

目 录

青少年信息学奥林匹克竞赛情况简介.....	5
第一章 计算机基础知识.....	7
1.1 计算机的基本常识.....	7
1.1.1 计算机的产生与发展.....	7
1.1.2 计算机系统及工作原理.....	7
1.1.3 计算机中有关数及编码的知识.....	8
1.1.4 原码、反码与补码.....	10
1.1.5 逻辑运算.....	10
1.2 操作系统.....	11
1.2.1 DOS(Disk Operating System)的组成.....	11
1.2.2 DOS 的文件和目录.....	11
1.2.3 DOS 命令.....	12
1.2.4 Windows 简介.....	12
1.3 计算机网络常识.....	13
1.3.1 网络基础知识.....	13
1.3.2 Internet 简介.....	14
1.4 计算机信息安全基础知识.....	16
1.4.1 计算机的网络安全.....	16
1.4.2 计算机病毒.....	17
1.4.3 病毒的分类.....	17
第 2 章 C++编程简介.....	19
2.1 机器语言、汇编语言和高级语言.....	19
2.2 C 语言与 C++的历史.....	20
2.3 C++标准库.....	20
2.4 结构化编程.....	21
2.5 简单程序.....	22
2.6 简单程序：两个整数相加.....	25
2.7 算术运算.....	27
2.8 判断：相等与关系运算符.....	29
2.9 新型头文件与名字空间.....	31
第 3 章 C++输入/输出流.....	33
3.1 简介.....	33
3.2 流.....	33
3.2.1 iostream 类库的头文件.....	34
3.2.2 输入/输出流类和对象.....	34
3.3 输出流.....	35

3.3.1	流插入运算符.....	35
3.3.2	连续使用流插入/流读取运算符.....	37
3.3.3	输出 <code>char*</code> 类型的变量.....	37
3.3.4	用成员函数 <code>put</code> 输出字符和 <code>put</code> 函数的连续调.....	38
3.4	输入流.....	39
3.4.1	流读取运算符.....	39
3.4.2	成员函数 <code>get</code> 和 <code>getline</code>	41
3.5	成员函数 <code>read</code> 、 <code>gcount</code> 和 <code>write</code> 的无格式输入/输出.....	44
3.6	流操纵算子.....	45
3.6.1	整数流的基数：流操纵算子 <code>dec</code> 、 <code>oct</code> 、 <code>hex</code> 和 <code>setbase</code>	45
3.6.2	设置浮点数精度(<code>precision</code> 、 <code>setprecision</code>).....	46
3.6.3	设置域宽(<code>setw</code> 、 <code>width</code>).....	47
3.6.4	用户自定义的流操纵算子.....	48
3.7	流格式状态.....	49
3.7.1	格式状态标志.....	50
3.7.2	尾数零和十进制小数点(<code>ios::showpoint</code>).....	50
3.7.3	对齐(<code>ios::left</code> 、 <code>ios::right</code> 、 <code>ios::internal</code>).....	51
3.7.4	设置填充字符(<code>fill</code> 、 <code>setfill</code>).....	53
3.7.5	整数流的基数：(<code>ios::dec</code> 、 <code>ios::oct</code> 、 <code>ios::hex</code> 、 <code>ios::showbase</code>).....	54
3.7.6	浮点数和科学记数法(<code>ios::scientific</code> 、 <code>ios::fixed</code>).....	55
3.7.7	大/小写控制(<code>ios::uppercase</code>).....	56
3.7.8	设置及清除格式标志(<code>flags</code> 、 <code>setiosflags</code> 、 <code>resetiosflags</code>).....	57
3.8	流错误状态.....	58
第 4 章	文件处理.....	61
4.1	简介.....	61
4.2	文件和流.....	61
4.3	建立并写入文件.....	61
4.4	读取文件中的数据.....	65
4.5	更新访问文件.....	67
第 5 章	C++ 的字符串流.....	68
5.1	流的继承关系.....	68
5.2	字符串流的输入操作.....	68
5.3	字符串流的输出操作.....	69
5.4	字符串流在数据类型转换中的应用.....	70
5.5	输入/输出的状态标志.....	71
第 6 章	控制结构.....	74
6.1	简介.....	74
6.2	算法.....	74
6.3	控制结构.....	74
6.4	<code>if</code> 选择结构.....	75
6.5	<code>if/else</code> 选择结构.....	76

6.6	while 重复结构.....	78
6.7	构造算法：实例研究 1(计数器控制重复).....	78
6.8	构造算法与自上而下逐步完善：实例研究 2(标记控制重复).....	80
6.9	构造算法与自上而下逐步完善：实例研究 3(嵌套控制结构).....	85
6.10	赋值运算符.....	88
6.11	自增和自减运算符.....	88
6.12	计数器控制循环的要点.....	91
6.13	for 重复结构.....	92
6.14	for 结构使用举例.....	94
6.15	switch 多项选择结构.....	97
6.16	do/while 重复结构.....	101
6.17	break 和 continue 语句.....	102
6.18	逻辑运算符.....	104
6.19	混淆相等(==)与赋值(=)运算符.....	105
6.20	结构化编程小结.....	106
第 7 章 函数.....		108
7.1	简介.....	108
7.2	数学函数库.....	108
7.3	函数.....	109
7.4	函数定义.....	109
7.5	头文件.....	112
7.6	作用域规则.....	113
7.7	递归.....	116
7.8	使用递归举例，Fibonacci 数列.....	118
7.9	递归与迭代.....	120
7.10	带空参数表的函数.....	121
7.11	内联函数.....	122
7.12	函数重载.....	123
第 8 章 数组.....		125
8.1	简介.....	125
8.2	数组.....	125
8.3	声明数组.....	126
8.4	使用数组的举例.....	126
8.5	将数组传递给函数.....	137
8.6	排序数组.....	141
8.7	查找数组：线性查找与折半查找.....	142
8.8	多维数组.....	147
第 9 章 指针与字符串.....		153
9.1	简介.....	153
9.2	指针变量的声明与初始化.....	153
9.3	指针运算符.....	154

9.4	按引用调用函数.....	156
9.5	指针与常量限定符.....	158
9.6	按引用调用的冒泡排序.....	163
9.7	指针表达式与指针算法.....	167
9.8	指针与数组的关系.....	169
9.9	指针数组.....	173
9.10	函数指针.....	173
9.11	字符与字符串处理简介.....	177
9.11.1	字符与字符串基础.....	177
9.11.2	字符串处理库的字符串操作函数.....	179
第 10 章	信息学奥赛中的常用算法.....	185
10.1	算法简介.....	185
10.2	枚举算法.....	187
10.3	回溯算法.....	191
10.4	递归算法.....	193
10.5	递推算法.....	196
10.6	分治算法.....	200
10.7	贪心算法.....	202
10.8	搜索算法一（深度优先）.....	205
10.9	搜索算法二（广度优先）.....	209
10.10	动态规划法.....	212
10.11	高精度计算.....	215
附 录	228
ASCII 表.....		228

青少年信息学奥林匹克竞赛情况简介

信息学奥林匹克竞赛是一项旨在推动计算机普及的学科竞赛活动，重在培养学生能力，使得有潜质有才华的学生在竞赛活动中锻炼和发展。近年来，信息学竞赛活动组织逐步趋于规范和完善，基本上形成了“地级市——省(直辖市)——全国——国际”四级相互接轨的竞赛网络。现将有关赛事情况简介如下：

全国青少年信息学（计算机）奥林匹克分区联赛：

在举办 1995 年 NOI 活动之前，为了扩大普及的面，并考虑到多数省、直辖市、自治区已经开展了多年省级竞赛，举办了首届全国青少年信息学（计算机）奥林匹克分区联赛。考虑到不同年级学生的知识层次，也为了鼓励更多的学生积极参与，竞赛设提高组、普及组，并分初、复赛进行，这样可以形成一个梯队，确保每年的竞赛活动有比较广泛扎实的基础。

从 1995 年起，至 2001 年共举办了七届全国青少年信息学奥林匹克分区联赛，每年举办一次（下半年十月左右），有选手个人奖项（省、国家级）、选手等级证书、优秀参赛学校奖项。

安徽省青少年信息学（计算机）奥林匹克复决赛（简称 AHOI）：

省级信息学奥赛是一个水平较高的、有较大影响力的学科竞赛。由各市组织代表队参赛，参赛名额实行动态分配制度，每年举办一次（上半年五月左右）。从 1984 年起安徽省奥林匹克竞赛活动得到了蓬勃发展。奖项有个人一、二、三等奖，女选手第一、二、三名，奖励学校团体总分 1-8 名、市团体总分 1-8 名。

全国青少年信息学（计算机）奥林匹克竞赛（简称 NOI）：

由中国计算机学会主办的、并与国际信息学奥林匹克接轨的一项全国性青少年学科竞赛活动。1984 年举办首届全国计算机竞赛。由各省市区组织参赛，每年举办一次。奖项有个人一、二、三等奖，女选手第一、二、三名，各省队团体总分名次排队。

国际青少年信息学（计算机）奥林匹克竞赛（简称 IOI）：

每年举办一次，由各参赛国家组队参赛。

全国青少年信息学（计算机）奥林匹克分区联赛竞赛大纲

一、初赛内容与要求：（#表示普及组不涉及，以下同）

计 基 算 本 机 常 的 识	* 诞生与发展 * 计算机系统的基本组成 * 计算机的工作原理# * 计算机信息安全基础知识	*特点 *在现代社会中的应用 *计算机中的数的表示 *计算机网络
计 基 算 本 机 操 的 作	* MS DOS 与 Windows 的使用基础 * 常用输入/输出设备的种类、功能、使用 * 汉字输入/输出方法 * 常用计算机屏显信息	
程 序 设 计 基 本 知	程序的表示	* 自然语言的描述 * PASCAL 或 BASIC 语言
	数据结构类型	* 简单数据的类型 * 构造类型：数组、字符串 * 了解基本数据结构（线性表、队列与栈）
	程序设计	* 结构化程序的基本概念 * 阅读理解程序的基本能力

识		* 具有完成下列过程的能力： 现实世界（指知识范畴的问题） —>信息世界（表达解法） —>计算机世界（将解法用计算机能实现的数据结构和算法描述出来）
	基本算法处理	* 简单搜索 * 字符串处理 * 排序 * 查找 * 统计 * 分类 * 合并 * 简单的回溯算法 * 简单的递归算法

二、复赛内容与要求：

计算机 软 件	*操作系统的使用知识 *编程语言的使用
数 据 结 构	*结构类型中的记录类型 *指针类型 *文件（提高组必须会使用文本文件输入） *链表 *树 *图#
程 序 设 计	*程序设计能力 *设计测试数据的能力 *运行时间和占用空间的估算能力#
算 法 处 理	*排列组合的应用 *进一步加深回溯算法、递归算法 *分治法 *搜索算法：宽度、深度优先算法 *表达式处理：计算、展开、化简等# *动态规划#

在初赛的内容上增加以下内容(2008 年修改稿)：

三、初赛试题类型：注：试题语言两者选一

（程序设计语言： FREE PASCAL、C、C++）

*判断 *填空 *完善程序 *读程序写运行结果 *问答

四、推荐读物：

*分区联赛辅导丛书 *学生计算机世界报及少年电世界杂志

第一章 计算机基础知识

1.1 计算机的基本常识

1.1.1 计算机的产生与发展

计算机的产生是 20 世纪最重要的科学技术大事件之一。世界上的第一台计算机（ENIAC）于 1946 年诞生在美国宾夕法尼亚大学，到目前为止，计算机的发展大致经历了四代：

① 第一代电子管计算机，始于 1946 年，结构上以 CPU 为中心，使用计算机语言，速度慢，存储量小，主要用于数值计算；

② 第二代晶体管计算机，始于 1958 年，结构上以存储器为中心，使用高级语言，应用范围扩大到数据处理和工业控制；

③ 第三代中小规模集成电路计算机，始于 1964 年，结构上仍以存储器为中心，增加了多种外部设备，软件得到了一定的发展，文字图象处理功能加强；

④ 第四代大规模和超大规模集成电路计算机，始于 1971 年，应用更广泛，很多核心部件可集成在一个或多个芯片上，从而出现了微型计算机。

我国从 1956 年开始电子计算机的科研和教学工作，1983 年研制成功 1 亿/秒运算速度的“银河”巨型计算机，1992 年 11 月研制成功 10 亿/秒运算速度的“银河 II”巨型计算机，1997 年研制了每秒 130 亿运算速度的“银河 III”巨型计算机。

目前计算机的发展向微型化和巨型化、多媒体化和网络化方向发展。计算机的通信产业已经成为新型的高科技产业。计算机网络的出现，改变了人们的工作方式、学习方式、思维方式和生活方式。

1.1.2 计算机系统及工作原理

1. 计算机的系统组成

计算机系统由软件和硬件两部分组成。硬件即构成计算机的电子元器件；软件即程序和有关文档资料。

(1) 计算机的主要硬件

输入设备：键盘、鼠标、扫描仪等。

输出设备：显示器、打印机、绘图仪等。

中央处理器（CPU）：包括控制器和运算器运算器，可以进行算术运算和逻辑运算；控制器是计算机的指挥系统，它的操作过程是取指令——分析指令——执行指令。

存储器：具有记忆功能的物理器件，用于存储信息。存储器分为内存和外存

① 内存是半导体存储器(主存)：

它分为只读存储器(ROM)和随机存储器(RAM)和高速缓冲存储器（Cache）；

ROM:只能读，不能用普通方法写入，通常由厂家生产时写入，写入后数据不容易丢失，也可以用特殊方法（如紫外线擦除（EPROM)或电擦除(EEPROM)存储器）；

RAM:可读可写，断电后内容全部丢失；

Cache:因为 CPU 读写 RAM 的时间需要等待，为了减少等待时间，在 RAM 和 CPU 间需要设置

高速缓存 Cache,断电后其内容丢失。

②外存：磁性存储器——软盘和硬盘；光电存储器——光盘，它们可以作为永久存器；

③存储器的两个重要技术指标：存取速度和存储容量。内存的存取速度最快(与 CPU 速度相匹配)，软盘存取速度最慢。存储容量是指存储的信息量，它用字节(Byte)作为基本单位，1 字节用 8 位二进制数表示，1KB=1024B，1MB=1024KB，1GB=1024MB

(2)计算机的软件

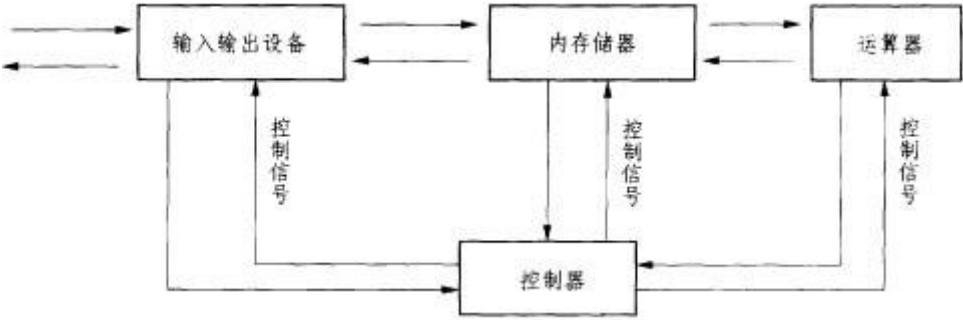
计算机的软件主要分为系统软件和应用软件两类：

①系统软件：为了使用和管理计算机的软件，主要有操作系统软件如，WINDOWS 95 / 98 / 2000 / NT4. 0、DOS 6. 0、UNIX 等；WINDOWS 95 / 98 / 2000 / NT4. 0 是多任务可视化图形 界面，而 DOS 是字符命令形式的单任务的操作系统。

②应用软件：为了某个应用目的而编写的软件，主要有辅助教学软件(CAI)、辅助设计软件(CAD)、文 字处理软件、工具软件以及其他的应用软件。

2.计算机的工作原理

到目前为止,电子计算机的工作原理均采用冯.若依曼的存储程序方式,即把程序存储在计算机内,由计算机自动存取指令（计算机可执行的命令=操作码+操作数）并执行它。工作原理图如下：



1.1.3 计算机中有关数及编码的知识

1.计算机是智能化的电器设备

计算机就其本身来说是一个电器设备，为了能够快速存储、处理、传递信息，其内部采用了大量的电子元件，在这些电子元件中，电路的通和断、电压高低，这两种状态最容易实现，也最稳定、也最容易实现对电路本身的控制。我们将计算机所能表示这样的状态，用 0，1 来表示、即用二进制数表示计算机内部的所有运算和操作。

2.二进制数的运算法则

二进制数运算非常简单，计算机很容易实现，其主要法则是：

$0+0=0$ $0+1=1$ $1+0=1$ $1+1=0$ $0*0=0$ $0*1=0$ $1*0=0$ $1*1=1$

由于运算简单，电器元件容易实现，所以计算机内部都用二进制编码进行数据的传送和计算。

3.十进制与二进制、八进制、十六进制数之间的相互转换

(1)数的进制与基数

计数的进制不同，则它们的基数也不相同，如表 1-1 所示。

进制	基数	特点
二进制	0,1	逢二进一
八进制	0,1,2,3,4,5,6,7	逢八进一

十六进制	0,1,2,...,9,A,B,C,D,E,F	逢十六进一
------	-------------------------	-------

(2) 数的权

不同进制的数，基数不同，每位上代表的值的大小（权）也不相同。

如：(219)₁₀=2*10²+1*10¹+9*10⁰

(11010)₂=1*2⁴+1*2³+0*2²+1*2¹+1*2⁰

(273)₈=2*8²+7*8¹+3*8⁰

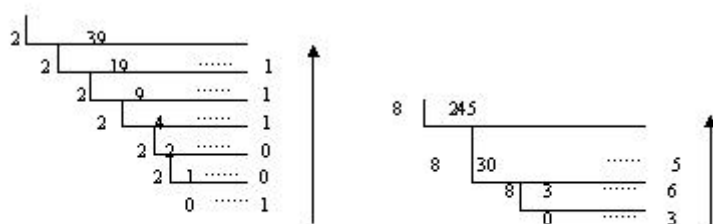
(27AF)₁₆=2*16³+7*16²+10*16¹+15*16⁰

(3) 十进制数转换任意进制

1) 将十进制整数除以所定的进制数,取余逆序。

(39)₁₀=(100111)₂

(245)₁₀=(365)₈



2)将十进制小数的小数部分乘以进制数取整,作为转换后的小数部分,直到为零或精确到小数点后几位。

如：(0.35)₁₀=(0.01011)₂ (0.125)₁₀=(0.001)₂

(4) 任意进制的数转换十进制

按权值展开:

如：(219)₁₀=2*10²+1*10¹+9*10⁰

(11010)₂=1*2⁴+1*2³+0*2²+1*2¹+1*2⁰=26

(273)₈=2*8²+7*8¹+3*8⁰=187

(7AF)₁₆=7*16²+10*16¹+15*16⁰=1867

4. 定点数与浮点数

定点数是指数据中的小数点位置固定不变。由于它受到字长范围的限制，所能表示的数的范围有限，计算结果容易溢出。

浮点数的形式可写成：N=M*2^E(其中 M 代表尾数,E 代表阶码) 其形式如下：

阶码	尾数（包括符号位）
----	-----------

5. ASCII 编码

由于计算机是电器设备,计算机内部用二进制数,这样对于从外部输入给计算机的所有信息必须用二进制数表示,并且对于各种命令、字符等都需要转换二进制数，这样就牵涉到信息符号转换成二进制数所采用的编码的问题，国际上统一用美国标准信息编码（ASCII）它可用 7 位二进制数表示，存储时用一个字节，它的最高位为 0。因此基本的 ASCII 字符集有 128 个如：

0-9: 48-57:00110000-...

A-Z:65-90 :01000001-...

a-z:97-122:01100000-...

6. 汉字编码与汉字输入法

(1) 机内码

ASCII 码不能表示汉字，因此要有汉字信息交换码，我国国家标准是 gb2312,它也被称作国际码。

它由两个字节组成，两个字节的最高位都为 1。gb2312 共收纳 6763 个汉字，其中，一级汉字（常用字）3755 个按汉字拼音字母顺序排列，二级汉字 3008 个按部首笔画次序排列。

(2) 汉字输入码（外码）

目前，汉字输入法主要有键盘输入、文字识别和语音识别。键盘输入法是当前汉字输入的主要方法。它大体可以分为：

流水码：如区位码、电报码、通信密码，优点重码律少，缺点难于记忆；

音码：以汉语拼音为基准输入汉字，优点是容易掌握，但重码律高；

形码：根据汉字的字型进行编码，优点重码少，但不容易掌握；

音形码：将音码和形码结合起来，能减少重码律同时提高汉字输入速度。

(3) 汉字字模

供计算机输出汉字（显示和打印）用的二进制信息叫汉字字形信息也称字模。通用汉字字模点阵规格有 16*16，24*24，32*32，48*48，64*64，每个点在存储器中用一个二进制位（bit）存储，如一个 16*16 点阵汉字需要 32 个字节的存储空间。

1.1.4 原码、反码与补码

在计算机中，数据是以补码的形式存储的：

在 n 位的机器数中，最高位为符号位，该位为零表示为正，为 1 表示为负；

其余 $n-1$ 位为数值位，各位的值可为 0 或 1。

当真值为正时：原码、反码、补码数值位完全相同；

当真值为负时：

原码的数值位保持原样，

反码的数值位是原码数值位的各位取反，

补码则是反码的最低位加一。

注意符号位不变。

如：若机器数是 16 位：

十进制数 17 的原码、反码与补码均为：0000000000010001

十进制数 -17 的原码、反码与补码分别为：1000000000010001、111111111101110、111111111101111

1.1.5 逻辑运算

1. 逻辑运算

逻辑与：同真则真

逻辑或：有真就真

逻辑非：你真我假

逻辑异或：不同则真

2. 按位运算

按位与 \cap ：同 1 则 1 如 $10010101 \cap 10110111 = 10010101$

按位或 \cup ：有 1 则 1 如 $10010101 \cup 10110111 = 10110111$

3. 逻辑化简

化简定律：

(1) 交换律： $A + B = B + A$ ， $A \cdot B = B \cdot A$

(2) 结合律： $(A + B) + C = A + (B + C)$ ， $(A \cdot B) \cdot C = A \cdot (B \cdot C)$

- (3) 幂等律: $A \cdot A = A$, $A + A = A$
 (4) 吸收律: $A \cdot (A + B) = A$, $A + (A \cdot B) = A$
 (5) 分配律: $A \cdot (B + C) = A \cdot B + A \cdot C$, $A + (B \cdot C) = (A + B) \cdot (A + C)$
 (6) 互补律: $A + \bar{A} = 1$, $A \cdot \bar{A} = 0$
 (7) 非深入: $A + B = A + A \cdot B$, $A \cdot B = A \cdot (A + B)$
 (8) 0-1 律: $A + 0 = A$, $A + 1 = 1$, $A \cdot 1 = A$, $A \cdot 0 = 0$

例:化简函数 $Q = AD + AD + AB + ACEF$ 。这个函数有 5 个自变量, 化简过程如下:

$$\begin{aligned} Q &= AD + AD + AB + ACEF \\ &= A + AB + ACEF \\ &= A + ACEF \\ &= A \end{aligned}$$

练习:求证: $(A+B)(A+C)=AB+AC$

1.2 操作系统

1.2.1 DOS(Disk Operating System)的组成

MS-DOS 采用模块结构, 它由五部分组成: ROM 中的 BIOS 模块、IO. SYS 模块、MSDOS. SYS 模块、COMMAND. COM 模块和引导程序。

- (1)BIOS 模块: 在 PC 机主板上有一个 ROM 芯片, 该芯片中存有系统自测试程序, CMOS 设置程序和基本输入输出程序(BIOS)。BIOS 是一组程序和参数表, 其中程序部份是可以通过中断方式调用的一组驱动程序, 参数给出外设的地址和参数。BIOS 是计算机硬件和操作系统之间的接口通过它操作系统管理计算机硬件资源。
- (2)IO. SYS 模块: IO. SYS 是 MS-DOS 和 ROMBIOS 之间的接口程序。它和 ROMBIOS 一起完成系统设备的管理。
- (3)MSDOS.SYS 模块: MSDOS. SYS 用于实现文件管理, 包括文件管理、目录管理、内存管理等功能。它以功能调用的形式实现用户和 MS-DOS 之间的程序级接口。
- (4)COMMAND. COM 模块: COMMAND. COM 的主要功能是负责接收、识别、解释和执行用户从键盘输入的 MS-DOS 命令。
- (5)引导程序: 引导程序又叫“引导记录”, 其作用是检查当前盘上是否有两个系统文件, 若有系统文件则把 DOS 系统从磁盘装入内存。一张系统盘上应该包含有: 引导记录、IO. SYS、MSDOS. SYS 和 COMMAND. COM 等模块。

1.2.2 DOS 的文件和目录

- 1)文件概念: 文件是指记录在存储介质(如磁盘、光盘)上的一组相关信息的集合。
- 2)文件标识:驱动器号+路径+文件名(1 到 8 各字符)+扩展名(1 到 3 个字符代表文件的类型)
- 3)通配符:*代表从该位置起的一个或多个合法字符; ?代表所在位置的任一个合法字符。
- 4) 树形目录: DOS 采用树形目录结构。由一个根目录和若干层子目录组成。这种目录结构一是能够解决文件重名问题, 即不同的目录可以包含相同的文件名或目录名; 二是能够解决文件多而根目录容量有限带来的问题。在查找某个子目录下的一个文件时, 要使用目录路径。指定路径有两种方法: 绝对路径和相对路径。绝对路径是从根目录开始到文件所在目录的路径。

例如要查找 UC DOS 子目录下的二级子目录 DATA 下的 README. TXT 文件, 绝对路径为: \ UC DOS \ DATA。路径中第一个 “\” 符号代表根目录。相对路径是从当前目录开始到文件所在目录的路径。当前目录指在不特意指定路径情况下 DOS 命令所处理的目录。例如系统提示符为: “C: \ UC DOS \ DATA>”, 则 DATA 是当前目录。

1.2.3 DOS 命令

1. 内部命令

1)内部命令: 当启动 DOS 系统时, 计算机引导程序将系统以及常用的命令处理模块驻留在计算机的内存中, 我们称之为内部命令。

2)常用的内部命令:

(1)目录命令:

DIR(显示文件目录)

MD、CD、RD(子目录的建立、进入、删除命令)

(2)文件操作命令:

COPY(复制命令)、DEL(删除命令)、REN(更改文件名)

TYPE(显示文本文件内容)

(3)其他内部命令

DATA、TIME、VER、CLS 等 •

2. 外部命令

1)外部命令: 存储在外存储器上的 DOS 可执行的文件, 这些文件程序所占的存储容量比较大, 当用户使用外部命令时, 计算机从外存调入内存, 当执行完外部命令, 就自动从内存中退出。

2)常用的外部命令

(1)磁盘格式化命令: FORMAT 盘符 [/S]/I/V]

其作用, 能够清除原盘中所有信息, 并将磁盘规范成计算机所能接受的格式, 以便有效存储信息。

(2)软盘复制命令: DISKCOPY [盘符 1:][盘符 2:]

其作用, 能够进行软盘之间的全盘复制(以磁道方式), 不仅可以复制系统文件而且可以复制隐含文件。

1.2.4 Windows 简介

Windows 是一个多任务图形用户界面, 该环境可以在基于 MS-DOS 的计算机上运行, 在多任务图形用户环境下, Windows 提供了一个基于下拉菜单、屏幕窗口和鼠标的界面, 在该环境下运行的应用程序必须进行专门的设计才能发挥这些特征的优点。

1. Windows 的特点

Windows 能够充分发挥计算机的作用, 其图形接口能够组织用户程序和文件、同时运行几个用户程序、在文档之间移动和复制信息、在平台上进行应用程序的切换等。为了提高效率, Windows 还提供了一些辅助程序, 如字处理器、画笔及其他标准应用程序等。

Windows 具有以下主要特点。

(1)图形化的用户界面

Windows 提供了一种不同于 DOS 系统下命令行的工作方式, 它通过对窗口、图标、选单、对话框、命令按钮、滚动框等图形符号与画面的操作来实现对计算机的各种操作。

(2)标准化的操作界面

在 Windows 中，所有的操作都是通过窗口中的图形界面进行的。

(3)多任务机制和执行性能

在 Windows 中，平稳的多任务机制可以同时运行多道程序以及执行多项任务，各程序与各任务之间不仅转换容易，而且还可以方便地交换数据。

(4)充分利用内存

Windows 利用虚拟内存技术，允许应用程序超过 640 阳常规内存的运行空间，从而最大限度地利用了计算机系统的所有内存资源，从而使内存较小的微机也能运行大型的应用程序。

(5)强大的联网功能

在 Windows 中，可以简单直观地实现网络的安装、配置、浏览，从而可以更加方便地实现网络管理和资源共享。

(6)丰富的多媒体功能

Windows 提供大量辅助程序，用以实现文字、图形、图像、声音、视频等多媒体功能，同时还支持其他厂商基于 Windows 标准开发的各种相应软件。

(7)TrueType 技术

TrueType(真实字体)属于内建式比例字体，可以任意平滑放大与缩小。这种字体能使屏幕上显示的效果与实际打印机输出的信息完全一致，这就是所谓的“所见即所得”。

[例]在 Windows 95 中，“任务栏”的作用是_____。

- A)显示系统的所有功能 B)只显示当前活动窗口名
C)只显示正在后台工作的窗口名 D)实现窗口之间的切换

解答：在任务栏中，显示了所有打开的程序的图标。

本题正确答案为 D。

1.3 计算机网络常识

1.3.1 网络基础知识

1.网络的概念

计算机网络是将地理位置不同的计算机,用通信链路连接起来,共同遵守一定的协议,以实现计算机软硬件资源共享为目标的通信系统。

2.网络的组成

计算机网络由网络硬件和网络软件组成。

网络软件包括网络操作系统、通信软件、通信协议（计算机之间实现数据通信共同遵守的相关规定）。

网络硬件包括网络的拓扑结构、网络服务器、网络工作站、传输介质和设备。

3.网络的分类

(1)按通信距离分:

局域网（LAN）:局限于某个范围(10 公里左右)的网络连接情（校园网）。

广域网（WAN）:跨地区的局域网，Internet 是覆盖全球的广域网。

(2)按网络的使用目的分:

共享资源网：使用者可分享网络的各种资源（如 Internet）。

数据处理网：用于数据处理（企业经营管理用的网络）。

数据传输网：用于数据的收集、交换和传输（情报检索网络）。

(3) 按网络的拓扑结构分:

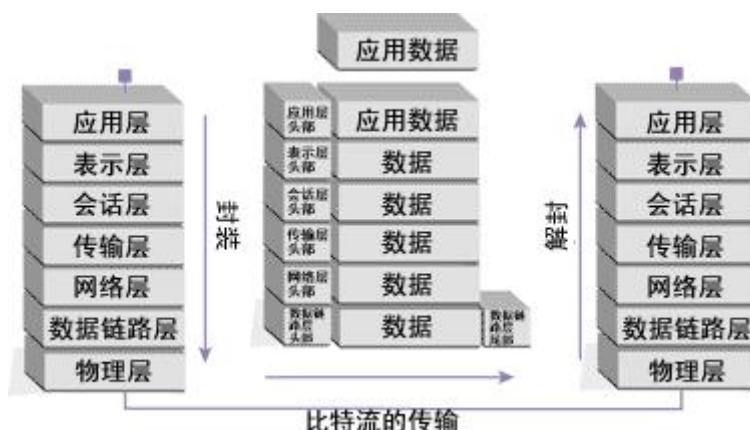
星形网: 以一台计算机为中心, 以放射状连接若干台计算机。

环形网: 传输线路构成一个封闭的环, 入网的计算机连到这个环形线路上。

总线网: 用一条通信线路作主干, 入网的计算机通过相应接口连到线路上。

4. 开放系统互联 模型 (OSI 模型)

OSI 模型分 7 层:



各层功能如下:

1. 物理层

物理层与移动二进制数和维护物理连接有关。

2. 数据链路层

数据链路层通过帧在一个给定的物理链路传输分组 (报文), 保持帧的有序以及发现检测到的各种错误, 包括传输错误, 但是数据链路层只了解在链路另一端的对等实体。数据链路层的地址是为了将网络中一点的数据帧送到另一点。

3. 网络层

网络层知道每个数据链路的对等进程, 并负责在链路间移动分组, 把它送到目的地。网络层地址是为了把单一分组从网络的一端送到目的地。

4. 传输层

传输层注意的是整个网络, 该层是第一个端到端层。其对等实体位于分组的最终目的地。传输层依靠网络层经过中间节点移动分组。传输层地址是为了把网络一端进程的完整信息送到最终目的地的对等进程。

5-7. 会话层、表示层和应用层提供了如下功能:

处理计算机间数据表示的差别。

确保数据在网络传输中不被窃取和泄露, 并且确保网络不允许未经授权就访问数据。

最高效地使用网络资源通过应用程序及活动同步来管理对话和活动。

在网络节点间共享数据。

1.3.2 Internet 简介

Internet 英文直译为“互联网”, 中文名为“因特网”。是世界上众多计算机网络的集合起源于 20 世纪 80 年代。

1. Internet 的 IP 地址、IP 地址类型和主机域名

(1) 在 Internet 网上采用统一的网络协议 TCP/IP, 与 Internet 相连的计算机必须具有唯一的主机地址, 称 IP 地址。IP 地址采用分段地址方式, 使用数字表示; 如: 207.46.130.14, 其中由三个

点隔开的四个数是十进制,其大小是 0-255,每个数对应一个 8 位二进制数,所以 IP 地址用 32 位二进制位存放站 4 个字节。

(2) IP 地址类型:最初设计互联网络时,为了便于寻址以及层次化构造网络,每个 IP 地址包括两个标识码(ID),即网络 ID 和主机 ID。同一个物理网络上的所有主机都使用同一个网络 ID,网络上的一个主机(包括网络上工作站,服务器和路由器等)有一个主机 ID 与其对应。IP 地址根据网络 ID 的不同分为 5 种类型, A 类地址、B 类地址、C 类地址、D 类地址和 E 类地址。

A 类 IP 地址

一个 A 类 IP 地址由 1 字节的网络地址和 3 字节主机地址组成,网络地址的最高位必须是“0”,地址范围从 1.0.0.0 到 126.0.0.0。可用的 A 类网络有 126 个,每个网络能容纳 1 亿多个主机。

B 类 IP 地址

一个 B 类 IP 地址由 2 个字节的网络地址和 2 个字节的主机地址组成,网络地址的最高位必须是“10”,地址范围从 128.0.0.0 到 191.255.255.255。可用的 B 类网络有 16382 个,每个网络能容纳 6 万多个主机。

C 类 IP 地址

一个 C 类 IP 地址由 3 字节的网络地址和 1 字节的主机地址组成,网络地址的最高位必须是“110”。范围从 192.0.0.0 到 223.255.255.255。C 类网络可达 209 万余个,每个网络能容纳 254 个主机。

D 类地址用于多点广播 (Multicast)。

D 类 IP 地址第一个字节以“1110”开始,它是一个专门保留的地址。它并不指向特定的网络,目前这一类地址被用在多点广播 (Multicast) 中。多点广播地址用来一次寻址一组计算机,它标识共享同一协议的一组计算机。

E 类 IP 地址

以“11110”开始,为将来使用保留。

全零 (“0. 0. 0. 0”) 地址对应于当前主机。全 “1” 的 IP 地址 (“255. 255. 255. 255”) 是当前子网的广播地址。

在 IP 地址 3 种主要类型里,各保留了 3 个区域作为私有地址,其地址范围如下:

A 类地址: 10.0.0.0~10.255.255.255

B 类地址: 172.16.0.0~172.31.255.255

C 类地址: 192.168.0.0~192.168.255.255

(3) 为了使用方便,在访问 Internet 上的主机时,通常使用主机域名而不是 IP 地址,但主机域名和 IP 地址一一对应,它由圆点分隔的一序列单词组成如 “Public.bta.net.cn”。

IP 地址如同电脑的身份证号码,而域名相当电脑的姓名。

2.Internet 的功能

(1) 信息浏览(WWW)

WWW(World Wide Web),中文名为“万维网”,是基于超文本的、方便用户信息浏览和信息搜索的信息服务系统。用户在浏览器中输入网址即可得到需要的信息。人们常用的浏览器有网景公司的 Netscape 浏览器和 Microsoft 公司的 Internet Explorer 浏览器。网址的输入是使用协议提供的服务+服务器地址 (IP 地址或主机域名) 如 http: //198.105.232.1;

ftp: //zsqs.com

(2) 文件传输(FTP)

FTP(File Transfer Protocol)是 Internet 的一种标准协议,这一协议使用户能在联网的计算机之间传送文件如上载(UPLOAD 把本地计算机上地文件复制到远程计算机上)和下载(DOWNLOAD 把远程计算机上的文件复制到本地计算机上)。

(3) 传送电子邮件 (E-mail)

电子邮件地址=用户名+@+主机域名；如：zhangming@yahoo.com

(4) 电子公告牌（BBS）

(5) 远程登录（telnet）

(6) 电子商务等

3.TCP/IP 参考模型

TCP/IP 协议的开发研制人员将 Internet 分为五个层次，以便于理解，它也称为互联网分层模型或互联网分层参考模型，如下表：

应用层（第五层）

传输层（第四层）

互联网层（第三层）

网络接口层（第二层）

物理层（第一层）

各层简要说明如下：

物理层：对应于网络的基本硬件，这也是 Internet 物理构成，即我们可以看得见的硬件设备，如 PC 机、互连网服务器、网络设备等，必须对这些硬件设备的电气特性作一个规范，使这些设备都能够互相连接并兼容使用。

网络接口层：它定义了将数据组成正确帧的规程和在网络中传输帧的规程，帧是指一串数据，它是数据在网络中传输的单位。

互联网层：本层定义了互联网中传输的“信息包”格式，以及从一个用户通过一个或多个路由器到最终目标的“信息包”转发机制。

传输层：为两个用户进程之间建立、管理和拆除可靠而又有效的端到端连接。

应用层：它定义了应用程序使用互联网的规程。

1.4 计算机信息安全基础知识

1.4.1 计算机的网络安全

1、不同环境和应用中的网络安全

运行系统安全，即保证信息处理和传输系统的安全。它侧重于保证系统正常运行，避免因为系统的崩溃和损坏而对系统存贮、处理和传输的信息造成破坏和损失，避免由于电磁泄漏，产生信息泄露，干扰他人，受他人干扰。

网络上系统信息的安全。包括用户口令鉴别，用户存取权限控制，数据存取权限、方式控制，安全审计，安全问题跟踪，计算机病毒防治，数据加密。

网络上信息传播安全，即信息传播后果的安全。包括信息过滤等。它侧重于防止和控制非法、有害的信息进行传播后的后果。避免公用网络上大量自由传输的信息失控。

网络上信息内容的安全。它侧重于保护信息的保密性、真实性和完整性。避免攻击者利用系统的安全漏洞进行窃听、冒充、诈骗等有益于合法用户的行为。本质上是保护用户的利益和隐私。

网络安全的特征

2、网络安全应具有以下四个方面的特征：

保密性：信息不泄露给非授权用户、实体或过程，或供其利用的特性。

完整性：数据未经授权不能进行改变的特性。即信息在存储或传输过程中保持不被修改、不被破坏和丢失的特性。

可用性：可被授权实体访问并按需求使用的特性。即当需要时能否存取所需的信息。例如网络环境下拒绝服务、破坏网络和有关系统的正常运行等都属于对可用性的攻击；

可控性：对信息的传播及内容具有控制能力。

3、主要的网络安全威胁

自然灾害、意外事故；

计算机犯罪；

人为行为，比如使用不当，安全意识差等；

“黑客”行为：由于黑客的入侵或侵扰，比如非法访问、拒绝服务计算机病毒、非法连接等；

内部泄密；

外部泄密；

信息丢失；

电子谍报，比如信息流量分析、信息窃取等；

信息战；

网络协议中的缺陷，例如 TCP/IP 协议的安全问题等等。

4、黑客常用的信息收集工具

信息收集是突破网络系统的第一步。黑客可以使用下面几种工具来收集所需信息：

SNMP 协议，用来查阅非安全路由器的路由表，从而了解目标机构网络拓扑的内部细节。

TraceRoute 程序，得出到达目标主机所经过的网络数和路由器数。

Whois 协议，它是一种信息服务，能够提供有关所有 DNS 域和负责各个域的系统管理员数据。（不过这些数据常常是过时的）。

DNS 服务器，可以访问主机的 IP 地址表和它们对应的主机名。

Finger 协议，能够提供特定主机上用户们的详细信息（注册名、电话号码、最后一次注册的时间等）。

Ping 实用程序，可以用来确定一个指定的主机的位置并确定其是否可达。把这个简单的工具用在扫描程序中，可以 Ping 网络上每个可能的主机地址，从而可以构造出实际驻留在网络上的主机清单。

1.4.2 计算机病毒

计算机病毒是一种程序,是人为设计的具有破坏性的程序。

计算机病毒具有破坏性、传播性、可激发性、潜伏性、隐蔽性等特点。

1.4.3 病毒的分类

(1)按病毒设计者的意图和破坏性大小，可将计算机病毒分为良性病毒和恶性病毒。

①良性病毒：这种病毒的目的不是为了破坏计算机系统，而只是为了编制者表现自己。此类病毒破坏性较小，只是造成系统运行速度降低，干扰用户正常工作。

②恶性病毒：这类病毒的目的是人为的破坏计算机系统的数据。具有明显破坏目标，其破坏和危害性都很大，可能删除文件或硬盘进行非法的格式化。

(2)计算机病毒按照寄生方式可以分为下列四类：

①源码病毒：在源程序被编译之前，就插入到用高级语言编写的源程序当中。编写这种病毒程序较困难。但是，一旦插入，其破坏性和危害性都很大。

②入侵病毒：是把病毒程序的一部分插入到主程序中。这种病毒程序也难编写，一旦入侵，

难以清除。

②操作系统病毒：是把病毒程序加入或替代部分操作系统进行工作的病毒。这种病毒攻击力强、常见、破坏性和危害性最大。

④外壳病毒：是把病毒程序置放在主程序周围，一般不修改源程序的一种病毒。它大多是感染 DOS 下的可执行程序。这种病毒占一半以上，易编制，也易于检测和消除。

在日常维护中应隔离计算机病毒的来源，经常要用杀毒软件检查计算机系统和存储器。

[例]设一张软盘已染上病毒，能清除病毒的措施是_____。

- A)删除该软盘上的所有文件
- B)格式化该软盘
- C)删除该软盘上的所有可执行文件
- D)删除该软盘上的所有批处理文件

解答：软盘染毒后，病毒隐藏在磁盘内部，并感染磁盘上的文件，而且可能通过磁盘的使用进而扩散到其他磁盘，造成更大的破坏。为了清除病毒，必须格式化软盘，从而彻底清除染毒文件和病毒本身。

本题正确答案为 B。

第 2 章 C++编程简介

2.1 机器语言、汇编语言和高级语言

程序员用各种编程语言编写指令,有些是计算机直接理解的,有些则需要中间翻译(tranlation)的步骤。如今使用的计算机语言有几百种,可以分为三大类:

1. 机器语言
2. 汇编语言
3. 高级语言

任何计算机只能直接理解本身的机器语言(machine language)。机器语言是特定计算机的自然语言,由计算机的硬件设计定义。机器语言通常由一系列数字组成(最终简化 0 和 1),让计算机一次一个地执行最基本的操作。机器语言非常繁琐,下面的机器语言程序将工龄工资和基础工资相加,并把结果保存在工资总额中:

```
+1300042774
+1400593419
+1200274027
```

随着计算机越来越普及,机器语言编程对大多数程序员显然太慢、太繁琐。程序员不用计算机直接理解的一系列数字,而是用类似英文缩写的助记将来表示计算机的基本操作,这些助记符构成了汇编语言(assembly language)。称为汇编器(assembler)的翻译程序以计算机的速度将汇编语言程序转换为机器语言。下列汇编程序也是工龄工资和基础工资相加,并将结果保存在总工资中,但要比相应的机器语言清晰得多:

```
LOAD    BASEPAY
ADD      OVERPAY
STORE    GROSSPAY
```

尽管这种代码对于人们一目了然,但计算机却无法理解,必须先翻译为相应的机器语言。

随着汇编语言的出现,计算机的使用迅速增加,然而即使是最简单的任务,也需要许多条指令才能完成。为了加速编程过程,人们开发了高级语言(high-level language),用一条语句完成大量任务。称为编译器(compiler)的翻译程序将高级语言程序变为相应的机器语言。高级语言使程序员能够编写更像英语的指令,可以包含常用的数学符号。将工龄工资和基础工资相加,并把结果保存在总工资中,可以用下列高级语言程序:

```
grossPay = basePay + overTimePay
```

显然,从程序员角度看,高级语言比机器语言和汇编语言都要强得多。C 和 C++是最强大最广泛使用的高级语言。

将高级语言程序编译为相应的机器语言的过程可能需要大量时间。解释器(interpreter)程序可以直接执行高级语言程序,而不必先将这些程序编译成相应的机器语言。尽管编译程序的执行速度比解释程序更快,但解释器在程序开发环境中更常用,因为增加新特性和纠正错误时经常需要重新编译程序。一旦程序开发完成,编译版本的运行最有效。

2.2 C 语言与 C++的历史

C++是从 C 语言演变而来的,而 C 语言又是从两个编程语言 BCPL 和 B 演变而来的、BCPL 是 Martin Richards 于 1967 年开发的,用于编写操作系统软件和编译器。Ken Thompson 在他的 B 语言(1970 年在贝尔实验室)。DCPL 和 B 都是“无类型”语言,每个数据项在内存中占一个“字”(word)长、如果要数据项作为整数或实数处理,编程的工作量会很大。

C 语言是从 D 语言演变而成的,由贝尔实验室的 Dennis Ritchie 开发,最初于 1972 年在 DEC PDP—11 计算机上实现。C 语言使用了 BCPL 和 B 的许多重要概念,同时增加了数据类型和其他特性。C 语言最初作为 UNIX 操作系统的开发语言而闻名于世。如今,大多数操作系统都是用 C/C++写成的。二十多年来,C 语言已经遍布在大多数计算机上。C 语言是硬件无关的,只要仔细设计。就可以编写能移植到大多数计算机上的 C 语言程序。

到 20 世纪 70 年代末期,C 语言演变成现在所谓的“传统 C”、“经典 C 或“Kernighan / Ritchie C”。1978 年 Prentice Hall 公司出版了 Kernighan 和 Ritchie 合作的著作《The C Programming Language》,引起了人们对 C 语言的广泛关注(见参考文献 Ke78)。

C 语言在各种不同类型计算机(有时称为硬件平台)上的普及导致了許多变形。它们虽然相似,但通常互不兼容。对需要为不同平台编写可移植程序的开发人员,这是个严重问题,显然需要有个标准的 C 语言版本。1983 年,美国国家计算机与信息处理标准委员会(x3)成立了 x3J11 技术分会,目的是提供无歧义性且与机器无关的语言定义。1989 年推出了这种语言标准。ANSI 与国际标准化组织(ISO)合作,在全球范围内将 C 语言标准化,1990 年推出了联合标准文档。称为 ANSI / ISO9899: 1990。这个文档可以从 ANSI 获得副本。1988 年推出的 Kernighan 和 Ritchie 著作的第二版体现了该版本(称为 ANSI C),这也是目前全世界使用的版本。

C++是 C 语言的扩展,是 20 世纪 80 年代初由贝尔实验室的 Bjarne Stroustrup 开发的。C++的许多特性是从 c 语言中派生的,但更重要的是,它提供了面向对象编程(object-oriented programming)的功能。

软件业正在酝酿一场革命,最终目标是更快、更正确、更经济地建立软件,新的、更强大的软件需求迫在眉睫。对象(object)实际上是模拟实际项目的可复用软件组件(component)。软件开发人员发现,利用模块化、面向对象的设计和实现方法与过去结构化编程方法相比较,可以使软件开发小组的生产率更高。面向对象编程的优势在于更容易理解、纠正和修改。

许多面向对象的语言也纷纷涌现,包括最著名的由 Xerox 的 Palo Alto 研究中心(PARC)开发的 small talk。Smalltalk 是纯粹的面向对象的语言,其所有的编程元素都是“对象”。C++则是一种“混合型语言”,可以用 C 语言方式、面向对象方式或兼用两种方式进行编程。2.9 节将介绍基于 C / C++的新语言——Java。

2.3 C++标准库

C++程序由类(class)和函数(function)组成。可以用多个小的软件模块构成 C++程序,但大多数 C++程序员会利用 C++标准库中已有的类和函数来编程。这样,C++“世界”中实际要学习两方面的知识,第一是学习 C++语言本身,第二是学习如何利用 C++标准库中现有的类和函数(本教程将介绍许多类和函数)。Plauger 的著作是程序员必读的,可以帮助程序员深入了解 C++中包括的 ANSI C 语言库函数,了解如何实现这些库函数,还可以了解如何用库函数编写可移植代码。标准库函数通常由编译器厂家提供。许多独立软件供应商(independent software vender)也提供各种专用类库。

常用数据类型与 Pascal 数据类型的比较：

Pascal	数据类型	C/C++
ShortInt	8 位有符号整数	char
Byte	8 位无符号整数	BYTE, unsigned short
SmallInt	16 位有符号整数	short
Word	16 位无符号整数	unsigned short
Integer, LongInt	32 位有符号整数	int, long
Cardinal, LongWord/DWORD	32 位无符号整数	unsigned long
Int64	64 位有符号整数	_int64
Single	4 字节浮点数	float
*Real48	6 字节浮点数	
Double	8 字节浮点数	double
*Extended	10 字节浮点数	long double
Currency	64 位货币类型	
TDate/TDateTime	8 字节日期/时间	
Variant, OleVariant	16 字节可变类型	VARIANT, ^Variant, ^OleVariant
Char, AnsiChar	1 字节字符	char
WideChar	2 字节字符	WCHAR
*ShortString	短字符串	string
AnsiString/String	长字符串	^AnsiString
WideString	宽字符串	^WideString
PChar, PAnsiChar	NULL 结束的字符串	char*
PWideChar	NULL 结束的宽字符串	LPCWSTR
Boolean, ByteBool	1 字节布尔类型	任何 1 字节
WordBool	2 字节布尔类型	任何 2 字节
BOOL, LongBool	4 字节布尔类型	BOOL
注：有*前缀的是向前兼容类型；有^前缀的是 C++Builder 特有类型。		

2.4 结构化编程

20 世纪 60 年代，许多大型软件的开发遇到了严重困难。常常推迟软件计划，因而使成本大大超过预算，而且最终产品也不可靠。人们开始认识到，软件开发是项复杂的活动，比原来所预想的要复杂得多。20 世纪 60 年代的研究结果是结构化编程(structured programming)的出现，用规定的方法编写程序比非结构化编程能产生更清晰、更容易测试 / 调试以及更容易修改的程序。本教程的第 2 章将介绍结构化编程原理。第 3 章到第 5 章则会开发多种结构化程序。

结构化编程研究的一个更具体结果是 1971 年 Niklaus Wirth 教授推出了 Pascal 语言。Pascal

语言是用 17 世纪著名数学家和哲学家巴雷斯·帕斯卡(Blaise Pascal)的名字命名的，常用于教学中讲解结构化编程。因而很快成为了大学中受欢迎的语言。但是这个语言缺乏在商业、工业和政府应用程序中所需要的许多特性，因此没有被大学以外的环境所接受。

Ada 语言是在 20 世纪 70 年代和 80 年代初由美国国防部资助开发的。在此之前，国防部的导弹命令与控制软件系统是由几百种不同语言生成的，国防部要求用一种语言来完成大多数工作。Ada 以 Pascal 为基础，但最终结构与 Pascal 大相径庭。这个语言是根据著名诗人 Lord Byron 的女儿(Ada Lovelace)的名字命名的。Ada Lovelace 在 19 世纪初编写了世界上第一个计算机程序，用于 Charles Babbage 设计的分析机引擎的计算设备。Ada 的一个最重要功能是多任务(multitasking)。程序员可以使多个活动任务并行发生。我们要介绍的其他常用高级语言(包括 C/C++)通常只让程序员编写一次只有一个活动任务的程序。

2.5 简单程序

C++使用非程序员可能感到奇怪的符号。我们首先介绍一个简单程序：打印一行文本。程序及其屏输出如图 2.2。

这段程序演示了 C++语言的几个重要特性。我们详细介绍程序的每一行。

// Fig.2.2:fig1_02.cpp

// A first program in C++

以//开头，表示该选项其余部分是注释语句(comment)。程序员手稿注释语句用来说明和提高程序的可读性。注释语句还可以帮助其它人阅读和理解程序。在运行程序时注释语句并不使计算机采用任何操作。C++编译器忽略注释语句，不产生任何机器目标码。注释语句"first program in C++"只是描述该程序的用途。以//开头的说明语句称为单行注释语句(single-line comment)，因为该注释语句在行尾结束(注意：C++程序员也可以用 C 语言注释语句样式，即注释语句以/*开头，以*/结束)。

```
1 // Fig. 2.2:fig01_02.cpp
2 // A first program in C++
3 #include <iostream>
4 using namespace std;
5 int main()
6 {
7     cout<<"Welcom to C++!\n";
8
9     rreturn 0;    // indicate that program ended sucessfully
10 }
```

输出结果：

Welcom to C++!

下列语句：

#include <iostream>

是条预处理指令(preprocessor directive)，是发给 C++预处理器的消息。预处理器先处理以#开头的一行语句之后再编译该程序。为一行预处理指令告诉预处理器要在程序中包括输入、输出的头文件 iostream.h 的内容。应该在任何使用 C++式输入、输出之前从键盘输入数据或向屏幕输出

数据的程序中包括这个头文件。注意，最新 ANSI、ISO C++ 标准实际上指定 `iostream.h` 和其它标准头文件不需要后缀 `.h`，如 `iostream`。我们在本教程中有时继续使用旧式头文件，因为有些编译器还不支持最新的 ANSI/ISO C++ 草案标准。

下列语句：

```
int main()
```

是每个 C++ 程序都包的语句。`main` 后面的括号表示 `main` 是个程序基本组件，称为函数 (function)。C++ 程序包含一个或几个函数，其中有且只有一个 `main` 函数即使 `main` 不是程序中的第一个函数，C++ 程序通常都从 `main` 函数开始执行。`main` 左边的关键字 `int` 表示 `main` 返回一个整数值。

左花括号 `{}` 应放在每个函数体 (body) 开头，对应右花括号 `}` 应放在每个函数的结尾。下列语句：

```
cout<<"Welcom to C++!\n";
```

让计算机在屏幕上打印引号之间的字符串 (string)。整个行称为语句 (statement)，包括 `cout<<` 运算符、字符串 `"Welcom to C++!\n"` 和分号 `;`。每条语句应以分号 (又称为语句终止符) 结束。C++ 中的输出和输入是用字符流 (stream) 完成的，这样，执行上述语句时，将字符流 `"Welcom to C++!"` 发送到标准输出流对象 (standard output stream object) `cout`，通常 `cout` 将其输出到屏幕。第 3 章“C++ 输入/输出流”中将详细介绍 `cout`。

运算符 `<<` 称为流插入符 (stream insertion operator)。执行这个程序时，运算符右边的值 (历操作数) 插入输出流中。历操作数通常按引号中的原样直接打印。但注意字符 `\n` 不在屏幕中打印。反斜杠 `\` 称为转义符 (escape character)，表示要输出特殊字符。字符串中遇到反斜杠时，下一个字符与反斜杠组合，形成转义序列 (escape sequence)。转义序列 `\n` 表示换行符 (newline)。使光标 (即当前屏幕位置的指示符) 移到下一行开头。表 2.3 列出了常用转义序列。

下列语句：

```
return 0;           // indicate that program ended sucessfully
```

放在每个 `main` 函数的末尾。C++ 的关键字 `return` 是退出函数的几种方式之一。`main` 函数末尾使用 `return` 语句时，数值 `0` 表示顺利结束。第 3 章将详细介绍和解释包括这个语句的原因。目前只要记住在每个程序中都要包括这个语句。否则在某些程序中编译器会产生警告消息。

右花括号 `}` 表示 `main` 函数结束。

转义序列	说明
<code>\n</code>	换行符，使屏幕光标移到屏幕中下一行开头
<code>\t</code>	水平制表符，使屏幕光标移到下一制表位
<code>\r</code>	回车符，使屏幕光标移到当前行开头，不移到下一行
<code>\a</code>	警告，发出系统警告声音
<code>\\</code>	反斜杠，打印反斜杠符
<code>\"</code>	双引号，打印双引号

图 2.3 常用转义序列

许多程序员让函数打印的最后一个字符为换行符 `(\n)`。这样可以保证函数使屏幕光标移到屏幕中下一行开头，这种习惯能促进软件复用，这是软件开发环境中的关键目标。确定一个喜欢的缩排长度，然后一直坚持这个缩排长度。可以用制表符生成缩排，但制表位可能

改变。建议用 1 / 4 英寸制表位或三个空格的缩排长度。

"Welcom to C++!"可用多种方法打印。例如，图 2.4 的程序用多条流插入语句，产生的程序输出与图 2.2 相同，因为每条流插入语句在上一条语句停止的位置开始打印。第一个流插入语句打印“Welcome”和空格，第二条流插入语句打印同一行空格后面的内容。C++允许以多种方式表达语句。

一条语句也可以用换行符打印多行，如图 2.5。每次在输出流中遇到\n 转义序列时，屏幕光标移到下一行开头。要在输出中得到空行，只要将两个\n 放在一起即可，如图 2.5。

```
1 // Fig. 2.4:fig01_04.cpp
2 // printing a line with multiple statements
3 #include <iostream>
4 using namespace std;
5 int main()
6 {
7     cout<<"Welcom ";
8     cout<<"to C++!\n";
9
10    return 0;
11 }
输出结果：
Welcom to C++!
```

图 2.4 用多条流插入语句打印一行

```
1 // Fig. 2.5:fig01_05.cpp
2 // printing multiple lines with a single statement
3 #include <iostream>
4 using namespace std;
5 int main()
6 {
7     cout<<"Welcom\nto\n\nC++!\n";
8
9     return 0;           // indicate that program ended sucessfully
10 }

输出结果：
Welcome
to

C++!
```

图 2.5 用一条流插入语句打印多行

2.6 简单程序：两个整数相加

下一个程序用输入流对象 `cin` 和流读取运算符 `>>` 取得用户从键盘中输入的两个整，计算这两个值的和，并将结果用 `cout` 输出。程序及其输出如图 2.6。

```
1 // Fig.2.6:fig01_06.cpp
2 // Addition program
3 #include <iostream>
4 using namespace std;
5 int main()
6 {
7     int integer1, integer2, sum;           // 声明三个变量
8
9     cout<<"Enter first integer\n";        // 提示信息
10    cin>>integer1;                          // 从键盘读一个整数
11    cout<<"Enter second integer\n";        // 提示信息
12    cin>>integer2;                          // 从键盘读一个整数
13    sum=integer1+integer2;                  // 两整数相加，值赋给变量 sum
14    cout<<"Sum is "<<sum<<endl;           // 输出和
15
16    return 0;                               // 返回 0 值表示程序运行成功。
17 }
```

输出结果：

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

图 2.6 两个整数相加

注释语句：

```
// Fig. 2.6: fig01_06.cpp
// Addition program
```

指定文件名和用途。C++预处理指令：

```
#include <iostream>
```

将 `iostream.h` 头文件的内容放进程序中。

前面介绍过，每个程序从 `main` 函数开始执行。左花括号表示 `main` 函数体开头，相应右花括号表示 `main` 函数体结束。下列语句：

```
int integer1, integer2, sum;
```

是个声明(declaration)。`integer1`, `integer2` 和 `sum` 是变量(variable)名。变量是计算机内存中的地址，存放程序使用的值。这个声明指定变量 `integer1`, `integer2` 和 `sum` 的数据类型为 `int`，表示这些变量保存整数值，如 7、-11、0、31914。所有变量都应先声明名称和数据类型后才能在程序中使用。

几个相同类型的变量可以在同一声明或多个声明中声明。我们可以一次只声明一个变量，但一次声明多个同类型变量更加简练。

稍后要介绍数据类型 `float`，(定义实数，即带小数点的数，如 3.4、0.0、-11.19)和 `char`(定义字符型数据。变量 `char` 只能保存一个小写字母、一个大写字母、一个数字或一个特殊字符，如 `x`、`$`、`7`、`*`等等)。

变量声明可以放在函数的任何位置，但变量声明必须放在程序使用变量之前。如果不用一条语句声明三个变量也可以分别声明。下列声明：

```
int integer1;
```

可以放在下列语句之前：

```
cin >> integer1;
```

下列声明：

```
int integer2;
```

可以放在下列语句之前：

```
cin >> integer2;
```

下列声明：

```
int sum;
```

可以放在下列语句之前：

```
sum = integer1 + integer2;
```

下列句：

```
cout<<"Enter first integer\n";
```

在屏幕上打印字符串 `Enter first integer`(`b` 也称为字符串直接量(string literal)或直接量(literal))，将光标移到下一行开头。这个消息称为提示(prompt)，提示用户进行特定操作。上述语句表示 `cout` 得到字符串 `"Enter first integer\n"`。

下列语句：

```
cin>>integer1;
```

用输入流对象 `cin` 和流读取运算符 `>>` 取得键盘中的值。利用流读取运算符 `cin` 从标准输入流读取输入(通常是键盘输入)。上述语句表示 `cin` 提供 `integer1` 的值。

计算机执行上述语句时，等待用户输入变量 `integer1` 的值。用户输入整数值并按 `Enter` 键(或 `Return` 键)，将数值发送给计算机。然后计算机将这个数(值)赋给变量 `integer1`。程序中后面引用 `integer1` 时都使用这个值。

`cout` 和 `cin` 流对象实现用户与计算机之间的交互。由于这个交互像对话一样，因此通常称为对话式计算(conversational computing)或交互式计算(interactive computing)。

下列语句：

```
cout<<"Enter second integer\n";
```

在屏幕上打印 `"Enter second integer"` 字样，然后移到下一行的开头。这个语句提示用户进行操作。

下列语句：

```
cin>>integer2;
```

从用户取得变量 `integer2` 的值。

赋值语句：

```
sum=integer1+integer2;
```

计算变量 `integer1` 和 `integer2` 的和，然后用赋值运算符(assignment operator) `"="` 将结果赋给变

量 sum。这个语句表示 sum 取得 integer1 加 integer2 的值。大多数计算都是在赋值语句中进行的。“=”运算符和前面的“+”运算符称为二元运算符，两个操作数是 integer1 和 integer2。而对于“=”运算符，两个操作数是 sum 和表达式 integer1+integer2 的值。

下列语句：

```
cout<<"Sum is"<<sum<<endl;
```

打印字符串"Sum is"和变量 sum 的数值，加上称为流操纵算子的 endl(end line 的缩写)。endl 输出一个换行符，然后刷新输出缓冲区，即在一些系统中，输出暂时在机器中缓存，等缓冲区满时再打印到屏幕上，endl 强制立即输出到屏幕上。

注意，上述语句输出多种不同类的值，流插入运算符知道如何输出每个数据。在一个语句中使用多个流插入运算符称为连接(concatenating)、链接(chaining)或连续使用流插入操作。这样，就不必用多条输出语句输出多个数据。

计算可以在输出语句中进行。可以将上述语句合二为一：

```
cout<<"Sum is"<<integer1+integer2<<endl;
```

从而不需要变量 sum。

右花括号告诉计算机到达了函数 main 的结尾。

C++的一个强大特点就是用户可以生成自己的数据类型(详见第 6 章)，然后可以告诉 C++如何使用>>和<<运算符输入或输出这种类型的值(称为运算符重载，见第 8 章)。

2.7 算术运算

算术运算符见图 2.10，注意这里使用了许多代数中没有使用的符号。星号(*)表示乘法、百分号(%)表示求模(modulus)。图 2.10 所示的算术运算符都是二元运算符，即这些运算符取两个操作数。例如，表达式"integer1+integer2"包含二元运算符“+”和两个操作数 integer1 和 integer2。

C++操作	算术运算符	代数表达式	C++表达式
加	+	f+7	f+7
减	-	p-c	p-c
乘	*	bm	b*m
除	/	x/y 或 $x \div y$	x/y
求模	%	r mod s	r%s

图 2.10 算术运算符

整除(即除数和被除数均为整数)取得整数结果。例如，表达式 7/4 得 1，表达式 17/5 得 3。注意，整除结果忽略分数部分，不用取整。

C++提供求模(modulus)运算符“%”即求得整除的余数。求模运算是个整数运算符，只能使用整数操作数。表达式 x%y 取得 x 除以 y 的余数，这样，7%4 得 3，17%5 得 2。后面几章将介绍求模运算符许多有趣的应用。如确定一个数是否为另一个数的倍数(确定一个数为奇数或偶数是这个问题的一个特例)。

C++中的算术运算表达式应以直线形式在计算机中输入。这样，a 除以 b 应输入为"a/b"，使所有常量、变量和运算符放在一行中。编译器通常不接受下列代数符号：

```
a
—
b
```

但有些特殊专业软件包支持复杂数学表达式更自然的表示方法。

C++表达式中括号的使用和代数表达式中相同。例如，要将 a 乘以 $b+c$ 的和，可以用：

$a*(b+c)$

C++中算术运算符的运算顺序是由运算符的优先级规则确定的，与代数中的相同：

1. 括号中的表达式先求值，程序员可以用括号指定运算顺序。括号具有最高优先级，对于嵌套括号，由内存向外层求值。
2. 乘法、除法、求模运算优先。如果表达式中有多个乘法、除法、求模运算，则从左向右求值。乘法、除法、求模的优先级相同。
3. 然后再进行加法和减法。如果表达式中有多个加法和减法，则从左向右求值。加法和减法的优先级相同。

运算符优先级保证 C++ 按正确顺序采用运算符。从左向右求值指的是运算符的结合律 (associativity)，也有一些运算符结合律是从右向左。图 2.11 总结了运算符优先级规则，引入其它 C++ 运算符时，这个表可以扩充，详细的运算符优先级请参见附录。

运算符	运算	求值顺序
()	括号	最先求值，如果有嵌套括号，则先求最内层表达式的值，如果同一层有几对括号，则从左向右求值。
*, /, 或 %	乘、除、求模	其次求值。如果有多个，则从左向右求值。
+或-	加、减	最后求值。如要有多个，则从左向右求值。

图 2.11 算术运算符优先级

下面用几个表达式说明运算符优先级规则。每个例子都列出代数表达式和对应的 C++ 表达式。

下例求五个值的算术平均值：

代数： $m = \frac{a+b+c+d+e}{5}$

C++： $m = (a+b+c+d+e)/5;$

括号是必须的，因为作法的优先级比加法高，要把整个和 $(a+b+c+d+e)$ 除以 5，如果不加括号，则 $a+b+c+d+e/5$ 的取值为：

$a+b+c+d+(e/5)$

下例是直线的方程：

代数： $y = mx+b$

C++： $y = m*x+b;$

不需要括号，乘法优先于加法，因此先乘后加。

下列包含模(%)、乘、除、加、减运算：

代数： $z = pr\%q+w/x-y$

C++： $z = p * r \% q + w / x - y;$

⑥ ① ② ④ ③ ⑤

语句下面的圆圈数字表示 C++ 采用运算符的顺序。乘法、求模和除法首先从左向右求值(结合律为从左向右)因为它们优先级高于加法和减法。然后进行加法和减法运算，也是从左向右求值。

并不是有多对括号的表达式都包含嵌套括号。例如下列表达式不包含嵌套括号：

`a * (b+c)+c*(d+e)`

这些括号在同一层。

要更好地了解运算符优先级规则，考虑二次多项式的求值：

`y = a * x * x + b * x + c;`

⑥ ① ② ④ ③ ⑤

语句下面的圆圈数字表示 C++采用运算符的顺序。C++中没有指数运算符，因此我们把 `x2` 表示为 `x*x`，稍后会介绍标准库函数 `pow(数)`。由于 `pow` 所需要的数据类型有一些特殊情况，因此放第 3 章再介绍。

假设变量 `a`、`b`、`c`、`x` 初始化如下：`a=2,b=3,c=7` 和 `x=5`。图 2.12 演示了上述二次多项式的运算符优先级。

上述赋值语句可以加上多余的括号，使代码更清晰：

`y=(a*x*x)+(b*x)+c;`

2.8 判断：相等与关系运算符

本节介绍简单的 C++ `if` 结构，使程序根据某些条件的真假做出判断。如果条件符合，即为真(true)，则执行 `if` 结构体的语句；如果不符合，即条件为假(false)，则不执行语句，稍后将举例说明。

`if` 结构中的条件可以用相等运算符(equality operator)和关系运算符(relational operator)表示，如图 2.13 关系运算符具有相同的优先级，结合律为从左向右。相等运算符的优先级也相同，但低于关系运算符的优先级，结合律也为从左向右。

标准代数相等与关系运算符	C++相等与关系运算符	C++条件举例	C++条件含义
<code>=</code>	<code>==</code>	<code>x==y</code>	<code>x</code> 等于 <code>y</code>
<code>≠</code>	<code>!=</code>	<code>x!=y</code>	<code>x</code> 不等于 <code>y</code>
关系运算符			
<code>></code>	<code>></code>	<code>x>y</code>	<code>x</code> 大于 <code>y</code>
<code><</code>	<code><</code>	<code>x<y</code>	<code>x</code> 小于 <code>y</code>
<code>≥</code>	<code>>=</code>	<code>x>=y</code>	<code>x</code> 大于或等于 <code>y</code>
<code>≤</code>	<code><=</code>	<code>x<=y</code>	<code>x</code> 小于或等于 <code>y</code>

图 2.13 相等与关系运算符

下例用六个 `if` 语句比较用户输入的两个数。如果其中任何一个 `if` 语句的条件成立，则执行与该 `if` 相关联的输出语句。图 2.14 显示了这个程序和三个示例输出。

注意图 2.14 的程序边疆使用流读取操作输入两个整数。首先将一个值读到 `num1` 中，然后将一个值读到 `num2` 中。`if` 语句的缩排是为了提高程序的可读性。另外，注意图 2.14 中每个 `if` 语句体中有一条语句。第 2 章将会介绍结构体中有多条语句的 `if` 语句(将语句体放在花括号“`{ }`”中)。

```
1 // Fig.2.14:fig01_14.cpp
2 // Using if statements,relational
```

```

3 // operators,and equality operators
4 #include <iostream>
5 using namespace std;
6 int main()
7 {
8     int num1,num2;
9
10    cout<<"Enter two integers,and I will tell you\n"
11    <<"the relationships they satisfy: ";
12    cin>>num1>>num2;        // 读取两个整数
13
14    if (num1==num2)
15        cout<<num1<<" is equal to"<<num2<<endl;
16
17    if (num1!=num2)
18        cout<<num1<<" is not equal to "num2<<endl;
19
20    if (num1<num2)
21        cout<<num1<<" is less than "<<num2<<endl;
22
23    if (num1>num2)
24        cout<<num1<<" is greater than "<<num2<<endl;
25
26    if (num1<=num2)
27        cout<<num1<<" is less than or equal to "
28        <<num2<<endl;
29
30    if (num1>=num2)
31        cout<<num1<<" is greater than or equal to "
32        <<num2<<endl;
33
34    return 0;        // 返回一个程序正常结束的标识
35 }

```

输出结果:

```

Enter two integers,and I will tell you
The relationships they satisfy: 3 7
3 is not equal 7
3 is less than 7
3 is less than or equal to 7

```

```

Enter two integers,and I will tell you
the relationships they satisfy:22 12
22 is not equal 12

```

```

22 is gretaer than 12
22 is greater than or equal to 12
Enter two integers,and I will tell you
the relationships they satisfy:7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7

```

图 2.14 使用相等和关系运算符

注意图 2.14 中空格的用法。在 C++ 语言中，空白字符(如制表符、换行符和空格)通常都被编译器忽略。因此，语句中可以根据程序员的爱好加上换行符和空格，但不能用换行符和空格分隔标识符。

图 2.15 显示了本章介绍的运算符优先级，运算符优先顺序从上向下递减。注意赋值运算符之后的所有运算符结合律均为从左向右。加法是左结合的，因此表达式 $x+y+z$ 求值为 $(x+y)+z$ 。赋值运算符是从右向左结合的，因此表达式 $x=y=0$ 求值为 $x=(y=0)$ ，首先将 0 赋给 y ，然后将结果 0 赋给 x 。

运算符	结合律	类型
()	从左向右	括号
* / %	从左向右	乘
+ -	从左向右	加
<< >>	从左向右	流插入/读取
< <= > >=	从左向右	关系
== !=	从左向右	等于
=	从右向左	赋值

图 2.15 运算符优先级和结合律

2.9 新型头文件与名字空间

本节是为使用支持 ANSI/ISO 草案标准的编译器用户提供的。草案标准指定了许多旧式 C++ 头文件的新名，包括 `iostream.h`，大多数新式头文件不再用扩展名 `.h`。图 2.16 改写图 2.2，演示新型头文件和两种使用标准库头文件的方法。

第 3 行：

```
#include <iostream>
```

演示新型头文件 名语法。

第 5 行：

```
using namespace std;
```

指定用 `std` 名字空间(namespace)，这是 C++ 中的新特性。名字空间可以帮助程序员开发新的软件组件而不会与现有软件组件产生命名冲突。开发类库的一个问题是类和函数名可能已经使用。名

名字空间能为每个新软件组件保持唯一的名称。

```
1 // Fig. 2.16:fig01_16.cpp
2 // Using new-style header files
3 #include <iostream.h>
4
5 using namespace std;
6
7 int main()
8 {
9     cout<<"Welcom to C++!\n";
10    std::cout<<"Welwcom to C++!\n";
11
12    return 0;
13 }
```

输出结果:

Welcom to C++!

Welcom to C++!

图 2.16 使用新型头文件

C++草案标准中的每个头文件用名字空间 `std` 保证今后 C++标准库操作的每个特性是唯一的，不会与其它程序员开发的软件组件混淆起来，程序员不能用名字空间定义新的类库。上述语句只是表示我们使用 C++标准库中的软件组件，要定义自己的类库，则可以将我们的所有类和函数放在名字空间 `deitel` 中，使我们的类库与所有其它公司的类库和 C++标准类库区别开来。

程序中出现"using namespace std"语句之后，就可以像第 9 行那样用 `cout` 对象将数值输出到标准输出流中。如果使用两个或几个类库，其中有的特性名称相同，则可能引起名称冲突。这时就要用名字空间来完全限定所用的名称，例如第 10 行的 `std::cout`:

```
std::cout<<"Welcom to C++!\n";
```

`cout` 的完全限定名为 `std::cout`，如果全部采用这种形式，虽然比较繁琐，但程序中第 5 行的"using namespace std"语句就没有必要了。using 语句可以在 C++标准库中使用每个名称的简写版本(或其它指定名字空间的名称)。我们将在本教程稍后详细介绍名字空间。目前并不是所有 C++环境都已经支持新的头文件的命名格式。为此，我们在本教程大部分地方使用旧式头文件，只在介绍 C++标准新特性时才使用新的头文件命名格式，使用新格式时将特别注明。

第 3 章 C++输入/输出流

3.1 简介

C++标准库提供了一组扩展的输入/输出(I/O)功能。本章将详细介绍 C++中最常用的一些 I/O 操作,并对其余的输入/输出功能做一简要的概述。本章的有些内容已经在前面提到,这里对输入/输出功能做一个更全面的介绍。

本章讨论的许多输入/输出功能都是面向对象的,读者会发现 C++的 I/O 操作能够实现许多功能。C++式的 I/O 中还大量利用了 C++的其他许多特点,如引用、函数重载和运算符重载等等。

C++使用的是类型安全(typesafe)的 I/O 操作,各种 I/O 操作都是以对数据类型敏感的方式来执行的。假定某个函数被专门定义好用来处理某一特定的数据类型,那么当需要的时候,该函数会被自动调用以处理所对应的数据类型。如果实际的数据类型和函数之间不匹配,就会产生编译错误。

因此,错误数据是不可能通过系统检测的(C 语言则不然。这是 C 语言的漏洞,会导致某些相当微妙而又奇怪的错误)。

用户既可以指定自定义类型的 I/O,也可以指定标准类型的 I/O。这种可扩展性是 C++最有价值的特点之一。

3.2 流

C++的 I/O 是以字节流的形式实现的,流实际上就是一个字节序列。在输入操作中,字节从输入设备(如键盘、磁盘、网络连接等)流向内存;在输出操作中,字节从内存流向输出设备(如显示器、打印机、磁盘、网络连接等)。

应用程序把字节的含义与字节关联起来。字节可以是 ASCII 字符、内部格式的原始数据、图形图像、数字音频、数字视频或其他任何应用程序所需要的信息。

输入/输出系统的任务实际上就是以一种稳定、可靠的方式在设备与内存之间传输数据。传输过程中通常包括一些机械运动,如磁盘和磁带的旋转、在键盘上击键等等,这个过程所花费的时间要比处理器处理数据的时间长得多,因此要使性能发挥到最大程度就需要周密地安排 I/O 操作。一些介绍操作系统的书籍(见参考文献)深入地讨论了这类问题。

C++提供了低级和高级 I/O 功能。低级 I/O 功能(即无格式 I/O)通常只在设备和内存之间传输一些字节。这种传输过程以单个字节为单位,它确实能够提供高速度并且可以大容量的传输,但是使用起来不太方便。

人们通常更愿意使用高级 I/O 功能(即格式化 I/O)。高级 I/O 把若干个字节组合成有意义的单位,如整数、浮点数、字符、字符串以及用户自定义类型的数据。这种面向类型的 I/O 功能适合于大多数情况下的输入输出,但在处理大容量的 I/O 时不是很好。

3.2.1 iostream 类库的头文件

C++的 iostream 类库提供了数百种 I/O 功能，iostream 类库的接口部分包含在几个头文件中。

头文件 iostream.h 包含了操作所有输入/输出流所需的基本信息，因此大多数 C++程序都应该包含这个头文件。头文件 iostream.h 中含有 cin、cout、cerr、clog 4 个对象，对应于标准输入流、标准输出流、非缓冲和经缓冲的标准错误流。该头文件提供了无格式 I/O 功能和格式化 I/O 功能。

在执行格式化 I/O 时，如果流中带有含参数的流操纵算子，头文件 iomanip.h 所包含的信息是有用的。

头文件 fstream.h 所包含的信息对由用户控制的文件处理操作比较重要。第 13 章将在文件处理程序中使用这个头文件。

每一种 C++版本通常还包括其他一些与 I/O 相关的库，这些库提供了特定系统的某些功能，如控制专门用途的音频和视频设备。

3.2.2 输入/输出流类和对象

iostream 类库包含了许多用于处理大量 I/O 操作的类。其中，类 istream 支持流输入操作，类 ostream 支持流输出操作，类 iostream 同时支持流输入和输出操作。

类 istream 和类 ostream 是通过单一继承从基类 ios 派生而来的。类 iostream 是通过多重继承从类 istream 和 ostream 派生而来的。

运算符重载为完成输入/输出提供了一种方便的途径。重载的左移位运算符(<<)表示流的输出，称为流插入运算符；重载的右移位运算符(>>)表示流的输入，称为流读取运算符。这两个运算符可以和标准流对象 cin、cout、cerr、clog 以及用户自定义的流对象一起使用。

cin 是类 istream 的对象，它与标准输入设备(通常指键盘)连在一起。下面的语句用流读取运算符把整数变量 grade(假设 grade 为 int 类型)的值从 cin 输入到内存中。

```
cin >> grade;
```

注意，流读取运算符完全能够识别所处理数据的类型。假设已经正确地声明了 grade 的类型，那么没有必要为指明数据类型而给流读取运算符添加类型信息。

cout 是类 ostream 的对象，它与标准输出设备(通常指显示设备)连在一起。下面的语句用流插入运算符 cout 把整型变量 grade 的值从内存输出到标准输出设备上。

```
cout << grade;
```

注意，流插入运算符完全能够识别变量 grade 的数据类型，假定已经正确地声明了该变量，那么没有必要再为指明数据类型而给流插入运算符添加类型信息。

cerr 是类 ostream 的对象，它与标准错误输出设备连在一起。到对象 cerr 的输出是非缓冲输出，也就是说插入到 cerr 中的输出会被立即显示出来，非缓冲输出可迅速把出错信息告诉用户。

clog 是类 ostream 的对象，它与标准错误输出设备连在一起。到对象 clog 的输出是缓冲输出。即每次插入 clog 可能使其输出保持在缓冲区，要等缓冲区刷新时才输出。

C++中的文件处理用类 ifstream 执行文件的输入操作，用类 ofstream 执行文件的输出操作，用类 fstream 执行文件的输入/输出操作。类 ifstream 继承了类 istream，类 ofstream 继承了类 ostream，类 fstream 继承了类 iostream。虽然多数系统所支持的完整的输入 / 输出流类层次结构中还有很多类，但这里列出的类能够实现多数程序员所需要的绝大部分功能。如果想更多地了解有关文件处理的内容，可参看 C++系统中的类库指南。

3.3 输出流

C++的类 `ostream` 提供了格式化输出和无格式输出的功能。输出功能包括：用流插入运算符输出标准类型的数据；用成员函数 `put` 输出字符；成员函数 `write` 的无格式化输出(3.5 节)；输出十进制、八进制、十六进制格式的整数(3.6.1 节)；输出各种精度的浮点数(3.6.2 节)、输出强制带有小数点的浮点数(3.7. 2 节)以及用科学计数法和定点计数法表示的浮点数(3.7. 6 节)；输出在指定域宽内对齐的数据(3.7. 3 节)；输出在域宽内用指定字符填充空位的数据(3.7. 4 节)；输出科学计数法和十六进制计数法中的大写字母(3.7. 7 节)。

3.3.1 流插入运算符

流插入运算符(即重载的运算符<<)可实现流的输出。重载运算符<<是为了输出内部类型的数据项、字符中和指针值。3.9 节要详细介绍如何用重载运算符<<输出用户自定义类型的数据项。

图 3.3 中的范例程序用一条流插入语句显示了输出的字符串。图 3.4 中的范例程序用多条流插入语句显示输出的字符串，该程序的运行结果与图 3.3 中程序的运行结果相同。

```
1 // Fig. 3.3: figl1 03.cpp
2 // Outputting a string using stream insertion.
3 #include <iostream>
4
5 using namespace std;
6 int main()
7     cout << "Welcome to C++!\n";
8
9     return 0;
10 }
```

输出结果：

Welcome to C++!

图 3.3 用一条流插入语句输出字符串

```
1 // Fig. 3.4: figl1O4.cpp
2 // Outputting a string using two stream insertions.
3 #include <iostream>
4 using namespace std;
5 int main()
6 {
7     cout << "Welcome to ";
8     cout << "C++!\n";
9
10     return 0;
```

```
11 }
```

输出结果:

Welcome to C++!

图 3.4 用两条流插入语句输出字符串

也可以用流操纵算子 `endl`(行结束)实现转义序列 `\n`(换行符)的功能(见图 3.5)。流操纵算子 `endl` 发送一个换行符并刷新输出缓冲区(不管输出缓冲区是否已满都把输出缓冲区中的内容立即输出)。也可以用下面的语句刷新输出缓冲区:

```
cout << flush;
```

3.6 节要详细讨论流操纵算子。

```
1 // Fig. 3.5: fig11_05.cpp
2 // Using the endl stream manipulator.
3 #include <iostream.h>
4 using namespace std;
5 int main()
6 {
7     cout << "Welcome to ";
8     cout << "c++!";
9     cout << endl; // end line stream manipulator
10
11     return 0;
12 }
```

输出结果:

Welcome to C++!

图 3.5 使用流操纵算子 `endl`

流插入运算符还可以输出表达式的值(见图 3.6)

```
1 // Fig. 3.6: fig11_O6.cpp
2 // Outputting expression values.
3 #include <iostream.h>
4 using namespace std;
5 int main()
6 {
7     cout << "47 plus 53 is ";
8
9     // parentheses not needed; used for clarity
10    cout << ( 47 + 53 ); // expression
11    cout << endl;
```

```

12
13     return O;
14 }

```

输出结果:

47 plus 53 is 100

图 3.6 输出一个表达式的值

3.3.2 连续使用流插入/流读取运算符

重载的运算符<<和>>都可以在一条语句中连续使用(见图 3.7)。

```

1  file:// Fig. 3.7: fig11_07.cpp
2  file:// Cascading the overLOadGd << OPeratOr.
3  #include<iostream.h>
4
5  int  main()
6  {
7      cout << "47 plus 53 is " << ( 47 + 53 ) << endl;
8
9      return  O;
10 }

```

输出结果:

47 plus 53 is 100

图 3.7 连续使用重载运算符<<

图中多次使用流插入运算符的语句等同于下面的语句:

```
(( ( cout << " 47 plus 53 is " ) << 47 + 53 ) << endl);
```

之所以可以使用这种写法,是因为重载的运算符<<返回了对其左操作数对象(即 cout)的引用,因此最左边括号内的表达式:

```
( cout << " 47 plus 53 is " )
```

它输出一个指定的字符串,并返回对 cout 的引用,因而使中间括号内的表达式解释为:

```
( cout << ( 47 + 53 ) )
```

它输出整数值 100,并返回对 cout 的引用。于是最右边括号内的表达式解释为:

```
cout << endl;
```

它输出一个换行符,刷新 cout 并返回对 cout 的引用。最后的返回结果未被使用。

3.3.3 输出 char*类型的变量

C 语言式的 I/O 必须要提供数据类型信息。C++对此作了改进,能够自动判别数据类型。但

是，C++中有时还得使用类型信息。例如，我们知道字符串是 `char*` 类型，假定需要输出其指针的值，即字符串中第一个字符的地址，但是重载运算符 `<<` 输出的只是以空(`null`)字符结尾的 `char*` 类型的字符串，因此使用 `void*` 类型来完成上述需求(需要输出指针变量的地址时都可以使用 `void*` 类型)。图 3.8 中的程序演示了如何输出 `char*` 类型的字符串及其地址，输出的地址是用十六进制格式表示。在 C++ 中，十六进制数以 `0x` 或 `0X` 打头，3.6.1 节、3.7.4 节、3.7.5 节和 3.7.7 节要详细介绍控制数值基数的方法。

```
1 // Fig. 3.8: fig11_08.cpp
2 // Printing the address stored in a char* variable
3 #include <iostream.h>
4 using namespace std;
5 int main()
6 {
7     char *string = "test";
8
9     cout << "Value of string is: " << string
10         << "\nValue of static cast< void *>( string ) is:"
11         << static_cast< void*>( string ) << endl;
12     return 0;
13 }
```

输出结果：

Value of string is:test

Value of staticcast <void*>(string)is:0x00416D50

图 3.8 输出 `char*` 类型变量的地址

3.3.4 用成员函数 `put` 输出字符和 `put` 函数的连续调用

用 `put` 成员函数用于输出一个字符，例如语句：

```
cout.put('A');
```

将字符 A 显示在屏幕上。

也可以像下面那样在一条语句中连续调用 `put` 函数：

```
cout.put('A').put("\n");
```

该语句在输出字符 A 后输出一个换行符。和 `<<` 一样，上述语句中圆点运算符(`.`)从左向右结合，`put`

成员函数返回调用 `put` 的对象的引用。还可以用 ASCII 码值表达式调用 `put` 函数，语句 `cout.put(65)` 也

输出字符 A。

3.4 输入流

下面我们要讨论流的输入，这是用流读取运算符(即重载的运算符<>)实现的。流读取运算符通常会跳过输入流中的空格、tab 键、换行符等等的空白字符，稍后将介绍如何改变这种行为。当遇到输入流中的文件结束符时，流读取运算符返回 0(false);否则，流读取运算符返回对调用该运算符的对象的引用。每个输入流都包含一组用于控制流状态(即格式化、出错状态设置等)的状态位。当输入类型有错时，流读取运算符就会设置输入流的 failbit 状态位；如果操作失败则设置 badbit 状态位，后面会介绍如何在 I/O 操作后测试这些状态位。3.7 节和 3.8 节详细讨论了流的状态位。

3.4.1 流读取运算符

图 3.9 中的范例程序用 cin 对象和重载的流读取运算符<>读取了两个整数。注意流读运算符可以连续使用。

```
1 // Fig. 3.9: figll_09.cpp
2 // Calculating the sum of two integers input from the keyboard
3 // with the cin oct and the stream-extraction operator.
4 #include <iostream>
5 using namespace std;
6 int main()
7 {
8     int x, y;
9
10    cout << "Enter two integers: ";
11    cin >> x >> y;
12    cout << "Sum of" << x << "and " << y << "is:"
13         << ( x + y ) << endl;
14
15    return 0;
16 }
```

输出结果:

```
Enter two integers: 30 92
Sum of 30 and 92 is: 122
```

图 3.9 计算用 cin 和流读取运算符从键盘输入的两个整数值的和

如果运算符<>和<<的优先级相对较高就会出现问题。例如，在图 3.10 的程序中，如果条件表达式没有用括号括起来，程序就得不到正确的编译(学员可以试一下)。

```
1 // Fig. 3.10: figlll0.cpp
2 // Avoiding a precedence problem between the stream-insertion
3 // operator and the conditional operator.
```

```

4 // Need parentheses around the conditional expression.
5 #include <iostream.h>
6 using namespace std;
7 int main()
8
9     int x, y;
10
11     cout << "Enter two integers: ";
12     cin >> x >> y;
13     cout << x << ( x == y ? "is" : "is not" )
14         << "equal to " << y << endl;
15
16     return 0;
17 }

```

输出结果:

```

Enter two integers: 7 5
7 is not equal to 5

```

```

Enter two integers: 8 8
8 is equal to 8

```

图 3.10 避免在流插入运算符和条件运算符之间出现优先级错误

我们通常在 while 循环结构的首部用流读取运算符输入一系列值。当遇到文件结束符时，读取运算符返回 0(false)。图 3.11 中的程序用于查找某次考试的最高成绩。假定事先不知道有多少个考试成绩，并且用户会输入表示成绩输入完毕的文件结束符。当用户输入文件结束符时，while 循环结构中的条件(cin>>grade)将变为 0(即 false)。

在图 3.11 中，cin>>grade 可以作为条件，因为基类 ios(继承 istream 的类)提供一个重载的强制类型转换运算符，将流变成 void*类型的指针。如果读取数值时发生错误或遇到文件结束符，则指针值为 0。编译器能够隐式使用 void*类型的强制转换运算符。

```

1 // Fig. 3.11: figll_ll.cpp
2 // Stream-extraction operator returning false on end-of-file.
3 #include <iostream.h>
4 using namespace std;
5 int main()
6 {
7     int grade, highestGrade = -1;
8
9     cout << "Enter grade (enter end-of-file to end): ";
10    while ( cin >> grade){
11        if ( grade > highestGrade )

```



```

12         highestGrade = grade;
13
14     cout << "Enter grade (enter end-of-file to end): ";
15 }
16
17 cout << "\n\nHighest grade is: " << highestGrade << endl;
18 return 0;
19 }

```

输出结果:

```

Enter grade (enter end-of-file to end): 67
Enter grade (enter end-of-file to end): 87
Enter grade (enter end of file to end): 73
Enter grade (enter end-of-file to end): 95
Enter grade (enter end-of-file to end): 34
Enter grade (enter end-of-file to end): 99
Entergrade (enter end-of-file to end): ^ z
Heighest grade is: 99

```

图 3.11 流读取运算符在遇到文件结束符时返回 false

3.4.2 成员函数 get 和 getline

不带参数的 get 函数从指定的输入流中读取(输入)一个字符(包括空白字符),并返回该字符作为函数调用的值;当遇到输入流中的文件结束符时,该版本的 get 函数返回 EOF。

图 3.12 中的程序把成员函数 eof 和 get 用于输入流 cin,把成员函数 put 用于输出流 cout。程序首先输出了 cin.eof()的值,输出值为 0(false)表明没有在 cin 中遇到文件结束符。然后,程序让用户键入以文件结束符结尾的一行文本(在 IBMPC 兼容系统中,文件结束符用<ctrl>-z 表示;在 UNIX 和 Macintosh 系统中,文件结束符用<ctrl>-d 表示)。程序逐个读取所输入的每个字符,并调用成员函数 put 将所读取的内容送往输出流 cout。当遇到文件结束符时,while 循环终止,输出 cin.eof()的值,输出值为 1(true)表示已接收到了 cin 中的文件结束符。注意,程序中使用了类 istream 中不带参数

的 get 成员函数,其返回值是所输入的字符。

```

1 // Fig. 3.12: figll_12.cpp
2 // Using member functions get, put and eof.
3 #include <iostream>
4 using namespace std;
5 int main()
6 {
7     char c;
8
9     cout << "Before input, cin.eof() is " << cin.eof()
10         << "\nEnter a sentence followed by end-of-file:\n";

```

```

11
12 while ( ( c = cin.get() ) != EOF )
13     cout.put( c );
14
15 cout << "\nEOF in this system is: " << c;
16 cout << "\nAfter input, cin.eof() is " << cin.eof() << endl;
17 return 0;
18 }

```

输出结果:

Before input, cin.eof() is 0

Enter a sentence followed by end-of-file:

Testing the get and put member functions^z

Testing the get and put member functions

EOF in this system is: -1

After input cin.eof() is 1

图 3.12 使用成员函数 get.put 和 eof

带一个字符型参数的 `get` 成员函数自动读取输入流中的下一个字符(包括空白字符)。当遇到文件结束符时, 该版本的函数返回 0, 否则返回对 `istream` 对象的引用, 并用该引用再次调用 `get` 函数。

带有三个参数的 `get` 成员函数的参数分别是接收字符的字符数组、字符数组的大小和分隔符(默认值为'\n')。函数或者在读取比指定的最大字符数少一个字符后结束, 或者在遇到分隔符时结束。为使字符数组(被程序用作缓冲区)中的输入字符串能够结束, 空字符会被插入到字符数组中。函数不把分隔符放到字符数组中, 但是分隔符仍然会保留在输入流中。图 3.13 的程序比较了 `cin`(与流读取运算符一起使用)和 `cin.get` 的输入结果。注意调用 `cin.get` 时未指定分隔符, 因此用默认的'\n'。

```

1 // Fig. 3.13: fig13.cpp
2 // contrasting input of a string with cin and cin.get.
3 #include <iostream.h>
4 using namespace std;
5 int main()
6 {
7     const int SIZE = 80;
8     char buffer1[ SIZE ], buffer2[ SIZE ];
9
10    cout << "Enter a sentence:\n";
11    cin >> buffer1;
12    cout << "\nThe string read with cin was:\n"
13         << buffer1 << "\n";
14
15    cin.get( buffer2, SIZE );

```

```

16  cout << "The string read with cin.get was:\n"
17      << buffer2 << endl;
18
19  return 0;
20 }

```

输出结果:

Enter a sentence:

Contrasting string input with cin and cin.get

The string read with cin was:

Contrasting

The string read with cin.get was:

string input with cin and cin.get

图 3.13 比较用 cin 和 cin.get 输入的字符串的结果

成员函数 `getline` 与带三个参数的 `get` 函数类似，它读取一行信息到字符数组中，然后插入一个空字符。所不同的是，`getline` 要去除输入流中的分隔符(即读取字符并删除它)，但是不把它存放在字符数组中。图 3.14 中的程序演示了用 `getline` 输入一行文本。

```

1 // Fig. 3.14: figll_14.cpp
2 // Character input with member function getline.
3 #include <iostream.h>
4
5 int main()
6 {
7     const SIZE = 80;
8     char buffe[ SIZE ];
9
10    cout << "Enter a sentence:\n";
11    cin.getline( buffer, SIZE );
12
13    cout << "\nThe sentence entered is:\n" << buffer << endl;
14    return 0;
15 }

```

输出结果:

Enter a sentence:

Using the getline member function

The sentence entered is:

Using the getline member function

图 3.14 用成员函数 `getline` 输入字符

3.5 成员函数 read、gcount 和 write 的无格式输入/输出

调用成员函数 read、write 可实现无格式输入/输出。这两个函数分别把一定量的字节写入字符数组和从字符数组中输出。这些字节都是未经任何格式化的，仅仅是以原始数据形式输入或输出。

例如：

```
char buffe[] = "HAPPY BIRTHDAY";  
cout.write(buffer, 10 );
```

输出 buffet 的 10 个字节(包括终止 cout 和<<输出的空字符)。因为字符串就是第一个字符的地址，所以函数调用：

```
cout.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10);
```

显示了字母表中的前 10 个字母。

成员函数 read 把指定个数的字符输入到字符数组中。如果读取的字符个数少于指定的数目，可以设置标志位 failbit(见 3.8 节)。

成员函数 gcount 统计最后输入的字符个数。

图 3.15 中的程序演示了类 istream 中的成员函数 read 和 gcount，以及类 ostream 中的成员函数 write。程序首先用函数 read 向字符数组 buffer 中输入 20 个字符(输入序列比字符数组长)，然后用函数 gcount 统计所输入的字符个数，最后用函数 write 输出 buffer 中的字符。

```
1 // Fig. 3.15: fig11_15.cpp  
2 // Unformatted I/O with read, gcount and write.  
3 #include <iostream.h>  
4 using namespace std;  
5 int main()  
6 {  
7     const int SIZE = 80;  
8     char buffer[ SIZE ];  
9  
10    cout << "Enter a sentence:\n";  
11    cin.read( buffer, 20 );  
12    cout << "\nThe sentence entered was:\n";  
13    cout.write( buffer, cin.gcount() );  
14    cout << endl;  
15    return 0;  
16 }
```

输出结果：

Enter a sentence:

Using the read, write, and gcount member functions

The sentence entered was:

Using the read,write

图 3.15 成员函数 read,gcount 和 write 的无格式 I/O

3.6 流操纵算子

C++提供了大量的用于执行格式化输入/输出的流操纵算子。流操纵算子提供了许多功能，如设置域宽、设置精度、设置和清除格式化标志、设置域填充字符、刷新流、在输出流中插入换行符并刷新该流、在输出流中插入空字符、跳过输入流中的空白字符等等。下面几节要介绍这些特征。

3.6.1 整数流的基数：流操纵算子 dec、oct、hex 和 setbase

整数通常被解释为十进制(基数为 10)整数。如下方法可改变流中整数的基数：插人流操纵算子 hex 可设置十六进制基数(基数为 16)、插人流操纵算子 oct 可设置八进制基数(基数为 8)、插人流操纵算子 dec 可恢复十进制基数。

也可以用流操纵算子 setbase 来改变基数，流操纵算子 setbase 带有一个整数参数 10、8 或 16。因为流操纵算子 setbase 是带有参数的，所以也称之为参数化的流操纵算子。使用 setbase 或其他任何参数化的操纵算子都必须在程序中包含头文件 iomanip.h。如果不明确地改变流的基数，流的基数是不变的。图 3.16 中的程序示范了流操纵算子 hex、oct、dec 和 setbase 的用法。

```
1 // Fig. 3.16: fig11_16.cpp
2 // Using hex, oct, dec and setbase stream manipulators.
3 #include <iostream>
4 #include <iomanip.h>
5 using namespace std;
6 int main()
7 {
8     int n;
9
10    cout << "Enter a decimal number: ";
11    cin >> n;
12
13    cout << n << "in hexadecimal is:"
14         << hex << n << '\n'
15         << dec << n << "in octal is:"
16         << oct << n << '\n'
17         << setbase( 10 ) << n << " in decimal is:"
18         << n << endl;
19
20    return 0;
21 }
```

输出结果：

Enter a decimal number: 20

20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20

图 3.16 使用流操纵算子 hex、oct、dec 和 setbase

3.6.2 设置浮点数精度(precision、setprecision)

在 C++中可以人为控制浮点数的精度，也就是说可以用流操纵算子 setprecision 或成员函数 precision 控制小数点后面的位数。设置了精度以后，该精度对之后所有的输出操作都有效，直到下一次设置精度为止。无参数的成员函数 precision 返回当前设置的精度。图 3.17 中的程序用成员函数 precision 和流操纵算子 setprecision 打印出了 2 的平方根表，输出结果的精度从 0 连续变化到 9。

```
1 // Fig. 3.17: fig11_17.cpp
2 // Controlling precision of floating-point values
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <math.h>
6 using namespace std;
7 int main()
8 {
9     double root2 = sqrt( 2.0 );
10    int places;
11
12    cout << setiosflags(ios::fixed)
13         << "Square root of 2 with precisions 0-9.\n"
14         << "Precision set by the"
15         << "precision member function:" << endl;
16
17    for ( places = 0; places <= 9; places++ ) {
18        cout.precision( places );
19        cout << root2 << '\n';
20    }
21
22    cout << "\nPrecision set by the"
23         << "setprecision manipulator:\n";
24
25    for ( places = 0; places <= 9; places++ )
26        cout << setprecision( places ) << root2 << '\n';
27
28    return 0;
29 }
```

输出结果:

Square root of 2 with pprecisions 0-9.

Precision set by the precision member function:

```
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Precision set by the setprecision manipulator:

```
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

图 3.17 控制浮点数的精度

3.6.3 设置域宽(setw、width)

成员函数 `ios.width` 设置当前的域宽(即输入输出的字符数)并返回以前设置的域宽。如果显示数据所需的宽度比设置的域宽小,空位用填充字符填充。如果显示数据所需的宽度比设置的域宽大,显示数据并不会被截断,系统会输出所有位。

图 3.18 中的程序示范了成员函数 `width` 的输入和输出操作。注意,输入操作提取字符串的最大宽度比定义的域宽小 1,这是因为在输入的字符串后面必须加上一个空字符。当遇到非前导空白字符时,流读取操作就会终止。流操纵算子 `setw` 也可以用来设置域宽。注意:提示用户输入时,用户应输入一行文本并按 Enter 键加上文件结束符(<ctrl>-z 对于 IDMPPC 兼容系统; <ctrl>-d 对于 UNIX 与 Macintosh 系统)。

```
1 // fig11_18.cpp
2 // Demonstrating the width member function
3 #include <iostream>
4 using namespace std;
```

```

5 int main()
6 {
7     int w = 4;
8     char string[ 10 ];
9
10    cout << "Enter a sentence:\n";
11    cin.width( 5 );
12
13    while (cin >> string ) {
14        cout.width( w++ );
15        cout << string << endl;
16        cin.width( 5 );
17    }
18
19    return 0;
20 }

```

输出结果:

Enter a sentence:

This is a test of the width member function

This

```

    is
      a
    test
      of
    the
    widt
      h
    memb
      er
    func
    tion

```

图 3.18 演示成员函数 width

3.6.4 用户自定义的流操纵算子

用户可以建立自己的流操纵算子。图 3.19 中的程序说明了如何建立和使用新的流操纵算子 bell、ret、tab 和 endl。用户还可以建立自己的带参数的流操纵算子,这方面内容可参看系统的手册。

```

1 /* Fig. 3.19: fig11_19.cpp
2 // Creating and testing user-defined, nonparameterized
3 // stream manipulators.
4 #include <iostream.h>

```



```

5 using namespace std;
6 // bell manipulator (using escape sequence \a)
7 ostream& bell( ostream& output ) { return output << '\a'; }
8
9 // ret manipulator (using escape sequence \r)
10 ostream& ret( ostream& output ) ( return output << '\r'; }
11
12 // tab manipulator (using escape sequence \t)
13 ostream& tab( ostream& output ) { return output << '\t'; ]
14
15 // endLine manipulator (using escape sequence \n
16 // and the flush member function)
17 ostream& endLine( ostream& output )
18 {
19     return output << '\n' << flush;
20 }
21
22 int main()
23 {
24     cout << "Testing the tab manipulator:" << endLine
25         << 'a' << tab << 'b' << tab << 'c' << endLine
26         << "Testing the ret and bell manipulators:"
27         << endLine << ".....  ";
28     cout << bell;
29     cout << ret << ..... << endLine;
30     return 0;
31 }

```

输出结果:

Testing the tab manipulator:

a b c

Testing the ret and bell manipulators:

-----.....

图 3.19 建立并测试用户自定义的、无参数的流操作算子

3.7 流格式状态

各种格式标志指定了在 I/O 流中执行的格式类型。成员函数 `serf`、`unserf` 和 `flag` 控制标志的设置。

3.7.1 格式状态标志

图 3.20 中的格式状态标志在类 `ios` 中被定义为枚举值，留到下几节解释。

虽然成员函数 `flags`、`serf` 和 `unseff` 可以控制这些标志，但是许多 C++ 程序员更喜欢使用流操纵算子(见 3.7.8 节)。程序员可以用按位或操作(`|`)把各种标志选项结合在一个 `long` 类型的值中(见图 3.23)。调用成员函数 `flags` 并指定要进行或操作的标志选项就可以在流中设置这些标志，并返回一个包含以前标志选项的 `long` 类型的值。返回值通常要保存起来，以使用保存值调用 `flags` 来恢复以前的流选项。

`flags` 函数必须指定一个代表所有标志设置的值。而带有一个参数的函数 `setf` 可指定一个或多个要进行或操作的标志，并把它们与现存的标志设置相“或”，形成新的格式状态。

参数化的流操纵算子 `setiosflags` 与成员函数 `serf` 执行同样的功能。流操纵算子 `resetiosflags` 与成员函数 `unsetf` 执行同样的功能。不论使用哪一种流操纵算子，在程序中都必须包含头文件 `iomanip.h`。

`skipws` 标志指示 `>>` 跳过输入流中的空白字符。`>>` 的默认行为是跳过空白字符，调用 `unsetf(ios::skipws)` 可改变默认行为。流操纵算子 `ws` 也可用于指定跳过流中的空白字符。

格式状态	说明
<code>ios::skipws</code>	跳过输入流中的空白字符
<code>ios::left</code>	在域中左对齐输出，必要时在右边显示填充字符
<code>ios::right</code>	在域中右对齐输出，必要时在左边显示填充字符
<code>ios::internal</code>	表示数字的符号应在域中左对齐，而数字值应在域中右对齐(即在符号和数字之间填充字符)
<code>ios::dec</code>	指定整数应作为十进制(基数 10)值
<code>ios::oct</code>	指定整数应作为八进制(基数 8)值
<code>ios::hex</code>	指定整数应作为十六进制(基数 16)值
<code>ios::showbase</code>	指定在数字前面输出进制(0 表示八进制，0x 或 0X 表示十六进制)
<code>ios::showpoint</code>	指定浮点数输出时应带小数点。这通常和 <code>ios::fixed</code> 一起使用保证小数点后面有一定位数
<code>ios::uppercase</code>	指定表示十六进制 Rx 应为大写，表示浮点科学计数法的 e 应为大写
<code>ios::showpos</code>	指定正数和负数前面分别加上正号或一号
<code>ios::scientific</code>	指事实上浮点数输出采用科学计数法
<code>ios::fixed</code>	指事实上浮点数输出采用定点符号，保证小数点后面有一定位数

图 3.20 格式状态标志

3.7.2 尾数零和十进制小数点(`ios::showpoint`)

设置 `showpoint` 标志是为了强制输出浮点数的小数点和尾数零。若没有设置 `showpoint`，浮点数 79.0 将被打印为 79，否则打印为 79.000000(或由当前精度指定的尾数零的个数)。图 3.21 中

的程序用成员函数 `setf` 设置 `showpoint` 标志，从而控制了浮点数的尾数零和小数点的输出。

```
1 // Fig. 3.21: fig11_21.cpp
2 // Controlling the printing of trailing zeros and decimal
3 // points for floating-point values.
4 #include <iostream.h>
5 #include <iomanip.h>
6 #include <math.h>
7
8 int main()
9 {
10     cout << "Before setting the ios::showpoint flag\n"
11         << "9.9900 prints as: "<< 9.9900
12         << "\n9.9000 prints as: "<< 9.9000
13         << "\n9.0000 prints as: "<< 9.0000
14         << "\n\nAfter setting the ios::showpoint flag\n";
15     cout.setf( ios::showpoint );
16     cout << "9.9900 prints as: "<< 9.9900
17         << "\n9.9000 prints as: "<< 9.9000
18         << "\n9.0000 prints as: "<< 9.0000 << endl;
19     return 0;
20 }
```

输出结果：

Before setting the ios::showpoint flag

9.9900 prints as: 9.99

9.9000 prints as: 9.9

9.9000 prints as: 9

After setting the ios::showpoint flag

9.9900 prints as: 9.99000

9.9000 prints as: 9.90000

9.0000 prints as: 9.00000

图 3.21 控制浮点数的尾数零和小数点的输出

3.7.3 对齐(ios::left、ios::right、ios::internal)

`left` 标志可以使输出域左对齐并把填充字符放在输出数据的右边，`right` 标志可以使输出域右对齐并把填充字符放在输出数据的左边。填充的字符由成员函数 `fill` 或参数化的流操纵算子 `setfill` 指定。图 3.22 用流操纵算子 `setw`、`setiosflags`、`resetiosflags` 以及成员函数 `serf` 和 `unsetf` 控制整数值在域宽内左对齐和右对齐。

```
1 // Fig. 3.22: fig11_22.cpp
2 // Left-justification and right-justification.
```

```

3 #include <iostream.h>
4 #include <iomanip.h>
5 int main()
6 {
7     int x = 12345;
8
9     cout << "Default is right justified:\n"
10         << setw(10) << x << "\n\nUSING MEMBER FUNCTIONS"
11         << "\nUse setf to set ios::left:\n" << setw(10);
12
13     cout.setf( ios::left, ios::adjustfield );
14     cout << x << "\nUse unsetf to restore default:\n";
15     cout.unsetf( ios::left );
16     cout << setw( 10 ) << x
17         << "\n\nUSING PARAMETERIZED STREAM MANIPULATORS"
18         << "\nUse setiosflags to set ios::left:\n"
19         << setw( 10 ) << setiosflags( ios::left ) << x
20         << "\nUse resetiosflags to restore default:\n"
21         << setw( 10 ) << resetiosflags( ios::left )
22         << x << endl;
23     return 0;
24 }

```

输出结果:

Default is right justified:

12345

USING MEMBER FUNCTIONS

Use setf to set ios::left:

12345

Use unsetf to restore default:

12345

USING PARAMETERIZED STREAM MANIPULATORS

Use setiosflags to set ios::left:

12345

Use resetiosflags to restore default:

12345

图 3.22 左对齐和右对齐

`internal` 标志指示将一个数的符号位(设置了 `ios::showbase` 标志时为基数, 见 3.7. 5 节)在域宽内左对齐, 数值右对齐, 中间的空白由填充字符填充。`left`、`right` 和 `internal` 标志包含在静态数据成员 `ios::adjustfield` 中。在设置 `left`、`right` 和 `internal` 对齐标志时, `setf` 的第二个参数必须是 `ios::adjustfield`, 这样才能使 `setf` 只设置其中一个标志(这三个标志是互斥的)。图 3.23 中的程序用流操纵算子 `setiosflags` 和 `setw` 指定中间的空白。注意, `ios::showpos` 标志强制打印了加号。

```

1 // Fig. 3.23: figl1_23.cpp
2 // Printing an integer with internal spacing and
3 // forcing the plus sign.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 int main()
8 {
9     cout << setiosflags( ios::internal | ios::showpos )
10         << setw( 10 ) << 123 << endl;
11     return 0;
12 }

```

输出结果:

```
+    123
```

图 3.23 打印中间带空白的整数并强制输出加号

3.7.4 设置填充字符(fill、setfill)

成员函数 fill 设置用于使输出域对齐的填充字符，如果不特别指定，空格即为填充字符。fill 函数返回以前设置的填充字符。图 3.24 中的程序演示了使用成员函数 fill 和流操纵算子，setfill 来控制填充字符的设置和清除。

```

1 // Fig. 3.24: figl1_24.cpp
2 // Using the fill member function and the setfill
3 // manipulator to change the padding character for
4 // fields larger than the values being printed.
5 #include <iostream.h>
6 #include <iomanip.h>
7
8 int main()
9 {
10     int x = 10000;
11
12     cout << x << "printed as iht right and left justified\n"
13         << "and as hex with internal justification.\n"
14         << "Using the default pad character (space):\n";
15     cout.setf( ios::showbase );
16     cout << setw( 10 ) << x << "\n";
17     cout.setf( ios::left, ios::adjustfield );
18     cout << setw( 10 ) << x << "\n";
19     cout.setf( ios::internal, ios::adjustfield );
20     cout << setw( 10 ) << hex << x;

```

```

21
22  cout << "\n\nUsing various padding characters:\n";
23  cout.setf( ios::right, ios::adjustfield );
24  cout.fill( '*' );
25  cout << setw( 10 ) << dec << x << '\n';
26  cout.setf( ios::left, ios::adjustfield );
27  cout << setw( 10 ) << setfill( '%' ) << x << '\n';
28  cout.setf( ios::internal, ios::adjustfield );
29  cout << setw( 10 ) << setfill( '^' ) << hex << x << endl;
30  return 0;
31 }

```

输出结果:

10000 printed as int right and left justified
and as hex with internal justification.

Using the default pad character (space):

10000

10000

0x2710

Using various padding characters:

*****10000

10000%%%%%%%%

图 3.24 用 fill 和 setfill 为实际宽度小于域宽的数据改变填充字符

3.7.5 整数流的基数: (ios::dec、ios::oct、ios::hex、ios::showbase)

静态成员 ios::basefield(在 setf 中的用法与 ios::adjustfield 类似)包括 ios::oct、ios::hex 和 ios::dec 标志位, 这些标志位分别指定把整数作为八进制、十六进制和十进制值处理。如果没有设置这些位, 则流插入运算默认整数为十进制数, 流读取运算按整数提供的方式处理数据(即以零打头的整数按八进制数处理, 以 0x 或 0X 打头的按十六进制数处理, 其他所有整数都按十进制数处理)。一旦为流指定了特定的基数, 流中的所有整数都按该基数处理, 直到指定了新的基数或程序结束为止。

设置 showbase 标志可强制输出整数值的基数。十进制数以通常方式输出, 输出的八进制数以 0 打头, 输出的十六进制数以 0x 或 0X 打头(由 uppercase 标志决定是 0x 还是 0X, 见 3.7.7 节)。图 3.25 中的程序用 showbase 标志强制整数按十进制、八进制和十六进制格式打印。

```

1 // Fig. 3.25: figl1_25.cpp
2 // Using the ios::showbase flag
3 #include <iostream.h>
4 #include <iomanip.h>
5

```

```

6 int main()
7 {
8     int x = 100;
9
10    cout << setiosflags( ios::showbase )
11        << "Printing integers preceded by their base:\n"
12        << x << '\n'
13        << oct << x << '\n'
14        << hex << x << endl;
15    return 0;
16 }

```

输入结果:

Printing integers preceded by their base:

100

0144

0x64

图 3.25 使用 ios::showbase 标志

3.7.6 浮点数和科学记数法(ios::scientific、ios::fixed)

ios::scientific 和 ios::fixed 标志包含在静态数据成员 ios::floatfield 中(在 setf 中的用法与 ios::adjustfield 和 ios::basefield 类似)。这些标志用于控制浮点数的输出格式。设置 scientific 标志使浮点数按科学记数法输出, 设置 fixed 标志使浮点数按照定点格式输出, 即显示出小数点, 小数点后边

有指定的位数(由成员函数 precision 指定)。若没有这些设置, 则浮点数的值决定输出格式。

调用 cout. setf(0, ios::floatfield)恢复输出浮点数的系统默认格式。图 3.26 中的程序示范了以定点格式和科学记数法显示的浮点数。

```

1 // Fig. 3.26: flg11_26.cpp
2 // Displaying floating-point values in system default,
3 // scientific, and fixed formats.
4 #include <iostream.h>
5
6 int main()
7 {
8     double x = .001239567, y = 1.946e9;
9
10    cout << "Displayed in default format:\n"
11        << x << '\t' << y << '\n';
12    cout.setf( ios::scientific, ios::floatfield );
13    cout << "Displayed in scientific format:\n"
14        << x << '\t' << y << '\n';

```

```

15  cout.unsetf( ios::scientific );
16  cout << "Displayed in default format after unsetf:\n"
17      << x << ',\t' << y << ',\n';
18  cout.setf( ios::fixed, ios::floatfield );
19  cout << "Displayed in fixed format:\n"
20      << x << '\t' << y << endl;
21  return 0;
22 }

```

输出结果:

Displayed in default format:

0.00123457 1.946e+009

Displayed in scientific format:

1.234567e-003 1.946000e+009

Displayed in default format after unsetf:

0.00123457 1.946e+009

Displayed in fixed format:

0.001235 1946000000.000000

图 3.26 以系统默认格式、科学计数法和定点格式显示浮点数

3.7.7 大/小写控制(ios::uppercase)

设置 ios::uppercase 标志用来使大写的 X 和 E 分别同十六进制整数和以科学记数法表示的浮点数一起输出(见图 3.27)一旦设置 ios::uppercase 标志, 十六进制数中的所有字母都转换成大写。

```

1 // Fig. 3.27: fig11_27.cpp
2 // Using the ios::uppercase flag
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     cout << setiosflags( ios::uppercase )
9         << "Printing uppercase letters in scientific\n"
10         << "notation exponents and hexadecimal values:\n"
11         << 4.345e10 << '\n' << hex << 123456789 << endl;
12     return 0;
13 }

```

输出结果:

Printing uppercase letters in scientific

notation exponents and hexadecimal values:

4.395E+010

图 3.27 使用 ios::uppercase 标志

3.7.8 设置及清除格式标志(flags、setiosflags、resetiosflags)

无参数的成员函数 `flags` 只返回格式标志的当前设置(long 类型的值)。带一个 long 类型参数的成员函数 `flags` 按参数指定的格式设置标志, 返回以前的标志设置。`flags` 的参数中未指定的任何格式都会被清除。图 3.18 的程序用成员函数 `flags` 设置新的格式状态和保存以前的格式状态, 然后恢复原来的格式设置。

```

1 // Fig. 3.28: fig11_28.cpp
2 // Demonstrating the flags member function.
3 #include <iostream.h>
4
5 int main()
6 {
7     int i = 1000;
8     double d = 0.0947625;
9
10    cout << "The value of the flags variable is:"
11        << cout.flags()
12        << "\nPrint int and double in original format:\n"
13        << i << '\t' << d << "\n\n";
14    long originalFormat =
15        cout.flags( ios::oct | ios::scientific );
16    cout << "The value of the flags variable is:"
17        << cout.flags()
18        << "\nPrint int and double in a new format\n"
19        << "specified using the flags member function:\n"
20        << i << '\t' << d << "\n\n";
21    cout.flags( originalFormat );
22    cout << "The value of the flags variable is:"
23        << cout.flags()
24        << "\nPrint values in original format again:\n"
25        << i << '\t' << d << endl;
26    return 0;
27 }

```

输出结果:

```

The value of the flags variable is: 0
Print int and double in original format:
1000    0.0947628

```

The value of the flags variable is: 4040
Print iht and double in a new format
Specified using the flags member function:
1750 9.476280e-002

The value of the flags variable is: 0
Print values in original format again:
1000 0.0947628

图 3.28 演示成员函数 flags

成员函数 `serf` 设置参数所指定的格式标志，并返回 `long` 类型的标志设置值，例如：

```
long previousFlagSettings=  
cout.setf(ios::showpoint | ios::showpos);  
带两个 long 型参数的 setf 成员函数。例如：  
cout.setf(ios::left, ios::adjustfield);
```

首先清除 `ios::adjustfield` 位，然后设置 `ios::left` 标志。该版本的 `setf` 用于与 `ios::basefield` (用 `ios::dec`、`ios::oct` 和 `ios::hex` 表示)、`ios::floatfield` (用 `ios::scientific` 和 `ios::fixed` 表示) 和 `ios::adjustfield` (用 `ios::left`、`ios::right` 和 `ios::internal` 表示) 相关的位段。

成员函数 `unsetf` 清除指定的标志并返回清除前的标志值。

3.8 流错误状态

可以用类 `ios` 中的位测试流的状态。类 `ios` 是输入/输出类 `istream`、`ostream` 和 `iostream` 的基类。

当遇到文件结束符时，输入流中自动设置 `eofbit`。可以在程序中使用成员函数 `eof` 确定是否已经到达文件尾。如果 `cin` 遇到了文件结束符，那么函数调用：

```
cin.eof()
```

返回 `true`，否则返回 `false`。

当流中发生格式错误时，虽然会设置 `failbit`，但是字符并未丢失。成员函数 `fail` 判断流操作是否失败，这种错误通常可修复。

当发生导致数据丢失的错误时，设置 `badbit`。成员函数 `bad` 判断流操作是否失败，这种严重错误通常不可修复。

如果 `eofbit`、`failbit` 或 `badbit` 都没有设置，则设置 `goodbit`。

如果函数 `bad`、`fail` 和 `eof` 全都返回 `false`，则成员函数 `good` 返回 `true`。程序中应该只对“好”的流进行 I/O 操作。

成员函数 `rdstate` 返回流的错误状态。例如，函数调用 `cout.rdstate` 将返回流的状态，随后可以用一条 `switch` 语句测试该状态，测试工作包括检查 `ios::eofbit`、`ios::badbit`、`ios::failbit` 和 `ios::goodbit`。

测试流状态的较好方法是使用成员函数 `eof`、`bad`、`fail` 和 `good`，使用这些函数不需要程序员熟知特定的状态位。

成员函数 `clear` 通常用于把一个流的状态恢复为“好”，从而可以对该流继续执行 I/O 操作。由于 `clear` 的默认参数为 `ios::goodbit`，所以下列语句：

```
cin. clear();
```

清除 cin，并为流设置 goodbit。下列语句：

```
cin.clear(ios::failbit)
```

实际上给流设置了 failbit。在用自定义类型对 cin 执行输入操作或遇到问题时，用户可能需要这么做。clear 这个名字用在这里似乎并不合适，但规定就是如此。

图 3.29 中的程序演示了成员函数 rdstate、eof、fail、bad、good 和 clear 的使用。

只要 badbit 和 failbit 中有一个被置位，成员函数 operator! 就返回 true。只要 badbit 和 failbit 中有一个被置位，成员函数 operator void* 就返回 false。这些函数可用于文件处理过程中测试选择结构或循环结构条件的 true/false 情况。

```
1 // Fig. 3.29: fig11_29.cpp
2 // Testing error states.
3 #include <iostream.h>
4
5 int main()
6 {
7     int x;
8     cout << "Before a bad input operation:"
9         << "\ncin.rdstate(): " << cin.rdstate()
10        << "\n    cin.eof(): " << cin.eof()
11        << "\n    cin.fail(): " << cin.fail()
12        << "\n    cin.bad(): " << cin.bad()
13        << "\n    cin.good(): " << cin.good()
14        << "\n\nExpects an integer, but enter a character: ";
15    cin >> x;
16
17    cout << "\nEnter a bad input operation:"
18        << "\ncin.rdstate(): " << cin.rdstate()
19        << "\n    cin.eof(): " << cin.eof()
20        << "\n    cin.fail(): " << cin.fail()
21        << "\n    cin.bad(): " << cin.bad()
22        << "\n    cin.geed()  " << cin.good() << "\n\n";
23
24    cin.clear();
25
26    cout << "After cin.clear()"
27        << "\ncin.fail(): " << cin.fail()
28        << "\ncin.good(): " << cin.good() << endl;
29    return 0;
30 }
```

输出结果：

Before a bad input operation:

cin.rdstate(): 0

```
cin.eof(): 0
cin.fail() 0
tin.bad(): 0
cin.good(): 0
```

Expects an integer, but enter a character: A

After a bad input operation:

```
cin.eof(): 0
cin.fail(): 2
cin.bad(): 0
cin.good(): 0
```

After cin.clear()

```
cin.fail(): 0
cin.good(): 1
```

图 3.29 测试错误状态

第 4 章 文件处理

4.1 简介

存储在变量和数组中的数据是临时的，这些数据在程序运行结束后都会消失。文件用来永久地保存大量的数据。计算机把文件存储在二级存储设备中(特别是磁盘存储设备)。本章要讨论怎样用 C++ 程序建立、更新和处理数据文件(包括顺序存储文件和随机访问文件)。我们要比较格式化与“原始数据”文件处理。后面将介绍从 `string` 而不是从文件输入和输出数据。

4.2 文件和流

C++ 语言把每一个文件都看成一个有序的字节流(见图 4.2)，每一个文件或者以文件结束符(end-of-file marker)结束，或者在特定的字节号处结束(结束文件的特定的字节号记录在由系统维护和管理的数据结构中)。当打开一个文件时，该文件就和某个流关联起来。第 11 章曾介绍过 `cin`、`cout`、`cerr` 和 `clog` 这 4 个对象会自动生成。与这些对象相关联的流提供程序与特定文件或设备之间的通信通道。例如，`cin` 对象(标准输入流对象)使程序能从键盘输入数据，`cout` 对象(标准输出流对象)使程序能向屏幕输出数据，`cerr` 和 `clog` 对象(标准错误流对象)使程序能向屏幕输出错误消息。

图 4.2 C++ 把文件看成 n 个字节

要在 C++ 中进行文件处理，就要包括头文件 `<iostream.h>` 和 `<fstream.h>`。`<fstream.h>` 头文件包括流类 `ifstream`(从文件输入)、`ofstream`(向文件输出)和 `fstream`(从文件输入，输出)的定义。生成这些流类的对象即可打开文件。这些流类分别从 `istream`、`ostream` 和 `iostream` 类派生(即继承它们的功能)。这样，第 2 章“C++ 输入，输出流”中介绍的成员函数、运算符和流操纵算子也可用于文件流。

4.3 建立并写入文件

因为 C++ 把文件看着是无结构的字节流，所以记录等等的说法在 C++ 文件中是不存在的。为此，程序员必须提供满足特定应用程序要求的文件结构。下例说明了程序员是怎样给文件强加一个记录结构。先列出程序，然后再分析细节。

图 4.4 中的程序建立了一个简单的访问文件，该文件可用在应收账款目管理系统中跟踪公司借贷客户的欠款数目。程序能够获取每一个客户的账号、客户名和对客户的结算额。一个客户的数

据就构成了该客户的记录。账号在应用程序中用作记录关键字，文件按账号顺序建立和维护。范例程序假定用户是按账号顺序键入记录的(为了让用户按任意顺序键入记录，完善的应收账款目管理系统应该具备排序能力)。然后把键入的记录保存并写入文件。

```
1 // Fig. 4.4; fig14_04.cpp
2 // Create a sequential file
3 #include <iostream.h>
4 #include <fstream.h>
5 #include <stdlib.h>
6 using namespace std;
7 int main()
8 {
9     // ofstream constructor opens file
10    ofstream fout( "clients.dat");
11
12    if ( ! fout ) { // overloaded ! operator
13        cerr << "File could not be opened" << endl;
14        exit( 1 ); // prototype in stdlib.h
15    }
16
17    cout << "Enter the account, name, and balance.\n"
18         << "Enter end-of-file to end input.\n? ";
19
20    int account;
21    char name[ 30 ];
22    float balance;
23
24    while (cin >> account >> name >> balance ) {
25        fout<< account << ' ' << name<< ' ' << balance << '\n';
26
27        cout << "? ";
28    }
29
30    return 0; // ofstream destructor closes file
31 }
```

输出结果：

Enter the account, name, and balance.

Enter end-of-file to end input.

? 100 Jones 24.98

? 200 Doe 345.67

? 300 White 0

? 400 Stone -42.16

? 500 Rich 224.62

? ^z

图 4.4 建立文件

现在我们来研究这个程序。前面曾介绍过，文件通过建立 `ifstream`、`ofstream` 或 `fstream` 流类的对象而打开。图 4.4 中，要打开文件以便输出，因此生成 `ofstream` 对象。向对象构造函数传入两个参数——文件名和文件打开方式。对于 `ostream` 对象，文件打开方式可以是 `ios::out`(将数据输出到文件)或 `ios::app`(将数据添加到文件末尾，而不修改文件中现有的数据)。现有文件用 `ios::out` 打开时会截尾，即文件中的所有数据均删除。如果指定文件还不存在，则用该文件名生成这个文件。下列声明(第 10 行)：

```
ofstream fout ("clients.dat");
```

生成 `ofstream` 对象 `fout`，与打开输出的文件 `clients.dat` 相关联。参数 `"clients.dat"` 和 `ios::out` 传入 `ofstream` 构造函数，该函数打开文件，从而建立与文件的通信线路。默认情况下，打开 `ofstream` 对象以便输出，因此下列语句：

```
ofstream fout ("clients.dat");
```

也可以打开 `clients.dat` 进行输出。图 14.5 列出了文件打开方式。

也可以生成 `ofstream` 对象而不打开特定文件，可以在后面再将文件与对象相连接。例如，下列声明：

```
ofstream fout;
```

生成以 `ofstream` 对象 `fout`。`ofstream` 成员函数 `open` 打开文件并将其与现有 `ofstream` 对象相连接，如下所示：

```
fout open("clients.dat");
```

文件打开方式		说明
<code>ios::app</code>	写入	将所有输出写入文件末尾
<code>ios::ate</code>		打开文件以便输出，并移到文件末尾(通常用于添加数据)数据可以
		文件中的任何地方
<code>ios::in</code>		打开文件以便输入
<code>ios::out</code>		打开文件以便输出
<code>ios::trunc</code>		删除文件现有内容(是 <code>ios::out</code> 的默认操作)
<code>ios::nocreate</code>		如果文件不存在，则文件打开失败
<code>ios::noreplace</code>		如果文件存在，则文件打开失败

图 4.5 文件打开方式

生成 `ofstream` 对象并准备打开时，程序测试打开操作是否成功。下列 `if` 结构中的操作(第 12 行到第 15 行)：

```
if ( ! fout ) {  
    cerr << "File could not be opened" << endl;  
    exit(1);  
}
```

用重载的 ios 运算符成员函数 `operator!` 确定打开操作是否成功。如果 `open` 操作的流将 `failbit` 或 `badbit` 设置，则这个条件返回非 0 值(`true`)。可能的错误是试图打开读取不存在的文件、试图打开读取没有权限的文件或试图打开文件以便写入而磁盘空间不足。

如果条件表示打开操作不成功，则输出错误消息 “File could not be opened”，并调用函数 `exit` 结束程序，`exit` 的参数返回到调用该程序的环境中，参数 0 表示程序正常终止。任何其他值表示程序因某个错误而终止。`exit` 返回的值让调用环境(通常是操作系统)对错误做出相应的响应。

另一个重载的 ios 运算符成员函数 `operator void*` 将流变成指针，使其测试为 0(空指针)或非 0(任何其他指针值)。如果 `failbit` 或 `badbit`(见第 11 章)对流进行设置，则返回 0(`false`)。下列 `while` 首部的条件自动调用 `operator void*` 成员函数：

```
while (cin >> account >> name >> balance )
```

只要 `cin` 的 `failbit` 和 `badbit` 都没有设置，则条件保持 `true`。输入文件结束符设置 `cin` 的 `failbit`。`operator void*` 函数可以测试输入对象的文件结束符，而不必对输入对象显式调用 `eof` 成员函数。

如果文件打开成功，则程序开始处理数据。下列语句(第 17 行和第 18 行)提示用户对每个记录输入不同域，或在数据输入完成时输入文件结束符：

```
cout << "Enter the account, name, and balance.\n"
    << "Enter EOF to and input.\n? ";
```

图 4. 6 列出了不同计算机系统中文件结束符的键盘组合。

计算机系统	组合键
UNIX 系统	<ctrl>d
IBM PC 及其兼容机	<ctrl>z
Macintosh	<ctrl>d
VAX(VMS)	<ctrl>z

图 14. 6 各种流行的计算机系统文件结束组合键

下列语句(第 24 行)：

```
while (cin >> account >> name >> balance )
```

输入每组数据并确定是否输入了文件结束符。输入文件结束符或不合法数据时，`cin` 的流读取运算符 `>>` 返回 0(通常这个流读取运算符 `>>` 返回 `cin`)，`while` 结构终止。用户输入文件结束符告诉程序没有更多要处理的数据。当用户输入文件结束符组合键时，设置文件结束符。只要没有输入文件结束符，`while` 结构就一直循环。

第 25 行和第 26 行：

```
fout << account << ' ' << name << ' ' << balance << '\n';
```

用流插入运算符 `<<` 和程序开头与文件相关联的 `fout` 对象将一组数据写入文件 “clients.dat”。可以用读取文件的程序取得这些数据(见 4.5 节)。注意图 4.4 中生成的文件是文本文件，可以用任何文本编辑器读取。

输入文件结束符后，`main` 终止，使得 `fout` 对象删除，从而调用其析构函数，关闭文件 `clients.dat`。程序员可以用成员函数 `close` 显式关闭 `ofstream` 对象，如下所示：

```
fout.close();
```


4.4 读取文件中的数据

为了在需要的时候能够检索要处理的数据，数据要存储在文件中。上一节演示了怎样建立一个顺序访问的文件。这一节要讨论按顺序读取文件中的数据。

图 4.7 中的程序读取文件"clients.dat"(图 4.4 中的程序建立)中的记录，并打印出了记录的内容。通过建立 ifstream 类对象打开文件以便输入。向对象传入的两个参数是文件名和文件打开方式。下列声明：

```
ifstream fin ( "clients.dat");
```

生成 ifstream 对象 fin，并将其与打开以便输入的文件 clients.dat 相关联。括号中的参数传入 ifstream 构造函数，打开文件并建立与文件的通信线路。

打开 ifstream 类对象默认为进行输入，因此下列语句：

```
ifstream fin ( "Clients.dat" );
```

可以打开 clients.dat 以便输入。和 ofstream 对象一样，ifstream 对象也可以生成而不打开特定文件，然后再将对象与文件相连接。

程序用 fin 条件确定文件是否打开成功，然后再从文件中读取数据。下列语句：

```
while (fin >> account >> name >> balance )
```

从文件中读取一组值(即记录)。第一次执行完该条语句后，account 的值为 100，name 的值为"John"，balance 的值为 24.98。每次执行程序中的该条语句时，函数都读取文件中的另一条记录，并把新的值赋给 account、name 和 balance。记录用函数 outputLine 显示，该函数用参数化流操纵算子将数据格式化之后再显示。到达文件末尾时，while 结构中的输入序列返回 0(通常返回 fin 流)，ifstream 析构函数将文件关闭，程序终止。

```
1 // Fig. 4.7: fig14_O7.cpp
2 // Reading and printing a sequential file
3 #include <iostream>
4 #include <fstream.h>
5 #include <iomanip.h>
6 #include <stdlib.h>
7
8 void outputLine( int, const char *, double );
9
10 int main()
11 {
12     // ifstream constructor opens the file
13     ifstream fin ( "clients.dat" );
14
15     if ( ! fin ) {
16         cerr << "File could not be opened\n";
17         exit( 1 );
18     }
19
20     int account;
21     char name[ 30 ] ;
```

```

22  double balance;
23
24  cout << setiosflags( ios::left ) << setw( 10 ) << "Account"
25      << setw( 13 ) << "Name" << "Balance\n";
26
27  while ( fin >> account >> name >> balance )
28      outputLine( account, name, balance );
29
30  return 0; // ifstream destructor closes the file
31 }
32
33 void outputLine( int acct, const char *name, double bal )
34 {
35     cout << setiosflags( ios::left ) << setw( 10 ) << acct
36         << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
37         << resetiosflags( ios::left )
38         << setiosflags( ios::fixed | ios::showpoint )
39         << bal << '\n';
40 }

```

输出结果:

Account	Name	Balance
i00	Jones	24.98
200	Doe	345.67
300	White	0.00
400	Stone	-42.16
500	Rich	224.62

图 4.7 读取并打印一个顺序文件

为了按顺序检索文件中的数据，程序通常要从文件的起始位置开始读取数据，然后连续地读取所有的数据，直到找到所需要的数据为止。程序执行中可能需要按顺序从文件开始位置处理文件中的数据好几次。istreatrl 类和 ostream 类都提供成员函数，使程序把“文件位置指针”(file position pointer，指示读写操作所在的下一个字节号)重新定位。这些成员函数是 istream 类的 seekg(“seekget”)和 ostream 类的 seekp(“seek put”)。每个 istream 对象有个 get 指针，表示文件中下一个输入

相距的字节数，每个 ostream 对象有一个 put 指针，表示文件中下一个输出相距的字节数。下列语句：

```
fin.seekg( 0 );
```

将文件位置指针移到文件开头(位置 0)，连接 fin。seekg 的参数通常为 long 类型的整数。

第二个参数可以指定寻找方向，ios::beg(默认)相对于流的开头定位，ios::cur 相对于流当前位置定位，ios::end 相对于流结尾定位。文件位置指针是个整数值，指定文件中离文件开头的相对位置(也称为离文件开头的偏移量)。下面是一些 get 文件位置指针的例子：

```

// position to the nth byte of fileObject
// assumes ios::beg
fileObject.seekg( n );
// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );
// position y bytes back from end of fileObject
fileObject.seekg( y, ios::end );
// position at end of fileObject
fileObject.seekg( o, ios::end );

```

ostream 成员函数 seekp 也可以进行类似的操作。成员函数 tellg 和 tellp 分别返回 get 和 put 指针的当前位置。下列语句将 get 文件位置指针值赋给 long 类型的变量 location。

```
location = filObject.tellg();
```

4.5 更新访问文件

4.4 节介绍了格式化和写入访问文件的数据修改时会有破坏文件中其他数据的危险。例如，如果要把名字"White"改为"Worthington"，则不是简单地重定义旧的名字。White 的记录是以如下形式写入文件中的：

```
300 White 0.00
```

如果用新的名字从文件中相同的起始位置重写该记录，记录的格式就成为：

```
300 Worthington 0.00
```

因为新的记录长度大于原始记录的长度，所以从“Worthington”的第二个“0”之后的字符将重定义文件中的下一条顺序记录。出现该问题的原因在于：在使用流插入运算符<<和流读取运算符>>的格式化输入，输出模型中，域的大小是不定的，因而记录的大小也是不定的。例如，7、14、-117、2047 和 27383 都是 int 类型的值，虽然它们的内部存储占用相同的字节数，但是将它们以格式化文本打印到屏幕上或存储在磁盘上时要占用不同大小的域。因此，格式化输入，输出模型通常不用来更新已有的记录。

也可以修改上述名字，但比较危险。比如，在 300 White 0.00 之前的记录要复制到一个新的文件中，然后写入新的记录并把 300 White 0.00 之后的记录复制到新文件中。这种方法要求在更新一条记录时处理文件中的每一条记录。如果文件中一次要更新许多记录，则可以用这种方法。

第 5 章 C++的字符串流

5.1 流的继承关系

在 C++ 中，有一个 `stream` 这个类，所有的 I/O 都以这个“流”类为基础的，里面包括了所有的输入输出类，今天我们就来介绍一下 `sstream.h`(字符串流)这个类：

C++ 引入了 `ostreamstream`、`istreamstream`、`stringstream` 这三个类，要使用他们创建对象就必须包含 `sstream.h` 头文件。

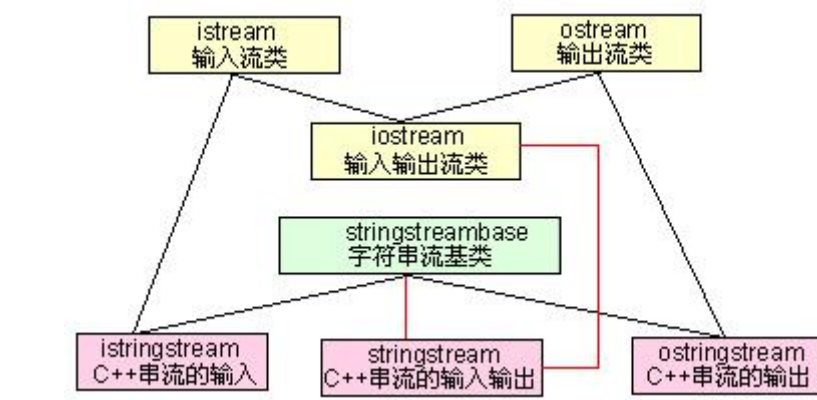
`istreamstream` 类用于执行 C++ 风格的串流的输入操作。

`ostreamstream` 类用于执行 C 风格的串流的输出操作。

`stringstream` 类同时可以支持 C 风格的串流的输入输出操作。

`istreamstream` 类是从 `istream`（输入流类）和 `stringstreambase`（c++字符串流基类）派生而来，`ostreamstream` 是从 `ostream`（输出流类）和 `stringstreambase`（c++字符串流基类）派生而来，`stringstream` 则是从 `iostream`(输入输出流类)和 `stringstreambase`（c++字符串流基类）派生而来。

他们的继承关系如下图所示：



5.2 字符串流的输入操作

`istreamstream` 是由一个 `string` 对象构造而来，`istreamstream` 类从一个 `string` 对象读取字符。

`istreamstream` 的构造函数原形如下：

```
istreamstream::istreamstream(string str);
```

```
#include <iostream>
```

```
#include <sstream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    istreamstream istr;
```

```
    istr.str("1 56.7",);
```

```

//上述两个过程可以简单写成 istreamstring istr("1 56.7");
cout << istr.str()<<endl;
int a;
float b;
istr>>a;
cout<<a<<endl;
istr>>b;
cout<<b<<endl;
system("pause");
}

```

上例中，构造字符串流的时候，空格会成为字符串参数的内部分界，例子中对 a, b 对象的输入“赋值”操作证明了这一点，字符串的空格成为了整型数据与浮点型数据的分解点，利用分界获取的方法我们事实上完成了字符串到整型对象与浮点型对象的拆分转换过程。

str() 成员函数的使用可以让 istreamstring 对象返回一个 string 字符串（例如本例中的输出操作 cout<<istr.str();）。

5.3 字符串流的输出操作

ostreamstring 同样是由一个 string 对象构造而来，ostreamstring 类向一个 string 插入字符。

ostreamstring 的构造函数原形如下：

```
ostreamstring::ostreamstring(string str);
```

示例代码如下：

```

#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
    ostreamstring ostr;
    ostr.str("abc");//如果构造的时候设置了字符串参数,那么增长操作的时候不会从结尾开始增加,而是修改原有数据,超出的部分增长
    ostr.put('d');
    ostr.put('e');
    ostr<<"fg";
    string gstr = ostr.str();
    cout<<gstr;
    system("pause");
}

```

在上例代码中，我们通过 put() 或者左移操作符可以不断向 ostr 插入单个字符或者是字符串，通过 str() 函数返回增长过后的完整字符串数据，但值得注意的一点是，当构造的时候对象内已经存在字符串数据的时候，那么增长操作的时候不会从结尾开始增加，而是修改原有数据，超出的部分增长。

对于 stringstream 来说，不用我多说，大家也已经知道它是用于 C++ 风格的字符串的输

入输出的。

stringstream 的构造函数原形如下：

```
stringstream::stringstream(string str);
```

示例代码如下：

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
    stringstream ostr("ccc");
    ostr.put('d');
    ostr.put('e');
    ostr<<"fg";
    string gstr = ostr.str();
    cout<<gstr<<endl;
    char a;
    ostr>>a;
    cout<<a
    system("pause");
}
```

5.4 字符串流在数据类型转换中的应用

除此而外，stringstream 类的对象我们还常用它进行 string 与各种内置类型数据之间的转换。

示例代码如下：

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
    stringstream sstr;
    //-----int 转 string-----
    int a=100;
    string str;
    sstr<<a;
    sstr>>str;
    cout<<str<<endl;
    //-----string 转 char[]-----
```

sstr.clear(); //如果你想通过使用同一 stringstream 对象实现多种类型的转换，请注意在每一次转换之后都必须调用 clear() 成员函数。

```
string name = "colinguan";
```

```

char cname[200];
sstr<<name;
sstr>>cname;
cout<<cname;
system("pause");
}

```

5.5 输入/输出的状态标志

接下来我们来学习一下输入/输出的状态标志的相关知识，C++中负责的输入/输出的系统包括了关于每一个输入/输出操作的结果的记录信息。这些当前的状态信息被包含在 `io_state` 类型的对象中。`io_state` 是一个枚举类型（就像 `open_mode` 一样），以下便是它包含的值。

`goodbit` 无错误

`Eofbit` 已到达文件尾

`failbit` 非致命的输入/输出错误，可挽回

`badbit` 致命的输入/输出错误，无法挽回

有两种方法可以获得输入/输出的状态信息。一种方法是通过调用 `rdstate()` 函数，它将返回当前状态的错误标记。例如，假如没有任何错误，则 `rdstate()` 会返回 `goodbit`。

下例示例，表示出了 `rdstate()` 的用法：

```

#include <iostream>
using namespace std;
int main()
{
    int a;
    cin>>a;
    cout<<cin.rdstate()<<endl;
    if(cin.rdstate() == ios::goodbit)
    {
        cout<<"输入数据的类型正确，无错误！"<<endl;
    }
    if(cin.rdstate() == ios_base::failbit)
    {
        cout<<"输入数据类型错误，非致命错误，可清除输入缓冲区挽回！"<<endl;
    }
    system("pause");
}

```

另一种方法则是使用下面任何一个函数来检测相应的输入/输出状态：

`bool bad();`

`bool eof();`

`bool fail();`

`bool good();`

下例示例，表示出了上面各成员函数的用法：

```

#include <iostream>
using namespace std;

```

```

int main()
{
    int a;
    cin>>a;
    cout<<cin.rdstate()<<endl;
    if(cin.good())
    {
        cout<<"输入数据的类型正确，无错误！"<<endl;
    }
    if(cin.fail())
    {
        cout<<"输入数据类型错误，非致命错误，可清除输入缓冲区挽回！"<<endl;
    }
    system("pause");
}

```

如果错误发生，那么流状态既被标记为错误，你必须清除这些错误状态，以使你的程序能正确适当地继续运行。要清除错误状态，需使用 `clear()` 函数。此函数带一个参数，它是你将要设为当前状态的标志值。，只要将 `ios::goodbit` 作为实参。

示例代码如下：

```

#include <iostream>
using namespace std;
int main()
{
    int a;
    cin>>a;
    cout<<cin.rdstate()<<endl;
    cin.clear(ios::goodbit);
    cout<<cin.rdstate()<<endl;
    system("pause");
}

```

通常当我们发现输入有错又需要改正的时候，使用 `clear()` 更改标记为正确后，同时也需要使用 `get()` 成员函数清除输入缓冲区，以达到重复输入的目的。

示例代码如下：

```

#include <iostream>
using namespace std;
int main()
{
    int a;
    while(1)
    {
        cin>>a;
        if(!cin)//条件可改写为 cin.fail()
        {
            cout<<"输入有错!请重新输入"<<endl;

```



```

cin.clear();
cin.get();
}
else
{
cout<<a;
break;
}
}
system("pause");
}

```

最后再给出一个对文件流错误标记处理的例子，巩固学习，代码如下：

```

#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ifstream myfile("c:.txt", ios_base::in, 0);
if(myfile.fail())
{
cout<<"文件读取失败或指定文件不存在!"<<endl;
}
else
{
char ch;
while(myfile.get(ch))
{
cout<<ch;
}
if(myfile.eof())
{
cout<<"文件内容已经全部读完"<<endl;
}
while(myfile.get(ch))
{
cout<<ch;
}
}
system("pause");
}

```

第 6 章 控制结构

6.1 简介

编写解决特定问题的程序之前，首先要彻底了解问题并认真计划解决问题的方法。编写程序时，还要了解可用的基本组件和采用实践证明的程序结构原则。本章将讨论结构化编程的理论和原理的所有问题。这里介绍的技术适用于大多数高级语言，包括 C++。

6.2 算法

任何计算问题都可以通过按特定顺序执行一系列操作而完成。解决问题的过程(procedure)称为算法(algorithm)，包括：

1. 执行的操作(action)
2. 执行操作的顺序(order)

下例演示正确指定执行操作的顺序是多么重要：

考虑每个人早晨起床到上班的“朝阳算法”：(1)起床，(2)脱睡衣，(3)洗澡，(4)穿衣，(5)吃早饭，(6)搭车上班。

总裁可以按这个顺序，从容不迫地来到办公室。假设把顺序稍作调换：(1)起床，(2)脱睡衣，(3)穿衣，(4)洗澡，(5)吃早饭，(6)搭车上班。

如果这样，总裁就得带着肥皂水来上班。指定计算机程序执行语句的顺序称为程序控制(program control)，本章介绍 C++程序的控制功能。

6.3 控制结构

通常，程序中的语句按编写的顺序一条一条地执行，称为顺序执行(sequential execution)。程序员可以用稍后要介绍的不同 C++语句指定下一个执行的语句不是紧邻其后的语句，这种技术称为控制转移(transfer of control)。

20 世纪 60 年代，人们发现，软件开发小组遇到的许多困难都是由于控制转移造成的。goto 语句使程序员可以在程序中任意指定控制转移目标，因此人们提出结构化编程就是为了清除 goto 语句。

Bohm 和 JMoP5n1 的研究表明，不用 goto 语句也能编写程序。困难在于程序员要养成不用 goto 语句的习惯。直到 20 世纪 70 年代，程序员才开始认真考虑结构化编程，结果使软件开发小组的开发时间缩短、系统能够及时交付运行并在预算之内完成软件项目。这些成功的关键是，结构化编程更清晰、更易调试与修改并且不容易出错。

BohM 和 J • jecopini 的研究表明，所有程序都可以只用三种控制结构(control structure)即顺序结构(sequence structure)、选择结构(selection structure)和重复结构(repetition structure)。顺序结

构是 C++ 内置的，除非另外指定，计算机总是按编写的顺序一条一条地执行。图 2.1 的流程图 (flowchart) 演示了典型的顺序结构。按顺序进行两次计算。

流程图是算法或部分算法的图形表示。流程图用一些专用符号绘制，如长方形、菱形、椭圆和小圆，这些符号用箭头连接，称为流程。

C++ 提供三种选择结构，本章将介绍这三种选择结构。if 选择结构在条件为 true 时执行一个操作，在条件为 false 时跳过这个操作。if/else 选择结构在条件为 true 时执行一个操作，在条件为 false 时执行另一个操作。switch 选择结构根据表达式取值不同而选择不同操作。

if 选择结构称为单项选择结构(single-selection structure)，选择或忽略一个操作。if/else 选择结构称为双项选择结构(double-selection structure)，在两个不同操作中选择。switch 选择结构称为多项选择结构(multiple-selection structure)，在多个不同操作中选择。

C++ 提供三种重复结构 while、do/while 和 for。if、else、switch、while、do 和 for 等都是 C++ 关键字(keyword)。这些关键字是该语言保留的，用于实现如 C++ 控制结构等不同特性。

C++ 只有七种控制结构：顺序结构、三种选择结构和三种重复结构。每个 C++ 程序都是根据程序所需的算法组合这七种控制结构。这种单入/单出控制结构(single-entry/single-exit control structure)使程序容易建立，只要将一个控制结构的出口与另一个控制结构的入口连接，即可组成程序。这点很像小孩子堆积木，因此称为控制结构堆栈(control-structure stacking)，还有另一种控制结构连接方法，称为控制结构嵌套(control-structure nesting)。

6.4 if 选择结构

选择结构在不同操作之间选择。例如，假设考试成绩 60 分算及格，则下列伪代码：

```
if student's grade is greater than or equal to 60
    print "Passed"
```

确定“学生成绩大于或等于 60 分”是 true 或 false，如果是 true，则该生及格，打印“Passed”字样，并顺序“执行”下一个伪代码语句(记住，伪代码不是真正的编程语言)。如果条件为 false，则忽略打印语句，并顺序“执行”下一个伪代码语句。注意这个选择结构第二行的缩排，这种缩排是可选的，但值得提倡，因为它能体现结构化程序的内部结构。将伪代码变成 C++ 代码时，C++ 编译器忽略空格、制表符、换行符等用于缩排和垂直分隔的空白字符。

上述伪代码的 if 语句可以写成如下 C++ 语句：

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

注意 C++ 代码与伪代码密切对应，这是伪代码的一个属性，使得其成为有用的程序开发工具。

注：伪代码常用于程序存设计期间“构思”程序，然后再将伪代码程序转换为 C++ 程序。

if 结构也是单入/单出结构。

6.5 if/else 选择结构

if 选择结构只在条件为 true 时采取操作，条件为 false 时则忽略这个操作。利用 if/else 选择结构则可以在条件为 true 时和条件为 false 时采取不同操作。例如，下列伪代码：

```
if student's grade is greater than or equal to 60
    print "Passed"
else
    print "Failed"
```

在学生成绩大于或等于 60 时打印 “Passed”，否则打印“Failed”。打印之后，都“执行”下一条伪代码语句。注意 else 的语句体也缩排。

上述伪代码 if/else 结构可以写成如下的 C++代码：

```
if(grade>=60)
    cout<<"Passed";
else
    cout<<"Failed";
```

C++提供条件运算符(?:)，与 if/else 结构密切相关。条件运算符是 C++中惟一的三元运算符(thrmary operator)，即取三个操作数的运算符。操作数和条件运算符一起形成条件表达式(conditional expression)。第一个操作数是条件，第二个操作数是条件为 true 时整个条件表达式的值。第三个操作数是条件为 false 时整个条件表达式的值。例如，下列输出语句：

```
cout<<(grade>=60? "Passed": "Failed");
```

包含的条件表达式在 grade=60 取值为 true 时，求值为字符串 “Passed”；在 grade>=60 取值为 false 时，求值为字符串“Failed”。这样，带条件表达式的语句实际上与上述 if/else 语句相同。可以看出，条件运算符的优先级较低，因此上述表达式中的话号是必需的。

条件表达式酌值也可以是要执行的操作。例如，下列条件表达式：

```
grade >=60? cout<<"Passed": cout<<"Failed";
```

表示如果 grade 大于或等于 60，则执行 cout<<"Passed"，否则执行 cout<<"Failed"。这与前面的 if/else 结构也是相似的。条件运算符可以在一些无法使用 if/else 语句的情况中使用。

嵌套 if/else 结构(nested if/else structure)测试多个选择，将一个 if/else 选择放在另一个 if/else 选择中。例如，下列伪代码语句在考试成绩大于或等于 90 分时打印 A。在 80 到 89 分之间时打印 B，在 70 到 79 分之间时打印 C，在 60 到 69 分之间时打印 D，否则打印 F。

```
if studen's grade is greater than or equal to 90

    print "A"
else
    If student's grade is greater than or equal to 80
        print "B"
    else
        If student's grade is greater than or equal to 70
            print "C"
        else
            If student's grade is greater than or equal to 60
```

```

        print "D"
    else
        print "F"
这个伪代码对应下列 C++代码:
    if (grade>=90)
        cout<<"A";
    else
        if (grade>=80)
            cout<<"B";
        else
            if (grade>=70)
                cout<<"C";
            else
                if (grade>=60)
                    cout<<"D";
                else
                    cout<<"F";

```

如果考试成绩大于或等于 90 分，则前 4 个条件都为 true，但只执行第一个测试之后的 cout 语句。执行这个 cout 语句之后，跳过外层 if/else 语句的 else 部分。许多 C++程序员喜欢将上述 if 结构写成：

```

if (grade>=90)
    cout<<"A";
else if (grade>=80)
    cout<<"B";
else if (grade>=70)
    cout<<"C";
else if (grade>=60)
    cout<<"D";
else
    cout<<"F";

```

两者形式是等价的。后者更常用，可以避免深层缩排便代码移到右端。深层缩排会使一行的空间太小，不长的行也要断行，从而影响可读性。

下例在 if/else 结构的 else 部分包括复合语句：

```

if (grade>=60)
    cout<<"Passed.\n";
else{
    cout<<"Failed.\n";
    cout<<"You must take this course again.\n";
}

```

如果 grade 小于 60，则程序执行 else 程序体中的两条语句并打印：

```

Failed.
You must take this course again.

```

注意 else 从句中的两条语句放在花括号中。这些花括号很重要，如果没有这些花括号，则下列语句：

```
cout<<"You must take this cours again.\n";
```

在 if 语句 else 部分之外，不管成绩是否小于 60 都执行。

本节介绍了复合语句的符号。复合语句可以包含声明(例如，和 main 程序体中一样)，如果这，则这个复合语句称为块(block)。块中的声明通常放在块中任何操作语句之前，但也可以和操作语句相混和。

6.6 while 重复结构

重复结构(repetition strucure)使程序员可以指定一定条件下可以重复的操作。下列伪代码语句：

```
While there are more items on my shopping list
Purchase next item and cross it off my list
```

描述购物过程中发生的重复。条件"there are more Items on my shopping list"(购物清单中还有更多项目)可真可假。如果条件为 true，则执行操作"Purchase next item and cross it off my list"(购买下一个项目并将其从清单中划去)。如果条件仍然为 true，则这个操作重复执行。while 重复结构中的语句构成 while 的结构体，该结构体可以是单句或复合句。最终，条件会变为 false(购买清单中最后一个项目并将其从清单中划去时)，这时重复终止，执行重复结构之后的第一条伪代码语句。

作为实际 while 的例子，假设程序要寻找 2 的第一个大于 1000 的指数值。假设整数变量 product 初始化为 2，执行下列 while 重复结构之后，product 即会包含所要值：

```
int product = 2;
while ( product <= 1000 )
    product = 2 * product;
```

进入 while 结构时，product 的值为 2。变量 product 重复乘以 2，连续取值 4、8、16、32、64、128、256、512 和 1024。当 product 变为 1024 时，while 结构条件 product<=1000 变为 false，因此终止重复，product 的最后值为 1024。程序继续执行 while 后面的下一条语句。

6.7 构造算法：实例研究 1(计数器控制重复)

要演示如何设计算法，我们要解决几个全班平均成绩的问题。考虑下列问题：

班里有 10 个学生参加测验，可以提供考试成绩(0 到 100 的整数)，以确定全班平均成绩。

全班平均成绩等于全班成绩总和除以班里人数。计算机上解决这个问题的算法是辅人每人的成绩，进行平均计算，然后打印结果。

下面用伪代码列出要执行的操作，指定这些操作执行的顺序。我们用计数器控制重复(counter-controlled repetition)一次一个地输入每人的成绩。这种方法用计数器(counter)变量控制一组语句执行的次数。本例中，计数器超过 10 时，停止重复。本节介绍伪代码算法(如图 6.6)和对

应程序(如图 6.7)。下节介绍如何开发这个伪代码算法。计数器控制重复通常称为确定重复(definite repetition)，因为循环执行之前，已知重复次数。

注意算法中引用了总数(total)和计数器。总数变量用于累计一系列数值的和。计数器变量用于计数，这里计算输入的成绩数。存放总数的变量通常应先初始化为 0 之后再在程序中使用，否则总和会包括总数的内存地址中存放的原有数值。

```
Set total to zero
Set grade counter to one
While grade counter is less than or equal to ten
    Input the next grade
    Add the grade i.to the total
    Add one to the grade counter
Set the class average to the total divided by ten
Print the class average
```

图 6.6 用计数器控制重复解决全班平均成绩问题的伪代码算法

```
1 // Fig. 2.7: fig0207.cpp
2 // Class average program with counter-controlled repetition
3 #include <iostream.h>
4
5 int main()
6 {
7     int total,           // sum of grades
8         gradeCounter,    // number of grades entered
9         grade,           // one grade
10        average;         // average of grades
11
12    // initialization phase
13    total = 0;             // clear total
14    gradeCounter = 1;      // prepare to loop
15
16    // processing phase
17    while ( gradeCounter <= 10 ) { // loop 10 times
18        cout << "Enter grade: "; // prompt for input
19        cin >> grade;           // input grade
20        total = total + grade;   // add grade to total
21        gradeCounter = gradeCounter + 1; // increment counter
22    }
23
24    // termination phase
25    average = total / 10;        // integer division
26    cout << "Class average is " << average << endl;
27
28    return 0; // indicate program ended successfully
```

29 }

输出结果:

Enter grade: 98

Enter grade: 76

Enter grade: 71

Enter grade: 87

Enter grade: 83

Enter grade: 90

Enter grade: 57

Enter grade: 79

Enter grade: 82

Enter grade: 94

Class average is 81

图 6.7 用计数器控制重复解决全班平均成绩问题的 C++程序和示例输出

根据使用情况, 计数器变量通常应先初始化为 0 或 1(下面会分别举例说明)。未初始化变量会包含垃圾值“garbage” value), 也称为未定义值(undefined value), 为该变量保存内存地址中最后存放的值。

切记: 一定要初始化计数器和总和变量。每个变量在单独一行中声明。

注意程序中的平均计算产生一个整数结果。实际上, 本例中的成绩总和是 81.7, 除以 10 时应得到 81.7, 是个带小数点的数, 下节将介绍如何处理这种值(称为浮点数)。

图 6.7 中, 如果第 21 行用 gradeCounter 而不是 10 进行计算, 则这个程序的输出显示数值 74。

6.8 构造算法与自上而下逐步完善: 实例研究 2(标记控制重复)

下面将全班平均成绩问题一般化, 考虑如下问题:

设计一个计算全班平均成绩的程序, 在每次程序运行时处理任意个成绩数。

在第一个全班平均成绩例子中, 成绩个数(10)是事先预置的。而本例中, 则不知道要输入多少个成绩, 程序要处理任意个成绩数。程序怎么确定何时停止输入成绩呢?何时计算和打印全班平均成绩呢?

一种办法是用一个特殊值作为标记值(sentinelvalue), 也称信号值(signalvalue)、哑值(dummy value)或标志值(flag value), 表示数据输入结束(“end of data entry”)用户输入成绩, 直到输入所有合法成绩。然后用户输入一个标记值, 表示最后一个成绩已经输入。标记控制重复(sentinel-controlled repetition)也称为不确定重复(indefinite repetition), 因为执行循环之前无法事先知道重复次数。

显然, 标记值不能与可接受的输入值混淆起来。由于考试成绩通常是非负整数, 因此可以用 -1 作标记值。这样, 全班平均成绩程序可以处理 95、96、75、74、89 和 -1 之类的输入流。程序计算并打印成绩 95、96、75、74 和 89 的全班平均成绩(不计入 -1, 因为它是标记值)。

我们用自上而下逐步完善(top-down, stepwise refinement)的方法开发计算全班平均成绩的程序, 这是开发结构化程序的重要方法。我们首先生成上层伪代码表示:

Determine the class average for the quiz

上层伪代码只是一个语句，表示程序的总体功能。这样，上层等于是程序的完整表达式。但上层通常无法提供编写 C++ 程序所需的足够细节。因此要开始完善过程。我们将上层伪代码分解为一系列的小任务，按其需要完成的顺序列出。这个结果就是下列第一步完善(first, refinement):

Initialize variables

Input, sum, and count the quiz grades

Calculate and print the class average

这里只用了顺序结构，所有步骤按顺序逐步执行。

要进行下一步完善(即第二步完善，second refinement)，我们要指定特定变量，要取得数字的动态和以及计算机处理的数值个数，用一个变量接收每个输入的成绩值，一个变量保存计算平均值。下列伪代码语句：

Initialize variables

可以细化成：

Initialize total to zero

Initialize counter to zero

注意，只有 total 和 counter 变量要先初始化再使用，average 和 grade 变量(分别计算平均值和用户输入)不需要初始化。因为它们的值会在计算或输入时重定义。

下列伪代码语句：

Input, sum, and count the quiz grades

需要用重复结构(即循环)连续输入每个成绩。由于我们事先不知道要处理多少个成绩，因此使用标记控制重复。用户一次一项地输入合法成绩。输入最后一个合法成绩后，用户输入标记值。程序在每个成绩输入之后测试其是否为标记值。如果用户输入标记值，则顺序循环终止。上述伪代码语句的第二步完善如下：

Input the first grade (possibly the sentinel)

While the user has not as yet entered the sentinel

Add this grade into the running total

Add one to the grade counter

Input the next grade (possibly the sentinel)

注意，在这个伪代码中，我们没有在 while 结构体中使用花括号，只是在 while 下面将这些语句缩排表示它们属于 while。伪代码只是非正式的程序开发辅助工具。

下列伪代码语句可以完善如下：

If the counter is not equal to zero

Set the average to the total divided by the counter

Print the average

else

Print "No grades were entered"

注意我们这里要测试除数为 0 的可能性，这是个致命逻辑错误，如果没有发现，则会使程序失败(通常称为爆炸或崩溃)。以上显示了全班平均成绩问题第二步完善的完整伪代码语句。伪代码中增加了一些空行，使伪代码更易读。空行将程序分成不同阶段。

图 6.8 所示的伪代码算法解决更一般的全班平均成绩问题，这个算法只进行了第二步完善，还

需要进一步完善。

```

Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

if the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print "No grades were entered"

```

图 6.8 用标记符控制重复解决全班平均成绩问题的伪代码算法

图 6.9 显示了 C++ 程序和示例执行结果。尽管只输入整数成绩，但结果仍然可能产生带小数点的平均成绩，即实数。int 类型无法表示实数，程序中引入 float 数据类型处理带小数点的数(也称为浮点数，floatingpoint number)，并引入特殊的强制类型转换运算符(cast operator)处理平均值计算。这些特性将在程序之后详细介绍。

```

1 // Fig. 2.9: fig02_09.cpp
2 // Class average program with sentinel-controlled repetition.
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int total,      // sum of grades
9     gradeCounter, // number of grades entered
10    grade;          // one grade
11    float average;  // number with decimal point for average
12
13    // initialization phase
14    total = 0;
15    gradeCounter = 0;
16
17    // processing phase
18    cout << "Enter grade, -1 to end: ";
19    cin >> grade;
20
21    while ( grade != -1 ) {
22        total = total + grade;

```

```

23  gradeCounter = gradeCounter + 1;
24  cout << "Enter grade, -1 to end: ";
25  cin >> grade;
26 }
27
28 // termination phase
29 if ( gradeCounter != 0 ) {
30     average = static_cast< float >( total ) / gradeCounter;
31     cout << "Class average is " << setprecision{ 2 }
32         << setiosflags( ios::fixed | ios::showpoint )
33         << average << endl;
34 }
35 else
36     cout << "NO grades were entered" << endl;
37
38     return 0;           // indicate program ended successfully
39 }

```

输出结果:

```

Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50

```

图 6.9 用标记符控制重复解决全班平均成绩问题的 C++程序和示例执行结果

注意图 6.9 中 while 循环中的复合语句。如果没有花括号，则循环体中的最后三条语句会放到循环以外，使计算机错误地理解如下代码：

```

while ( grade != -1 )
    total = total + grade;
gradeCounter = gradeCounter + 1;
cout << "Enter grade, -1 to end:";
cin >> grade;

```

如果用户输入的第一个成绩不是-1，则会造成无限循环。

注意下列语句：

```
cin >> grade;
```

前面用一个输出语句提示用户输入。

平均值并不一定总是整数值，而常常是包含小数的值，如 7.2 或 -93.5。这些值称为浮点数，用数据类型 `float` 表示。变量 `average` 声明为数据类型 `float`，以获得计算机结果中的小数。但 `total/gradeCounter` 的计算结果是整数，因为 `total` 和 `gradeCounter` 都是整数变量。两个整数相除是整除(integer division)，小数部分丢失(即截尾，truncated)。由于先要进行计算，因此小数部分在将结果赋给 `average` 之前已经丢失。要用整数值进行浮点数计算，就要先生成用于计算的临时浮点数值。

C++提供了一元强制类型转换运算符(unary cast operator)。下列语句：

```
average = static_cast< float >(total) / gradeCounter;
```

包括一元强制类型转换运算符 `static_cast<float>()`，生成用于计算的临时浮点数值(`total`)。这样使用强制类型转换运算符称为显式类型转换(explicit conversion)。`total` 中存放的值还是整数，而计算时则用浮点数值(`total` 的临时 `float` 版本)除以整数 `gradeCounter`。

c++编译器只能对操作数的数据类型一致的表达式求值。要保证操作数的数据类型一致，编译器对所选择的操作数进行提升(promotion)操作(也称为隐式类型转换，implicit conversion)。例如，在包含数据类型 `float` 和 `int` 的表达式中，`int` 操作数提升为 `float`。本例中，`gradeCounter` 提升为 `float` 之后进行计算，将浮点数除法得到的结果赋给 `average`。本章稍后将介绍所有标准数据类型及其提升顺序。任何数据类型都可用强制类型转换运算符，`static_cast` 运算符由关键字 `static cast` 加尖括号(<>)中的数据类型名组成。强制类型转换运算符是个一元运算符(unary operator)，即只有一个操作数的运算符。第 1 章曾介绍过二元算术运算符。C++也支持一元正(+)、负(-)运算符，程序员可以编写 -7、+5 之类的表达式。强制类型转换运算符从右向左结合，其优先级高于正(+)、负(-)运算符等其他一元运算符，该优先级高于运算符*、/和%，但低于括号的优先级。优先级表中用 `static_cast<type>()` 表示强制类型转换运算符。

图 6.9 中格式化功能将在第 2 章详细介绍，这里先做一简要介绍。下列输出语句中调用

```
setprecision(2);  
cout<<"Class average is" << setprecision(2)  
<< Setiosflags(ios::fixed | ios::showpoint)  
<<average<<endl;
```

表示 `float` 变量 `average` 打印小数点右边的位数为两位精度(precision)，例如 92.37，这称为参数化流操纵算子(parameterized stream manipulator)。使用这些调用的程序要包含下列预处理指令：

```
#include <iomanip.h>
```

注意 `endl` 是非参数化流操纵算子(nonparameterized stream manipulator)，不需要 `iomanip.h` 头文件。如果不指定精度，则浮点数值通常输出六位精度(即默认精度，default precision)，但稍后也会介绍一个例外。

上述语句中的流操纵算子 `setiosflags(ios::fixed | ios::showpoint)` 设置两个输出格式选项 `ios::fixed` 和 `ios::showpoint`。垂直条(|)分隔 `setiosflags` 调用中的多个选项(垂直条将在第 16 章详细介绍)。选项 `ios::fixed` 使浮点数值以浮点格式(而不是科学计数法，见第 2 章)输出。即使数值为整数，`ios::showpoint` 选项也会强制打印小数点和尾部 0，如 88.00。如果不用 `ios::showpoint` 选项，则 C++将该整数显示为 88，不打印小数点和尾部 0。程序中使用上述格式时，将打印的值取整，表示小数点位数，但内存中的值保持不变。例如，数值 87.945 和 67.543 分别输出为 87.95 和 67.54。

尽管浮点数算不总是 100%精确，但其用途很广。例如，我们说正常体温 98.6(华氏温度)时，并不需要精确地表示，如果温度计上显示 98.6 度。实际上可能是 98.5999473210643 度。这里显示 98.6 对大多数应用已经足够了。

另一种得到浮点数的方法是通过除法。10 除以 3 得到 3.333333……，是无限循环小数。计算机只分配固定空间保存这种值，因此只能保存浮点值的近似值。

6.9 构造算法与自上而下逐步完善：实例研究 3(嵌套控制结构)

下面介绍另一个问题。这里还是用伪代码和自上而下逐步完善的方法构造算法，然后编写相应的 C++ 程序。我们介绍过按顺序堆叠的控制结构，就像小孩堆积木一样。这里显示 C++ 中控制结构的另一种方法，称为嵌套控制结构。

考虑下列问题：

学校开了一门课，让学生参加房地产经纪人证书考试。去年，几个学生读完这门课并参加了证书考试。学校想知道学生考试情况，请编写一个程序来总结这个结果。已经得到了 10 个学生的名单，每个姓名后面写 1 时表示考试通过，写 2 时表示没有通过。

程序应分析考试结果，如下所示：

1. 输入每个考试成绩(即 1 或 2)，每次程序请求另一个考试成绩时，在屏幕上显示消息“Enter result”。
2. 计算每种类型的考试成绩数。
3. 显示总成绩，表示及格人数和不及格人数。
6. 如果超过 8 个学生及格，则打印消息“Raise tuition”。

认真分析上述问题后，我们做出下列结论：

1. 程序要处理 10 个考试成绩，用计数器控制循环。
2. 每个考试成绩为数字 1 或 2，每次程序读取考试成绩时，程序要确定成绩是否为数字 1 或 2。

我们的算法中测试 1，如果不是 1，则我们假设其为 2(本章末尾的练习会考虑这个假设的结果)。

3. 使用两个计数器，分别计算及格人数和不及格人数。
4. 程序处理所有结果之后，要确定是否有超过 8 个学生及格。

下面进行自上而下逐步完善的过程。首先是上层的伪代码表示：

Analyze exam results and decide if tuition should be raised

我们再次强调，顶层是程序的完整表达，但通常要先进行几次完善之后才能将伪代码自然演变成 C++ 程序。我们的第一步完善为：

Initialize variables

Input the ten quiz grades and Count passes and failures

Print a summary Of the exam results and decide if tuition should be raised

这里虽然有整个程序的完整表达式，但还需要进一步完善。我们要提供特定变量。要用两个计数器分别计算，用一个计数器控制循环过程，用一个变量保存用户输入。伪代码语句：

Initialize variables

可以细分如下：

Initialize passes to zero

Initialize failures to zero

Initialize student counter to One

注意：这里只初始化计数器和总和。伪代码语句：

Input the ten quiz grades and count Passes and faiLures

要求循环输入每个考试成绩。我们事先知道共有 10 个成绩，因此可以用计数器控制循环。在循环中(即嵌套在循环中)，用一个双项选择结构确定考试成绩为数字 1 或 2，并递增相应的计数器。上述伪代码语句细化如下：

```
while student counter is less than or equal to ten
Input the next exam result
if the student passed
Add one to Passes
else
Add One to failures
Add one to student counter
```

注意这里用空行分开 if/else 控制结构，以提高程序可读性。伪代码语句：

Print a sugary Of the exam results and declde if tuition should be raised

可以细化如下：

```
Print the number of passes
Print the number of filuies
if more than eight students Passed
Priht "Raise tuition"
```

图 6.10 显示了完整的第 2 步完善结果。注意这里用空行分开 while 结构，以提高程序可读性。

```
Initlalize passes to zero
Init±a lize failures to zero
Initlallze student counter to one
while student counter is less than or equal to ten
Input the next exam result
if the student Passed
Add one to passes
else
Add one to failures
Add one to student counter
Priht the number of passes
Priht the number of filures
if more than eight students passed
Print"Raise tuition"
```

图 6.10 检查考试成绩的伪代码

这个伪代码语句已经足以转换为 C++ 程序。图 6.11 显示了 C++ 程序及示例的执行结果。注意，我们利用 C++ 的一个特性，可以在声明中进行变量初始化。循环程序可能在每次循环开头要求初始化，这种初始化通常在赋值语句中进行。

```
1 // Fig. 2.11: fig02_1l.cpp
2 #include <iostream.h>
3
4
```

```

5 int main()
6 {
    // initialize variables in declarations
    int passes = 0,          // number of passes
        Passes = v;        // number or passes
        failures = 0,       // number of failures
        studentCounter = 1, // student counter
        result;             // oue exam result
    // process 10 students; counter-controlled loop
    while ( studentCounter <= 10 ) {
        cout << "Enter result (1=pass,2=fail): ";
        cin >> result;
        if ( result == 1 )      // if/else nested in while
            passes = passes + 1;
        else
            failures = failures + 1;
        studentcounter = studentCounter + 1;
    }
    // termination phase
    cout << "Failed" << failures << endl;
    if ( passes > 8 )
        cout << "Raise tuition " << endl;
    return 0; // successful termination
}

```

输出结果:

```

Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail)  1
Enter result (1=pass,2=fail): 2
Enter result {1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result )1=pass,2=fail): 2
Passed 6
Failed 4

```

```

Enter result (a=pass,2=Fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail)  1
Enter result (1-pass,2=fail): 2
Enter result {1=pass,2=fail): 1

```

```

Enter result (1=pass,2=fail): 1
Enter result {1=Pass,2=fail): 1
Enter result(1=pass, 2=fail): 1
Enter result(1=pass, 2=fail): 1
Enter result(1=pass, 2=fail): 1
Passed 9
Failed 1
Raise tuition

```

图 6.11 检查考试成绩的 C++程序及示例执行结果

6.10 赋值运算符

C++提供了几个赋值运算符可以缩写赋值表达式。例如下列语句：

```
c = c + 3;
```

可以用加法赋值运算符(addition assignment operator) “+=” 缩写如下：

```
c += 3;
```

+=运算符将运算符右边表达式的值与运算符左边表达式的值相加，并将结果存放在运算符左边表达式的值中。下列形式的语句：

```
variable = variable operator expression;
```

其中 operator 为二元运算符+、-、/或%之一(或今后要介绍的其他二元运算符)，均可写成如下形式：

```
variable operator = expression;
```

这样，赋值语句 c+=3 将 3 与 c 相加。图 2.12 显示了算术赋值运算符、使用这些算术赋值运算符的示例表达式和说明。

赋值运算符	示例表达式	说明	赋值
假设 int c=3,d=5,e=4,f=6,g=12;			
+=	e+=7	c=c+7	10 赋值给 e
-=	d-=4	d=d-4	1 赋值给 d
=	e=5	e=e*5	20 赋值给 e
/=	f/=3	f=f/3	2 赋值给 f
%=	g%=9	g=g%9	3 赋值给 g

图 6.12 算术赋值运算符

6.11 自增和自减运算符

C++还提供一元自增运算符(increment operator,++)和一元自减运算符(decrement operator)，见图 6.13。如果变量 c 递增 1，则可以用自增运算符++，而不用表达式 c=c+1 或 c+=1。如果将自

增和自减运算符放在变量前面，则称为前置自增或前置递减运算符 (preincrement 或 predecrement operator)。如果将自增和自减运算符放在变量后面，则称为后置自增或后置自减运算符 (postincrement 或 postdecrement operator)。前置自增(前置自减)运算符使变量加 1(减 1)，然后在表达式中用变量的新值。后置自增(后置自减)运算符在表达式中用变量的当前值，然后再将变量加 1(减 1)。

运算符	名称	示例表达式	说明
++	前置自增	++a	将 a 加 1，然后在 a 出现的表达式中使用新值
++	后置自增	a++	在 a 出现的表达式中使用当前值，然后将 a 加 1
--	前置自减	--b	将 b 减 1，然后在 b 出现的表达式中使用新值
--	后置自减	b--	在 b 出现的表达式中使用当前值，然后将 b 减 1

图 6.13 自增和自减运算符

图 6.14 的程序演示了++运算符的前置自增与后置自增计算之间的差别，后置自增变量 c 使其在输出语句中使用之后再递增，而前置自增变量 c 使其在输出语句中使用之前递增。

```
1 // Fig. 2.14:fig02 14.cpp
2 // Preincrementing and postincrementing
3 #include <iostream.h>
4
5 int main()
6 {
7     int c;
8
9     C = 5;
10    cout << C << endl;           // print 5
11    cout << C++ << endl;         // print 5 then postincrement
12    cout << c << endl << endl; // print
13
14    c = 5;
15    cout << c << endl;           // print 5
16    cout << ++c << endl;         // preincrement then print 6
17    cout << c << endl;
18
19    return 0;    // successful termination
20 }
```

输出结果:

```
5
5
6

5
```

6
6

图 6.14 前置自增与后置自增计算之间的差别

程序显示使用++运算符前后的 c 值，自减运算符的用法类似。

图 6.11 的三个赋值语句：

```
passes=passes+1;
failures=failures+1
student=student+1;
```

可以改写成更简练的赋值运算符形式：

```
passes+=1;
failures+=1;
student+=1;
```

使用前置自增运算符，如下所示：

```
++passes;
++failures;
++student;
```

或使用后置自增运算符，如下所示：

```
passes++
failures++
student++
```

注意，单独一条语句中自增或自减变量时，前置自增与后置自增计算之间的的结果一样，前置自减与后置自减计算之间的结果也相同。只有变量出现在大表达式中时，才能体现前置自增与后置自增计算之间的差别(和前置自减与后置自减计算之间的差别)。

目前只用简单变量名作为自增和自减的操作数(稍后会介绍，这些运算符也可以用于左值)。

图 6.15 显示了前面所介绍的运算符优先级和结合律，从上到下，优先级依次递减。第二栏介绍每一级运算符的结合律，注意条件运算符(?:)、一元运算符自增(++)、自减(--)、正(+)、负(-)、强制类型转换以及赋值运算符(=、+=、-=、*=、/=和%=)的结合律为从右向左。图 1.15 中所有其他运算符的结合律为从左向右。第三栏是运算符的组名。

运算符	结合律	类型
()		括号
++ -- + - static_cast<type>()	从左向右	一元
* / %	从右向左	乘
+ -	从左向右	加
<< >>	从左向右	插入/读取
< <= > >=	从左向右	关系
== !=	从左向右	相等
?:	从右向左	条件

=	+=	-=	*=	/=	%=	从右向左	赋值
,						从左向右	逗号

图 6.15 前面所介绍的运算符优先级和结合律

6.12 计数器控制循环的要点

计数器控制循环要求：

- 1.控制变量(或循环计数器)的名称(name)。
- 2.控制变量的初始值(initial value)。
- 3.测试控制变量终值(final value)的条件(即是否继续循环)。
- 4.每次循环时控制变量修改的增量或减量(increment decrement)。

考虑图 6.16 所示的简单程序，打印 1 到 10 的数字。声明：

```
int counter = 1;
```

指定控制变量(counter)并声明为整数，在内存中为其保留空间并将初始值设置为 1。需要初始化的声明实际上是可执行语句。在 C++中，将需要分配内存的声明称为定义(definition)更准确。

```
1 // Fig. 2.16: fig02_16.cpp
2 // Counter-controlled repetition
3 #include <iostream.h>
4
5 int main()
6 {
7     int counter = 1;           // initialization
8
9     while ( counter <= 10 ) {  // repetition condition
10         cout << counter << endl;
11         ++counter;             // increment
12     }
13
15 }
```

输出结果：

```
2
4
5
0
8
9
10
```

图 6.16 计数器控制循环

counter 的声明和初始化也可以用下列语句完成:

```
int counter;  
counter=1;
```

声明不是可执行语句, 但赋值是可执行语句。我们用两种方法将变量初始化。

下列语句:

```
++counter;
```

在每次循环时将循环计数器的值加 1。while 结构中的循环条件测试控制变量的值是否小于或等于 10(条件为 true 的终值)。注意, 即使控制变量是 10 时, 这个 while 结构体仍然执行。控制变量超过 10 时(即 counter 变成 11 时), 循环终止。

图 6.16 的程序也可以更加简化, 将 counter 初始化为。并将 while 结构换成:

```
while (++counter <= 10)  
    cout << counter << endl;
```

这段代码减少了语句, 直接在 while 条件中先增加计数器的值再测试条件。这段代码还消除了 while 结构体的花括号, 因为这时 while 只包含一条语句。

6.13 for 重复结构

for 重复结构处理计数器控制循环的所有细节。要演示 for 的功能, 可以改写图 6.16 的程序, 结果如图 6.17。

执行 for 重复结构时, 声明控制变量 counter 并将其初始化为 1。然后检查循环条件 counter<=10。由于 counter 的初始值为 1, 因此条件满足, 打印 Counter 的值(1)。然后在表达式 Counter++中递增控制变量 counter, 再次进行循环和测试循环条件。由于这时控制变量等于 2, 没有超过最后值, 因此程序再次执行语句体。这个过程一直继续, 直到控制变量 counter 递增到 11, 使循环条件的测试失败, 重复终止。程序继续执行 for 结构后面的第一条语句(这里是程序末尾的 return 语句)。

```
1 // Fig. 2.17:fig02 17.cpp  
2 // Counter-controlled repetition with the for structure  
3 #include <iostream.h>  
4  
5 int main()  
6 {  
7     // Initialization, repetition condition, and incrementing  
8     // are all included in the for structure header.  
9  
10    for ( iht counter = 1; counter <= 10; counter++ )  
11        cout << counter << endl;  
12  
13    return O;  
14 }
```

图 6.17 用 for 结构的计数器控制重复

图 6.18 更进一步研究了图 6.17 中的 for 结构。注意 for 结构指定计数器控制重复所需的每个项目。如果 for 结构体中有多条语句，则应把语句体放在花括号中。

注意图 6.17 用循环条件 `counter<=10`。如果循环条件变为 `counter<10`，则循环只执行 9 次，这种常见的逻辑错误称为差 1 错误。

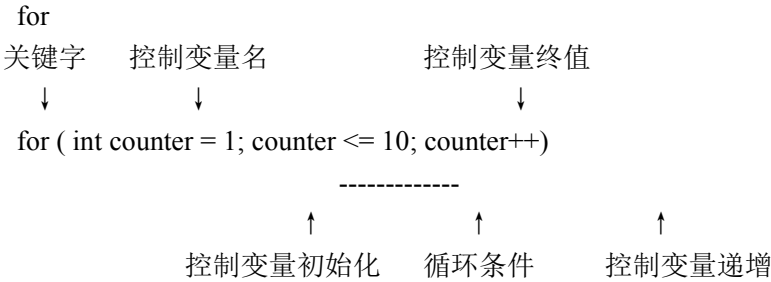


图 6.18 典型 for 首部的组件

for 结构的一般格式如下：

```
for (expression1; expression2; expression3)
    statement
```

其中 expression1 初始化循环控制变量的值，expression2 是循环条件，expression3 递增控制变量。大多数情况下，for 结构可以表示为等价的 while 结构：

```
expression1
while (expression2) {
    statement
    expression3;
}
```

如果 for 结构首部中的 expression1(初始化部分)定义控制变量(即控制变量类型在变量名前面指定)，则该控制变量只能在 for 结构体中使用，即控制变量值是 for 结构之外所未知的。这种限制控制变量名的用法称为变量的作用域(scope)。变量的作用域定义其在程序中的使用范围。

for 结构中的三个表达式是可选的。如果省略 expression2，则 C++ 假设循环条件为真，从而生成无限循环。如果程序其他地方初始化控制变量，则可以省略 expression1。如果 for 语句体中的语句计算增量或不需要增量，则可以省略 expression3。for 结构中的增量表达式就像是 for 语句体末尾的独立语句。因此，下列表达式：

```
counter = counter + 1;
counter += 1;
++counter;
counter++;
```

在 for 结构的递增部分都是等价的。许多程序员喜欢 counter++，因为递增在执行循环体之后发生，因此，后置自增形式似乎更自然。由于这里递增的变量没有出现在表达式中，因此前置自增与后置自增的效果相同。for 结构首部中的两个分号是必需的。

for 结构的初始化、循环条件和递增部分可以用算术表达式。例如，假设 x=2 和 y=10，如果

x 和 y 的值在循环体中不被修改，则下列语句：

```
for (int j= x; j <= 4* x* y; j += y/ x )  
等于下列语句：
```

```
for(int j=2; j<=80; j+=5)  
for 结构的增量也可能是负数(实际上是递减，循环向下计数)。
```

如果循环条件最初为 false，则 for 结构体不执行，执行 for 后面的语句。

for 结构中经常打印控制变量或用控制变量进行计算，控制变量常用于控制重复而不在 for 结构中提及这些控制变量。

for 结构的流程图与 while 结构相似。例如，图 6.19 显示了下列 for 语句的流程图：

```
for(int counter=1; counter>=10; counter++)  
cout<< counter<< endl;
```

从这个流程图可以看出初始化发生一次，并在每次执行结构体语句之后递增。注意，流程图(除了小圆框和流程之外)也只能包含表示操作的矩形框和表示判断的菱形框。这是我们强调的操作/判断编程模型。程序员的任务就是根据算法使用堆栈和嵌套两种方法组合其他几种控制结构，然后在这些框中填入算法所要的操作和判断，从而生成程序。

6.14 for 结构使用举例

下面的例子显示 for 结构中改变控制变量的方法。在每个例子中，我们都编写相应的 for 结构首部。注意循环中递减控制变量的关系运算符的改变。

a)将控制变量从 1 变到 100，增量为 1。

```
for(int i=1; i<=100; i++)
```

b)将控制变量从 100 变到 1，增量为-1。

```
for(int i=100; i>=1; i--)
```

c)控制变量的变化范围为 7 到 77。

```
for(int i= 7; i <= 77; i+= 7)
```

d)控制变量的变化范围为 20 到 2。

```
for(int i=20; i>=2; i-=2)
```

e)按所示数列改变控制变量值：2、6、8、11、14、17、20。

```
for(int j=2; j<=20; j+=3)
```

f)按所示数列改变控制变量值：99、88、77、66、55、44、33、22、11、0。

```
for(int j=99; j>=0; j-=11)
```

下面两个例子提供 for 结构的简单应用。图 2.20 所示的程序用 for 结构求 2 到 100 的所有整数的总和。

注意图 2.20 中 for 结构体可以用下列逗号运算符合并成 for 首部的右边部分：

```
for(int number = 2;                               // initialization  
    number <= 100;                                // continuation condition  
    sum += number, number += 2)                    // total and increment
```

初始化 sum=0 也可以合并到 for 的初始化部分。

```
1 // Fig. 2.20: fig02_20.cpp  
2 // Summation with for
```

```

3 #include <iostream.h>
4
5 int main()
6 {
7     int sum = 0;
8
9     for (int number = 2; number <= 100; number += 2 )
10        sum += number;
11
12    cout << "Sum is " << sum << endl;
13
14    return 0;
15 }

```

输出结果:

sum is 2550

图 6.20 用 for 语句求和

下列用 for 结构计算复利。考虑下列问题:

一个人在银行存款 1000.00 美元, 每利率为 5%。假设所有利息留在账号中, 则计算 10 年间每年年末的金额并打印出来。用下列公式求出金额:

$$a = P(1+r)^n$$

其中:

P 是原存款(本金)

r 是年利率

n 是年数

a 是年末本息

这个问题要用一个循环对 10 年的存款进行计算。解答如图 6.21

for 结构执行循环体 10 次, 将控制变量从 1 变到 10, 增量为 1。C++中没有指数运算符, 因此要用标准库函数 pow。函数 pow(x, y)计算 x 的 y 次方值。函数 pow 取两个类型为 double 的参数并返回 double 值。类型 double 与 float 相似, 但 double 类型的变量能存放比 float 精度更大的数值。C++把常量(如图 6.21 中的 1000.0 和 .05)当作 double 类型处理。

```

1 // Fig. 2.21: fig02_21.cpp
2 // Calculating compound interest
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <math.h>
6
7 int main()
8 {
9     double amount,           // amount on deposit
10        principal = 1000.0,   // starting principal
11        rate = .05;           // interest rate
12

```

```

13  cout << "Year" << setw( 21 )
14      << "Amount on deposit" << endl;
15
16  for ( int year = 1; year <= 10; year++ ) {
17      amount = principal * pow{ 1.0 + rate, year };
18      cout << setw( 4 ) << year
19          << setiosflags( ios::fixed | ios::showpoint )
20          << setw( 21 ) << setprecision( 2 )
21          << amount << endl;
22  }
23
24
25 }

```

输出结果：

Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.62
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

图 6.21 用 for 结构计算复利

这个程序必须包括 `math.h` 才能编译。函数 `pow` 要求两个 `double` 参数，注意 `year` 是个整数。`math.h` 文件中的信息告诉编译器将 `year` 值转换为临时 `double` 类型之后再调用函数。这些信息放在 `pow` 的函数原型(function prototype)中。第 3 章将介绍函数原型并总结 `pow` 函数和其他数学库函数。

程序中将变量 `amount`、`principal` 和 `rate` 声明为 `double` 类型，这是为了简单起见，因为我们要涉及存款数额的小数部分，要采用数值中允许小数的类型。但是，这可能造成麻烦，下面简单介绍用 `float` 和 `double` 表示数值时可能出现的问题(假设打印时用 `setprecision(2)`)：机器中存放的两个 `float` 类型的值可能是 14.234(打印 14.23)和 18.673(打印 18.67)。这两个值相加时，内部和为 32.907，打印 32.91。结果输出如下：

```

14.23
+ 18.67
-----
32.91

```

但这个加式的和应为 32.90。

输出语句:

```
cout<<setw(4)<<year
<<setiosflags(10s::fixed | 10s::showpoint)
<<  setw(21) << setprecision(2)
<<  amount<<endl;
```

用参数化流操纵算子 `setw`、`setiosflags` 和 `setprecision` 指定的格式打印变量 `year` 和 `amount` 的值。调用 `setw(4)` 指定下一个值的输出域宽(field width)为 4, 即至少用 4 个字符位置打印这个值。如果输出的值宽度少于 4 个字符位, 则该值默认在输出域中右对齐(right justified), 如果输出的值宽度大于 4 个字符, 则该值将输出域宽扩大到能放下这个值为止。调用 `setiosflag(10s::left)` 可以指定输出的值为左对齐(left justified)。

上述输出中的其余格式表示变量 `amount` 打印成带小数点的定点值(用 `setiosflags(10s::fixed | 10s::showpoint)` 指定), 在输出域的 21 个字符位中右对齐(用 `setw(21)` 指定), 小数点后面为两位(用 `setprecision(2)` 指定)。

注意计算 `1.0+rate` 作为 `pow` 函数的参数, 包含在 `for` 语句体中。事实上, 这个计算产生的结果在每次循环时相同, 因此是重复计算(是一种浪费)。

6.15 switch 多项选择结构

前面介绍了 `if` 单项选择结构和 `if/else` 双项选择结构。有时算法中包含一系列判断, 用一个变量或表达式测试每个可能的常量值, 并相应采取不同操作。C++ 提供的 `switch` 多项选择结构可以进行这种判断。

`switch` 结构包括一系列 `case` 标记和一个可选 `default` 情况。图 6.22 中的程序用 `switch` 计算学生考试的每一级人数。

```
1 // Fig. 2.22:fig02 22.cpp
2 // Counting letter grades
3 #include <iostream.h>
4
5 int main()
6 {
7     int grade,          // one grade
8     aCount = 0,         // number of A's
9     bCount = 0,         // number of B's
10    cCount = 0,         // number of C's
11    dCount = 0,         // number of D's
12    fCount = 0;         // number of F's
13
14    cout << "Enter the letter grades." << endl
15         << "Enter the EOF character to end input." << endl;
16
17    while ( ( grade = cin.get() ) != EOF ) {
```

```

18
19     switch ( grade ) {          // switch nested in while
20
21         case 'A': // grade was uppercase a
22         case 'a': // or lowercase a
23             ++aCount;
24             break; // necessary to exit switch
25
26         case 'B': // grade was uppercase B
27         case 'b': // or lowercase b
28             ++bCount;
29             break;
30
31         case 'C': // grade was uppercase C
32         case 'c': // or lowercase c
33             ++cCount;
34             break;
35
36         case 'D': // grade was uppercase D
37         case 'd': // or lowercase d
38             ++dCount;
39             break;
40
41         case 'F': // grade was uppercase F
42         case 'f': // or lowercase f
43             ++fCount;
44             break;
45
46         case '\n': // ignore newlines,
47         case '\t': // tabs,
48         case ' ': // and spaces in input
49             break;
50
51         default: // catch all other characters
52             cout << "Incorrect letter grade entered."
53
54             << "Enter a new grade." << endl;
55             break; // optional
56     }
57 }
58 cout << "\n\nTotals for each letter grade are:"
59     "\nA: " << aCount
60     << "\nB: " << bCount
61     << "\nC: " << cCount

```

```

62             << "\nD: " << dCount
63             << "\nF: " << fCount << endl;
64
65     return 0;
66 }

```

输出结果:

Enter the letter grades.

Enter the EOF character to end input.

```

A
B
C
C
A
D
F
C
E

```

Incorrect letter grade entered. Enter a new grade.

```

D
A
B

```

Totals for each letter grade are:

```

A: 3
B: 2
C: 3
D: 2
F: 1

```

图 6.22 使用 switch 举例

程序中，用户输入一个班代表成绩的字母。在 while 首部中：

```
while( (grade=cin.get()) !=EOF)
```

首先执行参数化赋值(`grade=cin.get()`)。`cin.get()`函数从键盘读取一个字符，并将这个字符存放在整型变量 `grade` 中。`cin.get()`中使用的圆点符号将在第 6 章中介绍。字符通常存放在 `char` 类型的变量中，但是，C++的一个重要特性是可以用任何整数数据类型存放字符，因为它们在计算机中表示为 1 个字节的整数。这样，我们根据使用情况，可以把字符当作整数或字符。例如，下列语句：

```
cout << "The character(" <<'a'<< ") has the value "
<< static_cast<int>('a')<< endl;
```

打印字符 `a` 及其整数值如下所示：

The character (a) has the value 97

整数 97 是计算机中该字符的数字表示。如今许多计算机都使用 ASCII(AmericanStandardCodefor

InformationInterehange, 美国标准信息交换码)字符集(character set), 其中 97 表示小写字母 “a”。附录中列出了 ASCII 字符及其十进制值的列表。

赋值语句的整个值为等号左边变量指定的值。这样,赋值语句 `grade=cin.get()` 的值等于 `cin.get()` 返回的值, 赋给变量 `grade`。赋值语句可以用于同时初始化多个变量。例如:

```
a = b = c = 0;
```

首先进行赋值 `c=0`(因为: 运算符从右向左结合), 然后将 `c=0` 的值赋给变量 `b`(为 0), 最后将 `b=(c=0)` 的值赋给变量 `a`(也是 0)。程序中, 赋值语句 `grade=cin.get()` 的值与 EOF 值(表示文件结束的符号)比较。我们用 EOF(通常取值为-1)作为标记值。用户输入一个系统识别的组合键, 表示文件结束, 没有更多要输入的数据。EOF 是 `<iostream.h>` 头文件中定义的符号化整型常量。如果 `grade` 的取值为 EOF, 则程序终止。我们选择将这个程序中的字符表示为 `int`, 因为 EOF 取整数值(通常取值为-1)。

用户通过键盘输入成绩。按 Enter(或 Return)键时, `cin.get()` 函数一次读取一个字符。如果输入的字符不是文件结束符, 则进入 `switch` 结构。关键字 `switch` 后面是括号中的变量名 `grade`, 称为控制表达式(controlling expression)。控制表达式的值与每个 `case` 标记比较。假设用户输入成绩 `c`, 则 `c` 自动与 `switch` 中的每个 `case` 比较。如果找到匹配的(`case'C:'`), 则执行该 `case` 语句。对于字母 C 的 `case` 语句中, `cCount` 加 1, 并用 `break` 语句立即退出 `switch` 结构。注意, 与其他控制结构不同的是, 有多个语句的 `case` 不必放在花括号中。

`break` 语句使程序控制转到 `switch` 结构后面的第一条语句。`break` 语句使 `switch` 结构中的其他 `case` 不会一起运行。如果 `switch` 结构中不用 `break` 语句, 则每次结构中发现匹配时, 执行所有余下 `case` 中的语句(这个特性在几个 `case` 要完成相同操作时有用, 见图 2.22 的程序)。如果找不到匹配, 则执行 `default case` 并打印一个错误消息。

每个 `case` 有一个或几个操作。`switch` 结构与其他结构不同, 多个语句的 `case` 不必放在花括号中。图 6.23 显示了一般的 `swish` 多项选择结构(每个 `case` 用一个 `break` 语句)的流程图。

从流程图中可以看出, `case` 末尾的每个 `break` 语句使控制立即退出 `switch` 结构。注意, 流程图(除了小圆框和流程之外)也只能包含矩形框和菱形框。这是我们强调的操作/判断编程模型。程序员的任务就是根据算法需要用堆栈和嵌套两种方法组合其他几种控制结构, 然后填入算法所要的操作和判断, 从而生成程序。嵌套控制结构很常见, 但程序中很少出现嵌套 `swilch` 结构。

在图 6.22 中的 `switch` 结构中, 第 46 到第 49 行:

```
case '\n':  
case '\t':  
case ' ' :  
break;
```

使程序跳过换行符、制表符和空白字符。一次读取一个字符可能造成一些问题。要让程序读取字符, 就要按键盘上的 Enter 键将字符发送到计算机中, 在要处理的字符后面输入换行符。这个换行符通常需要特殊处理, 才能使程序正确工作。通过在 `switch` 结构中包含 `case` 语句, 可以防止在每次输入中遇到换行符、制表符或空格时 `default case` 打印错误消息。

注意几个标号列在一起时(如图 6.22 中的 `case'D': case'd':`)表示每 `case` 发生一组相同的操

作。

使用 switch 结构时，记住它只用于测试常量整型表达式(constant integral expression)，即求值为一个常量整数值字符常量和整型常量的组合。字符常量表示为单引号中的特定字符(如, 'A')，而整型常量表示为整数值。

本教程介绍面向对象编程时，会介绍实现 switch 逻辑的更精彩方法。我们使用多态方法生成比使用 switch 逻辑的程序更清晰、更简洁、更易维护和扩展的程序。

C++之类可移植语言应有更灵活的数据类型长度，不同应用可能需要不同长度的整数。C++提供了几种表示整数的数据类型。每种类型的整数值范围取决于特定的计算机硬件。除了类型 int 和 char 外，C++还提供 short(short int 的缩写)和 long(long int 的缩写)类型。short 整数的取值范围是±32767。

对于大多数整数计算，使用 long 类型的整数已经足够。long 整数的取值范围是±2147483647。在大多数系统中，int 等价于 short 或 long。int 的取值范围在 short 和 long 的取值范围之间。char 数据类型可以表示计算机字符集中的任何字符，char 数据类型也可以表示小整数。

6.16 do/while 重复结构

do/while 重复结构与 while 结构相似。在 while 结构中，先在循环开头测试循环条件之后再执行循环体。do/while 重复结构执行循环体之后再测试循环条件，因此，do/while 结构至少执行循环体一次。do/while 结构终止时，继续执行 while 语句后面的语句。注意，如果结构体中只有一条语句，则不必在 do/while 结构中使用花括号。但通常还是加上花括号，避免混淆 while 与 do/while 重复结构。

例如：

```
while (condition)
```

通常当作 while 结构的首部。结构体中只有一条语句的 do/while 结构中不使用花括号时：

```
do
```

```
statement
```

```
while ( condition );
```

最后一行 while(condition)可能被误解成 while 结构包含空语句。这样，只有一个语句的 do/while 结构通常写成如下形式：

```
do {
```

```
statement
```

```
}while ( condltion);
```

图 6.24 所示的程序用 do/while 重复结构打印数字 1 到 10。注意控制变量 counter 在循环条件测试中是前置自增的。另外，只有一个语句的 do/while 结构也使用了花括号。

```
1 // Fig. 2.24: fig0224.cpp
2 // Using the do/while repetition structure
3 #include <iostream.h>
4
5 int main()
6 {
7     int counter = 1;
8
```

```

9   do {
10      cout << counter << " " ;
11   } while ( ++counter <= 10 );
12
13   cout << endl;
14
15   return 0;
16 }

```

输出结果:

```

1   2   3   4   5   6   7   8   9   10

```

图 6.24 使用 do/while 重复结构

do/while 重复结构如图 6.25。这个流程图显示循环条件要在至少进行一次操作之后才执行。注意，流程图(除了小圆框和流程之外)也只能包含矩形框和菱形框，这是我们强调的操作/判断编程模型。程序员的任务就是根据算法需要用堆栈和嵌套两种方法组合其他几种控制结构，然后填入算法所要的操作和判断，从而生成程序。

6.17 break 和 continue 语句

break 和 continue 语句改变控制流程。break 语句在 while、for、do/while 或 switch 结构中执行时，使得程序立即退出这些结构，从而执行该结构后面的第一条语句。break 语句常用于提前从循环退出或跳过 switch 结构的其余部分(如图 6.22)。图 6.26 演示了 for 重复结构中的 break 语句，if 结构发现 x 变为 5 时执行 break，从而终止 for 语句，程序继续执行 for 结构后面的 cout 语句。循环只执行四次。

注意这个程序中控制变量 x 在 for 结构首部之外定义。这是因为我们要在循环体中和循环执行完毕之后使用这个控制变量。

continue 语句在 while、for 或 do/while 结构中执行时跳过该结构体的其余语句，进入下一轮循环。在 while 和 do/while 结构中，循环条件测试在执行 continue 语句之后立即求值。在 for 结构中，执行递增表达式，然后进行循环条件测试。前面曾介绍过，while 结构可以在大多数情况下取代 for 结构。但如果 while 结构中的递增表达式在 continue 语句之后，则会出现例外。这时，在测试循环条件之前没有执行递增，并且 while 与 for 的执行方式是不同的。图 6.27 在 for 结构用 continue 语句跳过该结构的输出语句，进入下一轮循环。

```

1 // Fig. 2. 26: fig02 26.cpp
2 // Using the break statement in a for structure
3 #include<iostream. h>
4
5 int main()
6 {
7     // x declared here so it can be used after the loop
8     int x;
9

```

```

10    for ( x = 1; x <= 10; x++ ) {
11
12    if { x == 5 )
13        break;    // break loop only if x is 5
14
15    cout << x << " ";
16    }
17
18    cout << "\nBroke out of loop at x of" << x << endl;
19    return O;
20 }

```

输出结果:

```

1  2  3  4
Broke out of loop at x of 5

```

图 6.26 for 重复结构中的 break 语句

```

1  // Fig. 2.27: figO2_OT.cpp
2  // Using the continue statement in a for structure
3  #include <iostream.h>
4
5  int main()
6  {
7      for ( iht x = 1; x <= 10; x++ ) {
8
9          if {x==5)
10             continue; // skip remaining code in loop
11                     // only if x is 5
12
13             cout << x << " ";
14         }
15
16         cout << "\nUsed continue to skip printing the value 5"
17             << endl;
18         return O;
19 }

```

```

1  2  3  4  5  6  7  9  9  10
Used continue to skip printing the value 5

```

图 6.27 在 for 结构用 continue 语句

6.18 逻辑运算符

前面只介绍了 `counter<=10`、`total>1000` 和 `number!=sentinel Value` 之类的简单条件 (simplecondition)。我们用关系运算符 `>`、`<`、`>=`、`<=` 和相等运算符 `==`、`!=` 表示这些条件。每个判断只测试一个条件。要在每个判断中测试多个条件，可以在不同语句中或嵌套 `if(if/else)` 结构中进行这些测试。

C++ 提供的逻辑运算符 (logical operator) 可以用简单条件组合成复杂条件。逻辑运算符有逻辑与 (`&&`)、逻辑或 (`||`) 和逻辑非 (`!`)，下面将举例说明。

假设要保证两个条件均为 `true` 之后再选择某个执行路径，这时可以用 `&&` 逻辑运算符：

```
if { gender == 1 && age >= 65 }
    ++senior Females;
```

这个 `if` 语句包含两个简单条件。条件 `gender==1` 确定这个人是男是女，条件 `age>=65` 确定这个人是否为老年人。`&&` 逻辑运算符左边的简单条件先求值，因为 `::` 的优先级高于 `&&`。如果需要，再对 `&&` 逻辑运算符右边的简单条件求值，因为 `>=` 的优先级高级于 `&&` (稍后将会介绍，`&&` 逻辑运算符右边的条件只在左边为 `true` 时才求值)。然后 `if` 语句考虑下列组合条件：

```
gender==1&&age>=65
```

如果两边的简单条件均为 `true`，则这个条件为 `true`。最后，如果这个条件为 `true`，则 `seniorFemales` 递增 1。如果两边的简单条件有一个为 `false`，则程序跳过该递增，处理 `if` 后面的语句。上述组合条件可以通过增加多余的括号而变得更加清楚：

```
(gender==1)&&(age>=65)
```

图 6.28 的表格总结了 `&&` 逻辑运算符。表中显示了表达式 1 和表达式 2 的四种 `false` 和 `true` 值组合，这种表称为真值表 (truth table)。C++ 对所有包括逻辑运算符、相等运算符和关系运算符的所有表达式求值为 `false` 或 `true`。

表达式 1	表达式 2	表达式 1&&表达式 2
false	false	false
false	true	false
true	false	false
true	true	true

图 6.28 逻辑与(&&)运算符真值表

下面考虑逻辑或运算符 (`||`)。假设要保证两个条件至少有一个为 `true` 之后再选择某个执行路径，这时可以用逻辑或运算符，如下列程序段所示：

```
if(semesterAverage>=90 || finalExam>=90)
    cout<<"Student grade is A"<<endl;
```

上述条件也包含两个简单条件。条件 `semesterAverage>=90` 确定学生整个学期的表现是否为 "A"，条件 `finalExam>=90` 确定该生期末考试成绩是否优秀。然后 `if` 语句考虑下列组合条件：

```
semesterAverage>=90 || finalExam>=90
```

如果两个简单条件至少有一个为 `true`，则该生评为 "A"。注意只有在两个简单条件均为 `false` 时才不打印消息 "Student grade is A"。图 6.29 是逻辑或 (`||`) 运算符的真值表。

`&&` 逻辑运算符的优先级高于 `||` 运算符。这两个运算符都是从左向右结合。包含 `&&` 和 `||` 运算符

的表达式只在知道真假值时才求值。这样，求值下列表达式：

`gender==1 && age>=65`
将在 `gender` 不等于 1 时立即停止(整个表达式为 `false`)，而 `gender` 等于 1 时则继续求值(整个表达式在 `age>=65` 为 `true` 时为 `true`)。

表达式 1	表达式 2	表达式 1 && 表达式 2
false	false	false
false	true	true
true	false	true
true	true	true

图 6.29 逻辑或(`||`)运算符真值表

C++提供了逻辑非(`!`)运算符，使程序员可以逆转条件的意义。与`&&`和`||`运算符不同的是，`&&`和`||`运算符组合两个条件(是二元运算符)，而逻辑非(`!`)运算符只有一个操作数(是一元运算符)。逻辑非(`!`)运算符放在条件之前，在原条件(不带逻辑非运算符的条件)为 `false` 时选择执行路径,例如：

```
if(grade!=sentinelValue)
    cout<<"The next grade is"<<grade<<endl;
```

这种灵活性有助于程序员以更自然或更方便的方式表达条件。

图 6.31 显示了前面介绍的 c++运算符的优先级和结合律。运算符优先级从上到下逐渐降低。

运算符	结合律	类型
()	从左向右	括号
++ -- + - ! static_cast<type>()	从右向左	一元
* / %	从左向右	乘
+ -	从左向右	加
<< >>	从左向右	插入/读取
< <= > >=	从左向右	关系
== !=	从左向右	相等
&&	从左向右	逻辑与
	从左向右	逻辑或
?:	从右向左	条件
= += -= *= /= %=	从右向左	赋值
,	从左向右	逗号

图 6.31 运算符优先级和结合律

6.19 混淆相等(`==`)与赋值(`=`)运算符

这是 C++程序员常见的错误，包括熟练的 C++程序员也会把相等(`==`)与赋值(`=`)运算符相混淆。这种错误的破坏性在于它们通常不会导致语法错误，而是能够顺利编译，程序运行完后，因为运行时的逻辑错误而得到错误结果。

C++有两个方面会导致这些问题。一个是任何产生数值的表达式都可以用于任何控制结构的

判断部分。如果数值为 0，则当作 false，如果数值为非 0，则当作 true。第二是 C++ 赋值会产生一个值，即赋值运算符左边变量所取得的值。例如，假设把下列代码：

```
if(payCode==4)
    cout <<"You get a bonus!" << endl;
误写成如下形式：
if(payCode=4)
    cout <<  "You get a bonus!" << endl;
```

第一个 if 语句对 paycode 等于 4 的人发奖金。而第二个 if 语句则求值 if 条件中的赋值表达式为常量 4。由于非 0 值解释为 true，因此这个 if 语句的条件总是 true，则人人都获得一份奖金，不管其 paycode 为多少。更糟的是，本来只要检查 paycode，却已经修改了 Poycode。

变量名可称为左值(lvalue)，因为它可以放在赋值运算符左边。常量称为右值(rvalue)，因为它只能放在赋值运算符右边。注意，左值可以作为右值，但右值不能用作左值。

还有一个问题也同样麻烦。假设程序员要用下列简单语句给一个变量赋值：

```
x = 1;
但却写成：
x == 1;
```

这也不是语法错误，编译器只是求值条件表达式。如果 x 等于 1，则条件为 true，表达式返回 true 值。

如果 x 不等于 1，则条件为 false，表达式返回 false 值。不管返回什么值都没有赋值运算符，因此这个值丢失，x 值保持不变，有可能造成执行时的逻辑错误。但这个问题没有简单的解决办法。

6.20 结构化编程小结

结构化编程提倡简单性。Bohm 和 Jacopini 已经证明，只需要三种控制形式：

- 顺序(sequence)
- 选择(selection)
- 重复(repetmon)

顺序结构很简单，选择可以用三种方法实现：

- if 结构(单项选择)
- if/else 结构(双项选择)
- switch 结构(多项选择)

事实上很容易证明简单的 if 结构即可提供任何形式的选择，任何能用 if/else 结构和 switch 结构完成的工作，也可以组合简单 if 结构来实现(但程序可能不够流畅)。

重复可以用三种方法实现：

- while 结构
- do/while 结构
- for 结构

很容易证明简单的 while 结构即可提供任何形式的重复，任何能用 do/while 和 for 结构完成的工作，也可以组合简单 while 结构来实现(但程序可能不够流畅)。

根据以上结果，C++ 程序中所需的任何控制形式均可以用下列形式表示：

- 顺序

- if 结构(选择)
- while 结构(重复)

这些控制结构只要用两种方式组合，即嵌套和堆栈。事实上，结构化编程提倡简单性。

第7章 函数

7.1 简介

解决实际问题的大多数程序都比前几章介绍的程序要大得多。经验表明，要设计和修改大程序，最好的办法是从更容易管理的小块和小组件开始。这种方法称为“分而治之，各个击破”(divide and conquer)。本章介绍 C++ 语言中的许多关键特性，可以帮助设计、实现、操作和维护大程序。

7.2 数学函数库

数学函数库使程序员可以进行某些常见数学计算。我们这里用各种数学库函数介绍函数概念。本书稍后会介绍 c++ 标准库中的许多其他函数。

调用函数时，通常写上函数名，然后是一对括号，括号中写上函数参数(或逗号分隔的参数表)。例如程序员可以用下列语句计算和打印 900.0 的平方根：

```
cout << sqrt(900.0);
```

执行这个语句时，数学库函数 sqrt 计算括号中所包含数字(900.0)的平方根。数字 900.0 是 sqrt 函数的参数。上述语句打印 30。sqrtd 函数取 double 类型参数，返回 double 类型结果。数学函数库中的所有函数都返回 double 类型结果。要使用数学库函数，需要在程序中包含 math.h 头文件(这个头文件在新的 C++ 标准库中称为 cmath)。

函数参数可取常量、变量或表达式。如果 c1=13.0、d=3.0 和 f=4.0，则下列语句：

```
cout << sqrt(c1 + d * f);
```

计算并打印 $13.0 + 3.0 * 4.0 = 25.0$ 的平方根，即 5(因为 C++ 通常对没有小数部分的浮点数不打印小数点和后面的零)。

图 7.2 总结了一些数学库函数。图中变量 x 和 y 为 double 类型。

函数	说明	举例
ceil(x)	将 x 取整为不小于 x 的最小整数	ceil(9.2)=10.0 ceil(-9.8)=-9.0
cos(x)	x(弧度)的余弦	cos(0.0)=1.0
exp(x)	指数函数 e^x	exp(1.0)=2.71828 exp(2.0)=7.38906
fabs(x)	x 的绝对值	$x > 0, \text{fabs}(x) = x$ $x = 0, \text{fabs}(x) = 0.0$ $x < 0, \text{fabs}(x) = -x$
floor(x)	将 x 取整为不大于 x 的最大整数	floor(9.2)=9.0 floor(-9.8)=-10.0
fmod(x,y)	x/y 的浮点数余数	fmod(13.657,2.333)=1.992

<code>log(x)</code>	x 的自然对数(底数为 e)	<code>log(2.718282)=1.0</code> <code>log(7.389056)=2.0</code>
<code>log10(x)</code>	x 的对数(底数为 10)	<code>log(10.0)=1.0</code> <code>log(100.0)=2.0</code>
<code>pow(x,y)</code>	x 的 y 次方(xy)	<code>pow(2,7)=128</code> <code>pow(9,.5)=3</code>
<code>sin(x)</code>	x(弧度)的正弦	<code>sin(0.0)=0</code>
<code>sqrt(x)</code>	x 的平方根	<code>sqrt(900.0)=30.0</code> <code>sqrt(9.0)=3.0</code>
<code>tan(x)</code>	x(弧度的正切	<code>tan(0.0)=0</code>

7.3 函数

函数使程序员可以将程序模块化。函数定义中声明的所有变量都是局部变量(local variable), 只在所在的函数中有效。大多数函数有一系列参数, 提供函数之间沟通信息的方式。函数参数也是局部变量。

将程序函数化的目的有几个, “分而治之、各个击破”的方法使程序开发更容易管理。另一个目的是软件复用性(software reusability), 用现有函数作为基本组件, 生成新程序。软件复用性是面向对象编程的主要因素。有了好的函数命名和定义, 程序就可以由完成特定任务的标准化函数生成, 而不必用自定义的代码生成。第三个目的是避免程序中的重复代码, 将代码打包成函数使该代码可以从程序中的多个位置执行, 只要调用函数即可。

7.4 函数定义

前面介绍的每个程序都有一个 main 函数, 调用标准库函数完成工作。现在要考虑程序员如何编写自定义函数。

考虑一个程序, 用自定义函数, square 计算整数 1 到 10 的平方(如图 7.3)。

```

1 // Fig. 3.3: fig03_03.cpp
2 // Creating and using a programmer-defined function
3 #include <iostream.h>
4
5 int square( int ); // function prototype(函数原型)
6
7 int main( )
8 (
9     S
10    for ( in x = 1; x <= 10; x++ )
11    cout<< square( x ) << " " ;
12    cout << endl;
```

```

13    return 0;
14 }
15
16 // Function definition(函数定义)
17 int square( int y )
18 (
19     return y * y;
20 }

```

输出结果:

```

1    4    9    16    25  36    49    64    81    100

```

图 7.3 生成和使用自定义函数

main 中调用函数 square 如下所示:

```
square(x)
```

函数 square 在参数 y 中接收 x 值的副本。然后 square 计算 $y*y$ ，所得结果返回 main 中调用 square 的位置，并显示结果，注意函数调用不改变 x 的值。这个过程用 for 重复结构重复十次。

square 定义表示 square 需要整数参数 y。函数名前面的关键字 int 表示 square 返回一个整数结果。square 中的 return 语句将计算结果返回调用函数。

第 5 行:

```
int square(int);
```

是个函数原型(function prototype)。括号中的数据类型 int 告诉编译器，函数 square 要求调用者提供整数值。函数名 square 左边的数据类型 int 告诉编译器，函数 square 向调用者返回整数值。编译器通过函数原型检查 square 调用是否包含正确的返回类型、参数个数、参数类型和参数顺序。如果函数定义出现在程序中首次使用该函数之前，则不需要函数原型，这种情况下，函数定义也作为函数原型。如果图 33 中第 17 行到第 20 行在 main 之前，则第 5 行的函数原型是不需要的。函数原型将在第 7.6 节详细介绍。

函数定义格式如下:

```

return-value-type function-name(parameter-list)
{
    declarations and statements
}

```

函数名(function-name)是任何有效标识符，返回值类型(return-value-type)是函数向调用者返回值的类型，返回值类型 void 表示函数没有返回值。不指定返回值类型时默认为 int。

花括号中的声明(declaration)和语句(statement)构成函数体(function body)，函数体也称为块(block)，块是包括声明的复合语句。变量可以在任何块中声明，而且块也可以嵌套。任何情况下不能在一个函数中定义另一个函数。

将控制返回函数调用点的方法有三种。如果函数不返回结果，则控制在到达函数结束的右花括号时或执行下列语句时返回:

```
return;
```

如果函数返回结果，则下列语句:

return expression;

向调用者返回表达式的值。

第二个例子用自定义函数 `maximum` 确定和返回三个整数中的最大值(如图 7.4)。输入三个整数,然后将整数传递到 `maximum` 中,确定最大值。这个值用 `maximum` 中的 `return` 语句返回 `main`。返回的值赋给变量 `largest`, 然后打印。

```
1 // Fig. 7.4: fig0304.cpp
2 // Finding the maximum of three integers
3 #include <iostream.h>
4
5 int maximum( int, int, int ); // function prototype(函数原型)
6
7 int main( )
8 {
9     int a, b, c;
10
11     cout << "Enter three integers: ";
12     cin >> a >> b >> c;
13
14     // a, b and c below are arguments to
15     // the maximum function call(函数调用)
16     cout << "Maximum is: " << maximum( a, b, c ) << endl;
17
18     return 0;
19 }
20
21 // function maximum definition
22 // x, y and z below are parameters to
23 // the maximum function definition
24 int maximum( int x, int y, int z )
25 {
26     int max = x;
27
28
29
30
31     if { z > max )
32         max = z;
33
34     return max;
35 }
```

输出结果:

```
Enter three integers:  22  85  17
Maximum is:  85
```

```
Enter three integers:  92  35  14
```

```
Maximum is: 92

Enter three integers: 45 19 98
Maximum is: 98
```

图 7.4 自定义函数 maximum

7.5 头文件

每个标准库都有对应的头文件(header file), 包含库中所有函数的函数原型和这些函数所需各种数据类型和常量的定义。图 7.6 列出了 C++ 程序中可能包括的常用 C++ 标准库头文件。图 7.6 中多次出现的宏(macro)将在第 17 章“预处理器”中详细介绍。以 .h 结尾的头文件是旧式头文件。

对每个旧式头文件, 我们介绍新标准中使用的版本。

程序员可以生成自定义头文件, 自定义头文件应以.h 结尾。可以用#include 预处理指令包括自定义头文件。例如, square.h 头文件可以用下列指令:

```
#include "square.h"
```

放在程序开头。17.2 节介绍了包含头文件的其他信息。

旧式头文件	说明
旧式头文件(本书前面使用)	
<assert.h>	包含增加诊断以程序调试的宏和信息。这个头文件的新版本为<cassert>
<ctype.h>	包含测试某些字符属性的函数原型和将小写字母变为大写字母, 将大写字母变为小写字母的函数原型。这个头文件的新版本为<cctype>
<float.h>	包含系统的浮点长度限制。这个头文件的新版本为<cfloat>
<limits.h>	包含系统的整数长度限制。这个头文件的新版本为<climits>
<math.h>	包含数学库函数的函数原型。这个头文件的新版本为<cmath>
<stdio.h>	包含标准输入/输出库函数的函数原型及其使用信息。这个头文件的新版本为<cstdio>
<stdlib.h>	包含将数字变为文本、将文本变为数字、内存分配、随机数和各种其它工具函数的函数原型。这个头文件新版本为<cstdlib>
<string.h>	包含 C 语言格式的字符串处理函数的函数原型。这个头文件的新版本为<cstring>
<time.h>	包含操作时间和是期的函数原型和类型。这个头文件的新版本为<ctime>
<iostream.h>	包含标准输入/输出函数的函数原型。这个头文件的新版本为<iostream>
<iomanip.h>	包含能够格式化数据流的流操纵算子的函数原型。这个头文件的新版本为<iomanip>
<fstream.h>	包含从磁盘文件输入输出到磁盘文件的函数原型。这个头文件的新版本为<fstream>

标准库头文件	说明
<utility>	包含许多标准库头文件使用的类和函数
<vector>、<list>、	包含实现标准库容器的类的头文件。容器在程序执行期间用于存放数据。我们将在
<deque>、<queue>、	
<stack>、<map>、	“标准模板库”一章介绍这些头文件
<set>、<bitset>	
<functional>	包含用于标准库算法的类和函数
<memory>	包含用于向标准库容器分配内存的标准库使用的类和函数
<iterator>	包含标准库容器中操作数据的类
<algorithm>	包含标准库容器中操作数据的函数
<exception>	这些头文件包含用于异常处理的类(见第 13 章)
<stdexcept>	
<string>	包含标准库中 string 类的定义(见第 19 章)
<sstream>	包含从内存字符串输入和输出到内存字符串的函数原型
<locale>	包含通常在流处理中用于处理其它语言形式数据的类和函数(例如货币格式、
	排序字符串、字符表示等)
<limits>	包含定义每种计算机平台特定数字数据类型的类
<typeinfo>	包含运行时类型信息的类(在执行时确定数据类型)

图 7.6 常用 C++ 标准库头文件

7.6 作用域规则

程序中一个标识符有意义的部分称为其作用域。例如，块中声明局部变量时，其只能在这个块或这个块嵌套的块中引用。一个标识符的 4 个作用域是函数范围(function scope)、文件范围(filescope)、块范围(block scope)和函数原型范围(function-prototype scope)。后面还要介绍第五个——类范围(class scope)。

任何函数之外声明的标识符取文件范围。这种标识符可以从声明处起到文件末尾的任何函数中访问。全局变量、任何函数之外声明的函数定义和函数原型都取文件范围。

标号(后面带冒号的标识符，如 **start:**)是惟一具有函数范围的标识符。标号可以在所在函数中任何地方使用，但不能在函数体之外引用。标号用于 switch 结构中(如 case 标号)和 goto 语句中(见第 18 章)。标号是函数内部的实现细节，这种信息隐藏(information hiding)是良好软件工程的基本原则之一。

块中声明的标识符的作用域为块范围。块范围从标识符声明开始，到右花括号{ }处结束。函数开头声明的局部变量的作用域为块范围，函数参数也是，它们也是函数的局部变量。任何块都可以包含变量声明。块嵌套时，如果外层块中的标识符与内层块中的标识符同名，则外层块中的标识符“隐藏”，直到内层块终止。在内层块中执行时，内层块中的标识符值是本块中定义的，而不是同名的外层标识符值。声明为 static 的局部变量尽管在函数执行时就已经存在，但该变量

的作用域仍为块范围。存储时间不影响标识符的作用域。

只有函数原型参数表中使用的标识符才具有函数原型范围。前面曾介绍过，函数原型不要求参数表中使用的标识符名称，只要求类型。如果函数原型参数表中使用名称，则编译器忽略这些名称。

函数原型中使用的标识符可以在程序中的其他地方复用，不会产生歧义。

图 7.12 的程序演示了全局变量、自动局部变量和 `static` 局部变量的作用域问题。

```
1 // Fig. 7.12:fig03 12.cpp
2 // A scoping example
3 #include <iostream.h>
4
5 void a( void ); // function prototype
6 void b( void ); // function prototype
7 void c void ); // function prototype
8
9 int x = 1;      // global variable
10
11 int main( )
12 {
13     int x = 5; // local variable to main
14
15     cout << "local x in outer scope of main is " << x << endl;
16
17     {          // start new scope
18         int x = 7;
19
20         cout << "local x in inner scope of main is " << x << endl;
21     }          // end new scope
22
23     cout << "local x in outer scope of main is " << x << endl;
24
25     a();       // a has automatic local x
26     b();       // b has static local x
27     c();       // c uses global x
28     a();       // a reinitializes automatic local x
29     b();       // static local x retains its previous value
30     c();       // global x also retains its value
31
32     cout << "local x in main is " << x << endl;
33
34     return 0;
35 }
36
37 void a( void )
```

```

38 {
39     int x=25;          // initialized each time a is called
40
41     cout << endl << "local x in a is "<< x
42         << " after entering a" << endl;
43     ++x;
44     cout << "local x in a is "<< x
45         << "before exiting a" << endl;
46 }
47
48 void b( void )
49
50     static int x = 50; // Static initialization only
51                         // first time b is called.
52     cout << endl << "local static x is "<< x
53         << " - on entering b" << endl;
54     ++x;
55     cout << "local static x is" << x
56         << "on exiting b" << endl;
57 }
58
59 void c( void )
60
61     cout << endl << "global x is "<< x
62         << "on entering c" << endl;
63     x *= 10;
64     cout << "global x is "<< x << "on exiting c" << endl;
65 }

```

输出结果:

local x in outer scope of main is 5

local x in inner scope of main is 7

local x in outer scope of main is 5

local x in a is 25 after entering a

local x in a is 26 before exiting a

local static x is 50 on entering b

local static x is 51 on exiting b

global x is 1 on entering c

global x is 10 on exiting c

local x in a is 25 after entering a

```

local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5

```

图 7.12 说明变量作用域的例子

全局变量 `x` 声明并初始化为 1。这个全局变量在任何声明 `x` 变量的块和函数中隐藏。在 `main` 函数中，局部变量 `x` 声明并初始化为 5。打印这个变量，结果表示全局变量 `x` 在 `main` 函数中隐藏。然后在 `main` 函数中定义一个新块，将另一个局部变量 `x` 声明并初始化为 7，打印这个变量，结果表示其隐藏 `main` 函数外层块中的 `x`。数值为 7 的变量 `x` 在退出这个块时自动删除，并打印 `main` 函数外层块中的局部变量 `x`，表示其不再隐藏。程序定义三个函数，都设有参数和返回值。函数 `a` 定义自动变量 `x` 并将其初始化为 25。调用 `a` 时，打印该变量，递增其值，并在退出函数之前再次打印该值。每次调用该函数时，自动变量 `x` 重新初值化为 25。函数 `b` 声明 `static` 变量 `x` 并将其初始化为 10。声明为 `static` 的局部变量在离开作用域时仍然保持其数值。调用 `b` 时，打印 `x`，递增其值，并在退出函数之前再次打印该值。下次调用这个函数时，`static` 局部变量 `x` 包含数值 51。函数 `c` 不声明任何变量，因此，函数 `c` 引用变量 `x` 时，使用全局变量 `x`。调用函数 `c` 时，打印全局变量，将其乘以 10，并在退出函数之前再次打印该值。下次调用函数 `c` 时，全局变量已变为 10。最后，程序再次打印 `main` 函数中的局部变量 `x`，结果表示所有函数调用都没有修改 `x` 的值，因为函数引用的都是其他范围中的变量。

7.7 递归

前面介绍的程序通常由严格按层次方式调用的函数组成。对有些问题，可以用自己调用自己的函数。递归函数(`recursive function`)是直接调用自己或通过另一函数间接调用自己的函数。递归是个重要问题，在高级计算机科学教程中都会详细介绍。本节和下节介绍一些简单递归例子，本书则包含大量递归处理。图 7.17(7.14 节末尾)总结了本书的递归例子和练习。

我们先介绍递归概念，然后再介绍几个包含递归函数的程序。递归问题的解决方法有许多相同之处。调用递归函数解决问题时，函数实际上只知道如何解决最简单的情况(称为基本情况)。对基本情况的函数调用只是简单地返回一个结果。如果在更复杂的问题中调用函数，则函数将问题分成两个概念性部分：函数中能够处理的部分和函数中不能够处理的部分。为了进行递归，后者要模拟原问题，但稍作简化或缩小。由于这个新问题与原问题相似，因此函数启动(调用)自己的最新副本来处理这个较小的问题，称为递归调用(`recursive call`)或递归步骤(`recursive step`)。递归步骤还包括关键字 `return`，因为其结果与函数中需要处理的部分组合，形成的结果返回原调用者(可能是 `main`)。递归步骤在原函数调用仍然打开时执行，即原调用还没有完成。

递归步骤可能导致更多递归调用，因为函数-继续把函数调用的新的子问题分解为两个概念性部分。要让递归最终停止，每次函数调用时都使问题进一步简化，从而产生越来越小的问题，最终合并到基本情况。这时，函数能识别并处理这个基本情况，并向前一个函数副本返回结果，

并回溯一系列结果，直到原函数调用最终把最后结果返回给 `main`。这一切比起前面介绍的其他问题似乎相当复杂。下面通过一个例子来说明。我们用递归程序进行一个著名的数学计算。

非负整数 n 的阶乘写成 $n!$ ，为下列数的积：

$$n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$$

其中 1，等于 1, 0 定义为 1。例如，5! 为 $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ ，即 120。

整数 `number` 大于或等于 0 时的阶乘可以用下列 `for` 循环迭代(非递归)计算：

```
factorial = 1;
```

```
for { int counter = number; counter >= 1; counter-- }
```

```
factorial *= counter;
```

通过下列关系可以得到阶乘函数的递归定义：

$$n! = n \cdot (n-1)!$$

例如，5! 等于 $5 \cdot 4!$ ，如下所示：

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5 \cdot (4!)$$

求值 5! 的过程如图 7.13。图 7.13 a) 显示如何递归调用，直到 1! 求值为 1，递归终止。图 7.13 b) 显示每次递归调用向调用者返回的值，直到计算和返回最后值。

图 7.14 的程序用递归法计算并打印。到 10 的整数阶乘(稍后将介绍数据类型 `unsigned long` 的选择)。递归函数 `factorial` 首先测试终止条件是否为 `true`，即 `number` 是否小于或等于 1。如果 `number` 小于或等于 1，则 `factorial` 返回 1，不再继续递归，程序终止。如果 `number` 大于 1，则下列语句。

```
return number * factorial( number - 1 );
```

将问题表示为 `number` 乘以递归调用 `dactorial` 求值的 `number-1` 的阶乘。注意 `factorial(number-1)` 比原先 `factorial(number)` 的计算稍微简单一些。

```
1 // Fig. 7.14:fig03 14.cpp
2 // Recursive factorial function
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 unsigned long factorial( unsigned long );
7
8 int main( )
9 {
10     for ( int i = 0; i <= 10; i++ )
11         cout << setw( 2) << i << "! = " << factorial( i ) << endl;
12
13     return 0;
14 }
15
16 // Recursive definition of function factorial
17 unsigned long factorial( unsigned long number )
18 {
19     if ( number <= 1 ) // base case
```

```

20     return 1;
21     else                // recursive case
22         return number * factorial( number - 1 );
23 }

```

输出结果:

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

```

图 7.4 用递归法计算阶乘

函数 `factorial` 声明为接收 `unsigned long` 类型的参数和返回 `unsigned long` 类型的值。`unsigned long` 是 `unsigned long int` 的缩写，C++语言的规则要求存放 `unsigned long int` 类型变量至少占用 4 个字节(32 位)，因此可以取 0 到 4294967291 之间的值(数据类型 `long int` 至少占内存中的 4 个字节，可以取 ± 2147483647 之间的值)。如图 7.14，阶乘值很快就变得很大。我们选择 `unsigned long` 数据类型，使程序可以在字长为 16 位的计算机上计算大于 7! 的阶乘。但是，`factorial` 函数很快产生很大的值，即使 `unsigned long` 也只能打印少量阶乘值，然后就会超过 `unsigned long` 变量的长度。

练习中将会介绍，用户最终可能要用 `float` 和 `double` 类型来计算大数的阶乘，这就指出了大多数编程语言的弱点，即不能方便地扩展成处理不同应用程序的特殊要求。从本书面向对象编程部分可以看到，C++是个可扩展语言，可以在需要时生成任意大的数。

7.8 使用递归举例，Fibonacci 数列

Fibonacci 数列(斐波纳契数列):

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

以 0 和 1 开头，后续每个 Fibonacci 数是前面两个 Fibonacci 数的和。

自然界中就有这种数列,描述一种螺线形状。相邻 Fibonacci 数的比是一个常量 1.618..., 这个数在自然界中经常出现，称为黄金分割(golden ratio 或 golden mean)。人们发现，黄金分割会产生最佳的欣赏效果。因此建筑师通常把窗户、房子和大楼的长和宽的比例设置为黄金分割数，明信片的长宽比也采用黄金分割数。

Fibonacci 数列可以递归定义如下:

```
fibonacci(0)=0
```

```
fibonacci(1)=1
```

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

图 7.15 的程序用函数 fibonacci 递归计算第 i 个 Fibonacci 数。注意 Fibonacci 数很快也会变得很大，因此把 Fibonacci 函数的参数和返回值类型设为 unsigned long 数据类型。图 7.11 中每对输出行显示运行一次程序的结果。

```
1 // Fig. 7.15: fig03_1S.cpp
2 // Recursive fibonacci function
3 #include <iostream.h>
4
5 long fibonacci( long );
6
7 int main( )
8 {
9     long result, number;
10
11     cout << "Enter an integer: ";
12     cin >> number;
13     result = fibonacci( number );
14     tout << "Fibonacci{" << number << "} = " << result << endl;
15     return 0;
16 }
17
18 // Recursive definition of function fibonacci
19 long fibonacci( long n )
20 {
21     if ( n == 0 || n == 1 ) // base case
22         return n;
23     else // recursive case
24         return fibonacci( n - 1 ) + fibonacci( n - 2 );
25 }
```

输出结果如下：

```
Enter an integer: 0
```

```
Fibonacci(0) = 0
```

```
Enter an integer: 1
```

```
Fibonacci(1) = 1
```

```
Enter an integer: 2
```

```
Fibonacci(2) = 1
```

```
Enter an integer: 3
```

```
Fibonacci(3) = 2
```

```
Enter an integer: 4
```

```
Fibonacci(4) = 3
```

```
Enter an integer: 5
```

```
Fibonacci(5) = 5
```

```
Enter an integer: 6
Fibonacci(6) = 8
Enter an integer: 10
Fibonacci(10) = 55
Enter an integer: 20
Fibonacci(20) = 6765
Enter an integer: 30
Fibonacci(30) = 832040
Enter an integer: 35
Fibonacci(35) = 9227465
```

图 7.15 递归计算 Fibonacci 数列

main 中调用 fibonacci 函数不是递归调用，但后续所有调用 fibonacci 函数都是递归调用。每次调用 fibonacci 时，它立即测试基本情况(n 等于 0 或 1)。如果是基本情况，则返回 n。有趣的是，如果 n 大于 1，则递归步骤产生两个递归调用，各解决原先调用 fibonacci 问题的一个简化问题。图 9. 16 显示了 fibonacci 函数如何求值 fibonacci(3)，我们将 fibonacci 缩写成 f。

图中提出了 C++编译器对运算符操作数求值时的一些有趣的顺序问题。这个问题与运算符作用于操作数的顺序不同，后者的顺序是由运算符优先级规则确定的。图 7.16 中显示，求值 f(3) 时，进行两个递归调用，即 f(2)和 f(1)。但这些调用的顺序如何呢？

大多数程序员认为操作数从左向右求值。奇怪的是，C++语言没有指定大多数运算符的操作数求值顺序(包括+)，因此，程序员不能假设这些调用的顺序。调用可能先执行 f(2)再执行 f(1)，也可能先执行 f(1)再执行 f(2)在该程序和大多数其他程序中，最终结果都是相同的。但在有些程序中，操作数求值顺序的副作用可能影响表达式的最终结果。

C++语言只指定四种运算符的操作数求值顺序，即"&&"、"||"、逗号运算符(,)和"?:"。前三种是二元运算符，操作数从左向右求值。第四种是 C++惟一的三元运算符，先求值最左边的操作数，如果最左边的操作数为非 0，则求值中间的操作数，忽略最后的操作数；如果最左边的操作数为 0，则求值最后的操作数，忽略中间的操作数。

要注意这类程序中产生 Fibonacci 数的顺序。Fibonacci 函数中的每一层递归对调用数有加倍的效果，即第 n 个 Fibonacci 数在第 2n 次递归调用中计算。计算第 20 个 Fibonacci 数就要 220 次，即上百万次调用，而计算第 30 个 Fibonacci 数就要 230 次，即几十亿次调用。计算机科学家称这种现象为指数复杂性(exponential complexity)，这种问题能让最强大的计算机望而生畏。一般复杂性问题 and 指数复杂性将在高级计算机科学课程“算法”中详细介绍。

7.9 递归与迭代

前面几节介绍了两个可以方便地用递归与迭代实现的函数。本节要比较递归与迭代方法，介绍为什么程序员在不同情况下选择不同方法。

递归与迭代都是基于控制结构：迭代用重复结构，而递归用选择结构。递归与迭代都涉及重复：迭代显式使用重复结构，而递归通过重复函数调用实现重复。递归与迭代都涉及终止测试：

迭代在循环条件失败时终止，递归在遇到基本情况时终止。使用计数器控制重复的迭代和递归都逐渐到达终止点：迭代一直修改计数器，直到计数器值使循环条件失败；递归不断产生最初问题的简化副本，直到达到基本情况。迭代和递归过程都可以无限进行：如果循环条件测试永远不变成 `false`，则迭代发生无限循环；如果递归永远无法回推到基本情况，则发生无穷递归。

递归有许多缺点，它重复调用机制，因此重复函数调用的开销很大，将占用很长的处理器时间和大量的内存空间。每次递归调用都要生成函数的另一个副本(实际上只是函数变量的另一个副本)，从而消耗大量内存空间。迭代通常发生在函数内，因此没有重复调用函数和多余内存赋值的开销。那么，为什么选择递归呢？

大多数有关编程的教材都把递归放在后面再讲。我们认为递归问题比较复杂而且内容丰富，应放在前面介绍，本书余下部分会通过更多例子加以说明。图 7.17 总结了本书使用递归算法的例子和练习。

下面重新考虑书中重复强调的一些观点。良好的软件工程很重要，高性能也很重要，但是，这些目标常常是互相矛盾的。良好的软件工程是使开发的大型复杂的软件系统更容易管理的关键，而高性能是今后在硬件上增加计算需求时实现系统的关键，这两个方面如何取得折衷？

7.10 带空参数表的函数

在 C++ 中，空参数表可以用 `void` 指定或括号中不放任何东西。下列声明：

```
void print();
```

指定函数 `print` 不取任何参数，也不返回任何值。图 7.18 演示了 C++ 声明和使用带空参数表的函数的方法。

```
// Fig. 7.18: fig03_1$.cpp
2 // Functions that take no arguments
3 #include <iostream.h>
4
5 void function1( );
6 void function2( void );
7
8 int main( )
9 {
10     function1( );
11     function2( );
12
13     return 0;
14 }
15
16 void function1( )
17 {
18     cout << "function1 takes no arguments" << endl;
19 }
20
```

```

21 void function2( void )
22 {
23     cout << "function2 also takes no arguments" << endl;
24 }

```

输出结果:

```

function1 takes no arguments
function2 also takes no arguments

```

图 7.18 两种声明和使用带空参数表函数的方法

7.11 内联函数

从软件工程角度看，将程序实现为一组函数很有好处，但函数调用却会增加执行时的开销。

C++提供了内联函数(**inline function**)可以减少函数调用的开销，特别是对于小函数。函数定义中函数返回类型前面的限定符 **inline** 指示编译器将函数代码复制到程序中以避免函数调用。其代价是会产生函数代码的多个副本并分别插入到程序中每一个调用该函数的位置上(从而使程序更大)，而不是只有一个函数副本(每次调用函数时将控制传入函数中)。典型情况下，除了最小的函数以外编译器可以忽略用于其他函数的 **inline** 限定符。

图 7.19 的程序用内联函数 **cube** 计算边长为 **s** 的立方体体积。函数 **cube** 参数表中的关键字 **const** 表示函数不修改变量的值 **s**。

```

1 // Fig. 7.19:fig03 19.cpp
2 // Using an inline function to calculate
3 // the volume of a cube.
4 #include <iostream.h>
5
6 inline float cube( const float s ) { return s * s * s; }
7
8 int main( )
9 {
10     cout << "Enter the side length of your cube:  ";
11
12     float side;
13
14     cin >> side;
15     cout << "Volume of cube with side ,,
16         << side << " is "<< cube( side ) << endl;
17
18     return 0;
19 }

```

输出结果:

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

图 7.19 用内联函数计算立方体的体积

7.12 函数重载

C++允许定义多个同名函数，只要这些函数有不同参数集(至少有不同类型的参数)。这个功能称为函数重载(function overloading)。调用重载函数时,C++编译器通过检查调用中的参数个数、类型和顺序来选择相应的函数。函数重载常用于生成几个进行类似任务而处理不同数据类型的同名函数。

图 7.25 用重载函数 square 计算 int 类型值的平方以及 double 类型值的平方。

```
1 // Fig. 7.25: fig03_25.cpp
2 // Using overloaded functions
3 #include <iostream.h>
4
5 iht square( iht x ) { return x * x; }
6
7 double square( double y ) { return y * y; }
8
9 int main( )
10 {
11     cout << "The square of integer 7 is" << square( 7 )
12         << "\nThe square of double 7.5 is" << square( 7.5 )
13         << endl;
14
15     return 0;
16 }
```

输出结果:

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

图 7.25 使用重载函数

重载函数通过签名(signature)进行区别，签名是函数名和参数类型的组合。编译器用参数个数和类型编码每个函数标识符(有时称为名字改编或名字修饰)，以保证类型安全连接(type-safe linkage)。类型安全连接保证调用合适的重载函数并保证形参与实参相符。编译器能探测和报告连接错误。图 7.26 的程序在 Borland C++编译器上编译，我们不显示程序执行的输出，图中用汇编语言输出了由 Borland C++编译器产生的改编函数名。每个改编名用@加上函数名，

改编参数表以\$q开头。在函数 nothing2 的参数表中, zc 表示 char、i 表示 int、pf 表示 float*、pd 表示 double*。在函数 nothing1 的参数表中, i 表示 int、f 表示 float、zc 表示 char、pi 表示 int*。两个 square 函数用参数表区分, 一个指定 d 表示 double, 一个指定 i 表示 int。函数的返回类型不在改编名称中指定。函数名改编是编译器特定的。重载函数可以有不同返回类型, 但必须有不同参数表。

```

1 // Name mangling
2 int square( int x ) { return x * x; }
3
4 double square( double y ) { return y * y; }
5
6 void nothing1( int a, float b, char c, int *d )
7 {     } // empty function body
8
9 char *nothing2( char a, int b, float *c, double *d )
10 { return 0; }
11
12 int main( )
13 {
14     return 0;
15 }

```

输出结果:

```

public _main
public @nothing2$qzqipfpd
public @nothing1$qifzcp
public @square$qd
public @square$qi

```

图 7.26 名字改编以保证类型安全连接

编译器只用参数表区别同名函数。重载函数不一定要有相同个数的参数。程序员使用带默认参数的重载函数时要小心, 以免出现歧义。

第 8 章 数组

8.1 简介

数组(array)数据结构由相同类型的相关数据项组成。数组和结构是静态项目，在整个程序执行期间保持相同长度(当然，也可以用自动存储类，在每次进入和离开定义的块时生成和删除)。

8.2 数组

数组是具有相同名称和相同类型的一组连续内存地址。要引用数组中的特定位置或元素，就要指定数组中的特定位置或元素的位置号(position number)。

图 8.1 显示了整型数组 `c`。这个数组包含 12 个元素。可以用数组名加上方括号(1)中该元素的位置号引用该元素。数组中的第一个元素称为第 0 个元素(zeroth elemem)。这样，`c` 数组中的第一个元素为 `c[0]`，`c` 数组中的第二个元素为 `c[1]`，`c` 数组中的第七个元素为 `c[6]`，一般来说，`c` 数组中的第 `i` 个元素为 `c[i-1]`。数组名的规则与其他变量名相同。

方括号中的位置号通常称为下标(subscript)，下标应为整数或整型表达式。如果程序用整型表达式下标，则要求值这个整型表达式以确定下标，例如，假设 `a` 等于 5，`b` 等于 6，则下列语句：

```
c[a + b] += 2
```

将数组元素 `c[11]` 加 2。注意带下标的数组名是个左值，可用于赋值语句的左边。

图 8.1 中整个数组的名称为 `c`，该数组的 12 个元素为 `c[0]`、`c[1]`、`c[2]`...`c[11]`。的值为 -45、`c[1]` 的值为 6、`c[2]` 的值为 0，`c[7]` 的值为 62、`c[11]` 的值为 78。要打印数组 `c` 中前三个元素的和，用下列语句：

```
cout<< c[0]+c[1]+c[2] <<endl;
```

要将数组 `c` 的第 7 个元素的值除以 2，并将结果赋给变量 `x`，用下列语句：

```
x = c[6] / 2;
```

包括数组下标的方括号实际上是个 C++ 运算符。方括号的优先级与括号相同。图 8.2 显示了本书前面介绍的 C++ 运算符优先级和结合律。运算符优先级从上到下逐渐减少。

运算符	结合律	类型
() []	从左向右	括号
++ -- + - ! static_cast<type>()	从右向左	一元
* / %	从左向右	乘
+ -	从左向右	加
<< >>	从左向右	插入/读取
< <= > >=	从左向右	关系

==	!=	从左向右	相等
&&		从左向右	逻辑与
		从左向右	逻辑或
?:		从右向左	条件
=	+=	-=	*=
/=	%=	从左向右	赋值
,		从左向右	逗号

图 8.2 运算符的优先级和结合律

8.3 声明数组

数组要占用内存空间。程序员指定每个元素的类型和每个数组所要的元素，使编译器可以保留相应的内存空间。要告诉编译器对整型数组 `c` 保留 12 个元素，可以声明如下：

```
int c[12];
```

可以在一个声明中为几个数组保留内存。下列声明对整型数组 `b` 保留 100 个元素，对整型数组 `x` 保留 27 个元素：

```
int b[100],x[27];
```

数组可以声明包含其他数据类型。例如，`char` 类型的数组可以存放字符串。字符串及其与字符数组的相似性(C++从 `c` 语言继承的关系)和指针与数组的关系将在第 5 章介绍。在介绍面向对象编程后，我们将讨论成熟的字符串对象。

8.4 使用数组的举例

图 8.3 的程序用 `for` 重复结构将 10 个元素的整型数组 `n` 的元素初始化为 0，并用表格形式打印数组。第一个输出语句显示 `for` 结构中所打印列的列标题。记住，`setw` 指定下一个值的输出域宽。

可以在数组声明中用等号和逗号分隔的列表(放在花括号中)将数组中的元素初始化。程序 8.4 将七个元素的整型数组初始化并用表格形式打印数组。

```
1 // Fig. 8.3: fig04_03.cpp
2 // initializing an array
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int i, n[10];
9
10    for ( i = 0; i < 10; i++ )        // initialize array
11        n[ i ] = 0;
```

```

12
13 cout << "Element" << setw( 13 ) << "Value" << endl;
14
15 for(i=0;i<10;i++)          // print array
16     cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
17
18     return 0;
19 }

```

输出结果:

Element	value
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

图 8.3 将 10 个元素的整型数组 n 的元素初始化为 0

```

1 // Fig. 8.4:fig04 04.cpp
2 // Initializing an array with a declaration
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
9
10     cout << "Element" << setw( 13 ) << "Value" << endl;
11
12     for ( int i = 0; i < 10; i++ )
13         cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
14
15     return 0;
16 }

```

输出结果:

Element	Value
0	32

1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

图 8.4 用声明将数组中的元素初始化

如果初始化的元素比数组中的元素少，则其余元素自动初始化为 0。例如，可以用下列声明将图 8.3 中数组 n 的元素初始化为 0：

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

其显式地将第一个元素初始化为 0，隐式地将其余元素自动初始化为 0，因为初始化值比数组中的元素少。程序员至少要显式地将第一个元素初始化为 0，才能将其余元素自动初始化为 0。图 8.3 的方法可以在程序执行时重复进行。

需要初始化数组元素而没有初始化数组元素是个逻辑错误。

下列数组声明是个逻辑错误：

```
int n[5] = { 32, 27, 64, 18, 95, 14};
```

因为有 6 个初始化值，而数组只有 5 个元素。

初始化值超过数组元素个数是个逻辑错误。

如果带初始化值列表的声明中省略数组长度，则数组中的元素个数就是初始化值列表中的元素个数。例如：

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

生成五个元素的数组。

图 8.5 的程序将 10 个元素的数组 s 初始化为整数 2、4、6、…20，并以表格形式打印数组。这些数值是将循环计数器的值乘以 2 再加上 2 产生的。

```
1 // Fig. 8.5: fig0405.cpp
2 // Initialize array s to the even integers from 2 to 20.
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7
8     const int arraySize = 10;
9     int j, s[arraySize];
10
11     for ( j = 0; j < arraySize; j++ ) // set the values
12         s[j] = 2 + 2* j;
13
14     cout << "Element" << setw( 13 ) << "Value" << endl;
15
```



```

16     for ( j = 0; j < arraysize; j++ ) // print the values
17         cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
18
19     return 0;
20 }

```

输出结果:

Element	Value
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

图 8.5 将产生的值赋给数组元素

下列语句:

```
const int arraySlze: 10
```

用 `const` 限定符声明常量变量 `arraySize` 的值为 10。常量变量应在声明时初始化为常量表达式,此后不能改变(图 8.6 和图 8.7)。常量变量也称为命名常量(named constant)或只读变量(read-only variable)。

```

1 // Fig. 8.6: fig04_06.cpp
2 // Using a properly initialized constant variable
3 #include <iostream.h>
4
5 int main()
6 {
7     const int x = 7; // initialized constant variable
8
9     cout << "The value of constant variable x is:"
10         << x << endl;
11
12     return 0;
13 }

```

输出结果:

The value of constant variable x is: 7

图 8.6 正确地初始化和使用常量变量

```

1 // Fig. 8.7:fig04 07.cpp
2 // A const object must be initialized
3
4 int main()
5 {
6     const int x; // Error: x must be initialized
7
8     x = 7;       // Error: cannot modify a const variable
9
10    return 0;
11}

```

输出结果:

Compiling FIG04-7.CPP:

Error FIG04_7. CPP 6: Constant variable, x' must be initialized

Error FIG04_7. CPP 8: Cannot modify a const object

图 8.7const 对象应初始化

常量变量可以放在任何出现常量表达式的地方。图 8.5 中，用常量变量 `arraySize` 指定数组 `s` 的长度：

```
int j, s[arraySize];
```

用常量变量声明数组长度使程序的伸缩性更强。图 8.5 中，要让第一个 `for` 循环填上 1000 个数组元素，只要将 `arraySize` 的值从 10 变为 1000 即可。如果不用常量变量 `arraySize`，则要在程序中进行三处改变才能处理 1000 个数组元素。随着程序加大，这个方法在编写清晰的程序中越来越有用。

将每个数组的长度定义为常量变量而不是常量，能使程序更清晰。这个方法可以取消“魔数”，例如，在处理 10 元素数组的程序中重复出现长度 10 使数字 10 人为地变得重要，程序中存在的与数组长度无关的其他数字 10 时可能使读者搞乱。

图 5.8 中的程序求 12 个元素的整型数组 `a` 中的元素和，`for` 循环体中的语句进行求和。请注意，数组 `a` 的初始化值通常是用户从键盘输入的。例如，下列 `for` 结构：

```
for ( int j=0; j < arraySize; j++ )
    cin>>a[j];
```

一次一个地从键盘读取数值,并将数值存放在元素 `a[j]` 中。

下一个例子用数组汇总调查中收集的数据。考虑下列问题：

40 个学生用 1 到 10 的分数评价学生咖啡屋中的食品质量(1 表示很差，10 表示很好)。将 40 个值放在整型数组中，并汇总调查结果。

这是典型的数组应用(如图 8.9)。我们要汇总每种回答(1 到 10)的个数。数组 `responses` 是 40 个元素的评分数组。我们用 11 个元素的数组 `frequency` 计算每个答案的个数，忽略第一个元素 `frequency[0]`，因为用 1 分对应 `frequency[1]` 而不是对应 `frequency[0]` 更好理解。这样可以直接用回答的分数作为 `frequency` 数组的下标。

```

1 // Fig. 8.8:fig04 OS.cppf
2 // Compute the sum of the elements of the array
3 #include <iostream.h>
4
5 int main()
6 {
7     const iht arraySize = 12;
8     int a[ arraySize ] = { 1, 3, 5, 4, 7, 2, 99,
9                           16, 45, 67, 89, 45 };
10    int total = 0;
11
12    for (int i = 0; i < arraySize ; i++ )
13        total += a[ i ];
14
15    cout << "Total of array element values is "<< total << endl;
16    return 0;
17 }

```

输出结果:

Total of array element values is 383

图 8.8 计算数组元素和

```

1 // Fig. 8.9: fig04_09.cpp
2 // Student poll program
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7
8     const int responseSize = 40, frequencySize = 11;
9     int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
10    10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
11    5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
12    int frequency [ frequencySize ] = { 0 };
13
14    for (int answer = 0; answer < responseSize; answer++ )
15        ++frequency [ responses[ answer ] ];
16
17    cout << "Rating" << setw( 17 ) << "Frequency" << endl;
18
19    for ( int rating = 1; rating < frequencySize; rating++ )
20        cout << setw( 6 ) << rating

```

```

21         << setw( 17 ) << frequency [ rating ] << endl;
22
23     return 0;
24 }

```

输出结果:

Rating	Frequency
1	2
2	2
3	2
4	2
5	5
6	11
7	5
8	7
9	1
10	3

图 8.9 学生调查分析程序

第一个 for 循环一次一个地从 responses 数组取得回答, 并将 frequency 数组中的 10 个计数器 (frequency[1]到 frequency[10])之一加 1。这个循环中的关键语句如下:

```
++frequency[ responses[answer]>];
```

这个语句根据 responses[answer]的值相应递增 frequency 计数器。例如, 计数器 answer 为 0 时, responses[answer]为 1, 因此 “++frequency[responses[answer]>;” 实际解释如下:

```
++frequency[1];
```

将数组下标为 1 的元素加 1。计数器 answer 为 1 时, responses[answer]为 2, 因此 “++frequency[responses[answer]>;” 实际解释如下:

```
++frequency[2];
```

将数组下标为 2 的元素加 1。计数器 answer 为 2 时, response[answer]为 6, 因此 “++frequency[responses[answer]>;” 实际解释如下:

```
++frequency[6];
```

将数组下标为 6 的元素加 1 等等。注意, 不管调查中处理多少个回答, 都只需要 11 个元素的数组(忽略元素 0)即可汇总结果。如果数据中包含 13 之类的无效值, 则程序对 frequency[13]加 1, 在数组边界之外。C++没有数组边界检查, 无法阻止计算机到引用不存在的元素。这样执行程序可能超出数组边界, 而不产生任何警告。程序员应保证所有数组引用都在数组边界之内。C++是个可扩展语言, 第 8 章介绍扩展 C++, 用类将数组实现为用户自定义类型。新的数组定义能进行许多 C++内建数组中没有的操作, 例如, 可以直接比较数组, 将一个数组赋给另一数组, 用 cin 和 cout 输入和输出整个数组, 自动初始化数组, 防止访问超界数组元素和改变下标范围(甚至改变下标类型), 使数组第一个元素下标不一定为 0。

下一个例子(图 8.10)从数组读取数值, 并用条形图或直方图进行描述。打印每个数值, 然后

在数字旁边打印该数字所指定星号个数的条形图。嵌套 for 循环实现绘制条形图。注意用 endl 结束直方图。

```
1 // Fig. 8.10: fig04_10.cpp
2 // Histogram printing program
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     const int arraySize = 10;
9     int n[ arraySize ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
10
11     cout << "Element" << setw( 13 ) << "Value"
12         << setw( 17 ) << "Histogram" << endl;
13
14     for (int i = 0; i < arraySize ; i++) {
15         cout << setw( 7 ) << i << setw( 13 )
16             << n[ i ] << setw( 9 );
17
18         for ( int j = 0; j < n[ i ] ; j++)    // print one bar
19             cout << '*';
20
21         cout << endl;
22     }
23
24     return 0;
25 }
```

输出结果:

Element	Value	Histogram
0	19	*****
1	3	***
2	15	*****
3	7	*****
4	11	*****
5	9	*****
6	13	*****
7	5	*****
8	17	*****
9	1	

前面只介绍了整型数组，但数组可以是任何类型，下面要介绍如何用字符数组存放字符串。前面介绍的字符串处理功能只有用 `cout` 和 `<<` 输入字符串，“hello”之类的字符串其实就是一个字符数组。字符数组有几个特性。

字符数组可以用字符串直接量初始化。例如，下列声明：

```
char string1[] = "first";
```

将数组 `string1` 的元素初始化为字符串“first”中的各个元素。上述声明中 `string1` 的长度是编译器根据字符串长度确定的。注意，字符串“first”包括五个字符加一个特殊字符串终止符，称为空字符(`null character`)，这样，字符串 `string1` 实际上包含 6 个元素。空字符对应的字符常量为 `'\0'`(反斜杠加 0)，所有字符串均用这个空字符结尾。表示字符串的字符数组应该足以放置字符串中的所有字符和空字符。

字符数组还可以用初始化值列表中的各个字符常量初始化。上述语句也可以写成：

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' }
```

由于字符串其实是字符数组，因此可以用数组下标符号直接访问字符串中的各个字符。例如，`string1[0]`是字符‘f’，`string1[3]`是字符‘s’。

我们还可以用 `cin` 和 `>>` 直接从键盘输入字符数组。例如，下列声明：

```
char string2[20];
```

生成的字符数组能存放 19 个字符和一个 `null` 终止符的字符串。

下列语句：

```
cin >> string2;
```

从键盘中将字符串读取到 `string2` 中。注意，上述语句中只提供了数组名，没有提供数组长度的信息。程序员要负责保证接收字符串的数组能够放置用户从键盘输入的任何字符串。`cin` 从键盘读取字符，直到遇到第一个空白字符，它不管数组长度如何。这样，用 `cin` 和 `>>` 输入数据可能插入到数组边界之。

如果 `cin >>` 提供足够大的数组，则键盘输入时可能造成数据丢失和其他严重的运行时错误。

可以用 `cout` 和 `<<` 输出表示空字符终止字符串的字符数组。下列语句打印数组 `string2`：

```
cout << string2 << endl;
```

注意 `cout <<` 和 `cin >>` 一样不在乎字符数组的长度。一直打印字符串的字符，直到遇到 `null` 终止符为止。

图 8.12 演示直接用字符串初始化字符数组、将字符串读取到字符数组中、将字符数组作为字符串打印以及访问字符串的各个字符。

```
1 // Fig. 4_12: fig04_12.cpp
2 // Treating character arrays as strings
3 #include <iostream.h>
4
5 int main()
6 {
7     char string1[ 20 ], string2[] = "string literal";
8
9     cout << "Enter a string: ";
10    cin >> string1;
11    cout << "string1 is: " << string1
12         << "\nstring2 is: " << string2
```

```

13     << "string1 with spaces between characters is:\n";
14
15     for ( int i = 0; string1[ i ] != '\0'; i++ )
16         cout << string1[ i ] << ' ';
17
18     cin >> string1;    // reads "there"
19     cout << "\nstring1 is: " << string1 << endl;
20
21     cout << endl;
22     return 0;
23 }

```

输出结果:

```

Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
Hello
string1 is: there

```

图 8.12 将字符数组当作字符串

图 8.12 用 for 结构在 `string1` 数组中循环，并打印各个字符，用空格分开。for 结构的条件 `string1[i]!='\0'` 在遇到字符串的 null 终止符之前一直为真。

第 3 章介绍了存储类说明符 `static`。函数体中的 `static` 局部变量在程序执行期间存在，但只能在函数体中访问该变量。

图 8.13 显示了带有声明为 `static` 的局部数组的 `staticArrayInit` 函数和带自动局部数组的 `automaticArrayInit` 函数。`staticArrayInit` 调用两次，编译器将 `static` 局部数组初始化为 0。该函数打印数组，将每个元素加 5，然后再次打印该数组；函数第二次调用时，`static` 数组包含第一次调用时所存放的值。函数 `automaticArrayInit` 也调用两次，但自动局部数组的元素用数值 1、2、3 初始化。该函数打印数组，将每个元素加 5，然后再次打印该数组；函数第二次调用时，数组重新初始化为 1、2、3，因为这个数组是自动存储类。

```

1 // Fig. 8.13: fig0413.cpp
2 // Static arrays are initialized to zero
3 #include <iostream.h>
4
5 void staticArrayInit( void );
6 void automaticArrayInit( void );
7
8 int main()
9 {
10     cout << "First call to each function:\n";
11     staticArrayInit();

```

```

12     automaticArrayInit();
13
14     cout << "\n\nSecond call to each function:\n";
15     staticArrayInit();
16     automaticArrayInit();
17     cout << endl;
18
19     return 0;
20
21
22 // function to demonstrate a static local array
23 void staticArrayInit( void )
24 {
25     static int array1[ 3 ] ;
26     int i;
27
28     cout << "\nValues on entering staticArrayInit:\n";
29
30     for ( i = 0; i < 3; i++ )
31         cout << "array1[ " << i << " ] =" << array1[ i ] << " ";
32
33     cout << "\nValues on exiting staticArrayInit:\n";
34
35     for ( i = 0; i < 3; i++ )
36         cout << "array1[ " << i << " ] ="
37             << ( array1[ i ] += 5 ) << " ";
38 }
39
40 // function to demonstrate an automatic local array
41 void automaticArrayInit( void )
42 {
43     int i, array2[ 3 ] = { 1, 2, 3 };
44
45     cout << "\n\nValues on entering automaticArrayInit:\n";
46
47     for ( i = 0; i < 3; i++ )
48         cout << "array2[ " << i << " ] =" << array2[ i ] << " ";
49
50     cout << "\nValues on exiting automaticArrayInit:\n";
51
52     for ( i = 0; i < 3; i++ )
53         cout << "array2[ " << i << " ] ="
54             << ( array2[ i ] += 5 ) << " ";
55 }

```


输出结果:

First call to each function:

Values on entering staticArrayInit:

array1[0] = 0 array1[1] = 0 array1[2] = 0

Values on exiting staticArrayInit:

array1[0] = 5 array1[1] = 5 array1[2] = 5

Values on entering automaticArrayInit:

array2[0] = 1 array2[1] = 2 array2[2] = 3

Values on exiting automaticArrayInit:

array2[0] = 6 array2[1] = 7 array2[2] = 8

second call to each function:

Values on entering staticArrayInit:

array1[0] = 5 array1[1] = 5 array1[2] = 5

Values on exiting staticArrayInit:

array1[0] = 10 array1[1] = 10 array1[2] = 10

Values on entering automaticArrayInit:

array2[0] = 1 array2[1] = 2 array2[2] = 3

Values on exiting automaticArrayInit:

array2[0] = 6 array2[1] = 7 array2[2] = 8

图 8.13 比较 static 数组初始化和自动数组初始化

8.5 将数组传递给函数

要将数组参数传递给函数，需指定不带方括号的数组名。例如，如果数组 `hourlyTemperatures` 声明如下：

```
int hourlyTemperatures[24];
```

则下列函数调用语句：

```
modifyArray(hourlyTemperatures, 24);
```

将数组 `hourlyTemperatures` 及其长度传递给函数 `modifyArray`。将数组传递给函数时，通常也将其长度传递给函数，使函数能处理数组中特定的元素个数(否则要在被调用函数中建立这些信息，甚至要把数组长度放在全局变量中)。第 8 章介绍 `Array` 类时，将把数组长度设计在用户自定义类型中，每个 `Array` 对象生成时都“知道”自己的长度。这样，将 `Array` 对象传递给函数时，就不用把数组长度作为参数一起传递。

C++使用模拟的按引用调用，自动将数组传递给函数，被调用函数可以修改调用者原数组中

的元素值。数组名的值为数组中第一个元素的地址。由于传递数组的开始地址，因此被调用函数知道数组的准确存放位置。因此，被调用函数在函数体中修改数组元素时，实际上是修改原内存地址中的数组元素。

尽管模拟按引用调用传递整个数组，但各个数组元素和简单变量一样是按值传递。这种简单的单个数据称为标量(scalar 或 scalar quantity)。要将数组元素传递给函数，用数组元素的下标名作为函数调用中的参数。第 5 章将介绍标量(即各个变量和数组元素)的模拟按引用调用。

要让函数通过函数调用接收数组，函数的参数表应指定接收数组。例如，函数 `modifyArray` 的函数首部可能如下所示：

```
void modifyArray(int b[], int arraySize)
```

表示 `modifyArray` 要在参数 `b` 中接收整型数组并在参数 `arraySize` 中接收数组元素个数。数组方括号中的数组长度不是必需的，如果包括，则编译器将其忽略。由于模拟按引用调用传递数组，因此被调用函数使用数组名 `b` 时，实际上引用调用者的实际数组(上例中为数组 `hourlyTemperatures`)。第 5 章介绍表示函数接收数组的其他符号，这些符号基于数组与指针之间的密切关系。

注意 `modifyArray` 函数原型的表示方法：

```
void modifyArray(int[], int);
```

这个原型也可以改写成：

```
void modifyArray( int anyArrayName[], int( anyVariableName )
```

但第 3 章曾介绍过，C++编译器忽略函数原型中的变量名。

图 8.14 的程序演示了传递整个数组与传递数组元素之间的差别。程序首先打印整型数组 `a` 的五个元素,然后将 `a` 及其长度传递给函数 `modifyArray`，其中将 `a` 数组中的元素乘以 2，然后在 `main` 中重新打印 `a`。从输出可以看出，实际由 `modifyArray` 修改 `a` 的元素。现在程序打印 `a[3]` 的值并将其传递给函数 `modifyElement`。函数 `modifyElement` 将参数乘以 2。然后打印新值。注意在 `main` 中重新打印 `a[3]` 时，它没有修改，因为各个数组元素是按值调用传递。

```
1 // Fig. 8.14: fig0414.cpp
2 // Passing arrays and individual array elements to functions
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 void modifyArray( int [], int ); // appears strange
7 void modifyElement( int );
8
9 int main()
10 {
11     const int arraySize = 5;
12     int i, a[ arraySize ] = { 0, 1, 2, 3, 4 };
13
14     cout << "Effects of passing entire array call-by-reference:"
15         << "\n\nThe values of the original array are:\n";
16
17     for( i=0; i< arraySize; i++ )
```

```

18     cout<< setw( 3 ) << a[ i ];
19
20     cout << endl;
21
22     // array a passed call-by-reference
23     modifyArray( a, arraySize );
24
25     cout << "The values of the modified array are:\n";
26
27     for ( i = 0; i < arraySize; i++ )
28         cout << setw( 3 ) << a[ i ] ;
29
30     cout << "\n\n\n"
31         << "Effects of passing array element call-by-value:"
32         << "\n\nThe value of a[3] is "<< a[3] << "\n';
33
34     modifyElement( a[ 3 ] );
35
36     cout << "The value of a[ 3 ] is "<< a[ 3 ] << endl;
37
38     return 0;
39 }
40
41 void modifyArray( int b[ ], int sizeofArray )
42 {
43     for ( int j = 0; j < sizeofArray; j++ )
44         b[ j ] *= 2;
45 }
46
47 void modifyElement( int e )
48 {
49     cout << "Value in modifyElement is"
50         << (e *= 2 ) << endl;
51 }

```

输出结果:

Effects of passing entire array call-by-Value:

The values of the original array are:

0 1 2 3 4

The values of the modified array are:

0 2 4 6 8

Effects of passing array element call-by-value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6

图 8.14 向函数传递数组和数组元素

有时程序中的函数不能修改数组元素。由于总是模拟按引用调用传递数组，因此数组中数值的修改很难控制。C++提供类型限定符 `const`，可以防止修改函数中的数组值。数组参数前面加上 `const` 限定符时，数组元素成为函数体中的常量，要在函数体中修改数组元素会造成语法错误。这样，程序员就可以纠正程序，使其不修改数组元素。

图 8.15 演示了 `const` 限定符。函数 `tryToModifyArray` 定义参数 `const int b[]`，指定数组 `b` 为常量，不能修改。函数想修改数组元素会造成语法错误 “Canot modify const object”。`const` 限定符将在第 7 章再次介绍。

```
1 // Fig. 8.15: fig04_1S.cpp
2 // Demonstrating the const type qualifier
3 #include <iostream.h>
4
5 void tryToModifyArray( const int [] );
6
7 int main()
8 {
9     int a[] = {10,20,30};
10
11     tryToModifyArray( a );
12     cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
13     return 0;
14 }
15
16 void tryToModifyArray( const int b[] )
17 {
18     b[ 0 ] /= 2;    // error
19     b[ 1 ] /= 2;    // error
20     b[ 2 ] /= 2;    // error
21 }
```

输出结果：

Compiling FIG04_15.CPP:

Error FIG04_15.CPP 18: Canot modify a const object

Error FIG04_15.CPP 19: Canot modify a const object

Error FIG04_15.CPP 20: Canot modify a const object

Warning FIG04_15.CPP 21: Parameter 'b' is never used

图 8.15 演示 `const` 限定符

8.6 排序数组

排序(sort)数组(即将数据排成特定顺序,如升序或降序)是一个重要的计算应用。银行按账号排序所有支票,使每个月末可以准备各个银行报表。电话公司按姓氏排序账号清单并在同一姓氏中按名字排序,以便于找到电话号码。几乎每个公司都要排序一些数据,有时要排序大量数据。排序数据是个复杂问题,是计算机科学中大量研究的课题。本章介绍最简单的排序机制,在本章练习和第 15 章中,我们要介绍更复杂的机制以达到更高的性能。

图 8.16 的程序排列 10 个元素数组 a 的值,按升序排列。我们使用冒泡排序(bubble sort sinkingsort)方法,较少的数值慢慢从下往上“冒”,就像水中的气泡一样,而较大的值则慢慢往下沉。这个方法在数组中多次操作,每一次都比较一对相邻元素。如果某一对为升序(或数值相等),则将数值保持不变。如果某一对为降序,则将数值交换。

```
1 // Fig. 8.16:fig04 16.cpp
2 // This program sorts an array's values into
3 // ascending order
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 int main()
8 {
9     const int arraySize = 10;
10    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
11    int i, hold;
12
13    cout << "Data items in original order\n";
14
15    for ( i = 0; i < arraySize; i++ )
16        cout << setw( 4 ) << a[ i ] ;
17
18    for (int pass = 0; pass < arraySize - 1; pass++ ) // passes
19
20        for ( i = 0; i < arraySize - 1; i++ )        // one pass
21
22            if ( a[ i ] > a[ i + 1 ] ) {                // one comparison
23                hold = a[ i ];                          // one swap
24                a[ i ] = a[ i + 1 ];
25                a[ i + 1 ] = hold;
26            }
27
28    cout << "\nData items in ascending order\n";
29
30    for ( i = 0; i < arraySize; i++ )
31        cout << setw( 4 ) << a[ i ] ;
```

```

32
33     cout << endl;
34     return 0;
35 }

```

输出结果:

Data	items	in	original	order
2	6	4	8	10 12 89 68 45 37

Data	items	in	ascending	order
2	4	6	8	10 12 37 45 68 89

图 8.16 用冒泡法排序数组

程序首先比较 $a[0]$ 与 $a[1]$, 然后比较 $a[1]$ 与 $a[2]$, 接着是 $a[2]$ 与 $a[3]$, 一直到比较 $a[8]$ 与 $a[9]$ 。尽管有 10 个元素, 但只进行 9 次比较。由于连续进行比较, 因此一次即可能将大值向下移动多位, 但小值只能向上移动一位。第 1 遍, 即可把最大的值移到数组底部, 变为 $a[9]$ 。第 2 遍, 即可将第 2 大的值移到 $a[8]$ 第 9 遍, 将第 9 大的值移到 $a[1]$ 。最小值即为 $a[0]$, 因此, 只进行 9 次比较即可排序 10 个元素的数组。

排序是用嵌套 for 循环完成的。如果需要交换, 则用三条赋值语句完成:

```

hold = a[ i ];
a[i] = a[ i+1 ];
a[ i+1 ] = hold;

```

其中附加的变量 hold 临时保存要交换的两个值之一。只有两个赋值语句是无法进行交换的:

```

a[ i ] = a[ i+1 ];
a[ i+1 ] = a[ i ];

```

例如, 如果 $a[i]$ 为 7 而 $a[i+1]$ 为 5。则第一条赋值语句之后, 两个值均为 5, 数值 7 丢失, 因此要先用变量 hold 临时保存要交换的两个值之一。

冒泡排序的主要优点是易于编程。但冒泡排序的速度很慢, 这在排序大数组时更明显。练习中要开发更有效的冒泡排序程序, 介绍一些比冒泡排序更有效的方法。高级课题中将介绍更深入的排序与查找问题。

8.7 查找数组：线性查找与折半查找

程序员经常要处理数组中存放的大量数据, 可能需要确定数组是否包含符合某关键值(key value)的值。寻找数组中某个元素的过程称为查找(searching)。本节介绍两个查找方法: 简单的线性查找(linear search)方法和更复杂的折半查找(binary search)方法。练习 8.33 和练习 8.34 要求用递归法实现线性查找与折半查找。

图 8.19 的线性查找比较数组中每个元素与查找键(search key)。由于数组没有特定的顺序, 很可能第一个就找到, 也可能要到最后一个才找到。因此, 平均起来, 程序要比较数组中一半的元素才能找到查找键值。要确定哪个值不在数组中, 则程序要比较查找键与数组中每一个元素。

1 // Fig. 8.19: fig04_19.cpp

```

2 // Linear search of an array
3 #include <iostream.h>
4
5 int linearSearch( const int [], int, int );
6
7 int main()
8 {
9     const int arraySize = 100;
10    int a[ arraySize ], searchKey, element;
11
12    for(int x = 0; X < arraySize; x++ )    // create some data
13        a[ x ] = 2 * x;
14
15    cout << "Enter integer search key:" << endl;
16    cin >> searchKey;
17    element = linearSearch( a, searchKey, arraySize );
18
19    if ( element != -1 )
20        cout << "Found value in element " << element << endl;
21    else
22        cout << "Value not found" << endl;
23
24    return 0;
25 }
26
27 int linearsearch( const int arraay[], int key, int sizeofArray )
28 {
29     for (int n = 0; n < sizeofArray; n++ )
30         if( arraay[ n ] == key )
31             return n;
32
33     return -1;
34 }

```

输出结果:

Enter integer search key:

36

found value in element 18

Enter integer search key:

37

Value not found

图 8.19 数组的线性查找

线性查找方法适用于小数组或未排序数组。但是，对于大数组，线性查找是低效的。如果是排序数组，则可以用高速折半查找。折半查找算法在每次比较之后排除所查找数组的一半元素。这个算法找到数组的中间位置，将其与查找键比较。如果相等，则已找到查找键，返回该元素的数组下标。否则将问题简化为查找一半数组。如果查找键小于数组中间元素，则查找数组的前半部分，否则查找数组的后半部分。如果查找键不是指定子数组(原始数组的一部分)中的中间元素，则对原数组的四分之一重复这个算法。查找一直继续，直到查找键等于指定子数组中的中间元素或子数组只剩一个元素且不等于查找键(表示找不到这个查找键)为止。

在最糟糕的情况下，查找 1024 个元素的数组只要用折半查找进行十次比较。重复将 1024 除 2(因为折半查找算法在每次比较之后排除所查找数组的一半元素)得到值 512、256、128、64、32、16、8、4、1 和 1。数值 1024(2¹⁰)除 2 十次之后即变成 1。除以 1 就是折半查找算法的一次比较。1048576(2²⁰)个元素的数组最多只要 20 次比较就可以取得查找键。十亿个元素的数组最多只要 30 次比较就可以取得查找键。这比线性查找的性能大有提高，后者平均要求比较数组元素个数一半的次数。对于十亿个元素的数组，这是 5 亿次比较与 30 次比较的差别。折半查找所需的最大比较次数可以通过大于数组元素个数的第一个 2 的次幂的指数确定。

图 8.20 显示了函数 `binarySearch` 的迭代版本。函数取 4 个参数，一个整数数组 `b`、一个整数 `searchKey`、`low` 数组下标和 `high` 数组下标。如果查找键不符合于数组的中间元素，则调整 `low` 数组下标和 `high` 数组下标，以便查找更小的子数组。如果查找键小于中间元素，则 `high` 数组的下标设置为 `middle-1`，继续查找 `low` 到 `middle-1` 的元素。如果查找键大于中间元素，则 `low` 数组的下标设置为 `middle+1`，继续查找 `middle+1` 到 `high` 的元素。程序使用 15 个元素的数组。大于数组元素个数的第一个 2 的次幂是 16(2⁴)，因此寻找查找键最多只要四次比较。函数 `printHeader` 输出数组下标，函数 `printRow` 输出折半查找过程中的每个子数组。每个子数组的中间元素标上星号(*)，表示用这个元素与查找键比较。

```
1 // Fig. 8.20: fig0420.cpp
2 // Binary search of an array
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int binarySearch( int [], int, int, int, int );
7 void printHeader( iht ,
8 void printRow( int [], int, int, int, int );
9
10 int main()
11 {
12     const iht arraySize = 15;
13     int a[ arraySize ], key, result;
14
15     for ( int i = 0; i < arraySize; i++ )
16         a[ i ] = 2 * i;      // place some data in array
17
18     cout << "Enter a number between 0 and 28: ";
19     cin >> key;
```



```

20
21     printHeader( arraySize );
22     result = binarySearch( a, key, 0, arraySize - 1, arraySize );
23
24         if ( result != -1 )
25             cout << '\n' << key << "found in array element"
26                 << result << endl;
27         else
28             cout << '\n' << key << "not found" << endl;
29
30         return 0;
31 }
32
33 //
34 int binarysearch( int b[ ], int searchKey, int low, int high,
35                  int size )
36 {
37     int middle;
38
39     while ( low <= high ) {
40         middle = ( low + high ) / 2;
41
42         printRow( b, low, middle, high, size );
43
44         if ( searchKey == b[ middle ] ) // match
45             return middle;
46         else if ( searchKey < b[ middle ] )
47             high = middle - 1; // search low end of array
48         else
49             low = middle + 1; // search high end of array
50     }
51
52     return -1; // searchKey not found
53
54
55 // Print a header for the output
56 void printHeader( int size )
57 {
58     cout << "\nSubscripts:\n";
59     for (int i = 0; i < size; i++ )
60         cout << setw( 3 ) << i << ' ';
61
62     cout << '\n';
63
64     for ( i = 1; i <= 4 * size; i++ )

```

```

65         cout << '-';
66
67     cout << endl;
68 }
69
70 // Print one row of output showing the current
71 // part of the array being processed.
72 void printRow( int b[ ], int low, int mid, int high, int size )
73 {
74     for ( int i = 0; i < size; i++ )
75         if ( i < low || i > high )
76             cout << "    ";
77         else if ( i == mid )           // mark middle value
78             cout << setw( 3 ) << b[ i ] << '*';
79         else
80             cout << setw( 3 ) << b[ i ] << ' ';
81
82     cout << endl;
83 }

```

输出结果:

Enter a number between 0 and 28:25

Subscripts:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

0	2	4	6	8	10	12	14*	16	18	20	22	24	26	29
								16	18	20	22*	24	26	28
												24	26*	28
												24*		

25 not found

Enter a number between 0 and 28: 8

Subscripts:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

0	2	4	6	8	10	12	14*	16	18	20	22	24	26	28
0	2	4	6*	8	10	12								
				8	10*	12								
				8*										

8 found in array element 4

Enter a number between 0 and 28: 6

Subscripts:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

0	2	4	6	8	10	12	14*	16	18	20	22	24	26	28
0	2	4	6*	8	10	12								

6 found in array element 3

图 8.20 排序数组的折半查找过程

8.8 多维数组

C++中有多维数组。多维数组常用于表示由行和列组成的表格。要表示表格中的元素，就要指定两个下标：习惯上第一个表示元素的行，第二个表示元素的列。

用两个下标表示特定的表格或数组称为二维数组(double-subscripted array)。注意，多维数组可以有多于两个的下标，C++编译器支持至少 12 个数组下标。图 8.21 演示了二维数组 a。数组包含三行四列，因此是 3 x 4 数组。一般来说，m 行和 n 列的数组称为 m x n 数组。

数组 a 中的每个元素如图 8.21 所示，元素名字表示为 a[i][j]，i 和 j 是数组名，i 和 j 是惟一标识 a 中每个元素的下标。注意第一行元素名的第一个下标均为 0，第四列的元素名的第二个下标均为 3。

把二维数组元素 a[x][y]误写成 a[x, Y]。实际上，a[x, y]等于 a[y]，因为 C++将包含逗号运算符的表达式(x, y)求值为 y(逗号分隔的表达式中最后一个项目)。

图 8.21 三行四列的二维数组

多维数组可以在声明中初始化，就像一维数组一样。例如，二维数组 b[2][2]可以声明和初始化如下：

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

数值用花括号按行分组，因此 1 和 2 初始化 b[0][0]和 b[0][1]，3 和 4 初始化 b[1][0]和 b[1][1]。

如果指定行没有足够的初始化值，则该行的其余元素初始化为 0。这样，下列声明：

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

初始化 b[0][0]为 1、b[0][1]为 0、b[1][0]为 3、b[1][1]为 4。

图 8.22 演示了声明中初始化二维数组。程序声明三个数组，各有两行三列。array1 的声明在两个子列表中提供六个初始化值。第一个子列表初始化数组第一行为 1、2、3，第二个子列表初始化数组第二行为 4、5、6。如果从 array1 初始化值列表中删除每个子列表的花括号，则编译器自动先初始化第一行元素，然后初始化第二行元素。

array2 声明提供六个初始化值。初始化值先赋给第一行，再赋给第二行。任何没有显式初始化值的元素都自动初始化为 0，因此 array2[1][2]自动初始化为 0。

```
1 // Fig. 8.22: fig04_22.cpp
2 // Initializing multidimensional arrays
3 #include <iostream.h>
4
```

```

5 void printArray( int a[ 3 ] );
6
7 int main()
8 {
9     int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } },
10     array2[ 2 ][ 3 ] = { { 1, 2, 3, 4, 5 },
11     array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } } };
12
13     cout << "Values in array1 by row are:" << endl;
14     printArray( array1 );
15
16     cout << "Values in array2 by row are:" << endl;
17     printArray( array2 );
18
19     cout << "Values in array3 by row are:" << endl;
20     printArray( array3 );
21
22     return 0;
23 }
24
25 void printArray( int a[ 3 ] )
26 {
27     for (int i = 0; i < 2; i++ ) {
28
29         for ( int j = 0; j < 3; j++ )
30             cout << a[ i ][ j ] << ' ';
31
32         cout << endl;
33     }
34 }

```

输出结果:

Values in array1 by row are:

1 2 3

4 5 6

Values in array2 by row are:

1 2 3

4 5 0

Values in array3 by row are:

1 2 0

4 0 0

图 8.22 初始化多维数组

array3 的声明在两个子列表中提供 3 个初始化。第一行的子列表显式地将第一行的前两个元素初始化为 1 和 2，第 3 个元素自动初始化为 0。第二行的子列表显式地将第一个元素初始化为 4，最后两个元素自动初始化为 0。

程序调用函数 `printArray` 输出每个数组的元素。注意函数定义指定数组参数为 `int a[][3]`。函数参数中收到单下标数组时，函数参数表中的数组括号是空的。多下标数组第一个下标的长度也不需要，但随后的所有下标长度都是必须的。编译器用这些长度确定多下标数组中元素在内存中的地址。不管下标数是多少，所有数组元素都是在内存中顺序存放。在双下标数组中，第一行后面的内存地址存放第二行。

在参数声明中提供下标使编译器能告诉函数如何找到数组中的元素。在二维数组中，每行是一个一维数组。要找到特定行中的元素，函数要知道每一行有多少元素，以便在访问数组时跳过适当数量的内存地址。这样，访问 `a[1][2]` 时，函数知道跳过内存中第一行的 3 个元素以访问第二行(行 1)，然后访问这一行的第 3 个元素(元素 2)。

许多常见数组操作使用 `for` 重复结构。例如，下列 `for` 结构(见图 8.21)将数组 `a` 第三行的所有元素设置为 0：

```
for(column=0; column<4; column++)
    a[2][Column] =0;
```

我们指定第三行，因此知道第一个下标总是 2 (0 是第一行的下标，1 是第二行的下标)。 `for` 循环只改变第二个下标(即列下标)。上述 `for` 结构等同于下列赋值语句：

```
a[2][0]=0;
a[2][1]=0;
a[2][2]=0;
a[2][3]=0;
```

下列嵌套 `for` 结构确定数组 `a` 中所有元素的总和：

```
total = 0;
for ( row = 0; row < 3; row++ )
    for ( column = 0; column < 4; column++ )
        total += a [ row ] [ column ];
```

`for` 结构一次一行地求数组中所有元素的和。外层 `for` 结构首先设置 `row`(即行下标)为 0,使第一行的元素可以用内层 `for` 结构求和。然后外层 `for` 结构将 `row` 递增为 1，使第二行的元素可以用内层 `for` 结构求和。然后外层 `for` 结构将 `row` 递增为 2，使第三行的元素可以用内层 `for` 结构求和。结束嵌套 `for` 结构之后，打印结果。

图 8.23 的程序对 3 x 4 数组 `studentCrades` 进行几个其他常见数组操作。每行数组表示一个学生，每列表示学生期末考试中 4 门成绩中的一门成绩。数组操作使用 4 个函数进行。函数 `minimum` 确定该学期所有学生考试成绩中的最低成绩。函数 `maximum` 确定该学期所有学生考试成绩中的最高成绩。函数 `average` 确定每个学生该学期的平均成绩。函数 `printArray` 以整齐的表格形式输出二维数组。

```
1 // Fig. 8.23: fig04_23.cpp
2 // Double-subscripted array example
3 #include <iostream.h>
4 #include <iomanip.h>
```

```

5
6 const iht students = 3; // number of students
7 const iht exams = 4; // number of exams
8
9 int minimum( int [][] exams ], int, int );
10 int maximum(int [][] exams ], int, int );
11 float average( int [], int );
12 void printArray( int [][] exams ], int, int );
13
14 int main()
15 {
16     int studentGrades[ students ][ exams ] =
17         { { 77, 68, 86, 73 },
18           { 96, 87, 89, 78 },
19           { 70, 90, 86, 81 } };
20
21     cout << "The array is:\n";
22     printArray( studentGrades, students, exams );
23     cout << "\n\nLowest grade: "
24         << minimum( studentGrades, students, exams )
25         << "\nHighest grade:"
26         << maximum( studentGrades, students, exams ) << '\n';
27
28     for ( int person = 0; person < students; person++ )
29         cout << "The average grade for student" << person << "is"
30
31         << setprecision( 2 )
32         << average( studentGrades[ person ], exams ) << endl;
33
34     return 0;
35
36
37 // Find the minimum grade
38 int minimum( int grades[][] exams ], int pupils, int tests )
39 {
40     int lowGrade = 100;
41
42     for ( int i = 0; i < pupils; i++ )
43         for ( int j = 0; j < tests; j++ )
44
45             if ( grades[ i ][ j ] < lowGrade )
46                 lowGrade = grades[ i ][ j ];
47
48     return lowGrade;
49

```

```

50 }
51
52 // Find the maximum grade
53 int maximum( int grades[][ exams ], iht pupils, iht tests )
54 {
55     int highgrade = 0;
56
57     for ( int i = 0; i < pupils; i++ )
58
59         for (int j = 0; j < tests; j++ )
60
61             if ( grades[ i ][ j ] > highgrade )
62                 highgrade = grades[ i ][ j ];
63
64     return highgrade;
65 }
66
67 // Determine the average grade for a particular student
68 float average(int setofGrades[],int tests)
69 {
70     int total = 0;
71
72     for ( int i = 0; i < tests; i++ )
73         total += setofGrades[ i ];
74
75     return ( float ) total / tests;
76 }
77
78 // Print the array
79 {
80     cout <<"          [ 0 ]  [ 1 ]  [ 2 ]  [ 3 ]";
81     for (int i = 0; i < pupils; i++ ) {
82         cout << "\nstudentGrades[ "<< i << " ] ";
83
84         for(int j=0; j<tests;j++)
85             cout << setw( 5 )
86                 << grades[ i ][ j ];
87     }
88 }
89
90

```

输出结果:

The array is:

```

          [0] [1]  [2] [3]
StudentGrades[ 0 ]  77 68    86   73

```

```
StudentGrades[ 1 ] 96 87 98 78
StudentGrades[ 2 ] 70 90 86 81
```

Lowest grade: 68

Highest grade: 96

The average grade for student 0 is 76.00

The average grade for student 1 is 87.50

The average grade for student 2 is 81.75

图 8.23 使用二维数组举例

函数 `minimum`、`maximum` 和 `printArray` 取得三个参数——`studentGrades` 数组(在每个函数中调用 `grades`)、学生数(数组行)、考试门数(数组列)。每个函数用嵌套 `for` 结构对 `grades` 数组循环。下列嵌套 `for` 结构来自函数 `minimum` 的定义：

```
for(i = 0; i < pupils; i++)
    for(j = 0; j < tests; j++)
        if(grades[i][j] < lowGrade)
            lowGrade = grades[i][j];
```

外层 `for` 结构首先将 `i`(行下标)设置为 0,使第一行的元素可以和内层 `for` 结构体中的变量 `lowGrade` 比较。内层 `for` 结构比较 `lowGrade` 与 4 个成绩中的每一门成绩。如果成绩小于 `lowGrade`,则 `lowGrade` 设置为该门成绩。然后外层 `for` 结构将行下标加到 1,第二行的元素与变量 `lowGrade` 比较。接着外层 `for` 结构行下标加到 2,第三行的元素与变量 `lowGrade` 比较。嵌套 `for` 结构执行完成后,`lowGrade` 包含双下标数组中的最小值。函数 `maximum` 的执行过程与函数 `minimum` 相似。

函数 `average` 取两个参数：一个单下标数组为某个学生的考试成绩,一个数字表示数组中有几个考试成绩。调用 `average` 时,第一个参数 `studentGrades[student]` 指定将双下标数组 `studentGrades` 的特定行传递给 `average`。例如,参数 `studentGrades[1]` 表示双下标数组 `studentGrades` 第二行中存放的 4 个值(成绩的单下标数组)。双下标数组可以看作元素是单下标数组的数组。函数 `average` 计算数组元素的和,将总和除以考试成绩的个数,并返回一个浮点数结果。

第 9 章 指针与字符串

9.1 简介

本章介绍“C++编程语言一个最强大的特性——指针。指针是 C++中最难掌握的问题之一。前面介绍了引用可以用于实现按引用调用。指针使程序可模拟按引用调用，生成与操作动态数据结构，即能够伸缩的数据结构，如链表、队列、堆栈和树。本章介绍基本的指针概念，而且强调了数组、指针与字符串之间的密切关系，并包括一组很好的字符串操作练习。

把数组和字符串看成指针是从 C 语言演变而来的。本书后面会介绍把数组和字符串当作成熟的对象。

9.2 指针变量的声明与初始化

指针变量的值为内存地址。通常变量直接包含特定值，而指针则包含特定值变量的地址。因此可以说，变量名直接(directly)引用数值，而指针间接(indirectly)引用数值(如图 9.1)。通过指针引用数值称为间接引用。

指针和任何其他变量一样，应先声明后使用。下列声明：

```
int *countPtr, count;
```

声明变量 countPtr 的类型为 int*(即指向整型值的指针)，或者说成"countPtr 是 int 的指针"或"countPtr 指向整数类型的对象"。变量 count 声明为整数，而不是整型值的指针。声明中的*只适用于 countPtr。

声明为指针的每个变量前面都要加上星号(*)。例如，下列声明：

```
float *xPtr, *yPtr;
```

表示 xPtr 和 yPtr 都是指向 float 值的指针。声明中以这种方式使用*时，它表示变量声明为指针。指针可以声明为指向任何数据类型的对象。

假设对指针的声明会分配到声明中逗号分隔的指针变量名列表中的所有指针变量名，从而将指针声明为非指针。声明为指针的每个变量前面都要加上星号(*)。

图 9.1 直接和间接引用变量

指针应在声明时或在赋值语句中初始化。指针可以初始化为 0、NULL 或一个地址。数值为 0 或 NULL 的指针不指任何内容。NULL 是头文件<iostream.h>(和另外几个标准库头文件)中定义的符号化常量。将指针初始化为 NULL 等于将指针初始化为 0，但 C++中优先选择 0。指定 0 时，它变为指针的相应类型。数值 0 是惟一可以不将整数转换为指针类型而直接赋给指针变量的整数值。9.3 节将介绍将变量地址赋给指针。

9.3 指针运算符

&(地址)运算符是个一元运算符，返回操作数的地址。例如，假设声明：

```
int y = 5;
int *yPtr;
```

则下列语句：

```
yPtr = &y;
```

将变量 y 的地址赋给指针变量 yPtr。变量 yPtr “指向” y。图 9.2 显示了执行上述语句之后的内存示意图。图中从指针向所指对象画一个箭头表示“指向关系”。

图 9.3 显示了指针在内存中的表示，假设整型变量 y 存放在地址 600000，指针变量 yPtr 存放在地址 500000。地址运算符的操作数应为左值，(即要赋值的项目，如变量名)。地址运算符不能用于常量、不产生引用的表达式和用存储类 `register` 声明的变量。

“*”运算符通常称为间接运算符(indirection operator)或复引用运算符(dereferencing operator)，返回操作数(即指针)所指对象的同义词、别名或浑名。例如(图 9.2 再次引用)，下列语句：

```
cout << *yPtr << endl;
```

指向变量 y 的值(5)，如同下列语句：

```
cout << y << endl;
```

图 9.2 指针指向内存中整数变量的示意图

这里使用*的方法称为复引用指针(dereferencing a pointer)。注意复引用指针也可以用于赋值语句左边，例如下列语句：

```
*yPtr = 9;
```

将数值 9 赋给图 9.3 中的 y。复引用指针也可用于接收输入值，例如：

```
cin >> *yPtr;
```

复引用的指针是个左值。

	yPtr		y
500000	600000	600000	5

图 9.3 指针在内存中的表示

图 9.4 的程序演示了指针运算符。本例中通过<<用十六进制整数输出内存地址(十六进制整数见附录“数值系统”)。

注意 a 的地址和 aPtr 的值在输出中是一致的，说明 a 的地址实际赋给了指针变量 aPtr。&和*运算符是互逆的，如果两者同时作用于 aPtr，则打印相同的结果。图 9.5 显示了前面所介绍的

运算符的优先级和结合律。

```
1 // Fig. 9.4: fig05_04.cpp
2 // Using the & and * operators
3 #include <iostream.h>
4
5 int main()
6 {
7     int a;          // a is an integer
8     int *aPtr;      // aPtr is a pointer to an integer
9
10    a = 7;
11    aPtr = &a;       // aPtr set to address of a
12
13    cout << "The address of a is " << &a
14          << "\nThe value of aPtr is " << aPtr;
15
16    cout << "\n\nThe value of a is " << a
17          << "\nThe value of *aPtr is " << *aPtr;
18
19    cout << "\n\nShowing that * and & are inverses of"
20          << "each other.\n&*aPtr =" << &*aPtr
21          << "\n*&aPtr =" << *&aPtr << endl;
22    return 0;
23 }
```

输出结果：

```
The address of a is Ox0064FDF4
The value of aPtr is 0x0064FDF4
The value of a is 7
The value of *aPtr is 7
Showing that * and & are inverses of each other.
&*aPtr = 0x0064FDF4
*&aPtr = 0x0064FDF4
```

图 9.4 &与*指针运算符

运算符	结合律	类型
() []	从左向右	括号
++ -- + - static_cast<type>()	从右向左	一元
& *		
* / %	从左向右	乘

+	-	从左向右	加
<<	>>	从左向右	插入/读取
<	<=	从左向右	关系
>	>=	从左向右	关系
==	!=	从左向右	相等
&&		从左向右	逻辑 AND
		从左向右	逻辑或
?:		从右向左	条件
=	+=	从右向左	赋值
-=	*=	从右向左	赋值
/=	%=	从右向左	赋值
,		从左向右	逗号

图 9.5 运算符的优先级和结合律

9.4 按引用调用函数

C++用三种方式向函数传递数值：按值调用(call-by-value)、用引用参数按引用调用(call-by-reference reference argument)和用指针参数按引用调用(call-by-reference pointer argument)。第3章比较了按引用调用与按值调用，本章主要介绍用指针参数按引用调用。

第3章曾介绍过，return 可以从被调用函数向调用者返回一个值(或不返回值而从被调用函数返回控制)。我们还介绍了用引用参数将参数传递给函数，使函数可以修改参数的原有值(这样可以从函数“返回”多个值)，或将大的数据对象传递给函数而避免按值调用传递对象的开销(即复制对象所需的开销)。指针和引用一样，也可以修改调用者的一个或几个变量，或将大的数据对象指针传递给函数而避免按值调用传递对象的开销。

在C++中，程序员可以用指针和间接运算符模拟按引用调用(就像C语言程序中的按引用调用一样)。调用函数并要修改参数时，传递该参数地址，通常在要修改数值的变量名前面加上地址运算符(&)。第4章曾介绍过，数组不能用地址运算符(&)传递，因为数组名是内存中数组的开始位置(数组名等同于&arrayName[0])，即数组名已经是个指针。向函数传递参数地址时，可以在函数中使用间接运算符形成变量名的同义词、别名或浑名，并可用其修改调用者内存中该地址的值(如果变量不用const声明)。

图9.6和9.7的程序是计算整数立方函数的两个版本cubeByValue和cubeByReference。图9.6按值调用将变量number传递给函数cubeByValue。函数cubeByValue求出参数的立方，并将新值用return语句返回main，并在main中将新值赋给number。可以先检查函数调用的结果再修改变量值。例如，在这个程序中，可以将cubeByValue的结果存放在另一变量中，检查其数值，然后再将新值赋给number。

```

1 // Fig. 5,6: fig0506.cpp
2 // Cube a variable using call-by-value
3 #include <iostream.h>
4
5 int cubeByValue( int ); // prototype
6
7 int main()
8 {
9     int number = 5;

```

```

10
11  cout << "The original value of number is "<< number;
12  number = cubeByValue( number );
13  cout << "\nThe new value of number is" << number << endl;
14  return 0;
15 }
16
17 int cubeByValue( int n )
18 {
19     return n * n * n;  // cube local variable n
20 }

```

输出结果:

The original value of number is 5

The new value of number is 125

图 9.6 按值调用求出参数的立方

图 9.7 的程序按引用调用传递变量 `number`(传递 `number` 的地址)到函数 `cubeByReference`。函数 `cubeByReference` 取 `nPtr`(`int` 的指针)作为参数。函数复引用指针并求出 `nPtr` 所指值的立方,从而改变 `main` 中的 `number` 值。图 9.8 和 9.9 分别分析了图 9.6 和 9.7 所示程序。

```

1 // Fig. 9.7: fig05_07.cpp
2 // Cube a variable using call-by-reference
3 // with a pointer argument
4 #include <iostream.h>
5
6 void cubeByReference( int* );  // prototype
7
8 int main()
9 {
10     int number = 5;
11
12     cout << "The original value of number is "<< number;
13     cubeByReference( &number );
14     cout << "\nThe new value of number is "<< number << endl;
15     return 0;
16 }
17
18 void cubeByReference( int *nPtr )
19 {
20     *nPtr = *nPtr * *nPtr;  // cube number in main
21 }

```

输出结果:

The original value of number is 5

The new value of number is 125

图 9.7 用指针参数按引用调用求出参数的立方

接收地址参数的函数要定义接收地址的指针参数。例如，`cubeByReference` 的函数首部如下所示:

```
void cubeByReference(int *nPtr)
```

这个函数首部指定函数 `cubeByReference` 接收整型变量的地址(即整型指针)作为参数，在 `nPtr` 中局部存放地址，不返回值。`cubeByReference` 的函数原型包含括号中的 `int*`。和其他变量类型一样，不需要在函数原型中包括指针名。参数名仅用于程序中的说明，编译器将其忽略。

在需要单下标数组参数的函数首部和函数原型中，可以用 `cubeByReference` 参数表中的指针符号。编译器并不区分接收指针的函数和接收单下标数组的函数。当然，函数必须“知道”何时接收数组或要进行按引用调用的单个变量。编译器遇到形如 `int b[]` 的单下标数组函数参数时，编译器将参数变为指针符号 `int* const b`(`b` 是指向整数的常量指针)，`const` 见第 51 节介绍。声明函数参数为单下标数组的两种形式可以互换。

图 9.8 典型的按值调用分析

图 9.9 典型的用指针参数按引用调用分析

9.5 指针与常量限定符

`const` 限定符可以使程序员告诉编译器特定变量的值不能修改。

几年来，大量 C 语言遗留代码都是在没有 `const` 限定符的情况下编写的。因此，使用旧版 C 语言代码的软件工程有很大的改进空间。许多目前使用 ANSI C 和 C++ 的程序员也没有在程序中使用 `const` 限定符，因为他们是从 C 语言的早期版本开始编程的，这些程序员错过了许多改进软件工程的好机会。

函数参数使用或不用 `const` 限定符的可能性有六种，两种用按值调用传递参数，四种按引用调用传递参数，根据最低权限原则来进行选择。在参数中向函数提供完成指定任务所需的数据访问，但不要提供更多权限。

前面曾经介绍，按值调用传递参数时，函数调用中要生成参数副本并将其传递给函数。如果函数中修改副本，则调用者的原值保持不变。许多情况下，需要修改传入函数的值以使函数能够完成任务。但有时即使被调用函数只是操作原值的副本，也不能在被调用函数中修改这个值。假

设函数取一个单下标数组及其长度为参数，并打印数值。这种函数应对数组进行循环并分别输出每个数组元素。函数体中用数组长度确定数组的最高下标，以便在打印完成后结束循环。在函数体中不能改变这个数组长度。

将指针传递给函数有四种方法：非常量数据的非常量指针、常量数据的非常量指针、非常量数据的常量指针和常量数据的常量指针。每种组合提供不同的访问权限。

最高访问权限是非常量数据的非常量指针，可以通过复引用指针而修改，指针可以修改成指向其他数据。声明非常量数据的非常量指针时不用 `const`。这种指针可以接收函数中的字符串，用指针算法处理或修改字符串中的每个字符。图 9.10 中的函数 `convertToUpper` 声明参数 `sPtr(char*sPtr)` 为非常量数据的非常量指针。函数用指针算法一次一个字符地处理字符串 `string`。字符串中，`st` 到，`x`，的字符用函数 `toupper` 变为相应的大写字母，其余字符不变。函数 `toupper` 取一个字符作为参数。如果是小写字母，则返回相应的大写字母，否则返回原字符。函数 `toupper` 是字符处理库 `ctype.h` 中(见第 16 章)的一部分。

常量数据的非常量指针，指针可以修改成指向其他数据，但数据不能通过指针修改。这种指针可以接收函数的数组参数，函数处理数组每个元素而不修改数据。例如，图 9.11 的函数 `printCharacters` 将参数 `sPtr` 声明为 `const char*` 类型，表示“`sPtr` 是字符常量的指针”。函数体用 `for` 循环输出字符串中的每个字符，直到遇到 `null` 终止符。打印每个字符之后，指针 `sPtr` 递增，指向字符串中下一个字符。

```
1 // Fig. 9.10: fig0510.cpp
2 // Converting lowercase letters to uppercase letters
3 // using a non-constant pointer to non-constant data
4 #include <iostream.h>
5 #include <ctype.h>
6
7 void convertToUpper( char * );
8
9 int main()
10 {
11     char string[] = "characters and $32.98";
12
13     cout << "The string before conversion is: "<< string;
14     convertToUpper( string );
15     cout << "\nThe string after conversion is:  "
16     cout << string << endl;
17     return 0;
18 }
19
20 void convertToUpper( char *sPtr )
21 {
22     while ( *sPtr != '\0' ) {
23
24         if ( *sPtr >= 'a' && *sPtr <= 'z' )
25             *sPtr = toupper( *sPtr ); // convert to uppercase
```

```

26
27     ++sPtr; // move sPtr to the next character
28     }
29 }

```

输出结果:

The string before conversion is: characters and \$32.98

The string after conversion is: CHARACTERS AND \$32.98

图 9.10 将字符串变成大写

```

1 // Fig. 9.11:fig05 ll.cpp
2 // Printing a string one character at a time using
3 // a non-constant pointer to constant data
4 #include <iostream.h>
5
6 void printCharacters( const char * );
7
8 int main()
9 {
10     char string[] = "print characters of a string";
11
12     cout << "The string is:\n";
13     printCharacters( string );
14     cout << endl;
15     return 0;
16 }
17
18 // In printCharacters, sPtr is a pointer to a character
19 // constant. Characters cannot be modified through sPtr
20 // (i.e., sPtr is a "read-only" pointer).
21 void printCharacters( const char *sPtr )
22 {
23     for ( ; *sPtr != '\0'; sPtr++ ) // no initialization
24         cout << *sPtr;
25 }

```

输出结果:

The string is:

print characters of a string

图 9.11 用常量数据的非常量指针打印字符串(一次打印一个字符)

图 9.12 演示了函数接收常量数据的非常量指针，并试图通过指针修改数据在编译时产生的语

法错误消息。

```
1 // Fig. 9.12: fig05_12.cpp
2 // Attempting to modify data through a
3 // non-constant pointer to constant data.
4 #include <iostream.h>
5
6 void f( const int* );
7
8 int main()
9 {
10     int y;
11
12     f( &y );    // f attempts illegal modification
13
14     return 0;
15 }
16
17 // In f, xPtr is a pointer to an integer constant
18 void f( const int *xPtr )
19 {
20     *xPtr = 100; // cannot modify a const object
21 }
```

输出结果:

Compiling FIG05_12.CPP:

Error FIG05_12.CPP 20: Cannot modify a const object

Warning FIG05_12.CPP 21: Parameter 'xPtr' is never used

图 9.12 试图通过常量数据的非常量指针修改数据

众所周知,数组是累计数据类型,用同一名称存放相同类型的相关数据项。第 6 章将介绍另一种形式的累计数据类型——结构(structure),也称为记录(record)。结构可以用同一名称存放不同类型的相关数据项(例如,存放公司每个员工的信息)。调用带数组参数的函数时,数组模拟按引用调用自动传递给函数。但结构则总是按值调用,传递整个结构的副本。这就需要复制结构中每个数据项目并将其存放在计算机函数调用堆栈中的执行时开销(函数执行时用函数调用堆栈存放函数调用中使用的局部变量)。结构数据要传递给函数时,可以用常量数据的指针(或常量数据的引用)得到按引用调用的性能和按值调用对数据的保护。传递结构的指针时,只要复制存放结构的地址。在 4 字节地址的机器上,只要复制 4 字节内存而不是复制结构的几百或几千字节。

非常量数据的常量指针总是指向相同的内存地址,该地址中的数据可以通过指针修改。这里的数组名是默认的。数组名是数组开头的常量指针,数组中的所有数据可以用数组名和数组下标访问和修改。非常量数据的常量指针可以接收数组为函数参数,该函数只用数组下标符号访问数组元素。声明为 const 的指针应在声明时初始化(如果是函数参数,则用传入函数的指针初始化)。

图 9.13 的程序想修改常量指针，指针 `ptr` 的类型声明为 `int *const`，图中的声明表示“`ptr` 是整数的常量指针”，指针用整型变量 `x` 的地址初始化。程序要将 `y` 的地址赋给 `ptr`，但产生一个错误消息。注意数值 7 赋给 `*ptr` 时不产生错误，说明 `ptr` 所指的值是可修改的。

常量数据的常量指针的访问权限最低。这种指针总是指向相同的内存地址，该内存地址的数据不能修改。数组传递到函数中，该函数只用数组下标符号读取，而不能修改数组。图 9.14 的程序演示声明指针变量 `ptr` 为 `const int* const`，表示“`ptr` 是常量整数的常量指针”。图中显示了修改 `ptr` 所指数据和修改存放指针变量的地址时产生的错误消息。注意输出 `ptr` 所指的值时不产生错误，因为输出语句中没有进行修改。

```
1 // Fig. 9.13:fig05 13.cpp
2 // Attempting to modify a constant pointer to
3 // non-constant data
4 #include <iostream.h>
5
6 int main()
7 {
8     int x, y;
9
10    int * const ptr = &x; // ptr is a constant pointer to an
11                          // integer. An integer can be modified
12                          // through ptr, but ptr always points
13                          // to the same memory location.
14    *ptr = 7;
15    ptr = &y;
16
17    return 0;
18 }
```

输出结果：

Compiling FIG05 13.CPP:

Error FIG05 13.CPP 15: Cannot modify a const object

Warning FIGOS_13.CPP 18: 'y' is declared but never used

图 9.13 修改非常量数据的常量指针

```
1 // Fig. 9.141 fig05 14.cpp
2 // Attempting to modify a constant pointer to
3 // constant data.
4 #include <iostream.h>
5
6 int main()
7 {
8     int x = 5, y;
```

```

9
10 const iht *const ptr = &x; // ptr is a constant pointer to a
11                             // constant integer, ptr always
12                             // points to the same location
13                             // and the integer at that
14                             // location cannot be modified.
15 eout << *ptr << endl;
16 *ptr = 7;
17 ptr = &y;
18
19 return 0;
20 }

```

输出结果:

Compiling FIG05_14.CPP:

Error FIG05_14.CPP 16: Cannot modify a const object

Error FIG05_14.CPP 17: Cannot modify a const object

Warning FIG05_14.CPP 20: 'y' is declared but never used

图 9.14 修改常量数据的常量指针

9.6 按引用调用的冒泡排序

下面将图 8.16 的冒泡排序程序修改成用两个函数 bubbleSort 和 swap(如图 9.15)。函数 bubbleSort 进行数组排序, 它调用函数 swap, 变换数组元素 array[j]和 array[j+1]记住, C++强制函数之间的信息隐藏, 因此 swap 并不能访问 bubbleSort 中的各个元素。由于 bubbleSort 要求 swap 访问交换的数组元素, 因此 bubbleSort 要将这些元素按引用调用传递给 swap。每个数组元素的地址显式传递。尽管整个数组自动按引用调用传递, 但各个数组元素是标量, 通常按值调用传递。因此, bubbleSort 对 swap 调用中的每个数组元素使用地址运算符(&), 如下所示的语句:

```
swap( &array[ j ], array[j+ 1]);
```

实现按引用调用。函数 swap 用指针变量 element1Ptr 接收&array[j]。由于信息隐藏, swap 并不知道名称&array[j], 但 swap 可以用*element1Ptr 作为&array[j]的同义词。这样, swap 引用*element1Ptr 时, 实际上是引用 bubbleSort 中的&array[j]。同样, swap 引用*element2Ptr 时, 实际上是引用 bubbleSort 中的 array[j+1]。虽然 swap 不能用:

```
hold = array [ j ];
array[ j ] = array[ j + 1 ];
array[ j + 1 ] = hold;
```

但图 9.15 中的 swaP 函数用

```
hold = * element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

达到相同的效果。

```

1 // Fig. 9.15: fig05_15.cpp
2 // This program puts values into an array, sorts the values into
3 // ascending order, and prints the resulting array.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 void bubbleSort( int *, const int );
8
9 int main()
10 {
11     const int arraySize = 10;
12     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13     int i;
14
15     cout << "Data items in original order\n";
16
17     for ( i = 0; i < arraySize; i++ )
18         cout << setw( 4 ) << a[ i ];
19
20     bubbleSort( a, arraySize );           // sort the array
21     cout << "\nData items in ascending order\n";
22
23     for ( i = 0; i < arraySize; i++ )
24         cout << setw( 4 ) << a[ i ];
25
26     cout << endl;
27     return 0;
28 }
29
30 void bubbleSort( int *array, const int size )
31 {
32     void swap( int *, int * );
33
34     for (int pass = 0; pass < size - 1; pass++ )
35
36         for (int j = 0; j < size - 1; j++ )
37
38             if ( array[ j ] > array[ j + 1 ] )
39                 swap( &array[ j ], &array[ j + 1 ] );
40 }
41
42 void swap( int *element1Ptr, int *element2Ptr )
43 {

```

```

44  int hold = *element1Ptr;
45  *element1Ptr = *element2Ptr;
46  *element2Ptr = hold;
47 }

```

输出结果:

Data	item:	in	original	order
2	6	4	8	10
12	89	68	45	37

Data	items	in	ascending	order
2	4	6	8	10
12	37	45	68	89

图 9.15 按引用调用的冒泡排序

注意函数 `bubbleSort` 中的几个特性。函数首部中将 `array` 声明为 `int* array` 而不是 `int array[]`, 表示 `bubbleSort` 接收单下标数组作为参数(这些符号是可以互换的)。参数 `size` 声明为 `const` 以保证最低权限原则。尽管参数 `size` 接收 `main` 中数值的副本, 且修改该副本并不改变 `main` 中的值, 但是 `bubbleSort` 不必改变 `size` 即可完成任务。`bubbleSort` 执行期间数组的长度保持不变, 因此, `size` 声明为 `const` 以保证不被修改。如果排序过程中修改数组长度, 则排序算法无法正确执行。

`bubbleSort` 函数体中包括了函数 `swap` 的原型, 因为它是调用 `swap` 的惟一函数。将原型放在 `bubbleSort` 中, 使得只能从 `bubbleSort` 正确地调用 `swap`。其他函数要调用 `swap` 时无法访问正确的函数原型, 这通常会造成本文所述的语法错误, 因为 C++ 需要函数原型。

注意函数 `bubbleSort` 接收数组长度参数。函数必须知道数组长度才能排序数组。数组传递到函数时, 函数接收数组第一个元素的内存地址。数组长度要单独传递给函数。

通过将函数 `bubbleSort` 定义成接收数组长度作为参数, 可以让函数在排序任何长度单下标整型数组的程序中使用。

数组长度可以直接编程到函数内, 这样会把函数的使用限制在特定长度的数组并减少其复用性。程序中只有处理特定长度的单下标整型数组时才能使用这个函数。

C++ 提供一元运算符 `sizeof`, 确定程序执行期间的数组长度或其他数据类型长度(字节数)。采用数组名时(如图 9.16 所示), `sizeof` 算符返回数组中的总字节数为 `size_t` 类型的值, 通常是 `unsigned int` 类型。这里使用的计算机将 `float` 类型的变量存放在 4 字节内存中, `array` 声明为 20 个元素, 因此 `array` 使用 80 字节内存空间。在接收数组参数的函数中采用指针参数时, `sizeof` 运算符返回指针长度的字节数(4)而不是数组长度。

```

1 // Fig. 9.16: fig05_16.cpp
2 // Sizeof operator when used on an array name
3 // returns the number of bytes in the array.
4 #include <iostream.h>
5
6 size_t getSize( float * );
7
8 int main()

```

```

9 {
10     float array[ 20 ];
11
12     cout << "The number of bytes in the array is"
13         << sizeof( array )
14         << "\nThe number of bytes returned by getSize is"
15         << getSize( array ) << endl;
16
17     return 0;
18 }
19
20 size_t getSize( float *ptr )
21 {
22     return sizeof( ptr );
23 }

```

输出结果:

The number of bytes in the array is 80

The number of bytes returned by getSize is 4

图 9.16 采用数组名时，sizeof 运算符返回数组中的总字节数

数组中的元素个数也可以用两个 sizeof 操作的结果来确定。例如，考虑下列数组声明：

```
double realArray[ 22 ];
```

如果 double 数据类型的变量存放在 8 个字节的内存中，则数组 realArray 总共包含 176 个字节。

要确定数组中的元素个数，可以用下列表达式：

```
sizeof realArray/sizeof(double)
```

这个表达式确定 realArray 数组中的字节数，并将这个值除以内存中存放一个 double 值的字节数，图 9.17 的程序用 size 运算符计算我们所用的个人计算机上存放每种标准数据类型时使用的字节数。

```

1 // Fig. 9.17:fig05 17.cpp
2 // Demonstrating the sizeof operator
3 ~include <iostream.h>
4 ~include <iomanip.h>
5
76 ~nt main()
8     char c;
9     short s;
10    iht i;
11    long l;
12    float f;
13    double d;
14    long double ld;
15    int arras 20 ], *ptr = array;

```

```

16
17 cout << "sizeof c = " << sizeof c
18     << "\tsizeof(char) =" << sizeof( char )
19     << "\nsizeof s = " << sizeof s
20     << "\tsizeof(short) = " << sizeof( short )
21     << "\nsizeof i = " << sizeof i
22     << "\tsizeof(int) =" << sizeof (int )
23     << "\nsizeof l = " << sizeof l
24     << "\tsizeof(long) =" << sizeof( long )
25     << "\nsizeof f = " << sizeof f
26     << "\nsizeof d = " << sizeof d
27     << "\tsizeof(double) =" << sizeof( double )
28     << "\nsizeof ld = " << sizeof ld
29     << "\tsizeof(long double) = " << sizeof( long double )
30     << "\nsizeof array = " << sizeof array
31     << "\nsizeof ptr = " << sizeof ptr
32     << endl;
33 return 0;
34 }

```

输出结果:

```

sizeof c = 1    sizeof(char) = 1
sizeof s = 2    sizeof(short) = 2
sizeof i = 4    sizeof(int) = 4
sizeof l = 4    sizeof(long) = 4
sizeof f = 4    sizeof(float) = 4
sizeof d = 8    sizeof(double) = 8
sizeof ld = 8   sizeof(long double) = 8
sizeof array =
sizeof ptr = 4

```

图 9.17 用 sizeof 运算符计算存放每种标准数据类型时使用的字节数

sizeof 运算符可以用于任何变量名、类型名或常量值。用于变量名(不是数组名)和常量值时，返回存放特定变量或常量类型所用的字节数。注意，如果提供类型名操作数，则 sizeof 使用的括号是必需的；如果提供变量名操作数，则 sizeof 使用的括号不是必需的。记住，sizeof 是个运算符而不是个函数。

9.7 指针表达式与指针算法

指针是算术表达式、赋值表达式和比较表达式中的有效操作数。但是，通常并不是这些表达式中使用的所有运算符都在指针变量中有效。本节介绍可以用指针操作数的运算符及这些运算符的用法。

只有少量操作可以对指针进行。指针可以自增(++)或自减(--), 整数可以加进指针中(+或+=), 也可以从指针中减去整数(-或-=), 指针可以减去另一指针。

假设声明了数组 `int v[5]`, 其第一个元素位于内存地址 3000。假设指针 `vPtr` 已经初始化为指向数组 `v[0]`, 即 `vPtr` 的值为 3000。图 9.18 演示了在 32 位的机器中的这种情况。注意 `vPtr` 可以初始化为数组 `v` 的指针, 如下所示:

```
vPtr = v;  
vPtr = &v[ 0 ]
```

图 9.18 数组 `v` 和指向 `v` 的指针变量 `vPtr`

按照传统算法, $3000+2$ 得到 3002。而指针算法通常不是这样。将指针增加或减去一个整数时, 指针并不是直接增加或减去这个整数, 而是加上指针所指对象长度的这个倍数。这些字节数取决于对象的数据类型。例如, 下列语句:

```
vPtr += 2;
```

在用 4 字节内存空间存储整数时得到的值为 3008($3000+2*4$)。对数组 `v`, 这时 `vPtr` 指向 `v[2]` 如图 9.19。如果用 2 字节内存空间, 则上述结果得到 3004($3000+2*2$)。如果数组为不同数据类型, 则上述语句将指针递增指针所指对象长度的 2 倍。对字符数组进行指针算法时, 结果与普通算法相同, 因为每个字符的长度为一个字节。

图 9.19 经过指针运算之后的 `vPtr`

如果 `vPtr` 递增到 3016, 指向 `v[4]`, 则下列语句:

```
vptr -= 4;
```

将 `vPtr` 复位为 3000, 即数组开头。如果指针加 1 或减 1, 则可以用自增(++)和自减(--)运算符。下列语句:

```
++vptr;
```

```
vPtr++;
```

将指针移到数组中的下一个位置。下列语句:

```
--vPtr;
```

```
vPtr --;
```

将指针移到数组中的前一个位置。

指针变量还可以相减。例如, 如果 `vPtr` 包含地址 3000, `v2Ptr` 包含地址 3008, 则下列语句:


```
x = v2Ptr - vPtr;
```

将 `x` 指定为 `vPtr` 到 `v2Ptr` 的元素个数，这里为 2。指针算法只在对数组进行时才有意义。我们不能假设两个相同类型的变量在内存中相邻的地址存放，除非它们是数组的相邻元素。

如果两个指针的类型相同，则可以将一个指针赋给另一个指针。否则要用强制类型转换运算符将赋值语句右边的指针值转换为赋值语句左边的指针值。这个规则的例外是 `void` 的指针(即 `void*`)，该指针是个一般性指针，可以表示任何指针类型。所有指针类型都可以赋给 `void*` 指针而不需要类型转换。但是，`void*` 指针不能直接赋给另一类型的指针，而要先将 `void*` 指针转换为正确的指针类型。

`void*` 指针不能复引用。例如，编译器知道 `int` 指针指向 32 位机器中的 4 字节内存，但 `void*` 指针只是包含未知数据类型的内存地址，指针所指的字节数是编译器所不知道的。编译器要知道数据类型才能确定该指针复引用时的字节数。对于 `void*` 指针，无法从类型确定字节数。

指针可以用相等和关系运算符比较，但这种比较只在对相同数组成员进行时才有意义。指针比较是对指针存放的地址进行比较。例如，比较指向同一数组的两个指针可以表示一个指针所指的元素号比另一个指针所指的元素号更高。指针比较常用于确定指针是否为 0。

9.8 指针与数组的关系

C++ 中指针与数组关系密切，几乎可以互换使用。数组名可以看成常量指针，指针可以进行任何有关数组下标的操作。

假设声明了整数数组 `b[5]` 和整数指针变量 `bPtr`。由于数组名(不带下标)是数组第一个元素的指针，因此可以用下列语句将 `bPtr` 设置为 `b` 数组第一个元素的地址：

```
bPtr = b;
```

这等于取数组第一个元素的地址，如下所示：

```
bPtr = &b[0];
```

数组元素 `b[3]` 也可以用指针表达式引用：

```
*(bPtr + 3)
```

上述表达式中的 3 是指针的偏移量(offset)。指针指向数组开头时，偏移量表示要引用的数组元素，偏移量值等于数组下标。上述符号称为指针/偏移量符号(pointer / offset notation)。括号是必需的，因为*的优先顺序高于+的优先顺序。如果没有括号，则上述表达式将表达式 `*bPtr` 的值加上 3(即 3 加到 `b[0]` 中，假设 `bPtr` 指向数组开头)。就像数组元素可以用指针表达式引用一样，下列地址：

```
&b[3]
```

可以写成指针表达式：

```
bPtr + 3
```

数组本身可以当作指针并在指针算法中使用。例如，下列表达式：

```
*(b + 3)
```

同样引用数组元素 `b[3]`。一般来说，所有带下标的数组表达式都可以写成指针加偏移量，这时使用指针/偏移量符号，用数组名作为指针。注意，上述语句不修改数组名，`b` 还是指向数组中第一个元素指针和数组一样可以加下标。例如，下列表达式：

```
bPtr[1]
```

指数组元素 `b[1]`。这个表达式称为指针/下标符号(pointer / subscript notation)。

记住，数组名实际上是个常量指针，总是指向数组开头。因此下列表达式：

```
b += 3
```

是无效的，因为该表达式试图用指针算法修改数组名的值。

图 9.20 的程序用我们介绍的四种方法引用数组元素(数组下标、用数组名作为指针的指针 / 偏移量符号、指针下标和指针的指针 / 偏移量符号，打印数组的 4 个元素)。

要演示数组和指针的互换性，还可以看看程序 9.21 中的两个字符串复制函数 `copy1` 和 `copy2`。这两个函数都是将字符串复制到字符数组中。比较 `copy1` 和 `copy2` 的函数原型可以发现，函数基本相同(由于数组和指针具有互换性)。这些函数完成相同的任务，但用不同方法实现。

```
1 // Fig. 9.20: f~g05_20.cpp
2 // Using subscripting and pointer notations with arrays
3
4 #include <iostream.h>
5
6 int main()
7 {
8     int b[] = { 10, 20, 30, 40 };
9     int *bPtr = b; // set bPtr to point to array b
10
11     cout << "Array b printed with:\n"
12         << "Array subscript notation\n";
13
14     for(int i = 0; i < 4; i++ ),
15         cout << "b[" << i << "] = " << b[i] << '\n';
16
17
18     cout << "\nPointer/offset notation where\n"
19         << "the pointer is the array name\n";
20
21     for (int offset = 0; offset < 4; offset++ )
22         cout << "*" (b + " << offset << ") ="
23             << * ( b + offset ) << '\n';
24
25
26     cout << "\nPointer subscript notation\n";
27     for ( i = 0; i < 4; i++ )
28         cout << "bPtr[" << i << "] =" << bPtr[ i ] << '\n';
29     cout << "\nPointer/offset notation\n";
30
31     for ( offset = 0; offset < 4; offset++ )
32         cout << "*" (bPtr + " << offset << ") ="
33             << * ( bPtr + offset ) << '\n';
34
35     return 0;
```

38 }

输出结果:

Array b Printed with:

Array subscript notation

Pointer/offset notation where
the pointer is the array name

* (b + 0) = 10

* (b + 1) = 20

* (b + 2) = 30

* (b + 3) = 40

Pointer subscript notation

bPtr[0] = 10

bPtr[1] = 20

bPtr[2] = 30

bPtr[3] = 40

Pointer/offset notation

*(bPtr + 0) = 10

*(bPtr + 1) = 20

*(bPtr + 2) = 30

*(bPtr + 2) = 40

图 9.20 用我们介绍的四种方法引用数组元素

```
1 // Fig. 9.21: figOS_21.cpp
2 // Copying a string using array notation
3 // and pointer notation.
4 #include <iostream.h>
5
6 void copy1( char *, const char * );
7 void copy2( char *, const char * );
8
9 int main()
10 {
11
12     string3[ 10 ], string4[] = "Good Bye";
13
14     copy1( string1, string2 );
15     cout << "string1 =" << string1 << endl;
16
17     copy2( string3, string4 );
18     cout << "string3 =" << string3 << endl;
19
20     return 0;
```

```

21 }
22
23 // copy s2 to s1 using array notation
24 void copy1( char *s1, const char *s2 )
25 {
26     for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
27         ; // do nothing in body
28 }
29
30 // copy s2 to s1 using pointer notation
31 void copy2( char *s1, const char *s2 )
32 {
33     for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
34         ; // do nothing in body
35 }

```

输出结果:

string1 = Hello

string3 = Good Bye

图 9.21 使用数组和指针符号复制字符串

函数 `copy1` 用数组下标符号将 `s2` 中的字符串复制到字符数组 `s1` 中。函数声明一个作为数组下标的整型计数器变量 `i`。`for` 结构的首部进行整个复制操作，而 `for` 结构体本身是个空结构。首部中指定 `i` 初始化为 0，并在每次循环时加 1。`for` 的条件 “`(s1[i]=s2[i])!='\0'`”，从 `s2` 向 `s1` 一次一个字符地进行复制操作。遇到 `s2` 中的 `null` 终止符时，将其赋给 `s1`，循环终止，因为 `null` 终止符等于 `'\0'`。

记住，赋值语句的值是赋给左边参数的值。

函数 `copy2` 用指针和指针算法将 `s2` 中的字符串复制到 `s1` 字符数组。同样是在 `for` 结构的首部进行整个复制操作。首部没有任何变量初始化。和 `copy1` 中一样，条件 `(*s1=*s2)!='\0'` 进行复制操作。

复引用指针 `s2`，产生的字符赋给复引用的指针 `s1`。进行条件中的赋值之后，指针分别移到指向 `s1` 数组的下一个元素和字符串 `s2` 的下一个字符。遇到 `s2` 中的 `null` 终止符时，将其赋给 `s1`，循环终止。

注意 `copy1` 和 `copy2` 的第一个参数应当是足够大的数组，应能放下第二个参数中的字符串，否则可能会在写人数组边界以外的内存地址时发生错误。另外，注意每个函数中的第二个参数声明为 `const char*` (常量字符串)。在两个函数中，第二个参数都复制到第一个参数，一次一个地从第二个参数复制字符，但不对其做任何修改。因此，第二个参数声明为常量值的指针，实施最低权限原则。两个函数都不需要修改第二个参数，因此不向这两个函数提供修改第二个参数的功能。

9.9 指针数组

数组可以包含指针，这种数据结构的常见用法是构成字符串数组，通常称为字符串数组 (stringarray)。字符串数组中的每项都是字符串，但在 C++ 中，字符串实际上是第一个字符的指针。因此，字符串数组中的每项实际上是字符串中第一个字符的指针。下列字符串数组 `suit` 的声明可以表示一副牌：

```
char *suit[4] = { "Hearts", "Diamonds", "Clubs", "Spades"};
```

声明的 `suit[4]` 部分表示 4 个元素的数组。声明的 `char*` 部分表示数组 `suit` 的每个元素是 `char` 类型的指针。数组中的 4 个值为 "Hearts"、"Diamonds"、"Clubs" 和 "Spades"。每个值在内存中存放成比引号中的字符数多一个字符的 `null` 终止字符串。4 个字符串长度分别为 7、9、6、7。尽管这些字符串好像是放在 `suit` 数组中，其实数组中只存放指针(如图 9.22)。每个指针指向对应字符串中的第一个字符。这样，尽管 `suit` 数组是定长的，但可以访问任意长度的字符串，这是 C++ 强大的数据结构功能所带来的灵活性。

图 9.22 `suit` 数组的图形表示

`suit` 字符串可以放在双下标数组中，每一行表示一个 `suit`，每一列表示 `suit` 名的第一个字符。这种数据结构每一行应有固定列数，能够放下最长的字符串。因此，存放大量字符串而大部分字符串长度均比最长字符串短许多时，可能浪费很多内存空间。我们将在下一节用字符串数组帮助整理一副牌。

9.10 函数指针

函数指针包含函数在内存中的地址。第 4 章介绍了数组名实际上是数组中第一个元素的内存地址。同样，函数名实际上是执行函数任务的代码在内存中的开始地址。函数指针可以传入函数、从函数返回、存放在数组中和赋给其他的函数指针。

要演示如何使用函数指针，我们修改图 9.15 的冒泡排序程序，变成图 9.26 的程序。新程序包括 `main` 和函数 `bubble`、`swap`、`ascending` 和 `descending`。函数 `bubbleSort` 接收 `ascending` 或 `descending` 函数的函数指针参数以及一个整型数组和数组长度。程序提示用户选择按升序或降序排序。如果用户输入 1，则向函数 `bubble` 传递 `ascending` 函数的指针，使数组按升序排列。如果用户输入 2，则向函数 `bubble` 传递 `descending` 函数的指针，使数组按降序排列。图 9.27 显示了示例的执行结果。

```
1 // Fig. 9.26: fig0526.cpp
2 // Multipurpose sorting program using function pointers
3 #include <iostream.h>
4 #include <iomanip.h>
5
```

```

6 void bubble( int [], const int, int (*)( int, int ) );
7 iht ascending( int, int );
8 int descending( int, int );
9
10 int main()
11 {
12     const int arraySize = 10;
13     int order,
14         counter,
15     a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
16
17     cout << "Enter 1 to sort in ascending order,\n"
18         << "Enter 2 to sort    descending order: ";
19     cin >> order;
20     cout << "\nData items in original order\n";
21
22     for ( counter = 0; counter < arraySize; counter++ )
23         cout << setw( 4 ) << a[ counter ];
24
25     if ( order == 1 ) {
26         bubble( a, arraySize, ascending );
27         cout << "\nData items in ascending order\n";
28     }
29     else {
30         bubble( a, arraySize, descending );
31         cout << "\nData items in descending order\n";
32     }
33
34     for ( counter = 0; counter < arraySize; counter++ )
35         cout << setw( 4 ) << a[ counter ];
36
37     cout << endl;
38     return 0;
39 }
40
41 void bubble( int work[ ], const int size,
42             int (*compare)( int, int ) )
43 {
44     void swap( int *, int* );
45
46     for ( int pass = 1; pass < size; pass++ )
47
48         for ( int count = 0; count < size - 1; count++ )
49
50             if ( (*compare)( work[ count ], work[ count + 1 ] ) )

```

```

51     swap( &work[ count ], &work[ count + 1 ] );
52 }
53
54 void swap( int *element1Ptr, int *element2Ptr )
55 {
56     int temp;
57
58     temp = *element1Ptr;
59     *element1Ptr = *element2Ptr;
60     *element2Ptr = temp;
65     return b < a; // swap if b is less than a
66 }
67
68 int descending( int a, int b )
69 {
70     return b > a; // swap if b is greater than a
71 }

```

图 9.26 使用函数指针的多用途排序程序

输出结果:

```

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1
Data items in original order
2   6   4   8   10  12  89  68  45  37
Data items in ascending order
2   4   6   8   10  12  37  45  68  89
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1
Data items in original order
2   6   4   8   10  12  89  68  45  37
Data items in ascending order
89  68  45  37  12  10  9   6   4   2

```

图 9.27 使用函数指针的多用途排序程序的执行结果

注意这里只包括类型，程序员可以加上名称，但参数名只用于程序中的说明，编译器将其忽略。

if 语句中调用传入 bubble 的函数，如下所示：

```
if( (*compare)( work[ count ], work[ count + 1 ] ) )
```

就像复引用变量指针可以访问变量值一样，复引用函数指针可以执行这个函数。

也可以不复引用指针而调用函数，如下所示：

```
if( compare( work[ count ], work[ count + 1 ] ) )
```

直接用指针作为函数名。我们更愿意使用第一种通过指针调用函数的方法，因为它显式说明

compare 是函数指针，通过复引用指针而调用这个函数。第二种通过指针调用函数的方法使 compare 好像是个实际函数。程序用户可能搞糊涂，想看看 compare 函数的定义却怎么也找不到。

函数指针的一个用法是建立菜单驱动系统，提示用户从菜单选择一个选项(例如从 1 到 5)。每个选项由不同函数提供服务，每个函数的指针存放在函数指针数组中。用户选项作为数组下标，数组中的指针用于调用这个函数。

图 9.28 的程序提供了声明和使用函数指针数组的一般例子。这些函数(function1、function2 和 function3)都定义成取整数参数并且不返回值。这些函数的指针存放在数组 f 中，声明如下：

```
void(*f[ 3 ])(int) = { function1, function2, function3 }
```

声明从最左边的括号读起，表示 f 是 3 个函数指针的数组，各取整数参数并返回 void。数组用三个函数名(是指针)初始化。用户输入 0 到 2 的值时，用这些值作为函数指针数组的下标。函数调用如下所示：

```
(*f[ choice ])(choice);
```

调用时，f[choice]选择数组中 choice 位置的指针。复引用指针以调用函数，并将 choice 作为参数传入函数中。每个函数打印自己的参数值和函数名，表示正确调用了这个函数。练习中要开发一个菜单驱动系统。

```
1 // Fig. 9.28: fig05_28.cpp
2 // Demonstrating an array of pointers to functions
3 #include <iostream.h>
4 void function1( int );
5 void function2( iht );
6 void function3( int );
7
8 int main()
9 {
10     void (* f[ 3 ] )( int ) = { function1, function2, function3 };
11     int choice;
12
13     cout << "Enter a number between 0 and 2, 3 to end: ";
14     cin >> choice;
15
16     while ( choice >= 0 && choice < 3 ) {
17         (* f[ choice ] )( choice );
18         cout << "Enter a number between 0 and 2, 3 to end: ";
19         cin >> choice;
20     }
21
22     cout << "Program execution completed." << endl;
23     return 0;
24 }
25
26 void function1( int a )
27 {
28     cout << "You entered " << a
```



```

29         << "so function1 was called\n\n";
30 }
31
32 void function2( int b )
33 {
34     cout << "you entered " << b
35         << "so function2 was called\n\n";
36 }
37
38 void function3( int c )
39 {
40     cout << "You entered " << c
41         << "so function3 was called\n\n";
42 }

```

输出结果:

```

Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called
Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called
Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called
Enter a number between 0 and 2, 3 to end: 3
Program execution completed

```

图 9.28 声明和使用函数的指针数组

9.11 字符与字符串处理简介

本节要介绍一些字符串处理的标准库函数。这里介绍的技术适用于开发文本编辑器、字处理器、桌面排版软件、计算机化打字系统和其他文本处理软件。我们这里使用基于指针的字符串，本书稍后还将介绍把字符串作为成熟的对象。

9.11.1 字符与字符串基础

字符是 C++ 编程语言的基本组件。每个程序都是由一系列字符用有意义的方式组合而成的，计算机将其解释为一系列指令，用来完成一组任务。程序可能包含字符常量(character constant)。字符常量是表示为单引号字符的整数值。字符常量的值是机器字符集中该字符的整数值。例如，'z'表示 z 的整数值(在 ASCII 字符集中为 122)，'\n'表示换行符的整数值(在 ASCII 字符集中为 10)。

字符串就是把一系列字符当作一个单元处理。字符串可能包含字母、数字和+、-、*、/、\$等

各种特殊字符(special character)。C++中的字符串直接量(string literal)或字符串常量(string constant)放在双引号中如下所示:

“John Q. Doe” (姓名)
“9999 Nain Street” (街道)
“Waltham, Massachusetts” (州)
“(201)555—1212” (电话号码)

C++中的字符串是以 `null` 终止符(`'\0'`)结尾的字符数组。通过字符串中第一个字符的指针来访问字符串。字符串的值是字符串中第一个字符的地址(常量), 这样, `c++`中可以说字符串是个常量指针, 是指向字符串中第一个字符的指针。从这个意义上说, 字符串像数组一样, 因为数组名也是第一个元素的(常量)指针。

可以在声明中将字符串指定为字符数组或 `char*`类型的变量。下列声明:

```
char color[] = "blue";  
char *ColorPtr = "blue";
```

分别将变量初始化为“blue”。第一个声明生成 5 个元素的数组 `color`, 包含字符'b'、'l'、'u'、'e'和'w'。第二个声明生成指针变量 `colorPtr`, 指向内存中的字符串“blue”。

声明 `char color[]={"blue"};`也可以改写成:

```
char color[] = {'b','l','u','e','\0'};
```

声明包含字符串的字符数组时, 数组应足够大, 能存放字符串及其 `null` 终止符。上述声明自动根据初始化值列表中提供的初始化值的个数确定数组长度。

字符数组中没有分配能存放字符串及其 `null` 终止符的足够空间。

生成或使用不包含 `null` 终止符的字符串。

字符串可以用 `cin` 通过流读取赋给数组。例如, 下列语句将字符串赋给字符数组 `word[20]`:

```
cin >> word;
```

用户输入的字符串存放在 `word` 中。上述语句读取字符, 直到遇到空格、制表符、换行符或文件结束符。注意, 字符串不能超过 19 个字符, 因为还要留下 `null` 终止符的空间。第 2 章介绍的 `setw` 流操纵符可以用于保证读取到 `word` 的字符串不超过字符数组长度。例如, 下列语句:

```
cin >> setw( 20 ) >> word;
```

指定 `cin` 最多读取 19 个字符到数组 `word` 中, 并将数组第 20 个位置用于保存字符串的 `null` 终止符。`setw` 流操作符只能用于输入的下一个值。

有时, 可以将整行文本输入数组中。为此, C++提供了 `cin.getline` 函数。`cin.getline` 函数取三个参数: 存放该行文本的字符数组、长度和分隔符。例如, 下列程序段:

```
char sentence[80];  
cin.getline(sentence, 80, '\n');
```

声明 80 个字符的数组 `sentence`, 然后从键盘读取一行文本到该数组中。函数遇到分隔符'`\n`'、输入文件结束符或读取的字符数比第二个参数的长度少 1 时停止读取字符(最后一个字符位置用于保存字符串的 `null` 终止符)。如果遇到分隔符, 则读取该分隔符并不再读入下一字符。`cin.getline` 函数的第三个参数默认值为'`\n`', 因此上述函数调用也可以改写成:

```
cin.getline(sentence, 80);
```

第 2 章“C++输入/输出流”中详细介绍了 `cin.getline` 和其他函数。

将单个字符作为字符串处理可能导致致命的运行时错误。字符串是指针, 可以对应一个大整数。而单个字符是个小整数(0--255 的 ASCII 值)在许多系统中, 这会导致错误, 因为低内存地址是用于特殊用途的, 如操作系统中断处理器, 因此会发生“访问无效”的错误。

需要字符串参数时将字符传入函数可能导致致命的运行时错误。

需要字符参数时将字符串传入函数是个语法错误。

9.11.2 字符串处理库的字符串操作函数

字符串处理库提供许多操作字符串数据、比较字符串、搜索字符串中的字符与其他字符串、将字符串标记化(将字符串分成各个逻辑组件)和确定字符串长度的字符串操作函数。本节介绍字符串处理库(标准库)中常用的字符串操作函数。图 9.29 总结了这些函数。

注意图 9.29 中的几个函数包含 `size_t` 数据类型的参数。这是在头文件 `<stddef.h>` (标准库中的头文件, 标准库中还有许多其他标准库头文件, 包括 `<string.h>`) 中定义为 `unsigned int` 或 `unsigned long` 之类的无符号整数类型。

使用字符串处理库中的函数而不包括 `<string.h>` 头文件。

函数 `strcpy` 将第二个参数(字符串)复制到第一个参数(字符数组)中, 这个字符数组的长度应当足以放下字符串及其 `null` 终止符。函数 `strncpy` 与 `strcpy` 相似, 只是 `strncpy` 指定从字符串复制到字符数组的字符数。注意函数 `strncpy` 不一定复制第二个参数的 `null` 终止符, `null` 终止符要在复制的字符数比字符串长度至少多 1 时才复制。例如, 如果第二个参数为 “test”, 则只在 `strncpy` 的第三个参数至少为 5 (“test” 的长度加 `null` 终止符)时才复制 `null` 终止符。如果第三个参数大于 5, 则数组后面添加 `null` 终止符, 直到写入第三个参数指定的总字符数。

函数原型

函数说明

`char *strcpy(char *s1, const char *s2)`

将字符串 `s2` 复制到字符数组 `s1` 中、返回 `s1` 的值

`char *strncpy(char *s1, char *s2, size_t n)`

将字符串 `s2` 中最多 `n` 个字符复制到字符数组 `s1` 中, 返回 `s1` 的值

`char *strcat(char *s1, const char *s2)`

将字符串 `s2` 添加到字符串 `s1` 后面。`s2` 的第一个字符重定义 `s1` 的 `null` 终止符。返回 `s1` 的值

`char *strncat(char *s1, const char *s2, size_t n)`

将字符串 `s2` 中最多 `n` 个字符添加到字符串 `s1` 后面。`s2` 的第一个字符重定义 `s1` 的 `null` 终止符。返回 `s1` 的值

`int strcmp(const char *s1, const char *s2)`

比较字符串 `s1` 与字符串 `s2` 函数在 `s1` 等于、小于或大于 `s2` 时分别返回 0、小于 0 或大于 0 的值

`int strncmp(const char *s1, const char *s2, size_t n)`

比较字符串 `s1` 中的 `n` 个字符与字符串 `s2`。函数在 `s1` 等于、小于或大于 `s2` 时分别返回 0、小于 0 或大于 0 的值

`char *strtok(char *s1, const char *s2)`

用一系列 `strtok` 调用将 `s1` 字符串标记化(将字符串分成各个逻辑组件, 如同一行文本中的每个单词), 用字符串 `s2` 所包含的字符分隔。首次调用时包含 `s1` 为

第一个参数, 后面调用继续标记化同一字符串, 包含 `NULL` 为第一个参数。每次调用时返回当前标记的指针。如果函数调用时不再有更多标记, 则返回 `NULL`

`size_t strlen(const char *s)`

确定字符串的长度, 返回 `null` 终止符之前的字符数

图 9.29 字符串处理库的字符串操作函数

第三个参数小于或等于第二个参数的字符串长度时不在 `strncpy` 的第一个参数中添加 `null` 终止料可能造成严重的运行时错误。

图 9.30 的程序用 `strcpy` 将数组 `x` 中的整个字符串复制到数组 `y` 中，并用 `strncpy` 将数组 `x` 的前 14 个字符复制到数组 `z` 中。将 `null` 字符(`'\0'`)添加到数组 `z`，因为程序中调用 `strncpy` 时没有写入 `null` 终止符(第三个参数小于或等于第二个参数的字符串长度)。

```
1 // Fig. 9.30:fig05 30.cpp
2 // using strcpy and strncpy
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char x[] = "Happy Birthday to You";
9     char x[ 25 ], Z[ 15 ];
10
11     cout << "The string in array x is: "<< x
12         << "\nThe string in array y is: "<< strcpy( y, x )
13         << "\n";
14     strncpy( z, x, 14 ); // does not copy null character
15     z[ 14 ] = '\0';
16     cout << "The string in array z is: " << z << endl;
17
18     return 0;
19 }
```

输出结果:

The string in array x is: Happy Birthday to You

The stringin array y is: Happy Birthday to You

The string in array z is: Happy Birthday

图 9.30 使用 `strcpy` 和 `strncpy` 函数

函数 `strcat` 将第二个参数(字符串)添加到第一个参数(字符数组)中。第二个参数的第一个字符代替终止第一个参数中字符串的 `null` 终止符(`'\0'`)。程序员要保证存放第一个字符串的数组应足以存放第一个字符串、第二个字符串和 `null` 终止符(从第二个字符串复制)的合并长度。函数 `strncat` 从第二个字符串添加指定字符数到第一个字符串中，并在结果中添加 `null` 终止符。图 9.31 的程序演示了函数 `strcat` 和 `strncat`。

图 9.32 用 `strcmp` 和 `strncmp` 比较三个字符串。函数 `strcmp` 一次一个字符地比较第一个字符串参数与第二个字符串参数。如果字符串相等，则函数返回 0；如果第一个字符串小于第二个字符串，则函数返回负值；如果第一个字符串大于第二个字符串，则函数返回正值。函数 `strncmp` 等价于函数 `strcmp`，只是 `strncmp` 只比较到指定字符数。函数 `strncmp` 不比较字符串中 `null` 终止符后面的字符。

程序打印每次函数调用返回的整数值。

假设 `strcmp` 和 `strncmp` 在参数相等时返回 1 是个逻辑错误。`strcmp` 和 `strncmp` 在参数相等时返回 0(C++的 `false` 值)。因此测试两个字符串的相等性时，`strcmp` 和 `strncmp` 的结果应与 0 比较，确定字符串是否相等。

```
1 // Fig. 9.31:fig05 31.cpp
2 // using strcat and strncat
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char s1[ 20 ] = "Happy ";
9     char s2[] = "New Year ";
10    char s3[ 40 ] = "";
11
12    cout << "s1 =" << s1 << "\ns2 =" << s2;
13    cout << "\nstrcat(s1, S2) =" << strcat( s1, s2 );
14    cout << "\nstrncat(s3, s1, 6) =" << strncat( S3, s1, 6 );
15    cout << "\nstrcat(s3, s1) =" << strcat( S3, s1 ) << endl;
16
17    return 0;
18 }
```

输出结果：

s1 = Happy

s2 = New Year

strcat(s1, s2) = Happy New Year

\$trncat{s3, s1, 6) = Happy

strcat(s3, s1) = Happy Happy New Year

图 9.31 使用 `strcat` 和 `strncat` 函数

```
1 // Fig. 9.32: fig0532.cpp
2
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <string.h>
6
7 int main()
8 {
9     char *s1 = "Happy New Year";
10    char *s2 = "Happy New Year";
```

```

11  char *s3 = "Happy Holidays";
12
13  cout << "s1 =" << s1 << "\ns2 =" << s2
14      << "\ns3 =" << s3 << "\n\nstrcmp(s1, S2) ="
15      << setw( 2 ) << strcmp( s1, s2 )
16      << "\nstrcmp(s1, s3) =" << setw( 2 )
17      << strcmp( s1, s3 ) << "\nstrcmp(s3, s1) ="
18      << setw( 2 ) << strcmp( s3, s1 );
19
20  cout << "\n\nstrncmp(s1,  s3,  6) =" << setw( 2 )
21      << strncmp( si, s3, 6)      << "\nstrncmp(s1, s3, 7) ="
22      << setw( 2 ) << strncmp( s1, s3, 7 )
23      << "\nstrncmp(s3, s1, 7) ="
24      << setw( 2 ) << strncmp( s3, s1, 7 ) << endl;
25  return 0;
26 }

```

输出结果:

```

s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays
strcmp(s1, s2) = 0
strcmp( s1, s3) = 1
strcmp( s3, s1) = -1
strncmp(s1, s3, 6) = 0
strncmp(s1, s3, 7) = 1
strncmp(s3, s1, 7) = -1

```

图 9.32 使用 strcmp 和 strncmp 函数

要了解一个字符串大于或小于另一字符串的含义，可以考虑一系列姓氏的字母顺序表。读者一定会把“Jones”放在“Smith”之前，因为“Jones”的第一个字母在“Smith”的第一个字母之前。

但字母表中不仅有 26 个字母，而是个字母顺序表，每个字母在表中有特定位置。“z”不仅表示字母，而且是字母表中第二十六个字母。

计算机怎么知道一个字母在另一字母之前呢？所有字符在计算机中均表示为数字代码，计算机比较两个字符串时，实际上是比较字符串中字符的数字代码。

ASCII 和 EBCDIC 称为字符编码(character code)或字符集(character set)。字符串和字符操作实际上是在操作相应的数字代码，而不是操作字符本身。因此 C++ 中字符和小整数具有互换性。由于数字代码之间有大于、等于、小于的关系，因此可以将不同字符或字符串通过字符编码相互比较。

附录 A 列出了 ASCII 字符编码。

函数 strtok 将字符串分解为一系列标记(token)标记就是一系列用分隔符(delimiting chracter, 通常是空格或标点符号)分开的字符。例如，在一行文本中，每个单词可以作为标记，空格是分

隔符。

需要多次调用 `strtok` 才能将字符串分解为标记(假设字符串中包含多个标记)。第一次调用 `strtok` 包含两个参数, 即要标记化的字符串和包含用来分隔标记的字符的字符串(即分隔符): 在图 9.33 的例子中, 下列语句:

```
tokenPtr = Strtok(string " ");
```

将 `tokenPtr` 赋给 `string` 中第一个标记的指针。`strtok` 的第二个参数 `" "` 表示 `string` 中的标记用空格分开。函数 `strtok` 搜索 `string` 中不是分隔符(空格)的第一个字符, 这是第一个标记的开头。然后函数寻找字符串中的下一个分隔符, 将其换成 `null(, w,)` 字符, 这是当前标记的终点。函数 `strtok` 保存 `string` 中标记后面的下一个字符的指针, 并返回当前标记的指针。

后面再调用 `strtok` 时, 第一个参数为 `NULL`, 继续将 `string` 标记化。`NULL` 参数表示调用 `strtok` 继续从 `string` 中上次调用 `strtok` 时保存的位置开始标记化。如果调用 `strtok` 时已经没有标记, 则 `strtok` 返回 `NULL`。图 9.33 的程序用 `strtok` 将字符串 `" This is sentence with 7 tokens"` 标记化。分别打印每个标记。注意 `strtok` 修改输入字符串, 因此, 如果调用 `strtok` 之后还要在程序中使用这个字符串, 则应复制这个字符串。

没有认识到 `strtok` 修改正在标记化的字符串, 调用 `sstrtok` 后还在程序中使用这个字符串(以为还是原字符串) 函数 `strlen` 取一个字符串作为参数, 并返回字符串中的字符个数, 长度中不包括 `null` 终止符。

图 9.34 的程序演示了函数 `strlen`。

```
1 // Fig. 9.33:fig05 33.cpp
2 // Using strtok
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char string[] = "This is a sentence with 7 tokens";
9     char *tokenPtr;
10
11     cout << "The string to be tokenized is:\n" << string
12         << "\n\nThe tokens are:\n";
13
14     tokenPtr = strtok( string, " ");
15
16     while ( tokenPtr != NULL ) {
17         cout << tokenPtr << '\n';
18         tokenPtr = strtok( NULL, " ");
19     }
20
21     return 0;
22 }
```

输出结果:

The string to be tokenized is:

This is a sentence with 7 tokens

The tokens are:

This

is

a

sentence

7

tokens

图 9.33 使用 strtok 函数

```
1 // Fig. 9.34: fig05_34.cpp
2 // Using strlen
3 #include <iostream>
4 #include<string.h>
5
6 int main()
7 {
8     char *string1 = "abcdefghijklmnopqrstuvwxy";
9     char *string2 = "four";
10    char string3 = "Boston";
11
12    cout << "The length of \"\" << string1
13        << "\" is "<< strlen( string1 )
14        << "\n\nThe length of \"\" << string2
15        << "\" is "<< strlen( string2 )
16        << "\n\nThe length of \"\" << string3
17        << "\" is "<< strlen( string3 ) << endl;
18
19    return 0;
20 }
```

输出结果:

The length of "abcdefghijklmnopqrstuvwxy" is 26

The length of "four" is 4

图 9.34 使用 strlen 函数

第 10 章 信息学奥赛中的常用算法

10.1 算法简介

学习程序设计的人对算法这个词并不陌生，从广义上讲，算法是指为解决一个问题而采用的方法和步骤；从程序设计角度上讲，算法是指利用程序设计语言的各种语句，为解决特定的问题而构成的各种逻辑组合。我们在编写程序的过程就是在实施某种算法，因此程序设计的实质就是用计算机语言构造解决问题的算法。算法是程序设计的灵魂，一个好的程序必须有一个好的算法，一个没有有效算法的程序就像一个没有灵魂的躯体。

算法具有五个特征：

1、有穷性：一个算法应包括有限的运算步骤，执行了有穷的操作后将终止运算，不能是个死循环；

2、确切性：算法的每一步骤必须有确切的定义，读者理解时不会产生二义性。并且，在任何条件下，算法只有唯一的一条执行路径，对于相同的输入只能得出相同的输出。如在算法中不允许有“计算 $8/0$ ”或“将 7 或 8 与 x 相加”之类的运算，因为前者的计算结果是什么不清楚，而后者对于两种可能的运算应做哪一种也不知道。

3、输入：一个算法有 0 个或多个输入，以描述运算对象的初始情况，所谓 0 个输入是指算法本身定义了初始条件。如在 5 个数中找出最小的数，则有 5 个输入。

4、输出：一个算法有一个或多个输出，以反映对输入数据加工后的结果，这是算法设计的目的。它们是同输入有着某种特定关系的量。如上述在 5 个数中找出最小的数，它的输出为最小的数。如果一个程序没有输出，这个程序就毫无意义了；

5、可行性：算法中每一步运算应该是可行的。算法原则上能够精确地运行，而且人能用笔和纸做有限次运算后即可完成。

如何来评价一个算法的好坏呢？主要是从两个方面：

一是看算法运行所占用的时间；我们用时间复杂度来衡量，例如：在以下 3 个程序中，

(1) $x=x+1$

(2) for ($i=1; i \leq n; i++$)

$x=x+1$

(3) for ($i=1; i \leq n; i++$)

For ($j=1; j \leq n; j++$)

$x=x+1$

含基本操作“ x 增 1”的语句 $x:=x+1$ 的出现的次数分别为 1, n 和 n^2 则这三个程序段的时间复杂度分别为 $O(1)$, $O(n)$, $O(n^2)$ ，分别称为常量阶、线性阶和平方阶。在算法时间复杂度的表示中，还有可能出现的有：对数阶 $O(\log n)$ ，指数阶 $O(2^n)$ 等。在 n 很大时，不同数量级的时间复杂度有： $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$ ，很显然，指数阶的算法不是一个好的算法。

二是看算法运行时所占用的空间，既空间复杂度。由于当今计算机硬件技术发展很快，程序所能支配的自由空间一般比较充分，所以空间复杂度就不如时间复杂度那么重要了，有许多问题人们主要是研究其算法的时间复杂度，而很少讨论它的空间耗费。

时间复杂性和空间复杂性在一定条件下是可以相互转化的。在中学生信息学奥赛中，对程序的运行时间作出了严格的限制，如果运行时间超出了限定就会判错，因此在设计算法时首先要考虑的是时间因素，必要时可以以牺牲空间来换取时间，动态规划法就是一种以牺牲空间换取时间

的有效算法。对于空间因素，视题目的要求而定，一般可以不作太多的考虑。

我们通过一个简单的数值计算问题，来比较两个不同算法的效率（在这里只比较时间复杂度）。

例：求 $N!$ 所产生的数后面有多少个 0（中间的 0 不计）。

算法一：从 1 乘到 n ，每乘一个数判断一次，若后面有 0 则去掉后面的 0，并记下 0 的个数。为了不超出数的表示范围，去掉与生成 0 无关的数，只保留有效位数，当乘完 n 次后就得到 0 的个数。（C++程序如下）

```
//=====
// 程序名： N!.cpp
// 程序说明：求 N! 所产生的数后面有多少个 0
// 作者： 杨时龙
// 编写日期： 2011.2.22
//-----
#include <iostream>
#include <fstream>
#include <sstream>
//-----
using namespace std;
//-----
int main()
{
    long int i,t,n,sum;
    t=0;
    sum=1;
    cin>>n;
    for (i=1;i<=n;i++)
    {
        sum=sum*i;
        while (sum % 10 ==0)
        {
            sum=sum / 10;
            t++;
        }
        sum=sum % 1000;
    }
    cout<<t<<endl;
    system("pause"); //等待按任意键结束
}
//=====
```

算法二：此题中生成 0 的个数只与含 5 的个数有关， $n!$ 的分解数中含 5 的个数就等于末尾 0 的个数，因此问题转化为直接求 $n!$ 的分解数中含 5 的个数。

```
//=====
// 程序名： N!2.cpp
// 程序说明：求 N! 所产生的数后面有多少个 0
// 作者： 杨时龙
```

```

// 编写日期: 2011.2.22
//-----
#include <iostream>
#include <fstream>
#include <sstream>
//-----
using namespace std;
//-----

int main()
{
    int t,n;
    t=0;
    cin>>n;
    do
    {
        n=n/5;
        t=t+n;
    }
    while (n>5);
    cout<<t<<endl;
    system("pause"); //等待按任意键结束
}
//=====

```

分析对比两种算法就不难看出，它们的时间复杂度分别为 $O(N)$ 、 $O(\log N)$ ，算法二的执行时间远远小于算法一的执行时间。

在信息学奥赛中，其主要任务就是设计一个有效的算法，去求解所给出的问题。如果仅仅学会一种程序设计语言，而没学过算法的选手在比赛中是不会取得好的成绩的，选手水平的高低在于能否设计出好的算法。

下面，我们根据全国分区联赛大纲的要求，一起来探讨信息学奥赛中的基本算法。

10.2 枚举算法

枚举法，常常称之为穷举法，是指从可能的集合中一一枚举各个元素，用题目给定的约束条件判定哪些是无用的，哪些是有用的。能使命题成立者，即为问题的解。

采用枚举算法解题的基本思路：

- (1) 确定枚举对象、枚举范围和判定条件；
- (2) 一一枚举可能的解，验证是否是问题的解

下面我们就从枚举算法的优化、枚举对象的选择以及判定条件的确定，这三个方面来探讨如何用枚举法解题。

例 1：百钱买百鸡问题：有一个人有一百块钱，打算买一百只鸡。到市场一看，大鸡三块钱一只，小鸡一块钱三只，不大不小的鸡两块钱一只。现在，请你编一程序，帮他计划一下，怎么样买法，才能刚好用一百块钱买一百只鸡？

算法分析：此题很显然就是用枚举法，我们以三种鸡的个数为枚举对象（分别设为 x, y, z ），

以三种鸡的总数 ($x+y+z$) 和买鸡用去的钱的总数 ($x*3+y*2+z$) 为判定条件, 穷举各种鸡的个数。

下面是解这个百鸡问题的程序

```
//=====
// 程序名:  chicken.cpp
// 程序说明: 百钱习百鸡
// 作者:  杨时龙
// 编写日期:  2011.2.19
//-----

#include <iostream>
#include <fstream>
#include <sstream>
//-----

using namespace std;

//-----

int main()
{
    int x, y, z;
    for (x=0; x<=100; x++)
        for (y=0; y<=100; y++)
            for (z=0; z<=100; z++) //枚举所有的解
            {
                if ((x+y+z==100) && (x*3+y*2+z/3==100) && (z % 3==0))
                {
                    cout<<"x="<<x<<"    "<<"y="<<y<<"    "<<"z="<<z<<endl;
                }
            }
    system("pause");
}
//=====
```

上面的条件还有优化的空间, 三种鸡的和是固定的, 我们只要枚举二种鸡 (x, y), 第三种鸡就可以根据约束条件求得 ($z=100-x-y$), 这样就缩小了枚举范围, 请看下面的程序:

```
int main()
{
    int x, y, z;
    for (x=0; x<=100; x++)
        for (y=0; y<=100-x; y++)
        {
            z=100-x-y;
            if ((x+y+z==100) && (x*3+y*2+z/3==100) && (z % 3==0))
                cout<<"x="<<x<<"    "<<"y="<<y<<"    "<<"z="<<z<<endl;
        }
    system("pause");
}
```

```
}
```

未经优化的程序循环了 101^3 次，时间复杂度为 $O(n^3)$ ；优化后的程序只循环了 $(102*101/2)$ 次，时间复杂度为 $O(n^2)$ 。从上面的对比可以看出，对于枚举算法，加强约束条件，缩小枚举的范围，是程序优化的主要考虑方向。

在枚举算法中，枚举对象的选择也是非常重要的，它直接影响着算法的时间复杂度，选择适当的枚举对象可以获得更高的效率。如下例：

例 2、将 1, 2...9 共 9 个数分成三组, 分别组成三个三位数, 且使这三个三位数构成 1:2:3 的比例, 试求出所有满足条件的三个三位数。

例如: 三个三位数 192, 384, 576 满足以上条件. (NOIP1998pj)

算法分析：这是 1998 年全国分区联赛普及组试题（简称 NOIP1998pj，以下同）。此题数据规模不大，可以进行枚举，如果我们不加思地以每一个数位为枚举对象，一位一位地去枚举：

```
for (a=1;a<=9;a++)
    for (b=1;b<=9;b++)
        .....
        For (i=1;i<=9;i++)
```

这样下去，枚举次数就有 9^9 次，如果我们分别设三个数为 $x, 2x, 3x$ ，以 x 为枚举对象，穷举的范围就减少为 9^3 ，在细节上再进一步优化，枚举范围就更少了。程序如下：

```
Int main()
{
    Int t,x;
    String s,st;
    Char c;
    For (x=123;x<=321;x++) //枚举所有可能的解
    {
        t=0;
        str(x,st); //把整数 x 转化为字符串，存放在 st 中
        str(x*2,s); st=st+s;
        str(x*3,s); st=st+s;
        for (c='c';c<='9';c++) //枚举 9 个字符，判断是否都在 st 中
            if (pos(c,st)!=0)
                t++;
            else
                break; //如果不在 st 中，则退出循环
        if (t==9) cout<<x<<" "<<x*2<<" "<<x*3<<endl;
    }
}
```

在枚举法解题中，判定条件的确定也是很重要的，如果约束条件不对或者不全面，就穷举不出正确的结果，我们再看看下面的例子。

例 3 一元三次方程求解(noip2001tg)

问题描述 有形如： $ax^3+bx^2+cx+d=0$ 这样的一个一元三次方程。给出该方程中各项的系数 (a, b, c, d 均为实数)，并约定该方程存在三个不同实根(根的范围在 -100 至 100 之间)，且根与根之差的绝对值 ≥ 1 。

要求由小到大依次在同一行输出这三个实根(根与根之间留有空格),并精确到小数点后 2 位。

提示: 记方程 $f(x)=0$, 若存在 2 个数 x_1 和 x_2 , 且 $x_1 < x_2$, $f(x_1)*f(x_2) < 0$, 则在 (x_1, x_2) 之间一定有一个根。

样例

输入: 1 -5 -4 20

输出: -2.00 2.00 5.00

算法分析: 由题目的提示很符合二分法求解的原理, 所以此题可以用二分法。用二分法解题相对于枚举法来说很复杂很多。此题是否能用枚举法求解呢? 再分析一下题目, 根的范围在 -100 到 100 之间, 结果只要保留两位小数, 我们不妨将根的值域扩大 100 倍 ($-10000 \leq x \leq 10000$), 再以根为枚举对象, 枚举范围是 -10000 到 10000, 用原方程式进行一一验证, 找出方程的解。

有的同学在比赛中是这样做

```
Int main()
{
    Int k;
    double a, b, c, d, x;
    Cin >> a >> b >> c >> d;
    For (k = -10000; k <= 10000; k++)
    {
        x = k/100;
        if (a*x*x*x + b*x*x + c*x + d == 0)
            cout << x << endl;
    }
}
```

用这种方法, 很快就可以把程序编出来, 再将样例数据代入测试也是对的, 等成绩下来才发现这题没有全对, 只得了一半的分。

这种解法为什么是错的呢? 错在哪里? 前面的分析好象也没错啊, 难道这题不能用枚举法做吗? 看到这里大家可能有点迷惑了。

在上面的解法中, 枚举范围和枚举对象都没有错, 而是在验证枚举结果时, 判定条件用错了。因为要保留二位小数, 所以求出来的解不一定是方程的精确根, 再代入 ax^3+bx^2+cx+d 中, 所得的结果也就不一定等于 0, 因此用原方程 $ax^3+bx^2+cx+d=0$ 作为判断条件是不准确的。

我们换一个角度来思考问题, 设 $f(x)=ax^3+bx^2+cx+d$, 若 x 为方程的根, 则根据提示可知, 必有 $f(x-0.005)*(x+0.005) < 0$, 如果我们以此为枚举判定条件, 问题就逆刃而解。另外, 如果 $f(x-0.005)=0$, 那么就说明 $x-0.005$ 是方程的根, 这时根据四舍五入, 方程的根也为 x 。所以我们用 $(f(x-0.005)*f(x+0.005) < 0)$ 和 $(f(x-0.005)=0)$ 作为判定条件。为了程序设计的方便, 我们设计一个函数 $f(x)$ 计算 ax^3+bx^2+cx+d 的值, 程序如下:

```
Int k;
double a, b, c, d, x;
Double f(x:double)    //计算  $ax^3+bx^2+cx+d$  的值
{
    f = ((a*x+b)*x+c)*x+d;
}
Int main()
{
    Cin >> a >> b >> c >> d;
    For (k = -10000; k <= 10000; k++)
```

```

{
    x=k/100;
    if ((f(x-0.005)*f(x+0.005)<0) or (f(x-0.005)==0)) //若 x 两端的函数值异号或 x-0.005 刚好是方程的根, 则确定 x 为方程的根
        cout<<x<<endl;
}
}

```

用枚举法解题的最大的缺点是运算量比较大, 解题效率不高, 如果枚举范围太大 (一般以不超过两百万次为限), 在时间上就难以承受。但枚举算法的思路简单, 程序编写和调试方便, 比赛时也容易想到, 在竞赛中, 时间是有限的, 我们竞赛的最终目标就是求出问题解, 因此, 如果题目的规模不是很大, 在规定的时间与空间限制内能够求出解, 那么我们最好是采用枚举法, 而不需太在意是否还有更快的算法, 这样可以使你有更多的时间去解答其他难题。

10.3 回溯算法

如果上期的“百钱买百鸡”中鸡的种类数是变化的, 用枚举法就无能为力了, 这里介绍另一种算法——回溯法。

回溯法是一种既带有系统性又带有跳跃性的搜索法, 它的基本思想是: 在搜索过程中, 当探索到某一步时, 发现原先的选择达不到目标, 就退回到上一步重新选择。它主要用来解决一些要经过许多步骤才能完成的, 而每个步骤都有若干种可能的分支, 为了完成这一过程, 需要遵守某些规则, 但这些规则又无法用数学公式来描述的一类问题。下面通过实例来了解回溯法的思想及其在计算机上实现的基本方法。

例 1、从 N 个自然数 $(1, 2, \dots, n)$ 中选出 r 个数的所有组合。

算法分析: 设这 r 个数为 a_1, a_2, \dots, a_r , 把它们从大到小排列, 则满足:

- (1) $a_1 > a_2 > \dots > a_r$;
- (2) 其中第 i 位数 $(1 \leq i \leq r)$ 满足 $a_i \geq r-i$;

我们按以上原则先确定第一个数, 再逐位生成所有的 r 个数, 如果当前数符合要求, 则添加下一个数; 否则返回到上一个数, 改变上一个数的值再判断是否符合要求, 如果符合要求, 则继续添加下一个数, 否则返回到上一个数, 改变上一个数的值……按此规则不断循环搜索, 直到找出 r 个数的组合, 这种求解方法就是回溯法。

如果按以上方法生成了第 i 位数 a_i , 下一步的处理为:

- (1) 若 $a_i > r-i$ 且 $i=r$, 则输出这 r 个数并改变 a_i 的值: $a_i = a_i - 1$;
- (2) 若 $a_i > r-i$ 且 $i \neq r$, 则继续生成下一位 $a_{i+1} = a_i - 1$;
- (3) 若 $a_i \leq r-i$, 则回溯到上一位, 改变上一位数的值: $a_{i-1} = a_{i-1} - 1$;

算法实现步骤:

第一步: 输入 n, r 的值, 并初始化: $i=1; a[1] = n$;

第二步: 若 $a[1] > r-1$ 则重复:

- 若 $a[i] > r-i$, ①若 $i=r$, 则输出解, 并且 $a[i] = a[i] - 1$;
- ②若 $i \neq r$, 则继续生成下一位: $a[i+1] = a[i] - 1; i=i+1$;
- 若 $a[i] \leq r-i$, 则回溯: $i=i-1; a[i] = a[i] - 1$;

第三步: 结束;

程序实现

```

Int main()
{
    Int n, r, i, j;

```

```

Int a[10];
Cin>>n>>r;
i=1;a[1]=n;
do
    if (a[i]>r-i)  //{符合条件 }
        if (i==r)  //{输出}
        {
            For (j=1;j<=r;j++)
                Cout<<a[j];
            Cout<<endl;
            a[i]=a[i]-1;
        } else {  //{继续搜索}
            a[i+1]=a[i]-1;
            i++;
        }
    Else  //{回溯}
    {
        i--;
        a[i]=a[i]-1;
    }
while (a[1]==r-1);
}

```

下面我们再通过另一个例子看看回溯在信息学奥赛中的应用。

例 2 数的划分 (noip2001tg)

问题描述 整数 n 分成 k 份, 且每份不能为空, 任意两份不能相同(不考虑顺序)。

例如: $n=7, k=3$, 下面三种分法被认为是相同的。

1, 1, 5; 1, 5, 1; 5, 1, 1;

问有多少种不同的分法。

输入: n, k ($6 \leq n \leq 200, 2 \leq k \leq 6$)

输出: 一个整数, 即不同的分法。

样例

输入: 7 3

输出: 4 {四种分法为: 1,1,5; 1,2,4; 1,3,3; 2,2,3;}

算法分析: 此题可以用回溯法求解, 设自然数 n 拆分为 a_1, a_2, \dots, a_k , 必须满足以下两个条件:

- (1) $n = a_1 + a_2 + \dots + a_k$;
- (2) $a_1 \leq a_2 \leq \dots \leq a_k$ (避免重复计算);

现假设已求得的拆分数为 a_1, a_2, \dots, a_i , 且都满足以上两个条件, 设 $\text{sum} = n - a_1 - a_2 - \dots - a_i$, 我们根据不同的情形进行处理:

- (1) 如果 $i=k$, 则得到一个解, 则计数器 t 加 1, 并回溯到上一步, 改变 a_{i-1} 的值;
- (2) 如果 $i < k$ 且 $\text{sum} \geq a_i$, 则添加下一个元素 a_{i+1} ;
- (3) 如果 $i < k$ 且 $\text{sum} < a_i$, 则说明达不到目标, 回溯到上一步, 改变 a_{i-1} 的值;

算法实现步骤如下:

第一步: 输入自然数 n, k 并初始化; $t=0; i=1; a[i] = 1; \text{sum} = n-1; nk = n \div k$;

第二步: 如果 $a[1] \leq nk$ 重复:

若 $i=k$, 搜索到一个解, 计数器 $t=t+1$; 并回溯;

否则: ①若 $\text{sum} \geq a[i]$ 则继续搜索;

②若 $\text{sum} < a[i]$ 则回溯;

搜索时, $\text{inc}(i); a[i] = a[i-1]; \text{sum} = \text{sum} - a[i];$

回溯时, $\text{dec}(i); \text{inc}(a[i]); \text{sum} = \text{sum} + a[i+1] - 1;$

第三步：输出。

程序如下：

```
Int main()
{
    Int n,nk,sum,i,k;
    Long int t;
    Int a[6];
    Cin>>n>>k;
    nk=n/k;
    t=0; i=1; a[i]=1; sum=n-1; //初始化
    do
        if (i==k)    //判断是否搜索到底
        {
            t++; dec(i); a[i]++;
            sum=sum+a[i+1]-1;    //回溯
        } else {
            If (sum>=a[i])    //判断是否回溯
            {
                i++;
                a[i]=a[i-1];
                sum=sum-a[i];    //继续搜
            } else {
                dec(i); a[i]++;
                sum=sum+a[i+1]-1;    //回溯
            }
        }
    while (a[1]>nk);
    cout<<t<<endl;
}
```

回溯法是通过尝试和纠正错误来寻找答案，是一种通用解题法，在 NOIP 中有许多涉及搜索问题的题目都可以用回溯法来求解。

10.4 递归算法

递归算法的定义：如果一个对象的描述中包含它本身，我们就称这个对象是递归的，这种用递归来描述的算法称为递归算法。

我们先来看看大家熟知的一个的故事：

从前有座山，山上有座庙，庙里有个老和尚在给小和尚讲故事，他说从前有座山，山上有座庙，庙里有个老和尚在给小和尚讲故事，他说……

上面的故事本身是递归的，用递归算法描述：

```
void bonze-tell-story;
{
```

```

if (讲话被打断) 故事结束
else {
    从前有座山，山上有座庙，庙里有个老和尚在给小和尚讲故事；
    bonze-tell-story;
}
};

```

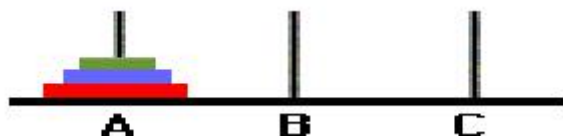
从上面的递归事例不难看出，递归算法存在的两个必要条件：

- (1) 必须有递归的终止条件；
- (2) 过程的描述中包含它本身；

在设计递归算法中，如何将一个问题转化为递归的问题，是初学者面临的难题，下面我们通过分析汉诺塔问题，看看如何用递归算法来求解问题；

例 1：汉诺塔问题，如下图，有 A、B、C 三根柱子。A 柱子上按从小到大的顺序堆放了 N 个盘子，现在要把全部盘子从 A 柱移动到 C 柱，移动过程中可以借助 B 柱。移动时有如下要求：

- (1) 一次只能移动一个盘子；
- (2) 不允许把大盘放在小盘上边；
- (3) 盘子只能放在三根柱子上；



算法分析：当盘子比较多时，问题比较复杂，所以我们先分析简单的情况：

如果只有一个盘子，只需一步，直接把它从 A 柱移动到 C 柱；

如果是二个盘子，共需要移动 3 步：

- (1) 把 A 柱上的小盘子移动到 B 柱；
- (2) 把 A 柱上的大盘子移动到 C 柱；
- (3) 把 B 柱上的大盘子移动到 C 柱；

如果 N 比较大时，需要很多步才能完成，我们先考虑是否能把复杂的移动过程转化为简单的移动过程，如果要把 A 柱上最大的盘子移动到 C 柱上去，必须先把上面的 N-1 个盘子从 A 柱移动到 B 柱上暂存，按这种思路，就可以把 N 个盘子的移动过程分作 3 大步：

- (1) 把 A 柱上面的 N-1 个盘子移动到 B 柱；
- (2) 把 A 柱上剩下的一个盘子移动到 C 柱；
- (3) 把 B 柱上面的 N-1 个盘子移动到 C 柱；

其中 N-1 个盘子的移动过程又可按同样的方法分为三大步，这样就把移动过程转化为一个递归的过程，直到最后只剩下一个盘子，按照移动一个盘子的方法移动，递归结束。

递归过程：

```

procedure Hanoi (N, A, B, C:integer); {以 B 柱为中转柱将 N 个盘子从 A 柱移动到 C 柱}
begin
    if N=1 then write(A, ' -> ', C) {把盘子直接从 A 移动到 C}
    else begin
        Hanoi (N-1, A, C, B); {以 C 柱为中转柱将 N-1 个盘子从 A 柱移动到 B 柱}
        write(A, ' -> ', C); {把剩下的一个盘子从 A 移动到 C}
        Hanoi (N-1, B, A, C); {以 A 柱为中转柱将 N-1 个盘子从 B 柱移动到 C 柱}
    end;
end;

```

从上面的例子我们可以看出,在使用递归算法时,首先弄清楚简单情况下的解法,然后弄清楚如何把复杂情况归纳为更简单的情况。

在信息学奥赛中有的问题的结构或所处理的数据本身是递归定义的,这样的问题非常适合用递归算法来求解,对于这类问题,我们把它分解为具有相同性质的若干个子问题,如果子问题解决了,原问题也就解决了。

例 2 求先序排列 (NOIP2001pj)

[问题描述]给出一棵二叉树的中序与后序排列。求出它的先序排列。(约定树结点用不同的大写字母表示,长度 ≤ 8)。

[样例] 输入: BADC BDCA 输出: ABCD

算法分析:我们先看看三种遍历的定义:

先序遍历是先访问根结点,再遍历左子树,最后遍历右子树;

中序遍历是先遍历左子树,再访问根结点,最后遍历右子树;

后序遍历是先遍历左子树,再遍历右子树,最后访问根结点;

从遍历的定义可知,后序排列的最后一个字符即为这棵树的根节点;在中序排列中,根结点前面的为其左子树,根结点后面的为其右子树;我们可以由后序排列求得根结点,再由根结点在中序排列的位置确定左子树和右子树,把左子树和右子树各看作一个单独的树。这样,就把一棵树分解为具有相同性质的二棵子树,一直递归下去,当分解的子树为空时,递归结束,在递归过程中,按先序遍历的规则输出求得的各个根结点,输出的结果即为原问题的解。

源程序

```
program noip2001_3;
var  z,h : string;
procedure make(z,h:string); {z 为中序排列,h 为后序排列}
var  s,m : integer;
begin
  m:=length(h); {m 为树的长度}
  write(h[m]); {输出根节点}
  s:=pos(h[m],z); {求根节点在中序排列中的位置}
  if s>1 then make(copy(z,1,s-1),copy(h,1,s-1)); {处理左子树}
  if m>s then make(copy(z,s+1,m-s),copy(h,s,m-s)); {处理右子树}
end;
begin
  readln(z);
  readln(h);
  make(z,h);
end.
```

递归算法不仅仅是用于求解递归描述的问题,在其它很多问题中也可以用到递归思想,如回溯法、分治法、动态规划法等算法中都可以使用递归思想来实现,从而使编写的程序更加简洁。

比如上期回溯法所讲的例 2 《数的划分问题》,若用递归来求解,程序非常短小且效率很高,源程序如下

```
var
  n,k:integer;
  tol:longint;
procedure make(sum,t,d:integer);
var i:integer;
```

```

begin
  if d=k then inc(tol)
  else for i:=t to sum div 2 do make(sum-i,i,d+1);
end;
begin
  readln(n,k);
  tol:=0;
  make(n,1,1);
  writeln(tol);
end.

```

有些问题本身是递归定义的，但它并不适合用递归算法来求解，如斐波那契(Fibonacci)数列，它的递归定义为：

$$F(n)=1 \quad (n=1, 2)$$

$$F(n)=F(n-2)+F(n-1) \quad (n>2)$$

用递归过程描述为：

```

Function fb(n:integer):integer;
Begin
  if n<3 then fb:=1
  else fb:=fb(n-1)+fb(n-2);
End;

```

上面的递归过程，调用一次产生二个新的调用，递归次数呈指数增长，时间复杂度为 $O(2^n)$ ，把它改为非递归：

```

x:=1;y:=1;
for i:=3 to n do
begin
  z:=y;y:=x+y;x:=z;
end;

```

修改后的程序，它的时间复杂度为 $O(n)$ 。

我们在编写程序时是否使用递归算法，关键是看问题是否适合用递归算法来求解。由于递归算法编写的程序逻辑性强，结构清晰，正确性易于证明，程序调试也十分方便，在 NOIP 中，数据的规模一般也不大，只要问题适合用递归算法求解，我们还是可以大胆地使用递归算法。

10.5 递推算法

所谓递推，是指从已知的初始条件出发，依据某种递推关系，逐次推出所要求的各中间结果及最后结果。其中初始条件或是问题本身已经给定，或是通过对问题的分析与化简后确定。

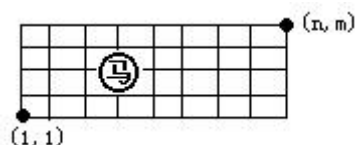
可用递推算法求解的题目一般有以下二个特点：

- (1) 问题可以划分成多个状态；
- (2) 除初始状态外，其它各个状态都可以用固定的递推关系式来表示。

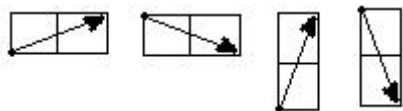
在我们实际解题中，题目不会直接给出递推关系式，而是需要通过分析各种状态，找出递推关系式。

例 1 骑士游历: (noip1997tg)

设有一个 $n \times m$ 的棋盘 ($2 \leq n \leq 50, 2 \leq m \leq 50$), 如下图, 在棋盘上任一点有一个中国象棋马,

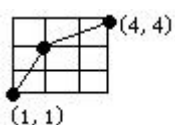


马走的规则为: 1. 马走日字 2. 马只能向右走, 即如下图所示:



任务 1: 当 N, M 输入之后, 找出一条从左下角到右上角的路径。

例如: 输入 $N=4, M=4$

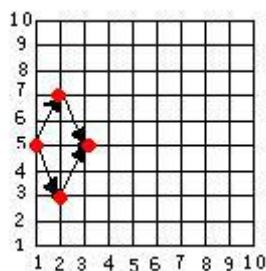


输出: 路径的格式: $(1, 1) \rightarrow (2, 3) \rightarrow (4, 4)$

若不存在路径, 则输出 "no"

任务 2: 当 N, M 给出之后, 同时给出马起始的位置和终点的位置, 试找出从起点到终点的所有路径的数目。

例如: ($N=10, M=10$), $(1, 5)$ (起点), $(3, 5)$ (终点)



输出: 2 (即由 $(1, 5)$ 到 $(3, 5)$ 共有 2 条路径)

输入格式: $n, m, x1, y1, x2, y2$ (分别表示 n, m , 起点坐标, 终点坐标)

输出格式: 路径数目 (若不存在从起点到终点的路径, 输出 0)

算法分析: 为了研究的方便, 我们先将棋盘的横坐标规定为 i , 纵坐标规定为 j , 对于一个 $n \times m$ 的棋盘, i 的值从 1 到 n , j 的值从 1 到 m . 棋盘上的任意点都可以用坐标 (i, j) 表示。对于马的移动方法, 我们用 K 来表示四种移动方向 (1, 2, 3, 4); 而每种移动方法用偏移值来表示, 并将这些偏移值分别保存在数组 dx 和 dy 中, 如下表

K	$Dx[k]$	$Dy[k]$
1	2	1
2	2	-1
3	1	2
4	1	-2

根据马走的规则, 马可以由 $(i - dx[k], j - dy[k])$ 走到 (i, j) 。只要马能从 $(1, 1)$ 走到

$(i-dx[k], j-dy[k])$ ，就一定能走到 (i, j) ，马走的坐标必须保证在棋盘上。我们以 (n, m) 为起点向左递推，当递推到 $(i-dx[k], j-dy[k])$ 的位置是 $(1, 1)$ 时，就找到了一条从 $(1, 1)$ 到 (n, m) 的路径。

我们用一个二维数组 a 表示棋盘，对于任务一，使用倒推法，从终点 (n, m) 往左递推，设初始值 $a[n, m]$ 为 $(-1, -1)$ ，如果从 (i, j) 一步能走到 (n, m) ，就将 (n, m) 存放在 $a[i, j]$ 中。如下表， $a[3, 2]$ 和 $a[2, 3]$ 的值是 $(4, 4)$ ，表示从这两个点都可以到达坐标 $(4, 4)$ 。从 $(1, 1)$ 可到达 $(2, 3)$ 、 $(3, 2)$ 两个点，所以 $a[1, 1]$ 存放两个点中的任意一个即可。递推结束以后，如果 $a[1, 1]$ 值为 $(0, 0)$ 说明不存在路径，否则 $a[1, 1]$ 值就是马走下一步的坐标，以此递推输出路径。

			-1, -1
	4, 4		
		4, 4	
2, 3			

任务一的源程序：

```
const
  dx:array[1..4]of integer=(2, 2, 1, 1);
  dy:array[1..4]of integer=(1, -1, 2, -2);
type
  map=record
    x, y:integer;
  end;
var
  i, j, n, m, k:integer;
  a:array[0..50, 0..50]of map;
begin
  read(n, m);
  fillchar(a, sizeof(a), 0);
  a[n, m].x:=-1; a[n, m].y:=-1; {标记为终点}
  for i:=n downto 2 do {倒推}
    for j:=1 to m do
      if a[i, j].x<>0 then
        for k:=1 to 4 do
          begin
            a[i-dx[k], j-dy[k]].x:=i;
            a[i-dx[k], j-dy[k]].y:=j;
          end;
        if a[1, 1].x=0 then writeln('no')
        else begin{存在路径}
          i:=1; j:=1;
          write('(', i, ', ', j, ')');
          while a[i, j].x<>-1 do
            begin
              k:=i;
              i:=a[i, j].x; j:=a[k, j].y;
              write('->(', i, ', ', j, ')');
```

```

    end;
end;
end.

```

对于任务二，也可以使用递推法，用数组 $A[i, j]$ 存放从起点 $(x1, y1)$ 到 (i, j) 的路径总数，按同样的方法从左向右递推，一直递推到 $(x2, y2)$ ， $a[x2, y2]$ 即为所求的解。源程序（略）

在上面的例题中，递推过程中的某个状态只与前面的一个状态有关，递推关系并不复杂，如果在递推中的某个状态与前面的所有状态都有关，就不容易找出递推关系式，这就需要对实际问题进行分析与归纳，从中找到突破口，总结出递推关系式。我们可以按以下四个步骤去分析：（1）细心的观察；（2）丰富的联想；（3）不断地尝试；（4）总结出递推关系式。

下面我们再看一个复杂点的例子。

例 2、栈（noip2003pj）

题目大意：求 n 个数通过栈后的排列总数。（ $1 \leq n \leq 18$ ）

如输入 3 输出 5

算法分析：先模拟入栈、出栈操作，看看能否找出规律，我们用 $f(n)$ 表示 n 个数通过栈操作后的排列总数，当 n 很小时，很容易模拟出 $f(1)=1$ ； $f(2)=2$ ； $f(3)=5$ ，通过观察，看不出它们之间的递推关系，我们再分析 $N=4$ 的情况，假设入栈前的排列为“4321”，按第一个数“4”在出栈后的位置进行分情况讨论：

- （1）若“4”最先输出，刚好与 $N=3$ 相同，总数为 $f(3)$ ；
- （2）若“4”第二个输出，则在“4”的前只能是“1”，“23”在“4”的后面，这时可以分别看作是 $N=1$ 和 $N=2$ 时的二种情况，排列数分别为 $f(1)$ 和 $f(2)$ ，所以此时的总数为 $f(1)*f(2)$ ；
- （3）若“4”第三个输出，则“4”的前面二个数为“12”，后面一个数为“3”，组成的排列总数为 $f(2)*f(1)$ ；
- （4）若“4”第四个输出，与情况（1）相同，总数为 $f(3)$ ；

所以有： $f(4)=f(3)+f(1)*f(2)+f(2)*f(1)+f(3)$ ；

若设 0 个数通过栈后的排列总数为： $f(0)=1$ ；

上式可变为： $f(4)=f(0)*f(3)+f(1)*f(2)+f(2)*f(1)+f(3)*f(0)$ ；

再进一步推导，不难推出递推式：

$f(n)=f(0)*f(n-1)+f(1)*f(n-2)+\dots+f(n-1)*f(0)$ ；

即 $f(n)=\sum_{0 \leq i \leq n-1} f(i)*f(n-i-1) \quad (n \geq 1)$

初始值： $f(0)=1$ ；

有了以上递推式，就很容易用递推法写出程序。

```

var
  a:array[0..18]of longint;
  n,i,j:integer;
begin
  readln(n);
  fillchar(a,sizeof(a),0);
  a[0]:=1;
  for i:=1 to n do
    for j:=0 to i-1 do a[i]:=a[i]+a[j]*a[i-j-1];
  writeln(a[n]);

```

end.

递推算法最主要的优点是算法结构简单，程序易于实现，难点是从问题的分析中找出解决问题的递推关系式。对于以上两个例子，如果在比赛中找不出递推关系式，也可以用回溯法求解。

10.6 分治算法

分治算法的基本思想是将一个规模为 N 的问题分解为 K 个规模较小的子问题，这些子问题相互独立且与原问题性质相同。求出子问题的解，就可得到原问题的解。

分治法解题的一般步骤：

- (1) 分解，将要解决的问题划分成若干规模较小的同类问题；
- (2) 求解，当子问题划分得足够小时，用较简单的方法解决；
- (3) 合并，按原问题的要求，将子问题的解逐层合并构成原问题的解。

例 1、 比赛安排 (noip1996)

设有 2^n ($n \leq 6$) 个球队进行单循环比赛，计划在 $2^n - 1$ 天内完成，每个队每天进行一场比赛。设计一个比赛的安排，使在 $2^n - 1$ 天内每个队都与不同的对手比赛。例如 $n=2$ 时的比赛安排为：

队	1	2	3	4
比赛	1-2	3-4	第一天	
	1-3	2-4	第二天	
	1-4	2-3	第三天	

算法分析：此题很难直接给出结果，我们先将问题进行分解，设 $m=2^n$ ，将规模减半，如果 $n=3$ (即 $m=8$)，8 个球队的比赛，减半后变成 4 个球队的比赛 ($m=4$)，4 个球队的比赛的安排方式还不是很明显，再减半到两个球队的比赛 ($m=2$)，两个球队的比赛安排方式很简单，只要让两个球队直接进行一场比赛即可：

1	2
2	1

分析两个球队的比赛的情况不难发现，这是一个对称的方阵，我们把这个方阵分成 4 部分 (即左上，右上，左下，右下)，右上部分可由左上部分加 1 (即加 $m/2$) 得到，而右上与左下部分、左上与右下部分别相等。因此我们也可以把这个方阵看作是由 $M=1$ 的方阵所成生的，同理可得 $M=4$ 的方阵：

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

同理可由 $M=4$ 方阵生成 $M=8$ 的方阵：

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

8	7	6	5	4	3	2	1
---	---	---	---	---	---	---	---

这样就构成了整个比赛的安排表。

在算法设计中,用数组 a 记录 2^n 个球队的循环比赛表,整个循环比赛表从最初的 1×1 方阵按上述规则生成 2×2 的方阵,再生成 4×4 的方阵,……直到生成出整个循环比赛表为止。变量 h 表示当前方阵的大小,也就是要生成的下一个方阵的一半。

源程序:

```
var
  i, j, h, m, n: integer;
  a: array[1..32, 1..32] of integer;
begin
  readln(n);
  m:=1; a[1, 1]:=1; h:=1;
  for i:=1 to n do m:=m*2;
  repeat
    for i:=1 to h do
      for j:=1 to h do begin
        a[i, j+h]:=a[i, j]+h; {构造右上角方阵}
        a[i+h, j]:=a[i, j+h]; {构造左下角方阵}
        a[i+h, j+h]:=a[i, j]; {构造右下角方阵}
      end;
    h:=h*2;
  until h=m;
  for i:=1 to m do
    begin
      for j:=1 to m do write(a[i, j]:4);
      writeln;
    end;
  end.
```

在分治算法中,若将原问题分解成两个较小的子问题,我们称之为二分法,由于二分法划分简单,所以使用非常广泛。使用二分法与使用枚举法求解问题相比较,时间复杂度由 $O(N)$ 降到 $O(\log_2 N)$, 在很多实际问题中,我们可以通过使用二分法,达到提高算法效率的目的,如下面的例子。

例2 一元三次方程求解 (noip2001tg)

题目大意: 给出一个一元三次方程 $f(x)=ax^3+bx^2+cx+d=0$, 求它的三个根。所有的根都在区间 $[-100, 100]$ 中, 并保证方程有三个实根, 且它们之间的差不小于 1。

算法分析: 在讲解枚举法时, 我们讨论了如何用枚举法求解此题, 但如果求解的精度进一步提高, 使用枚举法就无能为力了, 在此我们再一次讨论如何用二分法求解此题。

由题意知 $(i, i+1)$ 中若有根, 则只有一个根, 我们枚举根的值域中的每一个整数 $x (-100 \leq x \leq 100)$, 设定搜索区间 $[x_1, x_2]$, 其中 $x_1=x, x_2=x+1$ 。若:

- (1) $f(x_1)=0$, 则确定 x_1 为 $f(x)$ 的根;
- (2) $f(x_1)*f(x_2)<0$, 则确定根 x 在区间 $[x_1, x_2]$ 内。
- (3) $f(x_1)*f(x_2)>0$, 则确定根 x 不在区间 $[x_1, x_2]$ 内, 设定 $[x_2, x_2+1]$ 为下一个搜索区间;

若确定根 x 在区间 $[x_1, x_2]$ 内, 采用二分法, 将区间 $[x_1, x_2]$ 分成左右两个子区间: 左子区间 $[x_1, x]$ 和右子区间 $[x, x_2]$ (其中 $x=(x_1+x_2)/2$)。如果 $f(x_1)*f(x) \leq 0$, 则确定根在左区间 $[x_1,$

$x]$ 内, 将 x 设为该区间的右界值 ($x_2=x$), 继续对左区间进行对分; 否则确定根在右区间 $[x, x_2]$ 内, 将 x 设为该区间的左界值 ($x_1=x$), 继续对右区间进行对分;

上述对分过程一直进行到区间的间距满足精度要求为止 (即 $x_2-x_1<0.005$)。此时确定 x_1 为 $f(x)$ 的根。

源程序:

```
{ $N+ }  
var  
  x:integer;  
  a, b, c, d, x1, x2, xx:extended;  
function f(x:extended):extended;  
begin  
  f:=((a*x+b)*x+c)*x+d;  
end;  
begin  
  read(a, b, c, d);  
  for x:=-100 to 100 do  
    begin  
      x1:=x; x2:=x+1;  
      if f(x1)=0 then write(x1:0:2, ' ' )  
      else if f(x1)*f(x2)<0 then  
        begin  
          while x2-x1>=0.005 do  
            begin  
              xx:=(x1+x2)/2;  
              if f(x1)*f(xx)<=0 then x2:=xx  
              else x1:=xx;  
            end; {while}  
          write(x1:0:2, ' ' );  
        end; {then}  
      end; {for}  
    end.  
  end.
```

10.7 贪心算法

在求最优解问题的过程中, 依据某种贪心标准, 从问题的初始状态出发, 直接去求每一步的最优解, 通过若干次的贪心选择, 最终得出整个问题的最优解, 这种求解方法就是贪心算法。

从贪心算法的定义可以看出, 贪心法并不是从整体上考虑问题, 它所做出的选择只是在某种意义上的局部最优解, 而由问题自身的特性决定了该题运用贪心算法可以得到最优解。

我们看看下面的例子

例 1 均分纸牌 (NOIP2002tg)

[问题描述] 有 N 堆纸牌, 编号分别为 $1, 2, \dots, N$ 。每堆上有若干张, 但纸牌总数必为 N 的倍数。可以在任一堆上取若干张纸牌, 然后移动。移牌规则为: 在编号为 1 堆上取的纸牌, 只能移到编号为 2 的堆上; 在编号为 N 的堆上取的纸牌, 只能移到编号为 $N-1$ 的堆上; 其他堆上取的纸牌, 可以移到相邻左边或右边的堆上。现在要求找出一种移动方法, 用最少的移动次数使每堆上纸牌数都一样多。例如 $N=4$, 4 堆纸牌数分别为:

① 9 ② 8 ③ 17 ④ 6

移动 3 次可达到目的:

从 ③ 取 4 张牌放到 ④ ($9\ 8\ 13\ 10$) \rightarrow 从 ③ 取 3 张牌放到 ② ($9\ 11\ 10\ 10$) \rightarrow 从 ② 取 1 张牌放到 ① ($10\ 10\ 10\ 10$)。

[输入]: 键盘输入文件名。

文件格式: N (N 堆纸牌, $1 \leq N \leq 100$)

$A_1\ A_2\ \dots\ A_n$ (N 堆纸牌, 每堆纸牌初始数, $1 \leq A_i \leq 10000$)

[输出]: 输出至屏幕。格式为: 所有堆均达到相等时的最少移动次数。

[输入输出样例]

a. in:

4
9 8 17 6

屏幕显示: 3

算法分析: 设 $a[i]$ 为第 i 堆纸牌的张数 ($0 \leq i \leq n$), v 为均分后每堆纸牌的张数, s 为最小移到次数。

我们用贪心法, 按照从左到右的顺序移动纸牌。如第 i 堆 ($0 < i < n$) 的纸牌数 $a[i]$ 不等于平均值, 则移动一次 (即 s 加 1), 分两种情况移动:

(1) 若 $a[i] > v$, 则将 $a[i] - v$ 张纸牌从第 i 堆移动到第 $i+1$ 堆;

(2) 若 $a[i] < v$, 则将 $v - a[i]$ 张纸牌从第 $i+1$ 堆移动到第 i 堆;

为了设计的方便, 我们把这两种情况统一看作是将 $a[i] - v$ 张牌从第 i 堆移动到第 $i+1$ 堆; 移动后有: $a[i] := v$; $a[i+1] := a[i+1] + a[i] - v$;

在从第 $i+1$ 堆中取出纸牌补充第 i 堆的过程中, 可能会出现第 $i+1$ 堆的纸牌数小于零 ($a[i+1] + a[i] - v < 0$) 的情况。

如 $n=3$, 三堆纸牌数为 (1, 2, 27) 这时 $v=10$, 为了使第一堆数为 10, 要从第二堆移 9 张纸牌到第一堆, 而第二堆只有 2 张纸牌可移, 这是不是意味着刚才使用的贪心法是错误的呢?

我们继续按规则分析移牌过程, 从第二堆移出 9 张到第一堆后, 第一堆有 10 张纸牌, 第二堆剩下 -7 张纸牌, 再从第三堆移动 17 张到第二堆, 刚好三堆纸牌数都是 10, 最后结果是对的, 从第二堆移出的牌都可以从第三堆得到。我们在移动过程中, 只是改变了移动的顺序, 而移动的次数不变, 因此此题使用贪心法是可行的。

源程序:

```
var
  i, n, s: integer; v: longint;
  a: array[1..100] of longint;
  f: text; fil: string;
begin
  readln(fil);
```

```

assign(f, fil); reset(f);
readln(f, n); v:=0;
for i:=1 to n do begin
    read(f, a[i]); inc(v, a[i]);
end;
v:=v div n; {每堆牌的平均数}
for i:=1 to n-1 do
    if a[i]<>v then {贪心选择}
    begin
        inc(s); {移牌步数计数}
        a[i+1]:=a[i+1]+a[i]-v; {使第 i 堆牌数为 v}
    end; {then}
writeln(s);
end.

```

利用贪心算法解题，需要解决两个问题：

一是问题是否适合用贪心法求解。我们看一个找币的例子，如果一个货币系统有 3 种币值，面值分别为一角、五分和一分，求最小找币数时，可以用贪心法求解；如果将这三种币值改为一角一分、五分和一分，就不能使用贪心法求解。用贪心法解题很方便，但它的适用范围很小，判断一个问题是否适合用贪心法求解，目前还没有一个通用的方法，在信息学竞赛中，需要凭个人的经验来判断何时该使用贪心算法。

二是确定了可以用贪心算法之后，如何选择一个贪心标准，才能保证得到问题的最优解。在选择贪心标准时，我们要对所选的贪心标准进行验证才能使用，不要被表面上看似正确的贪心标准所迷惑，如下面的列子。

例 2 (NOIP1998tg) 设有 n 个正整数，将他们连接成一排，组成一个最大的多位整数。例如： $n=3$ 时，3 个整数 13, 312, 343, 连成的最大整数为：34331213

又如： $n=4$ 时，4 个整数 7, 13, 4, 246 连接成的最大整数为 7424613

输入：N

N 个数

输出：连接成的多位数

算法分析：此题很容易想到使用贪心法，在考试时有很多同学把整数按从大到小的顺序连接起来，测试题目的例子也都符合，但最后测试的结果却不全对。按这种贪心标准，我们很容易找到反例：12, 121 应该组成 12121 而非 12112, 那么是不是相互包含的时候就从小到大呢？也不一定，如：12, 123 就是 12312 而非 12112, 这样情况就有很多种了。是不是此题不能用贪心法呢？

其实此题是可以贪心法来求解，只是刚才的贪心标准不对，正确的贪心标准是：先把整数化成字符串，然后再比较 $a+b$ 和 $b+a$ ，如果 $a+b > b+a$ ，就把 a 排在 b 的前面，反之则把 a 排在 b 的后面。

源程序：

```

var
    s:array[1..20] of string;
    t:string;i,j,k,n:longint;
begin
    readln(n);
    for i:=1 to n do begin

```

```

    read(k);
    str(k, s[i]);
end;
for i:=1 to n-1 do
    for j:=i+1 to n do
        if s[i]+s[j]<s[j]+s[i] then
            begin{交换}
                t:=s[i];
                s[i]:=s[j];
                s[j]:=t;
            end;
        for i:=1 to n do write(s[i]);
    end.

```

贪心算法所作的选择可以依赖于以往所作过的选择，但决不依赖于将来的选择，也不依赖于子问题的解，因此贪心算法与其它算法相比具有一定的速度优势。如果一个问题可以同时用几种方法解决，贪心算法应该是最好的选择之一。

10.8 搜索算法一（深度优先）

在这里介绍两种基本的搜索算法：深度优先搜索和广度优先搜索法，以树的搜索为例，深度优先搜索法是优先扩展尚未扩展的且具有最大深度的结点；广度优先搜索法是在扩展完第 K 层的结点以后才扩展 K+1 层的结点。

深度优先搜索法与前面讲的回溯法差不多，主要的区别是回溯法在求解过程中不保留完整的树结构，而深度优先搜索则记下完整的搜索树，搜索树起记录解路径和状态判重的作用。为了减少存储空间，在深度优先搜索中，用标志的方法记录访问过的状态，这种处理方法使得深度优先搜索法与回溯法没什么区别了。在回溯法中，我们已分析了非递归的实现过程，在这里就只讨论深度优先的递归实现方法。

深度优先搜索的递归实现过程：

```

procedure dfs(i);

    for i:=1 to r do

        if 子结点 mr 符合条件 then 产生的子结点 mr 入栈;

            if 子结点 mr 是目标结点 then 输出

                else dfs(i+1);

            栈顶元素出栈（即删去 mr）;

```

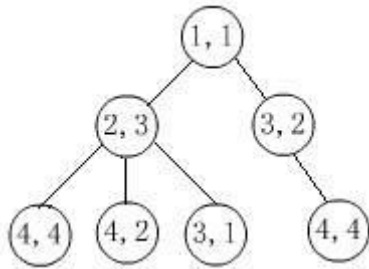
```
endif;
```

```
endfor;
```

在讲解递推法时，我们讨论了用递推法解骑士游历问题，在这里我们再看看如何用深度优先搜索法求解此题。

例 1 骑士游历: 设有一个 $n \times m$ 的棋盘，在棋盘上任一点有一个中国象棋马，马走的规则为: 1. 马走日字 2. 马只能向右走。当 N, M 输入之后，找出一条从左下角到右上角的路径。例如: 输入 $N=4, M=4$ ，输出: 路径的格式: $(1, 1) \rightarrow (2, 3) \rightarrow (4, 4)$ ，若不存在路径，则输出 "no"

算法分析: 我们以 4×4 的棋盘为例进行分析，用树形结构表示马走的所有过程 (如下图)，求从起点到终点的路径，实际上就是从根结点开始深度优先搜索这棵树。



马从 $(1, 1)$ 开始，按深度优先搜索法，走一步到达 $(2, 3)$ ，判断是否到达终点，若没有，则继续往前走，再走一步到达 $(4, 4)$ ，然后判断是否到达终点，若到达则退出，搜索过程结束。为了减少搜索次数，在马走的过程中，判断下一步所走的位置是否在棋盘上，如果不在棋盘上，则另选一条路径再走。

程序如下:

```
const
```

```
dx:array[1..4]of integer=(2, 2, 1, 1);
```

```
dy:array[1..4]of integer=(1, -1, 2, -2);
```

```
type
```

```
map=record
```

```
    x,y:integer;
```

```
end;
```

```
var
```

```
    i,n,m:integer;
```

```
    a:array[0..50]of map;
```

```
procedure dfs(i:integer);
```

```
var j:integer;
```

```
begin
```

```
    for j:=1 to 4 do
```

```
        if (a[i-1].x+dx[j]>0)and(a[i-1].x+dx[j]<=n)
```

```
and(a[i-1].y+dy[j]>0)and(a[i-1].y+dy[j]<=n) then {判断是否在棋盘上}
```

```
        begin
```

```
            a[i].x:=a[i-1].x+dx[j];
```

```

    a[i].y:=a[i-1].y+dy[j];{入栈}
    if (a[i].x=n)and(a[i].y=m) then
    begin
        write(' ',1,',',',1,')');
        for j:=2 to i do write(' ->(',a[j].x,',',',a[j].y,')');
        halt;{输出结果并退出程序}
    end;
    dfs(i+1);{搜索下一步}
    a[i].x:=0;a[i].y:=0;{出栈}
end;
end;
begin
    a[1].x:=1;a[1].y:=1;
    readln(n,m);
    dfs(2);
    writeln('no');
end.

```

从上面的例子我们可以看出，深度优先搜索算法有两个特点：

- 1、已产生的结点按深度排序，深度大的结点先得到扩展，即先产生它的子结点。
- 2、深度大的结点是后产生的，但先得到扩展，即“后产生先扩展”，与栈的工作原理相同，因此用堆栈作为该算法的主要数据结构，存储产生的结点。

对于不同的问题，深度优先搜索算法基本上是一样的，但在具体处理方法和编程技巧上又都不相同，甚至会有很大的差别。我们再看看另一个例子。

题二 选数（存盘名：NOIP2002pj）

[问题描述]:已知 n 个整数 x_1, x_2, \dots, x_n ，以及一个整数 $k(k < n)$ 。从 n 个整数中任选 k 个整数相加，可分别得到一系列的和。例如当 $n=4, k=3$ ，4 个整数分别为 3，7，12，19 时，可得全部的组合与它们的和为：3+7+12=22 3+7+19=29 7+12+19=38 3+12+19=34。现在，要求你计算出和为素数共有多少种。例如上例，只有一种的和为素数：3+7+19=29。

[输入]: 键盘输入，格式为：

```

n , k (1<=n<=20, k<n)
x1,x2, ..., xn (1<=xi<=5000000)

```

[输出]: 屏幕输出，格式为：

一个整数（满足条件的种数）。

[输入输出样例]:

输入：4 3

3 7 12 19

输出：1

算法分析：本题是求从 n 个数中选 k 个数的组合，并使其和为素数。求解此题时，先用深度优先搜索法生成 k 个数的组合，再判断 k 个数的和是否为素数，若为素数则总数加 1。

在程序实现过程中，用数组 a 存放输入的 n 个数，用 s 表示 k 个数的和， ans 表示和为素数的个数。为了避免不必要的搜索，程序对搜索过程进行了优化，限制搜索范围，在搜索过程 $dfs(i,m)$ 中，参数 m 为第 i 个数的上限，下限为 $n-k+i$ 。

源程序：

```
var
  n,k,i: byte;
  ans,s:longint;
  a: array[1 .. 20] of Longint;
procedure prime(s:longint);{判断 K 个数的和是否为素数}
var
  i:integer;
begin
  i:=2;
  while (sqr(i)<=s)and(s mod i<>0) do inc(i);
  if sqr(i)>s then inc(ans){若为素数则总数加 1}
end;

procedure dfs(i,m:byte);{搜索第 i 个数,  }
var
  j:byte;{j 表示第 i 个数的位置}
begin
  for j:=m to n-k+i do{枚举第 i 个数}
  begin
    inc(s,a[j]);{入栈}
    if i=k then prime(s)
    else dfs(i+1,j+1);{继续搜第 i+1 个数}
    dec(s,a[j]){出栈}
  end
end;
begin
  readln(n,k);
  for i:=1 to n do read(a[i]);
  ans:=0; s:=0;
  dfs(1,1);
  writeln(ans);
end.
```


从上面的两个例子我们可以看出，用递归实现深度优先搜索比非递归更加方便。

在使用深度搜索法解题时，搜索的效率并不高，所以要重视对算法的优化，尽可能的减少搜索范围，提高程序的速度。

在下一篇中将介绍另一种搜索方法——广度优先搜索法。

10.9 搜索算法二（广度优先）

在深度优先搜索算法中，深度越大的结点越先得到扩展，若把它改为深度越小的结点越先得到扩展，就是广度优先搜索法。

广度优先搜索基本算法：

```
program bfs;
初始化：建立队列 data;
设队列首指针 closed:=0;队列尾指针 open:=1;
repeat
    closed 增 1，取出 closed 所指结点进行扩展；
    for i:=1 to r do begin
        if 子结点符合条件 then begin
            open 增 1，并把新结点存入数据库队尾；
            if 新结点与原有结点有重复 then 删于该结点(open 减 1)
            else if 新结点即目标 then 输出并退出；
        end{if};
    end{for};
until closed>=open; {队列为空}
```

使用广度优先搜索时，离根结点最近的结点先扩展，所以广度优先搜索法比较适合求步数最少的解，由于深度优先使用了标志法，使得存储空间大大减少，而广度优先要保留所有搜索过的节点，随着搜索程度的加深，所需的存储空间成指数增加。因此在必要时我们采用双向搜索来减少搜索空间和存储空间，如下面的例子。

例 字符串变换（NOIP2002tg）

[问题描述]：已知有两个字符串 A\$, B\$ 及一组字符串变换的规则（至多 6 个规则）：

A1\$ -> B1\$ A2\$ -> B2\$ 规则的含义为：在 A\$ 中的子串 A1\$ 可以变换为 B1\$、A2\$ 可以变换为 B2\$ …。例如：A\$='abcd' B\$='xyz' 变换规则为：'abc' -> 'xu' 'ud' -> 'y' 'y' -> 'yz' 则此时，A\$ 可以经过一系列的变换变为 B\$，其变换的过程为：'abcd' -> 'xud' -> 'xy' -> 'xyz' 共进行了三次变换，使得 A\$ 变换为 B\$。

[输入]：键盘输入文件名。文件格式如下：

A\$ B\$

```

A1$ B1$ \
A2$ B2$  |-> 变换规则
... ... /

```

所有字符串长度的上限为 20。

[输出]:输出至屏幕。格式如下:

若在 10 步 (包含 10 步) 以内能将 A\$ 变换为 B\$, 则输出最少的变换步数; 否则输出 "NO ANSWER!"

[输入输出样例]

b. in:

abcd xyz

abc xu

ud y

y yz

屏幕显示: 3

算法分析: 此题是求变换的最少步数, 很显然可以使用广度优先搜索法, 如果直接从初状态搜到目标状态, 最坏情况下存储的结点数超过 6 的 10 次方幂, 搜索空间过大, 因此我们考虑使双向搜索, 同时从初始状态和目标状态向中间状态搜索, 当相遇时搜索结束。采用双向搜索, 存储的结点数还有可能超限, 我们在前向搜索队列中存储 5 步内变换的结点, 在后向搜索队列中, 由于第 5 步产生的结点只是用来与前向队列中的结点比较, 所以可以不存储在队列中, 后向搜索队列只需存储 4 步内的结点, 这样就解决了存储空间问题。

为了使用方便, 在程序设计中用一个数组 a[1..max] 存储两个队列, 前向搜索队列为 a[1..mid], 后向搜索队列为 a[mid..max], 用 st 存储搜索方向, st=0 表示前向搜索, st=1 表示后向搜索, 用 op[st] 和 cl[st] 分别表示队列尾指针和首指针, 用 be 表示队列起始位置, 循环产生每一个结点, 若在 10 内无解退出循环, 若在 10 内找到解则输出解并退出程序。

源程序:

```

const mid=12000;max=16000;
type
  node=record s:string;x:byte;end;
var
  i,mark:integer;
  a:array [1..max]of ^node;
  x:array[0..6,0..1]of string[20];
  d,fil:string;
  op,cl:array [0..1] of integer;
procedure Init;{读取数据, 初始化}
var f:text;t:string;
begin
  readln(fil);
  assign(f,fil);reset(f);i:=0;
  while not eof(f) do begin
    readln(f,t);

```

```

    x[i, 0] := copy(t, 1, pos(' ', t) - 1);
    x[i, 1] := copy(t, pos(' ', t) + 1, length(t));
    inc(i);
end; {while}
mark := i - 1; close(f);
end;
{判断是否到达目标状态}
procedure bool(be, st: integer);
begin
    for i := mid - be + 1 to cl[1 - st] do
        if a[cl[st]]^.s = a[i]^.s then begin
            writeln(a[cl[st]]^.x + a[i]^.x);
            halt;
        end; {if}
    end;
end;
{判断节点是否与前面的结点重复}
procedure check(be, st: integer);
begin
    for i := be + 1 to cl[st] - 1 do
        if a[i]^.s = a[cl[st]]^.s then
            begin dec(cl[st]); exit; end;
    end;
    bool(be, st);
end;
{扩展产生新节点}
procedure expand(be, st: integer);
var i, j, k, lx, ld: integer;
begin
    inc(op[st]); d := a[op[st]]^.s;
    k := a[op[st]]^.x; ld := length(d);
    for i := 1 to mark do begin
        lx := length(x[i, st]);
        for j := 1 to ld do begin
            if (copy(d, j, lx) = x[i, st]) then begin
                if (st <> 1) or (k <> 4) then begin
                    inc(cl[st]);
                    new(a[cl[st]]);
                end; {if}
                a[cl[st]]^.s := copy(d, 1, j - 1) + x[i, 1 - st] + copy(d, j + lx, ld);
                a[cl[st]]^.x := k + 1;
                check(be, st); {检查是否重复}
            end; {if}
        end; {for}
    end; {for}
end;
end;

```

```

procedure bfs;
var be,k,st:integer;
Begin
  for st:=0 to 1 do begin
    if st=0 then be:=0 else be:=mid;
    op[st]:=be+0;cl[st]:=be+1;
    new(a[cl[st]]);
    a[cl[st]].s:=x[0,st];
    a[cl[st]].x:=0;
  end; {for}
  repeat
    if (op[0]<cl[0])and(a[cl[0]].x<=5)then expand(0,0);
    if (op[1]<cl[1])and(a[cl[1]].x<=5)then expand(mid,1);
  until (op[0]>=cl[0])or (a[cl[0]].x>5)or (op[1]>=cl[1])or (a[cl[1]].x>5);
End;
BEGIN
  init;bfs;writeln('NO ANSWER!')
END.

```

两种搜索算法的比较:

搜索方式	扩展方式	数据结构	适合求解的问题
深度优先	后产生先扩展	栈	可行解或所有解
广度优先	先产生先扩展	队列	最优解

在选择搜索方式时,并不是完全遵循以上原则,具体还是要根据题目的要求而定。在求最优解时,如果搜索的深度不大,我们也可以考虑使用深度优先搜索;在求解可行解时,如果搜索的深度没有限制,或者搜索的代价与搜索的深度成正比,我们也应该使用广度优先搜索。

10.10 动态规划法

在学习动态规划法之前,我们先来了解动态规划的几个概念

- 1、阶段:** 把问题分成几个相互联系的有顺序的几个环节,这些环节即称为阶段。
- 2、状态:** 某一阶段的出发位置称为状态。
- 3、决策:** 从某阶段的一个状态演变到下一个阶段某状态的选择。
- 4、状态转移方程:** 前一阶段的终点就是后一阶段的起点,前一阶段的决策选择导出了后一阶段的状态,这种关系描述了由 k 阶段到 $k+1$ 阶段状态的演变规律,称为状态转移方程。

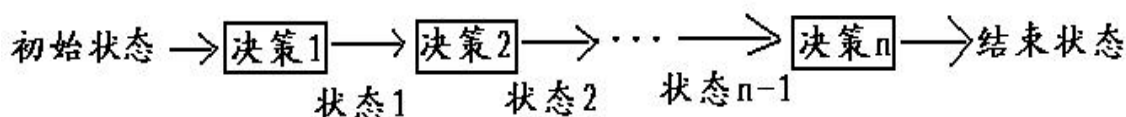
动态规划法的定义:在求解问题中,对于每一步决策,列出各种可能的局部解,再依据某种判定条件,舍弃那些肯定不能得到最优解的局部解,在每一步都经过筛选,以每一步都是最优解来保证全局是最优解,这种求解方法称为动态规划法。

一般来说,适合于用动态规划法求解的问题具有以下特点:

- 1、可以划分成若干个阶段,问题的求解过程就是对若干个阶段的一系列决策过程。
- 2、每个阶段有若干个可能状态

- 3、一个决策将你从一个阶段的一种状态带到下一个阶段的某种状态。
- 4、在任一个阶段，最佳的决策序列和该阶段以前的决策无关。
- 5、各阶段状态之间的转换有明确定义的费用，而且在选择最佳决策时有递推关系（即动态转移方程）。

动态规划法所处理的问题是一个多阶段最优化决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。这些决策形成了一个决策序列，同时确定了完成整个过程的一条活动路线。如下图：



动态规划设计都有着一定的模式，一般要经历以下几个步骤：

- 1、**划分阶段**：按照问题的时间或空间特征，把问题分为若干个阶段。
- 2、**确定状态**：将问题发展到各个阶段时所处的各种客观情况用不同的状态表示出来。
- 3、**确定决策并写出状态转移方程**：因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态，所以如果确定了决策，状态转移方程也就可以写出。
- 4、**寻找边界条件**：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。
- 5、**程序设计实现**：动态规划的主要难点在于理论上的设计，一旦设计完成，实现部分就会非常简单。

根据以上的步骤设计，可以得到动态规划设计的一般模式：

```
for k:=阶段最小值 to 阶段最大值 do {顺推每一个阶段}
  for I:=状态最小值 to 状态最大值 do {枚举阶段 k 的每一个状态}
    for j:=决策最小值 to 决策最大值 do {枚举阶段 k 中状态 i 可选择的每一种决策}
      f[ik]:=min (max) {f[ik-1]+a[ik-1, jk-1]| ik-1通过决策 jk-1可达 ik}
```

有了以上的设计模式，对于简单的动态规划问题，就可以按部就班地进行动态规划设计。

例 1：合唱队形（noip2004tg）

【问题描述】N 位同学站成一排，音乐老师要请其中的(N-K)位同学出列，使得剩下的 K 位同学排成合唱队形。合唱队形是指这样的一种队形：设 K 位同学从左到右依次编号为 1, 2, ..., K，他们的身高分别为 T₁, T₂, ..., T_K，则他们的身高满足 T₁ < T₂ < ... < T_i, T_i > T_{i+1} > ... > T_K (1 ≤ i ≤ K)。你的任务是，已知所有 N 位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

【输入文件】输入文件 **chorus.in** 的第一行是一个整数 N (2 ≤ N ≤ 100)，表示同学的总数。第一行有 n 个整数，用空格分隔，第 i 个整数 T_i (130 ≤ T_i ≤ 230) 是第 i 位同学的身高（厘米）。

【输出文件】输出文件 **chorus.out** 包括一行，这一行只包含一个整数，就是最少需要几位同学出列。

【样例输入】8

186 186 150 200 160 130 197 220

【样例输出】4

算法分析：此题采用动态规划法求解。先分别从左到右求最大上升子序列，从右到左求最大下降子序列，再枚举中间最高的一个人。算法实现起来也很简单，时间复杂度 O(N²)。

我们先考虑如何求最大上升子序列的长度，设 fl(i) 为前 i 个同学的最大上升子序列长度。若要求 fl(i)，必须先求得 fl(1), fl(2), ..., fl(i-1)，再选择一个最大的 fl(j) (j < i)，在前 j 个数中的最大

上升序后添加 T_i ，就可得到前 i 个数的最大上升子序列 $f1(i)$ 。这样就得到状态转移方程：

$$f1(i) = \max \{f1(j) + 1\} \quad (j < i, T_j < T_i)$$

边界条件： $f1(1) = 1$;

设 $f2(i)$ 为后面 $N-i+1$ 位排列的最大下降子序列长度，用同样的方法可以得到状态转移方程：

$$f2(i) = \max \{f2(j) + 1\} \quad (i < j, T_j < T_i); \text{ 边界值为 } f2(N) = 1;$$

有了状态转移方程，程序实现就非常容易了。

源程序：

```
var
    t,f1,f2:array[1..100]of byte;
    i,j,n,max:integer;
begin
    assign(input,'chorus.in');
    reset(input); readln(n);
    for i:=1 to n do begin
        read(t[i]);f1[i]:=1;f2[i]:=1;
    end;{for}
    close(input); max:=0;
    for i:=2 to n do
        for j:=1 to i-1 do begin
            if (t[i]>t[j])and(f1[j]>=f1[i]) then f1[i]:=f1[j]+1; {从左到右求最大上升子序列}
            if (t[n-i+1]>t[n-j+1])and(f2[n-j+1]>=f2[n-i+1]) then f2[n-i+1]:=f2[n-j+1]+1; {从右到左求最大下降子序列}
        end;{for}
    for i:=1 to n do if max<f1[i]+f2[i] then max:=f1[i]+f2[i]; {枚举中间最高的}
    assign(output,'chorus.ans');
    rewrite(output);
    writeln(n-max+1);
    close(output);
end.
```

运用动态规划法求解问题的关键是找出状态转移方程，只要找出了状态转移方程，问题就解决了一半，剩下的事情就是解决如何把状态转移方程用程序实现。

例 2、乘积最大 (noip2000tg)

题目大意：在一个长度为 N 的数字串中插入 r 个乘号，将它分成 $r+1$ 个部分，找出一种分法，使得这 $r+1$ 个部分的乘积最大。

算法分析：此题满足动态规划法的求解标准，我们把它按插入的乘号数来划分阶段，若插入 K 个乘号，可把问题看做是 K 个阶段的决策问题。设 $f[i, k]$ 表示在前 i 位数中插入 K 个乘号所得的最大值， $a[i, j]$ 表示从第 i 位到第 j 位所组成的自然数。用 $f[i, k]$ 存储阶段 K 的每一个状态，可以得状态转移方程：

$$f[i, k] = \max \{f[j, k-1] * a[j+1, i]\} \quad (k \leq j \leq i)$$

$$\text{边界值: } f[j, 0] = a[1, j] \quad (1 \leq j \leq i)$$

根据状态转移方程，我们就很容易写出动态规划程序：

```
for k:=1 to r do  
  
    for i:=k+1 to n do  
  
        for j:=k to I do  
  
            if f[i,k]<f[j,k-1]*a[j+1,I] then f[i,k]:=f[j,k-1]*a[j+1,i]
```

源程序（略）。

近年来，涉及动态规划的各种竞赛题越来越多，每一年的 NOIP 都至少有一道题目需要用动态规划法求解。而应用动态规划法解题是富于技巧性和创造性的，虽然在前面的求解过程中给出了一个解题的基本模式，但由于动态规划题目出现的形式多种多样，并且大部分题目表面上看不出与动态规划的直接联系，只有在充分把握其思想精髓的前提下大胆联想，多做多练才能达到得心应手，灵活运用的境界。

10.11 高精度计算

C/C++中的 int 类型能表示的范围是 $-2^{31}-2^{31}-1$ 。unsigned 类型能表示的范围是 $0-2^{32}-1$ ，即 $0-4294967295$ 。所以，int 和 unsigned 类型变量，都不能保存超过 10 位的整数。有时我们需要参与运算的数，可能会远远不止 10 位，例如，可能需要保留小数点后面 100 位（比如求 π 的值），那么，即便使用能表示的很大数值范围的 double 变量，但是由于 double 变量只有 64 位，所以还是不可能达到精确到小数点后面 100 位这样的精度。double 变量的精度也不足以表示一个 100 位的整数。一般我们称这种基本数据类型无法表示的整数为大整数。如何表示和存放大整数呢？基本的思想就是：用数组存放和表示大整数。一个数组元素，存放大整数中的一位。那么，如何解决类似大整数这样的高精度计算问题呢？比如，一个最简单的例子，给定两个不超过 200 位的整数，求他们的和。

[例题]大整数加法

问题描述

求两个不超过 200 位的非负整数的和。

输入数据

有两行，每行是一个不超过 200 位的非负整数，没有多余的前导 0。

输出要求

一行，即相加后的结果。结果里不能有多余的前导 0，即如果结果是 342，那么就不能输出为 0342。

输入样例

22222222222222222222

33333333333333333333

输出样例

Output Sample:

55555555555555555555

解题思路

首先要解决的就是存储 200 位整数的问题。显然，任何 C/C++固有类型的变量都无法保存它。最直观的想法是可以用一个字符串来保存它。字符串本质上就是一个字符数组，因此为了编程更方便，我们也可以用数组 `unsigned an[200]`来保存一个 200 位的整数，让 `an[0]`存放个位数，`an[1]`存放十位数，`an[2]`存放百位数……那么如何实现两个大整数相加呢？方法很简单，就是模拟小学生列竖式做加法，从个位开始逐位相加，超过或达到 10 则进位。也就是说，用 `unsigned an1[201]`保存第一个数，用 `unsigned an2[200]`表示第二个数，然后逐位相加，相加的结果直接存放在 `an1` 中。要注意处理进位。另外，`an1` 数组长度定为 201，是因为两个 200 位整数相加，结果可能会有 201 位。实际编程时，不一定要费心思去把数组大小定得正好合适，稍微开大点也无所谓，以免不小心没有算准这个“正好合适”的数值，而导致数组小了，产生越界错误。

参考程序：

```
1. #include <stdio.h>
2. #include <string.h>
3. #define MAX_LEN 200
4. int an1[MAX_LEN+10];
5. int an2[MAX_LEN+10];
6. char szLine1[MAX_LEN+10];
7. char szLine2[MAX_LEN+10];
8. int main()
9. {
10.     scanf("%s", szLine1);
11.     scanf("%s", szLine2);
12.     int i, j;
13.
14.     //库函数 memset 将地址 an1 开始的 sizeof(an1)字节内容置成 0
15.     //sizeof(an1)的值就是 an1 的长度
16.     //memset 函数在 string.h 中声明
17.     memset( an1, 0, sizeof(an1));
18.     memset( an2, 0, sizeof(an2));
19.
20.     //下面将 szLine1 中存储的字符串形式的整数转换到 an1 中去，
21.     //an1[0]对应于个位
22.     int nLen1 = strlen( szLine1);
23.     j = 0;
24.     for( i = nLen1 - 1; i >= 0 ; i --)
25.         an1[j++] = szLine1[i] - '0';
26.
27.     int nLen2 = strlen(szLine2);
28.     j = 0;
29.     for( i = nLen2 - 1; i >= 0 ; i --)
30.         an2[j++] = szLine2[i] - '0';
31.
```



```

32.   for( i = 0; i < MAX_LEN ; i ++ ) {
33.       an1[i] += an2[i]; //逐位相加
34.       if( an1[i] >= 10 ) { //看是否要进位
35.           an1[i] -= 10;
36.           an1[i+1] ++; //进位
37.       }
38.   }
39.   bool bStartOutput = false; //此变量用于跳过多余的 0
40.   for( i = MAX_LEN; i >= 0; i -- ) {
41.       if( bStartOutput)
42.           printf("%d", an1[i]); //如果多余的 0 已经都跳过，则输出
43.       else if( an1[i] ) {
44.           printf("%d", an1[i]);
45.           bStartOutput = true; //碰到第一个非 0 的值，就说明多余的 0 已经都跳过
46.       }
47.
48.   //-----
49.   return 0;
50. }

```

实现技巧

1. 再次强调：实际编程时，不一定要费心思去把数组大小定得正好合适，稍微开大点也无所谓，以免不小心没有算准这个“正好合适”的数值，而导致数组小了，产生越界错误。

2. 语句 25 是把一个字符形式的数字转换成 unsigned 型的数。比如，要把字符'8'转换成 unsigned 型数 8。char 类型的变量，本质上实际是 int 类型，值就是字符的 Ascii 码。由于字符'0'到字符'9'的 Ascii 码是连续递增的，因此 '8' - '0' 的值就是 8。

思考题 10.1：上面的程序，有一点点瑕疵。只在某一种情况下，不能输出正确的结果。指出这种情况并修正程序。提示：要改正此程序，程序中语句 47 前面的部分可以不做任何增删和修改，只需在其后添加一些代码即可。此思考题旨在培养阅读他人程序并发现 bug 的能力。

[例题]：大整数乘法

问题描述

求两个不超过 200 位的非负整数的积。

输入数据

有两行，每行是一个不超过 200 位的非负整数，没有多余的前导 0。

输出要求

一行，即相乘后的结果。结果里不能有多余的前导 0，即如果结果是 342，那么就不能输出为 0342。

输入样例

12345678900

98765432100

输出样例

1219326311126352690000

解题思路

在下面的例子程序中，用 unsigned an1[200]和 unsigned an2[200]分别存放两个乘数，用 aResult[400]来存放积。计算的中间结果也都存在 aResult 中。aResult 长度取 400 是因为两个 200 位的数相乘，积最多会有 400 位。an1[0]，an2[0]，aResult[0]都表示个位。计算的过程基本上和小学生列竖式做乘法相同。为编程方便，并不急于处理进位，而将进位问题留待最后统一处理。现以 835×49 为例来说明程序的计算过程。先算 835×9 。 5×9 得到 45 个 1， 3×9 得到 27 个 10， 8×9 得到 72 个 100。由于不急于处理进位，所以 835×9 算完后，aResult 如下：

下标	5	4	3	2	1	0	
aResult	...	0	0	0	72	27	45

图 7-2-1

接下来算 4×5 。此处 4×5 的结果代表 20 个 10，因此要 aResult[1]+=20，变为：

下标	5	4	3	2	1	0	
aResult		0	0	72	47	45

图 7-2-2

再下来算 4×3 。此处 4×3 的结果代表 12 个 100，因此要 aResult[2]+= 12，变为：

下标	5	4	3	2	1	0	
aResult	...	0	0	0	84	47	45

图 7-2-3

最后算 4×8 。此处 4×8 的结果代表 32 个 1000，因此要 aResult[3]+= 32，变为：

下标	5	4	3	2	1	0	
aResult	...	0	0	32	84	47	45

图 7-2-4

乘法过程完毕。接下来从 aResult[0]开始向高位逐位处理进位问题。aResult[0]留下 5，把 4 加到 aResult[1]上，aResult[1]变为 51 后，应留下 1，把 5 加到 aResult[2]上……最终使得 aResult 里的每个元素都是 1 位数，结果就算出来了：

下标	5	4	3	2	1	0	
aResult	...	0	4	0	9	1	5

图 7-2-5

总结一个规律，即一个数的第 i 位和另一个数的第 j 位相乘所得的数，一定是要累加到结果的第 i+j 位上。这里 i, j 都是从右往左，从 0 开始数。

参考程序：

```
1. #include <stdio.h>
2. #include <string.h>
3. #define MAX_LEN 200
4. unsigned an1[MAX_LEN+10];
5. unsigned an2[MAX_LEN+10];
6. unsigned aResult[MAX_LEN * 2 + 10];
```

```

7.  char szLine1[MAX_LEN+10];
8.  char szLine2[MAX_LEN+10];
9.  int main()
10. {
11.     gets( szLine1); //gets 函数读取一行
12.     gets( szLine2);
13.     int i, j;
14.     int nLen1 = strlen( szLine1);
15.     memset( an1, 0, sizeof(an1));
16.     memset( an2, 0, sizeof(an2));
17.     memset( aResult, 0, sizeof(aResult));
18.     j = 0;
19.     for( i = nLen1 - 1; i >= 0 ; i --)
20.         an1[j++] = szLine1[i] - '0';
21.     int nLen2 = strlen(szLine2);
22.     j = 0;
23.     for( i = nLen2 - 1; i >= 0 ; i --)
24.         an2[j++] = szLine2[i] - '0';
25.
26.     for( i = 0; i < nLen2; i ++ )    { //每一轮都用 an1 的一位，去和 an2 各位相乘
27.         //从 an1 的个位开始
28.         for( j = 0; j < nLen1; j ++ )    //用选定的 an1 的那一位，去乘 an2 的各位
29.             aResult[i+j] += an2[i]*an1[j]; //两数第 i, j 位相乘，累加到结果的第 i+j 位
30.     }
31.     //下面的循环统一处理进位问题
32.     for( i = 0; i < MAX_LEN * 2; i ++ )    {
33.         if( aResult[i] >= 10 ) {
34.             aResult[i+1] += aResult[i] / 10;
35.             aResult[i] %= 10;
36.         }
37.     }
38.     //下面输出结果
39.     bool bStartOutput = false;
40.     for( i = MAX_LEN * 2; i >= 0; i -- )
41.         if( bStartOutput)
42.             printf("%d", aResult[i]);
43.         else if( aResult[i] ) {
44.             printf("%d", aResult[i]);
45.             bStartOutput = true;
46.         }
47.     if(! bStartOutput )
48.         printf("0");
49.     return 0;
50. }

```

不一定一出现进位就马上处理，而是等全部结果算完后再统一处理进位，有时会方便些。

「例题」：大整数除法

求两个大的正整数相除的商

第 1 行是测试数据的组数 n，每组测试数据占 2 行，第 1 行是被除数，第 2 行是除数。每组测试数据之间有一个空行，每行数据不超过 100 个字符 输出要求 n 行，每组测试数据有一行输出是相应的整数商

3

2405337312963373359009260457742057439230496493930355595797660791082739646
2987192585318701752584429931160870372907079248971095012509790550883793197894

[illegible]

5409656775097850895687056798068970934546546575676768678435435345

1

0

10000000000000000000000000000000
5409656775097850895687056798068970934546546575676768678435435345

基本的思想是反复做减法，看看从被除数里最多能减去多少个除数，商就是多少。一个一个减显然太慢，如何减得更快一些呢？以 7546 除以 23 为例来看一下：开始商为 0。先减去 23 的 100 倍，就是 2300，发现够减 3 次，余下 646。于是商的值就增加 300。然后用 646 减去 230，发现够减 2 次，余下 186，于是商的值增加 20。最后用 186 减去 23，够减 8 次，因此最终商就是 328。所以本题的核心是要写一个大整数的减法函数，然后反复调用该函数进行减法操作。计算除数的 10 倍、100 倍的时候，不用做乘法，直接在除数后面补 0 即可。

```

1. #include <stdio.h>
2. #include <string.h>
3. #define MAX_LEN    200
4. char szLine1[MAX_LEN + 10];
5. char szLine2[MAX_LEN + 10];
6. int an1[MAX_LEN + 10];          //被除数,  an1[0]对应于个位
7. int an2[MAX_LEN + 10];          //除数,   an2[0]对应于个位
8. int aResult[MAX_LEN + 10];      //存放商, aResult[0]对应于个位
9. /* Substract 函数: 长度为 nLen1 的大整数 p1 减去长度为 nLen2 的大整数 p2

```

```

10. 减的结果放在 p1 里，返回值代表结果的长度
11. 如不够减返回-1，正好减完返回 0
12. p1[0]、p2[0] 是个位 */
13. int Subtract( int * p1, int * p2, int nLen1, int nLen2)
14. {
15.     int i;
16.     if( nLen1 < nLen2 )
17.         return -1;
18.     //下面判断 p1 是否比 p2 大，如果不是，返回-1
19.     bool bLarger = false;
20.     if( nLen1 == nLen2 ) {
21.         for( i = nLen1-1; i >= 0; i -- ) {
22.             if( p1[i] > p2[i] )
23.                 bLarger = true;
24.             else if( p1[i] < p2[i] ) {
25.                 if ( ! bLarger )
26.                     return -1;
27.             }
28.         }
29.     }
30.     for( i = 0; i < nLen1; i ++ ) { //做减法
31.         p1[i] -= p2[i]; //要求调用本函数时给的参数能确保当 i>=nLen2 时， p2[i] = 0
32.         if( p1[i] < 0 ) {
33.             p1[i] += 10;
34.             p1[i+1] --;
35.         }
36.     }
37.     for( i = nLen1 -1 ; i >= 0 ; i-- )
38.         if( p1[i] )
39.             return i + 1;
40.     return 0;
41. }
42. int main()
43. {
44.     int t, n;
45.     char szBlank[20];
46.     scanf("%d", &n);
47.     for( t = 0; t < n; t ++ ) {
48.         scanf("%s", szLine1);
49.         scanf("%s", szLine2);
50.         int i, j;
51.         int nLen1 = strlen( szLine1);
52.         memset( an1, 0, sizeof(an1));
53.         memset( an2, 0, sizeof(an2));

```

```

54.  memset( aResult, 0, sizeof(aResult));
55.  j = 0;
56.  for( i = nLen1 - 1; i >= 0 ; i --)
57.      an1[j++] = szLine1[i] - '0';
58.  int nLen2 = strlen(szLine2);
59.  j = 0;
60.  for( i = nLen2 - 1; i >= 0 ; i --)
61.      an2[j++] = szLine2[i] - '0';
62.  if( nLen1 < nLen2 ) {
63.  printf("0\n");
64.  continue;
65.  }
66.  nLen1 = Substract( an1, an2, nLen1, nLen2) ;
67.  if( nLen1 < 0 ) {
68.  printf("0\n");
69.  continue;
70.  }
71.  else if( nLen1 == 0 ) {
72.  printf("1\n");
73.  continue;
74.  }
75.  aResult[0] ++; //减掉一次了，商加 1
76.  //减去一次后的结果长度是 nLen1
77.  int nTimes = nLen1 - nLen2;
78.  if( nTimes < 0 ) //减一次后就不能再减了
79.  goto OutputResult;
80.  else if( nTimes > 0 ) {
81.  //将 an2 乘以 10 的某次幂，使得结果长度和 an1 相同
82.  for( i = nLen1 - 1; i >= 0; i -- ) {
83.  if( i >= nTimes )
84.  an2[i] = an2[i-nTimes];
85.  else
86.  an2[i] = 0;
87.  }
88.  }
89.  nLen2 = nLen1;
90.  for( j = 0 ; j <= nTimes; j ++ ) {
91.  int nTmp;
92.  //一直减到不够减为止
93.  //先减去若干个 an2×(10 的 nTimes 次方)，
94.  //不够减了，再减去若干个 an2×(10 的 nTimes-1 次方)，.....
95.  while( (nTmp = Substract(an1, an2+j, nLen1, nLen2-j)) >= 0 ) {
96.  nLen1 = nTmp;
97.  aResult[nTimes-j]++; //每成功减一次，则将商的相应位加 1

```

```

98.     }
99.     }
100.  OutputResult:
101.  //下面的循环统一处理进位问题
102.      for( i = 0; i < MAX_LEN; i ++ ) {
103.          if( aResult[i] >= 10 ) {
104.              aResult[i+1] += aResult[i] / 10;
105.              aResult[i] %= 10;
106.          }
107.      }
108.  //下面输出结果
109.  bool bStartOutput = false;
110.      for( i = MAX_LEN ; i >= 0; i -- )
111.          if( bStartOutput)
112.              printf("%d", aResult[i]);
113.          else if( aResult[i] ) {
114.              printf("%d", aResult[i]);
115.              bStartOutput = true;
116.          }
117.      if(! bStartOutput )
118.          printf("0\n");
119.      printf("\n");
120.  }
121.  return 0;
122.  }

```

常见问题

问题一、忘了针对每一组测试数据，都要先将 an1, an2 和 aResult 初始化成全 0，而是一共只初始化了一次。这导致从第二组测试数据开始就都不对了。

问题二、减法处理借位的时候，容易忽略连续借位的情况，比如 10000 - 87，借位会一直进行到 1。

[例题]：麦森数

问题描述

形如 2^p-1 的素数称为麦森数，这时 P 一定也是个素数。但反过来不一定，即如果 P 是个素数。 2^p-1 不一定也是素数。到 1998 年底，人们已找到了 37 个麦森数。最大的一个是 $P=3021377$ ，它有 909526 位。麦森数有许多重要应用，它与完全数密切相关。

你的任务：输入 P ($1000 < P < 3100000$)，计算 2^p-1 的位数和最后 500 位数字（用十进制高精度数表示）

输入数据

只包含一个整数 P ($1000 < P < 3100000$)

输出要求

第 1 行：十进制高精度数 2^p-1 的位数。 第 2-11 行：十进制高精度数 2^p-1 的最后 500 位数字。（每行输出 50 位，共输出 10 行，不足 500 位时高位补 0） 不必验证 2^p-1 与 P 是否为素数。

输入样例

输出样例

00
00
00000000000000104079321946643990819252403273640855
38615262247266704805319112350403608059673360298012
23944173232418484242161395428100779138356624832346
49081399066056773207629241295093892203457731833496
61583550472959420547689811211693677147548478866962
50138443826029173234888531116082853841658502825560
46662248318909188018470682222031405210266984354887
32958028878050869736186900714720710555703168729087

第一个问题，输出 2^p-1 有多少位。由于 2^p-1 的各位数只可能是 2, 4, 6, 8 所以 2^p-1 和 2^p 的位数相同。使用 C/C++ 标准库中在 `math.h` 里声明的，求以 10 为底的对数的函数 `double log10(double x)` 函数，就能轻松求得 2^p-1 的位数。 2^p 的值需要用一种高效率的办法来算。显然，对于任何 $p>0$ ，考虑 p 的二进制形式，则不难得到：

因而：

计算 2^p 的办法就是，先将结果的值设为 1，计算 2^1 。如果 a_0 值为 1，则结果乘以 2^1 ；计算 2^2 ，如果 a^1 为 1，则结果乘以 2^2 ；计算 2^4 ，如果 a^2 为 1，则结果乘以 2^4 ；……总之，第 i 步(i 从 0 到 n , a_n 是 1)就计算 2^{2^i} ，如果 a_i 为 1，则结果就乘以 2^{2^i} 。每次由 $2^{2^i} \times$ 就能算出 $2^{2^{i+1}}$ 。由于 p 可能很大，所以上面的乘法都应该使用高精度计算。由于题目只要求输出 500 位，所以这些乘法都是只须算出末尾的 500 即可。在上面的高精度计算中，我们用数组来存放大整数，数组的一个元素对应于十进制大整数的一位。本题如果也这样做，就会超时。为了加快计算速度，可以用一个数组元素对应于大整数的 4 位，即将大整数表示为 10000 进制，而数组中的每一个元素就存放 10000 进制数的 1 位。例如，用 `int` 型数组 `a` 来存放整数 6373384，那么只需两个数组元素就可以了，`a[0]=3384`，`a[1]=637`。由于只要求结果的最后 500 位数字，所以我们不需要计算完整的结果，只需算出最后 500 位即可。因为用每个数组元素存放十进制大整数的 4 位，所以本题中的数组最多只需要 125 个元素。

1. `#include <stdio.h>`
2. `#include <memory.h>`
3. `#define LEN 125` //每数组元素存放十进制的 4 位，因此数组最多只要 125 个元素即可。
4. `#include <math.h>`
5. `/* Multiply 函数功能是计算高精度乘法 a * b`


```

6. 结果的末 500 位放在 a 中
7. */
8. void Multiply(int* a, int* b)
9. {
10.     int i, j;
11.     int nCarry; //存放进位
12.     int nTmp;
13.     int c[LEN]; //存放结果的末 500 位
14.     memset(c, 0, sizeof(int) * LEN);
15.     for (i=0;i<LEN;i++) {
16.         nCarry=0;
17.         for (j=0;j<LEN-i;j++) {
18.             nTmp=c[i+j]+a[j]*b[i]+nCarry;
19.             c[i+j]=nTmp%10000;
20.             nCarry=nTmp/10000;
21.         }
22.     }
23.     memcpy( a, c, LEN*sizeof(int));
24. }
25. int main()
26. {
27.     int i;
28.     int p;
29.     int anPow[LEN]; //存放不断增长的 2 的次幂
30.     int aResult[LEN]; //存放最终结果的末 500 位
31.     scanf("%d", & p);
32.     printf("%d\n", (int)(p*log10(2))+1);
33.     //下面将 2 的次幂初始化为  $2^0$  ( $a^b$  表示 a 的 b 次方),
34.     // 最终结果初始化为 1
35.     anPow[0]=2;
36.     aResult[0]=1;
37.     for (i=1;i<LEN;i++) {
38.         anPow[i]=0;
39.         aResult[i]=0;
40.     }
41.
42.     //下面计算 2 的 p 次方
43.     while (p>0) { // p = 0 则说明 p 中的有效位都用过了, 不需再算下去
44.         if ( p & 1 ) //判断此时 p 中最低位是否为 1
45.             Multiply(aResult, anPow);
46.         p>>=1;
47.         Multiply(anPow, anPow);
48.     }
49.

```

```

50.  aResult[0]--; //2 的 p 次方算出后减 1
51.
52.  //输出结果
53.  for (i=LEN-1;i>=0;i--) {
54.      if (i%25==12)
55.          printf("%02d\n%02d", aResult[i]/100,
56.              aResult[i]%100);
57.      else {
58.          printf("%04d", aResult[i]);
59.          if (i%25==0)
60.              printf("\n");
61.      }
62.  }
63.  return 0;
64.  }

```

语句 17: j 只要算到 $LEN - i - 1$, 是因为 $b[i] \times a[j]$ 的结果总是加到 $c[i+j]$ 上, $i+j$ 大于等于 LEN 时, $c[i+j]$ 是不需要的, 也不能要, 否则 c 数组就越界了。

语句 18: $b[i] \times a[j]$ 的结果总是要加到 $c[i+j]$ 上, 此外还要再加上上次更新 $c[i+j-1]$ 时产生的进位。

语句 19: 由于 c 中的每一元素代表 10000 进制数的 1 位, 所以 $c[i+j]$ 的值不能超过 10000。

语句 20: 算出进位。

语句 43 到语句 48, 每次执行循环都判断 a_i (i 从 0 开始) 的值是否为 1, 如果是, 则将最终结果乘以 a_i 。接下来再由 a_i 算出 a_i 。

语句 54: 输出从万进制数的第 124 位开始, 万进制数的每一位输出为十进制数的 4 位, 每行只能输出 50 个十进制位, 所以发现当 $i\%25$ 等于 12 时, 第 i 个万进制位会被折行输出, 其对应的后两个十进制位会跑到下一行。

语句 55: “%02d” 表示输出一个整数, 当输出位数不足 2 位的时候, 用前导 0 补足到 2 位。本行将一个万进制位分两半折行输出。

语句 58: 将一个万进制位以十进制形式输出, 用前导 0 确保输出宽度是 4 个字符。

语句 59: 满足条件的话就该换行了。

常见问题

问题一: 没有想到用数学公式和库函数可以直接计算结果位数, 而是用其他办法大费周折。

问题二: 试图用最简单的办法, 做 p 次乘以 2 的操作, 结果严重超时。

问题三: 没有对数据规模有足够估计, 用数组表示十进制大整数而非万进制数, 结果超时。

思考题 10.3: 本题在数组中存储万进制大整数以加快速度。如果存储的是十万进制数, 岂不更快? 而且输出时十万进制数的一位正好对应于十进制的 5 位, 计算折行也会方便很多。这种想法成立吗? 这么写会真的会更方便吗?

练习题

1. 计算 2 的 N 次方

任意给定一个正整数 N ($N \leq 100$), 计算 2 的 N 次方的值。

2. 浮点数加法

求 2 个不超过 100 位的浮点数相加的和

3. 孙子问题浮点数加法

对于给定的正整数 a_1, a_2, \dots, a_n , 问是否存在正整数 b_1, b_2, \dots, b_n , 使得对于任意的一个正整数 N , 如果用 N 除以 a_1 的余数是 p_1 , 用 N 除以 a_2 的余数是 p_2 ……用 N 除以 a_n 的余数是 p_n , 那么 $M = p_1 * b_1 + p_2 * b_2 + \dots + p_n * b_n$ 能满足 M 除以 a_1 的余数也是 p_1 , M 除以 a_2 的余数也是 p_2 …… M 除以 a_n 的余数也是 p_n 。如果存在, 则输出 b_1, b_2, \dots, b_n 。题中 $1 \leq n \leq 10$, a_1, a_2, \dots, a_n 均不大于 50。

4. 浮点数求高精度幂

有一个实数 R ($0.0 < R < 99.999$), 要求写程序精确计算 R 的 n 次方。 n 是整数并且 $0 < n \leq 25$ 。

附 录

ASCII 表

八进制	十六进制	十进制	字符	八进制	十六进制	十进制	字符
00	00	0	nul	100	40	64	@
01	01	1	soh	101	41	65	A
02	02	2	stx	102	42	66	B
03	03	3	etx	103	43	67	C
04	04	4	eot	104	44	68	D
05	05	5	enq	105	45	69	E
06	06	6	ack	106	46	70	F
07	07	7	bel	107	47	71	G
10	08	8	bs	110	48	72	H
11	09	9	ht	111	49	73	I
12	0a	10	nl	112	4a	74	J
13	0b	11	vt	113	4b	75	K
14	0c	12	ff	114	4c	76	L
15	0d	13	er	115	4d	77	M
16	0e	14	so	116	4e	78	N
17	0f	15	si	117	4f	79	O
20	10	16	dle	120	50	80	P
21	11	17	dc1	121	51	81	Q
22	12	18	dc2	122	52	82	R
23	13	19	dc3	123	53	83	S
24	14	20	dc4	124	54	84	T
25	15	21	nak	125	55	85	U
26	16	22	syn	126	56	86	V
27	17	23	etb	127	57	87	W
30	18	24	can	130	58	88	X
31	19	25	em	131	59	89	Y
32	1a	26	sub	132	5a	90	Z
33	1b	27	esc	133	5b	91	[
34	1c	28	fs	134	5c	92	\
35	1d	29	gs	135	5d	93]
36	1e	30	re	136	5e	94	^
37	1f	31	us	137	5f	95	_
40	20	32	sp	140	60	96	'
41	21	33	!	141	61	97	a

42	22	34	"	142	62	98	b
43	23	35	#	143	63	99	c
44	24	36	\$	144	64	100	d
45	25	37	%	145	65	101	e
46	26	38	&	146	66	102	f
47	27	39	`	147	67	103	g
50	28	40	(150	68	104	h
51	29	41)	151	69	105	i
52	2a	42	*	152	6a	106	j
53	2b	43	+	153	6b	107	k
54	2c	44	,	154	6c	108	l
55	2d	45	-	155	6d	109	m
56	2e	46	.	156	6e	110	n
57	2f	47	/	157	6f	111	o
60	30	48	0	160	70	112	p
61	31	49	1	161	71	113	q
62	32	50	2	162	72	114	r
63	33	51	3	163	73	115	s
64	34	52	4	164	74	116	t
65	35	53	5	165	75	117	u
66	36	54	6	166	76	118	v
67	37	55	7	167	77	119	w
70	38	56	8	170	78	120	x
71	39	57	9	171	79	121	y
72	3a	58	:	172	7a	122	z
73	3b	59	;	173	7b	123	{
74	3c	60	<	174	7c	124	
75	3d	61	=	175	7d	125	}
76	3e	62	>	176	7e	126	~
77	3f	63	?	177	7f	127	del