

C 语言程序设计教程

——基于 Turbo C

李 莉 陈 哲 谢金达 主 编

严仲兴 张 捷 陈功文 副主编

科 学 出 版 社

内 容 提 要

本书循序渐进地讲解了 C 语言的基本语法和程序设计的基本方法。

全书共 13 章, 分别介绍了 C 语言的集成环境、数据类型、运算符及表达式、输入输出语句、选择结构和循环结构、函数、数组、指针、编译预处理、结构体与共用体、枚举与位运算、文件等内容。每章除讲解知识点外, 还穿插了较多的小程序, 同时提供了要点回顾和习题。为加强学生的理解, 最后给出一个综合程序设计实例, 并在附录中给出了 10 个上机实验指导。

本书可作为大中专、高职高专学校的教材, 也可以作为计算机等级考试的参考资料和自学用书。

图书在版编目 (CIP) 数据

C 语言程序设计教程——基于 Turbo C / 李莉, 陈哲, 谢金达编著.

—北京: 科学出版社, 2007

ISBN 978-7-03-018537-2

I. C… II. ①李… ②陈… ③谢… III. C 语言—程序设计—教材
IV. TP312

中国版本图书馆 CIP 数据核字 (2007) 第 017249 号

责任编辑: 刘秀青 张 楠 / 责任校对: 科 海

责任印刷: 科 海 / 封面设计: 林 陶

科学出版社 出版

北京东黄城根北街 16 号

邮政编码: 100717

<http://www.sciencep.com>

北京科普瑞印刷有限责任公司印刷

科学出版社发行 各地新华书店经销

*

2007 年 3 月第一版

开本: 16 开

2007 年 3 月第一次印刷

印张: 19.5

印数: 1-4000

字数: 475 千字

定价: 29.00 元

(如有印装质量问题, 我社负责调换)

本书编委会

主 编：李 莉 陈 哲 谢金达

副主编：严仲兴 张 捷 陈功文

编 委：安丰彩 石文华 高力勋

周爱霞 张瑞坤 宋玉璞

徐玉莲 彭 斌 刘冰洁

王 倩 张媛媛 冯焕婷

刘 燕 毕春华 李寒松

常明迪 李 艳

前 言

C 语言功能丰富、使用灵活、目标程序效率高、可移植性好，可用于开发系统软件和应用软件，是近年来被广泛应用的一种计算机语言。许多高校都开设了 C 语言课程，全国计算机等级考试也将其列为二级考试课程。作者在多年讲授 C 语言课程的基础上，结合全国计算机等级考试大纲，编写了本书，目的是使读者在学习程序设计语言的同时，培养自己的优良编程风格，掌握基本的编程方法和典型算法。

C 语言的数据类型多，运算规则复杂，语句形式灵活，初学者不易掌握。在本书编写过程中，力求做到语言简明、概念清晰、通俗易懂，同时将重点放在基本概念、基本方法上。讲解时，注意循序渐进，难点分散，对用到的基础知识都进行了简单介绍，对编程容易出错的地方给出提示，使读者学习时少走弯路。读者可以在没有太多计算机基础知识的情况下使用本书。

本书参照 ANSI 标准 C 编写，以 Turbo C 2.0 为程序运行环境，主要介绍 C 的数据类型、C 的结构化程序设计方法、函数间数据传递、C 的数组和指针的概念及应用、常用库函数及内存动态分配、结构体与共用体、位运算、文件的用法等。在内容安排上，注重知识的系统性和实用性，根据各章节的知识点，结合实用案例逐步给出新的编程方法并与原编程方法进行比较，并通过一些小型实例介绍典型算法，还给出每章的学习要点回顾和习题。为加强学生的理解，最后给出一个综合程序设计实例，并附有 10 个上机实验指导（根据课时安排，有些章节和例题可以留给学生自学）。

在本书出版过程中，得到科学出版社有关领导的大力支持和成洁编辑的热情帮助。在此表示衷心的感谢。

书中难免存在疏忽错误之处，诚挚地希望读者批评指正。

编 者

2007 年 1 月

目 录

第 1 章 C 语言基础知识	1
1.1 C 语言的发展及特点	1
1.2 C 程序的构成	2
1.3 C 语言风格和源程序书写格式	3
1.4 C 程序的编译和执行	4
1.5 TC 集成环境简介	6
1.5.1 Turbo C 2.0 的启动	6
1.5.2 Turbo C 2.0 的主屏幕	6
1.5.3 Turbo C 2.0 的子菜单	8
1.5.4 源程序的建立和编辑	10
1.5.5 源程序的编译、连接和运行举例	12
要点回顾	15
习题	15
第 2 章 数据类型、运算符及表达式	16
2.1 数制基础及计算机中数的表示	16
2.1.1 数的二进制、八进制和十六进制表示法	16
2.1.2 数在机器内部的表示方法	18
2.2 C 语言的数据类型及其取值范围	18
2.2.1 基本数据类型	18
2.2.2 基本类型数据的存储空间长度及数据取值范围	19
2.3 各种类型常量及其表示	20
2.3.1 整型常量	20
2.3.2 实型常量	20
2.3.3 字符常量	21
2.3.4 字符串常量	23
2.3.5 符号常量	23
2.4 变量及其类型定义	24
2.4.1 变量名	24
2.4.2 变量的数据类型	25
2.4.3 变量的定义	25
2.4.4 变量的初始化	25
2.5 C 语言运算符的分类、运算优先级和结合性	26

2.6 算术运算符和算术表达式	29
2.6.1 二元算术运算符	29
2.6.2 一元算术运算符	29
2.6.3 算术表达式	30
2.7 赋值运算符和赋值表达式	30
2.7.1 赋值运算符	30
2.7.2 赋值表达式	32
2.8 逗号运算符和逗号表达式	32
2.8.1 逗号运算符	32
2.8.2 逗号表达式	33
2.9 关系运算符和关系表达式	34
2.9.1 关系运算符	34
2.9.2 关系表达式	34
2.10 逻辑运算符和逻辑表达式	35
2.10.1 逻辑运算符	35
2.10.2 逻辑表达式	35
2.11 测试数据长度运算符 sizeof	37
2.12 不同类型数据间的转换与运算	37
2.12.1 算术表达式运算时的数据类型隐式转换规则	38
2.12.2 显式类型转换	38
2.12.3 赋值表达式的类型及赋值时的数据类型转换	39
要点回顾	41
习题	42

第3章 C语言的基本语句与输入输出

3.1 基本语句	44
3.1.1 表达式语句	44
3.1.2 空语句	45
3.1.3 复合语句	45
3.2 流程控制语句	46
3.3 数据的输入与输出	46
3.3.1 格式输出函数 printf()	47
3.3.2 格式输入函数 scanf()	51
3.3.3 字符输出输入函数 putchar()、getchar()	54
3.4 简单程序举例	56
要点回顾	58
习题	58

第 4 章 选择结构程序设计	60
4.1 计算机算法及其描述.....	60
4.2 结构化程序设计的概念	64
4.3 选择结构程序设计	64
4.3.1 if-else 分支语句.....	64
4.3.2 if-else 编程举例.....	67
4.4 选择结构的嵌套	69
4.4.1 if-else 语句的嵌套.....	69
4.4.2 if-else if 结构.....	72
4.5 用 switch 语句实现多分支选择结构.....	76
4.6 程序调试过程举例	78
要点回顾.....	80
习题	81
第 5 章 循环结构程序设计	83
5.1 for 语句.....	83
5.2 while 语句.....	86
5.3 do-while 语句	88
5.4 使用 goto 语句实现循环.....	91
5.5 多重循环	91
5.6 break 语句与 continue 语句.....	95
5.7 几种循环的关系与比较.....	98
5.8 应用程序举例	98
要点回顾.....	103
习题	103
第 6 章 函数.....	108
6.1 C 程序的模块结构.....	108
6.1.1 函数定义方法和函数的形参	110
6.1.2 函数声明与函数原型	111
6.2 函数调用时参数值的传递	113
6.2.1 函数的传值调用.....	113
6.2.2 函数的返回	115
6.3 局部变量和全局变量.....	117
6.3.1 局部变量	117
6.3.2 全局变量	118
6.4 变量的存储类别	120
6.5 函数的嵌套调用和递归调用	125

6.5.1 函数的嵌套调用	125
6.5.2 函数的递归调用	125
6.6 内部函数与外部函数	128
6.6.1 内部函数	128
6.6.2 外部函数	128
6.7 多文件编程举例	128
要点回顾	135
习题	136
第 7 章 数组	139
7.1 数组的定义和初始化	139
7.1.1 一维数组的定义	139
7.1.2 多维数组的定义	140
7.1.3 数组的存储结构	140
7.1.4 数组的初始化	141
7.2 数组元素的引用	142
7.3 数组的赋值	143
7.4 数值型数组的输入和输出	144
7.5 数组应用程序举例	145
7.6 字符串和字符型数组	153
7.6.1 字符串	153
7.6.2 字符型数组的定义和初始化	154
7.6.3 字符型数组的输入和输出	155
7.6.4 字符串处理函数	158
7.6.5 字符数组举例	159
要点回顾	160
习题	162
第 8 章 指针	166
8.1 地址以及和地址有关的运算	166
8.1.1 地址的概念	166
8.1.2 取地址运算符和访问地址运算符	167
8.2 指针的概念及指针变量的定义	169
8.2.1 指针变量的定义	170
8.2.2 将指针指向对象的方法、空指针和 void 型指针	170
8.2.3 指针的运算	172
8.3 通过指针引用变量、数组、字符串	173
8.3.1 用指针访问变量	173
8.3.2 用指针访问数组	173

8.3.3 用指针访问字符串	174
8.4 指针数组和二级指针	177
8.4.1 指针数组的概念	177
8.4.2 二级指针（指向指针的指针）	180
8.4.3 用二级指针访问数组或字符串	180
8.5 将指针作为函数参数	182
8.6 返回指针值的指针型函数	184
8.7 内存动态分配	185
8.7.1 内存动态分配的含义	185
8.7.2 内存动态分配函数	186
8.8 函数指针	188
8.9 main()函数的命令行参数	189
要点回顾	190
习题	192
第9章 编译预处理	198
9.1 宏定义	198
9.1.1 不带参数的宏定义	198
9.1.2 带参数的宏定义	201
9.1.3 宏定义的解除	204
9.2 文件包含	205
9.2.1 文件包含的格式	205
9.2.2 文件包含的功能	205
9.3 条件编译	207
要点回顾	209
习题	210
第10章 结构体与共用体	212
10.1 结构体类型	212
10.1.1 结构体类型的概念	212
10.1.2 结构体类型的定义	212
10.2 结构体变量	214
10.2.1 结构体变量的定义	214
10.2.2 结构体变量的初始化	216
10.2.3 结构体变量的成员引用	217
10.2.4 结构体变量的赋值和输入输出	218
10.3 结构数组	219
10.3.1 结构数组的定义	219
10.3.2 结构数组的初始化	219

10.3.3 结构数组的元素成员访问	220
10.4 结构指针	221
10.4.1 结构指针的定义	221
10.4.2 结构指针的初始化	221
10.4.3 结构指针的访问	221
10.5 递归结构和链表	223
10.5.1 递归结构的定义	223
10.5.2 递归结构的应用	224
10.6 共用体	227
10.6.1 共用体的定义	227
10.6.2 共用体变量的访问	228
10.6.3 共用体的存储	228
10.6.4 共用体的应用	229
10.7 类型定义	231
10.7.1 类型定义的形式	231
10.7.2 类型定义的使用	231
要点回顾	232
习题	234

第 11 章 枚举与位运算.....238

11.1 枚举	238
11.1.1 枚举类型的定义	238
11.1.2 枚举类型的应用	239
11.2 位运算	241
11.2.1 按位逻辑运算符	241
11.2.2 移位运算符	245
11.2.3 复合赋值运算符	246
11.2.4 不同长度的数据进行位运算	246
11.3 简单的位运算应用举例	246
11.4 位段	249
要点回顾	251
习题	252

第 12 章 文件操作.....254

12.1 文件概述	254
12.2 文件的打开与关闭	255
12.2.1 文件指针（FILE 类型指针）	255
12.2.2 文件的打开	256
12.2.3 文件的关闭	258

12.3 文件检测函数.....	259
12.3.1 检测文件末尾函数 feof().....	259
12.3.2 检测出错函数 ferror().....	259
12.4 文件读写函数.....	260
12.4.1 字符读写函数.....	260
12.4.2 字符串读写函数.....	261
12.4.3 文件格式读写函数.....	262
12.4.4 数据块读写函数.....	264
12.5 文件的定位与读写.....	266
12.5.1 文件的定位.....	266
12.5.2 文件的顺序读写和随机读写.....	267
要点回顾.....	270
习题.....	272
第 13 章 综合程序设计举例.....	275
附录 1 C 运算符的优先级与结合性.....	282
附录 2 ASCII 码表.....	283
附录 3 Turbo C 2.0 常用库函数及其标题文件.....	284
附录 4 Turbo C 2.0 编译错误提示和原因.....	288
附录 5 实验指导.....	291
实验 1 TC 集成环境的基本应用.....	291
实验 2 数据类型、运算符和表达式.....	292
实验 3 简单程序设计及基本调试方法.....	293
实验 4 分支结构程序设计.....	294
实验 5 循环程序设计.....	294
实验 6 函数应用.....	295
实验 7 数组应用.....	295
实验 8 指针应用.....	296
实验 9 结构体与共用体.....	296
实验 10 文件.....	297
参考文献.....	299

第 1 章 C 语言基础知识

本章的学习目标:

了解 C 语言的发展及特点,掌握 C 语言程序的构成,了解 C 语言风格和源程序书写格式,了解 C 语言程序的编译和执行步骤,熟悉 Turbo C 集成环境,熟练操作独立完成一个 C 程序的输入、编译、连接、运行的操作过程。

1.1 C 语言的发展及特点

C 语言是目前最为流行的计算机高级程序设计语言之一。它设计精巧,功能齐全,不仅是开发系统软件的理想工具,也是开发应用软件的理想程序设计语言,因此深受广大程序设计者的欢迎。

C 语言是在 1972 年由美国贝尔实验室的 D. M. Ritchie 为描述和实现 UNIX 操作系统而设计的,现在已经成为一个成熟的通用编程语言,并广泛应用于多种机型(如个人计算机、工作站和大型机)和操作系统(如 Windows、DOS 和 UNIX)。C 语言既可以处理数据库、网络、图形、图像等,又适合在工业控制、自动检测等方面应用,比如现在控制和检测领域的很多软件都是用 C 语言编写的。

C 语言的特点可大致归纳如下:

(1) C 语言短小精悍,基本组成部分紧凑、简洁。C 语言一般只用 32 个标准关键字、45 个标准运算符以及 9 种控制语句,不但语言组成精练、简洁,而且使用方便、灵活。

(2) C 语言运算符丰富,表达能力强。C 语言具有“高级语言”和“低级语言”双重特点,其运算符包含的内容广泛,所生成的表达式简练、灵活,有利于提高编译效率和目标代码的质量。

(3) C 语言数据结构丰富,程序结构化好。C 语言提供了编写结构化程序所需要的各种数据结构和控制结构,并且具有以函数调用为主的程序设计风格,所以利用 C 语言所编写的程序具有良好的结构。

(4) C 语言提供了某些接近汇编语言的功能,有利于编写系统软件。C 语言提供的一些运算和操作,能够实现汇编语言的功能,比如它可以直接访问物理地址,并能进行二进制位运算等,这为编写系统软件提供了方便。

(5) C 语言程序所生成的目标代码质量高。C 语言程序所生成目标代码的效率仅比用汇编语言描述同一个问题低 20% 左右,因此用 C 语言编写的程序执行效率高。

(6) C 语言程序的可移植性好。在 C 语言所提供的语句中,没有直接依赖于硬件的语句,与硬件有关的操作,如数据输入、输出等都是通过调用系统提供的库函数来实现的。因此,用 C 语言编写的程序能够很容易地从一种计算机环境移植到另一种计算机环境中。



当然，C 语言本身也有弱点：一是运算符的优先级较多，不容易记忆；二是由于 C 语言语法限制不太严格，在增强了程序设计灵活性的同时，一定程度上也降低了某些安全性，这对程序设计人员提出了更高的要求。

有些 C 语言的不足之处，在 C++ 里已经得到了改进。1983 年，贝尔实验室的 Bjarne Stroustrup 在 C 的基础上，推出了 C++。C++ 进一步扩充和完善了 C 语言，目前流行的版本有 Borland C++ 和 Microsoft Visual C++。C++ 语言支持面向对象技术，易于将问题空间直接映射到程序空间，为程序员提供了一种新的思维方式和编程方法。

C 是 C++ 的基础，C++ 语言和 C 语言在很多方面是兼容的；在 C++ 的环境中，许多 C 代码可以直接使用。因此，掌握了 C 语言后再学习 C++，就能以一种熟悉的语法来学习面向对象的语言，从而收到事半功倍的效果。另外，广泛应用于 Internet 的 Java 语言具有安全、跨平台、面向对象、简单、适用于网络等显著特点，它也采用了 C 语言中的大部分语法，所以学好 C 语言还有助于学习 Java 语言。

1.2 C 程序的构成

C 语言是函数式语言，一个 C 程序由一个或多个函数组成，其中必须有且只能有一个名为 `main` 的主函数。在程序中，主函数 `main()` 的位置没有特殊要求，但程序执行是从主函数开始，随主函数的结束而结束。主函数是整个程序的控制部分。在主函数执行过程中，可以调用 C 语言库函数或用户自定义函数。C 语言有丰富的库函数，可供用户直接调用。

例 1.1 C 程序基本结构举例。

```
#include <stdio.h>
main()
{
    printf("This is an example.");
}
```

本程序运行时输出一个字符串 “This is an example.”。

本程序有一个主函数 `main()`，花括号 `{}` 中的内容是 `main` 的函数体。一般函数体由若干条语句组成，包括说明语句和执行语句。说明语句用来定义或声明程序中出现的变量名称和类型，执行语句进行计算和输出。

程序中的第一条语句是文件包含语句，其作用是将标准输入输出头文件包含到当前文件中来。头文件又称为标题文件或包含文件，扩展名一般为 “.h”。头文件可以看作源文件之间的接口，程序设计时需要用文件包含命令 `include` 将头文件包含到程序文件中来。头文件内容一般为：类型声明、函数声明、常量定义、数据声明、枚举定义、包含指令、宏定义、注释等。

上例文件中要用到输出函数 `printf`，而该函数原型在标准输入输出头文件 `stdio.h` 中定义，所以程序开始要添加 `#include <stdio.h>` 语句，否则在编译时候会出现 `printf` 没有定义的错误信息。

程序的开始执行点是 `main()`。如果程序有多个函数，且在 `main` 函数的执行过程中调用其他函数，被调函数执行结束后会返回到主函数中对应的调用点；主函数执行到 `return` 语

句或右花括号“}”时，程序结束。

C 语言程序是用函数驱动的，函数要先定义或声明后才可使用。有关函数的更详细内容将在后续章节中学习，先通过下面例题了解 C 程序的一般组成。

例 1.2 两个函数组成的程序，功能是计算这两个整数的乘积。

```
#include <stdio.h>          /* 文件包含 */
main()                      /* 主函数名 */
{                            /* 函数体开始 */
    int product(int ,int );  /* 函数声明 */
    int x,y,z;              /* 局部变量类型定义 */
    printf("enter the value of x,y:"); /* 屏幕提示信息 */
    scanf("%d%d",&x,&y);    /* 输入变量值 */
    z=product(x,y);         /* 调用计算乘积的函数 */
    printf("x=%d,y=%d\n",x,y); /* 打印 x 和 y 的值 */
    printf("x*y=%d\n",z);   /* 打印乘积值 */
}                            /* 函数体结束 */
int product(int a,int b)    /* 被调函数及其形式参数 */
{
    int c;                  /* 定义局部变量 */
    c=a*b;                  /* 计算乘积 */
    return(c);              /* 返回值 */
}
```

这是一个简单的含多个函数的 C 程序，它含有一个主函数 `main()` 和一个子函数 `product()`。主函数负责输入两个变量的值，然后调用子函数，最后打印结果。子函数负责接收从主函数传送过来的两个数，并计算其乘积，最后将该乘积返回主函数。

程序运行时，屏幕首先显示提示信息：`enter the value of x,y:_`。

提示信息末尾的“_”是闪烁的光标，用户若从键盘输入 5 和 8（5 和 8 用空格隔开），按回车键后，程序输出为：

```
x=5,y=8
x*y=40
```

1.3 C 语言风格和源程序书写格式

C 程序风格灵活，书写格式自由，一般有以下特点：

（1）如果程序源文件由多个函数组成，每个函数在整个程序文件中的位置可以任意。但不管主函数位于程序中的何处，程序总是从主函数开始运行。

（2）标识符的命名应该“见名知意”，例如：`sum`，`average`，`student_no`；如果用多个单词表示一个标识符，中间可以用下划线分隔。标识符一般采用小写字母，大写字母常用作符号常数。

（3）每个语句以分号结束。既允许一行内写多条语句，也允许一条语句分多行书写。但为了看起来清楚，建议每行写一个语句。

（4）分级缩进格式。同一级别的语句采用同样的缩进格式，使程序看起来条理清晰，



有层次感，语句的执行顺序与书写时的缩进格式无关。

(5) 适当使用注释，帮助理解程序。注释写在“/* */”内，可以出现在程序中的任何位置，对源程序编译时将跳过注释语句，也就是注释语句对目标程序和可执行程序都没有任何影响。

例 1.3 程序书写格式举例。

下面程序可实现计算 $S=1+\frac{1}{2!}+\frac{1}{3!}+\dots+\frac{1}{100!}$ （暂时不讨论程序的设计思路，只是了解程序书写格式）。

```
#include <stdio.h>
main()
{
    double sum=0.0,t,s;           /*定义变量类型和名称 */
    int i,j;
    for(i=1;i<=100;i++)           /*外重循环开始 */
    {
        t=1;
        for(j=1;j<=i;j++)         /*内重循环，计算阶乘 */
            t=t*j;
        printf("%e\n",t);         /*输出 t 的值 */
        s=1/t;                    /*计算阶乘的倒数 */
        sum=sum+s;                /*求阶乘倒数的和 */
    }                             /*外重循环结束 */
    printf("\n%e\n",sum);         /*输出结果 */
}
```

该程序是一个双重循环，每重循环体内的语句均采用分级缩进格式书写。程序的缩进格式仅使程序看起来有层次感，对程序的逻辑结构和编译执行没有影响。

1.4 C 程序的编译和执行

C 语言是一种编译型的程序设计语言。一个 C 程序要经过编辑、编译、连接和运行 4 个步骤，才能得到运行结果。

1. 编辑

所谓编辑是指对 C 语言源程序进行输入和修改。编辑所使用的软件是各种编辑程序，如 DOS 系统提供的全屏幕编辑程序 edit，Windows 系统提供的记事本程序 notepad 等，都可以用来编辑 C 源程序。编辑结束后，将 C 源程序以文本文件的形式存放在磁盘上，文件名由用户自己选定，扩展名一般为“.c”，例如 f1.c，example.c 等。C 语言源程序是以 ASCII 代码形式输入和存储的，不能直接被计算机执行。

2. 编译

编译是把已编辑好的 C 源程序翻译成可重定位的二进制目标程序，编译过程由编译程

序完成。在编译时，编译程序自动对源程序进行句法和语法检查，当发现错误时，就将错误的类型和在程序中的位置显示出来，以帮助用户修改源程序中的错误。此时应重新进入编辑状态，对源程序进行修改后再重新编译，直到通过编译为止。编译完成后，编译程序自动生成二进制目标代码并对目标代码进行优化后生成目标文件。目标文件的名称由编译系统自动规定为和源文件同名，也可以由用户另外指定；目标文件的扩展名为“.obj”。

3. 连接

经编译后得到的二进制代码还不能直接执行，必须把经过编译的各个模块的目标代码与系统提供的标准模块（如 C 语言中的标准库函数）连接后才能运行。

连接也称链接或装配，是用连接程序将编译过的目标程序和程序中用到的库函数连接装配在一起，得到具有绝对地址的可执行文件，即计算机能直接执行的文件。此文件的扩展名由系统自动指定为“.exe”（例如 fl.exe）。

4. 运行

运行是将可执行文件投入运行，以获取程序的运行结果。运行可执行文件时，在 DOS 操作系统下是在提示符后直接键入可执行文件名，在 Windows 操作系统下是用鼠标双击可执行文件名。

C 程序的编译和执行过程如图 1-1 所示。

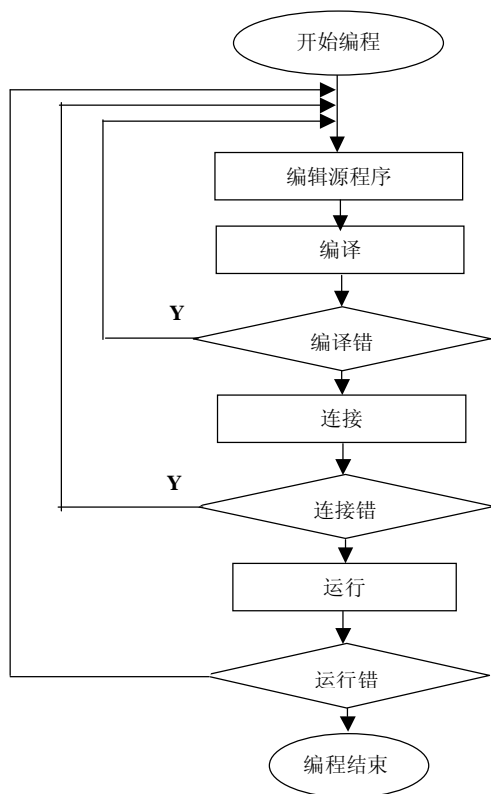


图 1-1 C 程序的编译和执行过程



1.5 TC 集成环境简介

常用的 C 编译程序有 Borland 公司的 TC 和微软公司的 QC，TC 的应用更为广泛。

TC 即 Turbo C，它提供一个集成环境，使程序的编辑、编译、连接、调试和运行等操作能在一个窗口内实现。

Turbo C 2.0 是美国 Borland 公司开发的一个 C 语言集成环境，它集编辑、编译、连接及运行功能于一体，使 C 程序的编辑、调试和测试非常简捷，编译和连接速度极快。软件本身占用空间小，可以执行 Install 文件进行安装，安装会产生一个 TC 文件夹及子文件夹和文件，或者直接将 TC 文件夹连同内容一起复制到硬盘后即可使用。

在 Turbo C 2.0 集成环境中的基本操作步骤为：在编辑菜单下进行输入和编辑，产生 C 源程序文件（扩展名为“.c”）；对该文件进行编译，生成可重定位的目标文件（扩展名为“.obj”）；编译过程中将对源程序进行语法检查，如果有错，则需要改错后重新编译；再将目标文件与系统库函数及其他目标程序进行连接装配，生成可执行文件（扩展名为“.exe”）；运行可执行文件即可获得运行结果。下面介绍 TC 集成环境。

1.5.1 Turbo C 2.0 的启动

Turbo C 2.0 可以在 DOS 或 Windows 下运行。

1. DOS 环境下启动

假设 Turbo C 2.0 安装在 C 盘根目录下的 TC 子目录中，要启动 Turbo C，只要先启动 DOS，然后按下面两种方式进入 Turbo C 集成环境。

(1) 直接进入 TC 子目录，启动 Turbo C，即（不带下划线的部分是键盘输入的）

```
C:\>cd tc ✓  
C:\tc>tc ✓
```

(2) 先进入用户子目录（假设用户子目录为 C:\USER），然后启动 Turbo C，即

```
C:\>cd user ✓  
C:\user>c:\tc\tc ✓
```

使用前一种方式进入 Turbo C，用户的程序文件将存放在 TC 子目录下；使用后者，用户的程序文件将直接存放在用户子目录下。

2. Windows 环境下启动

在 Windows 环境下，打开“资源管理器”，找到 Turbo C 所在的文件夹，用鼠标左键双击该文件夹下的“tc.exe”文件。

1.5.2 Turbo C 2.0 的主屏幕

不管在哪种环境下，Turbo C 启动后，将出现如图 1-2 所示的画面（主屏幕）。主屏幕

自上而下可分为 4 部分：顶行是主菜单，中间是编辑窗口和编译信息窗口，底行是功能键提示行。

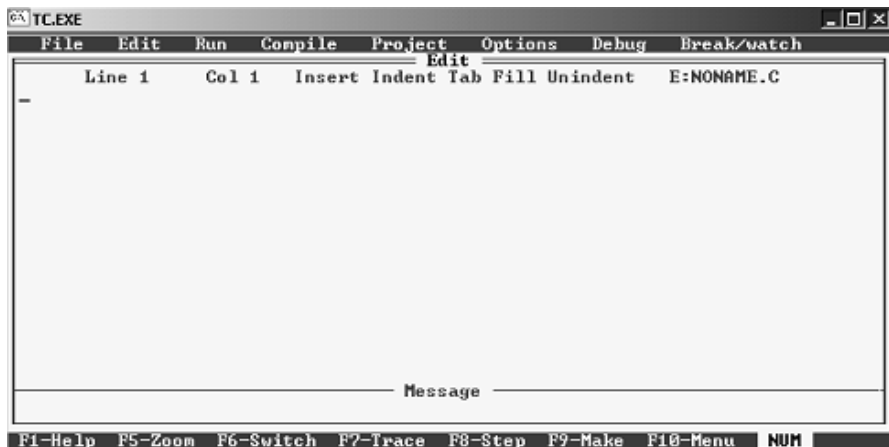


图 1-2 Turbo C 2.0 主屏幕

1. 主菜单

主菜单提供 Turbo C 环境的功能，共 8 个选项。

- File 文件管理，用于建立、装入和存储源程序文件，处理目录，切换到 DOS 以及退出 Turbo C 环境。
- Edit Turbo C 编辑器，用于建立、编辑和修改源程序。
- Run 运行，用于编译、连接和运行当前内存中的源程序。
- Compile 编译器，用于编译当前内存中的源程序。
- Project 项目管理，用于支持大型多程序文件的开发和维护。
- Options 可选项，用于设置 Turbo C 集成环境的各种操作方式。
- Debug 调试，用于设置各种调试选择项，以进行调试操作。
- Break/watch 中断/监视，用于对正在执行的程序的指定表达式进行监视，以及设置断点使程序执行时在指定的位置中断。

主菜单的选择方法有两种：

- 按 F10 进入主菜单，用“→”、“←”键将光标亮区移到所需选项上，然后按回车键。
- 按住 Alt 键，再按菜单项的第一个字母，大小写均可。例如，要选择 Edit，可按 Alt+E；要进入 File，可按 Alt+F。

2. 编辑窗口

编辑窗口是提供给用户的书写环境。窗口上部有一状态行，包括：

- 光标的当前位置（行号 line 和列号 col）。
- 插入（Insert）和改写转换。在插入方式下，输入的任何字符将插在光标之前，否



则输入的字符将替换光标处的字符。用 **Ins** 键可进行这两种方式的转换。

- 自动缩进格式转换 (**Indent** 和 **Unindent**)。可实现自动缩进排版或取消自动缩进，用 **Ctrl+OI** 进行切换。
- **Tab** 表示可用 **Tab** 键插入制表符或使 **Tab** 失效，用 **Ctrl+OT** 进行切换。
- **Fill** 表示可用任意一组空格或制表符填充一系列的空格。如果 **Fill** 打开，仅用空格。用 **Ctrl+OF** 进行切换。
- 右端显示当前正在编辑的文件名，当用户未给出文件名时，系统自动给出 **NONAME.C**。

3. 编译信息窗口

编辑窗口下面、带横线的“**Message**”以下部分是编译信息窗口。编译信息窗口用于显示编译、连接和调试时出现的警告和错误信息。

用 **F6** 键可以使光标在编辑窗口和编译信息窗口之间切换，用 **F5** 键可以扩大编辑窗口或编译信息窗口（操作对象是当前光标所在的活动窗口）。

4. 功能键提示行

功能键提示行给出当前操作中可以使用的主要功能键的功能说明。图 1-2 中下方显示的是主菜单下主要功能键的功能说明。

1.5.3 Turbo C 2.0 的子菜单

Turbo C 2.0 除 **Edit** 外，其他每个主菜单项都有一个下拉式子菜单。当选中某个主菜单项后，其下方出现子菜单。例如，图 1-3 是 **File** 下的子菜单。



图 1-3 Turbo C 集成环境的 File 子菜单

选择子菜单可用以下两种方法：

- 先选中主菜单项，然后用 **↑**、**↓** 键将光标亮区移到所需的子菜单上，再按回车键。

- 先选中主菜单项，然后键入所需子菜单项的第一个字母。例如在图 1-3 的 File 子菜单中，要选择 New，可按 N 或 n。

下面对常用的几个子菜单进行简单介绍。

1. 文件 (File)

File 子菜单有 9 个选择项，它们的功能是：

- Load 装入文件。
- Pick 显示最近装入过的文件的列表。
- New 编辑新文件，文件名为 NONAME.C。
- Save 将正在编辑的文件存盘。
- Write to 将正在编辑的文件改名存盘。
- Directory 显示当前工作目录的内容。
- Change dir 修改当前工作目录的路径名。
- OS shell 暂时退回到 DOS 环境，键入 EXIT 命令返回 Turbo C。
- Quit 退出 Turbo C，回到 DOS 环境。

2. 编辑 (Edit)

Edit 项不含子菜单，编辑程序的操作将在下面单独介绍。

3. 运行 (Run)

Run 子菜单含 6 个选择项，常用的有：

- Run 编译、连接和运行正在编辑的源程序。
- User screen 查看程序运行时所产生的输出屏幕。

4. 编译 (Compile)

Compile 子菜单含 6 个选择项，常用的有：

- Compile to OBJ 编译当前文件，并生成“.OBJ”文件。
- Make EXE file 直接将源程序编译和连接成可执行文件。
- Link EXE file 连接当前的“.OBJ”文件和库文件。
- Primary C file 指定待编译的主文件，而不是当前编辑的文件。
- Get info 显示当前对话的信息。

5. 可选项 (Options)

Options 设定集成环境的操作方式，它包含 7 个选择项，常用的有：

- Compiler 选择编译程序的存储模式。
- Environment 更改集成环境的工作方式。



- Directories 设定标题文件、库文件和可执行文件所在的目录。

6. 设置断点及监视 (Break/watch)

该子菜单有 7 个选择项, 功能是:

- Add watch 增加监视表达式
- Delete watch 删除监视表达式
- Edit watch 编辑监视表达式
- Remove all watches 删除所有监视表达式
- Toggle breakpoint 打开或关闭断点
- Clear all breakpoint 清除所有断点
- View next breakpoint 显示下一个断点

1.5.4 源程序的建立和编辑

1. 建立新文件

在主菜单下选中 **Edit** 选项, 或者先选中 **File** 项, 然后再选中 **File** 下的子菜单项 **New** (见图 1-3), 均可进入编辑新文件方式, 这时系统自动给出文件名 “NONAME.C”, 用户即可输入源程序。在输入过程中, 若使用分级缩进格式, 可使程序更清晰易读。如果自动缩进格式起作用, 每次按回车键后, 编辑程序会自动将光标放在与前输入行相同的缩进级上。

编辑文件后, 一般要给程序另外取名保存, 此时可以按 **Alt+F** 键调出 **File** 菜单, 从 **File** 菜单中选 **Save** 选项, 或者直接按快捷键 **F2**, 在出现的对话框中给文件另外起一个名字, 然后按回车键进行保存。

2. 装入老文件

若要调入已有的文件, 可先进入 **File** 项, 然后选中子菜单 **Load**, 或者直接按 **F3** 键, 屏幕上会提示用户输入文件名。有两种方法可以输入文件名:

- 如果知道文件名, 可以直接键入该文件名字。
- 如果对文件名没有把握, 什么也不要输入, 直接按回车键, **Turbo C** 会显示所有带 “.C” 扩展名的文件, 将光标移到所需文件上, 并按回车键选中即可。

确定文件名后, 文件内容会出现在编辑窗口内, 即可对文件进行编辑操作。

3. 编辑文件

在输入和编辑文件时, 可以使用系统提供的各种编辑功能。常用的编辑功能键列于表 1-1。

表 1-1 Turbo C 2.0 常用编辑功能键

类别	功能键	功能
光标移动	←或 Ctrl+S	光标左移一个字符
	→或 Ctrl+D	光标右移一个字符
	↑或 Ctrl+E	光标上移一行
	↓或 Ctrl+X	光标下移一行
	PgUp 或 Ctrl+R	光标上移一页
	PgDn 或 Ctrl+C	光标下移一页
	Home 或 Ctrl+QS	光标移至本行行首
	End 或 Ctrl+QD	光标移至本行行末
	Ctrl+QC	光标移至文件开头
	Ctrl+QP	光标移至文件末尾
插入删除	Ins 或 Ctrl+V	插入/替换开关
	Ctrl+N	光标处插入一个空行
	Ctrl+Y	删除光标所在行
	Ctrl+QY	从光标处删除到行末
	Ctrl+H 或 Backspace	删除光标左一个字符
	Del 或 Ctrl+G	删除光标处一个字符
块操作	Ctrl+KB	标记块首
	Ctrl+KK	标记块尾
	Ctrl+KT	标记当前光标处字符串
	Ctrl+KC	将标记块复制到光标处
	Ctrl+KV	将标记块移至光标处
	Ctrl+KY	删除标记块
	Ctrl+KH	消除块标记
	Ctrl+QB	光标移到块首
	Ctrl+QK	光标移到块尾
	Ctrl+KW	将标记块写入文件
	Ctrl+KR	将磁盘文件读至光标处
	Ctrl+KI	缩排一块
	Ctrl+KU	解除缩排



无论是新文件还是老文件，编辑完成后都可以直接按 F2 键存盘。也可以用文件菜单的 Save 选项直接存盘或 Write to 选项改名存盘。文件操作及程序调试的快捷键列于表 1-2。

表 1-2 文件操作及调试常用快捷键

功能键	功能
F1+F1	显示帮助索引
F1	显示与当前光标所在位置有关的提示信息
F2	将当前编辑文件存盘
F3	装入文件
F4	程序运行到光标所在行
F5	放大缩小活动窗口
F6	在编辑、消息或监视窗口间切换
F7	单步执行程序，可跟踪进函数内
F8	单步执行程序，不跟踪进函数内
F9	编辑、连接源程序，生成可执行文件
F10	进入主菜单
Alt+F1	显示上次显示的帮助信息
Alt+F3	在最近使用的文件中选取文件装入
Alt+F5	转入用户屏幕，按任意键返回
Alt+F6	在当前与最近文件间切换及在消息窗口和监视窗口间切换
Alt+F9	编译当前源程序生成“.OBJ”文件
Ctrl+F1	调用光标所在位置有关函数的上下文帮助信息
Ctrl+F2	结束当前调试，释放程序空间
Ctrl+F4	计算表达式的值
Ctrl+F7	增加一个监视表达式
Ctrl+F8	设置断点和删除断点
Ctrl+F9	运行程序

1.5.5 源程序的编译、连接和运行举例

TC 对应的可执行文件是 TC.EXE，但应用时需要多个文件的支持，如标题文件和库文件，这些文件分别存放在 TC 文件夹下的 INCLUDE 文件夹和 LIB 文件夹中。编译和连接时生成文件的存放位置也是确定的，如果这些文件的路径或名称发生改变，则 C 程序就不

能正常使用。一般情况下,编译器会到 TC\INCLUDE 文件夹下寻找包含文件,到 TC\LIB 文件夹寻找库文件;如果文件的路径不一致,则编译时会出现“Unable to open include file 'xxxx'”或连接时出现“Linker Error: Unable to open input file 'C0x.OBJ’”错误提示信息。

若想解决上述问题,可以在 Options/Directories 命令下重新设定搜索包含文件和库文件的路径,或者调整这些文件所在位置。但是这种修改是暂时的,当退出 TC 后再启动 TC 时会又回到原来的设置。如果需要永久修改,可以选择 Options/Save options 命令,将设置保存成 TCCONFIG.TC 文件。

下面举例说明。在 A 计算机上 C 盘根目录下的 TC 文件夹及相关文件可以正常使用,但将其复制到 B 计算机的 E 盘就会出问题。解决方法是可以复制文件到 B 计算机的 C 盘,或者修改 Options/Directories,将目录设为如图 1-4 所示的位置(用移动光标键选中某个项再按回车键可进行修改)。这时编译文件时就会自动到 E:\TC\INCLUDE 文件夹查找包含文件,连接文件时会自动到 E:\TC\LIB 文件夹查找库文件,这样再进行编译连接就正常了。



图 1-4 TC 环境设置

当建立新文件或装入老文件后,按 Alt+R 键进入 Run 菜单,选择 Run 选项后就可以进行编译、连接和运行(或者用 Alt+C 调出 Compile 菜单,将编译、连接分步完成后,再用 Alt+R 运行程序)。在编译和连接过程中,如果发现错误,编译信息窗口中会显示警告和错误信息(包括错误说明和错误位置),并将错误所在的行反相显示。用户可以根据错误提示信息,用移动键将光标移动到错误所在行进行修改;只要按回车键或者按 Alt+E 键,就可对出错程序行进行编辑修改。全部修改完成后,再次用 Alt+R 重新编译和连接。如果编译过程中未发现错误,系统将自动执行该程序,执行完毕后会返回到集成环境。若未看清屏幕显示的结果,可按 Alt+F5 键进入用户屏幕,阅读程序运行结果,然后按任意键返回到集成环境;也可按 Alt+FO 键暂时进入 DOS,同样可以看清屏幕上的结果,然后用 EXIT 命令返回到集成环境。

初学者输入源程序时往往会有各种各样的错误,比如关键词写错了,语句后没有分号,或者该有空格的地方漏写了空格,这些问题编译时都会出现错误信息。错误信息会指出在哪一行出现了哪种错误,但是有时出错提示并不准确:有时前一行少了分号,出错提示指出的是后一行;有时程序中只有前面的某一个语句有错,出错提示会给出很多错误信息。这需要读者多进行调试程序的实践,逐步掌握调试程序的技巧。



比如，在 TC 窗口中按 Alt+E 键输入源程序：

```
#include <stdio.h>
main()
{
    inta=1,b=2,c;
    c=a+b;
    printf("\na=%db=%dc=%d",a,b,c);
}
```

将源程序以 li.c 存盘，然后按 Alt+C 键调出 Compile 菜单，选中 Compile to Obj 选项并回车进行编译。因为在程序中第 3 行的 int 和 a 之间少了一个空格，会出现如图 1-5 所示界面。

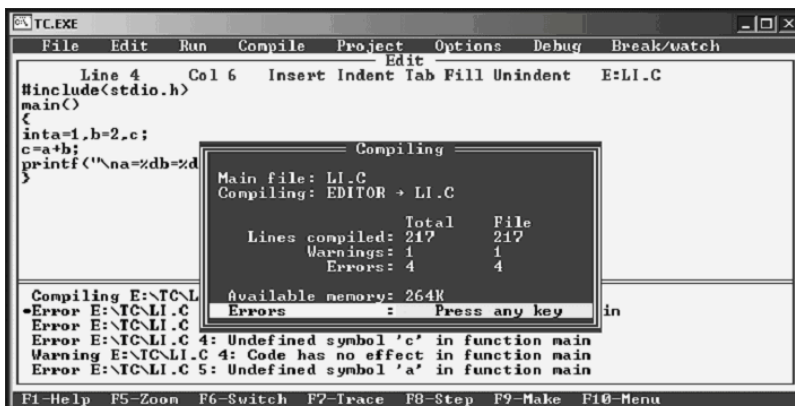


图 1-5 TC 编译时的界面

编译时显示编译了 217 行（stdio.h 被包含到当前文件参加编译），给出一个警告（Warning）、4 个错误提示（Error）。一般程序编译时都会提出警告和错误，错误一定要去掉，警告有的可以不用处理，而有的却可能预示着程序存在潜在的大问题。按任意键后，在 Message 窗口会显示出错误和警告信息的具体内容（所在行号，错误类型），见图 1-6。

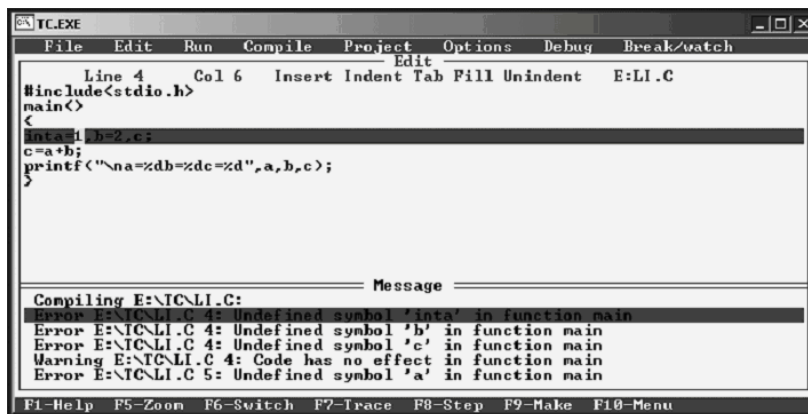


图 1-6 编译时发现错误的提示信息

这时按 **Alt+E** 键并进行修改, 修改后若编译无错, 用 **Alt+C** 键调出 **Compile** 菜单, 选择 **Link** 项进行连接。连接无错后, 按 **Alt+R** 键或 **Ctrl+F9** 键运行程序, 然后按 **Alt+F5** 键转到用户屏幕察看程序的输出结果。

也可以直接按 **Alt+R** 键将编译、连接、运行的步骤依次自动进行。若在某个步骤有错, 则会停止并给出错误提示, 改正后才可以再次运行。

要点回顾

1. C 兼有高级语言和低级语言的特点, 是应用广泛的程序设计语言。
2. C 语言程序由一个或多个函数构成, 每个程序有且只有一个主函数 **main()**, 程序执行由主函数开始和结束, 在主函数执行过程中可以调用其他函数。
3. C 语言的风格灵活, 源程序采用缩进格式表示程序的层次, 但缩进格式只影响外观, 对程序的执行逻辑没有影响。
4. C 程序要经过编辑、编译、连接和运行 4 个步骤, 这些步骤通常在集成开发环境中完成。
5. TC 集成开发环境的几个主菜单项为: 文件、编辑、编译、运行、选项。每个主菜单项又有子菜单, 可以由 **Alt** 和字母键调出子菜单, 也可以用移动键选中子菜单。

习 题

1. 简述 C 程序的特点。
2. 为什么要在程序上加注释? 怎样在程序上加注释? 加入注释对程序的编译和执行有没有影响?
3. 编写一个简单的 C 程序, 使其输出以下信息:

```
*****  
*   hello.c   *  
*****
```

4. 简述 C 程序的上机操作过程, 并将上题所编写的程序在机器上编辑、编译、连接和运行。
5. 在 TC 文件夹下建立一个名为 **newout** 的文件夹, 改变 **Options** 菜单下的 **Directories** 子菜单项, 使得输出文件 (*.obj 和 *.exe) 存放在 **newout** 文件夹下。



第 2 章 数据类型、运算符及表达式

本章的学习目标:

掌握进位计数制的概念和不同进位计数制之间的转换方法,了解常量、变量以及数据类型的含义,掌握变量定义的含义和变量定义的方法;了解运算符的分类及优先级的概念,掌握运算符的表示及其用法;掌握表达式的分类以及表达式运算时的数据类型转换规则。

程序设计的重要内容之一是数据结构。不同的应用领域,采用的数据结构是不同的。各种程序设计语言为了适应不同领域的需要,都明确规定了数据类型的概念,以处理不同的数据结构。所谓数据类型,是指程序设计语言所允许的变量和常量的种类,即每个常量、变量或表达式的值都属于确定的数据类型,并且有特定的运算。虽然在程序执行期间,变量的值在不断地改变,但是变量所有可能的取值以及所允许的操作都在程序中显式或隐含地被规定。

在程序设计语言中使用数据类型的概念,具有如下优点:

- 带来了程序的简明性。数据类型明确规定变量应该取什么类型的值,操作应该在什么类型的操作数上进行,从而能够进行类型匹配检查,这是高级语言较之机器语言的一大进步。
- 提高了程序的执行效率。当程序运行时,每个变量的当前值总是存储在一个或多个存储单元中。由于类型定义总是出现在程序执行部分之前,所以编译系统可以决定和预先分配存储这些变量的值所需要的存储空间,而不用等到运行时才去分配变量的存储空间,这就提高了程序的执行效率。

本章先介绍有关数制基础及计算机中数的表示,然后介绍 C 语言描述数据的方式以及对数据的基本操作。

2.1 数制基础及计算机中数的表示

2.1.1 数的二进制、八进制和十六进制表示法

人们通常习惯使用十进制数,十进制数有 0,1,2,⋯,9 十个数码。在计算机内部,为了表示方便,数据以二进制形式存放。这是因为二进制只有 0 和 1 两个数码,在电路中可以用电平的高低或者脉冲的有无来表示,容易实现。

下面首先介绍各数制之间如何转换。对于十进制数 123,可以表示为 $1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$,从右至左对应的 10 的 0 次幂、10 的 1 次幂、10 的 2 次幂称为该位数字的权。同理,二进制数从右至左,各位的权依次是 2 的 0 次幂,2 的 1 次幂,2 的 2 次幂,⋯。二进制数 1011 可以表示为 $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$,根据这一点,将二进制数各位数码乘以该位对

应的权再按照十进制规律相加，可以将一个二进制数化成十进制数。例如：

$$(101011)_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 43$$

将十进制数化为二进制数的方法是将十进制数除以 2 取余数，再把商不断除以 2，记下每一次的余数，直到商为 0。把每一次的余数排起来就是该十进制数对应的二进制数：第一次的余数是二进制数的最低位，最后一次的余数是二进制数的最高位。这样的方法称为除 2 取余法。例如，求 $x=13$ 的二进制表示的过程如下：

除数	被除数	余数
2	13	
2	6	1
2	3	0
2	1	1
	0	1

所以 13 的二进制表示是 1101。

虽然在计算机内部数据是以二进制形式存放的，但是人们还是使用十进制形式来输入输出数据，并通过程序自动完成进制间的转换。有时想知道一个数的二进制表示中某一位数字是 0 还是 1，这时直接使用数的二进制形式就十分直观。但是用二进制形式表示数的缺点是位数太多，不易读写。为此，在计算机科学中，人们经常采用八进制或十六进制的形式表示一个数。采用这两种进制表示一个数，位数既不很多又可以很容易地知道数中各个二进制位是 0 还是 1。C 语言中，可以用十进制、八进制和十六进制来表示一个常数，以 1~9 开头的数字表示十进制数，以 0 开头的数字表示八进制数，以 0x 开头的数字表示十六进制数。

十进制数和八进制数或十六进制数之间的转换与十进制数和二进制数之间的转换类似。二进制数和八进制数之间或二进制数和十六进制数之间的转换很容易，下面首先说明转换方法。

八进制数有 8 个数码，0,1,2,3,4,5,6,7，每一位八进制数可以对应 3 位二进制数，同样每 3 位二进制数对应一位八进制数。把一个二进制数从右端开始，每 3 个二进制位为一组，每组用一个八进制数字表示，就可以得到对应的八进制表示。例如：

$$10110011 = (10)(110)(011) = 0263$$

反之，把一个八进制数的各位数字表示成 3 位二进制形式，就得到该数的二进制表示。例如：

$$0147 = (001)(100)(111) = 001100111$$

人们也经常使用十六进制的形式表示数。十六进制数字是 0,1,2,⋯,9,a,b,c,d,e,f，其中 a,b,c,d,e,f (A,B,C,D,E,F) 分别表示数字 10,11,⋯,15。十六进制与二进制之间的转换关系同八进制与二进制的转换类似，不同的是每一位十六进制数字和 4 位二进制数字对应。

$$\text{例如：} 0x9bfa = 1001101111111010$$

$$1010001111001101 = 0xa3cd$$



2.1.2 数在机器内部的表示方法

任何数据在机器内部都是以二进制形式存放的，下面主要介绍整数在机器内部的表示方法。为了区分正整数和负整数，通常约定一个数的最高位（最左边的一位）为符号位，如果符号位为 0，则表示正整数，符号位为 1，表示负整数。在此约定下，有两种常用方法可以表示一个整数，它们分别是原码表示法和补码表示法。

一个整数在计算机中一般用 16 个二进制位（即 2 个字节）来表示。一个整数的原码的最高位是这个数的符号位，其余位是这个数绝对值的二进制形式。例如 5 的原码是 0000000000000101，即 0x0005；-5 的原码是 1000000000000101，即 0x8005。

一个正整数的补码与它的原码相同，一个负整数的补码则是把它的原码的各位（符号位除外）求反，即 0 变成 1，1 变成 0，然后在末位加 1。-5 的原码是 1000000000000101，-5 的补码是在 111111111111010 的末位加 1，等于 111111111111011，即 0xfffb。

补码在计算机中应用十分广泛，计算机中的带符号整数都是用补码形式表示的。

2.2 C 语言的数据类型及其取值范围

每种高级语言都根据其所应用的范围规定了它可以使用的数据类型。C 语言有丰富的数据类型：

- 基本类型（简单类型） 包括数值类型、字符类型（char）和枚举类型（enum）。
- 结构类型（组合类型） 包括数组类型、结构体类型（struct）、共用体类型（union）、文件类型（FILE）、指针类型、空类型（void）。结构类型数据是指由基本类型数据按一定的规律构造而成的，例如若干个整数的有序排列就构成一个整型数组。

2.2.1 基本数据类型

本节仅介绍基本数据类型（不包括枚举类型），其他类型在后续章节中介绍。C 语言提供的基本数据类型以及对应的关键字如表 2-1 所示。其中，字符型（char）通常描述一个字符；整型（int）描述我们日常使用的整数，整数在计算机中是准确表示的；浮点型（float）和双精度型（double）描述我们日常使用的实数，实数在计算机中一般是近似表示的，浮点型的近似程度比较低，而双精度型的近似程度则比较高，因此，浮点型有时也叫单精度型。

表 2-1 基本数据类型以及对应的关键字

数据类型	关键字
字符型	char
整型	int
浮点（单精度）型	float
双精度型	double

2.2.2 基本类型数据的存储空间长度及数据取值范围

C 语言对不同类型的数据分配不同长度的存储空间。一般说,有 1 个字节(8 位)、2 个字节(16 位)、4 个字节(32 位)、8 个字节(64 位)和 10 个字节(80 位)等,大小与宿主计算机的硬件特性有关。

- **char 型(字符型)** char 型数据占 1 字节存储空间,表示字符的 ASCII 码。
- **int 型(整型)** int 型数据又分为短整型(short)、普通整型、长整型(long)3 种。char 和 int 型还可以表示为有符号(signed)或无符号(unsigned),有符号和无符号数之间的区别在于对数的最高位的处理:有符号数将最高位视为符号位,0 表示正号,1 表示负号;无符号数则将最高位也视为数值位。IBM PC 系列计算机以定点二进制补码形式存储有符号数,一般都以 2 个字节(16 位)存放短整型和普通整型数,以 4 个字节(32 位)存放长整型数。
- **float 型(实型)** float 型数据占 4 字节存储空间,以浮点形式存储。有关定点数和浮点数的概念,请参阅其他书籍。
- **double 型(双精度型)** double 型数据占 8 字节存储空间,存储方式与 float 基本相同。

不同类型计算机上各种数据类型所占的二进制位(bit)数不同,可以用 sizeof 方法测试所用环境下的各种数据类型所占的字节数(byte)。

在 IBM PC 系列计算机中,C 语言基本数据类型的长度及其取值范围见表 2-2。

表 2-2 C 语言基本数据类型的长度和取值范围(IBM PC 系列)

关键字	数据类型	长度 (bit)	取值范围
Char	字符型	8	-128~127
[signed] char	有符号字符型	8	-128~127
unsigned char	无符号字符型	8	0~255
int	(普通) 整型	16	-32 768~32 767
[signed] int	有符号整型	16	-32 768~32 767
unsigned [int]	无符号整型	16	0~65 535
short int	短整型	16	-32 768~32 767
unsigned short [int]	无符号短整型	16	0~65 535
long int	长整型	32	-2 147 483 648~2 147 483 647
[signed] long [int]	有符号长整型	32	-2 147 483 648~2 147 483 647
unsigned long [int]	无符号长整型	32	0~4 294 967 295
float	(单精度) 实型	32	$-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$
double	双精度(实)型	64	$-1.7 \times 10^{308} \sim 1.7 \times 10^{308}$
long double	长双精度型	80	$-1.1 \times 10^{4932} \sim 1.1 \times 10^{4932}$

注:方括号内的关键字可省略。



2.3 各种类型常量及其表示

2.3.1 整型常量

整型常量可以用十进制、八进制或十六进制来表示。C 语言规定，以 1~9 开头的数字表示十进制数，以 0 开头的数字表示八进制数，以 0x 开头的数字表示十六进制数。例如：65 535（十进制数），-32 768（十进制负数），0 177 777（八进制数），0xffff（十六进制数），0xa3（十六进制数）。

在 16 位字长的计算机系统中，由于一个整数以 2 个字节储存，因此数的范围如前所述为 -32 768~32 767，用八进制数表示则为 0~0 177 777，用十六进制数表示为 0x0~0xffff。超出上述范围的整常数，要用长整数（32 位）表示。在 C 语言中，整数后加字母 l 或 L 为长整数。例如：-12L（十进制长整数），774545L（十进制长整数），076L（八进制长整数），0200000L（八进制长整数，等于十进制数 65 536），0x10000L（十六进制长整数，等于十进制数 65 536）。

在 16 位字长的机器中，一旦把一个常数表示成长整数，系统便为其扩充储存空间为 4 个字节。所以从值的大小上看，12L 与 12 没有区别，但它们占用的储存空间不相同。

下面各个数据不是合法的整形常量：

- (1) 0987 以 0 开头的数字表示八进制数，但后面的数字中又出现了数字 8 和 9；
- (2) 1faf 以 1~9 开头的数字表示十进制数，但后面的数字中又出现了 a 和 f；
- (3) 0x1hfg 以 0x 开头的数字表示十六进制数，但后面的数字中又出现了 h 和 g。

2.3.2 实型常量

实型常量只能用十进制形式表示，不能用八进制或十六进制形式表示。实型常量可以用小数形式或指数形式表示，如 34.5，0.345，345.0，3.14159，1e2，1.5e-3 等均为合法的实型常量。指数形式又称为科学记数法，用于表示浮点数，由尾数、e（或 E）和指数 3 部分组成，如 1.5e-3 中的 1.5 被称为尾数，-3 被称为指数，表示的数相当于 1.5×10^{-3} ，即 0.0015。指数形式必须有尾数，并且指数部分必须是整数，如 -3e1.5，e2 都是错误的。

实型常量不分单精度型和双精度型，大多数 C 编译系统都将实型常量隐含作为双精度数来处理。若要表示一个单精度型常量，应在其末尾加一个 f 或 F，如 1.2345f。实型常量可以赋给一个 float 型或 double 型变量。一个 float 型变量能接收 7 位有效数字，一个 double 型变量能接受 16 位有效数字，一个长双精度型变量能接受 17 位有效数字。

例 2.1 测试单精度型和双精度型变量的有效数字位数。

```
/* Example 2-1: testing for significant figure */
#include <stdio.h>
main()
{
    float f1=1234.5678;
    double d1=1234.5678901234567890;
```

```
printf("f1=%f,d1=%f\n",f1,d1);
printf("f1=%.20f,d1=%.20f\n",f1,d1);
}
```

程序运行结果为:

```
f1=1234.567749,d1=1234.567890
f1=1234.56774902343750000000,d1=1234.567890123456890000000
```

可以看出,单精度型有效数字位数为 7 位,双精度型有效数字位数为 16 位,默认小数位数都是 6 位。这两种类型虽然可以强制输出更多的位数,但不能提高其精确度。

2.3.3 字符常量

字符常量是指仅含单个 ASCII 字符的常量。它是用一对单引号括起来的一个字符,如 'a','A','?','#'。注意,单引号只是字符与其他部分的分隔符,或者说是字符常量的定界符,不是字符常量的一部分,当输入一个字符常量时不输入此单引号。不能用双引号代替单引号,如 "a" 不是字符常量。注意,单引号中的字符不能是单引号或反斜杠,如 "或 \" 不是合法的字符常量。

字符常量有两种表示方法。

(1) 对可显示字符,直接用单引号将该字符括住,即表示字符常数。如 'A','a','5','\$' 等。

C 语言将字符常量也视为 1 字节的整数,其值为对应的 ASCII 码值,可以像整数一样参加数值运算。例如, 'A' 的 ASCII 码十进制值是 65, 'A'+5 的值为 70。又如, 'a' 的 ASCII 码十进制值是 97, 'a'-'A' 的值即为 32。由于任何字母其大、小写的 ASCII 代码值均相差 32,这就很容易实现大小写字母之间的转换。

当用 printf 函数输出字符常量时,既可用 "%d" 将其按整数形式输出,也可用 "%c" 将其按字符输出。同样,对 0~255 之间的整数也可以用 "%c" 输出其对应的字符。

例 2.2 字符常量的表示方法。

```
/* Example 2-2: character constant */
#include <stdio.h>
main()
{
    printf("%d",'B');
    printf("%c",'B');
    printf("%c",66);
    printf("%c\n",'B'+32);
    printf("%d,%c,%d,%c\n",'\\n','\\0','\\0',0x41);
}
```

程序运行结果为:

```
66,B,B,b
10 , ,0,A
```

从以上结果可以看出,字符常量可以作为整数使用,整数也可以作为字符常量使用。特别对字符常量 '\0',按 "%c" 以字符格式输出时,留出一个字符的空位;按 "%d" 以十



进制整型格式输出时，则输出数值 0。

(2) 转义字符。对不可显示字符，主要是那些控制字符，如换行符、回车符、换页符等，还有一些在 C 语言中有特殊含义和用途的字符，如单引号、双引号、反斜杠，只能用转义字符表示。

C 语言规定：

(1) 用反斜杠开头后面跟一个字母代表一个控制字符；

(2) 用\\代表字符"\"，用\'代表单引号（'）字符。

(3) 用\后跟 1~3 个八进制代表 ASCII 码为该八进制数的字符；用\x 后跟 1~2 个十六进制数代表 ASCII 码为该十六进制数的字符。

因为反斜杠后面的字符已不再是原来字符的作用而转为新的含义，因而称转义字符，见表 2-3。如\n'是换行符，同样也可以用'\012'、'\xa'表示，因为换行符的 ASCII 代码八进制值 12，十六进制值是 a。

表 2-3 C 语言中的转义字符

转义序列	含义	十进制 ASCII 代码值	说明
\0	NUL	0	空字符（即无字符）
\a	BELL	7	报警铃响
\b	BS	8	退格符（backspace）
\f	FF	12	换页符
\n	NL(LF)	10	换行符
\r	CR	13	回车符
\t	HT	9	水平制表符（Tab）
\v	VT	11	垂直制表符
\\	\	92	反斜杠
\'	'	44	单引号
\"	"	34	双引号
\ddd			1~3 位八进制数所代表的字符
\xhh			1~2 位十六进制数所代表的字符

可显示字符也可以用转义字符表示，例如字母“A”的 ASCII 代码八进制值是 101，十六进制值是 41，所以，可以用'\0101'、'\x41'等形式表示字母“A”。

综上所述，用转义字符表示字符常量时，是通过转义字符将其后面的字母、八进制或十六进制数码转变为特定的含义。

例 2.3 转义字符的使用。

```
#include <stdio.h>
Main()
{
```

```
char ch;  
ch = '\45';          /* 将 ASCII 值为八进制数 45 的字符赋给 ch */  
printf("%c\n", ch);  
}
```

在 IBM PC 机上运行，可在屏幕上输出：%。

2.3.4 字符串常量

字符串常量是用双引号括起来的 0 个或多个字符的序列。例如：

"I am a student.", 是一个英文句子；

"x", 只含一个字母 x；

""，是一个空串，即不含任何字符的字符串；

"01234567"，全部由数字字符组成的字符串；

"I said, \"goog morning!\"", 包含有用转义序列 (\\) 表示的字符双引号，它代表下面的英文句子：I said, "goog morning!". 这是因为双引号已作为字符串常量的定界符有了特殊的含义，所以只能用转义序列表示，不能直接使用。

注意，字符串常量和字符常量在计算机中存储时，存放的都是对应字符的 ASCII 代码值，但它们又有明显的区别。

(1) 从表示方式上看，字符串常量是以双引号定界，而字符常量则以单引号定界，如 "A" 是字符串常量，'A' 则是字符常量。

(2) 任何字符常量都只占 1 个字节存储单元，在内存中以相应的 ASCII 代码存放；而字符串常量则占用一段连续的存储单元，其所占字节数为字符串长度加 1。例如，字符串常量 "A" 的存储方式为（十六进制表示）：4100，而字符常量 'A' 的存储方式为 41。又如 "\\0" 和 "\0"，一个是含一个空字符的字符串，一个是空字符，它们的存储格式也不一样。前者占 2 个字节：0000，后者只占一个字节 00。特殊情况是，空字符串 ""（即两个引号连写，不含任何字符），它只占 1 个字节的存储单元，即 00，而空字符常量 '\0' 也占 1 个字节的存储单元，它们的存储方式类似，但表示方法不同，用 printf 函数输出时，显示格式也不相同。

(3) 字符型常量通常存放在字符型变量中，而字符串常量则必须存放在字符型数组中。

(4) 字符型常量可以与整数混合运算，而字符串常量则不可以。

2.3.5 符号常量

C 语言除了提供各种类型的常量和变量外，还提供了符号常量（或称符号常数）。符号常数是用标识符表示的常数。从外表看它是标识符，像变量，但实质上它是常数，它的值不能通过赋值或输入改变。程序中采用符号常数具有以下好处：

(1) 用符号常数可以清晰地表示常量所代表的物理意义。例如，用符号常数 g 和 pi 表示重力加速度和圆周率，它们比写成 9.8 和 3.14159 更容易看出其代表的物理意义，同时又符合人们的习惯，这就增强了程序的可读性。

(2) 当程序中多次出现某一个常数时，需要多次书写，这样一是会耗费时间，二是可能出现不一致。用符号常数就可以用较短的符号代替，从而有效地避免这两个缺点。



(3) 当程序中多次出现同一常数需要修改时，必须逐个修改，则很可能出现漏改或错改。用符号常数只需修改定义，即可做到一改全改。

定义符号常数的方法将在后续章节中讨论。

2.4 变量及其类型定义

变量是指那些在程序运行过程中其值可以改变的量。变量代表着存储器中的一个位置，也可以说变量就是命名的内存单元。程序中用变量名来引用内存单元，变量在整个程序运行中其值可以改变，但某一时刻变量有惟一确定的值。变量有两个特性：名称和数据类型。在 C 语言中，变量具有如下 3 个属性：

- 变量分为不同的数据类型，数据类型决定了该变量可以存储什么数据以及可以进行什么运算。
- 变量分为不同的存储类别，存储类别决定了变量在计算机中的存储位置及其寿命（生存期）；有关变量的存储类别将在第 6 章中学习。
- 每个变量都有一个作用域，作用域是由变量在程序中被定义时的位置决定的。

2.4.1 变量名

一个变量应该有一个变量名作为标识。在 C 语言中，用来标识变量名以及符号常量名、函数名、数组名、类型名、文件名的有效字符序列，称为标识符。标识符只能由字母、数字或者下划线 3 种字符构成，且第一个字符必须为字母或者下划线，其长度（字符个数）无统一规定，随系统而不同。

有一些字符组合已被 C 语言本身使用使有特定的含义，这些字符组合称为 C 语言的关键字或保留字。ANSI 标准定义了下述 32 个关键字：

auto	double	int	struct	break	else
long	switch	case	enum	register	typedef
char	extern	return	union	const	float
short	unsigned	continue	for	signed	void
default	goto	sizeof	volatile	do	if
while	static				

标识符不能和 C 语言的关键字相同，也不应该和 C 语言的库函数名相同。在 C 语言中，大小写字母是有区别的，sum、SUM 和 Sum 是 3 个不同的标识符。

在选择变量名和其他标识符时，应尽量做到“见名知意”，即选择有含意的英文单词、汉语拼音（或其缩写）作为标识符，如 count, name, sum 等，以便于理解变量的内容。除了数值计算程序外，一般不提倡采用代数符号（如 a,b,c,x,y,z 等）作为变量名，以便提高程序的可读性（在本书的程序举例中，为了简便常用代数符号作为变量名）。

C 语言中，标识符的命名规则如下：

- 第一个字符必须是英文字母或下划线，后跟任意字符、数字或下划线。

- 不能是 C 语言中的关键字、已定义的函数名或 C 语言的库函数名。
- 在同一范围内不能重名。

2.4.2 变量的数据类型

与常量数据类型相似，C 语言中的变量也有字符型、整型、浮点型、双精度型等数据类型。变量的数据类型必须提前定义，而与变量名本身无关。为什么要规定变量的类型呢？原因如下。

- 不同类型的数据在内存中占据不同长度的存储区，并且采用不同的表示方式（指数数据在机器内部的表示方式）。例如，一般的微型机是用两个字节以定点形式存放一个整数，用 4 个字节以浮点形式存放一个实数。
- 类型决定了变量或表达式所能取值的范围。例如，整数的范围为 -32 768~32 767，单精度实数的范围约为 $-10^{38} \sim 10^{38}$ ，双精度实数的范围约为 $-10^{308} \sim 10^{308}$ 。
- 一种数据类型对应着一组允许的操作。例如，对整型数据，可以进行“求余”运算（5%2 的值为 1，即 5 除以 2 的余数为 1）；而实型数不能进行“求余”运算。

当变量的数据类型定义之后，就规定了该变量只能存储对应类型的数据。例如，定义了 int 型变量 x，则 x 中只能存放整型数，即使把其他类型的数据赋给它，如 x=3.14159；C 语言编译系统也会先将 3.14159 转换为整型数 3 后再存放到 x 中。

2.4.3 变量的定义

C 语言规定，一个 C 程序中使用到的任何变量都必须在使用前定义。定义变量时，一是定义变量的数据类型，二是定义变量的名称，三是说明变量的存储类别。在一个程序中，一个变量只能属于一个类型。定义变量的一般格式为：

【存储类别】数据类型 变量清单；

其中，“数据类型”用类型关键字标识；“变量清单”中可含多个变量名，其间用逗号隔开；“存储类别”分为寄存器变量（register）、自动变量（auto）、全局变量和静态变量（static），具体意义将在第 6 章中讨论。变量数据类型定义举例如下：

```
char ch1;           /* 指定变量 ch1 为字符型 */
int i,j,k;          /* 指定变量 i,j,k 为整型 */
short int si;       /* 指定变量 si 为短整型 */
unsigned ui;        /* 指定变量 ui 为无符号整型 */
float f1,f2;        /* 指定变量 f1 ,f2 为单精度实型 */
double profit,loss; /* 指定变量 profit, loss 为双精度实型 */
```

2.4.4 变量的初始化

变量的初始化，就是在定义变量的同时赋予其与类型相一致的初值。如：

```
register int a=2,b=3;
static double x,pi=3.1415926;
char ch1='a',ch2,ch3='b',ch4=ch1;
```



其中，第一条定义语句对两个寄存器变量 `a` 和 `b` 都赋了不同的初值；第二条定义语句只对变量 `pi` 赋了初值；第三条定义语句为 4 个字符型变量赋了初值，`ch1`、`ch2` 和 `ch3` 用字符常量赋初值，且 `ch2` 和 `ch3` 赋了相同的初值，`ch4` 则用已赋过初值的变量 `ch1` 赋初值。

对变量的初始化，需要说明以下几点：

- 寄存器变量和自动变量若未进行初始化，则应该在程序中通过赋值语句或输入语句进行赋值后再使用，否则它们的值是不确定的。
- 寄存器变量和自动变量的初始化是在程序执行期间完成的。它们所在的程序段每执行一次，都要重新进行初始化。
- 全局变量和静态变量的初始化是在编译阶段完成的。若不对这两类变量进行初始化，则编译系统自动为其赋零值。
- 全局变量只能用常数或常数表达式进行初始化，而寄存器变量、静态变量和自动型变量则既可以用常数进行初始化，也可以用先前已初始化的变量进行初始化。

例 2.4 一个学生数据（即学号、姓名、成绩）的赋值及输出。

```
#include<stdio.h>
main()
{
    int num;          /*整型量*/
    char name;        /*字符型*/
    float score;      /*浮点型*/
    num=1;
    name='w';
    score=88.5;
    printf("\n%d,%c,%f",num,name,score);
}
```

2.5 C 语言运算符的分类、运算优先级和结合性

运算（即操作）是对数据的加工。对于最基本的运算形式，常常可以用一些简洁的符号记述，这些符号称为运算符或操作符，被运算的对象（数据）称为运算量或操作数。C 程序中的数据运算主要是通过对表达式的计算完成的。表达式是将运算量用运算符连接起来组成的式子，其中的运算量可以是常量、变量或函数。由于运算量可以为不同的数据类型，每一种数据类型都规定了自己特有的运算或操作，这就形成了对应于不同数据类型的运算符集合。C 语言提供了很多数据类型，运算符也相当丰富，共有 13 类，详见表 2-4。

表 2-4 C 语言运算符分类

功能	运算符	运算量类型
算术运算	+ - * / % ++ --	数值型、指针、字符型
关系运算	> < >= <= == !=	数值型、字符型、成员
逻辑运算	! &&	数值型、指针、字符型
赋值运算	= += -= *= /= %= <<= >>= &= ^= =	数值型、指针、字符型、结构、联合
条件运算	?:	数值型、指针、结构、联合
位运算	<< >> ~ ^ &	整型
求字节运算	sizeof	类型说明符、表达式
下标运算	[]	数组、指针
指针运算	* &	指针、地址
取成员运算	. ->	结构、联合
逗号运算	,	表达式
括号运算	()	表达式
强制类型转换运算符	类型符	表达式
其他（如函数调用运算符()）		

在学习运算符时应注意以下几点：

(1) 运算符的功能。如加、减、乘、除等。

(2) 运算量的关系。即：

① 要求运算量的个数关系。例如，有的运算符要求有两个运算量参加运算（如+、-、*、/），称为双目（或二元）运算符；而有的运算符（如负号运算符、地址运算符&）只允许有一个运算量，称为单目（或一元）运算符。

② 要求运算量的类型关系。如+、-、*、/的运算的对象可以是整型或实型数据，而求余运算符%要求参加运算的两个运算量都必须为整型数据。

(3) 运算的优先级。如果不同的运算符同时出现在表达式中时，先执行“优先级”高的运算。所谓优先级，是指进行运算的优先次序。我们所熟知的四则运算法则之一是“先乘除，后加减”，说的就是优先级，即乘除优先于加减。

(4) 结合方向，即结合性，指的是同一优先级的运算符出现在同一表达式中时进行运算的顺序。在一个表达式中，有的是按从左到右的顺序运算，有的是按从右到左的顺序运算。如果在一个运算量的两侧有两个相同优先级的运算符，则按结合性处理。例如：3*5/6，在5的两侧分别为*和/，根据“先左后右”的原则，5先和其左面的运算符结合，这就称为“自左至右的结合方向（或称左结合性）”。在C语言中，并非都采取自左至右的结合方向，有些运算符的结合方向是“自右至左”的，即“右结合性”。例如，赋值运算符的结合方向就是“自右至左”的，因此对赋值表达式a=b=c=5，根据自右至左的原则，它相当于a=(b=(c=5))。表2-5列出了所有运算符的优先级和结合性。



表 2-5 C 语言运算符的优先级和结合性

优先级	运算符	功能	运算量个数	结合性
15	()	圆括号, 提高优先级		左
	[]	下标运算, 访问地址		
	->	指向结构或联合成员		
	++ -- (后缀)	取结构或联合成员		
14	++ -- (后缀)	自加、自减	1	右
	!	逻辑非		
	~	按位取反		
	++ (前缀)	自加		
	-- (前缀)	自减		
	-	负号运算符		
	(type)	强制类型转换		
	*	访问地址或指针		
13	&	取地址	2	左
	sizeof	测试数据长度		
	*	乘法		
12	/	除法	2	左
	%	求整除的余数		
11	+	加法	2	左
	-	减法		
10	<<	左移位	2	左
	>>	右移位		
9	< <= > >=	关系运算	2	左
8	== !=	等于 不等于	2	左
7	&	按位与	2	左
6	^	按位异或	2	左
5		按位或	2	左
4	&&	逻辑与	2	左
3		逻辑或	2	左
2	?:	条件运算	3	右
1	= += -= *= /= %= ^=	赋值运算	2	右
	= &= >>= <<=			
1	,	逗号运算		左

注：表中所列优先级，“15”级为最高，“1”级为最低。

(5) 注意所得结果的类型，即表达式值的类型，尤其当两个不同类型数据进行运算时，特别要注意结果值的类型。我们在后面要专门讨论这个问题。

本章仅介绍常用的运算符和表达式，下标运算符、指针运算符以及取成员运算符等将在后续章节中陆续介绍。

2.6 算术运算符和算术表达式

表达式描述了对哪些数据、以什么顺序施以什么样的操作。表达式由运算符与运算量组成，运算量可以是常量，变量，还可以是函数，例如， $a+3$ ， $t+\sin(a)$ ， $x=a+b$ ， $\text{Pi}*r*r$ 等都是表达式。

C 语言提供了丰富的运算符，能构成多种表达式。同时，它把许多基本操作都作为运算符处理，这就使得 C 语言表达式处理问题的范围宽、功能强。例如，在 C 语言中，可以在一个算术表达式中出现一个赋值表达式， $3+(a=5)*6$ ，它先执行括弧内的赋值运算，赋值表达式的值为 5，然后乘以 6 加上 3，得到整个表达式结果为 33；变量 a 的值为 5。

2.6.1 二元算术运算符

C 语言中有 5 个二元算术运算符，它们是 $+$ ， $-$ ， $*$ ， $/$ ， $\%$ ，其功能分别是加、减、乘、除、求余。这几个运算符的优先级为： $*$ ， $/$ ， $\%$ 同级， $+$ ， $-$ 同级但低于 $*$ ， $/$ ， $\%$ ，它们的结合方向均为“自左至右”。其中， $+$ ， $-$ ， $*$ ， $/$ 这 4 种运算符已为我们所熟悉。但要注意，除法运算根据运算量的不同效果也不同：若除数和被除数均为整数，则结果只取整数部分；若运算量中有一个为实数，则结果是双精度型实数。

求余运算符 $\%$ 用来求两个整数相除的余数，例如： $3\%2$ ，其值为 1； $8\%3$ ，其值为 2。

求余运算的一般规则是：假设两个整数分别为 a 和 b ，则： $a\%b=a-(a/b)*b$ 。例如：

$-19\%4=-19-(-19/4)*4=-19-(-4)*4=-3$

$19\%(-4)=19-(19/(-4))*(-4)=19-(-4)*(-4)=3$

$\%$ 要求运算量必须为整型，否则会出现错误信息。

C 语言中不含乘方运算符，不能直接进行乘方运算。所以在需要进行乘方运算时，可用连续相乘实现。

2.6.2 一元算术运算符

一元算术运算符指该运算符只能连接一个运算量，除正负号以外，还有 $++$ （加 1）、 $--$ （减 1）。加 1 或减 1 运算是对变量增加 1 或减少 1 的运算，也称为自加或自减运算。例如： $++i$ 表示 $i=i+1$ ； $--i$ 表示 $i=i-1$ 。

关于加 1 和减 1 运算符的说明：

(1) $++$ 或 $--$ 可以写在变量之前（称为前缀），也可以写在变量之后（称为后缀）。如果单独对一个变量施加前缀或后缀运算，其运算结果是相同的，例如设 $i=5$ ，则 $i++$ ；及 $++i$ ；运算后 i 的值均为 6。

(2) 如果对变量施加了前缀或后缀运算，并参与其他运算，则前缀运算是先改变变量的值再参与运算，而后缀运算是先参与运算后再改变变量的值。 $++i$ 表示先令 $i=i+1$ ，然后取 i 的值。 $j++$ 表示先取 j 的值，然后令 $j=j+1$ 。例如，令 $i=5$ ， $j=5$ ，则： $x1=++i$ ；执行后 $x1$ 的值为 6， i 的值也是 6； $x2=j++$ ；执行后 $x2$ 的值为 5， j 的值是 6。

例 2.5 前缀运算和后缀运算应用举例。



```
#include <stdio.h>
main()
{
    int a=10;
    printf("%d",a);
    printf("%d",++a);
    printf("%d",a++);
    printf("%d\n",a);
}
```

程序运行结果为:

```
10,11,11,12
```

(3) 自加和自减运算符作前缀使用时的结合方向是“自右至左”，作后缀使用时的结合方向是“自左至右”，运算对象只能是整型变量而不能是表达式或常数。例如： $5++$ 或 $(x+y)++$ 是错误的。另外，像 $a=b+++c$ ；应该理解为 $a=(b++)+c$ ，还是 $a=b+(++c)$ 呢？实际上在 C 编译处理时，从左至右尽量多地将若干个字符组成一个运算符（对标识符也如此），因此 $b+++c$ 被处理成 $(b++)+c$ 而不是 $b+(++c)$ 。但是像 $b+++c$ 这样的写法看起来不容易理解，应该避免使用，可以采用加括号的方法明确运算顺序。

2.6.3 算术表达式

用算术运算符连接数值型的运算量而得到的式子，就是算术表达式。例如：

```
a+(b++)*c
2+3+(5*sizeof(int))/345
5*3+6%4/2-1
(a+b)/(a-b)
(i++)+j
y+(x++)-(++i)
```

进行算术表达式运算时，要注意算术运算符的优先级和结合性。从表 2-5 中我们可以看到，一元运算符属于同一优先级，它们优先于二元运算符；二元运算符中， $*$ ， $/$ ， $%$ 属于同一优先级； $+$ ， $-$ 属于同一优先级。一元运算符的结合性为从右到左，二元运算符的结合性为从左到右。

例如，表达式 $3*5/4*2$ 中，同时出现了 3 个同一优先级的运算符，运算次序由结合性决定，即从左到右，该表达式的计算相当于： $((3*5)/4)*2$ 。又如，表达式 $-i++$ ，由于 $-$ 和 $++$ 属于同一优先级，其结合性为从右到左，因此 $-i++$ 相当于 $-(i++)$ 。

2.7 赋值运算符和赋值表达式

2.7.1 赋值运算符

在 C 语言中，通常把“ $=$ ”称为赋值号，也叫赋值运算符。它是一个二元运算符，需要连接两个运算量：左边必须是变量或数组元素，右边则是表达式。

除了“=”之外，C语言还提供了10种复合赋值运算符：

`+=` `-=` `*=` `/=` `%=` `<<=` `>>=` `&=` `|=` `^=`

其中，前5种是常用的算术运算，后5种适用于位操作。复合赋值运算符实际上是算术运算符与赋值运算符的合成或简化，例如：

<code>x+=e</code>	等价于 <code>x=x+e</code>
<code>x-=e</code>	等价于 <code>x=x-e</code>
<code>x*=e</code>	等价于 <code>x=x*e</code>
<code>x/=e</code>	等价于 <code>x=x/e</code>
<code>x%=e</code>	等价于 <code>x=x%e</code>

上述式子中的 `e` 表示表达式，使用复合赋值运算符连接两个运算量时，要把右边的运算量视为一个整体。例如：“`x*=y+5`”表示“`x=x*(y+5)`”，而不是“`x=x*y+5`”。

各种赋值运算符都属于同一优先级，且优先级仅比逗号运算符高，比其他所有运算符都低，其结合性为从右到左。

自加和自减运算符`++`、`--`是复合赋值运算符中的更特殊的情况，它们分别相当于`+=`和`-=`。

综上所述，凡赋值运算符及其变种（包括复合赋值运算符、自加自减运算符和正负号运算符）的结合方向都是自右至左的。

C语言允许在一个表达式中使用一个以上的赋值类运算符（包括赋值运算符、复合赋值运算符、自加、自减运算符等），使程序简洁，提高了程序设计效率，但同时也造成了阅读与理解程序的困难。所以用户应该有限制地使用复合赋值运算符；或者用圆括号加以说明。

复合赋值运算所引起的副作用表现在不易理解和结果不确定两个方面。

（1）不易理解

例如：“`c=b*=a+2`”容易误解为“`b=b*a+2; c=b+2;`”。为了使表达式清晰易懂，可以采用一些措施将费解处分解。例如将表达式“`c=b*=a+2`”分解为：“`a=a+2; b=b*a; c=b;`”或“`c=a+2; c=b*c;`”。将表达式“`x=(i++)+j`”分解为：“`x=i+j; i++;`”。

C语言的运算符丰富，但运算规则复杂、难记。为了避免出错，可以加一些“不必要”的括号表示优先顺序。例如，“`c=b*=a+2`”改写为：“`c=b*=(a+2);`”或“`c=(b*=(a+2));`”便清晰了。之所以称为“不必要”，是指不加这些括号计算机也不会算错，但是人容易理解错。

另外，注释是提高程序清晰性的有力工具。对C语言来说，注释不会影响程序的效率，又可以帮理解程序。例如：`c*=b=a+2;`可以写为：`c*=b=a+2; /* b=a+2; c=c*b */。`

（2）结果不确定

例如：

```
j=3;
i=(k=j+1)+(j=5);
```

执行程序时，不同机器上得到 `i` 的值可能是不同的。有的机器先执行 `(k=j+1)`，然后再执行 `(j=5)`，`i` 值得 9。而有的机器是先执行 `(j=5)`，后执行 `(k=j+1)`，`i` 的值就成了 11。



在 PC 机 Turbo C 2.0 环境下, i 的值为 9。

这是因为在数学中, $a+b$ 和 $b+a$ 是一样的, $(a+b) + (c+d)$ 也可以写成 $(c+d) + (a+b)$ 。换言之, $(a+b) + (c+d)$ 的求值顺序不影响结果 (可以先求 $a+b$, 也可以先求 $(c+d)$)。但在 C 语言中, 由于运算符类型丰富, 尤其是赋值类运算符的影响, 使交换律不再适用于 C 语言中的运算。且 C 语言对表达式的求值顺序 (方向) 无统一规定, 而是由各个 C 编译系统自己决定, 这就造成了同一程序在不同计算机系统中运行时会得到不同的结果。

为了提高程序的可移植性, 应当使表达式分解, 使之在任何机器上运行都能得到同一结果。因此上面语句可改为:

```
j=3;    k=j+1;
j=5;    i=k+j;
```

2.7.2 赋值表达式

用赋值运算符连接两个运算量就得到赋值表达式。例如, $d=a+b-c$ 、 $d=a=3$ 就是赋值表达式, 其右边的运算量是表达式。其前者是将算术表达式赋值给变量 d , 后者将赋值表达式再赋值给变量 d 。根据赋值运算符从右到左的结合性, 在表达式 $d=a=3$ 中, 先计算表达式 $a=3$, 变量 a 的值为 3, 该表达式的值也是 3, 再赋值给 d , 最后得到 d 的值也是 3。在赋值表达式中, 被赋予变量的值就是赋值表达式的值, 该赋值表达式的值也是 3。

赋值表达式可以出现在表达式可以出现的任何地方, 例如作为 `printf` 函数的输出项、算术表达式中的一个运算量, 还可以作为赋值运算符的一个运算量组成新的赋值表达式。如:

```
x=y=z=6          /* z=6,y=z,x=y */
j+=i--=1         /* i=i-1,j=j+i */
a+=a-=a*a        /* a=a-a*a,a=a+a */
k=5+(c=6)        /* c=6,k=5+c */
```

例 2.6 根据一个学生的 3 门课程成绩计算平均分

```
#include<stdio.h>
main()
{
    float score1,score2,score3;
    float aver;
    score1=78.9;
    score2=90;
    score3=87;
    aver=(score1+score2+score3)/3;
    printf(" %f ",aver);
}
```

2.8 逗号运算符和逗号表达式

2.8.1 逗号运算符

C 语言中, 逗号既作为分隔符, 也作为运算符。逗号用作分隔符时, 可将若干数据项

分开，主要用于分隔变量定义时的多个变量和函数形式参数表中的形参。如：

```
int a,b,c;  
printf("%d,%d,%d",a,b,c);  
void sub(int a,float b,double c);
```

其中的逗号作为分隔符使用。

逗号用作运算符时，是将若干个独立的表达式隔开，以依次计算其中各个表达式的值。例如“a=3*5,a*4,a+5”中的逗号是运算符，它将一个赋值表达式和两个算术表达式连接起来，构成一个新的表达式，其功能是从左到右依次计算 a=3*5 以及 a*4 和 a+5 的值。

逗号运算符亦称顺序运算符，它是一个二元运算符，在所有的 C 运算符中，它的优先级是最低的。如果若干个逗号同时出现在表达式中，其结合性为从左到右。

2.8.2 逗号表达式

用逗号运算符将几个表达式连接在一起，就构成逗号表达式。逗号表达式的值是逗号表达式中最右边的一个表达式的值。例如，设 a 是 int 型变量，则“a=4*6, a+2”中第一个表达式的值是 24，第二个表达式的值是 26，那么整个逗号表达式的值也是 26。

逗号的优先级是最低的，如果有“b=a=3*5, a+1”，则 a 的值为 15，b 的值为 15，整个表达式的值为 16。若改写成“b= (a=3*5, a+1)”，则 a 的值为 15，a+1 的值为 16，括号中的逗号表达式的值为 16，所以 b 的值为 16。这是因为表达式是赋值表达式，它把逗号表达式 (a=3*5, a+1) 的值 16 赋给变量 b。由于逗号的优先级低于赋值号，需要加括号把右边作为一个表达式。

例 2.7 逗号表达式。

```
#include <stdio.h>  
main()  
{  
    int a,b;  
    printf("a=%d,exp=%d\n ",a,(a=3*5,a+1));  
    printf("a=%d,b=%d,exp=%d\n ",a,b,(b=a=3*5,a+1));  
    printf("a=%d,b=%d,exp=%d\n ",a,b,b=(a=3*5,a+1));  
}
```

运行结果为：

```
a=15,exp=16  
a=15,b=15,exp=16  
a=15,b=16,exp=16
```

需要注意的是，Turbo C 规定，printf() 函数中出现多个表达式输出项时，先按从右到左的顺序计算各表达式的值，然后再显示结果；而不是从左到右计算结果，也不是每计算一个表达式就输出结果。



2.9 关系运算符和关系表达式

2.9.1 关系运算符

关系运算符有 6 种符号，即>，<，>=，<=，==和!=，分别表示大于、小于、大于或等于、小于或等于、等于和不等，见表 2-6。

表 2-6 关系运算符

优先级	运算符	功能	运算量	结合性
14	!	逻辑非	1	从右至左
10	>	大于	2	从左至右
	<	小于		
	>=	大于等于		
	<=	小于等于		
9	==	等于	2	从左至右
	!=	不等于		

关系运算符主要用来对两个算术表达式或赋值表达式进行比较运算，例如：

```
a+b>=c-d
x==y
score>90
```

关系运算的结果是一个逻辑值，即若比较成立得逻辑真 True，否则得逻辑假 False。C 语言中，逻辑值用整数 0 和 1 表示：1 代表逻辑真，0 代表逻辑假。如：

```
0>9          其值为 0
100!=20      其值为 1
```

2.9.2 关系表达式

用关系运算符连接两个算术表达式或赋值表达式，就构成了关系表达式。

关系运算符的优先级低于算术运算符，高于赋值运算符。因此， $x+y>=a+b$ 与 $(x+y)>=(a+b)$ 等价，而 $i=(c=b)$ 则与 $i==c=b$ 不等价。关系运算符结合性都是从左至右。

用“==”或“!=”连接两个关系表达式就构成判断相等表达式。在判断相等表达式中，先计算两个关系表达式的值，然后进行相等性检查。例如，“ $x>y!=y<z$ ”，先计算 $x>y$ 和 $y<z$ ，得到两个逻辑值，再检查它们是否满足不等性条件。

例 2.8 关系表达式。

```
#include <stdio.h>
main()
```

```
{
    int a=5,b=3;
    float x=1.23,y=4.56;
    printf("%d,%d\n",a+b!=a-b,x<=(y-=7.8));
}
```

程序运行结果为:

1,0

2.10 逻辑运算符和逻辑表达式

“关系”是指某个值与其他值的关系，通常包括比较大小、是与不是、真与假等；“逻辑运算”是把关系运算连接起来。在 C 语言中，关系运算和逻辑运算常与后续章节介绍的条件语句、开关语句、for 语句或 while 语句等连用。在程序设计中，关系运算和逻辑运算往往一起使用。

2.10.1 逻辑运算符

逻辑运算符有 3 种，即!、&&和|，分别表示逻辑非、逻辑与、逻辑或。逻辑运算符主要用来对关系运算进行逻辑连接。例如，数学不等式“ $0 \leq x \leq 9$ ”必须用逻辑表达式：“ $0 \leq x \&\& x \leq 9$ ”或“ $x \geq 0 \&\& x \leq 9$ ”表示，而不能写成“ $0 \leq x \leq 9$ ”。

逻辑运算符的运算量和运算结果都是逻辑量，逻辑量用 1 表示真，用 0 表示假。逻辑运算符的功能可用如表 2-7 所示的真值表来表示。

表 2-7 逻辑运算真值表

P	q	p&&q	p q	!p
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

运算量也可以是任意数值量，但是作为逻辑量来处理，非 0 值算做真，0 值算做假，逻辑运算的运算结果是逻辑值 0 或 1，0 表示假，1 表示真。例如：

25&&-3.5	其值得 1
0 356	其值得 1
!(-23)	其值得 0

2.10.2 逻辑表达式

用逻辑运算符连接关系表达式就构成逻辑表达式。一个逻辑表达式中，允许出现若干个逻辑运算符和关系表达式，此时需要按照优先级和结合性进行计算。逻辑运算符&&和|的优先级低于关系运算符，高于赋值运算符（参见表 2-5），结合性是从左至右。



值得注意的是，任何关系表达式和逻辑表达式的结果，其值只有 0 和 1 两种可能。另外，运用下述规律可以简化逻辑表达式的运算：

- 在一个&&表达式中，若&&的一端为 0，则不必再计算另一端的值，表达式的结果恒为 0；
- 在一个||表达式中，若||的一端为 1，则不必再计算另一端的值，表达式的结果恒为 1。

这样的运算规律称为短路运算。例如：在表达式 $a+1 \&\& b++ \mid c++ == 0$ 中，有算术运算符、关系运算符和逻辑运算符。若 $a=0$ ， $b=1$ ， $c=2$ ，按运算规则和运算符的优先级分析出运算顺序和结果见图 2-1（图中框起来的部分表示运算被短路，即不执行）；若 $a=-1$ ， $b=1$ ， $c=2$ ，按运算规则和运算符的优先级分析出运算顺序和结果见图 2-2。

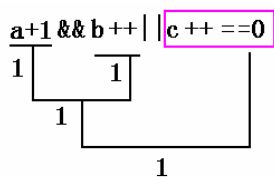


图 2-1 逻辑表达式的运算顺序

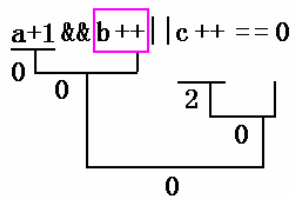


图 2-2 逻辑表达式的运算顺序

例 2.9 逻辑表达式。

```
#include<stdio.h>
main()
{
    int a=-1,b=1,c=2;
    printf("\n %d",a+1&& b++ | c++==0);
    printf("%d%d%d",a,b,c);
}
```

输出结果为：

```
0  -1  1  3
```

如果变量定义语句改为 “`int a=0,b=1,c=2;`”，则输出为：

```
1  0  2  2
```

例 2.10 逻辑表达式。

```
#include <stdio.h>
main()
{
    char c;
    int i,j,k;
    c='3';
    i=5,j=7,k=0;
    printf("%d,%d,%d,%d\n",i&&j,i&&k,i||j,i||k);
    printf("%d,%d\n",c>='0'&&c<='9',i<=j-1||c!=' ');
}
```

程序运行结果为:

```
1,0,1,1  
1,1
```

2.11 测试数据长度运算符 sizeof

C 语言中各种类型的数据占用多大的存储空间,与宿主机器的硬件特性有关。为了能够确定某一种类型数据的长度,C 语言提供了测试数据长度运算符 `sizeof`,它的一般格式为:

```
sizeof(exp)
```

其中, `exp` 可以是类型关键字、变量或表达式。`sizeof` 运算符的功能是给出 `exp` 所占用的内存字节数,例如: `sizeof(double)`, `sizeof(x)`, `sizeof(a+b)`, `sizeof(3*1.46/7.28)`等。

`sizeof` 是一元运算符,它的优先级与++, --, &, ~等相同,结合性为从右至左。下面举一个例子说明 `sizeof` 的作用,功能是给出 Turbo C 中各种数据类型的长度。

例 2.11 测试数据长度运算符 `sizeof()` 的使用。

```
#include <stdio.h>  
main()  
{  
    char ch;  
    float a=1.28f,b=3000.0f;  
    printf("char:%d\n",sizeof(char));  
    printf("short int:%d int:%d long int:%d\n",  
           sizeof(short int),sizeof(int),sizeof(long int));  
    printf("float:%d    long float:%d\n",sizeof(a),sizeof(long float));  
    printf("double:%d    long double:%d\n",sizeof(double),  
           sizeof(long double));  
    printf("float express:%d\n",sizeof(a+b));  
    printf("constant express:%d\n",sizeof(3*5/7.0f));  
}
```

程序运行结果为:

```
char:1  
short int:2    int:2    long int:4  
float:4 long float:8  
double:8      long double:10  
float express:8  
constant express:8
```

2.12 不同类型数据间的转换与运算

对于 C 表达式类型和求值规则,主要讨论算术表达式运算时的数据类型转换和赋值表达式赋值时的数据类型转换。



2.12.1 算术表达式运算时的数据类型隐式转换规则

算术表达式的数据类型就是该表达式的值的类型。由于算术表达式允许不同类型的数据混合运算，故其类型的确定遵循以下自动类型转换规则。

- 对一元运算符而言，因为只有一个运算量，故表达式的类型就是运算量的类型。
- 对二元运算符而言，有两个运算量参加运算，则表达式的类型确定方法为：
 - ❶ 若两个整型 (int) 运算量参加运算，则结果也是整型的。初学者要特别注意，两个整型数相除，其值也是整型数（即商的整数部分）。
 - ❷ 若不是两个整型的运算量参加运算，则 C 编译系统自动对它们进行转换，一般规则是把精度低的类型转换为精度高的类型，以保证不丢失精度。具体遵循如下规则：
 - ◆ 所有的 char 和 short int 都转换成 int，所有的 float 都转换成 double；
 - ◆ 如果其中的高精度数是 unsigned，则另一个操作数转换成 unsigned。
 - ◆ 如果其中的高精度数是 long，则另一个操作数转换成 long；
 - ◆ 如果其中的高精度数是 double，则另一个操作数转换成 double；
 - ◆ 如果其中的高精度数是 long double，则另一个操作数转换成 long double；

2.12.2 显式类型转换

当算术表达式中需要违反自动类型转换规则，或者说自动类型转换规则达不到目的时，可以使用强制类型转换，这叫做显式类型转换，一般形式为：

(类型标识符) 表达式

显式类型转换也是一种一元运算符，且与其他一元运算符具有相同的优先级，它的功能是将指定的表达式强制转换成指定的类型。例如：

(double)a 强制将变量 a 转换成 double 型
(int)(x+y) 强制将表达式(x+y)转换成 int 型
(int)x+y 强制将变量 x 转换成 int 型，然后与 y 相加

显式类型转换有时非常有用。例如，若 i 为整型变量，表达式(i/2)只能得到整数，而用(float)i/2 就能得到小数。又如，设 x 是 float 型变量，表达式 x%3 是错误的，用(int)x%3 就允许了。后面将要介绍的数组，当用 float 型或 double 型变量作下标时，也需要使用强制类型转换。还有，C 编译系统提供的数学函数大多数是 double 型的并且要求参数为 double 型，在调用这些函数时，可以用显式转换方法进行参数的类型转换或将函数值转换成需要的类型。例如：double cos((double)i)，double sqrt((double)i)，double exp((double)i) 等。

使用显式类型转换应注意以下几点：

- 在进行显式类型转换时，类型关键字必须用括号括住。例如(int)x 不能写成 int x。
- 在对一个表达式进行显式类型转换时，整个表达式应该用括号括住。例如，(float)(a+b)若写成(float)a+b，就只对变量 a 进行显式类型转换。

- 在对变量或表达式进行了显式类型转换后，并不改变原变量或表达式的类型。例如，设 x 为 float 型，y 为 double 型，则(int)(x+y)为 int 型，而 x+y 仍然是 double 型。
- 将 float 型或 double 型强制转换成 int 型时，对小数部分是四舍五入还是简单地截断，取决于具体的系统。Turbo C 采用的是截断小数的办法。

例 2.12 显式类型转换运算符的使用。

```
#include <stdio.h>
main()
{
    float x=2.0f;
    double y=3.3;
    printf("%d    %f\n", (int)(x*y), x*y);
}
```

程序运行结果为：

```
6          6.600000
```

2.12.3 赋值表达式的类型及赋值时的数据类型转换

赋值表达式的类型就是被赋值的变量或数组元素的类型。当赋值运算符两边的数据类型不一致时，C 编译系统自动将赋值运算符右边的数据类型转换成与左边变量或数组元素相同的类型。这种转换可分为两类：一类是将低精度类型转换成高精度类型，其转换规则见表 2-8；另一类是将高精度类型转换成低精度类型，其转换规则见表 2-9。

表 2-8 低精度类型向高精度类型转换规则

低精度类型	高精度类型	转换规则
char	unsigned char	忽略 char 的符号位
char	int	将 char 的 8 位存入 int 的低 8 位中，高 8 位符号扩展
unsigned char	int	将 char 的 8 位存入 int 的低 8 位中，高 8 位补 0
Int	unsigned int	忽略 int 的符号位
int	long int	将 int 的 16 位存入 long 的低 16 位中，高 16 位符号扩展
unsigned int	long int	将 unsigned 的 16 位存入 long 的低 16 位中，高 16 位补 0
int	float	将 int 型用浮点表示
float	double	将 float 的尾数和阶码分别存入 double 的尾数和阶码

注：符号扩展是指若低精度数据的最高位为 1，则将低精度数据存入高精度变量的低位部分，而高位部分全补 1；若低精度数据的最高位为 0，则将低精度数据存入高精度变量的低位部分，而高位部分全补 0。



表 2-9 高精度类型向低精度类型转换规则

高精度类型	低精度类型	转换规则
unsigned char	char	使符号位有效
int	unsigned char	截去 int 的高 8 位，低 8 位按无符号数处理
int	char	截去 int 的高 8 位
unsigned int	int	使符号位有效
long int	int 或 char	截去 long int 的高位
long int	unsigned int	截去 long int 的高位
float	int	截去 float 的小数部分
double	float	将 double 的多余小数部分四舍五入

对于表 2-8 和表 2-9 中未标明的类型转换，可逐步转换，直到完成为止。例如要从 double 转换成 int，可先将 double 转换成 float，再由 float 转换成 int。

例 2.13 低精度类型转换成高精度类型。

```
#include <stdio.h>
main()
{
    int a=-1;
    unsigned int b;
    b=a;
    printf("%d    %u\n",a,b);
}
```

程序运行结果为：

```
-1    65535
```

我们可以看到，a 的值没有发生变化，b 的值为 65535。这是因为在 Turbo C 中，int 型变量 a 为负数时，按补码存储，其二进制存储格式为 16 个 1，即：

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

将它转换成 unsigned int 型后，将最高位的符号位也视为数值位，所以其值为 65535。

例 2.14 高精度类型转换成低精度类型。

```
#include <stdio.h>
main()
{
    int i=8808;
    char ch;
    ch=i;
    printf("%d    %c\n",i,ch);
}
```

程序运行结果为：

```
8808  h
```

这是因为 `int` 型变量 `i` 占 16 位，其中存放数据 8808，二进制存储格式为：

0	0	1	0	0	0	1	0	0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

截去高 8 位后，余下的低 8 位的值是 104，它代表字符 `h`。

要 点 回 顾

1. 计算机内部数据是以二进制形式存放的。人们为了方便，经常采用八进制或十六进制的形式表示一个数，且习惯使用十进制形式来输入输出数据。计算机中的带符号数都是用补码形式表示的。一个正整数的补码与它的原码相同，一个负整数的补码则是把它的原码的各位（符号位除外）求反然后在最低位加 1，例如： $(-5)_{\text{补}} = 0\text{xffb}$ 。

2. C 语言的基本数据类型有：

- `char` 型（字符型） 数据占 1 字节存储空间，表示字符时代表 ASCII 码。
- `int` 型（整型） 又分为短整型（`short`）、普通整型（`int`）、长整型（`long`）3 种。`char` 和 `int` 型还可以表示为有符号（`signed`）或无符号（`unsigned`）。IBM PC 系列计算机以定点二进制补码形式存储有符号数，一般以 2 个字节（16 位）存放短整型和普通整型数，以 4 个字节（32 位）存放长整型数。最常使用的是普通整型。
- `float` 型（实型） 数据占 4 字节存储空间，以浮点形式存储。
- `double` 型（双精度型） 数据占 8 字节存储空间，存储方式与 `float` 基本相同。

3. C 语言并不规定各种类型的数据占用多大的存储空间，它与宿主机器的硬件特性有关。为了确定某一种类型数据的长度，可以利用测试数据长度运算符 `sizeof`，它的一般格式为 `sizeof(exp)`，其中 `exp` 可以是类型关键字、变量或表达式，其功能是给出 `exp` 所占用的内存字节数。

4. 整型常量可以用十进制、八进制和十六进制来表示。C 语言规定，以 1~9 开头的表示十进制数；以 0 开头的数字，表示八进制数；以 0x 开头的数字表示十六进制数。实型常量只能用十进制形式表示，也可以用小数值形式或指数形式表示。字符常量以单引号定界，占 1 个字节存储单元，在内存中以相应的 ASCII 代码存放；字符串常量以双引号定界，占用一段连续的存储单元。

5. C 程序中使用到的任何变量都必须在使用前进行定义。变量定义的一般格式为：

```
[存储类别] 数据类型 变量清单；
```

6. C 语言的运算符也很丰富，在学习运算符时应注意运算符的功能、运算量的关系、运算的优先级别、结合性以及表达式值的类型。

7. `++`或`--`可以写在变量之前（称为前缀），也可以写在变量之后（称为后缀）。如果单独对一个变量施加前缀或后缀运算，其运算结果是相同的；如果对变量施加了前缀或后缀



运算，并参与其他运算，则前缀运算是先改变变量的值再参与运算，而后缀运算是先参与运算后改变变量的值。自加和自减运算符作前缀使用时的结合方向是“自右至左”，作后缀使用时的结合方向是“自左至右”，它的运算对象只能是整型变量而不能是表达式或常数。

8. 当含有不同类型的数据时，算术表达式运算的数据类型转换默认按隐式类型转换，即从精度低的类型自动转换成精度高的类型；也可以按显式类型转换，一般形式为：

(类型标识符)表达式

9. 关系表达式和逻辑表达式的结果都是逻辑量，用“1”表示真，用“0”表示假。逻辑运算的运算量可以是任意数值量，在作为逻辑量来处理时，非 0 值算做真，0 值算做假。

习 题

- 在计算机内部，一切信息的存取、处理与传送均采用（ ）。
A) 二进制 B) BCD 码 C) 十六进制 D) ASCII 码
- 以下选项中属于 C 语言的数据类型是（ ）。
A) 复数型 B) 逻辑型 C) 双精度型 D) 集合型
- 在 C 语言中，不正确的 int 类型的常数是（ ）。
A) 32768 B) 0 C) 037 D) 0xAF
- 语句：printf("%d", (a=2)&&(b=-2)); 的输出结果是（ ）。
A) 无输出 B) 结果不确定 C) -1 D) 1
- 若 a 为 int 类型，且其值为 3，则执行完表达式 $a+=a-=a*a$ 后，a 的值是（ ）。
A) -3 B) 9 C) -12 D) 6
- 设 x, y, t 均为 int 型变量，则执行语句：x=y=3; t=++x||++y; y 的值为（ ）。
A) 不定值 B) 4 C) 3 D) 1
- 以下运算符中优先级最高的是（ ）。
A) ++ B) ?= C) != D) &&
- 已知各变量的类型定义如下：

```
int k,a,b;  
unsigned long w=5;  
double x=1.42;
```

则以下表达式中不符合 C 语言语法的是（ ）。

- A) $x \% (-3)$ B) $k=(a=2,b=3,a+b)$ C) $w+=-2$ D) $a+=a-=b=4$
- 表达式 $18/4*\text{SQRT}(4.0)/8$ 的值的数据类型是什么，其值是多少？
- 求下列算术表达式的值：
设 $a=2, b=3, x=3.5, y=2.5$ ，则 $(\text{float})(a+b)/2+(\text{int})x\%(\text{int})y$ 的值为_____。
设 $x=2.5, a=7, y=4.7$ ，则 $x+a\%3*(\text{int})(x+y)\%2/4$ 的值为_____。
- 设 a 为整形变量，初值为 3，求下列表达式的值。
A) $a+=a$ B) $a-=2$

- C) $a*=2+3$ D) $a/=a+a$
12. 写出下列各逻辑表达式的值 (设 $a=3$, $b=4$, $c=5$)。
- A) $a+b>c\&\&b==c$ B) $a\|b+c\&\&b-c$
- C) $!(a>b)\&\&!c\|1$ D) $!(x=a)\&\&(y=b)\&\&b+c/2$
- E) $c>b>a$



第 3 章 C 语言的基本语句与输入输出

本章的学习目标:

了解 C 语言的语句格式, 掌握输入/输出函数的用法、输入/输出函数中格式控制符和输出项的对应关系, 能够实现格式输出; 利用基本语句和输入/输出函数编写简单程序。

3.1 基本语句

程序是对计算机要执行的一组操作序列的描述, 高级语言源程序的基本组成单位是语句。语句按功能可以分为两类: 一类用于描述计算机要执行的操作运算(如变量定义语句、赋值语句等), 另一类是控制上述操作运算的执行顺序(如条件控制语句、循环控制语句等)。前一类称为操作运算语句, 后一类称为流程控制语句。

3.1.1 表达式语句

C 语言是一种表达式语言, 所有的运算操作都通过表达式来实现。由表达式组成的语句称为表达式语句, 它由一个表达式后接一个分号组成, 其中分号是语句结束的标志。

表达式语句可以分为以下几种类型:

表达式语句 { 赋值语句
函数调用语句
逗号表达式语句

(1) 简单的赋值语句由赋值表达式加一个分号组成。例如:

```
i=1;  
x=sin(y);
```

赋值语句可以对表达式进行计算, 然后把计算结果赋给指定的变量, 其基本格式为:

```
v=e;
```

其中, v 是变量名, e 是表达式。赋值语句的功能是先计算表达式 e 的值, 并把 e 的值转换成和 v 相同的数据类型后赋给变量 v 。

例 3.1 编程对两个数求和, 程序如下:

```
#include<stdio.h>  
main()  
{  
    int a,=0,b=1,sum;  
    sum=a+b;
```

```
printf("sum=%d\n",sum);  
}
```

程序运行结果如下：

```
sum=1
```

(2) 函数调用语句由函数调用表达式后跟一个分号组成。例如：`printf("this is an example");`

(3) C 语言还允许把几个表达式组合在一起，形成一个逗号表达式语句。组合的方法是，用逗号作为表达式间的分隔符，最后用分号结尾。例如：

```
++i, --j;  
i=1, j=2;
```

(4) `exit()`：该语句是一个特殊的表达式语句，使程序停止执行。

`exit()`是标准库(<stdlib.h>)中的一个函数。它的作用是立即停止当前程序并退回到操作系统状态。它也常常被作为一个特殊的表达式语句来控制程序停止执行。

使用 `exit()`函数，应在程序前部使用以下的预编译命令：

```
# include <stdlib.h>
```

`exit()`是带参数调用的，参数是 `int` 型：参数为 0 时，说明这个停止属正常停止；参数为其他值时，说明程序非正常退出。

3.1.2 空语句

空语句是只有一个分号而没有表达式的语句。其形式为：

```
;
```

它不产生任何实际操作运算，只作为形式上的语句，被填充在控制结构之中。例如：

```
for(i=0; i<1000; i++)  
;
```

空语句用作循环语句的循环体，表示什么也不做。事实上，这个循环的功能是延迟一小段时间。有时空语句也被用作转向点。

3.1.3 复合语句

C 语言允许把一组语句括在一对花括号之中，成为复合语句。例如：

```
{  
    c=getchar();  
    putchar( c );  
}
```



注意

一个复合语句的后花括号之后不需再写分号。若复合语句的后花括号之后写有分号，该分号为一个空语句，对程序的执行不产生任何影响。

复合语句在语法上是一个整体，相当于一个语句。一个复合语句中又可以包含另一个



或多个复合语句。凡是能使用简单语句的地方，都可以使用复合语句。复合语句中也可以定义变量，例如：

```
{
    int x;
    {
        int y;
        y=3;
        printf("%d\n",x+y);
    } /*复合语句（分程序）*/
}
```

复合语句又称为分程序，在分程序中定义的变量（如上面的 y）只能在本分程序中可用。

3.2 流程控制语句

前面所有语句是按照其在程序中出现的先后顺序执行的。实际上，程序执行的结果不仅与数据和计算公式有关，还与执行的顺序有关。对同样的一组表达式语句，以不同的流程执行，可以得到不同的结果。比如，对同样的数据，先乘除后加减与先加减后乘除得到的结果是不同的。所以，设计程序时不仅要设计合适的操作，而且要构造适当的流程，即执行顺序。高级语言一般以两种形式进行流程控制：

- 形成流程控制结构（如 if, switch, while, do-while, for, break, continue 语句）。这些语句能够实现程序的判断选择或者是循环执行。
- 简单的流程转向，如 return, goto 语句和语句标号的使用。

3.3 数据的输入与输出

对数据的一种重要的操作是输入和输出。因为程序在多次运行时，用到的数据可能是不同的，此时应该在程序运行时，允许用户临时输入所需的数据，以提高程序的通用性和灵活性。而且程序的运行结果要输出到外设。

C 语言的输入输出由函数来实现。C 语言提供了多种输入输出函数，使输入输出灵活、多样、方便、功能强。标准 I/O 函数库中，有一些公用的信息写在头文件 `stdio.h` 中，因此要使用标准 I/O 函数库中的 I/O 函数时，一般应在程序开头先写下面的命令：`#include <stdio.h>`，以便把 I/O 函数要使用的信息包含到程序中来。

输入输出的对象是数据，而数据是以介质为载体的，所以进行输入输出操作就要与各种外部设备发生联系，要指出从哪个设备（文件）读入数据，将数据输出到哪个设备（文件）上去。本节只讨论从标准输入设备（键盘）和输出到标准输出设备（显示器）的情况，并且只讨论使用得最广泛的 `scanf` 函数、`printf` 函数、`getchar` 函数和 `putchar` 函数。这 4 个函数在 `stdio.h` 中定义，所以程序开始要有 `#include<stdio.h>`，以便在编译连接时，将用户程序与标准库文件相连。

3.3.1 格式输出函数 printf()

格式输出函数 `printf()` 的功能是按照指定的格式，控制参数的值在标准输出设备（显示器）上的输出。`printf()` 的一般格式为：

```
printf("格式控制字符串",输出项目清单)
```

其中，`printf` 是函数名，其后的括号中是该函数的参数；“格式控制字符串”用双引号括住，用来规定输出格式；“输出项目清单”中包含零个或多个输出项，它们可以是常数、变量或表达式，相互之间用逗号隔开。例如：假设 `a` 和 `b` 是整型变量（`a=123,b=456`），`c` 是浮点型变量（`c=7.89`），则：

```
printf("a=%d,b=%d\nc=%f",a,b,c);
```

的输出结果为：

```
a=123,b=456
c=7.890000
```

1. 输出格式

输出格式由格式控制字符串加以规定，它关系到数据输出格局的安排。格式控制字符串由两种成分组成：格式说明符和普通字符。其中普通字符（包括转义符序列）将被简单地复制显示或执行（比如“`\n`”引起换行，“`a=`”原样输出）；而一个格式说明符将引起一个输出参数项的转换与显示，格式说明符是由“`%`”引出并以一个类型描述符结束的字符串，中间是一些可选的附加说明项。格式控制字符串可包括以下 3 种符号。

（1）格式转换说明符

最简单的格式转换说明符是在“`%`”后跟一个特定的字母，用来与输出项的数据类型相匹配：

<code>%d</code>	输出十进制整数（decimal）。
<code>%x</code>	按小写形式输出十六进制整数（hexadecimal）。
<code>%X</code>	按大写形式输出十六进制整数（hexadecimal）。
<code>%o</code>	输出八进制整数（octal）。
<code>%i</code>	输出十进制整型数（int）。
<code>%u</code>	输出无符号十进制整数（unsigned），把符号位视为数的一部分。
<code>%f</code>	输出浮点数，保留 6 位小数（float）。
<code>%g</code>	输出浮点数，保留 6 位有效数字（float）。
<code>%c</code>	输出单个字符（character）。
<code>%s</code>	输出一个字符串（string）。
<code>%%</code>	输出一个 <code>%</code> 。
<code>%e</code>	按指数格式 <code>[-]m.dddddde±nn</code> 输出浮点数（规格化表示）。
<code>%E</code>	按指数格式 <code>[-]m.dddddde±nn</code> 输出浮点数（规格化表示）。

（2）转义序列

转义序列由“`\`”后跟一个特定字母组成，用以输出某些特殊字符和不可见字符：



<code>\n</code>	回车换行 (Enter)。
<code>\t</code>	制表符 (Tab, 光标右移 8 位)。
<code>\a</code>	报警铃响 (alert)。
<code>\\</code>	倒斜杠(\)
<code>\"</code>	双引号(")

(3) 其他字符

除以上两项字符以外的其他字符都视为普通字符, 输出时照原样显示。

2. 说明

(1) 整个格式控制字符串必须用双引号括住。如果有输出项目, 则格式控制字符串与第一个输出项目之间用逗号隔开; 如果有多个输出项目, 各输出项目之间用逗号隔开。

(2) 格式转换说明符的个数应与输出项的个数相等, 且顺序和类型应一一对应, 例如:

```
printf("%d %d %d %d",x,y,z);
```

中, 4 个格式转换说明符对应 3 个输出项, 是错误的。如果 `x,y,z` 不是整型, 使用 “`%d`” 格式也是错误的, 虽然在编译时不出现错误信息, 但是可能得到错误结果。

(3) 将输出格式控制中的 3 种控制字符适当组合, 可形成不同的显示效果, 如表格、图案等。

(4) 格式转换说明符可以加下列修饰符:

① “`%m.nd`”, 其中 `m` 规定输出的字段总宽度, `n` 规定输出的最小有效位数。如果被输出量的实际位数小于 `m`, 则左端补空格; 如果被输出量的实际位数大于 `m`, 则按实际位数输出。如果被输出量的实际有效位数大于 `n`, 则 `n` 不起作用; 如果实际有效位数小于 `n`, 则左边补 0 直至 `n` 位。例如:

```
printf("%3d,%6d,%6.5d\n",1234,1234,1234);
```

输出结果为:

```
1234, 1234, 01234
```

② “`%m.nf`”, 其中 `m` 规定输出的字段总宽度, `n` 规定输出的小数位数。如果实际数据的整数位数小于 $(m-n-1)$, 则左端填充空格。如果实际数据的整数位数大于 $(m-n-1)$, 则 `m` 不起控制作用, 按实际数据应有的位数输出。如果不规定 `m` 和 `n`, 则 `m` 取被输出量的实际位数, `n` 取 6。例如:

```
printf("%10f,%10.3f,%.3f\n",123.456,123.456,123.456);
```

输出结果为:

```
123.456000, 123.456,123.456
```

③ “`%m.ne`”, 其中 `m` 规定输出的字段总宽度, $(n-1)$ 规定尾数部分的小数位数。例如:

```
printf("%10e,%10.4e,%.3e\n",123.456,123.456,123.456);
```

输出为规格化的指数形式, 即尾数部分的小数点前有一位整数。

```
1.23456e+02, 1.235e+02,1.23e+02
```

其中，“%10e”未指定 n ，系统自动按 $n=6$ 处理，即 5 位小数；因为输出的实际位数已超过 10 列，故输出 11 列；“%10.4e”指定 $m=10$ ， $n=4$ ，即输出总宽度为 10，小数 3 位，实际总位数为 9 位，所以左边留有一个空格；“%.3e”未指定 m ，则系统自动取 m 等于实际数据应占的长度，本例中为 8 位。

在上述 3 种情况下，如果在百分号与 $m.n$ 之间加一个 0，即写成 %0m.nd、%0m.nf 或 %0m.ne，则左边多余的空位用 0 而不是用空格填充。例如：

```
printf("%04d,%06d,%03d\n",1234,1234,1234);
```

输出结果为：1234,001234,1234

```
printf("%010.2f",123.456);
```

输出结果为：0000123.46

④ "%m.ns"，其中 m 规定输出的字段总宽度， n 规定只输出字符串中 n 个字符。若字符串长度小于 m ，则左边补充空格；若字符串长度大于 m ，则 m 不起作用；若字符串长度小于 n ，则 n 不起作用；若字符串长度大于 n ，则只输出左边 n 个字符，多余的字符被截断。例如：

```
printf("%10.5s,%10.5s\n","abcdefgh ", "abc");
```

输出结果为：

```
abcde,      abc
```

(5) 左对齐。在默认情况下，所有的输出均右对齐；如果指定的字段宽度大于实际位数，则左边填充空格或零。但在百分号后边加一个减号，如 %-m.nd、%-m.nf、%-m.ns 等，可使输出左对齐，并在右边填充空格。例如：

```
printf("%-10.2f,%-5.2f\n",123.456,1.23456e2);
```

输出为：

```
123.46      ,123.46
```

再例如下面 3 个语句：

```
printf("1234567890123456789012345678901234567890\n");
printf("%3s,%15s,%10.5s,%.8s\n","abc","fortran","C language","English");
printf("%-10s,%-5s,%-10.5s,%-.8s\n","abc","fortran","C language","English");
```

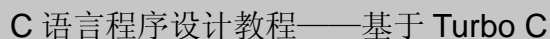
输出结果为：

```
1234567890123456789012345678901234567890
abc,          fortran,      C lan,English
abc          ,fortran,C lan      ,English
```

(6) 类型修饰。

① 在类型说明符 d, i, o, u, x, X 前面可以加字母 l (L) 或 h (H)，分别表示 long 或 short。例如 %ld 表示输出 long int 型，%hu 表示输出 short unsigned int 型。语句：

```
printf("%d,%8ld,%ld\n",7000,70000l,70000l);
```



7000, 70000, 700000

在 `printf` 函数中，控制符 `%o` 用来输出无符号形式的八进制数，`%lo` 用来输出无符号形式的八进制长整数；`%x` 或 `%X` 用来输出无符号形式的十六进制数，`%lx` 或 `%LX` 用来输出无符号形式的十六进制长整数。这些控制符都不能输出负数，而是根据数据的存储形式，将其视为无符号数输出。例如：

```
printf("%x,%o,%u,%d \n",-1,-1,-1,-1);
```

```
ffff,177777,65535,-1
```

[illegible]

若将它视为无符号十六进制，其值为 FFFF；若将它视为无符号八进制数，其值为 177777；若将它视为无符号十进制数，其值为 65535。又如：

```
printf("%o,%o,%lo\n",-01234,01234u,01234ul);
```

176544,1234,1234

```
printf("%x,%x,%lx\n",-0x6A3B,0X6A3BU,0X6A3Bu);
```

95c5, 6a3b, 6a3b

```
printf("%o,%x,%d",0177,0177,0177);
```

177, 7f, 127

```
printf("%o,%x,%d",0xABC,0xABC,0xABC);
```

5274,abc,2748

② 在类型说明符 f,e,E,g 前面可以加字母 L, 表示输出 long double 型, 因为用 “%f”, “%e”, “%E”, “%g” 格式输出 long double 型数时, 会输出错误结果。

(7) 当 printf() 函数中出现多个表达式输出项时, Turbo C 规定, printf 按照从右到左的

顺序计算各表达式的值，然后再输出结果。

输出语句的各个输出项依次是：字符常量 ('A')、零字符常量 ('\0')、字符串常量 ("A")、空字符串常量 ("") 及零字符串常量 ("\0")、字符串常量 ("A")：

```
printf("%c,%c,%s,%s,%s,%s\n",'A','\0',"A","", "\0","A");
```

当输出字符常数 '\0' 时，不显示任何字符，只空出一个字符的位置；当输出空字符串 "" 时，既不显示字符也不空出位置；输出字符串 "\0" 时，既不显示字符也不空出位置。所以输出结果为：

```
A, ,A, ,A
```

3.3.2 格式输入函数 scanf()

scanf() 函数的功能是从标准输入设备（键盘）接收数据，按格式转换说明符将数据转换成相应的格式并存放到对应的变量中。格式输入函数 scanf() 的一般形式为：

```
scanf("格式控制字符串", 输入项目清单);
```

其中，“格式控制字符串”通常只包含格式转换说明符，其含义与 printf() 类似，同样是用 % 后跟一个字母表示，“输入项目清单”要求至少有一个输入项，且必须是变量的地址（用变量名前加 & 表示）。

说明：

(1) 当程序执行到 scanf() 时，将停止继续执行，等待用户从键盘输入数据。如果只有一个输入项，则键入数据后再敲回车键；如果有多个输入项，则键入的数据之间用一个或多个空格、Tab（键盘上的 Tab 键）或回车键隔开。例如：

```
scanf("%d%d",&a,&b);
scanf("%f%f",&x,&y);
```

你可以这样输入：

```
5✓
8✓
1.2✓
3.5✓
```

也可以这样输入：

```
5      8✓
1.2    3.5✓
```

还可以这样输入：

```
5✓
8      1.2✓
3.5✓
```

(2) 在格式控制字符串中通常只出现格式转换说明符。scanf() 函数可以用于所有类型数据的输入，方法是采用不同的格式转换说明符将不同类型的数据从标准输入设备读入内存。常用的格式转换说明符示于表 3-1。



表 3-1 常用的 scanf()格式转换说明符

格式转换说明符	功能
%c	读入一个字符
%d(%D)或%i(%i)	读入一个十进制整数
%f	以小数形式读入一个浮点数
%e	以指数形式读入一个浮点数
%h	读入一个 short int 型数
%s	读入字符串，遇到空格、制表符或换行符时结束
%u	读入一个无符号十进制整数
%o	读入一个八进制整数
%x	读入一个十六进制整数

- 当用%d 控制 int 型变量时，键盘输入的数据必须是整数；当用%f 或%e 控制 float 型或 double 型变量时，键盘输入的数据可以是整数（不带小数点）、定点表示（带小数点）或指数形式（如 2e-3、3.6e4 等）表示的实数，函数会自动进行转换。
- 用%c 控制 char 型变量时，输入的字符不必加单引号，如果连续几个 char 型数据同时输入时，不要使用分隔符。
- 用%s 控制字符串时，输入的字符串不必加双引号，但遇到空格、制表符或换行符时将终止接收。

如果出现其他字符，这些字符应该照原样输入，否则 scanf 函数在输入数据中找不到这样的字符串时，就自行终止。例如：

```
scanf("%d,%d",&a,&b);
```

输入数据时，两个数据之间必须有逗号，即：

```
5,6↵
```

如果输入：

```
5 6↵
```

则变量 a 获得数值 5，scanf 找不到与格式控制字符串中对应的","，函数自行终止，导致变量 b 未获得数据。

因此，一般不要用 scanf 函数去显示一个提示信息，提示信息可用 printf 函数显示，用 scanf 函数输入数据，即：

```
printf("enter a value:");
scanf("%d",&x);
```

这样，屏幕上会先显示：

```
enter a value:_
```

用户只需键入一个数值就可以了，比如键入：

10✓

(3) 和 `printf` 函数一样, `scanf` 也要求格式转换说明符与输入项在个数、顺序、类型上一一对应。例如:

```
scanf("%d%d",&a,&b);
```

若键盘输入:

56.3145 35✓

则 `a` 和 `b` 的值就不能正确读入。

(4) 限制接收字符个数。在百分号和控制字符之间插进一个整数, 可以限制从输入数据中接收的字符个数。如果希望整型数不超过 4 位, 可用:

```
scanf("%4d%4d",&h,&k);
```

若输入:

123 45678✓

时, 函数将 123 赋给变量 `h` (输入的数据不足 4 位, 以输入的位数为准), 将 4567 赋给变量 `k` (输入的数据超过 4 位, 只取前 4 位)。

采用限制接收字符个数的方法还可以将一串字符赋给多个变量, 例如:

```
scanf("%5f%f",&x,&y);
```

当输入 1.23456789 时, 函数将 1.234 赋给变量 `x` (包括小数点在内共 5 位), 将 56789.0 赋给变量 `y` (余下的数据仍有效)。

再如, 这种限制常用在字符串的输入中, 可以防止因遇到空格等字符使输入提前终止。

```
scanf("%10s",str);
```

假设 `str` 是一个字符型数组, 长度为 20 个字符。若输入:

good morning✓

时, 接收的字符串为 “good morni”, 多余的字符被截断。如果不用 `%10s` 来限制, 同样的输入, `str` 只能接收到 “good”, 空格后面的字符被截断。

(5) `scanf` 函数与输入缓冲区。在输入数据时, 实际上并不是输入完一个数据项就被读入并送给一个变量, 而是在键入一行字符并按回车键之后才被输入, 这一行字符先放在一个缓冲区中, 然后按 `scanf` 函数格式说明的要求从缓冲区中读数据。如果输入的数据多于 `scanf` 函数所要求的个数, 余下的数据就可以为下一个 `scanf` 函数接着使用。

例 3.2 输入函数应用举例。

```
main()
{
    int a,b,c,d,e,f;
    scanf("%d %d",&a,&b);
    scanf("%d %d",&c,&d);
    scanf("%d",&e);
```




```
scanf("%d",&f);
printf("a=%d,b=%d,c=%d,d=%d,e=%d,f=%d",a,b,c,d,e,f);
}
```

运行情况如下：

```
12\34\56\78\90✓
91\92✓
a=12,b=34,c=56,d=78,e=90,f=91
```

图 3-1 表示缓冲区中数据的情况，第 1 个 scanf 函数从缓冲区读取前 2 个整数给 a 和 b (12 和 34)，第 2 个 scanf 接着读取下两个整数给 c 和 d (56 和 78)，第 3 个 scanf 函数接着读取下一个数据 (90) 给 e。此时缓冲区中数据已读完。从键盘再键入一行字符，第 4 个 scanf 函数只需取第一个整数 (91)，余下一个无用。

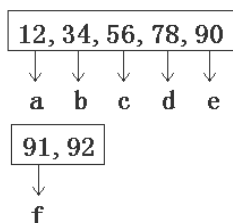


图 3-1 输入缓冲区

例 3.3 从键盘输入一个学生的多科成绩，计算出平均分，并以某一种格式输出。

```
#include<stdio.h>
main()
{
    float score1,score2,score3;
    float aver;
    scanf(" %f %f %f ",&score1,&score2,&score3);
    aver=(score1+score2+score3)/3;
    printf(" %5.2f ",aver);
}
```



注意

scanf()中的变量名前要加地址运算符。如果不加地址运算符，语法上没有出错提示，但是变量中不能得到正确的值。

3.3.3 字符输出输入函数 putchar()、getchar()

putchar()和 getchar()函数用于单个字符的输出和输入，它们使用起来比 scanf()和 printf()简洁。

1. putchar()函数

putchar()函数的一般格式为：

```
putchar(c);
```

其中，形式参数 `c` 是一个字符型常量或变量，也可以是一个取值不大于 255 的整型常量或变量。函数的功能是将字符变量 `c` 中的字符输出到标准输出设备上。

例 3.4 用 `putchar()` 函数输出一个字符。

```
#include <stdio.h>
main()
{
    int c;
    char d;
    c=65; d='A';
    putchar(c);
    putchar(d);
    putchar('A');
    putchar(65);
}
```

程序中的 4 个 `putchar()` 函数输出完全相同的结果，屏幕显示为：

```
AAAA
```

用 `putchar()` 函数还可以输出不可显示字符或控制字符。例如：

`putchar('\n');` 可以控制换行；

`putchar('\a');` 或 `putchar("\07");` 可以输出一声铃响。

2. `getchar()` 函数

`getchar` 函数的功能是从标准输入设备上读入一个字符。`getchar()` 函数的一般格式为：

```
getchar();
```

这是一个不带参数的函数，即圆括号中没有参数，但圆括号不能省略。

例 3.5 用 `getchar()` 函数输入一个字符。

```
#include <stdio.h>
main()
{
    char c;
    c=getchar();
    printf("%c\n",c);
    printf("%d\n",c);
}
```

程序运行时，等待用户从键盘输入字符，假设用户按了字母 `a` 及回车键，则屏幕显示：

```
a↓
a
97
```

其中 `a` 是 `printf()` 函数按字符格式显示 `c` 的值，`97` 是 `printf()` 函数按整数格式显示 `c` 的值。`getchar()` 还可以作为 `putchar()` 的参数，例如：



```
putchar(getchar());
```

其功能是显示用户从键盘输入的字符。

使用 `getchar()` 函数时，回车键也作为输入字符的一部分。尤其在连续使用该函数时要特别注意回车符可能已作为换行符被接收并存入另一个变量中。例如：

```
#include <stdio.h>
main()
{
    char c,d;
    c=getchar();
    d=getchar();
    printf("%c\n",c);
    printf("%d\n",d);
}
```

程序运行时，用户同样按了字母 a 及回车键，屏幕显示：

```
a↓
a
10
```

3.4 简单程序举例

例 3.6 已知北京与纽约的时差关系，当北京时间中午 12 时，纽约为上一天的 23 时，编程序实现根据键盘输入的北京时间自动转换为纽约时间。

用变量 `BeiJing_Time` 表示北京时间，用 `NewYork_Time` 表示纽约时间，则根据北京时间计算纽约时间的计算公式可写成： $\text{NewYork_Time} = (\text{BeiJing_Time} - 13 + 24) \% 24$ ，含义是为了避免时间出现负数，将二者时间差+24 转换为正常的时间表示，又因为 24 点即零点，所以要再取除以 24 的余数。

```
#include<stdio.h>
main()
{
    int BeiJing_Time,longdon_time,NewYork_Time;
    printf("\nPlease input the BeiJing_Time:");
    scanf("%d",&BeiJing_Time);
    NewYork_Time=(BeiJing_Time-13+24)%24;
    printf("\n\t*****");
    printf("\n\t%-15s%3d clock  ", " BeiJing_Time",BeiJing_Time);
    printf("\n\t%-15s%3d clock  ", " NewYork_Time",NewYork_Time);
    printf("\n\t*****\n");
}
```

程序运行情况如下：

```
Please input the BeiJing_Time: 9
*****
BeiJing_Time   9 clock
```

```
NewYork_Time 20 clock
*****
```

例 3.7 下面的程序定义了 2 个字符型变量, 1 个整型变量, 1 个浮点型变量; 接收从键盘输入的数据并按一定格式输出:

```
#include<stdio.h>
main()
{
    char c1,c2;
    int i;
    float x;
    printf("\n%s","Input two characters,an int and a float ");
    scanf("%c%c%d%f",&c1,&c2,&i,&x);
    printf("\nHere is the data that you typed in:\n");
    printf("%3c%3c%5d%17e",c1,c2,i,x);
    return 0;
}
```

如果程序运行时从键盘输入的数据为: AB1 22, 则显示结果如下:

```
Input two characters,an int and a float
Here is the data that you typed in:
  A B    1    2.20000e+01
```

如果从键盘输入的数据为 AB 1 22, 则不能得到正确的显示结果, 因为 A 后面的空格也是一个字符, scanf()中用%c 格式正常接收, 而后面的 B 用%d 格式就不能正常接收, 所以出现错误。

例 3.8 分析下面程序在执行时会输出什么?

```
main()
{
    int i=0x1a;
    int j=012;
    printf("\nas a decimal    i=%d\n",i);
    printf("as a octal      i=%o\n",i);
    printf("as a hexadecimal i=%x\n",i);
    printf("\nas a decimal    j=%d\n",j);
    printf("as a octal      j=%o\n",j);
    printf("as a hexadecimal j=%p\n",j);
}
```

程序中定义了两个整型变量 i 和 j, 并设 i 的初始值为十六进制数 1a, j 的初始值为八进制数 12, 然后分别将它们以十进制、八进制和十六进制输出。在表示十六进制常数时要加前缀 0x, 表示八进制常数时要加前缀 0; 输出时格式符 %o 表示以八进制输出, 但是输出的数据并没有前缀 0; 格式符 %x 或 %p 表示以十六进制输出, 输出的数据也没有前缀 0x, 所以格式符 %x 表示以十六进制小写形式输出, 格式符 %p 表示用 4 位十六进制大写形式输出 (相当于 %04X)。输出结果如下:

```
as a decimal    i=26
```



```
as a octal      i=32
as a hexadecimal i=1a
as a decimal    j=10
as a octal      j=12
as a hexadecimal j=000A
```

要 点 回 顾

1. C 语言的语句主要有表达式语句、复合语句、空语句和流程控制语句。表达式语句由各种表达式后面加分号组成；复合语句是用花括号括起来的语句，可以看作一个整体；空语句只有一个分号，什么也不执行；流程控制语句使程序可以根据条件改变执行顺序。

2. C 语言的输入输出是调用函数来实现的，输入函数 `scanf()` 的功能是接收键盘输入的数据给变量，输出函数 `printf()` 的功能是将数据以一定格式显示输出。

3. 输出函数的一般形式为：

```
printf("格式控制字符串",输出项目清单);
```

“格式控制字符串”由格式说明符、转义序列和普通字符组成。其中普通字符（包括转义序列）将被简单地复制显示（或执行）。而一个格式说明符将引起一个输出参数项的转换与显示。格式说明符是由“%”引出并以一个类型描述符结束的字符串，中间是一些可选的附加说明项。

4. 输入函数 `scanf()` 的一般形式为：

```
scanf("格式控制字符串",输入项目清单);
```

“格式控制字符串”与 `printf()` 函数中的类似，不过一般只含简单的格式说明符；“输入项目清单”至少有一个输入项，且必须是变量的地址（用变量名前加 & 表示）。

`printf()` 和 `scanf()` 都要求格式转换说明符与输入项在个数、顺序、类型上一一对应。

5. `putchar()` 函数的功能是将字符型量 `c` 中的字符输出到标准输出设备上。一般格式为：

```
putchar(c);
```

形式参数 `c` 是一个字符型常量或变量，或者是取值不大于 255 的整型常量或变量。

6. `getchar()` 函数的功能是从标准输入设备上读入一个字符。一般格式为：

```
getchar();
```

习 题

1. 以下选项中不是 C 语句的是（ ）。
A) {int i; i++; printf("%d\n",i); } B) ;
C) a=5,c=10 D) { ; }
2. 以下程序的输出结果是（ ）。
A) 0 B) 1 C) 3 D) 不确定的值

3. 若变量已正确定义为 int 类型, 要给 a、b、c 输入数据, 以下正确的输入语句是()。
A) read(a,b,c);
B) scanf("%d%d%d",a,b,c);
C) scanf("%D%D%D",&a,&b,&c);
D) scanf("%d%d%d",&a,&b,&c);
4. 以下语句的输出是()。
printf("%10.5f\n",12345.678);
A) |2345.67800|
B) |12345.6780|
C) |12345.67800|
D) |12345.678|
5. 若从终端输入以下数据, 要给变量 c 赋以 123.45, 则正确的输入语句是()。
123.4500✓
A) scanf("%f",c);
B) scanf("%f",&c);
C) scanf("6.2f",&c);
D) scanf("%8",&c);
6. 编写程序, 输入两个整数: 1500 和 350, 求出它们的商数和余数并进行输出。
7. 编写程序, 读入 3 个双精度型浮点数, 求它们的平均值并保留此平均值小数点后一位, 对小数点后第二位数进行四舍五入, 最后输出结果。
8. 编写程序, 读入 3 个整型数给 a,b,c, 然后交换它们中的值: 把 a 中原来的值给 b, 中原来的值给 c, 把 c 中原来的值给 a。
9. 判断下面程序段执行时的输出是什么, 上机验证输出结果与你的判断是否一致, 并分析原因。

```
unsigned int a=65535,b=6;
unsigned int c=a+b;
printf("%ld",c);
```



第 4 章 选择结构程序设计

本章的学习目标:

掌握算法的概念及其描述方法, 掌握程序的三种基本结构及结构化程序设计的基本方法; 掌握 if 语句的形式及条件表示式的用法, 掌握 switch 语句的形式及用法, 会利用 if 语句编写分支结构程序, 会利用 if 语句嵌套和 switch 语句实现多分支结构程序的设计。学会利用单步或设断点的方式调试程序, 能够应用 Break/watch 菜单监视程序运行过程中某些变量或表达式的值。

4.1 计算机算法及其描述

对于简单程序, 程序设计者往往能够立即完成软件的构思与编写; 而对于比较复杂的程序设计问题, 则需要科学合理的程序设计步骤。拿到问题, 如果立即着手编写程序代码, 往往是很难成功的, 通常需要首先解决做什么和怎么做的问题, 即确定算法问题。所谓算法, 就是为了解决问题而采取的方法和步骤。对于程序设计这个系统工程来说, 如何分析用户需求, 采取何种具体有效的计算机能够执行的方法和步骤来解决问题是非常重要的。计算机算法的表示方法很多, 例如: 自然语言; 流程图; 结构化流程图; 伪代码; PAD 图等。其中比较常用的是结构化流程图。

1. 传统流程图

使用规定的图形符号来描述算法, 形象、直观、易于理解。常用的图形符号有矩形框、菱形框、流程线等。每个框代表一段程序(一个或多个语句)的功能, 框内写明做什么工作。矩形框代表处理, 不进行比较和判断, 只有一个入口一个出口。菱形框代表条件判断, 有一个入口, 两个出口, 即根据判断结果有两个分支。菱形判断框两边用符号注明是否满足判断条件: 字母“Y”或“y”表示条件成立, 字母“N”或“n”表示条件不成立; 也可以用字母“T”或“t”表示条件成立, 字母“F”或“f”表示条件不成立。带箭头的横向或竖向直线代表执行流向, 称为流程线。



2. 结构化流程图

由于传统流程图对流程线的使用没有严格限制, 用户可以使流程随意转来转去, 从而容易造成流程混乱, 无法清晰表达算法的逻辑结构。因此人们开始设想使用几种规定的基

本结构按一定规律组织成一个算法结构。1966 年, Bohra 和 Jacopini 提出了以下 3 种基本结构作为表示一个算法的基本单元(见图 4-1)。

- 顺序结构 顺序地执行语句序列 S_1 和 S_2 , 只有当 S_1 执行完成之后才执行 S_2 。
- 选择结构 对条件进行判断, 当条件成立或不成立时分别执行语句序列 S_1 或 S_2 , 二者择其一, 不可能既执行 S_1 又执行 S_2 。不管执行哪一个语句序列, 执行结束后, 控制都转移到同一出口的地方。 S_1 和 S_2 两个框中可以有一个是空的, 即不执行任何操作。
- 循环结构 反复执行语句序列 S_1 (也称循环体), 直到条件不成立时终止循环, 控制转移到循环体外。

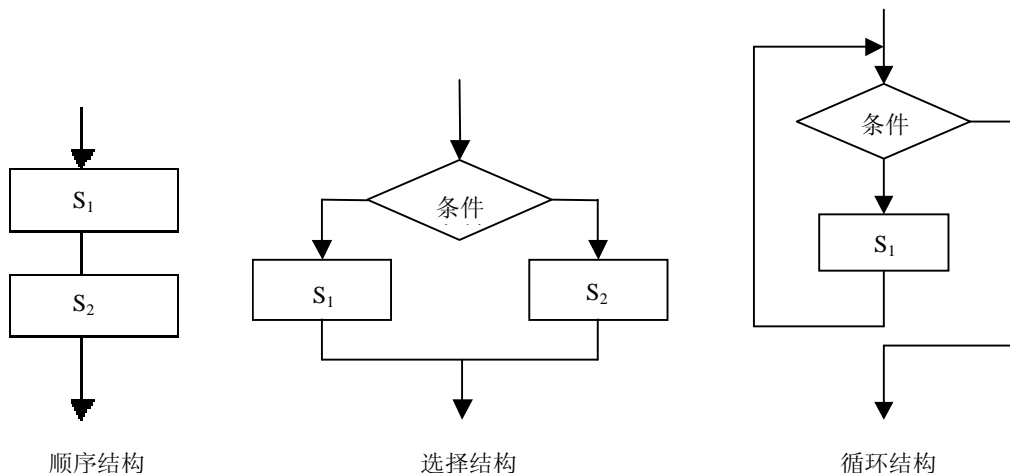


图 4-1 结构化程序的 3 种基本控制结构流程图

从图 4-1 可以看出, 这 3 种基本结构有一个共同的特点, 就是只有一个入口和一个出口, 结构内的每一部分都有机会被执行到, 结构中也没有无终止的循环(死循环)。进行程序设计时, 如果组成程序的各个分结构都只存在一个入口和一个出口这样简单的接口关系, 那么就可以相对独立地设计各个分结构, 静态地分析控制关系, 并验证它们的正确性, 而不必动态地转来转去。同样, 如果一个分结构需要修改, 则只要接口不变, 就不会影响到其他的分结构和整个程序。因此, 由这 3 种基本结构所构成的流程图称为结构化流程图, 由结构化流程图所表示的算法为“结构化”的算法, 具有这样结构的程序为结构化的程序。

3. N-S 流程图

由于使用 3 种基本结构的组合可以表示任何复杂的算法结构, 因此人们设想取消基本结构中的流程线。1973 年美国学者 I • Nassi 和 B • Shneiderman 提出了一种新的流程图形式。这种流程图形式去掉了带箭头的流程线, 将所有算法写在一个矩形框内, 该框内又可以包含其他的从属框。该流程图以两位发明者的英文姓名的第一个字母命名为 N-S 结构化流程图; 又由于这种流程图非常简捷, 如同一个多层盒子, 因此又被称为盒图(box diagram), 特别适合于表示结构化程序设计。

N-S 流程图使用流程图符号表示 3 种基本结构(如图 4-2 所示)。



- 顺序结构 由 S_1 框和 S_2 框组成。
- 选择结构 当条件成立时执行 S_1 操作，当条件不成立时执行 S_2 操作。
- 循环结构 当条件成立时反复执行 S_1 操作，直到条件不成立为止。

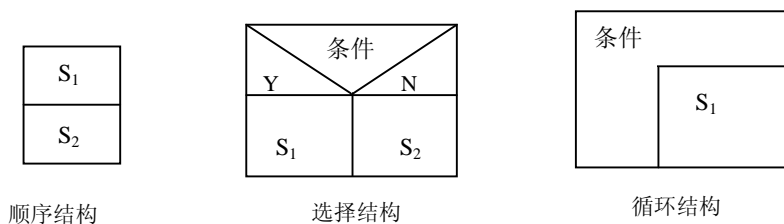


图 4-2 结构化程序的 3 种基本控制结构 N-S 图



注意

上图中的 S_1 或 S_2 框，可以表示一个简单的操作，也可以是 3 个基本结构之一。用这 3 种 N-S 流程图基本形式可以组成各种复杂的流程图，以表示算法。

当然，采用结构化程序设计方法，只要程序设计语言具有上述 3 种程序控制结构就很完备了。但为了方便用户使用，各种程序设计语言还是常常引进其他各种各样的控制结构，如多路选择结构、直到型循环结构、步长型循环结构等。这些结构尽管都可以用上述三种基本结构来表示或代替，但它们一般都具有更简洁的形式（如图 4-3、图 4-4 所示），直接使用会更加方便。

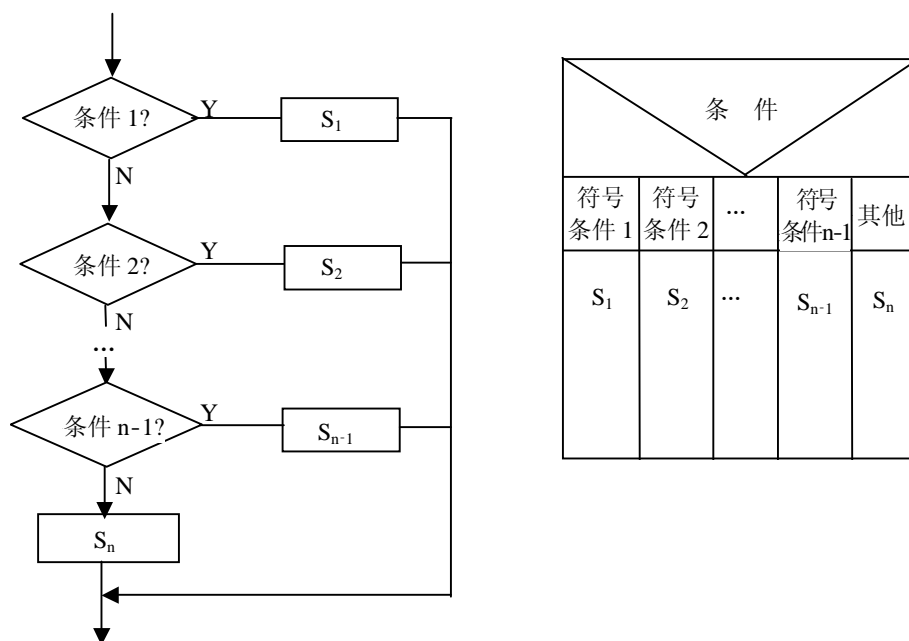


图 4-3 多路选择结构的流程图和 N-S 图

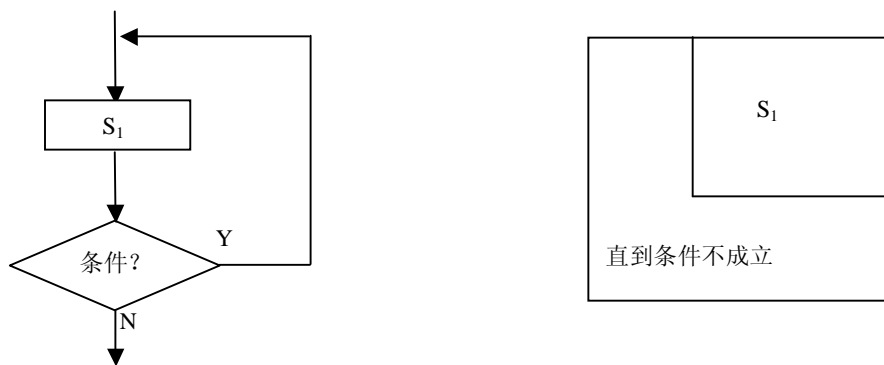


图 4-4 直到型循环结构的流程图和 N-S 图

例 4.1 分别用传统流程图和 N-S 图表示判断闰年的算法。判断闰年的方法为：①公元年数不能被 4 整除，则不是闰年；②公元年数能被 4 整除，而不能被 100 整除，则是闰年；③公元年数能被 400 整除也是闰年。结果见图 4-5 和图 4-6。

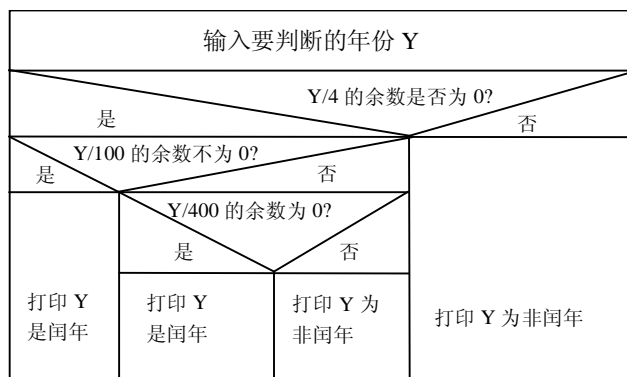


图 4-5 用 N-S 流程图表示的判断闰年算法

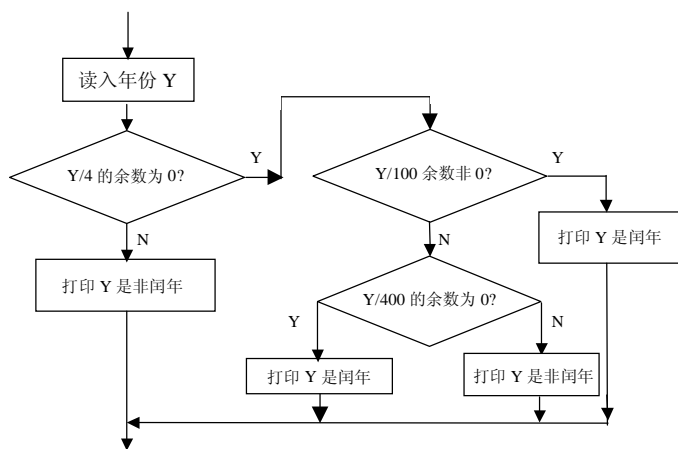


图 4-6 用传统流程图表示的判断闰年算法



4.2 结构化程序设计的概念

所谓结构化程序，是指程序结构清晰、易于阅读、易于验证。好结构的程序从效率上看不一定是最好的程序，但是它能提高程序的可读性、可理解性和可靠性，易于查错和维护。在计算机运行速度大大提高、存储容量不断扩大的情况下，程序的可靠性和可维护性已成为第一要求，除了系统的核心程序以及其他一些特殊要求的程序之外，在通常情况下，宁可降低一些效率，也要保证好的程序结构。除了面向对象的程序设计方法之外，目前普遍使用和比较成熟的程序设计方法，就是结构化程序设计方法。

结构化程序设计方法就是只采用 3 种基本的程序控制结构来编制程序，从而使程序具有好的结构。Bohra 和 Jacopini 于 1966 年证明：任何一个程序，不管多么复杂，均可用“顺序”、“选择”和“循环”这 3 种基本结构经排列、嵌套或组合而构成的程序来描述。这个定理称为结构化定理，它是结构化程序设计的理论基础。

从软件工程角度来说，开发一个应用程序，一般需要经过几个步骤：① 分析问题、确定算法；② 用流程图表示出程序算法；③ 根据流程图编写程序代码；④ 调试运行程序。

在以上几个步骤中，第一个步骤尤为重要。实质上，一个结构化的程序就是用计算机语言来描述一个结构化的算法。当程序员面临复杂的问题时，很难快速写出正确的程序算法，只有把复杂问题分模块、分阶段求解，才能更好地得到最终结果。具体说，应按照“自顶向下，逐步求解”的原则对问题进行分解，然后再用结构化程序开发步骤编制出各个分问题的程序模块，最后构造出整个问题的算法和程序。只有这样，才能得到质量较高的结构化算法，从而写出结构化的程序。

C 语言提供了控制语句来实现结构化程序设计，这些语句有条件分支控制语句（if, switch）、循环控制语句（for, while, do...while）和无条件转移语句（goto）。

4.3 选择结构程序设计

选择结构是结构化程序设计中的一种基本结构，是应用非常广泛的一种结构。我们经常会遇到需要计算机进行逻辑判断的情况，例如：比较两个数的大小，并输出判断结果；或根据职工不同的岗位，发放不同的工资和奖金。这些问题都是顺序结构程序所无法实现的，而是属于选择结构程序设计的范畴。C 语言提供了如下形式的选择语句来实现选择结构。

4.3.1 if-else 分支语句

形式 1:

```
if (条件表达式)
    语句 S1;
[ else
    语句 S2;
]
```

形式 2:

```
if (条件表达式)
    语句;
```

或

```
if (条件表达式)
{
    语句序列;
}
```

如图 4-7 所示的形式 1 中的 if-else 构造了一种二选一的分支选择结构, 其中 else 是可选项; 如图 4-8 所示的形式 2 中表示了 if-else 结构省略 else 部分的情况。if-else 语句形式 1 的执行过程是: 先判断条件表达式的结果, 如果结果为真 (成立, 值为非零), 则执行语句 S_1 , 而不执行语句 S_2 ; 否则若结果为假 (不成立, 值为 0), 则执行语句 S_2 , 而不执行语句 S_1 。这是一种最基本的选择结构。形式 2 的执行过程为: 如果条件表达式的结果为真, 则执行相应的语句序列, 若条件表达式的结果为假, 则不执行 if 后面的语句序列。条件表达式必须放在圆括号中; 语句 S_1 和语句 S_2 可以是单语句, 也可以是复合语句。在条件测试中, 当条件表达式的值为一切非 0 值 (包括正数和负数) 时, 就为真, 即条件成立; 只有表达式的值为 0 时, 才为假, 即条件不成立。

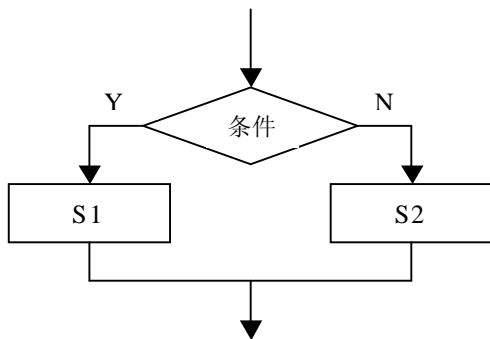


图 4-7 选择结构形式 1 的对应流程图

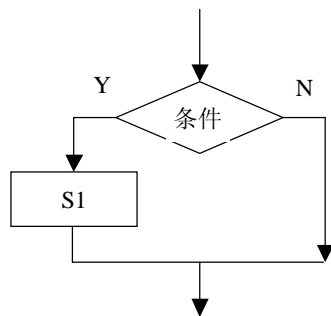


图 4-8 选择结构形式 2 的对应流程图



注意

这里的条件表达式可以是关系表达式或逻辑表达式, 也可以是任何数值表达式, 但计算结果必须为整型、浮点型或字符型。

下面对 if 语句的使用进行说明。

(1) C 语言中条件表达式的书写非常灵活, 初学者要引起足够的重视。例如:

```
if(a=3)printf("%d",a);
```

和

```
if(a==3)printf("%d",a);
```

执行结果就全然不同, 因为前者的条件是一个赋值表达式, 其值恒为 3, 为真; 后者的条件是一个关系表达式, 只有 a 等于 3 时才为真。又如,



```
if(a-2)
    printf("%d",a);
```

和

```
if(a!=2)
    printf("%d",a);
```

则是等价的。if(a-2)表示的含义为：如果表达式 a-2 的值为真，即 a-2 的值为非零，则输出 a 的值，因此与 if(a!=2)是等价的。

在 C 程序中，还经常用 if(!x)来代替 if(x==0)，用 if(x)代替 if(x!=0)，它们功能是等价的，使用 if(!x)和 if(x)形式更简洁，但是从理解的角度来看，后者的表达形式更容易理解。

(2) if 语句中出现语句块时，一定要用花括号括住，表示为复合语句，可写成如下格式：

```
if(条件表达式)
{
    语句块 1
}
else
{
    语句块 2
}
```

这里语句块 1 和语句块 2 用花括号括住，表示为一个复合语句。例如，设 a=1,b=2; 程序段：

```
if(a>b)
{
    t=a;a=b;b=t;
}
/* 复合语句，表示如果 a>b，则执行复合语句体 */
printf("a=%d,b=%d",a,b);
```

和

```
if(a>b)
    t=a;a=b;b=t;
printf("a=%d,b=%d",a,b);
/* 表示如果 a>b，则仅执行语句 t=a; */
```

前者输出 a=1,b=2，而后者输出 a=2,b 为不定值。可见，加不加花括号，其执行结果是完全不同的。

(3) 如果 if 语句的两个分支都是赋值语句，且是给同一个变量赋值的语句，可以用条件运算符来代替 if 语句。条件运算符是一个三元运算符，需要三个操作对象，条件运算符的使用格式为：

```
表达式 1? 表达式 2: 表达式 3
```

用条件运算符组成的表达式称为条件表达式。条件表达式的求解过程为：先求解表达式 1，若为真，则求解表达式 2 并将表达式 2 的值作为整个条件表达式的值；若条件表达式的值为假，则求解表达式 3 并将表达式 3 的值作为整个条件表达式的值。

例如：max=(a>b)?a:b;将条件表达式的结果赋值给 max 变量，实现的功能是选出 a,b 中

的较大数。等价的 if 语句如下：

```
if(a>b)
    max=a;
else
    max=b;
```

可见用条件运算符更简洁。但是很多情况下，条件运算符并不能替代 if 语句。

4.3.2 if-else 编程举例

例 4.2 求一个整数的绝对值。

分析：设有任意整数 x ，它的绝对值为：

$$|x| = \begin{cases} x & x \text{ 大于等于 } 0 \\ -x & x \text{ 小于 } 0 \end{cases}$$

算法如图 4-9 示。

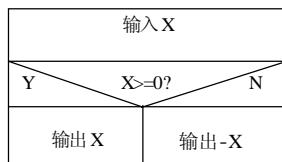


图 4-9 算法流程图

程序如下：

```
main()
{
    int x,y ;
    printf("please enter x: ");
    scanf("%d ", &x);
    if(x>=0)
    {
        y=x;
        printf(" y=%d\n",y);
    }
    else
    {
        y=-x;
        printf("y=%d\n" ,y);
    }
}
```

请读者上机调试程序并分析：如果 else 后面的花括号去掉，程序执行的流程如何？为什么会出现错误的结果？



例 4.3 从键盘输入一个整数，判断它的奇偶性。

分析：如果一个整数能被 2 整除，则其为偶数，否则为奇数。能被 2 整除，用数学方法来表达即为该数值除 2 的余数为 0。程序如下：

```
main()
{
    int x ;
    printf("please enter x: ");
    scanf("%d" ,&x);
    if(x%2==0)
        printf(" %d is an even number.\n",x);    /* 该数为偶数 */
    else
        printf("%d is an odd number.\n" ,x);    /* 该数为奇数 */
}
```

例 4.4 从键盘输入三角形的三条边 a,b,c，求三角形的面积。

分析：判断是否构成三角形的条件是任意两边之和大于第三边。if 条件表达式可以写为：

```
a+b>c&& a+c>b&& b+c>a
```

求三角形的面积时可以使用海伦公式：

面积=sqrt(s*(s-a)*(s-b)*(s-c))，其中 s 为(a+b+c)/2

这里 sqrt()为求平方根的函数，该函数的描述在 math.h 中，因此程序开头要用#include <math.h>把 math.h 包含进来。程序如下：

```
#include <stdio.h>
#include <math.h>
main()
{
    float a,b,c,area,s;
    printf("please enter a,b,c:\n");
    scanf("%f %f %f",&a,&b,&c);
    if(a+b>c&& a+c>b&& b+c>a)
    {
        s=(a+b+c)/2.0;
        area= sqrt(s*(s-a)*(s-b)*(s-c));
        printf("area=%f\n",area);
    }
    else
    {
        printf("data invalid");
    }
}
```

4.4 选择结构的嵌套

4.4.1 if-else 语句的嵌套

if-else 结构可以嵌套使用，即一个 if-else 语句中又包含一个或多个 if-else 语句。嵌套规则是 else 与上面距离最近并且没有其他 else 与其配对的 if 配对。例如有以下几种形式：

形式 1:

```
① if(条件)
   {
     if (条件)
       语句;
     else
       语句;
```

形式 2:

```
② if(条件)
   {
     if (条件)
       语句;
   }
else
  语句;
```

形式 3:

```
③ if (条件)
   {
     ④ if (条件)
        {
          语句;
        }
     else
       语句;
   }
else
   {
     ⑤ if (条件)
        {
          语句;
        }
     else
       语句;
```



注意

这里①②③④⑤表示的 if-else 为一对，其间的语句既可以为单个语句，又可以为一个 if-else 结构，又可以是复合语句，非常灵活。如果想改变以上配对规则，可采用花括号{}括起来，如②所示。

为了使嵌套层次清晰明了，在程序的书写上常常采用所谓的缩排格式，即不同层次的 if-else 出现在不同的缩排级上；但是 if-else 的匹配与缩排格式无关。

由于 C 语言的 if 语句没有终端语句，所以在 if 嵌套的情况下要特别注意 else 和 if 的配对关系，避免引起逻辑上的混乱。



例 4.5 根据键盘输入的 3 个数，找出最大数并输出。如果程序编写如下：

```
#include <stdio.h>
main()
{
    float x ,y ,z ,max ;
    printf("Enter 3 real numbers x ,y ,z :\n" );
    scanf("%f,%f,%f",&x,&y,&z);
    max=x;
    if(z>y)
        if(z>x)
            max=z;
        else
            if(y>x)
                max=y;
    printf("The max is %f\n",max );
}
```

它的一次运行情况为：

```
Enter 3 real numbers x ,y ,z:
1.2,5.6,3.4↓
The max is 1.200000
```

可以看出程序的运行结果是错的，那么为什么会出现错误结果呢？从书写形式上看，似乎 `else` 应与第一个 `if` 配对，即满足如图 4-10(a)所示的逻辑关系。实际上，C 语言规定：当 `if` 没有与它配对的其他 `else` 时，`else` 总是与离自己最近的 `if` 配对。本程序段中，`else` 与第二个 `if` 配对，它所描述的逻辑关系应该是图 4-10(b)。

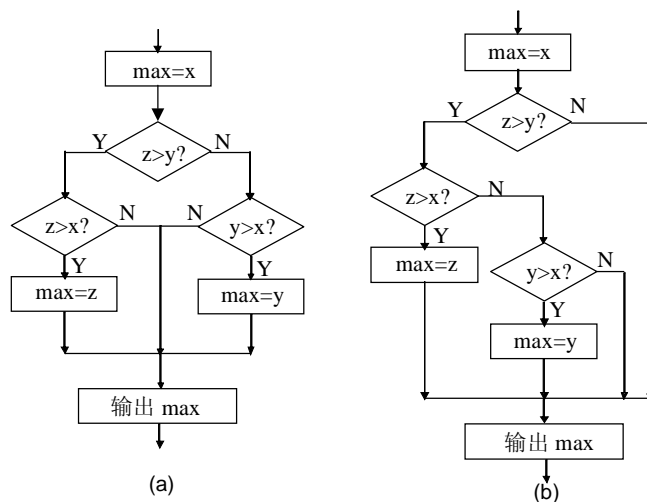


图 4-10 算法流程图

正确的程序如下：

```
#include <stdio.h>
main()
```

```
{
    float x ,y ,z ,max ;
    printf("Enter 3 real numbers x y z :\n" );
    scanf("%f%f%f",&x,&y,&z);
    max=x;
    if(z>y)
    {
        if(z>x)
            max=z;
    } /* 相对于出错程序，这里加了花括号 */
    else
    {
        if(y>x)
            max=y;
    }
    printf("\n max=%f",max);
}
```

程序中嵌套使用了 if 语句，有的是 if-else 结构形式，有的没有 else。如果一个分支内只是一个 if 语句，有时可以省略表示该分支的花括号，但要特别注意 if 和 else 语句的配对关系，不要出现本例程序中的类似问题。为避免混淆，建议不要省略花括号。

例 4.6 编写程序，求解一元二次方程 $ax^2+bx+c=0$ 的根。

设计算法见图 4-11。

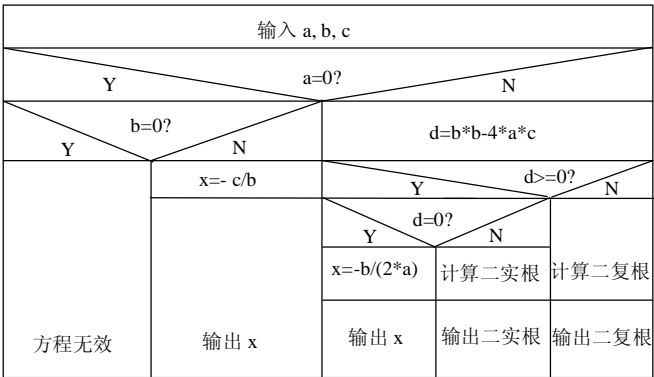


图 4-11 N-S 流程图表示的求一元二次方程算法

程序如下：

```
#include <stdio.h>
#include <math.h>
main()
{
    float a,b,c,d ,r,l,x1,x2;
    printf("\n ENTER a b c:");
    scanf("%f%f%f",&a,&b,&c);
    if(a==0)
        if(b==0)
```



```
        printf("\n a, b, c are illogical. They can't form an equation. ");
    else
    {
        x1= -c/b;
        printf("x=%f",x1);
    }
    else
    {
        d=b*b-4*a*c;
        if(d>=0)
            if(d==0)
            {
                x1=-b/(2*a);
                printf("x1=x2=%f ",x1);
            }
            else
            {
                x1=-b/(2*a)+sqrt(d)/(2*a);
                x2=-b/(2*a)-sqrt(d)/(2*a);
                printf("\n x1=%f x2=%f ",x1,x2);
            }
        else
        {
            r=-b/(2*a);
            I=sqrt(-d)/(2*a);
            printf("\n x1=%f + %f i",r,I);
            printf("\n x2=%f - %f i\n",r,I);
        }
    }
}
```

在此例中，将根的实部和虚部分别计算，并与格式控制符配合以实现复根的输出。

4.4.2 if-else if 结构

在 if-else 结构的嵌套过程中，如果嵌套的层数过多，每层又要往右缩进，会使程序看起来很繁琐。如果使用 if-else if 结构来处理就简捷一些。其形式如下：

```
if (条件表达式 1)
    语句 1;
else if (条件表达式 2)
    语句 2;
...
[else if (条件表达式 n)
    语句 n;]
[else
    语句 n+1;]
```

其中，最后的 else 及其下面的语句 n+1 为可选项，也就是说，如果所有的条件表达式的值均为假，则执行 else 语句；如果最后的 else 语句不存在，则不执行任何操作。if-else if

结构的流程是自顶向下执行的，如果发现某个条件表达式为真，则与该条件相连接的那个 if 语句将被执行，其他部分被忽略。如果所有条件表达式的值为假，则执行 else 后面的语句 n+1。其算法流程图表示如图 4-12。

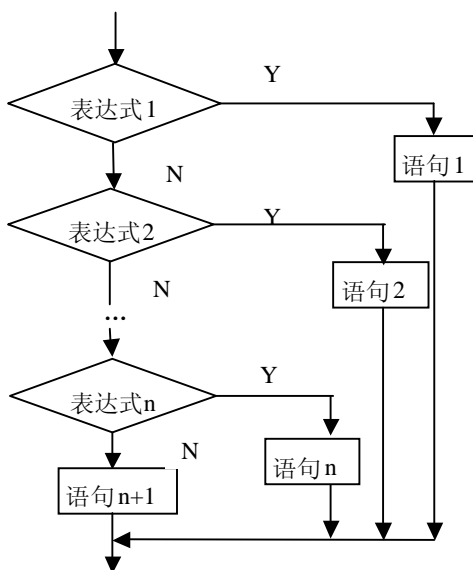


图 4-12 if-else if 算法流程图

下面我们用 if-else 结构的嵌套和 if-else if 语句来完成一个程序，以比较它们的使用。

例 4.7 用 if-else 嵌套和 if-else if 语句结构完成程序：从键盘输入任意年份，判断它是否为闰年。

前面我们已经分析了判断闰年的算法，在此不再赘述。if-else 嵌套方式的程序如下：

```
#include <stdio.h>
main()
{
    int year, leap;
    scanf("%d", & year);
    if(year%4)
        leap=0;          /* year 不能被 4 整除，非闰年 */
    else
    {
        if(year%100)
            leap=1;       /* year 能被 4 整除，不能被 100 整处，是闰年 */
        else
        {
            if(year%400)
                leap=0;
            else
                leap=1;
        }
    }
}
```



```
if(leap)
    printf("%d is a leap year.\n",year);
else
    printf("%d is a common year.\n",year);
}
```

用 if-else if 结构编写的程序如下:

```
#include <stdio.h>
main()
{
    int year, leap;
    scanf("%d", &year);
    if(year%4)
        leap=0;
    else if(year%100)
        leap=1;
    else if(year%400)
        leap=0;
    else
        leap=1;
    if(leap)
        printf("%d is a leap year.\n", year);
    else
        printf("%d is a common year.\n", year);
}
```

可以看出, 用 if-else if 结构写出的程序更为简洁易懂, 特别是层次多的时候, 其优点更为突出。

当然就这个例题来说, 由于判断闰年的表达式可以表示为: 若表达式 $(year\%4==0\&\&year/100!=0\|year\%400==0)$ 的结果为真, 则该年为闰年。因此上述程序可以更加简化为用一个 if 语句来实现, 请读者自己完成具体步骤。如果求平年, 逻辑表达式可表示为: 只要表达式 $!(year\%4==0\&\&year/100!=0\|year\%400==0)$ 的结果为真, 则该年为平年。

例 4.8 编写一个菜单程序, 用以完成如下的数制转换。

- 输入 1, 将十进制转换为十六进制。
- 输入 2, 将十六进制转换为十进制。
- 输入 3, 将十进制转换为八进制。

程序如下:

```
#include <stdio.h>
main()
{
    int choice, value;
    printf("\n1 to convert decimal into hex\n");
    printf("2 to convert hex into decimal\n");
    printf("3 to convert decimal into octal\n");
    printf("3 to convert octal into decimal\n");
}
```

```
printf("Please input your choice: ");
scanf("%d",&choice);
if( choice==1)
{
    printf("Please input a decimal number:");
    scanf("%d",&value);
    printf("%d be converted into a hex as %x ",value ,value);
}
else if( choice==2)
{
    printf("Please input a hex number: ");
    scanf("%x",&value);
    printf("%x be converted into a decimal number as %d ",value ,value);
}
else if( choice==3)
{
    printf("Please input a decimal number: ");
    scanf("%d",&value);
    printf("%d be converted into an octal number as %o ",value ,value);
}
}
```

例 4.9 根据键盘输入的成绩, 转换成 A、B、C、D、E 5 个级别。其中 90 分以上为 A, 80~89 分之间为 B, 70~79 之间为 C, 60~69 分之间为 D, 59 分以下为 E。

```
#include<stdio.h>
main()
{
    float score;
    char grade;
    printf("please input the score: ");
    scanf("%f",&score);
    if(score>100&&score<0)
        printf("data invalid");
    if(score>=90)
        grade='A';
    else if(score>=80)
        grade='B';
    else if(score>=70)
        grade='C';
    else if(score>=60)
        grade='D';
    else
        grade='E';
    printf("%c ",grade);
}
```



4.5 用 switch 语句实现多分支选择结构

尽管用 if-else if 结构或嵌套的 if 语句可以实现多分支，但当分支较多时，程序在结构上不够精巧。C 语言提供了一个更为方便的实现多分支结构的语句 **switch**，一般形式如下：

```
switch(表达式)
{
    case 常量表达式 1: 语句序列 1;
        [break; ]
    case 常量表达式 2: 语句序列 2;
        [break; ]
    ...
    case 常量表达式 n: 语句序列 n;
        [break; ]
    [ default: 语句序列 n+1; ]
}
```



注意

switch 后面括号中的表达式只能是整型、字符型或枚举型表达式，case 后面的常量表达式的类型必须与其匹配。所谓“常量表达式”，是指该表达式的值必须在运行前就是确定的，不能改变。例如下面的程序段是错误的。

```
int x=1,y=2;
switch(z)
{
    case x+y:      /*使用变量是错误的*/
    ...
}
```

若 switch 中表达式的值与某一 case 后面的常量表达式匹配时，就执行此 case 后面的所有语句序列，直到遇到 break 语句或 switch 语句的结束标志“}”，所以 case 通常与 break 语句连用，以保证多路分支的正确实现。在各 case 分支都带 break 的情况下，各 case 语句的出现次序可以任意。另外，多个 case 可以共用一组执行语句。

default 语句是可选的。如果表达式的值与所有 case 后的常量表达式都不匹配，就执行 default 后的语句；如果都不匹配时不执行任何操作，则可以没有 default 语句。

break 语句在语法上是一个可选项，用于结束与某个常量相联接的语句序列。

switch 语句与 if 语句的测试条件区别在于：switch 仅能判断一种逻辑关系，即相等关系，而不能进行大于、小于的判断，不能表达区间的概念，而 if 语句的条件表达式可以是任何一种条件关系，应用更为灵活。

在同一个 switch 语句中，两个 case 常量不能为同一值。

switch 语句经常用来处理有较多分支的情况，并且各分支的取向可以用一个常量值来表达。当对应多个常量表达式都执行同一个语句序列时，可以写在一起。switch 语句也可以嵌套。

例 4.10 用 switch 语句编写程序，根据键盘输入的成绩打印出等级。其中 90 分以上

为 A, 70~89 分之间为 B, 60~69 分之间为 C, 59 分以下为 D。

```
#include <stdio.h>
main()
{
    int score,temp;
    printf("\nPlease input the score: ");
    scanf("%d",&score);
    if(score==100)
        temp=9;
    else
        temp=score/10;
    switch(temp)
    {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:printf("D");break;
        case 6:printf("C");break;
        case 7:
        case 8:printf("B");break;
        case 9:printf("A");break;
        default:printf("the data is invalid");
    }
}
```

在程序中,是通过 `temp=score/10` 把 `score` 的区间值转化为一个常量值,从而可以使用 `switch` 开关语句。在此程序中,如果 `temp` 的取值为 0,则执行其后的分支,直到遇到 `break` 语句结束,所以当 `temp` 取值为 0、1、2、3、4、5 时,都是输出 D;同理,当 `temp` 的取值为 7 和 8 时,输出 B。

当 `temp` 的取值不在 0~9 之间时,程序执行 `default` 分支,打印“data invalid”。如果没有 `default` 分支,则什么也不输出,可见 `default` 分支会给逻辑检查带来很多方便。例如,如果用 `switch` 语句处理固定数目的条件分支,超出这些条件分支之外的任何值都属于逻辑错误,则可以使用 `default` 语句分支来识别逻辑错误。

请读者将例 4.9 和例 4.10 的程序对照,并将例 4.8 的程序用 `switch` 结构进行改写。

例 4.11 编写程序,根据用户输入的运算符和数据模拟简单计算器的功能。
其算法流程图见图 4-13。

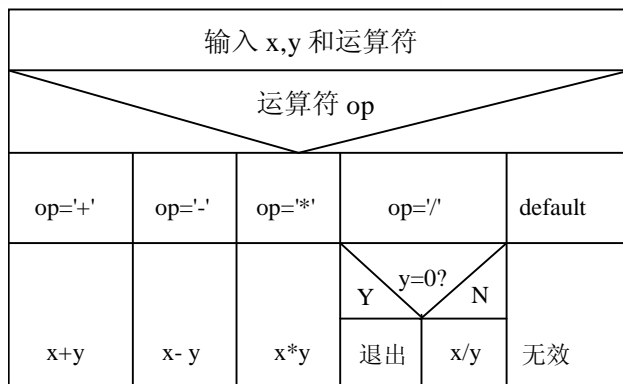


图 4-13 算法流程图

程序如下：

```
#include <stdio.h>
main()
{
    float x,y;
    char op;
    printf("\nPlease input a number x an operator and a number y:\n");
    scanf("%f%c%f",&x,&op,&y);
    switch(op)
    {
        case '+': printf("%.2f+%.2f=%.2f",x,y,x+y); break;
        case '-': printf("%.2f-%.2f=%.2f",x,y,x-y); break;
        case '*': printf("%.2f*%.2f=%.2f",x,y,x*y); break;
        case '/':
            if(y==0)
                printf("divided by zero.\n"); break;
            printf("%.2f / %.2f=%.2f",x,y,x/y); break;
        default:printf("the operator is invalid!\n");
    }
}
```

在程序中，当运算符为“/”时，case 后面的语句是一个 if 语句结构，判断如果被除数 y 是 0，就不进行除法运算。

4.6 程序调试过程举例

当程序不是简单的顺序结构时，有时会出现编译、连接时都已经没有出错提示，但程序运行时得到错误结果的现象。这是因为程序中出现了逻辑错误（编译器只能检查出语法错误），需要对算法进行进一步分析；还有可能程序中用到的变量的值超出范围（溢出）、数组下标越界，或指针指向的地址不对。这些情况编译时并不算出错，但是程序的结果错误却很严重，所以对程序中数组和指针的使用要重点检查；注意编译时给出的警告信息，必要时通过 F7 或 F8 单步执行程序来检查程序的执行步骤；如果程序代码行较多，则通过

设断点的方法对程序进行调试，在程序暂停时检查变量和表达式的值从而发现问题。

例如：输入例 4.5 中的错误程序，编译连接都正常。当运行时输入试验数据 1.2, 5.6, 3.4，得到结果 The max is 1.2，是错误的。

1. 监视变量

为了检查程序执行过程中哪里出现错误，可以单步跟踪程序执行并监视几个变量的变化情况。如图 4-14 所示，可以在程序开始运行前或者程序暂停时，用 Alt+B 键调出 Break/watch 菜单，添加（Add watch）或编辑（Edit watch）要监视的变量（按 F6 键可以使光标在 Edit 和 Watch 窗口之间切换）：如 x 是浮点型变量，则在出现的编辑框里输入 x,f；如果 x 是整型变量，则在编辑框里输入 x,d（以十进制显示）或 x,x（以十六进制显示），然后回车。

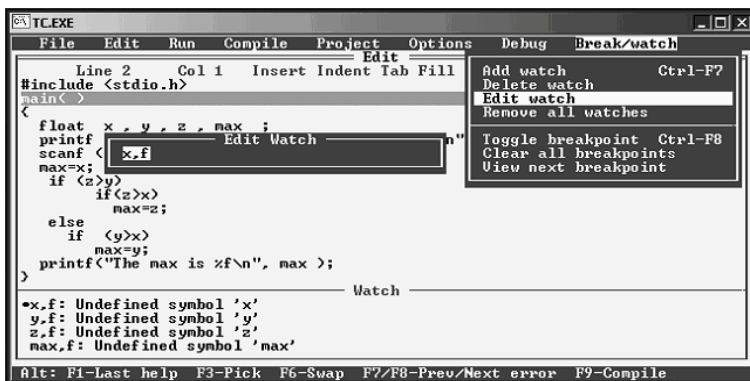


图 4-14 编辑监视变量

将光标移动到第一行，按 F7 键开始以单步方式跟踪执行程序。如图 4-15 所示，屏幕下部的 Watch 窗口中将随时显示各变量的当前值。在定义变量后且还没有给变量输入数据之前，变量的取值是随机数。

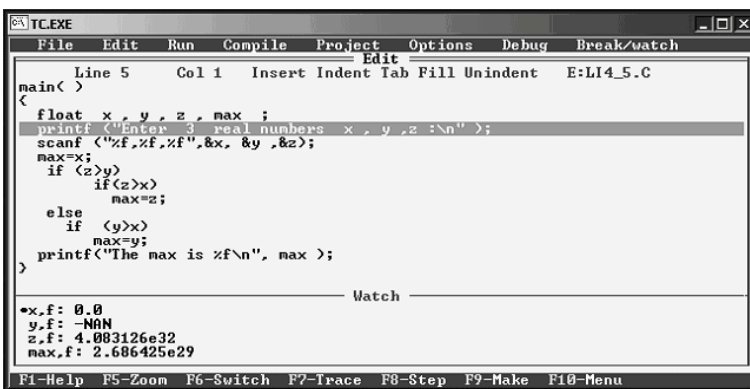


图 4-15 单步执行程序

执行完输入语句后，x,y,z 变量的值即为键盘输入的值。再执行完赋值语句后，max 变量的值为 x 变量的值。每按一次 F7 键，执行一条语句，亮条停在逻辑上的下个语句位置上，如图 4-16 所示。

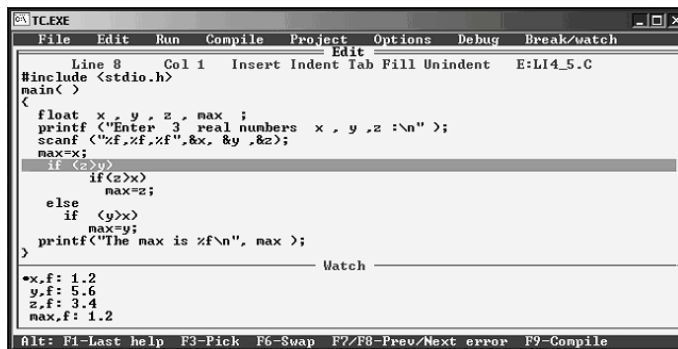


图 4-16 单步执行程序

按 F7 逐条语句执行程序并同时观察各变量的值，会发现执行 if(z>y)后接着执行的是输出语句，而并不是像希望的那样执行 else 后的语句。由此可发现程序中的逻辑错误，再对程序加以分析并改正过来。

2. 设置断点

设置断点的方法是先移动光标到需要设断点的语句行，然后用 ctrl+F8 或者 Break/watch 菜单中的 Toggle breakpoint 设断点，该操作相当于软开关，可以设断点，也可以清除已有的断点。

用 Alt+Run 或 Ctrl+F9 运行程序，到断点处，程序将暂停运行，再按 Ctrl+F9 继续运行到下一断点。

在单步和断点方式下执行程序，当程序暂停运行时，可以在 Watch 窗口察看各变量的值，也可以用 Alt+F5 转到用户屏幕察看部分程序的输出。

3. 添加输出语句

还有一种在调试程序时行之有效的方法，是在程序中有疑问的地方添加输出语句输出中间结果，并根据输出的中间结果进行分析来发现问题，程序调试好后再去掉输出语句。这种方法请读者在以后的程序调试过程中自己应用。

要 点 回 顾

1. 算法是程序设计的重要内容。常用的算法表示有自然语言描述、传统流程图和 N-S 流程图。

2. 任何复杂的程序，均可用“顺序”、“选择”和“循环”这 3 种基本结构经排列、嵌套或组合而构成的程序来描述。这 3 种基本结构有一个共同的特点，就是只有一个入口和一个出口，结构内的每一部分都有机会被执行到，结构中也没有无终止的循环（死循环）。结构化程序设计方法就是只采用 3 种基本的程序控制结构来编制程序。

3. 实现分支结构的语句是 if 语句，if 语句有如下形式：

```

if (条件表达式)
    语句 S1;
[else
    语句 S2;
]

```

如果“条件表达式”的结果为真（非 0），则执行语句 S₁，若为假（为 0），则执行语句 S₂（在省略 else 的情况下，什么也不执行）；S₁，S₂ 可以是复合语句。C 语言中条件表达式的书写非常灵活，可以是关系表达式或逻辑表达式，也可以是任何数值表达式。

4. 如果 if 语句的两个分支都是赋值语句，且是给同一个变量赋值的语句，可以用条件运算符来代替 if 语句。条件运算符需要 3 个操作对象，条件运算符应用的格式为：

表达式 1? 表达式 2: 表达式 3

用条件运算符组成的表达式称为条件表达式。当“表达式 1”为真时，以“表达式 2”的结果作为条件表达式的结果；当“表达式 1”为假时，以“表达式 3”的结果为条件表达式的结果。

5. 分支结构中的每一个分支还可以是另一个分支结构，即分支结构嵌套。可以用 if-else 语句实现分支结构的嵌套，也可以用 if-else if 来处理，嵌套层数多时后者更简捷。

6. 分支结构嵌套的程序要注意 else 是和哪一个 if 相对应，避免出现逻辑错误。

7. 可以用单步和设断点的方式跟踪程序实际执行的流程，并利用 Watch 窗口监视程序执行过程中变量和表达式的值。

习 题

1. 判断一个整数能否被 5 和 7 整除：如果能，将它打印出来；若不能，则提示输入错误。
2. 输入两个数，比较大小并输出结果。
3. 有 3 个整数 x, y, z，编程将 3 个数由小到大顺序排列输出。
4. 分别用嵌套 if 语句和 if-else if 语句编写程序，计算下面的分段函数（x 由键盘输入）：

$$y = \begin{cases} x & x < -1 \\ x^2 + x + 1 & -1 \leq x \leq 1 \\ \sqrt{9 - x^2} & 1 < x < 2 \\ -x^2 - 0.6x & 2 \leq x \end{cases}$$

5. 任意给出一个不多于 4 位的正整数，编程：①判断它是几位数；②输出每一位数字；③按逆序输出各位数字。
6. “神州行”是移动公司推出的一种手机服务，其话费计算方法如下：

市话	0.6 元/3 分钟
区间通话	0.8 元/3 分钟
省内漫游	1.3 元/分钟
省外漫游	1.6 元/分钟



分别用 if 语句和 switch 语句编写程序，实现如下功能：根据提示输入通话时间（以分钟为单位），再根据菜单提示输入通话方式（可以用数字或字符表示，如用 1 表示“市内电话”），计算相应的收费金额并输出（保留两位小数，对第 3 位小数四舍五入）。

7. 写一程序，实现从键盘接收字符，遇到'Q'或'q'时结束；并把大写字母变为小写字母、小写字母变为大写字母输出。

8. 若 x、y、z 均为整型变量，请分析以下程序段中的语句是否有错？为什么？

A)

```
switch(x*x+y*y)
{
    case 4:z=x-y;break;
    case 5:z=x/y;break;
    case 3+2:z=x*y;break;
}
```

B)

```
switch 3*x*y
{
    case 1:
    case 2:z=(x-y)/(x*y);break;
    case 4:z=exp(x)+exp(y);break;
    case 7:
    case 8:z=x*x-y*y;
}
```

C)

```
switch(x/10)
    default: printf("fail!\n");break;
    case 6,7: printf("pass!\n");break;
    case 8: printf("good!\n");break;
    case 9,10: printf("very good!\n");
```

D)

```
switch(x+y)
{
    case z:break;
    case z+3:printf("end!\n");break;
}
```

9. 某英语培训学校对不同性质的学员听课收费不同，具体规定如下：一级会员每课时 20 元；二级会员听课 10 学时及以下每学时 30 元，然后每增加一个学时收费 20 元；三级会员听课 10 学时及以下每学时 50 元，然后每增加一个学时收费 30 元。编程输入某个学员的学员类型、听课学时，并计算该学员应付的费用。

10. 采用单步和设断点的方式调试第 9 题的程序，输入多个试验数据，在程序暂停时观察表示费用的变量的值。

第 5 章 循环结构程序设计

本章的学习目标:

掌握循环结构程序的算法及其表示, 掌握循环语句的形式和用法及其特点, 会编写循环程序, 注意灵活使用循环的初始条件和终止条件, 会利用循环语句嵌套编写多重循环程序。

在结构化程序设计的 3 种基本结构中, 循环结构是应用相当广泛的一种结构。几乎所有应用程序都会涉及到重复计算问题, 重复可以通过循环结构来实现。例如求 1~100 的和, 根据前面学习的知识, 可以用“ $1+2+\cdots+100$ ”来求解, 但如果是求 1~1000 的和, 那繁琐程度就不可想象了。我们可以换个思路来考虑这个问题:

首先设置一个累加器 `sum`, 其初值为 0, 重复利用 `sum=sum+n` 来计算 (`n` 依次取 1、2、 \cdots 、100), 其具体步骤如下:

- ① 将 `n` 的初值置为 1;
 - ② 执行 `sum=sum+n`;
 - ③ `n` 的值增 1;
 - ④ 转到②去执行, 当 `n` 增到 101 时, 停止计算。此时, `sum` 的值就是 1~100 的累加和。
- 上述步骤可以通过循环结构来实现。C 语言提供了以下 4 种形式的语句来实现循环。

- 用 `for` 语句
- 用 `while` 语句
- 用 `do-while` 语句
- 用 `goto` 语句和 `if` 语句构成循环

下面我们分别对它们进行介绍。

5.1 for 语句

1. for 语句的一般语法格式

```
for([表达式 1]; [表达式 2]; [表达式 3])  
{  
    循环体语句;  
}
```

2. for 循环的执行过程

其执行过程用流程图表示如图 5-1 所示。

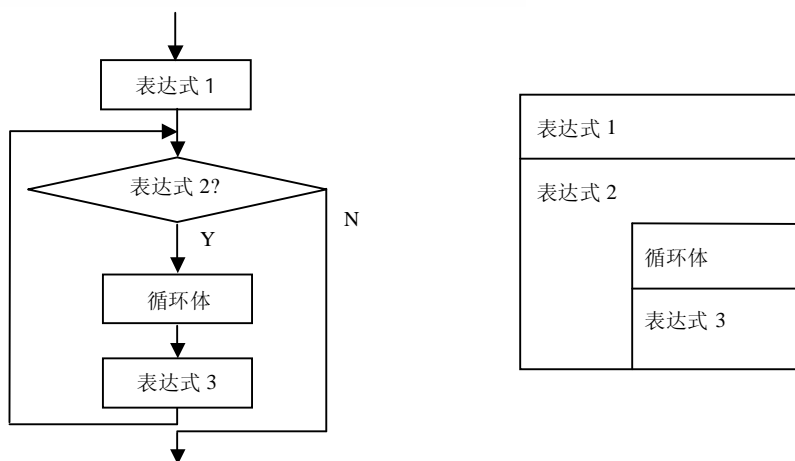


图 5-1 for 循环语句的流程图

- ① 先求解“表达式 1”；
- ② 求解“表达式 2”，若为 0 即表达式结果为“假”，则结束循环，转到⑤；
- ③ 若“表达式 2”结果为真，执行循环体，然后求解“表达式 3”；
- ④ 转回②；
- ⑤ 执行 for 语句下面的其他语句。

3. 说明

(1) 结构中的 3 个表达式均可缺省，甚至全部缺省，但其间的分号不能省略。例如，`for(;;)printf("hello world\n");` 3 个表达式全部缺省，for 循环成为一个无终止的循环(死循环)。

(2) “循环体”语句可以是单个语句、空语句或多个语句构成的复合语句。若循环体仅由一条语句构成，可以不使用复合语句形式。

(3) “表达式 1”通常是给循环变量赋初值的赋值表达式，但也可以是与此无关的其他表达式(如逗号表达式)。例如，`for(sum=0,i=1;i<=100;i++) sum += i;`。

(4) “表达式 2”是一个条件表达式，是一个逻辑量，可以是关系(或逻辑)表达式，也可以是算术(或字符)表达式等。若“表达式 2”省略，其循环条件为“真”。

(5) “表达式 3”通常用于循环体执行后对某些变量的值进行修改。例如，`for(sum=0;i<100;i+=2) sum+=i;`。

(6) for 循环的循环体可以为空，这常用来产生延时，不执行任何操作。例如，`for(i=0;i<100;i++);`。

例 5.1 求 $1+2+3+\cdots+100$ 的和。

算法如图 5-2 所示。其程序如下：

```
#include <stdio.h>
main()
{
    int i,sum=0; /*将 sum 初始化为 0*/
    for(i=1; i<=100; i++)
```

```

    sum += i;          /*实现累加*/
    printf("sum=%d\n",sum);
}

```

可以看出，用 for 结构来实现循环，程序非常简洁。

例 5.2 求 n 的阶乘 $n!$ 。

分析：

$1! = 1 * 1$

$2! = 1! * 2$

$3! = 2! * 3$

$n! = (n-1)! * n$

算法如图 5-3 所示。

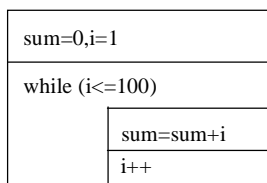


图 5-2 算法流程图

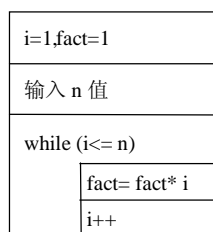


图 5-3 算法流程图

程序如下：

```

#include <stdio.h>
main()
{
    int i,n;
    long fact=1;          /*将 fact 初始化为 1*/
    printf("Enter n:\n ");
    scanf("%d",&n);
    for(i=1; i<=n; i++)
        fact *= i;        /*实现累乘*/
    printf("%d ! = %ld\n",n,fact);
}

```

请读者思考，① 若不对 fact 进行初始化，程序会不会受影响？② 若 n 数值较大，将 fact 定义为整型是否会影响最终结果？

例 5.3 编程输出 2000~2080 年之间的闰年。

分析：从例 4.1 可知，判定 year 是闰年的逻辑表达式可表示为： $year \% 4 == 0 \& \& year \% 100 != 0 \vee year \% 400 == 0$ ，如果上述逻辑表达式的结果为真，则是闰年。这里只要再设置一个循环变量控制年的变化即可。

程序如下：

```

#include <stdio.h>
main()

```




```
{  
    int year;  
    for(year=2000;year<=2080;year++)  
        if(year%4==0&&year/100!=0||year%400==0)  
            printf("%d is leap \n",year);  
}
```

5.2 while 语句

1. while 语句的一般语法格式

```
while(条件表达式)  
{  
    循环体语句组;  
}
```

2. while 语句执行过程

执行过程如图 5-4 所示。

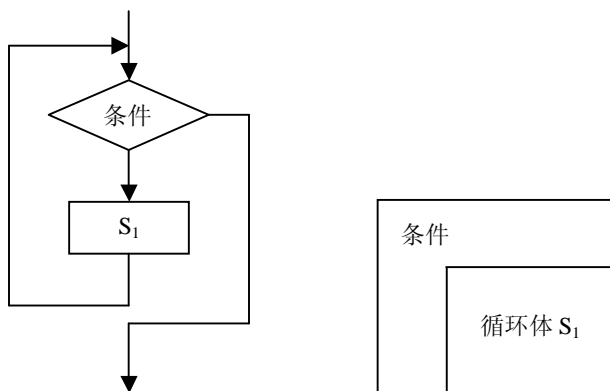


图 5-4 while 循环结构的算法描述

① 先求解“条件”表达式，如果其结果值为真（非 0），转②执行；若结果值为假，则转③执行。

② 执行循环体语句组，然后转①。

③ 执行 while 语句下面的其他语句。

其中， S_1 为循环体语句组。当条件为真时，执行 S_1 ，否则结束循环。可见，while 循环是 for 循环的一种简化形式。这里的循环体语句组同 for 循环相同，也可以是单个语句、空语句或多个语句构成的复合语句。

例 5.4 用 while 语句求 1~100 的累计和。

算法如图 5-2 所示，程序如下：

```
#include <stdio.h>  
main()
```

```

{
    int i=1,sum=0; /*初始化变量 i 和 sum, 相当于 for 循环中的表达式 1*/
    while( i<=100 ) /*条件判断, 相当于 for 循环中的表达式 2*/
    {
        sum += i; /*实现累加*/
        i++;
    } /*循环变量 i 增 1, 相当于 for 循环中的表达式 3*/
    printf("sum=%d\n",sum);
}

```

说明:

(1) 实质上, while 是将 for 循环中变量赋初值的“表达式 1”和修改循环变量的“表达式 3”位置进行了调整, 初学者应特别注意不要忘记循环变量的修正。

(2) while 循环体中可以是空语句。例如, while((ch=getchar())!='Q');该循环将简单地执行循环直至键入字符 Q。

(3) 可以在 while 条件表达式中赋值。例如, while(x=1) y=y+1;
请读者自己用 while 循环改写例 5.2 求 n! 的程序。

例 5.5 用 while 循环求解 $1! + 2! + 3! + \cdots + 100!$ 。
其算法如图 5-5 所示, 程序如下。

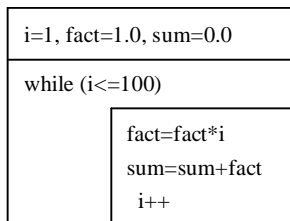


图 5-5 算法流程图

```

#include <stdio.h>
main()
{
    double sum=0.0,fact=1.0;
    int i=1;
    while(i<=100)
    {
        fact=fact * i;
        sum=sum+fact;
        i++;
    }
    printf("sum=%f",sum);
}

```



5.3 do-while 语句

1. do-while 语句的一般语法格式

```
do
{
    循环体语句组;
}while(条件表达式);    /*注意 while 后面的分号不能省略*/
```

2. 执行过程

如图 5-6 所示。

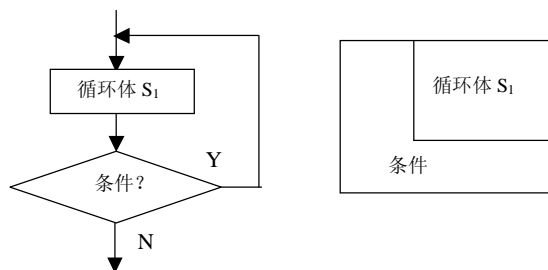


图 5-6 do-while 循环的算法流程图

- ① 先执行循环体语句组；
- ② 求解“条件”表达式。如果“条件”表达式的值为真（非 0），则转向①继续执行；否则，转向③继续执行；
- ③ 执行 do-while 下面的其他语句。

说明：

（1）do-while 循环语句的流程是先执行“循环体”语句，再在循环底部测试循环条件，即一个 do-while 循环至少要执行一次；这一点与 while 循环、for 循环是不同的。

（2）当“循环体”语句组仅由一条语句构成时，可以不使用复合语句形式。但为了提高可读性及避免与 while 混淆，一般将循环体语句用花括号括住，并把“while（条件表达式）”；直接写在循环体的“}”后面。

（3）while(条件表达式)之后的分号（;）不能省略。

例 5.6 用 do-while 语句求 1~100 的和。

其算法如图 5-7 所示，程序如下。

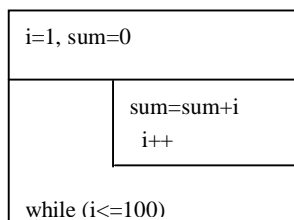


图 5-7 算法流程图

```
#include <stdio.h>
main()
{
    int i=1,sum=0; /*定义并初始化 i 和 sum*/
    do
    {
        sum += i;          /*累加*/
        i++;
    }while(i<=100); /*循环条件: i<=100*/
    printf("sum=%d\n",sum);
}
```

do-while 循环语句可以用来编写菜单程序，因为一个菜单选择程序至少要执行一次。然后在循环体的底部测试循环成立的条件，接收用户的选择是否为有效输入，例如有如下菜单程序段：

```
do
{
    printf( "please input your choice:\n");
    printf( " 1: learning English\n");
    printf( " 2: learning Math\n");
    printf( " 3: learning Chinese\n");
    printf( " 4: learning Computer\n");
    scanf("%d",&choice);
}while(choice<1||choice>4);
```

执行该程序段时，程序将循环等待直到用户输入一个有效的回答为止。

思考：请读者自己用 do-while 循环改写求 $n!$ 的程序。

例 5.7 用 3 种循环语句完成打印 Fibonacci 数列前 20 项的程序。

Fibonacci 数列问题是一个著名的古典数学问题，数列的规律为：它前两个数为 1, 1，从第 3 个数开始每个数是其前两个数之和，此数列的前几项为：1, 1, 2, 3, 5, 8, 13, 21, …。

这类问题为“递推”问题，即在一个序列中，后面一项的值与前面已知的值存在某种依赖关系，求某项的值要从前一项推算得出。由于递推关系是有一定规律的，因此采用循环语句解决比较方便。

该序列的递推关系用公式可表示为：

$$\begin{cases} f_1=1 & (n=1) \\ f_2=1 & (n=2) \\ f_n=f_{n-1}+f_{n-2} & (n \geq 3) \end{cases}$$

其算法如图 5-8 所示。

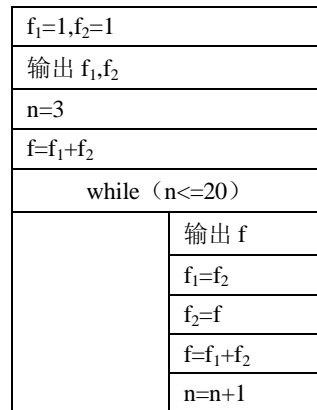


图 5-8 求 Fibonacci 数列的 N-S 图

用 for 循环语句实现的程序如下：

```
#include <stdio.h>
main()
{
    long int f1=1,f2=1,f=0;
    /* 定义并初始化数列的前两项 */
    int n=3;
    /* 定义并初始化循环变量 n */
    printf("%ld %ld",f1,f2);
    f=f1+f2;
    for(; n<=20 ; n++ )
    {
        printf("% ld ",f); /* 输出当前的 f */
        f1= f2;
        f2 = f;
        f=f1+f2;          /* 计算其余的项 */
    }
}
```

用 while 循环改写上面的程序如下：

```
#include <stdio.h>
main()
{
    long int f1=1,f2=1;
    int i=1;
    while(i<=10)          /*一次输出 2 项*/
    {
        printf("\n%ld %ld",f1,f2); /*输出当前的 2 项*/
        f1 += f2; f2 += f1;        /*计算后面 2 项*/
        i++;
    }
}
```

用 do-while 循环语句完成上面的程序如下：

```
#include <stdio.h>
```

```
main()
{
    long int f1=1,f2=1;
    int i=1;
    do
    {
        printf("\n%d %d",f1,f2);
        f1 += f2; f2 += f1;
        i++;
    }while(i<=10);
}
```

5.4 使用 goto 语句实现循环

除了上面介绍的三种循环语句之外，还有一种方法来实现循环，就是使用 goto 语句。goto 语句的一般语法格式为：

```
goto 标号
```

其功能为使程序执行顺序无条件地转向标号所在的语句行。

例如：使用 if-goto 语句实现计算 $1+3+5+7+\cdots+99$ 的程序可以如下：

```
#include <stdio.h>
main()
{
    int n=1,sum=0;
    loop: sum += n; n+=2;
    if(n<100)goto loop; /* "loop:"为语句标号 */
    printf("sum=%d\n",sum);
}
```

这里“loop:”为语句标号，标识当前所在的语句行，标号的命名遵循标识符命名规则。注意，标号定义要以冒号结尾，它标识的是程序中的一个特定位置，在程序中可以单独成行；它只能在 goto 语句中引用，在其他地方都不会被处理。

从以上例题可以看出，在这种循环结构中，需要 goto 语句和 if 语句配合使用以实现循环的退出。需要指出的是，虽然 if 和 goto 语句配合可以实现循环，但在结构化程序设计方法中，应该限制使用 goto 语句。因为过多地使用 goto 语句，会导致程序结构可读性和可维护性下降。另外，从功能上说，for 语句、while 语句、do-while 语句皆可替代 if-goto 语句的功能，所以该语句也不是必需的控制语句。尽管如此，但在有些特定情况下，使用 goto 语句会给程序设计带来方便。

在像 C 语言这样具有丰富的控制结构、又允许使用 break 和 continue 语句作为附加控制的语言中，一般很少使用 goto 语句。读者只需了解该语句的用法，编程时尽量避免使用。

5.5 多重循环

多重循环又称为循环的嵌套，即在循环语句的循环体内，又包含另一个完整的循环结构。在实际应用中，一重循环经常不能解决问题，所以循环的嵌套应用还是比较广泛的。



对所有高级语言来说，循环嵌套的概念是一样的。

C 语言提供的 3 种循环语句 for 语句、while 语句、do-while 语句都可以相互嵌套，例如以 for 循环作为外循环的二重循环形式如下（用 S 代表内循环体语句序列）：

(1) for 循环嵌套 for 循环。

```
for(表达式 11;表达式 12;表达式 13)
{
    ...
    for(表达式 21;表达式 22;表达式 23)
    {
        S;
    }
    ...
}
```

(2) for 循环嵌套 while 循环。

```
for(表达式 1;表达式 2;表达式 3)
{
    ...
    while(表达式)
    {
        S;
    }
    ...
}
```

(3) for 循环嵌套 do-while 循环

```
for(表达式 1;表达式 2;表达式 3)
{
    ...
    do
    {
        S;
    }while(表达式);
    ...
}
```

其他循环嵌套的形式可以依此类推，在此不再一一列举。



注意

在嵌套循环的使用中，被嵌套的循环可以不止一个，并且可以嵌套多层，但不论是哪种情况，内层循环和外层循环都必须是一个完整的结构，不允许有相互交叉的情况出现。如果出现图 5-9 所示的情况则是非法的嵌套，而图 5-10 所示为正确的嵌套。

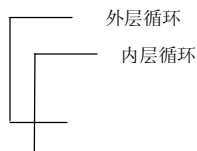


图 5-9 非法嵌套示意图

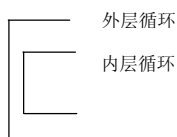


图 5-10 正确嵌套示意图

例 5.8 编程显示数字 1~9 的 1~4 次幂。

```
#include <stdio.h>
main()
{
    int i,j,k;
    long temp;
    /*打印一个表头*/
    printf("-----\n");
    printf("    i        i 的 2 次幂      i 的 3 次幂      i 的 4 次幂    \n");
    printf("-----\n");
    for(i=1;i<=9;i++)    /*外层循环控制数据 1~9 的变化*/
    {
        for(j=1;j<=4;j++)/*该内层循环控制 1~4 次幂的变化*/
        {
            temp=1;
            for(k=0;k<j;k++)/*该内层循环求得数 i 的 j 次幂 */
                temp=temp* i;
            printf("%12d",temp);
        }
        printf("\n");    /*输出一个数据的 4 次幂后换行*/
    }
}
```

循环嵌套使用时，内层循环体的执行次数等于外层循环的重复执行次数乘以每次外层循环执行时内层循环的重复次数。在上例中，外层循环执行了 9 次，第二层循环本身执行了 4 次，所以其循环体语句执行了 36 次。请读者自行分析最内层循环体的执行次数。

例 5.9 打印九九乘法表。

分析：常见的九九乘法表如下所示。观察式子的规律，可考虑使用双重循环，外循环控制被乘数的变化，内循环控制乘数的变化。

1*1=1	1*2=2	...	1*9=9
2*1=2	2*2=4	...	2*9=18
...	...		
9*1=9	9*2=18	...	9*9=81

其算法如图 5-11 所示。

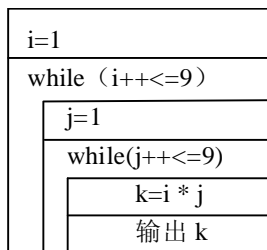


图 5-11 乘法表流程图

其程序如下：



```
#include <stdio.h>
main()
{
    int i,j,k;
    for(i=1;i<=9;i++)
    {
        for(j=1;j<=9;j++)
        {
            k=i*j;
            printf("%d*%d=%d    ",i,j,k);        /*控制输出格式*/
        }
        printf("\n");
    }
}
```

请读者练习编程打印三角形的 9-9 乘法表。

例 5.10 求 2!, 4!, 6!, 8!, 10!。

分析：前面已经介绍过求阶乘的方法，是用一个循环来实现的。本例要求 5 个整数的阶乘而且这 5 个整数是有固定规律的，即每个数之间的间隔为 2，所以设想使用循环的嵌套来解决，用外循环控制数的变化，内循环求阶乘。

如果编写如下程序：

```
#include <stdio.h>
main()
{
    int i,j;
    float fact=1.0;
    for(i=2;i<=10;i+=2)
    {
        for(j=1;j<=i;j++)    /*内循环求 i 的阶乘*/
            fact=fact*j;
        printf("%d !=%e\n",i,fact);
    }
}
```

该程序的运行结果为：

```
2!=2.00000e+00
4!=4.800000e+01
6!=3.45600e+04
8!=1.39346e+09
10!=5.05658e+15
```

可以看出，程序的运行结果并不正确。通过分析循环的执行情况可以看出，当 2! 求出以后，fact 的值为 2!；此时 i=4，执行内循环 for(j=1;j<=i;j++)fact=fact*j;的本意是求 4!，fact 的初值应为 1.0，然而此时 fact 的值为 2!，这样错误就产生了。这是初学者容易出错的地方，所以编写程序时应该仔细分析，并上机逐步追踪调试。

修改后正确的程序为：

```
#include <stdio.h>
main()
{
    int i,j;
    float fact;
    for(i=2;i<=10;i+=2)
    {
        fact=1.0;
        for(j=1;j<=i;j++)
            fact=fact*j;
        printf("%d!= %f\n",i,fact);
    }
}
```

程序的正确运行结果:

```
2!=2.000000
4!=24.000000
6!=720.000000
8!=40320.000000
10!=3628800.000000
```

5.6 break 语句与 continue 语句

在前面介绍选择结构时我们曾使用过 **break** 语句，在这里我们将讨论它在循环中的功能。另外，为了使循环控制更加灵活，C 语言还提供了 **continue** 语句。**break** 和 **continue** 语句可使应用程序从循环中非正规退出。

break 语句的一般语法格式：

```
break;
```

break 语句的功能是强行结束循环或结束当前 **switch** 选择结构，转向执行循环语句或 **switch** 开关语句的下一条语句。

continue 语句的一般语法格式：

```
continue;
```

continue 语句的功能是结束本次循环。对于 **for** 循环，跳过循环体其余语句，转向循环变量增量“表达式”3 的计算；对于 **while** 和 **do-while** 循环，跳过循环体其余语句，而转向循环条件的判定。

说明：

- **break** 能用于循环语句和 **switch** 语句中，**continue** 只能用于循环语句中。
- 循环嵌套时，**break** 和 **continue** 只影响包含它们的最内层循环，与外层循环无关。
- **continue** 语句结束本次循环，而不是结束整个循环，因此如果循环条件为真，则继续下一次循环；**break** 语句结束整个循环，不再进行循环条件的判定。

break 和 **continue** 语句对循环控制的影响如图 5-12 和图 5-13 所示。

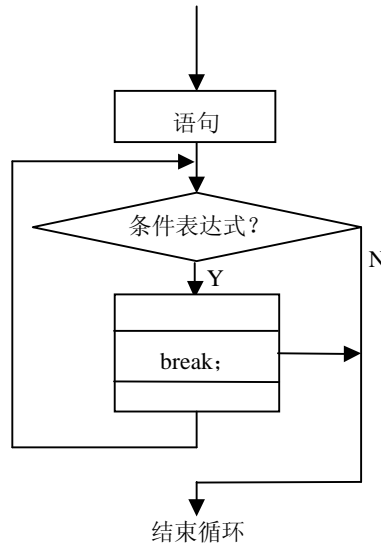


图 5-12 break 语句

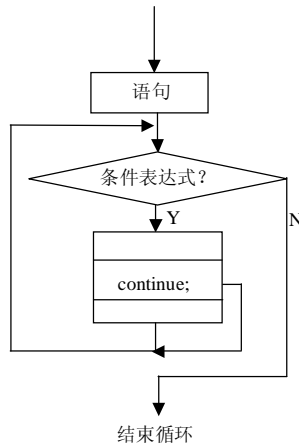


图 5-13 continue 语句

例 5.11 打印输出 1~100 之间偶数的和。

程序如下：

```
#include <stdio.h>
main()
{
    int i,sum=0;
    for(i=1;i<=100; i++)
    {
        if(i %2)continue ;
        sum=sum+i ;
    }
    printf( "sum=%d\n",sum);
}
```

例 5.12 输出 3~100 之间的全部素数。所谓素数 n 是指, 如果 n 是一个大于等于 2 的整数, 除 1 和 n 之外, 不能被 2~($n-1$) 之间的任何整数整除。

分析: 判断一个数是否能被另一个数整除, 可通过判断余数是否为 0 来实现。算法如图 5-14 所示。

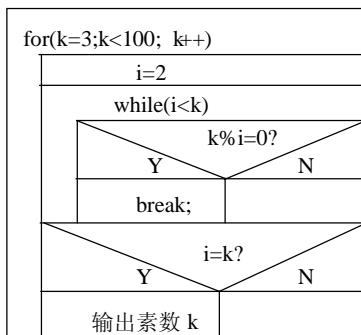


图 5-14 算法流程图

程序如下:

```
#include <stdio.h>
main()
{
    int k,i,counter=0;
    for( k=3; k<100; k+=2)/*外循环控制要判定的整数范围, 偶数非素数故排除*/
    {
        for(i=2; i<=k-1; i++)/*内循环判断整数 k 是否是素数*/
            if(k%i==0) /*k 不是素数*/
                break; /*退出内循环, 执行下面的 if 语句*/
        if( i == k ) /*说明 k 不能被 2~k-1 的整数整除, 所以 k 是素数*/
        {
            printf("%d ",k);
            counter++;
        }/*统计素数的个数*/
    }
    printf("\ncounter=%d\n",counter);
}
```

在该程序中, 外循环控制变量 k 的初值从 3 开始、增量为 2, 从而排除了偶数, 减少了循环次数, 提高了运行速度, 优化了程序。

请读者考虑, 对于这个程序还有没有更好的优化方法? (可从减少循环次数入手)

例 5.13 计算键盘输入的一批学生数据总人数、总分数、平均分。键盘输入的分数范围为 0~100, 当输入数据超出范围时, 表示输入结束。

```
#include <stdio.h>
main()
{
    int n=0; /*表示人数*/
    float score_sum=0;
```



```
float score_aver;
float score;          /*单个分数*/
do
{
    scanf("%f",&score);
    if( score<0.0 || score>100.0)
        break;
    score_sum+=score;
    n++;
}while(1);
score_aver=score_sum/n;
printf("the number of students is :%d\n",n);
printf("total score is :%.2f\n" ,score_sum);
printf("average score is :%.2f\n",score_aver);
}
```



提示

循环程序在调试时,可以用 F7 跟踪程序的执行过程,或者加一些输出语句来输出程序的中间结果以帮助分析程序,程序调试成功后再把多余的输出语句删除。如果程序是死循环无法正常结束,可以按 Ctrl+C 键或 Ctrl+Break 键使程序强行退出。

5.7 几种循环的关系与比较

在 C 语言的 3 种循环语句中,对于同一个问题,使用任一语句都可以,即 3 种循环语句可以互换。其中 for 语句最为灵活,不仅可用于循环次数已经确定的情况,也可用于循环次数虽不确定、但给出了循环继续条件的情况。

while 和 do-while 语句的循环变量初始化是在循环语句之前完成的,循环变量的修改是在循环体中完成的。for 语句中的第一个表达式可对变量进行初始化,第三个表达式可完成循环变量的修改,不需用户另写语句。

while 循环和 for 循环在执行流程时都是“先判断,后执行循环体”。do-while 语句则是先执行一次循环体语句,再判断条件,它比较适用于处理不论条件是否成立,先执行 1 次循环体语句的情况。除此之外,do-while 语句能实现的,for 语句和 while 语句也能实现,而且可能更简洁。

几种循环可以互相嵌套使用。

for 循环、while 循环和 do-while 循环都能使用 continue 语句结束本次循环,使用 break 语句终止循环。

在嵌套的循环结构中,可以利用 goto 语句由内层循环体转到外层循环体或外循环外,但不能从外循环外或内循环外转到内循环内部。

5.8 应用程序举例

例 5.14 计算并输出一个整数各位数字之和。如: 1234, 各位之和是 $1+2+3+4=10$ 。

分析：要计算各位数字之和，必须将该整数的每一位数字求出来，可考虑运算符“%”的使用，如 $1234\%10$ 的结果为个位数 4。其算法如图 5-15 所示。

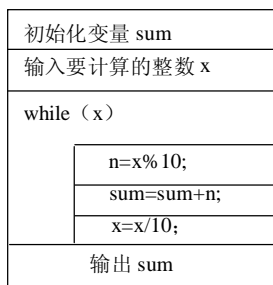


图 5-15 算法流程图

其程序如下：

```
#include <stdio.h>
main()
{
    int x,n,sum=0;
    scanf("%d",&x);
    while(x)
    {
        n=x%10; /*通过余数求得每一位数字*/
        sum=sum+n; /*求和*/
        x=x/10; /*得到商部分*/
    }
    printf("sum=%d\n",sum);
}
```

例 5.15 编写一个程序，从键盘输入字母，如遇小写字母则变为大写字母输出。

分析：小写字母变为大写字母，需要了解它们之间 ASCII 码的关系，如小写字母 a 的 ASCII 值为 97，大写字母 A 的 ASCII 值为 65，有关系“a”-“A”=32 存在。若输入 10 个字符后退出，其程序如下所示：

```
#include<stdio.h>
main()
{
    int i; char ch;
    for(i=0;i<10;i++)
    {
        ch=getchar(); /*接收一个字符*/
        if(ch>='a'&&ch<='z') /*判定是否为一个字母*/
            ch=ch-32; /*将小写字母转换为大写字母*/
        putchar(ch);
    }
}
```

例 5.16 编程求 e 的近似值，直到最后一项小于 $1e-6$ 为止。



已知近似计算公式： $e=1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\dots+\frac{1}{n!}$ 。程序如下。

```
#include <stdio.h>
main()
{
    float s;
    int i,n ;
    i=1;
    n=1;
    s=1.0;
    while(1.0/n>1e-6)
    {
        n=n*i;
        s=s+1.0/n;
        i++;
    }
    printf("%f\n",s);
}
```

例 5.17 将 1 元钱换成 1 分、2 分、5 分的硬币有多少种方法？请编程求解。

分析：设 x 为 1 分硬币数， y 为 2 分硬币数， z 为 5 分硬币数，则有如下方程：

$$x+2*y+5*z=100$$

可以看出，这是一个不定方程，没有惟一的解。这类问题我们无法使用解析法求解，只能将所有可能的 x 、 y 、 z 的值一个一个地去试，看是否满足上面的方程，如满足则求得一组解。这种方法称为“穷举法”。这种方法在数学中应用比较广泛，在前面学过的求素数的例题中，程序也是采用此种方法。使用穷举法的关键是确定正确的穷举范围：如果穷举的范围过大，则程序的运行效率将降低。

分析问题可知，最多可以换出 100 个 1 分硬币，最多可以换出 50 个 2 分硬币，最多可以换出 20 个 5 分硬币。所以 x 的可能取值为 0~100， y 的可能取值为 0~50， z 的可能取值为 0~20。据此可以恰当地确定穷举范围。

使用 3 重 for 循环，编写程序如下：

```
#include <stdio.h>
main()
{
    int x,y,z;
    int count=0;
    for(x=0; x<=100;x++)
        for(y=0;y<=50;y++)
            for(z=0;z<=20;z++)
                if((x+2*y+5*z)==100.0)
                {
                    printf("x=%d,y=%d,z=%d\n",x,y,z);
                    count++;
                }
    printf("there are %d methods",count);
}
```

}

例 5.18 使用迭代法求方程 $x^2 - 21.634434 = 0$ 的近似根，给定初值 $x_1 = 2.5$ ，精度要求为 10^{-5} 。迭代法的关键是确定迭代公式、迭代的初始值和精度要求。牛顿切线法是一种高效的迭代法，它的实质是以切线与 x 轴的交点作为曲线与 x 轴交点的近似值以逐步逼近解，如图 5-16 所示。

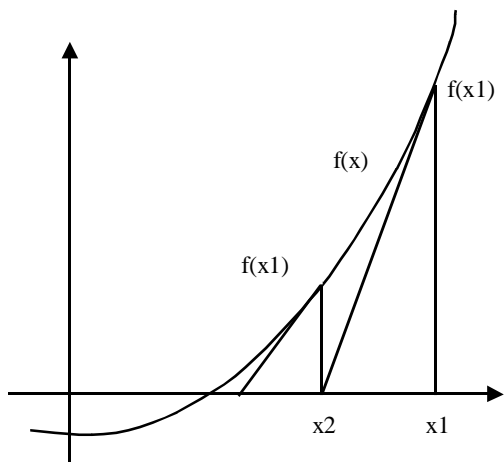


图 5-16 牛顿迭代法

方程 $f(x)=0$ 的等价形式为：
$$x = x - \frac{f(x)}{f'(x)} \quad (f'(x) \neq 0)$$

迭代公式为：
$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (i=0, 1, 2, \dots)$$

其中 $f'(x_i)$ 为 $f(x)$ 的一阶导数。

分析：已知 $f(x) = x^2 - 21.634434$ ，则 $f'(x) = 2x$ ，迭代公式为： $x_{i+1} = x_i - (x_i^2 - 21.634434) / (2x_i)$
程序如下：

```
#include <math.h>
main()
{
    float x1,x2,a;
    int i;
    x1=2.5;
    a=21.634434;
    x2= x1-(x1*x1-a)/(2*x1);
    for(i=0;fabs(x2-x1)>1e-5&& i<10;i++)/*使用 i<10 限制迭代次数*/
    {
        x1=x2;
        x2= x1-(x1*x1-a)/(2*x1);
    }
    printf("x=%f\n",x2);
}
```




可以看出,该程序实际上是求 a 的平方根,使用变量 $x1$ 存放迭代过程中 x_i 的值,变量 $x2$ 存放 x_{i+1} 的值。

例 5.19 用二分法求方程 $x^3-3x^2+x+1=0$ 在区间 $[-7,7]$ 上的近似根,允许误差为 10^{-6} 。

分析:如图 5-17 所示,若函数有实根,则函数与 x 轴应有交点,在交点的左右区间内,函数值的符号相反,即异号。利用这一原理,逐步缩小异号区间,如果区间的长度小于给定的精度要求,则认为求得一个根。设区间为 $[x1,x2]$,其具体步骤如下:

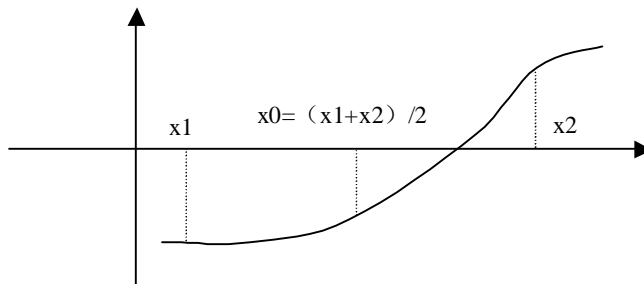


图 5-17 二分法求方程的根

- ① 判断 $f(x1)*f(x2)$ 是否小于 0,若是则进行第②步骤。
- ② 取区间 $[x1,x2]$ 的中点 $x0=(x1+x2)/2$;
- ③ 若 $f(x0)=0$,则 $(x1+x2)/2$ 为方程的根;
- ④ 否则,若 $f(x0)$ 与 $f(x1)$ 同号,则区间为 $[x0,x2]$;若 $f(x0)$ 与 $f(x1)$ 异号,则区间为 $[x1,x0]$;
- ⑤ 重复②~④,直到达到所要求精度为止。

程序如下:

```
#include <stdio.h>
#include <math.h>
main()
{
    float x0,x1,x2,y0,y1,y2;
    do
    {
        printf("Please input x1  x2 ");
        scanf("%f%f",&x1,&x2);
        y1=x1*x1*x1-3*x1*x1+x1+1;
        y2= x2*x2*x2-3*x2*x2+x2+1;
    }while(y1*y2>0);
    do
    {
        x0=(x1+x2)/2;
        y0= x0*x0*x0-3*x0*x0+x0+1;
        if((y0*y1)<0)
        {
            x2=x0;
            y2=y0;
        }
        else
```

```
{
    x1=x0;
    y1=y0;
}
}while(fabs(y0)>=1e-6);
printf("The root is %f\n",x0);
}
```

要点回顾

1. 循环结构是程序设计中应用最多的结构形式，循环语句 **for**、**while**、**do** 可以使程序段重复多次执行。其中 **for** 和 **while** 语句先判断条件，条件成立才执行循环体；**do** 语句先执行循环体再判断条件，可以使循环体至少执行一次。3 种循环语句的一般格式如下：

```
for([表达式 1]; [表达式 2]; [表达式 3])
{
    循环体语句;
}
while(条件表达式)
{
    循环体语句组;
}
do
{
    循环体语句组;
}while(条件表达式);    /*注意 while 后面的分号不能省略*/
```

许多情况下，3 种循环语句可以互相转换，其中 **for** 语句使用频率较高。

2. 有时候需要用到多重循环，又称为循环嵌套，即在循环语句的循环体内又包含另一个完整的循环结构。注意，循环嵌套不允许交叉。

3. 为了避免出现无终止的循环，程序设计者要注意循环结束条件的使用，也就是说在循环执行中，要修改循环变量；还要注意循环的初始条件。分析循环第一次和最后一次执行时的情况有助于写出正确程序。如果程序执行时出现了死循环无法而正常结束，可以用 **Ctrl+C** 键或 **Ctrl+Break** 键强行退出。

4. 可以用 **goto** 语句和 **if** 语句相配合实现循环，但 **goto** 可以任意改变程序流程，要慎用。**break** 可以从循环中退出，**continue** 可以跳过其后的语句直接转去判断循环条件，**return** 可以直接退出函数。这几个语句可以使循环的执行和退出更为灵活，但是不符合结构化的程序设计思想，建议少用。

5. 对典型循环问题，如累加求和、求数的阶乘、穷举法解方程，二分法解方程等，在设计算法时，要先找出问题中的规律。

习 题

1. 利用 3 种循环方法编写程序，求 1~n 之间的偶数之和、奇数之和。



2. 用 3 种循环语句编写求 s 的程序, 已知 $s=1+2^3+3^3+4^3+\Lambda+n^3$ 。
3. 在 $3\sim n$ 之间找出不能被 3、5、7 整除的数。
4. 输入两个整数 M 和 N , 计算并输出它们的最大公约数和最小公倍数。



提示

采用欧几里德算法。用 M 除以 N 算出余数; 如余数为零, 则最大公约数就是 N ; 如果余数不为零, 则以 N 代替原 M , 余数代替原 N 。再继续上述运算, 直至出现余数为零, 此时 N 为最大公约数。

5. 找出 $1\sim 100$ 之间的 3 个不相同的整数 x 、 y 、 z , 它们之间存在关系 $x*y*z=x+y+z$ 。
6. 计算 sum 的值, 直至 $|S_n-S_{n-1}|<10^{-5}$

$$\text{sum}=1+\frac{1}{2}+\frac{1}{4}+\frac{1}{7}+\frac{1}{11}+\frac{1}{16}+\frac{1}{22}+\Lambda+s_{n-1}+s_n$$

7. 输出九九乘法表, 要求打印格式为如下所示的三角形。

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

8. 打印所有的“水仙花数”。所谓“水仙花数”是指一个三位正整数, 其各位数字立方和等于该数本身。例如: $153=1^3+5^3+3^3$, 所以 153 是一个水仙花数。

9. 编程求 $S_n=a+aa+aaa+\cdots+\overbrace{aa\cdots a}^{n \uparrow a}$, 其中 a 是一个从键盘输入的数。
10. 写出程序运行结果。

A)

```
#include <stdio.h>
main()
{
    int i,j,s,m;
    s=0;m=0;
    for(i=1;i<3;i++)
    {
        s++;
        for(j=1;j<5;j++)
            if(j%2)
```

```
        continue;
    else
        m++;
    s=s+s+m;
}
printf("\n s=%d",s);
}
```

B)

```
#include<stdio.h>
main()
{
    int i,j,k;
    for(i=3;i>=1;i--)
    {
        for(j=1;j<=4-i;j++)
            printf(" ");
        for(k=1;k<2*i;k++)
            printf("*");
        printf("\n");
    }
}
```

C)

```
main()
{
    int sum,i,k;
    sum=10;
    for(i=8;i>=1;i--)
    {
        k=i/2;
        sum=sum+k;
    }
    printf("%d",sum);
}
```

D)

```
main()
{
    int y=9;
    for(;y>0;y--)
    {
        if(y%3==0)
        {
            printf("%d",--y);
            continue;
        }
    }
}
```



11. 分析程序段，并选择正确答案。

(1) 有以下程序段：

```
int k=0;
while(k=1)k++;
```

while 循环执行的次数是 ()。

A) 执行 1 次 B) 有语法错，不能执行 C) 一次也不执行 D) 无限次

(2) 有以下程序段：

```
int x=3;
do
{
    printf("%d",x-=2);
}while(!(--x));
```

其输出结果是 ()。

A) 1 B) 3 0 C) 1 -2 D) 死循环

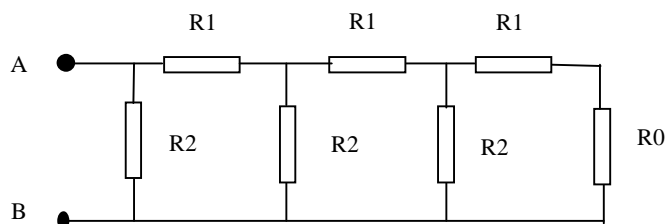
(3) 有以下程序段：

```
main()
{
    int i=1,sum=0;
    while(i<10)
        sum=sum+1;
    i++;
    printf("i=%d,sum=%d",i,sum);
}
```

其输出结果是 ()。

A) i=10,sum=9 B) i=9,sum=9 C) i=2,sum=1 D) 运行出现错误

12. 有一电路如下图所示，R0，R1 和 R2 由键盘输入，求 A、B 两点间的等效电阻。



13. 现有以下语句：

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

与上述语句序列等价的有哪些？

A)

```
for(i=1;;i++)
{
    sum+=i;
    if(i==100)
        break;
}
```

B)

```
for(i=1;i<=100;)
{
    sum+=i;
    i++;
}
```

C)

```
i=1;
for(;i<=100;)
{
    sum+=i;
}
```

14. 根据题意填空。计算 1~20 之间所有奇数之和与偶数之和。

```
#include <stdio.h>
main()
{
    int a,b,c,i;
    a=c=0;
    for(i=0;i<20;i+=2)
    {
        a+=i;
        _____;
        c+=b;
    }
    printf("偶数之和=%d\n",a);
    printf("奇数之和=%d\n",c);
}
```



第 6 章 函 数

本章的学习目标:

掌握 C 程序的模块化结构,掌握函数的定义和声明的含义,会进行函数的定义和声明,了解变量的存储类别及其作用域和生存期的概念,掌握函数的调用过程及函数调用时参数的传递。会利用多函数组成程序,也可以将多函数分别编译,再组成一个项目进行连接、运行。

6.1 C 程序的模块结构

C 语言是函数式语言,一个 C 程序由一个或多个函数组成,其中必须有一个名为 `main` 的主函数。主函数是整个程序的控制部分,在主函数执行过程中可以调用其他函数并允许嵌套调用,其他函数是 C 语言库函数或用户自定义函数,能实现特定的功能。

一个 C 程序的执行,总是从主函数的最外层左花括号“{”开始,依次执行花括号内的所有语句,直到主函数的最外层右花括号“}”程序才执行结束;而其他函数,只有在被调用时才能执行。图 6-1 中的 C 程序由一个主函数和 7 个其他函数组成;在执行 `main()` 时,调用了 `fun1()`、`fun2()` 和 `fun3()`;而在执行 `fun1()`、`fun2()` 和 `fun3()` 时,又分别调用了 `fun4()`、`fun5()`、`fun6()` 和 `fun7()`。

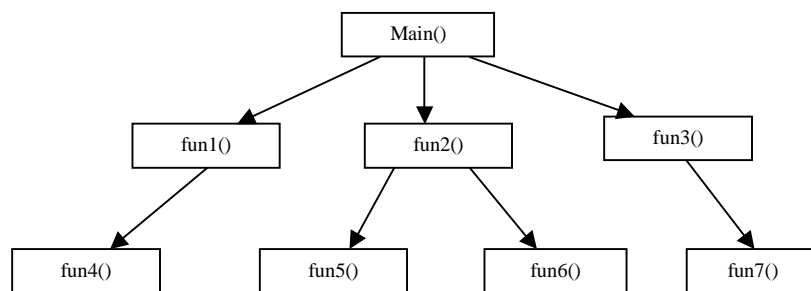


图 6-1 C 语言程序的模块结构

C 程序的一般格式如下:

```
全局变量定义或声明
main()
{
    局部变量定义
    语句序列
}
<类型标识符> f1(形式参数表)
```

```
{  
    局部变量定义  
    语句序列  
}  
<类型标识符> f2(形式参数表)  
{  
    局部变量定义  
    语句序列  
}  
.  
.  
.  
<类型标识符> fn(形式参数表)  
{  
    局部变量定义  
    语句序列  
}
```

其中 f1()~fn()表示用户定义的函数。

C 编译系统提供了很多非常有用的库函数，程序设计者可根据需要进行调用，但调用前要将相应的标题文件（又称头文件）包含到程序中来。前面程序举例中的第一行都是“#include”，这是一条预编译命令，作用是将 C 编译程序提供的标题文件包含到当前程序中。程序中用到的库函数 scanf()和 printf()都是在标题文件 stdio.h 中定义的（标题文件是未编译的源代码文件）。库函数不是 C 语言本身的组成部分，而是由 C 编译系统提供的一些非常有用的功能函数。库函数是编译过的文件。例如，C 语言没有输入输出语句，也没有直接处理字符串的语句，但是 C 编译系统以库函数的方式提供了这些功能。另外，还有大量的数学函数及其他函数可供用户直接调用。这些库函数的类型和宏定义都保存在相应的标题文件中，而对应的子程序则存放在运行库(.lib)中，用户只要在程序的函数外部用“#include <标题文件>”或“#include "标题文件"”包含指定的标题文件，就可以调用相关的库函数。#include 是预编译命令，位置一般在其他语句的前面。

Turbo C 常用的标题文件有：

- stdio.h 标准输入输出函数（ANSI C）
- math.h 数学函数（ANSI C）
- string.h 字符串处理函数（ANSI C）
- malloc.h 内存分配函数（ANSI C）

绝大多数 C 程序都包含对标准函数库的调用。一般程序都要包含 stdio.h 标题文件；如果要使用数学函数，还要包含 math.h，如果要处理字符串，还要包含 string.h。只有包含相应的标题文件以后，才可以使用特定的库函数。调用库函数的时候，还要注意函数形式参数个数、类型以及函数返回值的类型。

函数库是一种重要的软件资源，读者应在学习 C 语言本身的同时，逐步了解和掌握各种库函数的功能和用法。充分利用它们，可以提高编程效率。不同的 C 编译系统提供的库函数在数量、种类、名称及使用上都有些差异，甚至连标题文件的名称也不一定相同。



例如, ANSI C 标准建议的有 100 多个库函数, 而 Turbo C 在 ANSI C 标准建议的 100 多个库函数的基础上扩充到 374 个。但就 ANSI C 建议的 15 类标准函数而言, 各类 C 编译系统大致都是相同的。

用户定义的 C 函数, 从功能上讲可分为两类: 主函数和子函数。主函数有固定的名称 `main`, 通常描述程序的总体框架; 子函数由用户自己命名, 一般完成某种特定的子功能。一个函数必须在函数定义或函数声明后才能被调用。

6.1.1 函数定义方法和函数的形参

定义函数就是要求在编译时建立一个函数实体。函数定义的一般形式为:

```
[类型标识符] <函数名>([<形式参数说明>])  
{  
    <说明部分>  
    <语句部分>  
}
```

说明:

(1) 类型标识符。类型标识符指明函数返回值的类型。函数返回值由 `return` 语句实现, `return` 语句的格式为:

```
return 表达式;
```

函数在返回前, 先将表达式的值转换为所定义的类型, 再返回到主调函数中的调用表达式。如果函数定义时不指明类型, 系统隐含指定为 `int` 型。对于无返回值的函数, 应定义为空 (`void`) 类型, 并且可省略最后的 `return` 语句。

(2) 函数名。主函数有固定的名称 `main`, 通常包括了整个程序的轮廓, 由它再调用其他函数。其他函数则可以根据标识符的命名方法任意取名。函数名后的一对圆括号是函数的象征。

(3) 形式参数。又称形参、虚参或哑元。形式参数写在函数名后面的一对圆括号内, 它有两个作用: 其一表示将从主调函数中接收哪些类型的信息, 多个形式参数之间应以逗号分隔; 一个函数也可以没有形式参数, 此时圆括号中为空, 但圆括号不能省略。其二在函数体中形式参数可以被引用, 可以从键盘输入、被赋以新值或参与运算。

程序进行编译时, 并不为形式参数分配存储空间, 当函数被调用时, 才分配存储空间。形参从相应的实参得到值, 称为“虚实结合”。当调用结束, 流程返回 `main` 函数时, 形参所占空间也被释放了。所以形参只是在函数被调用范围内有效。

形式参数的名字并不重要, 关键在于它们的数量及类型。只要类型与数量确定了, 程序员便可以自己选择一些合适的标识符来做形参名。

(4) 函数体。函数名后花括号内的部分称为函数体。它主要有两大部分: 第一部分是本函数内部用到的局部变量定义 (C 语言中, 所有的变量都要先定义后使用, 在函数体中定义的变量只有在执行该函数时才存在), 第二部分是语句序列, 完成本函数的功能。

函数体中也可以不定义变量, 而只有语句, 也可以二者皆无。如:

```
void null(void)
{
}
```

这是一个空函数，调用它不产生任何有效操作，但却是一个符合 C 语言语法的合法函数。在模块化设计中，往往先把 **main** 函数写好，并预先确定需要调用的函数，有时一些函数还未编写好，可以用空函数形式放在程序中表示程序的总体框架，然后先调试程序的其他部分，以后再逐步补上。

例 6.1 函数定义举例。

```
#include <stdio.h>
main()
{
    long sum(int);          /* 做函数显式声明 */
    printf("%d,%d\n",100,sum(100));
}
long sum(int x)
{
    int i;
    long s=0;
    for(i=1;i<=x;i++)
        s=s+i;
    return s;
}
```

程序运行结果为：100，5050。



注意

这里所讲的是 ANSI C 标准的函数定义格式。还有一种老的 K&R 函数定义格式，现在不提倡，但是在一些已有的应用程序或老的参考书中仍然出现，其形式如下：

```
long sum(x)int x;
{
    int i;
    long s=0;
    for(i=1;i<=x;i++)
        s=s+i;
    return s;
}
```

可见，K&R 标准的参数说明部分是在括号里写出参数名，而参数类型的定义由另外一个语句实现，并且形参类型定义的语句出现在函数体前。

6.1.2 函数声明与函数原型

在主调函数中，要对在本函数中将被调用的函数事先做声明。所谓“函数声明”，是指向编译系统提供必要的信息：函数名，函数的类型，函数参数的个数、类型及顺序。编译



系统以函数声明中的信息为依据，对调用表达式进行检测，以保证调用表达式与函数之间的参数正确传递，如检测形参与实参类型是否一致、函数返回值的类型是否正确。如例 6.1 中主函数 `main()` 的第一条语句：`long sum(int);` 就是函数声明语句。

函数声明的一般格式为：

```
类型标识符 函数名 (类型标识符 形参, 类型标识符 形参, ...);
```

这些信息就是函数定义中的第一行（称函数头）的内容，也称函数原型或函数模型。设某一函数的定义为：

```
double func(double a,int b,float c){函数体}
```

正确完整的函数原型应为：

```
double func(double x,int y,float z); /*注意末尾的分号*/
```

函数原型的作用是通知编译系统要传递给函数的参数个数和类型以及函数返回值的类型。本例中的函数原型就表示 `func` 函数有 3 个参数：第一个参数是双精度型，第二个参数是整型，第三个参数是浮点型；同时函数返回值的类型是双精度型。

这里形参的名字是不重要的，重要的是类型标识符及顺序。函数声明也可以写成：

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

编译系统将默认实参与形参是匹配的。当实参与形参类型不同时，就按一般赋值规则进行转换，而不发出错误信息。但这种情况下，数据传递容易发生错误，并且错误不易检查，所以函数调用时要注意使实参和形参类型一致。

在下列情况下可以省略函数声明：

- 函数调用是在同一文件中且位于该函数定义之后。也就是被调函数定义写在前面，主调函数写在后面。
- 函数返回值为 `int` 类型。

但是为了便于阅读和理解，应当养成在调用之前都做函数显式声明的习惯。当一个函数要被一个文件中的多个函数调用时，可以将该函数声明写在所有函数之前。下面是一个函数声明与定义之间对应关系的例子。

```
float f(float x ,int y); /*放在所有函数之前，做统一声明*/
main()
{
    ...
}
fun1()
{
    ...
    f(a,b);          /*调用 f 函数 */
    ...
}
fun2()
{
```

```
...
    f(c,d);          /*调用 f 函数 */
}
float f(float x ,int y) /*函数定义*/
{
    ...
}
```

函数定义与函数声明的区别很明显：函数定义以函数体结尾，而函数声明不包含函数体；函数定义要求分配内存单元，用来存放经编译后的函数指令，而函数声明不要求分配空间。

6.2 函数调用时参数值的传递

6.2.1 函数的传值调用

函数调用由函数调用表达式实现，即：

```
<函数名>([<实际参数表>])
```

函数调用表达式可以出现在表达式可以出现的任何位置。例如：

```
printf("hello, world\n"); /*函数调用表达式后加分号组成函数调用语句*/
y=sqrt(x);                /*函数调用出现在赋值表达式的右边*/
printf("%f", sqrt(5.0));  /*出现在输出语句中作为输出项*/
y=sqrt(fabs(-5.0));       /*作为实参出现在其他函数调用中*/
```

参数是函数调用时的信息载体，主调函数和被调函数之间的数据传递可以通过实参与形参的结合实现。在 C 程序中，实参与形参是按传值方式结合的，也称为传值调用方式。传值调用的过程是：

(1) 形参与实参各占一个独立的存储空间。

(2) 形参的存储空间是函数被调用时才分配的。调用开始时，系统为形参开辟一个临时存储区，然后将各实参之值传递给形参。这种虚实结合方式是单向的，称为“值结合”。

(3) 函数返回时，由 `return` 语句带回函数值，形参的临时存储被撤消。

传值调用方式的函数只有一个入口（实参传值给形参）和一个出口（函数返回值）。函数受外界影响减小到最低程度，从而保证了函数的独立性。但这种“值结合”方式只能从函数返回一个值，有时是不够用的（在介绍指针以后就能用其他方法解决此问题）。

例 6.2 函数参数的传递（1）。

```
#include <stdio.h>
long power(int x,int n); /*放在所有函数之前，做统一声明*/
main()
{
    int w=8;
    long r;
    r=power(w,2); /*函数调用*/
    printf("the result is %ld\n",r);
}
```



```
long power(int x,int n) /*函数定义*/
{
    int i;
    long p=1;
    for(i=1;i<=n;i++)
        p*=x;
    return p;
}
```

运行结果为: the result is 64。

传值调用的特点是: 函数中对形参变量的操作不会影响到调用函数中的实参变量, 即形参值不能传回给实参。

例 6.3 函数参数的传递 (2)。

```
#include <stdio.h>
main()
{
    void swap(int,int);
    int a=10,b=5;
    printf("a=%d,b=%d\n",a,b);
    swap(a,b);
    printf("a=%d,b=%d\n",a,b);
}
void swap(int x,int y)
{
    int temp ;
    temp=x,x=y,y=temp;
    printf("x=%d,y=%d\n",x,y);
}
```

执行结果为:

```
a=10,b=5
x=5,y=10
a=10,b=5
```

这里, swap 函数的功能是交换两个参数的值。但运行的结果表示, 它只交换了函数 swap 中的两个形参变量 x 和 y 的值, 而没有交换 main() 中实参 a 与 b 的值。

在函数调用过程中, 为了能使传值正确地进行, 必须注意实参与形参在个数、类型上要匹配, 否则容易出错, 错误还不易被发现。

例 6.4 函数调用时实参与形参在个数、类型上不匹配的情况举例。

```
#include <stdio.h>
main()
{
    float add(unsigned int,unsigned int);
    float x=1.5,y=-5.7;
    printf("%f+%f=%f\n",x,y,add(x,y)); /*实参是浮点型*/
}
```

```
float add(unsigned int a,unsigned int b)/*形参是无符号整型*/
{
    printf("a= %u,b=%u\n",a,b);
    return(a+b); /*返回整型值，与函数类型不匹配*/
}
```

当实参和形参的类型不一致时，虚实结合按一般赋值规则进行转换。本例中，实参为实型，而形参为无符号整型，虚实结合后形参 a,b 的值并不是 1.5 和 -5.7。返回值 a+b 为无符号整型，但函数类型为实型，所以 add(x,y)得到的值也不正确。运行结果如下：

```
a=1,b=65531
1.500000+ -5.700000=65532.000000
```

显然是错误的。

6.2.2 函数的返回

函数执行的最后一个操作是返回。返回的作用有两个：

(1) 使流程返回主调函数，宣告函数的一次执行终结，在调用期间所分配的变量单元被撤消。

(2) 将函数值传到调用表达式中。但是这一点并非是必需的，有些函数有返回值，有些函数可以没有返回值。函数返回值也是函数间传递数据的一种手段。

函数返回值可以是整型的、实型的、双精度型的和字符型的数据，通过 return 语句将这些数据送回主调函数。

例 6.5 求一个整数绝对值的函数。

```
#include <stdio.h>
int absolute(int);
main()
{
    int x=-100;
    printf("%d,%d\n",x,absolute(x));
    x=100;
    printf("%d,%d\n",x,absolute(x));
}
int absolute(int x)
{
    return(x>=0? x:-x); /*返回三项条件运算表达式的值*/
}
```

运行结果为：

```
-100, 100
100, 100
```

这个函数的 return 语句返回一个表达式的值。如果 return 语句后面不带表达式，则不返回任何值，仅将流程转回主调函数。

例 6.6 打印 n 个星号 (*) 的函数。



```
#include <stdio.h>
void star(int);
main()
{
    star(10);
}
void star(int n)
{
    int i;
    printf("\n");
    for(i=0;i<n;i++)
        printf("*");
    return;
}
```

运行结果:

```
*****
```

这个函数只执行打印 n 个*的操作，不返回任何值到调用函数中，所以用 `void` 定义它的类型，表示函数是无返回值的。

C 语言中，当不带表达式的 `return` 语句位于函数体的最后时，允许将该 `return` 省略，用作函数体结束的右花括号也会将流程返回主调函数，所以上例中的 `return` 语句可去掉。

应当说明，如果函数类型不定义为 `void` 型，则函数类型默认为整型。即使在 `return` 后面不带表达式，或者省略 `return` 语句，函数也是有一个返回值的，只是返回值不确定。

上例中如果不定义 `star` 函数为 `void` 型，当在主调函数中有：

```
printf( "%d",star(10));
```

在输出 10 个*的同时，能输出 `star` 函数的返回值，但是这个值无意义。

为了防止误用函数返回值(例如，将一个不需返回值的函数当作有返回值的函数使用)，需要将函数定义为 `void` 类型，即无返回值类型。所以上面函数定义为：

```
void star(int n)
```

在调用它时不得使用其函数值。这时如果有 `printf(" %d ",star(10))`这样的引用，在编译时将给出错误信息。

在一个函数中允许有一个或多个 `return` 语句，流程执行到其中一个 `return` 时即返回主调函数。在一个函数中允许使用选择结构形成函数的多个条件出口。各个 `return` 语句在是否返回值以及返回值的类型上要保持一致。

例 6.7 一个函数可以有多个 `return` 语句。

```
#include <stdio.h>
int y(float);          /*函数声明*/
main()
{
    printf("%d,%d\n",-100,y(-100));
    printf("%d,%d\n",0,y(0));
}
```

```
printf("%d,%d\n",100,y(100));
}
int y(float x)
{
    if(x>0)
        return(1);
    else if(x<0)
        return(-1);
    else
        return(0);
}
```

运行结果为:

```
-100,-1
0,0
100,-1
```

虽然允许在一个函数中有多个 `return` 语句,但一般还是写成一个函数只有一个 `return` 语句的形式,即一个函数有一个入口,也只有一个出口,以保证程序完整的模块结构。如例 6.7 中函数 `y` 可以改写为:

```
int y(float x)
{
    int z;
    if(x>0)
        z=1;
    else if(x<0)
        z=-1;
    else
        z=0;
    return(z);
}
```

6.3 局部变量和全局变量

6.3.1 局部变量

我们前面用到的变量都是局部变量,又称内部变量,只在定义它的函数或复合语句内有效。也就是说,当局部变量所在的函数或复合语句被执行时,该局部变量被分配存储单元;当函数或复合语句结束时,该局部变量便消失。函数的形式参数是局部变量,只在本函数段内有效,当调用结束时,形式参数消失。

例如考虑下面两个函数:

```
void func1(void)
{
    int x;
    x=10;
    ...
}
```




```
}  
void func2(void)  
{  
    int x;  
    x=20;  
    ...  
}
```

在 `func1` 中定义一个 `x` 变量，在 `func2` 中又定义一个 `x` 变量。`func1` 中的 `x` 和 `func2` 中的 `x` 同名但毫无关系，因为每个 `x` 在各自的程序段中被定义，只能被所在的程序段所认可。

例 6.8 局部变量的使用。

```
#include <stdio.h>  
main()  
{  
    int a,b,c;  
    a=1;b=2;c=3; /* a=1,b=2,c=3 */  
    a=a+1;b=b+1;c=c+1; /* a=2,b=3,c=6 */  
    {  
        int b,c;  
        b=4; /* b=4 */  
        c=b*3; /* c=12 */  
        a=a+c; /* a=14 */  
        printf("first:a=%d,b=%d,c=%d\n",a,b,c);  
        a=a+c; /* a=26 */  
        printf("second:a=%d,b=%d,c=%d\n",a,b,c);  
    }  
    printf("third:a=%d,b=%d,c=%d\n",a,b,c);  
}
```

`main` 函数的函数体内还有一个用花括号括住的程序块（称为复合语句）。外层花括号内定义了变量 `a`, `b`, `c`，它们的作用范围从最外层左花括号开始，到最外层右花括号结束；内层花括号内又定义了变量 `b` 和 `c`，它们的作用范围从内层左花括号开始，到内层右花括号结束。尽管变量 `b` 和 `c` 在内外层花括号中的名称相同，但它们在其中的取值不同。变量 `a` 则在内外层花括号内都有效。因此程序运行结果为：

```
first: a=14,b=4,c=12  
second: a=26,b=4,c=12  
third: a=26,b=3,c=6
```

6.3.2 全局变量

在函数外部定义的变量称为外部变量或全局变量。在程序的整个执行期间，全局变量的值都被保留，并且可以被多个函数引用。也就是说，全局变量的作用域是整个程序，所以全局变量也可作为函数间传递数据的一种手段。

例 6.9 全局变量传递数据。

```
#include <stdio.h>
```

```
int c;          /*定义外部变量 c, 被 main 和 plus 共享 */
main()
{
    int a,b;
    void plus(int,int);
    printf("input a b =");
    scanf("%d%d",&a,&b);
    printf("a=%d,b=%d\n",a,b);
    plus(a,b);
    printf("a+b=%d\n",c);
}
void plus(int x,int y)
{
    c=x+y;
}
```

运行结果为:

```
input a b = 5 6
a=5,b=6
a+b=11
```

外部变量的作用域是从定义处开始, 直到整个文件结束; 在外部变量定义位置之前的函数不能共享它们。但若用 **extern** 关键字对外部变量进行声明, 在定义外部变量位置之前的函数也能共享它们。

例 6.10 外部变量的声明。

```
#include <stdio.h>
main()
{
    extern int a,b;  /*声明 a, b 是外部变量*/
    printf("%d\n",max(a,b));
}
max(int x,int y)
{
    int z;
    z=x>y?x:y;
    return z;
}
int a=13,b=-8;  /*定义外部变量 a, b*/
```

程序中, 全局变量 **a** 和 **b** 是在程序末尾定义的, 本来 **main** 函数不能引用它们, 由于在 **main** 函数中对它们进行了 **extern** 声明, 使这两个变量可以被 **main** 函数引用, 所以程序运行时输出 13。

需要说明的是, 利用全局变量实现函数间的数据传递, 加大了函数之间的数据影响, 但降低了整个程序的可靠性和通用性, 因此在程序设计中应限制使用这种方式来传递数据。



6.4 变量的存储类别

变量的存储类别指的是变量在计算机中的存放位置，它决定了变量的生存期。

当一个用户程序运行时，操作系统必须为它安排足够的存储空间。计算机中的存储空间可分为两类：一类是内存，一类是运算器中的通用寄存器。

在内存中，操作系统安排一段内存空间（称为用户区）供用户程序使用。用户区分为 4 部分：第一部分称为程序代码区，用于存放程序代码；第二部分称为全局变量区，也称静态存储区，用于存放程序中定义的全局变量和静态局部变量，这些数据所占用的存储空间在程序编译期间分配好，并在整个程序运行期间保持有效；第三部分称为动态存储区，这是一块自由存储区，程序可通过 Turbo C 的动态分配函数使用它；第四部分称为运行栈，用于存放程序运行过程中需要动态分配和释放的变量或数组，如子程序的形式参数、子程序调用时的现场保护信息和返回地址以及自动型局部变量等，这些数据在程序运行期间根据需要动态地产生和消失。结构参见图 6-2。



图 6-2 内存用户区示意图

运算器中的通用寄存器也可用于存放数据。由于通用寄存器可以直接与运算器中的算术逻辑单元进行信息交换，因此存取速度比内存快，但寄存器数量非常有限。

C 语言中，变量的存储类别分为 4 种，可以以不同的方式占用内存。

1. 自动类（auto）

这种变量存储在内存的运行栈中，它在程序执行时被分配存储空间，具有暂时存储性质。一旦该变量所在的程序执行结束变量就不复存在，其所占内存被自动释放。也就是说，自动变量的生存期只限于它所在的程序段执行期间。由于自动变量并不长期占用内存，这就使内存空间可以被若干变量多次覆盖使用。因此，C 语言程序中大量使用自动变量，其目的就是为了节省内存空间。自动变量的作用域为定义它的函数内。

2. 外部类（extern）

C 语言允许将若干个程序文件分别编译，然后再将它们连接起来，以提高编译速度并有助于大型工程的维护。例如，两个 C 程序文件 file1.c 和 file2.c，它们分别含有若干个函数，file1.c 和 file2.c 可以分别编译。file1.c 中定义了全局变量 x、y 和 ch，file2.c 要引用这 3 个全局变量，需要用 extern 声明这 3 个变量而不要再定义它们。变量的定义和声明是不同的：定义的含义是为变量安排存储空间；声明的含义只是通知编译系统，后面的变量类型和名称已在其他地方定义过，不必再为它们安排存储空间。如果在 file2.c 中不用 extern

进行声明，编译系统在对 file2.c 编译时就会认为这 3 个变量未定义；如果在 file2.c 中又定义了这 3 个变量，在连接时就发生连接错误。

```
/* file1.c */
int x,y;
char ch;
main()
{
    ...
}
func1()
{
    x=123;
}

/* file2.c */
extern int x,y;
extern char ch;
void func22(void)
{
    x=y/10;
}
void func23(void)
{
    y=10;
}
```

定义外部变量要在函数外进行，采用与定义自动类变量同样的形式，只要定义语句是在函数外，系统自动将定义的变量默认为外部变量；声明外部变量可以在函数外，也可以在函数内，并需要显式的 `extern`。外部变量的生存期为整个程序执行期；外部变量的作用域为从出现定义或声明的位置开始直到程序正文结束。

例 6.11 有两个文件 file1.c 和 file2.c，内容如下：

```
file1.c
#include<stdio.h>
int a=1,b=2,c=3;
int f(void);
int main(void)
{
    printf("%3d\n",f());
    printf("%3d%3d%3d\n",a,b,c);
    return 0;
}
file2.c
int f(void)
{
    extern int a;
    int b,c;
    a=b=c=4;
    return(a+b+c);
}
```

两个文件可以分别编译，第一个文件中定义了 3 个外部变量 a,b,c，第 3 个文件中将变量 a 声明为外部变量，即用 `extern` 通知编译系统 a 变量在其他地方（可能是本文件中也可能是其他文件中）被定义，第二个文件的 f 函数内又定义了同名的内部变量 b 和 c。

当 f 函数被调用时，内部变量优先，也就是说在 f 函数内，外部变量不起作用；函数退出时，内部变量消失。所以在 f 函数内，变量 a、b、c 被重新赋值并参与运算，返回主函数时，主函数中只有 a 变量的值改变，b、c 变量的值并不变化。将两个文件合并成一个项目并连接运行，程序的输出结果是：



```
12
4 2 3
```

当编写大型程序时，对文件分别编译是很有用的。

3. 静态类 (static)

静态变量存储在内存的全局变量区中，在程序编译时由系统分配存储空间，在整个程序运行期间始终占用该存储位置。因此，静态变量的生存期与整个程序的运行期相同，既当结束了相应函数调用后，static 变量的储存空间依然保留。使用静态变量，会长期占用内存，因而内存的开销比较大，但带来了程序易读的优点。

静态变量的定义必须用关键词 **static** 显式地加以说明。既可以在函数内部定义静态局部变量，又可以在函数外部定义静态全局变量。静态局部变量和静态全局变量的作用是不同的，静态局部变量使用较多，此处只讨论静态局部变量。

静态变量出现在函数内部就是静态局部变量。程序编译时，编译系统将这些局部变量存放在静态存储区即全局变量区中，使它们占用永久存储器。这样，不管这些变量所在的程序段或程序块是否执行结束，它们的值都将被保留下来。静态局部变量的生存期是整个程序的执行期，因此，静态局部变量有和外部变量一样的生存期。静态局部变量的作用域是定义它的函数内部，只有在函数内部可以对静态局部变量进行操作，所以相对于外部变量来说，它更安全，用程序设计的术语来说，静态变量起到了信息隐蔽的作用。

例 6.12 静态局部变量的使用。

```
#include <stdio.h>
long factorial(int n)
{
    static long int f=1; /*函数内的静态局部变量*/
    f=f*n;
    return f;
}
main()
{
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%ld\n",i,factorial(i));
}
```

程序的执行结果：

```
1!=1
2!=2
3!=6
4!=24
5!=120
```

main 函数要 5 次调用函数 factorial，编译时将静态局部变量 f 的值初始化为 1，每一次程序被调用时，初始化语句不再执行。因为 f 是 static 变量，其生存期是整个程序执行期，故每次函数 factorial 调用结束后，f 的值仍然保留。这样，第一次调用 factorial 后 f 的值是

1 的阶乘;第二次调用 factorial 后 f 的值是 $1! \times 2$, 即 2 的阶乘;依此类推,第 i 次调用 factorial 后 f 的值是 $(i-1)! \times i$, 即 i 的阶乘。

若将本例题中的语句 “static long int f=1;” 改写为 “static long int f; f=1;”, 编译和连接过程都不会出现任何错误, 但得到错误的运行结果, 试分析其原因?

4. 寄存器类 (register)

寄存器 (register) 变量只能出现在函数内部, 在它前面必须有关键词 register。当一变量被说明为 register 变量时, 其值存放在 CPU 运算器的通用寄存器中, 这样对寄存器变量的存取速度就很快。C 语言使用寄存器型变量时直接在寄存器中进行存取, 而不必到内存中去, 因而提高了程序的执行效率。程序中使用寄存器变量时, 由于运算器中通用寄存器的个数有限, 用户可以使用的通用寄存器更为有限, 一般微型机只允许 2~3 个, 小型机也只允许 3~5 个, 所以程序中定义的寄存器变量就不能超过限制; 并且寄存器变量只限于 char 型和 int 型, 而 long int 型、float 型和 double 型变量就不能定义为寄存器型。C 语言作了容错规定, 即若用户定义的寄存器型变量超过了通用寄存器数目的限制, 则自动将多余的变量作为 auto 型变量处理。

一般情况下, 变量的值是存放在内存中的, 自动变量和静态变量都是如此。当程序中用到哪一个变量时, 由控制器发出指令将该变量的值送到运算器中, 用运算器进行运算。如果需要存放, 再由控制器发出指令将运算器中的数据送到内存存放。如果有的变量使用很频繁, 就可以用寄存器存放该变量, 即将变量的存储类别设为寄存器类, 以提高程序执行速度。

例 6.13 寄存器变量实例。

```
#include <stdio.h>
long factorial(int n)
{
    register int i;
    long r=1;
    for(i=1; i<=n; i++)
        r*=i;
    return r;
}
main()
{
    int k;
    for(k=1;k<=5;k++)
        printf("%d!=%ld\n",k,factorial(k));
}
```

程序执行结果为:

```
1!=1
2!=2
3!=6
4!=24
```



5!=120

register 变量的作用域局限在相应的函数内部，生存期为相应函数被调用时。需要说明的是，Turbo C 2.0 对 register 变量当作自动变量处理，并不把它们真正放在寄存器中。

各类变量的特性总结见表 6-1。

表 6-1 存储类别小结

存储类别	出现范围	判别方法	作用域	生存期
自动类	函数内部	(1) 显式出现 auto (2) 在函数内部（包括 main）中省略存储类别定义的变量	定义该变量的函数内	定义该变量的函数被调用时
外部类	任何可出现说明部分的位置	(1) 显式出现 extern (2) 在函数外部省略存储类别定义的变量	出现定义或声明的位置开始直至程序正文结束	整个程序执行期
静态类(仅限函数内部情况)	函数内部	显式出现 static	定义该变量的函数内	整个程序执行期
寄存器类	函数内部	显式出现 register	定义该变量的函数内	定义该变量的函数被调用时

注意不同存储类别变量的初始化：用户没有进行初始化的外部变量和静态变量，系统自动将其值设为 0；而自动类和寄存器类变量，如果用户不对其进行初始化，其值一般是任意的。

例 6.14 变量存储类别举例。

```
#include <stdio.h>
int x=1,y=2; /*定义外部变量并初始化*/
int funcf(int x,int y)
{
    int z=2;
    z=x+y;
    return z;
}
main()
{
    int x=3,w;
    w=funcf(x,y);
    printf("%d\n",w);
}
```

/*自动变量 x,y,z 的作用范围*/

/*外部变量 x,y 的作用范围*/

/*自动变量 x,w 的作用范围*/

本程序中，既有外部变量 x，又有在函数 funcf 中作为形式参数的自动变量 x，y，还有在 main 中的自动变量 x。这些同名变量代表的是不同的对象，问题在于 funcf 以及 main 中出现的 x 是代表自动变量，还是外部变量？为此，C 语言规定：在同一个源程序中，若全局变量与自动变量同名，则在自动变量的作用域内，全局变量不起作用。因此，函数 funcf

中出现的 x , y 代表的是形参, 属局部自动变量; `main` 中出现的 x 在 `main` 函数中被定义为局部自动变量, 其值为 3 (不取外部变量的值 1), 而 `main` 中出现的 y 在 `main` 函数内没有再定义, 代表的是外部变量, 取值为 2。按照这种理解, 程序最终输出结果是 5。

6.5 函数的嵌套调用和递归调用

6.5.1 函数的嵌套调用

函数定义是平行的, 但函数调用允许嵌套, 即一个函数不能在另一个函数体中进行定义, 但一个函数被调用的过程中可以调用另一个函数。例如:

```
float f1(int a,float b)    float f2(float x,float y)
{
    float c;
    c=f2(b-1,b+1);        函数体;
    ...
}                          ...
}
```

`f1` 和 `f2` 是分别独立定义的函数, 互不从属。但在调用 `f1` 的过程中又要调用函数 `f2`, 其调用过程如图 6-3 所示。

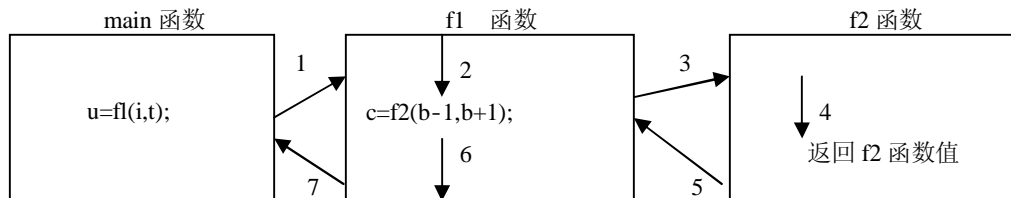


图 6-3 函数的嵌套调用及其返回过程

图中 1~7 为调用过程的步骤。即 1 将流程转到 `f1` 函数, 实参 i , t 分别传值给形参 a , b ; 2 执行 `f1` 函数体中语句直到遇到调用 `f2` 的语句; 3 调用 `f2` 函数, 将 `f1` 函数中的数据 $b-1$, $b+1$ 作为实参传送给 `f2` 函数的形参; 4 执行 `f2` 函数; 5 将 `f2` 函数返回值带回 `f1` 函数中的调用处 (即带回到函数调用表达式处); 6 继续执行 `f1` 函数中剩下的语句; 7 将 `f1` 函数的返回值带回函数 `main` 中, 并赋给变量 u 。每一次调用时, 形参的值自动压入运行栈, 每一层返回都是返回到本层函数被调用的位置, 返回时释放形参占用的内存。

6.5.2 函数的递归调用

一个函数直接或间接地调用函数本身, 称为递归调用, 前者称为直接递归, 后者称为间接递归。递归调用的函数称为递归函数。由于递归非常符合人们的思维习惯, 而且许多数学函数及许多算法或数据结构都是递归定义的, 因此, 递归调用颇具实用价值。

例如: 我们用 `factorial(n)` 表示计算 $n!$ 的函数, 则 `factorial(n-1)` 表示 $(n-1)!$, 则有关系: `factorial(n)=n* factorial(n-1)`, 这样的关系就是递归关系, 可以用如下函数实现。

例 6.15 递归计算 $n!$ 。

```
#include <stdio.h>
long factorial(int n);
main()
{
    printf("%d!=%d\n",3,factorial(3));
    printf("%d!=%d\n",5,factorial(5));
}
long factorial(int n)
{
    if(n<=1)
        return(1);
    else
        return(n* factorial(n-1));    /*调用自己*/
}
```

运行结果为：

```
3!=6
5!=120
```

对例 6.15 递归调用过程分析如下：

当形参 n 的值大于 1 时，函数的返回值为 $n * \text{factorial}(n-1)$ ，其中 $\text{factorial}(n-1)$ 又是一次函数调用，而调用的正是 factorial 函数，这就是在一个函数中调用自身函数的情况，即函数的递归调用。

函数的返回值是 $n * \text{factorial}(n-1)$ ，而 $\text{factorial}(n-1)$ 的值当前还不知道，要调用完才能知道。例如 $n=5$ 时，返回值是 $5 * \text{factorial}(4)$ ；而 $\text{factorial}(4)$ 调用的返回值是 $4 * \text{factorial}(3)$ ；仍然是个未知数，还要先求出 $\text{factorial}(3)$ ，而 $\text{factorial}(3)$ 也不知道，它的返回值为 $3 * \text{factorial}(2)$ ；而 $\text{factorial}(2)$ 的值为 $2 * \text{factorial}(1)$ 。直到 $\text{factorial}(1)$ 的返回值为 1，是一个已知数。这是依次调用的过程。

然后根据 $\text{factorial}(1)=1$ 求出 $\text{factorial}(2)=2 * \text{factorial}(1)=2 * 1=2$ ，将 $\text{factorial}(2)$ 的值乘以 3 求出 $\text{factorial}(3)$ ，将 $\text{factorial}(3)$ 乘以 4 得到 $\text{factorial}(4)$ ，再将 $\text{factorial}(4)$ 乘以 5，得到 $\text{factorial}(5)$ 。这是依次回代的过程。

可以看出，递归函数在执行时，有一系列的调用和回代的过程。当 $n=5$ 时，其调用回代过程如图 6-4 所示。

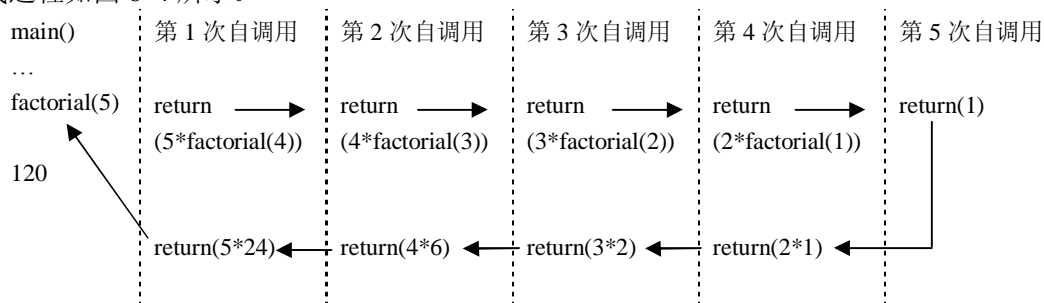


图 6-4 递归函数的调用和回代的过程分析

根据变量的作用域和生存期，递归程序在执行过程中每次对自身的调用，其形参尽管变量名相同，但每一次从实参传递来的值是不同的，而且每次调用完成后，形参的值自动压入运行栈，因此不会互相混淆。在“调用”阶段，形参的值依次压入运行栈，在“回代”阶段，形参的值再依次弹出，同时参加回代公式的计算，最终求出所需要的结果。

从图 6-4 可以看出，递归过程不应无限地进行下去，当调用若干次以后，就应当到达递归调用的终点得到一个确定值（例如本例中的 `factorial(1)=1`），然后进行回代。回代的过程是从已知值推出下一个值。

在设计递归函数时，应当考虑到递归的终止条件，在本例中递归终止的条件为：

```
if(n<=1)
    return(1);
```

任何有意义的递归总是由两部分组成的：递归方式与递归终止条件。递归方式根据问题本身来确定，递归终止条件一般用 `if` 语句或类似的语句来表示。要避免程序无休止地递归下去。

递归是一种非常有用的程序设计技术。当一个问题蕴含递归关系且结构比较复杂时，使用递归算法往往要自然、简洁、容易理解。但是递归函数的效率低，例如本例中求阶乘函数的每一次递归调用都会带来重复计算，并且每一次调用都需进行保留现场、传递参数及恢复现场等操作，大量反复的递归调用会降低效率。

例 6.16 求菲波那契（Fibonacci）数列的第 n 项。菲波那契数列的定义为：

$$\text{fib}(n)=\begin{cases} 0 & (n=0) \\ 1 & (n=1) \\ \text{fib}(n-1)+\text{fib}(n-2) & (n\geq 2) \end{cases}$$

用递归方法就可以很容易地写出程序如下：

```
#include <stdio.h>
long fib(int);
main()
{
    int n=5;
    printf("fib(%d)= %ld",n,fib(n));
}
long fib(int n)
{
    int f;
    if(n==0||n==1)
        f=n;
    else
        f=fib(n-1)+fib(n-2);
    return f;
}
```

运行结果为：`fib(5)=5`。该程序执行时的调用和回代过程请读者仿照例 6.15 自行分析。



6.6 内部函数与外部函数

一个 C 语言程序可以由多个函数组成，这些函数既可以在一个文件中，也可以分散在多个不同的文件中。根据这些函数的使用范围，可以把它们分为内部函数和外部函数。

6.6.1 内部函数

内部函数又被称为静态函数，它只能被定义它的文件中的其他函数调用，而不能被其他文件中的函数调用，亦即内部函数的作用范围仅仅局限于本文件。为了定义内部函数，需要使用关键字 `static`。如：

```
static long factorial(int x);
```

此时，函数 `factorial` 的作用范围仅局限于定义它的文件，而在其他源文件中不能调用此函数。如果在不同的源文件中存在同名的内部函数，它们互不干扰。

6.6.2 外部函数

因为函数与函数之间都是并列的，即函数不能嵌套定义，所以函数在本质上都具有外部性质。内部函数（静态函数）只能被定义它的源文件中的函数调用，而不能被其他源文件中的函数调用。除此之外，其余的函数既可被定义它的源文件中的函数调用，也可以被其他源文件中的函数所调用，即其作用范围不只局限于函数所在的源文件，而是整个程序的所有文件。有时为了明确这种性质，可以在函数定义和调用时使用关键字 `extern`，`extern` 既可用于外部函数的定义，也可用于外部函数的声明。

一个程序如果用到多个函数，允许把它们定义在不同的文件中，也允许一个文件中含有不同程序中的函数，即在一个文件中可以包含本文件中的程序用不到的函数。

在定义函数时，一个函数只能定义在别的函数的外部，它们都是平行的，互相独立的，一般省略关键字 `extern`。如果在一个文件中的函数要调用其他文件中定义的函数，一般先用 `extern` 声明被调用的函数，表示该文件在其他地方定义；在有些系统中，也可以不做声明。而 `static` 只用于内部函数的定义，内部函数不需要声明。

可以用 `#include` 命令将需要引用的文件包含到主文件中。在编译时，系统自动将被包含的文件放到 `#include` 命令所在的位置，作为一个整体编译。这时，这些函数被认为是在同一个文件中，不再是作为外部函数被其他文件调用；主调函数中原有的 `extern` 声明也可以不要。

6.7 多文件编程举例

当程序较大时，经常分成多个模块编程，并将不同的模块文件分别编译，然后组成一个项目文件（Project），再将项目文件连接并运行。

例 6.17 输入多个数，排序后再输出。由主函数调用子函数来完成输入、排序和输出的工作。

(1) 主函数和子函数可以写在同一个源程序文件中，如下例所示：

```
#include<stdio.h>
#define N 3
int num[N];
void input()/*输入数据的子函数*/
{
    int j;
    printf("please input %d numbers",N);
    for(j=0;j<N;j++)
    {
        scanf("%d",&num[j]);
    }
}
void sort() /*进行排序的子函数*/
{
    int t,k;
    int j;
    for(j=0;j<N-1;j++)
        for(k=j+1;k<N;k++)
            if(num[j]>num[k])
            {
                t=num[j];
                num[j]=num[k];
                num[k]=t;
            }
}
void output()/*输出结果的子函数*/
{
    int j;
    for(j=0;j<N;j++)
    {
        printf("%d \n",num[j]);
    }
}
main()
{
    input();
    sort();
    output();
}
```

(2) 也可以将主函数和被调函数写成两个文件单独编译，再组成项目文件连接并运行。

```
/*输入和排序子函数放在 sub.c*/
#define N 3
int j; /* j 是本程序文件中的外部变量，两个函数都可以引用*/
extern int num[N];
/* 声明为外部变量，表示该数组在其他地方定义 */
/* 如果没有这一句，编译此文件时会显示 num 没有定义*/
/* 如果写成 int num[N] 连接时会显示 num 重复定义 */
```



```
void input()
{
    printf("please input %d numbers",N);
    for(j=0;j<N;j++)
    {
        scanf("%d",&num[j]);
    }
}
void sort()
{
    int t,k;
    for(j=0;j<N-1;j++)
        for(k=j+1;k<N;k++)
            if(num[j]>num[k])
            {
                t=num[j];
                num[j]=num[k];
                num[k]=t;
            }
}
/* 主函数main()和输出子函数 output()写在 main.c 文件中 */
#include<stdio.h>
#define N 3
int num[N];
/* 下面将 output() 定义为静态函数或称为内部函数，只允许同一个源文件中的函数调用它。
sub.c 中的两个函数不可以写成 static ...*/
static void output() /* 也可写成 void output()将 output()函数定义为外部函数 */
{
    int j;          /*j 再定义为函数的内部变量，也可以用 extern 声明为外部变量*/
    printf("print numbers in sort ascending");
    for(j=0;j<N;j++)
    {
        printf("%d \n",num[j]);
    }
}
main()
{
    input();
    sort();
    output();
}
```

将上述两文件单独编译好，再建立一个项目文件，项目文件的内容如下：

```
main.c
sub.c
```

以 proj1.prj 为名保存文件，见图 6-5。

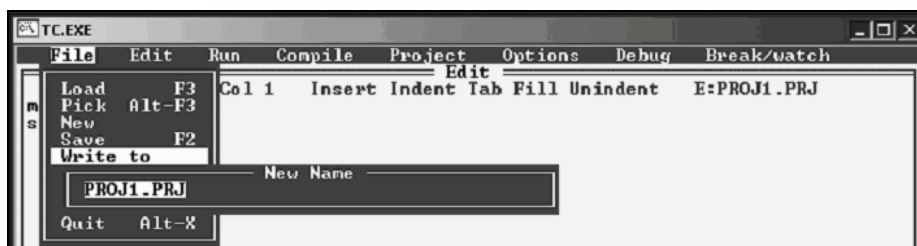


图 6-5 为项目文件命名

在 Project 菜单 Project name 子菜单下，输入项目名称 proj1.prj，然后在 Compile 菜单下连接、运行，见图 6-6。



图 6-6 输入项目名称

项目使用完后，要从 Project 菜单下运行 Clear project 清除项目，即移动光标到 Clear project 选项按回车键。如果不清除项目，则以后按 Alt+R 总是运行该项目，见图 6-7。

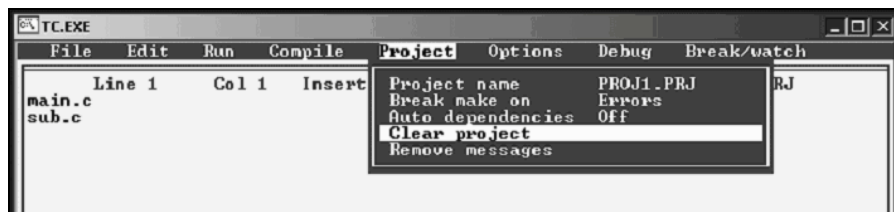
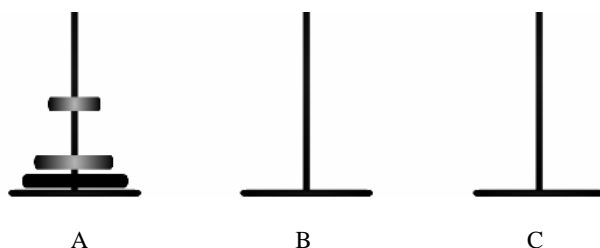


图 6-7 清除项目

例 6.18 “汉诺塔”问题是递归算法的典型问题。有 A、B、C 3 根柱子，A 上堆放了 n 个盘子，每个盘子都比它下面的盘子小一些。现在要把盘子全部搬到 C 上去，条件是每次只能搬动一个盘子，而且任何时候都不能放在比它小的盘子上面（显然，必须用到 B 作为中转）。怎么搬动这些盘子呢？



解决办法：

分析发现，想把 n 个盘子搬到 C，必须先把上面的 $n-1$ 个盘子搬到 B，然后把第 n 个



盘子搬到 C，最后再把 $n-1$ 个盘子搬过来。整个过程是这样的：

- (1) 把上面的 $n-1$ 个盘子从 A 搬到 B，以 C 作为中转；
- (2) 把第 n 个盘子从 A 搬到 C；
- (3) 把 $n-1$ 个盘子从 B 搬到 C，以 A 作为中转。

也就是说，要解决 n 个盘子的问题，先要解决 $n-1$ 个盘子的问题。而这个问题与前一个是类似的，可以用相同的办法解决。最终，我们会达到只有一个盘子的情况，这时直接把盘子从 A 搬到 C 即可。

例如，将 3 只盘子从 A 移到 C 可以分为如下三步：

- (1) 将 A 柱上的 1~2 号盘子借助于 C 移至 B 柱上；
- (2) 将 A 柱上的 3 号盘子移至 C 柱上；
- (3) 将 B 柱上的 1~2 号盘子借助于 A 柱移至 C 柱上。

步骤 (1) 又可分解成如下三步：

- ① 将 A 柱上的 1 号盘子从 A 柱移至 C 柱上；
- ② 将 A 柱上的 2 号盘子从 A 柱移至 B 柱上；
- ③ 将 C 柱上的 1 号盘子从 C 柱移至 B 柱上。

步骤 (3) 也可分解为如下三步：

- ① 将 B 柱上的 1 号盘子从 B 柱移至 A 柱上；
- ② 将 B 柱上的 2 号盘子从 B 柱移至 C 柱上；
- ③ 将 A 柱上的 1 号盘子从 A 柱移至 C 柱上。

综合上述移动，将三只盘子由 A 移到 C 需要如下的移动步骤：

- (1 号盘子 $A \rightarrow C$, 2 号盘子 $A \rightarrow B$, 1 号盘子 $C \rightarrow B$), 3 号盘子 $A \rightarrow C$,
(1 号盘子 $B \rightarrow A$, 2 号盘子 $B \rightarrow C$, 1 号盘子 $A \rightarrow C$)

可以把上面三步骤归纳为两类操作：

- (1) 将 $1 \sim (n-1)$ 号盘子从一个柱子移动到另一个柱子上；
- (2) 将 n 号盘子从一个柱子移动到另一个柱子上。

基于以上分析，可以将程序分成几个文件。标题文件 `hanoi.h` 中对子函数作声明，并定义外部变量 `cnt` 用来统计移动盘子的次数；`hanoi.c` 文件中是主函数，其中调用函数接收键盘输入的盘子个数及调用函数进行移动盘子；`get.c` 文件中是接收键盘输入盘子个数 n 的函数；`move.c` 文件中的函数是一个递归函数，可以实现将 n 个盘子从一个柱子借助于中间柱子移动到另一个柱子，如果 n 不为 1，以 $(n-1)$ 作实参调用自身，即将 $(n-1)$ 个盘子移动，依次调用自身，直到 n 等于 1，结束递归调用。各文件的内容如下：

```
/* hanoi.h */
#include<stdio.h>
#include<stdlib.h>
extern int cnt; /* the number of movement */
int get_n_from_user(void);
void move(int n,char a,char b,char c);
/* hanoi.c */
#include"hanoi.h"
int cnt=0;
```

```
int main(void)
{
    int n;
    n=get_n_from_user(); /* get number of plates */
    /*Move n disks from tower A to tower C,using tower B as an
    intermediate tower*/
    move(n,'A','B','C');
    return 0;
}
/* file get.c, the function name is get_n_from_user */
#include "hanoi.h"
int get_n_from_user(void)
{
    int n;
    printf("%s","TOWERS OF HANOI:\n"
        "the problem starts with n disks on Tower A.\n"Input n:");
    if(scanf("%d",&n)!=1 || n<1)
    {
        printf("\nERROR");
        exit(1);
    }
    return n;
}
/* move.c */
#include "hanoi.h"
void move(int n,char a,char b,char c)
{
    if(n==1)
    {
        ++cnt;
        printf("%5d: %s%d%s%c%s%c.\n",cnt,"Move disk ",
            1," from tower ",a," to tower ",c);
    }
    else
    {
        move(n-1,a,c,b);
        ++cnt;
        printf("%5d: %s%d%s%c%s%c.\n",cnt,"Move disk ",
            n ," from tower ",a," to tower ",c);
        move(n-1,b,a,c);
    }
}
```

将 3 个源文件分别编译完成，并建立项目文件 `hanoi.prj`，内容如下：

```
hanoi.c
get.c
move.c
```

将 `project` 菜单下的 `project name` 设为 `hanoi.prj`，然后连接、运行程序。若在运行程序过程中输入盘子个数为 3，则得到如下结果：



TOWERS OF HANOI:

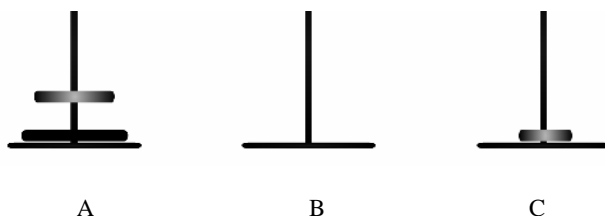
the problem starts with n disks on Tower A.

Input n: 3

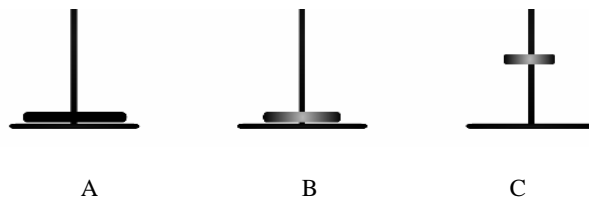
- 1: Move disk 1 from tower A to tower C.
- 2: Move disk 2 from tower A to tower B.
- 3: Move disk 1 from tower C to tower B.
- 4: Move disk 3 from tower A to tower C.
- 5: Move disk 1 from tower B to tower A.
- 6: Move disk 2 from tower B to tower C.
- 7: Move disk 1 from tower A to tower C.

第一次到第四次移动盘子的示意图如下。

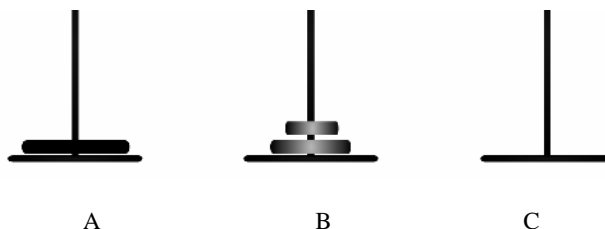
(1) 将 1 号盘从 A 移动到 C



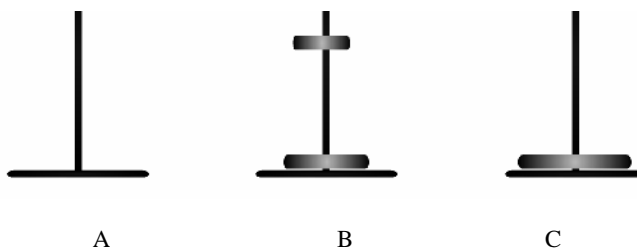
(2) 将 2 号盘从 A 移动到 B



(3) 将 1 号盘从 C 移动到 B



(4) 将 3 号盘从 A 移动到 C



要 点 回 顾

1. 函数是利用 C 语言进行结构化程序设计的最基本的概念，C 程序是由函数组成的。可以把一个复杂的程序分成多个模块进行设计，而每个模块是一个函数。`main()`是 C 程序中最重要函数，程序运行从 `main()`函数开始，也在 `main()`函数结束。在 `main()`函数的执行过程中，可以调用其他函数。

2. 函数定义的一般形式为：

```
[类型标识符] <函数名> ([<形式参数说明>])  
{  
    <说明部分>  
    <语句部分>  
}
```

“类型标识符”指明函数返回值的类型。如果函数定义时不指明类型，系统隐含指定为 `int` 型。“形式参数”又称形参、虚参或哑元，有两个作用：其一表示将从主调函数中接收哪些类型的信息，其二在函数体中形式参数可以被引用。

3. 函数返回值由 `return` 语句实现，`return` 语句的格式为：

```
return 表达式；
```

函数先将表达式的值转换为所定义的类型，然后返回到主调函数中的调用表达式。

4. 函数调用是通过函数调用表达式进行，即“<函数名>(<实际参数表>)”。当函数被调用时，机器才为形参分配存储空间，形参从相应的实参得到值，称为传值调用方式，也称为“虚实结合”，实参与形参在个数、类型上要匹配。当调用结束，流程返回主调函数时，形参所占空间被释放。

5. 函数调用前应该已经定义或声明，函数声明的一般格式为：

```
类型标识符 函数名 (类型标识符 形参, 类型标识符 形参, …) ;
```

与函数定义中的第一行（称函数头）内容一致（也称函数原型或函数模型）。函数定义以函数体结尾，而函数声明不包含函数体。函数定义要求分配内存单元，用来存放编译后的函数指令；而函数声明的作用只是通知编译系统函数的参数个数和类型以及函数返回值的类型。

6. 函数的形参及函数内定义的变量称为内部变量或局部变量，其作用范围在定义它的函数或复合语句内。

7. 在函数外部定义的变量称为外部变量或全局变量，其作用域是从定义或声明处到整个程序结束。

8. 变量的存储类别指的是变量在计算机中的存放位置，变量的存储类有：自动类（`auto`）、外部类（`extern`）、静态类（`static`）、寄存器类（`register`）。它们具有不同的生存期和作用域。

9. 一个函数被调用的过程中可以调用另一个函数，即函数调用允许嵌套。每一次调用时，形参的值自动压入运行栈，每一层返回都是返回到本层函数被调用的位置；返回时释放形参占用的内存。



10. 一个函数直接或间接地调用函数本身，称为递归调用。任何有意义的递归总是由两部分组成的：递归方式与递归终止条件。

11. 一般来说，一个函数，只要不是主函数，就可以被其他函数调用。可以认为函数默认是外部的。为了减少函数的互相影响，C 语言规定，有的函数只能被定义它的文件中的其他函数调用，而不能被其他文件中的函数调用，这样的函数称为内部函数或静态函数。定义内部函数时，需要使用关键字 **static**，不同文件中的静态函数可以同名而互不影响。

12. 当程序较大时，经常分成多个模块编程：将不同的模块文件分别编译，然后组成一个项目文件（Project），再将项目文件连接并运行。

项目文件的内容为各个模块文件的名字，项目文件的扩展名为 .prj；在 Project 菜单 Project name 子菜单下，输入项目名称，然后再连接、运行。项目使用完后，要从 Project 菜单下运行 Clear project 清除项目。

习 题

- 以下对 C 语言函数的有关描述中，正确的是（ ）。
 - 在 C 中调用函数时，只能把实参的值传送给形参，形参的值不能传送给实参
 - C 函数既可以嵌套定义，又可以递归调用
 - 函数必须有返回值，否则不能使用函数
 - 程序中有调用关系的所有函数必须放在同一个源程序文件中
- 以下叙述中不正确的是（ ）。
 - 在 C 中，函数中的自动变量可以赋初值，每调用一次赋一次初值
 - 在 C 中调用函数时，实参和对应形参在类型上只需赋值兼容
 - 在 C 中，外部变量的隐含类型是自动存储类别
 - 在 C 中，函数形参可以说明为 register 变量
- 在调用函数时，如果实参是简单变量，它与对应形参之间的数据传递方式是（ ）。
 - 地址传递
 - 单向值传递
 - 由实参传给形参，再由形参传回实参
 - 传递方式由用户指定
- 以下函数值的类型是（ ）。

```
fun( float x )  
{  
    float y;  
    y= 3*x-4;  
    return y;  
}
```

 - int
 - 不确定
 - void
 - float

5. 设有以下函数：

```
f( int a)  
{  
    int b=0;
```

```
static int c = 3;
b++;
c++;
return(a+b+c);
}
```

如果在下面的程序中调用该函数，则输出结果是什么？

```
main()
{
    int a = 2,i;
    for(i=0;i<3;i++)
        printf("%d\n",f(a));
}
```

6. 以下函数的功能是求 x 的 y 次方，请填空。

```
double fun( double x,int y)
{
    int i;
    double z;
    for(i=1,z=x; i<y;i++)
        z=z*(_____);
    return z;
}
```

7. 以下叙述中正确的是 ()。
- A) 构成 C 程序的基本单位是函数
 - B) 可以在一个函数中定义另一个函数
 - C) main() 函数必须放在其他函数之前
 - D) 所有被调用的函数一定要在调用之前进行定义
8. 有一函数：

$$y = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

以下程序段中不能根据 x 值正确计算出 y 值的是 ()。

- A) if(x>0)y=1;else if(x==0)y=0;else y= -1;
 - B) y=0;if(x>0)y=1;else if(x<0)y= -1;
 - C) y=0;if(x>=0)if(x>0)y=1;else y= -1;
 - D) if(x>=0)if(x>0)y=1;else y=0;else y= -1;
9. C 语言中，函数值类型的定义可以缺省，此时函数值的隐含类型是 ()。
- A) void
 - B) int
 - C) float
 - D) double
10. 以下函数的功能是计算 $s=1+1/2!+1/3!+\cdots+1/n!$ ，请填空。

```
double fun(int n)
{
```



```
double s=0.0,fac=1.0; int i;  
for(i=1;i<=n;i++)  
{  
    fac=fac_____;  
    s=s+fac;  
}  
return s;  
}
```

11. 分别用递归函数和静态变量两种方法编程计算下式: $\text{sum}=1+2!+3!+4!\cdots 20!$ 。

第 7 章 数 组

本章的学习目标:

了解数组的基本概念以及数组的存储结构,掌握数组的定义和初始化的方法,掌握引用数组元素的方法(注意数组边界),掌握和数组有关的一些典型算法如排序、检索等,会利用数组编写数据处理的程序。

7.1 数组的定义和初始化

数组用来表示具有相同数据类型且按顺序存放的一组变量,也就是说,数组是相同类型变量的集合。构成数组的变量称为数组元素,各个数组元素具有同一个数组名,不同的元素可按不同的下标区分。有一个下标的数组称为一维数组,有多个下标的数组称为多维数组;多维数组中以二维数组应用较多。

例如,定义一个整型数组 `a[10]`,就定义了 10 个有序的整型变量,每一个变量称为一个数组元素。第一个变量记为 `a[0]`,第二个变量记为 `a[1]`,这里的“0”和“1”称为下标,它指明了该变量在数组中的位置,所以也把数组元素称为下标变量。

同简单变量一样,数组也要先定义后使用。定义数组,就是要:

- (1) 规定数组名,它是数组中所有元素共有的名称,取名规则与标识符相同;
- (2) 规定数组类型,包括数据类型和存储类别。它代表数组中每个元素的类型,基本数据类型是必需的,存储类别则是可选的;
- (3) 规定数组大小,包括维数及每维中可容纳的元素个数。数组中维的概念类似于几何空间中维的概念,一维数组的元素有一个下标,二维数组的元素有两个下标, ..., n 维数组的元素有 n 个下标。这就像一维空间中的点要用一个坐标表示,二维空间的点要用两个坐标表示一样。

7.1.1 一维数组的定义

定义一维数组的一般形式为:

```
[存储类别] 数据类型 数组名 [常量表达式];
```

其中,“存储类别”可以是 `auto`, `static`, `register`, `extern` 之一,“数据类型”为 `int`, `float`, `char`, `double` 之一。数组名后面方括号中的“常量表达式”是仅由常数或符号常数组成的表达式,它规定该数组中可容纳的元素个数。该值称为维界,必须为正整数。该“常量表达式”的值是在编译时确定的。例如: `int data[10]`; 定义了一个一维整型数组 `data`, 它有 10 个元素; `static float array[3]`; 定义了一个静态的一维实型数组 `array`, 它有 3 个元素。

下标是从数组开始到数组元素的偏移量。所有的一维数组都用 `[0]` 作为第 1 个元素的下



标。因此，数组 `data` 的 10 个元素依次为 `data[0]~data[9]`，数组 `array` 的 3 个元素依次为 `array[0]~array[2]`。

7.1.2 多维数组的定义

有两个下标的数组是二维数组。定义二维数组的一般形式是：

```
[存储类别] 数据类型 数组名[常量表达式 1][常量表达式 2]
```

将它与一维数组的定义形式对照一下发现，除了数组名后面多了一对方括号外，其他完全相同。我们把“常量表达式 1”称为第二维或行，把“常量表达式 2”称为第一维或列。例如，要定义一个大小为 10 行 20 列的实型数组 `d`，用如下语句实现：

```
float d[10][20];
```

二维数组用 `[0][0]` 作为第一个元素的下标。因此，二维实型数组 `d` 描述了如下的矩阵：

	第 1 列	第 2 列	...	第 20 列
第 1 行	<code>d[0][0]</code>	<code>d[0][1]</code>	...	<code>d[0][19]</code>
第 2 行	<code>d[1][0]</code>	<code>d[1][1]</code>	...	<code>d[1][19]</code>
...
第 10 行	<code>d[9][0]</code>	<code>d[9][1]</code>	...	<code>d[9][19]</code>

C 语言还允许使用二维以上的数组，一个 `n` 维数组的定义形式为：

```
[存储类别] 数据类型 数组名 [常量表达式 1] [常量表达式 2]...[常量表达式 n]
```

例如：一个三维数组相当于多页二维表格，假设有 3 页表格，每页的表格有 4 行 5 列，则可以用数组 `table[3][4][5]` 来表示。第一个下标相当于页号，第二个下标相当于行号，第三个下标相当于列号。

其中使用较多的是二维数组。

7.1.3 数组的存储结构

进行数组的定义就是让编译程序为每个数组安排一片连续的存储单元来依次存放数组的各个元素。对一维数组来说，各个元素按下标由小到大顺序存放；对二维数组来说，先按行的顺序，再按列的顺序依次存放各个元素。每个元素占用几个字节的存储单元，取决于数组的数据类型，同一个数组的各个元素占用相同字节数的存储单元。例如有下面的定义：

```
float data[6];  
int a[2][3];
```

则浮点型数组 `data`、整型数组 `a` 的存储方式分别如图 7-1 所示。

因此，我们在定义数组的时候，就为 C 编译程序安排存储单元提供了依据。其中，数组名将作为该数组所占的连续存储区的起始地址；数据类型将决定该数组的每一个元素要占用多少字节的存储单元；存储类别则用来确定将该数组安排在运行栈，还是静态存储区中。

data		a	
data[0]	4 字节	a[0][0]	2 字节
data[1]	4 字节	a[0][1]	2 字节
data[2]	4 字节	a[0][2]	2 字节
data[3]	4 字节	a[1][0]	2 字节
data[4]	4 字节	a[1][1]	2 字节
data[5]	4 字节	a[1][2]	2 字节

图 7-1 数组存储方式示意图

7.1.4 数组的初始化

和变量一样，数组也可以进行初始化。在程序不提供初始值的情况下，被 C 编译程序安排在静态存储区的全局型数组和静态局部型数组自动初始化为零，安排在运行栈的局部自动型数组初始值不定。

1. 一维数组的初始化

一维数组的初始化是在定义数组时把所赋初值顺序放在等号右边的花括号中，各常量之间用逗号隔开。例如：

```
static int data[10]={1,2,3,4,5,6,7,8,9,10};  
float table[5]={1.2,3.4,5.6,7.8,9.0};
```

编译时，编译系统会自动地从第一个元素开始，将花括号中的常数顺序存放在各个数组元素的存储单元中。本例中，data[0]初值为 1，…，data[9]初值为 10；table[0]初值为 1.2，table[1]~table[4]的初值依次为 3.4，5.6，7.8 和 9.0。

C 语言规定：

(1) 如果赋初值的花括号中的常数个数少于数组元素个数，则编译程序会自动以零来补足。例如：int x[6]={1,2,3,4};由于数组 x 有 6 个元素，所给初值只有 4 个，这表示只给前 4 个元素赋了初值，即 x[0]=1，x[1]=2，x[2]=3，x[3]=4，其余的元素均为 0 值，即 x[4]=0，x[5]=0。

(2) 不允许初值的个数多于定义的数组元素个数，也不允许用跳过逗号的方式来省略某些元素值。例如：

```
int x[5]={1,2,3,4,5,6}; /*错，初始化值个数多于数组元素个数*/  
int x[5]={1,,2,3,4}; /*错，初始化的值不能省略*/  
int x[5]={}; /*语法格式错*/
```

2. 二维数组的初始化

二维数组的初始化，是将全部初值放在一对花括号中，每一行的初值又分别放在一对内嵌的花括号中。例如 int a[4][3]={ {1,2,3},{4,5,6},{7,8,9},{10,11,12}};其中代表每一行的内



层花括号也可以省略，直接写成：`int a[4][3]={1,2,3,4,5,6,7,8,9,10,11,12};`。

与一维数组初始化类似，二维数组允许每行花括号内的初值个数少于每行中的数组元素个数，这时每行中后面各行的元素也自动赋 0 值。例如：`int a[4][3]={{1,2},{4,5}};`等价于：

```
int a[4][3]={{1,2,0},{4,5,0},{0,0,0},{0,0,0}};或int a[4][3]={1,2,0,4,5,0,0,0,0,0,0,0};
```

全局数组的初始化在编译时候完成，而局部数组的初始化是在进入函数之后完成。

3. 不指定数组长度的初始化

对数组全部元素初始化的数组定义可以省略方括号中的数组大小，这时编译器会统计花括号之间的元素个数并自动确定数组大小。例如：

```
int a[]={2,4,6,8,10};
```

并没有指定 `a` 数组的大小，但全部列出了初始化数据的个数。C 编译系统会自动建立一个足以存放所有初始化数据的数组。上面的定义与 `int a[5]={2,4,6,8,10};`效果相同。

不指定数组长度的定义和初始化方式在用于多维数组时，必须指定除最左边维界之外的所有维界。例如对二维数组，不能省略第二个方括号中的表达式，即只能写成：`int a[][3]={{1,2,3},{4,5}};`它等价于：`int a[2][3]={{1,2,3},{4,5}};`或 `int a[2][3]={1,2,3,4,5,0};`不能写成：`int a[2][]={{1,2,3},{4,5}};`或 `int a[][]={{1,2,3},{4,5}};`

使用不定尺寸数组定义并初始化字符型数组，可以免除统计字符个数的工作。例如：

```
char str[]={'T','h','e',' ','s','t','r','i','n','g','.','\0'};
char str[]="The string.";
```

编译系统会根据初值给出的字符个数自动为该数组分配 12 个字节的存储空间。又如：

```
char language[][8]={"BASIC","FORTRAN","PASCAL","C","COBOL"};
```

编译系统会根据初始化中字符串常量的个数确定该数组的第二维维界是 5。这种情况下可以用 `sizeof` 可得到数组大小。

只能在有初始化的数组定义中可以省略数组大小。在定义数组的地方，无论如何，编译器必须要知道数组的大小。

7.2 数组元素的引用

定义数组后，其元素即可被引用，引用形式为：

```
数组名[下标]
```

例如：

```
a[1]=3;
a[2]=a[1];
printf("%d",a[1]);
```

下标可以是整常数或整型常量表达式，例如`[2+1]`（和为整型常量），在引用时应注意

下标值不要超过数组的边界。例如 `a` 数组的长度为 5，下标值应控制在 0~4 范围内。因为编译时不检查下标是否越界，如果引用了 `a[5]`，编译时不会指出下标越界的错误，而把 `a[4]` 后面的内容作为 `a[5]` 引用。但 `a[4]` 后面的单元可能是其他关键数据，如果对其引用或修改，将会造成无法预料的结果。

7.3 数组的赋值

欲将数据存入指定的数组中，除了初始化方式外，也可用赋值或输入的方式实现。

只能逐个对数组元素赋值，不能直接对数组名赋值。例如，定义了 `int i,a[5]` 后，要将 100、200、300、400、500 存入数组 `a` 中，可用如下程序段实现：

```
for(i=0;i<5;i++)
    a[i]=(i+1)*100;
```

或：

```
a[0]=100,a[1]=200,a[2]=300,a[3]=400,a[4]=500;
```

但不能将该程序段写成：

```
a={100,200,300,400,500};
```

或：

```
a[5]={100,200,300,400,500};
```

又如，建立一个 2×3 的数组 `b`，要求各元素值是其行下标和列下标之和。可用如下程序段实现：

```
int b[2][3],i,j;
for(i=0;i<2;i++)
    for(j=0;j<3;j++)
        b[i][j]=i+j;
```

在进行循环条件判断时，不能写成 `i<=2` 或 `j<=3`，这样会造成下标越界。

同样，对字符型数组的赋值，也只能对每个元素用字符常量赋值。由于字符串中的字符往往没有规律，因而通常不能用循环实现。例如要把字符串“ABC”赋给字符数组 `st`，可用如下程序段完成：

```
char st[4];
st[0]='A';
st[1]='B';
st[2]='C';
st[3]='\0';
```

但不能写成下面形式：

```
st[]="ABC";
```

或：



```
st[4]="ABC";
```

或:

```
st="ABC";
```

对字符型数组赋值时,还可以用本章后面介绍的字符串复制函数 `strcpy()` 将字符串常量复制到字符型数组中,以达到赋值的效果。

7.4 数值型数组的输入和输出

数值型数组的输入输出,一般用循环实现对各元素逐个进行。例如,下面的程序段可实现一维数组的输入:

```
float x[10];
int i;
for(i=0;i<10;i++)
    scanf("%f",&x[i]);    /* 注意, x[i]前面一定要加上地址运算符 "&" */
```

二维数组的输入输出则要用二重循环。若按行的顺序输入,则可以用下面语句:

```
int a[3][4],i,j;
for(i=0;i<3;i++)/*改变行号*/
    for(j=0;j<4;j++)/*改变列号*/
        scanf("%d",&a[i][j]);
```

若按列的顺序输入,则可以用下面语句:

```
int a[3][4],i,j;
for(j=0;j<4;j++)/*改变列号*/
    for(i=0;i<3;i++)/*改变行号*/
        scanf("%d",&a[i][j]);
```

例 7.1 将整型数组 `a` 的 10 个元素分两行输出。

```
#include <stdio.h>
main()
{
    int i,a[10]={1,2,3,4,5,6,7,8,9,10};
    for(i=0;i<10;i++)
    {
        printf("%2d",a[i]);
        if(i%5==4)    /*在下标为 4 和下标为 9 的元素后回车换行*/
            printf("\n");
    }
}
```

程序中增加了一个 `if` 语句,使每打印完 5 个元素就换到下一行。

程序运行结果显示如下:

```
1 2 3 4 5
6 7 8 9 10
```

例 7.2 按矩阵形式打印二维数组 b[3][4]。

```
#include <stdio.h>
main()
{
    int i,j;
    float b[3][4]={1.1,1.2,1.3,1.4,2.1,2.2,2.3,2.4,3.1,3.2,3.3,3.4};
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
            printf("%6.2f",b[i][j]);
        printf("\n");
    }
}
```

程序运行结果为：

```
1.10  1.20  1.30  1.40
2.10  2.20  2.30  2.40
3.10  3.20  3.30  3.40
```

7.5 数组应用程序举例

1. 数据统计

例 7.3 求一维数组 a[10]中的最大值元素及所在的位置。求数组中的最大或最小值是通过不断比较实现的。定义变量 p 来记录最大值所在的位置，设 p 的初始值为 0，用 a[p] 和 a[1] 比较，如果 a[1]>a[p]，则 p 的值变为 1，否则 p 保持原值；再用 a[p] 和 a[2] 比较，如果 a[2]>a[p]，则 p 的值变为 2；依此类推，最后 a[p] 元素就是最大值，p 变量记录的就是最大值所在的位置。

```
#include<stdio.h>
#define N 10
main()
{
    int p,j;
    int a[N]={1,3,5,7,9,2,4,6,8,0};
    p=0;
    for(j=0;j<N;j++)
        if(a[p]<a[j])p=j;
    printf("the max value is %d\n",a[p]);
    printf("the position is %d\n",p);
}
```

2. 排序

排序是将一批杂乱无序的数据按从小到大或从大到小的顺序整齐排列，主要操作是比较和交换。排序算法有很多，本节介绍两种常用的排序算法。



例 7.4 选择法排序。假设将 20 个数存放在 a 数组，要求将它们按从小到大的顺序排列。选择排序方法的基本思路是：

① 第一轮，先将 a[0] 与其后的数进行比较，发现比 a[0] 小的数就记下它的位置，再将该位置的数与其后的数进行比较，经过一轮（共比较 19 次）后，最小的数就选择出来了，将它与 a[0] 交换位置。这样，经过第一轮比较，最小数被放到数组的第一个位置。

② 第二轮，将第二个数 a[1] 与其后的数进行比较，发现比 a[1] 小的数就记下它的位置，再将该位置的数与其后的数进行比较，经过这一轮（共比较 18 次）后，次小的数就选择出来了，把它放在数组的第二个位置上。

③ 依此类推，最后将 a[18] 和 a[19] 进行比较，必要时交换它们的位置，这样，最大的数就沉到数组的底部，整个排序过程即告结束。

图 7-2 是以 6 个数的数组为例给出选择法排序的示意图。

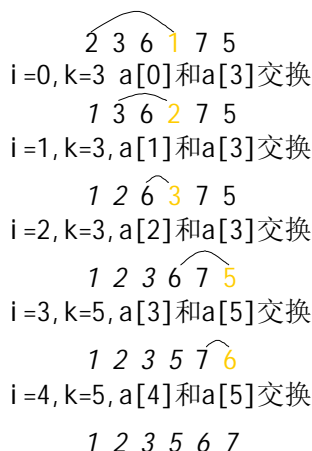


图 7-2 选择法排序示意图

选择排序共要进行 $(n-1)+(n-2)+\cdots+1=n(n-1)/2$ 次比较和最多 $(n-1)$ 次交换，是一种比较简单的排序方法。对应的程序如下：

```
#include<stdio.h>
#define N 10
main()
{
    int i,j,k;
    float a[N],t;
    for(i=0;i<N;i++)
        scanf("%f",&a[i]);
    for(i=0;i<N-1;i++)
    {
        k=i;
        for(j=i+1;j<N;j++)
            if(a[j]<a[k])
                k=j;
        if(i!=k) /* 如果第 k 个元素比第 i 个元素小，交换 */
        {
```

```

        t=a[i];
        a[i]=a[k];
        a[k]=t;
    }
}
for(i=0;i<N;i++)
{
    printf("%6.2f ",a[i]);
    if(i%10==9)
        printf("\n");
}
}

```

程序某次运行时输入数据和显示结果如下：

```

2 3 6 1 7 5 10 8 9 4✓
1.00  2.00  3.00  4.00  5.00  6.00  7.00  8.00  9.00 10.00

```

例 7.5 冒泡法排序。若按从小到大的顺序，冒泡排序的过程是：

① 第一轮，先将 $a[0]$ 与 $a[1]$ 比较，若 $a[0]>a[1]$ ，则将二者交换；然后将 $a[1]$ 与 $a[2]$ 比较，若 $a[1]>a[2]$ ，则将二者交换；然后再进行下面两个相邻元素的比较。经过这一轮 19 次比较后，最大数已沉到数组的底部，即 $a[19]$ 存放的是最大值。

② 第二轮，先将 $a[0]$ 与 $a[1]$ 比较，若 $a[0]>a[1]$ ，则将二者交换；然后将 $a[1]$ 与 $a[2]$ 比较，直到 $a[17]$ 与 $a[18]$ 比较后，就将次大数沉到数组的底部的上一个元素即 $a[18]$ 中。

③ 依此类推，最后一轮将 $a[0]$ 和 $a[1]$ 比较，最终将最小数上浮到数组顶部。

在排序过程中，把小数比成气泡，使之不断向数组顶部上浮，当气泡已一步一步上移到顶部时，排序过程即告结束。

冒泡法排序的比较次数和选择法排序相同，但交换次数比选择排序法多。图 7-3 给出了有 5 个数的数组进行冒泡法排序的示意图。

```

i=0-----
j=0  5 9 8 7 6
j=1  5 9 8 7 6
j=2  5 8 9 7 6
j=3  5 8 7 9 6

i=1-----
j=0  5 8 7 6 9
j=1  5 8 7 6 9
j=2  5 7 8 6 9

i=2-----
j=0  5 7 6 8 9
j=1  5 7 6 8 9

i=3-----
j=0  5 6 7 8 9
      5 6 7 8 9

```

图 7-3 冒泡法排序示意图



```
#include <stdio.h>
#define N 20
main()
{
    int i,j;
    float a[N],t;
    for(i=0;i<N;i++)
        scanf("%f",&a[i]);
    for(i=0;i<N-1;i++)
        for(j=0;j<N-1-i;j++)
            if(a[j]>a[j+1])/* 相邻数据两两交换 */
            {
                t=a[j];
                a[j]=a[j+1];
                a[j+1]=t;
            }
    for(i=0;i<N;i++)
    {
        printf("%6.2f ",a[i]);
        if(i%10==9)printf("\n");
    }
}
```

程序某次运行时输入数据和显示结果如下：

```
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1✓
1.00  2.00  3.00  4.00  5.00  6.00  7.00  8.00  9.00  10.00
11.00 12.00 13.00 14.00 15.00 16.00 17.00 18.00 19.00 20.00
```

3. 数据检索

在处理大量数据时，经常要按某种方法找出所需的数据，这个过程称为检索或查找。当要检索的数据已经排好序，若采用二分检索，搜索次数最多为 \log_2^n 次（其中 n 是数据总个数），是一种比较快的检索方法。

例 7.6 二分检索（对分检索、折半检索）。

假设有 n 个从小到大排好序的数存放在数组 v 中，并设立三个标记： low 指向搜索区间顶部， $high$ 指向搜索区间底部， $mid=(low+high)/2$ 指向搜索区间中部。开始时，令 $low=0$ ， $high=n-1$ ，即将整个数组作为搜索区间。当要查找某个 x 是否在数组 v 中时，按下列步骤进行：

① 令 $mid=(low+high)/2$ ，判断 $v[mid]$ 是否等于 x ，若 $v[mid]=x$ ，则已搜索到，查找结束；若 $v[mid]>x$ ，说明 x 在区间的前半部分，可令 $high=mid-1$ ， low 保持不变，即将该区间前半部分作为新的搜索区间。反之，若 $v[mid]<x$ ，令 $low=mid+1$ ， $high$ 保持不变。经过这次处理，搜索区间已缩小为原来的一半。

② 在新的搜索区间上令 $mid=(low+high)/2$ ，重复与①中相同的操作，继续将搜索区间进一步缩小。

③ 经过有限次（最多工员 \log_2^n 次，假设 v 数组含 16 个元素，则最多搜索 4 次即可确

定 x 是否在数组中) 搜索后, 最终结果是或者 x 已找到, 或者已出现 $low=high$, 即区间缩小为只有一个元素。如果是后一种情况, 就可以确切地得到 x 不在数组中的结论。

下面是二分检索的程序。首先读入 N 个有序数, 然后读入要查找的 x , 继而使用二分检索算法来查找, 并得出相应的结论。

```
#include <stdio.h>
#define N 20
main()
{
    int v[N],x,low,high,mid,i;
    printf("Please input the arranged original data:");
    for(i=0;i<N;i++)
        scanf("%d",&v[i]);
    printf("Please input the number looked for:");
    scanf("%d",&x);
    low=0;
    high=N-1;
    mid=(low+high)/2;
    while(low<high&&x!=v[mid])
    {
        if(x<v[mid])
            high=mid-1;
        else
            low=mid+1;
        mid=(low+high)/2;
    }
    if(x==v[mid])
        printf("%d is at the position %d.\n",x,mid);
    else
        printf("%d isn't in the original data.\n",x);
}
```

程序某次运行结果如下:

```
Please input the arranged original data:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20✓
Please input the number looked for:8✓
8 is at the position 7.
```

例 7.7 有一个二维数组 $a[3][4]$, 找出其中最大和最小的元素, 并指出它们所在的行号和列号。

先分析求最大值的情况: 定义一个 max 变量用来存放最大值, 定义 $row1$ 和 $column1$ 用来记录最大值所在的行号和列号, 先将第 0 行 0 列的元素传给 max , 然后用 max 与第 0 行 1 列的元素比较, 如果后面的元素大, 则将其值存入 max 中, 并且记录所在的行号到 $row1$ 和列号到 $column1$, 否则 max 、 $row1$ 、 $column1$ 的值保持不变。再用 max 与后面第 0 行的各列和后续行的各列比较, 处理方法与前面的相同。这样, 当所有的元素已经比较过, max 变量中的值就是最大值, $row1$ 、 $column1$ 是最大值所在的行号和列号。求最小值的思路类似, 用 min 代表最小值, $row2$ 和 $column2$ 表示最小值所在的行号和列号。



程序如下：

```
#include <stdio.h>
main()
{
    int a[4][4],i,j,max,row1,column1;
    int min,row2,column2;
    for(i=0;i<4;i++)
        for(j=0;j<4;j++)
            scanf("%d",&a[i][j]);
    max=a[0][0];row1=0;column1=0;
    min=a[0][0];row2=0;column2=0;
    for(i=0;i<4;i++)
        for(j=0;j<4;j++)
        {
            if(max<a[i][j])
            {
                max=a[i][j];
                row1=i;
                column1=j;
            }
            if(min>a[i][j])
            {
                min=a[i][j];
                row2=i;
                column2=j;
            }
        }
    printf("max=%d,row1=%d,column1=%d\n",max,row1,column1);
    printf("min=%d,row2=%d,column2=%d\n",min,row2,column2);
}
```

程序运行结果如下：

```
1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 -9↵
max=8,row1=1,column1=3
min=-9,row2=3,column2=3
```

例 7.8 输入成绩表数据并排序。用两个一维数组表示学号和分数，num[j]表示第 j 个学生的学号，score[j]表示第 j 个学生的分数。

```
#include <stdio.h>
#define N 4
main()
{
    int j,k,p;
    float t;
    int num[N];
    float score[N];
    /*输入学号和分数*/
    for(j=0;j<N;j++)
```

```
{
    printf("please input number score");
    scanf("%d%f",&num[j],&score[j]);
}
/*按分数从低到高排序*/
for(j=0;j<N-1;j++)
{
    p=j;
    for(k=j+1;k<N;k++)
        if(score[p]>score[k])
            p=k;
    t=score[j];
    score[j]=score[p];
    score[p]=t;
    t=num[j];
    num[j]=num[p];
    num[p]=t;
}
/*输出排序后的名次、学号和成绩*/
printf("place|--number--|----score\n");
for(j=0;j<N;j++)
{
    printf(" %d | %d | %5.1f\n",j+1,num[j],score[j]);
    printf("-----\n");
}
}
```

例 7.9 用二维数组处理成绩表。设数组的第 0 列表示学号，第 1 列表示分数，进行输入及排序。为了简单，学号和分数都用整型数表示。

```
#include <stdio.h>
#define N 4
main()
{
    int j,k,p;
    float t;
    int stu[N][2];
    /*输入学号和成绩*/
    for(j=0;j<N;j++)
    {
        printf("please input number score");
        scanf("%d%d",&stu[j][0],&stu[j][1]);
    }
    /*按成绩从低到高大排序*/
    for(j=0;j<N-1;j++)
    {
        p=j;
        for(k=j+1;k<N;k++)
            if(stu[p][1]>stu[k][1])
                p=k;
    }
}
```



```
t=stu[j][0];
stu[j][0]=stu[p][0];
stu[p][0]=t;
t=stu[j][1];
stu[j][1]=stu[p][1];
stu[p][1]=t;
}
/*输出排序后的名次、学号和分数*/
printf("place|--number--|--score\n");
for(j=0;j<N;j++)
{
    printf("  %d  |  %5d      |  %5d\n",j+1,stu[j][0],stu[j][1]);
    printf("-----\n");
}
}
```

例 7.10 成绩表转置。

```
#include <stdio.h>
#define N 4
main()
{
    int j,k,p;
    float t;
    int stu[N][2];
    int student[2][N];
    /*输入学号和分数并输出原数组*/
    for(j=0;j<N;j++)
    {
        printf("please input number score");
        scanf("%d%d",&stu[j][0],&stu[j][1]);
        printf("%d  %d",stu[j][0],stu[j][1]);
    }
    for(j=0;j<N;j++)/* 将 stu 数组的 j 行 k 列元素赋值给 student 数组的 k 行 j 列 */
        for(k=0;k<2;k++)
            student[k][j]=stu[j][k];
    /* 输出转置后的数组*/
    for(j=0;j<2;j++)
    {
        printf("\n%5d %5d %5d %5d\n",student[j][0],
            student[j][1],student[j][2],student[j][3]);
    }
}
```

例 7.11 根据平时成绩和期末成绩，计算总评成绩，计算公式为：

总评成绩= 平时成绩*20%+期末成绩*80%

定义 3 个二维数组，分别表示平时、期末和总评成绩表，程序如下：

```
#include <stdio.h>
#define N 4
```

```

main()
{
    int j,k,p;
    float t;
    int stu1[N][2],stu2[N][2],stu3[N][2];
    /*输入学号和平时成绩*/
    for(j=0;j<N;j++)
    {
        printf("please input number score as usual:");
        scanf("%d%d",&stu1[j][0],&stu1[j][1]);
    }
    /*输入学号和期末成绩*/
    for(j=0;j<N;j++)
    {
        printf("please input number score in final exam:");
        scanf("%d%d",&stu2[j][0],&stu2[j][1]);
    }
    /*总评成绩=平时成绩*20%+期末成绩*80%*/
    for(j=0;j<N;j++)
    {
        stu3[j][0]=stu1[j][0];
        stu3[j][1]=0.2*stu1[j][1]+0.8*stu2[j][1];
    }
    /*输出总评成绩*/
    printf("---number--|----score----\n");
    for(j=0;j<N;j++)
    {
        printf(" %5d    |    %5d\n",stu3[j][0],stu3[j][1]);
        printf("-----\n");
    }
}

```

7.6 字符串和字符型数组

7.6.1 字符串

字符串又叫字符串常量，在存储时，从左到右的顺序依次占用连续的存储单元，每个字符占用 1 个字节，存放其对应的 ASCII 代码值。C 编译系统还会自动在每个字符串常量的末尾追加一个零字符 NULL，即'\0'，作为字符串的结束标志。因此，如果一个字符串常量里含有 n 个字符，则它要占用的存储空间为 (n+1) 个字节。例如，字符串常量"I am a student"包含有 14 个字符（其中有 3 个空格字符），它要占用 15 个字节的存储单元，即：

I		a	m		a		s	t	u	d	e	n	t	\0
---	--	---	---	--	---	--	---	---	---	---	---	---	---	----

按其 ASCII 码存放（十六进制表示），存储格式为

49	20	61	6d	20	61	20	73	74	75	64	65	6e	74	00
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----



最后的 NULL 字符（'\0'，十六进制表示为 00）是系统自动加上的，输出时不显示。

C 语言可以通过字符数组来处理字符串，例如可以用 `char s[15]` 来存放字符串 “I am a student”，则每个数组元素的值为上面每个单元的值。注意，为了存放有 `n` 个字符的字符串，定义字符数组时元素个数至少设为 `(n+1)`。

7.6.2 字符型数组的定义和初始化

C 语言中有字符型常数和字符串常数，但只有字符型变量没有字符串变量，所以需要利用字符型数组存放和处理字符串。其中，一维字符型数组可存放一个字符串，二维字符型数组可存放多个字符串。字符型数组也要先定义才可以使用。

字符型数组的定义形式如下：

```
char <数组名>[<常量表达式>];
```

例如，下面语句定义两个字符型数组 `str1` 和 `str2`：

```
char str1[30];  
char str2[10][80];
```

则 `str1` 可存放 1 个长度不超过 29 个字符的字符串，`str2` 可存放 10 个字符串，每个字符串的长度都不超过 79 个字符。注意，每个字符串末尾都有 1 个零字符（'\0'），它要占 1 个字节的存储单元。

对字符型数组进行初始化，即在定义一个字符数组时给它指定初值。

1. 一维字符型数组的初始化

C 语言用一维字符型数组存放多个字符或一个字符串。一维字符型数组的初始化，有以下两种方式：

（1）用字符常数初始化。即将字符常数依次存放在花括号中，以逐个为数组中各元素指定初值字符，字符在内存中以相应的 ASCII 代码存放。例如：

```
char str[12]={ 'T','h','e',' ','s','t','r','i','n','g','.', '\0' };
```

这样，字符型数组 `str` 中就存放了一个字符串 “The string.”，每一个数组元素 `str[i]` 存放一个字符，最后一个元素 `str[11]` 存放串结束符 '\0'。注意，用这种方式初始化，如果是字符串，最后一个字符必须是 '\0'，不能是其他字符；否则只是存放的字符元素，不是字符串。

也可以部分初始化，如：`char str[8]={ 'a','b','c','d' }`；未赋值的部分自动为 '\0'。

对于全部元素指定初值的情况下，字符数组的大小可以不必定义。如：`char str[]={ 'a','b','c' }`；系统自动确定字符型数组 `str` 有 3 个元素。

（2）直接用字符串常量初始化，字符串常量加不加花括号均可。例如：`char str[12]="The string.";` 或 `char str[12]={ "The string." }`；

等价于 `char str[12]={ 'T','h','e',' ','s','t','r','i','n','g','.', '\0' }`；这时，C 编译程序会自动将字符串中各字符逐个地顺序赋给字符数组中各元素，最后自动在末尾增加一个串结束符 '\0'。需要注意的是，用这种方式初始化时，一定要使定义的数组的大小至少比所赋的字符串长度多 1。

在用字符串初始化一个一维字符数组时，可以不指定数组大小并且省略字符串常量外面的花括号。例如：`char str[]="The string."`；系统自动根据字符串大小为数组设定长度为 12。

2. 二维字符型数组的初始化

二维字符型数组相当于一维字符串数组，即一维数组的每个元素是字符串，其初始化也有两种方式。

(1) 字符常数方式。如：

```
char Language[3][8]={{'B','A','S','I','C','\0'},
{'F','O','R','T','R','A','N','\0'},{'C','\0'}};
```

它相当于定义了 3 个字符串，每个字符串最多可存放 7 个字符。

(2) 字符串常数方式。如：

```
char Language[3][8]={"BASIC","FORTRAN","C"};
```

显然，第 (2) 种方式比第 (1) 种方式简便。

7.6.3 字符型数组的输入和输出

1. 用 scanf()和 printf()实现字符型数组的输入和输出

(1) 用 “%c” 控制的 scanf()和 printf()可以逐个输入和输出字符数组中的各个字符。

例 7.12 用 scanf()和 printf()实现字符型数组的输入和输出。

```
#include <stdio.h>
main()
{
    int i;
    char ch[10];
    for(i=0;i<9;i++)
        scanf("%c",&ch[i]);
    ch[9]= '\0';
    for(i=0;ch[i]!='\0';i++)
        printf("%c",ch[i]);
}
```

用这种方式输入时，从键盘输入的字符数一定要比定义的长度少 1。程序自动将最后一个位置放入 '\0' 字符。输出时，可以用 `ch[i] != '\0'` 来作为继续循环的条件。

(2) 用 “%s” 控制的 scanf()和 printf()可以输入和输出字符串。例如，程序段：

```
char ch[16];
scanf("%s",ch);
printf("%s\n",ch);
```

注意，由于数组名代表数组起始地址，因此在 scanf()函数中只需写数组名 `ch` 即可，而不必写成 `scanf("%s",&ch)`。在 scanf()和 printf()中都用了数组名 `ch`，而没用地址运算符 `&`，这是因为 “%s” 是直接控制字符串的，它只要求某个字符串的起始地址作为参数。当 “%s”



用在 `scanf()` 中时，会自动把用户输入的回车符转换成 `\0` 并加在字符串的末尾；用在 `printf()` 时，从 `ch` 代表的地址开始逐个输出字符，遇到 `\0` 就结束输出。

若用 `scanf("%s",ch);` 语句向字符数组输入一个字符串，可在运行时从键盘输入并按回车键即可，不必在字符串两端加双引号。例如，可按以下方式输入字符串：

```
Computer✓
```

在按回车键后，回车键前面的字符作为一个字符串输入，系统自动在最后加一个字符串结束标志 `\0`，并且输入给数组中的字符个数是 9 而不是 8。



“%s” 用在 `scanf()` 中控制输入时，输入的字符串不能含有空格或制表符。因为 C 规定，用 `scanf()` 输入字符串是以空格或回车符作为字符串间隔符号；当注意 “%s” 遇到空格或制表符时，就认为输入结束。

如果有以下 `scanf()` 函数语句：

```
scanf("%s%s",str1,str2);
```

当输入：

```
good morning✓
```

时，则将 `good` 和 `\0` 输入到字符数组 `str1` 中，将 `morning` 和 `\0` 送到字符数组 `str2` 中。

如果输入的字符串中包含空格，例如，执行：

```
scanf("%s",ch);
```

当从键盘输入 “good morning” 时，`ch` 的值为 “good” 而不是 “good morning”。解决这一问题的办法是对输入长度加以限定，例如：

```
scanf("%13s",ch);
```

同样的输入可使 `ch` 的值为 “good morning”。

“%s” 用在 `printf()` 中时，不会因为遇到空格或制表符而结束输出。

2. 字符串输入函数 `gets()` 和字符串输出函数 `puts()`

用 `gets()` 可以直接输入字符串，直至遇到回车键为止，它不受输入字符串中空格或制表符的限制。使用 `gets()` 的一般格式为：

```
gets(字符型数组名);
```

用 `puts()` 可以输出字符串，字符串中空格或回车不影响，遇到 `\0` 结束串输出，而且自动把字符串末尾的 `\0` 字符转换成换行符（而 “%s” 控制的 `printf()` 则没有将字符串末尾的 `\0` 转换成换行符的功能，必须增加 “`\n`” 来实现换行）。调用它的一般格式为：

```
puts(字符型数组名 或 字符串常量);
```

例如：

```
puts("abc def \n hij");
```

输出为:

```
abc def
hij
puts("abc \0 def \n hij");
```

输出为:

```
abc
```

gets()和 puts()函数都要求数组名作参数,不用在数组名前加“&”。这两个函数定义在标题文件 stdio.h 中,使用前要用“#include <stdio.h>”语句将标题文件包含进来。

例 7.13 一维字符数组的输入和输出。

```
#include <stdio.h>
main()
{
    char name[11];
    printf("please input a name:");
    gets(name);
    puts(name);
}
```

程序运行情况如下:

```
please input a name:Bill Gates✓
Bill Gates
```

例 7.14 二维字符数组的输入和输出。

```
#include <stdio.h>
main()
{
    int i;
    char country[3][8];
    for(i=0;i<3;i++)
    {
        printf("Please input a country name:\n");
        gets(country[i]);
    }
    for(i=0;i<3;i++)
        puts(country[i]);
}
```

程序中用到“country[i]”这样的符号,它代表数组 country 第 i 行的首地址。程序运行情况如下:

```
Please input a country name:
China✓
Please input a country name:
America✓
Please input a country name:
```




```
Japan✓  
China  
America  
Japan
```

其中，前面的行是程序显示的提示行及从键盘输入的数据，后 3 行是输出的结果。

7.6.4 字符串处理函数

C 语言本身不提供字符串处理的能力，但是 C 编译系统提供了大量的字符串处理库函数，它们定义在标题文件 `string.h` 中，使用时只要包含这个标题文件，就可以使用其中的字符串处理函数。下面介绍几个常用的字符串处理函数。

1. 字符串连接函数 `strcat(字符数组 1, 字符数组 2)`

该函数用来连接两个字符数组中的字符串，即把“字符数组 2”接到“字符数组 1”的后面，结果放在“字符数组 1”中，函数调用后的返回值为“字符数组 1”的地址。

“字符数组 1”必须足够大，以便容纳连接后的新字符串。连接后在新串最后只保留一个串结束符 `'\0'`。

2. 求字符串长度函数 `strlen()`

该函数用来计算字符串的长度，即所给字符串中包含的字符个数（不计字符串末尾的 `'\0'` 字符），函数返回值为整型，其调用格式为：

```
strlen(字符串);
```

其中的参数可以是字符型数组名或字符串常数。

例 7.15 字符串长度的输出。

```
#include <stdio.h>  
#include <string.h>  
main()  
{  
    char s[]="good morning";  
    printf("%d\n",strlen(s));  
    printf("%d\n",strlen("good morning"));  
}
```

程序运行后，输出两行 12。

3. 字符串复制函数 `strcpy()`

该函数用来将一个字符串复制到另一个字符串中，函数类型为 `void`，其调用格式为：

```
strcpy(字符数组 1, 字符串 2);
```

它可以将“字符串 2”中的字符复制到“字符数组 1”中。其中“字符数组 1”必须定义的足够大，以容纳被复制的字符串。函数中的参数“字符数组 1”必须是字符型数组名，

“字符串 2”可以是字符型数组名或字符串常数。如：

```
char string1[20],string2[20];
strcpy(string1,"FORTRAN");
```

表示将字符串常数“FORTRAN”复制到 string1 中，实现了赋值的效果。而：

```
strcpy(string2,string1);
```

则表示将 string1 中的字符全部复制到 string2 中，这时要求 string2 的大小能容纳 string1 中的全部字符。

注意，不能用赋值语句将一个字符串常量或字符数组直接赋给一个字符型数组，只能用 strcpy 函数。

4. 字符串比较函数 strcmp()

该函数用来对两个字符串进行比较，看第一个字符串是大于、等于还是小于第二个字符串，函数类型为整型，它的调用格式为：

```
strcmp(字符串 1,字符串 2);
```

其中“字符串 1”和“字符串 2”可以是字符型数组名或字符串常数，其作用是对“字符串 1”和“字符串 2”从左至右逐个字符进行比较（按其 ASCII 代码值比较），直到出现不同的字符或遇到‘\0’为止，如果全部字符均相同，则认为两个字符串相等，函数返回 0 值；若出现不相同的字符，则以第一个不相同的字符的比较结果为准，比较后的结果作为函数值返回：若“字符串 1”>“字符串 2”，则函数返回一个正整数，其值为两个字符 ASCII 代码值的差；若“字符串 1”<“字符串 2”，则函数返回一个负整数，其值为两个字符 ASCII 代码值的差。

例如：

```
Printf("%d", strcmp("ab","cd"));
```

返回的结果是-2，因为第一个字符就不相等，字母 a 的 ASCII 代码值比 c 的 ASCII 代码值小 2。

7.6.5 字符数组举例

例 7.16 用字符*输出一个菱形图案。对于一个 5 行的菱形，可以先将一个二维数组初始化为一个三角形的形状，用双重循环输出菱形的上半部，再用一个双重循环输出数组的第二行和第一行使形成菱形的下半部。

```
#include<stdio.h>
main()
{
    int i,j;
    static char triangle[][5]={{' ',' ','*',' ',' '},
                                {' ','*',' ','*',' '},{ '*',' ',' ',' ','*'}};

    for(i=0;i<3;i++)
    {
```



```
        for(j=0;j<5;j++)
            printf("%c",triangle[i][j]);
        printf("\n");
    }
    for(i=1;i>=0;i--)
    {
        for(j=0;j<5;j++)
            printf("%c",triangle[i][j]);
        printf("\n");
    }
}
```

程序执行时输出如下图形:

```
*
* *
*  *
* *
*

```

例 7.17 编写一个字符串比较程序。键盘输入两字符串，一个存到 s1 数组，一个存到 s2 数组，对两字符串的大小进行比较：如果出现不一致的字符，则字符大的字符串大。

```
#include <stdio.h>
#include <string.h>
main()
{
    int i=0;
    char s1[10],s2[10];
    scanf("%s",s1);
    scanf("%s",s2);
    while(s1[i]==s2[i]&& s2[i]!='\0')
        i++;
    if(s1[i]==s2[i])
        printf("s1=s2\n");
    else if(s1[i]>s2[i])
        printf("s1>s2\n");
    else
        printf("s1<s2\n");
}
```

要 点 回 顾

1. 数组用来表示具有相同数据类型且按顺序存放的一组变量，构成数组的变量称为数组元素。数组要先定义后使用，定义一维数组的一般形式为：

[存储类别] 数据类型 数组名[常量表达式];

“数组名”后面方括号中的“常量表达式”规定该数组中可容纳的元素个数，其值称为维界，必须为正整数。

定义二维数组的一般形式是：

```
[存储类别] 数据类型 数组名[常量表达式 1][常量表达式 2]
```

把“常量表达式 1”称为第二维或行，把“常量表达式 2”称为第一维或列。

2. 进行数组的定义就是让编译程序为每个数组安排一片连续的存储单元来依次存放数组的各个元素。对一维数组来说，各个元素按下标由小到大顺序存放；对二维数组来说，先按行的顺序，再按列的顺序依次存放各个元素。数组名将作为该数组所占的连续存储区的起始地址；数据类型将决定该数组的每一个元素要占用多少字节的存储单元。

3. 数组初始化就是在定义时给数组元素设初始值。一维数组初始化时，把所赋初值按顺序放在等号右边的花括号中，各常量之间用逗号隔开。

4. 对数组全部元素初始化的数组定义可以省略方括号中的数组大小，编译器会统计花括号之间的元素个数并自动确定数组大小。不指定数组长度的定义和初始化用于多维数组时，必须指定除最左边维界之外的所有维界。

5. 数组在定义后其元素即可被引用，引用时须保证没有超出数组边界，C 编译系统不检查下标是否越界。引用形式为：数组名[下标]。

6. 只能逐个对数组元素赋值，数值型数组一般用循环实现对各个元素赋值。

7. 求一批数据中的最大或最小值以及进行排序和检索是利用数组的典型问题。

8. C 语言可以通过字符数组来处理字符串，字符型数组的定义形式：

```
char <数组名>[<常量表达式>];
```

9. C 语言用一维字符型数组存放多个字符或一个字符串。一维字符型数组的初始化，有以下方式：

(1) 用字符常数初始化，即将字符常数依次放在花括号中，以逐个为数组中各元素指定初值字符

(2) 直接用字符串常量初始化，字符串常量加不加花括号均可。

10. 可以用 `scanf()` 和 `printf()` 实现字符数组的输入和输出：

(1) 用“%c”控制的 `scanf()` 和 `printf()` 可以逐个输入和输出字符数组中的各个字符。

(2) 用“%s”控制的 `scanf()` 和 `printf()` 可以输入和输出字符串。

11. 用 `gets()` 可以直接输入字符串，直至遇到回车键为止，它不受输入字符中空格或制表符的限制。使用该函数的一般格式为：`gets(字符型数组名);`。

12. 用 `puts()` 可以输出字符串，字符串中可以含空格或回车，遇到“\0”结束串输出，而且自动把字符串末尾的“\0”字符转换成换行符。使用该函数的格式为：`puts(字符型数组名 或 字符串常量);`。

13. 常用的字符串处理函数有：

(1) 字符串连接函数 `strcat(字符数组 1, 字符数组 2)`。

(2) 求字符串长度函数 `strlen()`。

(3) 字符串复制函数 `strcpy()`。

(4) 字符串比较函数 `strcmp()`。



习 题

1. 合法的数组定义是 ()。

A) int a[]="string";

B) int a[5]={0,1,2,3,4,5};

C) vlst s="string";

D) char a[]={0,1,2,3,4,5};

2. 给出以下定义:

char x[]="abcdefg";

char y[]={ 'a','b','c','d','e','f','g'};

则正确的叙述为 ()。

A) 数组 x 和数组 y 等价

B) 数组 x 和数组 y 的长度相同

C) 数组 x 的长度大于数组 y 的长度

D) 数组 x 的长度小于数组 y 的长度

3. 下列描述中不正确的是 ()。

A) 字符型数组中可以存放字符串

B) 可以对字符型数组进行整体输入、输出

C) 可以对整型数组进行整体输入、输出

D) 不能在赋值语句中通过赋值运算符“=”对字符型数组进行整体赋值

4. 以下程序中, 主函数调用了 LineMax 函数, 实现在 N 行 M 列的二维数组中找出每一行上的最大值。请填空。

```
#define N 3
#define M 4
void LineMax(int x[N][M])
{
    int i,j,p;
    for(i=0; i<N;i++)
    {
        p=0;
        for(j=1; j<M;j++)
            if(x[i][p]<x[i][j])
                _____;
        printf("The max value in line %d is %d\n",i, _____);
    }
}
main()
{
    int x[N][M]={1,5,7,4,2,6,4,3,8,2,3,1};
    _____
}
```

5. 假定 int 类型变量占用两个字节, 若有定义: int x[10]={0,2,4};, 则数组 x 在内存中所占字节数是 ()。

A) 3

B) 6

C) 10

D) 20

6. 以下数组定义中不正确的是 ()。

A) int a[2][3];

- B) int b[][3]={0,1,2,3};
C) int c[100][100]={0};
D) int d[3][]={{1,2},{1,2,3},{1,2,3,4}};

7. 若已定义: int a[10];i;, 以下 fun 函数的功能是: 在第一个循环中给前 10 个数组元素依次赋值 1,2,3,4,5,6,7,8,9,10; 在第二个循环中使 a 数组前 10 个元素中的值对称折叠, 变成 1,2,3,4,5,5,4,3,2,1。请填空。

```
fun( int a[ ])
{
    int i;
    for(i=1; i<=10; i++)
        _____=i;
    for(i=0; i<5; i++)
        _____=a[i];
}
```

8. 若有定义语句: char s[100],d[100]; int j=0,i=0;, 且 s 中已赋字符串, 请填空以实现字符串复制 (注: 不得使用逗号表达式)。

```
while(s[i]){d[j]=_____;j++;}
d[j]=0;
```

9. 以下程序中函数 sort 的功能是对 a 数组中的数据进行由大到小的排序。

```
void sort(int a[],int n)
{
    int i,j,t;
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
            if(a[i]<a[j])
            {
                t=a[i];
                a[i]=a[j];
                a[j]=t;
            }
}
main()
{
    int aa[10]={1,2,3,4,5,6,7,8,9,10},i;
    sort(&aa[3],5);
    for(i=0;i<10;i++)printf("%d,",aa[i]);
    printf("\n");
}
```

程序运行后的输出结果是_____。

10. 以下程序中的函数 reverse 的功能是将 a 所指数组中的内容进行逆置。

```
void reverse(int a[],int n)
{
    int i,t;
```



```
    for(i=0;i<n/2;i++)
    {
        t=a[i];
        a[i]=a[n-1-i];
        a[n-1-i]=t;
    }
}
main()
{
    int b[10]={1,2,3,4,5,6,7,8,9,10};
    int i,s=0;
    reverse(b,8);
    for(i=6;i<10;i++)
        s+=b[i];
    printf("%d\n",s);
}
```

程序运行后的输出结果是_____。

11. 有以下程序:

```
main()
{
    int aa[4][4]={{1,2,3,4},{5,6,7,8},{3,9,10,2},{4,2,9,6}};
    int i,s=0;
    for(i=0;i<4;i++)s+=aa[i][1];
    printf("%d\n",s);
}
```

程序运行后的输出结果是_____。

12. 函数 fun 的功能是: 使一个字符串按逆序存放, 请填空。

```
void fun(char str[])
{
    char m;
    int i,j;
    for(i=0,j=strlen(str);i<_____;i++,j--)
    {
        m=str[i];
        str[i]=_____;
        str[j-1]=m;
    }
    printf("%s\n",str);
}
```

13. 以下程序用来对从键盘上输入的两个字符串进行比较, 然后输出两个字符串中第一个不相同字符的 ASCII 码之差。例如: 输入的两个字符串分别为 abcdef 和 abceef, 则输出为-1。请填空。

```
#include
main()
{
```

```
char str[100],str2[100],c;
int i,s;
printf("\n input string 1:\n");
gets(str1);
printf("\n input string 2:\n");
gets(str2);
i=0;
while((str1[i]==str2[i]&&(str1[i]!=_____))
    i++;
s=_____;
printf("%d\n",s);
}
```

14. 以下程序的功能是：从键盘上输入若干个学生的成绩，计算出平均成绩，并输出低于平均分的学生成绩，用输入负数结束输入。请填空。

```
main()
{
    float x[1000], sum=0.0,ave, a;
    int n=0, i;
    printf("Enter mark: \n");scanf("%f",&a);
    while(a>=0.0&& n<1000)
    {
        sum+_____;
        x[n]= _____;
        n++;
        scanf("%f",&a);
    }
    ave=_____;
    printf("output: \n");
    printf("ave=%f\n",ave);
    for( i=0;i<n;i++)
        if(x[i]<ave)
            printf("%f",x[i]);
}
```




第 8 章 指 针

本章的学习目标:

了解指针的概念和指针的定义方法,掌握用指针访问变量、数组以及用指针使用动态内存的方法,会利用指针处理数组和字符串。了解指针和函数之间的关系,会利用指针作为形式参数设计函数,会利用指针作函数的形参进行数组处理。了解指针数组和二级指针的含义,了解函数型指针和指针型函数,了解命令行参数的概念,会设计带参数的 `main()` 函数。

指针是指数据在内存中的地址,指针变量是指存放地址的变量。用指针可以直接访问内存单元中的数据,有时可以写出更紧凑和更有效的程序代码;指针支持内存的动态分配,还能有效地处理诸如链表、树、图等复杂的数据结构。指针是 C 语言的最强特性,但是指针的用法灵活,不易掌握,如果使用不当很容易使程序出错,并且产生的错误不易发现,所以指针应用又是 C 语言的难点。

8.1 地址以及和地址有关的运算

8.1.1 地址的概念

在计算机中,内存是一个连续编号或编址的空间,每一个存储单元(在微型计算机中通常是一个字节)都有一个固定的编号,这个编号称为地址。由于不同的数据类型占用不同字节的存储空间,而每一字节都有一个地址,当定义一个变量时,系统会为该变量分配内存空间。内存空间的首字节地址称为该变量的地址,内存空间的内容称为该变量的值。

例如,我们在程序中进行了如下的定义:

```
char ch='A';  
float a=34.0;  
int b[2];
```

编译系统会根据整个程序的情况,为变量 `ch` 分配 1 字节、为变量 `a` 分配 4 字节的存储空间,为数组 `b` 分配一片连续的存储空间,每个数组元素占 2 字节(如图 8-1 所示,每一方格为 1 字节)。每个字节单元都有地址编号,每个变量的首字节地址即为该变量的地址;每个数组元素的首字节地址即为该元素的地址,而 `b[0]` 元素的地址又是数组的首地址,也是数组名 `b` 的值。在实际运行时,变量的地址分配情况取决于当时计算机内存状况和程序状况,图 8-1 仅为示意。

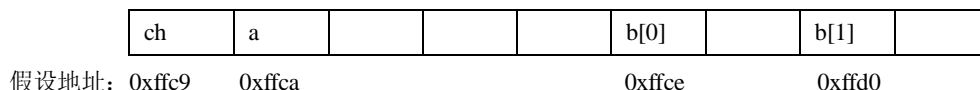


图 8-1 地址的概念



注意

数组名 `b` 的值是它的第一个元素 `b[0]` 的地址，即数组名是一个地址常数，它的值就是数组的首地址。

8.1.2 取地址运算符和访问地址运算符

1. 取地址运算符&

变量或数组元素的地址是由系统分配的，用 `&` 操作符可获取变量或数组元素的地址，它的格式是：

`&` 变量名或数组元素名

取地址运算符 `&` 是一个一元运算符，它的形式与位操作中的“按位与”运算符相同，但后者是一个二元运算符，所以一般在使用时不会发生混淆。

将 `&` 施加在一个变量或数组元素上，就可以得到该变量或数组元素的地址。在数组中，由于数组名本身是数组首地址，因此 `b` 和 `&b[0]` 等价。

下面，我们可以编一个小程序来获取程序中定义的变量及数组元素的地址。

例 8.1 取地址运算。

```
#include <stdio.h>
main()
{
    char ch='A';
    float a=34.0;
    int b[2]={1,2};
    printf("address of ch is %x\n",&ch);
    printf("address of a is %x\n",&a);
    printf("address of array b is %x\n",b);
    printf("address of b[0] is %x\n",&b[0]);
    printf("address of b[1] is %x\n",&b[1]);
}
```

程序中用 `%x` 来控制地址量的输出，得到用 4 位十六进制数表示的地址。程序运行结果如下（参考值）：

```
address of ch is ffc9
address of a is ffca
address of array b is ffce
address of b[0] is ffce
address of b[1] is ffd0
```



注意

不能对常数、符号常数、表达式进行取地址运算，如 `&25`、`&(x+y)` 等是错误的。

2. 访问地址运算符*和[]（指向运算符和下标运算符）

访问地址运算符用来访问特定地址中的数据。前面我们是用变量名或数组元素名来访问数据，此时系统首先要根据变量名或数组元素名找到它们在内存中的地址，然后才能访



问其中的数据。但访问地址运算符使系统可以根据给出的地址量直接访问数据。

C 语言中访问地址运算符有两种：`*`和`[]`。

(1) `*`运算符

`*`运算符的作用是将`*`施加在一个地址量上,以得到该地址中的数据。`*`又叫指向运算符,应用格式为:

`*地址量`

`*`也是一元运算符,它的形式与乘法运算符相同,因为后者是二元运算符,使用中不会产生混淆。

假设 `a` 是变量, `&a` 就是 `a` 在内存中的地址, `*(&a)` 就是访问 `a` 所在地址中的数据。同样,一维数组名 `d` 是一个地址量,代表数组 `d` 的首地址,即 `d` 和 `&d[0]` 是等价的,而 `*d` 就是访问 `d[0]` 所在地址中的数据,即 `*(&d[0])`,也就是 `d[0]` 的值。

运算符`*`和`&`处于同一优先级,结合性都是从右至左,因此`*&a`与`*(&a)`等价。

(2) `[]`运算符

`[]`运算符也用来访问地址中的数据,与下标表示形式相同,又叫下标运算符,应用格式如下:

`地址量[整型表达式]`

其中,“地址量”通常是数组名;`[]`运算符的功能是从地址量开始,向高地址方向移动若干个存储单元后,取该地址中的值。所移动的存储单元个数,与`[]`中整型表达式的值有关,还与所存放的数据类型有关。

比如 `d[n]` 表示数组 `d` 中第 `n` 个元素的值,其访问过程为: `d[n]` 表示从地址 `d` 向高地址方向移动 `n` 个数据单元后,取对应数据单元中的值。例如, `d[0]` 是在地址 `d` 的基础上向上移动 0 个存储单元,其地址没有发生变化, `d[1]` 则在 `d` 的基础上向上移动 1 个数据单元,其地址增加了 1 个数据单元的字节数。这里所说的“数据单元”是指数据占用的全部字节:对整型数组,一个数据单元表示 2 字节;对浮点型数组,一个数据单元表示 4 字节等,这是由计算机自动完成的。

(3) `*`和`[]`的等价性

如果定义了整型数组 `d[n+1]`,图 8-2 给出了数组元素的两种地址表示形式和访问数据的不同形式。

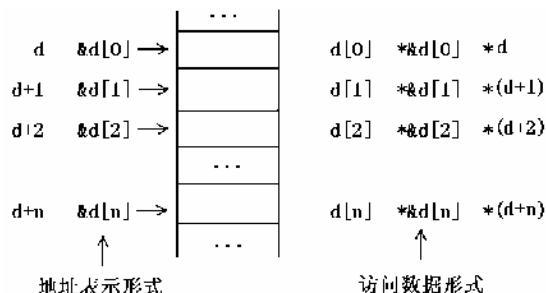


图 8-2 一维整型数组地址表示及数据访问等价形式

图中,&d[i]表示数组中第 i 个元素的地址,而 d+i 表示从 d 后移 i 个数据单元后的地址,也是第 i 个元素的地址,所以两者是等价的。从图中可以看出,数组名就是数组的首地址,是地址常量,所以不允许对数组名重新赋值。

*和[]都是访问地址运算,用来访问地址中的数据,它们是等价的。对任何地址量,都可以在其前面加上*或者在其后面加上[]来访问其中的数据。例如,对变量 x 而言,&x 是地址量,则 x、*x 和(&x)[0]等价;对一维数组元素 d[i]来说,第 i 个元素的地址用&d[i]表示,也可用 d+i 表示(即相对于数组首地址 d 移动 i 个数据单元后的地址),所以 d[i]、*&d[i] 和*(d+i)等价。

这样在表示数据时候就可以很灵活,例如表示简单变量 a 时也可以写成(&a)[0],表示数组元素 d[i]时可以写成*&d[i]和*(d+i)。

请分析下面的程序实例,以加深对访问数据的几种形式的理解。

例 8.2 根据访问地址运算的等价性,要输出变量和数组元素的值,可以有多种方法。

```
#include <stdio.h>
main()
{
    char ch='a';
    int b[2]={1,2};
    printf("ch=%c",ch);
    printf("ch=%c",(&ch)[0]); /* (&ch)表示先得到 ch 的地址 */
    printf("ch=%c\n",&ch);
    printf("b[0]=%d",b[0]); /* 用三种形式输出 b[0]的值 */
    printf("b[0]=%d",&b[0]);
    printf("b[0]=%d\n",*b);
    printf("b[1]=%d",b[1]); /* 用三种形式输出 b[1]的值 */
    printf("b[1]=%d",&b[1]);
    printf("b[1]=%d\n",*(b+1));
}
```

运行结果:

```
ch=a,ch=a,ch=a
b[0]=1,b[0]=1,b[0]=1
b[1]=2,b[1]=2,b[1]=2
```

8.2 指针的概念及指针变量的定义

采用访问地址的方法来存取数据,速度快,效率高。但是程序运行过程中数据在内存中的实际地址是由系统分配的,用户并不知道数据的具体地址。C 语言把地址存放在一种特殊类型的变量中,这样就可以把对地址的操作转换为对变量的操作,处理起来更加方便。专门用来存放地址的变量,就是指针变量,指针变量中存放的地址称为指针值。有时把指针变量和指针值统称为指针。



8.2.1 指针变量的定义

指针变量和其他类型的变量一样，使用前必须先定义。定义指针变量的一般形式为：

[存储类别] 数据类型 *指针变量名 1, *指针变量名 2, ...

这里的*只是一个说明符，它出现在定义语句中，表示后面的变量是指针变量。指针的数据类型是表示该指针可以指向何种类型的数据；每一种基本数据类型都有相应的指针类型，指针在引用中类型必须匹配。指针本身是整型的，指针变量的值表示地址。例如：

```
char * pc;  
int * pi;  
float * pf;
```

定义了字符型、整型和浮点型指针 pc、pi 和 pf，它们可分别用来指向字符型、整型和实型的变量或数组，也就是存放对应变量或数组的地址。而 pc、pi 和 pf 本身也有自己的地址，各占 2 个字节存储空间。

指针和普通变量一样，也有 auto、static、register 和 extern 四种存储类别，它决定了指针的生存期；指针定义在程序中的不同位置，决定了指针的作用范围。

8.2.2 将指针指向对象的方法、空指针和 void 型指针

1. 使指针指向对象的方法

指针只有指向某一对象，如变量、数组、函数、结构、文件等，才有实际意义。如果使用未指向特定对象的指针，就会出现错误，甚至使程序运行失败。要使指针指向某一对象，可以采用如下方式。

(1) 指针变量可以在定义时初始化。例如：

```
char c, *p1=&c, *p2=p1;  
float a;  
float *pa=&a; /* 但赋值语句 *pa=&a 是错的 */
```

定义了指向 char 型变量 c 的指针 p1 和 p2 以及指向实型变量 a 的指针 pa。在初始化过程中，指针 p1 先得到变量 c 的地址，再把 p1 的值赋给指针 p2，使 p1 和 p2 同时指向变量 c。这些变量和指针占用内存情况如图 8-3 所示。

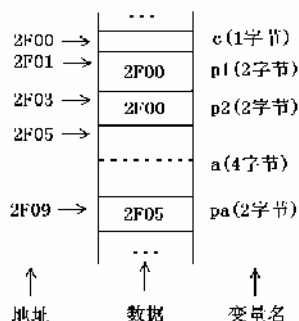


图 8-3 指针的概念

从图中可看到, 变量 `c` 占用地址为 `0x2F00` 的存储单元, 指针 `p1` 和 `p2` 分别占用从地址 `0x2F01` 和 `0x2F03` 开始的 2 字节存储单元, 变量 `a` 占用从地址 `0x2F05` 开始的 4 字节存储单元, 指针 `pa` 则占用从地址 `0x2F09` 开始的 2 字节存储单元; `p1` 和 `p2` 中存放的是变量 `c` 的地址, `pa` 中则存放变量 `a` 的地址。

(2) 先定义指针, 然后用赋值语句给指针变量赋值。例如:

```
int a,*pa;           /*定义一个整型变量 a 和一个指向整型量的指针变量 pa */
float b[3],*pb;      /*定义一个实型数组 b 和一个指向实型量的指针变量 pb */
pa=&a;                /*把变量 a 的地址赋给指针变量 pa, 使 pa 指向变量 a */
pb=b;                /*把数组首地址赋值给指针变量 pb, 使 pb 指向数组 b*/
```

(3) 把一个绝对地址值直接赋给一个指针变量。这种方法容易引起错误, 较少使用。

当指针指向变量后, 就可以用指向运算符 “*” 来表示该指针指向的变量。例如, 在上例中, `*pa=5;` 等价于 `a=5;`, 而 `printf("%d",a);` 也可以写成 `printf("%d",*pa);`。

2. 空指针和 void 型指针

如果指针未通过初始化或赋值的方式指向某个对象, 则指针的值是未定的, 这时如果用该指针去访问数据, 就会导致各种错误。例如:

```
int * pi;
*pi=58;           /* pi 指向的地址不定, 这种赋值很危险! */
```

一般应当给未定指针赋 `NULL` 值 (`NULL` 是 C 编译系统定义的宏, 它代表零字符), 以表明它未指向任何地方, 防止产生误解:

```
int *pi=NULL;
```

这时称该指针为空指针。由于 `NULL` 的代码值是 0, 所以也可以用 0 或 `\0` 代替 `NULL`。`\0`、0 或 `NULL` 既不表示指针取零字符, 也不表示指针指向地址为 0 的存储单元, 而是表示并未指向任何数据。如果有赋值: `*p=58;` 则程序执行时会显示 “Null pointer assignment”。

另外, ANSI C 标准允许定义 `void` 型指针, 例如:

```
void *p;
```

这是一种具有独特性质的指针, 仅表示指向内存的某个地址, 而它所指向的对象的数据类型并未指定, 因此, `void` 型指针又称无指定类型指针。对 `void` 型指针, 同样不能直接用来访问数据, 但可以通过强制类型转换将它转换成某种具体数据类型的指针 (这将在后面介绍)。

`NULL` 是一个指针值, 任何类型的指针都可赋予该值, 表示未指向任何数据; 而 `void *` 是一种类型, 表示空类型指针, 它不指向任何类型。空类型指针不能进行指针运算, 也不能进行间接引用。将其他指针的值赋给空类型指针是合法的, 将空类型指针的值赋给其他指针则不允许, 除非进行显式转换。



3. 使用指针应注意的问题

(1) 指针所指向的对象一定要在相应的指针之前定义。例如：

```
char *p=&c;  
char c;
```

是错误的。因为编译系统根据变量定义的先后次序安排存储单元，所以要先定义变量，然后才能定义指向该变量的指针，再通过赋值把指针指向变量。

(2) 指针变量必须存放地址量。例如：

```
int a,*p;  
p=a;
```

是错误的，虽然编译时不会出现错误提示，但是程序运行可能会严重错误。

(3) 未指向对象的指针不能引用。

(4) 不能将 `static` 型的指针初始化为指向 `auto` 或 `register` 型的变量。例如在函数内部有如下定义：

```
int local;  
static int *p=&local;
```

编译时就给出错误提示，原因是 `auto` 型变量在每次程序控制进入到该变量所在的程序段时才被分配空间，而静态指针的初始化在编译时进行。但是用赋值语句将自动型变量的地址赋给静态指针是可以的。

8.2.3 指针的运算

(1) 指针运算都是以数据类型为单位展开的。指针可以与整数相加减，指针变量加（减）一个整数是指将该指针变量的原值（地址）和它指向的变量所占用的内存单元字节数相加（减）。例如有以下定义：

```
int a[3], *pi=a;  
float f[3], *pf=f;
```

则 `pi+2` 的值在原 `pi` 值基础上增加了 $2*2$ ，`pf+2` 的值在原 `pf` 值的基础上增加了 $4*2$ ，分别表示从原地址增加 2 个数据单元后的地址：即 `pi+2` 表示数组元素 `a[2]` 的地址，`pf+2` 表示数组元素 `f[2]` 的地址。

下面程序段：

```
{  
    int a[3],*pi=a;  
    float f[3],*pf=f;  
    printf("%u\t%u\n",pi,pi+2);  
    printf("%u\t%u\n",pf,pf+2);  
}
```

运行时输出用无符号十进制格式表示的指针值，为：

```
65480    65484
```

65486 65494

经常对指针进行++或--的计算，表示指针移动，即指向相邻的数组元素。

(2) 如果两个指针指向同一数组的元素，则可以进行比较，结果是指向前面元素的指针变量小于指向后面元素的指针变量。如果两指针指向不同数组，进行比较则无意义。

(3) 两指针量也可以相减。如果两个指针变量指向同一个数组，则两指针值之差表示两个指针之间的元素个数。但两指针量的加法运算无意义。

指针运算的操作必须是有意义的，例如：

```
int a[5];
int *pi=&a[1];/*使pi 指向 a[1]*/
pi--;          /*指向 a[0]*/
*pi=3;         /*给数组 a[0]赋值，正常*/
pi--;          /*指向 a[-1],数组下标越界，危险*/
*pi=6;         /*可能出错,并且编译过程检查不出*/
```

8.3 通过指针引用变量、数组、字符串

8.3.1 用指针访问变量

我们可以用指针来访问它所指向的存储单元。如果定义了指向变量的指针，则对该变量的地址和变量的值可以不用变量名，而直接使用指针访问。例如，定义 `int a,*p=&a;`后，程序中凡能正确使用 `a` 的地方都可以用 `*p` 或 `p[0]`来代替(当然对简单变量而言，很少用 `p[0]`的形式，主要使用 `*p` 的形式)。见表 8-1，其中第一行为变量的使用形式，后两行是指针的使用形式。

表 8-1 用指针访问变量的各种形式

A	a=10	A++	x=a+b	a+=5	&a
*p	*p=10	*p++	x=*p+b	*p+=5	&*p
p[0]	p[0]=10	P[0]++	x=p[0]+b	p[0]+=5	&p[0]

8.3.2 用指针访问数组

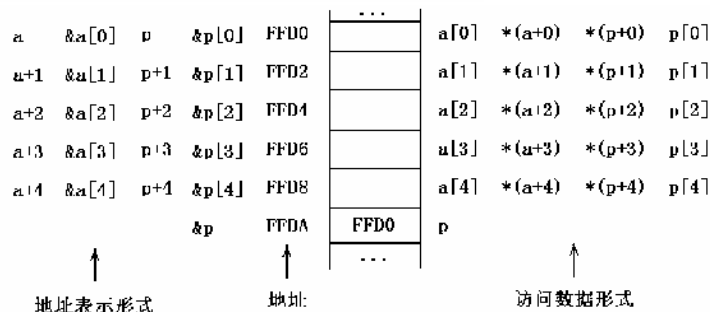
1. 指针与一维数组的关系

由于数组在内存中占用一片连续的存储单元，如果定义一个指向数组的指针，将该指针指向数组的第一个元素，通过改变指针的值，则可以存取数组的每一个元素。

假设有以下定义语句：

```
int a[5],*p=a;
```

指针 `p` 与 `a` 数组的各元素之间存在如图 8-4 所示的对应关系(设数组 `a` 的首地址为 `0xFFD0`)。



a+i 等价于 &a[i] 等价于 p+i 等价于 &p[i]

a[i] 等价于 *(a+i) 等价于 *(p+i) 等价于 p[i]

图 8-4 指针与一维数组的关系

2. 用指针访问一维数组

根据图 8-4，我们可以方便地用指针来访问一维数组的各个元素。

例 8.3 下面程序用下标、数组名和指针来实现一维数组的输入输出，从中可以看出它们之间的对应或等价关系。

```
#include <stdio.h>
main()
{
    int a[10]={1,2,3,4,5,6,7,8,9,10},i,*p;
    for(i=0;i<10;i++)
        printf("%d,",a[i]); /*用数组元素访问*/
    printf("\n");
    for(i=0;i<10;i++)
        printf("%d,",*(a+i)); /*用数组名访问*/
    printf("\n");
    for(p=a;p<a+10;p++)
        printf("%d,",*p); /*用指针访问(*运算)*/
    printf("\n");
    for(i=0,p=a;i<10;i++) /*用指针访问([]运算)*/
        printf("%d,",p[i]);
    printf("\n");
}
```

程序运行结果输出 4 行 1,2,3,4,5,6,7,8,9,10，可见 4 种方法的输出完全相同。

8.3.3 用指针访问字符串

1. 指针和字符串的关系

C 语言本身未提供字符串变量。前面已经学习了用字符型数组来存储和处理字符串，而用字符型指针处理字符串将更为方便和灵活。

如果我们定义：

```
char s[80],*st=s;
```

那么, 指针 `st` 是字符型指针, 就可以用上述方法通过指针处理字符型数组 `s`。在字符串中应用指针还有如下特点。

(1) 可以用字符型指针对字符串进行整体赋值。

我们已经知道, 除了在初始化中可对字符型数组名直接赋初值外, 不能在程序中直接对字符型数组名赋值, 也不能进行字符串的整体赋值。而字符型指针就好像是字符串变量一样, 既可以初始化也可以进行字符串的整体赋值。例如:

```
char *t="c language";
```

或:

```
char s[20],*t=s;  
t="c language";
```

都是允许的。

(2) 字符型指针还可以脱离字符数组独立使用以便处理字符串, 例如:

```
char *t;  
t="c language";
```

也是允许的。在这种情况下, 编译系统会安排一片连续的存储单元存放给定的字符串, 而把该字符串的首地址赋值给指针变量。

2. 用指针处理字符串

先举一个例子说明如何用指针来处理字符串。

例 8.4 把字符串 `t` 复制到字符串 `s` 中。库函数中的 `strcpy()` 可以实现字符串复制功能, 也可以自己编制这样的函数。为了体现对比效果, 用两种方法实现: 数组和指针。

(1) 用数组法。

```
#include<stdio.h>  
main()  
{  
    char s[11],t[]="c language";  
    int i=0;  
    while((s[i]=t[i])!='\0')  
        i++;  
    printf("\n%s",s);  
    printf("\n%s",t);  
}
```

(2) 用指针法。

```
#include<stdio.h>  
main()  
{  
    char *t="c language",*s,*s1=s,*t1=t; /*把字符串的首地址保存到 s1 和 t1*/  
    while((*s=*t)!='\0')
```



```
{
    s++;
    t++;
}
printf("\n%s",s1);      /*输出字符串，字符指针 s1 指向字符串的首地址*/
printf("\n%s",t1);
}
```

在 (2) 中，没有出现数组，只定义了两个字符型指针。对指针的操作就是对字符串的操作，因此两种方案的功能完全相同，程序运行结果为：

```
c language
c language
```

(3) 利用指针的特点，还可以将程序进一步简化如下。

```
#include <stdio.h>
main()
{
    char *t="c language",*s,*s1=s,*t1=t;
    while((*s++=*t++)!='\0');    /*或 while(*s++=*t++);*/
    printf("\n%s",s1);
    printf("\n%s",t1);
}
```

在 (3) 中，把移动 t 和 s 的操作移到了测试部分进行，因此，循环体变成了空语句。*t++ 和 *s++ 的功能是先取 t 所指处的字符并存入 s 所指的位置，然后将 t 和 s 各移向下一个字符。如果已复制到字符 '\0'，赋值表达式 (*s++=*t++) 的值就得 0，循环也就随之终止（严格来说，程序中指针 s 的用法有风险，程序举例暂时先这样用，以后利用动态内存分配可避免风险）。

由于字符串通常是顺序存取的，所以用指针处理字符串比用指针处理数值数组更能体现指针的优越性。这些优越性表现在以下几个方面：

(1) 使用指针可使程序变短，执行速度变快，所用的算法变得更简洁。

例 8.5 计算键盘输入的字符串长度。

```
#include <stdio.h>
main()
{
    int n=0;
    char *s;
    printf("please input a string\n");
    gets(s);
    for(n=0;*(s++);n++);
    printf("The length of string is %d",n);
}
```

程序中用一条不带循环体的 for 语句就能计算字符串的长度，*(s++) 表示循环结束的条件：如果 *(s++) 是字符串结束符，则循环结束，否则 n 的值加 1。循环结束时，n 的值就是字符串的长度，该长度表示字符串中字符个数，不包括串结束符。

(2) 用指针可以解决数组很难解决的问题。

例 8.6 进行两个字符串的交换。

```
#include <stdio.h>
main()
{
    char *str1="first string",*str2="second string";
    char *t;
    t=str1; str1=str2; str2=t;
    printf("str1=%s\tstr2=%s\n",str1,str2);
}
```

如果定义字符数组：

```
char str1[]="first string",str2[]="second string";
char *t;
```

就不能进行如下的操作：

```
t=str1;
str1=str2;
str2=t;
```

用字符数组进行字符串交换则需要更为复杂的方法。

(3) 字符指针是变量，它可以改变指向而指向别的变量，所以可对字符指针施加诸如++或--之类的赋值操作。而数组名代表数组的首地址，是常量，对数组名就不能施加这样的赋值操作。因此，用指针可以很容易得到子字符串。请看下面的例子。

例 8.7 求字符串的子串。

```
#include <stdio.h>
main()
{
    char *a="I Love China";
    a=a+7;          /*如果是数组名则不允许这样重新赋值*/
    printf("%c\n",*a); /*输出 a 指向的字符*/
    printf("%s\n",a);  /*输出从 a 指向位置开始的字符串*/
}
```

程序运行结果为：

```
C
China
```

当用“%c”控制输出时，输出的是 a 所指的一个字符；当用“%s”控制输出时，输出的是从 a 所指的存储单元开始的各个字符，直到遇到‘\0’为止，这就得到了一个子串。

8.4 指针数组和二级指针

8.4.1 指针数组的概念

如果一个数组，其每一个元素都是指针，这个数组就称为指针数组。指针数组的定义



格式为：

[存储类别] 数据类型 *指针数组名[常量表达式]

其中，方括号中的“常量表达式”的值应该是一个正整数，表示指针数组元素的个数。指针数组经常用来处理二维数组。

二维数组在逻辑上是二维空间，但是在存储器中则是以先行后列的顺序占用一片连续的内存单元，其存储结构是一维线性空间。

设有数组定义 `int a[2][3]`；其在内存中的存放顺序如下图示：

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
看作 a[0]			看作 a[1]		

据此，就可以把二维数组各元素视为一维数组元素来处理。还可以把二维数组看作特殊的一维数组，该数组的元素代表二维数组的每一行。即可以把 `a` 数组看作一维数组有两个元素 `a[0]` 和 `a[1]`，而 `a[0]` 和 `a[1]` 元素又是一维数组，其中 `a[0]` 有 `a[0][0]`、`a[0][1]`、`a[0][2]` 三个元素，`a[1]` 有 `a[1][0]`、`a[1][1]`、`a[1][2]` 三个元素。`a[0]`、`a[1]` 就表示每一行在内存中的开始地址，可以用指针数组来指向每一行。当然也可以用指针数组指向多个字符串。

例如：

```
int a[2][3];
int *p[2]={a[0],a[1]};
char * nation[]={ "China", "America", "France" };;
```

在指针数组的定义中，同样也出现了 `*` 和 `[]`，由于 `[]` 的优先级比 `*` 高，使 `p` 先与 `[]` 结合，说明它是一个数组，再与 `*` 结合，说明它是一个指针。`int *p[2]` 表示 `p` 是指针数组，有 2 个元素，每个元素是指针。

和普通数组一样，编译系统要为指针数组分配一片连续的存储单元来存放它的各个元素，数组名就是该数组的首地址，是一个常量。指针数组的初始化也与普通数组相同。

可以用指针数组来处理二维数组：先使指针数组的各元素分别保存二维数组各行的首地址，然后用指针数组的各元素来对它所指向的行进行操作。

指针数组的数组元素是一个指针，每个数组元素（指针）所指向的内容长度可以是不相同的，如上例中每个指针指向的字符串是不规则的。使用指针数组的显著优点是可以方便地处理多个字符串。

例 8.8 用指针数组处理多个长度不等的字符串。

```
#include <stdio.h>
main()
{
    static char *lq[]={"sun","mon","tues","wednes","thurs","fri","sat"};
    int n;
    while(1)
    {
        printf("请输入星期(0,1,2,3,4,5,6:");
        scanf("%d",&n);
```

```
        if(n<0||n>6)
            break;
        printf("weekday is %s\n",lq[n]);
    }
}
```

程序中用一个字符型的指针数组存放每周星期几的英文名称,它们的代号依次是0~6;根据用户输入的代号,程序即输出该代号所对应的英文名称。若输入的代号在0~6之外,程序自行终止。

例 8.9 用选择法对给定的 3 个字符串按从小到大的顺序排序后输出。我们用指针数组来编制程序,其中用到 `strcmp()` 函数和字符串交换方法。

```
#include <stdio.h>
#include <string.h>
main()
{
    static char * nation[]={"France","China","America"};
    char *temp;
    int i,j,k;
    for(i=0;i<3;i++)
    {
        k=i;
        for(j=i+1;j<3;j++)
            if(strcmp(nation[k],nation[j])>0)
            {
                k=j;
                temp=nation[k];
                nation[k]=nation[i];
                nation[i]=temp;
            } /*交换,使小的字符串前移*/
    }
    for(i=0;i<3;i++) /*输出排序后的字符串*/
        printf("%s\n",nation[i]);
}
```

程序中定义的指针数组 `nation[]`,其每一个元素指向一个国家名字符串。

例 8.10 用指针数组处理二维数组。定义一个二维数组 `num` 用来存放学号和成绩,定义一个指针数组 `pb` 并使其元素值指向二维数组的每一行。该程序要求用户输入 `N` 名同学的学号和成绩,按学号排序后输出学号和成绩。程序如下:

```
#include <stdio.h>
#define N 3
main()
{
    int num[N][2];
    int j,k,t;
    int *pb[N];
    pb[0]=num[0];
```



```
pb[1]=num[1];
pb[2]=num[2];
/* input */
printf("please input %d numbers and scores",N);
for(j=0;j<N;j++)/* input the data*/
    scanf("%d %d",(pb[j]),(pb[j]+1));
/* sort */
for(j=0;j<N-1;j++)
    for(k=j+1;k<N;k++)
        if(*pb[j]>*pb[k])
        {
            t=*pb[j];
            *pb[j]=*pb[k];
            *pb[k]=t;
            t=*(pb[j]+1);
            *(pb[j]+1)=*(pb[k]+1);
            *(pb[k]+1)=t;
        }
/* output */
for(j=0;j<N;j++)
    printf("%d %d\n",*(pb[j]),*(pb[j]+1));
}
```

8.4.2 二级指针（指向指针的指针）

前面学习的指针变量或指针数组元素直接指向数据对象（称之为一级指针），其内存单元中存放的是数据的地址。而指针变量自己也有地址，用来存放指针变量地址的指针称之为二级指针。所谓二级指针就是指向指针的指针，是一种间接指向数据目标的指针，如图 8-5 所示。

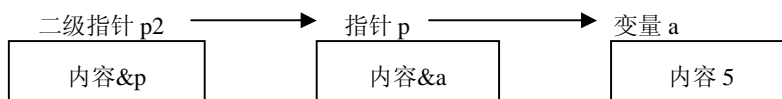


图 8-5 一级指针和二级指针

二级指针的定义格式：

[存储类别] 数据类型 **指针名

其中，指针名前面有两个*，表示是一个二级指针。二级指针的赋值和初始化与一级指针方法类似。例如：

```
int a=5,*p=&a,**p2=&p; /*定义 p 为指针并且使指向 a, p2 为二级指针使指向 p*/
printf("%d,%d,%d",a,*p,**p2); /*访问变量 a 中数据有三种等价形式*/
```

8.4.3 用二级指针访问数组或字符串

用二级指针可以处理二维数组。比如，我们作如下定义：

```
int a[3][4]={ {1,2,3,4},{5,6,7,8},{9,10,11,12}}; /*定义二维数组*/
```

```
int *p[3],**p2; /*定义指针数组和二级指针*/
p[0]=a[0];p[1]=a[1];p[2]=a[2]; /*使指针数组 p 的元素指向二维数组的每一行*/
p2=p; /* 使二级指针 p2 指向指针数组 p */
```

则有： $*p2==p[0]==*p$ ， $*(p2+1)==p[1]==*(p+1)$ ， $*(p2+2)==p[2]==*(p+2)$ 。可见，二级指针和指针数组是等价的。

进一步还有： $\&a[i][j]=p[i]+j=p2[i]+j$ ， $**p2=a[0][0]$ ， $**(p2+1)=a[1][0]$ ， $**(p2+2)=a[2][0]$ ， $*(*(p2+i)+j)=a[i][j]$ ($0 \leq i < 3, 0 \leq j < 4$)。

图 8-6 表示了二维数组、指针数组和二级指针之间的相互关系。图中的每一格表示一个数据单元。二维数组用二维表格的形式表示，各元素的地址实际上是连续的。

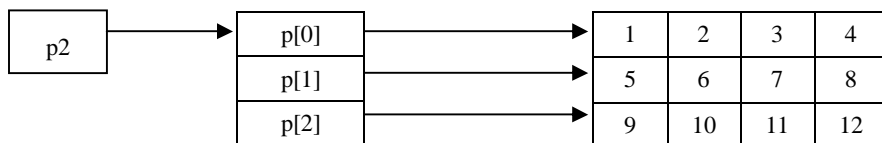


图 8-6 二维数组、指针数组和二级指针之间的相互关系

还可以用二级指针处理多个字符串，例如：

```
char * pc[]={ "good", "better", "best" }; /*定义指针数组，每个元素指向一个字符串*/
char ** p2c; /*定义二级指针 p2c*/
p2c=pc; /*二级指针 p2c 指向指针数组 pc 的首地址*/
```

例 8.11 用二级指针和指针数组处理二维数组。

```
#include <stdio.h>
main()
{
    static int a[3][4]={ {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
    static int *p[3]={a[0],a[1],a[2]},**p2=p; /*初始化*/
    int i,j;
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
            printf("%5d",*(*(p2+i)+j)); /*或者 printf("%5d",p[i][j]);*/
        printf("\n");
    }
}
```

例 8.12 用二级指针输出多个字符串。

```
#include <stdio.h>
main()
{
    static char *nation[]={ "China", "America", "France", "" };
    char **p=nation; /*定义二级指针并使之指向指针数组*/
    while(**p!='\0') /*如果存储单元非空，输出字符串*/
        printf("%s\n", *p++);
}
```




8.5 将指针作为函数参数

我们前面已经学习过：函数间通过参数的虚实结合传递参数是一种按值传递方式，在被调函数中形参的改变不会影响主调函数中的实参，这种方式较好地保持了程序的独立性。但是有时候我们需要通过调用函数来改变某些主调函数中实参的值，或有时在主调函数和被调函数之间需要传递数组，此时值传递方式就很难实现。

如果用指针型变量作形参，用地址做实参，则可以使被调函数中形参的值为地址，即被调函数中的形参可以指向主调函数中的参数，即可以通过在被调函数中处理指针指向的内容来处理主调函数的参数，也可以在被调函数中处理主调函数中的数组或字符串。

例 8.13 用简单变量做形参和用指针变量做形参的对照。下面程序实现利用函数 swap1 进行形参变量 x、y 的交换，swap2 进行两个实参变量的值互相交换。其中 swap1 用简单变量做形参，调用时变量 a、b 做实参，是传值方式的调用，被调函数中 x、y 变量的值交换，但主调函数的实参 a、b 的值没有交换。swap2 用指针变量做形参，调用时用变量 a、b 的地址做实参，虚实结合传递的是地址值，则在被调函数中可以通过指针方法对主调函数中的变量进行交换。

```
#include "stdio.h"
void swap1(int x,int y)
{
    int temp;
    temp=x;
    x=y;
    y=temp;
    printf("x=%d,y=%d\n",x,y);
}
void swap2(int *px,int *py)
{
    int temp;
    temp=*px;
    *px=*py;
    *py=temp;
    printf("*px=%d,*py=%d\n",*px,*py);
}
main()
{
    int a=5,b=8;
    swap1(a,b);
    printf("a=%d b=%d\n",a,b);
    swap2(&a,&b);
    printf("a=%d b=%d\n",a,b);
}
```



注意

*出现在形参说明中表示 px、py 是指针变量，而出现在函数体中表示访问指针指向的变量。

如果用函数来处理数组,也可以把形参写成数组的形式,此时实参要用数组的首地址。但实质上,函数中仍然是把形参作为指针来处理,可以对该指针进行递增和递减操作。

例 8.14 利用指针作形参在函数间传递数组,用被调函数实现对数组元素求和。

```
#include <stdio.h>
int sum(int x[ ],int n)          /* 数组形式表示的形参相当于指针变量 */
{
    int i,sum=0;
    for(i=0;i<n;i++,x++)
        sum+=*x;
    return sum;
}
main()
{
    int a[]={1,2,3,4,5,6,7};
    int b[]={2,4,6,8,10,12};
    printf("suma=%d\n",sum(a,7));    /*调用函数分别计算 a 数组和 b 数组的和*/
    printf("sumb=%d\n",sum(b,6));    /*实参是数组名,表示数组的首地址*/
}
```

例 8.15 利用指针变量作形式参数,函数调用时传递数组。在函数中对两个一维数组的处理,输入多个学生学号、成绩并排序后输出。

```
#include<stdio.h>
#define N 3
main()
{
    int num[N],score[N];
    void input(int *pa,int *pb,int n);
    void sort(int *pa,int *pb,int n);
    void output(int *pa,int *pb,int n);
    input(num,score,N);
    sort(num,score,N);
    output(num,score,N);
}
void input(int *pa,int *pb,int n)
{
    int j;
    printf("please input %d numbers and scores:\n",N);
    for(j=0;j<n;j++)
    {
        scanf("%d%d",pa,pb);
        pa++;
        pb++;
    }
}
void sort(int *pa,int *pb,int n)
{
    int t;
    int j,k;
```



```
for(j=0;j<n-1;j++)
    for(k=j+1;k<n;k++)
        if(pa[j]>pa[k])
        {
            t=pa[j];
            pa[j]=pa[k];
            pa[k]=t;
            t=pb[j];
            pb[j]=pb[k];
            pb[k]=t;
        }
}
void output(int *pa,int *pb,int n)
{
    int j;
    for(j=0;j<n;j++)
        printf("%d %d\n",pa[j],pb[j]);
}
```

8.6 返回指针值的指针型函数

我们已经学习过，所谓函数类型是指函数返回值的类型。函数的返回值可以是一般数据，也可以是地址量。如果函数的返回值是地址量，则函数的类型是指针型。

指针型函数定义的一般格式如下：

```
[存储类别][数据类型标识符] *函数名([形参表])
{
    内部变量定义语句;
    执行语句;
}
```

其中“函数名”之前加了“*”号表明这是一个指针型函数，即返回值是一个指针。“数据类型说明符”表示了返回的指针值所指向的数据类型。如：

```
int *ap(int x,int y)
{
    /*函数体*/
}
```

表示 `ap` 是一个返回指针值的指针型函数，它返回的指针指向一个整型变量。

例 8.16 指针型函数应用举例。利用指针函数，输入一个 1~7 之间的整数，输出对应的星期名。程序如下：

```
#include <stdio.h>
main()
{
    int i;
    char *day_name(int n);    /*指针型函数在引用之前先定义或声明*/
}
```

```
printf("input Day No: ");
scanf("%d",&i);
printf("Day No:%2d-->%s\n",i,day_name(i));
}
char *day_name(int n)      /*指针型函数的定义*/
{
    static char *name[]={ "Illegal day", "Monday",
                           "Tuesday", "Wednesday",
                           "Thursday", "Friday",
                           "Saturday", "Sunday" };
    return((n<1||n>7)? name[0] : name[n]);
    /*返回 name[0]或 name[n]表示的字串首地址*/
}
```

本例中定义一个指针型函数 `day_name`，它的返回值指向一个字符串。该函数中定义了一个静态指针数组 `name`，`name` 数组初始化赋值为 8 个字符串，分别表示各个英文星期名及出错提示。形参 `n` 表示与星期名所对应的整数。在主函数中，把输入的整数 `i` 作为实参，在 `printf` 语句中调用 `day_name` 函数并把 `i` 值传送给形参 `n`。`day_name` 函数中的 `return` 语句包含一个条件表达式，`n` 值若大于 7 或小于 1，则把 `name[0]` 指针返回主函数，输出出错提示字符串 “Illegal day”；否则将 `name[n]` 指针返回主函数，主函数中输出对应的星期名字符串。

指针型函数不能把在它内部定义的局部变量的地址作为返回值，只能返回全局或静态变量的地址。调用指针型函数时，接收返回值的变量一定是与被调用函数数据类型一致的指针，不得使用数组名接收指针型函数的返回值，因为数组名是地址常量。

8.7 内存动态分配

8.7.1 内存动态分配的含义

Turbo C 主要用两种方法使用内存。第一种是用全局变量和局部变量方式。程序的代码部分占用代码区；静态变量和全局变量占用静态存储区（也称全局变量区），由编译系统分配并在整个程序运行期间保持占用；局部变量（也称自动变量）存储在运行栈，动态地分配和释放所占内存，即当程序执行时才为它们分配存储空间，一旦它们所在的程序段（函数或复合语句）执行结束，会自动被销毁或释放。第二种是用内存动态分配方式，即使用动态存储区。这个存储区在用户的程序之外，不是由系统分配的，而是由用户在程序中通过动态分配获取的。使用动态内存分配至少有如下 3 个优点。

- 可以更有效地使用内存。例如，考虑到程序的通用性，可能需要建立 1 000 个字符串，每个字符串要容纳 30 个字符，总共需要 30 000 字节的存储空间。如果程序某次运行时，只使用了 30 个字符串，那么，就有 29 100 字节的存储空间被占而不用，造成浪费。使用动态分配，就可以减少这种浪费。
- 同一段内存可作为不同的用途。因为动态内存存在使用时申请，用完就释放，这样，这一段内存就可以被再次申请。



- 允许建立链表等动态数据结构（有关建立线性链表的内容在后面介绍）。

如前所述，当使用动态内存存放数组时，不是在程序中定义数组的大小，而是在程序运行时才确定它的大小，这可以让程序最高效地使用内存资源。

使用动态内存分配，必须遵循以下几步：

- (1) 要确切地规定需要多少内存空间，以避免存储空间的浪费，也可多为其他数据留有空间。
- (2) 利用 C 编译系统提供的动态内存分配函数来分配所需要的存储空间。
- (3) 使指针指向获得的内存空间，以便在该空间内实施运算或操作。
- (4) 当用完之后，一定要释放这一空间。如果不释放获得的存储空间，则可能把动态存储区的内存耗尽。只有正确使用内存动态分配，才能提高存储效率。

8.7.2 内存动态分配函数

使用内存动态分配，通常要用到 4 个函数：`calloc()`和 `malloc()`用于动态申请内存空间；`realloc()`用于重新改变已分配的动态内存的大小；`free()`用于释放不再使用的动态内存。ANSI 建议将这 4 个函数都定义在标题文件 `stdlib.h` 中。因此，如果在 Turbo C 环境下使用动态内存，一定要包含 `stdlib.h` 文件。也有的 C 环境将这几个文件放在 `malloc.h` 中。

1. 动态分配函数 `calloc()`和 `malloc()`

`calloc()`和 `malloc()`都用于内存的动态分配，前者是按照指定的数据对象分配，后者则是按照指定的字节分配。

(1) `calloc()`的调用形式为：

```
void *calloc(n,size)
```

其中，`n` 和 `size` 都是整数。函数的功能是动态分配 `n` 个长度为 `size` 字节的连续空间。调用该函数的结果：若分配成功，则函数返回一个指向分配内存区起始地址的指针。由于该地址内的数据类型无法确定，故为一个 `void` 型指针；若没有足够的内存满足要求，则返回空指针 `NULL`。因此在使用内存之前，验证该函数的返回值是不是空指针（`NULL` 或 0）非常重要。

为了给指定类型的数据动态分配内存，一般常用 `sizeof` 来确定该类型数据所占字节数。例如，下面的调用：

```
float *p;
p=(float *)calloc(500,sizeof(float));    /*将函数返回值强制转为实型指针*/
if(p==NULL)
    exit(1);
```

如果成功，可得到供 500 个实型数据使用的动态内存的起始地址，并把这个起始地址赋给指针变量 `p`，利用该指针就能对该区域里的数据进行操作或运算。如果不成功，退出程序。

(2) `malloc()`的调用形式为：

```
void *malloc(size)
```

其中, size 是整数。它的功能是动态分配 size 个字节的连续空间。例如:

```
char *p1,*q1;
if((p1=(char *)malloc(80))==NULL)
    exit(1);
if((q1=(char *)malloc(20))==NULL)
    exit(1);
```

如果成功,就分别得到 80 和 20 字节的动态内存,并由 p1 和 q1 分别指向。

2. 动态重分配函数 realloc()

realloc()的调用形式为:

```
void *realloc(p,size)
```

其中, p 是指向动态内存起始地址的指针。函数功能是对 p 所指向的已动态分配的内存区重新进行分配,新分配的内存区大小为 size 字节。调用该函数的结果是返回新分配内存区的起始地址。使用这个函数时,允许新内存区大于或小于老内存区。

例 8.17 对动态内存进行再分配。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
    char *p;
    if((p=(char *)malloc(19))==NULL)
        exit(1);
    strcpy(p,"This is the string");
    p=(char *)realloc(p,20);
    if(p==NULL)
        exit(1);
    strcat(p,".");
    puts(p);
}
```

程序中先用 malloc()申请 19 字节的内存存放 18 个字符的字符串,又用 realloc 重新申请 20 字节的内存(即在原申请内存的基础上增加了 1 字节),以便在字符串的末尾增加一个"."字符。

在 realloc(p,size)中,若 p=NULL,它就相当于 malloc(size)的功能;若 size=0,则相当于下面要介绍的 free()的功能。

3. 释放动态内存函数 free()

free()的调用形式为:

```
void free(p)
```

其中, p 是指向待释放内存空间首字节的指针。函数的功能是释放不再使用的动态内



存。该函数没有返回值，故定义为 `void` 型。如在上例程序的末尾增加一条语句：

```
free(p);
```

表示将申请的动态内存释放。

8.8 函数指针

在 C 语言中，一个函数总是占用一段连续的内存区，而函数名代表该函数所占内存区的首地址，即函数执行时的入口地址。当把函数名赋予一个指针变量时，该指针变量中的内容就是函数的入口地址，该指针是指向这个函数的指针，简称函数指针，通过指针变量就可以找到并调用函数。一般把用来存放函数入口地址的指针定义为函数型指针。函数型指针的定义形式如下：

```
[存储类别][数据类型标识符](*函数指针变量名)();
```

例：`static int(*pf)();`，其中“数据类型标识符”`int` 表示被指函数的返回值的类型是整型。“`(*pf)`”表示“*”后面的变量 `pf` 是定义的指向函数入口的指针变量，最后的空括号表示指针变量所指的是一个函数。用函数指针调用函数的一般形式为：

```
(*函数指针变量名)(实参表)
```

用函数指针调用函数的步骤如下：

- (1) 先定义函数型指针变量；
- (2) 把被调函数的入口地址（函数名）赋予该函数指针；
- (3) 用函数指针变量形式调用函数。

例 8.18 函数指针应用举例。用函数指针形式调用两函数，分别得到两数中的大数和小数。

```
#include <stdio.h>
int max(int a,int b)
{
    return(a>b)? a:b;
}
int min(int a,int b)
{
    return(a<b)?a:b;
}
main()
{
    int max(int a,int b); /*函数声明*/
    int min(int a,int b); /*函数声明*/
    int x,y,z;
    int(*pf)();           /*定义函数指针变量*/
    printf("input two numbers:\n");
    scanf("%d%d",&x,&y);
    pf=max;               /* max 函数的入口地址（函数名）赋予该函数指针*/
```

```
printf("max=%d\n",(*pf)(x,y)); /*用函数指针变量形式调用函数*/  
pf=min; /*min 函数的入口地址（函数名）赋予该函数指针*/  
printf("min=%d\n",(*pf)(x,y)); /*用函数指针变量形式调用函数*/  
}
```

说明：

(1) 函数指针变量不能进行算术运算，这是与数组指针变量不同的。数组指针变量加减一个整数可使指针移动指向后面或前面的数组元素，而函数指针的移动是毫无意义的。

(2) 函数调用中“(*函数指针变量名)”的两边的括号不可少，其中的*不应该理解为求值运算，应理解为指向函数的入口地址。

(3) 注意函数型指针变量和指针型函数这两者在写法和意义上的区别。如 `int(*p)()` 和 `int *p()` 是两个完全不同的量。`int(*p)()` 说明 `p` 是一个指向函数入口的指针变量，被指函数的返回值是整型量；`int *p()` 则说明 `p` 是一个指针型函数，其返回值是一个指向整型量的指针，`*p` 两边没有括号。作为指针型函数定义或声明，在括号内还要写入形式参数；定义时，`int *p()` 只是函数头部分，一般还应该写有函数体部分。

8.9 main()函数的命令行参数

除了在递归程序中可能出现 `main` 函数自己调用自己的情况外，`main` 函数很少被其他函数调用。所以，我们前面用到的 `main` 函数一概写成：

```
main()
```

也可以写成：

```
void main(void)
```

实际上，`main` 函数也可以有参数和返回值，本节介绍 `main` 函数的参数及应用。

1. 命令行参数

我们知道，C 语言可用来设计系统程序，像 DOS 的 `COPY` 命令完全可以用 C 语言编制。为此，要用到命令行参数的概念。

一般地，我们把操作系统状态下为了执行某个程序或命令而键入的一行字符称为命令行。通常命令行含有可执行文件名，有的还带有若干参数，并以回车符结束。例如：

```
C:\>copy file1.txt file2.txt
```

就是一个命令行，它的功能是要求操作系统进行文件复制，根据 `file1.txt` 复制产生一个 `file2.txt`。在这个命令行中，DOS 命令及其参数共有 3 个字符串，它们相互用空格隔开，这 3 个字符串统称为命令行参数。

2. main()函数的参数

将命令行的参数传递给主函数的方法是通过主函数的两个形式参数实现的。采用指针数组或二级指针作为 `main()` 函数的形参，形式如下：



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

或:

```
main(int argc, char **argv)
```

其中, 整型变量 `argc` 用来记录命令行中参数的个数, 由 C 程序运行时自动计算出来; 字符型指针数组 `argv` 用来存放命令行中的各个参数(或者用二级指针 `argv` 指向命令行中每个参数), 即将每一个以空格为界的参数视为一个字符串, 存放其首地址, `argv` 的容量由 `argc` 确定。在 `main()` 中使用 `argc` 和 `argv` 这两个参数, 就可以把用户在命令行中键入的文件名及参数传递到程序的内部进行处理。

例 8.19 编程实现将命令行参数输出。

```
#include<stdio.h>
main(int argc, char *argv[])
{
    int i;
    printf("argc=%d\n", argc);
    for(i=0; i<argc; i++)
        printf("argv[%d]:%s\n", i, argv[i]);
}
```

假设该程序以 `echo1` 为文件名, 执行该程序时从命令行提示符状态输入 `echo1 one two three`, 则屏幕显示如下输出:

```
argc=4
argv[0]:echo1.exe
argv[1]:one
argv[2]:two
argv[3]:three
```

程序自动计算出 `argc=4` 并确定数组 `argv` 的元素个数为 4, `argv[0]` 用于存放程序名, 本例中为 `echo1.exe`, 后面参数依次传给 `argv[1]~argv[3]` 并在程序中进行输出。`argv` 也可以采用二级指针的形式, 读者可以用二级指针作参数改写本程序。



注意

`argc` 和 `argv` 这两个参数的名称属于准保留字, 专门用作 `main` 函数的参数。用户也可以自己定义参数名, 但类型不可更改。

要 点 回 顾

1. 存储器中每一个存储单元都有一个地址。当定义一个变量时, 系统会为该变量分配内存, 则内存空间的首地址称为该变量的地址, 内存空间的内容称为该变量的值。变量或数组元素的地址是由系统分配的, 用取地址运算符“&”可获取变量或数组元素的地址, 使用的格式为: `&变量名`或`&数组元素名`。

2. 指向运算符“*”的作用是将“*”施加在一个地址量上, 以得到其中的数据。应用格式为: `*地址量`。此外, `*&a` 与 `*(&a)` 等价。

3. 下标运算符[]也用来访问地址中的数据,应用格式为:地址量[整型表达式]。
4. 整型数组的一个元素占2字节,浮点型数组的一个元素占4字节;对一维数组元素d[i]来说,表示形式d[i]、*&d[i] 和*(d+i)等价。
5. 专门用来存放地址的变量,就是指针变量,使用前需要先定义。定义指针变量的一般形式为:

[存储类别] 数据类型 *指针变量名1,*指针变量名2,...

6. 指针只有指向某一对象时,才有实际意义。要使指针指向某一对象,可以采用如下方式:

- (1) 指针变量可以在定义时初始化。
- (2) 先定义指针,然后用赋值语句给指针变量赋值。

7. 如果指针未通过初始化或赋值的方式指向某个对象,则指针的值是未定的。这时如果用该指针去访问数据,就会导致各种错误。

ANSI C 标准允许定义 void 型指针,void 型指针仅表示指向内存的某个地址,而它所指向的对象的数据类型并未指定。

8. 使用指针时要注意:(1) 指针所指向的对象一定要在相应的指针之前定义;(2) 指针变量必须存放地址量;(3) 未指向对象的指针不能引用。

9. 指针量可以进行运算,指针运算都是以数据类型为单位展开的。指针可以与整数相加减(指针值的变化和元素的数据类型有关);如果两个指针指向同一数组的元素,则可以进行比较;两指针量也可以相减。

10. 如果定义了指向变量的指针,则对该变量的地址和变量的值,可直接使用指针去访问。

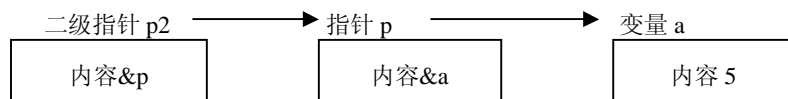
11. 可以方便地用指针来访问数组的各个元素。如果定义一个指针,并将该指针指向数组的某一个元素,则通过改变指针的值,可以存取数组的每一个元素。用字符型指针处理字符串更为方便和灵活。

12. 如果一个数组,其每一个元素都是指针,这个数组就称为指针数组。指针数组的定义格式为:

[存储类别] 数据类型 *指针数组名[常量表达式]

常用指针数组处理二维数组。

13. 指针变量自己也有地址:用来存放指针变量地址的指针称为二级指针。也就是说二级指针是指向指针的指针,是一种间接指向数据目标的指针,如图所示。



二级指针的定义格式:

[存储类别] 数据类型 **指针名

可以用二级指针访问数组或字符串。

14. 动态存储区在用户的程序之外,不是由系统分配的,而是由用户在程序中通过动



态分配获取的。有关内存动态分配的 4 个函数为：`calloc()`和 `malloc()`用于动态申请内存空间；`realloc()`用于重新改变已分配的动态内存的大小；`free()`用于释放不再使用的动态内存。

15. 如果用指针型变量作形参，用地址做实参，则可以使被调函数中的形参指向主调函数中的参数，从而通过在被调函数中处理指针指向的内容来处理主调函数的参数；用此方法也可以在被调函数中处理主调函数中的数组或字符串。

16. 如果函数的返回值是地址量，则函数的类型是指针型。指针型函数定义的一般格式如下：

```
[存储类别][数据类型标识符] *函数名([形参表])
{
    内部变量定义语句;
    执行语句;
}
```

17. 当把函数名赋予一个指针变量时，指针变量中的内容就是函数的入口地址，该指针是指向这个函数的指针，简称函数指针；通过指针变量就可以找到并调用这个函数。

函数指针的定义形式如下：

```
[存储类别][数据类型标识符](*函数指针变量名)();
```

18. 操作系统状态下，为了执行某个程序或命令而键入的一行字符称为命令行。通常命令行含有可执行文件名，有的还带有若干参数，并以回车符结束。为了将命令行的参数传递给程序的主函数，采用指针数组或二级指针作为 `main()`函数的形参，形式如下：

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

`argc` 用来记录命令行中参数的个数，由 C 程序运行时自动计算出来；`argv` 用来存放命令行中的各个参数。`argc` 和 `argv` 这两个参数的名称属于准保留字，专用作 `main` 函数的参数。用户也可以使用其他参数名，但类型不可更改。

习 题

1. 设 `p1` 和 `p2` 是指向同一个 `int` 型一维数组的指针变量，`k` 为 `int` 型变量，则不能正确执行的语句是 ()。

- a) `k=*p1+*p2;` b) `p2=k;` c) `p1=p2;` d) `k=*p1 *(*p2);`

2. 设有如下定义：

```
int arr[]={6,7,8,9,10};
int * ptr;
```

则下列程序段的输出结果为 ()。

```
ptr=arr;
*(ptr+2)+=2;
printf("%d,%d\n", *ptr, *(ptr+2));
```

A) 8,10

B) 6,8

C) 7,9

D) 6,10

3. 设有如下定义:

```
int(*ptr)();
```

则以下叙述中正确的是 ()。

- A) ptr 是指向一维数组的指针变量
 - B) ptr 是指向 int 型数据的指针变量
 - C) ptr 是指向函数的指针, 该函数返回一个 int 型数据
 - D) ptr 是一个函数名, 该函数的返回值是指向 int 型数据的指针
4. 若有以下定义和语句:

```
double r=99, *p=&r;
*p=r;
```

则以下正确的叙述是 ()。

- A) 两处的*p 含义相同, 都说明给指针变量 p 赋值
 - B) 在 “double r=99, *p=&r;” 中, 把 r 的地址赋值给了 p 所指的存储单元
 - C) 语句“*p=r;” 把变量 r 的值赋给指针变量 p
 - D) 语句“*p=r;” 取变量 r 的值放回 r 中
5. 用以下语句调用库函数 malloc, 使字符指针 st 指向具有 11 个字节的动态存储空间, 请填空。

```
st=(char*) _____;
```

6. 以下程序通过函数指针 p 调用函数 fun, 请在填空栏内, 写出定义变量 p 的语句。

```
void fun(int *x,int *y)
{
    ...
}
main()
{
    int a=10,b=20;
    _____; /定义变量 p */
    p=fun; p(&a,&b);
    ...
}
```

7. 设有定义: int n,*k=&n;以下语句将利用指针变量 k 读写变量 n 中的内容, 请将语句补充完整。

```
scanf("%d", _____);
printf("%d\n", _____);
```

8. 若有以下的定义和语句, 则在执行 for 语句后, *(*(pt+1)+2)表示的数组元素是

```
int t[3][3],*pt[3],k;
for(k=0;k<3;k++)
    pt[k]=&t[k][0];
```

- A) t[2][0] B) t[2][2] C) t[1][2] D) t[2][1]



9. 下面程序把数组元素中的最大值放入 a[0]中, 则在 if 语句中的条件表达式应该是 ()。

```
main()
{
    int a[10]={6,7,2,9,1,10,5,8,4,3}, *p=a, i;
    for(i=0; i<10; i++, p++)
        if(_____)
            *a=*p;
    printf("%d", *a);
}
```

- A) $p > a$ B) $*p > a[0]$ C) $*p > *a[0]$ D) $*p[0] > *a[0]$
10. 当调用函数时, 实参是一个数组名, 则向函数传送的是 ()。
- A) 数组的长度 B) 数组的首地址
- C) 数组每一个元素的地址 D) 数组每个元素中的值
11. 以下程序运行后的输出结果是什么?

```
main()
{
    char s[ ]="9876", *p;
    for( p=s ; p<s+2 ; p++)
        printf("%s\n", p);
}
```

12. 有以下程序:

```
#include <string.h>
main()
{
    char *p="abcde\0fghjik\0";
    printf("%d\n", strlen(p));
}
```

程序运行后的输出结果是_____。

13. 以下程序的输出结果是什么?

```
#include<stdio.h>
sub(int *a,int n,int k)
{
    if(k<=n)
        sub(a,n/2,2*k);
    *a+=k;
}
Main()
{
    int x=0;
    sub(&x,8,1);
    printf("%d\n",x);
}
```

14. 函数调用 `strcat(str1,str2),str3` 的功能是 ()。
- A) 将串 `str1` 复制到串 `str2` 中后再连接到串 `str3` 之后
 - B) 将串 `str1` 连接到串 `str2` 之后再复制到串 `str3` 之后
 - C) 将串 `str2` 复制到串 `str1` 中后再将串 `str3` 连接到串 `str1` 之后
 - D) 将串 `str2` 连接到串 `str1` 之后再将串 `str1` 复制到串 `str3` 中
15. 下列程序中字符串中各单词之间有一个空格, 则程序的输出结果是 ()。

```
#include
main()
{
    char str1[]="How do you do",*p1=str1;
    strcpy(str1+strlen(str1)/2,"es she");
    printf("%s \n", p1);
}
```

16. 函数 `sstrcmp()` 的功能是对两个字符串进行比较。当 `s` 所指字符串和 `t` 所指字符串相等时, 返回值为 0; 当 `s` 所指字符串大于 `t` 所指字符串时, 返回值大于 0; 当 `s` 所指字符串小于 `t` 所指字符串时, 返回值小于 0 (功能等同于库函数 `strcmp()`)。请填空。

```
#include <stdio.h>
int sstrcmp(char *s,char *t)
{
    while(*s&&*t&&*s==_____)
    {
        s++;t++;
    }
    return _____;
}
```

17. 若有以下定义和语句:

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

则不能表示 `a` 数组元素的表达式是 ()。

- A) `*p`
 - B) `a[10]`
 - C) `*a`
 - D) `a[p-a]`
18. 以下函数用来在 `w` 数组中插入 `x`, `w` 数组中的数已按由小到大顺序存放, `n` 指向的存储单元存放数组中数据的个数。插入后数组中的数仍有序。请填空。

```
void fun(char *w,char x,int *n)
{
    int i,p;
    p=0;
    w[*n]=x;
    while(x>w[p])
        _____;
    for(i=*n;i>p;i--)
        w[i]=_____i;
    w[p]=x;
```



```
++*n;  
}
```

19. 以下程序中 select 函数的功能是：在 n 行 m 列的二维数组中，选出一个最大值作为函数值返回，并通过形参传回此最大值所在的行下标。请填空。

```
#define N 3  
#define M 3  
select(int a[N][M],int *n)  
{  
    int i,j,row=1,column=1;  
    for(i=0;i<N;i++)  
        for(j=0;j<M;j++)  
            if(a[i][j]>a[row][column])  
            {  
                _____  
            }  
    *n=row;  
    return a[row][column];  
}  
main()  
{  
    int a[N][M]={9,11,23,6,1,15,9,17,20},max,n;  
    max=select(a,&n);  
    printf("max=%d,line=%d\n",max,n);  
}
```

20. 以下程序的功能是：从键盘上输入一行字符，将其存入一个字符数组中，然后输出该字符串。请填空。

```
#include"ctype.h"  
#include"stdio.h"  
main()  
{  
    char str[81],*sptr;  
    int i;  
    for(i=0;i<80;i++)  
    {  
        str[i]=getchar();  
        if(str[i]=='\n')  
            break;  
    }  
    str[i]=_____;  
    sptr=str;  
    while(*sptr)  
        putchar(*sptr_____);  
}
```

21. fun 函数的功能是：首先对 a 所指的 N 行 N 列的矩阵，找出各行中的最大的数，再求这 N 个最大值中的最小的那个数作为函数值返回。请填空。

```
#include <stdio.h>
#define N 100
int fun(int(*a)[N])
{
    int row,col,max,min;
    for(row=0;row<N;row++)
    {
        for(max=a[row][0],col=1;col<N;col++)
            if(_____)
                max=a[row][col];
        if(row==0)
            min=max;
        else if(_____)
            min=max;
    }
    return min;
}
```




第 9 章 编译预处理

本章的学习目标:

了解编译预处理语句的作用;掌握定义符号常量、定义带参数的宏以及解除宏定义的方法;掌握包含语句的用法以及被包含文件的含义;掌握条件编译语句的用法。

以“#”开始的行叫做编译预处理命令,这些行完成了与预处理器的通信。编译预处理命令由 ANSI C 标准统一规定,用来在程序编译前根据预处理命令对程序作相应处理。常用的编译预处理命令有宏定义、文件包含以及条件编译等。编译预处理语句的作用范围是从它在文件中的出现处到文件尾或者是被其他命令取消作用的位置。

9.1 宏 定 义

宏定义又称宏替换,主要功能是用一个指定的标识符(即宏名)代替一个字符串,从而使程序更加简洁。宏定义具体又分为不带参数的宏定义和带参数的宏定义两种。

9.1.1 不带参数的宏定义

1. 宏定义的格式

宏定义的一般格式是:

```
#define 标识符 字符串
```

其中,“#define”并不是 C 语言的关键字,而是发布给编译系统的预处理命令;“标识符”和“字符串”之间用空格隔开。标识符又称宏名,为了区别于一般变量,通常用大写字母表示(也可以用小写字母);字符串又称宏体,可以是常量、关键字、语句、表达式,还可以是空白,它没有类型和值的含义。于是,宏定义又可以描述如下:

```
#define 宏名 宏体
```

其作用是把宏名标识符定义为字符串。当宏名出现在程序中并进行编译预处理时,编译系统就能够把程序中出现的宏名标识符,一律用字符串去替换,然后再对替换处理后的源程序进行编译。把宏名置换为宏体的过程,叫做宏展开,又叫做宏替换。

例如:编程时经常犯的错误是在应写“==”的地方写成了“=”,为了避免此类错误,可以在程序开始处进行如下的宏定义:

```
#define EQ ==
```

然后在程序中需要用到“==”的地方写成 EQ,在程序预处理时会将 EQ 用“==”代替,从而避免了错误的发生。

又如：

```
#define YES 1
```

该宏定义就是前面多次用到的定义符号常量的形式，其作用是将 **YES** 定义为 1。

符号常量经过定义后，就可以在程序中作为常量使用。例如：

```
if(x==YES)
    printf("%d\n",YES);
```

经过编译预处理后，程序中的符号常量用定义它们的常数去替换，将得到如下的源程序：

```
if(x==1)
    printf("%d\n",1);
```

但是，若宏名出现在字符串中，则编译预处理不会对它进行替换。例如：

```
#define YES 1
char *ps;
ps="x= =YES";
printf("%s\n",ps);
```

变量 **ps** 右边出现的 **YES** 不会被置换，输出结果为 “x= =YES” 而不是 “x= =1”。除了常数外，宏体还可以是表达式或空。例如：

```
#define REG
```

在这个宏定义中，只有宏名，没有宏体。此时，**REG** 被定义为空，表示 **REG** 是一个被定义过的宏，可以和条件编译语句 **#ifdef**（参见第 9.3.1 节）配合使用。

一个 **#define** 只能定义一个宏，若需要定义多个宏，就要用多个 **#define**。



宏定义和文件包含一样，称为预处理命令行，不是 C 语句，故不能用分号结尾。

注意

2. 宏定义的嵌套

嵌套的宏定义就是用定义过的宏名去定义另一个宏名。例如：

```
#define WIDTH 80
#define LENGTH(WIDTH+40)
```

在第二个宏定义中，使用了前面定义过的宏名 **WIDTH**。在编译预处理时，程序中所有的 **WIDTH** 都被 80 所替换，所有的 **LENGTH** 又被 (80+40) 替换。所以如果程序中出现了如下语句：

```
var=LENGTH*20;
```

经过编译预处理后将变为：

```
var=(80+40)*20; /* var 的值为 2400 */
```

但是如按以下方式定义：



```
#define WIDTH 80
#define LENGTH WIDTH+40
var=LENGTH*20;
```

则经过编译预处理后变为：

```
var=80+40*20; /* var 的值为 880 */
```

也就是说，宏替换只是简单地用定义的宏体去替换宏名而不进行任何计算。因此，宏定义中若出现表达式时，有无圆括号其运算的结果将明显不同。所以为了保证宏定义在置换后仍保持正确的运算顺序，需要在宏定义中使用必要的圆括号将字符串括起来。

3. 宏定义的应用

(1) 定义符号常量或字符串

例如以下宏定义：

```
#define PI 3.1416
```

PI 就代表了常量 3.1416。因此，在 C 语言中把通过宏替换得到的标识符（如这里的 PI）称为符号常量。

定义符号常量是宏定义的一种应用，它可以提高程序的运行效率，因为编译程序处理常数的速度要比变量快；使用符号常量还可以方便地改变其值并达到一改全改的目的；用符号常量作为数组的维界说明，可以增加程序的通用性。

如果程序中需要多次用到某个字符串，也可以定义一个宏名来表示它，例如：

```
#define EMS "standard error on input\n"
```

定义完成后，若程序中出现语句“printf(EMS)”，则经过编译预处理后该语句被替换为：

```
printf("standard error on input\n");
```

(2) 定义更复杂的表达式或函数

此部分内容的详细讲解，请参见第 9.1.2 小节，这里只作为了解。

例 9.1 输入圆的半径，求圆周长和圆面积。

```
#include<stdio.h>
#define PI 3.1415926 /*定义符号常量*/
main()
{
    float r;
    double p ,s;
    scanf("%f" ,&r);
    p=2*PI*r;
    s=PI*r*r;
    printf("p=%f\n" ,p);
    printf("s=%f\n",s);
}
```

在上例中，当我们要把 PI 的精度提高到 3.14159265 时，只需更换宏定义即可；若不

引入宏定义,就需在源程序中做多次修改,由此可见引入宏定义的优越性。

使用宏定义时应注意以下几点:

- 宏定义必须以`#define` 开头,行末不加分号,因为它不是 C 语句。
- 每个`#define` 只能定义一个宏,且只占一行。
- `#define` 命令一般出现在函数外部,其有效范围是从出现处到所在源程序的文件结束处。
- 使用宏定义时可使用已定义过的宏,这称为宏定义的嵌套。
- 编译系统只对程序中出现的宏名用定义中的字符串作简单替换,而不作语法检查。
例如定义“`#define PI 3.14abc`”后,编译系统也会同样用字符串“`3.14abc`”替换 `PI`,而不管其含义是否正确,直到程序编译时才会提示出错。

ANSI C 中预定义了几个宏,如表 9-1 所示。它们可以被直接使用,并且不能在程序中取消定义,每一个宏名都有前后两条下划线。

表 9-1 预定义的宏及含义

预定义的宏	含义
<code>__DATE__</code>	表示当前日期的字符串
<code>__TIME__</code>	表示当前时间的字符串
<code>__FILE__</code>	表示当前文件名的字符串
<code>__LINE__</code>	表示当前行号的整数

9.1.2 带参数的宏定义

宏名可以带有一个或多个参数。带参数的宏定义的一般格式如下:

```
#define 标识符(形参表) 宏体
```

例如:

```
#define SQUARE(x) ((x)*(x))
```

其中, `SQUARE(x)` 为带参数的宏名, `x` 是它的形参; `((x)*(x))` 为宏体。在此定义之后,便可以在程序中利用 `SQUARE(x)` 来进行表达式 `((x)*(x))` 的计算了。若在程序中出现 `SQUARE(5)`,则被替换成 `(5*5)`; 若出现 `SQUARE(y)`,则被替换成 `(y*y)`; 若出现 `SQUARE(a+b)` 则被替换成 `((a+b)*(a+b))`, 此处形参的使用方法类似于函数中的形参。例如以下代码:

```
int i;
for(i=0;i<100;i++)
    printf("%d ",SQUARE(i));
```

该代码的作用是输出 0~99 的平方值。

在程序设计中,经常把那些反复使用的运算表达式或某些操作,定义为带参数的宏。这样,一方面使程序更加简洁;另一方面,可以使运算的意义更加明显。在定义带参数的宏时,对形参的数量没有限制。



带参数的宏也可以进行嵌套定义，例如：

```
#define MIN(x,y) (((x)<(y))?(x):(y))          /* 求两个量中的最小值 */
#define MIN4(a,b,c,d) MIN(MIN(a,b),MIN(c,d)) /* 求 4 个量中的最小值 */
```

下面给出几个常用的带参数的宏的实例：

```
#define MAX(x,y) (((x)>(y))?(x):(y))          /*求 x 和 y 中的较大的一个*/
#define ABS(x) (((x)>=0)?(x):0-(x))           /*求 x 的绝对值*/
#define PERCENT(x,y) (100.0*(x)/(y))         /*求 x 除以 y 的百分数值*/
#define ISODD(x) (((x)%2==1)?1:0)            /*判断 x 是否为奇数*/
#define SWAP(t,x,y) {t=x; x=y; y=t; }        /*交换 x 和 y 的值*/
```

1. 带参数的宏与函数的区别

注意，带参数的宏与函数在使用形式上虽有某些相似之处，但二者在本质上是不同的。

(1) 在程序控制上，函数的调用需要进行函数控制的转移；使用带参数的宏，则仅仅是表达式的运算。

(2) 带参数的宏，一般是一个运算表达式，所以它不像函数那样有固定的数据类型。宏的数据类型，可以认为是表达式运算结果的类型，随着使用的实参数据不同，运算结果也将呈现不同的数据类型。例如：

```
float x;
for(x=0;x<100;x+=1)
    printf("%f ",SQUARE(x));
```

上例得到的数据类型为实型。

(3) 在调用函数时，对使用的实参有一定的数据类型限制；而带参数的宏的实参，可以是任意数据类型。

(4) 在函数调用时，存在着从实参向形参传递数据的过程；而带参数的宏不存在这种过程。

2. 带参数的宏的特点

尽管带参数的宏和函数一样，可以作为程序模块而用于模块化的程序设计中，但是，使用带参数的宏，具有以下独有的特点：

(1) 程序中使用带参数的宏，由于不存在控制的转移和参数的传递，因而可以得到较高的程序执行速度；因此，对简短的表达式以及调用频繁、要求快速响应的场合，采用宏替换较好。但是由于定义代码的反复使用而使程序变大，因而在对源程序进行编译时，要花费较多的时间。

(2) 带参数的宏，除了使用运算表达式定义之外，还可以使用函数定义。在标准函数库中，经常使用以下这种形式：

```
#define getchar() fgetc(stdin)
```

其中 `getchar()` 是用另一个函数定义的宏，这样定义的宏替换与定义它的函数在性质上是相同的。

(3) 宏替换只是简单的字符替换而不进行计算, 因而一些过程是不能用宏替换去代替函数调用的(例如递归调用)。

(4) 宏定义如果使用不当, 会产生不易觉察的错误。

例 9.2 判断以下程序的输出。

```
#define Y(x)(x)*(x)
#define Z(x) x*x
#include<stdio.h>
main()
{
    int a=3,b=4,y,z;
    y=Y(a+b);
    z=Z(a+b);
    printf("\ny=%d,z=%d\n",y,z);
}
```

程序中使用了两个带参数的宏, 在编译预处理时将 $Y(a+b)$ 替换为 $(a+b)*(a+b)$, 将 $Z(a+b)$ 替换为 $a+b*a+b$ 。所以程序的运行结果为:

```
y=49,z=19
```

例 9.3 比较输出整数 1~10 的平方的两个程序。

(1) 采用函数调用的程序。

```
#include <stdio.h>
square(int n)
{
    return(n*n);
}
main()
{
    int i=1;
    while(i<=10)
        printf("%d\n",square(i++));
}
```

(2) 采用宏定义的程序。

```
#define SQUARE(n)((n)*(n))
#include <stdio.h>
main()
{
    int i=1;
    while(i<=10)
        printf("%d\n",SQUARE(i++));
}
```

运行这两个程序, 得到的结果分别是: (1,4,9,16,25,36,49,64,81,100)和(2,12,30,56,90)。很明显, 第一个程序是成功的, 而第二个程序没有达到预期的目的。其原因是在第二个程序中, 经宏替换后, `printf()`函数语句被置换为:



```
printf("%d\n", (i++)*(i++));
```

Turbo C 编译系统在处理函数实参求值时，采用自左而右逐项求值的方法，而 `i++` 是“先使用，再加 1”。因此，当 `i` 的初值为 1 时，在第一次循环中，先处理左边的 `(i++)`，即用 `i` 的原值 1 作为左边 `(i++)` 的值，该 `i` 经自加后变成 2 作为右边 `(i++)` 的 `i` 原值，结果是 `1*2`。计算后右边 `(i++)` 的 `i` 值自加 1 变成 3，再参加第二次循环。因此在第二次循环中进行的是 `3*4` 的运算，然后 `i` 变成 5。在第三次循环中进行的是 `5*6` 的运算。最后进行 `9*10` 的运算。

从上例可以看出，使用带参数的宏替换，将带来引入 `i++` 的副作用，而采用函数调用的程序时，则不会出现上述问题。

9.1.3 宏定义的解除

在程序开头使用的宏定义具有全局意义。如果想把宏定义的作用域限制在程序的某个范围内，可以使用 `#undef` 来解除已有的宏定义。其一般形式为：

```
#undef 宏名
```

其中，“宏名”是在此之前已定义过的。`#undef` 的功能是解除已定义的宏，使之不再起作用。例如：

```
#define SQUARE(x)((x)*(x))
...
#undef SQUARE
```

宏 `SQUARE` 只在 `#undef` 之前的程序中有效，在 `#undef` 之后就不能再使用这个宏了。注意解除带参数的宏定义时，只需给出宏名，而不必给出宏体。

在程序中将 `#define` 和 `#undef` 配合使用，就可以把宏定义的使用限制在二者之间的范围内，因此这也称为局部宏定义。

`#undef` 的另一个作用是可以重新进行宏定义。C 语言规定：符号常量和带参数的宏都不能重复定义，即程序中不能定义同名的宏。例如，在程序开头定义了 `SIZE` 是 256，在程序的另一个地方需要定义 `SIZE` 是 512，若使用以下代码则是不允许的。

```
#define SIZE 256
...
#define SIZE 512
```

但是，如果在定义 `SIZE` 为 512 之前，使用以下语句来解除原先的定义，就可以再定义 `SIZE` 为 512。

```
#undef SIZE
```

程序是由多个源文件组成的，在不同的源文件中可能会出现同一个宏名被定义为不同宏体的情况。若将这些源文件合并在一起，就会出现重复宏定义的错误，此时在每个源文件的末尾把使用过的宏定义均用 `#undef` 解除即可。

9.2 文件包含

文件包含的功能是把一个指定文件嵌入到现行的源程序文件中，再对嵌入后的源程序文件进行编译处理。这可以有效地减少重复编程。

9.2.1 文件包含的格式

文件包含的一般格式如下：

```
#include <文件名>
```

或：

```
#include "文件名"
```

其中，“文件名”是指磁盘中文本文件的名字。例如：

```
#include <stdio.h>
#include "file1.c"
```

其中，`stdio.h` 是由系统提供的标题文件；`file1.c` 则是用户提供的 C 程序文件。`#include` 的功能是将指定的文件包含到本源程序文件中，这样本源程序文件就可以调用被包含文件中的函数。一般来说，一个 `#include` 只能指定一个包含文件，如果需要把多个文件都嵌入到本源程序文件中，则必须使用多个 `#include`。

文件包含中，可以用尖括号或双引号将被包含文件括住，二者的差别在于编译系统将使用不同的方式搜索包含文件。

(1) 当用尖括号括住被包含文件时，编译系统将仅仅在系统设定的标准目录中搜索包含文件。例如：

```
#include <string.h>
```

则编译系统只在系统设定的子目录 `include` 下查找包含文件 `string.h`。如果在标准目录下不存在指定的文件，编译系统会发出错误信息，并停止编译过程。

(2) 当使用双引号括住被包含文件且文件名中无路径时，编译系统将首先在源文件所在的目录中查找；若未找到，再到系统设定的标准目录中查找。当使用双引号括住被包含文件并指定该文件的路径时，则系统就按指定的文件路径去查找。例如：

```
#include "c:\user\user.h"
```

编译系统将在 `c:\user` 子目录下查找被包含文件 `user.h`。

9.2.2 文件包含的功能

1. 包含标题文件

C 语言提供了若干标准库函数。`Turbo C 2.0` 提供了 26 个标题文件（又称为头文件或包含文件）。每个标准库函数都与某个标题文件相对应。标题文件中包括相应库函数中的适当说明，以及各种有用的数据结构说明和宏定义。用 `#include` 命令把某个标题文件包含到自



己的源程序文件中之后，编译系统在编译源程序前就会用指定的被包含文件的内容取代 `#include` 命令。也就是说，从磁盘中读取被包含文件并将它插入到源程序中 `#include` 所在的位置上，使它成为源程序的一部分。这样，用户程序就可以直接调用被包含文件中的函数而不必自己重新编写了。

例如，当没有用 `#include` 将标题文件 `string.h` 包含到源程序文件中，而程序中却使用了 `strlen()` 函数时，编译时就会出现错误信息，因为 `strlen()` 函数定义在标题文件 `string.h` 之中。如果包含了 `string.h` 文件，问题就会迎刃而解。

系统提供的标题文件，一般是以 “.h” 为后缀名，它们集中存放在由系统建立的子目录 `include` 中。

2. 包含用户文件

除了标题文件之外，文件包含的还可以是用户自己设计的包含文件。

当用户文件由多个源程序文件组成时，为了避免重复性的说明和定义、提高工作效率、提高程序的可靠性和可维护性，可以把各个源文件共同使用的函数以及符号常量的宏定义等组建为单独的用户标题文件，然后在各个源文件中用 `#include` 包含该用户的标题文件。这样不但可以使程序简洁明快，更保证了各个源程序文件中函数定义和符号常量定义的一致性。因此，文件包含也是模块化程序设计的手段之一。

例 9.4 用户标题文件的建立。

假设某数学演示程序，当用户输入半径后，程序就可输出该半径对应的圆周长、圆面积、球表面积和球体积。为此，可设计如下的程序：

```
#include "circle.h"
#include <stdio.h>
main()
{
    float r;
    printf("Input R:");
    scanf("%f",&r);
    printf("C=%.2f  ",CIRCLE(r));
    printf("A=%.2f  ",AREA(r));
    printf("S=%.2f  ",SURFACE(r));
    printf("V=%.2f\n",VOLUME(r));
}
```

该程序使用到了计算圆的面积和周长、球表面积和体积的宏定义。现把这些宏定义单独存放在标题文件 `circle.h` 中：

```
#define PI 3.141593
#define CIRCLE(r) 2*PI*(r)
#define AREA(r) PI*(r)*(r)
#define SURFACE(r) AREA(r)*4
#define VOLUME(r) SURFICE(r)*(r)/3
```

这样，当对源程序进行编译时，由于它已包含了标题文件 `circle.h`，所以编译程序会自动将

该标题文件插入到 `main()` 的前面以取代 `#include`，从而成为程序的一部分。程序的某次运行结果为：

```
Input R:2.5✓  
C=15.71 A=19.63 S=78.54 V=65.45
```

采用文件包含指令，不同的用户可以共享同一个标题文件，也可以共享同一个 C 源程序文件。例如：

```
#include<myheadfile.h>  
#include<mycfile.c>
```

其中，`myheadfile.h` 是用户自定义的头文件，而 `mycfile.c` 是用户自己编写的 C 源程序文件。

9.3 条 件 编 译

条件编译就是根据条件对 C 程序的某一部分进行编译，其他部分不编译。条件编译语句有 `#if` 语句、`#if-#elif` 语句、`#ifdef` 语句和 `#ifndef` 语句。条件编译可以使源代码更易于修改，兼容性更强，并使目标代码缩短。当程序在不同系统上编译、在同一系统的不同编译器上编译或进行不同目的的编译时，利用条件编译让编译预处理器根据条件忽略某些语句或只编译某些语句，这可以减少对程序语句的修改。

1. #if 语句

`#if` 语句的一般形式是：

```
# if 表达式  
    程序段 1  
# else  
    程序段 2  
# endif
```

`#if` 的执行过程是：如果“表达式”为真，就编译“程序段 1”，否则编译“程序段 2”。作为一种特例，当条件为假不执行任何操作时，可以省略 `#else`。

2. #if-#elif 语句

`#if-#elif` 语句的形式与 `if-else if` 语句的形式基本相同，`elif` 相当于 `else if` 的缩写形式，`#if-#elif` 语句的一般形式是：

```
# if 表达式 1  
    程序段 1  
# elif 表达式 2  
    程序段 2  
...  
# else  
    程序段 n  
# endif
```



如果“表达式 1”的值为真，就编译“程序段 1”，否则如果“表达式 2”为真，编译“程序段 2”；依此类推，若各表达式都不为真，则编译“程序段 n”。

3. #ifdef 语句

#ifdef 语句的一般形式是：

```
# ifdef 标识符
    程序段 1
# else
    程序段 2
# endif
```

如果“标识符”在此之前已经由#define 给出了定义，就编译“程序段 1”；否则编译“程序段 2”。如果没有“程序段 2”，则#else 可省略。

4. #ifndef 语句

#ifndef 语句的一般形式是：

```
# ifndef 标识符
    程序段 1
# else
    程序段 2
# endif
```

如果“标识符”在此之前未经定义，就编译“程序段 1”；否则编译“程序段 2”。如果没有“程序段 2”，则#else 可省略。

例 9.5 条件编译应用举例。要求根据不同的语言环境程序有不同的输出，方法是当使用的语言发生变化时，只需要将程序中的#define LANGUAGE SYMBOL 改变成不同的语言名，程序预编译时即可编译不同的程序段。实现代码如下：

```
#define ENGLISH 1
#define CHINESE 2
#define SYMBOL 3
#define LANGUAGE SYMBOL
#if LANGUAGE= =CHINESE /* 如果使用中文，则输出汉语拼音的函数被编译 */
void print()
{
    printf("nin hao !\n");
}
#elif LANGUAGE= =ENGLISH /* 如果使用英文，则输出英文的函数被编译 */
void print()
{
    printf("hello!\n");
}
#elif LANGUAGE= =SYMBOL /* 如果使用符号，则输出符号的函数被编译 */
void print()
{
```

```
printf(" :-)\n");

#else                                /* 如果是其他语言，则输出一串*的函数被编译 */
void print()
{
    printf("*****\n");
}
#endif
#include<stdio.h>
main()
{
    print();
}
```

在本程序中，因为 LANGUAGE 被定义为 SYMBOL，所以输出 “:-)”。

例 9.6 条件编译应用举例。为了调试程序，经常在程序中写一些 printf() 语句来输出程序执行时的中间变量的值，而当程序调试成功以后就不再需要输出这些值了。这可以用下面的条件编译来实现：

```
#define DEBUG 1
...
#if DEBUG
    printf("debug: a=%d\n",a);
#endif
```

当调试成功后，将符号常数 DEBUG 的值改为 0，则重新编译时，这些 printf() 就不会被编译且中间变量的值也不会被输出了。

或者写成如下代码：

```
#define DEBUG
...
#ifdef DEBUG
    printf("debug: a=%d\n",a);
#endif
```

当调试成功后，将定义 DEBUG 的语句删除再重新编译即可。

要 点 回 顾

1. 以 “#” 开始的行称为编译预处理命令，这些行完成与预处理器的通信，预处理的步骤在编译前自动执行。编译预处理语句的作用范围是从它在文件中的出现处到文件尾或者是被其他命令取消作用的位置。常用的编译预处理命令有宏定义、文件包含、条件编译等。

2. 宏定义又称宏替换，主要功能是用一个指定的标识符（即宏名）代替一个字符串。宏定义又分为不带参数的宏定义和带参数的宏定义两种。宏定义可以嵌套，即用定义过的宏名去定义另一个宏名。



不带参数宏定义的一般格式是：

```
#define 标识符 字符串
```

宏名可以带有一个或多个参数。带参数的宏定义的一般格式是：

```
#define 标识符(形参表) 宏体
```

带参数的宏与函数在使用形式上虽有某些相似之处，但二者在本质上是不同的。

3. 如果想把宏定义的作用域限制在程序的某个范围内，可以使用`#undef`来解除已有的宏定义。其一般形式为：

```
#undef 宏名
```

4. 文件包含的功能是把一个指定文件嵌入到现有的源程序文件中。文件包含的一般格式是：

```
#include <文件名>
```

或：

```
#include "文件名"
```

5. 条件编译语句有`#if`语句、`#if-#elif`语句、`#ifdef`语句和`#ifndef`语句。条件编译就是根据条件对 C 程序的某一部分进行编译，其他部分不编译。

习 题

1. 以下程序中循环执行的次数是_____。

```
#define N 2
#define M N+1
#define NUM (M+1)*M/2
main()
{
    int i;
    for(i=1; i<=NUM; i++)
        printf("%d\n",i);
}
```

2. 下面程序段的执行结果是什么？

```
#define Y(x)(x)*(x)
#define Z(x) x*x
#include<stdio.h>
main()
{
    int a=1,b=2,y,z;
    y=Y(a+b);
    z=Z(a+b);
    printf("\ny=%d,z=%d\n",y,z);
}
```

3. 下面程序段的执行结果是什么？

```
#define X 6
#define Y X+1
#define Z Y*X/2
#include<stdio.h>
main()
{
    int a; a=Y;
    printf("%d\n",Z);
}
```

4. 下面程序段的执行结果是什么？

```
#define US 1
#define ENGLAND 2
#define COUNTRY US
#if COUNTRY==US
    char currency[ ]= " dollar ";
#elif COUNTRY==ENGLAND
    char currency[ ]= " pound ";
# endif
#include<stdio.h>
main()
{
    printf("%s\n",currency);
}
```



第 10 章 结构体与共用体

本章的学习目标:

了解结构体类型和结构体变量的概念, 掌握结构体类型、结构体变量以及结构体数组的定义方法, 掌握结构体成员的引用方法; 了解结构体类型指针的概念、定义方法及用法, 会利用结构体类型指针对结构体变量或结构体数组元素的成员进行操作; 了解递归结构和数据链表的概念, 掌握数据链表的基本操作方法; 了解共用体类型和共用体变量的概念, 掌握共用体变量成员的引用方法; 了解类型定义 `typedef` 的用法, 会用 `typedef` 定义新类型名。

10.1 结构体类型

10.1.1 结构体类型的概念

结构体是不同数据类型变量的集合体, 又称为结构; 结构体类型是一种复合数据类型, 它可容纳各种数据类型的变量及数组, 相当于常用的记录。例如, 要描述一个学生的记录, 包括学号、姓名、年龄、性别、成绩和住址等数据项, 就可以把它们组合在一起, 形成一个新的数据类型, 即结构体类型或结构类型。其中学号、姓名、年龄、性别、成绩和住址都是其成员。

10.1.2 结构体类型的定义

结构体类型包含的数据项组合可能千差万别, 所以结构体类型不是惟一的, 不能像 `int`、`float` 那样用一个关键字来描述, 而是让用户自己去定义它的名称和包含的数据项。定义结构体类型的一般格式为:

```
struct 结构体类型名
{
    数据类型 成员 1;
    数据类型 成员 2;
    ...
    数据类型 成员 n;
};
```

其中, `struct` 是关键字, “结构体类型名” 和各成员名由用户确定。不同的结构体类型名代表不同的结构体类型, 它们的成员也各不相同。例如, 上面的学生记录可定义成一个名为 `struct student` 的结构体类型:

```
struct student
{
```

```
char number[5];          /* 也可以定义为 char *number */
char name[9];            /* 也可以定义为 char *name */
int age;
char sex;
float score;
char address[31];        /* 也可以定义为 char *address */
};                        /* 末尾的分号不能缺少 */
```

它的数据结构如图 10-1 所示。

number[5]	name[9]	age	sex	score	address[31]
-----------	---------	-----	-----	-------	-------------

图 10-1 数据结构

在定义一个结构体类型时，可以利用已定义的另一个结构体类型来定义其成员类型。
例如：

```
struct date
{
    int month;
    int day;
    int year;
};          /* 定义了一个 struct date 类型 */
struct student1
{
    char number[5];
    char name[9];
    struct date birthday; /* birthday 是一个结构体类型的成员 */
    char sex;
    float score;
    char address[31];
};
```

它的数据结构如图 10-2 所示。

number[5]	name[9]	birthday			sex	score	address[31]
		month	day	year			

图 10-2 嵌套的结构体

可以看出，struct date 作为一个结构体类型名，可以出现在结构体类型 struct student1 的定义中，也就是说一个结构体中可以嵌套另一个结构体，这种形式称为嵌套结构。这时，如果 stu1 被定义为 struct student1 类型，则它的值的形式可以如图 10-3 所示，它共占存储单元中的 56 个字节。其中各成员所占字节数均标于成员下方。

0301	zhangsan	7	15	1985	M	567. 0	Jinan er huan nan lu
5	9	2	2	2	1	4	31

图 10-3 嵌套结构体值的存储形式

对结构体类型的说明如下：



(1) 一个结构体类型中可以包含若干个数据项，每个数据项都属于一种已有的数据类型。这些数据项已不再是一般意义上的变量、数组或指针，C 语言称它们为结构体成员。

(2) 结构体类型和基本类型的不同之处在于：第一，结构体类型不是由系统定义的，而是由用户定义的；第二，结构体类型不是惟一的，用户可以定义多种不同的结构体类型；第三，结构体类型定义之后，与基本类型一样，只规定了内存分配模式，并不实际占用内存空间。

(3) 结构体类型定义的位置可以是函数内部，也可以是函数外部。在函数内部定义的结构体类型，只能在函数内部使用；在函数外部定义的结构体类型，其有效范围是从定义处开始，直到它所在的源程序文件结束。

(4) 结构体成员可以和程序中的其他变量同名，也可以和另一个结构体类型的成员同名。

10.2 结构体变量

10.2.1 结构体变量的定义

结构体变量是结构体类型的变量。结构体类型的作用与 `int`、`float` 等相同，并不代表哪个具体的数据，只规定了内存的分配模式，真正代表或存储数据的是结构体变量以及结构体数组。结构体变量和结构体数组需要先定义再使用。

定义一个结构体变量的方法有分别定义法、直接定义法和一次性定义法。

1. 分别定义

分别定义法，即先定义结构体类型，再定义结构体变量。定义的一般形式为：

```
struct 结构体名
{
    成员表
};
struct 结构体名 变量名列表;
```

例如：

```
struct student
{
    char number[5];
    char name[9];
    int age;
    char sex;
    float score;
    char address[31];
};
struct student stu1,stu2;
```

在上例中，先定义结构体类型名 `struct student`，再用它来定义结构体变量 `stu1` 和 `stu2`。

请注意, `struct student` 代表类型名 (类型标识符), 下面的写法是错误的:

```
struct stu1,stu2;    /*(没有声明是哪一种结构体类型)*/
student stu1,stu2;   /*(没有关键字 struct, 系统将不认为是结构体类型)*/
```

在定义了一个结构体类型后, 可以多次用它来定义变量。例如还可以用 `struct student` 来定义其他变量:

```
struct student stu3,stu4;    /*(再定义两个 struct student 类型的变量)*/
```

2. 直接定义

直接定义, 即在定义一个结构体类型的同时定义一个或若干个结构体变量。定义的一般形式为:

```
struct 结构体名
{
    成员表
}变量名表;
```

例如:

```
struct student
{
    char number[5];
    char name[9];
    int age;
    char sex;
    float score;
    char address[31];
}stu1,stu2;
```

其中, `stu1` 和 `stu2` 是 `struct student` 型的结构体变量, 每个结构体变量都包含该结构体类型中的 6 个成员。

这是第一种形式的紧凑形式, 既定义了类型, 又定义了变量。还可以用此 `struct student` 再定义其他变量:

```
struct student stu3,stu4;    /*(再定义两个 struct student 类型的变量)*/
```

3. 一次性定义

一次性定义即直接定义结构体类型的变量。一般形式为:

```
struct
{
    成员表
}变量名表;
```

例如:

```
struct
{
    char number[5];
```



```
char name[9];
int age;
char sex;
float score;
char address[31];
}stu1,stu2;
```

其中，stu1 和 stu2 是指上述结构体类型的结构体变量名。

此时只是直接定义了 stu1 和 stu2 这两个变量为具有上面花括号内结构的结构体类型，但没有定义此结构体类型的名字，因此不能再用它来定义其他变量。

在这 3 种定义方式中，前两种比较灵活，定义的结构体类型可以多次使用。而若采用第 3 种方式，因为没有结构体类型名，所以若需要定义同类型的结构体变量，只能重新定义结构体及变量。

对结构体变量定义的说明如下：

(1) 结构体变量的定义位置决定了它的作用范围，即可见性；结构体变量的存储类别决定了它的生存期，这方面的规则都和普通变量相同。但结构体变量不能用 register 进行存储类别说明。

(2) 结构体变量占用内存的情况取决于结构体中各成员占用内存的要求。在定义了变量 stu1 和 stu2 为 struct student 类型之后，它们就具有 struct student 结构体类型的特征，也就是说，变量 stu1 不是一个简单变量，它的值也不是一个简单的整数、实数或字符，而是由许多基本数据组成的复合值。例如，stu1 的值可以是：

0301	zhangsan	18	M	567.0	Jinan er huan nan lu
------	----------	----	---	-------	----------------------

在内存中，stu1 占用一片连续的存储单元，共 52 个字节。结构体变量数据的长度（字节数）可用语句“sizeof(结构体类型名)”或“sizeof(结构体变量名)”来算出。还可用 printf 函数输出它的值：

```
printf("%d",sizeof(struct student));
```

或：

```
printf("%d",sizeof(stu1));
```

10.2.2 结构体变量的初始化

结构体变量的初始化与数组的初始化非常相似，只要把对应的常数值放在花括号中（即初值表），各常数值之间用逗号隔开即可。例如：在定义了 struct student 结构体类型之后可以定义结构体变量并进行初始化。初始化时的语句如下：

```
struct student stu1={"0301", "zhangsan", 18, 'M', 567.0,
                    Jinan er huan nan lu" };
```

在初始化时，按照所定义的结构体类型的数据结构，依次写出每个初始值，在编译时系统自动将它们赋给此变量中的各成员。也可以写成如下的形式：

```
main()
```

```
{
    static struct student
    {
        char number[5];
        char name[9];
        int age;
        char sex;
        float score;
        char address[31];
    }stu1={"0301","zhangsan",18,'M',567.0," shandong Jinan "};
}
```

此时的 stu1 是一个静态局部结构体变量。但不要写成以下这种形式：

```
static struct student
{
    char number[5]= "0301";
    char name[9]= "zhangsan" ;
    int age=18;
    char sex='M';
    float score=567.0;
    char address[31]= " shandong Jinan ";
}stu1;
```

如果一个结构体类型中又嵌套另一个结构体类型，则初始化时仍需对各个基本数据类型的成员给予初值。

10.2.3 结构体变量的成员引用

由于一个结构体变量可以是一组数据类型不同的数据所组成的集合体，所以一般不对结构体变量整体进行操作，而是对其成员进行操作。为了访问或引用结构体变量的某一个成员（简称结构成员），必须指明该成员属于哪一个结构体变量。C 语言提供了访问结构成员的运算符“.”，该运算符可以将结构成员与结构名联系起来。访问结构成员的形式为：

结构体变量名.成员名

“.”称为成员运算符（也称点运算符），是一个二元运算符，它的运算优先级最高，结合性为从左至右。例如：“stu1.age”可以访问或引用结构体变量 stu1 的成员 age。访问过程是：先找到结构体变量 stu1 的起始地址，然后从 stu1 所占内存单元中找到 age 成员。这样，结构成员就可以和普通变量一样参加各种运算，例如可把成员 age 看成是整型变量，按照对整型变量的处理方法对其进行操作即可，例如以下代码：

```
scanf("%d%s",&stu1.age,stu1.name);
printf("%d%s\n",stu1.age,stu1.name);
```



注意

&stu1.age 表示取结构成员的地址，由于运算符“.”的优先级高于“&”，所以先进行“.”运算，即先取成员，然后再取该成员的地址，相当于&(stu1.age)。



例 10.1 结构成员的访问。

```
#include <stdio.h>
#include <string.h>
main()
{
    static struct student
    {
        char number[5];
        char name[9];
        int age;
        char sex;
        float score;
        char address[31];
    }stu1,stu2;
    strcpy(stu1.number,"0301"); /*用函数将字符串复制给结构成员字符型数组*/
    strcpy(stu1.name,"zhangsan");
    stu1.age=19;
    stu1.sex='m';
    stu1.score=560.5;
    strcpy(stu1.address,"shandong jinan");
    printf("%5s,%9s,%2d,%c,%7.2f,%31s\n",stu1.number,
        stu1.name,stu1.age,stu1.sex,stu1.score,stu1.address);
}
```

其中，strcpy 为字符串复制函数，功能为将字符串赋值给字符型指针或数组。程序运行结果为：

```
0301,zhangsan,19,m, 560.50, shandong jinan
```

如果在同一函数中又定义了一个 age 变量，则该变量需在内存中另分配单元，不在 stu1 范围之内。stu1.age 代表结构体变量中的成员，age 代表简单变量。

如果有两个变量 stu1 和 stu2 均定义为同一个结构体类型 struct student，为引用两个变量中的 age 成员，应该分别用 stu1.age 和 stu2.age，它们代表内存中不同的存储单元，引用结果也为不同的值。

如果一个结构体类型中又嵌套一个结构体类型，则访问一个结构成员时，应采取逐级访问的方法，即使用多个点操作符直到得到所需访问的结构成员为止。例如，对于图 10-2 所示的结构体变量，若要得到 zhangsan 的出生年份，可以应用以下形式：

```
stu1.birthday.year
```

而不能只写 year 或 birthday.year 或 stu1.year。

可以对结构体变量的结构成员进行各种有关的计算，允许计算的种类与相同类型的简单变量的种类相同。

10.2.4 结构体变量的赋值和输入输出

C 语言不允许对结构体变量名进行直接赋值、输入和输出的操作，但是可以将一个结

构体变量作为一个整体赋给另一个具有相同类型的结构体变量。例如，下面的赋值、输入、输出的操作是不允许的：

```
stu1={"0301","zhangsan",18,'M',567.0," shandong Jinan "};
scanf("%s %s %d %c %f %s",stu1,stu2);
printf("%s %s %d %c %f %s ",stu1,stu2);
```

但整体复制是允许的：

```
struct student stu1={"0301","zhangsan",18,'M',567.0," shandong
                    Jinan "},stu2;
stu2=stu1;
```

以上代码将结构体变量 `stu1` 中各个结构成员的值依次赋给同类型结构体变量 `stu2` 中相应的结构成员；也可以把一个结构体变量中的内嵌结构体类型成员赋给另一个结构体变量的相应部分。



注意

结构体类型名不同的两个变量不允许相互赋值，即使是两者包含同样的成员。不允许用赋值语句将一组常量赋给一个结构体变量。也不允许把一个结构体变量作为一个整体进行输入或输出的操作。对结构体变量的输入输出只能按照结构成员访问的方法进行。

10.3 结 构 数 组

具有相同结构体类型的一组变量可以组成结构体数组（或简称为结构数组）。在例 10.1 中，定义了两个描述学生记录的结构体变量 `stu1` 和 `stu2`，每个变量只能存放一个学生记录。如果有很多个学生记录，可以用结构数组来存放，结构数组的每个元素代表一个结构体类型数据。

10.3.1 结构数组的定义

结构数组必须先定义然后才可以引用。定义结构数组和定义结构体变量的方法类似，有直接定义、间接定义和一次性定义 3 种方式，不同的是，定义结构数组需要在结构名的后面加上数组维界说明符。例如，利用前面定义的结构类型 `struct student` 可以定义一个结构数组 `stu[20]`：

```
struct student stu[20];
```

结构数组 `stu` 中含有 20 个元素，每一个元素都是 `struct student` 型的结构体类型数据。

10.3.2 结构数组的初始化

同简单结构体变量的初始化类似，结构数组的初始化是在数组定义的同时将初始数据赋值给数组元素，初始数据应与数组元素的各成员一一对应。为清晰起见，每一个数组元素的初始数据都用花括号括住。例如：



```
struct student stu[2]={{"0301","zhangsan",19,'m',567.5,
                        " shandong Jinan " },
                       {"0302","lisi",20,'m',560.5,"jiangsu nanjing "}};
```

类似于以上代码所示的列出所有初始数据的情况，数组的维界也可以省略不写。

10.3.3 结构数组的元素成员访问

一个结构数组元素相当于一个结构体变量，因此访问结构数组元素的成员与访问结构体变量的成员具有相同的规则。例如：`stu[i].age` 表示结构数组第 *i* 个元素中成员 `age` 的值；`&stu[i].age` 表示结构数组第 *i* 个元素中成员 `age` 的地址。因为“.”运算符优先于“&”和“*”运算符，因此，`&stu[i].age` 与 `&(stu[i].age)` 等价。

例 10.2 用结构数组处理成绩表。设结构体成员为学号、姓名、成绩，用结构数组存放一批学生的数据。

```
#include<stdio.h>
#define N 4
struct student
{
    int number;
    char name[9];
    int score;
};
struct student stu[N] ,temp;
main()
{
    int j,k,p;
    /*输入学号、姓名和成绩*/
    for(j=0;j<N;j++)
    {
        printf("please input number name score");
        scanf("%d %s %d",&stu[j].number,stu[j].name,&stu[j].score);
    }
    /*从小到大排序*/
    for(j=0;j<N-1;j++)
    {
        p=j;
        for(k=j+1;k<N;k++)
            if(stu[p].score > stu[k].score)
            {
                p=k;
                temp=stu[j];
                stu[j]=stu[p];
                stu[p]=temp;
            }
    }
    /*输出排序后的名次、姓名和数学成绩*/
    printf("place|--number--|--name--|-- score \n");
    for(j=0;j<N;j++)
```

```
{
    printf(" %d | %d | %s | %5d\n",
           j+1, stu[j].number, stu[j].name, stu[j].score);
    printf("-----\n");
}
```

10.4 结 构 指 针

指向结构体变量或结构数组的指针称为结构指针。指针的值是结构体变量或结构数组元素的地址。

10.4.1 结构指针的定义

定义结构指针和定义一般指针相似，形式为：

```
[存储类别] 结构类型 *指针 1, *指针 2, ...;
```

例如：

```
static struct student *sp;
```

该代码定义了一个静态的 `struct student` 结构类型的指针 `sp`。其中，`student` 是已经定义过的结构类型名。

10.4.2 结构指针的初始化

把实际存在的某个结构变量或结构数组元素的地址赋给指针，使指针指向该变量或数组元素，这一过程称为结构指针的初始化。例如：

```
static struct student *sp=&stu1;
```

该代码定义了一个指向 `static struct student` 结构类型的指针 `sp`，并使其指向结构体变量 `stu1`（`stu1` 也是 `static` 类）。也可以使用赋值的方法使指针指向 `stu1`：

```
static struct student *sp;
sp=&stu1;
```

10.4.3 结构指针的访问

定义了结构指针并初始化之后，就可以用结构指针来访问结构成员了。指针访问结构成员的形式如下：

```
(*指针名).成员名
```

该形式等价于前面学习过的用结构体变量名访问结构成员的形式：结构体变量名.成员名。

由于指向结构的指针使用得非常频繁，故 C 语言提供了另一种简便的取结构成员的运算符“`->`”，该运算符称为指向成员运算符，它由一个减号和一个大于号组成。因此用指针访问结构成员，又可以写成：



指针名->成员名

运算符“->”和“.”都是访问结构成员运算符，同处于最高优先级，结合性也都是从左到右。

假设已定义：

```
struct student stu1,*sp=&stu1;
```

若要访问其结构成员 `name`，则下面三种形式等效：`stu1.name`、`(*sp).name` 和 `sp->name`。

显然，当利用结构体变量名访问结构成员时应用“.”比较方便；当利用指针访问结构成员时，应用“->”较为方便。

例 10.3 指向结构体变量的结构指针应用举例：用结构指针输出结构成员。

```
#include <stdio.h>
main()
{
    static struct student
    {
        char number[5];
        char name[9];
        int age;
        char sex;
        float score;
        char address[31];
    }stu1={ "0301", "zhangsan", 19, 'm', 560.5, " shandong Jinan " },*sp=&stu1;
    printf("%5s,%9s,%2d,%c,%7.2f,%31s\n", sp->number, sp->name,
        sp->age, sp->sex, sp->score, sp->address);
}
```

程序运行结果同例 10.1。

例 10.4 指向结构数组的结构指针应用举例：判断下面程序的执行结果。

```
#include<stdio.h>
main()
{
    static struct data
    {
        int x;
        int *y;
    }*p;
    static int z0[]={11,12};
    static int z1[]={31,32};
    static struct data z[2]={{100,z0},{300,z1}};
    /*成员 y 分别指向一维数组 z0 和 z1*/
    p=z; /*结构指针 p 指向结构数组 z*/
    printf("\na=%d,b=%d,c=%d,d=%d", *p->y, (p++)->x, ++p->x, p->x);
}
```

分析：当 `printf` 中有多个输出项时，先从右向左进行每一项的计算，最后一起输出。P

原指向 `z[0]`, `p->x` 等价于 `z[0].x`, 值为 100; `++p->x` 等价于 `++(p->x)`, 值为 101, 并且 `z[0].x` 的值变为 101; `(p++)->x` 等价于 `z[0].x`, 值为 101; 执行 `p++` 后, `p` 指向 `z[1]`, 所以 `*p->y` 等价于 `z[1].y`, 值为 31。所以输出结果为:

```
a=31, b=101, c=101, d=100
```

当运算符 “.”、“->” 以及其他多种运算符同时出现在一个表达式中时, 一定要注意运算符的优先级和结合性。本例中: `*p->y` 等价于 `*(p->y)`, `++p->x` 等价于 `++(p->x)`。

10.5 递归结构和链表

结构体变量的结构成员是另一个结构体变量, 这种情况称为结构嵌套。若结构成员不能是自身类型的结构体变量, 只能应用同类型的结构指针作为其结构成员, 则这种结构称为递归结构, 又称自嵌套结构, 即结构成员是具有该结构类型的结构指针。

10.5.1 递归结构的定义

递归结构的定义形式如下:

```
struct 结构类型名
{
    结构成员说明;
    struct 结构类型名 *指针名;
};
```

例如:

```
struct info /*定义递归结构*/
{
    char name[10];
    int score;
    struct info *pNext;
};
```

该结构有 3 个成员, 其中一个成员 `pnext` 是指向同类型 `info` 的结构指针。这样的递归结构描述的是链表。链表中的每个结构体变量称为结点, 如图 10-4 所示。

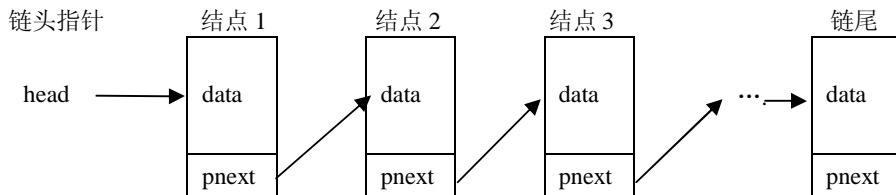


图 10-4 单向链表的递归结构

该链表的每个结点有两个域: 数据域 `data` 和指针域 `pnext`。数据域可能是简单的数据, 也可能是数组、结构等复合型数据, 例如上面定义的数据域包含 `name` 数组和 `score`; 指针域存放下一个结点的地址。



10.5.2 递归结构的应用

递归结构常用来处理动态数据结构的问题，用指针和结构体构成链表，可以实现单向链表的建立、遍历、插入、查找与删除等操作。

例 10.5 定义 3 个递归结构变量 a、b、c，并组成单向链表。

```
#include<stdio.h>
main()
{
    struct info    /*定义递归结构*/
    {
        int num;
        struct info *pNext;
    }a,b,c;
    a.num=1;a.pNext=&b;
    b.num=2;b.pNext=&c;
    c.num=3;c.pNext=NULL;
}
```

例 10.6 针对图 10-4 所示的单向链表进行建立、遍历、插入、查找与删除操作。先作如下定义：

```
struct info        /*定义递归结构*/
{
    char name[10];
    int score;
    struct info *pNext;
};
```

(1) 建立链表。

一个链表总是包含一个链首指针，操作链表时一般都由链首指针开始，最后一个结点的 pnext 值为空 (NULL)。首先定义链首指针和结点指针，并使其指向新建立的链首结点，然后通过结点指针为链首结点输入数据，再建立多个结点并且输入数据。

程序中可构造任意长的链表，每建立一个新结点都要用动态存储分配函数为其分配空间，使指针指向它，并给结点赋值。

```
struct info *creat(int n)    /*建立 n 个结点链表的函数*/
{
    int i;
    struct info *head,*p;
    printf("creat a list,the length is %d\n",n);
    p=head=(struct info *)malloc(sizeof(struct info));
    /*head 表示指向链首的指针，通过动态内存分配获取地址，p 指向新结点的指针*/
    printf("please input name");
    scanf("%s",p->name);
    printf("please input score");
    scanf("%d",&p->score);
    for(i=2;i<=n;i++)
    {
```

```

    p->pnext=(struct info *)malloc(sizeof(struct info));
    /* 建立新结点*/
    printf("please input name");
    scanf("%s",p->name); /* 给新结点赋值*/
    printf("please input score");
    scanf("%d",&p->score);
}
p->pnext=NULL; /*建立多个结点,并使最后结点的 pnext 指向 NULL。*/
return(head);
}

```

(2) 遍历链表,即处理链表中的所有结点,利用以下代码即可输出链表中的所有结点信息。

```

void list(struct info *head)/*显示各结点信息的函数*/
{
    struct info *p1;
    p1=head;
    while(p1!=NULL)
    {
        printf("%-15s%d\n",p1->name,p1->score);
        p1=p1->pnext;
    }
}

```

上述代码中, head 接收调用函数传递过来的链表首地址,从该地址开始逐个输出结点信息,方法是处理完一个结点后,利用语句“p1=p1->pnext”下移指针,直到链表尾。因为链表尾结点的 pnext 的值为 NULL,可作为判断循环结束的条件。

(3) 插入结点。

插入结点的过程是:将要插入位置结点的指针成员指向后一个结点,将前一个结点的指针成员指向插入的结点。具体是首先在链表中找到插入的位置并将结点插入链表中,再将新插入结点的指针成员指向插入点位置后的结点。

```

void insert(struct info *mid)/* 在 mid 后插入结点的函数*/
{
    struct info *pnew;
    printf("insert information to list\n");
    pnew=(struct info *)malloc(sizeof(struct info)); /* 构造一个新结点*/
    printf("please input name"); /* 输入新结点的数据 */
    scanf("%s",pnew->name);
    printf("please input score");
    scanf("%d",&pnew->score);
    pnew->pnext=mid->pnext; /*新结点指向后结点*/
    mid->pnext=pnew; /*原插入点指向新结点*/
}

```

(4) 查找符合某个条件的结点。

```

struct info *search(struct info *head)          /*从已有链表中查找信息*/
{

```



```
struct info *p;
int t;
char temp1[10];
printf("please input the name you want to find");
p=head;
gets(temp1);
while((t=strcmp(p->name,temp1))!=0) /*查找信息*/
{
    if(p->pnext==NULL)
        break;
    else
        p=p->pnext;
}
if(t==0) /* 表示找到了想找的信息 */
{
    printf("the information is: %s %d\n",p->name,p->score);
    return(p);
}
else
{
    printf("not found"); /* 显示找不到*/
    return(NULL);
}
}
```

(5) 删除一个结点，即将其前一个结点的指针成员指向它的后一结点。方法是首先找到要删除的结点，然后将删除结点的指针成员值赋给前一结点的指针成员，使其前一结点的指针成员直接指向后一结点。

```
struct info *dele(struct info *phead,struct info *pdel)
/* 删除已找到的结点的函数*/
{
    struct info *temp;
    char y;
    printf("delete information from a list\n");
    printf("the information will be deleted is:
           %s %d\n",pdel->name,pdel->score);
    if(phead==pdel)
        phead=pdel->pnext; /*如果第一个结点是要删除的结点，删除*/
    else
    {
        temp=phead;
        while(temp->pnext!=pdel)
            temp=temp->pnext; /* 找到前相邻的结点 */
        temp->pnext=pdel->pnext; /* 使前结点指向后结点*/
    }
    free(pdel); /* 释放结点 pdel 所占内存空间 */
    return(phead);
}
```

phead 是链首指针, pdel 指向要删除的结点。当删除正常完成时, 返回链首指针的值。

(6) 主函数。

包含以上各个函数的主函数代码如下:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    char m;
    struct info *phead,*p,*pnew;
    p=phead=creat(3); /*调用建立链表的函数建立链表,并将指针指向链首*/
    while(1)/* 显示菜单以供用户选择操作*/
    {
        printf("\n1    search\n");
        printf("2    insert\n");
        printf("3    delete\n");
        printf("4    list\n");
        printf("5    quit\n");
        printf("please select\n");
        scanf("%d",&m);
        if(m==5)exit(0);
        switch(m)
        {
            case 1:p=search(phead);break; /* find element */
            case 2: insert(p);break; /* insert element */
            case 3: phead=dele(phead,p);break; /* delete from a link */
            case 4: list(phead);
        }
    }
}
```

10.6 共用体

共用体是共享存储单元的不同变量的集合体(又称为联合 union), 它的特点是各个成员共享同一存储区域。

10.6.1 共用体的定义

共用体的定义和结构体非常类似, 其一般格式如下:

```
union 共用体名
{
    类型 成员名1;
    类型 成员名2;
    ...
    类型 成员名n;
}[共用体变量名];
```



例如：

```
union un
{
    char c;
    int i;
    float f;
}un1,un2;
```

上述代码定义了一个名为 `un` 的共用体类型，同时还定义了两个 `union un` 类型的变量 `un1` 和 `un2`，它们都有 3 个成员：`c`、`i` 和 `f`。

一般地，共用体的成员除了可以是简单变量外，还可以是数组、共用体或结构等。

10.6.2 共用体变量的访问

对共用体变量的访问也是采用访问成员的方式，共用体成员的引用方式和结构体一样：可以定义指向共用体的指针，利用“.”或“->”运算符可访问共用体的成员。例如可利用以下语句对共用体变量成员进行赋值。

```
un1.i=10;
un1.c='a';
un1.f=25.6;
```

共用体也有自己的地址，用共用体变量名表示。和结构体一样，共用体也不能进行整体赋值、输入和输出等操作，但相同类型的共用体变量可以相互复制。例如：

```
un2=un1;
```

10.6.3 共用体的存储

共用体的各个成员占用同一存储区域，其中占用最多存储单元的成员的 length 就是共用体的 length，共用体的各个成员的地址都是同一地址，即首地址。例如下面代码所定义的共用体变量 `un1` 的存储分配如图 10-5 所示。

```
union un
{
    char c;
    int i;
    float f;
}un1;
```

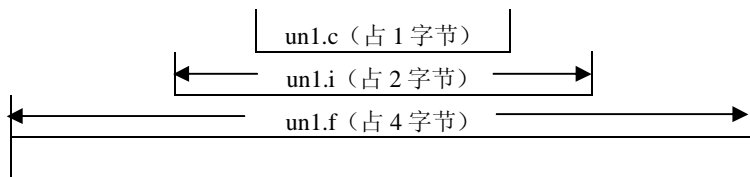


图 10-5 共用体变量 `un1` 存储分配示意图

因此，一个共用体中可能有若干个不同类型的成员，但在每一瞬间只有一个成员起作用。

用。严格地讲，只有最近一次赋值的成员起作用。例如，有下列顺序的赋值语句：

```
un1.i=10;
un1.c='a';
un1.f=25.6;
```

执行后，只有 `un1.f` 是有效的，`un1.i` 和 `un1.c` 均变成无意义，这时就不能获得 `un1.i` 或 `un1.c` 的值。但若存在相同类型的成员，则该成员就变的有意义。假设共用体变量 `un1` 中还有一个成员是 `float x`，则经过上述赋值后，`un1.x` 的值也是 25.6，用户可以引用它。

共用体不能进行初始化，即在定义的同时不能设初始值，只能定义之后才可对成员进行赋值操作。

10.6.4 共用体的应用

程序中使用共用体可以让各个成员共享存储单元，这有如下 3 点主要优点：

- 增加程序的可移植性：由于不同数据类型的长度是依赖于机器的，使用共用体可使不同类型的数据存放在同一片存储单元中，编译程序会记录共用体中各成员的实际大小，这样编程者可以不关心数据类型的长度。
- 增加程序的灵活性：在系统软件中常常需要处理符号表，符号表中包含符号名、类型名和值，此时可以使用结构存放符号表，可以使用共用体将不同类型的值存放在一起。根据共用体的使用规则，只有同类型的数据才能在相应的共用体成员中起作用，也就是能用一个符号名处理所有同类型的数据。
- 进行数据类型转换：使用共用体将不同的数据类型作为自己的成员，就可使单个的变量能合法的拥有几种类型中的任何一种。

例 10.7 判断以下程序的输出结果

```
#include <stdio.h>
main()
{
    union
    {
        unsigned int n;
        unsigned char c;
    }u1;
    u1.c='A';
    printf("%c\n",u1.n);
}
```

因为 `u1.n` 和 `u1.c` 共用存储单元，`u1.n` 低位的内容即是 `u1.c` 的值，所以按照字符格式输出时，输出字符 A。

例 10.8 判断以下程序的输出结果。

```
#include <stdio.h>
main()
{
```




```

union u
{
    int a;
    char b[2];
}c;
c.a=0x1234;
printf("\nc.a=%x  c.b[0]=%x  c.b[1]=%x",c.a,c.b[0],c.b[1]);
printf("\n&c=%x  &c.a=%x  &c.b[0]=%x  &c.b[1]=%x",&c,
        &c.a,&c.b[0],&c.b[1]);
}

```

赋值以后共用存储区的内容如图 10-6 所示。

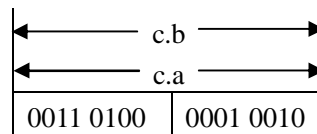


图 10-6 共用存储区示意图

所以输出为:

```

c.a=1234  c.b[0]=34  c.b[1]=12
&c=ffda  &c.a=ffda  &c.b[0]=ffda  &c.b[1]=ffdb

```

注意，地址是由系统自动分配的。

例 10.9 判断以下程序的输出结果。

```

main()
{
    union
    {
        unsigned char c ;
        unsigned int n;
    }u1;
    u1.n=65345;
    printf("\nu1.n=%u,u1.c=%c\n",u1.n ,u1.c);
    printf("\nu1.n=%x,u1.c=%x\n",u1.n ,u1.c);
}

```

赋值以后共用存储区的内容如图 10-7 所示。

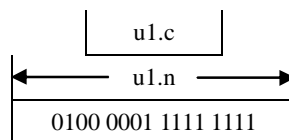


图 10-7 共用存储区示意图

所以输出为:

```

u1.n=65345,u1.c=A
u1.n=ff41,u1.c=41

```

10.7 类型定义

在进行结构体类型的定义或共用体类型的定义时，定义的内容可能比较多，写起来比较繁琐。此时，用户可以用类型定义 `typedef` 将其定义成新的类型标识符，然后新的类型标识符即可当作原标识符使用。

10.7.1 类型定义的形式

类型定义的一般形式为：

```
typedef 已有的类型标识符 新的类型标识符；
```

其中，`typedef` 是类型定义关键字，功能是将已有的类型标识符，如 `int`、`float` 和已定义过的结构体、共用体等，用新的类型标识符来代替。`typedef` 只是对已有的类型名增加一个新的替换名，并不能创造新的类型，也不是取代现有的类型名。

例如，熟悉 FORTRAN 语言的用户可能比较熟悉 `INTEGER`、`REAL`、`CHARACTER` 等类型标识符，于是可以进行如下定义：

```
typedef int INTEGER;
typedef float REAL;
typedef char CHARACTER;
```

定义完成后就可应用新定义的类型标识符去定义变量、数组等：

```
INTEGER i,j;
REAL a,b;
CHARACTER ch1,ch2;
```

它们等价于：

```
int i,j;
float a,b;
char ch1,ch2;
```

一般地，将 `typedef` 定义的类型名用大写字母表示，以便与系统提供的标准类型标识符相区别。

10.7.2 类型定义的使用

`typedef` 除了可以定义简单的数据类型外，还可以定义其他各种已经定义过的类型。

(1) 定义数组。

```
typedef char STRING[80];
STRING s1,s2; /* 等价于 char s1[80],s2[80]; */
```

(2) 定义指针。

```
typedef float *PFLOAT;
PFLOAT p1,p2; /* 等价于 float *p1,*p2; */
```

(3) 定义函数。



```
typedef char FCH();
FCH fa;      /* 等价于 char fa(); */
typedef FCH *PFCH;
PFCH pfb;    /* 等价于 char(*pfb)(); */
```

(4) 定义结构和共用体。

```
typedef struct st
{
    int year;
    int month;
    int day;
}DATE;
DATE birthday,*p; /* DATE 等价于 struct st */
```

对类型定义的说明如下：

(1) `typedef int INTEGER` 与 `#define INTEGER int` 有相似之处，它们都是用 `INTEGER` 代表 `int`，但 `#define` 是在预编译时处理的，它只能作简单的字符串替换，而 `typedef` 是在编译时处理的，并不是作简单的字符串变换。例如前面介绍的定义数组：

```
typedef char STRING[80];
STRING s;
```

以上代码不是将 `STRING[80]` 替换 `char`，而是相当于定义：

```
char s[80];
```

(2) 使用 `typedef` 可以为程序设计带来方便。当不同源文件中用到同一类数据类型，尤其是像数组、指针、结构体、共用体时，常用 `typedef` 定义一些数据类型，并把它们单独放在一个文件中，然后在需要用到它们的文件中用 `#include` 命令把它们包含进来。

(3) 使用 `typedef` 还有利于程序的移植。有时程序会依赖于硬件特性。例如，有的计算机系统 `int` 型数据用 2 个字节，而另外一些机器则用 4 个字节存放一个整数；如果把一个 C 程序从一个以 4 个字节存放整数的计算机系统移植到以 2 个字节存放整数的系统，按一般的办法需要将定义变量中的每一个 `int` 改为 `long`，如果程序中有多处 `int`，则需要依次修改。为了实现一改全改，可以用一个 `typedef` 定义：

```
typedef int INTEGER;
```

在程序中用 `INTEGER` 定义变量。在移植时只需要改动 `typedef` 定义体即可：

```
typedef long INTEGER;
```

要 点 回 顾

1. 结构体是不同数据类型变量的集合体，又称为结构；结构体类型是一种复合数据类型，它可容纳各种数据类型的变量及数组。定义结构体类型的一般格式为：

```
struct 结构体类型名
{
```

```
数据类型 成员 1;  
数据类型 成员 2;  
...  
数据类型 成员 n;  
};
```

不同的结构体类型名代表不同的结构体类型，它们的成员也各不相同。在定义一个结构体类型时，也可以利用已定义的另一个结构体类型来定义其成员的类型。

2. 结构体类型只规定了内存的分配模式，真正代表或存储数据的是结构体变量以及结构体数组。结构体变量定义的一般形式为：

```
struct 结构体名  
{  
    成员表  
};  
struct 结构体名 变量名表列;
```

结构体变量占用内存的情况取决于结构体中各成员占用内存的要求。结构体变量不是一个简单变量，它的值也不是一个简单的整数、实数或字符等，而是由许多基本数据组成的复合值。

3. 结构体变量的初始化与数组的初始化非常相似，只要把对应的常数值放在花括号中（即初值表），各常数值之间用逗号隔开即可。

4. 一般不对结构体变量整体进行操作，而是对其成员进行操作。访问或引用结构体变量的某一个成员（简称结构成员）时，使用运算符“.”将结构成员与结构体变量名联系起来。访问结构成员的一般形式为：

```
结构体变量名.成员名
```

5. 不允许对结构体变量名直接进行赋值、输入和输出等操作，但可以将一个结构体变量作为一个整体赋给另一个具有相同类型的结构体变量。

6. 具有相同结构体类型的一组变量可以组成结构体数组（或简称为结构数组）。定义结构数组和定义结构变量类似，只要在结构变量名的后面加上数组维界说明符即可。

7. 结构数组的初始化是在数组定义的同时将初始数据赋值给数组元素，初始数据应与数组元素的各成员一一对应。为清晰起见，每一个数组元素的初始数据都用花括号括住。

8. 访问结构数组元素的成员与访问结构体变量的成员具有相同的规则。

9. 指向结构体变量或结构数组的指针称为结构指针。指针的值是结构体变量或结构数组元素的地址。定义结构指针和定义一般指针相似，形式为：

```
[存储类别] 结构类型 *指针 1, *指针 2, ...;
```

10. 结构指针必须通过初始化或赋值，把实际存在的某个结构变量或结构数组的首地址赋给它，使它指向该变量或数组。定义了结构指针并确定结构指针指向之后，即可利用结构指针来访问结构成员。形式如下：

```
(*指针名).成员名
```

又可以写成：



指针名->成员名

11. 结构体变量允许用同类型的结构指针作为成员。这种结构称为递归结构，又称自嵌套结构，即结构的成员是具有该结构类型的结构指针。递归结构的定义形式如下：

```
struct 结构类型名
{
    结构成员说明;
    struct 结构类型名 *指针名;
};
```

递归结构常用来处理动态数据结构的问题，用指针和结构体构成链表，可以实现单向链表的建立、输出、插入与删除等操作。

12. 共用体是共享存储单元的不同变量的集合体（又称为联合 **union**），它的特点是各个成员共享同一存储区域。共用体的定义形式和结构非常类似：

```
union 共用体名
{
    类型 成员名 1;
    类型 成员名 2;
    ...
    类型 成员名 n;
}[共用体变量名];
```

对共用体变量的访问也需采用访问成员的方式。共用体的各个成员占用共同的存储区域，其中占用最多存储单元的成员的 length 就是共用体的 length，共用体的各个成员的地址都是同一地址，即首地址。一个共用体中可能有若干个不同类型的成员，但在每一瞬间只有最近一次赋值的成员起作用。

13. 在进行结构体类型的定义或共用体类型的定义时，定义的内容可能比较多，写起来比较繁琐，用户可以用类型定义 **typedef** 将其定义成新的类型标识符。然后，新的类型标识符即可当作原标识符使用。类型定义的一般形式为：

```
typedef 已有的类型标识符 新的类型标识符;
```

习 题

1. 设有如下定义：

```
struct sk
{
    int a;
    float b;
}data;
int *p;
```

若要使 **p** 指向 **data** 中的 **a** 域，正确的赋值语句是（ ）。

A) **p=&a;** B) **p=data.a;** C) **p=&data.a;** D) ***p=data.a**

2. 有以下程序：

```

struct STU
{
    char num[10];
    float score[3];
};
main()
{
    struct STU s[3]={{"20021",90,95,85},{"20022",95,80,75},
                     {"20023",100,95,90}},*p=s;

    int i;
    float sum=0;
    for(i=0;i<3;i++)
        sum=sum+p->score[i];
    printf("%6.2f\n",sum);
}

```

程序运行后的输出结果是 ()。

A) 260.00 B) 270.00 C) 280.00 D) 285.00

3. 有以下程序:

```

#include <stdlib.h>
struct NODE
{
    int num;
    struct NODE *next;
};
main()
{
    struct NODE *p,*q,*r;
    p=(struct NODE *)malloc(sizeof(struct NODE));
    q=(struct NODE *)malloc(sizeof(struct NODE));
    r=(struct NODE *)malloc(sizeof(struct NODE));
    p->num=10;q->num=20;r->num=30;
    p->next=q;q->next=r;
    printf("%d\n",p->num+q->next->num);
}

```

程序运行后的输出结果是 ()。

A) 10 B) 20 C) 30 D) 40

4. 假定建立了以下链表结构, 指针 p、q 分别指向如图 10-8 所示的结点, 则以下可以将 q 所指结点从链表中删除并释放该结点的语句组是 ()。

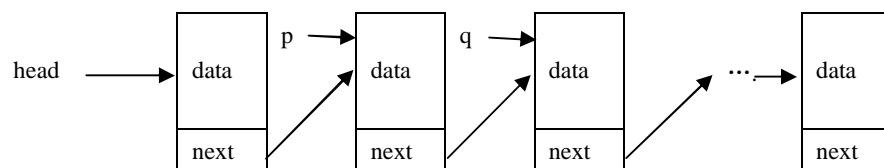


图 10-8 链表结构



- A) free(q); p->next=q->next;
 B) (*p).next=(*q).next; free(q);
 C) q=(*q).next; (*p).next=q; free(q);
 D) q=q->next; p->next=q; p=p->next; free(p);
5. 下面程序的运行结果是 ()。

```
typedef union student
{
    char name[10];
    long sno;
    char sex;
    float score[4];
}STU;
main()
{
    STU a[5];
    printf("%d\n",sizeof(a));
}
```

6. 有以下说明和定义:

```
typedef int *INTEGER;
INTEGER p,*q;
```

- 则以下叙述正确的是 ()。
- A) p 是 int 型变量
 B) p 是指向整型量的指针变量
 C) q 是指向整型量的指针变量
 D) 程序中可用 INTEGER 代替 int 类型名
7. 以下各选项试图说明一种新的类型名, 其中正确的是 ()。
- A) typedef v1 int; B) typedef v2=int;
 C) typedef v1 int v3; D) typedef v4: int;
8. 以下程序段用于建立一个简单的单向链表, 请填空。

```
struct STRU
{
    int x,y ;
    float rate;
    _____p;
}a,b;
a.x=0; a.y=0; a.rate=0; a.p=&b;
b.x=0; b.y=0; b.rate=0; b.p=NULL;
```

9. 若有如下结构体定义, 请填空, 以完成对 t 数组的定义, t 数组的每个元素为该结构体类型。

```
struct STRU
{
```

```
int a,b ;  
char c;  
double d;  
struct STRU p1,p2;  
};  
_____t[20];
```




第 11 章 枚举与位运算

本章的学习目标:

了解枚举类型及其用法; 了解对二进制位进行操作的基本方法和作用; 掌握位操作运算符的表示形式及用法; 掌握利用位运算将二进制数的某些位置 1、清 0 或取反; 了解移位运算的过程; 了解位段的概念及用法; 掌握编写简单的对二进制位进行操作的程序。

11.1 枚 举

枚举类型规定变量只可能取哪几个值。在实际问题中, 有些变量的取值被限定在一个有限的范围内, 例如逻辑量的值只有真和假, 一个星期内只有 7 天, 一年只有 12 个月。C 语言提供了一种“枚举”类型, 在“枚举”类型的定义中列举出所有可能的取值, 被说明为该“枚举”类型的变量取值不能超过定义的范围。应该说明的是, 枚举类型是一种基本数据类型, 而不是一种构造类型, 因为它不能再分解为任何基本类型。

11.1.1 枚举类型的定义

枚举类型定义的一般形式为:

```
enum 枚举名
{
    枚举值表
};
```

在枚举值表中应罗列出所有可能的值, 这些值称为枚举元素。例如:

```
enum weekday
{
    sun, mon, tue, wed, thu, fri, sat
};
```

该枚举名为 `weekday`, 枚举值共有 7 个, 即一周中的 7 天。凡被说明为 `weekday` 类型的变量的取值只能是上述 7 个枚举值中的某一个。

枚举变量的定义, 如同结构体和共用体一样, 共有 3 种定义方式, 即先定义类型后定义变量、定义类型的同时定义变量、直接定义枚举类型。设要将变量 `a`、`b`、`c` 定义为上述的 `weekday` 枚举类型, 可采用下列任一种方式:

```
enum weekday
{
    sun, mon, tue, wed, thu, fri, sat
};
```

```
weekday a,b,c;
```

或者为:

```
enum weekday
{
    sun,mon,tue,wed,thu,fri,sat
}a,b,c;
```

或者为:

```
enum
{
    sun,mon,tue,wed,thu,fri,sat
}a,b,c;
```

11.1.2 枚举类型的应用

枚举类型在使用中有以下规定:

(1) 枚举值是常量,不是变量,不能在程序中用赋值语句对它赋值。例如对枚举 `weekday` 的元素作以下赋值都是错误的:

```
sun=5;
mon=2;
sun=mon;
```

(2) 枚举元素本身由系统定义了一个表示序号的数值:从 0 开始顺序定义为 0, 1, 2…。如在 `weekday` 中, `sun` 值为 0, `mon` 值为 1, `sat` 值为 6。

例 11.1 输出枚举元素的序号。

```
main()
{
    enum weekday
    {
        sun,mon,tue,wed,thu,fri,sat
    }a,b,c;
    a=sun;
    b=mon;
    c=tue;
    printf("%d,%d,%d",a,b,c);
}
```

则输出的枚举元素的序号为: 0, 1, 2。

例 11.2 改变枚举元素的值,在定义时另外指定。

```
#include <stdio.h>
main()
{
    enum team{one,two,three=10,four=three+2,five};
    printf("%d,%d,%d,%d,%d",one,two,three,four,five);
}
```



程序中 three 被重新赋值为 10, four 为 12, five 依次加 1 后为 13。所以输出为: 0, 1, 10, 12, 13。

(3) 只能把枚举值赋予枚举变量, 不能把枚举元素的序号直接赋予枚举变量。如: 语句 “a=sun;b=mon;” 是正确的。而 “a=0; b=1;” 是错误的。如一定要用数值的方法给枚举变量赋值, 则必须用强制类型转换, 例如: “a=(enum weekday)2;” 其意义是将顺序号为 2 的枚举元素赋予枚举变量 a, 相当于: “a=tue;”, 还应该说明的是, 枚举元素不是字符常量, 也不是字符串常量, 使用时不要加单、双引号。

例 11.3 假设某个月有 30 天, 根据键盘输入 1 日是星期几, 然后要求输出每天的日期号及对应的星期。每行输出 7 天, 第 1 列为星期日。

```
#include <stdio.h>
main()
{
    enum body
    {
        sun,mon,tue,wed,thur,fri,sat
    }month[31],j;
    int i,k;
    printf("\nplease input weekday No of the first day: ");
    scanf("%d",&j);
    k=j;
    printf("\n");
    for(i=1;i<=k;i++)
        printf("%7c",0);
    for(i=1;i<=30;i++)
    {
        month[i]=j;
        j++;
        if(j>sat)
            j=sun;
    }
    for(i=1;i<=30;i++)
    {
        switch(month[i])
        {
            case 0:printf("%3d%4s",i,"sun"); break;
            case 1:printf("%3d%4s",i,"mon"); break;
            case 2:printf("%3d%4s",i,"tue"); break;
            case 3:printf("%3d%4s",i,"wed"); break;
            case 4:printf("%3d%4s",i,"thu"); break;
            case 5:printf("%3d%4s",i,"fri"); break;
            case 6:printf("%3d%4s",i,"sat"); break;
            default:break;
        }
        if((i+k)%7==0)
            printf("\n");
    }
}
```

```
printf("\n");  
}
```

设该月第一天是星期五，程序输出结果为：

```
please input weekday No of the first day:5✓  
1 fri 2 sat  
3 sun 4 mon 5 tue 6 wed 7 thu 8 fri 9 sat  
10 sun 11 mon 12 tue 13 wed 14 thu 15 fri 16 sat  
17 sun 18 mon 19 tue 20 wed 21 thu 22 fri 23 sat  
24 sun 25 mon 26 tue 27 wed 28 thu 29 fri 30 sat
```

11.2 位 运 算

C 语言具有类似于汇编语言的实现位操作的功能。前面章节介绍的运算都是以字节作为基本单位进行的，但在有些系统程序中常要求以位（bit）为单位进行运算或处理。例如为了节省内存空间，在有些系统软件中常将多个标志状态简单地组合在一起，存储到一个字节（或字）中。C 语言提供了按位运算的功能，可以实现将标志状态从标志字节中分离出来，这使得 C 语言也能像汇编语言一样用来编写系统程序。所谓位运算，是指对字节或字节内部的二进制位进行测试、设置、移位或逻辑运算。位操作运算符共 6 种，即“~”，“<<”，“>>”，“&”，“^”和“|”，分别表示按位取反、左移位、右移位、按位与、按位异或和按位或，如表 11-1 所示。

表 11-1 位操作运算符

优先级	运算符	功能	运算量	结合性
14	~	按位取反	1	右至左
11	<<	左移位	2	左至右
	>>	右移位	2	左至右
8	&	按位与	2	左至右
7	^	按位异或	2	左至右
6		按位或	2	左至右

位操作运算符只能对整型数（包括 short、long、signed、unsigned）施加运算，不能对 float、double、void、long double 以及更为复杂的类型进行操作。这 6 种运算符大体可分为两大类：按位逻辑运算和移位运算。它们经常应用于设备驱动程序中，如在调制解调器程序中用于屏蔽某些位，实现奇偶校验；在磁盘文件管理中如果希望文件内容不可识别，可用加密子程序将文件中的字符逐位取反；有些 C 编译系统用左移位实现乘 2 运算，用右移位实现除 2 运算，其速度非常快。

11.2.1 按位逻辑运算符

按位逻辑运算符包括“~”、“&”、“^”和“|”，它们对某一个或某一对二进制位进行



操作。假设 p 和 q 分别表示一个二进制位，则按位逻辑运算符的真值表如表 11-2 所示。

表 11-2 按位逻辑运算符的真值表

P	Q	$p \& q$	$p \wedge q$	$p q$	$\sim p$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	0	1	0

1. 按位与 (&)

按位与运算符是最常用的，它对参加运算的两个二进制位进行运算：如果其中有一个位是 0，则结果为 0；两个位均为 1 时，结果才为 1。格式：

$x \& y$

例如，令 $x=23$ ， $y=42$ ，则 $z=x \& y$ 为：

x:	00010111	(操作数 x: 十进制整数 23)
y:	<u>& 00101010</u>	(操作数 y: 十进制整数 42)
z:	00000010	(结果 z: 十进制整数 2)

主要用途：

(1) 清零。若要将某个单元清零，只要将该单元的内容与同样长度的数据零进行与运算即可。

(2) 取一个数的某些位或保留一个数的某些位，其余各位置 0。

例如：一个整数 x 长度为 2 个字节，若想取其中的低位字节，只需将 x 与 $y=0x00ff$ 按位与即可。

x:	00100101	01100011
y:	<u>& 00000000</u>	<u>11111111</u>
z:	00000000	01100011

这里 $z=x \& y$ ，只保留了 x 的低位字节；如果要取 x 的高位字节，则只需将 x 与 $0xff00$ 按位与；如果要取 x 的 2、4、6 位，则只需将 x 与 $0x0054$ 相与即可。

2. 按位或 (|)

按位或和按位与相反，如果两个操作数中有一个是 1，结果就是 1；只有两个操作数都是 0 时，结果才是 0。格式：

$x | y$

例如，令 $x=23$ ， $y=42$ ，则 $z=x | y$ 为：

x:	00010111	(操作数 x: 十进制整数 23)
y:	<u> 00101010</u>	(操作数 y: 十进制整数 42)
z:	00111111	(结果 z: 十进制整数 63)

主要用途：将一个数的某些位置 1，其余各位不变。例如 x 是一个长度为 2 个字节的整数，若想将其中的低位字节置 1，高位字节保持不变，则只需将 x 与 $y=0x00ff$ 按位或即可。

```

x:      00100101  01100011
y:      | 00000000  11111111
z:      00000101  11111111

```

3. 按位异或 (\wedge)

按位异或的运算规则是：参加运算的两个操作数相应的位相同（都是 1 或都是 0），则结果为 0；两个操作数相应的位不同时，结果为 1。格式：

$x \wedge y$

例如，令 $x=23$ ， $y=42$ ，则 $z=x \wedge y$ 为：

```

x:      00010111      (操作数 x: 十进制整数 23)
y:      ^ 00101010      (操作数 y: 十进制整数 42)
z:      00111101      (结果 z: 十进制整数 61)

```

“异或”的规则有时又被简化为相异（位值不同）为 1，相同（位值相同）为 0。

主要用途：

(1) 使特定位的值翻转（即原来为 1 的位翻转变为 0，为 0 的位翻转变为 1），其余位不变。因为 $1 \wedge 1$ 结果为 0，而 $1 \wedge 0$ 又成为 1，所以如果想翻转一个变量中的某一位，可构造一个该位置 1 其余位全置 0 的数与该变量进行异或运算，如要使变量 x 的第 4 位翻转，可构造新值 y （第 4 位为 1 其余各位为 0）与 x 进行异或运算，如下所示。

```

x:      00110101      (0x35)
y:      ^ 00010000      (构造的新值 0x10)
z:      00100101
          ↑
        此位翻转

```

可见， $x \wedge y$ 的值与 x 相比第 4 位改变了，而其余位保持不变。

当然，不仅可以翻转某一位，而且可以使若干位“翻转”。例如：设 $x=00010111$ ，想使其低 4 位翻转，可将 x 与 00001111 进行异或运算。

```

x:      00010111
y:      ^ 00001111
z:      00011000

```

可以看出，结果 z 的低 4 位是数 x 的翻转，而它的高 4 位与 x 的高 4 位一致。使用这种方法时注意，要翻转哪些位就将其与进行异或运算的数中相对应的位置为 1，因为值为 1 的位与 1 进行异或运算的结果为 0，而值为 0 的位与 1 进行异或运算的结果为 1。如果将数与 0 进行异或运算，则保留原值。

(2) 清零。将一个数与其本身进行异或运算，结果为 0。

(3) 如果使 y 两次与 x 进行异或运算，其值仍为原值，即 $y \wedge x \wedge x == y$ ，过程如下所示。

```

y:      00110101      (0x35)

```



```

x:      ^ 00010000      (0x10)
z:      00100101      (0x25)
x:      ^ 00010000      (0x10)
z^x:    00110101      (0x35)

```

4. 按位取反 (~)

按位取反运算比较简单，它是一个单目运算符，运算量写在运算符之后。取反运算符的作用是使一个数据中所有位都取其反值，即 0 变 1，1 变 0，相当于求该数的反码。格式：

```
~x
```

例如，令 $x=23$ ，则 $z=\sim x$ 为：

```

x:      ~ 00010111      (操作数 x: 十进制整数 23)
z:      11101000      (结果 z: 视为无符号数，十进制整数 232)

```

主要用途：间接地构造一个数，以增强程序的可移植性。例如，在 IBM-PC 机中，若想使数 x (x 为 2 个字节的十进制整数 23) 的最低一位为 0，则可以用 $x=x \& 0xfffe$ (二进制数为 111111111111110)。

```

00000000000010111
& 1111111111111110
00000000000010110

```

使得 x 的最后一位变为 0。但若将涉及该运算的 C 源程序移植到用 32 位存放一个整数的计算机系统中，此时整数 x 将用 4 个字节表示，要将最后一位变为 0 就不能用 $x\& 0xfffe$ 了，为了适应 32 位存放一个整数的计算机系统，应使用 $x\& 0xffffffe$ ，若这样改动程序，则其可移植性就差了。

但可以使用： $x=x \& \sim 1$ 。

对以 16 位存放一个整数的计算机系统来说，1 的补码为：

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

~ 1 的补码为：

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

对以 32 位存放一个整数的计算机系统来说， ~ 1 的补码为：

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

这样整个程序不用修改，可移植性就已改善了。



“~”运算符的优先级高于算术运算符、关系运算符、逻辑运算符及其他位运算符。

另外需要指出的是，前面章节中讨论的由关系和逻辑运算符构成的表达式结果只有两个值，即 0 (假) 或 1 (真)，而按位逻辑运算则可产生任意值。以 $x=23$ ， $y=42$ 为例，比较如表 11-2 所示。

表 11-2 逻辑运算和按位运算的比较

逻辑运算	按位运算
$x\&y=1$	$x\&y=2$
$x y=1$	$x y=63$
$!x=0$	$\sim x=232$ （若视为有符号补码，应为-24）

11.2.2 移位运算符

移位运算符“<<”和“>>”是将操作数中的所有二进制位按指定的位数左移和右移。
格式：

```
x<<n
x>>n
```

其中，x 是操作数，可以为变量或表达式；n 必须是正整数，表示要移动的位数。

1. 左移运算符（<<）

左移运算符用来将操作数中的二进制位按指定的位数左移，低位补 0，高位溢出。例如，假设有整型变量 a=25（即二进制数 0000000011001），进行左移位运算 a<<3，结果为 0000000011001000，转化成十进制为 200，如图 11-1 所示。

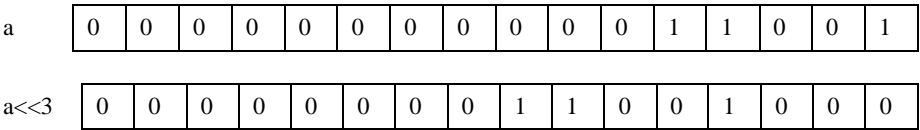


图 11-1 左移示例

左移位相当于乘 2 运算，左移一位相当于该数乘以 2，左移 n 位相当于该数乘以 2ⁿ，该结论只适应于下面两种情况：

- 对补码表示的正数左端移出的全是 0，且移位后的最高位仍为 0。
- 对补码表示的负数左端移出的全是 1，且移位后的最高位仍为 1。

如果不符合上述情况，左移一位就不相当于乘以 2。例如，有符号整型数 64（即二进制数 01000000），左移一位结果是-128，左移两位则结果是 0。这是因为 64 向左移 1 位时，左端移出的是 0，数据的最高位变成了 1，结果为数据-128；而左移 2 位时，左端移出的高位中包含 1，此时结果变为 0。

由于左移比乘法运算快得多，有利于提高程序的执行效率，所以有些 C 编译系统自动将乘 2 的运算用左移来实现。

而对于无符号数，左移 n 位后无溢出时相当于该数乘以 2ⁿ。

2. 右移运算符（>>）

右移运算符用来将操作数中的二进制位按指定的位数右移，移出的低位舍弃；高位的变化如下：



- 对无符号数和有符号数中的正数，高位补 0。
- 有符号数中的负数，取决于所使用的系统。有的左端空位一律补 0，称为“逻辑右移”；有的左端空位全部补 1（即符号扩展），称为“算术右移”。Turbo C 采用的是算术右移。

采用算术右移相当于除 2 运算。右移 1 位，其值等于该数除以 2，右移 n 位，其值等于该数除以 2^n 。例如，假设有变量 $a = -8192$ （补码），右移位运算 $a \gg 2$ ，如果使用“逻辑右移”，表示将 a 的各个二进制位顺序右移 2 位，移出右端之外的位被舍弃，左端空出的位补 0，结果为 14336；如果使用“算术右移”，移出右端之外的位被舍弃，左端空出的位补 1，结果为 -2048。

$a = -8192$ （补码表示）：

A	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

逻辑右移 2 位后，其值为 14336。

$a \gg 2$	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0
-----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

算术右移 2 位后，其值为 -2048。

$a \gg 2$	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
-----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

11.2.3 复合赋值运算符

前面章节中曾介绍过算术运算符与赋值运算符的结合，实际上位运算符也可与赋值运算符结合构成复合的赋值运算符，如： $\&=$ 、 $|=$ 、 $\gg=$ 、 $\ll=$ 、 $\wedge=$ 等。

$a \&= b$ 相当于 $a = a \& b$ ； $a |= 3$ 相当于 $a = a | 3$ ； $a \gg= 3$ 相当于 $a = a \gg 3$ ； $a \ll= 3$ 相当于 $a = a \ll 3$ ； $a \wedge= 3$ 相当于 $a = a \wedge 3$ 。

11.2.4 不同长度的数据进行位运算

若进行位运算的两个数据长度不同，系统会按照类型转换规则将低精度数转换为高精度数再进行运算。

例如 a 为 long 型，而 b 为 int 型，进行 $a \& b$ 运算时，系统将按如下原则对 b 进行符号扩展，将 b 转换为 long 型：

- 若 b 为正数，则左侧 16 位补 0
- 若 b 为负数，则左侧 16 位补 1。

11.3 简单的位运算应用举例

例 11.4 用按位与的方法可以测试一个数的某几位是否为 1。

```
#include <stdio.h>
main()
```

```
{
    unsigned char a,b;
    scanf("%x",&a);
    scanf("%x",&b);
    printf("%02x",a&b);
}
```

a 和 b 都是从键盘输入的十六进制数，a 是用来测试 b 的某一位是否为 1 的数。运算情况如下：

```
10✓      (测 b 的第 4 位是否为 1)
12✓
10
```

这是按位与运算的应用示例。运行时输入 a 的值为十六进制数 10，即二进制 00010000，目的是测试 b 的第 4 位是否为 1；b 的值为十六进制数 12，即二进制数 00010010，a&b 的值为十六进制数 10，与 a 相同。

由于 a 的输入值保持测试位为 1，其余位为 0，如果 a&b 的结果与 a 相同，则说明 a 中为 1 的位，b 中对应位也为 1。上面用的 a 只有一位为 1，实际上可以推广到测试若干位是否为 1。例如要测试 b 的低 4 位是否为 1，可以使 a 的值为 0x0f，如果 a&b 的结果是 0x0f，则说明 b 的低 4 位全为 1。

例 11.5 编程将 a 的高 4 位与 b 的低 4 位合成一个新值输出。

```
#include <stdio.h>
main()
{
    unsigned char a,b;
    printf("\nenter two hex numbers:");
    scanf("%x",&a);
    scanf("%x",&b);
    a=a&0xf0;
    b=b&0x0f;
    printf("%02x",a|b);
}
```

或运算可用来组合两个变量中的某些位，使其成为一个新值，方法是：将不被包含到新值去的那些位变为 0，然后进行或运算即可。

例 11.6 取反运算。

```
#include <stdio.h>
main()
{
    unsigned char x; /*x 定义为字符型，只占 1 个字节*/
    printf("\nenter a hex number:");
    scanf("%x",&x);
    printf("%02x",~x);
}
```



这是按位取反运算的应用。运行情况如下：

```
enter a hex number: 13
ec
```

若把变量 `x` 定义为整型变量 (`unsigned int`)，则可以对 16 个二进制位进行取反运算，此时运行情况如下：

```
enter a hex number: 13
ffec
```

例 11.7 从键盘上输入一个正整数为 `int` 变量 `num` 赋值，并按二进制位输出该数。

```
#include <stdio.h>
main()
{
    int num,mask,i;
    printf("Input a integer number: ");
    scanf("%d",&num);
    mask = 1<<15; /*构造一个最高位为 1、其余各位为 0 的整数(屏蔽字)*/
    printf("%d=",num);
    for(i=1; i<=16; i++)
    {
        putchar(num&mask ? '1' : '0'); /*输出最高位的值(1 或者 0)*/
        num <<= 1; /*左移一位，将次高位移到最高位上以便输出*/
        if( i%4==0 )
            putchar(' '); /*4 位一组，用空格分开*/
    }
    printf("\n");
}
```

该程序通过 `for` 循环利用与运算 (`num & mask`) 将数据 `num` 一位一位地输出。

例 11.8 从键盘上输入一个正整数给 `int` 型变量 `num`，构造一个由 `num` 的 3~6 位构成的数。

分析：这是利用与运算来取一个数中的某几位的例子，若取 `a` 的低 4 位，则用 `0x0f&a` 即可，若取低 5 位，则用 `0x1f & a` 即可。因此可按如下步骤编程：

(1) 使变量 `num` 右移 3 位，将 3~6 位移到低 4 位上。

(2) `num` 与整数 `0x0f` 进行按位与运算。

具体程序如下：

```
#include <stdio.h>
main()
{
    int num,result;
    printf("\nEnter a integer number: ");
    scanf("%d",&num);
    num >>=3; /*右移 3 位，将 3~6 位移到低 4 位上*/
    result =num&0x0f;
    printf("result=0x%04x\n",result);
}
```

```
}
```

请读者进一步思考, 是否可以使用“ $\sim(\sim 0 \ll 4)$ ”间接构造一个低 4 位为 1、其余各位为 0 的整数? 答案是肯定的。因此语句“`result = num & 0x0f;`”可以改写为“`result = num & ~(~0 << 4);`”, 这种取一个数中的某几位的方法称为“屏蔽方法”, 即用 0 屏蔽一个数中的某些位, 用 1 保留剩余的位。

11.4 位 段

在计算机用于数据采集、过程控制等领域时, 控制或状态信息往往只占一个或几个二进制位, 此时可以用一个字节存放多个信息, 从而节约存储空间。对二进制位的操作可以用位运算符, 也可以采用位段的方法访问字节中的某些位。

位段结构方式的操作实际上是仿照结构变量中对其成员进行操作的方式来对位段进行操作。位段是一种特殊的结构体中的成员, 这种成员的长度以位为单位。位段结构的定义和操作类似于结构体变量。

位段结构定义的一般形式如下:

```
struct [位段结构名]
{
    unsigned 位字段名 1: 表达式 1;
    unsigned 位字段名 2: 表达式 2;
    unsigned 位字段名 3: 表达式 3;
    ...
    unsigned 位字段名 n: 表达式 n;
}位段结构变量名;
```

其中“位段结构名”为可选项, 可以省略; 表达式为常量表达式, 表示位段占用的二进制位数。

例如, CPU 的状态寄存器, 按位段类型定义如下:

```
struct status
{
    unsigned sign:      1; /*符号标志*/
    unsigned zero:      1; /*零标志*/
    unsigned carry:     1; /*进位标志*/
    unsigned parity:    1; /*奇偶/溢出标志*/
    unsigned half_carry: 1; /*半进位标志*/
    unsigned negative:  1; /*减标志*/
}flag;
```

显然, 对 CPU 的状态寄存器而言, 使用位段类型仅需 1 个字节。

也可以先定义一个位段结构类型, 然后再定义位段结构变量, 如果只用位段结构类型定义一个位段结构变量, 可以省略位段结构名。

对位段定义的说明如下:

(1) 因为位段类型是一种结构类型, 所以位段类型的定义、位段变量的定义以及对位



段（即位段类型中的成员）的引用，均与结构类型和结构变量类似。例如：

```
flag.zero  
flag.carry  
if(flag.zero==0)
```

如果使用指针变量，则：

```
struct status flag, *p;  
p=&flag;  
p->zero  
p->carry
```

(2) 对位段赋值时，要注意取置范围。一般地说，长度为 n 的位段，其取值范围是： $0 \sim (2^n - 1)$ 。例如：

```
struct exam  
{  
    unsigned a: 5;  
    unsigned b: 2;  
}data;
```

其中，位段 b 只有两位，它的最大取值为 $2^2 - 1 = 4 - 1 = 3$ ，若引用时写： $data.b = 8$ ，则超出了 b 的定义范围，此时系统自动取赋予该数据的低位。由于 8 的二进制数形式为 1000，最低两位为 00，因此 $data.b$ 最后取得的值为 0。

(3) 位段可以不命名，此时称为无名位段。无名位段只有冒号和长度，用它来填充特殊字段，以便使位字段结构变量的空间与 int 所占用的空间一致。

例如，有如下定义：

```
struct exam  
{  
    unsigned a: 5;  
    unsigned b: 2;  
    : 9; /*无名字段*/  
};
```

而使用长度为 0 的无名位段，可使其后续位段从下一个存储单元开始存储。例如：

```
struct status  
{  
    unsigned sign:1;  
    unsigned zero: 1;  
    unsigned carry:1;  
    unsigned :0; /*长度为 0 的无名位段*/  
    unsigned parity:1;  
    unsigned half_carry: 1;  
    unsigned negative: 1;  
}flag;
```

原本 6 个标志位是连续存储在一个单元中的。由于加入了一个长度为 0 的无名位段，所以其后的 3 个位段，从下一个单元开始存储，一共占用两个存储单元。

(4) 一个位段必须存储在同一个存储单元中, 不能跨两个单元。如果本单元空间不能容纳某位段, 则该空间不用, 从下一个单元起开始存储该位段。例如:

```
struct exam
{
    unsigned a:3;
    unsigned b:2;
    int i;
}data;
```

其中 a 占 3 位, b 占 2 位, a 和 b 共占不到一个字节, i 为整型占 16 位, a 和 b 所占的单元还剩下 3 位, 不能容纳 i, 此时这 3 位将空余不用, i 从下一个单元开始存放。

(5) 可以用 %d、%x、%u 和 %o 等格式字符, 以整数形式输出位段。例如:

```
Printf("%d,%d",flag.zero,flag.carry);
```

(6) 在数值表达式中引用位段时, 系统自动将位段转换为整型数。例如:

```
flag.zero+flag.carry
```

(7) 位段不能是数组, 对位段不能进行取地址运算 (&)。

要 点 回 顾

1. 枚举类型规定变量只可能取哪几个值, 枚举类型定义的一般形式为:

```
enum 枚举名
{
    枚举值表
};
```

在枚举值表中应罗列出所有可能的值, 这些值也称为枚举元素。

2. 枚举变量的定义, 可用已定义过的枚举类型定义枚举变量。

3. 枚举类型在使用中有以下规定:

(1) 枚举值是常量, 不是变量, 不能在程序中再对它赋值。

(2) 枚举元素本身由系统定义了一个表示序号的数值, 从 0 开始顺序定义为 0, 1, 2...

(3) 只能把枚举值赋予枚举变量, 不能把枚举元素的序号数值直接赋予枚举变量。

4. 位运算是指对字节或字节内部的二进制位进行测试、设置、移位或逻辑运算。位操作运算符共 6 种, 即 “~”, “<<”, “>>”, “&”, “^” 和 “|”, 分别表示按位取反、左移位、右移位、按位与、按位异或和按位或。位操作运算符只能对整型数 (包括 short、long、signed、unsigned) 施加运算。

5. 按位与运算符 (&) 是最常用的, 它对参加运算的两个二进制位进行运算: 如果其中有一个位是 0, 则结果为 0; 两个位均为 1 时, 结果才为 1。格式: x&y。

6. 按位或 (|) 和按位与相反, 如果两个操作数中有一个是 1, 结果就是 1; 只有两个操作数都是 0 时, 结果才是 0。格式: x|y。

7. 按位异或 (^) 的运算规则是: 参加运算的两个操作数相应的位相同 (都是 1 或都



是 0)，则结果为 0；两个操作数相应的位不同时，结果为 1。格式： $x \wedge y$ 。

8. 按位取反（~）是一个单目运算符，运算量写在运算符之后。取反运算符的作用是使一个数据中所有位都取其反值，即 0 变 1，1 变 0。格式：~x。

9. 移位运算符（<<和>>）是将操作数中的所有二进制位按指定的位数左移和右移。格式：

```
x<<n  
x>>n
```

其中，x 是操作数，可以为变量或表达式；n 必须是正整数，表示要移动的位数。

10. 位运算符也可与赋值运算符结合构成复合的赋值运算符，如：&=、|=、>>=、<<=、^=等。

11. 对二进制位的操作可以用位运算符，也可以采用位段的方法访问字节中的某些位。位段结构方式的操作实际上是仿照结构变量中对其成员进行操作的方式来对位段进行操作。位段是一种特殊的结构体中的成员，这种成员的长度以位为单位。位段结构的定义和操作类似于结构体变量。

习 题

1. 判断下面程序的输出结果。

```
#include<stdio.h>  
main()  
{  
    enum as{sat=5,sun};  
    char *w[]={ "sun", "mon", "tus", "wed", "thu", "fri", "sat"};  
    printf("%s%s\n",w[sat],w[sun]);  
}
```

2. 编写一函数，对一个 16 位的二进制数取出它的偶数位（即 0，2，4，…，14）。

3. 编写函数实现左右循环移位。函数名为 rotate(value,n)，其中 value 为要循环移位的数据，n 为所移的位数。若 n 为正数表示右移，若 n 为负数表示左移。如 n=5，表示右移 5 位；n=-2，表示左移 2 位。

4. 编写程序实现从一个 16 位的单元中取出从第 4 位开始、第 7 位结束的几位组成的一个新值（即该几位取出后保持原值，其余位为 0）。

5. 分析程序的输出结果。

(1)

```
main()  
{  
    int p,y=0,x=0;  
    p=x<<8|~y>>8;  
    printf("%d",p);  
    p+=(p+=2);  
    printf("%d\n",p);  
}
```

```
}
```

(2)

```
main()
{
    int x=3,y=2,z=1;
    printf("s=%d\n", "x/y&z", x/y&z);
    printf("s=%d\n", "x^y&~z", x^y&~z);
}
```




第 12 章 文件操作

本章的学习目标:

了解缓冲文件系统和非缓冲文件系统的概念; 了解文本文件和二进制文件的特点; 了解文件类型的概念及文件类型指针的定义方法; 掌握 C 语言中对文件操作的方法; 掌握文件指针和文件的关系; 掌握文件指针和文件的读写位置指针的区别; 掌握基本的文件操作函数; 掌握利用文件操作函数对文本文件和二进制文件进行读写的操作。

12.1 文件概述

文件是程序设计中的一个重要概念, 是实现程序和数据分离的重要方式。从广义上讲, 文件是存储在外部介质上的数据或信息的集合, 它可以是一段程序、一段文字或一批数据。文件有很多种类型, 如文本文件、图形文件、声音文件、可执行文件等。按照文件中数据的存储方式, 文件可分为文本文件和二进制文件。

- 文本文件是由字符组成的文件, 通常文件中存储的是字符的 ASCII 码, 存取方便。程序源文件就是文本文件。
- 二进制文件是指数据按二进制形式组成的文件, 该文件节省存储空间, 存取速度快, 但是存取复杂。C 语言中对这两类文件的操作采用不同的方法。

操作系统对外部介质上的信息是以文件为单位进行管理的, 用户只要给出文件名, 就可以对文件进行存取。C 语言中的文件操作是通过标准库函数来实现的, 常用的文件操作库函数列于表 12-1 中, 本章主要介绍如何使用文件操作函数来存取用户自己的数据文件。

表 12-1 常用文件操作库函数

函数名	函数原型说明	功能	返回值
clearerr	void clearerr(FILE *fp)	清除与文件指针 fp 有关的所有出错信息	无
Fclose	int fclose(FILE *fp)	关闭 fp 所指的文件	出错返回非 0, 否则返回 0
Feof	int feof(FILE *fp)	检查文件是否结束	遇文件结束返回非 0, 否则返回 0
Fgetc	int fgetc(FILE *fp)	从 fp 所指文件中读取一个字符	出错返回 EOF, 否则返回所读字符数
Fgets	char fgets(char *str,int num,file *fp)	从 fp 所指文件中读取一个长度为 num-1 的字符串, 存入 str 中	返回 str 地址, 若文件结束返回 NULL

(续表)

函数名	函数原型说明	功能	返回值
Fopen	FILE *fopen(const char *fname,const char *mode)	以 mode 方式打开文件 fname	成功时返回文件指针, 否则返回 NULL
fprintf	int fprintf(FILE *fp,const char *format,arg-list)	将 arg-list 的值按 format 指定的格式写入 fp 所指文件中	返回实际输出的字符数
Fputc	int fputc(int ch,FILE *fp)	将 ch 中的字符写入 fp 所指文件	成功时返回该字符, 否则返回 EOF
Fputs	int fputs(const char *str,FILE *fp)	将 str 中的字符串写入 fp 所指文件中	成功时返回 0, 否则返回非 0
fread	size_t fread(void buf,size_t size,size_t count,FILE *fp)	从 fp 所指文件读取长度为 size 的 count 个数据项, 写入 fp 所指文件中	返回读取的数据项个数
fscanf	int fscanf(FILE *fp,const char *format,arg-list)	从 fp 所指文件按 format 指定的格式读取数据存入 arg-list 中	返回读取的数据个数, 出错或遇文件结束返回 0
fseek	int fseek(FILE *fp,long offset,int origin)	移动 fp 所指文件的读写位置指针	成功时返回当前位置, 否则返回 -1
ftell	long ftell(FILE *fp)	求出 fp 所指文件的当前读写位置	读写位置
fwrite	size_t fwrite(const void *buf,size_t size,size_t count,FILE *fp)	将 buf 所指内存区中的 count*size 个字节写入 fp 所指文件中	写入的数据项个数

C 语言中使用以下两种文件系统:

- 缓冲文件系统, 又称为标准文件系统或高级文件系统, 是一种高效、方便且常用的文件系统, 与具体机器无关, 通用性好, 功能强, 使用方便。在缓冲文件系统中使用文件指针来标识文件。后面要学习的文件操作函数都是基于缓冲文件系统的。
- 非标准缓冲文件系统, 又称为低级文件系统, 该系统与机器有关, 节省内存, 执行效率高, 但是应用难度大。

12.2 文件的打开与关闭

12.2.1 文件指针 (FILE 类型指针)

C 语言通过文件指针对文件进行操作。当文件指针与某个文件连接之后, 用户就可以通过文件指针而不是通过文件名来存取文件。文件指针是一种结构体类型指针, C 编译系统已将这一结构体定义好, 并命名为 **FILE**。**FILE** 是一个特殊的结构体类型标识符, 该结构体类型的成员可以表示文件名、文件状态标志、文件读写位置指针及缓冲区大小等信息。结构体类型 **FILE** 在标题文件 **stdio.h** 中被定义, 内容如下:

```
typedef struct
{
```



```
short level;          /*缓冲区满或空的程度 fill/empty level of buffer */
unsigned flags;       /* 文件状态标志 File status flags */
char fd;              /* 文件描述符 File descriptor */
unsigned char hold;   /* 若无缓冲区不读取字符 Ungetc char if no buffer */
short bsize;          /* 缓冲区大小 Buffer size */
unsigned char *buffer; /* 数据缓冲区 Data transfer buffer */
unsigned char *curp;   /* 当前读写位置指针 Current active pointer */
unsigned istemp;       /* 临时文件指示器 Temporary file indicator */
short token;          /* 用于有效性检查 Used for validity checking */
}FILE;
```

对此结构体类型成员的含义读者可以不去深入探讨。

因此,在使用文件前,要包含标题文件 `stdio.h`,还要定义 `FILE` 型的文件指针。定义文件指针的格式如下:

```
FILE *fp;
```

相当于:

```
struct
{
    ...
}*fp;
```

这样, `fp` 即成为指向 `FILE` 类型结构体的指针变量(简称文件指针),它可以通过 `fopen()` 函数连接到指定的文件上(称为指向文件)。实际上 `fp` 指向的是一个 `FILE` 型的结构体,通过结构体中的信息可以访问对应的文件。

一般而言,一个文件指针指向一个文件。因此有几个文件就应该有几个文件指针,不允许一个文件指针同时指向两个或两个以上的文件,也不允许几个指针指向同一文件。

C 语言中,把所有的外部设备都当作文件(称为设备文件),通常从键盘输入数据,而把数据输出到显示器,所以把键盘作为标准输入设备文件,把显示器作为标准输出设备文件。标准输入、输出设备文件的文件指针由系统命名为 `stdin` 和 `stdout`,在编写程序时可以直接使用这两个指针。

12.2.2 文件的打开

打开文件要使用库函数 `fopen()`,其调用的一般格式为:

```
fp=fopen(fname,mode);
```

其中, `fp` 是已经定义过的 `FILE` 型指针; `fname` 是要打开的文件名,是一个字符串常数或字符型数组,文件名可以带路径; `mode` 表示文件的读写方式。文件读写方式及其含义如表 12-2 所示。

表 12-2 文件读写方式及含义

读写方式	含义
r	打开文本文件，只读
w	建立文本文件，只写
a	打开文本文件，追加
rb	打开二进制文件，只读
wb	建立二进制文件，只写
ab	打开二进制文件，读写
r+	打开文本文件，读写
w+	建立文本文件，读写
a+	打开或建立文本文件，读写
r+b	打开二进制文件，读写
w+b	建立二进制文件，读写
a+b	打开或建立二进制文件，读写
rt	打开文本文件，只读
wt	建立文本文件，只写
at	打开文本文件，追加
r+t	打开文本文件，读写
w+t	建立文本文件，读写
a+t	打开或建立文本文件，读写

对表 12-2 中的读写方式说明如下。

(1) **w 方式（写方式）**：该方式只能用于向打开的文本文件写入数据。若文件不存在，则按用户指定的名字创建新文件；若文件已存在，则先删除文件中的全部内容。文件打开时，文件读写位置指针指向文件头，从文件头开始向文件写入数据。

(2) **r 方式（读方式）**：该方式只能用于打开一个已存在的文本文件并从中读出数据。文件打开时，文件读写位置指针指向文件头，读操作从文件读写位置指针所指处开始。当文件不存在时，出现错误信息。

(3) **a 方式**：该方式用于向文本文件末尾添加数据。若文件存在，则将它打开，并将文件读写位置指针指向文件末尾；若文件不存在，则创建一个新文件，这时文件读写位置指针既是文件头，也是文件尾。追加操作也是从读写位置指针所指处开始。

(4) **r+、w+、a+ 方式**：这 3 种方式针对已存在的文本文件，打开文件后，既可以读，也可以写。它们的区别有如下几点。

- **w+**：用该方式打开文件后，如文件有内容则以覆盖方式写入，即写入的内容覆盖原文件中的内容。



- **r+**: 用该方式打开文件后, 文件原有内容全部丢失, 这时只能先向文件写入数据, 然后再读出。
- **a+**: 用该方式打开文件后, 将文件内容保留。读时从文件开头读, 写时则追加到文件末尾。

(5) 上述 6 种方式加上字母 “t” 后仍表示对文本文件进行打开操作。

(6) 上述 6 种方式加上字母 “b” 后就表示对二进制文件进行打开操作。这时, 可以以数据项为单位进行文件的读写。

文件正常打开后, `fopen()` 函数的返回值传给 `fp`, 即 `fp` 指向文件 `fname` (其实是指向和 `fname` 文件对应的 `FILE` 型结构体)。此后对文件的操作就通过文件指针进行, 而不再使用文件名 `fname`。如果不能打开指定的文件, 则返回 `NULL`。打开和建立一个文件的程序示例如下:

```
#include<stdio.h>
FILE *fp;
main()
{
    ...
    fp=fopen("a:data.txt","w");
    ...
}
```

上述代码的功能是如果 A 盘上有该文件存在, 则刷新文件内容, 可以写入新数据; 如果文件不存在, 则建立以 `data.txt` 为文件名的新文件, 等待写入数据。

为确保文件操作的正常进行, 有必要在程序中检测文件打开操作是否成功。常用下面的程序段来检测:

```
if((fp=fopen("fname","w"))==NULL)
{
    printf("cannot open file");
    exit(1);
}
```

即文件不能正常打开时, 屏幕提示 “cannot open file”; `exit()` 函数的功能是终止程序运行, 关闭文件并返回操作系统。若文件成功打开, 程序就可以继续向下运行。

12.2.3 文件的关闭

文件使用完毕后, 必须用 `fclose()` 函数来关闭文件, 以保证本次文件操作有效。使用方式为:

```
fclose(fp);
```

`fp` 是由 `fopen()` 函数打开文件时使用的文件指针。该函数执行成功时, 返回 0 值; 否则, 返回非 0 值。文件关闭后, 文件指针与文件名脱钩, 文件指针可重用, 即可以用 `fopen` 函数使它再次指向某个文件。文件打开和关闭的程序示例如下:

```
#include<stdio.h>
```

```
FILE *fp1,*fp2;
main()
{
    ...
    fp1=fopen("fname1","w");
    ...
    fclose(fp1);
    fp2=fopen("fname2","r+");
    ...
    fclose(fp2);
    ...
}
```

12.3 文件检测函数

12.3.1 检测文件末尾函数 feof()

feof()用来检测文件读写位置指针是否已到文件末尾，以便能正确地进行文件的存取。

在文本文件中，C 编译系统规定 EOF 为文件结束标志，值为-1。由于任何 ASCII 码都不可能取负值，所以它不会在文件中产生冲突。但是在二进制文件中，-1 有可能是一个有效的数据，用 EOF 作为文件的结束标志就不太合适。因此，C 编译系统定义 feof()函数作为二进制文件的结束标志。该函数也可作为文本文件的结束标志。

feof()的调用格式为：

```
feof(fp);
```

fp 是文件类型指针，如果文件读写位置指针已到文件末尾，函数返回非 0 值；否则，函数返回 0 值。例如：

```
while(!feof(fp))
    getc(fp);
```

可利用上述代码循环读取字符直到文件末尾。

所谓文件读写位置指针是文件指针，指向的文件型结构的一个成员。注意：文件指针本身并不指向文件数据流中的单个字符，这是许多初学者容易混淆的概念。

12.3.2 检测出错函数 ferror()

该函数可用来检测输入输出函数的每次调用是否有错，函数的调用格式是：

```
ferror(fp);
```

正常时函数返回 0 值，出错时函数返回非 0 值。一般在调用输入输出函数后立即调用该函数，以检查输入输出函数的引用是否正确。例如：

```
c=fgetc(fp);
if(ferror(fp))
    printf("I/O error! \n");
```



12.4 文件读写函数

文件的存取，即文件的读写。一个文件打开后，就可以对文件进行读写操作。C 语言支持文件读写操作的常用库函数已列于表 12-1 中。

12.4.1 字符读写函数

`getc()`、`fgetc()`和 `putc()`、`fputc()`这两组字符读写函数用于从文件中读出一个字符，或把一个字符写入文件。它们与函数 `getchar()`和 `putchar()`的不同之处在于，前者是针对文件操作的，而后者是针对标准输入输出设备操作的。

1. `fputc()`和 `putc()`

这两个函数用来将一个字符写入文件。它们的功能和使用方法完全相同，差别在于前者是函数原型，后者是由前者定义的宏。它们的调用格式为：

```
fputc(ch,fp);  
putc(ch,fp);
```

其中，`ch` 是字符常数或字符型数组变量，`fp` 是文件指针。该语句的功能是把字符型数组 `ch` 中的字符串写入到 `fp` 所指向的文件中：如果函数执行成功，则返回所写入的字符；否则，返回 EOF。EOF 是 C 编译系统在标题文件 `stdio.h` 中定义的文件结束标志，其值为 -1。

2. `fgetc()`和 `getc()`

`fgetc()`和 `getc()`用于从指定文件中读出一个字符，它们的关系与 `fputc()`和 `putc()`的关系相同。其调用格式为：

```
c=fgetc(fp);  
c=getc(fp);
```

该语句的功能是从 `fp` 所指向的文件中读出一个字符并赋给变量 `c`。如果读取正确，返回读取的字符；否则，当读取错误或遇到文件结束标志 EOF 时，返回 EOF。

例 12.1 按字符方式读写文本文件的实例。

```
#include <stdio.h>  
main()  
{  
    FILE *fpi;  
    int c;  
    fpi=fopen("text1","r"); /*以读方式打开文件，并用 fpi 指向文件*/  
    while(!feof(fpi))  
    {  
        c=fgetc(fpi); /*逐个读入文件中的字符*/  
        fputc(c,stdout); /* 输出到标准输出设备，也可以写成 putchar(c); */  
    }  
    fclose(fpi);  
}
```

```
}
```

该程序可以显示与程序文件在同一个目录下的文件 `text1` 中的字符。

12.4.2 字符串读写函数

函数 `fgets()` 和 `fputs()` 用于从指定文件中读出一个字符串或把一个字符串写入指定的文件。

1. fgets()

函数 `fgets()` 可以从指定文件中读入一行以 '\0' 或 EOF 结尾的字符串并赋给字符型数组。其调用格式为：

```
fgets(s,n,fp);
```

其中, `s` 是字符型数组名, 用字符型数组存放读进来的字符串; `n` 是指定读入的字符个数 (注意该语句最多只能读取 `n-1` 个字符), 并将字符串存入字符型数组 `s` 中。函数读取字符直至达到指定的字符个数、接收到换行符或遇到文件结束标志 EOF 为止。如果执行成功, 返回的函数值为字符型数组的地址; 否则返回空值, 这时 `s` 中的内容不定。

2. fputs()

函数 `fputs()` 可以将一个字符串写入指定的文件。它的调用格式为：

```
fputs(s,fp);
```

其中, `s` 是一个字符型数组名或指向字符串的指针, 表示要写入文件的字符串。如果函数执行成功, 则返回一个换行符; 否则, 返回 EOF。

例 12.2 对文本文件读写字符串的实例: 把一个文本文件的内容读出并存入另一个文本文件中。

```
#include <stdio.h>
main()
{
    FILE *fp1,*fp2;
    char string[80];
    if((fp1=fopen("file1.txt","r"))==NULL)
    {
        printf("cannot open file1");exit(1);
    }
    if((fp2=fopen("file2.txt","w"))==NULL)
    {
        printf("cannot open file2");exit(2);
    }
    while(!feof(fp1))                /*如果不到文件尾, 继续读写下一行*/
    {
        fgets(string,79,fp1);
        fputs(string,fp2);
    }
}
```




```
printf("\nsuccess\n");  
fclose(fp1);  
fclose(fp2);  
}
```

程序执行完成后,可以得到一个和 file1.txt 内容相同的文本文件。file1.txt 和 file2.txt 都与本程序的可执行文件在同一目录下。

12.4.3 文件格式读写函数

函数 fscanf()和 fprintf()可以实现文件的格式读写。它们与 scanf()和 printf()的不同之处在于, fscanf()和 fprintf()函数只是对文件进行读写,而不是对标准输入输出设备进行读写。

1. fprintf()

函数 fprintf()的使用格式为:

```
fprintf(fp,format,arg1,arg2,...,argn);
```

其功能是按转换控制字符串 format 给定的格式,将输出项 arg1, arg2, ..., argn 的值写入 fp 所指向的文件中。函数如果执行成功,则返回实际写入文件的字符个数;若出现错误,则返回负数。

例 12.3 从键盘输入 N 个学生的学号、姓名和成绩,并按一定格式存入文件 score.txt 中。

```
#include<stdio.h>  
#define N 4  
typedef struct student  
{  
    int number;  
    char name[8];  
    int score;  
}st;  
st stu[N];  
main()  
{  
    FILE *fp;  
    int j,k,p; float t;  
    /*键盘输入学号、姓名和分数*/  
    for(j=0;j<N;j++)  
    {  
        printf("please input number name score:");  
        scanf("%d %s %d",&stu[j].number,stu[j].name,&stu[j].score);  
    }  
    /*将学号、姓名和分数写入文件 score.txt*/  
    if((fp=fopen("score.txt","w+"))==NULL)  
    {  
        printf("can't open file");  
        exit();  
    }
```

```
}
for(j=0;j<N;j++)
{
    fprintf(fp,"%6d%8s%6d\n",stu[j].number,stu[j].name,stu[j].score);
}
fclose(fp);
}
```

文件中学号用“%6d”的格式输出,姓名用“%8s”的格式输出,分数用“%6d”的格式输出,数据右对齐。根据某次输入数据,score.txt文件的内容及格式如下:

```
1  zhang  99
2  wang   98
3  zhao   97
4  liu    96
```

2. fscanf()

函数 fscanf() 的用法为:

```
fscanf(fp,format,arg1,arg2,...,argn);
```

其功能是从文件指针 fp 所指的文件中读取数据,按转换控制字符串 format 所给定的格式赋给输入项 arg1, arg2, ..., argn。如果该函数执行成功,返回输入项的个数;如果遇到文件末尾,返回 EOF;如果赋值失败,返回 0。

例 12.4 将上例中的文件 score.txt 中的数据读入到结构体数组中,再对成绩求平均分,并将平均分写在原数据文件的末尾。

```
#include<stdio.h>
#define N 4
typedef struct student
{
    int number;
    char name[8];
    int score;
}ST;
ST stu[N];
main()
{
    FILE *fp;
    int j,k,p;
    float aver;
    /*从 score.txt 读入学号、姓名和成绩*/
    if((fp=fopen("score.txt","rt"))==NULL)/*将文件以只读方式打开*/
    {
        printf("can't open file");
        exit(0);
    }
    for(j=0;j<N;j++)/*读入数据到结构数组*/
    {
```



```
fscanf(fp,"%6d %8s %6d\n",&stu[j].number,
      stu[j].name,&stu[j].score);
}
/*求分数平均值*/
for(j=0;j<N;j++)
    aver+=stu[j].score;
aver/=N;
/*将文件以追加方式再次打开，将平均分写到文件末尾*/
if((fp=fopen("score.txt","at"))==NULL)
{
    printf("can't open file");
    exit();
}
fprintf(fp,"\\naverage=%.2f",aver);
fclose(fp);
}
```

程序运行后文件内容如下：

```
1  zhang      99
2  wang       98
3  zhao       97
4  liu        96
average=97.50
```

在使用 `fscanf()` 和 `fprintf()` 时，如果是对标准输入输出设备进行读写，有下面的关系：

语句 “`fscanf(stdin,"格式符",输入项);`” 等价于 “`scanf("格式符",输入项);`”。

语句 “`fprintf(stdout,"格式符",输出项);`” 等价于 “`printf("格式符",输出项);`”。

利用 `fprintf()` 和 `fscanf()` 函数读写文件非常方便，容易理解，但效率不高。其原因是，`fscanf()` 读取的只是字符串，需将其转换成二进制才能存入内存；输出时，`fprintf()` 又需将二进制数转换为字符串才能输出到文件。文件的读、写都需转换，故花费时间较多。在要求速度的情况下，最好采用 `fread()` 和 `fwrite()` 函数。

12.4.4 数据块读写函数

可利用函数 `fread()` 和 `fwrite()` 对文件进行数据块的读写操作，一次可以读写一组数据。

1. `fread()`

`fread()` 用于从文件中读出一个数据块，它的调用格式为：

```
fread(buf,size,count,fp);
```

其中，`buf` 是指向数组的指针，数组用于存放读入的数据；`size` 是无符号整型量，用于计算要读入的每个数据项的长度（字节数）；`count` 是整型量，用于指定数据项的个数。该函数的功能是从 `fp` 所指的文件中，一次读出长度为 `size` 字节的 `count` 个数据项，然后存放在 `buf` 所指的数组中。函数返回值为实际读入的数据项个数，出错或读到文件末尾的情况必须用检测函数 `ferror()` 和 `feof()` 来判定。

2. fwrite()

fwrite()用于向文件写入一个数据块，其调用格式为：

```
fwrite(buf,size,count,fp);
```

其中，各个参数的含义与 fread()中的相同。它的功能是向 fp 所指的文件中写入一个由 buf 指向的数据块，该数据块共有 count 个数据项，每个数据项有 size 个字节。如果执行成功，返回实际写入的数据项个数；若所写实际数据项少于需要写入的数据项，则出错。

前面进行文件复制时采用了逐个字符或逐行读出再写入的办法，程序执行效率低，如果用读写数据块的方法则可以提高程序的执行效率。

例 12.5 文件数据块读写举例，利用数据块读写函数进行文件的复制。

```
#include <stdio.h>
FILE *fp1,*fp2;
main()
{
    char buffer[200];
    clrscr();
    if((fp1=fopen("file1.txt","r"))==0)
    {
        printf("Cannot open file file1.txt\n");
        exit(1);
    }
    if((fp2=fopen("file2.txt","w"))==0)
    {
        printf("Cannot open file file2.txt\n");
        exit(1);
    }
    while(1)
    {
        if(fread(buffer,sizeof(buffer),1,fp1) != 1)
            fwrite(buffer,sizeof(buffer),1,fp2);
        else if(!feof(fp1))
        {
            printf("premature end of file\n");
            break;
        }
        else
        {
            printf("file read end\n");
            break;
        }
    }
    fclose(fp1);
    fclose(fp2);
}
```

程序中用 fread()的返回值来判断是否已读入指定的数据项个数，如果所读数据项比指



定的个数少，说明已发生错误或遇到文件末尾，这时可用函数 `feof()` 或 `ferror()` 进行判定。

12.5 文件的定位与读写

12.5.1 文件的定位

文件的定位是指移动文件读写位置指针到指定位置。与文件定位有关的函数有 `fseek()` 函数、`ftell()` 函数、`rewind()` 函数等。

1. `fseek()` 函数

`fseek()` 函数的作用是使文件读写位置指针移动到所需要的位置，它的调用形式为：

```
fseek(文件类型指针, 位移量, 起始点);
```

“起始点”是指以什么地方为基准进行移动，值只能取在表 12-3 中列出的符号名（也可以用对应的值）之一，注意使用符号名时要大写。

表 12-3 起始点的取值

符号名	值	含义
SEEK_SET	0	文件开头
SEEK_CUR	1	文件当前位置
SEEK_END	2	文件末尾

“位移量”是指以“起始点”为基点向前移动的字节数；如果它的值为负数，表示向后移。所谓“向前”，是指从文件开头向文件末尾移动的方向，位移量应为 `long` 型数据，这样当文件长度很长时（例如大于 64 k），位移量仍在 `long` 型数据表示范围之内。下面是几个函数调用的例子：

```
fseek(fp, 10, 0);          /*将读写位置指针移到离文件开始处 10 个字节处*/
fseek(fp, -20L, 1);        /*将读写位置指针从当前位置向后移 20 个字节*/
fseek(fp, -50L, 2);        /*将读写位置指针从文件末尾后移 50 个字节*/
```

如果 `fseek()` 函数执行成功，函数值返回 0，否则返回一个非 0 值。

`fseek()` 函数常用于二进制文件的随机读写。用于文本文件时，因字符转换问题定位常出错误。

2. `ftell()` 函数

`ftell()` 函数能告知用户文件读写位置指针的当前位置，例如 `ftell(fp)` 的返回值是 `fp` 所指向的文件读写位置指针的当前值，返回值为长整型数，表示相对于文件头的字节数，出错时返回 -1L。

`ftell()` 函数调用格式为：

```
ftell(fp);
```

例如，以下程序段：

```
long i;
if((i=ftell(fp))!= -1L)
    printf("A file error has occurred at %ld.\n",i);
```

该程序段可以通知用户在文件的何处出现了文件错误。

3. rewind()函数

使用 `rewind()` 函数可使文件读写位置指针重新返回文件的开头处。调用格式为：

```
rewind(fp);
```

函数调用成功则返回 0 值；否则，返回非 0 值。例如，下面的程序段：

```
while(!feof(fp))
    putchar(fgetc(fp));
rewind(fp);
while(!feof(fp))
    putchar(fgetc(fp));
```

以上代码可两次显示 `fp` 所指定的文件内容。

12.5.2 文件的顺序读写和随机读写

对文件的顺序读写是指从文件的开头逐个数据进行读或写。文件中的“读写位置指针”指向当前读或写的位置。在顺序读写时，每读或写完一个数据后，该位置指针就自动移到它后面的一个位置。如果读写的数据项包含多个字节，则对该数据项读写完毕后读写位置指针移到该数据项之末（即下一数据项的起始地址）。

文件的随机读写就是移动读写位置指针到所需要的地方再进行读写。若希望能直接读到某一数据而不是按物理顺序逐个查找，只要能移动读写位置指针到所需要的地方，就能实现随机读写。方法是用 `fseek()` 函数指定读写位置指针的值，然后进行读写。

在文件的存取中，经常用 `w`、`r`、`a` 方式打开文件，其中，`w` 用于建立文件，`r` 用于读取文件，`a` 则用于在文件的末尾添加数据。如果在同一个程序中既要写又要读，就需要用 `w+` 等方式。对字符串为主的文件（例如一篇文章）而言，建议采用文本方式打开文件；对数值数据为主或字符和数值兼而有之的文件（例如档案）来说，则建议采用二进制文件方式打开文件。为了提高文件读写的速度，经常将数据组织成结构类型，以便使用 `fread()` 和 `fwrite()` 来进行数据块的读写。

例 12.6 练习将文件逆序输出：将文件以二进制读的方式打开，从文件尾按逆序逐个读出字符并显示。

```
/*write a file backwards. */
#include<stdio.h>
#define MAXSTRING 100
int main(void)
{
```



```

char fname[MAXSTRING];
int c;
FILE *ifp;
printf("\n input a filename: ");
scanf("%s",fname);
ifp=fopen(fname,"rb");/*以二进制读的方式打开文件*/
    fseek(ifp,0,SEEK_END);/* 定位到文件尾*/
fseek(ifp,-1,SEEK_CUR); /* 读写位置指针从当前位置向文件头方向退一个字符*/
while(ftell(ifp)>0)      /* 当未退到文件头时,循环 */
{
    c=getc(ifp);    /* 读文件中一个字符,读写位置指针向文件尾方向移动一个字符*/
    putchar(c);
    fseek(ifp,-2,SEEK_CUR);/*读写位置指针从当前位置向文件头方向退 2 个字符*/
}
return 0;
}

```

如果在可执行文件所在的目录中存在一个文件 fl.txt, 文件的内容为 “good morning”。则程序运行情况如下:

```

input a filename: fl.txt✓
gninrom doog

```

例 12.7 文件中一个独立的数据单位常称为一个记录,例如在学生成绩文件中,学号、姓名、分数等就构成一个记录,可以用结构体数据表示记录。在例 12.3 中的程序是将键盘输入的学生信息数据并按照输入时的顺序存入文本文件的。要根据学号查找某个记录时,可以用格式输入函数将记录依次读入结构体变量,然后将学号和要查找的学号进行比较,如果相同,则显示找到的信息,如果没有相同的记录,则显示查找失败。但是如果文件较大,而要查找的数据又在后边,则这种方法效率偏低。

此时如果将数据直接写入二进制文件,采用直接定位的方式可以提高效率,即从键盘输入学号,程序自动定位到相应的记录位置,以二进制形式写入信息。查询时根据输入的学号定位,读出信息并显示。若输入学号 0,则结束查询。程序代码如下:

```

#include<stdio.h>
#define N 4
typedef struct student
{
    int number;
    char name[8];
    int score;
}st;
st stu[N],st1;
main()
{
    FILE *fp;
    int j,k,p;
    for(j=0;j<N;j++)                /*读入 N 个学生信息*/
    {
        printf("please input number,name,score");
    }
}

```

```

        scanf("%d %s %d",&stu[j].number,stu[j].name,&stu[j].score);
    }
    /*将信息以二进制形式写入文件 score.dat*/
    if((fp=fopen("score.dat","w+"))==NULL)
    {
        printf("can't open file");
        exit();
    }
    if(fwrite(&stu,sizeof(st),4,fp)!=4)          /*将数组一起写入文件*/
    {
        puts("Write file error.");
        exit(1);
    }
    fclose(fp);
    /*从 score.dat 读入学号、姓名和成绩*/
    if((fp=fopen("score.dat","rb"))==NULL)      /*将文件以只读方式打开*/
    {
        printf("can't open file");
        exit(0);
    }
    printf("input a number for searching:");      /* 查询开始 */
    scanf("%d",&j);                               /* 输入要查询的学号 */
    do
    {
        fseek(fp,(long)(j-1)*sizeof(st),0);      /* 文件读写位置指针定位 */
        fread(&st1,sizeof(st),1,fp);             /* 读取指定的数据 */
        printf("num=%-6dname=%-8sscore=%-6d\n",
            st1.number,st1.name,st1.score);
        printf("input number for searching:");
        scanf("%d",&j);
    }while(j>0);
    fclose(fp);
}

```

例 12.8 可以按数据块处理二进制文件，也可以按字节处理二进制文件。其实任何文件在内存中都是以二进制形式存放的，不过由文件头部分标识出不同的文件格式而已。下面的程序以字节方式对文件进行读写，以进行 256 色位图文件图片的反转。

```

/*write a file backwards. */
#include<stdio.h>
#define MAXSTRING 100
int main(void)
{
    char fname[MAXSTRING];
    int i;char c;
    FILE *ifp,*ofp;
    printf("\n input source filename : ");
    scanf("%s",fname);
    if((ifp=fopen(fname,"rb"))==0)
        exit(0);

```




```
printf("\n input destination filename: ");
scanf("%s",fname);
if((ofp=fopen(fname,"wb"))==0)
    exit(1);
for(i=0;i<1078;i++)
/*文件开始部分的 1078 个字节是位图文件的文件头，存储了文件格式、图片尺寸、
颜色表等信息，原样复制到目的文件。*/
{
    c=fgetc(ifp);
    fputc(c,ofp);
}
fseek(ifp,0,SEEK_END); /*源文件定位到尾部*/
fseek(ifp,-1,SEEK_CUR);
while(ftell(ifp)>1077)
{
    /*源文件从后向前读，目的文件从前向后写*/
    c=fgetc(ifp);
    fputc(c,ofp);
    fseek(ifp,-2,SEEK_CUR);
}
fclose(ifp);
fclose(ofp);
return 0;
}
```

执行该程序时，如果键盘输入源文件名为 test1.bmp，目的文件名为 test2.bmp，则由 test1.bmp 可以反转复制成 test2.bmp，效果如图 12-1 所示。

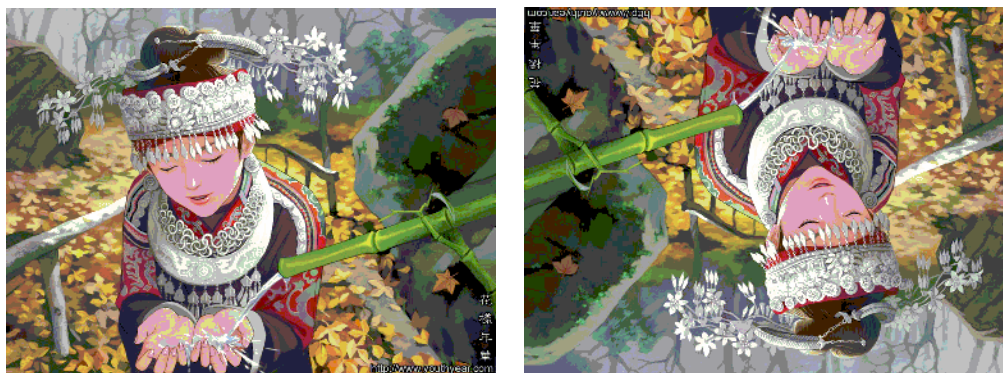


图 12-1 效果图

请读者自行练习应用 fread()和 fwrite()函数按数据块读写完成同样的工作。

要 点 回 顾

1. 文件就是存储在外部介质上的数据或信息的集合。按照文件中数据的存储方式，文件分为文本文件和二进制文件。

2. C 语言使用两种文件系统：缓冲文件系统和非缓冲文件系统。在缓冲文件系统中，是使用文件指针来标识文件。

3. 文件指针是一种 **FILE** 类型指针，**FILE** 是一个特殊的结构体类型标识符，定义文件指针的格式如下：

```
FILE *fp;
```

fp 为指向 **FILE** 类型结构体的指针变量（简称文件指针），它可以通过 **fopen()** 函数连接到指定的文件上（称为指向文件）。标准输入输出设备文件的文件指针由系统命名为 **stdin** 和 **stdout**。

4. 打开文件要使用库函数 **fopen()**，该函数的常用形式为：

```
fp=fopen(fname,mode);
```

其中 **mode** 表示文件读写方式，用 **w**、**r**、**a** 等表示，**w** 用于建立文件，**r** 用于读取文件，**a** 则用于在文件的末尾添加数据。

5. 文件使用完毕，必须用 **fclose()** 函数来关闭文件，以保证本次文件操作有效。使用方式为：

```
fclose(fp);
```

6. **feof()** 函数用来检测文件读写位置指针是否已到文件末尾，以便能正确地进行文件的存取。**feof()** 的调用格式为：

```
feof(fp);
```

到文件尾时函数返回非 0 值。

7. **ferror()** 函数用来检查输入、输出函数的每次调用是否有错，函数的调用格式为：

```
ferror(fp);
```

正常时函数返回 0 值，出错时返回非 0 值。

8. **fputc()** 和 **putc()** 函数用来将一个字符写入文件；**fgetc()** 和 **getc()** 用于从指定文件中读出一个字符。这几个函数的使用格式为：

```
fputc(ch,fp); putc(ch,fp); fgetc(fp); getc(fp);
```

9. 函数 **fgets()** 可以从指定文件中读入一行以 '\0' 或 EOF 结尾的字符串并赋给字符型数组。其调用格式为：

```
fgets(s,n,fp);
```

其中 **s** 是字符型数组名，用字符型数组存放读进来的字符串；**n** 是指定读入字符的个数。

10. **fputs()** 函数可以将一个字符串写入指定的文件。它的调用格式为：

```
fputs(s,fp);
```

其中，**s** 是一个字符型数组名或指向字符串的指针，表示要写入文件的字符串。

11. 函数 **fscanf()** 和 **fprintf()** 可以实现对文件的格式读写。它们与 **scanf()** 和 **printf()** 的不同之处在于，**fscanf()** 和 **fprintf()** 只是对文件进行读写，而不是对标准输入输出设备进行读写。



函数 `fprintf()` 的使用格式为：

```
fprintf(fp, "格式说明字符串", 输出项表);
```

12. 用 `fprintf()` 和 `fscanf()` 函数读写文件非常方便，容易理解，但效率不高。因为 `fscanf()` 读取的只是字符串，需将其转换成二进制才能存入内存；`fprintf()` 又需将二进制数转换为字符串才能输出到文件。文件的读、写都需转换，故花费时间较多。在要求速度的情况下，最好采用 `fread()` 和 `fwrite()` 函数。

13. `fread()` 用于从文件中读出一个数据块，它的调用格式为：

```
fread(buf, size, count, fp);
```

其中，`buf` 是指向数组的指针，数组用于存放读入的数据；`size` 是无符号整型量，用于计算要读入的每个数据项的长度（字节数）；`count` 是整型量，用于指定数据项的个数。该函数的功能是从 `fp` 所指的文件中，一次读出长度为 `size` 字节的 `count` 个数据项，然后存放在 `buf` 所指的数组中。

14. `fwrite()` 用于向文件写入一个数据块，其调用格式为：

```
fwrite(buf, size, count, fp);
```

其中，各个参数的含义与 `fread()` 函数中的参数相同。它的功能是向 `fp` 所指的文件中写入一个由 `buf` 指向的数据块，该数据块共有 `count` 个数据项，每个数据项有 `size` 个字节。

15. `fseek()` 函数的作用是使文件读写位置指针移动到所需要的位置，`fseek()` 函数的调用形式为：

```
fseek(文件类型指针, 位移量, 起始点);
```

16. `ftell()` 函数告知用户文件读写位置指针的当前位置，调用格式为：

```
ftell(fp);
```

17. `Rewind()` 函数可使文件读写位置指针重新返回到文件的开头处。调用格式为：

```
rewind(fp);
```

函数调用成功返回 0 值；否则，返回非 0 值。

18. 对文件的顺序读写是从文件的开头开始对逐个数据进行读或写。在顺序读写时，每读或写完一个数据后，文件读写位置指针就自动移到它后面的一个位置。

19. 文件的随机读写就是移动读写位置指针到所需要的地方再进行读写，通常用 `fseek()` 函数指定读写位置指针的值，然后再进行读写。

习 题

1. 以下程序将磁盘中的一个文件复制到另一个文件中，两个文件名在命令行中给出（假定文件名无误），请填空。

```
#include "stdio.h"
main(int argc, char *argv[ ] )
```

```

{
    FILE    *f1,f2;
    char    ch;
    if(argc<____)
    {
        printf("命令行参数错! ");
        exit(0);
    }
    f1=fopen(argv[1],"r");
    f2=fopen(argv[2],"w");
    while(____)
        fputc(fgetc(f1),____);
    _____;
    _____;
}

```

2. 以下程序的功能是：从键盘上输入一个字符串，把该字符串中的小写字母转换为大写字母，且输出到文件 test.txt 中，然后从该文件中读出字符串并显示出来。请填空。

```

#include<stdio.h>
main()
{
    FILE    *fp;
    char    str[100];
    int    i=0;
    if((fp=fopen("text.txt", _____))= =NULL)
    {
        printf("can't open this file.\n");
        exit(0);
    }
    printf("input astring:\n");
    gets(str);
    while(str[i])
    {
        if(str[i]='a'&&str[i]<='z')
            str[i]=_____;
        fputc(str[i],fp);
        i++;
    }
    fclose(fp);
    fp=fopen("test.txt", _____);
    fgets(str,100,fp);
    printf("%s\n",str);
    fclose(fp);
}

```

3. 以下程序由终端键盘输入一个文件名，然后把终端键盘输入的字符依次存放到该文件中，用#号作为结束输入的标志。请填空。

```

#include    "stdio.h"
main()

```



```
{
    FILE *fp;
    char ch, fname[10];
    printf("Enter the name of file\n");
    gets(fname);
    if((fp=fopen(fname, "w")) == NULL)
    {
        printf("Open error\n");
        exit(0);
    }
    printf("Enter data : \n");
    while((ch=getchar()) != '#')
        fputc(ch, fp);
    fclose(fp);
}
```

4. 请调用 `fputs()` 函数，把 10 个字符串输出到文件中；再从此文件中读入这 10 个字符串放在一个字符串数组中；最后把字符串数组中的字符串输出到终端屏幕，以检验所有操作是否正确。

5. 从键盘输入 10 个浮点数，以二进制形式存入文件中。再从文件中读出数据并显示在屏幕上。修改文件中的第 4 个数。再从文件中读出数据显示在屏幕上，以验证修改是否正确。

6. 从键盘输入一批学生信息，如姓名、年龄、联系电话、通讯地址等，存成一个通讯录（`txl.txt`）文件。

7. 打开通讯录文件，从键盘输入一个学生姓名，并根据姓名查找并显示该学生的信息。

第 13 章 综合程序设计举例

本章的学习目标:

综合应用前面章节所学习的知识,进行较大程序的编写、调试。可以由学生自己阅读分析程序,并对程序进行调试、修改等操作,然后自己编写一个类似的程序。

本章将进行一个综合程序设计。通过综合程序设计,读者可进一步熟悉各种基本结构程序的语句及实现方法,掌握 C 语言的函数、数组、指针、结构体、文件等的用法,练习对数组进行排序、检索、删除、统计等操作。

要求完成如下任务。

(1) 建立文件 `score.txt`, 用该文件存储多个学生的学号、姓名、成绩等信息, 并随意输入一些学生数据。

(2) 追加数据: 在文件末尾追加数据。

(3) 列表: 以特定的格式显示文件内容。

(4) 排序: 读出文件 `score.txt` 的内容, 按成绩排序后显示且存入 `sort.txt` 中, 并显示出最高分和最低分。

(5) 删除某个数据: 根据键盘输入的学号, 删除相应的信息。

(6) 查询输出: 根据键盘输入的姓名, 在 `score.txt` 中找到对应的学生数据并显示。

(7) 数据统计: 计算并显示各分数段的人数及平均分。

各功能分别设计成函数并由主函数进行调用, 参考程序如下:

```
/* 将表示学生信息的结构体数组和一个结构体变量定义为外部类, 各个函数都可以直接使用, 再定义一个可以供各个文件使用的文件类型指针。*/
#include<stdio.h>
typedef struct student
{
    int number;
    char name[8];
    int score;
}ST;
ST stu[60],st1;
FILE *fp;
/* creat 函数实现从键盘输入多个学生的学号、姓名和成绩, 并将信息用文件格式输出语句存入文件 score.txt 中, 当输入的学号为 0 时表示数据输入完成。*/
void creat()
{
    if((fp=fopen("score.txt", "w+"))==NULL)
    {
        printf("can't open file");
        exit(1);
    }
}
```



```
printf("please input number name score");
scanf("%d %s %d",&st1.number,st1.name,&st1.score);
/*键盘输入学号、姓名和分数*/
while(st1.number!=0)
{
    /*将学号、姓名和分数写入文件 score.txt*/
    fprintf(fp,"%6d%8s%6d\n",st1.number,st1.name,st1.score);
    printf("please input number name score");
    scanf("%d %s %d",&st1.number,st1.name,&st1.score);
}
fclose(fp);
}
/*append()函数以追加方式打开文件，在文件尾部追加数据*/
void append()
{
    if((fp=fopen("score.txt","at"))==NULL)
    {
        printf("can't open file");
        exit(2);
    }
    printf("please input number name score");
    scanf("%d %s %d",&st1.number,st1.name,&st1.score);
    fprintf(fp,"%6d%8s%6d\n",st1.number,st1.name,st1.score);
    fclose(fp);
}
/*list()函数列表显示所有数据*/
void list()
{
    if((fp=fopen("score.txt","rt"))==NULL)
    {
        printf("can't open file");exit(0);
    }
    printf("---number--|--name--|--score--\n");
    while(!feof(fp))
    {
        fscanf(fp,"%6d%8s%6d\n",&st1.number,st1.name,&st1.score);
        printf(" %8d | %6s | %5d\n",st1.number,st1.name,st1.score);
        printf("-----\n");
    }
}
/*find()函数根据键盘输入的学号进行查找*/
void find()
{
    int num1;
    if((fp=fopen("score.txt","rt"))==NULL)
    {
        printf("can't open file");
        exit(0);
    }
    printf("input the number you want to find");
```

```
scanf("%d",&num1);
while(!feof(fp))
{
    fscanf(fp,"%6d%8s%6d\n",&st1.number,st1.name,&st1.score);
    if(st1.number==num1)
    {
        printf("%6d%8s%6d\n",st1.number,st1.name,st1.score);
        break;
    }
}
if(feof(fp))
    printf("did not find\n");
fclose(fp);
}
/* dele()函数根据输入的学号删除数据 */
void dele()
{
    int j,n,num1;
    if((fp=fopen("score.txt","rt"))==NULL)
    {
        printf("can't open file");exit(0);
    }
    printf("input the number you want to delete");
    scanf("%d",&num1);
    j=0;
    while(!feof(fp))/*从文件读输入到数组*/
    {
        fscanf(fp,"%6d%8s%6d\n",&stu[j].number,
            stu[j].name,&stu[j].score);
        if(stu[j].number!=num1)
        {
            j++;
            continue;
        }
        else
            continue;
    }/*如果学号和要删除的学号一致,直接读下一数据*/
    n=j;
    fclose(fp);
    if((fp=fopen("score.txt","wt"))==NULL)/*以写方式打开文件*/
    {
        printf("can't open file");
        exit(0);
    }
    for(j=0;j<n;j++)
        fprintf(fp,"%6d%8s%6d\n",stu[j].number,
            stu[j].name,stu[j].score);
    fclose(fp);
}
/*sort()函数将数据读出排序后显示并存入 sort.txt 文件*/
```




```
void sort()
{
    int j,k,p,t,s,n;
    char tname[8];
    if((fp=fopen("score.txt","rt"))==NULL)
    {
        printf("can't open file");exit(0);
    }
    j=0;
    while(!feof(fp))
    {
        fscanf(fp,"%6d%8s%6d\n",&stu[j].number,
            stu[j].name,&stu[j].score);
        j++;
    }
    n=j;
    printf("sort on number :1\n"); /*可以由用户选择按学号或分数排序*/
    printf("sort on score :2\n");
    scanf("%d",&s);
    switch(s)
    {
        case 1:
        {
            for(j=0;j<n-1;j++)
            {
                p=j;
                for(k=j+1;k<n;k++)
                    if(stu[p].number>stu[k].number)
                        p=k;
                t=stu[j].number;
                stu[j].number=stu[p].number;
                stu[p].number=t;
                strcpy(tname,stu[j].name);
                strcpy(stu[j].name,stu[p].name);
                strcpy(stu[p].name,tname);
                t=stu[j].score;
                stu[j].score=stu[p].score;
                stu[p].score=t;
            }
        }
        break;
        case 2:
        {
            for(j=0;j<n-1;j++)
            {
                p=j;
                for(k=j+1;k<n;k++)
                    if(stu[p].score>stu[k].score)
                        p=k;
                t=stu[j].number;
```

```
        stu[j].number=stu[p].number;
        stu[p].number=t;
        strcpy(tname,stu[j].name);
        strcpy(stu[j].name,stu[p].name);
        strcpy(stu[p].name,tname);
        t=stu[j].score;
        stu[j].score=stu[p].score;
        stu[p].score=t;
    }
}
break;
default:printf("select error");return;
}
if((fp=fopen("sort.txt","wt"))==NULL)
{
    printf("can't open file");
    exit(0);
}
/*显示排序后的名次、姓名和成绩*/
printf("place|--number--|--name--|--score\n");
for(j=0;j<n;j++)
{
    printf("%4d | %8d | %6s | %5d\n",j+1,
        stu[j].number,stu[j].name,stu[j].score);
    printf("-----\n");
    fprintf(fp,"%6d%8s%6d\n",stu[j].number,stu[j].name,stu[j].score);
    /*同时写文件*/
}
}
/* statistic()函数统计各分数段的人数并计算平均分 */
void statistic()
{
    int s06=0,s67=0,s78=0,s89=0,s910=0;
    int j=0,temp;
    float sum=0,aver;
    if((fp=fopen("score.txt","rt"))==NULL)
    {
        printf("can't open file");exit(0);
    }
    while(!feof(fp))
    {
        fscanf(fp,"%6d%8s%6d\n",&stu[j].number,
            stu[j].name,&stu[j].score);
        sum+=stu[j].score;
        temp=stu[j].score/10;
        if(temp==10)
            temp=9;
        switch(temp)
        {
            case 0:
```



```
        case 1:
        case 2:
        case 3:
        case 4:
        case 5: s06++;break;
        case 6: s67++;break;
        case 7: s78++;break;
        case 8: s89++;break;
        case 9: s910++; break;
    }
    j++;
}
aver=sum/j;
printf("0--60:%d\n60--70:%d\n70--80:%d\n80--90:
      %d\n90--100:%d\n",s06,s67,s78,s89,s910);
printf("average=%.2f\n",aver);
}
/*主程序显示菜单并根据键盘输入的选项调用各个函数*/
main()
{
    int m;
    while(1)
    {
        printf("\n\t*****\n");
        printf("\t*   1   creat       *\n\n");
        printf("\t*   2   append      *\n\n");
        printf("\t*   3   list        *\n\n");
        printf("\t*   4   sort        *\n\n");
        printf("\t*   5   delete     *\n\n");
        printf("\t*   6   find        *\n\n");
        printf("\t*   7   statistic   *\n\n");
        printf("\t*   0   quit       *\n\n");
        printf("\t*****\n");
        printf("\tplease select");
        scanf("%d",&m);/*accept the enter*/
        if(m==0)
            break;
        switch(m)
        {
            case 1: creat();break;
            case 2: append();break;
            case 3: list();break;
            case 4: sort();break;
            case 5: dele();break;
            case 6: find();break;
            case 7: statistic();break;
        }
    }
}
```

为简化思路，本程序假设学生数接近 60，所以程序中预先定义了一个 60 个元素的结构体数组，但这种方法存在如下不足：如果某个班学生数只有 20，则数组多占的空间造成浪费；如果学生数超过 60，程序处理时下标越界可能会出现难以预料的错误。较好的处理办法是采用指针，每次程序运行时，由键盘输入学生数，再根据学生数分配内存。请读者自己尝试用指针的方法改写本程序。

还有一个问题就是上面参考程序是用文本文件处理数据：如果数据较多，则文件大，用文本文件的方法效率低。请读者自己练习用二进制文件的方法保存数据并进行各种操作。

改进的工作可以由几个同学合作完成，讨论确定分工安排，注意函数间的关系和调用时的参数传递，分别写不同的函数，编译无错后，合成一个项目文件连接并执行即可。



附录 1 C 运算符的优先级与结合性

优先级	运算符	功能	运算量个数	结合性
15	()	圆括号, 提高优先级	2	左
	[]	下标运算, 访问地址		
	->	指向结构体或共用体成员		
	•	取结构体或共用体成员		
	!	逻辑非		
	~	按位取反		
	++	加 1		
14	--	减 1	1	右
	-	负号运算符		
	(type)	强制类型转换		
	*	访问地址或指针		
	& sizeof	取地址 测试数据长度		
13	*	乘法	2	左
	/	除法		
	%	求整数余数		
12	+	加法	2	左
	-	减法		
11	>>	左移位	2	左
	<<	右移位		
10	< <= > > >=	关系运算	2	左
9	==	等于	2	左
	!=	不等于		
8	&	按位与	2	左
7	^	按位异或	2	左
6		按位或	2	左
5	&&	逻辑与	2	左
4		逻辑或	2	左
3	?:	条件运算	3	右
2	= += -= *= /= %= ^=	赋值运算	2	右
	= &= >>= <<=			
1	,	逗号运算		左

注: 表中所列优先级, “15” 级为最高, “1” 级为最低。

附录 2 ASCII 码表

字符	ASCII 码	字符	ASCII 码	字符	ASCII 码	字符	ASCII 码	字符	ASCII 码
NUL	0	SUB	26	4	52	N	78	H	104
SOH	1	ESC	27	5	53	O	79	i	105
STX	2	FS	28	6	54	P	80	j	106
ETX	3	GS	29	7	55	Q	81	k	107
EOT	4	RS	30	8	56	R	82	l	108
EDQ	5	US	31	9	57	S	83	m	109
ACK	6	Space	32	:	58	T	84	n	110
BEL	7	!	33	;	59	U	85	o	111
BS	8	“	34	<	60	V	86	p	112
HT	9	#	35	=	61	W	87	q	113
LF	10	\$	36	>	62	X	88	r	114
VT	11	%	37	?	63	Y	89	s	115
FF	12	&	38	@	64	Z	90	t	116
CR	13	‘	39	A	65	[91	u	117
SO	14	(40	B	66	\	92	v	118
SI	15)	41	C	67]	93	w	119
DLE	16	*	42	D	68	^	94	x	120
DCI	17	+	43	E	69	_	95	y	121
DC2	18	,	44	F	70	`	96	z	122
DC3	19	-	45	G	71	a	97	{	123
DC4	20	.	46	H	72	b	98		124
NAK	21	/	47	I	73	c	99	}	125
SYN	22	0	48	J	74	d	100	~	126
ETB	23	1	49	K	75	e	101	del	127
CAN	24	2	50	L	76	f	102		
EM	25	3	51	M	77	g	103		

注：表中所列 ASCII 码值为十进制数。



附录 3 Turbo C 2.0 常用库函数及其标题文件

1. I/O 函数（标题文件 stdio.h）

下面的数据类型中有一些类型是 stdio.h 中定义的，例如，size_t 表示整型字节数。

函数名	函数原型说明	功能	返回值
clearerr	void clearerr(FILE *fp)	清除与文件指针 fp 有关的所有出错信息	无
fclose	int fclose(FILE *fp)	关闭 fp 所指的文件	出错返回非 0， 否则返回 0
feof	int feof(FILE *fp)	检查文件是否结束	遇文件结束返回非 0， 否则返回 0
fgetc	int fgetc(FILE *fp)	从 fp 所指文件读取一个字符	出错返回 EOF， 否则返回所读字符数
fgets	Char fgets(char *str,int num,FILE *fp)	从 fp 所指文件读取一个长度为 num-1 的字符串，存入 str 中	返回 str 地址， 遇文件结束返回 NULL
fopen	FILE *fopen(const char *fname,const char *mode)	以 mode 方式打开文件 fname	成功时返回文件指针， 否则返回 NULL
fprintf	int fprintf(FILE *fp,const char *format,arg-list)	将 arg-list 的值按 format 指定的格式写入 fp 所指文件中	返回实际输出的字符数
fputc	int fputc(int ch,FILE *fp)	将 ch 中的字符写入 fp 所指文件	成功时返回该字符， 否则返回 EOF
fputs	int fputs(const char *str,FILE *fp)	将 str 中的字符串写入 fp 所指文件中	成功时返回 0， 否则返回非 0
fread	size_t fread(void buf,size_t size,size_t count,FILE *fp)	从 fp 所指文件读取长度为 size 的 count 个数据项，写入 fp 所指文件中	返回读取的数据项个数
fscanf	int fscanf(FILE *fp,const char *format,arg-list)	从 fp 所指文件按 format 指定的格式读取数据并存入 arg-list 中	返回读取的数据个数， 出错或遇文件结束返回 0
fseek	int fseek(FILE *fp,long offset,int origin)	移动 fp 所指文件读写位置指针的位置	成功时返回当前位置， 否则返回 -1
ftell	long ftell(FILE *fp)	求出 fp 所指文件的当前读写位置	读写位置
fwrite	size_t fwrite(const void *buf,size_t size,size_t count,FILE *fp)	将 buf 所指的内存区中的 count*size 个字节写入 fp 所指文件中	写入的数据项个数

(续表)

函数名	函数原型说明	功能	返回值
Getc	int getc(FILE *fp)	从 fp 所指文件中读取一个字符	返回读取的字符, 出错或遇文件结束时返回 EOF
getch	int getch(void)	从标准输入设备读取一个字符, 不必用回车键, 不在屏幕上显示	返回所读字符, 否则返回-1
getche	int getche(void)	从标准输入设备读取一个字符, 不必用回车键并在屏幕上显示	返回所读字符, 否则返回-1
getchar	int getchar(void)	从标准输入设备读取一个字符, 以回车键结束, 并在屏幕上显示	返回所读字符, 否则返回-1
gets	char *gets(char *str)	从标准输入设备读取一个字符串, 遇回车键结束	返回读取的字符串
getw	int getw(FILE *fp)	从 fp 所指文件中读取一个整型数	返回读取的整数
printf	int printf(const char *format, arg-list)	将 arg-list 中的数据按 format 指定的格式输出到标准输出设备	返回输出的字符个数
putc	int putc(int ch, FILE *fp)	同 fputc	同 fputc
putchar	int putchar(int ch)	将 ch 中的字符输出到标准输出设备	返回输出的字符, 出错返回 EOF
puts	int puts(const char *str)	将 str 所指内存区中的字符串输出到标准输出设备	返回换行符, 出错返回 EOF
remove	int remove(const char *fname)	删除 fname 所指文件	成功返回 0, 否则返回-1
rename	int rename(const char *oldfname, const char newfname)	将名为 oldfname 的文件更名为 newfname	成功返回 0, 否则返回-1
rewind	void rewind(FILE *fp)	将 fp 所指文件读写位置指针指向文件开头	无
scanf	int scanf(const char *format, arg-list)	从标准输入设备按 format 指定的格式读取数据, 并存入 arg-list 中	返回已输入的字符个数, 出错返回 0

2. 字符判别和转换函数 (标题文件 ctype.h)

函数名	函数原型说明	功能	返回值
isalnum	int isalnum(int ch)	检查 ch 是否为字母或数字	是则返回 1, 否则返回 0
isalpha	int isalpha(int ch)	检查 ch 是否为字母	是则返回 1, 否则返回 0
isascii	int isascii(int ch)	检查 ch 是否为 ASCII 字符	是则返回 1, 否则返回 0
isctrl	int isctrl(int ch)	检查 ch 是否为控制字符	是则返回 1, 否则返回 0
isdigit	int isdigit(int ch)	检查 ch 是否为数字	是则返回 1, 否则返回 0



(续表)

函数名	函数原型说明	功能	返回值
isgraph	int isgraph(int ch)	检查 ch 是否为可打印字符, 即不包括控制字符和空格	是则返回 1, 否则返回 0
islower	int islower(int ch)	检查 ch 是否为小写字母	是则返回 1, 否则返回 0
isprint	int isprint(int ch)	检查 ch 是否为字母或数字	是则返回 1, 否则返回 0
ispunch	int ispunch(int ch)	检查 ch 是否为标点符号	是则返回 1, 否则返回 0
isspace0	int isspace(int ch)	检查 ch 是否为空格	是则返回 1, 否则返回 0
isupper	int isupper(int ch)	检查 ch 是否为大写字母	是则返回 1, 否则返回 0
isxdigit	int isxdigit(int ch)	检查 ch 是否为十六进制数字	是则返回 1, 否则返回 0
tolower	int tolower(int ch)	将 ch 中的字母转换为小写字母	返回小写字母
toupper	int toupper(int ch)	将 ch 中的字母转换为大写字母	返回大写字母

3. 字符串函数 (标题文件 string.h/mem.h)

函数名	函数原型说明	功能	返回值
strcat	char *strcat(char *str1,const char *str2)	将字符串 str2 连接到 str1 后面	返回 str1 的地址
strchr	char *strchr(const char *str,int ch)	找出 ch 字符在字符串 str 中第一次出现的位置	返回 ch 的地址, 找不到返回 NULL
strcmp	int strcmp(const char *str1,const char *str2)	比较字符串 str1 和 str2	str1<str2, 返回负数 str1=str2, 返回 0 str1>str2, 返回正数
strcpy	char *strcpy(char *str1,const char *str2)	将字符串 str2 复制到 str1 中	返回 str1 的地址
strlen	size_t strlen(const char *str)	求字符串 str 的长度	返回 str1 包含的字符数 (不含末尾的\0)
strlwr	char *strlwr(char *str)	将字符串 str 中的字母转换为小写字母	返回 str 的地址
strncat	char *strncat(char *str1,const char *str2,size_t count)	将字符串 str2 中的前 count 个字符连接到 str1 的后面	返回 str1 的地址
strncpy	char *strncpy(char *dest,const char *source,size_t count)	将字符串 str2 中的前 count 个字符复制到 str1 中	返回 str1 的地址
strstr	char *strstr(const char *str1,const char *str2)	找出字符串 str2 在字符串 str1 中第一次出现的位置	返回 str2 的地址, 找不到返回 NULL
strupr	char *strupr(char *str)	将字符串 str 中的字母转换为大写字母	返回 str 的地址

4. 数学函数 (标题文件 math.h)

函数名	函数原型说明	功能	返回值
acos	double acos(double x)	计算 $\arccos(x)$ 的值	计算结果
asin	double asin(double x)	计算 $\arcsin(x)$ 的值	计算结果
atan	double atan(double x)	计算 $\arctan(x)$ 的值	计算结果
atan2	double atan2(double y, double x)	计算 $\arctan(x/y)$ 的值	计算结果
ceil	double ceil(double num)	求不小于 num 的最小整数	计算结果
cos	double cos(double x)	计算 $\cos(x)$ 的值	计算结果
cosh	double cosh(double x)	计算 $\cosh(x)$ 的值	计算结果
exp	double exp(double x)	计算 e^x 的值	计算结果
fabs	double fabs(double num)	计算 x 的绝对值	计算结果
floor	double floor(double num)	求不大于 x 的最大整数 (双精度)	计算结果
fmod	double fmod(double x, double y)	求 x/y 的余数 (即求模)	计算结果
frexp	double frexp(double num, int *exp)	将双精度数分成尾数部分和指数部分	计算结果
hypot	double hypot(double x, double y)	计算直角三角形的斜边长	计算结果
log	double log(double num)	计算自然对数	计算结果
log10	double log10(double num)	计算常用对数	计算结果
modf	double modf(double num, int *i)	将双精度数 num 分解成整数部分和小数部分, 整数部分存放在 i 所指的变量中	返回小数部分
pow	double pow(double x, double y)	计算幂指数 x^y	计算结果
pow10	double pow10(int n)	计算指数函数 10^n	计算结果
sin	double sin(double x)	计算 $\sin(x)$ 的值	计算结果
sinh	double sinh(double x)	计算 $\sinh(x)$ 的值	计算结果
sqrt	double sqrt(double num)	计算 num 的平方根	计算结果
tan	double tan(double x)	计算 $\tan(x)$ 的值	计算结果
tanh	double tanh(double x)	计算 $\tanh(x)$ 的值	计算结果

5. 动态分配函数 (标题文件 stdlib.h)

函数名	函数原型说明	功能	返回值
calloc	void *calloc(size_t num, size_t size)	为 num 个数据项分配内存, 每个数据项大小为 size 个字节	返回分配内存空间的起始地址, 分配不成功返回 0
free	void *free(void *ptr)	释放 ptr 所指的内存	无
malloc	void *malloc(size_t size)	分配 size 个字节的内存	返回分配内存空间的起始地址, 分配不成功返回 0
realloc	void *realloc(void *ptr, size_t newsize)	将 ptr 所指的内存空间改为 newsize 字节	返回新分配内存空间的起始地址, 分配不成功返回 0



附录 4 Turbo C 2.0 编译错误提示和原因

错误提示	错误原因
Array bounds missing] in function xxx	函数 xxx 中的数组缺少方括号
Array size too large in function xxx	函数 xxx 中数组定义的太大
Bad file name format in include directive	在 include 行中文件名格式错（缺引号或尖括号）
Bad ifdef(或 ifndef,或 undef)directive syntax	条件编译行中无标识符
Both return and return of a value used in function xxx	在函数 xxx 中的 return 表达式中使用了函数名
Case outside of switch in function xxx	函数 xxx 中缺关键字 switch 或 case 出现在 switch 语句的外面
Case statement missing: in function xxx	函数 xxx 中的 case 常量表达式后面缺少冒号
Character constant too large in function xxx	函数 xxx 中的字符常量缺少右引号
Code has no effect in function xxx	原因是多方面的,如:变量无类型/存储类别说明;使用了指针相加;数组或结构未给尺寸;使用了非 C 语句等
Compound statement missing} in function xxx	复合语句缺少右花括号
Declaration missing ; in function xxx	说明语句缺分号
Declaration need type or storage class	变量无数据类型/存储类别说明,或 struct 的类型名后少右花括号等
Declaration syntax error in function xxx	数组缺左方括号;字符型变量赋了两个以上的字符; struct/union 说明后少分号;函数调用时参数不匹配;用关键字 typedef 作变量使用;变量无数据/存储类别说明
Default outside of switch in function xxx	default 定义在 switch 之外
Define directive needs identifier in function xxx	#define 行少标识符
Do statement must have while in function xxx	do 语句后无 while 对应
Do-while statement missing(/);in function xxx	do-while 语句少左圆括号、右圆括号或分号
Duplicate case in function xxx	case 的常量表达式重复定义
Enum syntax error in function xxx	enum 语法错误
Expression syntax in function xxx	表达式语法错误或“->”左边无结构指针
For statement missing(/)in function xxx	for 语句缺少左或右圆括号
Function call missing)in function xxx	函数调用时缺右圆括号或参数之间不是逗号隔开
Goto statement missing lable in function xxx	goto 语句少标号

(续表)

错误提示	错误原因
If statement missing(/)in function xxx	if 语句少左或右圆括号
Illegal character '#' in function xxx	"#" 后不是宏指令名
Illegal octal digit in function xxx	出现非法八进制数
Illegal use of floating point in function xxx	浮点数使用场合错
Illegal use of pointer in function xxx	指针运算中使用了非法运算符
Incorrect number format in function xxx	十六进制整数中使用了小数点
Incorrect use of default in function xxx	default 后面无冒号
Invalid macro argument separator in function xxx	带参数宏的参数之间的分隔符不是逗号
Invalid use of arrow in function xxx	"->" 右边无结构成员
Invalid use of dot in function xxx	"." 右边无结构成员
value required in function xxx	赋值号左边不是变量
Misplaced break in function xxx	在 switch 外使用了 break
Misplaced decimal point in function xxx	指数部分使用了小数点
Misplaced else directive in function xxx	没有与 #else 或 #end if 配对的 #if
Misplaced else in function xxx	多用了 else
No file name ending	在 #include 行中, 文件名后少右引号或右尖括号
Nonportable pointer assignment in function xxx	发生指针与指针相减; 把指针值赋给非指针变量; putchar() 使用了非单个字符以及 0 作除数等
Switch statement missing(/)in function xxx	switch 后少左或右圆括号
Too many decimal point in function xxx	小数点太多
Too many default cases in function xxx	switch 中, default 多于一个
Too many types in declaration in function xxx	数据类型关键字多于一个; struct 的类型名后少右花括号等
Two consecutive dots in function xxx	使用了两个句号
Type mismatch in parameter 'c' in call to '_fputc' in function xxx	putchar() 使用了非单个字符; 指针值赋给非指针变量以及 0 作除数等
Undefined symble 'xxx' in function xxx	使用了未定义的变量名、结构名、标号及函数名, 或者将它们用错; 变量无类型/存储类别说明; 企图对硬件寄存器取地址; struct 类型名后少右花括号等
Unexpected end of file in comment started on line n	第 n 行的注释少 "*/"
Unexpected end of file in conditional started on line n in function xxx	条件编译中少 #endif 或 #endif 拼错
Unknown preprocessor directive 'xxx' in function xxx	#后面不是预编译命令



(续表)

错误提示	错误原因
Unreachable code in function xxx	出现不能检索的源码，如数字作语句用
Unterminated string or character constant	字符或字符串常量少右引号
While statement missing(/)in function xxx	while 语句少左或右圆括号
Wrong number of arguments in call 'xxx' in function xxx	调用宏时，参数多于定义的参数
'xxx'is assigned a value which is never used in function xxx	定义的变量未使用

附录 5 实验指导

实验 1 TC 集成环境的基本应用

1. 实验目的

- (1) 了解所用的计算机系统的基本操作方法，学会独立使用该系统。
- (2) 了解 TC 的基本环境，观察 TC 的各个菜单并对菜单项做修改。
- (3) 了解在该系统上如何编程、编译、链接和运行一个 C 程序。
- (4) 通过运行简单的 C 程序，初步了解 C 程序的特点。

2. 实验内容和步骤

(1) 在 Windows 环境下搜索 C 盘，找到 TC 文件夹，将 TC 文件夹连同所有文件复制到 D 盘下。双击 TC 文件夹，观察 TC 文件夹下有哪些文件及子文件夹。在 D 盘的 TC 文件夹下再建一个 USER 文件夹，双击 TC.exe 文件进入 TC 集成环境。修改路径，使用户源程序都放到 USER 文件夹下；即选中 Options 的 Directories 子菜单，使该目录改变为：D:\TC\USER，按回车键使改变生效。再以同样的方法将 Directories | Include Directories 设为 D:\TC\INCLUDE，将 Library 设为 D:\TC\LIB，将 Output Directory 设为 D:\TC\USER，然后按 Esc 键返回上一级菜单，再到 Save Options 子菜单中保存设置，最后按 Esc 键返回主菜单。

(2) 在 TC 集成环境中，先用上、下、左、右键观察各个菜单的大概组成，然后选择 File | New 命令并按回车键，在“编辑”窗口中输入并编辑一个源程序。编辑输入时观察窗口顶部左端 line 和 col 后面数字的变化。

(3) 输入如下源程序：

```
#include <stdio.h>
main()
{
    printf("this is my first C program");
}
```

(4) 执行。

该程序执行时显示“this is my first C program”。按 Alt+C 键进入编译菜单，再分别进入编译（Compile to OBJ）、连接（Link EXE file）子菜单进行编译连接；按 Alt+R 键进入运行菜单，选择 Run 命令运行程序。观察编译、连接、运行的各个步骤，如果出现错误信息（可以有意写错以进行观察），则改正错误后再进入下一步骤。按 Alt+F5 键或 Alt+R 键观察用户屏幕，该屏幕将显示出相应的文字信息。

(5) 保存。

将程序以文件名 first.c 进行存盘，最后按 Alt+X 键退出 TC。



实验 2 数据类型、运算符和表达式

1. 实验目的

(1) 掌握 C 语言数据类型, 熟悉整型、字符型和实型变量的定义方法以及对它们赋值的方法。

(2) 了解所用系统的各种数据类型占用存储单元的字节数, 掌握不同的类型数据之间赋值时的转换规律。

(3) 学会使用 C 的有关算术运算符以及包含这些运算符的表达式, 特别是自加 (++) 和自减 (--) 运算符的使用。

(4) 进一步熟悉 C 程序的编程、编译、连接和运行的过程。

2. 实验内容和步骤

(1) 输入并运行下面的程序:

```
main()
{
    char c1,c2;
    int i1, i2;
    c1='a';
    c2='b';
    i1=97;
    i2=98;
    printf("%c%c%c%c\n",c1,c2,i1,i2);
    printf("%d%d%d%d\n",c1,c2,i1,i2);
}
```

运行此程序, 并分析结果。

(2) 输入并运行下面的程序:

```
main()
{
    printf("size of char=%d\n",sizeof(char));
    printf("size of int=%d\n",sizeof(int));
    printf("size of long int=%d\n",sizeof(long));
    printf("size of float=%d\n",sizeof(float));
    printf("size of double=%d\n",sizeof(double));
}
```

(3) 输入并运行下面的程序:

```
main()
{
    char c1='a',c2='b',c3='c',c4='\101',c5=101;
    printf("a%c b%c\tc%c\tabc\n",c1,c2,c3);
    printf("\t\b%c%c",c4,c5);
    c4=65535;
    c5=-1.2345;
```

```
printf("%d%d",c4,c5);  
}
```

在上机前先分析程序, 写出应得结果, 上机后将二者对照。

(4) 输入并运行下面的程序:

```
main()  
{  
    int i,j,k,l,m,n;  
    i=3;  
    j=5;  
    k=++I;  
    l=j++;  
    m += i++;  
    n -= --j;  
    printf("%d,%d,%d,%d,%d,%d",i,j,k,l,m,n);  
}
```

上机前先判断 i、j、k、l、m、n 各变量的值, 再运行程序并与分析结果做比较。

实验3 简单程序设计及基本调试方法

1. 实验目的

- (1) 掌握 C 语言中赋值语句和输入输出函数的使用方法。
- (2) 掌握程序的基本调试方法, 练习用单步方式和设断点方式执行程序。
- (3) 掌握变量的观察方法。

2. 实验内容和步骤

自己编写一个简单的计算 $c=a+b$ 的程序并运行, 从键盘输入变量 a、b 的值, 用按 F7 键或 F8 键单步执行方式观察各个变量值的变化情况。

参考程序如下:

```
#include<stdio.h>  
main()  
{  
    int a,b,c;  
    printf("please input the value of a and b");  
    scanf("%d,%d",&a,&b);  
    c=a+b;  
    printf("a=%d,b=%d,a+b=%d",a,b,c);  
}
```

在程序中设置断点, 方法是先将光标置于要设断点的语句上, 按 Ctrl+F8 键设置断点; 可以在多个语句行设断点, 也可以将光标放在已设断点的语句上再按 Ctrl+F8 键取消断点。然后按 Alt+R 键运行程序, 观察设断点的程序的运行情况并且在程序断点处观察变量的值, 最后清除断点。

实验中注意练习用菜单及快捷键两种方法设置断点。



修改程序，将变量 a、b、c 设为不同的数据类型，观察程序运行中变量值的变化情况。

实验 4 分支结构程序设计

1. 实验目的

- (1) 掌握 if 语句和 switch 语句的用法及所用到的条件表示法。
- (2) 学会正确使用逻辑运算符和逻辑表达式。
- (3) 掌握一些简单算法。
- (4) 进一步熟悉调试程序的方法。

2. 实验内容

本实验要求事先编好解决下面问题的程序，然后上机输入程序并调试运行程序。

(1) 有如下函数：

$$y = \begin{cases} 0 & (x=0) \\ 1 & (x>0) \\ -1 & (x\leq 0) \end{cases}$$

用 scanf 函数输入 x 的值，求 y 值并输出。

运行程序，使输入的值分别为等于 0、大于 0、小于 0 三种情况，检查输出的值是否正确。

(2) 给出一个百分制成绩，要求输出成绩等级 A、B、C、D、E。90 分以上为 A，80~89 分为 B，70~79 分为 C，60~69 分为 D，60 分以下为 E。当输入数据大于 100 或小于 0 时，通知用户“输入数据错”，程序结束。

要求分别用 if 语句和 switch 语句实现。运行程序并检查结果是否正确，如果出错则进行修改从而得到正确结果。

实验 5 循环程序设计

1. 实验目的

- (1) 掌握用 for 语句、while 语句和 do-while 语句实现循环的方法。
- (2) 掌握用循环的方法实现一些常用算法。
- (3) 练习 break 和 continue 的使用。

2. 实验内容

根据下面的要求编程序并上机调试运行。

(1) 编程计算 $2! + 4! + 6! + \cdots + 20!$ 。

(2) 编程判断键盘输入的数 n 是否为素数，如果是素数，屏幕输出“YES”，否则屏幕输出“NO”。所谓 n 为素数是指，如果 n 是一个大于等于 2 的整数，除 1 和 n 之外，不能被 2~(n-1) 之间的任何整数整除。

(3) 使用迭代法求 85 的平方根的近似值，即方程 $X^2 - 85 = 0$ 的近似解，要求误差小

于十万分之一，分别设根的初始值为 9 和 6，并观察循环次数。

实验 6 函数应用

1. 实验目的

- (1) 掌握函数定义和函数声明的方法。
- (2) 掌握函数调用时实参与形参的对应关系以及“值传递”的方式。
- (3) 掌握函数的嵌套调用和递归调用的方法。
- (4) 掌握全局变量、局部变量、自动型变量、静态型变量的概念和使用方法。
- (5) 学习对多文件程序的编译和运行。

2. 实验内容和步骤

(1) 编写一个计算阶乘的函数。在主函数中，通过调用阶乘函数来计算 $2!+4!+\cdots+10!$ 。用递归调用和局部静态型变量两种方法编写函数。用 F7 键和 F8 键跟踪程序执行，同时观察变量值的变化。

(2) 分析下面的程序，判断输出结果并运行程序，从而验证分析是否正确。

```
#include<stdio.h>
int a=1,b=2;
int add(int x,int y)
{
    a=x+y;
    printf("a=%d,b=%d\n",a,b);
    return a;
}
main()
{
    int a=1,b=2,m=11,n=22;
    b=add(m,n);
    printf("a=%d,b=%d\n",a,b);
}
```

(3) 预习实验 7，将实验 7 的程序预先编写好。

实验 7 数组应用

1. 实验目的

- (1) 掌握一维数组和二维数组的定义、赋值和输入输出的方法。
- (2) 掌握字符数组和字符串函数的使用。
- (3) 掌握一些典型算法（如排序、查找、逆序等）。

2. 实验内容和步骤

编写程序并上机调试运行。



(1) 用选择法对 10 个整数进行排序。10 个整数用 `scanf()` 函数输入，排序以后再将数组按 5 个元素一行的格式进行输出。

(2) 修改上面的程序，接收键盘输入的一个数，对排好序的数组用折半查找法找出该数是数组中第几个元素。如果该数不在数组中，则输出 “can not find”。

(3) 找出二维数组元素最小值及最小值的所在位置。

实验 8 指针应用

1. 实验目的

(1) 掌握指针的概念，并掌握用指针处理所指向的变量或数组的方法。

(2) 掌握用指针处理字符串的方法。

(3) 掌握用指针变量做函数形参的方法。

2. 实验内容和步骤

编写程序并上机调试运行程序（要求用指针处理）。

(1) 编写函数实现将一个数组的各个元素值改变为原来的 2 倍。`Main()` 函数中定义两个数组，调用函数将两个数组的元素值都改变为原来的 2 倍。

(2) 将一个 3×3 的矩阵转置，用一函数实现该功能。在主函数中用 `scanf()` 函数输入以下矩阵元素：

```
11    12    13
21    22    23
31    32    33
```

以数组名作为函数实参的调用函数，在执行函数的过程中实现矩阵转置，函数调用结束后返回主函数，在主函数中输出已转置的矩阵。

(3) 用一个函数实现字符串的复制，即自己写一个 `strcpy()` 函数，函数原型为：

```
void strcpy(char *p1, char *p2);
```

设 `p1` 指向字符数组 `s1`，`p2` 指向字符串 `s2`，将 `s2` 的各字符复制到 `s1` 中。

字符数组 `s1` 和字符串 `s2` 由 `main()` 函数定义，`main()` 函数调用 `strcpy()` 实现字符串的复制。

实验 9 结构体与共用体

1. 实验目的

(1) 掌握结构体变量的定义和使用。

(2) 掌握结构体数组的概念和使用。

(3) 掌握链表的概念，初步学会对链表进行操作。

(4) 掌握共用体的概念与使用。

2. 实验内容和步骤

编写程序，然后上机调试运行。

(1) 有 10 个学生，每个学生的数据包括学号、姓名、3 门课的成绩，从键盘输入 10 个学生的数据，要求打印出 3 门课总平均成绩以及个人平均分最高的学生数据（包括学号、姓名、3 门课的成绩、个人平均分）

要求用 `input()` 函数输入 10 个学生数据；用 `average()` 函数求总平均成绩和个人平均分；用 `max()` 函数找出个人平均分最高的学生数据；总平均成绩和个人平均分最高的学生的数据都在主函数中输出。

(2) 建立一个链表，每个结点包括：学号、姓名、性别、年龄。输入一个学号，如果链表中的一个结点所包含的学号等于此学号，则将此结点删去。

(3) 输入和运行以下程序：

```
union data
{
    int i[2];
    float a;
    long b;
    char c[4];
}u;
main()
{
    scanf("%d,%d",&u.i[0],&u.i[1]);
    printf("i[0]=%d,i[1]=%da=%fb=%ldc[0]=%c,c[1]=%c,c[2]=%c,c[3]=%c",
        u.i[0],u.i[1],u.a,u.b,u.c[0],u.c[1],u.c[2],u.c[3]);
}
```

输入两个整数 10000、20000，分别赋给 `u.i[0]` 和 `u.i[1]`，分析运行结果。

然后将 `scanf` 语句改为：

```
scanf("%ld",&u.b);
```

输入 60000 给 `b`，并分析运行结果。

实验 10 文件

1. 实验目的

- (1) 掌握文件以及缓冲文件系统的概念。
- (2) 学会用文件指针对缓冲文件系统中的文件进行简单的操作。
- (3) 学会使用文件打开、关闭、读、写等文件操作函数。

2. 实验内容和步骤

编程实现小学二年级数学测试题的随机输出试卷的功能。题目要求是一个两位数和一个一位数的加减运算，且个位有往十位的进位或借位，高位没有往上的进位或借位，即结果仍为两位数。将程序产生的试卷以 `test.txt` 的文件名保存。参考程序如下：



```
#include<stdio.h>
#include<stdlib.h>
main()
{
    int a,b,i,j=1;
    FILE *fp;
    if((fp=fopen("test.txt","w"))==NULL)
    {
        printf("文件 test.txt 打开错误!");
        exit(1);
    }
    printf("文件 test.txt 成功打开");
    fprintf(fp,"\n\t\t\t\t\t=====二年级数学测试题=====\\n");
    /*写文件*/
    fprintf(fp,"\n\t\t\t\t\t班级_____学号_____姓名_____成绩_____\\n");
    fprintf(fp,"\\_____\\n");
    randomize;
    for(i=1;i<=1000;i++)/*产生 1000 个 0~99 之间的随机数*/
    {
        a=random(99);
        b=random(9);
        if(b<10&&a>10&&(a%10+b)>10)
        {
            fprintf(fp,"\\t%d+%d=\\t",a,b);
            j++;
            if(j%4==1)
                fprintf(fp,"\\n\\n");
        }
    }
    for(i=1;i<=1000;i++)
    {
        a=random(100);
        b=random(10);
        if(b<10&&a>10&&(a%10-b)<0)
        {
            fprintf(fp,"\\t%d-%d=\\t",a,b);j++;
            if(j%4==1)
                fprintf(fp,"\\n\\n");
        }
    }
    fclose(fp);
}
```

参 考 文 献

- [1] 谭浩强. C 程序设计. 北京: 清华大学出版社, 2000
- [2] 谭浩强, 张基温, 唐永炎. C 语言程序设计教程. 北京: 高等教育出版社, 2000
- [3] 谭浩强. C 程序设计解题与上机指导. 北京: 清华大学出版社, 2003
- [4] 廖雷. C 语言程序设计. 北京: 高等教育出版社, 2003
- [5] 孙宏昌, 王燕来. C 语言程序设计. 北京: 高等教育出版社, 1999