C 语言简明教程

计 算 机 教 程

青苹果电子图书系列

C 语言简明教程

陈　赞　　编著

张　晋　　审校

北京

前　言

C 语言是目前国内外使用最广泛的程序设计语言之一。它处理功能丰富、表达能力强、使用方便灵活、执行程序效率高、可移植性强；既有高级语言的特点，又有汇编语言的特点。它具有较强的系统处理能力，可直接实现对系统硬件和外部接口的控制。它采用了自顶向下、逐步求精的结构化程序设计技术。另外，它的函数式结构也为实现程序的模块化提供了强有力的保障。因此，它被广泛地应用于系统软件和应用软件的开发。

由于 C 语言涉及的概念和规则比较多，使用起来虽然灵活但是容易出错，不少初学者感到困难，希望能有一本合适的教材，本书就是为了满足这一需求而编写的。

全书共分九章，并有 C 语言语法提要及系统函数库等附录。

第一章为 C 语言简介，叙述 C 语言的发展历程、主要特点、C 语言的结构特点以及 C 语言的编译和执行过程等内容。第二章介绍的是用 C 语言编程必须掌握的一些基础知识，包括最基本的数据类型、对数进行运算的运算符和数据的输入输出。第三章介绍了 C 语言程序的三种基本结构：顺序结构、选择结构、循环结构。

第四~七章分别讲述了数组、指针、函数和结构体、联合和枚举等内容，这部分是本书的重点章节，反映了 C 语言主要的特征。部分内容相对来说较难掌握，例如指向函数的指针、结构类型的递归应用、联合的概念及应用等。

第八、九章分别介绍了 C 语言的标准函数库和文件系统以及 C 语言的预编译程序两部分内容。

最后要感谢所有为编写本书提供帮助的人，他们是协助调试程序的周华、彭小琦以及为本书绘制了大量插图的高英丽同志。

由于作者水平有限，书中难免有错误和疏漏之处，恳请广大读者批评指正。

编　者

内容提要

随着 C 语言的不断普及和广泛应用，学习它的人越来越多。本书共组织了 9 个章节的内容，介绍了 C 语言的基本概念、语法规则和利用 C 语言进行程序设计的方法，并提供了大量实例和习题，是广大用户学习和使用 C 语言的良师益友。

本书可作为大专院校师生学习 C 语言的教材，也可供参加计算机等级考试的读者作为辅导教材。

目　　录

第一章　C 语言简介

C 语言是目前国际上广泛流行的一种结构化程序设计语言，它既适合作为系统描述语言，也可用来开发应用软件。因此，它深受广大程序设计者的欢迎。

本章，我们将带领读者学习 C 语言的发展史，并且掌握 C 语言程序的结构特点、C 语言程序的编译和执行等内容，为后面的学习打下基础。

1.1　C 语言的历史及特点

1.1.1　C 语言的历史

C 语言诞生之前，操作系统等系统软件主要是用汇编语言编写的（包括 UNIX 在内）。由于汇编语言依赖于计算机硬件，程序的可读性和可移植性都比较差。为了提高可读性和可移植性，最好改用高级语言，但一般高级语言难以实现汇编语言的某些功能（汇编语言可以直接对硬件进行操作，例如，对内存地址的操作、位操作等）。人们设想能否找到一种既具有一般高级语言特性，又具有低级语言特性的语言。于是，C 语言就在这种情况下应运而生了。

C 语言是在 20 世纪 70 年代初由美国贝尔实验室的 D.M.Ritchie 设计的，最初的 C 语言是为描述和实现 UNIX 操作系统而提供的一种工作语言。

到了 1973 年，K.Thompson 和 D.M.Ritchie 两个人合作把 UNIX 的 90%以上内容用 C 语言进行了改写，即 UNIX 第五版。C 是为开发 UNIX 操作系统而研制的，它随着 UNIX 的出名而闻名。C 语言的广泛应用又不断推出新的 C 语言版本，其性能也越来越强。

到了 1975 年 UNIX 第六版的推出和随着面向对象程序设计技术的出现，C 语言的突出优点引出了人们的普遍关注。

1978 年以后，C 语言逐渐风靡全世界，成为世界上应用最广泛的计算机语言之一。到目前为止，又发展到演变出了目前可在微机上运行的 Microsoft C/C++、Turbo C、Quick C、Borland C、Visual C/C++等版本。

1.1.2　C 语言的特点

C 语言之所以能被推广并被广泛使用，概括地说主要有如下特点：

- 语言简洁，使用方便。C 语言一共只有 32 个关键字，9 种控制语句，程序书写形式自由，主要用小写字母表示，压缩了一切不必要的成分。
- 运算符丰富。C 语言的运算符包含的范围很广泛，共有 34 种运算符。C 语言把括号、赋值、强制类型转换等都作为运算符处理。从而使 C 语言的运算类型极其丰富，表达式类型多样化，灵活使用各种运算符可以实现在其他高级语言中难以实现的运算。
- 数据结构丰富，具有现代化语言的各种数据结构。C 语言的数据类型有：整型、实型、字符型、数组类型、指针类型、结构体类型、共用体类型等。能用来实现各种复杂的数据结构（如链表、树、栈等）的运算。尤其是指针类型数据，使用起来比其他高级语言更为灵活、多样。
- 具有结构化的控制语句（如 if…else 语句、while 语句、do…while 语句、switch 语句、for 语句）。用函数作为程序模块以实现程序的模块化。是结构化的理想语言，符合目前通行的编程风格要求。
- 语法限制不太严格，程序设计自由度大。例如，对数组下标越界不作检查，由程序编写者自己保证程序的正确。对变量的类型使用比较灵活，例如，整型变量与字符型数据以及逻辑型数据可以通用。一

般的高级语言语法检查比较严，能检查出几乎所有的语法错误，而 C 语言允许程序编写者有较大的自由度，程序员应当仔细检查程序，保证其正确，而不要过分依赖 C 语言编译程序去查错。“ 限制 ”与“ 灵活 ” 是一对矛盾。限制严格，就失去灵活性；而强调灵活，就必然放松限制。一个不熟练的人员，编一个正确的 C 语言程序可能会比编一个其他高级语言程序难一些。也就是说，对用 C 语言的人，要求对程序设计更熟练一些。

- C 语言允许直接访问物理地址，能进行位（bit）操作，能实现汇编语言的大部分功能，可以直接对硬件进行操作。因此 C 语言既具有高级语言的功能，又具有低级语言的许多功能，可用来写系统软件。C 语言的这种双重性，使它既是成功的系统描述语言，又是通用的程序设计语言。有人把 C 语言称为“ 高级语言中的低级语言 ”，也有人称它为“ 中级语言 ”，意为兼有高级和低级语言的特点。
- 生成目标代码质量高，程序执行效率高。一般只比汇编程序生成的目标代码效率低 10%~20%。
- 用 C 语言写的程序可移植性好（与汇编语言比）。基本上不作修改就能用于各种型号的计算机和各种操作系统。

C 语言的以上特点，读者现在也许还不能深刻理解，待学完 C 语言以后再回顾一下，就会有比较深的体会。

我们从应用的角度出发对 C 语言和其他高级语言作一简单比较：

从掌握语言的难易程度来看，C 语言比其他语言难一些，BASIC 是初学者入门的较好的语言，FORTRAN 也比较好掌握。

科学计算多用 FORTRAN；对商业和管理等数据处理领域，用 COBOL 为宜，C 语言虽然也可用于科学计算和管理领域，但并不理想，C 语言的特长不在这里。

对操作系统和系统实用程序以及需要对硬件进行操作的场合，用 C 语言明显地优越于其他高级语言，有的大型应用软件也用 C 语言编写。

从教学角度，由于 PASCAL 是世界上第一个结构化语言而曾被认为是计算机专业的比较理想的教学语言，目前在“ 数据结构 ” 等课程中一般用 PASCAL 语言举例，但 PASCAL 语言难以推广到各实际应用领域，到目前为止基本上只是教学语言，C 语言也是理想的结构化语言，用描述能力强，同样适于教学，而且“ 操作系统 ” 课程多结合 UNIX 讲解，而 UNIX 与 C 语言不可分，因此，C 语言有可能取代 PASCAL 而成为被广泛使用的教学语言，而且 C 除了能用于教学外，还有广泛的应用领域，因此更有生命力。

PASCAL 和其他高级语言的设计目标是通过严格的语法定义和检查来保证程序的正确性，而 C 则是强调灵活性，使程序设计人员能有较大的自由度，以适应宽广的应用面。

总之，C 语言对程序员要求较高。程序员使用 C 语言编写程序会感到限制少、灵活性大、功能强，可以编写出任何类型的程序。现在，C 语言已不仅用来编写系统软件，也用来编写应用软件。

1.2 C 语言程序的结构特点

任何一种计算机语言，都有特定的语法规则和表现形式，C 语言也不例外。程序的构成规则和书写格式则是其表现形式的重要方面，下面就来介绍 C 语言最基本的结构特点。

1.2.1 构成 C 语言的基本字符和标识符

C 语言规定了其所需的基本符号和标识符，这些是初学者首先应掌握的。

1. 字符集

满足 C 语言语法要求的字符集如下：

- 英文字母 a～z，A～Z；
- 阿拉伯数字 0～9；
- 特殊符号（如表 1-1 所示）。

表 1-1 C 语言中可以使用的特殊符号

+	-	*	/	%	_下划线	=	<
>	&	~	(	)	[	]	.
{	}	:	?	;	"	!	#
空格	'	|	^				

2．标识符

C 语言的标识符主要用来表示常量、变量、函数和类型等的名字，是只起标识作用的一类符号。它包括如下 3 类：

（1）保留字

所谓保留字，就是这样一类标识符，其每一个都有特定的含义，不允许用户把它们当作变量名使用，C 语言的保留字都用英文小写字母表示，共有 33 个保留字，如表 1-2 所示。

表 1-2 C 语言的保留字

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

（2）预定义标识符

除了上述保留字外，还有一类具有特殊含义的标识符，它们被用作库函数名和预编译命令，这类标识符在 C 语言中称为预定义标识符。一般来说不要把标识符再定义为其他标识符（用户定义标识符）使用。预定义标识符包括预编译程序命令和 C 编译系统提供的库函数名。其中预编译程序命令有：define、undef、include、ifdef、ifndef、endif、line。

（3）用户定义标识符

用户定义标识符是程序员根据自己的需要定义的一类标识符，用于标识变量、符号常量、用户定义函数、类型名和文件指针等。这类标识符主要由英文字母、数字和下划线构成，但开头字符一定是字母或下划线。下划线（_）起到字母的作用，它还可用于一个长名字的描述，如：

numbergoodstudent

可写为：

number_good_student

上面的写法中，用下划线把名词隔开，以增加可读性。

在 C 语言中，大小写字母的变量含义是不同的，如 TOTAL、Total、…、total 等是完全不同的名字。通常变量名用小写字母，常数名用大写字母。一个变量名字可由许多字符组成，但其长度是有限的，对于 ANSIC 只有前 31 个字符有效。对旧标准是前 8 个字符有效，例如 student_AAA 和 Student_BBB 编译程序把它们视为同一个名字。

为了使程序清晰、易读，建议在定义标识符时，应注意如下几点：

- 名字要有明确的含义，应尽量选用具有一定含义的英文单词来命名，使读者“见其名而知其意”。例如，代表总和的标识符用 sum 要比用 st 好，代表平均数的标识符用 average 而不用 a 等。如果所选用的英文单词太长，可采用公认的缩写方式。例如，圆周率用 PI 来命名。
- 标识符一般采用常用取简、专用取繁的原则。即常用的标识符应当定义为既简单又易识别的符号。
- 对于由多个单词描述的标识符，建议用下划线将各单词隔开，以增强可读性。例如，average_salary。
- 对于标识变量的标识符，可用特定的字符作其前缀来表示变量的数据类型。例如，用“i”表示整数、“I”表示长整数、“c”表示字符型、“sz”表示串类型等。

1.2.2 C 语言程序的实例

为了说明 C 语言程序的结构特点，先看几个简单的 C 语言程序实例，以便使读者有一个初步的认识。

【例 1-1】编写显示字符串“Hello! C Program”的 C 语言程序。

具体程序代码如下：

```
# include < stdio .h>
main（）
 {
     printf（"Hello ! C Program \ n "）；
}
```

这是一个最简单的 C 语言程序，它把字符串“ Hello! C Program ”显示在屏幕上。该程序由一个函数 main（）（叫主函数）构成。任何一个程序都必须有此函数，花括号{}所括的内容是 main 的函数体，每个 C 语言程序的函数都至少有一对{}。

printf（）是由系统提供的标准库函数，它完成输出功能，C 语言的输出是由函数来完成的，而与系统无关，这是它的特点之一。“ Hello! C Program ”是要输出的内容。“ \ n ”表示换行字符，它是由“ \ ”和“ n ”二字符构成，属转义字符，有关转义字符，在后面将会具体介绍。printf（）后的分号是语句结束符，C 的每一个语句都以“ ;”终止。

#include 是预编译程序命令，它把头文件“ stdio.h ”的内容展开在#include <stdio.h>所在的行位置处。其中，#include <stdio.h>也可以写成#include“ stdio.h ”，“ stdio.h ”文件中定义了 I/O 库所用到的某些宏和变量。因此，在每一个引用标准库函数的程序中都必须带有该#include <stdio.h>命令行。

【例 1-2】计算两个数之和的 C 语言程序。

具体程序代码如下：

```
# include <stdio.h>                          / * 计算两数之和的 C 语言程序 * /
main（）
 {
     float a，b，c；                          / * 定义 a，b，c 的数据类型为实型 * /
     printf（"Please input the two datas \ n：")；
     scanf（" % f % f"，&a，&b）；              / * 输入 a，b 两个数 * /
     c = a + b；                             / * 求和 * /
     printf（" \ n sum = % f \ n"，c）；
}
```

运行该程序时，首先提示输入两个数 a 和 b，然后计算出它们的和，并把结果以如下形式显示在屏幕上：

```
     sum=…
```

在此程序中，/ * … * /是一个注释语句，其中包含注释的内容，它在程序的编译过程中不产生任何执行代码，只是在编程中起到备忘录的作用。“ float a，b，c ;”是数据类型说明语句，它把 a、b 和 c 定义为实型数。值得注意的是，C 语言程序中的变量，在使用之前都要定义其数据类型。

“ scanf（…）;”是输入语句，scanf（）是格式化输入函数，它是一个由系统提供的标准库函数，其后的括弧内为参数表，“ % f % f ”为格式串，% f 表示实型数格式，指明给 a、b 等要求输入实型数。执行该语句时，数据从键盘上输入。

“ c=a+b ;”是赋值语句(或表达式语句),等号(=)是赋值运算符,表示把右边表达式的运算结果赋给 average。

“ printf（“ \ n sum=% f \ n ”，c）;”为输出语句，它首先在新的一行输出字符串“ sum= ”，然后按实型数格式（% f）输出变量 c 的值，并使光标移至下一行。

【例 1-3】求 a 和 b 两个数中的较大的值。

具体代码如下：

```
# include <stdio.h>                          / * 计算较大值的 C 语言程序 * /
main（）                                     / * 主函数 * /
 {
     int a，b，imax；                         / * 定义变量类型 * /
     printf（"please input two datas a，b：\ n"）；
     scanf（" % d % d "，& a，& b）；          / * 在此输入 a，b 的值 * /
     imax = max（a，b）；                     / * 调用求最大值的函数 * /
```

```
        printf（" \ n maximum is % d "，imax）；   / * 输出结果 * /
}
int max（x，y）；                          / * 求最大值函数 * /
int x，y；
{
    int m；
    if（x>y）
        m = x；
    else
        m = y；
    return（m）；                          / * 向主程序返回最大值结果 * /
}
```

此程序由两个函数组成，除了主函数 main（）之外，还有一个计算最大值的函数 max（）。“ int max（x，y）”说明函数的返回值类型为 int（整型），函数名字为 max，函数的参数为 x，y。“ int x,y; ”说明各参数的类型，“ return（m）”将求解结果返回给主函数。

该程序的执行是从 main（）函数开始，当主函数执行到 imax=max（a，b）语句时，控制被传递给 max（）函数，当执行 return（m）语句时，则结束 max（）函数，控制又被传递给 main（）函数，并把 max（）的计算结果带给 main（）函数。当主函数执行结束时，整个程序的执行也就结束了。

1.2.3 C 语言程序的结构特点

由上面几个简单的 C 语言程序实例，可以看出 C 语言程序的结构有如下几个特点：

1. C 语言程序由一个或多个函数组成

其中必须有一个主函数，主函数名为 main。其余函数的名字由程序设计者自定。

程序的执行是从主函数开始，其他函数都是在开始执行 main 函数以后，通过函数调用或嵌套调用而得以执行的。主函数是整个程序的控制部分。

主函数以外的其他函数可以是系统提供的库函数，也可以是用户根据自己的需要而编制的函数。为了便于程序设计，各种 C 语言的版本都提供了大量的库函数，供程序设计者引用。

2. 函数组成

C 语言函数的定义包括函数说明和函数体两个部分。函数说明指明函数的类型、属性、函数名、参数和参数说明等，如例 1-3 中的 max 函数的说明部分为：

```
int max（x，y）；
int x,y；
```

函数体是花括号所括的部分，它包括局部变量的说明语句一组执行语句。每个语句都由分号“ ；”结束。综上所述，一般函数的结构如下：

```
数据类型标识符 函数名（形参表）
形参说明：
{
    局部变量说明语句；
    执行语句；
}
```

3. 外部说明

在函数定义之外还可包含一个说明部分，该说明部分叫外部说明，它可包括预编译命令（如例 1-3 中的 #include）、外部变量的说明等。

1.3 C 语言程序的编译和执行

当把 C 语言程序编写好之后，就可以在机器上运行它了。我们把编写好的 C 语言程序叫 C 源程序。由于 C 语言是一种高级程序设计语言，它很容易被人们看懂和接受，但是，对于计算机来说，却不能接受这种语言，它只能接受机器语言。为此，首先必须把 C 语言程序翻译成相应的机器语言程序，这个工作叫编译。

1．源文件的编辑

为了编译 C 源程序，首先要用系统提供的编辑器建立一个 C 语言程序的源文件。一个 C 语言源文件是一个编译单位，它以文本格式存放在计算机的文件系统中（硬盘上）。源文件名自定，文件的扩展名（或后缀名）为“.c”。例如：

myfile.c

file.c

一个大的 C 语言程序往往可划分为若干模块，每个模块由不同的人或小组负责编写。对每个模块可建立一个源文件。因此，一个大的 C 语言程序可包含多个源文件。

2．编译

源文件建立好后，经检查无误后就可进行编译。编译是由系统提供的编译器完成，编译命令随系统的不同而异，具体操作时可参考相应的系统手册。例如，对于 Turbo C，一般通过 Turbo C 的编辑环境界面中的 Compile 菜单中的 Compile 命令进行编译，编译器在编译时对源文件进行语法和语义检查，并给出所发现的错误。用户可根据错误情况，使用编辑器进行修改，然后对修改后的源文件再度编译。用户也可以在 Compile 菜单中选 Make 命令进行编译，它能直接生成可执行的文件，此时如果系统发现用户的程序有语法错误，就发出错误的参考信息，提示用户进行错误代码的修改，然后用户再重新进行编译。

3．连接编辑

在上述步骤中，若用户选择 Compile 命令进行编译时，编译所生成的目标文件（*.obj）是相对模块，还不能直接执行，必须用连接编辑器把它和其他目标文件以及系统所提供的库函数进行连接装配，生成可执行文件存于文件系统中。可执行文件的名字可自由指定，扩展名为“.exe”。如图 1-1 所示，是 C 语言程序的操作过程。

4．执行

执行文件生成后，就可执行它了。若执行的结果达到预想的结果，则说明程序编写正确。否则，就需进一步检查修改源程序，重复上述步骤，直至得到正确的运行结果为止。

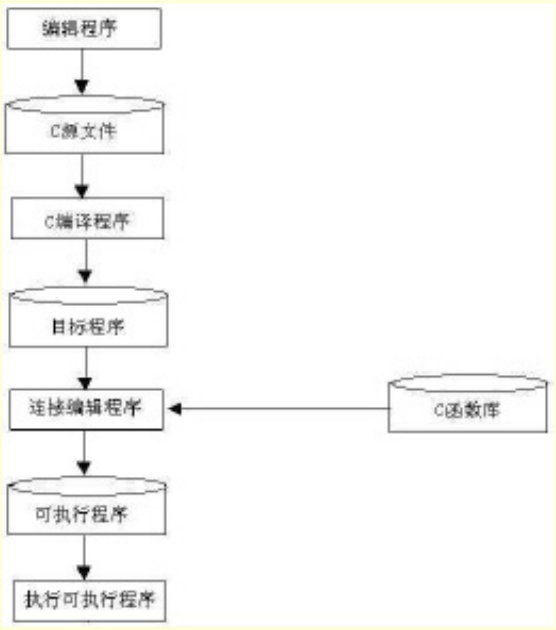


图 1-1 C 语言程序的操作过程

1.4 小 结

（1）C 语言的保留字是 C 语言中具有特定含义和用处的标识符。不能用保留字作变量名或函数名等。保留字都用小写英文字母表示。

（2）预定义标识符是 C 语言中的另一类特殊标识符，它们被用作库函数名，预编译命令等。这类标识符虽不是保留字，也准许程序设计者重新定义作用户定义标识符，但是，为了增强程序的可读性，最好还是不要这样做。

（3）用户定义标识符是程序设计者根据需要定义的一类标识符，用于标识变量名、函数名、类型名、符号常数名和文件名等。其构成规则是：

- 由英文字母和数字组成，但一定字母或下划线开头。
- 下划线“_”起字母作用，它还可用于分隔一个长的用户定义标识符。
- 大小写英文字母含义不同，习惯上是常数名用大写，变量名用小写。
- 不要用保留字作为用户定义的标识符，建议也不要用预定义标识符来定义用户定义标识符。
- 用户定义标识符的有效长度为 8 个字符。

（4）C 语言程序的结构特点如下：

- 它由一个或多个函数所组成，其中只能有一个主函数，其函数名为 main。
- 函数由说明和函数体两部分组成，说明部分用来说明函数的类型、名字、参数和参数说明等。函数体是由一对花括号括起来的程序块，它包括局部变量的说明语句（如果有的话）和一组可执行语句。每个语句末用分号终结。
- 在函数之外还包含一个说明部分，它包含外部变量的定义、说明和预编译程序命令。可在程序的任何位置加注释。

（5）在编写 C 语言程序时，一定要使书写的程序具有良好的风格，注意使层次清晰，以增强程序的可读性和可维护性。

习 题

1．C 语言的主要特点有哪些?

2．C 语言的主要用途是什么?它和其他高级语言有什么不同?

3．C 语言程序的结构特点有哪些?

4．编写一个程序，输出以下信息：

```
******************************
**     Hello,C Program!        **
******************************
```

5．编写一个程序，输入 a、b、c 三个值，输出其中最大者。

第二章　C 语言编程基础知识

本章介绍的是用 C 语言编程必须掌握的一些基础知识，包括最基本的数据类型、对数据进行运算的运算符和数据的输入输出。

C 语言中数据类型虽然比较丰富，但也很容易掌握；C 语言的运算符主要是大家比较熟悉的算术运算符和逻辑运算符；至于输入输出，输出内容就是对特定问题进行解答而得到的结果，输入内容就是对特定问题进行求解时所必须接受的数据。

2.1　C 语言的数据类型

数据类型是程序设计中的一个重要概念，它规定了该类型中数据的值域。例如，数值类型的数据，其值域是计算机所能表示的数的范围内的所有数据；逻辑类型的数据取值范围是“真”（TRUE）或“假”（FALSE）；字符类型的数据值域是某一字符集中的所有元素；指针类型的数据值域是计算机存储单元的绝对地址或相对地址的集合。

数据类型定义了一个运算集。例如，对数值型数据可施加算术运算；对逻辑型数据可施加逻辑运算；对字符型数据可施加连接和求子串运算；对指针型数据准许进行加、减运算，而不准许进行乘除运算等。当然，不同数据类型也可以进行混合运算，其结果为数据类型中字节最多的数据类型。

数据类型同时也定义了数据在内存中的存储方式。例如，一个字符型数据在计算机内存中占一个字节。长度为 n 的字符串在计算机内占用连续的 n+1 个字节。一个整型数在计算机内存中占 2 个字节，一个单精度浮点数在计算机内存中占 4 个字节，一个双精度浮点数在计算机内存中占 8 个字节等。

在高级语言中，每一个数据都属于一定的数据类型，不存在不属于某种数据类型的数据。

C 语言提供如图 2-1 所示的数据类型。数据包含常量和变量，它们都属于上述某种数据类型，本章主要介绍基本数据类型，其他数据类型将在以后的章节中逐步介绍。

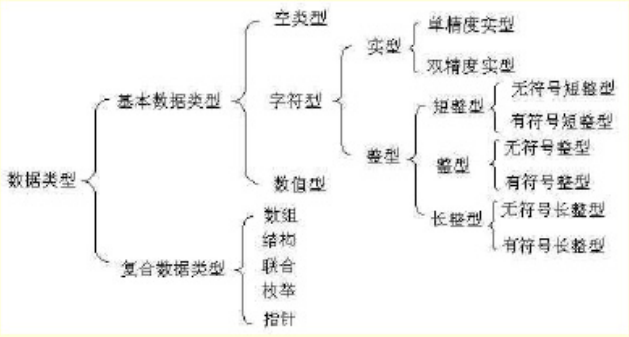


图 2-1　C 语言的数据类型

2.2　常　量

常量是程序中其值不发生变化的量。在 C 语言中，常量有数、字符和字符串三种。除此之外，C 语言程序中还经常使用另外一种表示形式的常量，叫符号常量，它由宏定义语句 define 来定义。

2.2.1 数

C 语言中的数有整型数和实型数两种，其中实型数还分为单精度实型数和双精度实型数，其有效数值范围也不一样。

1. 整型数

整型数有十进制数、八进制数和十六进制数三种，其表示形式如表 2-1 所示。

表 2-1 整型数的三种形式

进 制	表达方式	样 例
八进制数	由数字 0 开头	012，022
十六进制数	由 0x 或 0X 开头	0x12，0x22，0xff
十进制数	必须是数字 1，2…9 之中的一个数开头	12，22

值得注意的是，不同进制的数，表面上看数字是一样的，如 012、0x12 和 12，它所代表的真正数值是不同的。例如：

012 是八进制数，其代表的十进制数为 10；

0x12 是十六进制数，其代表的十进制数为 18；

12 是十进制数。

整型数又可分为正整型数和负整型数，分别在其前面加上：“ + ”或“ - ”表示，正整型数的符号“ + ”可省略。例如：

444，+666，-111，0222，-0222，0x3bf，-0x4de

整型数有短整型数、一般整型数和长整型数。对多数计算机系统而言，短整型数一般占用两个字节，一般整型数占用两个字节（即 16 位二进制位），其取值范围是-2^{15}～$2^{15}-1$ 即-32768~32767。超过该范围的整型数用长整型数表示。长整型数占用 4 个字节，其取值范围是-2^{31}～$2^{31}-1$，即-2147483648~2147483647。长整型数的表示方法是在数的末尾加一个字符 I 或 L，例如：

218I 333L 0x7dfL

2. 实数

实数又称之为浮点数，只用在十进制数中。它有单精度和双精度之分。其表示形式分为一般形式和指数形式两类，指数形式也称为科学计数法。一般形式的实数由整型数部分、小数点和小数部分所组成。例如：

3.14159 .0666-678.777 999.0 888． 0.88

指数形式的实数由尾数、e 或 E 和指数三部分组成，例如：

0.55e5 333E-3 8.88e+18

其中 0.55、3.33 和 8.88 为尾数，e 或 E 后面的 5、-3 和+18 均是指数。值得注意的是，用指数形式表示的浮点数必须有尾数。

2.2.2 字符常量

字符常量是用一对单引号括起来的单一字符，它在计算机的存储中占据一个字节。单引号是定界符，它并不是字符常量的一部分。下面显示的都是字符常量：

'a'，'A'，'?'，'2'，'*'

字符常量的值就是该字符在其所属字符集（如 ASCII）中的编码，例如'A'的 ASCII 值是 65，'a' ASCII 值是 97，'2'的 ASCII 值是 50 等。由于字符常量中的单引号（'）已作为定界符使用，于是单引号的字符常量表示形式为“ \' ”；而反斜杠（\）的字符常量表示形式为“ \ \ ”。所以“ ' ”和' \ '都是错误的表示形式。由于字符常量在计算机中是以其编码形式存放，因此可以说一个字符常量实际上是一个字节的整型数，它可以参与各种运算，例如：

x='b'；

y='b'+15；

z='?'+'A'；

它们分别相当于下列运算（在运算过程中将字符型数据转化成其相应的 ASCII 值参与运算）：

x=98；

y=98+15；

z=63+65；

在 C 语言程序中，字符常量通常用于字符之间的比较。

一般来说，字符可以直接写出，如“A”和“B”等。但制表符、回车－换行字符、退格字符等一些控制字符却不能直接写出，为此，C 语言提供了一类转义字符或称换码序列，用于表示那些无法在键盘上直接表示的字符。这些特殊字符有常用的控制字符（它们没有相应的印刷符号）和用于功能定义的字符（如单引号、双引号、反斜线等），用转义字符表示法就可在字符常量和字符串常量中书写它们了。常用的转义字符如表 2-2 所示。

表 2-2 转义字符

转换字符	含 义
\n	回车－换行
\t	水平制表（TAB）
\v	垂直制表
\b	退格字符（向左删除一字符）
\r	回车
\f	换页
\a	响铃警报
\'	单引号（'）
\“	双引号（”）
\0	空字符（NULL）
\ddd	表示一个字节的代码（或一个字符代码），其中 ddd 为三位八进制数，如\101 表示字符 A，\010 表示\b
\xddd	也是表示一个字节的代码（或一个字符代码），其中 ddd 为三位十六进制数，如\x041 表示字符 A，\x008 表示\b

2.2.3 字符串常量

字符串常量是用一对双引号括起来的一串字符，字符的个数称为字符串的长度，字符串常量简称字符串，下列所示都是字符串常量：

“Tsinghua”、“C Program”、“University”

长度为 n 的字符串，在计算机的存储中占用 n+1 个字节，分别存放各字符的编码，最后一个字节是 NULL 字符（或叫空字符，该字符在 ASCII 字符集中的编码为 0。为了书写方便，在 C 语言程序中用'\0'来表示该字符）。也就是说，任何一个字符串在机内都是以'\0'结尾。

双引号和反斜线字符在字符串中的表示形式类似单引号和反斜线在字符常量中的表示形式，应该以“\””或“\\”的形式出现，而不应该是“””或“\”。例如字符串：

“\”Tsinghua University\””

其中\”表示双引号字符

由上所述可知，字符常量与字符串常量在表示形式和存储形态上是不同的，例如'A'和“A”是两个不同的常量。字符常量'A'可以赋给字符型变量，而字符“A”只能赋给字符型数组。字符串“”表示空串，它在存储中占一个字节，其值为 NULL 字符的代码。

2.2.4 符号常量

在 C 语言程序中，可对常量进行命名，即用符号代替常量。该符号叫符号常量。符号常量一般用大写字母表示，以便与其他标识相区别。符号常量要先定义后使用，定义的一般格式是：

#define 符号常量 常量

例如：

```
# define NULL   0
#define EOF    -1
# define PI    3.1415926
```

这里的# define 是预编译命令，一个＃define 命令只能定义一个符号常量，且用一行书写，不用分号结尾。

符号常量一旦定义，就可在程序中代替常量使用。

【例 2-1】求圆柱体体积的程序。

```
# define        PI        3.1415926              / * 定义符号常量，即宏定义 * /
main（）
{
float r，h，v；
scanf（" % f % f "，& r，& h）；
v=PI * r * r * h；                              / * 求体积时用到了常量 */
printf（"Volume= % f "，v）；
}
```

使用符号常数有如下两点好处：

- 增强可读性：符号常量在程序中代替具有一定含义的常量，如用 EOF 代替-1（表示结尾），用 PI 代替 3.1415926 等，可增强程序的可读性，而且值在程序中不变。

- 增强程序的可维护性：如果一个大的程序中有多处使用同一个常量，这时可把该常数定义为符号常数。这样，当需要对某一个数在多处进行修改时，只需在其定义的地方做修改即可，不必做多处改变，这样可以避免偶然的错误。当调试、扩充或移植一个程序时，如果需经常改变某些常量的话，则把它定义为符号常量将大有好处。

2.3 数据类型及变量

数据类型及变量是 C 语言编程中首先面对的问题，下面将分别介绍。

2.3.1 基本数据类型

在 2.1 节中我们已简要地列出了 C 语言的数据类型。本节将进一步说明基本数据类型。C 语言的基本数据类型有如下不同的类型：

- 从长度上分，有 8 位、16 位、32 位和 64 位。
- 从数据的符号来分，有无符号数和有符号数。
- 按照数据的数学性质，分为整型、实型和字符型。

数值型数据的类型及表示形式等如表 2-3 所示。C 语言中各种基本数据类型的长度和范围随 CPU 的类型和编译器的实现不同而异，但对于大多数微机而言，其长度和范围如表 2-3 所示。

表 2-3 C 语言的基本数据类型

类型标识符	名 字	长度（二进制位）	范 围
Char	字符型	8	ASCII 字符代码
Unsigned char	无符号字符型	8	0 至 255
Signed char	有符号字符型	8	-2^7 至 2^7-1
Int	整型	16	-2^{15} 至 $2^{15}-1$
Unsigned int	无符号整型	16	0 至 $2^{16}-1$
Singned int	有符号整型	16	同 int
Short int	短整型	16（有的 8）	同 int
unsigned short int	无符号短整型	16（有的 8）	同 unsigned int
signed short int	有符号短整型	16（有的 8）	同 short int
long int	长整型	32	-2^{31}~$2^{31}-1$
singned long int	有符号长整型	32	同 long int
unsigned long int	无符号长整型	32	0 至 $2^{32}-1$
float	浮点	32	10^{-38}~10^{38}
double	双精度型	64	10^{-308}~10^{308}
void	空值型	0	无值

void 类型有两种用法：其一是指定函数返回值的类型，其二是用来设置类属指针。

2.3.2 变量及变量的定义

变量是在程序执行过程中，其值可能发生变化的一种量。每一个变量都对应计算机内存中相应长度的存储单元，以存放变量所取的值。下面是一个变量 x 在程序中的应用实例。

```
main（）
{
float x；
x=2.001；
x=2*x；
x=x+2.2；
printf（"f",x）；
}
```

每一个变量都用一个名字来标识，称之为变量名。变量名实质上是计算机内存单元的命名，变量的地址就是该内存单元的开始地址。变量名命名规则与用户定义标识符一样。

同常量一样，任何一个变量都属于某一数据类型。若它为整型变量，则在其作用域内只能取整型数值；若定义它为实型变量，则在其作用域内只能取实数值。

变量定义就是按照特定的方式为其使用的变量指定标识名、类型和长度等。在 C 语言程序中，所使用的每一个变量在引用前都必须先定义，否则编译程序将在程序编译时发出错误提示信息。

简单变量定义的方法是在类型标识后跟一个变量或变量表，变量之间用逗号分开，然后以分号结尾。下面是一些变量定义的例子：

```
int x，y，z；                / *定义整型数变量 x，y 和 z，变量表之间用逗号隔开* /
float sum，average；         / *定义实型变量 sum 和 average * /
double voleme；              / *定义双精度型变量 volume* /
unsigned long distance；     / *定义无符号长整型数 distance* /
char cl，c2；                / *定义字符型变量 c1 和 c2* /
```

2.3.3 变量的初始化

上述的变量定义只是指定了变量名字和数据类型，并没有给它们赋初值，给变量赋初值的过程称为初始化。值得注意的是，没有赋初值的变量并不意味着该变量中没有数值，而只表明该变量中尚未定义的值。因此，在使用该变量时，它所标识的内存单元尚保留先前使用该单元时留下的内容，于是，引用这样的变量就可能产生莫名其妙的结果，C 语言准许在定义变量时对其初始化。例如：

```
int a=2001；                  / *定义变量 a 为整型，初始值为 888 */
double p=15.5，d=0.1；        / *定义 p 和 d 为双精度实型，初始值分别为 15.5 和 0.1*/
float x ，y，z=4.53；         / *只给一个变量设置初始值*/
short int i=j=k=555；         / *给 i，j，k 设置同一个初始值 555*/
char c='a'；                  / *定义字符型变量 c，其值为字符 a*/
```

2.4 数据类型转换

在 C 语言的表达式中，准许对不同类型的数值型数据进行某一操作或混合运算。当不同类型的数据进行操作时，应当首先将其转换成相同的数据类型，然后进行操作。数据类型转换有两种形式，即隐式类型转换和显示类型转换。

2.4.1 隐式类型转换

所谓隐式类型转换就是在编译时由编译程序按照一定规则自动完成，而不需人为干预。因此，在表达式中如果有不同类型的数据参与同一运算时，编译器就在编译时自动按照规定的规则将其转换为相同的数据类型。

C 语言规定的转换规则是由低级向高级转换。例如，如果一个操作符带有两个类型不同的操作数时，那么在操作之前行先将较低的类型转换为较高的类型，然后进行运算，运算结果是较高的类型。更确切地说，对于每一个算术运算符，则遵循图 2-2 所示的规则。

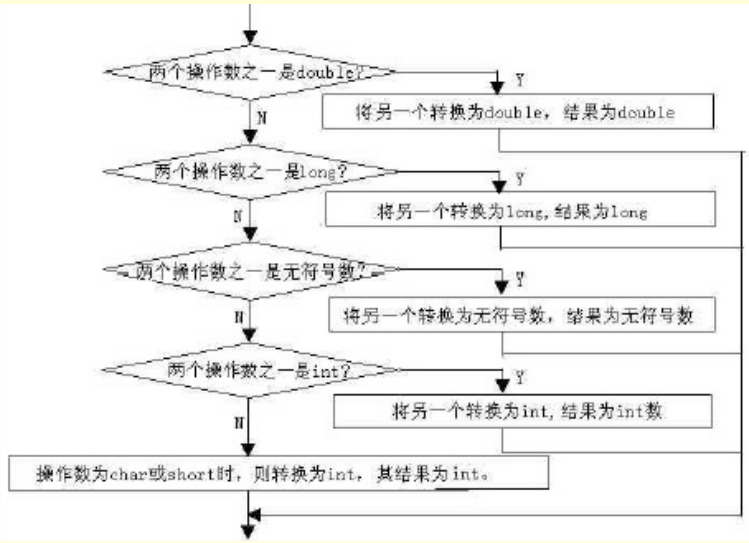


图 2-2 数据类型转换规则之一

注意：在表达式中，所有的 float 类型都转换为 double 型以提高运算精度。

在赋值语句中，如果赋值号左右两端的类型不同，则将赋值号右边的值转换为赋值号左边的类型，其结果类型还是左边类型。

因为函数参数是表达式，因此，当参数传递给函数时，也发生类型转换。具体地说，char 和 short 均转换为 int；float 转换为 double。这就是为什么我们把函数参数说明为 int 和 double，尽管调用函数时用 char 和 float .

也可以将图 2-2 所示的规则用图 2-3 表示。图 2-3 中的水平箭头表示必定转换，纵向箭头表示两个操作对象类型不同时的转换方向。

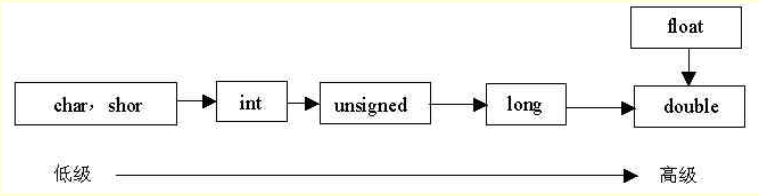


图 2-3 数据类型转换规则之二

下面举行说明类型转换的规则。例如执行：

```
x=100+'a'+1.5 * u+f/'b'-s * 3.1415926
```

其中，u 为 unsigned 型，f 为 float 型，s 为 short 型，x 为 float 型。式中右面表达式按如下步骤处理：

（1）首先将'a'、'b'和 s 换成 int，将 1.5 和 f 转换为 double 型。

（2）计算 100+'a'，因'a'已转换为 int 型，于是此运算结果为 197。

（3）计算 1.5*u，由于 1.5 已转换为 double，u 是 unsigned 型，于是首先 u 转换为 double，然后进行运算，运算结果为 double。

（4）计算 197+1.5 * u，先将 197 转换为 double（如 197.00…00），其结果为 double。

（5）计算 f/ 'b'，f 已转换为 double，'b'已转换为 int，于是先将'b'再转换为 double，其结果为 double。

（6）计算（197+1.5 * u）+f / 'b'，者均为 double，于是结果也为 double。

（7）计算 s * 3.1415926，先将 s 由 int 转换为 double，然后进行运算，其结果为 double。

（8）最后与前面得的结果相减，结果为 double。

（9）最后将表达式的结果转换为 float 并赋给 x。

2.4.2 显式类型转换

显示类型转换又叫强制类型转换，它不是按照前面所述的转换规则进行转换，而是直接将某数据转换成指定的类型。这可在很多情况下简化转换。例如，

```
int i；
…
i=i+9.801
```

按照隐式处理方式，在处理 i=i+9.801 时，首先 i 转换为 double 型，然后进行相加，结果为 double 型，再将 double 型转换为整型赋给 i。

```
int i；
…
i=i+（int）9.801
```

这时直接将 9.801 转换成整型，然后与 i 相加，再把结果赋给 i。这样可把二次转换简化为一次转换。

显示类型转换的方法是在被转换对象（或表达式）前加类型标识符，其格式是：

（类型标识符）表达式

例如，有如下程序段：

```
main（）
 {
int a，b；
float c；
b=a+int（c）；
printf（"b=d% \ n"，b）；
}
```

在上述程序的运行过程中，在执行语句 b=a+int（c）时，将 c 的值临时强制性转化为 int 型，但变量 c 在系统中仍为实型变量，这一点很重要，不少初学者在这个问题上忽略了这个问题。

2.5 运算符和表达式

2.5.1 运算符和表达式概述

1．表达式

一个表达式包含一个或多个操作，操作的对象称作运算元（或叫作操作数），而操作本身通过运算符体现的。例如 a、a-b、c=9.801 等都是一个表达式。

一个表达式完成一个或多个操作，最终得到一个结果，而结果的数据类型由参加运算的操作决定。最简单的表达式是只含一个常量或变量的表达式，即只含一个操作数而不含运算符。

C 语言中表达式的种类十分丰富，主要有如下一些：

- 算术表达式：进行一般的计算。
- 赋值表达式：进行赋值操作。
- 关系表达式：进行比较判断。
- 逻辑表达式：进行逻辑比较判断。
- 条件表达式：进行条件满足与否的判断。
- 逗号表达式：实际上是一种复杂运算，可以包含多个算术表达式。

2．C 语言的操作符

C 语言的特点之一是具有丰富和使用灵活的运算符，概括起来它有如下的几类运算符：

- 算术运算符。
- 赋值运算符（包括符合赋值运算符）。
- 关系运算符。
- 逻辑运算符。
- 条件运算符。
- 逗号运算符。
- 位运算符。
- 指针运算符。
- 求字节运算符（可以归并到函数的应用中去，它是通过函数 sizeof（）来进行运算的）。
- 强制类型转换运算符。

这些运算符如表 2-4 所示。

表 2-4　C 语言中的运算符

名　称	操作符
自增，自减	+ +，- -
逻辑与、或、非	& &，｜｜，!

续　表 2－4

名　称	操作符
指针操作及引用	*，&
加、减、乘、除、求模运算	+，－，*，/，%
关系操作符	<,<=,>,>=,= =,! =
按位与、或、异或、求反	&,｜,^,~
逗号表达式	，
类型转换	（）
移位运算	< <,> >
条件运算	？ ：
求占用的字节数	sizeof
赋值	=,+ =,－ =,*=,/ =,% =

2.5.2　算术运算符及算术表达式

1．基本算术运算符

+、－、*、/和求模（%）运算为基本算术运算符，当进行两个整型数相除时，其结果仍为整型数．小数部分被略去。例如：

```
27/5=5
27/7=3
```

取模运算符“％”用于计算两个数相除后得到的余数，只适用于两个整型数取模，不能用于其他数据的运算。在“x%y”表达式中，x 是被除数，y 是除数。

2．自增自减运算符

自增“++”和自减“——”运算符为变量的增 1 和减 1 提供了紧凑而方便的表达形式。但这两种运算符都有前置和后置之分，其一般用法如下：

```
i++或++ i
i——或——i（相当于 i=i－1）
```

其中 i 是一个整型变量。对一个变量实行前置或后置运算其结果是相同的，即都使它增 1 或减 1。前置运算是在该变量参与其他运算之前先增 1 或减 1，而后置运算是在它参与其他运算之后才增 1 或减 1。这一点是要区别的。例如：

```
k=3；
j=5；
i=3；
m=（++k）* j；
```

```
n=（i++k）* j；
```

在计算 m=（++k）* j 时，k 首先增 1 变为 4，然后与 j 相乘，最后将 20 赋给 m。在计算 n=（i++k）* j 时，由于对 i 实施的是后置运算；因此 i 是用 3 的值参与乘运算的。于是（i++）*j=3*5=15。在参与乘操作之后，i 才增 1 变为 4；最后赋给 n 的值是 15。下边是自减的例子：

```
i=4；
j=5；
k=4；

m=（i--）* j；
n=（--k）* j；
```

在计算 m=（i--）* j 时，由于对 i 实施的后置减 1，所以（i--）* j＝4*5＝20。i 在参与乘之后才减 1，此时 m=20，i 的值变为 3；在计算 n 时，k 首先减 1，变为 3，然后与 j 进行乘，结果为 15，于是 n=15。

使用自增自减运算时，应注意如下几点：

- 自增自减运算的操作数一定是变量，而不能是其他表达式。例如，

```
i++              / *是合法的* /
（i+j）++        / *非法的，自增或自减只对变量进行* /
10++             / *是非法的* /
```

- 一个变量的前、后置运算只表明该变量参与其他运算与其自身变化之间的先后关系，并不影响它在表达式中参与其他运算的顺序。
- “ ++ ”和“ -- ”运算符是自右向左结合的，单目运算符“ - ”也是自右向左结合的。因此，对-i++就理解为-（i++），而不应理解为（-i）++。（-i）++是非法的，因为它是表达式而不是变量。
- 自增和自减运算常用于数组下标改变循环次数控制。例如，

```
x=10；
i=5；
…
a［i++］＝x；              / *把 10 赋给 a［5］，然后 I 变为 6*/
```

对于下面的运算：

```
int m=5，n；
n=（m++）+（m++）；
```

其计算过程是先进行 m 的三次相加，5+5+5=15，即 n=15，然后再进行 m 的三次自加，最后 m 的值为 6。对于下面的运算：

```
int p=5，q；
q=（++p）+（++p）+（++p）；
```

其计算过程是先进行三次的 p 的自加，此时 p 的值为 8，然后在自加后 p 等于 8 的基础上进行 p 的三次相加，即 8+8+8＝24，即 q 的值为 24。

3. 算术表达式

算术表达式是由算术运算符和操作数所组成的表达式。在计算算术表达式时，其计算顺序应按照操作符的优先次序进行，如果有括号的话，则括号应配对。对于自增减运算，要注意运算符的结合性，其中运算符++、--和-是自右向左结合，+、-、*、/和%是自左向右结合。

2.5.3 赋值运算符和赋值表达式

赋值运算符包括简单赋值运算符和复合赋值运算符，复合赋值运算符又包括算术复合赋值运算符和位复合赋值运算符，如表 2-5 所示。

表 2-5 赋值运算符

运算符分类	名 称	赋值运算符	举 例	等价于
简单赋值	赋值	=	Y=x	
算术复合赋值	加赋值	+=	Y += x	Y = y-x
算术复合赋值	减赋值	-=	Y -= x	Y = y-x
算术复合赋值	乘赋值	*=	Y *= x	Y = y*x
算术复合赋值	除赋值	/=	Y /=x	Y = y/x
算术复合赋值	取余赋值	%=	Y %= x	Y = y%x
位操作复合赋值	位与赋值	&=	y&= x	Y= y&x
位操作复合赋值	位或赋值	|=	y |= x	y = y | x
位操作复合赋值	位异或赋值	^=	Y ^=x	Y = y > >x
位操作复合赋值	右移赋值	>>=	y >>= x	Y = y > > x
位操作复合赋值	左移赋值	<<=	Y <<= x	Y = y < < x

1．赋值运算符

赋值运算符“=”是将其右边表达式的值赋给左边的变量，赋值号左边一定是变量，右边是表达式。如果右边表达式的类型与左边变量的类型不一致时，则先将右边表达式的值转换为与左边变量相同的类型，然后进行赋值。例如：

```
i=d+3
```

其中 i 为 int 型，d 为 double 型。于是此运算的处理过程是先将 3 转换为 double 型（3.0），再执行 d+3.0，结果为 double 型，最后再把 double 的结果转换为 int 型，并赋给 i。

2．赋值表达式

由赋值运算符将一个变量和一个表达式连起来的式子叫赋值表达式。赋值表达式的一般形式如下：

```
v op expr
```

其中 v 是变量，op 是赋值运算符，expr 是表达式。例如，“x=10”是一个赋值表达式，其处理过程是：先计算赋值号右边表达式（10）的值，其值为 10，再将 10 赋给 x，于是表达式“x=10”的值是 10，其中 x 的值也是 10。赋值号右边的表达式还可以是赋值表达式。例如，下式是合法的：

```
x=10*（y=5）
```

按照赋值操作的结合规则（自右向左结合），首先处理表达式“（y=5）”，该表达式的值是 5，整个赋值表达式的值为 50。

赋值表达式中也可以包含复合赋值运算符，例如：

```
x+=y+=z*z
```

设 x，y 和 z 的初始值分别为 10，20 和 30，此例等价于

```
x=x+（y=y+z*z）；
```

其中（y=y+z*z）的值为 920，y 的值也为 920，于是整个表达式的值为 930，x 也为 930。对于如下的运算：

```
int t=5；
t+=t-=t*t；
```

对于这样的表达式，其运算要注意其顺序，这样的表达式是右结合式，即先计算：

```
t-=t*t；
```

它相当于：

```
t=t-t*t；
```

此时的结果是：

```
t=5-5*5=-20
```

然后再计算：

```
t+=-20；
```

最后的结果是 t=-40。

【例 2-2】熟悉数据类型及其用法（本例重点要求掌握在输入、输出过程中容易出错的地方）。

```
main（）
 {float x，y；                    /* 定义 x 和 y 为实型变量 */
```

```
int i，j，k，m，n，p，q，s，t；        / * 定义一系列整型变量 * /
x=3.6；                                / * 对 x 值进行初始化 * /
i =（int）x；                          / * 强制类型转换 * /
y =（float）i；                        / * 强制类型转换* /
printf（"1- - [x=3.6，i =（int）x，y=（float）i] - -x=% f i=% d y=% f \n"，x，i，y）；
k=i；
printf（"2- - [k=i] - - -k=% d \ n"，k）；
j= + + i；
printf（"3- - [j= + + i] - - - j= % d \ n"，j）；
m=k + +；
printf（"4- - [m=k + +] - - -m=% d k= % d \ n"，m，k）；
p=m；
printf（"5- - [p=m] - - - p=% d \ n"，p）；
n=（m+ +）+（m+ +）+（m+ +）；
printf（"6- - [n=（m + +）（m+ +）（m+ +）] - - -m=% d n=% d\n"，m，n）；
q=（+ +p）+（+ +p）+（+ +p）；
printf（"7- - [q=（+ +p）（+ +p）（+ +p）] - - -p=% d q=% d \ n"，m，n）；
s=q+ + +p；
printf（"8- - [s=q+ + +p] - - - s= % d p= % d q= % d \ n "，s，p，q）；
printf（"9- - s= % d s= % d s= % d \ n"，s，s+ +，s- -）；
t=s;
printf（"10--[t=s]---t=% d \ n",t）;
t + = t -= t * t;
printf（"11--[t + =t - =t * t]---t=% d \ n",t）;
}
```

程序运算结果如下：

```
1 - - [ x = 3.6，i =（int）x， y =（float）i ] - - x =3.6 i=3 y = 3.0
2 - - [ k = i ] - - - k = 3
3 - - [ j = + + i ] - - - j = 4
4 - - [ m= k + + ] - - - m = 3 k = 4
5 - - [ p = m ] - - - p =3
6 - - [ n = ( m + + ) + ( m + + ) ( m + + ) ] - - - m = 6 n = 9
7 - - [ q = ( + + p ) + ( + + p ) + ( + + p ) ] - - - p = 6 q = 18
8 - - [ s = q + + + p ] - - - s = 24 p = 6 q = 19
9 - - s =24 s= 23 s =24
10 - - [ t =s ] - - - t = 24
11 - - [ t + = t - = t * t ] - - - t = - 1104
```

上例中，语句：

```
s=q+++p；
```

相当于：

```
s=（q++）+p；
```

由于 q 的自加运算是后运算，因此，s 的值应该是 q+p 的值，然后 q 的值再自增 1。

对于语句：

```
printf（"9--s=%ds=%d\n"，s，s + +，s - -）；
```

它是右向结合，因此，先输出 s-的运算结果，然后输出 s++的运算结果，最后输出 s 的值。

2.5.4 关系运算符和关系表达式

1. 关系运算符

关系运算符是对两个操作量进行大小比较的运算符，其操作结果是“真”或“假”。由于 C 语言中没有逻辑类型的数据，所以通常以非零表示真，实际上经常用整型数“1”表示“真”，“0”表示“假”。C 语言中有六种关系运算符，即：

- ＞＝（大于等于）
- ＜＝（小于等于）
- ＝＝（等于）
- ！＝（不等于）
- ＞（大于）
- ＜（小于）

2. 关系表达式

关系表达式就是用关系运算符把操作对象连接起来而构成的式子，操作对象可以是各种表达式，对于关系表达式或逻辑表达式，应把其值理解为 1（真）或 0（假）。例如表达式：

5＞（4＜5）

由于（4＜5）是“真”，所以其值为 1，于是该表达式成立，其值为 1（即“真”）。

又如，假设 x=10，y=5，则表达式

x= =y+5　　其值为“真”

（x=3）<5+y　　其值也为“真”

x=y<y+5　　则 x=1，因为 y<y+5 是“真”

下边都是合法的关系表达式：

a>b　　a+5>b-3　　（a=100）>（y=50）　　'a'<'b'　　（a>b）>、（b<c）

2.5.5 逻辑运算符和逻辑表达式

1. 逻辑运算符

逻辑运算符是对逻辑量进行操作的运算符。逻辑量只有两个值，“真”和“假”，它们分别用 1 和 0 表示。C 语言中有三个逻辑运算符，即：

- ！（逻辑非）
- ＆＆（逻辑与）
- ｜｜（逻辑或）

逻辑运算符＆＆和｜｜是双目运算符，！是单目运算符，它们的操作对象是逻辑量或表达式（可以是关系表达式或逻辑表达式），其操作结果仍是逻辑量。例如：

x && y 当 x 和 y 均为“真”时，其结果为“真”，只有二者均为“假”时，其结果才为“假”。

x | | y 当 x 和 y 之一为“真”时，其结果为“真”，只有二者均为“假”时，其结果才为“假”。

！（a>b）当（a>b）为假时，其结果为“真”，否则为“假”。

（x>y）&&（a>b）当 x>y 和 a>b 之中有一个满足时，其结果为真。

（x>y）| |（a>b）当 x>y 和 a>b 之中有一个满足时，其结果为真。

2. 逻辑表达式

逻辑表达式是用逻辑运算符把操作对象（可以是关系表达式或逻辑表达式）连起来所构思的一种运算式子，其操作结果是“真（非零）”或“假（零）”。在处理逻辑表达式时要注意逻辑运算符的优先级及结合性，逻辑运算符的优先级从高到低的顺序依次是“！”、“&&”、“｜｜”。“&&”和“｜｜”的优先级低于关系运算符和算术运算符，而“！”高于基本算术运算符。“&&”和“｜｜”的结合性是自左至右，而“！”是自右至左，例如：

x>y&& a<c-5　　相当于（x>y）&&（a<c-5）

x! =y && a>=c+5　　相当于（x! =y）&&（ a>=c+5）

! x && a = = c　　相当于（! x）（&& a = = c）

有时为了提高程序的可读性，经常把逻辑运算符两边的表达式加上一对括号，如上面的“x>y&& a<c-5”就

可以写成“（x>y）&&（a<c-5）”，这样，当逻辑运算符两边的表达式为复杂的表达式时，容易辨认和阅读。

在 C 语言中，逻辑表达式或关系表达式的取值，“真”和“假”用 0 表示。但在处理逻辑表达式中，当判断一个量是否为“真”时，是看它是否为非 0，对于非 0 则视为“真”，对于 0 则视为“假”。例如：

当 x=5，y=1.5，z='a'时，则！x，！y，！z 均为“假”，即为 0。

当 x=5，y=3 时，x & & y 的值为 1，因为 x 和 y 均为非 0。

2.5.6 三项条件运算符

三项条件运算符是 C 语言中惟一具有三个操作对象的运算符，它具有如下的语法形式：

表达式 1？ 表达式 2： 表达式 3

其处理过程是先处理“表达式 1”，当“表达式 1”为“真”时，则对“表达式 2”进行计算；否则对“表达式 3”进行计算。用三项条件运算符构成的表达式叫三顶条件运算表达式，其结果是一个算术值，它或者是“表达式 2”，或者“表达式 3”。例如：

```
a>8？b+10：b-20
```

先处理条件表达式 a>8，当它为“真”时，则计算 b+10，而此时该三项条件运算表达式的值为 b+10；否则计算 b-20，该三项条件运算表达式的结果为 b-20。在处理三项条件运算表达式时应注意如下几点：

- 在三项条件运算表达式中，“表达式 1”是条件表达式，而“表达式 2”和“表达式 3”可以是其他的表达式
- 三项运算表达式准许嵌套，这种嵌套是右结合的，即先计算右边的三项运算。例如：

```
int a，b，c，d，e
a=15；
b=20；
c=25；
d=30；
e=a>b？c：c>d？b：d
```

按照三顶条件运算符的结合规则，上边二式相当于：

```
e=a>b？c：（c>d？b：d）
```

上述表达式中，先求解（c>d？b：d），由于 c 比 d 小，“表达式 1”的结果为“假”，因此其结果为“表达式 3”的值，即 30，然后求解 e=a>b？c：30，由于 a 比 b 小，此时“表达式 1”的值仍为假，结果仍取“表达式 3”的值，最后结果 e=30。

- 在程序中，常把三顶条件运算表达式的结果赋给某个变量。例如：

```
result=（x % 2= = 0）？0：1；
```

当 x 是偶数时，则 result＝0，否则 result＝1。

```
y=x>0？x：-x
```

该三项条件计算表达式将 x 的绝对值赋给 y。

```
ch=（c>='a' & & c<='z'）？c-'a'+'A'：c
```

该式把 c 中字母转换为大写赋给 ch 变量。

2.5.7 其他运算符

1．逗号运算符和逗号表达式

C 语言中逗号也是一种运算符。用逗号把几个运算表达式连接起来所构成的表达式叫逗号表达式。例如：

```
a=15，b=a*5，z-=y，a+6
```

上式是下一个逗号表达式。它是由四个表达式结合而成。逗号表达式的运算次序是自左而右逐个进行运算，最后一个表达式的结果就是逗号表达式的运算结果。例如逗号表达式：

```
a=15，a*10，a+8
```

上式结果是 158。

在 C 语言程序中，也可以用逗号表达式来给一个变量赋值。例如：

z=（x=15，y=x+25，y * x + 30）

那么 z 的值是 630。

2．求字节数运算符

sizeof 是求其操作对象所占用字节数的运算符。它是在编译源程序时，求出其操作对象所占字节数的，它实际上是一个函数。其操作对象可以是类型标识符，也可以是表达式。它有如下两种表达形式：

sizeof（类型标识）

sizeof 表达式

如：

sizeof（double）的值是 8，表明双精度浮点数占用 8 个字节。

float b［10］；

sizeof（b）的值是 40，因为一个 float 对象占 4 个字节。

sizeof（char）的值为 1，说明字符型数据占 1 个字节。

2.6 位运算符

由于 C 语言是介于高级语言和汇编语言之间的一种中级语言，它是为开发系统软件而设计的，它可以直接对地址进行运算，因此，C 语言提供了位运算的功能。

计算机中的位运算是针对二进制代码进行的。每一个二进制位的取值只有 0 或 1。位运算符的操作对象是一个二进制位集合，如一个字节（8bit）。C 语言提供了如表 2-7 所示的位运算符。

表 2-7 位运算符

运算符	名称	使用格式
~	按位取反	~表达式
< <	左移位	表达式 1< <表达式 2
> >	右移位	表达式 1> >表达式 2
&	按位与	表达式 1 &表达式 2
^	按位异或	表达式 1 ^表达式 2
|	按位或	表达式 1 |表达式 2

2.6.1 按位取反运算符

按位取反运算符就是将其操作对象中的据有二进制位全部改变状态，即“逢 0 变 1，逢 1 变 0”。例如，八进制数 0217，（即二进制 10001111），其按位取反后为八进制数 0160（即二进制位 01110000）。所以~0217 的值是 0160。又如，

```
unsigned char x=0137；          / *即二进制 01011111 * /
x=~ x                           /* x 的二进制结果为 10100000 * /
```

2.6.2 移位运算符

移位运算符有左移运算符和右移运算符。

1．左移运算符

左移运算符是将其操作对象向左移动指定的位数，每左移 1 位相当于乘以 2，移 n 位相当于乘以 2 的 n 次方。一个二进制位组在左移时右边补 0，移几位右边补几个 0。其一般书写格式为：

表达式 1＜＜表达式 2

其中“表达式 1”是被左移对象，“表达式 2”给出左移位数。例如，表达式 x << 4 的结果就是将 x 左移 4 位。左边移出的位被舍弃。例如，表达式 0377＜＜4，其结果为 0360，即 11111111 左移 4 位后为 11110000。

2．右移运算符

右移运算符是将其操作数向右移动指定的位数，右移操作相当于除以 2，右移 n 位相当于除以 2 的 n 次方。在进行右移时，右边移出的二进制位被舍弃。其一般书写格式为：

表达式 1＞＞表达式 2

其中“表达式 1”是被移对象，“表达式 2”给出移动位数。例如，表达式 x＞＞2 的结果就是将 x 右移 2 位。例如，表达式 0377＞＞4 的结果为 017。它相当于将二进制数 11111111 右移 4 位，结果为 00001111。

2.6.3 按位“与”、按位“或”、按位“异或”

1. 按位“与”

按位“与”的一般书写格式为：

表达式 1＆表达式 2

其中“表达式 1”和“表达式 2”均为整型表达式。

按位“与”遵循这样的原则：当两个操作对象的相应二进制位都为 1 时，则该位的结果为 1，否则为 0，即“两 1 为 1，其余为 0”。值得注意的是，按位“与”的运算，两个表达式之间用一个“＆”符，而逻辑“与”，两个表达式之间用两个“＆＆”，这是初学者经常出错的地方。例如：

15＆26

第一个操作数的二进制表示为 00001111，第二个操作数的二进制表示为 00011010，按位“与”后的二进制表示为 00001010。

我们可利用按位与来获取指定位的值。例如，假设 x 是一般的 unsigned 类型的整型数（2 个字节），我们想获取其低字节的值时，只需将 x 与 0377 相与既可。例如：

```
y=x & 0377
```

其中 x 的二进制表示为 0101000101110010，0377 的二进制表示为 0000000011111111，y 的二进制表示为 0000000001110010 还可以利用按位“与”来测试指定的位是否为 0。例如，想测试上例中 x 的从左数第四位是否为 0，则只需将它与 010000（八进制）按位“与”(即 y=x &010000)。若 y 为 0 说明被测位为 0，否则为 1。x 的二进制表示为 0101000101110010，010000 的二进制表示为 0001000000000000，计算结果为 0001000000000000。

2. 按位“或”

按位“或”遵循这样的原则：当两个操作对象的相应二进制位都为 0 时，则该对应位的按位“或”结果才为 0，否则为 1，可以简记为“两 0 为 0，其余为 1”。例如：

35｜41：

35 的二进制表示为 00000000001000011，41 的二进制表示为 0000000000101001，按位“或”的结果为 0000000000101011。

按位“或”的一般书写格式为：

表达式 1｜表达式 2

其中“表达式 1”和“表达式 2”是整型表达式。

我们可利用按位“或”来将指定位设置为 1。例如，将 x 的从右数第三位（二进制位）设置为 1，则只需执行如下表达式即可：

```
x = x | 04
```

x 的二进制表示为 0111100001000011，04 的二进制表示为 0000000000000100，按位“或”的结果为 0111100001000111。

3. 按位“异或”

按位“异或”遵循这样的原则：当其两个操作对象的相应位相同时，则该对应位“异或”的结果为 0。可以简记为“相同为 0，不同为 1”，即 0 ^ 0＝0，0 ^ 1＝1，1 ^ 1＝0。按位“异或”也可称不进位加，即两个操作对象执行二进制相加，但不向高位进位。例如：

73 ^ 81

73 的二进制表示为 0000000001001001，81 的二进制表示为 0000000001010001，按位“异或”结果为 0000000000011000。所示“异或”的意思是判断两个相应的二进制位是否相“异”。如果相“异”则结果为“真”（即为 1），否则为“假”(即为 0)。利用按位“异或”可使一个数的各二进制位翻转。例如要使 x 的各位翻转，只需执行如下的表达式：

x=x ^ 0177777

其中：x 的二进制表示为 0101000101110010，“异或”后的结果为 1010111010001101。

2.7 C 语言的基本输入/输出函数

C 语言中的基本输入输出函数，是初学者必须熟练掌握的基本内容之一。由于 C 语言本身不像其他某些高级语言一样有输入和输出语句，其输入和输出是由标准的输入和输出函数完成的。本节将介绍标准函数库中部分常用的输入和输出函数，这些函数的原则均在特定的头文件中定义，因此，使用输入和输出函数，在程序的开头要嵌入相应的头文件。要正确使用各个函数，必须从以下几方面理解它们：

- 函数的引用格式
- 函数的功能
- 函数参数的个数、类型、含义及顺序
- 返回值的类型及含义
- 要使用的头文件

标准函数虽然是外部函数，但引用时用户不需另加说明，因为他们的说明已包含在相应的头文件中。引用标准文件的输入/输出函数时，要求使用 stdio.h 头文件。

2.7.1 字符输入/输出函数

1．字符输入函数 getchar（）

（1）功能

该函数从标准输入设备（通常是键盘）上读入一字符。

（2）调用格式

c=getchar（）

当执行此函数调用语句时，变量 c 获得一个从标准设备上读取的字符代码值。当从键盘上输入 ^ z（即 ctrl 与 z 键同时按下）时，c 得到的值是-1。 ^ z 称文件结尾，在程序中经常使用符号常量 EOF 表示（End of File）。用此函数时，也要求在程序第一行有预编译命令（也称嵌入头文件）＃include <ctdio.h>。

【例 2-3】getchar（）函数的应用。

```
# include < stdio.h >
main（）
{
int c;
printf（"input a character:"）；
c=getchar（）；                                  / * 从键盘读入字符 * /
printf（"The character inputed is % c\ n",c）；    / * 在屏幕上输出字符 * /
}
```

运行该程序时，提示作者输入一个字符（input a character），然后在终端上以字符格式输出所输入的字符。其操作及输出情况如下：

```
input a character：A＜CR＞
A
```

2．字符输出函数 putchar（）

（1）功能

该函数向标准输出设备（通常是显示终端）输出一字符。使用此函数时，要求在程序首行有#include <stdio.h>预编译命令。

（2）调用格式

putchar（c）；

其中 c 是一个字符型变量或整型变量，其值被看作是要输出字符的代码，它被输出到显示终端上。

【例 2-4】利用 putchar（）函数将字符输出到显示终端上。

```
# include < stdio.h >
main（）
{
int a;                                  / *  定义整型变量名  */
a=98;                                  / *  给变量赋值  */
printf（"output the character \ n"）;
putchar（a）;                            / *  输出字符  * /
}
```

执行该程序，将把字母 b（其 ASCII 码为 98）输出在显示屏上，其输出形式如下：

```
output the character
b
```

【例 2-5】把输入的小写字母变成大写的形式，然后显示在屏幕上。

```
# include < stdio.h >
main（）
{
int c;
while（（c =getchar（））  ! =EOF）
if（ c >= 'a' & & c < = 'z '）
putchar（c - 'a '+ 'A '）;                    / *  小写变大写的算法  */
else
putchar（c）;                           / *  大写保持不变  * /
}
```

2.7.2 字符串输入/输出函数

1．字符串输入函数 gets（）

（1）格式

```
gets（char * s）
```

（2）功能

该函数从标准输入文件（一般是键盘）读取下一个字符串，存入 s 所指向的内存区内。当输入遇到＜CR＞字符时，结束串的输入，并自动将＜CR＞字符转换为' \ 0 '（即 NULL），存放在输入串的末尾，使其构成下一个字符串。

（3）参数说明

s 是一个字符型指针，它指向所取字符串的首地址。

（4）返回值

正常返回时，取到字符串的首地址。如果遇到文件尾或出错时返回 NULL。NULL 定义的形式为“ #define NULL 0 ”，它包含在 stdio.h 头文件中。

【例 2-6】输入一个字符串，然后在屏幕上输出。

```
# include < stdio.h >
main（）
{
char str[100];
if（gets（str）  ! = NULL）                    / *  遇 NULL 结束输入* /
printf（" % s \ n",str）;
}
```

如运行此程序，从键盘输入 Tsinghua，则在屏幕上会输出 Tsinghua 的字符串。

2．字符串输出函数 puts（）

（1）格式

int puts（char * s）；

（2）功能

将 s 所指向的字符串输出到标准输出文件中，并将末尾字符' \ 0 '变换为＜CR＞输出。

（3）参数

s 指向要输出的字符串。

（4）返回值

正常返回值是 0，错误返回值为 EOF。

【例 2-7】将输入的字符串原样输出。

```
# include <stdio.h >
main（）
{
char * string;
puts（"test string I/O"）;
while（gets（string） ! = NULL）
puts（string）;
}
```

2.7.3 格式化输入/输出函数

C 语言的输入/输出是由系统提供的库函数完成，C 语言的标准函数库提供了所需的输入/输出函数。

1．格式化输出函数

（1）格式

printf（“ 输出格式描述串 ”，输出项表列）

（2）功能

该函数按照指定的格式，将输出项表列中的诸项输出到标准输出文件中。

（3）参数说明

输出格式描述串是由一系列格式转换说明符组成，格式转换说明符的描述形式如下：

% [+] [-] 0 m [.n] l ＜形式字母＞

■ 形式字母。形式字母指定输出格式，形式字母的种类如表 2-8 所示。

表 2-8 形式字母表

形式字母	输出格式	使用举例	输出结果
d	十进制整型数	int y=25;print("%d",y）;	25
x	十六进制整型数	int y=35;printf("%x",y）;	23
o	八进制整型数	int y=35;printf("%o",y）;	43
u	无符号十进制整型数	int y=35;printf("%u",y）;	35
c	单个字符	char c='A"printf("%c",c）;	A
s	字符串	shatic char s[]="This";printf("%s",s）;	This
e	指数形式的浮点数	float y =475.3751;printf("%e",y）;	4.753751e+002

续 表

形式字母	输出格式	使用举例	输出结果
f	小数形式的浮点数	float y =475.3751;printf("%f",y）;	475.375100
g	e 和 f 中较短的一种	float y =475.3751;printf("%g",y）;	475.3751
%	符号“ % ”本身	printf("%%"）;	%

■ l。l 指定输出精度。如果形式字母是 d、x、o 和 u，则可指定如下几类精度：

l：long 型输出精度

h：short 精度

缺省时为 int 型精度

例如：long x =12345678；printf（“ %ld ”，x）；

如果形式字母是 e、f 和 g 时，则指定 1 时为 double 精度，不指定时为 float 精度。

■ m［.n］。指定输出长度。如果输出的是实数，则 m 表示该项输出占用字符位置的总长度，n 表示小数部分的字符总长度。例如：

```
float x =3.1415
printf（" % 7.4f "，x）；
```

% 7.4 指出输出 x 时，总共占用 7 个字符位置，其中小数部分占 4 位，输出结果是：

```
b3.1415
```

其中 b 表示下一个空白。

如果输出的是字符串，则 m 表示输出字符占用的总字符长度，而 n 表示实际输出的字符数。例如：

```
static char c［］＝"ABCDE"；
printf（" %5.3"，c）；
```

则输出结果是：

```
bbABC
```

其中 bb 表示两个空白（下同），表示该输出共占 5 个字符位置，实际输出 3 个字符。

■ 0。指定不使用的空位置填 0（数字零）。如果不指定 0 时，则不使用的位置为空白。

该项仅对数值输出时才可指定，对字符串输出不用指定。例如：

```
int x=456；
printf（" %05d"，x）；
```

二者的输出形式分别为：

```
00456
bb456
```

■ ［+］［-］。指定输出位置。如果指定“ + ”缺省时，则输出字符靠右端。如果指定“ - ”时，则输出字符靠左端。例如：

```
static char str［］＝"ABC"；
printf（"% -5s"，str）；
printf（"% +5s"，str）；
```

二者的输出分别为：

```
ABCbb
bbABC
```

■ %。是一个格式转换说明符的开始标识记号。

程序员要想按照要求的格式输出指定的项，就必须弄清楚各种各样转换说明符。

4. 输出项表

输出项表指定要输出的数据，可以指定多个输出数据，数据之间用逗号（,）分隔，要输出的数据可以是变量，也可以是表达式。输出数据的个数要与格式转换说明符的个数相一致。例如：

```
printf（"x=% f n=% d s=% s \ n"，x，n，s）；
```

其中：输出项 x 与格式转换符% f 相对应，以实型格式输出；输出项 n 与格式转换符%d 相对应，以整型格式输出；输出项 s 与格式转换符%s 相对应，以字符串格式输出；

【例 2-8】请指出下面程序的输出结果。

```
main（）
{
int x,y,z;                          / *  定义整型变更名  */
x = 10;                             / *  变量初始化  * /
y = 15;                             / *  变量初始化  * /
```

```
z = 25;                                    / * 变量初始化 * /
printf（"output-result:\ n"）;              / *屏幕输出打印提示 * /
printf（"x = % d y = % d z =% d \ n",x,y,z;  / * ，输出结果 * /
prntf（"x + y = % d \ n x + y + z = % d \ n",x + y + z）;     / * 输出计算结果 * /
}
```

其输出结果为：

```
output - result：
x = 10 y = 15 z = 25
x + y = 25
x + y + z = 50
```

此例可以看出，输出格式转换说明符不仅规定了输出项的输出格式，也规定了其输出位置，例如 y **的值输出在“** y = **”之后。另外还看出输出项可以是表达式。**

2．格式化输入函数

（1）格式

scanf（**“格式描述串”，输入项表列）；**

（2）功能

该函数从标准输入设备（通常是键盘）按照指定的格式为指定的输入项输入数据。如果有多个输入项时，从键盘上输入的各个数据之间可以用空格、TAB **键或回车作为分隔符。**

（3）参数说明

与 printf（）**类似，格式描述串也是由一系列“格式转换说明”所组成。每个格式转换说明符的完整描述如下：**

```
% m［.n］1＜形式字母＞
```

- **形式字母，有** d、o、x、c、e **和** f，**含义同** printf（）**中的描述。**
- l。**指定输入数据的精度。在形式字母** d、o、x、c **和** f **前可指定** l，**则表示输入** long **型或** double **型精度。如果指定** h，**则表示精度为** short。
- m［.n］。**表示输入数据的长度，含义类似** printf（）**中所述。对于字符串输入可限制输入串的长度。如** scanf（**“％** 20s **”，**str）；**表示输入串不能超过** 20 **个字符。**
- **输入分隔符的指定。输入分隔符可用以下两种方法指定：**
- **在格式转换说明符之间用空格字符（空格、**TAB **和＜**CR**＞作分隔符。**
- **在格式转换说明符之间用字符作分隔符。**

例如：

```
scanf（“ %d：%d：%f"，& x，&y，&z）；
```

从键盘输入以下内容：

```
35：45：4.45＜CR＞
```

其中 35 **赋给** x，45 **赋给** y，4.45 **赋给** z。**又如：**

```
scanf（" %4d %2d % d",&x，&y，&z）；
```

键盘输入以下内容时：

```
20531754
```

其中 2053 **赋给** x，01 **赋给** y，754 **赋给** z。

（4）输入项表

输入项表指定输入项。如果多个输入项，则输入项之间用逗号隔开。输入项必须以形式给出。例如：

```
float x；
char string［20］；
scanf（" % f % s"，x，string）；
```

此例是错误的。因为，x **不是以地址形式给出。下例说明** scanf（）**函数的功能。**

【例 2-9】请给出下面程序的输入并指出其输出结果。

```
main（）
{
int a,b,c;                                  / * 定义整型变量名 * /
float average;                              / * 定义实型变量名 * /
printf（" \ n please input a,b,c:"）；          / * 输入提示 * /
scanf（" % d % d % d,&a,&b,&c,）；           / * 读入数据 * /
printf（" \ n a = % d b= % d c= % d",a,b,c）；  / * 输出 a、b 和 c 的值 * /
average=（a+b+c）/3;                        / * 计算平均值 * /
printf（" \ n average = % f ",average）；      / * 输出平均值 * /
}
```

该程序以人机对话方式为变量 a、b 和 c 输入数据，并显示该输入数据和计算的平均值。它的某次执行情况如下：

```
please input a，b，c：75 85 95＜CR＞
a =75b=85c=95
average = 85.0
```

【例 2-10】本例要求掌握不同输入格式下数据的输入方式。

```
# include < stdio.h >
main（）
{
char a,b,c,d,e                              / * 定义字符型变量 * /
int m,n,u,v;                                / * 定义整型变量 * /
a=' N ';b=' a '; c= 'm ';d='e ';            / * 初始化字符变量 * /
putchar（a）；
putchar（b）；
putchar（c）；
putchar（d）；
putchar（' \ n '）；
putchar（a）;putchar（' \ n '）；
putchar（b）; putchar（' \ n '）；
putchar（c）; putchar（' \ n '）；
putchar（d）; putchar（' \ n '）；
printf（"Please input a character ,and then press ENTER key ! \ n"）；
e=getchar（）
printf（"The character inputed is "）；
putchar（e）；
putchar（ ' \ n '）；
printf（"请输入两个整型数，然后按回车键！\ n"）；
printf（"请注意，两个整型数之间用＜空格＞隔开 \ n"）；
scanf（" % d % d,& m,& n）；
printf（"The two datas inputed are"）；
printf（" % d and % d \ n",m,n）；
printf（"---------- \ n"）；
printf（"请输入两个整型数，然后按回车键！\ n"）；
printf（请注意，两个整型数之间用＜逗号＞隔开 \ n"）；
scanf（" % d % d ,& u, & v）；
```

```
printf（"The two datas inputed are"）；
printf（" % d and % d \ n",u,v）；
printf（"------------------------------------------------------------ \ n"）；
printf（"请输入两个整型数，然后按回车键！\ n"）；
printf（"请注意，两个整型数之间用＜：＞隔开 \ n"）；
scanf（" % d: % d,& u, & v）；
printf（"The two datas inputed are "）；
printf（" %d and %d \ n",u,v）；
}
```

从以上的程序可以知道，用 scanf（）函数进行输入时，输入格式要严格根据输入格式中定义的方式进行，尤其是当连续输入若干个整型数（其他类型的数据也一样）时，如果控制格式符没有任何其他符号时，一定要用一个“空格”分隔开，否则系统将无法确认输入的数据，例如，要连续输入 123 和 456 两个数，如果不用空格分隔开这两个数，那就变成 123456，系统无法区分输入的是 12 和 3456，还是 1 和 23456，更不能理解用户的意图——输入 123 和 456 两个数。输入格式中有“空格”时，那就必须输入两个空格来分隔两个数，其中下一个空格是用来与格式控制符中的空格匹配，另一个空格用来分隔两个数。此外，当在格式控制符中用“：”来分隔两个数时，用户在输入数据时，就必须在两个数据之间用“：”来分隔这两个数，以实现输入的数据格式与格式控制符匹配。用户可以把【例 2-10】运行一遍，以更好地了解和体会格式控制符在输入和输出过程中的作用。

【例 2-11】求解 $ax^2 + bx + c = 0$ 的根，假设 $b^2 - 4ac >= 0$。

```
# include "math.h"        / * 由于程序中用到 sqrt（）这个开平方函数，必须有此语句* /
# include "stdio.h"
main（）
{
float a,b,c,disc,x1,x2,p,q;     / * 定义数据类型，a，b 和 c 是言和的系数，x1、x2 为
                                方程的根 * /
scanf（" % f % f % f",& a,& b,& c）；        / * 输入方程的三个系数 * /
printf（"a= % f \ nb=% f\ nc=% f\ n",a,b,c）；   / * 输出已输入的系数，确认输入是否正
                                                确 * /
disc = b * b - 4 * a * c;
p = -b/（2 * a）；
q= sqrt（disc）/（2 * a）；
x1=p + q;                                  / * 第一个根 */
x2=p - q;                                  / * 第二个根 * /
printf（"\ n\ nx1= %5.2f\ nx2= % 5.2f \ n",x1,x2）；   / * 输出计算结果 * /
}
```

2.8 小 结

（1）数据类型

C 语言提供了丰富的数据类型，不同的数据类型决定了数据所准许的操作以及不同的存储形式和表示形式。任何数据（不论常量和变量）都属于某种数据类型。

（2）数据的存储形式

不同类型的数据其存储形式也不相同。为了理解不同类型数据之间的转换，弄清其存储形式。

- 字符型数据。下一个字符占用下一个字节，其值就是其所在字符集中的编码，对 ASCII 字符集，就是其 ASCII 编码。例如'a'的编码是 97，其内存表示形式为 01100001。
- 整型数。整型数分为短整型数、整型数和长整型数，一般而言短整型数和整型数都占两个字节，长整型数占 4 个字节。最高二进制位是符号位，正数用原码形式表示，负数以补码形式表示。

于是，125 的 short、int 和 long 的表示形式分别为：

short，int　　0 000 000 001 111 101

long　　00 000 000 000 000 000 000 000 001 111 101

-125 的存储形式为：

short，int　　1 111 111 110 000 011

long　　11 111 111 111 111 111 111 111 110 000 011

整型数又分为符号型和无符号型，有符号型的存储形式同 short、int 和 long。

无符号型整型数则把最高二进制位视为有效数据，而不是符号位，于是它表示的范围要比有符号型大一倍。

■ 实数。实数分 float 和 double 两种类型，它们分别占用 4 个字节和 8 字节。

（3）数据的表示形式

数据的表示形式就是在程序中的书写形式。不同类型的数据有不同的表示形式。字符常量是以单引号括住的单个字符，例如'a'、'A'和'3'等。字符串常量是用双引号括起来的一串字符，例如：“ This is a string ”。

十进制整型数的表示形式是：[－] nn…n，其中 n 是 0～9 之间的数字。

八进制整型数的表示形式是：0nn…n，其中 n 的取值域是 0、1…7，开头的 0 是数字 0。

十六进制数的表示形式是：0xnn…n，其中 n 的取值域是 0、1…、A～F（或 a～f），开头 0x 是数字 0 和字母 x（或大写 X）。实数的表示形式有两种：小数形式和指数形式。另外，对整型数来说，如果在常量的后边加上一个 l，则认为是 long 型整型数。读者要弄清数的表示形式，才能写出正确的程序。

（4）变量

数据有常量和变量，变量是其值在程序执行过程中不断发生变化的量，每个变量都有一个名字，名字的命名规则同用户定义标识符。变量是和内存单元相关联的，其对应的内存是在编译时（对全局变量和静态变量）或程序执行时（对于局部变量）分配的。为了分配存储，所有的变量必须先定义或说明，然后才能引用。对全局变量而言，同一个变量在整个程序中只能定义一次，但是可以说明多次；定义是要分配内存的，而说明并不分配内存；如果一个变量是在 x 源文件的开头定义，则它在 y 源文件中只需说明就可引用，而在 x 源文件中不需再说明就可使用。

（5）运算符和表达式

C 语言的特点之一是有丰富而书写灵活的运算符，这些运算符都有结合性和优先级。表达式的处理规则就是按照运算符的优先规则和结合性来决定的。所以读者必须理解其优先级和结合性才能写出正确的表达式。

（6）数据类型转换

在一个表达式中，参与操作的各操作对象的数据类型可能不一致，这就需进行数据类型转换。

转换有显示转换和隐式转换，显示转换也叫强制类型转换；隐式转换是按照规定的转换规则自动进行的。

（7）C 语言的输入/输出由专门的函数来完成。

习　题

1．C 语言的数据类型有哪些？

2．什么是常量？在 C 语言中，常量有哪几三种？

3．字符常量与字符串常量有什么区别？

4．写出以下程序的运算结果。

```
main（）
{char c1 = 'a'; c2 = 'b'; c3 = 'c'; c4 ='101';c5 = '\116';
printf ("a%cb%c\tc%c\tabc\n",c1,c2,c3 );
printf ("\t\b%c    %c", c4, c5);
}
```

5．求下面算术表达式的值。

（1）x+a%3*(int)(x+y)%2/4

设 x=2.5, a=7, y =4.7

（2）(float)(a+b)/2 + (int)x % (int)y

a=2,b=3,x=3.5,y=2.5

6.**写出程序运算结果。**

```
main()
{int i,j,m.n;
i= 8; j=10;
m= + + i;n = j + +;
printf("%d,%d,%d,%d",i,j,m,n);
}
```

第三章　C 语言程序的控制结构

C 语言的最大特点之一是结构化，结构化使程序结构清晰、易读性强，可以提高程序设计的质量和效率。结构化程序由若干基本结构组成，每一个基本结构可以包含一个或若干个语句。在 C 语言中，有三种基本结构：顺序结构、选择结构、循环结构，本章将分别讲述这三种基本结构。

3.1　算法及结构化程序设计

3.1.1　算法及其特征

当今计算机的迅猛发展已使软件开发步入工程化阶段。一个大型软件的开发一般需要经历“规划”、“要求分析”、“设计”、“编码”、“测试”和“运行维护”等几个阶段。它就象是一个大型的工程。其中，正确的“需求分析”对程序设计是至关重要的，而“编码”就是对需求的具体实现。在“编码”过程中，算法是非常重要的，良好的算法，往往会得到事半功倍的效果，算法思想决定了程序的质量和性能。

而目前许多软件的开发不仅需要处理大量的关系复杂的数据，这些数据通常都具有一定的结构性。因此，软件设计的实质就是设计合适的数据结构和基于这个数据结构的算法。于是著名的计算机专家 N.沃恩教授提出了一个著名的公式：

程序＝数据结构＋算法

算法是程序设计中一个非常重要的概念，所谓算法就是处理问题的方法和步骤。更确切地说，算法是为解决特定的问题而要一步一步执行的有穷操作的描述。

一个完整的算法应具有如下特征：

■　有穷性

任何一个算法都必须能在执行有限步之后结束。例如：

N！=1*2*3*…*（N-1）*N

其中 N 是下一个特定数，例如 50，则这个描述可称之为一个算法。又如：

sum=1+2+3+…+N+…

上式就不能称之为算法，因为该描述在执行有限步后仍不能结束，它只能称之为一个计算方法。

■　确定性

算法的每一步执行，其顺序和内容都必须有确切的规定，不能有二义性。

■　可执行性

可执行性是指算法的所有操作都是能做到的，即称之为可操作性。

■　0 个或多个输入；1 个或多个输出

算法既然是为解决特定问题而设计的，那么它至少包含一个输出步骤，来告知处理结果。无任何输出信息的算法是无意义的。

为了更深入地理解算法，先看几个有关算法的实例。

【例 3-1】解一元二次方程的算法。

设有一元二次方程 a*x*x+b*x+c=0，其中 a、b 和 c 是不等于 0 的实数。解此方程可按如下的方法和步骤：

（1）首先计算 p=b*b-4*a*b*c。

（2）判断 p 是否＞＝0 若结果为“真”则执行步骤（3），若结果为“假”，则执行步骤（4）。

（3）计算二实根

x1=（-b+sqrt（p））/（2*a）

x2=（-b-sqrt（p））/（2*a）

结束计算。

（4）计算复根

x1=-b/（2*a）+isqrt（-p）/（2*a）

x2=-b（2*a）+ isqrt（-p）/（2*a）

结束计算。

以上四步就是解一元二次方程的一个算法。其中 sqrt（）**是求平方根函数，其程序如下：**

```
# include < stdio.h >
main（）
{float a,b,c,x1,x2,p,q,q1,q2;
scanf（" % f% f% f",a,b,c）;
p=b * b - 4 * a *c;
if（p > =0）
{
x1=（-b +sprt（p））/（2 * a）;                    / *求实根 * /
x2-（-b -sprt（p））/（2 * a）;
printf（" x1= % f \ n",x1）;
printf（" x2=% f \ n",x2）;
}
else
{
q1= -b/（2 * a）;                              / * 求虚根 * /
q2=sqrt（-p）/（2 * a）;
printf（" x1= % f + i% f \ n",q1,a2）;
printf（" x2= % f -I% f \ n",q1,a2）;
}
}
```

3.1.2 算法和类型与结构

计算机可处理的算法，一般归纳为“数值算法”和“非数值算法”两类。“数值算法”常用于科学计算，而“非数值算法”则广泛用于各类数据的数据处理，它常常要涉及大量数据和复杂的数据结构。不管是何种算法，都是由“结构”和“原操作”所构成。最基本的结构有顺序、分支和循环三种，原操作包括输入、输出、表达式求值、变量赋值、比较两个变量等等。下面分别描述三种基本结构。

1．顺序结构

顺序结构是由一组顺序执行的程序块所组成的，每一个程序块可以是一个非转移语句、分支语句、循环语句或这些语句的排列、嵌套，它是任何一个算法都离不开的一个基本主体结构，它是由若干顺序执行的处理块所组成。例如有三个处理块所构成的顺序结构如图 3-1 **所示。**

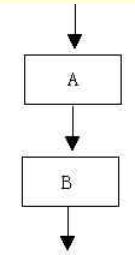


图 3-1 **顺序结构示意图**

2．分支结构

分支结构是根据分支条件的取值来决定程序执行的走向，分支结构的示意图如图 3-2 所示：

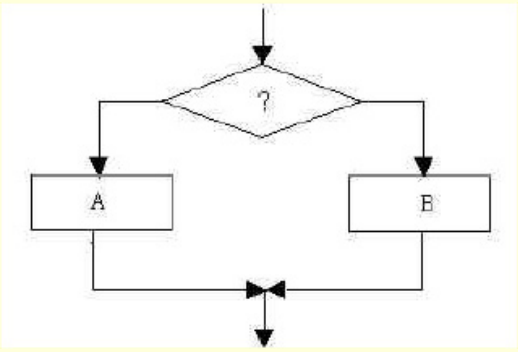
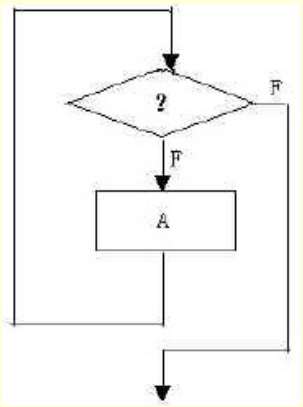


图 3-2　分支结构示意图

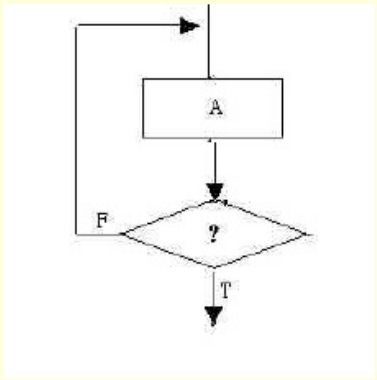
分支结构也是一种基本和常用的一种结构，它向我们提供了根据条件取值来选择不同处理块的方法。

3．循环结构

循环结构是一种对某一处理块反复执行指定次数的结构，图 3-3 是循环结构的示意图，循环结构又分为“ 当循环 ”和“ 直到型循环 ”，“ 当循环 ”的特点是先判断，后执行循环体；“ 直到型循环 ”是先执行，后判断。图 3-3a 和图 3-3b 分别是 “ 当循环 ” 和 “ 直到型循环 ” 的示意图。



a）



b）

图 3-3　循环结构示意图

循环中反复处理部分叫循环体，循环次数由条件取值决定。

从软件工程的观点来看，一个结构化的程序远比非结构化的程序质量高。由于结构化的程序具有结构清晰、可读性强、易维护的特点，而非结构化程序是程序设计者随意施展自己技能所构成的程序，所以这样的程序可读性相对来说就差一些，而且不易维护。因此，当前广泛采用结构化程序设计方法。结构化程序设计的实质是将问题按 “ 自顶向下，逐步求精 ” 的原则进行分解，使其分解成若干较小的问题，然后再用结构化编码技术编制出各个较小问题的程序块，进而构造出整个问题的求解程序。

结构化程序设计是以有条理的方式构造程序的一种技术。一个结构化程序的构造应符合以下几条规则：

（1）任何一个程序单位（例如一个函数）都只能由顺序、分支和循环三种基本结构所组成。

（2）任何一个程序单位可视为若干程序块排列而成。

（3）每个程序块只能有一个入口和一个出口。

（4）下列任何一个程序结构都可视为一个程序块。

- 一个或若干个不进行转移的执行语句（视为顺序结构）是一个程序块。
- 一个如下形式的分支或 switch 结构（视为分支结构）是一个程序块。

if…else…

switch…case…

- **一个如下形式的循环结构，也视为一个程序块。**

  while…

  do…while

  for（e1；e2；e3）

- **由若干程序块按顺序排列所构成的结构也视为一个程序块。所以，一个结构化程序实质上是由有限个顺序、分支和循环三种基本结构经排列、嵌套而成。**

编写结构化程序应注意如下两点：

（1）要保证结构的完整性

不允许结构层次之间的交叉，例如图 3-4 所示的结构是符合完整性要求的，而图 3-5 则不符合。因此，循环结构和循环结构、分支结构与分支结构、循环与分支结构的交叉都是不准许的。

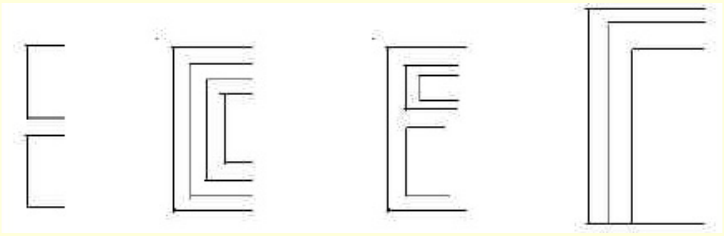


图 3-4　符合完整性要求的结构

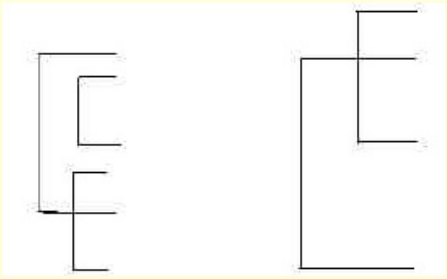


图 3-5　不符合完整性要求的结构

（2）操作上的完整性

对于结构化的程序设计，一个基本结构就是下一个完整的操作单元，程序只能从入口进入，出口退出，不能从其外部进入内部，也就是说，一个程序结构不能有多个入口和多个出口。但是程序设计语言往往并不限定操作上的完整性，例如 C 语言，在循环内部可用 goto 语句退出循环（或 exit（）结束等），但这是不符合结构化程序设计的完整性要求的。

【例 3-2】语法上是合法的，但不符合结构化程序设计的完整性要求。

```
float a;
…
scanf（" % f,& a）;
sum = 0;
while（a ! = 0.0）
{
sum + =a;
if（sum > =100）
goto endloop;          / * 从内部退出循环，不符合结构化程序设计的完整性要求 * /
scanf（"% f",&a）;
}
……
endloop:
```

```
printf（"sum= % d",sum）;
…
```

因此，图 3-6 是符合操作完整性的，而图 3-7 是不符合的。

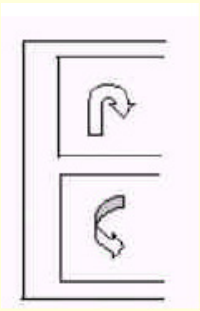


图 3-6　符合完整性要求的结构

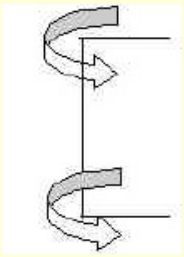


图 3-7　不符合完整性要求的结构

3.2　顺序结构程序设计

顺序结构的程序设计是最简单的程序设计。顺序结构的程序是由一组顺序执行的程序块所组成。最简单的程序块是由若干个顺序执行的语句所构成。这些语句可以是赋值语句，输入/输出语句等。为此，我们首先介绍赋值语句和简单的输入/输出语句，然后说明顺序程序设计的方法。

3.2.1　赋值语句

以分号结尾的赋值表达式叫表达式语句，也叫赋值语句。例如：

```
a=b+c-d*e；
```

在赋值语句中，首先计算等号左边的表达式的值，然后将其值赋给等号左边的变量。如果等号右边的表达式的类型与左边变量的类型不一致，系统将自动把等号右边的表达式的值转换为与左边变量相同的类型，然后再赋值。

3.2.2　顺序程序设计及举例

计算机处理的问题难易程序差别很大。简单问题的程序，其结构可能完全是顺序的，即在执行时，是顺序逐句执行。而复杂问题的程序则不仅包含顺序结构，还可能包含分支结构和循环结构。本节主要介绍顺序结构的程序设计方法。下面通过一些简单的顺序结构程序设计的例子，使读者了解并总结顺序结构程序设计的方法与特点。

【例 3-3】从键盘上输入一个小写字母，要求用其对应的大写字母输出。

算法设计：解决问题分为三步：输入一个小写字母，将小写字母转换为对应的大写字母，输出转换后的大写字母。其程序如下：

```
# include < stdio.h >                          / * 嵌入头文件 * /
main（）
{
char c1，c2                                    / * 定义字符型变量 c1 和 c2 * /
printf（"please input a Lower-caae:"）;          / * 输出提示语句 * /
c1 = getchar（）;                               / * 从键盘接收一个字符 */
printf（" \ n Lower-case is % c",c1）;
c2 = c1 - 32;                          / * 将小写字母通过 ASCII 值的计算转换成大写 */
printf（ " \ n upper - case is % c",c2）;          / * 输出转换后的结果 * /
}
```

运行时的输出情况如下：

```
please input a Lower-case：'d'＜CR＞
Lower-case is d
upper-case is D
```

【例 3-4】从键盘分别输入两个复数的实部和虚部，求它们的和、差、积、商并分别在屏幕上输出。

算法设计：每个复数均有实部和虚部，所以输入两个复数实际上是输入四个实数，然后按数学公式顺序地求出它们的和、差、积、商（每个结果都是两个实数）并使用 printf 函数按复数的格式打印出来。应该注意的是，既可以定义若干个变量来存放计算的结果，也可以将计算的表达式放在 printf 函数的变量列表中直接打印出来。其程序如下：

```
# include <stdio.h >
main（）
{float a,b,c,d,resultR,resultI;
printf（"please input the first complex number: \ n"）;
scanf（" % f % f,&a,&b）;                                    / *  输入第一个复数的实部和虚部  */
printf（"please input the second complex number: \ n"）;
scanf（" % f %    f",&c,&d）;                                / *  输入第二年复数的实部虚部  * /
printf（"the sum of them is %    f + % f i\ n",a + c,b+d）;    / *求两实数的和并打印  * /
printf（"the difference between them is % f + % f i \ n",a-c,b-d）;
resultR=a * c-b * d;
resultI=a * d + b * c;                                       / *  求两实数乘积的实部和虚部* /
printf（"the mass of them is % f+ % f    i \ n",resultR,resultI）
resultR=（a * c + b *d）/（c * c + d * d）;
resultR=（b * c - a *d）/（c * c + d * d）;
printf（  "the quotient of them is % f+ % f    i \ n",resultR,resultI）;
}
```

执行情况如下：

```
please input the first complex number：
3.4       5.6＜CR＞
please input the second complex number：
7.8 -99.1＜CR＞
the sum of them is 11.200000+-93.499999；
the difference of them is -4.400000+104.699998；
the mass of them is 581.479980+-293.260010；
the quotient of ehem is -0.053477+0.038518；
```

【例 3-5】从键盘输入一个字符，求出其前后相邻的两个字符，然后按由大到小的顺序输出这三个字符及对应的 ASCII 码。

算法设计：输入字符的前面下一个字符，其 ASCII 码比此字符小 1，同样，后一个字符的 ASCII 码比此字符大 1，对字符型变量进行算术运算时，使用的正是它们的 ASCII 码，所以直接将输入的字符加 1 或减 1，就可以得到它前后的相邻字符。输出时，使用格式控制符%c 可输出字符本身，而使用%d 则可输出字符对应的 ASCII 码。其程序如下：

```
# include < stdio.h >
main（）
{
char c,cf,cb;                                  / *  定义字符型变量* /
printf（" \ n please input a character:"）;
c =getchar（）;                                  / *从键盘读入字符* /
```

```
cf =c - 1;                          / *求前导字符 * /
cb =c + 1;                          / * 求后续字符 * /
printf（" % c % c % c\ n",cf,c,cb）;      / * 输出字符 * /
printf（" % d % d % d\ n",cf,c,cb）;      输出字符的 ASCII 值 * /
}
```

执行情况如下：

```
please input a character：m＜CR＞
1       m          n
108     109        110
```

从上述三个例子中，我们可以看到编程的关键是设计算法。一个简单的问题，其处理算法比较简单，我们一下子就能把它的算法设计出来。但对处理较复杂问题的算法，就很难一下子分析得十分透彻，而需要逐步分析清楚。即先从总体上分析其粗略的算法框架，再对粗略算法的每一步进一步细化，如此下去，直至细化到每一步都足够简单，能用一个或几个 C 的语句来实现。于是就出现算法的顶层设计，第二层设计…。这是十分有效而且常用的算法设计方法，即“ 自顶向下，逐步求精 ”的设计方法。

【例 3-6】从键盘输入一个正三棱锥的边长、分别计算其表面积和体积，并打印输出，输入输出时都要有提示信息，输出结果只保留 3 位小数。其程序如下：

```
# include < stdio.h >
# include < math.h >
main（）
{float x;
printf（"please input thegth of the edge of the pyramid:\ n"）;
scanf（" % f",& x）;
printf（"the total area of the sruface is % -8.3f\ n",pow（3,0,5）*x * x）;
/ * 求出并打印表面积。* /
printf（"the volume of the pyramid is % -8.3f\ n",pow（2,0,5）*pow（x,3）/12）;
/ * 求出并打印体积。* /
}
```

由于使用了标准数学函数，所以应在程序开头添加包含命令＃include <math.h>**。**

3.3 分支结构程序设计

与顺序结构一样，分支结构也是程序的基本结构之一，也是常用的一种结构。所谓分支结构，就是根据不同的条件，选择不同的处理块（或程序块，或分程序）。例如，对于形如 aX2-bX+c **的方程，应根据系数** a **是否等于零分别当作一次方程和二次方程求解。因此，分支结构又叫条件分支结构。**

在 C **语言中，条件分支结构可通过** if **语句和** switch **语句实现。**if **语句有** if**、**if-else **和** if-else if **三种形式。**

3.3.1 If-else 分支

If-else **分支是标准形式的条件分支，其执行流程如图** 3-8 **所示。**

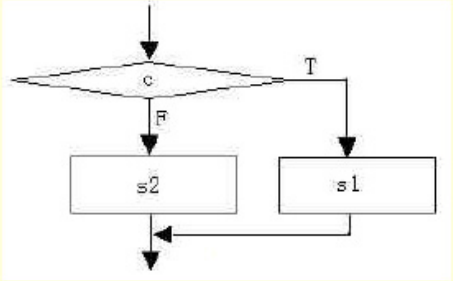


图 3-8 if-else **条件分支**

其语句形式为：

```
if（条件表达式 c）
 {
程序段 sl；
 {
else
 {
程序段 s2；
}
```

其中，程序段 sl，s2 **可以是语句或者复合语句。**

if-else **语句的处理过程是，先计算** if **后面的条件表达式** c **；若结果为非** 0 **值（即逻辑上为真），则执行程序段** sl，**否则执行程序段** s2。**可见，**if-else **是二选一的分支结构。**

【例 3-7】**输入一个字符，判断它是否是数字。**

```
main（）
{
char c;
printf（"\ nplease input a character:"）;
c =getchar（）                    / * 从键盘输入一个字符 * /
if（c > =48 & & c < =57）         / * 根据输入字符的 ASCII 码判断其是否是一个数字* /
printf（"It is a number. \ n"）;  / * 若是数字，输出相应的提示 * /
else
printf（"It is not a number.\ n"）; / * 若不是数字，也输出相应的提示 * /
}
```

该程序执行时，当输入数字时，则显示“ It is a number. **”，否则显示“** It is not a number. **”。**if **后面用圆括号“（”和“）”括住的表达式经常是条件表达式或逻辑表达式（但可以是任何表达式），表达式必须用圆括号括住。**

3.3.2 if **分支**

if **分支是** if-else **分支的特殊情况，其执行流程如图** 3-9 **所示。**

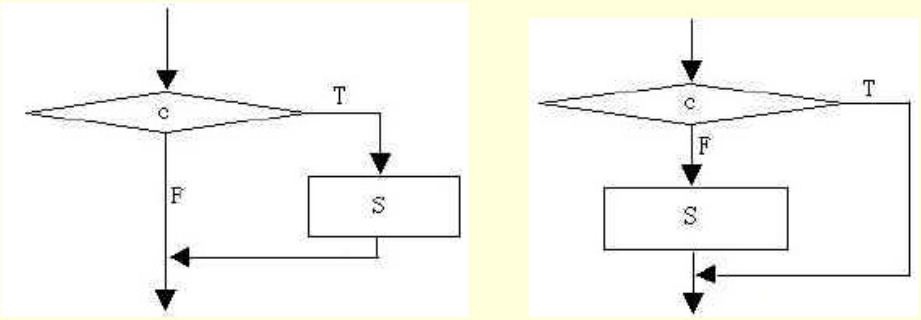


图 3-9 if **分支的流程图**

其程序形式如下：

```
if（c）          或       if（! c）
程序段 s；                程序段 s；
```

其中 c **是条件表达式。**

if **分支实际是** if-else **分支形式的简化，若** if-else **的两个分支中的一个为空，则可以写成** if **分支的形式：**

```
if（c）          或       if（c）
程序段 s；                ；
```

```
else                    else
；                      程序段 s；
```

【例 3-8】判断输入的字符是否是小写字母。

```
main（）
{
char c;
printf（" \ n Input a character:"）；
c =getchar（）；
if（c > =97 & &c < =122）                    / * 判断所输入的字符是否是小写字母 * /
printf（"Yes,it a small letter ! \ n"）；
}
```

3.3.3 条件分支的嵌套

进行程序设计时，经常要用到条件分支嵌套。所谓条件分支嵌套就是在一个分支中可以嵌套另一个分支。例如，在下面的条件分支中：

```
if（条件表达式 c）
程序段 s1；
else
程序段 s2；
```

程序段 s1 和 s2 中又可包含条件分支。利用条件分支嵌套可以实现多分支控制。下面是一个条件分支嵌套的例子。

【例 3-9】从键盘输入一串字符，将其中的大写字母转换成小写字母，小写字母保留不变，其他字符忽略后输出。

```
# include < stdio.h >
main（）
{char c;
printf（"input a $: \ n"）；
while（（c =getchar（））  ! _' \ n"）
{
if（c > =97 & &c < =122）
putchar（c）；
else
{
if（c > =65 & &c < =90）
putchar（c+32）；
else;
   }
}
}
```

在此例中，内嵌的花括号可以省略。因为它仅括住了一个语句，即 if-else 语句。如果被括住的是多个语句，则花括号不能省略。在多个分支嵌套中，如果缺省花括号时，要特别注意 if 和 else 的配对关系，一个 else 总是与其上面距它最近的，并且没有其他 else 与其配对的 if 相配对，例如，有如下的嵌套形式：

```
if（条件表达式 c1）              （1）
  if（条件表达式 c2）            （2）
    程序段 s1；                  （3）
else
```

```
if（条件表达式 c3）          （4）
    程序段 s2；              （5）
else                         （6）
    程序段 s3；
```

其 if 和 else 的配对关系应当如图 3-10 所示。

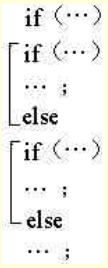


图 3-10　if 与 else 配对关系

由图 3-10 可以看出，整个嵌套形式相当于：

```
if（条件表达式 c1）
{if（条件表达式 c2）
    程序段 s1；
else
{ if（条件表达式 c3）
    程序段 s2；
 else
    程序段 s3；
 }
}
```

使用括号可以改变 if 和 else 的配对关系，比如，想让 if（1）和 else（3）配对，可采用如下方法：

```
if（条件表达式 c1）          （1）
{if（条件表达式 c2）         （2）
    程序段 s1；
else
{ if（条件表达式 c3）        （3）
    程序段 s2；              （4）
 else（5）
程序段 s3；
```

其实，为了使程序清晰易读并减少出错几率，建议；对所有的嵌套 if…else 语句组，都使用括号来明确其配对关系。

3.3.4　if-else if 结构

if-else if 结构是条件分支嵌套常用的一种形式，其一般形式如下：

```
if（条件表达式 c1）
    程序段 s1；
else if（条件表达式 c2）
    程序段 s2；
else if（条件表达式 c3）
    程序段 s3；
……
```

else if（条件表达式 cn）

程序段 sn+1；

该结构实质上是 if-else 分支的多层嵌套。因为如果嵌套的层数过多，会使程序写的很靠右，因此，把它简化为 if-else 结构。其流程是从条件表达式 c1 开始向下逐一判断，一旦满足（即条件 ci 的逻辑值为 1），则执行对应的程序段 si，若所有的条件表达式的值都为零，就执行 sn+1 语句。图 3-11 是 if-else if 结构的示意图。

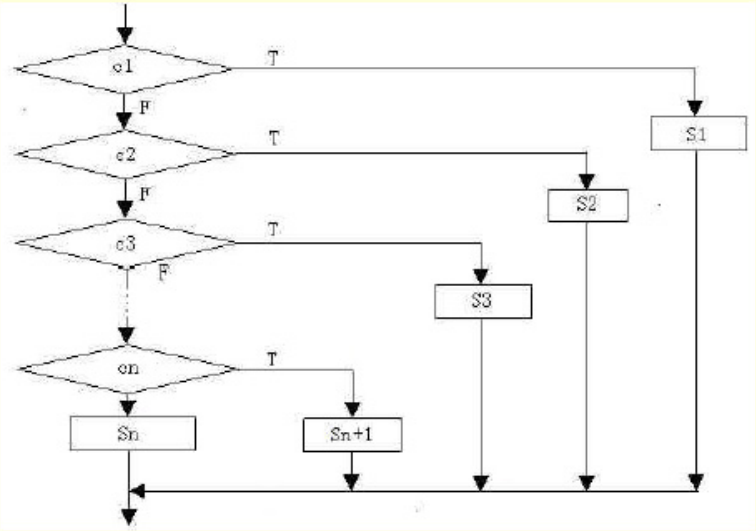


图 3-11 if-else if 结构的流程图

【例 3-10】有一个函数，定义如下：

$$y=f(x)=\begin{cases}0 & (x<0)\\ x & (0<=x<=10)\\ x^2+1 & (x>10)\end{cases}$$

请使用 if…else if…else…语句编写程序，根据用户输入的自变量 x 的值，计算函数值。

程序如下：

```
# include < stdio.h >
main（）
{float x;
printf（" input the value of x:"）；
scanf（" % f",&x）；
if （x <0.0）
   printf（"y =0 \ n"）；
else if（x > =0.0& &x < =10.0）
   printf（"y =% .2f \ n",x）；
else
   printf（"y = % .2f \ n",x * x * + 1）；
}
```

应该注意的是，一般情况下，嵌套 if…else 语句 if…else…if…else…语句和下面将要讲到的 switch 语句之间是可以相互转换的。

3.3.5 开关（switch）分支结构

开关分支是分支结构的另一种形式，执行时它根据条件的取值为选择程序中的一个分支。

其程序形式如下：

```
switch（表达式 c）
 {
  case  常量表达式 c1：
        程序段 s1；
        break；
…
case    常量表达式 s2；
        程序段 s2；
        break；
…
case    常量表达式 cn：
        程序段 sn；
        break；
default：
        程序段 sn+1；
        break；
}
```

其执行流程是首先计算表达式的值，然后判断表达式的值与 c1、c2、…cn 中的哪个值相等，若与某个 ci 值相等，则执行其下的 si 语句组。若不与任何一个 ci 值相等，则执行 sn+1 语句组。在执行某一分支中的语句组时，遇 break 语句则退出 switch-case 结构，即程序控制转移至该结构中花括号之后的语句，图 3-12 是开关分支的流程图。

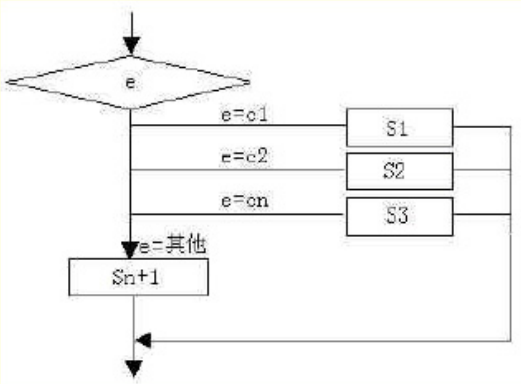


图 3-12　开关分支的流程图

【例 3-11】根据学生成绩的等级打印出分数段。

具体的程序如下：

```
main（ ）
{ char grade;
printf（"input the grade（A,B,C,D,D）:"）;
scanf（" % c",& grade）;
switch（trade）
{case'A':printf（"90-100 \ n"）;break;
case'B':printf（"80-89 \ n"）;break;
case'C':printf（"70-79 \ n"）;break;
case'D':printf（"60-69 \ n"）;break;
case'E':printf（"0-59 \ n"）;break;
default:printf（"error \ n"）;
}
```

```
}
```

在使用 switch-case 分支时，应注意以下几点：

（1）switch 后面的表达式可以是整型、字符型或枚举类型表达式。

（2）case 后面的判断值要求是一个整常量表达式，它可以是一个整数、字符常量、枚举常量或整常量表达式。

（3）各分支语句组中的 break 语句使控制退出 switch 结构。若没有 break 语句，则程序将继续执行下面一个 case 中的语句组。例如：

```
switch（表达式 C）
 {
case 'a':
x++；
case'b'
x+ +
default：
total+ +；
}
```

在此开关分支语句中，若表达式 C 的取值是'a'，则三个分支都执行，若 C 的取值是'b'，则只执行最后两个分支（即 x+ +和 total+ +），若 C 的取值既不是'a'也不是'b'，则只执行 total + +。

（4）在开关分支结构中，各个 case 及 default（default）之后有 break 语句时）的次序是任意的，但各 case 后的判断值必须不同。

（5）在开关分支中，default 部分不是必须的，如果没有 default 部分，则当表达式的值与各 case 的判断值都不一致时，则程序不执行该结构中的任何部分。例如：

```
switch（表达式 C）
 {
case 0
   x+   =2；
   break；
case 1：
   y+   =2；
   break；
case2：
   z+   =2
   break；
}
```

当 C 的值既不是 0 也不是 1 或 2 时，则程序不执行该开关分支中的任何语句。为了使程序能进行错误检查或逻辑检查，应该使用 default 分支。例如：

```
switch（ch）
 {
   case 'x'：printf（"YES\n"）；
            break；
   case' y'：printf（"NO\n"）；
            break；
}
```

当 ch 的值是其他字符时，就会什么都不输出。如果加上 default 分支，且在 default 分支中输出一定的提示信息，那么，遇到 default 分支时，就可输出提示信息。

增加 default 分支会给逻辑检查带来很多方便。例如，如果用 switch 语句来处理数目固定的条件，而且认为

这些条件之外的值都属于逻辑错误，则可以增加一个 default 分支来识别逻辑错误。

（6）尽管最后一个分支之后的 break 语句可以省略，但我们推荐保留它。在最后一个分支之后有 break 语句是程序设计的好习惯。因为你写的程序要被你自己或他人维护，例如要在最后一个分支之后增加几个 case 分支，如果没有注意到最后一个分支之后没有 break 语句，则原来的最后一个分支就会受新增分支的干扰而失效。

（7）在 switch 分支结构中，如果对表达式的多个取值都执行相同的语句组时，则对应的多个 case 可共同使用同一个语句组。例如在下面的程序段中，case1、case2 和 case3 均共用 case3 的语句。

```
switch（grade）
 {case'A'：
case 'B'：
case 'C'：printf（"pass!\n"）；break；/ *前三个月 case 共用一个 printf 语句。* /
case 'D'：printf（"NO pass! \ n"）；break；
default：printf（"error\n"）；break；
}
```

3.3.6 条件分支程序设计举例

本小节通过一些实例来进一步说明分支程序设计的方法。

【例 3-12】有两个圆台，圆心的坐标分别是（3，3）、（-3，-3），各圆台的半径均为 1.5，高度分别为 10 米和 5 米，圆台以外的地面的高度均为 0 米。输入一个坐标，求此处的高度。

算法设计：判断输入的坐标是否在某一个圆台的范围之内。若在第一个圆台（3，3）之内，则认为此处的高度是 10 米；在第二个圆台内，认为此处的高度是 5 米；否则，认为此处的高度是 0 米。其程序如下：

```
# include < stdio.h >
# include <math.h >
main（）
{float x,y,dist1,dist2;
printf（"input the coordinate:"）；
scanf（" % f% f,& x,& y）；
dist1 =pow（（x -3） * （x -3） + （y -3） * （y -3）,0.5）；
dist2 =pow（（x +3） * （x +3） + （y +3） * （y +3）,0.5）；
if（dist1 < =1.5）
printf（"height=10\ n"）；
else if（dist2 < =1.5）
printf（"height = 5\ n"）；
else
printf（"height = 0 \ n"）；
}
```

执行结果：

```
input the coordinate：21＜CR＞
height=0
input the coordinate：32＜CR＞
height=10
input the coordinate：-3-1.5＜CR＞
height=5
```

【例 3-13】请编写求解一元二次方程根的程序。

算法设计：对于一元二次方程 $ax^2+bx+c=0$**，输入 a、b、c 三个系数后，首先应该判断：若 a 等于零，则将方程当作一次方程求解，否则，求解二次方程，具体方法是根据** b^2-4ac **的值大于 0，小于 0，等于 0 分别求解。可见整个求解过程将使用一个多重嵌套的 if 语句。**

```
main（）
{
float a,b,cp,rpart,ipart,x1,x2;                    / * 定义实型变量名 * /
printf（" \ n Enter a,b,c,:"）;                      / * 输入方程的系数 * /
scanf（" % f% f% f,&a,&b,&c）;
if（a= = 0）                                        / * 等于零，则方程为一次方程 * /
{
if（b = =0）                                       / * 判断 b 是否等于 0 * /
printf（" \ n a,b,c are illegal !"）;
else
{
x1=- c / b;
printf（"There is one root: x = % f",x1）;
}
}                                                  / * 此方程是一个二次方程 * /
else
{
p =b * b - 4 * a * c;
if（p > = 0）
if（p= = 0）
{
x1 =-b /  （2 * a）;
printf（"double root x = % f",x1）;
}
else
{
x1= -b  （2 * a）+sqrt（p）  /  （2 * a）;
x2= -b/（2 * a）-sqrt（p）  /  （2 * a）;
printf（" \ n x1 = % f x2 = % f",x1,x2）;
}
else
{
rpart = -b /  （2 * a）;
ipart= sqrt（-p）  /  （2 * a）;
printf（" \ n bas has comphex roots:"）;
printf（" \n x1 = % f + % fi",rpart,ipart）;
printf（"\n x2 = % f - % fi",rpart,ipart）;
}
}
}
```

【例 3-14**】请用** switch **结构和** if-else **结构分别编制某运输公司计算运费的程序。运行程序时，用户输入运输距离和运量，程序将给出单价和总金额。**

收费标准为：s<500km **时没有优惠，单价为** 5 **元/（吨·公里）；**500km≤s＜1000km **优惠** 2%**；**1000km≤s＜2000km **优惠** 5%**；**2000km≤s＜3000km **优惠** 8%**；**3000km≤s **优惠** 10%**。**

其程序如下:

```
# include < stdio.h >
main（）
{int distance,ton,grade;
float unit_price,total_price;
printf（"input the distance and weight:"）；
scanf（" % d% d,& distance,& 100）；
grade = distance/500;
switch（grade）
{
case 0:unit_price=5;break;
case 1:unit_price=5*0.98;break;
case 2:
case 3:unit_price=5*0.95;break;
case 4:
case 5:unit_price=5 *0.92;break;
default:unit_price=5*0.9;break;
}
total_price=unit_price*ton*distance;
printf（"the unit price is % .3f.\nthe total price is %.3f n",unit_price,total_price）；
}
```

此例用了 switch **结构的分支，当然也可以用如下的** if-else **分支代替** switch**：**

```
if（distance<500）
unit_price=5;
else if（distance>=500&distance<1000）
unit_price=5*0.98;
else if（distance>=1000&&distance<2000）
unit_price=5*0.95;
else if（distance>=2000&&distance<3000）
unit_price=5*0.92;
else unit_price=5*0.9;
```

【例 3-15**】写程序，判断某一年是否闰年。**

在本例中，以变量 leap **代表是否闰年。如果是闰年，令** leap=1**。非闰年，**leap=0**。最后判断** leap **是否为** 1**（真）；若是，则输出“ 闰年 ”信息。**

程序如下：

```
main（）
{
int year,leap;
scanf（"%d",&year）；
if（year%4 = = 0）
   {if（year%100 = =0）
      {if（year%400 = =0）
           leap = 1;
        else leap = 0;}
```

```
        else
        leap = 1;}
    else
      leap = 0;
    if（leap）
    printf（"%d is",year）；
    else
    printf（"%d is not",year）;
    printf（"a leap year\n"）；
    }
```

运行情况如下：

（1）2001

```
2001 is not a leap year
```

（2）2004

```
2004 is a leap year
```

也可以用一个逻辑表达式包含所有的闰年条件，将例子中的 if 语句用下面的 if 语句来代替：

```
if（（year %4 = = 0 && year %100 ! = 0）||（year % 400 = = 0））  leap = 1;
else leap = 0;
```

3.4 循环结构程序设计

循环结构是结构化程序设计的基本结构之一，其应用相当广泛，几乎所有的应用程序都会用到循环结构。C 语言提供了 while、do-while 和 for 三种语言来实现循环，下面分别介绍这三种语言。

3.4.1 while 语句

用 while 语句实现当型循环结构。其算法描述如图 3-13 所示。

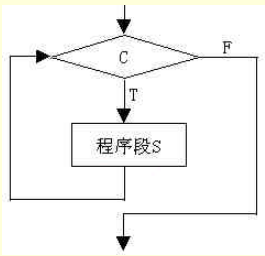


图 3-13 while 的算法描述

其程序形式如下：

```
while（条件表达式 C）
 {
        程序段 S；
}
```

当 C 取值非 0 时（即逻辑值为“真”），则执行程序段 S，否则跳出循环。程序段 S 可以是单个语句、空语句或复合语句，称之为循环体。

【例 3-16】使用 while 循环求两个正整数的最小公倍数。

其程序如下：

```
# include < stdio.h >
main（）
```

```
{int m,n,result;
printf（"input two integers:"）；
scanf（" % d% d",& m,& n）；
result=m <n? n:m;
while（ !（result % m= =0& &result % n= =0））
result+ +;
printf（"the lease common multiple of m and n is % d \ n",reault）；
}
```

注意到执行 while **循环时，是先对条件进行判断，满足条件（逻辑值为“真”）则执行循环体，否则跳出循环，这与** do-while **循环的执行方式是不同的。下面介绍** do-while **循环的使用。**

3.4.2 do-while 语句

do-while **语句用于实现直到型循环结构，其算法描述如图** 3-14 **所示。**

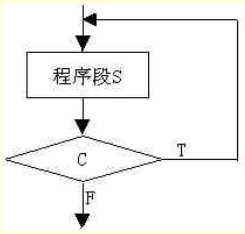


图 3-14 do-while **循环**

其程序形式如下：

```
do｛
    程序段 S；
｝
while（条件表达式 C）；
```

程序段 S **是循环体，它可以是单个语句、空语句或复合语句。**

从流程图中可以看出，do-while **循环的执行流程是先执行循环体，再计算表达式** C**，然后判断它是否非** 0**，如果非** 0**，则返回重新执行循环体，否则结束循环。所以** do-while **循环是先执行一次循环体，再判断是否继续循环。**

【例 3-17**】使用** do-while **循环求两个正整数的最小公倍数。**

其程序如下：

```
# include < stdio.h >
main（）
{int m,n,result;
printf（"input two integers:"）；
scanf（"% d % d,& m,& n）；
result =m > n? m:n;
result - -;
do{
result+ +;
{while（result % m!=0||result % n!=0）；
printf（"the lease sommon multiple of m and n is % d \ n",result）；
}
```

对比使用 while **循环和** do-while **循环求解同样的问题可以看出，其不同之处主要在对初值的处理上。使用** do-while **语句时，应该注意以下几点：**

（1）在 do-while 循环中，while（c）之后的分号（；）不要忘掉。（在 while 循环中，while（c）之后是没有分号的。）

（2）在 do-while 循环中，不管循环体是否为单一语句，习惯上都用花括号把它括起来，并把“ while（c）；”直接写在“ ｜ ”的后面，以免把“ while（c）：”部分误认为是一个新的 while 循环的开始。

【例 3-18】用 do-while 语句求 $\sum_{n=1}^{100} n$。

程序如下：

```
main（）
{
int i,sum=0;
i=1;
do
{sum=sum+i;
i++;
}
while （i<=100）;
printf（"%d",sum）;
}
```

可以看到，对同一个问题可以用 while 语句处理，也可以用 do-while 语句处理。do-while 结构可以转换成 while 结构。可见，do-while 结构是由一个语句加一个 while 结构构成的。

在一般情况下，用 while 语句和用 do-while 语句处理同一问题时，若二者的循环体部分是一样的，它们的结果也一样。但在 while 后面的表达式一开始就为假（0 值）时，两种循环的结果是不同的。

【例 3-19】while 和 do-while 循环的比较。

（1）

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

运行情况如下：

```
1↙
55
```

（2）

```
main（）
{int sum=0,i;
scanf（"%d",&i）;
      do
{sum=sum+i;
i++;
}

    printf（"%d",sum）;
 }
```

运行情况如下：

```
1↙
55
```

可以看到：当输入 i 的值小于或等于 10 时，二者得到结果相同。而当 i＞10 时，二者结果就不同了。这是因为此时对 while 循环来说，一次也不执行循环体（表达式“ i＜＝10 ”为假），而对 do-while 循环来说则要执行一次循环体。可以得到结论：当 while 后面的表达式的第一次的值为“ 真 ”时，两种循环得到的结果相同。否则，二者结果不相同（指二者具有相同的循环体的情况）。

还要注意一点：do-while 循环是先执行循环体语句，后判断表达式，从这点来说，它类似于直到型循环，但它与其他语言中的 until 型循环（不同，do-while 循环是当表达式为真时反复执行循环体，表达式为假时结束循环。而典型的 until（直到型）循环结构则是表达式为真时结束循环。

3.4.3 for 语句

循环结构中，for 语句的功能比 while 语句更强，使用也更为灵活方便。原则上，所有的 while 循环都可以

用 for 循环来替代。for 循环的处理流程如图 3-15 所示。

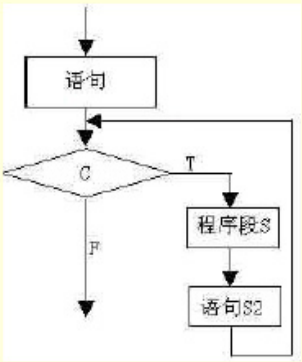


图 3-15　for 循环

程序形式为：

```
for（语句 S1；条件表达式 C；语句 S2）
 {
    程序段 S；
}
```

语句 S1、S2，条件表达式 C 的作用各不相同。原则上说，S1 和 S2 可以是一般的语句，但实际上 S1 常常是一个在初次进入循环之前给某些变量赋值的表达式，也叫初值表达式；S2 常常是一个在每次执行循环体后对某些变量进行修改的表达式，也叫修改表达式。条件表达式 C 用于指定循环条件，其作用同 while 语句中的表达式 C 相同，它通常是关系表达式或逻辑表达式。S1、S2、C 中的任何一个均可省略，但其后面的分号决不能省略。如果省略 C，则循环条件将恒为“真”。例如：

```
for（：：）
 {
    …
}
```

上述结构是下一个无限循环，需要通过其他手段（比如 break 语句）结束循环。

程序段 S 是循环体，它可以是单个语句、空语句或复合语句，如果是单个语句或空语句，可以省去花括号。如果循环体是由多个语句组成，则必须用花括号括起来。

for 循环的执行流程如下：

（1）执行语句 S1。

（2）计算机表达式 C，判断其值，若为“真”（即非 0），则执行（3）；若为“假”（即 0），则结束循环。

（3）执行循环体 S。

（4）执行语句 S2。

（5）重复执行（2）。

【例 3-20】利用 for 循环计算：$1^2+2^2+3^2+4^2+\cdots 20^2$

其程序如下：

```
# # include < stdio.h >
main（）
{int k;
lont result=0;
for（k=1;k< =20;k+ +）
result+ =k * k;
printf（"the result: %ld \ n",result）；
}
```

for 循环语句和与之等价的 while 循环语句的对应关系如下：

for 语句 对应的 while 语句

for（S1；C；S2） S1；

{

S； while（C）

{

} S；

S2

}

3.4.4 三种循环的比较

C 语言提供的三种循环语句可以用来处理同一问题，一般情况下可以互换。但其功能和灵活程度不同，对这三种循环的比较如下：

（1）for 语句功能最强，最方便灵活，使用最多，任何循环都可以用 for 实现。while 和 do-while 用的最少（但这并非它没用，对某种情况它还是很有价值的）。

（2）while 和 do-while 的循环变量初始化是在循环语句之前完成，而 for 语句循环变量的初始化是在 for 中的 S1 语句（常常是一个表达式）中实现。

（3）for 循环中的第一个和第三个语句（S1、S2）可以是逗号表达式，这是 for 语句的一个很有用的特性。它扩充了 for 的作用范围，使它有可能同时对若干参数（如循环变量，重复计算参数等）进行初始化和修改等。

（4）for 和 while 循环是先判断循环条件，后执行循环体；而 do-while 循环则是先执行一次循环体，然后才判断循环条件。因此，后者不管什么情况下，都至少要执行一次循环体。

【例 3-21】判断某字符串是否是前后对称的。

其程序如下：

```
# include < stdio.h >
# include <stirng.h >
main（）
{int j,k,sign;
char str[50];
printf（"input a str;"）；
scanf（"% s",str）；
for（j=0,k=strlen（str）-1,sign=0;k>j;j+ +,k- -）
if（str[j]!=str[k]）
sign=1;
if（sign= =1）
printf（"not a symmetrical string. \ n"）；
else
printf（"a symmetrical string. \ n"）；
}
```

该例中用到了数组，其中 s［i］表示数组中的第 i 个元素。s［i］＝s［j］表示把数组中第 j 个元素中的字符赋给数组中的第 i 个元素。有关数组的概念请参看第四章。在程序中，由于可以用逗号表达式，就使得循环灵活多了。

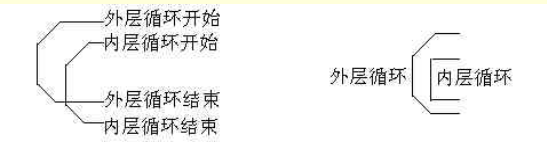
3.4.5 多重循环

在一个循环的循环体内又包含另一个循环，这称之为循环的嵌套。被嵌入的循环又可以嵌套循环，这就是多重嵌套，又称作多重循环。在实际应用中，经常要用到多重循环。

C 语言提供的三种循环语句都可以互相嵌套。

在使用循环相互嵌套时，被嵌套的一定是一个完整循环结构，即两个循环结构不能相互交叉。如图 3-16 所

示，其中（a）是非法的嵌套，（b）是合法的嵌套。



(a) (b)

图 3-16 循环的合法与非法嵌套

下面是一个多重循环的例子。

【例 3-22】有一个数 43634，其左右对称，求比它大的对称数中最小的那一个。

本程序使用两重嵌套循环，外层循环用于由小到大逐一列举比 43635 大的整数，直到找到一个对称数，内层循环用于判断列举到的每一个是否是对称的，具体方法是将此整数的每一位分离出来进行比较。

其程序如下：

```
# include < stdio.h >
main（）
{long int i=43634,j;
int count,ch[5],sign=0;
do|i+ +;
j=i;
count=0;
while（j）
{ch[count]=j% 10;
j=j/10;
count+ +;
}
if（ch[0]= =ch[4]& &ch[1]= =ch[3]）
sign=1;
}while（! sign）;
printf（"% ld \ n",i）;
}
```

3.4.6 循环和开关（switch）分支的中途退出

前面我们讨论了 C 语言提供的三种循环，它们都是根据循环判断表达式为 0 时来控制循环结束，这种结束是正规的循环结束。在实际应用中，往往还要求在循环的中途退出循环，这是一种非正规的循环退出。实现非正规循环退出的语句有 break 和 continue。

1．break 语句

break 语句的功能是退出当前循环或当前 switch 结构。如果 break 语句处在一个循环体中，则执行它时退出当前所在的循环，即结束当前层循环。如果 break 语句处在 switch 语句的一个分支中，则执行它时退出当前所在的 switch 结构。

【例 3-23】使用 break 语句改写例【3-21】的程序。

在例【3-21】的循环中，if 语句对一个标志变量 sign 进行设置，而 sign 则作为判断是否应跳出循环的标志，可以在使用变量 sign，而在 if 语句中直接使用 break 语句跳出循环。更改后的程序如下：

```
# include < stdio.h >
# include < string.h >
main（）
{ int j,k;
```

```
char str[50];
printf（"input a str:"）;
scanf（"% s",str）;
for（j=0,k=strlen（str）-1;k >j;j+ +,k- -）
break;
if（k>j）
printf（"not a symmetrical string. \ n"）;
else
printf（"a symmetrical string. \ n"）;
```

从程序中可以看出，当发现不对称的字符时，立即跳出循环。C 语言中规定，break 语句只能用于循环语句和 switch 语句中，不能用于其他任何语句之中。

2．continue 语句

continue 语句用于结束本次循环。其一般形式如下：

```
continue；
```

continue 与 break 的区别是：continue 结束本次循环，而不是结束整个循环结构，break 是结束整个循环结构，不再进行循环条件判断而 continue 则还要进行循环条件判断，又可能继续下一次循环。另外，continue 语句只能用在循环语句中，而不能用在 switch 和其他语句中。

使用 continue 语句时需注意：结束本次循环后控制转移到什么地方。下面举例说明 continue 的用法。

【例 3-24】输入一个串字符，统计其中小写字母的个数。

其程序如下：

```
# include < stdio.h >
main（）
{char c;
int num=0;
while（（c=getchar（））!=' \ n'）{
if（c<97 | | c >122）
continue;
num+ +;
}
printf（" % d \ n",num）;
}
```

程序接受用户输入的下一个字符串（以回车终止输入），并使用循环逐一地检查每一个字符，若发现当前字符不是小写字母，立即使用 continue 结束本次循环，因而计算变量的值不增加；否则，执行整个循环体，使计数变量的值加 1。

3.4.7 goto 语句

goto 语句是一个条件分支语句，其功能是将控制转移到指定位置继续执行。goto 的一般格式如下：

```
goto 标号；
```

goto 语句中必须有标号，当执行该语句时，控制被转向标号所标位置的语句继续执行。标号是一个标识符，它标识程序中的下一个特定位置。标号的构成文法同一般用户定义标识符。标号被定义在某个执行语句的前面。可以和该执行语句处在同一行，也可以单独成行，处在该执行语句的上一行。被定义的标号一定以冒号结尾。它和其所标的语句之间可有一个或多个空格字符，但不许出现别的内容。标号只能在 goto 语句中引用，在其他地方都不会被引用。带标号的语句称作标号语句。

goto 语句的使用范围只局限于一个函数内部，也就是说，标号的作用范围是在其所在的函数内部。因此，只能用 goto 语句把控制从函数的一个位置转到另一个位置，而不可能将控制从一个函数的某点转到另一函数的指定位置。goto 在程序中最常见的用法有两种：一是实现控制从多重循环内部退出；二是构成循环。

在结构化程序设计中，不提倡使用 goto 语句，因为控制的自由分支会使程序的基本结构受到破坏，从而降低程序的可读性和可维护性。但是，在有些特定情况下，使用 goto 又为程序设计带来方便和提高程序执行速度。因此，只能有节制地使用它。下面是表示 goto 语句作用的例子。

【例 3-25】用户输入一串字符，以' \ n '作为结束标志，请使用 goto 语句跳出循环。

其程序如下：

```
# include < stdio.h >
main（）
{ char c;
printf（"input a $"）；
do{
scanf（"% c",& c）；
if（c = = ' \ n'）
goto end;
}while（1）；
end:
printf（"end! \ n"）；
}
```

显然，可以使用 break 语句或更改 for 语句中的循环结束的条件来去掉 goto 语句。

【例 3-26】利用 goto 语句构成循环，计算：$1^2+2^2+3^2+4^2+\cdots 20^2$。

其程序如下：

```
# include < stdio.h >
main（）
{int k=1;
long result=0;
loop:
result + = k* k;
k+ +;
if（k < =20）
goto loop;
printf（"result: % ld \ n",result）；
}
```

由于 for 循环、while 循环、do-while 循环能圆满而方便地解决所有的循环问题，所以由 goto 语句构成循环的方法并不经常使用。

3.5 结构化程序举例

【例 3-27】将小于 n 的所有个位不等于 9 的素数在屏幕上打印出，n 的具体值由用户输入来确定。素数即为只能被 1 和本身整除的整数。要求每行输出 10 个数，分行输出。

分析：可以使用循环从 2 到 n 逐一检查每一个数 i，在循环体中判断其是否满足“是素数且个位不等于 9”的条件，判断数 i 是否是素数时，可以再使用一个内层循环，从 2 到 i -1 逐一检查每一个数是否是它的约数，若都不是，则证明其为素数。然后，使用表达式 i %10 求出 i 的个位的数字（注：请读者自己考虑两个问题，一是是否需要逐一检查从 1 到 n 的每一个数，二是判断 i 是否为素数时检查约娄得否有必要从 1 循环到 i-1）。

其程序如下：

```
# include < stdio.h >
main（）
```

```
{ long int n;
int line=0;
printf（" \ n please input n:"）;
scanf（" % d",& n）;
if（n < =1）
{printf（"No number to output ! \ n"）;
return（1）;
}
{printf（"No number to output ! \ n"）;
return（1）;
}
for（int i=2;i< =n;i+ +） / *逐一地判断各个数是否是素数 * /
{for（int j=2;j<i;j+ +）
if（i/j= =i） / * 若此数有约数.* /
break;
if（j= =i &&i%10! =9） / 若此数为素数且个数不等于 9. / *
{printf（" % d,",i）;
line + +;
if（line= =10）
{printf（" \ n"）;
line=0;
}
}
}
return（1）;
}
```

【例 3-28】编程求 0～9 的有一个数字 x、y、z 使其满足：xyz+zyx=123（注：xyz 指由 x、y、z 在三个数字组成的三位数）。

算法设计：使用三重循环穷举 x、y、z 在取值范围内的所有组合，对每一种组合代入方程进行验证。其程序如下：

```
# include < stdio.h >
main（）
{int i,j,k;
for（i=1;i< =9;i+ +）
{for（j=0;j< =9;j+ +）
{for（k=0;k< =9;k+ +） / * 穷举了 X，Y，Z 所有的取值的组合 * /
{if（101 * i +20 * j +101 * k= =1231） / * 若满足方程 * /
printf（" x= % d,y=% d,z=% d:xyz=%d % d % d,zyx=% d % d % d \ n",
i,j,k,i,k,k,j,i）;
}
}
}
return（1）;
}
```

【例 3-29】本程序为一个菜单程序。运行时，首先显示下一个菜单画面用以提示输入操作选择，这时，操作员从菜单上选择一个操作（即输入相应的代码 1，2，…），程序接收该选择后调用相应的函数完成操作。设菜

单画面格式如图 3-17 所示，并且系统提供了 select（）、Insert（）、update（）和 delete（）函数，用以实现相应的操作。

图 3-17 菜单格式

算法设计：程序编制过程主要有五个方面，即：生成菜单画面、提示和接收操作员选择、判断操作员选择、执行相应操作和重复以上操作。其程序如下：

```
# include < stdio.h >
void insert （） ,select （） ,delete （） ,update （） ;
main （）
{
char op;                / * generate menu * /
printf （" \ n* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *"） ; / *生成菜单 */
printf （" \ n *                 Menu section            *"） ;
printf （" \ n *                 1.  Insert              *"） ;
printf （" \ n *                 2.  Select              *"） ;
printf （" \ n *                 3.  Delete              *"） ;
printf （" \ n *                 4.  Update              *"） ;
printf （" \ n *                 5.  Exit                *"） ;
printf （" \ n* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *"） ; / *生成菜单 */
while  （1）            / * selection opration * /
{
printf （" \ n Please enter selection:"） ;
scantf （"% d,& op） ;
switch （op）                        / * 根据输入，选择分支走向 * /
{
case ' 1 ': insert （） ;
break;
case' 2 ':select （） ;
break;
case ' 3 ':delete （） ;
break;
case ' 4 ':update （） ;
break;
case' 5 ':break;
default:
printf （" \ n Selection error !"） ;
break;
}
if （op = = ' 5 '）
break; / * 退出循环 * /
```

```
}
}
```

【例 3-30】打印某月的日历，已知当月共有 30 天，而 1 号是星期三。

算法设计：本题的关键是打印格式的控制。第一行等间隔地打印好周日至周六七个符号后，下一行开始打印日期，每一个日期与其对应的星期对齐。其中，本月 1 号对齐星期三，每次打印到星期六对应的那一天后换行。判断换行的方法是：当（data+2）%7＝＝6 时换行。其程序如下：

```
# include < stdio.h >
main（）
{ int k;
printf（"- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - \ n"）; / * 打印前分隔符 * /
printf（ " Sun Mon Tue Wed Thu Fri Sat \ n"）;
printf（" "）;
for（k=1;k< =30;k+ +）{                    / * 打印日期 * /
printf（ " % - 2d",k）;
if（（k+2）% 7 = =6）                      / * 打印到星期六时，换行 * /
printf（" \ n"）;
}
printf（"- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - \ n"）; / * 打印后分隔符 * /
return 1;
}
```

【例 3-31】用 $\frac{p}{4} \approx 1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$ 公式求 π 的近似值，直到最后一项的绝对值小于 10^{-6} 为止。

程序如下：

```
#include <math.h>
main（）
{
int s;
float n,t,pi;
t=1;pi=0;n=1.0;s=1;
while（（fabs（t））＞=1e-6）
{pi=pi+t;
n=n+2;
s=-s;
t=s/n;
}
pi=pi*4;
printf（"pi=%10.6f\n",pi）;
}
```

运行结果为

```
pi= 3.141397
```

【例 3-32】求 Fibonacci 数列：1，1，2，3，5，8，……的前 40 个数，即

```
F1＝1              （n=1）
F2＝1              （n=2）
Fn=Fn-1+Fn-2       （n≥3）
```

程序如下：

```
main（）
{
long int f1,f2;
int i;
f1=1;f2=1;
for（i=1;i<=20;i++）
{
printf（"%12ld  %12ld  ",f1,f2）;
if（i%2= =0）printf（"\n"）;
f1=f1+f2;
f2=f2+f1;
}
}
```

运行结果为：

1	1	2	3
5	8	13	21
34	55	89	144
233	377	610	987
1597	2584	4181	6765
10946	17711	28657	46368
75025	121393	196418	317811
514229	832040	1346269	2178309
3524578	5702887	9227465	14930352
24157817	39088169	63245986	102334155

程序中在 printf 函数中输出格式符用“ %12 ld ”，而不是用“ %12d ”，这是由于在第 22 个数之后，整数值已超过微型机的整数最大值 32767，因此必须用“ %ld ”格式输出。

if 语句的作用是使输出 4 个数后换行。因为 i 是循环变量，当 i 为偶数时换行，而 i 每增值 1，就要计算和输出 2 个数（f1,f2）,因此 i 每隔 2 换一次行相当于每输出 4 个数后换行。

【例 3-33】判断 m 是否素数。

我们采用的算法是这样的：让 m 被 2 到 $\sqrt{m}$ 除，如果 m 能被 2~$\sqrt{m}$ 之中任何一个整数整除，则提前结束循环，此时 i 必然小于或等于 k（即 $\sqrt{m}$）；如果 m 不能被 2~k（即 $\sqrt{m}$）之间的任一整数整除，则在完成最后一次循环后，i 还要加 1，因此 i=k+1，然后才终止循环。在循环之后判别 i 的值是否大于或等于 k+1，若是，则表明未曾被 2~k 之间任一整数整除过，因此输出“ 是素数 ”。

程序如下：

```
#include <math.h >
main（）
{
int m,i,k;
scanf（"%d",&m）;
k=sqrt（m）;
for（i=2;i<=k;i++）
if（m%i= =0）break;
if（i>=k+1）
printf（"%d is a prime number\n"n",m）;
else
printf（"%d is not a prime number\n",m）;
```

```
}
```

运行情况如下：

```
17↙
17 is a prime number
```

【例 3-34】求 100~200 间的全部素数。

在【例 3-33】的基础上，对本题用一个嵌套的 for 循环即可处理。程序如下：

```
#include <math.h >
main（）
{
int m,k,i,n=0;
for（m=101;m<=200;m=m+2）
{
if（n % 10 = =0）
printf（"\n"）;
k=sqrt（m）;
for（i=2;i<=k;i++）
if（m % i = =0）break;
if（i>=k+1）
printf（"%d    ",m）;n=n+1;
}
}
```

运行结果如下：

```
101   103   107   109   113   127   131   137   139   149
151   157   163   167   173   179   181   191   193   197
199
```

n 的作用是累计输出素数的个数，控制每行输出 10 个数据。

【例 3-35】输入一行字符，分别统计出其中英文字母、空格、数字和其他字符的个数。

程序如下：

```
#include <stdio.h>
main（）
{
char c;
int letters=0,space=0,digit=0,other=0;
printf（"请输入一行字符：\n"）;
while（（c=getchar（））!='\n'）
{
if（c>='a'&&c<='z'‖c<='A'&&c<='z'）
letters++;
else if（c= =' '）
space++'
else if（c>='0'&&c<='9'）
digit++;
else
other++;
}
printf（"其中：字母数＝%d 空格数＝%d 数字数＝%d 其他字符数＝%d\n",letters,space,digit,other）;
```

```
}
```

运行结果：

请输入一行字符：

I am 16.at 156——3.↙

其中：字母数＝5 **空格数＝**4 **数字数＝**6 **其他字符数＝**4

【例 3-36**】打印出所有的“水仙花数”，所谓“水仙花数”是指一个三位数，其各位数字立方和等于该数本身。例如：**153 **是一“水仙花数”，因为** 153＝1^3+3^3+5^3**。**

程序代码如下：

```
main（）
{
int i,j,k,n;
printf（"水仙花数是："）;
for（n=100;n<1000;n++）
{
i=n/100;
j=n/10-i*10;
k=n%10;
if（i*100+j*10+k= =i*i*i+j*j*j+k*k*k）
{
printf（"%d",n）;
}
}
printf（"%n"）;
}
```

运行结果：

水仙花数是：153 370 371 407

【例 3-37**】一个数如果恰好等于它的因子之和，这个数就称为“完数”。例如：**6 **的因子为** 1、2、3，**而** 6＝1+2+3，**因此** 6 **是“完数”。编程序找出** 1000 **以内的所有完数，并按下面格式输出其因子。**

6 **是一个‘完数’，它的因子是** 1，2，3.

程序代码如下：

```
#define M 1000/*定义寻找范围*/
main（）
{
int k0,k1,k2,k3,k4,k5,k6,k7,k8,k9;
int i,j,n,s;
for（j=2;j<-M;j++）
{
n=0;
s=j;
for（i=1;i<j;i++）
{
if（（j%i）= =0）
{
n++;
s=s-i;
switch（n）/*将每个因子赋给 k0,k1……k9*/
```

```
{
case 1:
k0=i;
break;
case 2:
k1=i;
break;
case 3:
k2=i;
break;
case 4:
k3=i;
case 5:
k4=i
berak;
case 6:
k5=i
berak;
case 7:
k6=i;
berak;
case 9:
k8=i;
break;
case 10:
k9=i;
break;
}
}
}
if（s= =0）
{
printf（"%d 是一个'完数'.它的因子是",j）；
if（n>1）
printf（"%d,%d",k0,k1）；
if（n>2）
printf（",%d",k2）；
if（n>3）
printf（",%d",k3）；
if（n>4）
printf（",%d",k4）；
if（n>5）
printf（",%d",k5）；
if（n>6）
printf（",%d",k6）；
if（n>7）
```

```
printf（",%d",k7）;
if（n>8）
printf（",%d",k8）;
if（n>9）
printf（",%d",k9）;
printf（"\n"）;
}
}
}
```

运行结果：

6 是一个'完数'，它的因子是 1，2，3

28 是一个'完数'，它的因子是 1，2，4，7，14

496 是一个'完数'，它的因子是 1，2，4，8，16，31，62，124，248

3.6 小 结

（1）算法就是处理问题的方法和步骤。在编写程序之前首先要进行算法设计。任何一个算法必须具有“有穷性、确定性、有效性、有 0 个或多个输入和有一个或多个输出”等特征。算法有三种基本结构：顺序、分支和循环。任何一个程序都可由这三种基本结构组成。

（2）C 语言提供了四类语句：控制语句、表达式语句、空语句和复合语句。控制语句有 if、for、while、do-while、continue、break、switch、goto 和 return 等九个语句。

（4）表达式语句是用分号结尾的表达式。C 语言表达式的多样性，也就决定了表达式语句的多样性。函数调用语句也属表达式语句。

（5）空语句是只包含一个分号的语句。

（6）复合语句是由花括号括起来的一段程序，它可以包含说明部分，复合语句又叫局部程序块或分程序。

（7）C 语言提供了 if 和 switch 两种语句来实现分支结构。if 语句还有 if、if-else 和 if-else if 三种形式，其中，if-else 是最基本的形式。if 后面的表达式不限于关系表达式或逻辑表达式，可以是任意表达式。if 语句可以嵌套，在嵌套的 if 中，else 子句总是与前面最近的还没 else 的 if 相配对。if-else if 语句是 if-else 语句嵌套的简化表示形式，用它可以方便地实现多分支结构。

switch 语句用于实现多分支结构，如果有两个以上基于同一个数字型变量的条件表达式时，选用 switch 要比选用 if 更好。

（8）C 语言提供了 while、for、do-while 三种语句来实现循环结构。for 语句功能最强、更灵活、使用最多。for 和 while 语句是先判断后执行循环体，而 do-while 则是执行一次循环体后才判断。

（9）break 语句用于结束其所在的 switch 分支结构或循环结构，continue 语句用于结束本次循环。

（10）goto 语句用于使控制转向指定点执行，在结构化程序设计中要求有节制地使用 goto 语句。

习 题

1．在 C 语言中，有哪几种基本结构？

2．编写程序，从键盘上输入 4 个整数，输出其中最小的一个。

3．求 $1^2/(1+1)+2^2/(2+1)+3^2/(3+1)+\cdots\cdots+n^2/(n+1)$ 的值，n 由用户确定。请使用两种以上的不同循环分别编写本题。

4．编写程序，输入一个字符串，将其中的空格和数字删除后输出。

5．编写程序，打印出 sinx 在 -180^0~180^0 之间的值，要求每 10^0 计算一次，结果以下列形式输出，要求：每输出 10 行停顿一次，按下回车键后继续执行。

sin:

-180----------xxx

-170----------xxx

……

第四章　数组及其应用

迄今为止，我们使用的都是属于基本类型（整形、字符型、实型）的数据，C 语言还提供了构造类型的数据，它们有：数组类型、结构体类型、共用体类型。构造类型数据是由基本类型数据按照一定的规则组成的，因此有的书又称它们为“导出类型”。

本章将只介绍数组。数组是具有相同数据类型且按一定次序排列的一组变量的集合体，构成一个数组的这些变量称为数组元素。数组有一个统一的名字叫数组名，每个数组元素并没有另外的名字，它可通过数组名及其在数组中的位置（叫下标）来确定。即数组元素是用数组名后跟用方括号（[]）括住的下标来表示。例如 name［15]、list［5］［10］等。数组按下标个数分类，有一维、二维和三维数组等，二维以上数组统称为多维数组。

4.1　一维数组

4.1.1　一维数组的定义

一维数组是数组名后只有一对方括号的数组，其定义方式为：

数据类型　　数组名［元素个数］；

例如：

```
char str［50］
```

此语句定义了一个由 50 个元素组成的一维数组，数组名为 str，这 50 个元素分别为 str［0]、str［1]、str［2]、str［3］…str［49]，每个元素都是字符型变量。关于数组的定义，应注意如下几点：

（1）数组名后用方括号括住数组元素的个数，不能使用圆括号。

（2）元素个数可以是整型常量，也可以是整型常量表达式，但决不能含有变量，因为此表达式的值是在编译时计算出来的，而编译时系统并不能确定变量的取值。

（3）数组元素个数必须大于或等于 1。

（4）数组元素的下标是从 0 开始编号的，因此，对于定义：float a［8]，其第一个元素是 a［0］而不是 a［1]，最后一个元素是 a［7]，而不是 a［8]。

根据上述要点可以判断，下面四个定义是非法的：

```
int size1，size2；
folat height［size1］；                    （使用变量做下标是错误的）
float width［size1+size2+1］；             （用含变量的表达式做下标是错误的）
int number［-8］                           （使用负数做下标是错误的）
int m（8）                                 （数组名后面用（）是错误的）
```

而下面的数组定义是合法的：

```
# define STRSIZE 50
char string［STRSIZE］；
int m［15* STRSIZE］；
float x［3*32+1］；
```

4.1.2 一维数组的存储形式

一维数组在内存中存储时，按下标递增的次序连接存放。对于 int a［15］数组名 a 或 a［0］是数组存储区域的首地址，即数组第一个元素存放的地址。其中＆是地址运算符，它表示取 a［0］的地址。因此，数组名是一个地址常量，不能对其进行赋值和进行＆运算。

4.1.3 一维数组的引用

与变量类似，任何一个数组都应先定义或说明，然后再引用。在 C 语言中不能对数组整体进行操作，例如不能对整个数组进行赋值或其他各种运算。只能对数组元素进行操作。数组的引用形式为：

数组名［下标］

其中下标既可以是整型常量表达式，也可以是含有变量的整型表达式。例如，在例中的 x［k+2］。

```
# define SIZE 4
# include < stdio.h >
main（）
{
float x[SIZE],sum= =0.0;
int k =0;
scanf（" % d % d % d",x）；                    / * 对数组进行整体输入是错误的 * /
scanf（" % d",& x[3]）；                       / * 正确的输入方式 * /
while（k <SIZE）{
sum + = x[k];                                  / * 正确的引用方式 * /
k + +;
}
k=0
printf（" % d % d % d % d \ n",x）；             / * 对数组进行整体输出是错误的 * /
printf（" % d % d \ n",x[k+1],x[k+2]）；        / * 正确的引用方式 * /
printf（" % d \ n",x[SIZE]）；                  / * 下标越界 * /
}
```

在 C 语言中，编译和执行程序时，系统并不自动检查数组下标是否越界，因此可能访问到数组所占的内存空间以外的存储单元，而这种访问往往是十分危险的，因此程序员要特点注意下标越界的问题。

4.1.4 一维数组的初始化

初始化就是给数组元素赋初始值，有如下两种初始化方法：

1．用赋值语句初始化

用赋值语句初始化是在程序执行时实现的。

2．在数组定义时初始化

这种初始化是在编译时进行的。其一般形式如下：

数据类型 数组名［数组元素个数］＝｛值 1，值 2，…值 n｝；

对上述形式说明如下：

- 花括号中的值是初始值，用逗号分开。例如：

  int m［3］＝｛0，1，2｝；

那么各数组元素的初始值为：m［0］＝0，m［1］＝1，m［2］＝2。

- 如果花括号中值的个数少于数组元素的个数，则多余的数组元素初始化为 0。例如：

  int m［3］＝｛0，1｝；

则各数组元素的初始值为：m［0］＝0，m［1］＝1，m［2］＝0。

- 可对数组中的部分元素赋值，这时对不赋值的数组元素可在括号中缺少相应的值，但逗号不能省略，而缺省值视为 0。例如：

int m［3］＝｛0，，1｝；

此时各数组元素的初始值为：m［0］＝1，m［1］＝0，m［2］＝1。

- 在数组定义中，可缺省方括号（[]）中的元素个数，而用花括号中初始值的个数来决定数组元素个数。例如：

int m［］＝｛0，1，2｝；

相当于：

int m［3］＝｛0，1，2｝；

- 对于静态或全局类型的数组，如果不在定义显示初始化，则多数编译系统都将其初始化为 0。

4.1.5 一维数组的应用举例

【例 4-1】有一递推数列，满足：f（0）＝0，f（1）＝1，f（2）＝2，f（n+1）=f（n）+2f（n-1）f（n-2）（n>=2）。使用数组编写程序，顺序打印出 f（0）到 f（10）的值。

分析：可以定义一个整型数组，用于存放 f（0）到 f（10）这 11 个整数。先将 f（0）、f（1）、f（2）的值直接赋给数组的前三个元素，然后建立一个循环，每次取出数组中的三个元素（最后一个元素必须是上一次进行循环体时刚计算出来的），通过递推公式求出下一项的值并存入数组中，直到 11 项都被计算出来为止。

程序如下：

```
# include < stdio.h >
main（）
{
int f[11],k;

f[0]=0,f[1]=1,f[2]=2;                    / * 前三项赋初值 * /
for[k=3;k <11;k + +）
f[k]=f[k-1]+ 2 * f[k-2] * f[k-3];        / * 递推求解每一项的值 * /
for（k=0;k<11;k+ +）
printf（" % d,",f[k]）;
}
```

从上例中可以看出，使用数组的方便之处之一是通过下标存取数组元素正好同循环控制变量及其表达式相匹配，因此，数组+循环就可以简单地实现顺序地、逆序地或跳跃地对大量数据进行连续处理，这是仅使用基本数据类型无法办到的。

【例 4-2】请用户输入一个含有 12 个浮点数的一维数组，请分别计算出数组中所有的正数的和以及所有的负数的和。

分析：显然应该建立一个循环，从头到尾地将整个数组搜索一遍，同时将其中所有的正数和负数分别累加起来以求得结果。程序如下：

```
# include < stdio.h >
main（）
{float data[12];                         / * 存放浮点数的一维数组 * /
float result1=0.0,result2=0.0;           / * 将要用于分别存放正数和，负数和 * /
int i;
printf（"please input 12 float numbers: \ n"）;
for（i=0;i<12;i++）
scanf（" % f",& data[i]）;               / * 逐一输入数据 * /
for（i=0;i<12;i+ +）
{ if（data[i]>0.0）
result 1 + =data[i];                     / * 累加正数 * /
else
```

```
result2 + =data[i];                    / * 累加负数 * /
}
printf（" the sum of all the positive numbers is % .3f \ n",result1）；
printf（" the sum of all the positive numbers is % .3f \ n",result2）；
}
```

【例 4-3】使用直接插入法对 12 个整数进行排序（按从小到大的顺序排列）。

直接插入排序的算法描述如下：

比较待排序数组中前两个数的大小，按要求的顺序排列好；将第三个数加入到由前两个数组成的有序子序列中，保证此三个数的排列依然保持所要求的顺序；以此类推，当前 N 个数已按从小到大的顺序排好后，将第 N+1 个数依次同前面的各数相比较，直到找到一个合适的位置将其插入，使前 N+1 个数保持从小到大的顺序排列：重复上述过程，直到整个数组中的所有元素都被处理过为止。

程序如下：

```
# include < stdio.h >
main（）
{
int array[12],i,j;
printf（"input 12 integers: \ n"）；
for（i =0;i<12;i+ +）
scanf（" % d",& array[i]）；        / * 逐一输入数据 * /
for（i=0;i<12;i+ +）               / * 从第二个数开始，逐一将当前数据插入到此数之
                                   前的数据序列中，所以，当前数以前的数据序列总是有序 * /
{ int temp =array[i];
for（j =i-1;j > =0;j - -）         / * 逐一搜索当前数以前的有序数据序列 * /
{if（array[j]>temp）               / * 未达到插入点，则将原序列中的数据依次后移 * /
{if（array[j+1]=array[j];
else
{array[j+1]=temp;
break;
}
}
if（j= = -1）                      / * 前序列中的第一个数都比此数小，则插入到第一的位置 * /
array[0]=temp;
}
for（i=0;i<12;i+ +）
printf（"% d,",array[i]）；
}
```

【例 4-4】用数组来处理求 Fibonacci 数列问题。

程序如下：

```
main（）
{
int i;
static int f[20]={1,1};
for（i=2;i<20;i++）
f[i]=f[i-2]+f[i-1];
for（i=0;i<20;i++）
```

```
{
if（i%5= =0）printf（"\n"）;
printf（"%12d",f[i]）;
}
}
```

运行结果如下：

```
          1          1          2          3          5
          8         13         21         34         55
         89        144        233        377        610
        987       1597       2584       4181       6765
```

if **语句用来控制换行，每行输出** 5 **个数据。**

4.2 多维数组

数组名后有两对方括号的数组叫二维数组，有三对方括号的数组叫三维数组，方括号对数大于或等于二的数组统称多维数组。

4.2.1 多维数组的定义

多维数组定义的一般形式如下：

数据类型标识符 数组名［常量表达式 e1］［**常量表达式** e2］…；

多维数组定义的数组元素个数为：e1*e2…

同一维数组一样，多维数组每一维元素的下标也都从 0 **开始。例如：**

char c［2］［2］；

此二维数组共有四个元素，分别为：c［0］［0］，c［0］［1］，c［1］［0］，c［1］［1］。**这四个数组元素的类型均为字符型。又如：**

int n［2］［3］［2］；

此三维数组共有 12 **个元素，分别为：**

n［0］［0］［0］ n［0］［0］［1］ n［0］［1］［0］ n［0］［1］［1］ n［0］［2］［0］ n［0］［2］［1］

n［1］［0］［0］ n［1］［0］［1］ n［1］［1］［0］ n［1］［1］［1］ n［1］［2］［0］ n［1］［2］［1］

这些元素均为整型变量。一维数组定义需注意的问题对多维数组也适用，此处不再一一例举。

4.2.2 多维数组的存储形式

多维数组在内存中按下标顺序依次存储在内存的连续空间中。

对于二维数组，是按先行后列的顺序存放，如：char c［2］［2］；**先存放第一行，且顺序为** c［0］［0］，c［0］［1］，**然后再存放第二行，且顺序为** c［1］［0］，c［1］［1］，**数组** c［2］［2］**在内存中的连续存放顺序如图** 4-1a **所示。**

C **语言允许使用多维数组，下面介绍多维数组是如何存储的。**

以三维数组为例，如 int n［2］［3］［2］；**则先存放** n［0］［0］［0］，n［0］［0］［1］，**然后再存放** n［0］［1］［0］，n［0］［1］［1］，…，**最后存放** n［1］［2］［0］，n［1］［2］［1］，**如图** 4-1b **所示。**

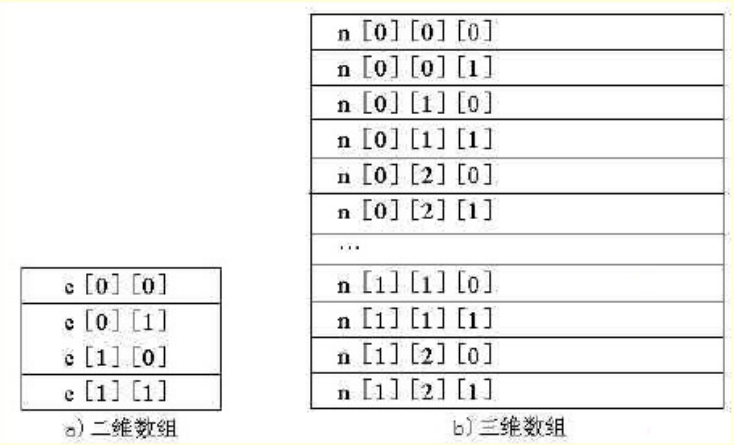


图 4-1　数组元素在内存中的连续存放示意图

4.2.3　多维数组的引用

不论是一维数组还是多维数组，都不能对其进行整体引用，只能对具体元素进行引用。引用格式与一维数组类似。

二维数组的引用形式：

数组名［e1］［e2］；

三维数组的引用形式：

数组名［e1］［e2］［e3］

其中 e1 e2 e3 **是值大于或等于** 0 **的整型表达式，这些表达式可包含变量。例如，对于数组：**

int n［10］［15］［12］；

以下的引用方式都是合法的：

n［0］［1］［2］，n［k+1］［0］［4］，n［3*k+2］［j+3］［11］

在数组引用中要特别注意下标越界。因为系统不检查下标越界问题，所以程序设计者就要特别注意。例如，对于数组：

int b［4］［5］；

则引用 b［4］［5］ **是错误的，因为该引用下标越界。**

4.2.4　多维数组的初始化

同一维数组一样，对于全局类型和静态类型的数组可在定义的同时进行初始化。初始化的方式有如下两种：

1．把初始值括在一个花括号内

例如，对二维数组 c［2］［2］**，可用如下方法初始化：**

int c［2］［2］= {'a'，'b'，'c'，'d'}

于是有：

c［0］［0］='a'，c［0］［1］='b',c［1］［1］='d'。

可以看出，实际上系统是按数组元素在内存中的存放顺序将初始化列表中的值依次赋给各数组元素的。从下面的例子中也可以得到同样的结论：

int n［2］［3］［2］= {0，1，2，3，4，5，6，7，8，9，10，11}；

于是有：

n［0］［0］［0］=0 n［0］［0］［1］=1

n［0］［1］［0］=2 n［0］［1］［1］=3

n［0］［2］［0］=4 n［0］［2］［1］=5

n［1］［0］［0］=6 n［1］［0］［1］=7

n［1］［1］［0］=8 n［1］［1］［1］=9

n［1］［2］［0］=10 n［1］［2］［1］=11

2．把多维数组分解成多个一维数组

二维数组又可看作一维数组，该一维数组的每一个元素又是一个一维数组。例如：

int a［2］［3］；

可把它看成是具有两个元素的一维数组 a［0]、a［1]。而 a［0]、a［1］又都具有三个元素的一维数组，即：

a［0］：a［0］［0］，a［0］［1］，a［0］［2］

a［1］：a［1］［0］，a［1］［1］，a［1］［2］

因此，上例对二维数组的初始化又可分解成对多个一维数组的初始化：

char c［2］［2］={ {'a'，'b'} ， {'c'，'d'} }

其效果与不分解完全一样。又如对三维数组：

int n［2］［3］［2］；

也可按上述方法逐步分解，可分解成 2 个二维数组：

n［0]

n［1]

它们各有 3×2＝6 个元素。每个二维数组又可分解为 3 个一维数组，所以共分界出 6 个一维数组：

n［0］［0]

n［0］［1]

n［0］［2]

n［1］［0]

n［1］［1]

n［1］［2]

它们又各包含 2 个元素。于是这个三维数组可按如下方式初始化：

```
int n［2］［3］［2］= {
                        {0，1，2，3，4，5} {6，7，8，9，10，11}
                    } ；
```

也可这样来初始化：

```
int y［2］［3］［4］= {
                        { {0，1} {2，3} {4，5} }
                        { {6，7} {8，9} {10，11} }
                    } ；
```

这种逐步分解、降低维数的方法，有助于理解数组的存储形式和初始化。

4.2.5 多维数组应用举例

【例 4-5】用户输入一个 4×4 的整数矩阵，编写程序求其对角线上元素的和并提示用户逐行输入矩阵的元素。

分析：矩阵可以使用二维数组来存放。其程序如下：

```
#include < stdio.h >
main（）
{int matrix[4] [4];
int k,j,result 1=0,result2=0;
for（k=0;k<4;k+ +）
{printf（"one line:"）;
for（j=0;j<4;j+ +）
{scanf（" % d",& matrix[k][j]）;
if（k = =j）                    / * 累加对角线上的元素 * /
result1+ =matrix[k][j];
```

```
if（k +j= =3）                    / *  累加反对角线上的元素  * /
result2 + =matrix[k][j];
}
}
printf（"the result:% d,% d \ n",result1,result2）;
}
```

【例 4-6】有六个人，来自六个不同的国家日、美、英、法、德和西班牙，设此六个人分别为 1，2，3，4，5，6，已知条件有：

（1）1 不是来自日、英、法、西班牙。

（2）2 不是来自日、英、法。

（3）3 不是来自日、英、德、西班牙。

（4）4 不是来自日、法、西班牙。

（5）5 不是来自日、法、西班牙。

（6）6 不是来自法国。

请利用二阶矩阵编写程序求解，这六个人各来自哪个国家。

分析：此题有一种比较特殊的解决方案：建立一个二维矩阵，每一行表示一个人，每一列表示一个国家，对于已经明确的关于某人不属于某一国家的条件，将其对应的行列交点的矩阵元素的值赋为 0，其余的矩阵元素的值赋为 1，然后使用循环一行一行地检查，若发现某一行只有五个非零的元素，则确定此行对应的人属于此列对应的国家，立即在另一个数组中记录下此信息，再将本列其他所有的数组元素的值全部赋为零（因为其他的人不可能属于此国家了）。重新开始检查每一行，重复上述过程，直到所有的人所在的国家都已经确定为止。

其程序如下：

```
#include < stdio.h >
main（）
{     static int judge[6][6],sign[6];
int sing1=0;              / *sing1 被用于记录已经明确了谁是其的国民的国家的个数* /
char nation[6][10]={{"Jap"},{"USA"},{"Britain"},{"France"},
{"Germany"},{"Spain"}};
for（int i=0;i<6;i+ +）
{ judge[i][1]=1
{ judge[3][i]=1
{ judge[i][4]=1
{ judge[5][i]=1
}
judge[2][4]=0;judge[5][3]=0;
judge[4][2]=1;judge[1][5]=1;judge[5][5]=1;
/ *  已明确某人不是某国人时，其对应节点的值被赋为 0，否则，其节点的值暂时都被赋为 1 * /
while（sign1<6）              / *  当六个国家尚未被处理完时的继续  * /
{ int sign2,record;          / * sign2 用于标志某一列的非零元素的个数  * /
for（i=0;i<6;i+ +）
{ if（sign[i]!=0）             / *  若某列已被处理过了则此列对应的国家已经明确了谁
是他的国民  * /
continue;
sign2=0;
for（int j=0;j<6;j+ +）
if（judge[j][i]= =1）
{ sign2 + +;record=j;}
```

```
if（sign2 = =1）              / * 找到某列的非零元素的个数为 1，则确定此非零元素对
应的人是此列对应的国家的公民 * /
{ sign[i]=record;              / * 标志此列已经被处理过了，并记录下国家和人的关系 * /
sign1 + +;                      / * 已经处理过的国家个数加 1 * /
for（int k=0;k<6;k+ +）
if（k! =i）
judge[record][k] =0;           / * 明确此人不可能是其他国家的人了，立即修改 judge
数组 * /
}
}
}
for（i=0;i<6;i+ +）
printf（" \ nperson % d comes from % s",sign[i] +1,naiton[i]）;
return（1）;
}
```

【例 4-7】已知一个包括年、月、日的日期，编程求该日是该年的第多少天。

分析：首先应判断所输入的年份是否为闰年，利用数组，将闰年和非闰年每月的天数各自列成一个表，于是，求某些月份的总天数的问题就转换成了对数组元素的累加。由于月份是 1 开始的，数组的下标是从 0 开始的，因此，在 month_tab 数组中就增加了 0 月份，它有 0 天。定义的数组的意义如下：

```
month_tab［］［］：存放闰年与不闰年各月的天数
month_tab［0］：存放不闰年时各月的天数
month_tab［1］：存放闰年时各月的天数
```

其程序如下：

```
# include < stdio.h >
main（）
{
static int month_tab[2][13]=
{
{0,31,28,31,30,31,30,31,31,30,31,30,31},      / * 利用数组，将非闰年的每月天数列
成一个表 * /
{0,31,29,31,30,31,30,31,31,30,31,30,31}       / * 闰年的每月天数对应的表 * /
}
int year,month,day;
int yearday,leap;
printf（"Enter year-month-day:"）;
sacnf（"% d % d % d",& year,& month,& day）;    / * 输入年、月、日 * /
leap=（（year % 4 = =0）&&（year % 100! =0）||（year % 400 = =0））;
/ * 判定是否为闰年 * /
yearday=day;
for（i=1;i<month;i + +）
yearday + =month_tab[leap][i];            / * 计算是该年的第几天 /*
printf（"yearday = % d",yearday）;
}
```

4.3 字符型数组与字符串

4.3.1 字符型数组的概念

字符型数组是数据类型为 char 的数组。因此，在前面两节中介绍的数组的定义、存储形式和引用等也都适用于字符型数组。字符型数组用于存放字符或字符串，每一个数组元素存放一个字符，它在内存中占用一个字节。例如：

```
char e［18］；
…
c［0］='T'，    c［1］='h'，    c［2］='i'，    c［3］='s'，    c［4］=' '，
c［5］='i'，    c［6］='s'，    c［7］=' '，    c［8］='a'     c［9］=' '，
c［10］='s'，   c［11］='t'，   c［12］='r'，   c［13］='i'，
c［14］='n'，   c［15］='g'，   c［16］='!'，   c［17］='\0'；
```

即该字符数组中存放如下 17 个字符：

```
This is a string!
```

常用字符数组存放字符串，由于 C 语言中所有的处理都约定字符串以'\0'结尾。因此，在用字符数组存放字符串时，也要在字符串的末尾用'\0'结束。于是，在定义用来存放字符串的数组时，其数组元素个数要比字符串中字符个数多 1，以便保留字符'\0'。

4.3.2 字符型数组的初始化

关于字符数组的初始化有如下两种方式：

1．用字符常量初始化数组

例如：

```
char str［20］= {'T'，'s'，'i'，'n'，'g'，'h'，'u'，'a'
                ''，'U'，'n'，'i'，'v'，'e'，'r'
               's'，'i'，'t'，'y'，'\0'}
```

即该数组被初始化为“Tsinghua University”。

2．用字符串常量初始化数组

例如：

```
char str［18］="This is a program"；
```

该初始化自动在末尾一个字符后加'\0'。

以上两种初始化方式效果完全一样，但后者要简单的多。在初始化中，如果提供的字符个数多于数组元素的个数，则作为语法错误处理。如果字符个数小于元素个数，则多余的数组元素自动赋空格字符。在用字符常量初始化时，如果字符末尾没有'\0'字符，则该字符数组不能作为字符串处理，只能作为字符逐个处理。初始化时是否加'\0'就要看是否作字符串处理。

对二维字符数组可以看作一维字符串数组。例如：

```
char string［3］［5］；
```

可看作包含三个字符串的数组。于是可以这样来初始化它：

```
char string［3］［6］= {"WANG"
                       "ZHANG"
                       "SHEN"
                      }；
```

其存储形式如图 4-2 所示。

W	A	N	G	/0	
Z	H	A	N	G	/0
S	H	E	N	/0	

图 4-2　用二维数组存放字符串

4.3.3　字符型数组的输入/输出

在标准库函数中，系统提供了大量有关字符和字符串的操作函数，其中包括输入/输出函数，这里简单说明一下如何利用它们对字符型数组进行输入/输出。我们可以用 getchar（）**或** scanf（）**两函数对字符数组进行赋值，而用** putchar（）**或** printf（）**输出其内容。下面举几个例子来说明。**

1．用 getchar（）或 scanf（）的' % c'格式符对数组进行字符赋值

例如，对于数组 char c［15］：

用 getchar（）**赋值：**

```
for（i=0；i<14；i++
    e［i］=getchar（）；
```

也可用 scanf（）**赋值：**

```
for（i=0；i<14；i++
    scanf（" % c"，& c［i］）
```

用 getchar（）**赋值，一次只接收一个字符，用** scanf（）**赋值，一次可接收一个字符串。**

2．用 scanf（）的' % s'格式赋值（即字符串赋值）

```
char c［15］；
scanf（" % s"，c）；
```

或

```
scanf（" % s"& c［0］）；
```

当在键盘上输入"program"**并回车时，**c **数组中自动包含一个以**' \ 0 '**结尾的字符串**"program"**。**" % s "**格式要求操作数是地址，所以上述两种写法均正确。但下面的写法是不正确的：**

```
scanf（" % s"& c）
```

3．用字符输出函数输出字符数组中的内容

例如，假设下面的 c **数组各元素已赋值，**

```
char c［15］；
…
for（i=0；i<15；i+ +）
putchar（c［i］）；
```

或

```
for（i=0；i<15；i+ +）
printf（" % s"，c）；
```

4．用 printf（）的"%s"进行字符串输出

此时要求字符数组一定要以'\ 0'**结尾。例如：**

```
char c[15]={"program"};
printf（"%s",c）;
```

此时将输出字符串：

```
program
```

值得提出的是 putchar（）**一次只能输出一个字符。**

4.3.4　字符型数组的应用举例

【例 4-8**】指出下面数组引用哪些是错误的，为什么？**

```
char str［8］，c；
（1）str［1］='3'；
（2）scanf（" % s"，str）；
（3）str="program"；
（4）str=getchar（）；
（5）c=str+'3'；
（6）str［8］='a'；
```

答案：（3）、（4）和（5）引用是错误的，因为不能对数组整体进行操作，只能对数组元素进行操作。（6）也是错误的，因为数组 str 的下标从 0 开始，最大下标为 7，所以 str［8］越界。

【例 4-9】编写不用 strcat 函数，而将两个字符串中的字符拼接起来的程序。

分析：此程序先顺序搜索第一个字符串，找到结尾时停止，然后将第二个字符串中的字符从此位置开始逐一拷贝入第一个字符串直到第二个字符串结束。其程序如下：

```
#include < stdio.h >
main（）
{ char str0[41],strt[21];
int j,k;
printf（"please input firt $（1-20 characters）: \ n "）;
scanf（" % s",str0）;
printf（" please input the second one（1-20 characters）: \ n"）;
scanf（ "% s",strt）;
for（j=0;str0[j]! =' \ 0';j+ +）;      / * 搜索到第一个字符串的结尾处 * /
for（k=0;str0[k]! =' \ 0';k+ +）;     / * 将第二个字符串中的字符逐一拷贝到第一个字符串的后面 * /
}
str0[j]=strt[k];
j+ +;
}
str0[j]=' \ 0';                        / * 加字符串结尾标志 * /
printf（" % s",str0）;
}
```

【例 4-10】输入一个含若干字符的字符串，分别找其中的字母和数字，其余的字符一概忽略。将字母序列和数字序列分别输出。

分析：本题可使用的方法较多，其中一种是定义两个字符型数组，分别存放字母和数字，在使用循环接受输入时即对每一个字符作判断，若是数字放入其中一个数组，若是字母放入另一个数组，否则，直接丢弃而不作任何处理。得到分别存有输入的字母和数字的数组后，输出即可。其程序如下：

```
#include < stdio.h >
main（）
{    char str1[100],str2[100];      / * str1 用于存放输入的数字，str2 用于存放输入的
字母 * /
char c;
int I=0,j=0;
printf（" \ nplease input a $（0-99）: \ n"0）;
while（（c=getchar（））!=' \ n'& & i<99& & j<99）
{    if（c> =48 & &c < =57）
str1[i + +]+c;                / * 输入的数字，则存放入 str1 中 * /
else if（c> =65& &c <=90||c> =97& &c< =122）
str2[j+ +]=c;                 / * 输入的是字母，则存放入 str2 中 * /
```

```
else;                          / * 既不是数字，又不是字母，忽略 * /
}
str1[i]= ' \ 0 ';              / * 加字符串结束标志 * /
str2[j]=' \ 0 ';
printf（"the number array: % s \n",str1）;      / * 打印结果 * /
printf（"the letter array: % s \ n",str2）;
}
```

4.4 综合应用举例

【例 4-11】下面是对数组进行初始化的语句，指出哪些是错误的，错在哪里？

（1）int m［］= {1，2，3，4，5} ；

（2）char str［2］［］= {"boys"，"girls"}；

（3）int n［］［］= { {1，2，3} ， {5，6，7} } ；

（4）char str［20］= {"I am a student"}

答案：（2）和（3）初始化是错误的，因为数组的第二个下标不能缺省。

【例 4-12】下面是一段关于二维数组的程序，指出其中的错误并改正。

```
#include < stdio.h >
main（）
{   int a[3],[]={{1,2,4},{},{5,7}};
int b[3],[3];
int sum 1=0,sum2=0;
for（int j=1;j< =3;j+ +）
{   for（int k=1;k< =3;k+ +）
scanf（" % d",b[j][k];}
for（k=1;k< =3;k+ +）
{   sum1+ =a[k][k];
sum2+ =b[k][k];
}
printf（" % d,% d",sum1sum2）;
}
```

答案：错误①改 int a［3］，［］= {{1，2，4}，{}，{5，7}}，**为** int a［］［3］= {{1，2，4}，{}，{5，7}}；**第二维的大小是不能省略的，而两下标之间的逗号纯属画蛇添足。错误②：改** int b［3］，［3］；**为** int b［3］［3］；**错误③：改** for（int j=1；j+ +）for（int k=1，k<=3；k+ +）**和** for（k=1；k<=3；k+ +）**为** for（int j=0；j<3；j+ +）for（int k=0；k<3；k+ +）**和** for（k=0；k<3；k+ +）。**不管是一维数组还是多维数组，下标始终应该从零开始。**

【例 4-13】匹配搜索程序的编写。先根据用户的输入建立下一个字符串，再提示用户输入另一个字符串，在原串中搜索，是否存在与此相同的子串。若存在，给出此子串的起始位置，否则，给出没有找到的信息。

分析：匹配算法中最直观的一种（当然也是效率最低的一种）是先将待匹配的串与原串的开头对齐，逐一检查两串的对应字符是否相等，一旦发现不等则此次匹配失败，将待匹配的串与原串的第二个字符对齐，依此类推，直到找到完全匹配的子串或者原串已经结束。显然，此过程可以使用一个二重循环来实现。

```
# include < stdio.h >
# include <string.h >
main（）
{   char strsource[50],straim[50];
int sign;
```

```
printf（" \ ninput the source str: \ n"）        / * 输入原串。* /
scanf（" % s",straim）;
for（int i=0;i< =（strlen（strsource）-strlen（straim））;i+ +）
{     int point=i;
sign=1;
for（int j=0;j< strlen（straim）;j+ +,point+ +）        / * 向后匹配。* /
{ if（strsource[point]! =straim[j]）                    / * 若匹配失败。* /
{    sign=0;
break;
}
}
if（sign= =1）              / * 匹配成功。* /
{ printf（"find! the location is % d \ n",i+1）;
break;
}
}
if（! sign）          / * 找不到匹配字符串。* /
printf（"No find! \ n"）;
}
```

【例 4-14】输出两个字符串 str1 和 str2，要求各串中无重复的字符，求两者的交集。若交集的每一字符，在 str2 中搜索，若找到相同字符，则认为此字符在 str1 和 str2 的交集中而将其记录下来，否则将它丢弃。其程序如下：

```
# include < stdio >
main（）
{ char str[10],str2[10];          / * 存放输入的两个字符串。* /
char str3[10];                    / * 存放输入的两个字符串的交集。* /
int    count1,count2;
int    count3;
count1=0;
count2=0;
count3=0;
scanf（"% s",str1）;
scanf（"% s",str2）;
while（str2[count1]! =' \ 0'）      / * 逐一考查 str1 中的每一个字符。* /
{ count2=0;while（str2[count2]! = ' \ 0'）      / * 在 str2 中搜索相同的字符。* /
{    if（str1[count1]= =str2[count2]）          / * 若找到相同字符。* /
{ str3[count3]=str1[count1]:            / * 将当前字符存入交集数组中。* /
count3+ +;
break;
}
count2+ +;
}
count1+ +;
}
str3[count3]=' \ 0';
if（str3[0]=' \ 0'）
```

```
printf（" % s \ n",str3）；
}
```

【例 4-15】输入一行字符，统计其中有多少个单词，单词之间用空格分隔开。

程序如下：

```
#include <stdio.h>
main（）
{
char string[81];
int i,num=0,word=0;
char c;
gets（string）；
for（i=0;（c=string[i]）!='\0';i++）
if（c= =' '）word=0;
else if（word= =0）
{
word=1;
num++;
}
printf（"There are %d words in the line\n",num）；
}
```

运行情况如下：

```
I am a boy.↙
There are 4 words in the line
```

程序中，变量 i **作为循环变量，**num **用来统计单词个数，**word **作为判别是否单词的标志，若** word=0 **表示未出现单词，如出现单词** word **就置成** 1**。算法见图** 4-3 **所示。**

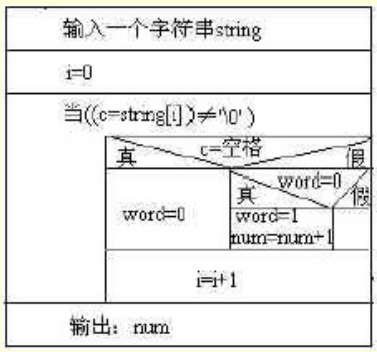


图 4-3 **【例** 4-15**】算法**

解题的思路是这样的：

单词的数目可以由空格出现的次数决定（连续的若干个空格作为出现一次空格；一行开头的空格不在内）。如果测出某一个字符为非空格，而它的前面的字符是空格，则表示“新的单词开始了”此时使 num**（单词数）累加。如果当前字符为非空格而其前面的字符也是非空格，则意味着仍然是原来那个单词的继续，**num **不应累加** 1**。前面一个字符是否空格可以从** word **的值看出来，若** word=0**；则表示前一个字符是空格，如果** word=1,**意味着前一个字符为非空格。**

程序中 for **语句中的“循环条件”为**

```
（c=string[i]!='\0'）
```

它的作用是先将字符数组的某一元素（一个字符）赋给字符变量 C**。此时赋值表达式的值就是该字符，然**

后再判定它是否结束符。这个“循环条件”包含了一个赋值操作和一个关系运算。可以看到用 for 循环可以使程序简练。

4.5 小 结

（1）数组是具有相同数据类型且按一定次序排列的一组变量的集合体，数组必须先定义后使用。

（2）数组被存入连续相邻的存储单元中，数组名是存储区域的起始地址。一维数组按下标的次序存放，二维数组是按先行后列的次序存放，对多维数组的存储次序是按右边的下标先变化，左边的下标后变化的规律来存放。

（3）数组只能对其元素进行操作，不能对数组整体进行操作。数组元素的表示方法是：

数组名［下标］，

数组名［下标］［下标］…

其中下标指明该数组元素在数组中的位置。下标从 0 开始。

（4）数组初始化有用赋值语句初始化和在定义时初始化两种方法。对于全局类型和静态类型的数组才可在定义时初始化。

（5）在使用数组时，要防止下标越界。

习 题

1．用选择法对 10 个整数进行排序。

2．打印出以下的杨辉三角形（要求打印出 10 行）。

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10   10   5   1
……
```

3．有一篇文章，共有 3 行文字，每行有 80 个字符。编写程序统计出其中英文大写字母、小写字母、数字、空格以及其他字符的个数。

4．编写程序，不使用 strcmp 函数，比较两个字符串的大小。

5．编写程序，求两个 3×2 的矩阵的和的程序。

6．编写程序，打印出以下图案。

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

7．编写一程序，输入八个职工的信息（包括姓名、职工号、工资），使用三个数组存放起来，然后输入一个职工的姓名，输出他的职工号和工资。

8．有一行电文，已按下面规律译成密码：

A→Z　　a→z

B→Y　　b→y

C→Z　　c→x

… …

即第 1 个字母变成第 26 个字母，第 i 个字母变成第（26-i+1）个字母。非字母字符不变。要求编程序将密码译回原文，并打印出密码和原文。

第五章 指 针

指针是 C 语言中一个重要的概念，也是 C 语言的一个重要特色。正确而灵活地运用指针，可以直接对内存中的数据进行快速处理、实现函数间的通信等。

虽然指针具有很强的功能，而且用起来很方便，但对初学者，使用不当也会产生严重的错误。因此，希望读者要多思考、多实践，逐步掌握指针的有关内容。

5.1 指针的基本概念

5.1.1 什么是指针

为了说清楚什么是指针，必须弄清楚数据在内存中是如何存储的，又是如何读取的。

如果在程序中定义了一个变量，在编译时就给这个变量分配内存单元。系统根据程序中定义的变量类型，分配一定长度的空间。例如，许多微机的 C 系统对整型变量分配 2 个字节，对实型变量分配 4 个字节，字符型变量为 1 个字节，……。内存区的每一个字节有一个编号，这就是“地址”，它相当于旅馆中的房间号。在地址所标志的单元存放数据，这相当于旅馆中各个房间中住旅客一样。

读者一定要弄清楚一个内存单元的地址与内存单元的内容这两个概念的区别，如图 5-1 所示。

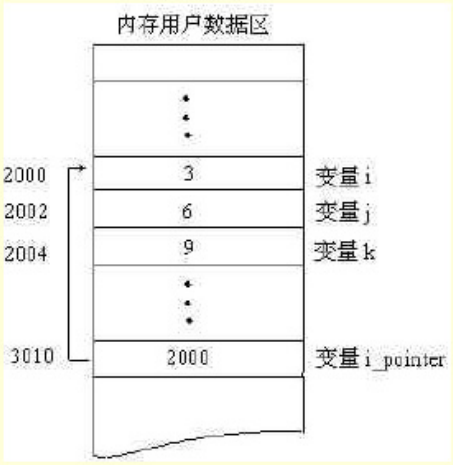


图 5-1 内存数据的“直接访问”模式

假设程序已定义了三个整型变量 i、j、k，编译时系统分配 2000 和 2001 两个字节给变量 i，2002、2003 字节给 j，2004、2005 给 k。在内存中已没有 i、j、k 这些变量名了，对变量值的存取都是通过地址进行的。例如，printf（"%d"，i）的执行是这样的：根据变量名与地址的对应关系，找变量 i 的地址 2000，然后从由 2000 开始的两个字节中取出数据（即变量的值 3），把它输出。输入时如果用 scanf（"%d"，&i），在执行时，就把从键盘输入的值送到地址为 2000 开始整型存储单元中。如果有 k=i+j，则从 2000、2001 字节取出 i 的值（3），再从 2002、2003 字节取出 j 的值（6），将它们相加后再将其和（9）送到 k 所占用的 2004、2005 字节单元中。这种按变量地址存取变量值的方式称为“直接访问”方式。

还可以采用另一种称之为“间接访问”的方式，将变量 i 的地址存放在另一个内存单元中。C 语言规定可以在程序中定义整型变量、实型变量、字符变量等，也可以定义这样一种特殊的变量，它是存放地址的。假设我们定义了变量 i_pointer 是存放整型变量的地址的，它被分配为 3010、3011 字节。可以通过下面语句将 i 的地址

存放到 i_pointer 中。

```
i_pointer=&i；
```

这时，i_pointer 的值就是 2000，即变量 i 所占用单元的起始地址。要存取变量 i 的值，也可以采用间接方式：先找到存放“ i 的地址 ”的单元地址（3010、3011），从中取出 i 的地址（2000），然后到 2000、2001 字节取出 i 的值（3）。

打个比方，为了开一个 A 抽屉，有两种办法，一种将 A 钥匙带在身上，需要时直接找出该钥匙打开抽屉，取出所需要的东西。另一种办法是：为安全起见，将该钥匙（A）放到另一抽屉 B 中锁起来。如果需要打开 A 抽屉，就需要先找出 B 钥匙，打开 B 抽屉，取出 A 钥匙，再打开 A 抽屉，取出 A 抽屉中之物，这就是“ 间接访问 ”。

图 5-2 表示直接访问和间接访问的示意图。

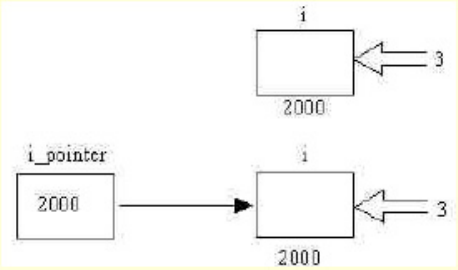


图 5-2　内存数据的“ 间接访问 ”模式

为了表示将数值 3 送到变量中，可以有两种表达方法：

（1）将 3 送到变量 i 所占的单元中。见图 5-2 上。

（2）将 3 送到变量 i_pointer 所“ 指向 ”的单元中。见图 5-2 下。

所谓“ 指向 ”就是通过地址来体现的。i_pointer 中的值为 2000，它是 i 的地址，这样就在 i_pointer 和 i 之间建立起一种联系，即通过 i_pointer 能知道 i 的地址从而找到变量 i 的内存单元。图 5-2 中以箭头表示这种“ 指向 ”关系。

由于通过地址能找到所需的变量单元，我们可以说，地址“ 指向 ”该变量单元（如同说，房间号“ 指向 ”某一房间一样）。一个变量的地址称为该变量的“ 指针 ”。例如，地址 2000 是变量 i 的指针。如果有一个变量专门用来存放另一变量的地址（即指针）的，则它称为“ 指针变量 ”。上述的 i_pointer 就是一个指针变量。指针变量的值（即指针变量中存放的值）是指针（地址）。请区分“ 指针 ”和“ 指针变量 ”这两个概念。例如，可以说变量 i 的指针是 2000，而不能说 i 的指针变量是 2000。

5.1.2　指针的目标变量

若把某一地址量赋给指针，该指针就指向了由该地址开始的一片内存空间。于是就可以对该区域中存放的数据进行操作。该区域中所存放的数据称为该指针的目标。如果指针所指向的区域是五个变量的存储空间，则这个变量称为指针的目标变量。即指针所指向的变量就是目标变量。指针除了可以指向一般变量外，还可以指向数组变量、结构变量、联合变量和函数目标变量。指针的目标用指针名前加星号（*）为表示。例如，假设 point 是指针：

```
int m，n；
point=& m；
n=m；
n=* point；
```

这里 point 是五个指针变量，*point 是指针目标，x 是目标变量。*是指针运算符，它的操作对象必须是地址量，其功能是访问地址内所存放的内容。语句 n=m；和 n=* point；是等效的。

指针的概念类似于汇编语言中的间接寻址。在间接寻址中，一个存储单元中存放着另一个存放单元的地址。前者就类似于指针。

5.1.3 指针运算符

“ * ”运算符称为指针运算符，“ & ”是地址运算符，二者是互逆运算。例如，

```
int n；
point = & n；
```

则&（* point）等效于&n，其结果为 point 的内容，即* point 的地址就是 n 的地址。*（& n）等效于 n，即访问变量 n 的地址，其结果就是 n。

在进行指针运算时，读者应理解清楚 piont＝& n 与* point=n 的区别。前者是把目标变量 n 的地址赋给指针变量，从而使 point 指向 n，这时* point 和 n 占据相同的存储区。所以，读者应区分 point、* point 和& point 的区别：

- point：是指针变量，其内容是地址量。
- * point：是指针变量的目标变量，其内容是数据。
- & point：是指针变量本身所占据的存储地址。

5.2 指针的定义与初始化

指针既然是一种变量，那么就应当同其他变量一样，在引用之前必须先定义，在定义的同时也可初始化。

5.2.1 指针的定义

指针定义的一般形式为：

```
类型标识符          *指针名［，*指针名，…］；
```

例如：

```
int * pn；
char * pch，* pname；
float * pf；
```

类型标识符指出指针目标的数据类型，而不是指针变量本身的数据类型。因为指针本身的数据总是地址量。若指针指向变量，则数据类型就是目标变量的数据类型。通常把目标变量的数据类型称为指针的数据类型。例如：

```
char * pch1，* pch2；
```

说明指针 pc1 和 pc2 都是指向 char 类型的数据。

指针名的命名规则同用户定义标识符的命名规则。在同一行中可以定义多个同类型的指针，指针名之间用逗号分开。在定义指针时，应注意在指针名前加“ * ”号。

5.2.2 指针的初始化

在指针定义的同时，赋给它初始值，这叫指针的初始化。初始化的一般形式为：

```
类型标识符 * 指针名=初始地址值；
```

例如：

```
int m，n［8］；
char c；
int * pm=& m；
int * pn=n；        /* 数组 a 的首地址赋给指针变量* /
char * pc=& c；
```

对于指针的初始化，应注意以下几点问题：

（1）这里的初始化是对指针变量的初始化，而不是对指针目标变量的初始化。例如上面的三个指针初始化例子中，是把目标变量的地址& m、n 和& c 分别赋给了指针变量 pm、pn 和 pc，而不是赋给指针的目标变量 * pm、* pn 和 pc。

（2）当把一个变量的地址作为初始值赋给指针变量时，该变量必须在这个指针初始化之前定义过。即不能把一个没有定义的变量的地址赋给指针。因为没有定义过的变量，其地址没有定义。

（3）指针目标变量的数据类型必须与指针的数据类型相一致。下例中的初始化是错误的，因为类型不一致。

```
double x；
int * px=& x；
```

（4）可以把一个指针的值赋给另一指针。下例是正确的初始化。

```
int n；
int * pn=&n；
int * qn=pn；
```

（5）在指针初始化时，不能把一般的整数数据（非地址量）赋给指针，如果这样做的话，就会把该数值作为内存地址，对这种地址进行读写将会造成可怕的后果。

（6）可以把一个指针初始化为下一个空指针。下例中赋值 0 值的指针不指向任何对象。

```
int * pn=0；
```

（7）不能用 auto 类型变量的地址去初始化一个 static 或全局变量类型的指针，因为 auto 类型变量的存储分配是动态的，控制每进入其有效作用域时就分配，退出时就要释放。因此，auto 类型变量的地址是不固定的。而 static 类型指针的内容则长期保持的。有关 auto、static 存储类型将在函数一章描述。例如：

```
float global；
main（）
{   static float sta；
float local；
static float * p1=& local；
static float * p2=& sta；
static float * p3=& global；
float * p4 = & local；
…
}
```

在上例中 p1 指针的初始化是错误的，而其他三个指针的初始化是合法的。因 local 变量是 auto 存储类型的变量。下面是指针的各种基本用法及其说明，请读者从中弄清指针的有关概念。

```
int x，y，z；
int * px=&x；                    / * 使指针 px 指向 x * /
y= * px * 2                      / * 将 x * 2 赋给 y * /
printf（" % d"，* px）；          / * 打印出 x 的当前值 * /
z=sqrt（（double）* px）          / * 将 x 转换成 double 型，然后求平方根，运
                                 算结果赋给整型变量 z * /
y=*（px+1）；                    / * px 指针加 1 后，将新地址中的内容赋给 y * /
% px=0；                         / * 即 x=0 * /
* px + =1                        / * 即 x=x+1 或（* px）+ + * /
（* px）+ +                      / * 即 x+ + * /
* px + +                         / * 即*（px+ +）* /
```

5.3 指针的运算

由于指针所包含的内容是地址量，所以指针的运算实际上是地址的运算。C 语言具有一套地址运算方法，这是一套适合于指针和数组等地址计算的规则化方法。指针变量在含义和种类上都和一般数据的运算不同。在 C 语言中，指针只有如下三种运算：

（1）算术运算。
（2）关系运算。
（3）赋值运算。

5.3.1 指针的算术运算

指针的算术运算有三种，即：指针与整数的加、减运算、指针的加 1 和减 1 运算、和指针相减运算。

（1）指针与整数的加、减运算

指针 point 加上或减去一个整数 n，将相对于当前位置前移或后移 n 个存储单元。一个存储单元的长度（即占的字节数）取决于指针的类型，因此，p+n 所表示的实际内存地址（以字节编号）是：

（point）+n * sizeof（指针的数据类型）（单位：字节）

例如：

```
int n，* pn；
```

则对于 pn+5 运算后的结果是：

（pn）+5 * sizeof（int）=（pn）+5 * 2=（pn）+ 10（字节）

又如：

```
float y，* py；
```

则对于 py-5 运算后的结果是：

（py）-5 * sizeof（long）=（py）-5 * 4（py）-20（字节）

（2）指针加 1 和减 1 运算

这是指针与整数的加减运算的特例，指针加 1 运算后就指向内存中下一个数据位置，而减 1 以后就指向前一个数据位置。下面都是指针加 1 或减 1 运算的例子：

```
int * pn，n，m；
pn=& n；
m=* pn+ +                    / * 即 m=*（pn+ +）* /
m=* + + pn；                 / * 即 m=*（+ + pn）* /
m=（* pn）+ +                / * 先给 m 赋值，后使指针指向的变量的值加 1 * /
m=+ +（* pn）；              / * 将指针指向的变量的值加 1 后赋给 m      * /
```

（3）指针相减运算

对于同一类型的指针变量 p1 和 p2，p2-p1 的意义是两个指针间数据的个数，而不是两指针间的字节数。从下式就可以看出这一点：

p2-p1=（（p2）-（p1））/sizeof（指针目标变量的数据类型）

例如，若 p2=2048，p1=1024，p2 和 p1 的目标变量的数据类型为 float，则 p2 和 p1 之间有 256 个 float 型的数据，其计算方法如下：

```
p2-p1=（（p2）-（p1））/sizeof（float）
      =（2048-1024）/4
      =256
```

【例 5-1】使用指针将输入的两个整数交换。

其程序如下：

```
# include < stdio.h >
main（）
{int m， * pm， *pn，temp；
pm=& m；
pn=& n；
scanf（"% d % d"，pm，pn）；
temp=* pm；
* pm=* pn；
```

```
* pn=* temp；
printf（"% d % d \ n"，m，n）；
printf（"% d % d \ n"，* pm，* pn）；
return 1；
}
```

请读者思考一下，倘若程序象下面这样写，将会打印出怎样的结果，m，n 两变量的值交换过来了吗？

```
int * temp；
temp=pm；
pm=pn；
pn=pn；
pn=temp；
printf（"% d，% d"，* pm，* pn）；
```

5.3.2 指针的关系运算

两个指向同一组数据类型相同的数据的指针之间可以进行如表 5-1 所示的各种关系运算。两个指针之间的关系运算表示它们的目标变量的地址位置之间的关系。

表 5-1 指针的关系运算

关系运算符	例 子	含 义
<	Px < py	Px < py 成立，则表示 px 的目标变量在 py 目标变量之前
= =	Px = = py	若关系成立，则表示二指针的目标变量的位置相同
< =	Px < = py	若关系成立，则表示 px 的目标变量的位置与 py 目标位置相同，或在它之前
>	Px > = py	若关系成立，则表示 px 的目标变量在 py 目标变量之后
>=	Px > = py	若关系成立，则表示 px 目标变量的位置与 py 目标变量的位置相同，或在它之前
!=	Px ! = py	若关系成立，则说明两个指针变量的位置不同

在进行指针关系运算时需要以下几点：

- **两个不同数据类型的指针之间的关系运算是无意义的。**
- **指针与一般整数之间的关系运算也是没有意义的。**
- **不可用指向不同数组的两个指针作关系运算。**
- **指针可以和 0 进行“ = = ”或“ ! = ”比较，用以判断是否为空指针。**

5.3.3 指针的赋值运算

可以对指针进行赋值运算，但所赋的值一定是地址常量，而不是一般整数。常见的赋值形式有以下几种：

（1）可以把一个变量的地址赋给与其具有相同数据类型的指针。例如：

```
int n，* pn；
px = & n；
```

（2）具有相同数据类型的两个指针可以相互赋值。例如，

```
float * px，* py，y；
px= & y；
py=px；
```

（3）可以把一个数组的地址赋给与其相同数据类型的指针。例如：

```
double a［30］，* pa. * pb；
pa = a；
pb = & a［0］；
```

（4）其他常用的赋值运算

设 px 和 py 是具有相同数据类型的两个指针，则以下这些赋值运算是合法的。

```
px=py+n；
```

```
px=py-n；
px + =n；
px - =n；
```

5.4 指针与数组

大家知道，数组的名字是数组的首地址，而且数组的元素在内存中占用连续的一片空间，因此，在 C 语言中可以用指针访问数组中的诸元素，根据前面所讲述的指针概念，可定义一个指针指向数组。例如：

```
float x［10］，* px；
```

可以通过如下的赋值语句使该指针指向数组 x；

```
px = x；或 px =& x［0］；
```

下面通过几个例子来进一步说明指针和数组的关系。

【例 5-2】使用指针对数组元素进行输入输出。

程序如下：

```
# include < stdio.h >
main（）
{float data[5]，* pt =data；
int k；
printf（"input datas: \ n"）；
for（k=0；k<5；k+ +）
scan（"% f"，pf+k）；
for（k=0；k<5；k+ +）
printf（"% .1f，"， *（pf+ +））；                /* 依次输出数组各元素 */
pf=data；
printf（" \ n"）；
for（k=0；k<5；k+ +）
printf（"%.1f，"，*（pf+k））；                  /* 依次输出数组各元素 */
printf（" \ n"）；
}
```

当 pf=data；后，*（pf+k）与 data［k］是等价的，都是访问数组的 k 个元素。其实，使用上述两种方式对数组元素进行访问时，系统都是先进行地址计算（计算结果为 data+k），然后访问该地址的目标，运算符*的作用正是访问地址的目标。*（pf+ +）是另一种访问数组元素的方法，通常被称作指针移动法。即对指针本身作加减运算，使指针先指向将访问的元素，再访问之。下面的例题将进一步说明使用指针访问数组元素的方法。

【例 5-3】使用指针对数组元素进行访问的进一步说明。

程序如下：

```
# include < stdio.h >
main（）
{ int m[5]={5，4，3，2，1，}；
int * p=m；
printf（"% d，% d，% d，% d"，m[3]，*（m+3），p[3]，*（p+3））；
}
```

执行该程序的输出结果为：

```
2，2，2，2
```

综合例题 2 和例题 3，访问数组元素的方法一共有三种：

- 下标法，如：m［3］，p［3］。下标法对数组名和指针都适用。

- 偏移量法，如：*（m+3），*（p+3）。偏移量法同样是对数组名和指针都适用。
- 指针移动法，如：pf+ +。显然，指针移动法只对指针适用。

最后，对指针和数组在使用时应注意几点加以归纳：

（1）用指针和数组名在访问地址中的数据时，它们的表现形式是等价的，因为它们都是地址量。

（2）指针和数组名在本质上又是不同的。首先指针是地址变量，其值可以发生变化，可以对其进行赋值和其他运算。例如，指针的以下运算都是合法的。

```
int m［10］，* pm；
pm=m；
pm+ +；
pm- -；
pm+ =k；
```

由于数组名是地址常量，不能对其赋值和其他运算。因此，下面的操作是非法的；

```
int m［10］；
m+ +；
m- -；
m=px；
```

（3）指针在使用前必须赋初值，而数组名不需赋初值。

下面是分别用数组和指针实现字符串比较的实例，从中可以看出数组和指针在用法上的区别。

【例 5-4】使用指针编写一程序，将一个整数数组中存放的数据完全颠倒顺序。

其程序如下：

```
# include < stdio.h >
main（）
{    int data[7]={1，2，3，4，5，6，7，}，temp，k， * ph， * pt；
ph=data；
pt=data+6；      / * 令两个指针一个指向数组的第一个元素，一个指向数组的最
后一个元素 * /
while（pt>ph）{
temp=* ph；
* ph= * pt；
* pt =temp；      / * 数据进行交换 * /
ph + +；
pt- -；            / * 两指针向中间移动* /
}
for（k=0；k<7；k+ +）
printf（"% d，"， * （data + k）；
printf（" \ n"）；
}
```

题中使用了两个指针，初始时，一个指针指向数组的第一个元素，另一个指针指向数组的最后一起元素，然后一边交换数据，一边向中间移动，直到在数组的正中相遇。这种处理问题的方法显然是十分简单、自然而形象的，这利益于指针对数组元素存取的灵活性。

5.5 字符指针和字符串

类型为 char 的指针用于指向字符变量，因而被称为字符指针。字符指针是 C 语言中最常用的指针类型，因为几乎所有的字符串操作都是通过字符指针来实现的。

字符指针初始化的方法有如下两种形式：

（1）在指针定义的同时进行初始化。例如：

```
char * p=“ a string ”；
```

需要注意的是对字符指针初始化，就是将字符串的首地址赋给指针，这类似于数组的指针操作，而不是将字符串本身复制到指针中。指针初始化就是使指针指向该字符串。也可以用如下形式进行指针初始化：

```
char str［20］；
char * psl= str；
char * ps2=ps1；
```

如果数组 str 中包含一字符串，则可以把该数组的首地址赋给指针。用已初始化的指针来初始化另一指针也是可行的方法。

（2）利用赋值语句来初始化指针：

```
…
char * s；
…
s="string"；
```

于是指针 s 就指向字符串"string"。同样，该赋值语句也是把字符串"string"的首地址赋给 s，而不是把字符串本身赋值给 s。

【例 5-5】使用指针编写程序，比较两个字符串的大小（不使用字符函数）。

分析：使用两个指针分别指向两个字符串，然后通过移动指针逐步对字符进行比较。其程序如下：

```
# include < stdio.h >
main（）
{    char str1[50]，str2[50]；
char * ps1=str1，  * ps2=str2；
printf（"input a $:"）；
scanf（" % s"，ps1）；
printf（"another $:"）；
scanf（"% s"，ps2）；
while（ * ps1! = ' \ 0' & & + ps2! = ' \ 0'）{
  if（ * ps1! = * ps2）          / * 逐一比较两字符串中的字符，发现不等时跳出
循环 * /
break；
ps1+ +；
ps2 + +；
  }
  if（ * ps2> * ps1）
printf（"string1<string2 \ n"）；
  else
printf（"string1=string2 \ n"）；
}
```

【例 5-6】使用指针编写程序，进行字符串复制（不使用字符函数）。

其程序如下：

```
# include < stdio.h >
main（）
{
char s[20]，buf[20]；
char * ps = s，* pb =buf；
```

```
printf（" \ n Enter string:"）；
scanf（"% s"，pb）；
while（ * ps+ += * pb+ + ! = ' \ 0'）      / * 将 but 中的字符拷贝到 s 中 * /
printf（" \ n % s，% s"，s，- -ps）；
}
```

某次运行情况为：

```
Enter string：
student＜CR＞
student
```

由于在程序复制结束时 ps 指向字符串末尾，所以什么也没有输出。这使我们注意到，由于指针移动灵活，所以编写程序时必须随时关注各指针的位置。最后顺便提醒一下，使用 Printf 函数的% s 格式输出字符时，是从给定的地址开始，到遇上第一个' \ 0 '时结束。据此请读者写出下面程序段的输出结果：

```
char str［20］="abcdefghijk"；
char * ps=str+4；
printf（" % s \ n"，ps）；
```

5.6 指针数组

5.6.1 指针数组的概念

（1）什么是指针数组

指针数组是一个数组，该数组是指针变量的集合，即它的每一个元素都是一个指针变量，这些指针变量具有相同的数据类型。例如：

```
int * p［3］；
char * pc ［5］；
```

定义的都是指针数组，其中 p 数组包含 3 个指针元素，而且它们都是 int 类型的指针。数组 pc 包含 5 个指针元素，它们都是 char 型指针。

指针数组同其他数组一样，按数组下标的次序被存放在一片连续的存储空间，数组名是其所占区域的首地址。指针数组在使用之前必须先定义。例如下面是对指针数组的初始化：

```
main（）
 {
static int m［2］ ［5］；
static int * pm［2］ = {& m［0］ ［1］，& m［1］ ［2］； / * 指针数组的初始化 * /
…
}
```

这时指针 pa［0］指向 a［0］，pa［1］指向 a［1］。如图 5-3 所示。

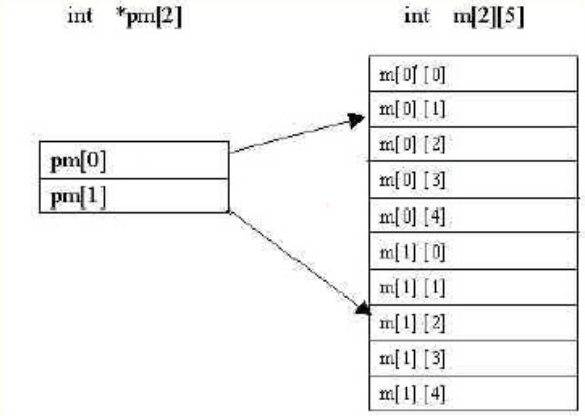


图 5-3 指针数组的初始化

字符指针数组初始化时，可直接使用多个字符串，即把每个字符串的首地址赋给字符指针数组的各元素。

【例 5-7】使用字符指针数组输出若干个人的名字。

其程序如下：

```
# include < stdio.h >
main（）
{    static char name[5][20]；
static char   * p[5]={name[0]，name[1]，name[2]，name[3]，name[4]；
int k；
for（k=0；k<5；k+ +）
scanf（" % s"，p[k]）；
for（k=0；k <5；k+ +）
printf（"% s \ n"，name[k]）；
return 1；
}
```

在本题中，指针数组中存放了一系列一维数组（字符串）的首地址。

5.6.2 指针数组的应用

可用指针数组来处理多维数组，且最常用于处理多个字符串。下面通过实例来进一步说明其在处理多维数组和字符串方面的应用。首先列出使用指针数组对二维数组元素的进行引用的各种方式。

```
int m［3］［6］；
int * prn［3］；
pm［0］=m［0］；
pm［1］=m［1］；
pm［2］=m［2］；
printf（" % d" ，m［2］［4］）；
pritnf（" % d" ，（m［2］+4））；
pritnf（" % d" ，pm［2］［4］）；
pritnf（" % d" ，*（pm［2］+4））；
pritnf（" % d" ，*（*（pm+2）+4））；/ * 5 个
printf 语句将打印出同样的结果即 m［2］［4］的值* /点。
```

【例 5-8】输入星期号（从星期日到星期六分别是 0 到 6），输出对应的英文名称，请使用指针数组实现。

分析：定义一个指针数组，其中的每一个指针指向 7 个字符串中的一个，这样，字符串的意义和指针的标号对应起来，可以方便地使用。其程序如下：

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

```
main（）
{    char  * p[7]={"Sunday"，"Monday"，"Tuesday"，"wednesday"，"Thursday"，
"Friday"，"Saturay"}；
int code；
printf（"input the code of the day（0-6）:"）；
scanf（"% d"，& code）；
printf（"Today is & s"，p[code]）；
}
```

【例 5-9**】输入一个** 2×3 **的整数矩阵和一个** 3×2 **的整数矩阵，请使用指针数组实现这两个矩阵的相乘。其程序如下：**

```
# include < stdio.h >
main（）
{    static int a[2][3]，b[3][2]，result[2][2]；
int  * p[3]；
p[0]=a[0]；p[1]=b[0]；p[2]=result[0]；
/ * 三个指针分别指向三个二维数组的起始地址 * /
printf（" \ nthe lst matrix: \ n"）；
for（int i=0；i <2；i+ +）
{    printf（"new line: \ n"）；
for（int j=0； j <3；j+ +）
scanf（"% d"，p[0]+3 * i + j）；
}  / * 下面的三重循环将两个矩阵乘起来 * /
for（i=0；i <2；i+ +）
    for（int j=0；j <3；j+ +）   / * 前两重循环穷举了 aij * /
for（int k=0；k <2；k+ +）   / * 对于每一个 aij，列举将与之相乘的 bjk * /
*（p[2]2 * i + k）  + = *  （p[0] + 3 * i + j）  *  （ *  （p[1] + 2 * j + k））；
/ *  aij *  bjk，结果累加入 cik */
for（i=0；i <2；i+ +）
{    printf（" \ n"）；
for（j=0；j <2；j + +）
printf（" % d，"，*（p[2]+ 2 * i + j））；
}
}
```

5.7 多级指针

5.7.1 多级指针的概念

（1）什么是多级指针

指向指针的指针叫多级指针。例如：

```
char * ps［3］；
```

是一个指针数组，它有三个元素：ps［0］、ps［1］**和** ps［2］**。这三个元素都是指针，如果在定义的同时，还进行初始化，如：**

```
char * ps［3］= {"Mathematics"，"Physics"，"Chimistry"}；
```

则 ps［0］**指向**"Mathematics"，ps［1］**指向**"Physics"，ps［2］**指向**"Chimistry"。**如果我们再定义一个指针**

变量 p，使它指向指针数组 ps，则 p 就是一个指向指针的指针。这时指针 p、ps 和字符串之间的关系如图 5-4 所示。由于指针 p 指向指针数组 ps，而 ps 指针数组又指向处理的字符串，所以 p 是一个二级指针。由于指针 p 的目标变量是* p 即 ps［0］；而则是 ps［0］的目标变量是* ps［0］，即* * p，其目标为字符' M '。同样道理，* *（p+1）的目标为' P '，* *（p+2）的目标为字符' C '。

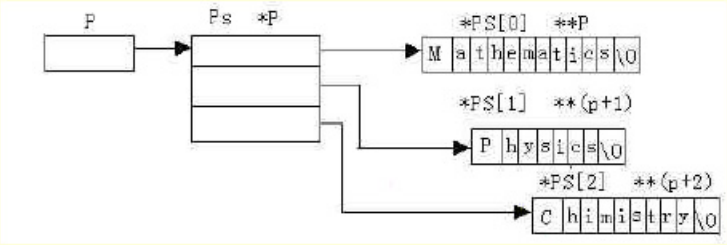


图 5-4 指针 p、ps 和字符串之间的关系

多级指针在引用之前也必须先定义。定义的一般形式如下。

类型标识符 * * … * 指针名

指针名前有一个“ * ”，称一级指针，简称指针。有两个“ * * ”，则称为二级指针；有三个星号“ * * * ”的叫三级指针；其余类推。类型标识是指针最终目标变量的类型。下面是多级指针的举例。

（2）多级指针定义举例

二级指针定义例一：

```
char * * p；
char * pp="Mulpointer"；
p = & pp；
```

其示意图如图 5-5 所示。

图 5-5 二级指针的定义

二级指针定义例二：

```
char * * p；
static char * pp［3］= {"Fortran"，"Pascal"，"Basic"} ；
p =pp；
```

其示意图如图 5-6 所示。

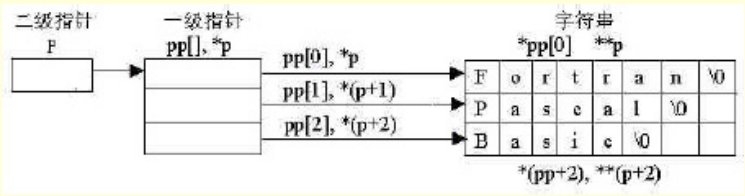


图 5-6 多级指针示意图

三级指针定义举例：

```
char * * * p；
static char * ppp［4］= {"Program"，"Sourcefile"，"Filename"，"Declarepart"} ；
static char * * pp［4］= {ppp［0］，ppp［3］；ppp［1，］ppp［2］} ；… p =pp；
```

其示意图如图 5-7 所示。

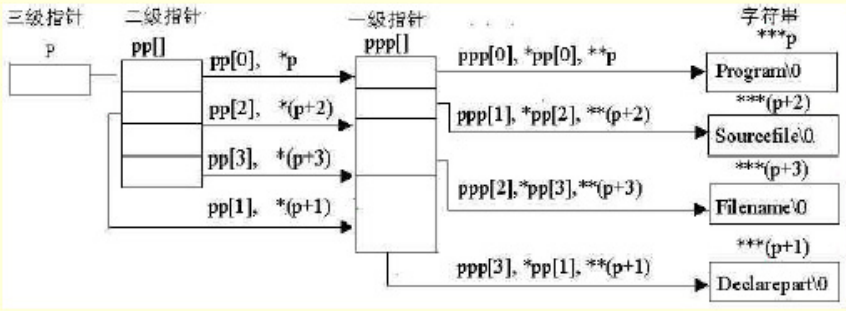


图 5-7　多级指针示意图

在上述各定义中，指针的类型实际是标识最终目标变量的数据类型。而不是指针本身的类型。

5.7.2　多级指针应用举例

在 C 语言中，多级指针主要用来处理多个字符串，用的较多的是二级指针（通常叫指向指针的指针），三级以上指针用的较少。下面是一个应用的例子。

【例 5-10】使用二级指针来改写例题 8 的程序。

分析：可以定义一个二级指针来指向指针数组的首地址，然后使用此二级指针来引用数组中的指针，从而间接引用字符串。

```
# include < stdio.h >
main（）
{    char   * p[7]={"Sunday"，"Monday"，"Tuesday"，"Wednesday"，"Thursday"，
"Friday"，"Saturday"}；
char   * * pp=p；      / * 指向指针的指针指向指针数组的首地址 * /
int code；
printf（"input the code of the day（0-6）:"）；
scanf（"% d"，& code）；
printf（"Today is % s"，  *  （pp+code）；    / * 指向指针的指针间接引用字符串 * /
}
```

本程序的运行结果与例题 8 完全一致。指针间接引用数组元素的形式不唯一，那么二级指针间接引用的形式也是不唯一的，除本题中的*（pp+code）以外，也可以写成 pp［code］的形式，当然，使用二级指针移动法去引用也是正确的。

5.8　综合应用举例

【例 5-11】下面是一段使用指针排序数组元素的程序，找出其中的错误并改正之。

```
# include < stdio.h >
main（）
{    int a[10]；
int * p1；
int * * p2=p1；
*p1=a；
for（int k=0；k <10；  k+ +）
   scanf（"% d"，  * p1 + +）；    / * 逐一输入每个数组元素的值。* /
for（k=0；  k <9；k + +）
{    for（m=0；m <9 - k；m+ +）
{    if（ * p2[m] > * p2[m+1]）
```

```
{    int change= * p2[m]；
* p2[m]= * p2[m+1]；
* p2[m+1]=change；
}
}
}
for（k=0；k<9；k+ +）
   printf（"% d"， * p2 + k）；      / *输出已经排好序的各元素。* /
return（1）；
}
```

解答：

错误（1）：改 int * * p2=p1；**为** int * * p2= & p1；**变量** p2 **是指向指针的指针，应该把指针** p1 **的地址而不是值赋给** p2。

错误（2）：改 * p1=a；**为** p1=a；**在定义指针时赋初值可写成** int * p1=a；**但单独使用赋值语句时应写成** p1=a；**的形式，因为*** p1 **表达的是** p1 **指向的那个变量的值，而数组名** a **是一个地址，两者类型不同，故不能相互赋值。**

错误（3）：改 scanf（" % d"，* p1+ +）；**为** scanf（" % d"，p1+ +）；**我们知道，**C **语言的** scanf **函数的变量列表中应该列出变量的地址而不是变量的标志符，**p1 **相当于数组元素的值或标志符，**p1 **才是数组元素的地址。**p1+ +**的作用是将指针** p1 **不断向后移动，依次指向各数组元素，将输入的数据存入其中。**

错误（4）：改：

```
if（* p2［m］> *p2［m+1］）
 {int change=* p2［m］；
  * p2［m］=* p2［m+1］；
  * p2［m+1］=change；
}
```

为：

```
if（（* p2）［m］>（* p2）［m+1］）
 { int change=（* p2）［m］；
   （* p2）［m］，（* p2）［m+1］；
   （* p2）［m+1］=change；
}
```

我们可以使用指针 p1 **来引用数组元素，方式是：**pl［m］，**而此程序中** p2 **指向** p1，**即***p **等价于** p1，**所以我们用（*** p2）［m］**来引用数组中的数据元素完全等价。那么*** p2［m］**是否和（*** p2）［m］**等价呢？由于运算符［］的优先级较*****更高，***p2［m］**只能等价于***（*（p2+m））**，而后面这个表达式完全错误，因为** p2+m **指向谁，我们都不知道。**

错误（5）：改 printf（" % d"，* p2+k）；**为** printf（" % d"，*（* p2+k））；printf **函数的变量列表中应该给出变量标志符而不是其地址。**

【例 5-12**】下面这些关于指针的定义、赋值和运算的语句是否有错。**

```
int   * pm，* pn，  * pk，  * point1，  * point2；
int m=5，n，data[8]；
* mp =m；
pn=n；
point1 = & data；
point2 = & data[7]；
point1+point2；
point2-point1；
scanf（" % d"，pn）；
```

scanf（" % d"， * pm）；

解答：

错误（1）：改 * pm=m；**为** pm=& m；，**因为指针** pm **未指向任何变量，不能对***pm **赋值，只能对** pm **赋地址。**

错误（2）：改 pn=n；**为** pn=& n；**因为对于指针变量，只能将地址赋给它。**

错误（3）：改 Point1=& data；**为** point1=data；**因为数组名** data **本身已经是一个地址。**

错误（4）：point1+point2 **没有意义，因为指针是不能相加的。**

错误（5）：改 scanf（" % d，* pm）；**为** scanf（" % d"，pm）；**因为** scanf **函数中的列表必须是地址。**

【例 5-13**】有如下程序段，题中选项正确的是（）。**

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

②int * p1=a；

③int p2=& a［0］［0］；

④int（* p3）［4］=a

⑤int（* p4）［4］=& a［0］［0］；

⑥printf（" % d \ n"，*（p1+2））；

⑦printf（" % d \ n"，*（p2+5））；

⑧printf（" % d \ n"，*（*（p3+1）+5））；

⑨printf（" % d \ n"，*（p3［1］+1））；

⑩printf（" % d \ n"，*（*（p4+1）+2））；

（A）②③④⑤**四句定义赋值都正确。**

（B）**只有②③两句定义赋值正确。**

（C）**只有③④两句定义赋值正确，②⑤两句虽然编译时不出错，但指针不能正确存取。**

（D）**只有③④两句是正确的，②⑤两句定义赋值在编译时将报错。**

答案 D。**二维数组的数组名相当于二级指针，不能赋值给一级指针** p1，**这是语句②错误的原因。**p4 **是指向一维数组的指针，相当于是二级指针，不能把整型变量的地址赋给它，这是语句⑤错误的原因。**

【例 5-14**】有如下函数，题中选项正确的是（）**

void xxx（char * p1，char * p2）

｛while（* p1+ + = * p2 + +）；｝

（A）**此函数的功能是将** p2 **指向的字符串拷贝到** p1 **所指的字符串中。**

（B）**此函数的功能是将** p1、p2 **指向的字符串中的字符的** ASCII **码值都加** 1。

（C）**此函数不能正确地执行，必将陷入死循环中。**

（D）**此函数能执行，但是不能完成任何有效的操作。**

答案：A。**请注意*** p1+ +，* p2+ +**等价于***（p1+ +），*（p2+ +），**故指针将不断移动，结果将** p2 **指向的字符串中的字符逐一拷贝到了** p1 **指向的字符串中。**

5.9 小 结

（1）**指针是** C **语言的又一个重要特色，它是一种特殊的变量，其内容是所指对象的存储地址。指针所指区域的数据是指针目标，如果存目标的空间也是某变量的空间，则该变量也叫该指针的目标变量。指针所指的对象可以是简单变量，也可以是数组、结构、联合或函数等。如果** px **是指针，则*** px **表示该指针的目标。**

（2）**指针在使用之前应当先定义，定义的一般格式为：**

类型标识符 *指针名［，*指针名…］；

其中，类型标识符指明目标变量的数据类型，而不是指针本身的类型，目标变量的类型简称指针变量的类型。

（3）**指针在定义时可以初始化，初始化的一般形式为：**

类型标识符 *指针名＝初始值

初始化是对指针变量的初始化，而不是目标变量。初始值一定得是地址量，而不能是一般数。

（4）指针可进行算术运算、关系运算和赋值运算。其中算术运算包括指针变量与整数的加、减，二指针变量相减等。

（5）在 C 语言程序中可用指针访问数组。如果 px 是指向数组 x []的指针，则 x [i]与*（px+i）或（x+i）是等价的表现形式。但 px 和 x 也有本质区别，px 是指针变量，而 x 是地址常量。

（6）字符指针可以用字符串初始化，其含义是将字符串的首地址赋给指针。

（7）可以定义指针数组，常用指针数组来处理多个字符串。

（8）指向指针的指针叫多级指针，多级指针也是主要用于处理多个字符串。

习 题

1．输入三个整数，按由小到大的顺序输出。

2．有 n 个整数，使前面各数顺序向后移 m 个位置，最后 m 个数变成最前面 m 个数，如图 5-8 所示。写一个函数实现以上功能，在主函数中输入 n 个整数和输出调整后的 n 个数。

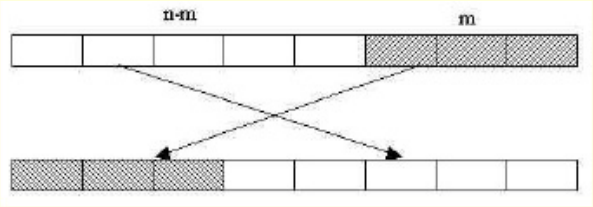


图 5-8 移数示意图

3．编写一个程序，输入月份号，输出该月的英文月名。例如，输入数字“ 5 ”，则输出“ May ”，要求用指针数组处理。

4．编写程序，输入两个 4×4 的整数矩阵，相加后再转置。

5．下面程序的功能为输出字符串 abc 的第一个字符和最后一个字符，请指出程序中的错误。

```
main（）
{
char str[4]，c；
str = "abc"；
a = &str；
c = a[4]；                    /*将 str 的最后一个字符赋给 c*/
printf（"%c"，c）；
printf（"%c"，*a+1）；              /*打印 str[1]*/
}
```

6．下面程序的功能为向程序输入 5 个字符，并一一输出这 5 个字符，请指出程序中的错误。

```
main（）
{
char c[5]，*p；
int i；
*p = c；
for（i=0；i<5；i++）
{
p=getchar（）；
printf（"%d"，（*p）++））；/*输出数组元素的值，并使指针增 1*/
}
}
```

第六章　函　数

函数是 C 语言程序的基本构件，也可以说一个 C 语言程序是一个或多个函数的集合体。正因为如此，本章专门讨论函数的定义、引用及其参数传递的方法等问题。

6.1　概　述

一个较大的程序一般应分为若干个程序模块，每一个模块用来实现一个特定的功能。所有的高级语言中都有子程序这个概念，用子程序来实现模块的功能。在 C 语言中，子程序的作用是由函数来完成的。一个 C 程序可由一个主函数和若干个函数构成。由主函数调用其他函数，其他函数也可以互相调用。同一个函数可以被一个或多个函数调用任意多次。图 6-1 所示是一个程序中函数调用的示意图。

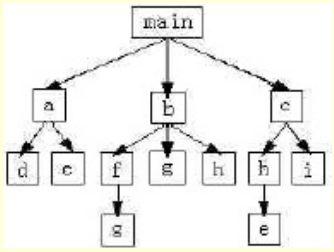


图 6-1　C 程序中的函数调用示意图

在程序设计中，常将一些常用的功能模块编写成函数，放在函数库中供公共选用。要善于利用函数，以减少重复编写程序段的工作量。

先看一个简单函数调用的例子。

【例 6-1】

```
main（）
{printstar（）；                    / *  调用 printstar 函数  * /
print_message（）；                 / *  调用 print_message 函数  * /
printstar（）；                     / *  调用 printstar 函数  * /
}
printstar（）                        / * printstar 函数  * /
{
print_message（）                    / * print_message 函数  * /
{
printf（"        How do you do ! \n"）；
}
```

运行情况如下：

```
* * * * * * * * * * * * * * * * * *
        How do you do !
* * * * * * * * * * * * * * * * * *
```

printstar 和 print_message 都是用户定义的函数名，分别用来输出一排星号和一行信息。

说明：

（1）一个源程序文件由一个或多个函数组成。一个源程序文件是一个编译单位，即以源文件为单位进行编译，而不是以函数为单位进行编译。

（2）一个 C 程序由一个或多个源程序文件组成。对较大的程序，一般不希望全放在一个文件中，而将函数和其他内容（如预定义）分别放到若干个源文件中，再由若干源文件组成一个 C 程序。这样可以分别编写、分别编译，提高调试效率。一个源文件可以为多个 C 程序公用。

（3）C 程序的执行从 main 函数开始，调用其他函数后流程回到 main 函数，在 main 函数中结束整个程序的运行。main 函数是系统定义的。

（4）所有函数都是平行的，也就是说在定义函数时，不同的函数是互相独立的，一个函数并不从属于另一函数，即函数不能嵌套定义，但可以互相调用，但要注意：不能调用 main 函数。

（5）从用户使用的角度看，函数有两种：

① 标准函数，即库函数。这是由系统提供的，用户不必自己定义这些函数，可以直接使用它们。应该说明，每个系统提供的库函数的数量和功能不同，当然有一些基本的函数是共同的。

② 用户自己定义的函数，以解决用户的专门需要。

（6）从函数的形式看，函数分两类

① 无参函数。如例 6-1 中的 printstar 和 print_message 就是无参函数。在调用无参函数时，主调函数并不将数据传送给被调用函数，一般用来执行指定的一组操作（如例 6-1 中 printstar 函数的作用就是输出 18 个星号）。无参函数可以带回或不带回函数值，但一般以不带回函数值的居多。

② 有参、在调用函数时，在调用函数和被调用函数之间有参数传递，也就是说，主调函数可以将数据传给被调用函数使用，被调用函数中的数据也可以带回来供主调函数使用。

6.1 函数的定义和引用

6.1.1 函数的定义

函数定义的一般形式如下：

```
[数据类型标识符] 函数名（[形式参数表]）
[形式参数说明；]
{
内部数据说明语句；
可执行语句；
}
```

例如：

```
int max（i，j） /*函数的定义 */
int i，j；/*形参定义 */
{
int k；/* 内部数据说明语句*/
k=i
if（k<j）/*函数内部的执行语句 */
k=j；
return（k）；/*返回结果 */
}
```

此例中，max 是函数名，其类型为 int，即返回值为 int 型。i 和 j 是该函数的形式参数。“ int i，j ；” 是形式参数的说明。该例比较 i 和 j 值的大小，用 return（k）返回 x 和 y 中较大者的值。

在一般函数定义中，数据类型标识符指出要定义函数的返回值的数据类型，缺省时，为整型无返回值，无返回值的函数类型也可以用 void 定义。函数名的构成规则同一般用户定义标识符的命名规则。函数可以有一个

或多个形式参数，也可以没有形式参数。没有形式参数时也必须有“（）”。圆括号“（）”是函数运算符。形式参数的具体值由调用函数提供。形式参数要在函数名和函数体之间说明，但其有效范围只局限于该函数之内，即形式参数是局部变量。

形式参数可以是变量、指针或数组名等，但不能是表达式或常量等。形式参数的数目从文法上讲并没有具体限制，但过多时会影响函数的速度，而且使函数显得不精练。

由“{”和“}”所括的内容是函数体。函数体中的内部数据说明语句从理论上讲可以出现在任何分程序的开始，推荐把说明部分写在第一对花括号的开始。如果函数有返回值，则应包含 return 语句，return 语句的一般格式为：

return（表达式）；/ *　有返回值　* /

如果没有返回值时，可以写为

return；

无返回值时也可以省略 return 语句。

在定义函数时需要注意的是：函数的定义不能嵌套，即不能在函数的定义体内又包含另一个函数的定义。这就保证了每一个函数是一个独立的和功能单一的程序单元。在由多个函数组成的 C 语言程序中，函数定义的先后顺序与其被引用的先后次序无关，即函数的定义次序不影响其引用次序。由此看出，一个 C 语言的程序实质上是一系列相互独立的函数的定义，函数之间只存在引用和被引用的关系。

下面是一个返回值为整数的函数定义实例：

```
fun（n）
int n;
{
int i;
i =n;
while（- -n > 0）
i =i +n
return（i）;
}
```

此函数实现 1+2+3+…+n，定义中省略了数据类型标识符，因为函数的返回值是整数时，数据类型标识符可以缺省。

C 语言程序准许定义“空函数”，其一般形式为：

```
类型标识符　函数名（）
    {
    }
```

例如：

```
dummy（）　{}
```

调用此函数时，什么工作也不做，它只表明此处要调用一个函数，而现在尚未明确调用什么具体函数，等待以后功能扩充时再补充上去。例如一个系统包括若干模块，每个模块都对应一些函数，在系统开发的初期，只包括一些基本的功能模块，而其他一些模块待以后再开发，这时这些没开发出来的部分可以用 dummy（）函数替代。这样做使程序结构清晰、便于扩充。

6.1.2　函数的引用

在 C 语言程序中，函数引用的一般形式如下：

函数名（［实参表］）

如果函数定义中包含有形式参数，则在函数引用中应包含实际参数（简称实参），而且实参的个数、数据类型和顺序应当与形参相同。函数引用可以用以下三种形式中的任何一种：

（1）以函数引用语句的形式出现，例如：

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

这两个语句都是函数引用语句，简称函数语句。

（2）在表达式中出现，例如

```
y=max（x，y）* fun（n）；
```

其中 max（）和 fun 均是函数，其定义在前面已经出现。

（3）在函数引用中以实参的形式出现，例如

```
y=cos（tg（x））；
```

函数 tg（x）以实参的身份出现在函数 cos（）的引用中。

总之，在 C 语言程序中，凡是能引用变量或表达式的地方，均可引用函数。同变量一样，函数也应该先说明（或定义）后引用。函数说明的目的是告诉编译程序该函数返回什么类型的值，以便使编译程序能够检查对函数的调用是否正确，会不会发生错误的类型转换等。函数说明的一般形式如下：

```
［数据类型标识符］函数名（）；
```

例如：

```
float sqrt（），avarage（）；
```

在函数说明中应注意如下几点：

（1）如果被调用函数的说明放在源文件的开头，则该说明对整个源文件都有效。

（2）如果被调用函数的说明是在调用函数定义的内部，则该说明仅对该调用函数有效。

例如：

```
main（）
{
int max（）；              / *  该说明位于 main （）函数内部  * /
int i,k,m;
scanf（" % d % d",& i,& j,& k）；
m = max（i,j,k）；
printf（" \ n % d:",m）；
}
max（a,b,c）
int a,b,c;
{
int z;
if（a > b）
z=a;
else
z=b;
if（c > z）  z = c;
return（z）；
}
```

如果被调用函数的说明是在函数之外，则该说明对此后的所有调用都有效。

（3）下面几种情况可省略说明：

- 如果调用函数和被调用函数是在同一个源文件中，而且被调用函数的定义是在调用函数之前，此时可以省略说明。例如：

```
max（x,y,z）
int x,y,z;
{
int result;
```

```
if（x > y）
   result = x;
else
   result =y;
if（ z > result）
   result = z;
}
main（）
{
int i,j,k,m;
scanf（"% d,% d,% d",& i,&j,&k）;
m = max（i,j,k）;
printf（" \ n max = % d",m）;
}
```

此例中 max（）函数的定义是在 main（）之前，故在调用函数 main（）中可以省略说明。

- 如果函数的返回值是整数或字符时，可省略说明。
- 如果所有被调用函数的说明都是在源文件开头，则在该源文件内的所有调用函数中不必再对被调用函数进行说明。
- 如果被调用函数不是在源文件开头，而是在源文件中间且在所有函数之外某处被说明时，则在被说明之后引用它时，不必再说明，但在被说明之前引用它时，需要对其进行说明。

（4）如果调用函数和被调用函数的定义是在同一源文件中，应该先说明被调用函数之前的引用。函数引用应该注意如下几点：

- 实参应在个数、类型和顺序上与形参相一致。
- 实参可以是常量、变量名、数组名、数组元素或表达式，即必须具有确定的值。
- 为了保证函数引用的正确性，在引用之前应当首先弄清楚被引用函数的功能、输出参数、返回值等，然后再进行引用。

【例 6-2】编制求两数相乘的程序

```
main（）
{
float mul（）;
float x,y,z;                    / * 进行两数相乘的函数 * /
scanf（" %f,% f",&x,&y）;       / * 定义主函数内部的局部函数 * /
z=mul（x,y）;                   / * 输入要进行相乘的两个数 * /
printf（"The product is % f",z）  / * 调用函数，进行两数的相乘 * /
}
float mul（float x,float y）          / * 函数及形参类型定义 * /
{
float z;                        / * 定义浮点变量 * /
z=x * y;                        / * 两数相乘 * /
return（z）;                    / * 返回结果 * /
}
```

6.1.3 C 语言程序的执行过程

C 语言程序从启动、执行到结束的执行过程顺序如下：

（1）启动程序

当程序启动时，控制由操作系统传给该程序的 main（）函数，于是程序上的主函数开始执行。

（2）执行程序

在主函数的执行过程中，每当遇到函数调用，如图 6-2 中调用 fun1（）函数，就按下述步骤调用该函数：

- 传实参给被调用函数；
- 将控制传给被调用函数，于是被调用函数开始执行；
- 被调用函数保存调用函数的执行现场，其中包括断点等；
- 执行被调用函数的函数体，每遇到调用其他函数时，就重复（2）的步骤调用其他函数；
- 被调用函数执行结束时（如遇到 return 语句，或遇到函数体的结束括号“}”时），则恢复现场，控制返回调用函数（如主函数）的调用处。

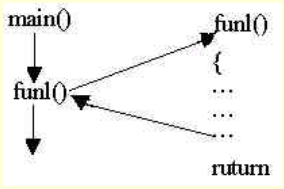


图 6-2 C 语言程序中的函数调用过程

（3）程序的结束

主函数执行结束时，控制退出主函数返回操作系统。所以，程序的执行是从主函数进入，又是从主函数结束，主函数是整个程序的主控程序。C 语言程序的执行过程如图 6-2 所示。

6.2 变量的存储类型及作用域

C 语言变量的存储类型决定了该变量分配的存储区类型，存储区域的类型又决定了它的作用域（即可见性）和生命期（即变量值的存在时间）。C 语言变量有如下四种存储类型：

- 自动型或堆栈型（auto 型）
- 外部型，或全局型（extern 型）
- 寄存器型（register 型）
- 静态型（static 型）

一个 C 语言程序在运行时所用的存储空间，通常包括以下三部分：

（1）程序区

主要用于存放执行的代码和静态变量。

（2）静态存储区

存放程序的外部变量。

（3）运行栈

程序运行时，系统为其分配一个运行栈，该栈用于存放以下数据：

- 函数调用时，保存调用点的现场，如寄存器中的值，返回点等。
- 函数调用的参数，在调用时给形参分配空间，以存放调用时传送给被调用函数的实际参数。
- 为自动量或寄存量分配空间。
- 存放函数的返回值等。

下面分别对各种存储类型进行说明。

6.2.1 自动型变量

自动型变量是在函数内部定义的一种变量，它局限于该函数，或所在的分程序。故也称为局部变量。自动变量定义的一般形式如下：

[auto]数据类型标识符 变量名表；

其中 auto 可以省略，例如：

```
funcname（a,b）
int a,b;
{
int i,j;
char cl,c2;
{
float f1,f2;
int m,n;
double d1,d2;
{
auto int k;
…
}
}
}
```

自动型变量定义在函数内部，更确切地说是定义在某对花括号之内的开始位置，即左花括号“{”之后。当函数被调用时，每当控制进入一对花括号（即分程序）时，就在运行栈中为其分配存储空间（如果有自动量定义时），当该分程序执行结束（即遇到相应的右花括号）时，就释放为其所分配的空间，变量的值也随之消失。所以，自动型变量的作用范围是其所在的一对花括号，即局限于所在的花括号。其生命期是在执行所属的分程序这段时间区间。函数的形参也属于自动型变量，当函数调用时在运行栈为其分配空间，函数执行结束时，释放其空间。因此，它的作用域是局限于该函数，当该函数执行时，其生命期即开始，当函数结束时，其生命期就告终。

6.2.2 外部变量

外部变量是全局变量，它在函数之外定义，其定义格式如下：

［extern］数据类型标识符 变量名表：

其中 extern 标识符可以省略。

例如：

```
int i,j;
fun（）
{
float x,y;
char ch[10];
double d;
…
float x,y;
double d;
…
}
```

其中 i、j 是在函数之外定义，所以它是全局变量，x、y、ch［］和 d 是在函数内部定义，故是自动型变量或局部变量。在编译时，在静态存储区为外部变量分配内存。同静态变量一样，所分配的内存在整个程序执行过程中始终归该变量所有，其值不会消失。也就是说，外部变量的生命期也是全局的。

外部变量的作用域是整个程序，即外部变量对程序中所有函数都是可见的。但是需要注意如下几点：

（1）如果外部变量是在源文件中各个函数之前定义，则该源文件中的各个函数都可以使用它，不需另加说明。

（2）如果外部变量是在一个源文件中间定义，则在其定义之前的函数中使用它时，应该用 extern 进行说明，

说明可以在函数之外，也可以在函数之内进行，其一般格式如下：

extern 数据类型标识符 变量名表：

例如：

```
extern float x,y;     / *  外部变量 x 和 y 在定义之前的函数中使用应说明* /
fun（）
{
…
y=x+2.456;
…
}
float x,y;     / * x,y 在该源文件中间定义  * /
main（）
{
…
}
```

注意：变量的说明和定义是不同的。说明只是指出变量的特征（长度、类型等），并不分配存储，而定义还要分配存储。

（3）在一个源文件中定义的外部变量，可以在另一个源文件中引用，但必须用 extern 进行说明。说明的方法同（2）。例如，一个程序包含 filel.c 和 file2.c 两个源文件：

文件 file1.c 的内容为：

```
int i，j；                    / *该源文件定义了外部变量 x 和 y * /
main（）
 {
…
}
fun（）
 {
…
}
```

文件 file2.c 中的内容为：

```
extern int i，j；/ *在该文件中使用第一个文件中定义的外部变量应说明* /
fl（）
 {
…
j=i * i+1；
…
}
f2（）
 {
…
printf（" % d % d"，i，j）；
…
}
```

i 和 j 在 file1.c 源文件中被定义为外部变量，在第二个源文件 file2.c 中使用时，就应在第二个源文件的开头用 extern 说明之。有了这个说明之后，file2.c 中的各个函数都可以使用该外部变量了。

（4）如果外部变量与函数内的局部变量同名，则在该函数内的同名变量的作用域内，局部变量优先，即局

部变量有效，而外部变量暂不起作用。下面是外部变量应用的例子：

```
int i;
static int j;
main（）
{
i=1;
j=2;
printf（" \ ni= % d j= % d",i,j）；
add（）；
printf（" \ ni = % d j= % d",i,j）；
}
add（）
{
i=i + j;
}
```

运行结果：

```
i=1        j=2
i=3        j=4
```

在此例中定义了两个外部变量 i 和 j，其中 j 是外部静态变量，两个函数中均可使用，其值一直保持到程序执行结束。

6.2.3 寄存器变量

寄存器变量是与硬件相关的一类变量。我们知道在每一种计算机的 CPU 内都包含若干个能用寄存器，硬件在对数据操作时，通常都是先把数据取到寄存器（或一部分取到寄存器中），然后进行操作。CPU 对寄存器中数据的操作速度要远远快于内存中数据的操作速度。为了加快操作速度，C 语言特引入寄存器变量。寄存器变量一般在程序执行时分配 CPU 的能用寄存器。由于 CPU 中能用寄存器的数目有限，所以，通常都把使用频繁的变量定义为寄存器变量。例如，在循环次数特别大的循环中，可把循环变量定义为寄存器变量，以提高处理速度。

寄存器变量的引用使 C 语言具有汇编语言的特征。由于硬件中寄存器的数目有限，因而在一个函数内准许定义的寄存器变量个数也有限，一般是 2 至 4 个，这要视机器类型而定，超过规定数目的，就按自动变量来处理。在有些计算机系统的 C 语言中，虽然准许定义寄存器变量，但并不对其分配寄存器，而是作为自动变量来分配其存储空间。

一般数据类型为 long、float 和 double 的变量不能定义为寄存器类型，因为这些数据类型的长度超过了寄存器本身的长度。例如 long 型（长整型）变量占 4 个字节，而通常寄存器是 16 位，2 个字节，所以通常只有 int 、short 和 char 类型的变量才准许定义为寄存器类型。

寄存器变量是局部变量，它只适用于自动变量和函数的形式参数。所以，它只能在函数内或分程序内定义，因此它同自动变量一样是局限于函数或分程序。其作用域和生命期同自动变量一样。如果函数的形式参数被定义为寄存器变量，则当函数被调用时，为其分配寄存器；如果没有多余的寄存器供分配，就在运行栈中为其分配存储空间。当函数执行结束时，就释放其分配。因此，其作用域是整个函数，其使命期是函数的每次调用。如果函数的某一分程序中定义了一个寄存器变量，则每当执行该分程序时，就为其进行分配，每当该分程序执行结束时，就释放其分配；所以它的作用域是该分程序，而其生命期为该分程序的每次调用。寄存器变量定义的一般形式为：

```
register 数据类型标识符  变量名表；
```

例如：

```
register int i，j；
```

寄存器变更和自动变量都是在函数执行时才进行分配的，因此，它们都是动态分配的。下面是寄存器变量

定义的例子。

【例 6-3】编写程序，求 $1^4+2^4+3^4+\cdots+n^4$。

```
main（）
{
lont int sum;                          / * 定义 sum 为长整型 * /
int n;
register i;                            / * 定义寄存器类整型变量 * /
printf（"Please input integer n:"）;      / * 提示输入 n * /
scanf（" % d",& n）;
sum = 0;                               / * 赋初值 * /
for（i=1;i < =n;i + +）
sum=sum+i * i * i *i;                  / * 求和 * /
printf（" \ n sum = % d",sum）;          / *输出结果 * /
}
```

【例 6-4】编写程序计算阶乘 n!。

```
long fact （n）
register int n;                        / * 定义寄存器类整型变量 * /
{
int i;
long facto;
facto=1;
for（i=1;i < =n;i + +）
facto * =i;                            / * 求阶乘 * /
return（facto）;                       / * 返回阶乘结果 * /
}
```

6.2.4 静态变量

静态变量是与自动变量和寄存器变量不同的另一类变量，它不是动态分配，而是静态分配的变量。即在编译时，在特定的存储区内为其分配相应的存储空间，所分配的存储空间在整个程序运行中自始至终归该变量使用。静态变量分内部静态变量和外部静态变量两种。

（1）内部静态变量

内部静态变量同自动变量，也是在函数内定义，更确切地说是在分程序内定义，它局限于定义它的分程序。其定义的一般形式为：

```
static  数据类型标识符 变量名表；
```

例如：

```
static int i，j；
static char ch［100］；
```

但内部静态变量又不同于自动变量，它不是在每当函数被调用且执行到所在的分程序时才存在，退出分程序时就消失，而是在整个程序执行过程中始终保持存在。也就是说，在函数被编译时，在该函数所处的程序区内为其所定义的内部静态变量分配专用的、永久性存储单元。内部变量的作用域是定义它的分程序，而在该分程序之外是不可见（也就是说不能使用）的。其生命期是程序的整个执行过程，而不是函数的每次调用。一个内部静态变量的值在函数的一次调用结束时并不消失。它在下次调用时不可继续使用原来保留的值。所以，内部静态变量具有局部的可见性和全局的生命期。

（2）外部静态变量

外部静态变量是在函数外部定义的变量，其作用域是定义它的源文件。即对于定义它的源文件是全程知道的，而对该源文件之外的文件则是不可见的。因此，外部静态变量的名字与其他源文件中的同名变量互不相干。

外部静态变量是在编译时，在包含它的源文件所在的代码区中为其分配存储空间，该空间在整个程序执行过程中都归该变量所有，直到程序执行结束时才释放，其生命期是整个程序的执行期间。外部静态变量一般在源文件的开始和所有函数之外定义，其定义形式同内部静态变量：

```
static 类型标识符 变量名表；
```

例如：

```
static float f1，f2；              / * 外部静态变量定义* /
fun（）
 {
int i，j；
static float x，y；                / * 内部静态变量定义* /
static char c；
…
}
```

外部静态变量提供了把数据隐藏起来的一种手段，使得外部不能访问它们，同时，也使不同源文件中的同名外部静态变量不会冲突。

【例 6-5】内部静态变量的作用域。

```
main（）
{
int i,l;
static int j;
register int k;
i =1
j =2
k =3
l =4
printf（" \ n…IN MAIN…\ n"）；
printf（"i= % d j = % d k = % d \ n",i,j,k,l）；
fun（）；
printf（"…IN MAIN…\ n"）；
printf（"i = = % d j = % d k = % d \ n",i,j,k,l）；
}
fun（）
{
int i,l;
static int j;
register k;
i =10;
j =20;
k =30;
l =40;
printf（"…IN FUNCTION…\ n"）；
printf（"i = = % d j = % d k = % d \ n",i,j,k,l）；
}
```

此程序的运行结果如下：

```
…  IN MAIN…
i=1 j=2 k=3 l=4；
```

… IN FUNCTION…

i=10 j=20 k=30 l=40；

… IN MAIN…

i=1 j=2 k=3 l=4；

由此例可知，函数 main（）中定义的变量 i、j、k、l 与 fun（）函数中定义的变量名字虽然完全相同，但它们是相互独立的。因为两个函数中的自动变量在运行栈中分配不同的存储区域，二者互不相干。两个函数中的内部静态变量在各自的程序代码区域中分配空间，二者也是互不相干。寄存器变量也是如此。表 6-1 描述了变量的存储类型分类、定义、作用域和生命期。

表 6-1 变量的存储类型

分类	存储类名	定义位置	定义举例	作用域	生命期	说 明
局部变量	自动变量	函数之内	Auto int x,y;	定义它的函数或分程序内	函数或分程序的一次执行	如果在函数体开始定义，则作用域是该函数。如果在分程序内定义，则作用域是该分程序。
局部变量	内部静态变量	函数之内	static int x,y;	同自动型	同自动型	同自动型
局部变量	寄存器变量	函数之内	register int x,y;	同自动型	同自动型	同自动型
全局变量	外变量	函数之外	int x,y;	整个程序	整个程序执行过程	在定义位置之前或在其他源文件中使用时，需加说明。

续 表

分类	存储类名	定义位置	定义举例	作用域	生命期	说 明
全局变量	外部静态变量	函数之外	static int x,y;	定义它的源文件	整个程序执行过程	

在论述了存储类型之后，可给出变量的完整定义或说明，其一般格式如下：

［存储类］数据类型标识符 变量名表；

例如：

```
static char * p；
register int k，l；
static float f1，f2，f3，f［］ = {0.1，0.2，0.3}
int i，j，c［100］ ；
auto char ch［200］
```

指针的存储类型是指指针变量本身的存储类，数据类型是指指针目标变量的数据类型。

对于函数来说，由于函数的定义总是在其他函数之外，所以，从本质上讲函数的存储类型都是外部的。但根据需要，又可把函数分为外部静态和一般外部两类。外部静态函数也称静态或内部函数。函数定义的完整格式为：

```
［存储类型］［数据类型标识符］函数名（形参表）
形参说明
 {
内部数据说明语句；
执行语句；
}
```

外部存储类型用 extern 标识，静态用 static 标识；对于外部类型，可以省略 extern。

静态函数局部于它所在的源文件，即对它所在源文件中的各函数是可见的，而对别的源文件中的函数是不可见的（即不能引用它）。所以，不同源文件中的内部函数可重名。

在函数定义中，如果缺省存储类型说明，则认为是外部函数。外部函数的作用域是整个程序，因此，在该作用域内的任何其他函数都可以引用。一个函数若在别的源文件中引用它，或虽在定义它的源文件中引用，但却是在这定义之前时，需加如下说明：

extern［数据类型标识符］函数名（）；

6.3 函数间的通信方式

C 语言程序是由函数所组成，这些函数都是为解决某一问题而聚集在一起的。它们如何构成一个有机的整体，在运行时得到所需的结果呢？其办法就是在程序执行期间，通过相互之间传递信息和数据来实现。在 C 语言中，实现函数间的通信有如下几种办法：

（1）参数传递方式，包括传值方式和传地址方式。

（2）函数返回值。

（3）全局变量，但一般不提倡使用。

（4）文件。

下面分别叙述这几种方式。

6.3.1 传值方式

传值方式又称数据复制方式，它是把实参值本身复制给被调用函数的形参，使形参获得初始值。在传值方式的函数调用中，形式参数是变量，而实际参数可以是变量、常量、函数调用和表达式等。例如：

```
float aver（float f1,float f2,float f3,float f4）
{
float average;
average =（f1+f2+f3+f4）/4;
return（average）;
}
main（）
{
float a,b,c,d,average;
printf（"Please enter a,b,c,d:"）;
scanf（"% f % f % f",&a,&b,&c,&d）;
average = aver（a,b,c,d）;
printf（" \ naverate = % f \ n",average）;
}
```

某一次程序运行结果：

```
please enter a，b，c，d：4 4 5 4＜CR＞
Averate=4.250000
```

当在 main（）函数中调用到 aver（）函数时，就在运行栈动态为 aver（）函数的形参分配存储，并把调用它所提供的实参值复制到运行栈形参所占用的存储中，这时 aver（）函数就利用传递给形参的值进行相应的计算。例如下面的数据交换程序：

```
swap（f1,f2）
float f1,f2;
{
float temp;
temp=f1;
f1=f2;
f2=temp;
}
main（）
```

```
{
float x,y;
printf（"Please enter x and y:"）；
scanf（"%f %f",& x,&y）；
printf（"\ n x = % f y = % f",x,y）；
printf（"\ n- - - do swap - - - \n"）；
swap（x,y）；
printf（"x = % f y = % f \ n",x,y）；
}
```

某一次的运行结果如下：

```
please enter x and y：1.2 5.78＜CR＞
x=1.200000y=5.780000
- - - do swap- - -
a=1.200000y=5.780000
```

从运行结果可以看出，在调用 sway（x，y）**时，并没有使** x **和** y **发生交换。这就是说传值方式没有副作用。**

6.3.2 地址复制方式

地址复制方式又叫传地址方式，它是把地址常量而不是数据传递给被调用函数的形参。这种方式一般以地址常量作实参，如数组名等；形参一般是指针变量或数组名。下面是地址复制方式的例子：

```
swap（f1,f2）
float * f1, * f2;
{
float temp;
temp = * f1;
* f1 = * f2;
* f2 = temp;
}
main（）
{
float x,y;
printf（"Please enter x and y:"）；
scanf（" % f % f",&x,&y）；
printf（" \ n x=% f y=% f \ n",x,y）；
printf（"…do swap…\ n"）；
swap（&x,&y）；
printf（" \ n x = % f y= % f",x,y）；
}
```

某一次运行的结果如下：

```
Please enter x and y：1.2 5.78＜CR＞
x=1.200000y=5.780000
.do swap…
x=5.780000 x=1.200000
```

此程序确实实现了二数相交换，即被调用函数 swap（）**改变了调用函数中** x **和** y **的值，这就是地址复制方式的副作用。地址复制方式实现的原理是：当函数** swap（& x，& y）**被调用时，地址常量**& x **和**& y **被复制到运行栈中形参所占用的存储中，被调用函数通过引用传递到运行栈中的地址，来对原参数进行间接访问。在** C **语言程序中利用地址复制方式不仅可以传递变量地址，还可以传送数组地址和其他数据结构的地址。例如：**

```
long int muln（a,n）
int a[],n;
{
long int r =1
int i;
for（ie =0; i < =n;i+ +）
r * =a[i];
return（r）;
}
main（）
{
int a[100],i=0;
long int product;
printf（"Please enter integers into a[] \ n"）;
scanf（" % d",& a[i]）;
while（a[i] ! =0）                / * 以 0 作为输入结束标志 * /
{i =i =1;
scanf（"% d",& a[i]）;
}
i=i-1;
product = muln（a,i）;
printf（" \ n product = % ld",product）;
}
```

某一次程序运行结果：

```
Please enter integers into a［］
3 4 5 0＜CR＞
product =60
```

此函数实现 n 个数相乘，当函数 muln（）被调用时，把地址常量 a 传递给被调用函数。

6.3.3 利用参数返回结果

当函数被调用时，其处理结果可以以返回值的形式传递给调用函数。如果要求返回多个结果值时，还可利用参数返回处理结果。例如，标准库函数 scanf（" % d % d"，& x，& y）就是通过参数 x 和 y 返回结果（输入值）的。

我们知道，当使用地址复制方式传递参数时，被调用函数可以改变调用函数中的数据，利用参数返回处理结果是根据该特征来实现的。下面是将输入字符串中的小写字母改为大写字母的程序，此程序就是利用参数返回函数处理结果的。

```
higher（ch）
char * ch;
{
if（ * ch > ='a '& & * ch < =' z '）
   * ch = * ch +  （'A'-'a'）;
}
main（）
{
char str[100];
int i;
```

```
printf（"Please enter a string:"）;
scanf（" % s",str）;
for（i=0;str[i]! =' \ 0';i=i +1）
higher（& str[i]）;
printf（" \ n % S",str）;
}
```

某一次程序运行结果：

```
Please enter a string：hello＜CR＞
HELLO
```

此例中，传送的参数是被转换字符的地址，于是 higher（）**函数可以利用该地址进行间接操作，即把转换后的结果存入被转换字符的地址中。也可以设置专门的变量，用于返回函数的处理结果。例如，下面是求输入串和小写和大写字母个数的程序，**high **用于返回大写字母个数，**low **用于返回小写字母个数。**

```
main（）
{
char str[100];
int high,low;
printf（"Please enter a string:"）;
scanf（" % s",str）;
high = 0;
low =0;
size（str,& high,& low）;
printf（"upcase=% d lowcase = % d",high,low）;
}
size（p,q,r）
int * q,* r;
char p[];
{
int i=0;
while（p[i]> ='a' &&p[i]< ='z'）
* r + =1;
else（p[i] > ='A' & & p[i]< ='Z'）
* q + =1;
i + +;
}
}
```

6.3.4 利用函数返回值传递数据

一般来说，函数被调用时都有一个返回值，这也是数据传递的一种方式。函数的返回值是通过 return **语句来实现的。**return **语句的一般格式为：**

```
return［（表达式）］；
```

其中表达式可以是常量、变量、函数引用、数组元素、地址常量和其他形式的表达式等。没有函数返回值时，（表达式）可以省略，而写为：

```
return；
```

一个函数可以有多个 return **语句，即多个出口点。从结构化程序设计的角度来讲，一般不提倡多个出口点。**return **语句的功能有：**

- **如果函数有返回值时，用该语句实现返回值。**

■ 终止函数执行，使控制返回调用者函数。

如果函数没有返回值时，该语句可以省略。在省略 return 语句的情况下，函数执行到最后一个右花括号时结束执行，并返回到调用者函数。例如，符号函数：

```
sign（x）
float x；
{
    if（x>0）
      return（1）；
  else if（x= =0）
      return（0）;;
 else
     return（-1）；
}
```

该函数有三个出口点。当 x＞0 时，返回值为 1；当 x=0 时，返回值为 0；当 x<0 时，返回值为-1。又如，计算 a 的 n 次方函数：

```
double power（a，n）
double x；
int n
 {
   double temp；
   for（temp=1；n>0；- -n）
      gemp * =x；
    return（temp）；
}
```

6.3.5 利用全局变量传递数据

全局变量对于程序中的所有函数都是可见的，因此可以利用它来实现函数间的通讯。利用全局变量进行函数间的通讯，不但简单，而且程序的运行效率高。但是，如果函数间使用过多的全局变量，就增加了函数间的联系，降低了函数的独立性。而且，由于各个函数都可以对全局变量进行操作，于是空间出错，而且，出错后比较难确定其错误发生的位置。所以过多的全局变量不易调试和维护。因此，程序中要有节制地使用全局变量。除了大多数函数都使用的数据用全局变量外，一般都用参数传递。下面是使用全局变量传递参数的例子；

```
int sum;
main（）
{
int i,j;
printf（"Please enter i and j:"）；
scanf（" % d % d",&i,&j）；
plus（i,j）；
printf（" \ n i + j = % d",sum）；
}
plus（x,y）
int x,y;
{
sum = x + y;
}
```

此例中使用了一个全局变量 sum，它用于存放两数之和。它把和从函数 plus（）中传递到函数 main（）。其

某次运行结果如下：

```
Please enter i and j：9 12＜CR＞
i+j=21
```

上面讨论了函数之间的通讯方法。但是，在设计一个通用函数（即能够被用在各种情况下的函数，或大家共同使用的函数）时，不应该把函数建立在全局变量上，即不应该使用全局变量。函数所需要的数据都应该用参数传递。使用参数传递，除了有助于函数的能用性之外，还能提高函数代码的可读性，减少函数因副作用带来的错误。

6.4 数组与函数

我们知道数组是具有相同数据类型数据的有序集合。如何把数组从调用函数传递给被调用函数，并在被调用函数中进行加工处理呢？按数据复制的方式传递显然不合理。因为要把数组的各个元素传递给被调用函数，就应该在被调用函数的定义中设置与数组元素个数相同的形式参数。如果数组元素很多时，形参的个数也就很多。这样既不便使用，执行效率又低。因此，一般都不采用这种方法传递。比较合理的方法有：

- 在调用函数中，将数组的首地址作为实参来引用函数；而在被调用函数的定义中，用指针型形参来接收它。这样，一旦把数组的首地址复制给指针变量，被调用函数就可以利用该指针来处理该数组了。
- 在调用函数中，将数组首地址作为实参数来调用函数，而在被调用函数的定义中，用数组名作形参，数组的定义只写“数组名 []”，而不写元素个数。利用该数组名接收传来的数组首地址。于是利用形参中的数组名就可处理传递来的数组的诸元素了。

实际上不论被调用函数定义中的形参是指针或数组，实质都是一样的，是等价的两种表示形式。下面是几个传递数组的例子：

【例 6-6】编写程序，求 100 名职工的平均薪水。

```
float average（aray）                    / * 定义函数，数组名作为形参* /
float array[100]                        / * 形参说明 * /
{
int i;
float aver,sum=array[0];
for（i=1;i<100;i+ +）                   / * 求薪水总和 * /
sum=sum + array[i];
aver=sum/100                          / * 求平均值 * /
return（aver）;                         / * 返回所求的平均值 * /
}
main（）
{
float salary[100],aver;
int i;
printf（"Please enter 100 salaries: \ n"）;
for（i=0;i<100;i+ +）
scanf（"% f",& salary[i]）;                    / * 输入职工薪水 * /
printf（" \ n"）;
aver=average（salary）;                          / * 调用平均值函数 * /
printf（"average salary is % 6.3f",aver）;        / * 输出结果 * /
}
```

在程序中，array（）为形参数组，salary（）为实参数组，salary（）内存有 100 个职工的薪水。

【例 6-7】编制程序，求矩阵乘积函数。

```
matrixmul（x,y,z,m,k,n）
int x[][3],y[][5],z[][5],m,k,n;
{
int i,j,l;
for（i=0;i<m;i+ +）                    / * 矩阵 x 为 m 行 k 列 * /
   for（j = 0;j < n;j + +）             /* 矩阵 y 为 k 行 n 列 * /
   {
z[i][j] =0;                          / * 矩阵 z 为 m 行 n 列 * /
for（1 =0;1 < k;1 + +）
   z[i][j] + =x[i][l] * y[l][j];        / * 矩阵相乘 * /
}
}
main（）
{
int a[5][3],b[3][5],c[5][5];
int i ,j;
printf（" \ nPlease enter matrix a: \ n"）;
for（i =0;i <5;i + +）                  / * 初始化矩阵 a * /
   for（i =0;i <3;i + +）
scanf（" % d",& a[i][j]）;
printf（" \ n Please enter matrix b:\ n"）;
   for（i =0;i <3;i + +）                  / * 初始化矩阵 b * /
    for（i =0;i <5;i + +）
scanf（" % d",& b[i][j]）;
matrixmul（a,b,c,5,3,5）;
printf（" \ n The product matrix of a and b: \ n"）;
for（i =0;i <5;i + +）                  / * 输入矩阵 c * /
{for（j =0;i <5;i + +）
printf（" % d",c[i][j]）;
printf（" \ n"）;
}
}
```

程序某次运行结果为：

```
Please enter matrix a：
274＜CR＞
383＜CR＞
421＜CR＞
590＜CR＞
657＜CR＞
Please enter matrix b：
17897＜CR＞
24316＜CR＞
57422＜CR＞
The product matrix of a and b：
36 70 53 33 64
34 74 60 41 75
```

```
13 43 42 40 42
23 71 67 54 89
51 111 91 73 86
```

【例 6-8】编写程序，比较数组的大小，假设有数组 A 和 B，若 A 数组中的元素大于 B 数组中相应元素的数目，则认为 A 数组大于 B 数组，如 a［i］>b［i］8 次，b［i］> a［i］3 次，则 A 数组大于 B 数组。

```
main（）
{
int a[10],b[10],i,n=0,m=0,k=0;
printf（"enter array a: \ n"）;
for（i=0;i<10;i + +）
scanf（" % d",&a[i]）;                    / * 输入数组 A * /
printf（" \ n"）;
printf（"enter array b: \ n"）;                    / * 输出数组 A * /
for（i=0;i<10;i + +）
scanf（"% d",& b[i]）;                    / * 进行数组元素大小的比较 * /
printf（" \ n"）;
for（i=0;i<10;i + +）
{
  if（large（a[i],b[i]）= =1）n =n + 1;          / * 数组 A 大于数组 B * /
else if（large（a[i],b[i]）= =0）m = m + 1;      / * 数组 A 和 B 相等 * /
k = k +1;                                  / * 数组 A 小于数组 B * /
}
printf（"a[i]>b[i]%d times \na[i]=b[i]%times\na[i]<b[i]%d time\n",n,m,k）;
/ * 输出结果 * /
if（n>k）printf（"array a is large than array b \ n"）;
else if（n<k）printf（"array a is small than array b \ n"）;
else
printf（"array a is equal to array b \ n"）;
}
large（x,y）                                  / * 数组比较函数 * /
int x,y;
{
int flag;                                  / * 标记变量 * /
if（x>y）flag=1; / * A >B,flag =1 * /
else if（x<y）flag =-1; / * A<B,flag=-1 * /
else flag =0; / * A=B,flag =1 * /
return （flag）;
}
```

6.5 字符串和函数

在 C 语言程序中，经常遇到字符串处理。如何将字符串传递给函数，进而对其处理呢？其方法类似于对数组的处理。因为 C 语言是使用字符数组来处理字符串的，所以处理字符串的函数与处理数组的函数在本质上是相同的。字符串传递给函数时，一般用地址复制方式把字符串的首地址传递给函数，此时函数的形参应是字符指针或字符数组方式。下边用几个例子说明函数间如何传递字符串。

【例 6-9】在字符组中存有一组字符串，将字符串排序，并按 ASCII 的顺序输出字符串。

```
main（）
{
static char * weekday[] ={"Sunday","Monday","Tuesday","Wednesday",
"Thursday","Friday","Seturday"};
        / * 初始化字符型指针数组 * /
int i;
sortstr（weekday,7）;                    / * 调用排序函数 * /
for（i=0,i<7;i+ +）
printf（"% s \ n",weekday[i]）;      / * 输出结果 * /
}
sortstr（p,n）                                / * 排序函数 * /
char * p[];
int n;
{
int i,j,gap;                          / * 定义整型变量 * /
char * temp;                          / * 定义字符型指针变量 * /
  for（gap= n/2;gap>0;gap / =2）        / * 排序过程 * /
  for（i=gap;i<n;i + +）
for（j=i-gap;j >=0;j- =gap）
{
  if（strcmp（p[j],p[j +gap]）< =0）
break;
  temp = p[j];
  p[j]=p[j+gap];
  p[j+gap]=temp;
}
}
```

此函数的运行结果如下：

```
Friday
Monday
Saturday
Sunday
Thursday
Wednesday
```

在该程序中，sortstr（）**函数的功能是把多个字符串按字典排序规则由小到大排序。形式参数** p **作为字符指针数组使用。在主函数中，函数调用语句的实参是字符串指针数组的首地址。**

【例 6-10】编写程序，求字符串长度，形参用指针。

```
stringlenth（str）                    / * 求字符串长度的函数 * /
char * str
{
int i;
for（i=0; * str ! = ' \ 0 ';s + +）
  i + +
return（i）;
}
```

```
main（）
{
char s[100];
int lenth;
printf（" \ nPlease enter a string:"）；
scanf（" % s",s）；
lenth = stringlenth（s）；
printf（" \ n The lenth of the first string is % d",lenth）；
lenth=stringlenth（"This is a c program."）；
printf（" \ n The lenth of the second string is % d",lenth）；
}
```

此例中，第一次调用 stringlenth(s)，**实参是字符数组的首地址，第二次调用** stringlenth("This is a c program.")，**实参是字符串首地址。被调用函数的形参是指针。在调用时把字符串首地址传递给形参，于是被调用函数就可对其处理。**

6.6 指针型函数

前面几节主要讨论了参数为变量、数组、字符串时函数的定义和引用等问题。本节从返回值方面讨论函数的性质。

6.6.1 指针型函数的定义和引用

在前面的讨论中，我们已经讲过，有返回值的函数每次调用时都返回具有某种数据类型的值。其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时，则称此函数是指针型函数。指针型函数定义一般格式如下：

```
［存储类型］［数据类型标识符］ * 函数名（［形参表］）
［形参说明；］
{
内部数据说明语句；
执行语句；
}
```

其中：

存储类型：是指该函数本身的存储类型，与一般函数一样，有外部和 static **两种。**

指针型函数返回的地址量可以是变量的地址、数组的首地址，指针变量或后面将要讲的结构、联合等结构的首地址。但不能是其内部定义的自动量或寄存器变量的地址。应该是外部量或静态量的地址。因为自动变量或寄存器变量在定义它的函数执行结束时，分配给它的存储空间已释放，从而其地址也就不存在了，所以不能返回这些量的地址。指针型函数的引用应注意如下几点：

- **同其他函数一样，指针型函数在引用之前应先定义或说明，什么条件下应该说明，以及在什么位置说明相同与其他函数。说明的格式如下：**

  ```
  ［存储类型］［数据类型标识符］*函数名（）；
  ```

- **在调用指针型函数时，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，不能进行赋值。**

6.6.2 指针型函数的应用举例

下面是两个指针型函数的应用实例，从中可以看出如何定义和引用指针型函数。

【例 6-11**】编写用二分法查找字符串的函数。**

所谓二分法就是将要查找的对象由小到大排序构成一个有序表。查找时，首先与表中的中间对象进行比较；若相等，则查找，若要查找的对象小于表中的对象，则取表中值较小的一半继续查找。若要查找的对象大于表中的对象，则取表中值较大的一半继续查找。如此继续下去，直至查到或表被查完。每次查找总是选中间元素进行比较。

假设 s 是指向要查找的字符串指针，sp 是指向被查找的若干字符串的指针数组，n 是字符串个数。查到时，返回查到的字符串的首地址，和字符串在表中的序号，没查到时返回 0。该函数的定义如下：

```
char * bin（sp,str,n,addr）
char * sp[],* str;
int n, * addr;
{
int left,right,mid;
left = 0;
right = n-1;
while（left< =right）
{
mid=（left+right）/2;
if（strcmp（str,sp[mid]）<0）
   right = mid -1;
else if（strcmp（str,sp[mid]）>0）
   left = mid + 1;
else
{
   return（sp[mid]）;
   * addr=mid;
}
}
return（0）;
}
```

【例 6-12】从键盘输入一个数 n，打印第 n 月份的英文名字。若输入错误，打印"illegal month"。

```
char * mooth_name（n）                    / *  定义字符型指针数组  * /
int n;
{
static char * name[]={"illegal month","January","February","March","April","May","Jun
e","July","August","September","October","November","December"
};
return（（n<1|| n > 12）? name[0]:name[n]）;
main（）
{
int n;
printf（" \ nPlease enter n:"）;
acanf（" % d",& n）;
printf（" \ nMonth No.% d- > % s \ n",n,month_name（n）;
}
```

某三次程序运行结果：

```
Please enter n：2＜CR＞
Month No.2->Febrary
```

```
Please enter n：8＜CR＞
Month No.8->August
Please enter n：13＜CR＞
Month No.13->illegal month
```

该程序调用一次 month_name（n），返回一个指向相应月的英文名的指针。month_name（）函数内只包含一个可执行语句（return），内含一个三项表达式，根据不同的条件返回 name［0］必须是 static 存储类型，如果是自动型，则调用函数中就无法用它输出。如果在主函数中变量接收返回值，则主函数应该改为如下形式：

```
main（）
{
int n;
char * p;
printf（" \ n Please enter n:"）;
scanf（" % d", &n）;
p =month_name（n）
printf（"Month No.%d- >% s \ n",n,p）;
}
```

6.7 指向函数的指针

6.7.1 函数指针的概念

在前面的数组章节中，告诉我们数组名是地址常量，它代表数组的首地址。如果把数组的首地址赋给一个指针的话，则该指针指向这个数组。

同样，C 语言中的函数名表示函数的首地址，即函数执行时的入口地址。例如，程序中定义了如下的函数：

```
int example（n）
int n；
 {
…
}
```

其中函数名 example（）就是该函数的入口点，当用函数名 example（）调用该函数时，控制就被传给该函数。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该函数的入口地址。这时称该指针是指向这个函数的指针，简称函数指针。所以，函数指针就是指向函数的指针。函数指针的定义说明形式如下：

［存储类型］［数据类型标识符］（*函数指针名）（）；

例如：

```
static int（* example）（）；
```

其中：存储类型是函数指针本身的存储类型。数据类型标识符表示指针所指向的函数的数据类型。

函数指针与数据指针在性质上是相同的，例如，它们所饮食的内容都是地址常量，程序中不能使用不定向的指针等。但是，它们也有区别，其主要区别是：

（1）对于数据指针，其访问目标的运算符“＊”是访问数据，例如：

```
* sp=j
i=* sp
```

对于有一定指向的函数指针，其访问目标的运算符（*）就是把控制传给该指针所指向的函数，使该函数的函数体被执行。数据指针指向的数据存储区，而函数指针指向的是程序代码存储区。

（2）函数指针与数据指针在定义或说明形式上不同，例如：

函数指针：int（* example）（）

数据指针：int * sp；

其中包围函数指针名的圆括号是绝对不能省略的。同时，它与指针型函数也不同，例如：

int * example（）；

它是指针型函数，而不是函数指针，它表示该函数的返回值是指向数据类型为 int **的指针。**

6.7.2 函数指针的应用

函数指针的主要用途是在函数之间实现函数传递。函数传递实质上是函数地址的传递。在函数调用中，如果想在调用函数中把一个函数传递给被调用函数，只需把要传递的函数名作为实参即可。被调用函数的形参是接收函数入口地址的函数型指针。指针函数的指针可以是数组，在函数指针数组中，每个数组元素都是指向函数的指针。下面是函数指针应用的实例。

【例 6-13**】编制函数，实现加、减、乘和除。**

```
main（）
{
int add（）,sub（）,mul（）,vid（）;
int（ * function[4]） （）;
int x,y,i;
static char c[] ={' + ', ' - ',' * ',' / ' };
function[0] = add;
function[1] = sub;
function[2] = mul;
function[3] = vid;
x = 10;
y = 5;
for（i =0;i <4;i + +）
printf（ "% d % c % d = % d \ n",x,c[i],y,execute（x,y,runction[i]））;
}
execute（x,y,func）
int x,y,（ * func） （）;
{
  return（ （ * func）（x,y） ）;
}
add（x,y）                          / * 加函数 * /
int x,y;
{
  return（x + y）;                  / * 返回加的结果 * /
}
sub（x + y）                        / * 减函数 * /
int x,y;
{
return（x - y）;                    / * 返回减函数 * /
}
mul（x + y）                        / * 乘函数 * /
int x,y;
{
  return（x * y）;                  / * 返回乘函数结果 * /
}
```

```
vid（x,y）                    / * 除函数 * /
int x,y;
{
   return（x/y）;             / * 返回除函数结果 * /
}
```

该程序实现 x+y、x-y、x*y、x/y。这四个操作中的任何一个的实现都是通过调用函数 execute（），由它再去执行函数指针数组所指向的函数。function［］是一个函数指针数组，其中 function［0］是指向 add（）的函数指针，function［1］是指向 sub（）的函数指针，function［2］是指向 mul（）的函数指针，function［3］是指向 vid（）的函数指针。该程序的执行结果如下：

```
10+5=15
10-5=5
10*5=50
10/5=2
```

【例 6-14】在函数间传递函数。

```
check（p1,p2,compare）
char   * p1, * p2;
int（ * compare）（）;
{
if（ ！ （ * compare）（p1,p2） ）
   printf（" \ nThe two strings are equal !"）;
else
   printf（" \ nThe two strings are not equal !"）;
}
main（）
{
int strcmp（）;
char strl[100],str2[100];
printf（" \ nPlease enter string str1:"）;
scanf（" % s",str1）;
printf（" \ nPlease enter string str2:"）;
scanf（" % s",str2）;
check（str1,str2,strcmp）;
}
```

在 main（）函数的“check（s1，s2，strcmp）;”语句中，strcmp 是函数名，它要被传递给 check（）函数。在 check（）函数的定义中，型参 compare 是一个函数型指针，它接收“check（s1，s2，strcmp）;”语句中传来的函数 strcmp（）。if 语句中的（* compare）（p，q）返回 0，则 p 和 q 所指的字符串相等。当一个函数在不同的调用中，要求它引用不同的函数时，使用函数传递比较方便灵活。

【例 6-15】检查两个输入数据或字符串是否相等。

我们定义一个函数，来实现检查字符串相等或数相等。这时用传递函数比较方便。

```
# include < stdio.h >
check（p1,p2,compare）                    / * 字符串比较函数 * /
char   * p1, * p2;
int（ * compare）（）;
int（ * compare）（）;
{
if（！ （ *compare）（p1,p2）
```

```
    printf（" \ nThe two strings are equal!"）；
else
    printf（" \ nThe two strings are not equal!"）；
}
numcmp（a,b）                                    / * 数比较函数 */
char * a, * b;
{
if（ atoi（a）= =atoi（b） ）                    / * stoi（）为字符转数字函数 * /
    return（0）；
else
    return（1）；
}
main（）
{
int strcmp（）,numcmp（）；
char strl[100],str2[100];
printf（" \ nPlease enter two strings or numbers:"）；    / * 提示输入两个字符串* /
scanf（" % s",str1）；                                   / * 输入字符串 * /
scanf（" % s",str2）；
printf（" \ nPlease enter ' n ' for number compare or enter ' e ' for string compare:"）；
if （getchar（） = =' n '）                                / * 数比较 * /
    check（str1,str2,numcmp）；                          / * 调用数比较函数 * /
else
    check（str1,str2,strcmp）；                          / * 调用字符串比较函数 * /
}
```

6.8 递归函数与递归程序设计

6.8.1 递归函数的概念

在 C 语言的函数定义中，如果出现直接或间接的调用自己，则称该函数为递归定义函数，或称递归函数。函数调用中的直接或间接调用自己称为递归调用。例如：下边函数的定义就出现了自己调用自己，这种调用是直接调用。

```
int f（n）
int n
 {
int i，j；
…
j=f（i）
…
}
```

下边函数的定义中出现间接调用自己，a（）的定义内部调用了 b（），而 b（）的定义内部又调用了 a（）：

```
int a（n）
int n；
 {
```

```
int d，c，b（）；
…
d=b（c）
…
int b（m）
int m
 {
int e，i；
…
e=a（i）；
…
}
```

以上定义的函数也是递归函数。

递归函数要注意容易造成死循环，一般要有根据某一条件是否成立来决定是否停止继续调用。递归函数的典型例子是计算 n!，前面已讲过 n!的程序实例，这里讨论如何用递归函数实现 n!。

我们知道，数学中计算 n!的公式如下：

```
n!=1*2*3…* n
```

在递归算法中 n!是通过如下两个公式计算的：

```
1!=1
n!=n（n-1）！    （当 n>1 时）
```

例如，求 4!的递归过程如下：

```
4!=4 * 3!
3!=3 * 2!
2!=2* 1!
1!=1
```

按上述相反的过程回溯计算，就得到 4!的计算结果：

```
1!=1
2!=2 * 1!=2
3!=3 * 2!=6
4!=4 * 3!=24
```

按上面给出的递归算法来构造函数时，就得到了 n!的递归函数。下面是用递归函数编制的计算 n!的程序：

```
int fact（n）
int n；
 {
if（n<=1）
return（1）；
else
return（n * fact（n-1））；
}
main（）
 {
int n；
int facto；
printf（" \ n Please enter n："）；
scanf（" % d "，& n）；
facto=fact（n）；
```

```
printf（" % d!=% d"，n，facto）；
}
```

这里的 fact（）函数就是一个递归函数。n!函数的非递归写法如下：

```
fact（n）
int n；
 {
int s；
for（s=1；n>1；n- -）
s *=n
retun（s）；
}
```

6.8.2 递归程序设计

利用函数的递归特性，我们可以实现递归程序设计。递归函数是在函数定义内部包含自调用结构的函数。于是递归程序就是包含递归函数的程序。要实现递归程序设计，首先要分析问题是否具有递归处理过程，而且要弄清该递归过程是什么，这样就可以很容易编写递归函数。递归程序设计可以使程序结构变得非常简单、明了、清晰。但是，递归程序的最大缺点是效率低，这是因为：

（1）递归函数会带来大量的重复计算，例如 n!程序，在不递归的算法中，没有重复计算；而在递归程序中，函数的每一次递归调用都会带来前面的重复计算，这无疑会大大降低效率。

（2）由于函数的每一次调用都需保留现场，传递参数以及恢复现场等操作。而大量反复地递归调用显然会降低效率。

下面我们通过一些实例来说明如何进行递归程序设计。

【例 6-16】用递归算法编写计算 x 的 n 次方的程序。

分析：从数学运算中知道：

```
f（x，n）表示 x 的 n 次方：
f（x，n）=1                          当 n=0 时
f（x，n）=（x * x）的 y/2 次方        当 n 为偶数时
f（x，n）=x 乘以 x 的（n-1）次方      当 n 为奇数时
```

即：

```
f（x，n）=1                          当 n=0 时
f（x，n）=f（x * x，n/2）             当 n 为偶数时
f（x，n）=x * f（x，n-1）             当 n 为奇数时
```

例如：

```
f（4，3）=4 * f（4，2）
        = 4 * f（4 * 4，2/2）
        = 4 * f（16，1）
        = 4 * 16
        = 64
```

其程序如下：

```
f（x,n）
int x,n;
{
if（n = = 0）
   return（1）;
else if（n % 2 = = 0）
   return（f x * x,n / 2））;
```

```
else
retrun（x * f（x,n -1））；
}
```

【例 6-17**】用递归算法编写程序，求** x **和** y **的最大公约数。**

算法分析：设 gcd（n，m）**表示计算** x **和** y **的最大公约数。其递归计算如下：**

gcd（n，m）=m **当** n=0 **时**

gcd（n，m）=gcd（n，m-n）**当** n<=m **时**

gcd（n，m）= gcd（m，n）**当** n>m **时**

例如，求 gcd（8，4）**：**

gcd（8，4）= gcd（4，8）

gcd（4，8）= gcd（4，8-4）= gcd（4，4）

gcd（4，4）= gcd（4，4-4）= gcd（4，0）

gcd（4，0）= gcd（0，4）=4

其函数如下：

```
ged（n,m）
int ,n,m;
{
if（n = = 0）
  return（m）;
else if  （m < n）
  return（ged（m,n）;）
else
  return（ged（n,m - n）;）
}
```

6.9 命令行参数

在操作系统的命令中，有很多都是含有参数的命令，这些命令可以通过带参数的 main **函数结合指针的应用来实现。在使用操作系统的命令参数时，常常要写出命令名和其所需的参数。例如，在** MS-DOS **操作系统下，常用如下一些命令：**

```
del filename＜CR＞
ren oldname   newname＜CR＞
copy filename1 filename2＜CR＞
…
```

其中 copy、ren **和** del **等都是命令名，通常是可执行程序的文件名。而命令名后的字符串是其所需要的参数。当这些命令发出后，就会根据命令行中的参数来进行处理。例如，当执行** copy **命令时，就会根据命令中给定的参数来进行处理，即把** filename1 **文件中的内容拷贝到文件** filename2 **中。那么如何将参数传递给相应的程序呢？这就是本节要阐述的问题。**

在 C **语言的程序中，将命令行上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来实现的。其格式如下：**

```
main（argc，argv）
int argc；
 {
…
使用传递的参数
```

…

}

其中：

argc：**指出命令行中命令名和参数的总个数。如上边** copy **和** ren **二命令中，**argc **均为** 3**。而** del **命令中** argc **为** 2**。**

argv［］：**是一个指针数组，各个元素所指向的目标如下：**

argv［0］：**命令名**

argv［1］：**第一个参数**

argv［2］：**第二个参数**

…

例如有一个 C **语言的程序，该程序的执行文件名为** test**，其命令行格式为：**

test sourcepro1.c namelist.txt

由于它有两个参数，再加上命令名，故在程序运行时，argc **被初始化为** 3**。而** argv ［］**进行如下的初始化：**

argv［0］="test"；

argv［1］="sourcepro1.c"；

argv［2］="namelist.txt"；

因此，argc **的值和** argv ［］**的元素个数取决于参数个数。于是在** C **语言中就可以使用** argc **和** argc ［］**来接收和处理命令行参数。**

【例 6-18**】请编写一个程序，实现其命令行参数的输出。**

```
# include < stdio.h >
main（argc,argv）
int argc;
char * argv[];
{
int I;
printf（"argc = % d \ n",arge）;
printf（"Command name:% s \ n",argv[0]）;
for（i =1;i<argc;i + +）
   printf（"Argument No.% d:% s \ n",i,argv[i]）;
}
```

假设该程序的执行文件名为 echo**，则执行它的某次结果如下：**

```
c：\ >echo welcome world peace＜CR＞
argc=4
command name：echo
Argument No.1：welcome
Argument No.2：world
Argument No.3：peace
```

本程序也可以通过如下的方式进行编程：

```
main（argc，argv）                    / *  分行输出命令参数  * /
int argc；                            / *  命令行中参数的总和* /
char * argv［］ ；                     / *  存放命令行参数的字符型指针数组*/
 {while（argc>1）                     / *  如果 argc>0 时，那么，输出时，将把
                                      命令字符串也输出来* /
 {+ + argv；                          / *  第一个参数是程序名，跳过不取* /
printf（" % s \ n"，* argv）；
- - argc；
```

```
}
}
```

6.10 综合应用实例

【例 6-19】输入 10 个整数，将其中最小的数与第一个数对换，把最大的数与最后一个数对换。写三个函数：①输入 10 个数；②进行处理；③输出 10 个数。

```
main（）/ * 将 10 个数中最小数与第一个对换，最大数与最后一个对换.* /
{
int number[10];
input （number）;/ * 调用输入 10 个数的函数 * /
max_min_value（number）;/ * 调用交换函数 * /
output （number）;/ * 调用输出函数 * /
}
input（number）/ * 输入 10 个数的函数 * /
int number[10];
{
int i;
printf（"请以，号间隔输入 10 个数： "）；
for（i=0;i<9;i++）
scanf（"%d,",&number[i]）;
scanf（"%d",&number[9]）;
}
max_min_value（array） / * 交换函数 * /
int array[10];
{int * max, * min;
int *p, *array-end=array+10;
max=min=array;
for（p=array+1;p<array-end;p++）
if（* p> * max） max=p;/ * 将大数地址赋给 max * /
else if（ * p< * min）min=p;/ * 将小数地址赋给 min * /
* p=array[0];array[0]= * min; * min= * p;/ *将最小数与第一个数交换 * /
* p=array[9];array[9]= * max; * max= * p;/ * 将最大数与最后一个数交换 * /
return;
}
output（array）/ * 输出函数 * /
int array[10];
{
int * p;
printf（"交换后 10 个数为："）
for（p=array;p<array+9;p++）
printf（"%d,", * p）;
printf（"%d\n",array[9]）;
}
```

运行结果：

请以，号间隔输入 10 个数：32，24，56，78，1，98，36，44，29，6↙

交换后 10 个数为：1，24，56，78，32，6，36，44，29，98

【例 6-20】将一个 5×5 的矩阵中最大的元素放在中心，四个角分别放四个最小的元素（顺序为从左到右，从上到下顺序依次从小到大存放），写一个函数实现之。用 main 函数调用。

```
main（）/ * 有序排列 5×5 矩阵（中心大，四角小） * /
{
int a[5][5], * p,i;
printf（"请按行输入 5×5 矩阵：\n"）;
for（i=0;i<5;i++）
scanf（"%d %d %d %d %d",&a[i][0],&a[i][1],&a[i][2],&a[i][3],&a[i][4]）;
p=&a[0][0];
change（p）;
printf（"交换后的矩阵为：\n"）;
for（i=0;i<5;i++）
printf（"%d %d %d %d %d\n",a[i][0],a[i][1],a[i][2],a[i][3],a[i][4]）;
}
change（p）/ * 交换函数 * /
int i,j,change;
int * pmax, * pmin;
pmax=p;
pmin=p;
for（i=0;i<5,i++）/ * 找最大最小值地址，并赋给 pmax,pmin * /
for（j=0;j<5;j++）
{
if（ * pmax< * （p+5 * i+j））pmax=p+5 * i+j;
if（ * pmin> * （p+5 + i+j））pmin=p+5 * i+j;
}
change= * （p+12）;/ * 将最大值换给中心元素 * /
* （p+12）= * pmax;
* pmax=change;
change= * p;/ *将最小值换给左上角元素 * /
* p= * pmin;
* pmin=change;

pmin=p+1;
for（i=0;i<5;i++）/ * 找第二个最小值地址赋给 pmin * /
for（j=0;i<5;i++）
if（（（p+5 * i+j）!=p）&&（ * pmin> * （p+5 * i+j））） pmin=p+5 * i+j;
change= * pmin;/ * 将第二个最小值换给右上角元素 * /
 * pmin= * （p+4）;
 * （p+4）=change;
pmin=p+1;
for（i=0;i<5;i++）/ * 找第三个最小值地址赋给 pmin * /
for（j=0;j<5;j++）
if（（（p+5 * i+j）!=（p+4））&&（（p+5 * i+j）!=p）&&（ * pmin> * （kp+5 * i+j））） pmin=p+5 * i+j;
change= * pmin;/ * 将第三个最小值换给左下角元素 * /
```

```
 * pmin= * （p+20）;
 * （p+20）=change;
pmin=p+1;
for（i=0;i<5;i++）/ * 找第四个最小值地址赋给 pmin * /
for（j=0;j<5;j++）
if（（（p+5 * i+j）!=p）&&（（p+5 * i+j）!=（p+4））&&（（p+5 * i+j）!=（p+20））&&（ * pmin> * （p+5 * i+j））））
pmin=p+5 * i+j;
change= * pmin;/ * 将第四个最小值换给右下角元素 * /
 * pmin = * （p+24）;
 * （p+24）=change;
}
```

运行结果：

```
请按行输入 5×5 矩：
35  34  33  32  31↙
30  29  28  27  26↙
25  24  23  22  21↙
20  19  18  17  16↙
15  14  13  12  11↙
```

交换后的矩阵为：

```
11  34  33  32  12
30  29  28  27  26
26  24  35  22  21
20  19  18  17  16
13  23  15  31  14
```

6.11 小 结

（1）C 语言程序是由一个或多个函数所组成，其中至少有一个主函数。每个函数都可看作是相互独立、功能单一的模块。因此，C 语言可以看作是一种模块化程序设计语言。C 语言程序中所包括的函数除主函数外，还有其他函数。其他函数有用户定义函数和标准库函数。标准库函数是由系统提供的。

（2）C 语言函数一般都有返回值并具有某种数据类型。如果函数的返回值是数值型，则其数据类型可以是有符号或无符号的 char、chort、int 或 long，或者是 float、double。若返回值是地址，则其数据类型是指针型。如果无返回值，则其数据类型是 void 型。

（3）函数定义的一般格式为：

```
[存储类型] [数据类型标识符] 函数名（[形式参数表]）
[形参说明；]
{
[内部数据说明语句；]
可执行语句；
}
```

需要注意的是函数定义不准许嵌套。

（4）函数有外部函数和静态函数。外部函数的作用范围是整个程序。静态函数的作用范围是其定义所在的源文件，即局部于源文件，因此也把静态函数称作内部函数。定义静态函数的目的是使其对其他源文件不可见。

（5）函数引用的一般格式是：

```
函数名（[实参表]）
```

实参表与其对应的形参表在个数、类型和顺序上应一致。

在一个源文件中，如果被引用的函数是在引用点之前定义，则不用另行说明即可引用。如果是在引用点之后定义或在另一个源文件中定义，则必须先说明后引用。说明的一般格式为：

［存储类型］［数据类型标识符］函数名（）；

函数说明的目的是告诉编译程序该函数的返回值是什么数据类型，以便在编译时检查函数的调用是否正确。对于静态函数来说，不能在非定义它的源文件中引用它。

（6）C 程序的执行顺序是：

- 在程序启动时，控制由操作系统传递给该程序的 main（）函数，main（）函数开始执行。
- 在 main（）函数的执行过程中，如果遇到调用其他函数，则控制从 main（）函数传递给其他函数。其他函数执行结束，控制又返回 main（）函数。在其他函数执行过程中又可调用另外的函数，…这叫做函数嵌套调用。C 语言就是通过函数的调用或嵌套调用来实现其功能的。
- 当 main（）函数执行结束时，控制才从 main（）函数返回操作系统，从而结束程序的执行。

（7）C 语言的变量有如下四种存储类型，它们分别是：auto 类型、register 类型、static 类型和外部类型，如果变量的存储类型不同，则其作用域和生命期也不同，变量定义或说明的完整格式如下：

［存储类型］［数据类型标识符］变量表；

（8）函数间的通讯可以通过如下 4 种方式实现：

- 参数传递方式：有数据复制方式和地址复制方式
- 函数返回值
- 全局变量
- 文件：例如可在一个函数内写文件，而在另一个函数中读文件，从而实现函数间的通讯。

（9）函数中传递数组的方法是：在被调用函数的定义中，将接收数组的形参定义为指针型，或将形参指定为数组名，并在形参说明中，按如下格式说明：

［数据类型标识符］数组名［］；

而在调用该函数时，将数组名作为实参即可。

（10）指针型函数就是其返回值为地址量的函数，其定义的一般格式为：

```
［存储类型］［数据类型标识符］*函数名（［形参表］）
［形参说明；］
{
内部数据说明语句；
执行语句；
}
```

（11）函数指针就是指向函数的指针，其定义或说明的格式是：

［存储类型］［数据类型标识符］（*函数指针名）（）；

读者要区别函数指针和指针型函数。

（12）递归函数就是在函数定义的内部又直接或间接地引用其自身。如果是直接引用自身，则称该函数为直接递归函数，如果是间接引用自身，则称其为间接递归函数。

习　题

1．编写两个函数，分别求两个整数的最大公约数和最小公倍数。

2．编写判断一个量是否为素数的函数。如果有素数，则返回 1，否则返回 0。

3．函数 getch（ ）实现从 buf []中读指针当前所指向的字符。另一函数 ungetch （ ）实现把刚读的字符退回 buf[]缓冲区内。

4．分别用递归和非递归算法编写求 1 + 2! + 3! + … + n!的程序。

5．编写从字符串 str 中删除字符 c 的函数。如果 str 中存在字符 c，则在完成删除后还返回 c 在串中的位置，如果不存在，则返回 0。

6．编写字符串中的字符替换函数 replacechar（str,c1,c2），即把字符串 str 中的字符 c1 替换为 c2。

7．编写从多个字符串中寻找最长串的函数，并返回该串的首地址。函数形式为：maxlenthstring（s,n）。其中 s 是指向多个字符串的指针数组，n 是字符串的个数。

8．有一个字符串存放在缓冲区 buf[]中，该缓冲区的指针 bufp 指向当前要读的字符。请编写两个函数，一个用递归和非递归算法编写把字符串 s 逆转的函数。

9．用递归算法和非递归算法编写 itoa 函数，即把一个整数转换为字符串。

10．编写一个函数，统计一个字符串中所含字母、数字、空格和其他字符的个数，要求用参数传递被统计的字符串。

11．编写一个函数 func（n）返回 n 的数字和（其中 n 为 1~9999 之间的正整数）。

第七章　结构体、联合和枚举

7.1　结构体的说明和定义

7.1.1　什么是结构体

前面的章节已讲过数组。数组是具有相同数据类型的数据所组成的集合体，它的引入为程序设计者带来了很大的方便。但是，在程序设计中，还经常遇到一些关系密切、数据类型不同的数据。对于这些数据，为了处理方便，经常把它们组织在一起，并为其取一个名字。这类数据在 Pascal 和 COBOL 语言中叫记录，而在 C 语言中叫结构体。

结构体通常是由不同类型的数据所组成的集合体。构成结构体的数据称之为结构体成员（或称结构体元素），每一个成员具有不同的名字和数据类型（特殊情况下，也可具有相同的数据类型）。为了处理方便，每个结构体都有一个名字，而所有的成员都组织在该名字之下。结构体的典型例子是学生的记录，它包括学号、姓名、性别、年龄、地址和电话号码等，这些数据的类型都可以不同。因此，结构体的每一个成员都是通过其名字来引用的，而不像数组是通过下标来引用。

结构体的引入为处理复杂的数据结构体提供了有力的手段，也为函数间传递一组不同数据类型的数据提供了方便。特别是对于数据结构体比较复杂的大型程序提供了方便。

7.1.2　结构体的说明及结构体变量的定义

1．结构体的说明

由于结构体是由不同数据类型的数据所组成的集合体，它包含若干成员。因此，在使用结构体进行数据处理时，首先应对结构体的组成进行描述。这种描述称之为说明。结构体的说明实质上是宣布该结构体是由哪些成员所组成，以及成员的数据类型。

结构体说明的一般格式如下：

```
struct      结构体名
 {
结构体成员表列；
}
```

其中 struct 是保留字，“ struct 结构体名 ”称结构体类型标识符，或称结构体类型名。大括号中的结构体成员表列称结构体体。成员表包含若干成员，每一个成员都具有如下的形式：

```
数据类型标识符  结构体成员名；
```

例如：

```
struct      date
 {
int         year；
int         month；
int         day;
int         yearday；
}  ；
```

其中，date 是结构体名。而 year、month、day、yearday 均是该结构体的成员。

有关结构体说明请注意如下几点：

- 结构体说明描述了结构体的组织形式，但在编译时并不为它分配存储空间。只是规定了一种特定的数据结构体类型及它所占用的存储模型。
- 结构体的成员可以是简单变量、数组、指针、结构体或联合等。
- 结构体说明可以在函数内部，也可以在函数外部。在内部说明的结构体，只在函数内部可见，在外部说明的结构体，从说明点到源文件尾之间的所有函数都可见。
- 结构体可以嵌套使用，即一个结构体也可以成为另一个结构体的成员。
- 结构体成员的名字可以同程序中的其他变量同名，二者不会相混。

2．结构体变量的定义

结构体变量同其他变量一样也必须先定义或说明，然后才能引用。在说明了一个结构体之后，就可定义该结构体的对象了。一个结构体的对象或实例称之为该结构体类型的结构体变量。可以采用如下三种方式定义结构体变量。

（1）在结构体说明的同时定义结构体变量

其一般形式如下：

```
［存储类型］struct 结构体名
 {
结构体成员表列；
} 结构体变量名表列；
```

其中，存储类型表示结构体变量的存储类型（下同）。

例如：

```
struct example
 {
char * name；
int age；
double salary；
char * address；
} guo，zhang，xiao，yang；
```

该例的结构体名为 example，用它定义了 guo、zhang、xiao 和 yang 四个结构体变量，这四个变量具有相同的结构体组成，即具有相同的类型。这种定义方法的特点是：定义一次结构体变量之后，在该定义之后任何位置还可用该结构体类型来定义其他结构体变量。

（2）直接定义结构体变量

其一般形式如下：

```
［存储类型］struct
 {
结构体成员表列；
} 结构体变量名表列；
```

例如：

```
struct
{
char * name;
int age;
double salary;
char * address;
}guo,zhang,xiao,yang;
```

这里定义的四个结构体变量与（1）中定义的完全相同。这种定义方式的特点是：不能在别处用来另行定义别的结构体变量，要想定义就得将“ struct{…} ”这部分重新写。

（3）把定义和说明分开

其定义的一般形式如下：

［存储类型］struct 结构体名 结构体变量表；

例如：

```
struct example
 {
char * name；
int age；
double salary；
char * address；
} ；
```

利用结构体类型“ struct example ”来定义结构体变量 guo、zhang、xiao 和 yang 的形式为：

```
struct example guo，zhang，xiao，yang；
```

这种定义方式的特点是：可把其结构说明部分作为文件存放起来，这样就可借助于“ # include ”语句把它复制到任何源文件中，用以定义同类型的其他结构体变量。

有关结构体变量的定义还需如下几点：

- 结构体变量分配内存，而结构体说明不分配内存。
- 结构体变量一般不用 register 型。
- 结构体变量的定义一定要在结构体说明之后或与结构体说明同时进行，对尚未说明的结构体类型，不能用它来定义结构体变量。
- 结构体变量占用实际内存的大小可用 sizeof（）运算来求出。sizeof（）的一般格式如下：

```
sizeof（运算符）
```

其中，运算量可以是简单变量、数组、结构体或数据类型名。

例如：

```
sizeof（struct example）=15
sizeof（int）=2
```

- 结构体变量中的成员可以单独使用，其地位与一般变量相同。

7.2 结构体成员的引用与结构体变量的初始化

7.2.1 结构体成员的引用

在对结构体进行引用时，一般只能对其成员进行直接操作，而不准许结构体变量整体直接进行操作。

引用有两种方式：

（1）用指针方式

即定义一个指针，使它指向该结构体变量，这时就可用指针和成员名来引用结构体成员了。

（2）用结构体成员运算符方式。

用结构体成员运算符引用结构体成员的一般形式如下：

结构体变量名.成员名

其中“ . ”是结构体成员运算符，其结合性是自左至右。下面是用结构体成员运算符引用结构体成员的例子：

```
struct example
 {
long int idnumber；
char * name；
char address［100］；
```

} guo；

各成员的引用形式如下：

guo.idnumber

guo.name

guo.address **或** guo.address［i］

对于* guo.name **形式，由于运算符“ . ”优先于“ * ”所以，*** guo.name **等价于***（guo.name)。**其含义是访问** guo.name **的目标变量。**

【例 7-1】**将某年某月某日转换成该年的第几天。**

这里我们把与时间有关的变量定义为结构体变量，其程序如下：

```
main()
{
   struct date                          / *  定义结构体  * /
     {
       int year;                        / *  定义结构体成员及其数据类型  * /
int month;
int day;
int yearday;
char * monthname;
}date1;                             / *  定义结构体变量  * /
int leap;
int i;
static int month[2][13] ={
{0,31,28,31,30,31,30,31,31,30,31,30,31},     / *定义静态数组  * /
{0,31,29,31,30,31,30,31,31,30,31,30,31}};
printf(" \ n Please enter year、 month and day:");        / *  提示输入年、月、日  * /
scanf(" % d,% d,% d",& date1.year,&date1.month,&date1.day);    /*提示年、月、日* /
i =date1.year;
leap =0;
if(( i %4= =0)& & (i % 100! =0))leap=1;              / *  判断是否闰年  * /
   if(i%400= =0)leap =1;
   date1.yearday = date1.day;
   for(i=1;i<date1.month;i+ +)
date1.yearday + = month[leap][i];printf("yearday = % d \ n",date1.yearday);
}
```

通过该例，可以看出结构体变量的成员，即成员变量，可以像一般变量一样参与各种操作。

7.2.2 结构体变量的初始化

所谓结构体变量的初始化，就是在定义结构体变量的同时，对其成员变量赋初值。结构体变量初始化的一般形式如下：

struct **结构体名 结构体变量名**= **｛初始数据｝ ；**

例如：

```
struct date
  {
int year；
int month；
int day；
```

```
    int yearday；
    char monthname［3］；
    }；
    struct date date1=｛1979，10，29，"OCT"｝；
```

在对结构体变量初始化时，应注意如下几点：

- 只能对外部和静态结构体变量初始化，不能对 auto 类型的结构体变量初始化。
- 初始化数据之间用逗号（,）隔开。
- 初始化数据的个数要与成员的个数相同。
- 初始化数据的类型要与相应的成员变量的类型一致。

7.3 结构体数组

7.3.1 结构体数组的定义及初始化

1. 结构体数组的定义

结构体数组是其元素都是具有相同结构体变量。同一般数组一样，结构体数组也必须先定义或说明，才能引用。结构体数组定义的一般格式如下：

［存储类型］struct 结构体名 结构体数组名［元素个数］［结构体数组名］［元素个数］，…］；

其中“ struct 结构体名 ”是已定义过的结构体类型。存储类型是结构体数组的存储类型。

2. 结构体数组的初始化

一个外部的或静态的结构体数组在定义的同时可以初始化。其一般格式是在定义之后 紧跟一个用花括号括起来的一组初始数据：

［存储类型］struct 结构体名 结构体数组名［］=｛初始数据｝；

其中“ struct 结构体名 ”是预先说明的结构体类型。

或：

```
［存储类型］struct 结构体名
{
结构体成员表列；
｝结构体成员表［］=｛初始数据｝；
```

例如，初始化前面已说明过的 keytab［］结构体数组：

```
struct key keytab［］=｛｛"shift"，0｝，
                       {"ctrl"，0｝，
                       {"del"，0｝，
                         …
                       {"pause"，0}
                    }；
```

在初始化时，应当使初始数据的个数与结构体数组的元素个数以及每个数组元素的成员个数匹配。为了增强可读性，最好使每一个数组元素的初始数据都用花括号括起来。

7.3.2 结构体数组的应用举例

结构体数组适合于处理一组具有相同结构体类型的数据，下面举例说明其应用。

【例 7-2】统计全年级男、女生人数及 1977 年到 1980 年出生的人数（含 1977 年和 1980 年）。

```
main
{
  struct stud                          / *  定义学生登记表一结构体数组  * /
```

```
    {
char name[30];
char sex;
int year;
    };
    struct stud grad[300];
    int malenumber,femalenumber,count;
    char sex[2],year[5];
    int i;
    countm = countf = count78 = 0;              / *  建立学生名册  * /
for(i =0;i <50;i + +)
{
readline(grade[i].name,sex,year);
grade[i].sex = sex[0];
grade[i].year = atoi(year);
}
for(i=0;i <300;i+ +)              / *  统计男女生人数和 1977 年到 1980 年出生人数* /
{
if((grade[i].year> =1977)& &(gradep[i].year< =1980))
count ++;
if(grade[i].sex = =' m '||grade[i].sex = =' m ')
malenumber + +;
else
femalenumber + ;
}
printf(" \ n male: % d",countm);            / *  输出统计结果* /
printf(" \ n 女生人数：% d",femalenumber);
printf(" \ n1977 年到 1980 年出生的人数：% d",count);
}
readlin(pname,* psex, * pyear)               / *  读记录函数  * /
char * pname, * psex, * pyear;
{
printf(" \ nPlease enter name:");            / *  读入记录的内容  * /
scanf(" % s",pname);
printf(" \ n Please enter sex:");
scanf(" % s",psex);
printf(" \ n Please enter birth year:");
scanf(" % s",pyear);
}
```

【例 7-3】用结构体数组处理通讯录。

```
# include < stdio.h >                    / *  嵌入头文件  * /
# include < stdlib.h >
# define MAXIMUM 20                 / *  宏定义  * /
struct stud
{
char name[30];                      / *  定义结构体  * /
```

```
int age;
char sex;
char telnumber[8];
char address[100];
}
main()
{
struct stud student[MAXIMUM];       / *  定义结构体数组 student[MAXIMUM]* /
char str[10];
int i;
for(i=0;i<MAXIMUM;i + +)            / *  给结构体成员初始化  * /
{
   printf(" \ n Please enter name:");
   gets(student[i].name);
   printf(" \ n Please enter sex:");
   gets(str);
   student[i].sex = str[0];
   printf(" \ n Please enter age?");
   gets(str);
 student[i].age = atoi(s);
   printf(" \ n Please enter telnumber:");
   gets(student[i].telnumber);
   printf(" \ n Please enter address:");
   gets(student[i].address);
}
printf(" \ n name age sex telnumber address \ n");
printf("                                        \ n");
for(i =0; i<MAXIMUM;i + +)
{
printf(" % -14s %-7d",student[i].name,student[i].age);    /*输出结果  * /
printf(" % -7c %-10s %-25s",student[i].age,student[i].telnumber,student[i].address);
}
}
```

从该例中可以看出，使用结构体数组可以使用循环，使程序十分简练。

7.4 结构体指针

7.4.1 结构体指针及其定义

结构体指针即指向结构体变量的指针，它是一个指针变量，而且其目标变量是一个结构体变量，其内容是结构体变量的首地址。结构体指针的定义类似一般指针，其一般格式如下：

［存储类型］ struct **结构体名 *结构体指针名；**

例如：

```
static struct example *pexample；
```

其中“ struct example **”是预先说明的结构体类型。它定义了一个结构体指针** pexample**，其存储类型是** static**。**

在定义结构体指针变量时，应注意结构体名必须是已说明过的结构体；而且结构体指针在使用之前必须通过初始化赋值操作，以便把某个结构体变量的首地址赋给它，使它指向该结构体变量，结构体指针所指向的结构体变量必须与定义时所规定的结构体类型一致。对于结构体指针的运算，其原则与一般指针相同。可对它进行加 1 或减 1 等运算，其含义就是将指针指向下一个或上一个结构体变量。例如：

```
struct key keytab［31］，* ps；
ps=keytab；
```

或：

```
ps=& keytab［0］；
```

指针 ps 指向结构体数组 keytab ［］ 的示意图，如图 7-1 所示。

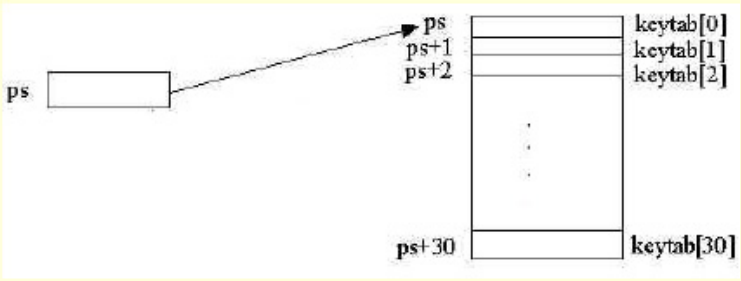


图 7-1　指向结构体数组的指针

如果指针开始向结构体数组的首地址，即指向 keytab ［0］，则指针加 1 后就指向 keytab ［1］，再加 1 后就指向 keytab ［2］。如果指针当前指向 keytab ［30］，则 ps 减 30 就指向 keytab ［0］。因此，对于指向结构体数组的指针，加 1 后使指针实际指向下一个结构体变量。

7.4.2　通过指针引用结构体成员

前面讲过通过结构体运算符“ . ”引用结构体成员的方法，在引入结构体指针以后，就可以用指针来引用结构体的成员了。用指针引用结构体成员的方式如下：

```
（* 结构体指针名）.成员
```

或

```
结构体指针名－＞成员名
```

例如：

```
struct key
 {
char * keyword；
int keycount；
}   * pkey；
```

则以下的引用都是合法的：

```
（* pkey）.keyword
（* pkey）.keycount
pkey- >keyword
pkey- >keycount
```

其中（* pkey）. keyword 和 pkey- >keyword 等价，（* pkey）.keycount 和 pkey- >keycount 等价。

至此，对结构体成员的引用共有如下三种形式：

- 结构体变量名.结构体成员名
- （*结构体指针名）.结构体成员名
- 结构体指针名->结构体成员名

由于运算符- >、.、（）和 ［］ 的优先级最高。因此，当结构体成员出现在表达式时要注意书写形式。表 7-1 是这些运算符的例子。

表 7-1 结构体成员的引用及计算

结构体样例	引 用	意 义
struct { int x; int *y; }*p;	(*p).y	是引用结构体成员 y
	*p.y	表示 p.y 是一个指针，* p.y 是取该指针的目标
	+ +p->x	该式等价于+ +（p- > x），即结构体成员 x 加 1，而不是+ +p
	(+ + p)->x	表示在存取 x 之前 p 先加 1
	(p + +)->x	在存取 x 之后 p 才加 1
	p + +->x	在存取 x 之后 p 才加 1
	* p - > y	在存取 x 之后 p 才加 1
	* p - > y + +	表示在存取 y 所指向的内容，它等价于*（p- > y）
	(* p - > y) + +	表示 y 所指向的内容加 1

7.4.3 结构体指针的应用举例

【例 7-4】指向结构体变量指针的应用实例。用结构体管理学生的学号、姓名和成绩。

```
main()
{
struct stud                          / *  定义结构体 stud * /
{
   char    * idnumber;               / *  定义结构体成员  * /
   char name[30];
   int score;
};
struct stud student, *sp;            / *  定义结构体变量及结构体指针变量  * /
sp = & student;                      / *  使指针 sp 指向结构体变量 student * /
sp - > idnumber="971811";
strcpy(( *sp).name,"Guo Lin");       / *  结构体成员初始化  * /
student.score = 97;
printf(" \ n student No.: % s\n name: % s \ n score: % d \ n", / *输出结果  * /
sp - > idnumber,sp - >name,sp - >score);
printf(" \ n student No.:% s name: % s score: % d",
( * sp).idnumber,( * sp).name,(* sp).score);
}
```

该程序先说明了一个结构体类型“ struct stud ”，然后定义了一个该结构体类型的结构体变量 student 和结构体指针 sp，并使 sp 指向 student。程序中的两个输出语句分别以不同的形式引用结构体成员。程序的运行结果如下：

```
student No.：971811
name：Guo Lin
score：97
student No.：971811 name：Guo Lin score：97
```

可见两个 printf（）的输出结果是相同的。这说明如下写法是等价的。

（* sp）.idnumber 和 sp- >idnumber

（* sp）.name 和 sp- >name

（* sp）.score 和 sp- >score

【例 7-5】指向结构体数组指针的应用实例。用指向结构体数组的指针变量处理学生的学号、姓名和成绩。

```
struct stud                          / *  定义结构体  * /
{
char * idnumber;                / *  定义结构体成员  * /
char name[30];
```

```
int score;
};
struct stud student[3] ={{"970001","Guo LIN",97},     / *  定义并初始化结构体数组* /
{"970002","Li Ping",83},
{"970003","Yang Wer",88}
};
main()
{
struct stud     * sp;                          / *  定义结构体指针  * /
printf("%-20s %-20s %4d\n",sp->idnumber,sp->name,sp->score);
}
```

该程序的输出结果如下：

```
Student     No.          Name             score
970001                   Guo Lin          97
970002                   Li Ping          83
970003                   Yang Wei         88
```

此例中，定义了一个结构体数组 student［]，**它是外部变量，因此可对其初始化。在** main（）**函数内部，定义了类型为“** student stud **”的结构体指针** sp。**在** for **循环内，开始时指针指向结构体数组成员** student［0]，**第二次循环时，**sp **指向** student［1]，**第三次循环时** sp **指向** student［2]。**所以，指针加** 1 **使它指向下一个结构体数组成员（指针变量内的实际地址值增加了下一个结构体数组成员的空间长度)。**

【例 7-6】**用结构体指针处理通讯录数据。假设通讯条件包含姓名、年龄、性别、电话号码和地址等信息。前面用结构体数组处理通讯录，这里用结构体指针来改写此例如下：**

```
# define < stdio.h >
# define < stdlib.h >
# define MAXIMUM 20
struct stud
{
char name[30];
int age;
char sex;
char telnumber[8];
char address[100];
};
main()
{
struct stud student[MAXIMUM],* sp;          / *  定义结构体数组及指针  * /
char str[10];
int i;
for(i =0;i <MAXIMUM;i + +)                        / *  建立通讯录  * /
sp=student;
{printf(" \ n Please enter name:");
gets(sp - > name);
printf(" \ n Please enter age:");
gets(str);
( * sp).age = atoi(str);
printf(" \ n Please enter sex :");
```

```
gets(str);
sp - >sex = str[0];
printf(" \ n Please enter telnumber:");
gets(sp - >telnumber);
printf( " \ n Please enter address:");
gets((* sp).address);
sp + +;
  }
  printf(" \ n Name                    sex        old          Tel          Address");      * /输出通讯  * /
  printf(" \ n- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - ");
  sp = student;
  for (i=0;i<MAXIMUM;i + +)
  {
Printf(" \ n % - 14s % -7d",sp- >name,sp- >age);
Printf(" % -7 % -10 % -25s",sp - >age,sp - >telnumber,sp- >address);
sp + +;
}
}
```

本例是建立一个学生通讯录，我们定义了一个结构体数组和一个结构体指针，数组的每一个元素表示一个学生的通讯信息。在建立通讯记录或输出通讯录时，首先把结构体数组的首地址（即 student）赋给指针，使它指向结构体数组的第一个元素 student［0］。在第一次循环中，把第一个学生的通讯信息输入 student［0］中，或从中输入，然后 sp 加 1，使它指向第二个数组元素 student ［1］，继续处理第二个学生的通讯信息，直至结束。在程序中，（* sp）.age 和（* sp）.address 等价于 sp- >age 和 sp- >address。

7.5 结构体在函数间的传递

在讲函数的调用时，讲到调用函数和被调用函数之间可以通过参数进行通讯。函数间不仅可以传递简单变量、数组、指针等类型的变量，而且还可以传递结构体类型的变量。函数之间的结构体变量的传递同一般变量一样，可以用数据复制方式传递，也可以用地址复制方式传递。

7.5.1 结构体变量的传递

1．用地址复制方式传递结构体变量

采用地址复制方式传递结构体变量时，实参是结构体变量的首地址，形参是与实参有相同结构体类型的指针，该指针用来接收传递的结构体变量首地址。被调用函数通过该指针来处理实参所对应的结构体变量的各成员。用地址复制方式传递结构体变量是常用的传递方式。

【例 7-7】用地址复制方式传递结构体变量应用实例。

给出年月日计算它是该年的第几天。其处理采用了地址复制方式。

```
struct date
{
int year;
int month;
int day;
int yearday;
};
main()
```

```
{
struct date date1;
printf(" \ n");
printf("Please enter year ,month,day:");
scanf(" % d,% d,% d",& date1.year,& date1.mouth,& date1.day);
days( & date1);
printf(" \ n 所输入日期是该年的第% d 天。",date1.yearday);
}
days(sp)
struct date * sp;
{
static int monthtable[][13] = {
{0,31,28,31,30,31,30,31,31,30,31,30,31},
{0,31,29,31,30,31,30,31,31,30,31,30,31}};
int i,leap;
sp - >yearday = sp - >day;
i =sp- >year;
leap =0;
if((i % 4= =0)&&(i % 100! =0))leap =1;       / * 判断是否为闰年* /
if(ie % 400 = =0)leap=1;
for(i =1;i <sp - >month;i + +)
sp - >yearday + = monthtable[leap][i];
}
```

该程序某次运行的输出结果如下：

```
Please enter year，month，day：2000，10，10＜CR＞
所输入日期是该年的第 284 天。
```

在 main（）**函数中定义了一个结构体变量** date1，**通过** scanf（）**函数给该结构体变量输入值。在调用** days（）**函数时，其实参是结构体类型的指针变量，该指针变量接收传递过来的实参首地址，即结构体变量** date1 **的首地址。函数通过** sp **访问的结构体变量是** date1，**并可修改其成员值，例如** sp- >yearday。**所以，利用地址复制方式传递结构变量时，被调用函数可以修改该结构体变量的成员。**

2．用数据复制方式传递结构体变量

用复制方式传递结构体变量时，可以把一个结构体变量作为普通变量来处理，例如两个类型相同的结构体之间可以进行赋值，可以把一个结构体变量作为一个参数以复制方式传递给被调用函数，函数也可以返回一个结构体类型的值等。下面是用复制方式把结构体变量传递给被调用函数的例子。

【例 7-8】**用数据复制方式传递结构体变量实例。**

```
struct person                          / * 在主函数之前定义一个结构体 * /
{
char name[30];
float salary;
float comm;
float sum;
};
main()
{
struct person person1;                         / * 定义结构体变量 person1 * /
printf(" \ n 请输入姓名:");
```

```
scanf("% s",person1.name);
person1.salary = 315.10;
person1.comm = 350.60;
person1.sum = person1.salary+ person1.comm;
printf(" \ n% s % f",person1.name,person1.salary);
printf("% f % f",person1.comm,person1.sum);
printf(" \ n 运行子程序");
fun(person1);
printf(" \ n 运行完子程序后");
printf(" \ n% s % f",person1.name,person1.salary);
printf("% f % f",person1.comm,person1.sum);
}
fun(sp)
struct person sp;
{
printf(" \ n % s % f % f",sp.name,sp.salary,sp.comm,sp.sum);
sp.salary = 519.50;
sp.comm = 410.50;
sp.cum = sp.salary + sp.comm;
printf(" \ n %s % f % f",sp.name,sp.salary,sp.comm,sp.sum);
}
```

执行该程序的输出情况如下：

```
请输入姓名：Wei Fang＜CR＞
Wei 315.100006 350.600006 665.700012
运行子程序
Wei 315.100006 350.600006 665.700012
Wei 519.500000 410.500000 930.000000
运行完子程序后
Wei 315.100006 350.600006 665.700012
```

在 main（）函数中，我们定义了一个自动存储类型的结构体变量 person1。对它进行输入和赋值后，调用 fun（）函数。调用时把结构体变量 person1 作为实参传递给被调用函数 fun（）。被调用函数 fun（）的形参 sp 与 person1 具有相同的结构体类型。sp 接收调用时传递给调用函数的数据。结构体变量 sp 和 person1 在运行时占用互不相同的空间。在传递参数时，把 person1 的各个成员的值复制给 sp 的各个成员。如图 7-2 所示。

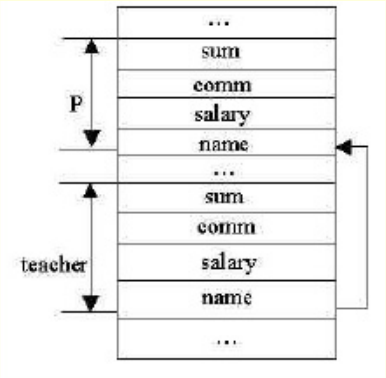


图 7-2　用数据复制方式传递结构体变量

由于调用和被调用函数之间是数据复制方式传递数据，所以 fun（）函数对 sp 的修改并不能修改 main（）

中定义的自动变量 person1，从输出结果中可以看到这一个特征。在以数据复制方式传递数据时应注意如下两点：

- 调用时的实参与被调用函数定义中的形参都是结构体变量名。
- 形参和实参代表两个不同的结构体变量，其结构体类型相同，但运行时分配不同存储空间。因此，被调用函数不能修改实参所对应的结构体变量。

7.5.2 结构体数组在函数间的传递

函数间不仅可以传递一般结构体变量，而且还可以传递结构体数组。在传递结构体数组时，实参是数组名，即结构体数组的首地址；形参是指针，它接收传递过来的数组首地址，使它指向实参所表示的结构体数组。下面是传递数组的实例：

【例 7-9】编程建立和显示全年级学生成绩册。

```
# include <stdio.h>
# define MAXIMUM 300
struct stud
{
char idnumber[6];
char name[30];
int score;
};
main()
{
struct stud student[MAXIMUM];              / * 定义结构体数组 * /
int i;
for(i=0;i<MAXIMUM;i+ +)                    / * 建立学生成绩册 * /
{if (input(&student[i])= = 0)
break;p
printf("\ n");
display(student,i);
}
}
input(sp)
struct stud * sp;
{
char str[10];
printf("\ n 请输入学号：");
gets(sp- >idnumber);
printf("\ n 请输入姓名：");
gets(sp- >name);
if(sp- >name[0]= =\ 0')                    / * 不输入学生姓名作为输入结束标志* /
return(0);
printf("\ n 请输入成绩：");
gets(str);
sp- >score = atoi(str);
return(l);
}
display(sp,n)
struct stud * sp;
```

```
int n;
{
int i;
printf("Idnumber Name        Score \ n");
printf("-------------------------------- \ n");
for(i=0;i<n;i+ +,sp + +)
printf("%-10s %-30s %d \ n",sp->idnumber sp- >name,
sp- >score);
}
```

在 main（）函数中，我们定义了一个结构体数组 student［]，用于存放成绩册。函数 input（）用于建立成绩册，函数调用 input（& student［i]）中的实参是结构体数组 student［i］的首地址。input（）函数用于输入一个学生成绩的有关信息，其形参 sp 是下一个 struct stud 结构体类型的指针，用于接收结构体数组元素的首地址，当返回值为 1 时，表示输入了一个学生的信息，当返回值为 0 时，结束输入。函数 display（）用于显示学生的成绩册。其形参 sp 是 struct stud 结构体类型的指针，函数调用时，接收结构体数组首地址。当调用函数 input（）时，将结构体数组元素的首地址（这也是一个结构体变量的首地址）传给了 input 函数。当调用函数 display（）时，将结构体数组的首地址传给函数 display（）。

7.6 结构体型和结构体指针型函数

7.6.1 结构体指针型函数

当函数的返回值是结构体变量的首地址时，称该函数为结构体指针型函数，该函数的数据类型为结构体指针型。结构体指针型函数定义的一般形式为：

```
struct 结构体名 * 函数名（［形参表］）
［形参说明；］
{
内部数据说明语句；
执行语句；
}
```

其说明形式为：

```
struct 结构体名 *函数名（）；
```

在调用结构体指针型函数时，用于接收结构体指针函数返回值的变量应该是指向与该函数具有相同结构体类型的结构体指针变量。下面是结构体指针型函数的举例。

【例 7-10】编一程序实现电话号码查询，假设有一个电话号码表，给定一个电话号码，找出该号码所对应的信息。

```
# define Null 0
struct data                          / * 定义结构体 * /
{
int telnumber;
char name[30];
char address[100];
}
struct data person[500];
personlist()                           / * 建立电话号码表函数* /
{
```

```
int i;
for(i=0;i<500;i+ +)
{
printf("\ n 请输入电话号码：");
scanf("% d",&person[i].telnumber);
printf("\ n 请输入姓名:");
scanf("%s",person[i].name);
printf("\ n 请输入地址:");
printf("%s",person[i].address);
if(person[i].telnumber = = Null)
break;
}
return;
}
struct data * found(n)                    / *给定电话号码查对应信息  * /
int n;
{
int i;
for(i= 0;person[i].telnumber ! = Null;i+ +)
if(person[i].idnumber = =n)
break;
return(&person[i]);
}
main()
{
int number;
struct data * sp;
personlist();
for(;;)
{
printf("\ n 请输入电话号码:");
scanf("% d",&number);
if(number = = Null)
break;
sp = found(number);
if(sp- >idnumber ! = Null)
{
printf("\ n 电话号码：%d",sp- >idnumber);
printf("\ n 姓名：%s",sp- >name);
printf("\ n 地址：%s",sp- >address);
}
else
printf("\ n 没找到相应资料！")
}
}
```

该例把电话号码表存放在一个外部结构体数组中，电话号码表以电话号码 0 结尾。在 found（）函数中，如

果找到，则返回查到信息所存放的结构体首地址（或结构体数组元素的地址）。需要注意的是：结构体指针型函数返回结构体首地址，但该结构体一定不能是该函数中定义的自动型结构体变量。

7.6.2 结构体型函数

前面已经讲过，函数具有数据类型，其数据类型是由返回值的数据类型所决定。如果函数返回值是某一结构体变量时，则称该函数是结构体型函数，该函数的数据类型就是这个结构体变量所具有的结构体类型。结构体型函数定义的一般形式如下：

```
struct 结构体名 函数名（［形参表］）
［形参说明；］
{
内部数据说明语句；
执行语句；
}
```

结构体型函数说明的一般形式如下：

```
struct 结构体名 函数名（）；
```

下面是结构体型函数的实例。

【例 7-11】返回值为结构体变量的函数的应用实例。假设某结构体有三个成员，通过返回结构体变量的函数向主函数传递在结构体函数中定义的结构体成员值。

```
# include <string.h>
struct data
{
char s[30];
int n;
float x;
}
struct data example()
{
struct data emp;
strcpy(emp.s,"This is an example program!");
emp.n = 68;
emp.x = 213.52;
printf("emp.s[ ] = %s emp.n = %d emp.x = %f \ n",emp.s,emp.n,emp.x);
printf("函数运行完后：\ n");
return((emp);
}
main()
{
struct data redata;
printf("\ n 运行函数：\ n");
redata = example();
printf("redata.s[] = %s redata.n = %d redata.x = %f \ n",redata.s,redata.n,reada.x);
}
```

该函数执行时的输出结果为：

```
运行函数：
emp.s［］=This is an example program!emp.n=68 emp.x=213.520000
函数运行完后：
```

redata.s［］=This is an example program!redata.n=68 redata.x=213.520000

函数 example（）被定义为结构体型函数，其类型为“ struct data ”。在定义中结构体类型和函数名分作两行书写，这在结构体类型名和函数名较长时，常常采用这种书写形式。在 main（）函数中，redata 也是“ struct data ”类型的结构体变量，它与 example 的类型相同，用 redata 接收函数 example（）的返回值。从运行结果可以看到，结构体变量 redata 中各成员的值与 emp 结构体变量相同。因此，redata=example（）可以看作结构体变量之间的赋值操作。

7.7 结构体嵌套

7.7.1 什么是结构体嵌套

如果某结构体的成员又是结构体类型，则称这样的结构体是嵌套结构体类型。例如，

```
struct date                                  / * 结构体定义* /
 {
int year；                                   / * 结构体成员* /
int month；
int day；
} ；
struct persondata
 {
char idnumber［6］；
char name［30］；
char sex；
struct date birthday；                       / * 结构体的成员又是一个结构体类型，
                                               结构体在此嵌套* /
} ；
```

在说明 persondata 结构体时又引用了 date 结构体类型。因此，persondata 结构体是嵌套型结构体类型。其中，date 的说明是内层结构体说明，而 persondata 的说明是外层结构体说明。

如果在一个结构体的成员中又引用了自身的结构体类型，则称该结构体类型是递归结构体类型，这种说明称递归说明。例如，定义一个如图 7-3 所示的链表：

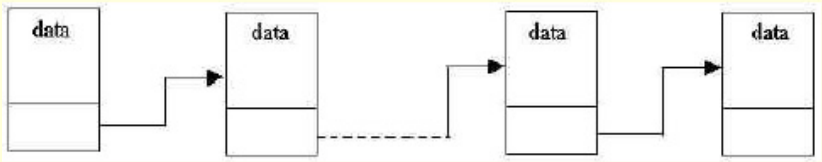


图 7-3 链表

```
struct example
 {
int data；
struct example * next；
} ；
```

则称 example 是一个递归型结构体类型。在说明嵌套型结构体类型时，应当注意如下两点：

- 从理论上讲嵌套的层次是无限的，但过多的层次将会增加引用的复杂性。
- 在嵌套说明中，内层结构体类型的说明必须在外层结构体类型说明之前进行说明，否则在编译时，将会发生错误。

7.7.2 嵌套结构体类型变量的引用

对于一般结构体变量，通常都是由其成员参加运算；而对于嵌套型结构体变量，通常都是由最内层的结构体成员参与运算。对于嵌套型结构体变量，其成员具有层次性。例如：

```
struct address
{
int post;
char address［100］;
char telnumber［8］；
}；
struct person
 {
char name［30］；
struct address workaddress；
struct address homeaddress；
}；
struct person Li；
```

嵌套结构体变量成员引用的一般形式有如下两种形式：

（1）用成员运算符“.”，其一般形式为：

结构体变量名.外层成员名.内层成员名

例如，引用各个 Li 结构体变量成员的形式如下：

```
Li.name
Li.homeaddress.post
Li.homeaddress.address
Li.homeaddr.telnumber
Li.workaddress.post
Li.workaddress.address
Li.workaddress.telnumber
```

（2）用指针运算“->”，例如：

```
struct person * pman；
pman->name
pman->workaddress.post
pman->workaddress.address
pman->workaddress.telnumber
pman->homeaddress.post
…
```

又如：

```
struct example * p；
p->data
p->next->data        /*下一个结点项 */
```

无论是运算符“.”或是“->”，二者都具有相同的优先级和自左向右的结合规则。因此，当嵌套型结构体变量出现在表达式中时，总是自左向右运算，即从外层向内层逐步访问结构体成员。

7.7.3 结构体嵌套应用举例

在结束本节之前，我们给出一个结构体嵌套的应用实例，从中进一步弄清结构体成员的引用格式。

【例 7-12】用嵌套型结构体变量处理通讯录。

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

```
# include <stdlib.h>
# define MAXIMUM 300
struct address                                  / *  定义结构体  * /
{
char * postcode;
char addess[100];
char telnumber[8];
};
struct person                                   / *定义结构体  * /
{
char name[30];
struct address workaddress;                     / *结构体成员嵌套定义  * /
struct address homeaddress;                     / *  结构体成员嵌套定义  * /
};
struct person grade[MAXIMUM];                   / *定义结构体变量  * /
inputdata()
{
int i;
char s[10];
for(i = 0;i<100;i+ +)
{
printf("\ n 请输入姓名:");gets(grade[i].name);
if(grade[i].name[0] = = \ 0')
return;
printf("\ n 请输入工作地址的邮政编码:");
gets(grade[i].workaddress.postcode);
printf("\ n 请输入工作地址:");gets(grade[i].workaddress.address);
printf("\ n 请输入办公室电话号码:");gets(grade[i].workaddress.telnumber);
printf("\ n 请输入家庭所在地的邮政编码:");gets(grade[i].homeaddress.postcode);
printf("\ n 请输入家庭地址:");gets(grade[i].homeaddress.address);
printf("\ n 请输入家中电话号码:");gets(grade[i].homeaddress.telnumber);
return;
}
}
main()
{
int i;
printf("\ n Name postcode workaddress telnumber postcode homeaddress telnumber");
printf("\ n----------------------------------------------------------------------------------");
for(i = 0;i<MAXIMUM;i+ +)
{
if(grade[i].name[0] = = \ 0')
break;
printf("\ n %s",grade[i].name);
printf("%d %s %s",grade[i].wordaddress.postcode,grade[i].workaddress.address,grade[i].
wordaddress.telnumber);
```

```
printf("%d %s %s",grade[i].homeaddress.postcode,grade[i].homeaddress.address,grade[i].
homeaddress.telnumber);
}
}
```

7.8 联　合

联合是一种类似于结构体的构造型数据类型，它准许不同类型和不同长度的数据共享同一块存储空间。也就是说，具有联合类型的变量所占用的空间，在程序运行的不同时刻，可能维持不同数据类型和不同长度的数据。这些不同类型和不同长度的数据都是从该共享空间的起始位置开始占用该空间。所以，联合提供了在相同的存储域中操作不同类型和不同长度数据的方法。联合实质上是采用了覆盖技术，准许不同类型数据互相覆盖。在程序设计中，采用联合要比使用结构体节省空间，但访问速度较慢。

7.8.1 联合的说明及联合变量的定义

1．联合的说明

联合的说明与结构体类似，其一般方法如下：

```
union 联合名
 {
数据类型标识符 成员名 1；
数据类型标识符 成员名 2；
…
数据类型标识符 成员名 n；
} ；
```

联合的说明仅规定了联合的一种组织形式，它并不分配存储。也就是说，联合说明只规定了联合使用内存的一种模式，即规定了一种类型（叫联合类型），联合名就代表这种类型，并不代表一个变量。例如：

```
union uniontype
 {
int i；
flag f；
char * p；
}
```

上例中规定了一种名为 uniontype 的联合类型。它说明该类型是由三个不同数据类型的成员所组成，它们共享同一块内存空间。

2．联合变量的定义

一旦一个联合被说明，就可以用它来定义联合变量了。联合变量定义的一般方式有如下三种：

（1）说明与定义分开，其一般形式如下：

```
union 联合名
 {
数据类型标识符 成员名 1；
数据类型标识符 成员名 2；
…
数据类型标识符 成员名 n；
} ；
union 联合名 联合变量名表列；
```

例如：

```
union uniontype
 {
int i；
char c；
short s；
long l；
} ；
union uniontype udata；
```

（2）定义与说明合在一起，其一般形式如下：

```
union 联合名
 {
数据类型标识符 成员名 1；
数据类型标识符 成员名 2；
…
数据类型标识符 成员名 n；
} 联合变量名表列；
```

例如：

```
union uniontype
 {
int i；
char c；
short s；
long l；
} udata；
```

（3）定义和说明合在一起，但缺省联合名，其一般形式如下：

```
union
 {
数据类型标识符 成员名 1；
数据类型标识符 成员名 2；
…
数据类型标识符 成员名 n；
} 联合变量名表列；
```

例如：

```
union
 {
int i；
char c;
short s；
long 1；
} udata；
```

在以上三种定义形式的例子中，所定义的联合变量 udata **是等价的。联合变量** udata **有四个成员（**i、c、s **和** l**）。编译时为它分配内存空间，编译系统按最长的成员分配内存。由于** long **最长，它占用** 4 **字节，故为该联合变量分配** 4 **个字节的内存。结构体与联合可以互相嵌套，例如：**

```
union uniontype
 {
int i；
```

```
float f；
} ；
struct structtype
 {
short s；
long l；
} ；
struct sutype                          / * 结构体中嵌套联合 * /
 {
char c；
union uniontype u；
} ；
union ustype                             / * 联合中嵌套结构体 * /
 {
int i；
struct structtype st；
} ；
```

类似于结构体，我们也可以定义联合数组，另外联合中的成员也可以是数组。例如，

```
union uniontype u［40］；                  / * u［］即为联合数组 * /
union                                    / * 联合的成员是数组 * /
 {
long l；
int a［40］；
} ；
```

事实上，一个联合也可以看作是一个结构体，只是其所有成员在结构体中的位置的相对位移量均是 0，其长度要大到足以容纳下每一个成员，边界要适合于联合中的所有类型。

3. 联合变量成员的引用

联合变量成员的引用类似结构体变量，也是利用运算符“ - > ”和“ . ”。例如：

```
union uniontype
 {
int i；
char c；
} ；
union uniontype udata，*p；
p=& udata；
```

则下边引用都是合法的：

```
udata .i，udata.c，p- >i，p- >c
```

7.8.2 使用联合变量应注意的问题

（1）联合变量的应用类似于结构体变量。在程序中，一般参与运算或操作的只能是联合变量的成员，而不能是联合变量整体。但可以对联合变量进行取地址运算（即 &）。在 ANSI 新标准中，允许对联合变量进行如下操作：

- 两个具有相同联合类型的联合变量可以互相赋值。
- 联合变量可以作为参数传递给被调用函数。
- 联合变量可以作为函数的返回值，从被调用函数返回调用函数。

（2）在对联合变量的成员进行赋值或引用时，要特别注意该成员当前存放的是什么类型的数据，一般需要

引用的数据类型要与当前存放的数据类型相一致，否则会出现错误。

（3）联合变量与结构体变量在说明、定义和引用方面都很类似，但二者也有区别，其主要区别是：

- 在程序执行的任何特定时刻，仅有一个联合成员留驻在该联合变量所占用的空间中；而结构体变量是所有的成员都同时留驻在该结构体变量所占用的空间中。
- 在初始化时，结构体变量的所有成员都可以初始化，而联合变量只需要初始化第一个成员即可。

（4）联合变量的地址与其各成员的地址相同，因为各成员地址的分配都是从联合变量空间的起点开始。

（5）不能对联合变量名赋值，也不能通过引用联合名来得到其成员的值。另外，在定义时也不能对联合名初始化。

下面举一个联合的应用的例子：

【例 7-13】编程建立一个教师和学生登记表，其中包括识别号、姓名、性别、身份和职称。如果身份是‘ student ”时，则职称一栏填年级；如果身份是“ teacher ”时，则职称栏填职称。

其程序如下：

```
struct persontype
{
int idnumber;
char name[30];
char sex;
char job[10];
union
{
int grade;
char position[10];
}level;
}person[10];
main()
{
int n,i;
printf("\ n 请输入个人信息（识别号，姓名，性别，身份，年级或职称）：\ n");
for(i=0;i<10;i+ +)
{
scanf("%d %s %c %s",&person[i].idnumber,person[i].name,
&person[i].sex,person[i].job);
if(strcmp(person[i].job,“ student ”))
scanf("%d",&spersonp[i].level.grade);
else if(strcmp(person[i].job,“ teacher ”))
scanf("%s",person[i].level.position);
else
printf(“ 输入错误！\ n ”);
}
printf("Idnumber name sex    job             grade/position\ n");
printf("-------------------------------------------------------------------------\ n");
for(i=0;i<10;i+ +)
{
if(strcmp(person.[i]job,"student"))
printf("% - 15d % - 10s% - 6c % - 20s % -15d \ n",
person[i].idnumber,person[i].name,person[i].sex,person[i],job,person[i].level.grade);
```

```
else
printf("% - 15d % - 10s % - 6c % - 20s % - 15s \ n",person[i].idnumber,
person[i].name,person[i].sex,person[i].job,person[i].level.position);
}
}
```

某次程序运行结果如下：

```
请输入个人信息（识别号，姓名，性别，身份，年级或职称）：
970 Guo f student 3
971 Zhang m student 3
972 Li m teacher professor
973 Wei f teacher vprofessor
974 Yang f student 1
975 Tang m student 4
976 Liu f teacher assistant
977 Wang f student 2
978 Chi m student 3
979 Chen m teacher professor
```

Idnumber	name	sex	job	grade/posion
970	Guo	f	student	3
971	Zhang	m	student	3
972	Li	m	teacher	professor
973	Wei	f	teacher	vprofessor
974	Yang	f	student	1
975	Tang	m	student	4
976	Liu	f	teacher	assistant
977	Wang	f	student	2
978	Chi	m	student	3
979	Chen	m	teacher	professor

该例中定义了一个外部结构体变量，该变量内嵌套了下一个联合变量。其中的成员 job **起着标志变量的作用，它标志联合中当前存放的是“** teacher **”或“** student **”。**

7.9 枚举类型

7.9.1 什么是枚举类型

枚举类型是 87 ANSI **标准中新增加的数据类型。如果一个变量只可能取某几种值，或者说只能赋给它某几种值，而不能取其他值，则可将它定义为枚举类型。所谓枚举就是变量在定义时，将它所有可能的取值都一一列举出来。这种在定义时就明确规定变量只可能取哪几个值，而不能取其他值的这类数据类型叫枚举类型。**

7.9.2 枚举类型的说明

枚举类型说明的一般形式为：

enum **枚举名｛元素名** 1，**元素名** 2，…**元素名** n｝；

例如：

enum month ｛Jan，Feb，mar，Apr，May，Jun，Jul，Aug，Sep，Oct，Nov，Dec｝；

/* 该枚举类型只准许取 12 种值 */

enum weekday {Sun，Mon，Tue，Wed，Thu，Fri，Sat} ；

/* 该枚举类型只准许取 7 种 */

enum flag {true，false} ； /* 只准许两个值 */

枚举类型的说明仅规定该类型只准许取哪几种值，而不许可取其他值，它并不分配内存。也可以说，说明只规定了枚举的类型，枚举名实际是枚举类型名。所以，month、weekday 和 flag 是三个枚举类型名。

7.9.3 枚举型变量的定义

枚举型变量定义的一般形式为：

enum［枚举名］［{元素名 1，元素名 2，…，元素名 n}］枚举变量名表列；

例如，下面几种定义形式都是合法的：

（1）说明与定义合一，如 enum flag {true，false} firstflag，secondflag；

（2）用无名枚举类型，如 enum {true，false} firstflag，secondflag；

（3）说明与定义分开，如 enum flag {true，false}；enum flag firstflag，secondflag；

上面三种定义形式所定义的枚举型变量 firstflag 和 secondflag 都只能取 true 和 false 两个值，而不能取其他值。在编译时，为定义的变量分配内存，一个枚举型变量占用的内存空间与 int 型相同。

7.9.4 如何正确使用枚举型变量

为了正确使用枚举型变量，应注意如下几点：

（1）C 编译系统把枚举类型说明中的元素作为常量名处理，它不是变量名。因此，不能对其赋值。

（2）C 编译系统在编译时，对这些常量按照元素的定义顺序分别使它们表示 0，1，…，n-1。例如：

enum flag {true，false} firstflag；

其中，元素 true 的值是 0，false 的值是 1。又如：

enum weekday {Sun，Mon，Tue，Wed，Thu，Fri，Sat} workday；

其中各元素的值分别为 0、1、2、3、4、5、6。

（3）枚举元素的值也可在定义时指定。例如：

enum weekday {Sun=7，Mon=1，Tue，Wed，Thu，Fri，Sat} workday；

对于没有指定值的元素，其取值原则仍按所处的顺序取，所以：

Tue 是 2，Wed 是 3，…，Sat 是 6。

（4）可用枚举变量进行判断或比较操作。例如：

```
enum flag {true，false} firstflag；
…
if（firstflag= =true）
```

（5）可对枚举变量进行赋值操作。例如：

```
enum flag fitstflag；
enum month this_month；
enum weekday workday；
```

但是，一个整数不能直接赋给枚举变量，要先进行强制类型转换才能赋值。例如：

workday= 3；

是错误的。而下列格式是正确的：

workday=（enum weekday）3；

以上格式相当于：

workday=Wed；

同样，也可以把一个表达式赋给枚举变量。如下面的赋值是合法的：

```
workday=（enum weekday）（5-2）；
month_value=7；
```

this_month=（enum month）（month_value-2）；

int month_value；

（6）枚举类型变量的存储类型也有自动、外部和静态之分。例如，

static enum flag complete_flag；

是静态枚举变量。

main（）

｛

enum flag endflag；

…

｝

endflag 就是一个自动枚举变量。

另外，如果枚举类型的说明是在函数内部，则只能在该函数内定义该枚举类型的变量。如果是在函数外说明，则可以在该说明之后的各函数内或外定义该枚举类型的变量。

【例 7-14】某科学小组有 10 人，现要选 3 人去执行某科研任务，假设每人参加的概率均等，问有哪几种选择方案。

分析：假设这 10 人分别叫 m1、m2、m3、m4、…m10。因为只能是这 10 人，不能是他人，故可用如下形式的枚举型表示它：

enum group{m1,m2,m3,m4,m5,m6,m7,m8,m9,m10};

其中 get1、get2 和 get3 表示一种选择方案中的三个人。并且：

get1!=get2!=get3。

n 为记录选择方案的总数。

```
main()
{
enum group{m1,m2,m3,m4,m5,m6,m7,m8,m9,m10};
enum group get1,get2,get3,pri;
int n,loop;
n = 0;
for(get1 = m1;get1 < = m10;get1 + +)
for(get2 = get1 + 1;get2 < = m10;get2 + +)
if(get1! = get2)
{
for(get3 = get2 +1;get3 < =m10;get3+ +)
if((get3! = get1)& &(get3 ! = get2))
{/ *输出一种选择方案 * /
n + +;
printf("%-4d",n);
for(loop = 1;loop < = 3;loop+ +)
{
switch(loop)
{
case 1:pri = get1;break;
case 2;pri = get2;break;
case 3:pri = get3;berak;
default:break;
}
switch(pri)
```

```
{
case m1:printf("% - 10s","m1");break;
case m2:printf("% - 10s","m2");break;
…
case m10:printf("% - 10s","m10");break;
default:break;
}
}
printf（"\ n"）;
}
}
printf("\ n total = %5d",n);
}
```

7.10 自定义类型

7.10.1 自定义类型（typedef）的含义及表示形式

所谓类型定义就是给已经存在的数据类型重新命名。例如，数据类型 float 可重新命名为 REAL：

```
typedef float REAL；
```

类型定义的一般形式为：

```
typedef oldtype newtype;
```

其中：typedef 是保留字，oldtype 是已有的数据类型名，newtype 是要命名的新类型名。

类型定义语句“ typedef oldtype newtype；”的功能就是把 oldtype 重新命名为 newtype，即把 newtype 说明为与 oldtype 相同的类型。例如：

```
typedef int INTEGER；
```

上述类型为 int 重新命名了一个新名字，于是 int i，j；与 INTEGER i，j；是完全等价的两个变量定义语句。同样，typedef float REAL；把 float 重新命名为 REAL，于是 float x，y；与 REAL x，y；也是等效的定义语句。

所以，类型定义实际上是为已有的类型重新命名，它并不产生新的类型，也不分配内存。使用类型定义语句定义的新名字来定义的变量与系统规定的类型名定义的变量具有相同的性质。事实上 typedef 和预编译命令＃define 在此点上是完全类似的。但二者还有如下几点区别：

- ＃define 是预编译命令，而 typedef 是 C 语言的一般语句。
- ＃define 是在预编译时处理，而 typedef 是编译时处理。
- 在书写形式上，＃define 以“＃”开头，而 typedef 不是；typedef 语句以“ ；”结束，而＃define 不是。
- ＃define 的功能是实现宏替换，而 typedef 是为程序增添新的类型名。

7.10.2 自定义类型的优点

使用自定义类型可以给程序带来如下两点好处：

1．增强程序的可读性

可以利用类型定义把已有的类型名重新命名为意义更加明确的名字，使程序更易理解。例如，对指向字符串的指针可以定义为：

```
typedef char * STRING：
```

这时 STRING 与 char * 同义，实际上 STRING 代表了一种数据类型，于是就可以用它来定义指向字符串的指针变量了。例如，定义：

```
char * ps；
```

```
char * lineptr［10］，* alloc（）；
```

与

```
STRING ps；
STRING lineptr［10］，alloc（）；
```

是等价的。又如：

```
typedef struct tnode
 {
char * word；
int count；
struct tnode * left；
struct tnode * right；
} TREENODE，*TREEPTR；
```

又命名了两个新的类型名：TREENODE 和 TREEPTR。其中 TREENODE 代表树结点类型“ struct tnode ”，而 TREEPTR 是指向树结点“ struct tnode ”的指针。于是，可以把为树结点分配内存的函数 talloc（）重写。

该函数原写法是：

```
struct tnode * talloc（）
 {
char * alloc（）；
return（（struct tnode*）alloc（size of（struct tnode））；
}
```

现改写为：

```
TREEPTR talloc（）
 {
char *alloc（）；
return（（TREEPTR）alloc（size of（TREENODE））；
}
```

可以把新的类型名取成与对象性质相关且容易理解的标志符。例如：

```
int（*strcmp）（），（*numcmp）（），（* swap）（）；
```

说明了三个函数指针，这些函数的返回值均为 int 型（其含义可用 Pointer Function Int PF1 来描述）。于是，该说明采用如下的方式将更易理解：

```
typedef int（* PF1）（）；
PF1 strcmp，numcmp，swap；
```

这样，在上下文中就容易知道它是函数指针型，所指向函数的返回值为 int 型。由此可以看出，typedef 能处理的置换已超出 # define。

2．增强程序的可移植性

一个程序与具体的机器关系越密切，其移植就越困难。例如，假设某程序原来是在甲机上开发的，该机上的 short 占用 2 字节，int 占用 4 字节等。考虑到这两个数据类型在不同的机器上，其长度可能不同。为了提高可移植性，在编码时可以采用如下的 typedef 语句：

```
typedef int INTEGER；
typedef short s_INT；
```

如果要把该程序移植到乙机器上，而该机上的 short 占用 1 字节，int 占用 2 字节，long 占用 4 字节等。这时只需如下改动这两个 typedef 语句即可：

```
typedef int s_INT；
typedef long INTEGER；
```

7.11 位字段结构体

在 C 语言的程序设计中，为了节约存储空间，常常用一个或若干个进制位来表示一个对象的某些属性。

这种用来表示一个对象属性的二进制位或二进制位组称为“位字段”。所以，位字段是由一个或多个二进制位组成，它是数据的一种压缩形式。位字段数据在整体上没有具体意义，具体到某个位字段才有意义。采用这种表示方法就要涉及到对“位字段”的存取。在 C 语言中，对位字段的存取有如下两种方式：

- 位操作方式
- 位字段结构体方式

7.11.1 位操作方式

C 语言中提供了位操作运算符，如：|、&、~、<<、>>、…。这些操作符已在第二章中描述过。可以利用这些位操作运算符进行字段操作。例如，有一个标志字 flag，其中记录着某标识的有关属性标志，我们要想知道某标志位的状态，可以分别对相应的二进制位定义对应的表征码。例如：

```
# define KEYWORD 01
# define EXTERNAL 02
# define STATIC    04
```

对 flag 某位字段的存取可以通过将 flag 与相应的表征码进行相应的位操作即可。例如：

```
int flag=0，temp；
…
flag &= ~（EXTERNAL|STATIC）；      / * 将 flag 的 external 和 static 位置 0 * /
flag|=EXTERNAL|STATIC;              / * 将 flag 的 external 和 static 位置 1* /
temp=flag & STATIC；                / * 取 flag 的 static 位* /
if（（flag &（KEYWORD|STATIC））！=0 / * 如果 flag 的 key word 和 static
                                    位均为 0 时，表达式为假* /
…
```

7.11.2 位字段结构体方式

C 语言中通过结构体变量将不同的类型的数据组织在一起，以便进行操作。仿照此思想，可以把不同的位字段按照结构体类型的形式组织在一起，这就是位字段结构体。所以，位字段结构体方式的操作实际上是依照结构体变量中对其成员进行操作的方式来对位字段进行操作。位字段结构体的定义和操作类似于结构体变量。

1．位字段结构体的定义

位字段结构体定义的一般形式是：

```
struct［位字段结构体类型名］
 {
unsigned 位字段名 1：e1；
unsigned 位字段名 2：e2；
…
unsigned 位字段名 n：en；
} 位字段结构体变量名；
```

其中：位字段结构体类型名可简称位字段结构体名。[] 表示该项可以省略。e1、e2、…en 是常量表达式，它表示位字段占用的二进制位数。例如：

```
struct id_flag
 {
unsigned is_keyword：1；
unsigned is_external：1；
```

```
unsigned is_static：1；
unsigned is_dummy：5；
unsigned is_char：1；
unsigned is_int：1；
unsigned is_long：1；
unsigned is_float：1；
unsigned is_double：1；
unsigned is_dummybit：3；
} flag；
```

上述定义了一个字的位字段结构体变量 flag，**它有** 10 **个位字段（即结构体成员），其中** is_dummy **和** is_dummybit **表示两个不使用的位字段。也可以先说明一个位字段结构体类型，然后再定义位字段结构体变量。例如：**

```
struct id_flag
 {
unsigned is_keyword：1；
unsigned is_external：1；
unsigned is_static：1；
unsigned is_dummy：5；
unsigned is_char：1；
unsigned is_int：1；
unsigned is_long：1;
unsigned is_float：1；
unsigned is-double：1；
unsigned is_dummybit：3；
} ；
struct id_flag flag；
```

如果只用字段结构体类型定义一个位字段结构体变量时，可以省去位字段结构体名。如：

```
struct
 {
unsigned is_keyword：1；
unsigned is_external：1；
unsigned is_static：1；
unsigned is_dummy：5；
unsigned is_char：1；
unsigned is_int：1；
unsigned is_long：1;
unsigned is_float：1；
unsigned is-double：1；
unsigned is_dummybit：3；
} flag；
```

2．**位字段的引用**

位字段的引用方法同结构体成员的引用方法。例如上面定义的位字段结构体变量 flag，**其成员的引用如下：**

```
flag.is_int
flag.is_float
flag.is_static
…
```

uf（（flag.is_keyword= =0）&&（flag.is_static= =0））

如果使用指向位字段结构体变量的指针，其引用也同一般结构体变量。例如，

```
struct id_flag flag，* a;
a=& flag；
a- >is_int
a->is_long
…
```

3. 关于位字段结构体变量定义时应注意的问题

- 位字段结构体变量占用的存储空间与 int 型数据相同，即使位字段结构的总长度小于 int 型的长度，也要占用一个 int 型大小的空间。当位字段结构体变量的总长度超过 int 型的长度时，它还将占用地址与前一个相连续的 int 型空间。但是，不准许任何一个成员（位字段）跨越两个 int 型空间的边界。若位字段较长，可能会跨越边界，则该位字段应从下一个 int 型边界开始。例如：

```
struct try
 {
unsigned al：2；
unsigned a2：5；
unsigned dummybit：4；
unsigned a4：6；
…
} ；
```

如果成员 a4 紧挨 a3 定义时，就会跨越 int 边界。为此引入不使用的位字段 dummybit，使 a4 从下一个 int 占用的空间开始。

- 位字段可以不命名，称为无名位字段。无名位字段只有冒号和长度，用它来填充特殊字段，以便使位字段结构体变量的空间与 int 所占用的空间边界对齐。例如：

```
struct test
 {
unsigned w1：3；
unsigned w2：4；
unsigned w9：9；
            } ；
```

- 位字段只能是 unsigned 型，它只能存于 int 类型的变量中。
- 位字段不能是数组，对位字段不能取地址运算（&）。
- 位字段的空间分配受系统影响，有的机器是从高位向低位分配，有的是从低位向高位分配。另外，不同的机器系统，int 的长度也不同。因此，使用位字段结构体会影响程序的移植性。

7.11.3 位字段结构体的应用

在过程控制、数据通讯、参数检测以及系统软件设计中，常用到位字段。下面的例子是在数据通讯中使用的 RS-232 接口，简称 8251A 接口。该接口定义为如下形式的位字段结构体：

```
struct bit
 {
unsigned bund_bit：2；
unsigned char_bit：2；
unsigned pari_bit：1；
unsigned even_bit：1；
unsigned stop_bit：2；
unsigned dummybit：8；
```

```
} ;
```

下面给出 8251A **接口中状态字的位置和读出方法的例子。**

【例 7-15】8251A **接口中状态字的设置，该例要求从终端输出状态字的各个位字段的值，然后从整体上输出该状态字的八进制值。其程序如下：**

```
struct bit
{
unsigned bund_bit:2;
unsigned char_bit:2;
unsigned pari_bit:1;
unsigned even_bit:1;
unsigned stop_bit:2;
unsigned eummybit:8;
};
main()
{
int input();
unsigned int * pmodeword;
struct bit m;
m.bund_bit = input("bund_bit:0 to 3");
m.char_bit = input("char_bit:0 to 3");
m.pari_bit = input("pari_bit:0 or 1");
m.even_bit = input("even_bit:0 or 1");
m.stop_bit = input("stop_bit:0 to 3");
m.dummybit = 0;
pmodeword = (unsigned int *)& m;
printf("\ n---------------------------------");
printf("\ n The mode word is %o \ n", * pmodeword);
}
input(s)
char * s;
{
int bits;
printf("\ nPlease enter %s:",s);
scanf("%d",&bits);
return(bits);
}
```

该程序的某次运行结果如下：

```
Please enter bund_bit（0 to 3）：3＜CR＞
Please enter char_bit（0 to 3）：1＜CR＞
Please enter pari_bit（0 or 1）：0＜CR＞
Please enter even_bit（0 or 1）：1＜CR＞
Please enter stop_bit（0 to 3）：3＜CR＞
--------------------------------------------------
The mode word is 347
```

程序说明：该程序先调用函数 input（）**向各位字段赋值，然后把位字段结构体变量** mode **的首地址赋给指针变量** pmodeword**。在表达式：**pmodeword=（unsigned int *）& m**；中，使用了强类型转换。因为** pmodeword

是指向 unsigned int 型的指针，而& m 是位字段结构体类型变更的首地址，二者类型不一样，故需要转换。此后，就可以用 printf（）把 pmodeword 指向的数以 8 进制形式输出。

7.12 动态存储分配及其应用

至此，我们已全部介绍 C 语言的功能及其程序设计技术。读者经过上述的，已基本具备 C 语言开发各种应用程序的能力。为了进一步提高读者程序设计的能力，下面将重点讨论动态存储分配及有关应用问题。

7.12.1 动态存储分配

在前面各章中，我们讨论了各种变量，其中包括全局变量和局部变量。这些变量的分配使用了两种技术。第一种是采用静态分配技术，如外部（全局）和静态变量。它们是在编译时由系统为其分配所需空间，而在运行期间始终保持不变。第二种是动态存储分配，例如对局部（自动）变量的存储分配。它们是在程序运行期间，由系统在运行栈为其分配所需要的空间，当使用结束时，就立即释放之。这种动态存储分配，虽然也有效地解决内存的高效利用问题，但是，还不能满足实际的需要。为此，本节讨论含义更广泛的动态存储分配技术。所谓动态存储分配，就是在程序运行期间，根据需要随机地为某种数据结构分配所需要的内存空间，当不需要时，可随时释放所占用的空间。这种分配和释放是受程序控制的，其分配的空间取自系统的自由空间，称之为堆（非运行栈），释放时又归堆空间。

为了实现动态存储分配，各种 C 语言都在标准函数库中提供了相应的标准函数。ANSI 标准建议的函数参考附录 A。本节重点介绍如下几个：

1．malloc（）函数

（1）格式

```
# include<stdlib.h>                    / * 有些系统是 malloc.n * /
void * malloc（size）
unsigned int size；
```

（2）功能

该函数从内存的堆空间中分配大小为 size 字节的内存空间给程序。

（3）参数说明

size：是要求分配内存空间的字节数。

（4）返回值

正常返回：分配空间的首地址。

异常返回：空指针（NULL），当没有足够的堆空间可分配时，返回空指针。

使用该函数应注意如下几点：

- 由于该函数的返回值是 void 型指针。因此，在把返回值赋给具有某种数据类型的指针时，应该对返回值实行强制类型转换。
- malloc（）函数的参数中经常使用 sizeof（）运算来测试数据类型的大小，然后再乘以指定的数据个数。这样做主要是为了能使程序适应不同硬件系统中不同数据类型占用不同的字节数这一情况。
- 在使用该函数分配内存时，经常要检查其返回值是否为 0，以确定所需的空间是否顺利分配。
- 对于 ANSI 标准的内存分配函数，要求使用 stdlib.h 头文件，但有不少系统是使用 malloc.h 头文件。还有的系统使用 malloc.h 和 stdlib.h 两个头文件，如 MSC5.0 编译系统。

【例 7-16】为一结构 addr 分配所需的内存空间。

```
#define NULL 0
# include <stdlib.h>                  / *嵌套头文件 * /
struct address                        / *定义结构体 * /
{
char name[20];
```

```
char street[40];
char city[40];
char state[3];
char zip[10];
}
struct address * get_struct()                    / *  定义结构体指针形函数  * /
{
struct address * pa;                              / *定义结构体指针  * /
pa =(struct address *)malloc(sizeof(struct address));/ *开辟存储空间  * /
if(pa = = NULL)
{
printf("allocation error－aborting");
exit(1);
}
return(pa);                                       / *返回新开辟存储空间起始地址  * /
}
main()
{
struct address * paddr;
paddr = get_struct();
strcpy(paddr- >name,"kane");
printf("\ nThe name is %s",paddr- >name);
free(p);
}
```

2．calloc（）函数

（1）格式

```
# include<stdlib.h>                / *  有许多系统使用 malloc.h * /
void * calloc（num，size）
unsigned int num；
unsigned int size；
```

（2）功能

该函数为程序分配 num *size **字节大小的内存空间。**

（3）参数说明

size**：表示一个实体所占用的字节数。**

num**：表示实体个数。**

（4）返回值

正常返回：被分配空间的首地址。

异常返回：如果没有足够的内存空间供分配时，返回空指针（NULL**）。**

该函数的使用注意事项同 malloc**（）函数。**

【例 7-17**】分配存储** 10 **个整数的内存空间。**

```
# include <stdlib.h>
int * get_mem()                                   / *定义指针形函数  * /
{
int * p;                                          / *定义指针变量  * /
p =(int *)calloc(10,sizeof(int));                 / *分配内存空间  * /
if(! p)
```

```
{
printf("allocation failure-aborting");
exit(1);
}
retrun(p);                                    / * 返回所分配空间的起始地址 * /
}
```

3．realloc（）函数

（1）格式

```
# include<stdlib.h>                  / * 有许多系统使用 malloc.h * /
void * relloc（ptr，size）
void * ptr；
unsigned int size；
```

（2）功能

该函数将由 ptr 指出的已分配的存储空间之长度改变为 size。size 的值可以比原来已分配的存储区长度大或者小。为了增加存储区长度，原分配的存储区可能进行必要的移动。如果发生移动，原来空间的内容会自动拷贝到新移动的区域内，以保证不会丢失信息。由于这种原因，该函数应返回一个改变后的空间首地址。

（3）参数说明

ptr：指向原分配空间的首地址。

size：改变后的新空间的长度。

（4）返回值

正常返回：新空间的首地址。

异常返回：如果没有足够的空间供分配，则返回空指针，且原来已分配的空间作废。因此，调用该函数后，测试调用是否成功是非常重要的。

【例 7-18】realloc（）函数的应用

```
# include <stdlib.h>
main()
{
char * s;
s =(char *)malloc(18);
if(! s)
{
printf("allocation error-aborting");
exit(1);
}
strcpy(s,"my first name is");
s =srealloc(s,24);
if(! s)
{
printf("allocation error-aborting");
exit(1);
}
strcat(s,"kane!");
printf("%s",s);
free(s);
}
```

此程序的执行结果为：

```
my first name is kane!
```

4. free（）函数

（1）格式

```
# include<stdlib.h>                    / *  许多系统使用 malloc.h * /
void free（ptr）
void * ptr；
```

（2）功能

该函数向堆空间交还由 ptr 所指出的存储空间。ptr 所指出的空间必须是先前用 malloc（）函数或 calloc（）分配的空间。否则，可能会产生错误。交还到堆中的空间以后可以再次进行分配。

（3）参数说明

ptr：指出要交还（释放）空间的首地址。

（4）返回值：无

【例 7-19】为用户的输入字符串分配空间，从键盘输入，从屏幕输出，然后释放之。

```
#define NULL 0
# include <stdlib.h>
main()
{
char *string[80];
int i;
for(i=0;i<80;i+ +)
{
if((string[i] =(char *)malloc(128))= = NULL)
{
printf("allocation error-aborting");
exit(1);
}
gets(string[i]);
}
/ * now free the memory * /
for(i = 0;i<80;i+ +)
{
puts(string[i]);
free(string[i]);
}
}
```

7.12.2 动态数据结构及链表

1. 动态数据结构

在前面的章节中，讲到过各种数组，其中有变量数组、指针数组和结构数组等。这些数组由系统根据说明在编译时（对全局和静态类型的数组）或在程序运行时（对局部类型的数组）对其分配大小固定的内存空间。该空间在程序运行期间（或更确切地说是在其生命中），始终保持不变。也就是说，数组中能容纳的数据个数是不变的。我们把类似数组的这种数据结构称之为静态数据结构。在静态数据结构中，各数据元素的位置是固定不变的。因此，可以有效地访问它们中的任何一个元素。但是在数组这种结构中，要想删除和插入一个数据元素是比较困难的，因为它可能引起大量数据的移动，而且数据元素的插入还受到它所占用内存空间大小的限制。

在实际应用中，经常需要这样一种数据结构，该数据结构所占用的内存空间大小在程序运行期间可以动态变化。这类数据结构称之为动态数据结构。在动态数据结构中，其数据元素个数可以随机变化（增加或减少）。

当元素增加时，就增加占用的空间；元素减少时，就减少占用的空间。而且，这些数据元素在逻辑上是连续排列的，但在物理上并不占用连续的存储空间。动态数据结构中比较典型的例子是链表和二叉树。本章将把它作为重点加以介绍。

2．链表

链表是动态数据结构中最典型的一种，它有多种形式，下面主要介绍单链表。为了便于叙述我们把单链表简称链表，其数据结构形式如图 7-4 所示。

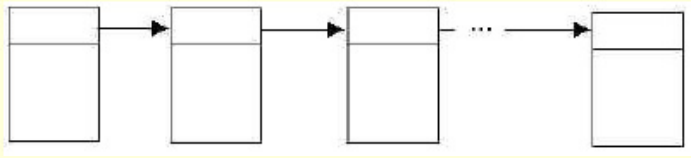


图 7-4　链表的数据结构

组成链表的每一个元素称为一个结点。每个结点具有相同的数据结构类型，它们都是由两部分组成：数据部分和指向下一个结点指针。在 C 语言中，我们可以用动态分配技术和递归结构类型来建立和处理链表。例如，处理学生成绩的链表结点，可以是如下形式的结构类型：

```
struct student
 {
long num；
float score；
struct node * next；
}
```

其中，结构成员 next 是指向同一个结构类型的指针。所以说这种结构类型是一个递归结构。使用递归结构的链表示意图如图 7-5 所示。

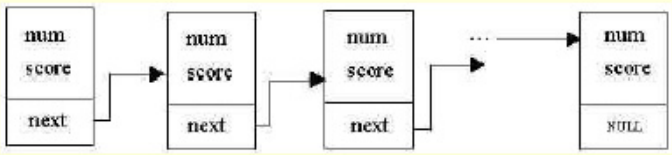


图 7-5　用递归结构表示的链表

下面以学生成绩为例，分别讨论链表的建立、遍历、结点插入和删除。

（1）链表的建立

首先为建立链表定义三个指针：struct student * phead,* ptail,* pnew。

其中：

phead：是指向链表结点的指针

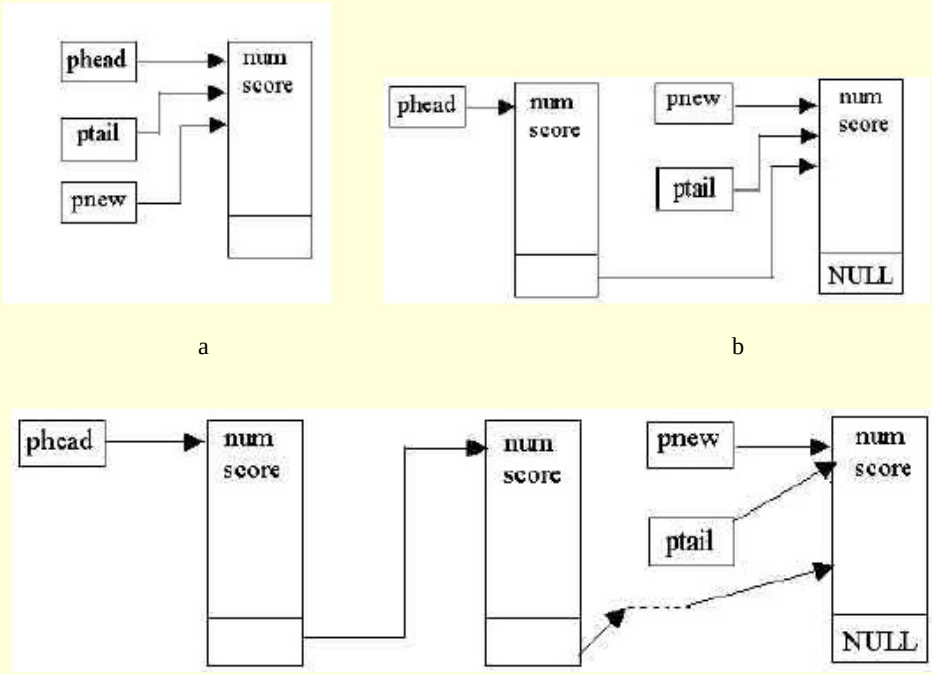
ptail：是指向链表尾结点的指针

pnew：是指向新插入结点的指针

建立表头结点的过程：

- 用 malloc（）函数为头结点分配存储空间，并使指针指向它。如图 7-6a 所示。

```
phead=（struct student * ）malloc（sizeof（struct student））；
ptail=phead；
pnew=phead；
```



a b

c

图 7-6 链表的创建过程

- **通过指针** pnew **把数据赋给链头结点，并把** NULL **赋给** link。

scanf（" % ld % f"，& pnew->num，& pnew->score）；

pnew->next=NULL；

建立其他结点的过程：

- **为结点分配空间**

pnew=（struct student * ）malloc（sizeof（struct student））；

- **通过** pnew **向新结点赋数据：**

scanf（" % ld % f"，& pnew->num，& pnew->score）；

- **将新结点链接在链表上：**

ptail->next=pnew；

- **调整链尾指针**

ptail=pnew；

一个新的结点建立结束，如图 7-6b **所示。**

最后重复建立其他结点的过程，直至把链表完成。其程序如下：

```
# define NULL 0                    / * 宏定义 * /
struct student
{
long num;
float score;
struct score;
struct student * next;
};
struct student * creattab(n)       / *定义结构体指针变量 * /
int n;                             / *结点数 * /
{
int i;
struct student * pnew, * phead, * ptail; / *定义结构体指针变量 * /
```

```
pnew = (struct student *)malloc(sizeof(struct student));/ *创建新节点 * /
phead = pnew;
ptail = pnew;
printf("\ nEnter number and score:");
scanf("%ld %f",&pnew - >num,&pnew- >score);       / * 输入节点内容 * /
pnew - >next = NULL;
for(i = 1;i<n;i+ +)
{
pnew =(struct student *)malloc(sizeof(struct student));/ * 创建新的存储空间 * /
printf("\ nEnter number and score:");
scanf("%ld %f",&pnew- >num,&pnew- >score);        / * 插入新节点内容 * /
pnew- >next = NULL;
ptail - >next = pnew;
ptail = pnew;
}
return(phead);                                    / * 返回链表头指针 * /
}
```

（2）遍历链表

遍历链表就是查询链表上的任意结点。现在以输出链表中的内容为例来看一下遍历的方法。其程序如下：

```
#define NULL 0
findtab（phead）
struct student * phead                        / * 定义链表头指针 * /
 {
struct student * p;                           / * 定义遍历指针 * /
p=phead；
printf（"num              score \ n"）；
printf（"-------------------------\ n"）；
while（p !=NULL）
 {
printf（" % ld % f \ n"，p->num，p->score）；
p=p->next；                                   / * 使指针指向下一个结点 * /
}
}
```

在该函数中，phead **是调用函数传递过来的链表首地址。根据该地址就可以找到链表的头结点。头结点处理完成后，可通过：**

```
p=p->next；
```

来使指针指向下一个结点。继续处理下一个结点，直至结束。因为，对于链表中的尾结点成员 next **中的内容是** NULL**，所以可用它来判断结束。**

（3）插入结点

链表建成后，经常需要指定结点后加入一个新的结点。插入新的结点的关键是与链表上插入点前后结点的连接问题。如图 7-7 **所示。**

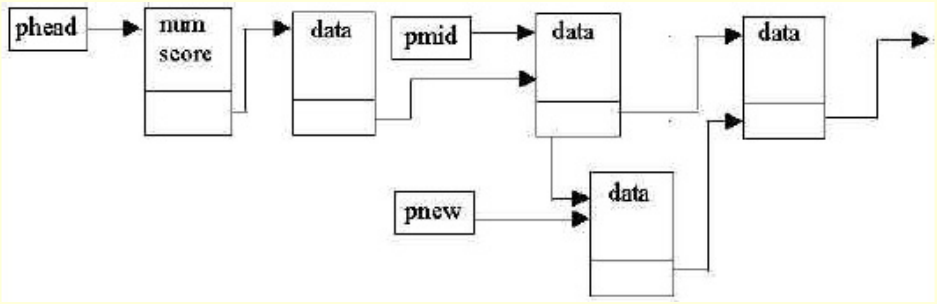


图 7-7　链表点的插入

其程序如下：

```
insert（pmid，pnew）
struct student * pmid，* pnew；
 {
pnew- >next=pmid- >next；                    / * 新结点与后结点相连 * /
pmid- >next=pnew；                           / * 新结点与前结点相连 * /
}
```

其中 pmid 指针指向插入位置，pnew 指向新插入的结点。

（4）删除一个结点

删除指定结点的程序如下：

```
delete(phead,pdel)                                  / * 创建删除节点函数 * /
struct student * phead, * pdel;                      / * 定义要删除的节点指针 * /
{
struct student * temp;
if(phead = = pdel)
phead = pdel - >next;
else
{
temp = phead;
while(temp - > next! = pdel)
if(temp - > next! = NULL)
temp = temp - >next;                  / * 检查下结点是否为删除结点 * /
else
{
printf("Can not find! \ n");          / * 没找到要删结点 * /
return(0);
}
temp - > next = pdel - > next;          / * 找到要删除的结点 * /
return(1);
}
}
```

在该函数中，phead 是链头指针，pdel 指定要删除的结点。当删除正常完成时，返回 1；当删除的结点不存在时，返回 0。

7.13 综合应用实例

【例 7-20】定义一个结构体变量（包括年、月、日）。计算该日在本年中是第几天?注意闰年问题。

方法一：

变量说明：结构 date **中的各元素对应于输入的年、月、日。**days **为天数。**

流程图如图 7-8 **所示。**

```
/ * 计算天数 * /
struct
{ int year;
   int month;
   int day;
}   date;
main（）
{   int days;
printf（"请输入日（年，月，日）"）；
scanf（"%d,%d,%d",&date.year,&date,month,&date.day）;
switch（date.month）
{   case  1:   days=date.day;        break;
    case  2:   days=date.day+31;    break;
    case  3:   days=date.day+59;    break;
    case  4:   days=date.day+90;    break;
    case  5:   days=date.day+120; break;
    case  6:   days=date.day+151; break;
    case  7:   days=date.day+181; break;
    case  8:   days=date.day+212; break;
    case  9:   days=date.day+243; break;
    case  10: days=date.day+273; break;
    case  11: days=date.day+304; break;
    case  12: days=date.day+334; break;
}
if（（date.year%4= =0 && date.year%100! =0
‖ date.year%400= =0）&&date.month>=3）
days+=1;
printf（"\n %d 月 %d 日是 %d 天.",date.month,date.day,date.year,days）;
}
```

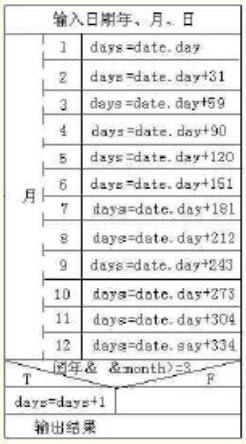


图 7-8 【例 7-20】流程图

运行结果：

请输入日期（年，月，日）1990，9，20↙

9 **月** 20 **日是** 1990 **年的第** 263 **天**.

方法二：

```
struct date{
int year;
int month;
int day;
} date;
main（）
{    int i,day-sum;
     static int day-tab[13]={0,31,28,31,30,31,30,31,31,30,31,30,31}
printf（"请输入年，月，日：\n"）;
scanf（"%d,%d,%d",&date.year,&date.month,&date.day）;
day-sum=0;
for（i=1;i<datre.month;i++）
day-sum+=day-tab[i];
day-sum+=date.day;
if（（date.year%4= =0 &&date.year%100!=0 ‖ date.year%400= =0）
&& date.month>=3）
day-sum+=1;
printf（"%d 月%d 日是  %d 的第%d 天"，date.month,date.day,day-sum）;
}
```

运行结果：

请输入年，月，日：

1990，2，10↙

2 **月** 10 **日是** 1990 **的第** 41 **天**

【例 7-21**】写一个函数** days**，实现上面的计算。由主函数将年、月、日传递给** days **函数，计算后将日数传回主函数输出。函数** days **的程序结构基本与【例** 7-20**】相同。**

方法一：

```
/ *  计算天数. * /
struct dt
{    int year;
     int month;
```

```
    int day;
}   date;
days（date）
struct dt date;
{
int sum;
switch（date.month）
{  case  1:  sum=date.day;        break;
   case  2:  sum =date.day+31;    break;
   case  3:  sum =date.day+59;    break;
   case  4:  sum =date.day+90;    break;
   case  5:  sum =date.day+120; break;
   case  6:  sum =date.day+151; break;
   case  7:  sum =date.day+181; break;
   case  8:  sum =date.day+212; break;
   case  9:  sum =date.day+243; break;
   case  10: sum =date.day+273; break;
   case  11: sum =date.day+304; break;
   case  12: sum =date.day+334; break;
}
if（（date.year%4= =0 && date.year%100!=0
‖date.year%400= =0）&& date.month>=3）
sum+=1;
return（sum）;
}
main（）
{  printf（"请输入日期（年，月，日）：\n"）;
scanf（"%d,%d,%d",&date.year,&date.month,&date.day）;
printf（"\n"）;
printf（"%d 月%d 日是 %d 年的第 %d 天.",date.month,date.day,date.year,days（date））;
}
```

运行结果：

```
请输入年，月，日：
1990，2，10↙
2 月 10 日是 1990 年的第 41 天.
```

方法二：

```
struct dt{
int year;
int month;
int day;
} date;
main（）
{   int i,day-sum;
    printf（"请输入年，月，日:\n"）;
    scanf（"%d,%d,%d",&date.year,&date.month,&date.day）;
    printf（"\n %d 月%d 日是%d 的第%d 天"，date,month,date.day,date.year,days
```

```
（date.year,date.month,date.day））；
}
days（year,month,day）
int year,month,day;
{    int day-sum,i;
     static int day-tab[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};
day-sum=0;
for（i=1;i<month;i++）
  day-sum+=day;
if（（year%4= =0 && year%100!=0 ‖year%400= =0）&&month>=3）
day-sum+=1;
return（day-sum）;
}
```

运行结果：

请输入日期（年，月，日）：

1991，2，15↙

2 **月** 15 **日是** 1991 **的第** 46 **天**

【例 7-22**】将一个链表按逆序排列，即将链头当链尾，链尾当链头。**

方法一：流程图如图 7-9 **所示。**

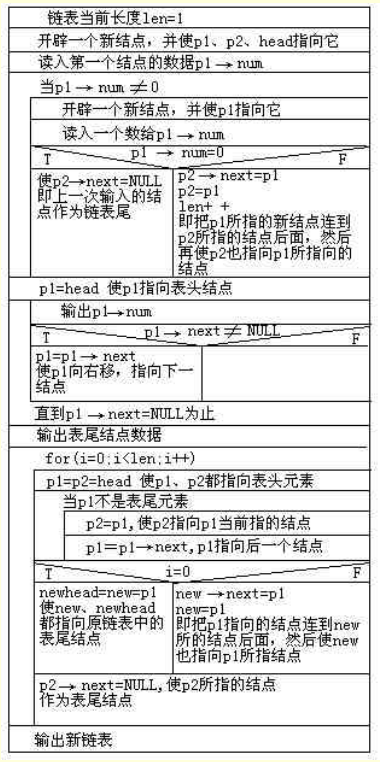


图 7-9 **【例** 7-22**】流程图**

```
#define NULL 0
struct line
{   int num;
    struct line * next;
}
main（）
{   int len=1,i;
```

```
    struct line * p1, * p2, * head, * new, * newhead;
p1=p2=head=（struct line * ）malloc（sizeof（struct line））;
printf（"请输入链表中各结点的数据（输入 0 代表结束）：\n 结点值：")；
scanf（"%d",&p1－>num）;
while（p1－>num!=0）
{ p1=（struct line * ）malloc（sizeof（struct line））;
printf（"结点值：")；
scanf（"%d",&p1－>num）;
if（p1－>num= =0）
  p2－>next=NULL;
else
{
p2－>next=p1;
p2=p1;
len++;
}
}
p1=head;
printf（"\n"  原链表： "\n"）;
do{
printf（"%4d",p1－>num）;
if（p1－>next !=NULL）
p1=p1－>next;
}while （p1－>next != NULL）;
printf（"%d",p1－>num）;
for（i=0;i<len;i++）
{
p2=p1=head;
while（p1－>next != NULL）
p2=p1,p1=p1－>next;
if（i= =0）newhead= new=p1;
else new = new－>next =p1;
p2－>next=NULL;
}
printf（"\n\n 新链表 ：\n"）;
p1=newhead;
for（i=0;i<len;i++）
{printf（"%4d",p1－>num）;
 p1=p1－>next;}
printf（"\n"）;
}
```

运行情况如下：

```
请输入链表中各结点的数据（输入 0 代表结束）：
结点值：12↙
结点值：6↙
结点值：9↙
```

结点值：32↙
结点值：7↙
结点值：0↙
原链表：
12 6 9 32 7
新链表：
7 32 9 6 12

说明：在建立链表时，当输入完链表中所有结点的数据后输入 0，表示链表至此结束。最后输入的 0 不包括在链表中。

方法二：

在看懂上面程序基础上，可以参考下面的程序。在下面的程序中分成几个函数分别完建、按逆序重新排列和输出的功能。同时不用输入“0”来控制建表的结束，而用输入一个或多个字符（用“end”或其他任意字符串均可）。当一个 scanf 函数成功地执行时返回值为 1，否则为 0，因此当向一个整型变量 p1→num 输入一个字符串时，scanf 函数返回 0，用它来控制建表的结束。

```
#define NULL 0
struct line
{int num;
 struct line * next;
} * p1, * p2;

struct line * creat（）
{int temp;
struct line * head=NULL;
printf（"\n 请输入链表（非数表示结束）\n 结点值："）；
while（scanf（"%d",&temp））
{p1=（struct line * malloc（sizeof（struct line））；
  （head= =NULL）?（head=p1）:（p2－>next=p1）；
 p1－>num=temp;
 printf（"结点值："）；
p2=p1;
}
p2－>next=NULL;
return（head）；
}
output（outhead）
struct line * outhead;
{
for（p1=outhead;p1!=NULL;printf（"%4d",p1－>num）,p1=p1－>next）；
}
struct line * turnback（head）
struct line * head;
{
struct line * new, * newhead=NULL;
do{   p2=NULL;
      P1=head;
while（p1－>next !=NULL）  p2=p1,p1=p1－>next;
```

```
if（newhead= =NULL）newhead=p1,new=newhead－＞next=p2;
new=new－＞next=p2;
p2－＞next=NULL;
}while（head －＞next !=NULL）;
return（newhead）;
}

main（）
{
struct line * head;
head=creat（）;
printf（"\n 原来表：")；
output（head）;
head=turnbadck（head）;
printf（"\n\n 反转表：")；
output（head）;
}
```

运行情况如下：

```
请输入链表（非数表示结束）
结点值：12↙
结点值：34↙
结点值：9↙
结点值：0↙
结点值：7↙
结点值：6↙
结点值：2↙
结点值：end↙
原来表：12    34   9    0    7    6    2
反转表：2 6    7    0    9    34   12
```

7.14 小 结

（1）结构体的定义

结构体是由多个数据成员组成的集合体，每个成员都有一个不同的名字，其数据类型可以互不相同。结构体的引入为处理复杂的数据提供了有力的手段。

（2）结构体的说明

结构体说明是对其组成的描述，它说明该结构体是由哪些成员所组成，以及这些成员的数据类型。结构体说明并不分配内存，只是定义了一种结构体类型。结构体说明的一般形式如下：

```
struct 结构体名
 {
结构体成员表列；
} ；
```

结构体成员表包含若干成员，每一个成员的形式为：

```
数据类型标识符  变量名
```

成员可以是简单变量、数组、指针、结构体或联合等。结构体可以在函数内说明，也可以在函数内说明，也可以在函数外说明。在函数内说明时，只在该函数内用它来定义结构体变量。

（3）结构体变量的定义

结构体变量的定义就是按照结构体说明中规定的结构体类型来定义变量，并为其分配内存。定义的一般形式如下：

```
[存储类型] struct [结构体名]
[ {
结构体成员表列；
} ] 结构体变量表列；
```

（4）结构体成员的引用

一般不许对结构体变量的整体进行操作，而只能对其成员直接进行操作。有两种引用结构体成员的方法：

- 用结构体成员运算符“ . ”，其一般形式为：

```
结构体变量名.成员名
```

- 用结构体指针，其一般形式为：

```
（* 结构体指针名）.成员名
```

或

```
结构体指针名- >成员名
```

（5）结构体变量的初始化

结构体变量的初始化就是在定义它的同时对其成员赋初值。只能对外部和静态结构体变量初始化。结构体变量初始的一般形式如下：

```
[存储类型] struct 结构体名 结构体变量名= {初始数据} ；
```

（6）结构体数组

结构体数组就是具有相同结构体类型的一组结构体变量的集合体，可以对外部和静态结构体数组初始化。初始化的一般格式为：

```
[存储类型] struct 结构体名 结构体数组名 [] = { {第一个数组元素的初始值} ，
                                              {第二个数组元素的初始值} ，
                                                …
                                              {最后一个数组元素的初始值}
                                                } ；
```

（7）结构体指针

指向结构体变量的指针叫结构体指针。结构体指针定义的一般形式为：

```
[存储类型] struct 结构体名 * 结构体指针名；
```

当把一个结构体变量的首地址赋给结构体指针时，则该指针主指向这个结构体变量。结构体指针的运算同一般指针。

（8）结构体变量可以在函数间传递。传递方式有二种：

- 数据复制方式：调用中实参是结构体变量名，被调用函数定义中的的形参也是一个结构体变量名。
- 用地址复制方式：调用中的实参是结构体变量的首地址，被调用函数定义的形参是结构体指针。如果传递的是结构体数组，则实参是数组名，形参仍是结构体指针。

（9）结构体型函数

如果函数的返回值是结构体变量，则称该函数为结构体函数。结构体型函数定义的一般形式是：

```
struct 结构体名 函数名（ [形参表] ）
[形参定义；]
{
内部数据定义；
执行语句；
}
```

说明的一般形式为：

```
struct 结构体名 函数名（）；
```

（10）结构体指针型函数

如果函数的返回值是结构体变量的首地址，则称函数为结构体指针型函数。结构体指针型函数定义的一般形式为：

```
struct 结构体名 * 函数名（［形参］）
［形参定义；］
{
内部数据定义;
执行语句；
}
```

说明形式为：

```
struct 结构体名 * 函数名（）；
```

（11）位字段结构体

用来表示一个对象属性的二进制位或位组叫位字段。位字段的存取方式有二种：

- 位操作方式：通过 C 语言提供的操作运算符进行操作。
- 位字段方式：通过定义位字段结构体变量来实现。位字段结构体变量定义的方式是：

```
struct ［位字段结构体名］
［｛
结构体成员名表列；
｝］位字段结构体变量名；
```

结构体成员名表包含若干成员，每个成员的格式是：

```
unsigned 位字段名：常量表达式；
```

位字段结构体变量的引用同一般结构体变量。

（12）联合

联合是这样一种数据类型，它准许不同类型和不同长度的变量共享同一块内存。联合定义的一般形式是：

```
［存储类型］union［联合名］
［｛
成员说明表列；
｝］联合变量名表列；
```

其中成员说明表包含若干成员说明。每个成员说明的格式为：

```
数据类型标识符　成员名；
```

数据类型标识符可以是一般类型标识符，也可以是结构体、联合、指针等。联合变量的引用类似结构体变量。

（13）枚举

枚举类型变量就是在定义它是就将它的所有可能取值一一列举出来。枚举类型变量定义的一般形式为：

```
［存储类型］enum［枚举名］［｛元素名表列｝］枚举变量名表；
```

元素名表规定了所有可能的取值，元素名称实际是常量名。因此，不能对其赋值。元素间用逗号隔开。

（14）类型定义 typedef

类型定义就是给已有的类型重新命名，其目的是增强程序的可读性和移植性。类型定义的一般形式是：

```
typedef 已有的类型名 新类型名；
struct year_date
{
int day；
int month；
int year；
}；
```

```
struct student
  {
int id；
char sex；
struct year_date birthday；
} ；
```

在结构体 student 的说明中，就引用了已说明过的结构体 date 来说明 birthday。因此，birthday 具有“ struct year_date ”类型。

习 题

1．编写一个 input( )函数，把一串字符从终端读入到缓冲区 buffer 内，buf 的长度为 100。当输入遇到'#' 或回车时结束，函数返回 buffer 的首地址。

2．如果某班有 35 名学生，共上 4 门课，请编写一个程序，完成以下工作：

（1）从键盘上输入每个学生的学号、姓名、4 门课的成绩。

（2）求出每个学生的总成绩和平均成绩。

（3）求出全班 4 门课各自的平均分数。

（4）输出班上总分最高和最低学生的学号、姓名、总分及平均分数。

3．编写一个程序，在屏幕上显示一行字符串中的重复出现的字符，并按重复的次数由小到大顺序输出，同时要求在每个字符前输出该字符重复出现的次数。

4．编制一档案管理程序，档案内容自己建，工作证号码编码范围为 6 位数要求：

（1）管理内容包括工作证号（6 位数）、姓名、职称（Prof、Assiprof、lecturer）、年龄、工作年限。

（2）输入任意工作证号，显示该职工的情况。

（3）屏幕输出所有职工的情况，按年龄由大到小顺序。

（4）屏幕输出年龄在 50 岁到 60 岁（含 50 岁和 60 岁）、职称为教授的人数。

5．编制一库房管理程序，要求对某些产品进行管理，产品数量不限。要求：

（1）建立库存（文件名 dat1），包括产品编号（001-999）、名称，数量、单价。

（2）显示任意编号产品的库存记录。

（3）遇到无货要有提示。

（4）根据产品数量由大到小进行排序，并显示排序结果。

（5）输入新的库存信息。

（6）删除某些货源。

（7）修改某些货物的库存量。

第八章　标准库函数和文件系统

由于 C 语言本身并不提供输入/输出机制，程序中的输入/输出功能完全是由函数来实现。这些输入/输出由 C 语言的编译系统提供。这些 I/O 函数加上其他功能的函数的集合构成了 C 的标准函数库。标准库中的每一个函数都以目标文件的形式存放。用户的源程序经编译后生成目标文件，再用连接装配程序将些目标文件与标准库中的有关函数相连接生成可执行文件。

标准函数库中不仅包含各种输入/输出函数，而且还配备有各种常用的数学函数、字符及字符串处理函数、内存管理及分配函数等。

C 语言标准函数库中的输入/输出函数都是面向文件处理的。因此，在介绍这些函数之前，首先讨论有关文件的概念。

8.1　文件概述

8.1.1　C 语言文件的概念

文件是程序设计中的重要概念。一般来说，文件是数据的集合体。例如，程序文件是程序代码的集合体，数据文件是处理数据的集合体。文件的物理存储位置是外部存储设备，例如硬盘、软盘、磁带、屏幕等。对文件的处理过程就是对文件的读写过程，或输入/输出过程。

计算机对文件的输入/输出过程是通过操作系统进行管理的。C 语言中文件的处理是通过标准函数库中的输入/输出函数来实现的。在 C 语言中，所有的外部设备均被作为文件对待，这种文件称为设备文件。对外部设备的输入/输出处理就是读写设备文件的过程。例如，在把打印机作为设备文件（系统命名为 PRN 文件）时，向 PRN 文件的输出信息就是向打印机设备输出打印信息；当把显示屏幕作为设备文件（系统一般称为标准输出文件）时，向标准输出文件输出的信息就是向显示设备输出的信息；当把键盘作为设备文件（系统一般称为所有的设备文件（硬盘、软盘、打印机、显示屏幕、键盘…）都作为相同的逻辑文件对待，从而对它们的输入/输出都可采用相同的方法进行。

C 语言中的设备文件有标准文件和一般文件。其中标准文件包含如下三个文件，它们由系统分配和控制：

- 标准输入文件：系统分配为键盘。
- 标准输入文件：系统分配为显示器。
- 标准错误输入文件：系统分配为显示器。

C 语言把文件看作是字符（或字节）序列，即文件是由一个个字符（或字节）数据顺序组成。如果文件是由一个个字符数据组成，则称之为文本文件（text）；如果是由一个个字节数据组成，则称之为二进制文件。C 语言文件是一个个字符流或二进制流，也就是说它是由一串字符（或字节）所组成，而不是由记录所组成。因此，C 语言文件也叫流式文件。流式文件允许对一个字符进行存取，这就增加了处理的灵活性。

在 C 语言的过去版本中，有两种文件的处理方法。一种叫“缓冲文件系统”（有时也叫格式化系统、文本系统或高级系统），另一种叫“非缓冲文件系统”（有时也叫非格式化系统、二进制系统或类 UNIX 系统）。“缓冲文件系统”是指系统为这类文件的处理自动在内存开辟一个确定大小的缓冲区，使输入/输出都先通过缓冲区过渡，以提高效率。“非缓冲文件系统”是系统不自动开辟确定大小的缓冲区，而是由程序为每个文件设定一个缓冲区，但目前的 ANSI C 提倡使用“非缓冲文件”。

8.1.2　文件类型指针

文件类型指针是“缓冲文件系统”的一个重要概念。我们知道 C 语言程序可同时处理多个文件，为了对每

个文件进行有效的管理，就需要为其开辟一个“文件信息描述区”，以记录文件的当前状态（例如文件名、文件状态等）。该信息描述区是用一个结构变量来的，该结构变量叫文件结构变量。文件结构变量的类型是由系统定义，并取名为 FILE。通常它被存放在“stdio.h”头文件中。FILE 类型的格式各个编译系统可能有微小差别，但大致如下：

```
struct_iobuf
 {
int_cnt；                  / * 输入缓冲区中等待取的字符个数 * /
int_fd；                   / * 文件号，或通道号 * /
int_ flag；                / * I/O 错误标志，若有错则该标志为非 0 * /
char * _base；             / * 指向 I/O 缓冲区的首地址 * /
char * _ptr；              / * 指向当前所取的字符 * /
char * _fname；            / * 指向文件识别名 * /
…
}
typedef struct_Iobuf FILE；
```

对于每一个要操作的文件，都必须定义一个指针变量，并使它指向该文件结构变量，该指针叫文件类型指针。于是可通过该指针找到被操作文件的描述信息，进而对其进行读写。

8.1.3 文件的处理过程

C 语言文件的处理过程主要有打开文件、读/写文件和关闭文件三种操作。

文件打开时，系统自动为该文件定义一个文件结构变量（即文件信息描述区），使该文件与其对应的文件结构变量建立联系。程序中可定义一个文件类型指针，使它指向该文件结构变量。此后对该文件的读写处理都是通过指向该文件结构变量的指针进行。文件打开是由标准函数 fopen（）实现。指向文件结构变量的指针可按如下形式定义：

```
FILE * 文件类型指针名；
```

如果程序中同时要处理几个文件，则应该定义几个文件类型指针，例如：

```
FILE * fp1，*fp2，…，*fpn；
```

其中 fp1，fp2，…fpn 是文件类型指针。这些指针中的地址值由函数 fopen（）提供，例如：

```
fp3=fopen（"filename","r"）；
```

由于标准文件是受系统控制的，其打开和关闭都是由系统自动实现，用户程序不用控制其打开和关闭。三个标准文件的文件类型指针由系统命名如下：

stdin：标准输入文件的文件类型指针

stdout：标准输出文件的文件类型指针

stderr：标准错误输出文件的文件类型指针

这三个文件类型指针不需用户定义或说明，可直接使用。

文件打开后，就可通过其文件类型指针进行读写了。例如：

```
gete（fp2）
putc（fp3）
…
```

文件读写结束时，需关闭该文件。关闭文件就是释放所分配的文件结构变量，使文件与文件结构变量的联系切断。

8.2 一般文件的打开和关闭

对于标准文件进行输入/输出时，不需要用户程序打开和关闭文件，就可以直接引用标准文件输入/输出函数

进行输入/输出。但是，对于一般文件的读/写，必须先打开它，然后才能读/写，读/写完成后还应关闭该文件。一般文件的打开用 fopen（）函数，而关闭用 fclose（）函数。本节介绍这两个函数。

8.2.1 文件的打开函数

（1）格式

```
FILE * fopen（pname，mode）
char * pname；
char * mode；
```

（2）功能

该函数按照 mode 指定的方式打开 pname 指向的文件。

（3）参数说明

pname：是指向文件名字符串的指针，该文件名即为要打开的文件。

mode：是指向文件处理方式字符串的指针，处理方式字符串给出文件打开后的处理方式，各种方式的含义如表 8-1 所示。

（4）返回值

当文件正常打开时，系统为该文件定义“文件结构变量”，然后把该结构变量的地址作为返回值返回。若文件不能打开，则返回 NULL。

（5）使用注意事项

- 在程序中必须用一个文件结构类型的指针变量来接收 fopen（）函数的返回值。例如，程序中要打开一个名为 testf 的文件，其处理方式为“只读”时，可这样处理：

```
FILE * fp；
fp=fopen（"testf"，"r"）；
```

文件打开可以利用文件类型指针得到对该文件进行处理所需要的各种信息了。在 C 语言中，大多数文件的输入/输出函数者是通过文件类型指针来处理的。

- 在程序中，使用 fopen（）函数打开文件时，一般要检查打开的正确性，以便确定程序能否继续执行下去。例如：

```
if（fp=fopen（"testf"，"r"）= =NULL）
 {
printf（"This file can not opened! \ n"）；
exit（i）；
}
```

其中 exit（）函数是停止程序的执行，使控制返回操作系统。通常返回值为 0 时，表示正常返回，非 0 表示非正常返回。

- 表中所列的文件打开方式是 ANSI C 标准的规定，但目前使用的系统，有的提供了这些功能，有的可能只提供一部分。请读者使用时注意。

表 8-1 文件处理方式

mode	处理方式	指定文件不存在	指定文件存在	含义
r	只读	出错	正常打开	为输入打开一个文本文件
w	只写	建立新文件	文件原有内容丢失	为输出打开一个文本文件
a	追加	出错	在文件原有内容后追加	为追加打开一个文本文件
rb	只读	出错	正常打开	为输入打开一个二进制文件
wb	只写	建立新文件	文件原有内容丢失	为输出打开一个二进制文件
ab	追加	出错	文件原有内容后追加	为追加打开一个二进制文件
r+	读写	出错	正常打开	为读/写打开一个文本文件
w+	读写	建立新文件	正常打开	为读写建立一个新的文本文件（先写后读）
a+	读写	出错	正常打开	为读写打开一个二进制文件
rb+	读写	出错	正常打开	为读写打开一个二进制文件
wb+	读写	建立新文件	正常打开	为读写打开一个二进制文件
ab+	读写	出错	正常打开	为读写打开一个二进制文件

8.2.2 文件关闭函数

（1）格式

```
int fclose（fp）
FILE * fp；
```

（2）功能

该函数关闭 fp 所指向的文件。我们知道，在打开时系统为打开的文件分配一个文件结构变量；而关闭时，则释放该文件所拥有的文件结构变量。

（3）参数说明

fp 是文件类型指针。

（4）返回值

正常返回时，返回值为 0；当关闭发生错误时，则返回值为 EOF。可以用 ferror（）函数来测试。

（5）使用注意事项

- 当文件不再读/写时，应注意及时关闭文件，这样可以及时释放系统的资源（文件结构变量）。而且，当程序结束时，如果缓冲区尚未满，而又未关闭文件，则停留在缓冲区中的数据就会丢失。
- 如果用 fclose（）函数关闭文件，则会把尚未装满的缓冲区中的数据写入文件，可以避免数据丢失。

8.3 一般文件的读写

文件打开之后，就可对它进行读写了。用于读写的函数有三种：字符、字符串和格式化输入/输出函数。它们都要求使用 stdio.h 头文件。这些函数都是面向一般文件进行输入/输出处理的，下面介绍这些函数。

8.3.1 一般文件的字符输入/输出函数

1．一般文件的字符输入函数

（1）格式

```
int fgetc（fp）          int getc（fp）
FILE * fp；              FILE * fp；
```

（2）功能

该二函数都是从 fp 所指定的文件中读取一个字符（即一字节代码值）。

（3）参数说明

fp 是一个文件类型指针，它指定要读的文件。

（4）返回值

正常返回：返回读取的字符代码。

非正常返回：返回 EOF（当文件结束时也返回 EOF）。

【例 8-1】编制程序，运行该程序后可以显示指定文件的内容，本例要求被显示文件的文件名在执行时的命令行中给出。例如，假设编译后生成 typefile.exe 文件，则命令行的格式为：

```
typefile filename.txt＜CR＞
```

其程序如下：

```
/ * File display program。File Name is typefile.c * /
# include <stdio.h>
main(argc,argv)
int argc;
char * argv[];
{
```

```
int ch;
FILE *fg;
if ((fp = fopen(argv[1],"r")= = NULL)
{
printf("\ 7The file:%s can not be lpened \ n",argv[1]);
exit(1);
}
ch = fgetc(fp);
while(ch ! = EOF)
{
putchar(ch);
ch=fgetc(fp);
}
printf("argc is %d",argc);
fclose(fp);
}
```

程序执行时，argc **等于** 2**，说明命令行有两个参数，其中第一个是“** typefile **”，第二个是“** filename.txt **”。数组指针** argv［0］**指向字符串“** typefile **”，而** argv［1］**指向“** filename.txt **”。在对文件进行处理时，首先打开该文件，把分配给它的文件结构变量首地址赋给文件类型指针** fp**。如果文件打开失败，则显示“** The file：filename can not be opened **”，并停止该程序执行，控制返回操作系统。字符‘** \ 7 **’为使机器的蜂鸣器发音。如果文件成功打开，则用** fgetc（）**函数每次读一个字符，用标准文件输出函数** putchar（）**把输入的字符显示在屏幕上。当读到文件结束时，关闭该文件，程序执行到此结束。该程序的功能类似于** MS-DOS **中的** type **命令。**

从上例可以看出，文件处理是通过文件结构类型指针完成的（下面的其他几个输入/输出函数也是如此）。如果 fgetc **函数使用标准输入文件类型指针** stdin**，即：**

```
fgetc（stdin）
```

则它与 getchar（）**函数功能完全相同。实质上** getchar（）**是如下形式定义的宏：**

```
# define getchar（）fgetc（stdin）
```

它被放在 stdio.h **头文件中。从用户的角度来看，使用** getchar（）**要比使用** fgetc（stdin）**方便，通常人们也把它叫作函数，而不称它为宏。**

2．一般文件的字符输出函数

（1**）格式**

```
int fputc（ch.fp）          int putc（ch.fp）
int ch；                    int ch；
FILE * fp；                 FILE * fp；
```

（2**）功能**

该函数是把字符 ch **写入** fp **所指定的文件中。**

（3**）参数说明**

ch**：是一个整型变量（或字符常量），其中包含一个要输出的字符。**

fp**：是文件类型指针。**

（4**）返回值**

正常返回值：要写入文件中的字符代码。

错误返回值：EOF**。**

标准输出文件的字符输出函数 putchar（ch）**与** fputc（ch，stkout）**功能相同。实质上** putchar（ch）**是如下定义的宏：**

```
# define puchar（ch）fputc（ch，stdout）；
```

其中，stdout **是标准输出设备文件的文件类型指针。另外，**putc **和** getc **的功能与** fputc **和** fgetc **的功能一般**

是相等的。它们其实是把 fputc 和 fgetc 定义为宏名 putc 和 getc（在 stdio.h 中）：

```
# define putc（ch，fp）fputc（ch，fp）
# define getc（fp）fgetc（fp）
```

【例 8-2】文件复制程序，将文件 infile 中的内容复制到文件 outfile 中。假设该程序的执行文件为 copyfile.exe，现要求执行该程序的命令行格式如下：

```
copyfile infile.txt outfile.txt＜CR＞
```

其中：infile.txt 是被复制的源文件，outfile.txt 是复制后生成的新文件。该例的程序如下：

```
/ * file copy program。The file Name is copyfile.c * /
# include <stdio.h>
main(argc,argv)
int argc;
char * argv[];
{
int ch;
FILE * in, * out;
if(argc! = 3)
{puts("\ n Error in format,Usage:copyfile filenamel filename2");exit(1);}
if((in = fopen(argv[1]，"r")) = = NULL)
{printf("\ n The file %s can not be opened \ n",argv[1]);exit(0);}
if(out = fopen(argv[2],"w"))= = NULL)
{ printf("\ n The file %s can not be opened \ n",argv[2]);exit(0);}
ch = fgetc(in);
while(ch! = EOF)
{fputc(ch,out);ch = fgfetc(in);}
fclose(in);
fclose(out);
}
```

该程序把一个文件复制为另一个文件。程序一开始首先检查命令行格式是否正确。按要求 argc 应该等于 3，如果不等于 3 则给出提示信息。然后程序打开复制的源文件和目标文件。如果打开有错误，则程序停止执行，控制返回操作系统。打开正确后，开始文件复制。复制结束时，关闭两文件。该程序的功能类似于 MS-DOS 中的 copy 命令。

【例 8-3】显示文件的 8 进制代码，并显示其对应的字符。若代码对应的字符是非显示字符，则显示“ # ”。该程序的功能类似于 DEBUG 调试工具中的 dump 功能。

假设该程序的可执行文件为 dumpfile.exe，则要求程序有如下形式的命令行：

```
dumpfile filename＜CR＞
```

该程序如下：

```
# include <stdio.h>
main(argc,argv)
int argc;
char * argv[];
{
char str[9];                    / * 保存一行字符 * /
int ch,i,count;                 / * 行号 * /
FILE * FP;
if(argc! = 2)
{puts("\ 7 Error in foumat,Usage:dumpf filename"); / * 输出提示字符串 * /
```

```
exit(1);
}
if(fp = fopen(argv[1],"r"))= = NULL)
{printf(" \ 7 The file %s can not be opened \ n",argv[1]);
exit(1);
}
count = 0;
do
{
i = 0;
printf("%06o:",count * 8);
ch = fgetc(fp);
while(ch! = EOF)
{
printf("%4o",ch);
if(ch< ' '  ||  ch > 0x7e)
str[i] = '#';                                   / *  记录非显示字符  * /
else
str[i] = ch;
if(+ + i = = 8)break;
}
str[i] = ' \ 0';
if(i! = 8)
for(;i<8;i + +)
printf("    ");                                 / *  输出两个空格字符  * /
printf("%s \ n",str);
count + +;
}while(ch! = EOF);
fclose(fp);
}
```

该程序的某次执行结果如下：

```
dumpfile exam8_1.c＜CR＞
000000：43 151 156 143 154 165 144 145 # include
000010：40 40 74 163 164 144 151 157 <stdio
000020：56 150 76 121 11 155 144 151 h>main…
…
```

8.3.2 一般文件的字符串输入/输出函数

1．字符串输入函数

（1）格式

```
char fgets（str，n，fp）
char * str；
int n；
FILE * fp；
```

（2）功能

从指定的文件中读取一字符串。

（3）参数说明

str：接收读取到的字符串的内存地址。可以是指针，也可以是数组。

n：限制读取的字符个数。

fp：指定读取的文件。

（4）返回值

正常返回：读取到的字符串首地址。

读到文件尾或出错时，返回 NULL。

（5）注意事项

用 fgets（）函数读取字符串时，当下列条件之一满足时，该字符串结束：

- 已读取 n-1 个字符。
- 读取到＜CR＞字符。
- 读到文件尾。

读取结束后，fgets（）再向 str 所指的缓冲区送一个字符‘\0’。当读取到＜CR＞字符时，fgets（）也把＜CR＞字符送到 str 所指的缓冲区。

当 fgets（）函数使用 stdin 作 fp 参数时，其功能与 gets（）函数并不相同。最大区别是：gets（）函数把读取到的＜CR＞字符转换为‘\0'字符，而 fgets（）函数则把读取到的＜CR＞字符也作为字符串中的一个字符传送，然后再追加‘\0'字符。

2．字符串输入函数

（1）格式

```
int fputs（str，fp）
char * str；
FILE * fp；
```

（2）功能

该函数把 str 所指定的字符串写入 fp 所指定的文件中。

（3）参数说明

str：指定输出的字符串，它可以是指针、数组名或字符串常量。

fp：指定输出文件。

（4）返回值

正常返回：返回输出的字符个数。

错误时：返回 EOF。

（5）注意事项

如果用标准输出文件的文件结构变量指针 stdout 作为 fputs（）函数中的 fp，这时 fputs（）与 puts（）函数在功能上是有区别的。主要不同在于 fputs（str，stdout）舍去字符串结尾字符‘\0’，而 puts（str）则将字符串结尾字符‘\0’转换为＜CR＞字符输出。下面是 fgets（）和 fputs（）函数应用的两个例子。

【例 8-4】有两个文本文件，编写程序实现把第二个文本文件接在第一个文本文件后面的功能。假设程序的可执行文件为 linkfile.exe，两个被连接的文件为 file1 和 file2，连接生成后的新文件为 file1。要求执行该程序的命令行格式如下：

```
linkfile file1 file2＜CR＞
```

程序如下：

```
# include <stdio.h>
# define SIZE 512
main(argc,argv)
int argc;
char * argv[];
{
int i;
```

```
char buffer[SIZE];
FILE * fp1, * fp2;
if(argc! = 3)
{printf("\ n Error in format,Usage:linkfile filename1 filename2");exit(0);}
if((fp1 = fopen(argv[1],"a")= = NULL)
{printf("The file:%s can not opened \ n",argv[2]);exit(0);}
if((fp2 = fopen(argv[2],"r"))= = NULL)
{printf("The file:%s can not opened \ n",argv[2]);exit(0);}
while(fgets(buffer,SIZE,fp1)! = NULL)
fputs(buffer,fp1);
fclose(fp1);
fclose(fp2);
if((fp1 = fopen(argv[1],"r"))= = NULL)
{printf("The file: %s can not opened \ n",argv[1]);exit(0);}
while(fgets(buffer,SIZE,fp1)! = NULL)
printf("\ n %s",buffer);
fclose(fp1);
}
```

8.3.3 一般文件的格式化输入/输出函数

与标准设备文件的格式化输入/输出函数 printf（）和 scanf（）相对应，一般文件也有格式化输入/输出函数 fprintf（）和 fscanf（）。

1．一般文件的格式化输出函数

（1）格式

fprintf（fp，“输出格式描述串”，输出项表列）

FILE * fp;

（2）功能

该函数将输出项表列中的各项，按照指定的格式输出到 fp 所指定的文件中。

（3）参数说明

fp：是文件类型指针，指定输出文件。

输出格式描述符串：给出输出格式，格式与 printf（）函数相同。

输出项表列：指定输出项，各项用逗号分开。

（4）返回值

无。

由上述可以看出，函数 fprintf（）与函数 printf（）在格式和功能上基本相同，不同之处仅是输出方向不同。printf（）函数是向标准输出文件输出，而 fprintf（）是向 fp 所指定的文件输出。如果 fp 用“ stdout ”替代时，则 fprintf（stdout…）与 printf（…）功能完全相同。fprintf（）函数向文件输出的是字符（即 ASCII 码），对数值型数据输出时，它首先把二进制的数据转换为字符，然后才输出到文件。例如：

```
b=13；
fprintf（fp."% d "，b）；
```

首先把 a 中的二进制数据（00000…01101）转换为字符‘ 1 ’和‘ 3 ’，然后将其 ASCII 代码 49 和 51 输出。如果要把数值写入文件，应该用其他函数，如 putw（）等。

【例 8-5】编写程序，实现显示文件内容、存储和为文件加行号的功能，如果实现此功能的程序的执行文件为 display.exe，则要求执行该程序的命令行格式为：

```
display［-i］［-s］filename＜CR＞
```

其中［-i］和［-s］是两个可选项，指定这两个可选项的意义如表 8-2 所示。

表 8-2 -p 和-n 选项的意义

可选项	命令行	显示的内容
不指定-i 和-s	display filename<CR>	只显示 filename 文件的内容
指定选项-s	display -s filename<CR>	显示并存储 filename 文件的内容
指定选项-I	display -i filename<CR>	对显示文件的每一个加行号
同时指定-i 和-s	display -i -s filename<CR>	把文件的每一行加行号后显示和存储

该例题的程序如下：

```
# include <stdio.h>
# define SIZE 100
main(argc,argv)
int argc;
char * argv[];
{
char bufffer[SIZE], * filename;
FILE * in, * out;
int n,count,sflag,iflag;
sflag = 0;
iflag = 0;
if(argc < 2 ┃┃ argc > 4)
{
puts ("\ 7 Command line error,Usage should be:display[-i][-s]filename");
exit(i);
}
for(n=1;n<argc;n+ +)
{
if(! strcmp(argv[n],"-s"))       sflag = 1;
else if(! strcmp(argv[n],"-i")) iflag = 1;
else                                filename = argv[n];
}
if((in = fopen(filename,"r")) = = NULL)
{printf("\ 7 Ther file:%s can not opened \ n",filename);exit(0);}
if((out = fopen("save.txt","w"))= = NULL)
{printf("\ 7Ther file:save.txt can not opened \ n");exit(0);}
conut = 1;
while(fgets(buffer,SIZE,in)! = NULL)
{
printf("%3d:%s",count,buffer);              / *  显示  * /
if (sflag = = 1)
fprintf(out,"%s",buffer);                    / *  储存  * /
}
count + +;
}
fclose(in);
fclose(out)
}
```

其中：

sflag：储存标志

iflag：：加行号标志

count：行计数器

2. 一般文件的格式化输入函数

（1）格式

fscanf（fp，“ 输入格式描述串 ”，输入项表列）

FILE * fp；

（2）功能

该函数从 fp 所指定的文件中按照指定格式读入数据到相应的输入项列表中。

（3）参数说明

fp：指定输入文件，是文件指针。

“ 输入格式描述串 ”：scanf（）函数。

输入项且列：把从指定文件读入的数据赋给相应的输入项。各输入项之间用逗号分开。

（4）返回值

无。

fscanf（）函数与 scanf（）函数在功能和使用格式上基本相同，唯一不同之处在于两个函数的输入源，scanf（）函数是从标准输入文件输入，而 fscanf（）函数是从 fp 所指定的文件输入。如果 fscanf（）函数中的 fp 是 stdin，则 fscanf（stdin，…）与 scanf（…）功能完全相同。fscanf（）和 fprintf（）函数的应用如下例。

【例 8-6】从键盘输入一个学生的姓名、班级、年龄、学号，读入文件 student.txt 中，再从该文件读取这些资料，在屏幕上输出：

```
/ * fscanf() and fprintf() example * /
# include < stdio.h>
main()
{
char name[20],class[20];
int age;
long number;
FILE * fp;
if((fp = fopen("student.txt","w"))= =NULL)     / *  打开输出文件  student.txt * /
{puts("\ 7 \ n This file can not be opened");exit(0);}
fscanf(stdin,"%s%s%d%ld",name,class,&age,&number);    / *从键盘输入  * /
fprintf(fp,"%s %s %4d %8ld",name,class ,age,number);       / *  写入 student.txt 文件  * /
fclose(fp);
if((fp = fopen("studen.txt","r"))= =NULL)                        / *  打开输入文件  * /
{puts("The input file can not be opened");exit(0);}
fscanf(fp,"$s%s%d%ld",name,class,&age,&number);         / *  从 student.txt 文件读  * /
fprintf(stdout,"%s %s %4d %8ld",name,class,age,number);    / *  在屏幕上输出  * /
fclose(fp);
}
```

该程序首先将从键盘上输入的某学生的姓名、班级、年龄、学号写入文件 student.txt 中。然后再把从 student.txt 中读的该学生的这些信息显示在屏幕上。其某次执行结果如下：

```
"Rose""class 1"18 991155<CR>      / *  键盘输入  * /
Rose class 1 15 991155                  / *  屏幕显示  * /
```

在“ fscanf（fp，" % s % s% d % d"，name，class，& age，& number）；”语句中，从文件读取的是 ASCII 码（即字符），对于数值数据来说，把读取的字符数据先转换为二进制再赋给变量 age 和 number。

由于上所述可知：fprintf（）和 fscanf（）函数使用方便，但由于用它们进行数值型数据的输入或输出时，

要进行二进制转换，如果输入/输出的数据量大时，将会明显降低执行速度。因此，对磁盘文件进行数值型数据的输入/输出时，最好使用 fread（）和 fwrite（）函数。

8.3.4 二进制形式的输入/输出函数

前面介绍的输入/输出函数均是按字符格式进行输入/输出，实际上，大量的输出需要以二进制的形式进行，下面介绍以二进制形式输入/输出的函数。

1．fread 函数

（1）格式

```
int fread（buffer，size，number，fp）
数据类型标识符 * buffer；
unsigned size，number；
FILE * fp；
```

（2）功能

该函数从 fp 所指定的文件中以二进制形式读取数据块。

（3）参数说明

buffer：是指针型变量，指出读入数据存放区域的首地址。

size：一次读入的字节数。

number：读的次数。

fp：文件指针。

例如：

```
fread（a，4，5，fp）；
```

从 fp 所指定的文件中一次读取 4 个字节，共读取 5 次，读的结果存入 a 缓冲区中。如果读的文件中存放的是实数，则一次读取一个实数，共读取 5 个实数，a 可视为数组。又如：

```
struct student
 {
int number；
char name［20］；
｝stu［30］；
…
for（i=0；i=30；i+ +）
    fread（& stu［i］，sizeof（struct student），1，fp）；
```

一次从 fp 指定的文件中读取一个学生的信息，每执行一次 fread（）函数只读取一个学生的信息。

（4）返回值

正常返回值：返回 number 的值，即读取多少次 size 字节。

遇文件结束或发生错误，则返回 0。

2．fwrite（）函数

（1）格式

```
int fwrite（buffer，size，number，fp）；
数据类型标识符 * buffer；
unsigned size，number；
FILE * fp；
```

（2）功能

该函数将 buffer 缓冲区中的数据以二进制形式写入 fp 所指定的文件中。

（3）参数说明

buffer：是指针型变量，指出一个缓冲区的首地址，该缓冲区中存放要输出的数据。

size：要输出的字节数。

number：调用一次该函数要求写 num 次 size 字节数据。

fp：指定输出文件。

例如：

```
struct student stu［30］；
for（i=0；i<30；i+ +）
    fwrite（& stu［i］，sizeof（struct student），1，fp）；
```

每调用一次该函数，就输出一个学生信息，即输出一个结构变量。

（4）返回值

正常返回值：number 的值。

异常返回值：0，表示文件输出结束或出错。

【例 8-7】现有一个公司工人的信息的磁盘文件 file1，请把该文件存入另一个文件 file2 中。其程序如下：

```
# include <stdio.h>
# define SIZE 4                    / *宏定义* /
struct worker                      / *定义结构体 * /
{
int number;                 / *  结构体成员  * /
char name[20];
int age;
};
main()                    / *  主函数  * /
{
struct worker wk;       / *  定义结构体变量  * /
int n;
FILE * in ,* out;         / *  定义文件类型指针  * /
if((in = fopen("file1.txt","rb")) = = NULL)       / *    打开文件  * /
{puts("The fil1 can not be opened \ n");exit(0);}
if((out = fopen("file2.txt","wb")) = =NULL)    / *  打开文件* /
{puts("The file2 can not be opened \ n");exit(0);}
while(fread(&wk,sizeof(struct worker)),1 ,in)     / *  读文件  * /
{
fwrite(&wk,sizeof(struct worker),1,out);    / *  写文件* /
}
fclose(in);                                  / *  关闭文件  * /
fclose(out);                                 / *  关闭文件  * /
}
```

该函数每次以二进制形式从 file1 文件中读取一个结构类型（worker）的数据，然后把该数据写到 file2 文件中，并显示在屏幕上。当 fread（）函数的返回值不等于 1 时，就意味着读出错，或读结束，此时结束循环。

3．getw（）函数

（1）格式

```
int get w（fp）
FILE * fp；
```

（2）功能

该函数从 fp 指定的文件中以二进制形式读取一个整数值。

（3）参数说明

fp：是文件指针。

（4）返回值

正常返回值：所取的二进制整数。

异常返回值：EOF（即-1），**表示文件结束或出错。**

get w（）**函数并非** ANSI C **标准规定的函数，如果** C **编译系统没有提供该函数，则可用如下的方法定义：**

```
get w（fp）
FILE * fp；
 {
int i；
char * str；
str = & i;
str［0］=getc（fp）；          / * 取正数的第一个字节 * /
str［1］=getc（fp）；          / * 取正数的第二个字节 * /
return（i）；
}
```

有的 C **编译系统虽提供了该功能的函数，但函数名不一定是** get w（），**请用户注意。**

4. putw（）**函数**

（1**）格式**

```
int put w（n，fp）
int n；
FILE * fp；
```

（2**）功能**

该函数以二进制形式将一个 int **型的数据写到** fp **所指定的文件中。**

（3**）参数说明**

n**：是要写到** fp **所指文件中的整型数据。**

fp**：是文件指针。**

（4**）返回值**

正常返回值：输出的整数。

异常返回值：EOF**（输出错误）。**

该函数也非 ANSI C **的标准函数，如果所用** C **编译系统没有提供该函数，则可按如下方法定义它：**

```
int put w（n，fp）
int n；
FILE * fp；
 {
char * str；
str=& n；
putc（str［0］，fp）；          / * 输出整数的第一字节 * /
putc（str［1］，fp）；          / * 输出整数的第二字节 * /
return（n）；
}
```

与 get w（）**函数类似，有的** C **编译系统虽提供了该功能的函数，但名字不一定是** put w（）。

8.3.5 文件状态检查函数

1. 文件读/写结束检查函数

在用 fgetc（fp）**或其他函数读文件时，当返回值是** EOF**（即-**1**）时，表示文件结束。这个结论只能适合于字符形式的读，因为** ASCII **编码中没有编码为-**1 **的字符。如果是以二进制形式进行读时，返回的-**1 **也可能是数据。在这种情况下就不能断定返回-**1 **时就一定是文件结束。为了确定是否文件结束，特引入函数** feof（）。

（1**）格式**

```
int feof（fp）
FILE * fp；
```

（2）功能

用该函数判定文件是否结束。

（3）参数说明

fp：**是文件指针。**

（4）返回值

1：**表示文件结束。**

0：**表示文件未结束。**

【例 8-8】**文件复制并显示。**

如果该程序的执行文件为 copyfile，**要求以如下的命令执行该程序：**

```
copy file filename1 filename2<CR>
```

该程序的执行结果是将文件 filename1 **复制到** filename2。**其程序如下：**

```
# include <stdio.h>                    / *  嵌入头文件  * /
main(arge,argv)                        / *  带形参的主函数  * /
int argc;
char * argv[];
{
FILE * in ,* out;                      / *  定义文件类型指针  * /
char ch;
if(afrgc!=3)
{
puts("Error in command! \ n");         / *  错误提示  * /
exit(0);
}
if((in = fopen(argv[1],"r")) = =NULL)
{puts("The input file can not be opened \ n");exit(0);}
if((out = fopen(argv[2],"w")) = = NULL)
{PRINTF("The output file can not be opened! \ n");exit(0);}
while(! feof(in))
{ch = fgetc(in);                       / *  文件复制  * /
putchar(ch);
}
fclose(in);                            / *  关闭文件  * /
fclose(out);
}
```

该程序中就是用 feof（）**函数来判断文件结束的。**

2．ferror（）**函数**

（1）格式

```
int ferror（fp）
FILE * fp；
```

（2）功能

在调用各种输入/输出函数时，可用该函数检查是否出错。

（3）参数说明

fp：**文件指针。**

（4）返回值

正常返回值：0。

调用出错时的返回值：非 0。

由于该函数是用来检查输入/输出函数的每次调用是否有错，因此应该在调用输入/输出函数后立即调用该函数，以检查输入/输出的调用是否正确。例如：

```
ch=fgetc（fp）；
if（ferror（fp））
printf（"Error in I/O! \ n"）；
```

3．clearerr（）函数

（1）格式

```
void clearerr（fp）
FILE * fp；
```

（2）功能

该函数用于将文件的错误标志设置为 0。**因为文件** I/O **发生错误时，其错误标志被置为非** 0，**该值一起保持到再一次调用** I/O **函数（文件打开时），错误标志也设置为** 0 **或** clearerr（）**函数时才改变。**

（3）参数说明

fp：**文件指针。**

（4）返回值

无。

8.3.6 文件定位函数

在每一个文件的文件结构变量中，都有一个指向当前读写位置的指针。在顺序读写一个文件时，每次读写都要修改指针的指向，使它指向下一次要读写的位置。如果想强制这个规律，可以使用文件定位函数，使指针指向特定的位置。

1．rewind（）函数

（1）格式

```
void rewind（fp）
FILE * fp；
```

（2）功能

该函数使文件的读/写位置指针重新指向文件的开头。

（3）参数说明

fp：**文件指针。**

（4）返回值

无。

例如：

```
main（）
 {
FILE * in，* out；
in=fopen（"filename1"，"r"）；
out=fopen（"filename2"，"w"）；
while（! feof（in））
   putc（getc（in），out）；
rewind（fp1）；                          / * 文件指针重定位 * /
while（! feof（in））
    putechar（getc（in））；
fclose（in）；
fclose（out）；
```

```
}
```

该函数在第一次读完 filename1 文件后，文件的读/写位置指针指向文件末尾。执行函数 rewind（fp1）后，文件读/写位置指针又指向文件的开头。于是，不需要重新打开就可以将该文件输出到屏幕上。

2．fseek（）函数

（1）格式

```
int fseek（fp，offse，base）
FILE * fp；
long offset；
int base；
```

（2）功能

该函数使 fp 所指定的文件的读/写位置指针被设置为相对于 base 的相对位移量为 offset 的位置。

（3）参数说明

fp：文件指针

offset：相对位移量（即相对于 base 的位移量）。

base：是计算相对位移量的基点，它可取 0、1 和 2 三个值之一。ANSI C 标准为这三个值规定了如下的名字和含义：

base 取值	命名	含义
0	SEEK_SET	文件开头
1	SEEK_CUR	文件当前位置
2	SEEK_END	文件末尾

例如：

```
fseek（fp，2001L，SEEK_SET）；/ *  将文件读/写位置指针移到距文件开头 200 个字节的位置* /
fseek（fp，150L，1）；/ *将文件读/写位置指针移到距当前位置 150 个字节的位置* /
fseek（fp，-100L，2）：/ *将文件读/写位置指针移到距末尾 100 个字节的位置* /
```

（4）返回值

正常返回：当前指针位置。

异常返回：-1。

利用该函数可实现文件的随机读/写，即可读/写文件中任意指定的字符（字节），而不是按顺序排列的后序字符。下面是利用 fseek（）函数进行随机读写的例子。

【例 8-9】在文件 stufile 中，存有 100 个学生的信息，他们按学号 1、2、3、…、100 的顺序排列。现要求把单号职员的信息显示在屏幕上。

其程序如下：

```
struct stu_type                          / *  定义结构体  * /
{
int num;                                 / *  定义结构体成员  * /
char name[20];
int age;
char class;
}stud;
main()                                   / *  主函数  * /
{
int n;
FILE * fp;                               / *  定义文件类型指针  * /
if((fp = fopen("stufile","rb"))= = NULL)   / *  打开文件  * /
{
puts("The file can not be opened \ n");
```

```
exit(1);
}
for(n=1;n<100;n+ =2)
{
fseek(fp,n *sizeof(struct stu_type),0);       / *  指针移动  * /
fread(&stud,sizeof(struct stu_type),1,fp); / *  读文件  * /
printf("%d%s%d%s \ n",stud.num,stud.name,stud.age,stud.class);
}
fclose(fp);                                   / *  关闭文件  * /
}
```

该程序并非顺序读取学生信息，而是隔一个读一个。

3. ftell（）函数

（1）格式

long ftell（fp）

FILE * fp；

（2）功能

该函数用于取得流式文件当前的读写位置，用相对于文件形状的位移量来表示该位置。

（3）参数说明

fp：**文件指针。**

（4）返回值

正常返回值：位移量。

异常返回：-1，**表示出错。**

例如：

```
n=ftell（fp）；
if（n= = -1）
printf（" \ 7Error \ n"）；
```

8.4 综合应用实例

【例 8-10】**有一磁盘文件“** employee **”，内存放职工的数据。每个职工的数据包括：职工姓名、职工号、性别、年龄、住址、工资、健康状况、文化程度。要求将职工名、工资的信息单独抽出来另建一个简明的职工工资文件。**

结构化流程图如图 8-1 **所示。**

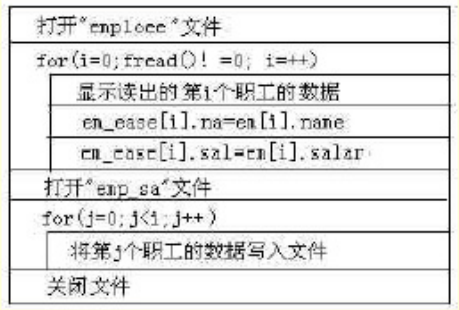


图 8-1 **【例** 8-10】**流程图**

程序代码如下：

```
/ *  建立简明职工文件  * /
```

```
#include <stdio.h>
struct emploee{
char        num[6];
char        name[8];
char        sex[3];
int         age;
char        add[20]'
float       salar;
char        state[8];
char        class[4];
} em[10];

struct emp{
char na[8];
float sal;
} em_ease[10];
main（）
{ FILE * fp1, * fp2;
int i,j;
if（（fp1=fopen（"emploee"."r"））= =NULL）
{   printf（"文件打不开"）；
exit（0）;
}
printf（"\n    职工号    姓名    性别    年龄    住址    工资    健康状况    文化程度\n"）;
for（i=0;fread（&em[i],sizeof（struct emploee）,1,fp1）!=0;i++）
{   printf（"\n%8s%6s%6d%8s%8.2f%8s%8s",em[i].num,em[i].name,em[i].sex,
em[i].age,em[i].add,em[i].salar,em[i].state,em[i].class）;
            strcpy（em_ease[i].na,em[i].name;
em_ease[i].sal= em[i].salar;
}
printf（"\n\n       * * * * * * * * * * * * * * * * * *            "）
fp2=fopen（"emp_sa","w"）;
for（j=0;j<i;j++）
{   fwrite（&em_ease[i],sizeof（struct emp）,1,fp2）;
    printf（"\n       %12%10.2f",em_ease[j].na,em_ease[j],sal）;
}
printf（"\n       * * * * * * * * * * * * * * * * * *            "）;
fclose（fp1）;
fclose（fp2）;
}
```

运行结果如下：

职工号	姓名	性别	年龄	住址	工资	健康状况	文化程度
010	李明	男	25	东城	125.50	健康	大学
011	王亚军	女	40	西城	89.00	一般	大专
012	张力	男	35	朝阳	131.00	良好	中专
013	赵巍	男	50	崇文	155.50	健康	大学

```
* * * * * * * * * * * * * * * * * * *
李明      125.50
王亚军    89.00
张力      131.00
赵巍      155.50
* * * * * * * * * * * * * * * * * *
```

说明：数据文件“ emploee **”是事先建立好的，其中已有职工数据，而“** emp_sa **”文件则是由程序建立的。建立“** emploee **”文件的程序如下：**

```
/ *  建立简明职工文件  * /
#include <stdio.h>
struct emploee{
char            num[6];
char            name[8];
char            sex[3];
int             age;
char            add[20];
float           salar;
char            state[8];
char            class[4];
} em[10];
main（）
{   FILE * fp;
int i,j;
printf（"      职工号  姓名  性别  年龄  住址  工资  健康状况  文化程度\n"）;
for（i=0;i<4;i++）
scanf（"%s %s %s %d %s %f %s %s",em[i].num,em[i].name,em[i].sex,
&em[i].age,em[i].add,em[i].state,em[i].salar,em[i].class）;
/ *  将数据写入文件  * /
fp=fopen（"emploee","w"）;
for（i=0;i<4;i++）
fwrite（&em[i],sizeof（struct emploee）,1,fp）;
fclose（fp）;}
```

【例 8-11**】从上题的简明“职工工资”文件中删去一个职工的数据，再存回原文件。结构化流程图如图** 8-2 **所示。**

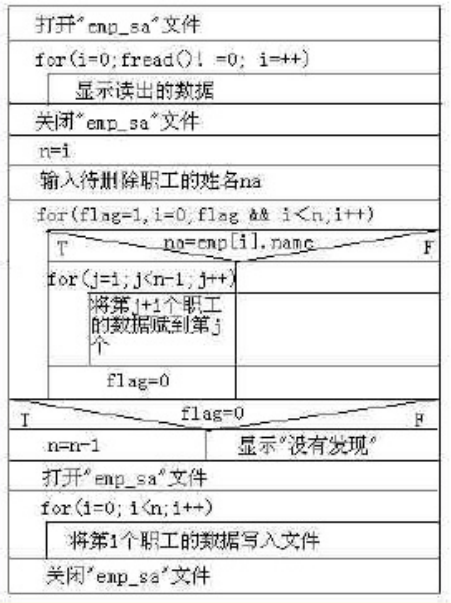


图 8-2 【例 8-11】流程图

```
/ *  删除职工数据    * /
# include <stdio.h>
# include <string.h>

struct emploee{
char name[8];
float salar;
} emp[20],tmp;

main（）
{   FILE * fp;
int i,j,n,flag;
char na[8];
float sa;

if（（fp=fopen（"emp_sa","r+"））= =NULL）
{    printf（"文件打不开"）
exit（0）;
}
printf（"\n 职工文件内容：\n      姓名      工资"）;
for（i= 0; fread（& emp[i],sizeof（struct emploee）,1,fp）!=0; i++）
printf（"\n %8s    %7.2f",emp[i].name,emp[i].salar）;
fclose（fp）;
n=i;
printf（"\n 输入待删除的职工姓名："）
scanf（"%s",na）;

for（flag = 1,i = 0;flag && i<n;i++）
{
if（strcmp（na,emp[i].name）= =0）
```

```
{    for（j+i;j<n－1;j++）
{
strcpy（emp[i].name,emp[j+1].name）;
emp[j].salar = emp[j+1].salar;
}
flag=0;
}
}
if（!flag）
n- =1;
else
printf（"\n 没有发现这个职工 !"）；
printf（"\n 删除后的文件内容：\n    姓名    工资    \n"）;
fp=fopen（"emp_sa","w"）;
for（i=0;i<n;i++）
{   fwrite（&emp[i],sizeof（struct emploee）,1,fp）;
}
fclose（fp）;
fp=fopen（"emp_sa","r"）;
for（i=0;fread（&emp[i],sizeof（struct emploee）,1,fp）  != 0;i++）
printf（"\n8s      %7.2f",emp[i].name,emp[i].salar）;
fclose（fp）;
}
```

运行情况如下：

```
职工文件内容：
姓名      工资
李明      125.50
王亚军    89.00
张力      131.00
赵巍      155.50
输入待删除的职工姓名：李明↙
删除后的文件内容：
姓名      工资
王亚军    89.00
张力      131.00
赵巍      155.50
```

8.5 小 结

本章主要介绍了 C 语言的文件系统和标准库函数。

（1）文件是数据的集合体，C 语言的文件是由一个个字符或字节所组成，因此，称之为流式文件。C 语言对所有设备的输入/输出都归结为对设备文件（简称文件）的输入/输出。C 语言把文件分为标准文件和一般文件。C 语言都把键盘设备文件规定为标准输入文件，而把屏幕文件规定为标准输入文件和标准错误输出文件。把其他文件（如磁盘文件、打印文件、磁盘文件等）规定为一般文件。

（2）C 语言的输入/输出功能是由一系列输入/输出函数实现，这些函数放在标准函数库中。C 语言编译系

统提供的输入/输出函数分为“缓冲文件系统”和“非缓冲文件系统”两类。用“缓冲文件系统”中的函数进行输入/输出时，系统自动为其提供一个缓冲区，而用“非缓冲文件系统”中的函数进行输入/输出时，系统并不提供缓冲区，需要程序为其提供缓冲区。ANSI C 标准规定只用“缓冲文件系统”，而不用“非缓冲文件系统”。

（3）标准函数库中不仅包含所有的输入/输出函数，还包含各种常用的数学函数、字符和字符串处理函数及存储分配函数等。所有这些函数都是以目标代码的形式存放在库中。当用户程序经编译生成目标文件后，可以通过连接装配程序将它与所调用的标准函数库中的函数相连接，生成可执行程序。

（4）对缓冲文件系统来说，为了管理文件的输入/输出，系统为每一个文件都提供一个结构体，结构体中的成员描述了文件的信息。文件结构变量的类型系统命名为 FILE。对于每一个要操作的文件，都必须定义一个文件类型指针，该指针指向要操作文件的文件结构变量。所有的输入/输出都是通过该指针来进行的。

（5）对所有文件的处理都必须经过打开、读/写和关闭三个步骤。打开时为文件分配一个文件结构变量，关闭时则释放这个变量。

（6）在引用每一个标准函数中的函数时，程序设计者必须弄清：函数的功能、调用格式、参数和返回值，以便正确引用。

习 题

1．编一程序，要求：

（1）将 infile.txt 文件显示在屏幕上。

（2）将 infile.txt 文件打印出来；

（3）将 infile.txt 文件拷贝到另一文件 outfile.txt 上；

2．现有一磁盘文件 textfile.txt，请编一程序统计该文件所包含的字母、数字和其他字符的个数。

3．将从键盘上输入的大写字符串变成小写，并加行号显示在屏幕上，然后再输出到磁盘文件 output.txt 中。

4．使用命令行参数编写一程序，分页显示指定文件的内容，每页显示 25 行，按任意键继续显示下一页。

5．编写一程序，从键盘上输入学生的学号、姓名和英语、数学、计算机三门课的成绩，并将这些信息和每人的总分及平均成绩按学号顺序写入磁盘文件 stu1.txt 中。

6．从上题 stu1.txt 文件中读出每个学生的信息按数学成绩降序排序显示在终端上，并写到另一磁盘文件 stu2.txt 中。

7．从键盘上输入一个学生的第 5 题中的信息，按数学成绩降序的顺序插入到 stu2.txt 文件中。

第九章　C 语言的预编译语句

在 C 语言的源程序中，除了说明语句和可执行语句外，还有一种特殊语句，称为预编译语句。预编译语句的作用是告诉编译系统，在正式编译源程序之前，应该首先做些什么预处理工作。这些预编译语句虽不是 C 语言的组成部分，但却扩充了 C 语言的程序设计环境。

ANSI 标准定义的 C 语言包括如下 4 类预编译语句：

（1）文件包括语句

include

（2）宏定义语句

define，# undef

（3）条件编译语句

if - # else - # endif

ifdef - # else - # endif

ifndef - # else - # endif

（4）其他预编译语句

line，# pragma

为了把预编译语句 C 语言的语句区分开来，预编译语句都用符号“ # ”开头。

一个预编译语句占用一行，其结尾不用分号“ ;”作结束。在编码时，一个语句行一般从一行的首列位置开始，“ # ”号前面不留空格。C 编译系统对源程序的处理过程是先进行预编译，然后进行编译。预编译后生成的源文件中就不包含预编译语句了，而完全是 C 语言的语句。

9.1　文件包括语句

在前面各章中，曾经不只一次见到这样的语句：

include <stdio.h>

其含义是在预编译时用 stdio.h 头文件的内容替换该语句。

文件包括语句的一般格式是：

include <filename>

或

include 0169filename0169

其中 filename 是被包括文件的文件名，它是一个磁盘文件。该预编译语句的功能是要将 filename 文件的全部内容包含在该# include 语句所在的源文件中。更确切地说是在预编译时，用 filename 文件的全部内容替换该# include 语句行，使该文件成为这个源文件的一部分。

在# include 语句的书写格式中，被包括文件的文件名（即 filename）可用尖括号（<、>）括住，也可用比引号（0169、0169）括住。当用尖括号时，其含义是指示编译系统按系统设定的标准目录搜索文件 filename。而用双引号括住时，表示按指定的路径搜索；若未指定路径名时，则在当前目录中搜索。例如：

（1）从指定的路径搜索 input.h 文件。

include 0169d：\ user \ include \ input.h0169

（2）先从目录中搜索 input.h 文件。找不到时，再去标准目录中检索。

include 0169input.h0169

（3）从系统设定的标准目录中搜索。

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

文件包括语句是很有用的，特别是对包括多个源文件的大程序来说，可以把各个文件中共同使用的函数说明、符号常量定义、外部量说明、宏定义和结构类型定义等写成一个独立的包括文件、在需要这些说明的源文件中，只需在源文件的开头用一个# include 语句把该文件包括进来即可，这样可避免重复劳动。

在第八章中，我们介绍过在引用输入/输出函数时，需要包括一些头文件（以.h 为后缀的文件）。这些头文件中一般包含输入/输出函数所需要的符号常数、宏和函数说明等。

做成包括文件的另一个好处是：当这些常量、宏定义等需要修改时，只需修改这个包括文件即可，而不必修改各个源文件。

在使用# include 语句时，应注意以下两点：

- 一个# include 语句只能包含一个包括文件。如果需要包括 n 个文件时，就需要 n 个# include 语句。
- 文件包括语句可以嵌套。

9.2 宏定义

C 编译系统的预编译程序提供了宏定义机制，利用这种机制可以定义符号常量和宏。下面叙述符号常量和宏的定义。

9.2.1 符号常量的定义

符号常量用宏定义语句定义，宏定义语句的一般形式如下：

```
# define  符号常量名  字符串
```

该语句把符号常量名定义为指定的字符串。在进行预编译处理时，用该字符串替代程序中出现的符号常量名。例如：

```
# define  TRUE    1
# define  FALSE   0
```

把 TRUE 定义为 1，FALSE 定义为 0。在符号常量定义之后，就可以用它来编码了。例如：

```
if（i= =TRUE）
  printf（0169Congratulations! You are right! \ n0169）；
else if（i= =FALSE）
  Printf（0169Sorry! You are wrong! \ n0169）；
```

对于该程序段，在进行预编译时，就把程序中出现的 TRUE 和 FALSE 分别用 1 和 0 替代，于是就变为：

```
if（i= =1）
  printf（0169Congratulations! You are right! \ n0169）；
else if（i= =0）
  Printf（0169Sorry! You are wrong! \ n0169）；
```

在符号常量定义语句中，字符串可以是一个数值数据、表达式或字符串。例如：

```
# define   PI      3.14159
# define   S       （PI * r * r）
# define   PRT     printf
# define   A       （20-（3 * 4））
```

如果字符串是一个运算表达式时，一般应该用括号括住它，以便把它视为一个操作对象与其他操作数进行计算，否则，会由于操作优先级的问题而发生错误。例如：

```
text=A *8
```

预编译后，该表达式变为：

```
text=（20-（3 *4））*2
```

如果 A 定义为：

```
# define A 20-3 * 4
```

则表达式 var=EXPR * 8 经预编译后变为：

```
var=20-3 *4 *2
```

这就不符合原意。因此，在宏定义语句中的字符串为一般表达式（而不是一个操作数）时，为了保证正确的运算次序，应该用括号括住它。在程序设计中，使用符号常量有如下两点好处：

（1）增强程序的可读性

由于符号常量含义明确，于是采用符号常量书写的程序要比不用符号常量可读性更强。例如：

```
# define        LENGTH        40
# define        WIDTH         120
# define        HEIGTH        80
```

在程序中使用 LENGTH、WIDTH、HEIGTH 时，一看就知道它们分别代表长、宽、高，而 40、120、80 则很难猜出它们是长、宽、高。

（2）增强程序的可维护性

如果一个常量在程序的多处被引用，可把它定义为符号常量。这样，在以后需改动该常量时，只需改动它的宏定义语句即可，而不必对每一个引用它的地方进行修改。这不但可以减少改错的工作量，而且可以避免漏改。

使用符号常量应注意以下几点：

- 符号常量名一般用大写字母（也可以用小写字母），以便与其他标识符相区别。符号常量名的命名规则同一般标识符一致。
- 宏定义语句不是 C 语言的语句，因此，不能用分号结尾。如果使用分号结尾，则分号被视为字符串的一部分。例如：

```
# define        AGE     50；
```

以上格式表示 AGE 被定义为“ 50 ;”而不是 50。于是，在预编译时，程序中凡是出现 AGE 的地方，都用“ 50 ;”替换。这就不符合原意了。

- 宏定义语句应放在函数定义之外，符号常量的有效范围是从定义它的宏定义语句开始至所在源文件的末尾。一般宏定义语句都放在源文件的开头，以便使它对整个源文件都有效。
- 可用# undef 预编译语句结束宏定义的作用域。例如：

```
# define AGE 30
main（）
 {
…
}
# undef AGE                        / * AGE 为 30 的宏定义结束 * /
func（）
 {
…
}
```

该符号常量的有效域是到“ # undef AGE ”语句为止。

- 在定义符号常量时，可以引用已定义的符号常量。例如：

```
# define     PI       3.1415926
# define     R          10
# define     S          PI * R * R
main（）
 {
printf（0169S=% f0169，S）；
```

```
}
```

预编译后，该程序变为：

```
main（）
 {
printf（0169S=% f0169，3.1415926 * 10 * 10）；
}
```

- 如果程序中用双引号括住的字符串内包含与符号常量相同的名字时，预编译时并不进行替换。例如，上面的 printf（）语句中，预编译时并不替换 0169S= % f0169 中的 S。

9.2.2 带参数的宏定义

利用# define 语句不仅可以定义符号常量，也可以定义带参数的宏。带参数的宏的一般定义形式如下：

```
# define  宏名（参数表）  字符串
```

例如：

```
# define  min（a，b）（（（a）<（b））？（a）：（b））
```

其中 min（a，b）是带参数的宏，a 和 b 是形式参数。该定义把 min（a，b）定义为（((a）<（b)）？（a)：（b))。在定义了该宏之后，就可在程序中用 min（a，b）替代定义它的运算表达式（((a）<（b)）？（a)：(b))。宏的使用方法类似于函数。例如，在需要求两个数的最小值时，就可使用已定义的宏：

```
c=min（20，21）；
```

在进行预编译时，预编译程序根据宏定义式来替换程序中出现的带参数宏，其中定义式中的形式参数用相应的实际参数替换。于是，上面的赋值语句变为：

```
c=（（20＜21）？20：21）；
```

在程序设计中，经常要把反复使用的运算表达式定义为带参数的宏。例如

```
# define   percent（a，b）（1000.0 *  （a）/（b））     / *求 a 是 b 的千分之几* /
# define   abs（x）    （（x）> =0）？（x）：-（x）/   / *求 x 的绝对值 * /
# define   max（a，b）（（（a）<（b））？（a）：（b））  / *求两个数中较大者* /
# define   isadd（x）    （（（x）% 2=1）？0：1）       / *判断是否为奇数 * /
```

引入带参数的宏有如下两点好处：

- 使程序更加简洁，减少不必要的重复书写。
- 增强程序的可读性，一般用一个含义明确的宏名代替一个较复杂的运算式，会使读者一目了然。

从上面的叙述中可以体会到，带参数的宏和函数在使用形式和特性上都很相似。但是，二者又有本质区别，主要表现在：

- 函数调用时，要保留现场和返回点，而后把控制转移给被调用函数。当被调用函数执行结束后，又要恢复现场和把控制返回到调用函数。而对带参数宏的使用不存在控制的来回转移，它只是表达式的运算。
- 函数有一定的数据类型，且数据类型是不变的。而带参数的宏一般是一个运算表达式，它没有固定的数据类型，其数据类型就是表达式运算结果的数据类型。同一个参数的宏，随着使用实参类型的不同，其运算结果的类型也不同。
- 函数定义和调用中使用的形参和实参都受数据类型的限制，而带参数宏的形参和实参可以是任意数据类型。
- 函数调用中存在参数的传递过程，而带参数宏的引用不存在参数传递过程。
- 使用函数可缩短程序占用的内存空间，但由于控制的来回转移，会使程序的执行效率降低。而带参数的宏则相反，多次使用宏会增加程序占用的存储空间，但其执行效率要比函数高。

除了使用运算表达式来定义带参数的宏外，还可使用函数来定义它，标准函数库中经常采用这种方式。例如：

```
# define    getchar（）     fgetc（stdin）
# define    putchar（ch）   fputc（ch，stdout）
```

为了正确地定义和使用带参数的宏，请注意如下几点：

■ 对宏定义语句中的定义和形式参数，要根据需要加上圆括号，以免发生运算错误。例如：

```
# define S（a） （（a）*（a）*（a））
…
a1=5；
a2=10；
s=S（a1+a2）；
```

经过预编译，该赋值语句变为：

```
s=（（a1+a2）*（a1+a2）*（a1+a2））；
```

如果定义式中不使用相应的括号，则预编译后的赋值语句变为：

```
s= a1+a2 * a1+a2 * a1+a2
```

这样和原意根本符合了。所以在定义带参数的宏时，一定要注意加相应的括号。

■ 在定义带参数的宏时，在宏名和带参数的圆括号之间不应包含空格，否则空格之后的字符串将视为替代字符串。例如：

```
# define min（a，b） （（a< =b） ？a：b）
```

将把 min 视为符号常量名，它代表字符串“（a，b）（（a< =b） ？a：b）”。这显然是错误的。下面是几个宏定义的表面积和体积。

【例 9-1】计算球的表面积和体积。

设球的半径为 r。其程序如下：

```
# define    PI    3.14159              / *  宏定义  * /
main（）
{
float r,s,v;
printf（0169Enter r:0169）;
scanf（“% f”,&r）;
s = 4 * PI * r * r;                    / *  引用 PI 的宏定义  * /
v = 3\4 * PI * r * r * r;              / *  引用 PI 的宏定义  * /
printf（0169 \ n Area = % f volume = %f \ n0169,s,v）;
}
```

【例 9-2】将数组 b［］中的 8 个数由大到小排序，要求用带参数的宏实现。

其程序如下：

```
# include <stdio.h>
# define change（a,b）    {int temp;temp = a;a = b;b = temp;}      / *  函数宏定义  * /
main（）
{
static int b[8] = {7,2,15,4,3,21,5,10};
int n,m;
for（n=0;n<7;n+ +）
{
for（m = n+1;m<8;m+ +）
if（b[n]<b[m]）
change（b[n],b[m]）;                                              / *  引用宏定义函数  * /
printf（0169b[%d] = %d0169,n,b[n]）;
}
printf（0169b[%d] = %d0169,n,b[n]）;
}
```

编预编译后 main（）函数变为如下形式：

```
main（）
{
static int b[8] = {7,2,15,4,3,21,5,10};
int   n,m;
for（n=0;n<7;n+ +）
{
for（m = n+1;m<8;m+ +）
if（b[n]<b[m]）
{int temp;temp = b[n];b[n]=b[m];b[m]=temp;};
printf（0169b[%d]=%d0169,n,b[n]）;
}
printf（0169b[%d] = %d0169,n,b[n]）;
}
```

9.3 条件编译

在一般情况下，源程序中的所有行都要参加编译。但是，有时希望参加编译的内容能够有所选择，即希望在满足其一条件时编译某一部分内容，不满足时编译另一部分内容。条件编译是由条件编译语句实现。条件编译语句有如下几种形式：

（1）由# ifdef - # else - # endif 语句实现

其一般形式如下：

```
# ifdef    标识符
程序段 1
# else
程序段 2
# endif
```

这种格式的作用是：当标识符被定义（用# define 定义）时，程序段 1 被编译，而程序段 2 被删除；否则程序段 1 被删除，而程序段 2 被编译。其中# else 部分可以没有，即：

```
# ifdef    标识符
程序段 1
# endif
```

条件编译语句中的# ifdef 和# endif 决定了编译范围，在此范围之外的源程序不存在条件编译问题。条件编译对于提高程序的移植性很有帮助。例如，开发一个应用程序时，希望能在两个不同的硬件环境下运行。这两个硬件环境的主要区别是：一个机器是用 32 位二进制表示一个整数，而另一个机器是用 16 位二进制表示。这时可用如下的条件编译语句组：

```
# ifdef              CONDITION
# define             INT－SIZE     32
# else
# define             INT－SIZE     16
# endif
```

如果想使应用程序在整数为 32 位的硬件环境下运行，可在此条件编译语句组前定义 CONDITION。如果不定义的话，则编译的应用程序只能在整数为 16 位的环境下运行。CONDITION 可定义为任何字符串，例如：

```
# define     CONDITION    CONA
```

这样，程序只需加一条或删一条该# define 语句，其他不做任何修改就能实现在不同的环境下运行。当然这

只是对最简单情况的实现方法，实际情况远不是整数长短的区别，但读者可以根据此思路来设计它。

（2）用# ifndef - # else - # endif 来实现

其一般形式为：

```
# ifndef       标识符
程序段 1
# else
程序段 2
# endif
```

该形式与（1）中所描述形式的惟一区别就是第一行# ifndef 和# endif。该形式的条件编译语句的作用是：当标识符未定义时，编译程序段 1，否则编译程序段 2。例如：

```
# ifndef       UNPRN
  printf（0169Name= % s     salary = % f0169，name，s）；
# else
  printf（0169 % s % f0169，name，s）；
# endif
```

当 UNPRN 在该程序段之前未定义时，则只编译“ printf（0169Name= % s　　salary = % f0169，name，s）;”。如果在该程序段之前加一行：

```
# define      UNPRN       1
```

则只编译“ printf（0169 % s f0169，name，s）;”。其中 UNPRN 可定义为任何字符串。

（3）用# if - # else - # endif 实现

其一般形式如下：

```
# if       表达式
程序段 1
# else
程序段 2
# endif
```

其作用是：当表达式值为非 0 时，编译程序段 1，不编译程序段 2；否则，只编译程序段 2。其中# else 语句可以省略。例如，在程序设计的测试阶段经常需要显示一些变量的信息，以检查是否正确，而在正式执行时，却不要显示这些信息。这时，可以用如下形式的条件编译：

```
# define        DEBUG      1
# if         DEBUG
  printf（0169a= % d b= % f c=% s0169，a，b，c）；
```

以上形式的条件编译适用于测试阶段，如果程序测试完成，要编译正式的执行代码时，只需把 DEBUG 定义为 0 即可。在进行条件编译时，可根据情况决定选择哪种条件编译语句。

9.4　预定义的宏名和其他预编译语句

9.4.1　预定义的宏名

为了有利于编译程序的处理和程序员诊断错误，在 ANSI 标准的 C 编译系统中，规定了五个宏名，它们是：

- LINE：用于指出源文件中语句的行编号
- FILE：用于指定当前的源文件名
- DATE：用于表示源文件被编译时的日期，形式为月/日/年
- TIME：用于表示源文件被编译时的时间，形式为时：分：秒
- STDC：如果实现是标准的，则含有常量 1，否则是非标准实现

如果所用的编译系统不是 ANSI 标准，则可能不支持以上几个宏名，或者只支持其中几个。

9.4.2 # line

该语句用于改变 LINE－和－FILE－的内容。其一般格式为：

line number [0169filename0169]

其中：0169filename0169 是源文件名，指出 FILE 的内容应该为该名，它指出当前要编译的源文件名。该项是可选的，如果缺省，则 FILE 中已记下的文件名不变。

number 为任何正整数，它指定源文件中该# line 语句之后的语句行的当前行号。

line 语句主要用于调试和其他一些特殊应用。例如，下面的# line 语句说明源文件中该位置的行计数从 100 开始；printf（）语句显示数 102，因为它是# line 语句之后的第三行。

```
# line    100                          / * 设置行计数器为 100 * /
main（）                               / * 行号 100 * /
 {                                     / * 行号 101 * /
printf（0169 % d\ n0169，－LINE－）；    / * 行号 102 * /
```

9.5 综合应用实例

【例 9-3】请设计输出实数的格式，包括：(1）一行输出一个实数；(2）一行输出两个实数；(3）一行输出三个实数。实数用“ 6.2f ”格式输出。

程序代码如下：

```
/ * 实数输出格式 * /
# define PR printf
# define NL "\n"
# define Fs"%f"
# define F"%6.2f"
# define F1 F NL
# define F2 F"\t"F NL
# define F3 F"\t"F"\t" F NL
main（）
{
float a,b,c;
PR（"请输入三个实数：\n"）；
scanf（Fs,&a）；
scanf（Fs,&b）；
scanf（Fs,&c）；
PR（NL）；
PR（"每行输出一个数：\n"）；
PR（F1,a）；
PR（F1,b）；
PR（F1,c）；
PR（NL）；
PR（"每行输出两个数：\n"）；
PR（F2,a,b）；
PR（F2,c）；
PR（NL）；
```

```
PR（"每行输出三个数:\n"）;
PR（F3,a,b,c）;
}
```

运行情况如下：

```
请输入三个实数：
2.4↙
5.9↙
9.1↙
每行输出一个数：
2.40
5.90
9.10
每行输出两个数：
2.40    5.90
9.10
每行输出三个数：
2.40    5.90    9.10
```

【例 9-4】设计所需的各种各样的输出格式（包括整数、实数、字符串等），用一个文件名"format.h"，把这些信息都放到此文件内，另编一个程序文件，用#include "format.h"命令以确保能使用这些格式。

程序代码如下：

```
/ * 不同输出格式 * /
/ * format.h 文件 * /
# define DIGIT（d）printf（"整数输出：%d\n",d）
# define FLOAT（f）printf（"整数输出：%8.2f\n",f）
# define STRING（s）printf（"字符串输出：%s\n",s）

/ * 以下为用户自己编写的程序，其中需要用"format.h"文件中的输出格式 * /
# include"fomat.h"
main（）
{
int d,num;
float f;
char s[80];
printf（"请选择输入形式：1-整数，2-实数，3-字符串："）；
scanf（"%d",&d）;
switch（num）
{case 1:printf（"请输入一个整数："）；
scanf（"%d",&d）;
DIGIT（d）;
break;
case 2:printf（"请输入一个实数："）；
scanf（"%f",&f）;
FLOAT（f）;
break;
case 3:printf（"请输入一个字符串："）；
scanf（"%s",s）;
```

```
STRING（s）;
break;
default:printf（"输入错误！"）；
}
}
```

运行结果如下：

①请选择输入形式：1-**整数，**2-**实数，**3-**字符串：**1↙
请输入一个整数：20↙
整数输出：20

②请选择输入形式：1-**整数，**2-**实数，**3-**字符串：**2↙
请输入一个实数：3.79↙
实数输出：3.79

③请选择输入形式：1-**整数，**2-**实数，**3-**字符串：**3↙
请输入一个字符串：student↙
字符串输出：student

④请选择输出形式：1-**整数，**2-**实数，**3-**字符串：**4↙
输入错误！

【例 9-5**】用条件编译方法实现以下功能：**

输入一行电报文字，可以任选两种输出，一为原文输出；一为将字母变成其下一字母（如"a"**变成**"b"**，…，**"z"**变成**"a"**。其他字符不变）。用**#define **命令控制是否要译成密码。例如：**

```
#define    CHANGE   1
```

则输出密码。若

```
#define    CHANGE   0
```

则不译成密码，按原码输出。

程序代码如下：

```
/ *  翻译电码  * /
#include "studio.h"
#define MAX 80
#define CHANGE 1
main（）
{
char str[MAX];
int i;
printf（"请输入文本行：\n"）;
scanf（"%s",str）;
#if（CHANGE）
{
for  （i=0;i<MAX;i++）
{
if  （str[i]!='\0'）
if  （str[i]>='a'&& str[i]<'z' ‖ str[i]>='A'&& str[i]<'z'）
str[i]+=1;
else if（str[i]）= ='z' ‖ str[i]= ='z'）
```

```
str[i]- =25;
}
}
# endif
printf（"输出电码为：\n%s",str）;
}
```

运行结果：

```
请输入文本行：
A-Lazy-Brown-Fox-Jumps-Over-A-Dog.
输出电码为：
B-Mbaz-Cpy-Kvnqt-Pwfs-B-Eph.
```

9.6 小 结

（1）C 语言的所有预编译语句均以“#”开头，不以分号结尾。C 编译系统包括预编译和编译两部分，源程序首先进行预编译处理，然后才进行编译。因此，所有预编译语句的处理都在编译之前完成。

（2）利用宏定义语句可以定义符号常量（也叫不带参数的宏）和带参数的宏。符号常量定义的一般形式：

```
# define      符号常量名      字符串
```

符号常量名也称宏名，在预编译时，将程序中出现的符号常量名用对应的字符串替换，这个过程叫宏替换。

（3）带参数宏定义的一般形式

```
# define      宏名（形参表）      字符串
```

字符串中应包含形参表中的形参。预编译时将带实参的宏名用定义语句中的字符串替换，且形参用相应的实参替换，非形参字符保留不变，这个过程称为宏展开。

（4）不论符号常量和带参数的宏，其宏名的作用域都是从其宏定义语句开始到源文件尾或遇# undef 为止。宏定义中可以引用已定义过的宏名。在宏替换中，对双引号括住的字符串中出现的与宏名同名的字符串一般不进行替换。

（5）文件包括语句的一般形式

```
# include      <文件名>
或
# include      0169 文件名 0169
```

文件名应是 C 语言的源文件名，可以是系统提供的，也可以是用户编写的，通常称为头文件，习惯用“.h”作为后缀名。文件名用尖括号括住表示直接从系统规定的标准目录中检索；用双引号括住表示首先从指定的目录或当前目录中检索，找不到时再从系统规定的标准目录中检索。在预编译时，用# include 语句中指定的文件内容替换该# include 语句。文件包括语句可以嵌套。

（6）条件编译是根据指定的条件是否成立，决定相关的程序段是否被编译。条件编译对程序的测试和提高其可移植性很有帮助。

习 题

1．选出在任何情况下都不会引起二义性的宏定义_____。

（1）＃define POWER（x） （x*x）

（2）＃define POWER（x） ((x) * (x))

（3）＃define POWER（x） x*x

（4）＃define POWER（x） （x）*（x）

2．若有如下定义，则执行 y=5*（a+x（5））;语句后的结果为_____。

```
# define a 4
# define x（b） （（a-3）*b）
```

（1）y=45　　（2）y 无定值　　（3）y=98　　（4）语句有错误

3．输入两个整数，求它们的相除的余数，要求用带参数的宏来实现求余数。

4．定义一个求 a、b、c 三个数中最小的值的宏，并编写一个程序验证该宏的正确性。

5．给出任意一个年份，判断该年是否为闰年，若为闰年请输出该闰年的年号。要求用带参数的宏实现。

6．输入 20 个整数，请分别显示出偶数和奇数。要求用带参数的宏实现判断一个数是偶数或奇数。若是奇数，则宏的值为 0，否则为 1。

附　　录

附录 A　ASCⅡ字符编码表

ASCⅡ值	控制字符	ASCⅡ值	字　符	ASCⅡ值	字　符	ASCⅡ值	字　符
0	NUL	32	(space)	64	@	96	、
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	—	127	DEL

附录 B　C 语言中的关键字

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

附录 C　运算符和结合性

优先级	运算符	含　义	要求运算对象的个数	结合方向
1	（ ） ［ ］ － ＞	圆括号 下标运算符 指向结构体成员运算符 结构体成员运算符		自左至右
2	！ ~ ＋ ＋ － － － （类型） * & sizeof	逻辑非运算符 按位取反运算符 自增运算符 自减运算符 负号运算符 类型转换运算符 指针运算符 地址与运算符 长度运算符	1 （单目运算符）	自右至左
3	* / %	乘法运算符 除法运算符 求余运算符	2 （双目运算符）	自左至右
4	+ －	加法运算符 减法运算符	2 （双目运算符）	自左至右
5	＜＜ ＞＞	左移运算符 右移运算符	2 （双目运算符）	自左至右
6	＜ ＜=＞ ＞=	关系运算符	2 （双目运算符）	自左至右
7	＝ ＝ ！ ＝	等于运算符 不等于运算符	2 （双目运算符）	自左至右
8	&	按位与运算符	2 （双目运算符）	自左至右
9	^	按位异或运算符	2 （双目运算符）	自左至右
10	¦	按位或运算符	2 （双目运算符）	自左至右
11	& &	逻辑与运算符	2 （双目运算符）	自左至右
12	¦ ¦	逻辑或运算符	2 （双目运算符）	自左至右
13	？：	条件运算符	3 （三目运算符）	自右至左
14	＝ +＝ －＝ *＝ /＝ %＝ >>= <<= &= ^= ¦=	赋值运算符	2	自右至左
15	，	逗号运算符 （顺序求值运算符）		自左至右

说明：

（1）同一优先级的运算符优先级别相同，运算次序由结合方向决定。例如，*与/具有相同的优先级别，其结合方向为自左至右，因此，3*5/4 的运算次序是先乘后除。－和＋＋为同一优先级，结合方向为自右至左，因此－ i ＋＋相当于－（i ＋＋）。

（2）不同的运算符要求有不同的运算对象个数，如＋（加）和－（减）为双目运算符，要求在运算符两侧各有一个运算对象（如 3＋5、8－3 等）。而＋ ＋和－（负号）运算符是一元运算符，只能在运算符的一侧出现一个运算对象（如－a、i ＋ ＋、— － i、(float) i、sizeof（int)、*P 等）。条件运算符是 C 语言中惟一的一个三个三目运算符，如 x ? a:b。

（3）从上述表中可以大致归纳出各类运算符的优先级：

初等运算符　()　[]　－＞·

↓

单目运算符
↓
算术运算符（先乘除，后加减）
↓
关系运算符
↓
逻辑运算符（不包括！）
↓
条件运算符
↓
赋值运算符
↓
逗号运算符

以上的优先级别由上到下递减。初等运算优先级最高，逗号运算符优先有最低。位运算符的优先级比较分散（有的在算术运算符之前（如~）有的在关系运算符之前（如＜＜和＞＞），有的在关系运算符之后（如＆、^、|））。为了容易记忆，使用位运算符时可以圆弧号。

附录 D　C 语言常用语法提要

为读者查阅方便，下面列出 C 语言语法中常用的一些部分的提要。为便于理解，没有采用严格的语法定义形式，只是备忘性质，供参考。

1．标识符

可由字母、数字和下划线组成。标识符必须以字母或下划线开头。大、小写的字母分别认为是两个不同的字符。不同的系统对标识符的字符数有不同的规定，一般允许 7 个字符。

2．常量

（1）整型常量

十进制常数。

八进制常数（以 0 开头的数字序列）。

十六进制常数（以 0x 开头的数字序列）。

长整型常数（在数字后加字符 L 或 t）。

（2）字符常量

用单引号（撇号）括起来的一个字符，可以使用转义字符。

（3）实型常量（浮点型常量）

小数形式。

指数形式。

（4）字符串常量

用引号括起来的字符序列。

3．表达式

（1）算术表达式

整型表达式　参加运算的运算量是整型量，结果也是整型数。

实型表达式　参加运算的运算量是实型量，运算过程中先转换在 double 型，结果为 double 型。

（2）逻辑表达式

用逻辑运算连接的整型量，结果为一个整数（0 或 1）。逻辑表达式可以认为是整型表达式的一种特殊形式。

（3）字位表达式

用位运算符连接的整型量，结果为整数。字位表达式也可以认为是整型表达式的一种特殊形式。

（4）强制类型转换表达式

用“（类型）”运算符使表达式的类型进行强制转换。如（float）a。

（5）逗号表达式（顺序表达式）

形式为

表达式 1，表达式 2，……表达式 n

顺序求出表达式 1，表达式 2，……表达式 n 的值。结果为表达式 n 的值。

（6）赋值表达式

将赋值号“＝”右侧表达式的值赋给赋值号左边的变量。赋值表达式的值为执行赋值后被赋值的变量的值。

（7）条件表达式

形式为

逻辑表达式？表达式 1：表达式 2

逻辑表达式的值或为非零，则条件表达式的值等于表达 1 值；若逻辑表达式的值为零，则条件表达式的值等于表达式 2 的值。

（8）指针表达式

对指针类型的数据进行运算。例如，p－2、p1－p2、＆a 等（其中 p、p1、p2 均已定义为指针变量），结果为指针类型。

以上各种表达式可以包含有关的运算符，也可以是不包含任何运算符的初等量（例如，常数是算术表达式的最简单的形式）。

4. 数据定义

对程序中用到的所有变量都需要进行定义。对数据要定义其数据类型，需要时要指定其存储类别。

（1）类型标识符

int
short
long
unsigned
char
float
double
struct 结构体名
union 联合体名
用 typedef 定义的类型名

（若省略数据类型，则按 int 处理）

结构体与联合体的定义形式为：

struct 结构体名
{成员表列}；
union 联合体名
{成员表列}；

用 typedef 定义类型名的形式为：

typedef 已有类型 新定义类型；

如：

typedef int COUNT；

（2）存储类别

auto
static
register
extern

（如不指定存储类别，作 auto 处理）

变量的定义形式为：

存储类别　数据类型　变量表列

例如：

```
static    float    a,b,c;
```

注意：外部数据定义只能用 extern 或 static，而不能用 auto 或 register。

5．函数定义

形式为：

存储类别　数据类型　函数名（形参表列）

形参说明；

函数体

函数的存储类别只能用 extern 或 static。函数体是用花括弧括起来的，可包括数据定义和语句。函数的定义举例如下：

```
static int max（x,y）
int   x,y;
{int z;
z=x＞y?x:y;
return（z）;
}
```

6．变量的初始化

可以在定义时对变量或数组指定初始值。

静态变量或外部变量如未初始化，系统自动使其初值为零（对数值型变量）或空（对字符型数据）。对自动变量或寄存器变量，若未初始化，若未初始化，则其初值为一不可预测的数据。

只有静态或外部数组才能初始化。

例如：

```
static    int    array［4］［2］＝{{1,2},{3,4},{5,6},{7,8}};
```

或

```
static    int    array［4］［2］＝{1,2,3,4,5,6,7,8}；
```

7．语句

（1）表达式语句；

（2）函数调用语句；

（3）控制语句；

（4）复合语句；

（5）空语句。

其中控制语句包括：

① if（表达式）语句

或

if（表达式）　语句 1

else　语句 2

② while　（表达式）　语句

③ do　语句

while　（表达式）；

④ for　（表达式 1；表达式 2；表达式 3）

语句

⑤ switch　（表达式）

{case 常量表达式 1：语句 1；

```
case 常量表达式 2：语句 2；
┆
case 常量表达式 n: 语句 n；
default;语句 n+1;
}
```

前缀 case 和 default 本身并不改变控制流程，它们只起标号，作用，在执行上一个 case 所标志的语句后，继续顺序执行下一个 case 前缀所标志的语句，除非上一个语句最后用 break 语句使控制转出 switch 结构。

⑥ break 语句

⑦ continue 语句

⑧ return 语句

⑨ goto 语句

8. 预处理命令

define 宏名 字符串

define 宏名（参数 1，参数 2，……，参数 n）字符串

undef 宏名

include "文件名"（或＜文件名＞）

if 常量表达式

ifdef 宏名

ifndef 宏名

else

endif

附录 E C 语言的标准函数库

每一种 C 编译系统都提供了一批标准库函数，这些库函数并不是 C 语言的一部分。不同的编译系统所提供的标准函数库的数目、名字和功能是不完全相同的。

ANSI C 标准建议提供的标准函数库包括了目前多数 C 编译系统提供的函数库，但也有一些是某些 C 编译系统未曾实现的。由于 C 库函数的种类和数目很多，限于篇幅，本附录不能全部介绍，只能列出 ANSI C 标准建议提供的、最常用的和最基本的一部分。读者在编制 C 程序时，如果用到更多的函数时，可查阅有关的系统手册。

E.1 数学函数

函数名	函数类型和形参类型	功 能	返回值	说 明
acos	double acos（x） doublex;	计算 $\cos^{-1}$（x）	计算结果	x 应在-1 到 1 范围内
asin	double asin（x） double x;	计算 $\sin^{-1}$（x）	计算结果	x 应在-1 到 1 范围内

续 表

函数名	函数类型和形参类型	功 能	返回值	说 明
atan	double atan（x） double x;	计算 $\tan^{-1}$（x）	计算结果	
atan2	double atan2（x,y） double x,y;	计算 $\tan^{-1}$（x/y）	计算结果	
cos	double cos（x） double x;	计算 cos（x）	计算结果	x 的单位为弧度
cosh	double cosh（x）	计算 cosh（x）	计算结果	

	double x;			
exp	double exp（x） double x;	求 e 的值。	计算结果	
fabs	double fabs（x） double x;	求 x 的绝对值。	计算结果	
floor	double floor（x） double x;	求不大于 x 的最大整	该整数的双精度实数	
fmod	double fmod（x,y） double x,y;	求整除 x/y 的余数	返回余数的双精度数	
frexp	double frexp（val,eptr） double val; int eptr;	把数 val 分解为数字部分（尾数）x 和以 2 为底的指数 n	返回数字部分，n 存放在 eptr 指向的变量中	val=x*2n 0.5x<1
log	double $\log_{10}$（x） double x;	求 $\log_e x$	计算结果	
log10	double $\log_{10}$（x） double x;	求 $\log_{10} x$	计算结果	
modf	double modf（val,iptr） double val; double *iptr	把双精度数 val 分解为 val 的整数部分和小数部分	val 的小数部分，把整数部分存到 iptr 指向的单元	
pow	double pow（x,y） double x,y;	计算 xy 的值	计算结果	
sin	double sin（x） double x;	计算 sinx 的值	计算结果	x 单位为弧度
sinh	double sinh（x） double x;	计算 x 的双曲正弦函数	计算结果	
sqrt	double sqrt（x） double x;	计算 $\sqrt{x}$	计算结果	x≥0。
tan	double tan（x） double x;	计算 tan（x）的值	计算结果	X 的单位为弧度
tanh	double tanh（x） double x;	计算 x 的双曲正切函数的值	计算结果	

E.2 字符函数和字符串函数

ANSI C 标准要求在使用字符串函数时要包含头文件"string.h"，在使用字符串函数时要包含头文件"ctype.h"。有的 C 编译不遵循 ANSI C 标准的规定，而用其它名称的头文件，使用时需要查阅其他的查考手册。

函数名	函数和形参类型	功 能	返回值	包含文件
isalnum	int isalnum（ch） int ch;	查看 ch 是否是字母（alpha）或数字（numeric）	是字母或数字返回 1，否则返回 0	ctype.h
isalpha	int isalpha（ch） int ch;	检查 ch 是否是字母	是返回 1，否则返回 0	ctype.h
iscntrl	int iscntrl（ch） int ch;	检查 ch 是否是控制字符（即代码在 00xlf）之间	是，返回 1，否则返回 0	ctype.h
isdigit	int isdigit（ch） int ch;	检查 ch 是否数字（09）	是，返回 1，否则返回 0	ctype.h
isgraph	int isgraph（ch） int ch;	检查 ch 是否则可打印字符（即代码在 0x21~0x7e）之间空格除外	是，返回 1，否则返回 0	ctype.h
islower	int islower（ch） int ch;	检查 ch 是否是小写字母（a~z）	是，返回 1，否则返回 0	ctype.h
isprint	int isprint（ch） int ch;	检查 ch 是否是可打印字符（即代码在 0x20~0x7e）之间包括空格	是，返回 1	否则返回 0
ispunct	int ispunct（ch） int ch;	检查 ch 是否是标点字符（即除空格、数字、字母之外的所有可打印字符）	是，返回 1，否则返回 0	ctype.h
isspace	int isspace（ch） int ch;	检查 ch 是否是空格、制表符或换行字符。	是，返回 1，否则返回 0	ctype.h

isupper	int isupper（ch） int ch;	检查 ch 是否是大写字母（A~Z）	是，返回 1，否，则返回 0	ctype.h
isxdigit	int isxdigit（ch） int ch;	检查 ch 是否是 16 进制数字符（0~9，A~F 或 a~f）	是，返回 1，否，则返回 0	ctype.h
strcat	char *strcat（str1,str2） char *strh, *str2	把字符串 str2 接到字符串 strl 之后，strl 之后的'\0'取消	str1	string.h
strchr	char *strchr（str,ch） char *str; int ch;	找出字符 ch 在 str 指向的字符串中第一次出现的位置	返回该字符在串中第一次出现时指针的内容；找不到时，返回空指针	string.h
strcmp	char *strcmp（str1,str2） char *strl,*str2;	str1 和 str2 二字符串进行比较	str1<str2, 返回负数；str1=str2,返回 0；str1>str2 返回正数	string.h
strcpy	char *strcpy（strl,str2） char *strl, *str2;	把 str2 所指向的字符串拷贝到 strl 所指向的区域中。	返回 str1	string.h
strlen	unsigned int strlen（str） char *str;	计算 str 所指向的字符串之长度。不包含串结束符'\0'。	返回字符个数	string.h
strstr	char *strstr（sftr1,str2） char *str1,str2;	找出字符串 str2（不包含串结束符'\0'）在串 strl 中第一次出现的位置。	返回 str2 串在 strl 中第一次出现时指针的内容；找不到时，返回空指针	string.h
tolower	int tolower（ch） int ch;	将 ch 字母转换为对应的小写字母	转换后的小写字母代码	string.h
toupper	int toupper（ch） int ch;	将 ch 字母转换为对应的大写字母	转换后的大写字母代码	string.h

E.3 输入和输出函数

凡用以下的输入输出函数，应该使用# include "stdio.h"**把** stdio.h **头文件包含到源程序文件中。**

函数名	函数和形参类型	功 能	返回值	说 明
fseek	int fseek（fp,offset,base） FILE *fp; long offset; int base;	将 fp 所指向的文件的位置指针，以 base 为基准以 offset 为位移量进行移动	返回当前位置，失败时返回-1	
ftell	long ftell（fp） FILE *fp;	返回 fp 指文件的读写位置	返回 fp 所指文件的读写位置	
fwrite	intfwrite（ptr,size,n,fp） char *ptr; unsigned size; unsigned n; FILE *fp;	把 ptr 所指向的 size*n 个字符输出到 fp 所指向的文件内。	写到文件中的数据项个数	
getc	int getc（fp） FILE *fp;	从 fp 所指向的文件内读入一个字符	返回所读的字符；若文件结束或出错，返回 EOF	
getchar	int getchar（）	从标准输入设备文件内读入一个字符	返回所读的字符；若文件结束或出错，返回 EOF	
getw	int getw（fp） FILE fp;	从 fp 所指向的文件内读取下一个字（整数）	返回所读的整数；若文件结束或出错，返回-1	非 ANSI 标准函数
open	int open（filename,mode） dhar *filename; int mode;	以 mode 指定的方式打开已存在的名为 filename 的文件	返回文件号（整数）；若打开失败，则返回-1	非 ANSI 标准函数
printf	int pritf（format,args,…） char *format;	将输出项 args,…按格式 format 输出到标准输出文件中	输出的字符个数；若出错，返回负数	format 指出输出格式，可以是字符串或字符数组名
putc	int putc（ch,fp）	将字符 ch 输出到 fp 所指向	输出的字符出；若出	

	int ch; FILE fp;	的文件中	错，返回 EOF。	
putchar	int putchar（ch） int ch;	将字符 ch 输出到标准输出文件中	输出的字符；若出错，返回 EOF	
puts	int puts（str） char *str;	将 str 指向的字符串输出到标准输出文件中，把'\0'转换为回车换行符	返回换行符；若出错，返回 EOF	
putw	int putw（w,fp） int w; FILE fp;	将一个整数（即一个字）写到 fp 所指向的文件中	输出的整数；若出错，返回 EOF	非 ANSI 标准函数
read	int read（fd,buf,count） int fd; char *buf; unsigned count;	从文件号 fd 所指的文件中读取 count 字节到 buf 所指的缓冲区中	读的字节数；遇文件结束，返回 0；出错，返回-1	非 ANSI 标准函数
rename	int rename（oldname,new-name） char *oldname, *newname;	重新命名文件名	成功返回 0；失败返回-1	
rewind	void rewind（fp） FILE *fp;	将 fp 所指文件的位置指针置于文件开头位置，并清除文件结束标志和错误标志	无	
scanf	int fscan（format,args,…） char *format;	从标准输入文件按格式 format 读取数据到 args…输入项中	正常，返回输入的数据个数；遇文件结束，返回 EOF；错，返回 0	format 指出输入格式，可以是字符串或字符数组名，args 为地址量
write	int write（fd,buf,count） int fd; char *buf; unsigned count;	从 buf 所指的缓冲区中输出 count 个字符到 fd 所指示的文件中	输出的字节数错，返回-1	非 ANSI 标准函数

E.4 动态存储分配函数

函数名	函数和形参类型	功 能	返回值
calloc	void（或 char） *calloc（n,size） unsigned n,size;	分配 n 个数据项所需的连续空间，每个数据项的大小为 size	分配成功时，返回分配区域的开始地址；失败时，返回 0
free	void free（p） void （或 char） *p;	释放 size 字节的内存空间	无
malloc	void（或 char） *malloc（size） unsigned size;	分配 size 字节的内存空间	分配成功时，返回分配区域的开始地址；失败时，返回 0
realloc	void（或 char） *realloc（p,size） void（或 char） *p; unsigned size;	把 p 所指的已分配的内存区的大小改为 size,size 可比原来分配的区域大或小	返回重新分配区域的开始地址