Threshold(IplImage *src)                //求取阈值
08  {
09      int width = src->width;                //源图像宽度
10      int height = src->height;              //源图像高度
11      ...
12      for(int i = 0; i < 256; i++)            //初始化数组变量
13      {
14          ...
15      }
16      for(int i = 0; i < height; i++)          //统计灰度级每个像素值在整幅图像中的个数
17      {
18          ...
19      }
20      for(int i = 0; i < 256; i++)            //计算每个像素在整幅图像中的比例
21      {
22          ...
23      }
24      for(int i = 0; i < 256; i++)            //遍历灰度级[0,255]
25      {
26          ...
27      }
28      return threshold;                      //返回阈值
29  }
30  void main()
31  {
32      IplImage* src;                         //源图像指针
33      IplImage* gray;                        //灰度化后图像内存
34      src = cvLoadImage("../Common/005.jpg", 3); //加载源图像
35      if(&src == NULL)
36          cout<<"加载源图像失败!"<<endl;
37      else
38      {
39          gray = cvCreateImage(cvGetSize(src), src->depth, 1); //创建gray的内存
40          cvCvtColor(src, gray, CV_BGR2GRAY);                  //灰度化
41          int threNumber;
42          threNumber = Threshold(gray);                         //调用 otsu 函数求阈值
43          cout<<"该张图像进行二值化处理的阈值为: "<<threNumber<<endl;
44      }
45      cvReleaseImage(&src);                                     //释放源图像指针
46      cvReleaseImage(&gray);                                    //释放灰度图像指针
47  }

```

【代码解析】

第 07~29 行是函数 Threshold()求源图像阈值，该函数的自适应算法的具体原理在此不详细介绍，请参看代码对应的注释。第 32~34 行定义图像变量及加载源图像，第 39、40 行对源图



像进行灰度化处理, 第 42 行调用函数 `Threshold()`, 第 43 行输出图像阈值。



注意: 函数 `Threshold()` 完全可以被当作已封装好的函数再次使用, 因此对于它的实现原理不需要深入理解。



实例 278 二值化源图像

【实例描述】

在实例 277 中求取图像二值化阈值后, 对图像数据进行统一赋值, 即可实现二值化, 运行效果如图 15-5 所示。

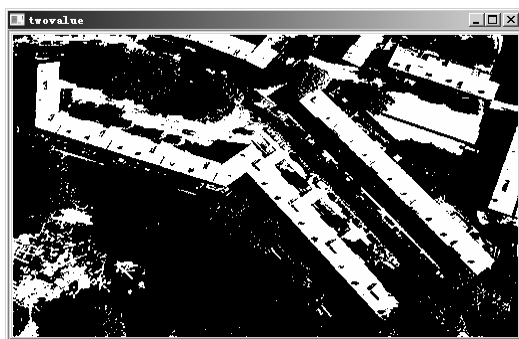


图 15-5 源图像的二值化结果

【实现过程】

定义函数 `binary()`, 在阈值的基础上, 对源图像的像素值进行判断, 并进行相应的赋值。然后在灰度图像上调用函数 `binary()` 实现二值化, 代码如下:

```
01 #include "../Common/Common.h"
02 #include "highgui.h"
03 #include "cv.h"
04 #include <iostream>
05 using namespace std;
06
07 int Threshold(IplImage *src)           //求取阈值
08 {
09     ...
10 }
11 IplImage* binary(IplImage *src)
12 {
13     IplImage* imgOtsu;                 //二值化图像
14     imgOtsu= cvCreateImage(cvGetSize(src),IPL_DEPTH_8U,1); //创建
15     cvCopy(src,imgOtsu);               //复制数据
16     int threNumber;                     //阈值
17     threNumber=Threshold(imgOtsu);      //调用 Threshold 函数求阈值
18     CvScalar otsu;                      //像素
19     int temp;                           //像素值
20     for(int i=0;i<src->height;i++)      //二值化
21     {
```



```

22         for(int j=0;j<src->width;j++)
23         {
24             otsu=cvGet2D(imgOtsu,i,j);           //获取源图像素
25             temp=(int)otsu.val[0];                //像素值
26             if(temp>=threNumber)                  //不小于阈值
27                 temp=255;                          //白
28             else                                   //小于阈值
29                 temp=0;                             //黑
30             otsu.val[0]=temp;
31             cvSet2D(imgOtsu,i,j,otsu);            //写入二值化图像
32         }
33     }
34     return imgOtsu;
35 }
36 void main()
37 {
38     ...
39     else
40     {
41         ...
42         cvCvtColor(src,gray,CV_BGR2GRAY);          //灰度化
43         imgOtsu=binary(gray);                      //二值化
44         cvShowImage("twovalue",imgOtsu);           //显示
45         ...
46     }
47     ...
48 }

```

【代码解析】

第 07~10 行是函数 Threshold()求图像阈值。第 11~35 行为二值化函数，其中，第 13~15 行定义二值化图像变量 imgOtsu，并对其创建及初始化。第 16、17 行获取图像阈值，第 18、19 行为二值化算法所需变量。第 20~33 行对源图像进行行列扫描，判断当前像素是否满足条件（第 26、28 行）。

在 main()函数中，先对源图像进行灰度化（第 42 行），然后调用函数 binary()对图像进行二值化（第 43 行）处理，显示二值化图像（第 44 行）。



注意：同样，二值化函数 binary()也可以作为封装函数用于二次开发。



实例 279 保存目标图像

【实例描述】

从实例 278 的目标图像可以看出，虽然二值化可以凸显物体，但是图像中还有噪声，比如独立的小白点，这些信息并不是所需要的。为了将这些噪声去除，需要在目标图像上进行进一步操作。本实例首先保存二值化图像，它需要调用函数 cvSaveImage()，保存成功后输出提示信息，效果如图 15-6 所示。

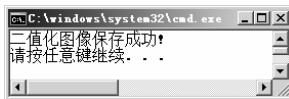


图 15-6 保存目标图像提示



【实现过程】

在实例 278 的基础上，在显示二值化图像后添加保存图像函数和保存成功提示信息，代码如下：

```
01  ...
02  void main()
03  {
04      ...
05          cvShowImage("twovalue",imgOtsu);           //显示
06          if (cvSaveImage("../Common/twovalue.jpg",imgOtsu)) //保存图像
07              cout<<"二值化图像保存成功!"<<endl;
08      ...
09  }
10      ...
11  }
```

【代码解析】

第 06 行判断是否成功保存图像（使用函数 `cvSaveImage()`，第 1 个参数为图像保存路径，第 2 个参数为保存图像数据）。如果保存成功，则输出第 07 行的提示信息。



实例 280 去除图像噪声（形态学开运算）

【实例描述】

在保存完二值化图像后，接下来使用形态学方法对图像进行去噪处理。本实例先使用形态学开运算去噪，运行效果如图 15-7 所示。

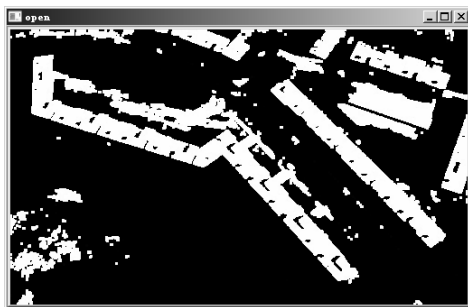


图 15-7 形态学开运算去噪效果

【实现过程】

利用形态学去噪可以使用函数 `cvMorphologyEx()`，它需要 3 个变量，分别为源图像、目标图像和临时变量数据。去噪后，显示目标图像并保存，代码如下：

```
01  #include "cv.h"
02  #include "highgui.h"
03  #include "../Common/Common.h"
04  #include <iostream>
05  using namespace std;
06
07  void main()
```



```

08  {
09      cvNamedWindow("open");                //命名窗口
10      IplImage * src = cvLoadImage("../Common/twovalue.jpg", 0);
11      IplImage * temp = cvCreateImage(cvGetSize(src), 8, 1);    //创建临时变量
12      IplImage * dst = cvCreateImage(cvGetSize(src), 8, 1);    //创建目标变量
13      cvCopyImage(src, temp);                //初始化临时
14      cvCopyImage(src, dst);                //初始化目标
15      cvMorphologyEx(                        //开运算
16          src,                                //源变量
17          dst,                                //目标变量
18          temp,                                //临时变量
19          NULL,                                //模板大小, 默认为 3*3
20          CV_MOP_OPEN,                        //运算种类
21          1);                                //迭代次数
22      cvShowImage("open", dst);                //显示图像
23      cvSaveImage("../Common/open.jpg", dst); //保存图像
24      cvWaitKey(0);
25      cvReleaseImage(&temp);                //释放
26      cvReleaseImage(&src);
27      cvReleaseImage(&dst);
28      cvDestroyAllWindows();                //销毁窗口
29  }

```

【代码解析】

第09行命名窗口,第10行加载源图像,第11、12行定义开运算所需变量 `temp` 和 `dst`。第13、14行初始化上述两个变量。第15~21行调用函数 `cvmorphologyEx()` 进行开运算,其中,第1个参数为源图像,第2个参数为目标图像,第3个参数为参与运算的临时变量,第4个参数为运算模板(此处为默认的3*3),第5个参数为运算类型(此处选为开运算 `CV_MOP_OPEN`),最后一个参数为运算的迭代次数。



注意: 由效果图可知,形态学开运算是将独立的小白点去除,但是面积大的亮区域融合在一起形成较大的区域。



实例 281 去除图像噪声（形态学闭运算）

【实例描述】

本实例利用形态学闭运算去二值化图像进行去噪处理,效果如图15-8所示。

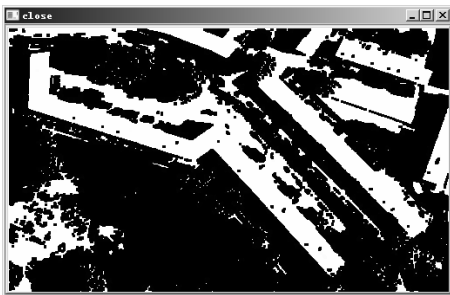


图 15-8 形态学闭运算去噪



【实现过程】

在实例 280 的基础上，只改变窗口名字及函数 `cvMorphologyEx()` 的第 5 个参数，即可完成闭运算。代码如下：

```
01  ...
02  void main()
03  {
04      ...
05      cvMorphologyEx (          //闭运算
06          src,                  //源
07          dst,                  //目标
08          temp,                 //临时
09          NULL,                 //默认模板 3*3
10          CV_MOP_CLOSE,        //闭运算
11          1);                   //迭代次数
12      cvShowImage("close", dst); //显示目标图像
13      cvSaveImage("../Common/close.jpg", dst); //保存图像
14      ...
15      cvDestroyAllWindows();    //销毁窗口
16  }
```

【代码解析】

第 05~11 行为形态学闭运算，其中第 5 个参数 `CV_MOP_CLOSE` 表示形态学闭运算参数。第 12、13 行为显示和保存图像，第 15 行为销毁所有的窗口（如果创建窗口较多，使用函数 `cvDestroyAllWindows()`）。



注意：读者可以自行改变迭代次数的值，运行效果有较大的差别。



实例 282 获取图像内物体轮廓（Canny 检测）

【实例描述】

本实例利用 OpenCV 已有函数 `cvCanny()` 实现对物体轮廓的 Canny 检测，其中输入参数有两个阈值，可以调节其数值，更新检测结果，运行效果如图 15-9 所示。

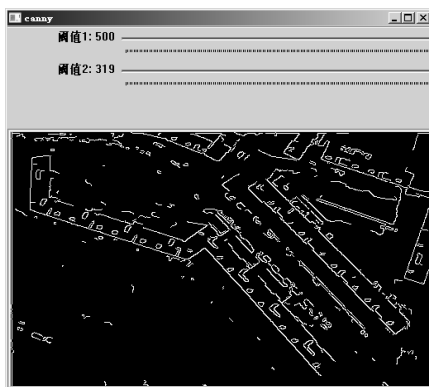


图 15-9 物体轮廓的 Canny 检测



【实现过程】

在本实例中出现变量条，使用函数 `on_trackbar()` 更新其值。`cvCanny()` 函数做边缘检测，其中，有两个阈值变量 `threshold1` 和 `threshold2`，实行双阈值检测。代码如下：

```
01 #include "../Common/Common.h"
02 #include "highgui.h"
03 #include "cv.h"
04 #include <iostream>
05 using namespace std;
06
07 void on_trackbar(int value)           //变量条
08 {}
09 void main()
10 {
11     int threshold1=20;                //阈值 1
12     int threshold2=20;                //阈值 2
13     IplImage* src;                    //源图像指针
14     IplImage* gray;                   //灰度化后图像内存
15     IplImage* edge;                  //检测图像
16     src = cvLoadImage("../Common/005.jpg", 3); //加载源图像
17     if(&src == NULL)
18         cout<<"加载源图像失败!"<<endl;
19     else
20     {
21         cvNamedWindow("canny");       //创建图像窗口
22         gray = cvCreateImage(cvGetSize(src), src->depth, 1); //创建gray的内存
23         cvCvtColor(src, gray, CV_BGR2GRAY); //灰度化
24         edge = cvCreateImage(cvGetSize(src), IPL_DEPTH_8U, 1);
25         //创建变量范围条
26         cvCreateTrackbar("阈值 1", "canny", &threshold1, 500, on_trackbar);
27         cvCreateTrackbar("阈值 2", "canny", &threshold2, 500, on_trackbar);
28         while(1)
29         {
30             cvCanny(gray, edge, threshold1, threshold2, 3); //Canny 检测
31             cvShowImage("canny", edge); //显示图像
32             if(cvWaitKey(100)>0) //等待用户响应
33                 break;
34         }
35         cvSaveImage("../Common/canny.jpg", edge); //保存图像
36     }
37     cvDestroyWindow("canny"); //销毁窗口
38     cvReleaseImage(&src); //释放源图像指针
39     cvReleaseImage(&gray); //释放灰度图像指针
40     cvReleaseImage(&edge); //释放均衡图像指针
41 }
```

【代码解析】

第 07、08 行为变量条函数 `on_trackbar()` 的定义，第 11、12 行定义 Canny 检测的双阈值变量。第 13~16 行定义图像处理变量及初始化。在第 22、23 行进行灰度化处理，第 24 行创建 Canny 检测后对变量进行初始化。第 26、27 行使用函数 `cvCreateTrackbar()` 创建变量范围条，第 30 行进行 Canny 检测，第 31 行显示图像。



注意：为了及时更新变化的轮廓图，在第 28 行的 `while` 循环采用无限循环，其内如果满足条件，则强制退出循环。



实例 283 物体轮廓直线化（Hough 变换）

【实例描述】

本实例实现在 Canny 检测的基础上，利用 Hough 变换使物体边缘轮廓直线化。Hough 变换的运行效果如图 15-10 所示。

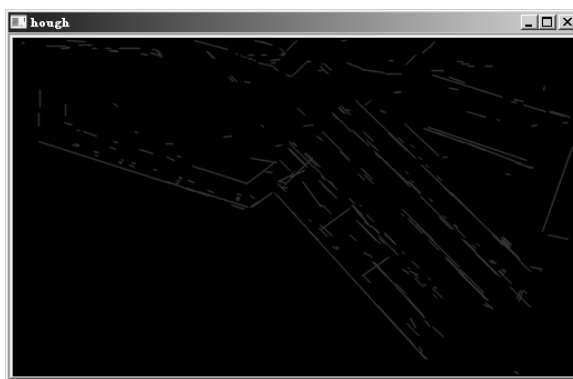


图 15-10 Hough 变换

【实现过程】

在实例 282 的基础上，添加实现 Hough 变换的代码。实现 Hough 变换的函数是 cvHoughLines2()。为了绘制直线，利用函数 showLines()完成，具体代码如下：

```
01  ...
02  IplImage* showLines(CvSeq *lines, IplImage *img)
03  {
04      IplImage* dst;
05      dst = cvCreateImage(cvGetSize(img), IPL_DEPTH_8U, img->nChannels);
06      cvCopy(img, dst, NULL);
07      for(int i = 0; i < lines->total; i++)                //循环所有的直线
08      {
09          CvPoint* line = (CvPoint*)cvGetSeqElem(lines, i); //获取直线
10          cvLine(dst, line[0], line[1], CV_RGB(255, 0, 0), 1, CV_AA, 0);
                                                    //直线用红色输出
11      }
12      return dst;
13  }
14  ...
15  void main()
16  {
17      ...
18      int param0 = 400, param1 = 216, threshold = 33, hpl = 2, hp2 = 3;
19      int rho = 1, theta = 2;                        //hough 变换变量
20      ...
21      IplImage* hough;                                //hough 变换
22      ...
23      else
24      {
25          ...
26          hough = cvCreateImage(cvGetSize(src), IPL_DEPTH_8U, 3); //创建 hough 变量
```



```

27         CvSeq* line;
28         CvMemStorage * storage = cvCreateMemStorage(0);
29         ...
30         while(1)
31         {
32             ...
33             line = cvHoughLines2(edge, storage, CV_HOUGH_PROBABILISTIC,
34                                 rho, CV_PI/180, threshold, hp1, hp2); //hough 变换
35             cvShowImage("canny", edge); //显示边缘检测图像
36             cvShowImage("hough", showLines(line,hough)); //显示直线
37             ...
38         }
39         cvSaveImage("../Common/canny.jpg",edge); //保存边缘检测图像
40         cvSaveImage("../Common/hough.jpg",showLines(line,hough)); //保存 hough 直线
41     }
42     ...
43 }

```

【代码解析】

第 02~13 行是绘制直线函数 showLines()的定义体,其中,第 07 行循环所有的直线,第 09 行获取直线的起始和终止点,第 10 行用红色将直线绘制。接下来在 main()函数中,第 18、19 行定义实现 Hough 变换的所有变量,第 21 行定义 Hough 变换后存储的图像数据。

第 26 行创建变量 hough 值,第 27、28 行定义 Hough 变换所需的辅助变量 line 和 storage。第 33、34 行调用函数 cvHoughLines2()实现 Hough 变换,并返回所有的直线。第 36 行获取函数 showLines()的返回值绘制直线。第 40 行保存 Hough 变换直线图。



实例 284 绘制图像灰度直方图

【实例描述】

本实例演示如何绘制图像的灰度直方图,它的目的是更直观地观察图像的灰度分布,以利于后期利用灰度进一步研究图像,效果如图 15-11 所示。

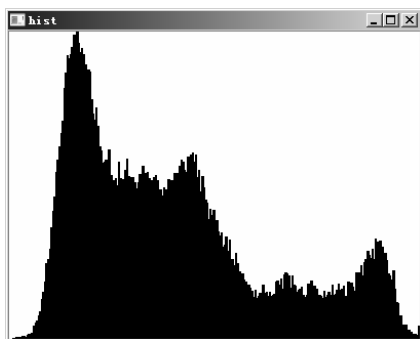


图 15-11 绘制图像灰度直方图

【实现过程】

首先获取源图像的灰度,然后计算各级的灰度总数相对于总灰度数的比例。最后将这些比



例值存储到图像缓冲区，并绘制显示。代码如下：

```
01 void main()
02 {
03     IplImage* src=cvLoadImage("../Common/005.jpg ",0); //加载图像
04     int width=src->width; //图像宽度
05     int height=src->height; //图像高度
06     int step=src->widthStep; //图像字节
07     uchar* data=(uchar *)src->imageData; //图像数据
08     int hist[256]={0}; //灰度级
09     int max=0; //最大值
10     IplImage* dst=cvCreateImage(cvSize(400,300),8,0); //目标图像
11     cvSet(dst,cvScalarAll(255),0); //初始化
12     double bin_width; //宽度间隔
13     double bin_unith; //相对 dst 高度间隔
14     for(int i=0;i<height;i++) //行
15     {
16         for(int j=0;j<width;j++) //列
17         {
18             hist[data[i*step+j]]++; //对应灰度级加 1
19         }
20     }
21     for(int i=0;i<256;i++) //级
22     {
23         if(hist[i]>max) //求最大值
24         {
25             max=hist[i];
26         }
27     }
28     bin_width=(double)dst->width/256;
29     bin_unith=(double)dst->height/max;
30     for(int i=0;i<256;i++)
31     {
32         CvPoint p0=cvPoint(i*bin_width,dst->height); //起始点
33         CvPoint p1=cvPoint((i+1)*bin_width,dst->height-hist[i]*bin_unith); //终止点
34         cvRectangle(dst,p0,p1,cvScalar(0,0,0),-1,8,0); //黑色
35     }
36     cvNamedWindow("hist",1);
37     cvShowImage("hist",dst);
38     cvWaitKey(0);
39     cvDestroyAllWindows();
40     cvReleaseImage(&src);
41     cvReleaseImage(&dst);
42 }
```

【代码解析】

第 03 行加载图像函数的第 2 个参数为 0，表示将读取的图像已经强制转化为灰度图。第 04~07 行获取图像信息，比如宽度、高度、每行字节数和图像数据。第 08 行定义 256 个灰度级数组，并初始化为 0。第 09 行定义灰度级的最大值变量 max，第 10 行创建目标图像变量 dst，用于绘制灰度直方图，第 11 行对其进行初始化，第 12 行定义目标图像各个长方形的宽度间隔和高度间隔。

第 14~20 行计算各个灰度级的个数。第 21~27 行求灰度级的最大值，存于变量 max 中。第 28、29 行给变量 bin_width 和 bin_unith 赋值，然后用于第 30~35 行的直方图绘制。第 36~41 行显示直方图图像，然后释放内存。



注意：如果第 03 行中第 2 个参数不是 0，必须在计算灰度级前先把图像灰度化。



实例 285 缩放图像

【实例描述】

OpenCV 有函数 `cvResize()` 可以实现图像的缩放，本实例实现该函数的简单功能。可以利用结构体 `IplImage` 中的元素 `imageData`，程序运行效果如图 15-12 所示。



图 15-12 缩放图像

【实现过程】

定义源/目标图像指针变量 `src` 和 `dst`、缩放倍数 `scale`、源/目标图像的宽高度、源/目标图像的字节和内存数据。在赋值过程中，如果有不存在的点，赋为 255。代码如下：

```
01 void main()
02 {
03     IplImage *src;                //源图像指针
04     IplImage *dst;                //目标图像指针
05     float scale=1.2;              //缩放倍数
06     int sW,sH;                    //源图像宽高
07     int dW,dH;                    //目标图像宽高
08     int d_step,s_step;            //图像字节
09     uchar *dst_data;              //目标数据
10     uchar *src_data;              //源数据
11     src=cvLoadImage("../Common/005.jpg ",0); //载入文件
12     sW=src->width;                 //图像宽度
13     sH=src->height;                //图像高度
14     s_step=src->widthStep;          //源图像字节
15     src_data=(uchar*)src->imageData; //源数据获取
16     dW=sW*scale;
17     dH=sH*scale;                  //目标图像高度
18     dst=cvCreateImage(cvSize(dW,dH),src->depth,src->nChannels); //构造目标图像
19     d_step=dst->widthStep;          //目标图像字节
20     dst_data=(uchar*)dst->imageData; //指向目标数据
21     for(int i=0;i<dH;i++)          //行
```



```
22     {
23         for(int j=0;j<dW;j++)                //列
24         {
25             int i0=i/scale;                  //坐标还原
26             int j0=j/scale;
27             if((j0>=0)&&(j0<sW)&&(i0>=0)&&(i0<sH))    //在源图范围内
28             {
29                 //指向源图像第 i0 行第 j0 列像素并赋值
30                 *(dst_data+(dH-1-i)*d_step+j)=*(src_data+s_step*(sH-1-i0)+j0);
31             }
32             else                                //在源图范围外
33             {
34                 *(dst_data+(dH-1-i)*d_step+j)=255;    //赋值 255
35             }
36         }
37     }
38     cvNamedWindow("src");                    //显示源图像窗口
39     cvNamedWindow("dst");                    //显示目标图像窗口
40     cvShowImage("src",src);                  //显示源图像
41     cvShowImage("dst",dst);                  //显示目标图像
42     cvWaitKey(-1);                           //等待用户响应
43     cvReleaseImage(&src);                     //释放内存
44     cvReleaseImage(&dst);
45     cvDestroyAllWindows();
46 }
```

【代码解析】

第 03、04 行定义源/目标图像指针，第 05 行定义缩放倍数。第 06~10 行定义源图像的各种信息，第 11 行加载源图像。第 12~15 行获取源图像信息，第 16、17 行计算目标图像宽高度。第 18 行创建目标图像，第 19、20 行给目标图像变量初始化。第 21~37 行的两层循环完成缩放功能。

其中，先进行行扫描，再进行列扫描。第 25、26 行还原目标图像坐标到源图像，第 27 行判断 $i0$ 和 $j0$ 的值是否在源图像范围内。如果在范围内，完成赋值（第 30 行），否则赋值为 255（第 34 行）。



注意：对图像进行缩放操作也是在灰度图的基础上进行的。



实例 286 图像格式转换

【实例描述】

图像格式有很多种，相互之间也可以转换。本实例提供程序转换图像格式，支持的格式有 5 种，JPG、BMP、TIF、PNG 和 PPM。程序运行效果如图 15-13 所示。

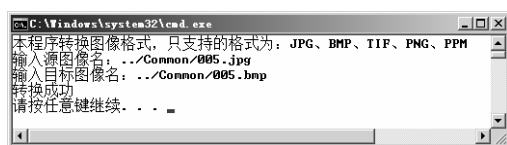


图 15-13 图像格式转换



【实现过程】

定义 `IplImage` 型变量存储载入图像数据, 以及 4 个 `string` 型变量 `nsrc`、`ndst`、`csrc` 和 `cdst`, 分别表示图像名和图像后缀名。函数 `cvLoadImage()` 载入图像, `cvSaveImage()` 存储图像。代码如下:

```
01 void main()
02 {
03     IplImage* src;                //源图像
04     string nsrc;
05     string ndst;                  //源目标图像名
06     string csrc;
07     string cdst;                  //源目标图像名的最后 3 字节
08     cout<<"本程序转换图像格式, 只支持的格式为: JPG、BMP、TIF、PNG、PPM\n";
09     cout<<"输入源图像名: ";
10     cin>>nsrc;
11     csrc=nsrc.substr(nsrc.length()-3,3);    //源图像的后缀
12     if(csrc!="jpg"
13         && csrc!="bmp"
14         && csrc!="tif"
15         && csrc!="png"
16         && csrc!="ppm")
17     {
18         cout<<"源图像格式不支持\n";
19     }
20     else
21     {
22         cout<<"输入目标图像名: ";
23         cin>>ndst;
24         cdst=ndst.substr(ndst.length()-3,3);    //目标图像的后缀
25         if(cdst!="jpg"
26             && cdst!="bmp"
27             && cdst!="tif"
28             && cdst!="png"
29             && cdst!="ppm")
30         {
31             cout<<"目标图像格式不支持\n";
32         }
33         else
34         {
35             src=cvLoadImage(nsrc.c_str(),-1);    //加载源图像
36             if(src!=0)                            //读取成功
37             {
38                 if(cvSaveImage(ndst.c_str(),src)!=0)    //保存图像
39                     cout<<"转换成功\n";
40             }
41             else
42                 cout<<"读取失败\n";
43         }
44     }
45     cvReleaseImage(&src);                        //释放
46 }
```

【代码解析】

第 03 行定义源图像变量 `src`。第 04~07 行定义 4 个 `string` 型变量。第 08 行输出提示语。第 10 行获取图像路径。第 11 行获取图像的后缀名。第 12~16 行判断源图像后缀是否符合要求, 如果不符合要求, 执行第 18 行, 否则执行第 22~43 行。

第 22 行提示输入转换后的图像名, 第 23 行提取变量 `ndst` 的值。第 24 行获取目标图像的



后缀名。第 25~29 行判断目标图像后缀是否符合要求, 如果不符合要求, 输出第 31 行, 否则执行第 34~43 行。

第 35 行加载源图像, 第 36 行判断是否读取成功, 如果成功, 保存目标图像, 如第 38 行。如果转换成功, 输出第 39 行。另外, 如果读取失败, 则输出第 42 行。



注意: 除了上述格式, 还有 PBM、PGM、EXR、SR、RAS 等格式图像文件。



实例 287 播放视频

【实例描述】

OpenCV 不仅可以处理图像, 还可以加载视频。本实例实现视频 (AVI 格式) 的播放, 效果如图 15-14 所示。



图 15-14 视频播放

【实现过程】

视频的播放是基于显示每帧图像, 首先获取该视频的帧率。然后求出播放每帧图像所需时间。最后利用函数 `cvShowImage()` 显示当前帧, 再调用函数 `cvWaitKey()` 等待每帧播放时间完成视频的播放。代码如下:

```
01 #include "../Common/Common.h"
02 #include <cv.h>
03 #include <highgui.h>
04 #include <iostream>
05 using namespace std;
06
07 void main()
08 {
09     double fps;                //视频帧率
10     int vfps;                  //每帧播放时间(毫秒)
11     IplImage *frame;           //每帧图像数据
12     float pos_ratio;           //当前帧在 AVI 中相对位置的比例
13     CvCapture *capture=cvCreateFileCapture("../Common/Megamind_buggy.avi");
```



```
14                                     //读入 AVI 文件
15     if (capture==NULL)               //变量值为空
16     {
17         cout<<"没有视频文件"<<endl;    //输出提示语
18         return;                       //返回
19     }
20     fps=cvGetCaptureProperty(capture,CV_CAP_PROP_FPS); //读取视频的帧率
21     vfps=1000/fps;                   //计算每帧播放的时间
22     cvNamedWindow("AVI",1);          //创建窗口
23     while(1)                         //循环
24     {
25         frame=cvQueryFrame(capture);    //抓取当前帧
26         //读取该帧在视频中的相对位置
27         pos_ratio=cvGetCaptureProperty(capture,CV_CAP_PROP_POS_AVI_RATIO);
28         if(!frame)                   //没有帧
29             break;                   //退出循环
30         cvShowImage("AVI",frame);      //显示当前帧
31         cvWaitKey(vfps);              //等待当前帧播放完
32     }
33     cvReleaseImage(&frame);
34     cvReleaseCapture(&capture);        //释放视频
35     cvDestroyWindow("AVI");           //销毁窗口
36 }
```

【代码解析】

第 09~12 行定义变量 `fps`（视频帧率）、`vfps`（每帧播放所用时间）、`frame`（每帧图像）和 `pos_ratio`（当前帧在视频中的位置比例）。第 13 行获取 AVI 文件数据，第 15~19 行判断变量 `capture` 是否为 `NULL`，并做相应的操作。

第 20 行读取视频的帧率。第 21 行以毫秒为单位计算每帧播放时间。第 25 行抓取当前帧，第 27 行获取当前帧在视频中的相对位置。在循环过程中，如果全部读完视频帧，则退出循环（如第 28、29 行）。第 30 行显示当前帧。第 31 行等待 `vfps` 时间，再读下一帧。



注意：在获取帧的相对位置时，必须将第 25 行代码放在第 26 行代码之前，否则 `pos_ratio` 的值会有问题。

第 16 章 三维仿真技术专题

本章主要讲解三维仿真技术的知识，所用关键技术是 C++与 OSG 语言的结合。本章意在演示如何利用 C++语言驾驭第三方语言，实例 288 给出了 OSG 语言应用的环境设置。关于应用环境的配置，读者可上网查看有关资料。



实例 288 OSG 语言应用的环境设置

【实例描述】

当安装 Visual Studio 开发环境后，利用 CMake 配置相应版本的 OSG。本实例讲解在环境设置中，添加 include 和 lib 路径，并且给出 lib 文件的包含命令。

【实现过程】

本实例工程添加一个头文件 Common.h 和源文件 Common.cpp，其头文件内容如下：

```
01 #ifdef _DEBUG
02 #pragma comment(lib, "osgd.lib")
03 #pragma comment(lib, "osgDBd.lib")
04 #pragma comment(lib, "osgViewerd.lib")
05 #pragma comment(lib, "OpenThreads.d.lib")
06 #pragma comment(lib, "osgGAd.lib")
07 #pragma comment(lib, "osgUtil.d.lib")
08 #pragma comment(lib, "osgTextd.lib")
09 #pragma comment(lib, "osgWidgetd.lib")
10 #pragma comment(lib, "glu32.lib")
11 #pragma comment(lib, "opengl32.lib")
12 #pragma comment(lib, "osgShadowd.lib")
13 #else
14 #pragma comment(lib, "osg.lib")
15 #pragma comment(lib, "osgDB.lib")
16 #pragma comment(lib, "osgViewer.lib")
17 #pragma comment(lib, "OpenThreads.lib")
18 #pragma comment(lib, "osgUtil.lib")
19 #pragma comment(lib, "osgGA.lib")
20 #pragma comment(lib, "osgText.lib")
21 #pragma comment(lib, "osgWidget.lib")
22 #pragma comment(lib, "glu32.lib")
23 #pragma comment(lib, "opengl32.lib")
24 #pragma comment(lib, "osgShadow.lib")
25 #endif
```

源文件中只写一个空的 main()函数体，代码如下：

```
01 #include "Common.h"
02 void main()
03 {}
```



【代码解析】

在头文件中,第 02~12 行是 debug 编译版本所需包含的 lib 文件,第 14~24 行是 release 版本所需包含的 lib 文件。



注意: 在使用 OSG 语言开发功能时,有些插件不支持 debug 版本,所以编译 release 版本。



实例 289 加载和显示三维资源

【实例描述】

OSG 同 OpenCV 语言一样,也有自己的窗口。本实例读取模型文件 glider.osg,并在窗口中显示,运行效果如图 16-1 所示。

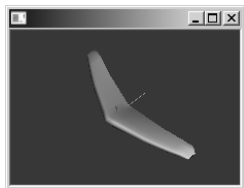


图 16-1 加载和显示三维资源

【实现过程】

OSG 语言的变量大都由智能指针申请,为了读取和显示模型,需要将其存入节点(变量 node),然后由视图变量(viewer)显示,代码如下:

```
01 #include "../Common/Common.h"
02 #include <osgViewer/Viewer>
03 #include <osgDB/ReadFile>
04 #include <osgViewer/ViewerEventHandlers>
05
06 void main()
07 {
08     osg::ref_ptr<osgViewer::Viewer> viewer = new osgViewer::Viewer; //视图
09     osg::ref_ptr<osg::Node> node = new osg::Node; //根节点
10     node = osgDB::readNodeFile("../data/glider.osg"); //载入资源
11     viewer->setSceneData(node.get()); //设置场景
12     viewer->addEventHandler(new osgViewer::WindowSizeHandler); //改变窗口大小
13     viewer->run(); //运行
14 }
```

【代码解析】

第 01~04 行是实现该功能必须包含的头文件,第 08、09 行定义视图变量 viewer 和节点变量 node。第 10 行读取资源 glider.osg 到变量 node 中,第 11 行设置场景显示功能。第 12 行添加事件响应功能,即可以改变窗口大小,第 13 行开始运行。



注意: 事实上,OSG 语言是由 OpenGL 语言发展而来的。而 OpenGL 语言必须在 C++ 语言的语法支持下才能进一步封装。



实例 290 绘制长方体

【实例描述】

在三维仿真技术中,需要绘制很多实物,而这些实物的建模少不了面片、直线、点等的模



拟。本实例利用函数 `Box()` 实现长方体的绘制，运行效果如图 16-2 所示。

【实现过程】

定义函数 `CreateCube()` 以实现长方体的创建，其中可以设置其位置、大小和颜色等属性，代码如下：

```
01 #include "../Common/Common
02 #include <osgViewer/Viewer>
03 #include <osgDB/ReadFile>
04 #include <osg/Geode>
05 #include <osg/ShapeDrawable>
06 #include <osgViewer/ViewerEventHandlers>
07
08 osg::ref_ptr<osg::Node> CreateCube()           //创建长方体
09 {
10     osg::ref_ptr<osg::Geode> gnode = new osg::Geode;           //节点
11     osg::ref_ptr<osg::ShapeDrawable> cube = new osg::ShapeDrawable(
12         new osg::Box(osg::Vec3(0, 0.0, 0), 5.0, 1.0, 10.0)); //绘制长方体
13     cube->setColor(osg::Vec4(1.0, 0.0, 0.0, 1.0));           //设置颜色
14     gnode->addDrawable(cube);                               //添加节点
15     return gnode;                                           //返回节点
16 }
17 void main()
18 {
19     osg::ref_ptr<osgViewer::Viewer> viewer = new osgViewer::Viewer; //视图
20     osg::ref_ptr<osg::Node> node = CreateCube();
21     viewer->setSceneData(node);                               //设置场景
22     viewer->addEventHandler(new osgViewer::WindowSizeHandler); //改变窗口大小
23     viewer->run();                                           //运行
24 }
```

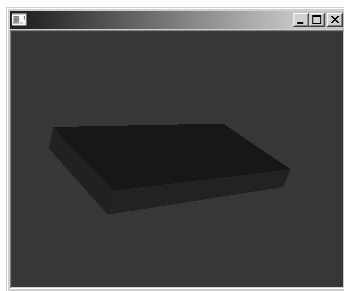


图 16-2 绘制长方体

【代码解析】

第 08~16 行是创建长方体的函数 `CreateCube()` 定义体，第 10 行定义节点变量 `gnode`，第 11、12 行创建长方体。其中，第 1 个参数是长方体所在位置，后 3 个参数是在 X、Y 和 Z 轴向上的长度。第 13 行设置颜色，其为红色。第 14 行添加节点，第 20 行加载长方体资源。



实例 291 模型贴图

【实例描述】

实例 290 绘制的长方体只做了基本属性的设置，在虚拟场景的仿真中可以利用长方体模拟建筑物。除了需要绘制长方体，还需要贴图技术的辅助以显示其仿真的真实感。运行效果如图 16-3 所示。

【实现过程】

本实例实现的功能只需在函数 `CreateCube()` 中添加新内容即可，定义二维纹理变量 `texture2D` 和图像变量 `image`。先从文件 `texture.jpg` 中读取图像数据，再映射到 `texture2D` 中，最后开启节点的纹理状态，即可实现。具体代码如下：



图 16-3 模型贴图



```

01 #include "../Common/Common.h"
02 #include <osgViewer/Viewer>
03 #include <osgDB/ReadFile>
04 #include <osgViewer/ViewerEventHandlers>
05 #include <osg/StateSet>
06 #include <osg/Geode>
07 #include <osg/ShapeDrawable>
08 #include <osg/Image>
09 #include <osg/Texture2D>
10
11 osg::ref_ptr<osg::Node> CreateCube()           //创建长方体
12 {
13     osg::ref_ptr<osg::Geode> gnode = new osg::Geode;           //节点
14     osg::ref_ptr<osg::ShapeDrawable> cube = new osg::ShapeDrawable(
15         new osg::Box(osg::Vec3(0, 0.0, 0), 5.0, 1.0, 10.0)); //绘制长方体
16     osg::ref_ptr<osg::Texture2D> texture2D = new osg::Texture2D; //纹理
17     osg::ref_ptr<osg::Image> image;           //图像
18     cube->setColor(osg::Vec4(1.0, 0.0, 0.0, 1.0));           //设置颜色
19
20     image = osgDB::readImageFile("../data/texture.jpg");     //设置纹理
21     if(image.valid())           //有效
22         texture2D->setImage(image.get());           //映射图像
23     gnode->getOrCreateStateSet()->setTextureAttributeAndModes(
24         0, texture2D.get(), osg::StateAttribute::ON);           //开启纹理状态
25     gnode->addDrawable(cube);           //添加节点
26     return gnode;           //返回节点
27 }
28 void main()
29 {
30     ...
31 }

```

【代码解析】

第 01~09 行是本实例实现贴图功能必须包含的头文件，从字面意思可以看出，第 08、09 行是图像和纹理的头文件。第 16、17 行定义纹理和图像变量。第 20 行读取图像 texture.jpg，第 21、22 行判断图像是否读取成功，如果成功，则执行第 22 行。第 23、24 行开启该节点的纹理设置状态。



注意：C++语言可以被封装为实现某种领域功能的新语言，比如 OpenGL、OpenCV 等。



实例 292 Shader 着色器

【实例描述】

Shader 是目前较先进的技术，本实例使用 Shader 着色器给模型上色。程序运行效果如图 16-4 所示。

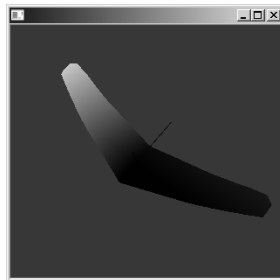


图 16-4 Shader 着色器应用

【实现过程】

定义静态变量 `vert` 和 `frag`，分别作为顶点着色器和片元着色器，利用它们对模型进行着色。具体实现代码如下：



```

01 #include "../Common/Common.h"
02 #include <osgViewer/Viewer>
03 #include <osgDB/ReadFile>
04 #include <osg/Program>
05 #include <osg/Shader>
06 #include <osgViewer/ViewerEventHandlers>
07
08 static const char* vert={                                //写一个顶点着色器
09     "varying vec4 color;\n"
10     "void main()\n"
11     "{\n"
12     "    color = gl_Vertex;\n"
13     "    gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;\n" //位置
14     "}\n"
15 };
16 static const char* frag={                                //片元着色器
17     "varying vec4 color;\n"
18     "void main()\n"
19     "{\n"
20     "    gl_FragColor = clamp(color, 0.0, 1.0);\n" //颜色
21     "}\n"
22 };
23 void main()
24 {
25     osg::ref_ptr<osgViewer::Viewer> viewer = new osgViewer::Viewer;//视图变量
26     osg::ref_ptr<osg::Node> nd=osgDB::readNodeFile("../data/glider.osg");
27                                     //加载资源
28     osg::ref_ptr<osg::StateSet> ss=nd->getOrCreateStateSet();//设置状态
29     osg::ref_ptr<osg::Program> pg=new osg::Program;
30     pg->addShader(new osg::Shader(osg::Shader::VERTEX, vert));//顶点着色器
31     pg->addShader(new osg::Shader(osg::Shader::FRAGMENT, frag));//片元着色器
32     ss->setAttributeAndModes(pg, osg::StateAttribute::ON); //设置属性
33     viewer->setSceneData(nd); //设置场景
34     viewer->addEventHandler(new osgViewer::WindowSizeHandler);//改变窗口大小
35     viewer->run();
36 }

```

【代码解析】

第 08~15 行是顶点着色器 `vert` 的定义，其中，第 12 行是颜色变量 `color`，第 13 行是位置变量的定义。第 16~22 行是片元着色器 `frag` 的定义，其中，第 20 行定义片元颜色。第 27 行定义变量 `ss` 用于设置状态。第 28~31 行设置顶点和片元着色器变量。第 32 行设置场景。第 34 行开始运行程序。



注意：第 08~22 行是 Shader 着色器的书写格式。



实例 293 虚拟场景漫游

【实例描述】

本实例模拟虚拟场景的漫游，即按键 A、D、W 和 S 实现向左、右、前和后移动视点。按键 End 和 Home 分别实现向下和向上移动视点。具体运行效果如图 16-5 所示。

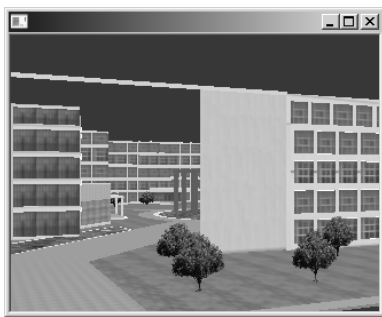


图 16-5 虚拟场景的漫游

【实现过程】

定义类 `TravelManipulator` 实现场景的漫游，按键事件的响应由函数 `handle()` 完成。首先列出类 `TravelManipulator` 的申请及漫游接口的实现函数定义体，代码如下：

```

01 #include "../Common/Common.h"
02 #include <osgViewer/Viewer>
03 #include <osgDB/ReadFile>
04 #include <osgGA/GUIEventAdapter>
05 #include <osgViewer/ViewerEventHandlers>
06 #include <osg/Matrixd>
07
08 class TravelManipulator:public osgGA::MatrixManipulator
09 {
10 public:
11     TravelManipulator();
12     //实现得到和设置矩阵接口
13     virtual void setByMatrix(const osg::Matrixd& matrix);
14     virtual void setByInverseMatrix(const osg::Matrixd& matrix);
15     virtual osg::Matrixd getMatrix() const;
16     virtual osg::Matrixd getInverseMatrix()const;
17     //相应事件
18     virtual bool handle(const osgGA::GUIEventAdapter& ea, osgGA::GUIActionA
19                         dapter&us);
20     void ChangePosition(osg::Vec3d&delta);           //改变位置
21 private:
22     osg::Vec3 m_vPosition;                          //视点
23     osg::Vec3 m_vRotation;                          //朝向
24     float m_vStep;                                  //移动步长
25 };
26 TravelManipulator::TravelManipulator()
27 {
28     m_vPosition=osg::Vec3(0,0,10.0);               //位置
29     m_vRotation=osg::Vec3(osg::PI_2,0,0);          //角度
30     m_vStep=1.0;                                    //改变步长
31 }
32 void TravelManipulator::setByMatrix(const osg::Matrixd & matrix)
33 {}
34 void TravelManipulator::setByInverseMatrix(const osg::Matrixd& matrix)
35 {}
36 osg::Matrixd TravelManipulator::getMatrix() const
37 {
38     osg::Matrixd mat;
39     mat.makeTranslate(m_vPosition);
40     return osg::Matrixd::rotate(m_vRotation[0],osg::X_AXIS,m_vRotation[1],

```



```
41  osg::Y_AXIS,m_vRotation[2],osg::Z_AXIS)*mat;
42  }
43  osg::Matrixd TravelManipulator::getInverseMatrix() const
44  {
45      osg::Matrixd mat;
46      mat.makeTranslate(m_vPosition);
47      return osg::Matrixd::inverse(osg::Matrixd::rotate(m_vRotation[0],osg::X_AXIS,
48  m_vRotation[1],osg::Y_AXIS,m_vRotation[2],osg::Z_AXIS)*mat);
49  }
```

接下来，给出函数 `handle()` 和 `ChangePosition()` 的定义体，代码如下：

```
01  bool TravelManipulator::handle(const osgGA::GUIEventAdapter& ea,
02  osgGA::GUIActionAdapter& us)
03  {
04      switch(ea.getEventType())
05      {
06          case osgGA::GUIEventAdapter::KEYDOWN:                //按键事件响应
07              {
08                  if(ea.getKey()=='w' || ea.getKey()=='W')      //向前
09                  {
10                      int data_x, data_y;                      //新的视点坐标
11                      data_x = m_vStep*cosf(osg::PI_2+m_vRotation._v[2]);
12                      data_y = m_vStep*sinf(osg::PI_2+m_vRotation._v[2]);
13                      ChangePosition(osg::Vec3d(data_x,data_y,0)); //改变视点位置
14                      return true;
15                  }
16                  else if(ea.getKey()=='s' || ea.getKey()=='S') //向后
17                  {
18                      int data_x, data_y;                      //新的视点坐标
19                      data_x = -m_vStep*cosf(osg::PI_2+m_vRotation._v[2]);
20                      data_y = -m_vStep*sinf(osg::PI_2+m_vRotation._v[2]);
21                      ChangePosition(osg::Vec3d(data_x,data_y,0)); //改变视点位置
22                      return true;
23                  }
24                  else if(ea.getKey()=='a' || ea.getKey()=='A') //向左
25                  {
26                      int data_x, data_y;                      //新的视点坐标
27                      data_x = m_vStep*sinf(-osg::PI_2+m_vRotation._v[2]);
28                      data_y = m_vStep*cosf(osg::PI_2+m_vRotation._v[2]);
29                      ChangePosition(osg::Vec3d(data_x,data_y,0)); //改变视点位置
30                      return true;
31                  }
32                  else if(ea.getKey()=='d' || ea.getKey()=='D') //向右
33                  {
34                      int data_x, data_y;                      //新的视点坐标
35                      data_x = m_vStep*sinf(osg::PI_2+m_vRotation._v[2]);
36                      data_y = -m_vStep*cosf(osg::PI_2+m_vRotation._v[2]);
37                      ChangePosition(osg::Vec3d(data_x,data_y,0)); //改变视点位置
38                      return true;
39                  }
40                  else if(ea.getKey()==osgGA::GUIEventAdapter::KEY_Home) //向上
41                  {
42                      ChangePosition(osg::Vec3d(0,0,m_vStep)); //改变视点位置
43                      return true;
44                  }
45                  else if(ea.getKey()==osgGA::GUIEventAdapter::KEY_End) //向下
46                  {
47                      ChangePosition(osg::Vec3d(0,0,-m_vStep)); //改变视点位置
48                      return true;
49                  }
49      }
```



```

50         else
51         {}
52     }
53 }
54     return false;
55 }
56 void TravelManipulator::ChangePosition(osg::Vec3d &delta)
57 {
58     m_vPosition+=delta;           //实现位置变换
59 }

```

最后, 给出 main()函数的实现, 首先加载场景资源 ceep.ive, 然后设置漫游事件。代码如下:

```

01 void main()
02 {
03     osg::ref_ptr<osgViewer::Viewer> viewer = new osgViewer::Viewer; //视图
04     osg::ref_ptr<osg::Node> gp = new osg::Node; //节点
05     gp=osgDB::readNodeFile("../data/ceep.ive"); //加载资源
06     viewer->setSceneData(gp.get()); //设置场景
07     viewer->setCameraManipulator(new TravelManipulator()); //设置漫游
08     viewer->addEventHandler(new osgViewer::WindowSizeHandler); //设置窗口大小
09     viewer->run();
10 }

```

【代码解析】

在第 1 段代码中, 第 08~25 行是类 TravelManipulator 的声明, 其中, 第 11 行是构造函数, 第 13~20 行是成员函数, 第 22~24 行是成员变量。第 26~31 行的构造函数用于确定初始视图位置及角度, 变量 m_vStep 用于确定每次移动位置时的步长。第 32~49 行的函数用于定义矩阵计算接口。

在第 2 段代码的 handle()函数中, 第 06 行表示按键事件响应, 其中, ea.getKey()分别获取按键字符, 并做相应的位置移动。第 56~59 行是函数 ChangePosition()的定义体, 实现变量 m_vPosition 的改变。

在第 3 段代码中, 第 05 行加载资源, 在第 06 行设置场景后由第 07 行设置漫游事件。



实例 294 HUD 应用（显示二维文字）

【实例描述】

实例 293 实现了在虚拟场景中的漫游功能, 为了显示说明性文字, 本实例给出 HUD 的应用, 运行效果如图 16-6 所示。



图 16-6 显示说明性文字



【实现过程】

在实例 293 的基础上, 添加函数 `CreateHUD()` 以实现说明性文字的显示, 具体代码如下:

```
01 #include "../Common/Common.h"
02 #include <osgViewer/ViewerEventHandlers>
03 #include <osgDB/ReadFile>
04 #include <osgViewer/Viewer>
05 #include <osg/Geode>
06 #include <osg/CameraNode>
07 #include <osgText/Text>
08 #include <osg/Matrixd>
09
10 osg::ref_ptr<osg::Camera> createHUD(unsigned int width,unsigned int height)
11 {
12     osg::ref_ptr<osg::Camera> camera = new osg::Camera; //说明性文字
13     osg::Geode* geode = new osg::Geode();
14     std::string ziti("../data/simhei.ttf");
15     float size=25*width/1366; //字体大小
16     osg::StateSet* stateset = geode->getOrCreateStateSet(); //设置状态, 关闭灯光
17     stateset->setMode(GL_LIGHTING,osg::StateAttribute::OFF);
18     osgText::Text* text1 = new osgText::Text;
19     osgText::Text* text2 = new osgText::Text;
20     osgText::Text* text3 = new osgText::Text;
21     osgText::Text* text4 = new osgText::Text;
22     osgText::Text* text5 = new osgText::Text;
23     osgText::Text* text6 = new osgText::Text;
24
25     geode->addDrawable(text1); //第 1 行
26     text1->setFont(ziti); //字体类型, 下同
27     text1->setCharacterSize(size); //字体大小, 下同
28     text1->setPosition(osg::Vec3(10*width/1366, (398*height/768), 0)); //字体位置, 下同
29     text1->setText(L"按键'w/W': 前进"); //每行内容, 下同
30     text1->setColor(osg::Vec4f(1.0f,1.0f,1.0f,1.0f)); //字体颜色
31
32     geode->addDrawable(text2); //第 2 行
33     text2->setFont(ziti);
34     text2->setCharacterSize(size);
35     text2->setPosition(osg::Vec3(10*width/1366, (358*height/768), 0));
36     text2->setText(L"按键's/S': 后退");
37     text2->setColor(osg::Vec4f(1.0f,1.0f,1.0f,1.0f));
38
39     geode->addDrawable(text3); //第 3 行
40     text3->setFont(ziti);
41     text3->setCharacterSize(size);
42     text3->setPosition(osg::Vec3(10*width/1366, (318*height/768), 0));
43     text3->setText(L"按键'a/A': 左移");
44     text3->setColor(osg::Vec4f(1.0f,1.0f,1.0f,1.0f));
45
46     geode->addDrawable(text4); //第 4 行
47     text4->setFont(ziti);
48     text4->setCharacterSize(size);
49     text4->setPosition(osg::Vec3(10*width/1366, (278*height/768), 0));
50     text4->setText(L"按键'd/D': 右移");
51     text4->setColor(osg::Vec4f(1.0f,1.0f,1.0f,1.0f));
52
53     geode->addDrawable(text5); //第 5 行
54     text5->setFont(ziti);
55     text5->setCharacterSize(size);
```



```

56     text5->setPosition(osg::Vec3(10*width/1366, (238*height/768), 0));
57     text5->setText(L"按键'Home': 上升");
58     text5->setColor(osg::Vec4f(1.0f, 1.0f, 1.0f, 1.0f));
59
60     geode->addDrawable(text6); //第6行
61     text6->setFont(ziti);
62     text6->setCharacterSize(size);
63     text6->setPosition(osg::Vec3(10*width/1366, (198*height/768), 0));
64     text6->setText(L"按键'End': 下降");
65     text6->setColor(osg::Vec4f(1.0f, 1.0f, 1.0f, 1.0f));
66     //设置相机及透视矩阵
67     camera->setProjectionMatrix(osg::Matrix::ortho2D(0, width, 0, height));
68                                     //透视矩阵
69     camera->setReferenceFrame(osg::Transform::ABSOLUTE_RF);
70     camera->setViewMatrix(osg::Matrix::identity()); //视点矩阵
71     camera->setClearMask(GL_DEPTH_BUFFER_BIT); //清除机制
72     camera->setRenderOrder(osg::Camera::POST_RENDER); //设置渲染顺序
73     camera->addChild(geode);
74     return camera;
75 };

```

然后给出本实例的 main() 函数定义，代码如下：

```

01 void main()
02 {
03     unsigned int W, H; //宽度高度
04     //获得系统分辨率
05     osg::GraphicsContext::WindowingSystemInterface* wsi=
06         osg::GraphicsContext::getWindowingSystemInterface();
07     if(!wsi)
08         return;
09     wsi->getScreenResolution(osg::GraphicsContext::ScreenIdentifier(0), W, H);
10     //设置图形环境特性
11     osg::ref_ptr<osg::GraphicsContext::Traits> traits=new osg::GraphicsCon
12     text::Traits();
13     traits->x=0;
14     traits->y=0;
15     traits->width=W;
16     traits->height=H;
17     traits->windowDecoration=false;
18     traits->doubleBuffer=true;
19     traits->sharedContext=0;
20     //创建图形环境特性
21     osg::ref_ptr<osg::GraphicsContext> gc=
22         osg::GraphicsContext::createGraphicsContext(traits.get());
23     if(gc->valid())
24     {
25         osg::notify(osg::INFO)<<"创建已成功"<<std::endl;
26         gc->setClearColor(osg::Vec4f(0.2f, 0.2f, 0.6f, 1.0f));
27         gc->setClearMask(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
28     }
29     else
30     {
31         osg::notify(osg::NOTICE)<<"创建失败"<<std::endl;
32         osg::ref_ptr<osgViewer::Viewer> viewer = new osgViewer::Viewer; //视图
33         osg::ref_ptr<osg::Node> nd = new osg::Node; //节点
34         osg::ref_ptr<osg::Group> gp=new osg::Group;
35         nd=osgDB::readNodeFile("../data/ceep.ive"); //加载资源
36         gp->addChild(nd.get()); //添加资源节点
37         gp->addChild(createHUD(W, H)); //创建 HUD
38         viewer->setSceneData(gp.get()); //设置场景

```



```

37     viewer->setCameraManipulator(new TravelManipulator()); //设置视口
38     viewer->addEventHandler(new osgViewer::WindowSizeHandler); //设置窗口大小
39     viewer->run();
40 }

```

【代码解析】

在第 1 段代码中,第 12 行定义相同的节点,用于返回值。第 13 行定义字体节点,第 14 行为字体类型,第 15 行定义字体的大小。第 16、17 行关闭字体的灯光,第 18~23 行定义字体变量。第 25~65 行定义每行字体的属性,第 67~72 行定义字体相机的属性及其透视矩阵。

在第 2 段代码中,第 03 行定义屏幕的分辨率保存值,分为宽度及高度。第 05~25 行获取屏幕分辨率。第 33 行加载外部资源 ceep.ive。第 35 行添加节点,以创建 HUD。



注意: 获取屏幕分辨率的目的是当屏幕尺寸发生变化时,文字的位置及大小也能随之改变。



实例 295 显示三维文字

【实例描述】

除了可以实现二维文字的显示,还可以实现三维文字的应用。本实例演示三维文字的显示,运行效果如图 16-7 所示。



图 16-7 显示三维文字

【实现过程】

定义函数 create3DText() 以实现 3D 文字的绘制,其返回值的类型为 Geode。本实例实现具体代码如下:

```

01 #include "../Common/Common.h"
02 #include <osgViewer/ViewerEventHandlers>
03 #include <osgText/Text3D>
04 #include <osgText/Font3D>
05 #include <osgViewer/Viewer>
06 #include <osg/Geode>
07 #include <osg/Group>
08
09 osg::ref_ptr<osg::Geode> create3DText() //3D 文字
10 {

```



```

11     osg::ref_ptr<osg::Geode> geode = new osg::Geode();           //节点
12     osg::ref_ptr<osgText::Text3D> text=new osgText::Text3D();   //3D 文字
13     text->setText(L"Hello, 祝您新年快乐");                       //内容
14     text->setFont("../data/simhei.ttf");                          //字体
15     text->setCharacterSize(60.0f);                                //大小
16     text->setPosition(osg::Vec3(0.0f,0.0f,0.0f));                 //位置
17     text->setRenderMode(osgText::Text3D::PER_GLYPH);             //渲染模式
18     text->setCharacterDepth(100.0f);                              //字体深度
19     geode->addDrawable(text.get());                               //添加文字
20     return geode.get();
21 }
22 void main()
23 {
24     osg::ref_ptr<osgViewer::Viewer> viewer=new osgViewer::Viewer(); //视图
25     osg::ref_ptr<osg::Group> gp=new osg::Group;                  //节点
26     gp->addChild(create3DText());                                //添加 3D 文字
27     viewer->addEventHandler(new osgViewer::WindowSizeHandler()); //设置窗口大小
28     viewer->setSceneData(gp.get());                               //设置场景
29     viewer->run();
30 }

```

【代码解析】

第 01~07 行是程序实现所需包含的头文件，第 09~21 行是函数 create3DText() 的定义体。其中，第 11 行定义节点变量 **geode**。第 12 行定义 3D 字体变量 **text**。第 13 行定义字体内容。第 14~18 行定义 3D 字体的各类属性，比如大小、位置及字体深度。第 19 行将文字添加到节点 **geode**。第 26 行添加 3D 文字到 **osg::Group** 型节点 **gp** 处，以在虚拟场景中显示。



注意：如果字体的类型只是英文字体，当文字内容有中文时，将不显示中文内容。



实例 296 添加光源

【实例描述】

本实例模拟光源的应用，介绍如何给模型添加光源，使用类 **osg::Light** 实现设置。本实例的光源位于负 X 轴的 1.0 和负 Y 轴的 1.0 处，运行效果如图 16-8 所示。

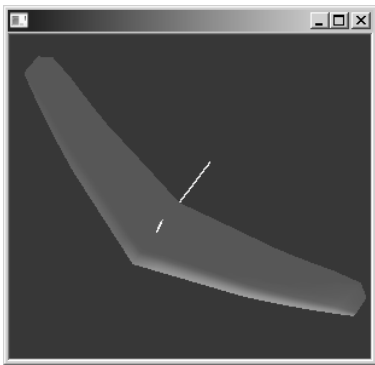


图 16-8 给模型添加光源



【实现过程】

定义函数 `createlight()` 实现光源的创建，返回变量类型是 `osg::Group`，具体代码如下：

```
01 #include "../Common/Common.h"
02 #include <osgViewer/Viewer>
03 #include <osg/Group>
04 #include <osg/Light>
05 #include <osg/LightSource>
06 #include <osgViewer/ViewerEventHandlers>
07 #include <osgDB/ReadFile>
08
09 osg::ref_ptr<osg::Group> createlight(osg::ref_ptr<osg::Node> node)
10 {
11     osg::ref_ptr<osg::Group> lightgroup=new osg::Group();
12     lightgroup->addChild(node);
13     osg::ref_ptr<osg::StateSet> stateset=new osg::StateSet();//设置状态
14     stateset=lightgroup->getOrCreateStateSet();
15     stateset->setMode(GL_LIGHTING,osg::StateAttribute::ON); //开启光源
16     osg::ref_ptr<osg::Light>light=new osg::Light();           //light 对象
17     light->setLightNum(0);                                     //设置光数量
18     light->setDirection(osg::Vec3(1.0f,0.0f,-1.0f));           //设置光方向
19     light->setPosition(osg::Vec4(-1.0,-1.0,0.0,1.0));          //设置光位置
20     light->setAmbient(osg::Vec4(1.0f,1.0f,1.0f,1.0f));        //设置环境光
21     light->setDiffuse(osg::Vec4(1.0f,1.0f,1.0f,1.0f));        //设置散射光
22     osg::ref_ptr<osg::LightSource> lightsource=new osg::LightSource();
                                                                    //创建光源
23     lightsource->setLight(light.get());                         //添加光变量
24     lightgroup->addChild(lightsource.get());                   //添加光源
25     return lightgroup.get();
26 }
27 void main()
28 {
29     osg::ref_ptr<osgViewer::Viewer>viewer=new osgViewer::Viewer();
30     osg::ref_ptr<osg::Group> gp=new osg::Group();
31     osg::ref_ptr<osg::Node> nd=osgDB::readNodeFile("../data/glider.osg");
32     gp->addChild(createlight(nd));
33     viewer->setSceneData(gp.get());
34     viewer->addEventHandler(new osgViewer::WindowSizeHandler());
35     viewer->run();
36 }
```

【代码解析】

第 12 行给光源节点添加模型节点，第 13~15 行设置节点状态属性，将光源属性开启。第 17~21 行设置光的各种属性，比如数量、方向、位置、环境光和散射光等。第 22 行设置光源。第 23 行添加光变量节点。第 24 行将光源节点添加到总节点。



实例 297 缩放模型

【实例描述】

本实例实现模型的缩放，利用 `osg::PositionAttitudeTransform` 类不仅可以缩放模型，还可以绕某个轴旋转模型。运行效果如图 16-9 所示，其下边是原模型，右上角是缩放一半后的模型。

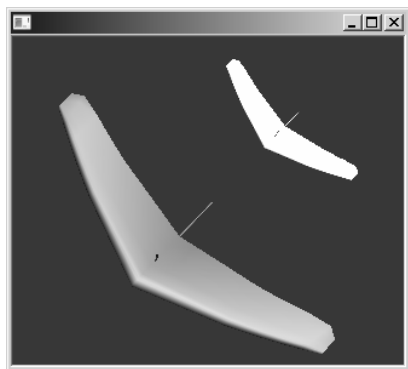


图 16-9 模型的缩放

【实现过程】

定义 `osg::PositionAttitudeTransform` 型变量 `pat`，以添加 `glider.osg` 节点，实现模型的缩放。具体代码如下：

```

01 #include "../Common/Common.h"
02 #include <osgViewer/Viewer>
03 #include <osgDB/ReadFile>
04 #include <osgViewer/ViewerEventHandlers>
05 #include <osg/PositionAttitudeTransform>
06
07 void main()
08 {
09     osg::ref_ptr<osgViewer::Viewer> viewer=new osgViewer::Viewer;
10     osg::ref_ptr<osg::Node> nd=osgDB::readNodeFile("../data/glider.osg");
11     osg::ref_ptr<osg::Group> gp=new osg::Group;
12     osg::ref_ptr<osg::PositionAttitudeTransform>pat=
13         new osg::PositionAttitudeTransform();           //位置变量
14     pat->addChild(nd.get());                             //添加节点
15     pat->setAttitude(osg::Quat(0,osg::X_AXIS));          //绕 X 轴旋转 0 度
16     pat->setScale(osg::Vec3(0.5f,0.5f,0.5f));           //缩放一半
17     pat->setPosition(osg::Vec3(1.0,0.0,0.0));           //设置位置
18     gp->addChild(nd.get());                             //原模型
19     gp->addChild(pat.get());                             //缩放后模型
20     viewer->setSceneData(gp.get());                      //设置场景
21     viewer->addEventHandler(new osgViewer::WindowSizeHandler()); //设置窗口大小
22     viewer->run();                                       //运行
23 }
```

【代码解析】

第 12、13 行定义位置变量。第 14 行添加模型节点。第 15 行设置绕 X 轴旋转。第 16 行设置模型尺寸的缩放。第 17 行设置目标模型位置。第 18、19 行是将原模型和缩放后的模型都添加到节点 `gp` 中。



注意：上述 OSG 代码是利用 C++ 语言实现不同功能的函数应用。因此，学习本章有助于加深对 C++ 语言应用的认识。



实例 298 利用粒子系统制作火焰

【实例描述】

本实例利用粒子及结合纹理贴图实现火焰的仿真，在粒子系统中需要创建模板、设置粒子大小、关联粒子发射器及更新器等。火焰仿真效果如图 16-10 所示。

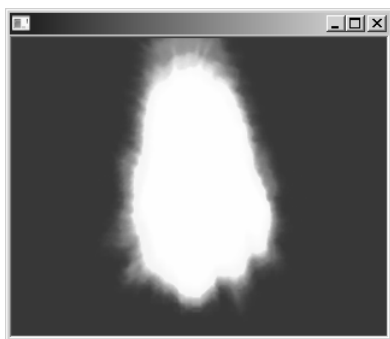


图 16-10 粒子系统仿真的火焰

【实现过程】

创建粒子系统模板、设置粒子的颜色、位置及速度等属性，以实现火焰的仿真，具体代码如下：

创建发射器和计数器，调整每一帧增加的粒子数目。

```
01 #include "../Common/Common.h"
02 #include <osgViewer/Viewer>
03 #include <osg/Geode>
04 #include <osgParticle/ParticleSystemUpdater>
05 #include <osgParticle/ModularEmitter>
06 #include <osgParticle/ModularProgram>
07 #include <osgParticle/LinearInterpolator>
08 #include <osgViewer/ViewerEventHandlers>
09
10 void main()
11 {
12     osg::ref_ptr<osgViewer::Viewer> viewer=new osgViewer::Viewer;
13     osg::ref_ptr<osg::Group> gp=new osg::Group;
14     osg::ref_ptr<osg::Geode> geode=new osg::Geode;
15     osgParticle::Particle ptemplate; //粒子系统模板
16     ptemplate.setShape(osgParticle::Particle::QUAD); //粒子形状
17     ptemplate.setLifeTime(1.0); //生命周期单位: 秒
18     ptemplate.setSizeRange(osgParticle::rangef(2.0, 1.0)); //大小
19     ptemplate.setColorRange(osgParticle::rangev4(
20         osg::Vec4(1.0f, 0.5f, 0.3f, 1.0f),
21         osg::Vec4(0.0f, 0.7f, 1.0f, 0.0f))); //颜色
22     ptemplate.setRadius(1.2f); //半径
23     ptemplate.setMass(10.0f); //设置面积
24     ptemplate.setPosition(osg::Vec3(0.0f, 0.0f, 0.0f)); //设置位置
25     ptemplate.setVelocity(osg::Vec3(0.0f, 0.0f, 0.0f)); //速度
26     ptemplate.setSizeInterpolator(new osgParticle::LinearInterpolator);
27     ptemplate.setAlphaInterpolator(new osgParticle::LinearInterpolator);
```



```

28     osg::ref_ptr<osgParticle::ParticleSystem> ps=
29         new osgParticle::ParticleSystem();           //创建粒子系统
30     ps->setDataVariance(osg::Node::STATIC);
31     ps->setDefaultAttributes("../data/1.bmp", true, false); //设置纹理
32     ps->setDefaultParticleTemplate(ptemplate);         //添加模板
33     osg::ref_ptr<osgParticle::RandomRateCounter> counter=
34         new osgParticle::RandomRateCounter();         //计数器
35     counter->setRateRange(osgParticle::range(30, 50)); //每秒添加的粒子个数
36     counter->setDataVariance(osg::Node::DYNAMIC);     //自动改变粒子
37     osg::ref_ptr<osgParticle::PointPlacer> placer=
38         new osgParticle::PointPlacer();               //设置一个点放置器
39     placer->setCenter(osg::Vec3(0.0f, 0.0f, 0.05f));   //设置位置
40     placer->setDataVariance(osg::Node::DYNAMIC);       //数据变化
41     osg::ref_ptr<osgParticle::RadialShooter> shooter=
42         new osgParticle::RadialShooter();             //弧度发射器
43     shooter->setInitialSpeedRange(osgParticle::range(1, 5)); //初始速度
44     shooter->setDataVariance(osg::Node::DYNAMIC);     //数据变化属性
45     shooter->setThetaRange(-0.01f, 0.01f);           //弧度值
46     osg::ref_ptr<osgParticle::ModularEmitter> emitter=
47         new osgParticle::ModularEmitter();           //粒子放射器
48     emitter->setDataVariance(osg::Node::DYNAMIC);
49     emitter->setParticleSystem(ps.get());             //关联粒子系统
50     emitter->setCounter(counter.get());               //关联计数器
51     emitter->setPlacer(placer.get());                 //关联点放置器
52     emitter->setShooter(shooter.get());               //关联发射器
53     gp->addChild(emitter.get());                       //把放射器添加到节点
54     osg::ref_ptr<osgParticle::ModularProgram> program=
55         new osgParticle::ModularProgram();           //标准编程器对象
56     program->setParticleSystem(ps.get());             //关联粒子系统
57     osg::ref_ptr<osgParticle::ParticleSystemUpdater> psu=
58         new osgParticle::ParticleSystemUpdater();    //更新器
59     psu->addParticleSystem(ps.get());                 //关联粒子系统
60     gp->addChild(psu.get());                           //添加更新器
61     gp->addChild(program.get());
62     geode->addDrawable(ps);
63     gp->addChild(geode);
64     viewer->setSceneData(gp);
65     viewer->addEventHandler(new osgViewer::WindowSizeHandler());
66     viewer->run();
67 }

```

【代码解析】

第12~14行定义实现火焰的视图及节点变量,第15行定义粒子系统模板,第16~27行是对粒子模板属性的设置,第28行创建粒子系统,第29~32行设置粒子系统属性,第33、34行定义计数器变量,第35、36行设置计数器属性,第37~45行定义并设置点放置器和弧度发射器,第46~52行将上述变量关联到放射器变量 emitter,最后在第53~63行实现粒子系统各相关元素的关联,并将粒子节点添加到总节点 gp。



注意: 上述的各类放置器、发射器还有很多不同的类型,请根据具体的应用功能选择不同的类型。比如,除点放置器外,还有线或面放置器。



实例 299 模拟雾效

【实例描述】

利用实例 298，改变参数可以实现不同的粒子特效。当然，OSG 本身也实现了很多既定的特效，例如雾、雨、雪等。本实例实现虚拟城市的雾效，运行效果如图 16-11 所示。



图 16-11 模拟雾效

【实现过程】

定义函数 `createFog()` 创建雾效节点，在该函数中设置雾效的颜色、密度及模式等属性，具体代码如下：

```
01 #include "../Common/Common.h"
02 #include<osgViewer/Viewer>
03 #include<osg/Node>
04 #include<osg/Fog>
05 #include<osg/Group>
06 #include <osgViewer/ViewerEventHandlers>
07 #include<osgDB/ReadFile>
08
09 //创建雾
10 osg::ref_ptr<osg::Fog> createFog()
11 {
12     osg::ref_ptr<osg::Fog> fog=new osg::Fog();           //雾对象
13     fog->setColor(osg::Vec4(1.0f,1.0f,1.0f,1.0f));       //颜色
14     fog->setDensity(0.01f);                               //密度
15     fog->setMode(osg::Fog::EXP);                          //设置模式
16     fog->setStart(1.0);                                    //设置近距离
17     fog->setEnd(500.0);                                    //设置远距离
18     return fog.get();
19 }
20 void main()
21 {
22     osg::ref_ptr<osgViewer::Viewer> viewer=new osgViewer::Viewer;
23     osg::ref_ptr<osg::Group> group=new osg::Group;
24     osg::ref_ptr<osg::Node> node=osgDB::readNodeFile("../data/ceep.ive");
25     group->addChild(node.get());
26     group->getOrCreateStateSet()->setAttributeAndModes(
27         createFog(),osg::StateAttribute::ON);            //设置雾效开启状态
28     viewer->setSceneData(group.get());
29     viewer->addEventHandler(new osgViewer::WindowSizeHandler());
30     viewer->run();
31 }
```



【代码解析】

第 10~19 行是函数 `createFog()` 的定义体, 其中第 12 行定义雾对象 `fog`, 第 13~17 行定义颜色、密度、模式、近距离及远距离等属性, 第 26、27 行设置雾效开启状态, 其中, 第 1 个参数设置雾效节点, 第 2 个参数将雾效打开。



实例 300 响应回调事件

【实例描述】

回调类主要应用于物体状态的实时更新, 比如位置的改变、角度的改变等。回调的基类是 `NodeCallback`, 由此派生的子类可以应用于不同事物的回调。本实例利用回调子类实现模型绕 Z 轴以 $\pi/100$ 的增量旋转。运行效果如图 16-12 所示。

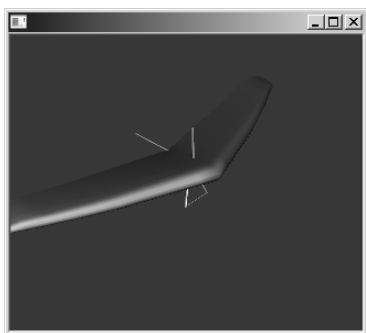


图 16-12 响应回调事件

【实现过程】

定义类 `RotateCallBack`, 用以实现模型绕 Z 轴角度的改变。在 `main()` 函数中, 利用 `osg::PositionAttitudeTransform` 变量调用该回调类, 具体代码如下:

```
01 #include "../Common/Common.h"
02 #include <osg/Quat>
03 #include <osg/PositionAttitudeTransform>
04 #include <osgViewer/ViewerEventHandlers>
05 #include <osgDB/ReadFile>
06 #include <osgViewer/Viewer>
07
08 class RotateCallBack:public osg::NodeCallback           //回调类
09 {
10 public:
11     RotateCallBack():angleZ(0.0) {}                      //构造函数
12     virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
13     {
14         osg::PositionAttitudeTransform* pat
15         = dynamic_cast<osg::PositionAttitudeTransform*>(node);
16         if(pat)
17         {
18             pat->setAttitude(osg::Quat(angleZ,osg::Z_AXIS)); //设置当前角度
19             angleZ +=osg::PI/100;                             //角度增量
20         }
21         traverse(node, nv);
```



```
22     }
23 private:
24     double angleZ;           //绕 z 轴角度
25 };
26 void main()
27 {
28     osg::ref_ptr<osgViewer::Viewer> viewer=new osgViewer::Viewer;
29     osg::ref_ptr<osg::Node> nd=new osg::Node;
30     nd=osgDB::readNodeFile("../data/glider.osg");
31     osg::ref_ptr<osg::PositionAttitudeTransform> ps=
32         new osg::PositionAttitudeTransform();           //定义变量
33     ps->addChild(nd);
34     ps->setUpdateCallback(new RotateCallBack());         //回调事件
35     viewer->setSceneData(ps.get());
36     viewer->addEventHandler(new osgViewer::WindowSizeHandler);
37     viewer->run();
38 }
```

【代码解析】

第 08~25 行是回调类 `RotateCallBack` 的声明和定义，其中第 11 行是构造函数对成员变量进行初始化。第 12~22 行是该回调类的核心函数，其中，第 18 行设置当前模型的角度，第 19 行对角度加增量 `osg::PI/100`。第 24 行是成员变量的定义。第 31、32 行定义位置节点变量 `ps`。第 34 行利用变量 `ps` 调用回调事件 `RotateCallBack()`。



注意：当设置有第一个回调事件后，还需要加另一个回调事件，此时应用 `addUpdateCallback()` 函数。