《Linux C 编程从初学到精通》

作者：华清远见

第 2 章 C 语言编程基础

本章目标

C 语言是国际上广泛使用的计算机高级编程语言，C 语言最初用于描述和支持 UNIX 系统，后来逐渐被广大程序员所接受，成为备受欢迎的编程语言。在其后的发展过程中，C 语言不断吸收计算机方面新的成果，使该语言逐渐完善起来。作为 Linux 系统的开发语言，C 语言在 Linux 编程开发中扮演着重要的角色。本章将向读者详细讲解 C 语言的相关编程基础知识。本章内容：

C 语言产生的历史背景。 C 语言的特点。 C 语言的基本数据类型。
运算符与表达式。 C 程序的 3 种基本结构。 C 语言中的数据输入与输出。
函数、数组、指针、结构体和共用体、链表。
位运算符和位运算。 C 语言的预处理命令。

2.1 C 语言的历史背景

C 语言的原型是 A 语言（ALGOL 60 语言）。1963 年剑桥大学将 A 语言发展成为 CPL（Combined Programming Language）语言。1967 年剑桥大学的 Matin Richards 对 CPL 语言进行了简化，于是产生了 BCPL 语言。1969 年美国贝尔实验室的 Ken Thompson 将 BCPL 进行了修改，提炼出它的精华并为它起名为“B 语言”。并且他用 B 语言写了第一个 UNIX 操作系统。而在 1973 年美国贝尔实验室的 D.M.Ritchie 在 B 语言的基础上设计出一种新的语言，他取了 BCPL 的第二个字母作为这种语言的名字，这就是 C 语言。

为了使 UNIX 操作系统得到推广，1977 年 D.M.Ritchie 发表了不依赖于具体机器系统的 C 语言编译文本《可移植的 C 语言编译程序》，即著名的 ANSI C。1978 年由 AT&T（美国电话电报公司）贝尔实验室正式发表了 C 语言。同时由 B.W.Kernighan 和 D.M.Ritchit 合著了著名的《THE C PROGRAMMING LANGUAGE》一书。通常简称为《K&R》，也有人称之为《K&R》标准。但是，在《K&R》中并没有定义一个完整的标准 C 语言，后来由美国国家标准协会（American National Standards Institute）在此基础上制定了一个 C 语言标准，于 1983 年发表。通常称之为 ANSI C。

1987 年，随着微型计算机的日益普及，出现了许多 C 语言版本。由于没有统一的标准，使得这些 C 语言之间出现了一些不一致的地方。为了改变这种情况，美国国家标准研究所（ANSI）为 C 语言制定了一套 ANSI 标准，成为现行的 C 语言标准。

1990 年，国际化标准组织 ISO（International Standard Organization）接受了 87 ANSI C 为 ISO C 的标准（ISO 9899-1990）。1994 年，ISO 修订了 C 语言的标准。目前流行的 C 语言编译系统大多是以 ANSI C 为基础进行开发的，但不同版本的 C 语言编译系统所实现的语言功能和语法规则略有差别。

2.2 C 语言的特点

C 语言之所以能被世界计算机界广泛接受，正是由于它自身具备的突出特点，从语言体系和结构上讲，它与 Pascal、ALGOL 60 等语言相类似，是结构化程序设计语言。归纳起来，C 语言具有下列特点：

C 语言是中级语言，它把高级语言的基本结构和语句与低级语言的实用性结合起来。例如，位、字节和地址是计算机最基本的工作单元，而 C 语言可以像汇编语言一样对这三者进行操作。

C 语言是结构式语言。结构式语言的显著特点是代码及数据的分隔化，即程序的各个部分除了必要的信息交流外彼此独立，这种结构化方式可使程序层次清晰，便于使用、维护及调试。C 语言是以函数作为程序的模块单位，用户可方便地调用这些模块，并可通过多种循环、条件语句控制程序流向，从而使程序完全结构化。

C 语言功能齐全。C 语言提供多种数据类型，能用来实现各种复杂的数据结构，例如通过引入了指针来使程序效率更高。另外 C 语言包含了 34 种运算符，丰富的运算符使其具有强大的计算功能和逻辑判断功能。

C 语言适用范围广。C 语言适用于多种操作系统（如 DOS、UNIX），也适用于多种机型。在对操作系统、系统应用程序及需要对硬件进行操作时，都选择使用 C 语言。而且，用 C 语言编写的程序，只要稍加修改就可移植到不同型号的计算机上。

C 语言对程序员要求也高，程序员用 C 语言编写程序会感到限制少、灵活性大、功能强，但较其他高级语言在学习上要困难一些。上面只介绍了 C 语言的一般特点。相信通过后续章节的实践，读者能够体会到 C 语言更多其他特点。

2.3 C 语言的基本数据类型

在计算机中，数据的性质和表示方式可能不同。所以需要将相同性质的数据归类，并用一定数据类型描述。任何数据对用户都呈现常量和变量两种形式。常量是指程序在运行时其值不能改变的量。常量不占内存，在程序运行时它作为操作对象直接出现在运算器的各种寄存器中。变量是指在程序运行时其值可以改变的量。变量的功能就是存储数据。C 语言的基本数据类型包括整型、实型和字符型。下面我们将分别进行介绍。

2.3.1 整型

整型即整数数据类型。分整型常量和整型变量两部分进行介绍。

1. 整型常量

整型常量就是整常数。在C语言中，使用的整常数有十进制、八进制和十六进制三种。

（1）十进制整常数是数码为 0～9 的十进制数字串，没有前缀。例如，237、-568、65 535、1 627。

（2）八进制整常数必须以 0 开头，即以 0 作为八进制数的前缀。数码取值为 0～7。八进制数通常是无符号数。例如，015（十进制为 13）、0101（十进制为 65）、0177777（十进制为 65535）。

（3）十六进制整常数的前缀为 0X 或 0x。其数码取值为 0～9，A～F 或 a～f。例如，0X2A（十进制为 42）、0XA0（十进制为 160）、0XFFFF（十进制为 65535）。

2. 整型变量

在 C 语言中，整型变量有 6 种类型：基本整型、无符号基本整型、短整型、无符号短整型、长整型和无符号长整型。

表 2.1 列出了 ANSI 标准定义的各种整型变量所分配的内存字节数，以及数的表示范围。

表 2.1 ANSI 标准规定的整型变量

类型	类型说明符	字节	数值范围
基本整型	[signed] int	2	-32 768～32 767，即-2^{15}～（$2^{15}-1$）
无符号基本整型	unsigned [int]	2	0～65 535，即 0～（$2^{16}-1$）
短整型	[signed] short [int]	2	-32 768～32 767，即-2^{15}～（$2^{15}-1$）
无符号短整型	unsigned short [int]	2	0～65 535，即 0～（$2^{16}-1$）
长整型	[signed] long [int]	4	-214 783 648～214 783 647，即-2^{31}～（$2^{31}-1$）
无符号长整型	unsigned long [int]	4	0～4 294 967 295，即 0～（$2^{32}-1$）

2.3.2 实型

实型即实数数据类型，也称为浮点型。分实型常量和实型变量两部分进行介绍。

1. 实型常量

实型常量也称为实数或者浮点数。在C语言中，实数只采用十进制。它有两种形式：十进制小数形式和指数形式。

（1）十进制小数形式：由数码 0～9 和小数点组成。如 0.0、25.0、5.789、0.13、5.0、300.、-267.8230 等，均为合法的实数。注意，必须有小数点。

（2）指数形式：由十进制数、加阶码标志“e”或“E”及阶码（只能为整数，可以带符号）组成。例如：

2.1E5（等于 2.1×10^5）
3.7E-2（等于 3.7×10^{-2}）
0.5E7（等于 0.5×10^7）
-2.8E-2 (等于-2.8×10^{-2})

2. 实型变量

实型变量分为：单精度（float 型）、双精度（double 型）和长双精度（long double 型）三类。实型数据的属性如表 2.2 所示。

表 2.2 实数基本类型表

类型说明符	比特数（字节数）	有效数字	数的范围
float	32(4)	6～7	10^{-37}～10^{38}
double	64(8)	15～16	10^{-307}～10^{308}
long double	128(16)	18～19	10^{-4931}～10^{4932}

实型变量定义的格式和书写规则与整型相同。例如：

```
float x,y;     /*x,y 为单精度实型量*/
double a,b,c;  /*a,b,c 为双精度实型量*/
```

由于实型变量是由有限的存储单元组成的，因此能提供的有效数字总是有限的。实型数据有时存在舍入误差。

2.3.3 字符型

文字处理是计算机的一个重要应用领域，这个应用领域的程序必须能够使用和处理字符形式的数据。字符型数据存储的是字符的 ASCII 码，一个字符占一个字节。

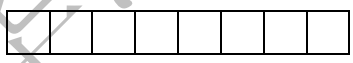
比如字符‘x’的十进制 ASCII 码是 120，字符‘y’的十进制 ASCII 码是 121。对字符变量 a、b 赋予'x'和'y'值：

```
a='x';
b='y';
```

实际上是在 a、b 两个单元内存放 120 和 121 的二进制代码：

a：

b：



所以也可以把字符看成是整型量。C 语言允许对整型变量赋予字符值，也允许对字符变量赋予整型值。在输出时，允许把字符变量按整型格式输出，也允许把整型量按字符格式输出，如程序 2.1。

向字符变量赋予整数：test1.c。

```
#include <stdio.h>
main()
{
   char a,b;                  /*定义两个字符型变量*/
   a=120;                     /*给字符型变量赋整数值*/
   b=121;
   printf("%c,%c\n",a,b);     /*以字符格式输出变量值*/
   printf("%d,%d\n",a,b);     /*以整型格式输出变量值*/
}
```

程序运行结果如下：

```
x,y
120,121
```

可以看到，当以字符格式输出变量值时，结果是 ASCII 码值 120 和 121 分别对应的字符 x 和 y，而以整型格式输出变量值时，结果是一个整型数。

说明

printf()函数称为格式输出函数，它的调用形式为：

printf ("格式控制字符串", 输出项表);

对于 printf()函数格式串中的格式符，当格式符为“%c”时，对应输出的变量值为字符，当格式符为“%d”时，对应输出的变量值为整数。详细格式请参见 2.6.3 小节。

除了以上形式的字符常量外，还有一种特殊的字符常量叫转义字符。转义字符以反斜线“\”开头，后跟一个或几个字符。转义字符具有特定的含义，不同于字符原有的意义，故称“转义”字符。转义字符主要用来表示那些用一般字符不便于表示的控制代码。表 2.3 列出了 C 语言中常用的转义字符。

表 2.3 常用的转义字符及其含义

转义字符	转义字符的含义	ASCII 代码
\n	回车换行	10
\t	横向跳到下一制表位置	9
\b	退格	8
\r	回车	13
\f	走纸换页	12
\\	反斜线符“\”	92
\'	单引号符	39
\"	双引号符	34
\a	鸣铃	7
\ddd	1～3 位八进制数所代表的字符	
\xhh	1～2 位十六进制数所代表的字符	

广义地讲，C 语言字符集中的任何一个字符均可用转义字符来表示。表中的\ddd 和\xhh 正是为此而提出的。ddd 和 hh 分别为八进制和十六进制的 ASCII 代码。如\101 表示字母"A"，\102 表示字母"B"，\134 表示反斜线，\XOA 表示换行等。

字符变量用来存储字符，一个字符型变量占用一个字节的内存容量，字符变量的类型说明符是 char。字符变量类型定义的格式和书写规则都与整型变量相同。例如：

```
char a,b;
```

字符串常量是由一对双引号引起来的字符序列。例如："CHINA"，"C program"，"$12.5" 等都是合法的字符串常量。

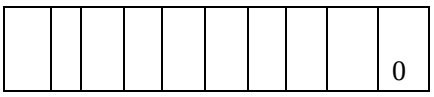
注意，字符串常量和字符常量是不同的量，它们之间主要有以下区别：

字符常量由单引号引起来，字符串常量由双引号引起来。

字符常量只能是单个字符，字符串常量则可以包含一个或多个字符。

可以把一个字符常量赋予一个字符变量，但不能把一个字符串常量赋予一个字符变量（在 C 语言中没有相应的字符串变量），但是可以用一个字符数组来存放一个字符串常量（数组将在 2.10 节予以介绍）。

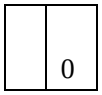
字符常量占一个字节的内存空间，字符串常量占的内存字节数等于字符串中字节数加 1。增加的一个字节中存放字符"\0"（ASCII 码为 0），它是字符串结束的标志。例如，字符串 "C program" 在内存中所占的字节为：



字符常量'a'和字符串常量"a"虽然都只有一个字符，但在内存中的情况是不同的。

'a'在内存中占一个字节，可表示为：

"a"在内存中占两个字节，可表示为：



2.4 运算符与表达式

C 语言中有很多种运算符和表达式，如算术运算、赋值运算、逗号运算、自增、自减、关系运算、逻辑运算、位运算、条件运算等。正是由于 C 语言具有丰富的多种类型的表达式，才得以体现出 C 语言所具有的表达能力强、使用灵活、适应性好的特点。本节向读者介绍算术、赋值和逗号运算符，其他的运算符将在本章中结合有关内容陆续进行介绍。

2.4.1 算术运算符与算术表达式

1. 基本的算术运算符

C 语言的基本算术运算符如表 2.4 所示。

表 2.4 算术运算符

运算符	含 义	运算对象个数	结合方向	例子
+	加法运算或取正值运算	双目、单目运算符	自左至右	a+b, +5
-	减法运算或取负值运算	双目、单目运算符	自左至右	a-b, -5
*	乘法运算	双目运算符	自左至右	a*b
/	除法运算	双目运算符	自左至右	a/b
%	模运算（求余运算）	双目运算符	自左至右	5%7

这里需要说明以下几点：

（1）“+”、“-”作为单目运算符（如-x，-5）时，具有左结合性。作为单目运算符使用时其优先级高于双目运算符。

（2）除法运算符“/”在使用时要注意数据类型。参与运算量均为整型时，结果也为整型，舍去小数。如果运算量中有一个是实型，则结果为双精度实型。例如，20/7，-20/7 的结果均为整型，小数全部舍去；而 20.0/7 和-20.0/7 由于有实数参与运算，其结果也为实型。

（3）求余运算符（模运算符）“%”要求参与运算的量均为整型，其结果等于两数相除后的余数。

2. 算术表达式

C 语言的算术表达式是由常量、变量、函数、运算符和圆括号组成的。例如：

3+5，3.2*5.6+7，-5*(18%4+9)，x/(y+z)，sin(x)+sin(y)。

它们都是合法的算术表达式。使用算术表达式时必须注意两个问题：一是双目运算符两侧的运算对象类型必须一致；二是括号可以改变表达式的运算顺序，先计算括号中的表达式，再计算括号外的表达式。

2.4.2 赋值运算符与赋值表达式

赋值运算符记为“=”，由“=”连接的式子称为赋值表达式。其一般形式为：

变量 = 表达式

赋值表达式的功能是先计算表达式的值，再赋予左边的变量。赋值运算符具有右结合性。例如：

```
pi=3.14;        /*将常数 3.14 赋值给变量 pi*/
```

```
S=(a+b)*h/2;   /*先计算算术表达式(a+b)*h/2 的值，再赋值给变量 S*/
L=2*pi*r;      /*先计算算术表达式 2*pi*r 的值，再赋值给变量 L*/
```

按照 C 语言规定，任何表达式在其末尾加上分号就构成语句，如上面的例子中就构成了 C 语言的赋值语句。

另外，如果赋值运算符两边的数据类型不相同，系统将自动进行类型转换，即把等号右边的类型转换成左边的类型，具体规定如下：

实型赋予整型，舍去小数部分。
整型赋予实型，数值不变，但以浮点形式存放，即增加小数部分（小数部分的值为 0）。
字符型赋予整型，由于字符型为一个字节，而整型为两个字节，故将字符的 ASCII 码值放到整型量的低八位中，高八位为 0。
整型赋予字符型，只把低八位赋予字符量。

程序 2.2 说明了 C 语言赋值运算中的类型转换规则，代码如 test2.c 所示。
赋值运算中的类型转换规则：test2.c。

```
#include <stdio.h>
main()
{
   int a,b=322;
   float x,y=8.88;
   char c1='k',c2;
   a=y;                /*给整型变量赋实型值*/
   printf("a=%d\n",a);
   x=b;                /*给实型变量赋整型值*/
   printf("x=%f\n",x);
   a=c1;               /*给整型变量赋字符型值*/
   printf("a=%d\n",a);
   c2=b;               /*给字符型变量赋整型值*/
   printf("c2=%c\n",c2);
}
```

程序运行结果如下：

```
a=8
x=322.000000
a=107
c2=B
```

可以看到，由于 a 为整型，所以赋予实型变量 y 值 8.88 后只取整数部分 8。x 为实型，赋予整型变量 b 值 322 后增加了小数部分。字符型变量 c1 赋予 a 变为整型，整型量 b 赋予 c2 后取其低八位成为字符型（b 的低八位为 01000010，即十进制 66，按 ASCII 码对应于字符 B）。

在赋值符“=”之前加上其他二目运算符可构成复合赋值符。如+=,-=,*=, / =,%=,<<=,>>=,&=,^=,|=。

它们等价于各自相应的运算符，例如：

a+=5 等价于 a=a+5。
x*=y+7 等价于 x=x*(y+7)。
r%=p 等价于 r=r%p。

复合赋值符这种写法，对初学者可能不习惯，但十分有利于编译处理，能提高编译效率并产生质量较高的目标代码。

2.4.3 逗号运算符与逗号表达式

在C语言中逗号“,”也是一种运算符，称为逗号运算符，其功能是把两个表达式连接起来组成一个表达式，称为逗号表达式。其一般形式为：

```
表达式1, 表达式2
```

逗号表达式的求值过程是，分别求出两个表达式的值，并以表达式 2 的值作为整个逗号表达式的值，如程序 2.3 中的代码。

逗号表达式的运算规则：test3.c。

```
#include <stdio.h>
main()
{
   int a=2,b=4,c=6,x,y;
   y=((x=a+b),(b+c));          /*用逗号表达式对y赋值*/
   printf("y=%d, x=%d",y,x);   /*显示x、y的值*/
}
```

程序运行结果如下：

```
y=10, x=6
```

从结果可以看出，y 等于整个逗号表达式的值，也就是表达式 2 的值 10，而 x 是第一个表达式的值 6。

2.5 C 程序的 3 种基本结构

算法的实现过程是由一系列操作组成的，这些操作之间的执行次序就是程序的控制结构。计算机科学家证明：任何简单或复杂的算法都可以由顺序、选择和循环这三种基本结构组合而成。所以这三种结构就被称为程序设计的三种基本结构。

2.5.1 顺序结构

顺序结构的程序设计是最简单的，程序中的各个操作按照它们出现的先后顺序执行，其流程如图 2.1 所示。

图中 S1 和 S2 表示两个处理步骤。整个顺序结构只有一个入口点和一个出口点。这种结构的特点是程序从入口点开始，按顺序执行所有操作，直到出口点，所以称为顺序结构。不论程序中包含了什么样的结构，程序的总流程都是顺序结构。程序 2.4 给出一个顺序结构程序设计的例子。

设有变量 x 和 y，编程序实现两个变量值的互换。

实现两个变量值互换的方式有很多种，本例中我们使用中间变量 t 来实现这个功能。先把 x 的值保存在变量 t 中，即 t=x；然后执行 x=y；此时，虽然 x 的值被 y 的值取代，但 x 的值事先已经保存在另一个变量 t 中，所以在使用 y=t 时，就可以把原 x 的值赋给 y，从而实现 x、y 值的互换，程序流程如图 2.2 所示。代码实现如 test4.c。

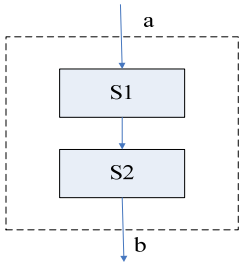


图 2.1 顺序结构

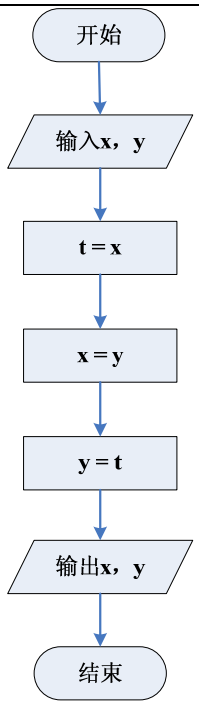


图 2.2 两个变量值互换

test4.c：

```
#include <stdio.h>
main()
{
	int x,y,t;
	printf("Enter x and y:\n");   /*提示用户输入数据*/
	scanf("%d %d",&x,&y);         /*通过格式输入 scanf 读取输入值*/
	t=x;                          /*交换算法*/
	x=y;
	y=t;
	printf("x=%d, y=%d\n",x,y);  /*显示交换结果*/
}
```

程序运行结果如下（□表示空格，↙表示回车）：

```
Enter x and y:
10□5↙
x=5, y=10
```

可以看到，变量 x 和 y 的值进行了互换。

2.5.2 选择结构

选择程序结构用于判断给定的条件，根据判断的结果来控制程序的流程。在选择结构中，程序的处理步骤出现了分支，它需要根据某一特定的条件选择其中的一个分支执行。选择结构有单选择、双选择和多选择 3 种形式。

单选择结构是最简单的选择结构，如图 2.3 所示，如果条件满足则执行 S1，否则向下到流出口处。也就是说，当条件不满足时，什么也没执行。C 语言用 if 语句实现这种功能。其一般形式为：

```
if (表达式) 语句序列
```

if 语句的执行过程与 if else 相似（稍后将会看到），只是在判断条件为“假”时，直接跳过语句序列，执行 if 的下一条语句。

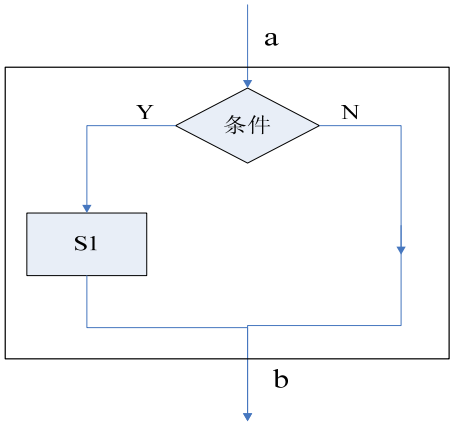


图 2.3　单选择结构

双选择结构如图 2.4 所示，程序流程出现了两个可供选择的分支，如果条件满足就执行 S1 处理，否则执行 S2 处理。两个分支中只能选择一条且必须选择一条执行，但不论选择了哪一条分支执行，最后流程都一定到达结构的出口点处。C 语言用 if else 语句实现这种功能。其一般形式为：

```
if (表达式) 语句 1
else  语句 2
```

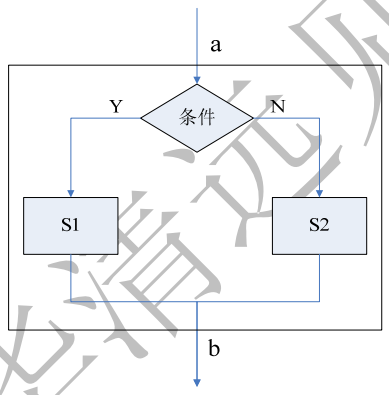


图 2.4　双选择结构

程序 2.5 给出了一个简单的选择结构程序的例子。

求两个数中的最大值：test5.c。

```
#include <stdio.h>
main()
{
       int x,y;
       printf("Enter x and y:\n");
       scanf("%d %d",&x,&y);
       if( x > y )
printf("max=%d\n",x);       /*如果条件满足执行此条语句*/
       else
 printf("max=%d\n",y);       /*如果条件不满足执行此条语句*/
}
```

程序运行结果如下（□表示空格，↙表示回车）：

```
Enter x and y:
10□5↙
max=10
```

多选择结构如图 2.5 所示，程序出现多个分支，程序执行方向将根据条件确定。如果满足条件 1 则执行 S1，如果满足条件 n 则执行 Sn。总之，要根据条件选择多个分支中的一个执行，不论选择哪一条分支，最后流程要到达同一个出口，如果所有分支条件都不满足，则直接到达出口。

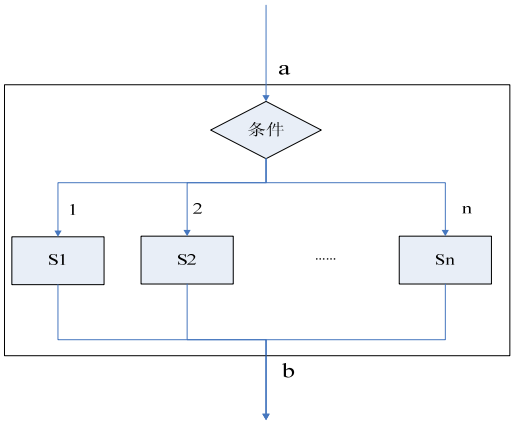


图 2.5　多选择结构

在 C 语言中，用嵌套 if 语句可以实现多分支结构程序，但分支较多时就显得很复杂，可读性差。C 语言中的 switch 语句专用于实现多分支结构程序。switch 语句的调用形式如下：

```
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;
}
```

switch 语句的执行过程可描述为：首先计算表达式的值，然后依次与常量表达式 i（i=1，2，3，…，n）进行比较，若表达式的值与某常量表达式相等，则从该常量表达式处开始执行，直到 switch 语句结束。若所有的常量表达式 i（i=1，2，3，…，n）的值均不等于表达式的值，则从 default 处开始执行。程序 2.6 是一个关于多分支选择结构的例子。

输入某学生的成绩，输出该学生的成绩和等级（A 级：90～100，B 级：80～89，C 级：60～79，D 级：0～59）。

为了区分各分数段，将[0，100]每十分划分为一段，则 x/10 的值为 0 到 10，它们表示 11 段：0～9 为 0 段，10～19 为 1 段，……，90～99 为第 9 段，100 为第 10 段，用 case 后的常量表示段号。例如，x=66，则 x/10 的值为 6，所以 x 在第 6 段，即 60≤x<70，属于 C 级。若 x 不在[0，10]段内，则表示 x 是非法成绩，在 default 分支处理。控制流程如图 2.6 所示，其代码实现如 test6.c。

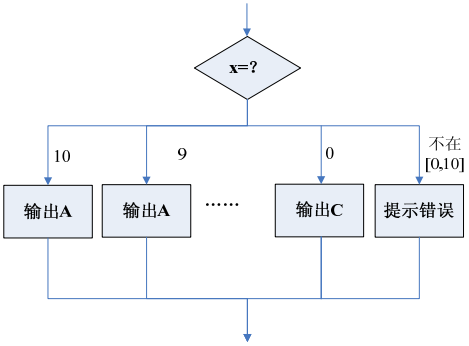


图 2.6　多分支结构

程序 2.6：test6.c。

```
#include <stdio.h>
main()
{
      int x;
printf("Please input x:\n");
scanf("%d",&x);                    /*输入学生成绩*/
switch(x/10)                       /*判断条件*/
{
        case 10: printf("x=%d -> A\n",x);break;     /*分段显示结果*/
        case 9: printf("x=%d -> A\n",x);break;
        case 8: printf("x=%d -> B\n",x);break;
        case 7: printf("x=%d -> C\n",x);break;
        case 6: printf("x=%d -> C\n",x);break;
        case 5: printf("x=%d -> D\n",x);break;
        case 4: printf("x=%d -> D\n",x);break;
        case 3: printf("x=%d -> D\n",x);break;
        case 2: printf("x=%d -> D\n",x);break;
        case 1: printf("x=%d -> D\n",x);break;
        default: printf("x=%d data errer!\n",x);   /*如果 x 不在[0,10]段内，则出错*/
      }
}
```

程序运行结果如下（↙表示回车）：

```
Please input x:
65↙
x=65 -> C
```

可以看到，当输入的成绩为 65 时，我们得到了等级 C。

说明

break 语句用于终止它所在的 switch 语句的执行。如果没有 break 语句，本例在执行完 case 6 后，还会依次执行 case 5、case 4 等后面的语句。读者可以自己验证一下不带 break 语句的情况。一般在多分支选择结构的程序中，在得到正确结果后，立即使用 break 语句来终止 switch 语句的执行。

2.5.3 循环结构

循环结构表示程序反复执行某个或某些操作，直到某些条件为假时才可终止循环。循环结构可以减少源程序重复书写的工作量，用来描述重复执行某段算法的问题，这是程序设计中最能发挥计算机特长的程序结构。

循环结构的基本形式有两种：当型循环和直到型循环，其流程分别如图 2.7(a)、(b)所示。

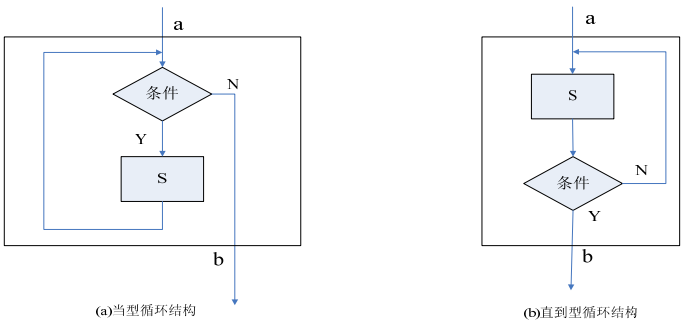


图 2.7 循环结构

当型循环结构的执行过程是，首先判断条件，当满足条件时执行循环体，执行完后自动返回循环入口；如果条件不满足，则退出循环体直接到达流程出口处，所以是先判断后执行。C 语言中用 while 语句实现当型循环结构。while 语句的调用形式为：

```
while (表达式) 循环体语句
```

直到型循环结构的执行过程是，从结构入口直接执行循环体，在循环终点处判断条件，如果条件不满足，返回入口处继续执行循环体，直到循环判断条件为真时再退出循环到达流程出口处，属于先执行后判断。C 语言中用 do while 语句来实现直到型循环结构。do while 语句的调用形式为：

```
do
  循环体语句
while(表达式)
```

程序 2.7 是一个使用 do while 语句来实现直到型循环结构的例子。

求 n!（n! =1*2*3*…*(n-1）*n)。

这是若干项的连乘问题，执行流程如图 2.8 所示，代码实现如 test7.c。

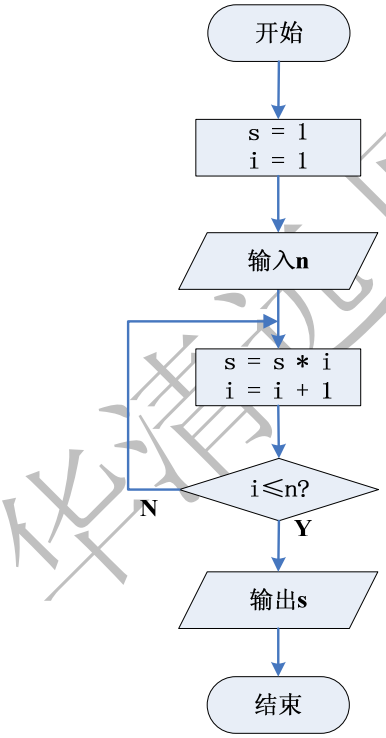


图 2.8　用 do_while 循环求 n！

程序 2.7：test7.c。

```
#include <stdio.h>
main()
{
       int i,n;
long s;
       s=1;
       i=1;
       printf("Please input n: ");
       scanf("%d",&n);
       do{s*=i;           /*循环体*/
          i++;
       } while ( i<=n );   /*循环结束的判断条件*/
```

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

程序运行结果如下（↙表示回车）：

```
Please input n: 5↙
5!=120
```

C 语言中，还有一种 for 循环语句。for 语句的调用形式如下：

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

它等价于下列 while 语句：

```
表达式 1;
while(表达式 2){
循环体语句;
表达式 3;
}
```

for 语句的执行过程如图 2.9 所示。首先计算表达式 1 的值，然后检测表达式 2 的值，若其值为“真”，则执行循环体语句，执行完毕后，再计算表达式 3，然后检测表达式 2 的值，若为“真”，则继续执行循环体语句，如此循环，直到表达式 2 的值为“假”时终止循环。

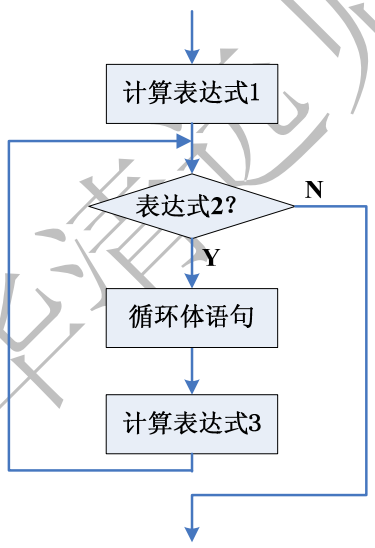


图 2.9　for 循环执行流程

表达式 1 通常为循环变量赋初值的表达式，表达式 2 为控制循环的表达式，表达式 3 通常是改变循环变量的表达式。程序 2.8 给出了一个使用 for 语句的简单例子。

用 for 循环计算 1～100 的整数累加和：test8.c。

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

程序运行结果如下：

```
n=5050
```

根据 for 循环的判断表达式 2，循环体一共执行了 100 次，即“n+=i;”语句一共被执行了 100 次。

2.6 C 语言中的数据输入与输出

C 语言中没有输入输出语句，为了实现输入输出功能，在 C 语言的库函数中提供了一组输入输出函数，其中 scanf 和 printf 函数是针对标准输入输出设备（键盘和显示器）进行格式化输入输出的函数，而 getchar 和 putchar 是专门对单个字符进行输入输出的函数。要使用它们，必须将文件“stdio.h”包含在程序文件中。

2.6.1 字符输出函数 putchar

putchar 函数是字符输出函数，其功能是在终端（显示器）输出单个字符。其一般调用形式为：

```
putchar(字符变量);
```

例如：

```
putchar('A');            /*输出大写字母 A*/
putchar(x);          /*输出字符变量 x 的值*/
putchar('101');      /*也是输出字符 A*/
putchar('\n');       /*换行*/
```

注意，对控制字符则执行控制功能，不在屏幕上显示。

2.6.2 字符输入函数 getchar

getchar 函数的功能是从键盘上读一个字符作为函数值。其一般调用形式为：

```
getchar();
```

通常把输入的字符赋予一个字符变量，构成赋值语句，如程序 2.9 中代码。

输入输出单个字符：test9.c。

```
#include <stdio.h>
void main()
{
   char c;                        /*定义字符变量 c*/
   printf("input a character:");     /*打印提示信息*/
   c=getchar();                      /*将读取的字符赋给变量 c*/
   putchar(c);                       /*输出字符变量 c*/
}
```

程序运行结果如下（↙表示回车）：

```
input a character:b↙
b
```

2.6.3 格式输出函数 printf

printf 函数称为格式输出函数。其功能是按照用户指定的格式，把指定的数据输出到显示器屏幕上。printf 函数调用的一般形式为：

```
printf(“格式控制字符串”, 输出项表);
```

其中格式控制字符串用来说明输出列表中各输出项的输出格式。输出项表列出了要输出的项，各输出项之间用逗号分开。输出项表也可以没有，这时输出的是格式字符串本身。

格式控制字符串有两种：格式字符串和非格式字符串。非格式字符串在输出时原样打印，在显示中只起提示作用。格式字符串是以“%”开头的字符串，在“%”后面跟有各种格式字符，以说明输出数据的类型、形式、长度、小数位数等。格式字符串的形式为：

```
% [输出最小宽度][.精度][长度]类型
```

如：%d，%9.3f 等。其中%d 格式符表示用十进制整型格式输出，而%f 表示用实型格式输出，附加格式说明符“9.3”表示输出宽度为 9（包括小数点），并含 3 位小数。printf 函数常用的输出格式符及其含义如表 2.5 所示。

表 2.5　输出格式符

格式字符	含义
d，i	以十进制形式输出带符号整数（正数不输出符号）
o	以八进制形式输出无符号整数（不输出前缀 0）
x	以十六进制形式输出无符号整数（不输出前缀 0x）
u	以十进制形式输出无符号整数
f	以小数形式输出单、双精度实数
e	以指数形式输出单、双精度实数
g	以%f 或%e 中较短输出宽度的一种格式输出单、双精度实数
c	输出单个字符
s	输出字符串

printf 函数输出整型、实型、字符型数据: test10.c。

```
#include <stdio.h>
main()
{
    int a=16;
    float b=123.4567;
    char c='A';
    printf("a=%d\n",a);           /*输出整型变量 a 的值*/
    printf("b=%9.4f\n",b);        /*输出实型变量 b 的值，注意运行结果的格式*/
    printf("c=%c,%s\n",c, "China");   /*输出字符变量 c 和字符串*/
}
```

程序运行结果如下（□表示空格）：

```
a=16
b=□123.4567
c=A,China
```

在程序 2.10 中，第一次用%d 格式输出整型数；第二次用%9.4f 格式输出实型数，宽度为 9（包括小数点），并含 4 位小数，不足 9 列，则左端补空格；第三次是用%c 格式输出单个字符，用%s 格式输出字符串。

2.6.4　格式输入函数 scanf

scanf 函数称为格式输入函数，即按照格式字符串规定的格式，从键盘上把数据输入到指定的变量之中。scanf 函数调用的一般形式为：

```
scanf(“格式控制字符串”, 输入项地址表);
```

其中，格式控制字符串的作用与 printf 函数相同，但不能显示非格式字符串，也就是不能显示提示字符串。地址表项中给出各变量的地址，地址是由地址运算符“&”后跟变量名组成的（如&a, &b）。

scanf 函数中格式字符串的构成与 printf 函数基本相同，但使用时有几点不同。

（1）格式说明符中，可以指定数据的宽度，但不能指定数据的精度。例如：

```
float a;
scanf(“%10f”,&a);    /*正确*/
scanf(“%10.2f”,&a);  /*错误*/
```

（2）输入 long 型数据必须使用%ld，输入 double 数据必须使用%lf 或%le。

（3）附加格式说明符“*”使对应的输入数据不赋给相应的变量，如程序 2.11。

用 scanf 函数读取输入的变量，并检查读取结果：test11.c。

```
#include <stdio.h>
main()
{
int a;
float b;
char c;
float d;                    /*定义 4 个不同数据类型的变量*/
scanf("%d",&a);          /*把输入的十进制整数赋给整型变量*/
printf("a=%d\n",a);
scanf("%10f",&b);        /*把输入的实数赋给实型变量*/
printf("b=%f\n",b);
scanf("%s",&c);          /*把输入的字符赋给字符型变量*/
printf("c=%c\n",c);
scanf("%*d",&d);         /*输入数据，不赋给相应的变量*/
printf("d=%f\n",d);
}
```

程序运行结果如下（↙表示回车）：

```
654↙
a=654
1.23↙
b=1.230000
e↙
c=e
4.6↙
d=-107374176.000000
```

2.7 函数

C 源程序是由函数组成的。最简单的程序有一个主函数 main()，但实用程序往往由多个函数组成，由主函数调用其他函数，其他函数也可以互相调用。函数是 C 源程序的基本模块，程序的许多功能是通过对函数模块的调用来实现的，学会编写和调用函数可以提高编程效率。

2.7.1 函数的定义

函数的定义通常包含以下内容：

```
类型  函数名(形参表说明)     /*函数首部*/
{
说明语句                          /*函数体*/
   执行语句
}
```

对上面的定义形式进行以下说明：

（1）“类型”是指函数返回值的类型。函数返回值不能是数组，也不能是函数，除此之外任何合法的数据类型都可以是函数的类型，如：int，long，float，char 等。函数类型可以省略，当不指明函数类型时，系统默认的是整型。

（2）函数名是用户自定义的标识符，在 C 语言函数定义中不可省略，须符合 C 语言对标识符的规范，用于标识函数，并用该标识符调用函数。另外函数名本身也有值，它代表了该函数的入口地址，使用指针调用函数时，将用到此功能。

（3）形参又称为“形式参数”。形参表是用逗号分隔的一组变量说明，包括形参的类型和形参的标识符，其作用是指出每一个形参的类型和形参的名称，当调用函数时，接收来自主调函数的数据，确定各参数的值。

（4）用{ }括起来的部分是函数的主体，称为函数体。函数体是一段程序，确定该函数应完成的规定的运算，应执行的规定的动作，集中体现了函数的功能。函数内部应有自己的说明语句和执行语句，但函数内定义的变量不可以与形参同名。花括号{ }是不可以省略的。

根据函数定义的一般形式，可以定义一个最简单的函数：

```
add()
{  ;
}
```

这是 C 语言中一个合法的函数，函数名为 add。它没有函数类型说明，也没有形参表，同时函数体内也没有语句。实际上函数 add 不执行任何操作和运算，它是一个空函数，在一般情况下是没有用途的，但在程序开发的过程中有时是需要的，常用来代替尚未开发完毕的函数。

2.7.2 函数的调用

主调函数使用被调函数的功能，称为函数调用。在 C 语言中，只有在函数调用时，函数体中定义的功能才会被执行。C 语言中，函数调用的一般形式为：

```
函数名(类型 形参，类型 形参…);
```

对无参函数调用时则无实际参数表。实际参数表中的参数可以是常数、变量或其他构造类型数据及表达式，各实参之间用逗号分隔。

在 C 语言中，可以用以下几种方式调用函数。

（1）函数表达式：函数作为表达式中的一项出现在表达式中，以函数返回值参与表达式的运算。这种方式要求函数是有返回值的。例如：

```
z=max(x,y);
```

是一个赋值表达式，把 max 的返回值赋予变量 z。

（2）函数语句：函数调用的一般形式加上分号即构成函数语句。例如：

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

都是以函数语句的方式调用函数。

（3）函数实参：函数作为另一个函数调用的实际参数出现。这种情况是把该函数的返回值作为实参进行传送，因此要求该函数必须是有返回值的。例如：

```
printf("%d",max(x,y));   /*把 max 调用的返回值作为 printf 函数的实参*/
```

在主调函数中调用某函数之前应对该被调函数进行声明。在主调函数中对被调函数进行说明的目的是使编译系统知道被调函数返回值的类型，以便在主调函数中按此种类型对返回值进行相应的处理。其一般形式为：

```
类型说明符 被调函数名(类型 形参，类型 形参…);
```

需要注意的是，函数的声明和函数的定义有本质上的不同。主要区别在以下两个方面：

（1）函数的定义是编写一段程序，应有函数的具体功能语句——函数体；而函数的声明仅是向编译系统的一个说明，不含具体的执行动作。

（2）在程序中，函数的定义只能有一次，而函数的声明可以有多次。

通过程序 2.12 中的代码，读者可以简单了解函数定义、说明、调用的全过程。

编写一个 max 函数，实现选取两个数中较大值的功能：test12.c。

```
#include <stdio.h>
int max(int a, int b);            /*函数 max 的说明*/
main()
{
int a, b;
printf("Enter a b:");
scanf("%d %d",&a, &b);
   printf("max = %d\n", max(a,b)); /*函数 max 的调用*/
}
int max(int a, int b)           /*函数 max 的定义*/
{
int p;
       p=a>b?a:b;
       return (p);
}
```

程序运行结果如下（↙表示回车，□表示空格）：

```
Enter a b:100□99↙
max=100
```

可以看到，程序中对两个输入整数的比较，调用了 max()函数。

注意

在下列情况下可以省去主调函数中对被调函数的函数说明。

（1）如果被调函数的返回值是整型或字符型时，可以不对被调函数进行说明，而直接调用。这时系统将自动对被调函数返回值按整型处理。

（2）当被调函数的函数定义出现在主调函数之前时，在主调函数中也可以不对被调函数再进行说明而直接调用。

（3）如在所有函数定义之前，在函数外预先说明了各个函数的类型，则在以后的各主调函数中，可不再对被调函数进行说明。

（4）对库函数的调用不需要再进行说明，但必须把该函数的头文件用 include 命令包含在源文件前部。

2.7.3 变量的存储类别

在 C 语言中，变量是对程序中数据所占内存空间的一种抽象定义，定义变量时，用户定义变量的名、变量的类型，这些都是变量的操作属性。不仅可以通过变量名访问该变量，系统还通过该标识符确定变量在内存中的位置。在计算机中，保存变量当前值的存储单元有两类，一类是内存，另一类是 CPU 的寄存器。变量的存储类型关系到变量的存储位置，C 语言中定义了 4 种存储属性，即自动变量、外部变量、静态变量和寄存器变量，它关系到变量在内存中的存放位置，由此决定了变量的保留时间和变量的作用范围。

变量的保留时间又称为生存期，从时间的角度，可将变量分为静态存储和动态存储两种情况。静态存储是指变量存储在内存的静态存储区，在编译时就分配了存储空间，在整个程序的运行期间，该变量占有固定的存储单元，程序结束后，这部分空间才释放，变量的值在整个程序中始终存在；动态存储是指变量存储在内存的动态存储区，在程序的运行过程中，只有当变量所在的函数被调用时，编译系统才临时为该变量分配一段内存单元，函数调用结束，该变量空间释放，变量的值只在函数调用期存在。

变量的作用范围又称为作用域，从空间角度，可以将变量分为全局变量和局部变量。局部变量是在一个函数或复合语句内定义的变量，它仅在函数或复合语句内有效，编译时，编译系统不为局部变量分配内存单元，而是在程序运行过程中，当局部变量所在的函数被调用时，编译系统根据需要，临时分配内存，调用结束，空间释放；全局变量是在函数之外定义的变量，其作用范围为从定义处开始到本文件结束，编译时，编译系统为其分配固定的内存单元，在程序运行的自始至终都占用固定单元。

1. 自动变量

函数中的局部变量，如不专门声明为 static 存储类别，都是动态地分配存储空间的，数据存储在动态存储区中。函数中的形参和在函数中定义的变量（包括在复合语句中定义的变量）都属此类，在调用该函数时系统会给它们分配存储空间，在函数调用结束时就自动释放这些存储空间。这类局部变量称为自动变量。自动变量用关键字 auto 进行存储类别的声明，例如声明一个自动变量：

```
int fun(int a)
{
auto int b,c=3;     /*定义 b，c 为自动变量*/
}
```

a 是函数 fun()的形参，b、c 是自动变量，并对 c 赋初值 3。执行完 fun()函数后，自动释放 a、b、c 所占的存储单元。

2. 外部变量

外部变量（即全局变量）是在函数的外部定义的，它的作用域为从变量定义处开始，到本程序文件的末尾。如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件末尾。如果在定义点之前的函数想引用该外部变量，则应该在引用之前用关键字 extern 对该变量进行“外部变量声明”。表示该变量是一个已经定义的外部变量。有了此声明，就可以从“声明”处起，合法地使用该外部变量，如程序 2.13。

用 extern 声明外部变量，扩展程序文件中的作用域：test13.c。

```
#include <stdio.h>
int max(int x,int y)          /*定义 max 函数*/
{
int z;
   z=x>y?x:y;
   return(z);
}
main( )
{
extern A,B;                   /*声明外部变量*/
   printf("%d\n",max(A,B));       /*调用 max 函数*/
}
int A=13,B=-8;                    /*在文件末尾定义外部变量*/
```

在本程序文件的最后 1 行定义了外部变量 A、B，但由于外部变量定义的位置在主函数 main 之后，因此本来在 main 函数中不能引用外部变量 A、B。但由于在 main 函数中用 extern 对 A 和 B 进行了“外部变量声明”，就可以从“声明”处起，合法地使用该外部变量 A 和 B 了。

3. 静态变量

有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值，这时就应该指定局部变量为静态局部变量，用关键字 static 进行声明。考察静态局部变量的值，如程序 2.14。

静态局部变量的使用：test14.c。

```
#include <stdio.h>
fun(int a)          /*定义函数 fun*/
{
auto b=0;               /*定义自动变量 b，并初始化为 0*/
   static c=3;       /*定义静态变量 c，并初始化为 3*/
   b=b+1;
   c=c+1;
   return(a+b+c);
}
main()
{
int a=2,i;
   for(i=0;i<3;i++)  /*重复调用函数 f，比较函数返回值*/
     printf("%d ",fun(a));
}
```

程序运行结果如下（□表示空格）：

```
7□8□9
```

在上面的程序中，对于自动变量 b，每次调用函数 fun 时，其值都被初始化为 0，而静态变量 c 则保留上一次调用结束时的值，即逐次加 1，所以程序运行的结果也是逐次增加 1。

提示

局部变量与自动变量的不同：

（1）静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在整个程序运行期间都不释放。而自动变量（即动态局部变量）属于动态存储类别，占动态存储空间，函数调用结束后即释放。

（2）静态局部变量在编译时赋初值，即只赋初值一次；而对自动变量赋初值是在函数调用时进行，每调用一次函数重新给一次初值，相当于执行一次赋值语句。

（3）如果在定义局部变量时不赋初值的话，则对静态局部变量来说，编译时自动赋初值 0（对数值型变量）或空字符（对字符变量）。而对自动变量来说，如果不赋初值则它的值是一个不确定的值。

4. 寄存器变量

为提高效率，C 语言允许将局部变量的值存放在 CPU 的寄存器中，这种变量叫做寄存器变量，用关键字 register 声明。使用寄存器变量需要注意以下几点：

（1）只有局部自动变量和形式参数可以作为寄存器变量。

（2）一个计算机系统中的寄存器数目有限，不能定义任意多个寄存器变量。

（3）不能使用取地址运算符“&”求寄存器变量的地址。

定义一个函数，实现阶乘功能，函数中使用寄存器变量：test15.c。

```
#include <stdio.h>
int fac(int n)                      /*定义阶乘函数*/
{
      register int i,f=1;           /*定义寄存器变量 i,f*/
```

```
        for(i=1;i<=n;i++)                /*用循环实现阶乘的功能*/
          f=f*i;
        return(f);                       /*返回计算结果 f*/
}
main()
{
        int i;
        for(i=0;i<=5;i++)
          printf("%d! = %d\n",i,fac(i));    /*调用阶乘函数*/
}
```

程序运行结果如下：

```
0!=1
1!=1
2!=2
3!=6
4!=24
5!=120
```

最后，表 2.6 对这 4 种变量存储类别的特性进行了总结。

表 2.6 4 种存储类型的特性

性能	自动变量	外部变量	静态变量		寄存器变量
			外部	内部	
记忆能力	无	有	有	有	无
多个函数共享	否	是	是	否	否
整个程序的不同文件共享性	否	是	否	否	否
初始化时未显示赋值的取值	不确定	0	0	0	不确定
变量初始化	由程序控制	编译器	编译器	编译器	由程序控制
数组与结构初始化	是	是	是	是	否
作用域	当前函数	整个程序	文件	函数	当前函数

2.8 数组

数组是同类型有序数据的集合，可以为这些数据的集合起一个名字，称为数组名。该集合中的各个数据项称为数组元素，每个元素可用数组名和下标表示。在 C 程序设计中，数组是一个十分有用的数据类型，下面将对数组进行详细介绍。

2.8.1 一维数组的定义和使用

在 C 语言中使用数组必须先进行定义，一维数组的定义方式如下：

```
类型说明符 数组名 [常量表达式];
```

其中类型说明符是任意一种基本数据类型或构造数据类型，它定义了全体数组成员的数据类型；数组名是用户定义的数组标识符；方括号中的常量表达式表示数据元素的个数，也称为数组的长度。例如：

```
float a[5],b[10];
```

该语句表示：

（1）定义了浮点型数组 a 和 b，其数组元素的类型都是 float。

（2）a 数组有 5 个数组元素，b 数组有 10 个数组元素。

（3）a 数组的数组元素是 a[0]、a[1]、a[2]、a[3]和 a[4]，共 5 个数组元素。所以 a 数组元素的下标大于等于 0，且小于 5。

（4）定义了 float 型数组 a，编译程序将为 a 数组在内存中开辟 5 个连续的存储单元，用来存放 a 数组的 5 个数组元素，a[0]代表这片存储区的第一个存储单元。数组名 a 代表 a 数组的首地址，即 a[0]的地址。

数组元素是组成数组的基本单元，数组元素也是一种变量，其标识方法为数组名后跟一个下标。下标表示元素在数组中的顺序号。引用数组元素的一般形式为：

```
数组名[下标]
```

其中下标只能为整型常量或整型表达式。例如 a[5]、a[i+j]、a[i++]都是合法的数组元素。

数组元素通常也称为下标变量。必须先定义数组，才能使用下标变量。在 C 语言中只能逐个使用下标变量，而不能一次引用整个数组。

给数组赋值的方法除了用赋值语句对数组元素逐个赋值外，还可采用初始化赋值和动态赋值的方法。数组初始化赋值是指在数组定义时给数组元素赋予初值。数组初始化是在编译阶段进行的，这样将减少程序运行时间，提高效率。

初始化赋值的一般形式为：

```
类型说明符 数组名[常量表达式]={初始值, 初始值, …… 初始值};
```

例如：

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

相当于：

```
a[0]=0;a[1]=1...a[9]=9;
```

在输出数组时，通常使用循环语句逐个输出各下标变量。程序 2.16 是关于数组初始化与输出的简单例子。

定义一个数组，逐个对其赋值，然后输出各个元素值：test16.c。

```
#include <stdio.h>
main()
{
   int i,a[10];         /*定义数组 a*/
   for(i=0;i<=9;i++)    /*使用 for 循环依次对数组中的各个元素赋初值*/
     a[i]=i;
   for(i=9;i>=0;i--)    /*使用 for 循环依次输出数组的每个元素*/
     printf("%d ",a[i]);
}
```

程序运行结果如下（□表示空格）：

```
9□8□7□6□5□4□3□2□1□0
```

程序 2.16 中首先使用 for 循环依次对数组 a 中的各个元素赋初值，再用 for 循环依次输出数组的各个元素值。

2.8.2 二维数组的定义和使用

当数组元素具有两个下标时，该数组称为二维数组，同样地，n 维数组具有 n 个下标。在实际问题中有很多量是二维的或多维的，多维数组元素有多个下标，以标识它在数组中的位置，所以也称为多下标变量。二维数组定义的一般形式是：

```
类型说明符 数组名[常量表达式1][常量表达式2];
```

其中常量表达式 1 表示第一维下标的长度，常量表达式 2 表示第二维下标的长度。

例如：

```
int a[2][3];
```

该语句表示：

（1）定义了整型二维数组 a，其数组元素类型是 int。

（2）a 数组有两行三列，共 6 个元素。

（3）该数组的行下标为 0、1，列下标为 0、1、2。数组元素分别是：

```
a[0][0],a[0][1],a[0][2],a[1][0],a[1][1],a[1][2];
```

（4）定义了 int 型数组 a，编译程序将为 a 数组在内存中开辟 6 个连续的存储单元，用来存放 a 数组的 6 个元素。存储方式为按行存放，即先依次存放第 0 行的 3 个元素，然后再接着存放第 1 行的 3 个数组元素。数组名 a 代表数组的首地址。

（5）在 C 语言中，二维数组 a 的每一行都可以看做是一维数组，a[0]表示第 0 行的 3 个元素构成的一维数组。

同一维数组一样，引用二维数组，也是引用它的数组元素。其表示形式为：

```
数组名[行下标][列下标]
```

其中下标应为整型常量或整型表达式。

二维数组初始化也是在类型说明时给各下标变量赋予初值，二维数组可按行分段赋值，也可按行连续赋值。例如对数组 a[4][3]的赋值如下。

（1）按行分段赋值可写为：

```
int a[4][3]={ {80,75,92},{61,65,71},{59,63,70},{85,87,90} };
```

（2）按行连续赋值可写为：

```
int a[4][3]={ 80,75,92,61,65,71,59,63,70,85,87,90};
```

这两种赋初值的结果是完全相同的。程序 2.17 是二维数组定义与使用的简单例子。

一个学习小组共有 5 人，每个人有三门课的考试成绩，求各科的平均成绩和全组成员的总平均成绩。他们的成绩与科目如下表所示：

	张	王	李	赵	周
Math	80	61	59	85	76
C	75	65	63	87	77
Foxpro	92	71	70	90	85

程序 2.17：test17.c。

```
#include <stdio.h>
main( )
{
   int i,j,s=0, average,v[3];
   int a[5][3]={{80,75,92},{61,65,71},{59,63,70},{85,87,90},{76,77,85}};/*定义二维数组*/
   for(i=0;i<3;i++)          /*用两层循环嵌套的方式访问数组的每个元素*/
   {
   for(j=0;j<5;j++)
           s=s+a[j][i];      /*变量 s 的值为各科的总成绩*/
         v[i]=s/5;               /*各科的平均成绩*/
```

```
            s=0;
        }
    average=(v[0]+v[1]+v[2])/3;      /*总平均成绩*/
        printf("Math:%d\nC language:%d\nFoxpro:%d\n",v[0],v[1],v[2]);
        printf("total:%d\n", average);
    }
```

程序运行结果如下（□表示空格）：

```
Math:72
C language:73
Foxpro:81
total:75
```

2.8.3 字符数组和字符串

用来存放字符的数组称为字符数组。字符数组的各个元素依次存放字符串的各字符，字符数组的数组名代表该数组的首地址，这为处理字符串中个别字符和引用整个字符串提供了极大的方便。

字符数组的定义形式与前面介绍的数值数组相同。例如：

```
char c[10];
```

字符数组也允许在定义时进行初始化赋值。例如：

```
char c[6]={'c', ' h ', 'i', 'n', 'a' , '\0' };
```

对字符数组的各个元素逐个赋值后，各元素的值为：

```
c[0]= 'c',c[1]= 'h',c[2]= 'i',c[3]= 'n',c[4]= 'a',c[5]= '\0'
```

其中，‘\0’为字符串结束符。如果不对 c[5]赋任何值，‘\0’会由系统自动添加。

字符数组也可采用字符串常量的赋值方式，例如：

```
char a[ ]={"china"};
```

2.8.4 常用字符串处理函数

C 语言提供了丰富的字符串处理函数，大致可分为字符串的输入、输出、合并、修改、比较、转换、复制、搜索等几类，使用这些函数可大大减轻编程的负担。用于输入输出的字符串函数，在使用前应包含头文件 stdio.h，使用其他字符串函数则应包含头文件 string.h。下面介绍几个最常用的字符串处理函数。

1. 字符串输出函数 puts

格式：

```
puts (字符数组名);
```

功能：把字符数组中的字符串输出到显示器，即在屏幕上显示该字符串，如程序 2.18。

用 puts 函数输出一个字符串：test18.c。

```
#include <stdio.h>
main()
{
    char c[]="BASIC\nBASE";      /*定义一个字符串数组*/
    puts(c);                     /*输出字符串*/
}
```

程序运行结果如下：

```
BASIC
BASE
```

字符串数组中的“\n”为转义字符，意思为换行。

提示

puts 函数完全可以由 printf 函数取代。当需要按一定格式输出时，通常使用 printf 函数。

2. 字符串输入函数 gets

格式：

```
gets(字符数组名);
```

功能：从标准输入设备上输入一个字符串，如程序 2.19。

用 gets 函数读取一个字符串：test19.c。

```
#include <stdio.h>
main()
{
  char st[15];                /*定义一个字符串数组*/
   printf("input string: ");
    gets(st);                 /*输入字符串*/
    puts(st);                 /*输出字符串*/
}
```

程序运行结果如下（□表示空格，↙表示回车）：

```
input string: abcde□fg↙
abcde□fg
```

提示

gets 函数并不以空格作为字符串输入结束的标志，而是以回车作为输入结束的标志，这与 scanf 函数是不同的。

3. 字符串连接函数 strcat

格式：

```
strcat (字符数组名 1, 字符数组名 2);
```

功能：把字符数组 2 中的字符串连接到字符数组 1 中字符串的后面，并删除字符串 1 后的结束标志‘\0’，函数的返回值是字符数组 1 的首地址。如程序 2.20。

用 strcat 函数连接两个字符串：test20.c。

```
#include <string.h>                    /*字符串处理函数头文件*/
main()
{
   static char st1[30]="My name is ";      /*定义字符串数组 st1*/
   int st2[10];                            /*定义数组 st2 为整型*/
   printf("input your name: ");
   gets(st2);                           /*输入字符串 st2*/
   strcat(st1,st2);                     /*将字符串 st2 连接到 st1 的后面*/
   puts(st1);                           /*输出字符串 st1*/
}
```

程序运行结果如下（↙表示回车）：

```
input your name: LiLei↙
My name is LiLei
```

从程序 2.20 中也可以看出，整型的字符串数组和字符型的字符串数组是可以相互赋值的，在 C 语言中，二者可以看做是等同的。

注意

在使用 strcat 函数时，字符数组 1 应定义足够的长度，否则不能全部装入被连接的字符串。

4. 字符串拷贝函数 strcpy

格式：

```
strcpy (字符数组名 1, 字符数组名 2);
```

功能：把字符数组 2 中的字符串拷贝到字符数组 1 中，字符串结束标志‘\0’也一同拷贝。字符数组 2 也可以是一个字符串常量，这时相当于把一个字符串赋给一个字符数组。如程序 2.21。

用 strcpy 函数将 str2 中的字符串拷贝到 str1 中去：test21.c。

```
#include <string.h>                    /*字符串处理函数头文件*/
main()
{
   char st1[15],st2[]="C Language";    /*定义两个字符串数组*/
   strcpy(st1,st2);                    /*将 str2 中的字符串拷贝到 str1 中*/
   puts(st1);                          /*输出字符串 st1*/
}
```

程序运行结果如下：

```
C Language
```

注意

同 strcat 函数一样，使用 strcpy 函数时，字符数组 1 也应定义足够的长度，否则不能全部装入所拷贝的字符串。

5. 字符串比较函数 strcmp

格式：

```
strcmp(字符数组名 1, 字符数组名 2);
```

功能：按 ASCII 码值的大小逐个比较两个字符串数组中的各个字符，直到出现不同的字符或遇到‘\0’为止。函数的返回值有以下 3 种情况：

字符串 1＝字符串 2，返回值为 0。
字符串 1＞字符串 2，返回值为一正整数。
字符串 1＜字符串 2，返回值为一负整数。

strcmp 函数也可用于比较两个字符串常量，或比较数组和字符串常量，如程序 2.22。

比较两个字符串的大小：test22.c。

```
#include <string.h>                    /*字符串处理函数头文件*/
main()
{
 int k;
    static char st1[15],st2[]="abcd";  /*定义两个字符串数组*/
   printf("input a string: ");
   gets(st1);                          /*输入字符串 st1*/
   k=strcmp(st1,st2);                  /*比较字符串 st1 和 st2*/
   if(k==0) printf("st1=st2\n");       /*比较结果*/
   if(k>0) printf("st1>st2\n");
```

```
    if(k<0) printf("st1<st2\n");
}
```

程序运行结果如下（↙表示回车）：

```
input a string: abck↙
st1>st2
```

本程序中把输入的字符串和数组 st2 中的字符串比较，比较结果返回给变量 k，根据 k 值再输出比较结果。

6. 求字符串长度函数 strlen

格式：

```
strlen(字符数组名);
```

功能：求字符串的实际长度（不含字符串结束标志‘\0’），并作为函数返回值，如程序 2.23。

验证 strlen 函数的功能：test23.c。

```
#include <string.h>              /*字符串处理函数头文件*/
main()
{
int k;
   static char str[]="abcde";
   k=strlen(str);                  /*求字符串 str 的长度*/
   printf("The lenth of the string is %d\n",k);
}
```

程序运行结果如下：

```
The lenth of the string is 5
```

可以看到，求取字符串的长度时，是指字符串的实际长度，并不包含在内存中自动添加的字符串结束标识符‘\0’。

2.9 指针

指针是 C 语言中的一种数据类型。掌握指针型数据的使用，是深入理解 C 语言特性和掌握 C 语言编程技巧的重要环节，正确灵活地使用指针，可以有效地描述各种复杂的数据结构，能够动态地分配内存空间，能够方便地操作字符串，还可以自由地在函数之间传递各种类型的数据，使程序简洁、紧凑，执行效率高。

2.9.1 地址和指针

首先需要了解程序中的数据在内存中是怎样进行存储的。在对程序进行编译时，程序中定义的变量会被分配到内存中的某一个单元，内存单元的长度由变量的类型决定。例如，int 型变量分配 2 个字节，float 型变量分配 4 个字节，char 型变量分配 1 个字节。C 程序中的变量在内存中占有一个内存单元，每个内存单元由若干个字节组成，每个字节都有自己的编号（即地址），而一个变量的地址是指该变量的内存单元中第一个字节的编号。C 语言允许在程序中使用变量的地址，并可以通过地址运算符“&”得到变量的地址，例如：

```
float x;
int a[10];
```

通过&x 和数组名 a，就可以获得变量 x 和数组变量 a 的地址。

C 语言通过直接访问和间接访问两种方式来读取内存中的变量。直接访问是通过变量名或地址访问变量的存储区，例如：

```
scanf ("%d", &x );
x=sqrt(x);
printf ("%d", x );
```

间接访问是将一个变量的地址存放在另一个变量中。如图 2.10 所示，变量 x 的存储单元地址为 1010，将变量 x 的地址值存放在变量 p 中，访问 x 时先找到 p，再由 p 中存放的地址值找到 x。

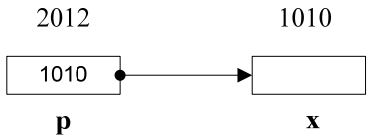


图 2.10　变量的间接访问

知道了数据在内存中的存储和读取方式后，指针的概念就不难理解了。一个变量的指针就是该变量的地址（指针就是地址），如变量 x 的指针即为 1010。

指针变量就是指存放某个变量的地址的变量，它用来指向另一个变量，如图 2.10 所示中的 p。

2.9.2　指针的定义和使用

对指针变量定义的一般形式为：

```
类型说明符 *变量名;
```

其中，*表示这是一个指针变量，变量名即为定义的指针变量名，类型说明符表示该指针变量所指向的变量的数据类型。例如：

```
int *p;   /*p 是指向整型变量的指针变量*/
```

说明

该语句表示 p 是一个指针变量，它的值是某个整型变量的地址，或者说 p 指向一个整型变量。至于 p 究竟指向哪一个整型变量，应由向 p 赋予的地址来决定。

在使用指针变量时，要首先对指针变量赋初值，使指针变量指向一个具体值。为指针变量赋值的方式有两种，使用赋值语句为指针赋初值和在定义指针变量的同时进行初始化。例如：

```
int a, *pa;
pa=&a;        /*方式一: 使用赋值语句为指针赋初值*/
int *pb=&a;   /*方式二: 定义指针变量的同时进行初始化*/
```

在指针定义和使用的过程中，经常会用到“&”和“*”这两个运算符。“&”是取地址运算符，“*”为指针运算符。例如：

```
int x=10, *p, y;
p=&x;      /*把变量 x 的地址赋给指针变量 p*/
y=* p;     /* *p 表示指针变量 p 所指单元的内容，即变量 x 的值，则 y=10 */
```

提示

在这个例子中，虽然第一条语句和第三条语句都出现了“*p”，但它们的意义却不同，这是因为“*”在类型说明和取值运算中的含义是不同的，初学者要多加注意。

2.9.3　数组与指针

前面我们已经知道，通过数组下标可以确定数组元素在数组中的顺序和存储地址。由于每个数组元素相当于一个变量，因此指针变量可以指向数组中的元素，也就是说可以用指针方式访问数组中的元素。

对一个指向数组元素的指针变量的定义和赋值方法，与指针变量相同。例如：

```
int a[10];       /*定义 a 为包含 10 个整型数据的数组*/
int *p;          /*定义 p 为指向整型变量的指针*/
p=&a[0];         /*把 a[0]元素的地址赋给指针变量 p*/
```

C 语言规定，数组名代表数组的首地址，也就是第 0 号元素的地址。因此：

```
p=a;         /*等价于 p=&a[0]; */
int *p=a;    /*等价于 int *p=&a[0]; */
```

对于指向首地址的指针 p，p+i（或 a+i）就是数组元素 a[i]的地址，*（p+i）（或*(a+i)）就是 a[i]的值。如果指针变量 p 已指向数组中的某一个元素，则 p+1 指向同一数组中的下一个元素。

引入指针变量后，就可以用以下两种方法来访问数组元素：

（1）下标法，即用 a[i]形式访问数组元素，在前面介绍数组时都是采用这种方法。

（2）指针法，即采用*(a+i)或*(p+i)形式，用间接访问的方法来访问数组元素，其中 a 是数组名，p 是指向数组的指针变量，其初值 p=a。具体实现过程如程序 2.24 中的代码。

定义一个数组，并用指针法访问数组的元素：test24.c。

```
#include <stdio.h>
main()
{
int i;
int a[5]={0,1,2,3,4};                    /*声明一个数组并对其进行初始化*/
int *p=a;                               /*把数组的首地址赋给指针变量 p*/
for(i=0; i<5; i++)
     printf("a[%d]=%d\n",i,*(a+i));  /*通过数组名计算元素的地址，找出元素的值*/
for(i=0; i<5; i++)
     printf("a[%d]=%d\n",i,*(p+i));  /*用指针变量指向元素*/
}
```

程序运行结果如下：

```
a[0]=0
a[1]=1
a[2]=2
a[3]=3
a[4]=4
a[0]=0
a[1]=1
a[2]=2
a[3]=3
a[4]=4
```

2.9.4 字符串与指针

前面我们已经讨论过字符数组与字符串，字符指针也可以指向一个字符串，可以用字符串常量对字符指针进行初始化。例如：

```
char *str = " This is a string." ;
```

这是对字符指针进行初始化。此时，字符指针指向一个字符串常量的首地址。

还可以用字符数组来存放字符串，例如：

```
char string[ ] = " This is a string.";
```

在这个语句中，string 是数组名，代表字符数组的首地址。因此可以通过数组名 string 来访问字符串。

字符串指针和字符串数组两种方式都可以访问字符串，但它们有着本质的区别：字符指针 str 是个变量，可以改变 str 使它指向不同的字符串，但不能改变 str 所指向的字符串常量的值。而 string 是一个数组，可以改变数组中保存的内容。读者应注意字符串指针和字符串数组的区别。

2.9.5 指向函数的指针

在定义一个函数之后，编译系统为每个函数确定一个入口地址，当调用该函数的时候，系统会从这个“入口地址”开始执行该函数。存放函数入口地址的变量就是一个指向函数的指针，简称为函数指针。函数指针定义的一般形式如下：

```
类型标识符 (* 指针变量名) ( );
```

类型标识符为函数返回值的类型。在 C 语言中，( )的优先级比*高，因此，“* 指针变量名”外部必须用括号，否则指针变量名首先与后面的( )结合。

函数指针必须赋初值，才能指向具体的函数。由于函数名代表了该函数的入口地址，因此可以直接用函数名为函数指针变量赋初值，即：

```
函数指针变量名 = 函数名;
```

例如：

```
double fun( );           /*函数说明*/
double (* f )( );    /*函数指针说明*/
f=fun;               /*f 指向 fun 函数*/
```

函数指针经定义和赋值之后，在程序中可以应用该指针，目的是调用被指针所指的函数。由此可见，使用函数型指针，增加了函数调用的方式。

2.10 结构体和共用体

在实际生活中，有大量由不同性质的数据构成的实体，如通信录就是由姓名、地址、电话、邮编等信息组成的。对于这种实体，用数组是难以描述的。因此，C 语言提供了一种被称为结构体的构造型数据类型，结构类型为处理复杂数据类型提供了便利的手段。

2.10.1 定义和引用结构体

结构体与数组类似，都是由若干分量组成的，与数组不同的是，结构体的分量可以是不同类型，可以通过成员名来访问结构体的元素。

结构体的定义说明了它的组成成员，以及每个成员的数据类型。定义一般形式如下：

```
struct 结构类型名
{
数据类型  成员名 1;
数据类型  成员名 2;
……
数据类型  成员名 n;
};
```

例如，定义一个结构体 address，用它来记录通信录信息：

```
struct address
{
       char name[30];          /*姓名，字符数组作为结构体中的成员 */
       char street[40];     /*街道*/
       unsigned long tel;    /*电话，无符号长整型作为结构体中的成员 */
       unsigned long zip;    /*邮政编码*/
}
```

结构的定义说明了变量在结构中的存在格式，要使用该结构就必须说明结构类型的变量。结构变量说明的一般形式如下：

```
struct 结构类型名称 结构变量名;
```

定义结构体便是定义了一种由成员组成的复合类型，而用这种类型说明了一个变量才会产生具体的实体。与说明基本数据类型的变量一样，系统会按照结构定义时的内部组成，为说明的结构变量分配内存空间。结构变量的成员在内存中占用连续的存储区域，所占内存大小为结构中每个成员的长度之和。

我们可以将变量 student1 说明为 address 类型的结构变量：

```
struct address student1;
```

虽然，结构体作为若干成员的集合是一个整体，但在使用结构时，不仅要对结构的整体进行操作，还经常要访问结构中的每一个成员。在程序中使用机构中成员的方法为：

```
结构变量名.成员名称
```

如 student1.tel 表示结构变量 student1 的电话信息。

和其他类型的变量一样，结构变量也可以进行初始化。结构初始化的一般形式如下：

```
struct 结构类型名 结构变量 =
{ 初始化数据 1, …… 初始化数据 n };
```

例如对变量 student1 的初始化可以用如下形式：

```
struct address student1 =
{ “wang”,”Road 1”, 123456, 1111 };
```

2.10.2 结构体数组

结构体数组是一个数组，其数组的每一个元素都是结构体类型。在实际应用中，经常用结构体数组来表示具有相同数据结构的一个群体，如一个班的学生档案，一个车间职工的工资表等。定义结构体数组和结构体变量相仿，只需说明它为数组类型即可。

比如定义一个结构体数组 student，包含 3 个元素：student[0]、student[1]、student[2]，每个数组元素都具有 struct address 的结构形式，并对该结构体数组进行初始化赋值：

```
struct address
{
char name[30];          /*姓名，字符数组作为结构体中的成员 */
char street[40];     /*街道*/
unsigned long tel;   /*电话，无符号长整型作为结构体中的成员 */
unsigned long zip;   /*邮政编码*/
}student[3]={
{"Zhang","Road NO.1",111111,4444},
{"Wang"," Road NO.2",222222,5555},
{"Li"," Road NO.3",333333,6666}
}
```

说明

当对全部元素进行初始化赋值时，也可不给出数组的长度。

访问结构体数组成员的一般格式为：

```
结构数组名[下标].成员名
```

比如通过语句：

```
printf("%s",student[1].name);
```

便可实现打印 student 数组中第 1 个元素的 name 成员值。

2.10.3 指向结构体的指针

当一个指针用来指向一个结构体变量时，称之为结构体指针变量。结构体指针变量中的值是所指向的结构变量的首地址，通过结构指针即可访问该结构变量。这与数组指针和函数指针的情况是相同的。结构体指针变量定义的一般形式为：

```
struct 结构类型名 *结构指针变量名
```

例如，我们在 2.12.1 节中定义了 struct address 结构类型，如要定义一个指向该结构类型的指针变量 pstu，可写为：

```
struct address *pstu;
```

当然也可在定义 struct address 结构类型时同时说明 pstu。与前面讨论的各类指针变量相同，结构指针变量也必须要先赋值后使用。

赋值是把结构变量的首地址赋予该指针变量，不能把结构名赋予该指针变量。如果 student1 是被说明为 struct address 类型的结构变量，则：

```
pstu = &student1;
```

就是对结构指针进行赋值。有了结构指针变量，就能更方便地访问结构变量的各个成员。其访问的一般形式为：

```
(*结构指针变量).成员名
```

或者：

```
结构指针变量->成员名
```

例如：

```
(*pstu).name
```

或者：

```
pstu->name
```

都是对 student1 结构体的 name 成员的访问。

注意

(*pstu)两侧的括号不可少，因为成员符"."的优先级高于"*"，如去掉括号写做*pstu.num 则等效于*(pstu.num)，这样，意义就完全不对了。

通过总结，不难发现，我们可以使用以下 3 种方式访问结构体中的成员：一是结构变量.成员名；二是(*结构指针变量).成员名；三是结构指针变量->成员名。这 3 种用于表示结构成员的形式是完全等效的。

2.10.4 共用体

在 C 语言中，允许几种不同类型的变量存放到同一段内存单元中，也就是使用覆盖技术，几个变量互相覆盖。这种几个不同的变量共同占用一段内存的结构，被称为共用体类型结构，简称共用体。一般定义形式为：

```
union 共用体名
{
    数据类型  成员名 1;
    数据类型  成员名 2;
    ……
    数据类型  成员名 n;
}变量名表列;
```

只有先定义了共用体变量，才能在后续的程序中引用它。不能直接引用共用体变量，而只能引用共用体变量中的成员。引用方法如下：

```
共用体变量名.成员名
```

共用体类型数据具有以下特点：

同一个内存段可以用来存放几种不同类型的成员，但是在每一瞬间只能存放其中的一种，而不是同时存放几种。换句话说，每一瞬间只有一个成员起作用，其他的成员不起作用，即不是同时都存在和起作用的。

共用体变量中起作用的成员是最后一次存放的成员，在存入一个新成员后，原有成员就失去作用。

共用体变量的地址和它的各成员的地址都是同一地址。

不能对共用体变量名赋值，也不能企图引用变量名来得到一个值，并且，不能在定义共用体变量时对它进行初始化。

不能把共用体变量作为函数参数，也不能是函数返回共用体变量，但可以使用指向共用体变量的指针。

共用体类型可以出现在结构体类型的定义中，也可以定义共用体数组。反之，结构体也可以出现在共用体类型的定义中，数组也可以作为共用体的成员。

程序 2.25 是一个关于共用体的定义与使用的例子。程序中首先声明一个名为 date 的共用体，并定义了共用体变量 a，然后根据输入的字符判断对共用体的哪一个成员变量进行赋值，最后输出该值。

共用体的定义与使用：test25.c。

```
#include <stdio.h>
main()
{
       char i;
       union date                              /*声明共用体数据类型*/
       {
          int day;                             /*共用体中的成员变量*/
          char month[20];
      int year;
       }a;                                     /*定义共用体变量 a*/
       scanf("%c",&i);                             /*输入判断字符 i*/
       if(i=='d') scanf("%d",&a.day);              /*若为 d，则输入的是 day 成员的值*/
       else if(i=='m') scanf("%s",&a.month);  /*若为 m，则输入的是 month 成员的值*/
       else if(i=='y') scanf("%d",&a.year);        /*若为 y，则输入的是 year 成员的值*/
       else printf("error input!\n");              /*错误字符*/
       if(i=='d') printf("a.day=%d\n",a.day); /*下面是输出共用体变量 a 的某个成员*/
       if(i=='m') printf("a.month=%s\n",a.month);
       if(i=='y') printf("a.year=%d\n",a.year);
}
```

程序运行结果如下（↙表示回车）：

```
d↙
30↙
a.day=30
m↙
September↙
a.month= September
y↙
2009↙
a.year=2009
```

需要提醒读者的是，由于同一时刻只能使用共用体变量中的某一个成员变量，而不是同时使用几个，所以程序 2.25 中必须使用判断字符 i，以判断将要使用共用体中的哪一个成员（day、month 或者是 year）。

2.10.5 使用 typedef 定义类型

在 C 语言中，除系统定义的标准类型和用户自定义的结构体、共用体等类型之外，还可以使用类型说明语句 typedef 定义新的类型来代替已有的类型。typedef 语句的一般形式是：

```
typedef 已定义的类型 新的类型;
```

例如：

```
typedef int INTEGER;        /*指定用 INTEGER 代表 int 类型*/
typedef float REAL;         /*指定用 REAL 代表 float 类型*/
```

在具有上述 typedef 语句的程序中，下列语句就是等价的：

```
int i, j; 与 INTEGER i, j;
float pi; 与 REAL pi;
```

2.11 链表

现实生活中存在大量需要动态存储和表示的数据，例如排队、数据排序等，这些问题都需要用链表的方式表示和处理。下面将对链表进行详细的介绍。

2.11.1 链表概述

链表是一种动态的数据结构。它是动态进行存储分配的一种结构。通常，对于大批的数据可以采用数组的方式保存，但使用数组保存存在明显的问题。首先，在 C 语言中，数组的大小在使用之前必须是确定的，一旦数据增加超过了数组的容量，就会发生数组溢出。为了保证不会发生数组溢出，当使用之前不能确定数组规模时，往往需要开设一个很大的数组，从而造成空间的浪费。如果在程序中采用动态数组的方式，即在数组增长的时候重新分配内存，然后将原始数据复制到新的数组中，这虽然是一种可行的办法，但效率太低。第二，如果要在数组中删除数据，就要将数组中删除点之后的数据向前移动；如果要在数组中插入数据，则必须将插入点后的元素向后移动。这种数据移动方式的效率同样是比较低的。链表正是针对数组的这些缺点而设计的一种存储数据的动态数据结构。

在链表中，所有数据元素都分别保存在一个具有相同数据结构的节点中，节点是链表的基本存储单位，一个节点与一个数据元素对应。每个节点在内存中使用一块连续的存储空间，每个节点可以使用不连续的存储空间，节点之间通过指针连在一起，连接节点的指针也称为链。

节点的存储结构在内存空间中通常分为两个部分：信息数据部分（也称为数据域）和连接节点的指针（也称为指针域）。节点定义采用结构体类型，一般形式为：

```
struct node
{
      datatype  data;     /*信息数据，根据实际数据定义*/
struct node * link;       /*指向节点 node 的指针*/
};
```

在这里，节点的数据类型名称是 struct node，data 是实际需要的结构成员分量，datatype 是实际分量所需要的数据类型，link 是一个指向 struct node 类型的结构指针，即 link 指向的对象是一个同样类型的数据节点，它是一个动态的指针，用来存放下一个节点的地址，通过 link 指针，一个个节点被依次连接起来，形成链表。

一个链表一般由 3 部分组成，分别为头指针、表头节点和数据节点。它们之间的关系如图 2.11 所示。

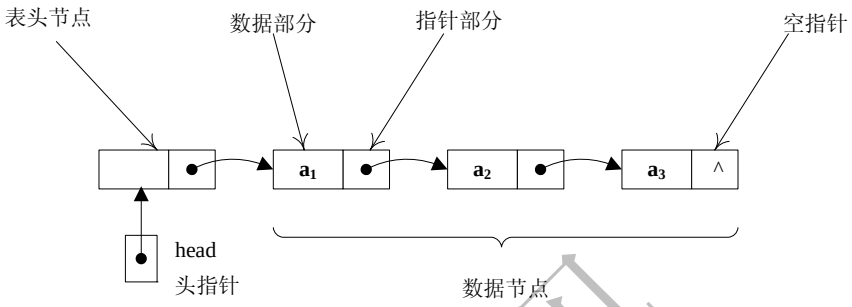


图 2.11　单链表结构

2.11.2　建立动态单向链表

建立链表首先要定义一个包含数据域和指针域的的结构类型，然后建立指向表头节点的头指针 head，最后通过 malloc 函数动态申请一块内存作为表头节点。

```
typedef struct node
{
      int data;              /*信息*/
      struct node *link;         /*指针*/
}NODE;                       /*定义节点*/
NODE *head;                  /*定义头指针 head */
```

定义结构类型和头节点之后，我们要建立不包含数据的表头节点，可以按下列语句进行操作。

```
NODE *p;                            /*说明一个指向节点的指针变量 p */
p=(NODE*) malloc(sizeof(NODE));  /*申请表头节点*/
p->link = NULL;                  /*将表头节点的 link 置为 NULL */
head=p;                             /*head 指向表头节点 p*/
```

此时链表的状态如图 2.12 所示，由于此时链表中只有一个表头节点，没有数据节点，所以称为空链表。

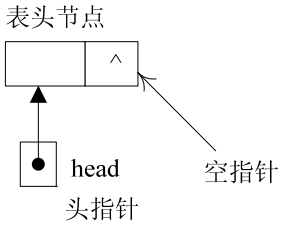


图 2.12　空链表

为了在链表中保存数据，可以从表头位置将数据节点插入到链表中，例如，插入一个数据节点：

```
p=(NODE*) malloc(sizeof(NODE));      /*申请一个数据节点*/
gets(p ->data);                      /*输入一个新的数据*/
p->link=head->link;                 /*建立链接关系。将表头节点的 link 存入 p 的 link 中*/
head->link=p;                          /*将数据节点插在表头节点之后成为第一个数据节点*/
```

插入第一个数据节点后链表如图 2.13 所示，然后继续插入下一个数据节点。

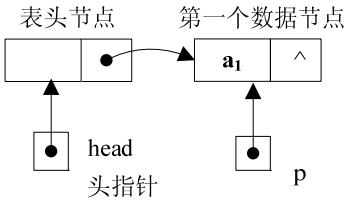


图 2.13　插入一个节点后的链表

根据上面的链表建立过程，可以写出函数 create 建立有 n 个数据节点的链表，如下所示：

```
create(NODE *head,int n)
{
      NODE *p;
for(; n>0;n--)
{
        p=(NODE*) malloc(sizeof(NODE));
        if(p==NULL)
           exit(0);
        gets(p->data);
        p->link = head->link;
        head->link = p;
}
}
```

2.11.3　单向链表的输出

将单向链表中各节点依次输出，首先要知道链表第一个节点的地址，然后设一个指针变量 p，先指向第一个节点，输出 p 所指的节点，然后使 p 后移一个节点再输出。直到链表的尾节点。如下面这段代码：

```
void print(NODE *head)
{
NODE *p;
      p=head;
      if(head!= NULL)
      do
      {
printf("%d /n", p->data);
         p=p->link;
      }while(p!=NULL)
}
```

2.11.4　对单向链表的删除操作

要删除链表中第 i 个节点的基本算法如下：

（1）定位第 i-1 个节点。指针 q 指向第 i-1 个节点，指针 p 指向被删除的节点。

（2）摘链。q->link = p->link。

（3）释放 p 节点。free(p)。

具体操作过程如图 2.14 所示。如图 2.15 所示是删除第 i 个节点后链表的状态。

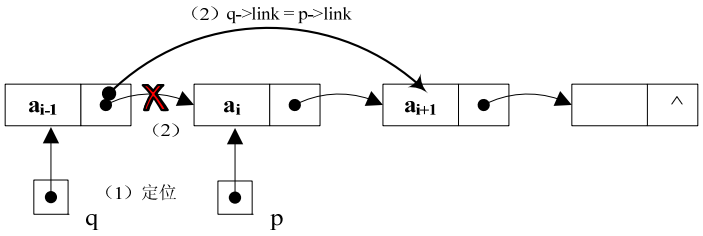


图 2.14　删除第 i 个节点的过程

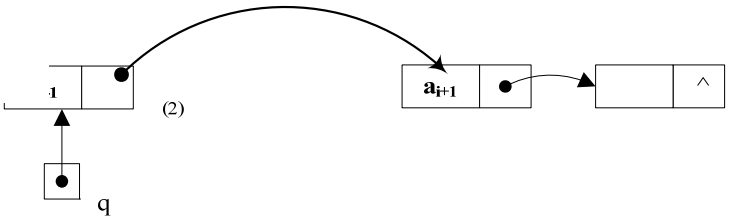


图 2.15　删除第 i 个节点后链表的状态

根据上述算法的基本思想，可以写出删除链表中第 i 个节点的程序如下：

```
delete_node(NODE *head, int i)
{
NODE *q, *p;
int n;
for(n=0,q=head;n<i-1&&q->link!=NULL;++n)
       q = q->link;              /*（1）定位第 i-1 个节点*/
if(i<0&&q->link!=NULL)
      {
p = q->link;                    /*p 指向被删除的第 i 个节点*/
         q->link = p->link;          /*（2）摘链*/
         free(p);               /*（3）释放 p 节点*/
      }
}
```

2.11.5　对单向链表的插入操作

在链表的第 i 个节点的后面插入一个新节点的基本算法如下：

（1）定位第 i 个节点。让指针 q 指向第 i 个节点，指针 p 指向需要插入的节点。
（2）链接后面指针。p-link = q->link。
（3）链接前面指针。q->link=p。

具体操作如图 2.16 所示。如图 2.17 所示是在第 i 个节点之后插入新节点后链表的状态。

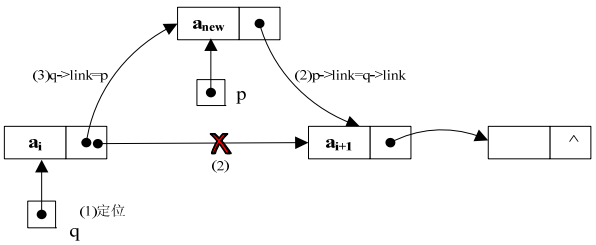


图 2.16　在第 i 个节点的后面插入数据节点的过程

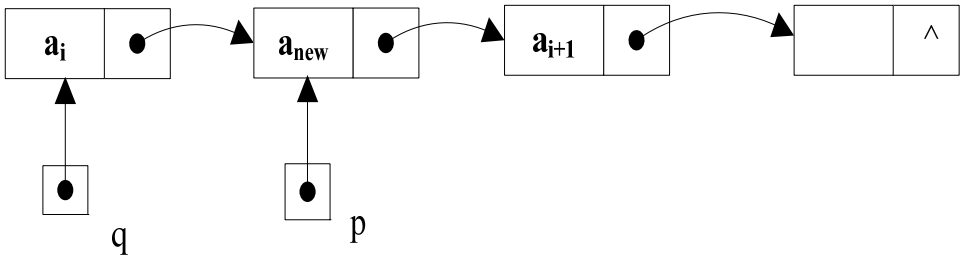


图 2.17　在第 i 个节点之后插入新数据节点后链表的状态

根据上述算法的基本思想，可以写出在链表第 i 个节点之后插入一个新数据节点 p 的程序如下：

```
insert_node(NODE *head, NODE *p, int i)
{
NODE *q;
       int n=0;
for(q=head; n<i&&q->link!=NULL;++n)
        q=q-link;              /*（1）定位第 i 个节点*/
p->link=q->link;          /*（2）链接后面指针*/
q->link=p;                /*（3）链接前面指针*/
}
```

2.11.6　循环链表

循环链表是另一种形式的表示线性聚集的链表，它的节点与单链表相同，与单链表不同的是链表中表尾节点的指针域中不是 NULL，而是存放了一个指向链表表头节点的指针，这样，只要知道表中任何一个节点的地址，就能遍历表中其他任一节点，如图 2.18 所示。

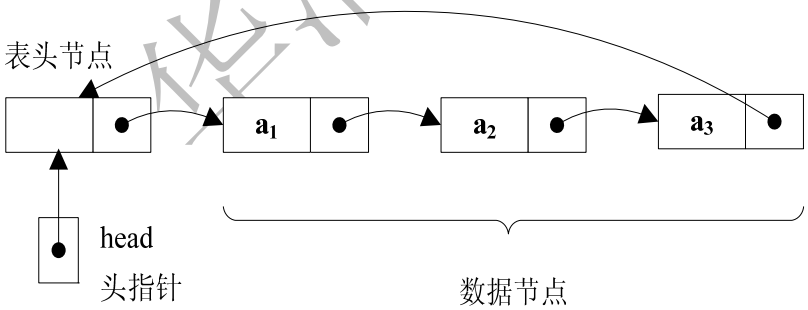


图 2.18　循环链表

循环链表的运算与单链表类似，但在涉及链头与链尾处理时稍有不同。例如，在实现循环链表的插入运算时，如果是在表的最前端插入，必须改变链尾最后一个节点的 link 域的值，这就需要搜索到最后一个节点。

2.11.7　双向链表

在单链表中，搜索一个指定节点的后继节点非常方便，只要该节点的 link 域的内容不为空，就可以通过 link 域找到该节点的后继节点地址。但是要搜索一个指定节点的前驱节点十分不容易，必须从链头开始，沿着 link 链顺序检测，直到某一节点的后继节点为该指定节点，则此节点即为该节点的前驱节点。为克服这一缺点，可以考虑双向链表。

在双向链表的每个节点中，应有两个链接指针作为它的数据成员：lLink 指示它的前驱节点，rLink 指示它的后继节点。因此双向链表的每个节点至少有 3 个域：

lLink（左链指针）	Data（数据）	rLink（右链指针）

节点之间的链接关系如图 2.19 所示。

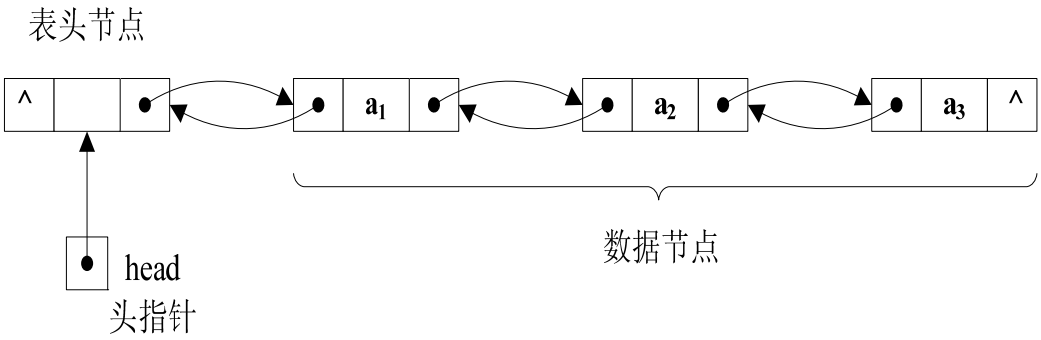


图 2.19　带表头的双向链表

指针 p 指向双向循环链表的某一节点，那么，p->lLink 指示 p 所指节点的前驱节点，p->lLink->rLink 中存放的是 p 所指前驱节点的后继节点的地址，即 p 所指节点本身。同样的，p->rLink 指示 p 所指节点的后继节点，p->rLink->lLink 也指向 p 节点本身。因此有 p == p->lLink->rLink == p->rLink->lLink，其过程如图 2.20 所示。

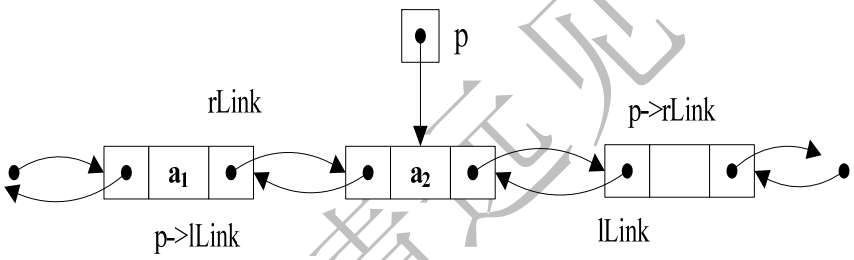


图 2.20　节点指针的指向

双向链表的的插入、删除操作的思想与单向链表是一样的，但操作时稍复杂一些。

如果要在当前指针 current 所指节点后面插入一个新节点 p，需修改 5 个指针，把新节点链入两个方向的链中，具体操作如下：

```
p->lLink=current;  p->rLink = current->rLink;  /*改新节点的两个链域*/
current->rLink=p;  current = current->rLink;   /*改前驱节点的后继链域*/
current->rLink->lLink=current;                  /*改后继节点的前驱链域*/
```

如果要删除双向链表中的一个节点，删除过程分两步，第一步先把当前节点从链中分离出来，修改前驱节点的后继指针和后继节点的前驱指针：

```
current->rLink->lLink=current->lLink;       /*从 lLink 链中摘下*/
current->lLink->rLink=current->rLink;       /*从 rLink 链中摘下*/
```

第二步再把当前节点释放。

2.12 位运算符和位运算

位运算是一种 C 语言提供的对二进制位的操作功能。它应用于整型数据，即把整型数据看成固定的二进制序列，然后对这些二进制序列进行按位运算。C 语言提供了 6 种基本位运算功能：按位与、按位或、取反、异或、左移、右移，下面将分别进行介绍。

2.12.1 “按位与”运算符 (&)

按位与运算是指对两个运算量相应的位进行逻辑与，“&”的运算规则与逻辑与“&&”相同。按位与表达式为 c = a & b，运算规则如图 2.21 所示。

	a:	1010,	1001,	0101,	0111
&	b:	0110,	0000,	1111,	1011
	c:	0010,	0000,	0101,	0011

图 2.21 按位与运算

2.12.2 “按位或”运算符（|）

按位或运算是指对两个运算量相应的位进行逻辑或，“|”的运算规则或逻辑与“||”相同。按位或表达式为 c = a | b，运算规则如图 2.22 所示。

	a:	1010,	1001,	0101,	0111
|	b:	0110,	0000,	1111,	1011
	c:	1110,	1001,	1111,	1111

图 2.22 按位或运算

2.12.3 “取反”运算符（~）

按位取反运算是将二进制表示的运算对象按位取反，即将 1 变为 0，将 0 变为 1。按位取反表达式为 c=~a，运算规则如图 2.23 所示。

	1	1	0	0
:	010,	001,	101,	111
	0	0	1	1
:	101,	110,	010,	000

图 2.23 取反运算

2.12.4 “异或”运算符（^）

按位异或运算的规则是：两个运算量的相应位相同，则结果为 0，相异则结果为 1。按位异或表达式为 c = a ^ b，运算规则如图 2.24 所示。

	a:	1010,	1001,	0101,	0111
^	b:	0110,	0000,	1111,	1011
	c:	1100,	1001,	1010,	1100

图 2.24 按位异或运算

2.12.5 移位运算符（<<和>>）

左移和右移是把整数作为二进制位序列，求出把这个序列左移若干位或者右移若干位后得到的序列。它们的一般形式为：x<<n 或 x>>n（x 是要被移位的量；n 是要移动的位数）。

左移运算规则是将 x 的二进制位全部向左移动 n 位，将左边移出的高位舍弃，右边空出的低位补 0。右移运算是将 x 的二进制位全部向右移动 n 位，将右边移出的低位舍弃，左边高位空出要根据原来量符号位的情况进行补充。对无符号数则补 0；对有符号数，若为正数则补 0，若为负数则补 1。

例如，设 a=5，则：

（1）b = a<<3 即 b = 0000,0101<<3 = 0010,1000 = 40。

（2）c = a>>2 即 c = 0000,0101>>2 = 0000,0001 = 1。

另外，左移运算等效于将整数值乘以 2 的幂；右移运算等效于将整数值除以 2 的幂，幂的大小即为左移或右移的位数。

2.12.6 位域

有些信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用 1 位即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为“位域”。

所谓“位域”，就是是把一个字节中的二进位划分为几个不同的区域，并说明每个区域的位数，每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。位域的定义和位域变量的定义形式为：

```
struct 位域结构名
{
类型说明符 位域名 1: 位域长度;
类型说明符 位域名 2: 位域长度;
……
类型说明符 位域名 n: 位域长度;
};
```

为了节省内存空间，可以把几个数据压缩到少数的几个类型空间中，比如需要表示两个 3 位的二进制数和一个 2 位的二进制数，则可以用一个 8 位的字符表示。如下所示定义一个 8 位的位域：

```
struct
{
char a : 3;
char b : 3;
char c : 2;
};
```

可以看到，这个结构体所占空间为一个字节（8 位），节省了内存空间。

位域的说明与结构变量说明的方式相同，可采用先定义后说明、同时定义说明或者直接说明这 3 种方式。使用位域时应注意以下几点：

一个位域必须存储在同一个字节中，不能跨两个字节。

位域的长度不能大于一个字节的长度，也就是说不能超过 8 位。

可以定义无名位域，这时它只能用来做填充或调整位置，无名位域在程序中是不能使用的。

2.13 C 语言预处理命令

预处理命令可以改变程序设计环境，提高编程效率，它们并不是 C 语言本身的组成部分，不能直接对它们进行编译，必须在对程序进行编译之前，先对程序中这些特殊的命令进行“预处理”。经过预处理后，程序就不再包括预处理命令了，最后再由编译程序对预处理之后的源程序进行编译处理，得到可供执行的目标代码。C 语言提供的预处理功能有三种，分别为宏定义、文件包含和条件编译，下面将对它们进行简单介绍。

2.13.1 宏定义

在 C 语言源程序中允许用一个标识符来表示一个字符串，称为“宏”，被定义为“宏”的标识符称为“宏名”。在编译预处理时，对程序中所有出现的宏名，都用宏定义中的字符串去代换，这称为“宏代换”

或“宏展开”。

宏定义是由源程序中的宏定义命令完成的，宏代换是由预处理程序自动完成的。在 C 语言中，宏分为有参数和无参数两种。无参宏的宏名后不带参数，其定义的一般形式为：

```
#define 标识符 字符串;
```

其中“#”表示这是一条预处理命令（在 C 语言中凡是以“#”开头的均为预处理命令），“define”为宏定义命令，“标识符”为所定义的宏名，“字符串”可以是常数、表达式、格式串等。符号常量的定义就是一种无参宏定义。

此外，常对程序中反复使用的表达式进行宏定义。例如：

```
#define M (y*y+3*y);
```

它的作用是指定标识符 M 来代替表达式(y*y+3*y)。在编写源程序时，所有的(y*y+3*y)都可由 M 代替，而对源程序进行编译时，将先由预处理程序进行宏代换，即用(y*y+3*y)表达式去置换所有的宏名 M，然后再进行编译。

C 语言允许宏带有参数。在宏定义中的参数称为形式参数，在宏调用中的参数称为实际参数。对于带参数的宏，在调用中，不仅要宏展开，而且要用实参去代换形参。

带参宏定义的一般形式为：

```
#define 宏名(形参表) 字符串;
```

在字符串中含有各个形参。

带参宏调用的一般形式为：

```
宏名(实参表);
```

例如：

```
#define M(y) y*y+3*y          /*宏定义*/
……
k=M(5);                       /*宏调用*/
……
```

在上面的宏调用时，用实参 5 去代替形参 y，经预处理宏展开后的语句为：

```
k=5*5+3*5;
```

程序 2.26 给出了一个宏定义和调用的完整实例。

定义一个名为 MAX 的带参数的宏，可以通过它来选出参数 a、b 中的较大值：test26.c。

```
#include <stdio.h>
#define MAX(a,b) (a>b)?a:b     /*带参数的宏定义*/
main()
{
   int x,y,max;
   printf("input two numbers: ");
   scanf("%d %d",&x,&y);
   max=MAX(x,y);                  /*宏调用*/
   printf("max=%d\n",max);   }
```

程序运行结果如下（□表示空格，↙表示回车）：

```
input two numbers: 2009□2010↙
max=2010
```

可以看到，宏替换相当于实现了一个函数调用的功能，而事实上，与函数调用相比，宏调用更能提高 C 程序的执行效率。

2.13.2 文件包含

文件包含是 C 预处理程序的另一个重要功能，文件包含命令行的一般形式为：

```
#include "文件名"
```

或者

```
#include <文件名>
```

文件包含命令的功能是把指定的文件插入该命令行位置取代该命令行，从而把指定的文件和当前的源程序文件连成一个源文件。

在程序设计中，文件包含是很有用的。一个大的程序可以分为多个模块，由多个程序员分别编程，有些公用的符号常量或宏定义等可单独组成一个文件，在其他文件的开头用包含命令包含该文件即可使用。这样，可避免在每个文件开头都去书写那些公用量，从而节省时间，并减少出错。

这里对 C 语言的文件包含命令进行以下几点说明：

（1）包含命令中的文件名可以用双引号引起来，也可以用尖括号引起来。例如以下写法都是允许的：

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

但是这两种形式是有区别的：使用尖括号表示在包含文件目录中去查找（包含目录是由系统的环境变量进行设置的，一般为系统头文件的默认存放目录，比如 Linux 系统在/usr/include 目录下），而不在源文件的存放目录中查找；使用双引号则表示首先在当前的源文件目录中查找，若未找到才到包含目录中去查找。用户编程时可根据自己文件所在的目录来选择某一种命令形式。

（2）一个 include 命令只能指定一个被包含文件，若有多个文件要包含，则需用多个 include 命令。

（3）文件包含允许嵌套，即在一个被包含的文件中又可以包含另一个文件。

2.13.3 条件编译

预处理程序提供了条件编译的功能，可以按不同的条件去编译不同的程序部分，因而产生不同的目标代码文件，这对于程序的移植和调试是很有用的。条件编译可分为三种形式。

第一种形式如下：

```
#ifdef 标识符
  程序段 1
#else
  程序段 2
#endif
```

它的功能是如果标识符已被 #define 命令定义过则对程序段 1 进行编译；否则对程序段 2 进行编译。如果没有程序段 2（为空），本格式中的#else 可以没有，即可以写为：

```
#ifdef 标识符
    程序段
#endif
```

第二种形式如下：

```
#ifndef 标识符
    程序段 1   #else
    程序段 2   #endif
```

与第一种形式的区别是将“ifdef”改为“ifndef”。它的功能是如果标识符未被#define 命令定义过则对程序段 1 进行编译，否则对程序段 2 进行编译。这与第一种形式的功能正好相反。

第三种形式如下：

```
#if 常量表达式
   程序段1  #else
   程序段2  #endif
```

它的功能是如果常量表达式的值为真（非 0），则对程序段 1 进行编译，否则对程序段 2 进行编译。因此可以使程序在不同的条件下完成不同的功能。

2.13.4 #error 等其他常用预处理命令

除了上面介绍的之外，C 语言还有#erroe、#line、#pragma 等其他常用的预处理命令，在很多 C 语言的程序中也是经常可见的。下面向读者简单介绍一下它们。

1. #error

#error 指令强制编译程序停止编译，它主要用于程序调试。#error 指令的一般形式是：

```
#error error-message
```

注意，宏串 error-message 不用双引号引起来。遇到#error 指令时，错误信息被显示，可能同时还显示编译程序作者预先定义的其他内容。

2. #line

#line 指令改变__LINE__和__FILE__的内容。__LINE__和__FILE__都是编译程序中预定义的标识符。标识符__LINE__的内容是当前被编译代码行的行号，__FILE__的内容是当前被编译源文件的文件名。#line 的一般形式是：

```
#line number “filename”
```

其中，number 是正整数并变成__LINE__的新值；可选的“filename”是合法文件标识符并变成__FILE__的新值。#line 主要用于调试和特殊应用。

3. #pragma

#pragma 是编译程序实现时定义的指令，它允许由此向编译程序传入各种指令。例如，一个编译程序可能具有支持跟踪程序执行的选项，此时可以用#pragma 语句选择该功能，编译程序忽略其不支持的#pragma 选项。使用#pragma 预处理命令可提高 C 源程序对编译程序的可移植性。

2.14 本章小结

本章较详细地讲解了 C 语言编程的基础知识，介绍的内容包括 C 的数据类型与运算规则、程序设计基本结构、数组、字符数据处理、指针、函数、结构体及其他构造类型、链表和预处理等。这些是在 Linux 下阅读和编写 C 程序的基础，读者务必熟练掌握和应用 C 语言。

实战演练

1. 编写一个程序，输出以下信息：

```
************************************************************
Hello, Linux world!
************************************************************
```

2. 编写一个程序，接受用户从键盘输入的字符，如果是小写字母则转换为大写字母，如果是大写字母，则原样输出。

3. 有一函数：

$$y=\begin{cases}x(x<1)\\2x-1(1\le x<10)\\3x-8(x\ge 10)\end{cases}$$

试编写一个 C 程序，输入 x 值，输出 y 值。

4. 编写一个程序，首先让用户在下面两个选项中选择一个：

A. 把温度从摄氏度转换为华氏度。

B. 把温度从华氏度转换为摄氏度。

然后提示用户输入温度值，输出转换后的新值。提示：把输入的值乘以 1.8，然后加 32，即可把摄氏度转换成华氏度。用输入的值减去 32，然后乘以 5，再用 9 除得到的结果，即可把华氏度转换成摄氏度。

5. 编写一个程序，从键盘读入 5 个 double 值，把它们存放到数组中。计算每个值的倒数（x 的倒数即 1.0/x），然后把它们存储在另外一个数组中。输出这些倒数值，计算并输出倒数的和。

6. 有一个 3×4 的矩阵，编写一个 C 程序，求所有元素中的最大值，以及该元素所在的行号和列号。

7. 编写一个程序，使用递归函数计算用户输入整数的阶乘值。

8. 编写一个程序，计算从键盘输入的任意多个浮点数的平均数。把所有值存储在开始计算之前动态分配的内存中，然后显示平均数，注意不能要求用户先声明要输入多少个值。

9. 定义一个 struct 类型，存放一个人的名字和他的电话号码。在一个程序中使用这个 struct，该程序允许输入一个或多个人的名字和对应的电话号码，然后把它们存储在一个结构数组的元素中。该程序应该能接受输入第二个人名，并输出与这个名字对应的所有号码。

10. 设 ha 和 hb 分别是两个带表头节点的非递减有序单链表的表头指针，试设计一个算法，将这两个有序链表合并成一个非递减有序的单链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其他的存储空间。表中允许有重复的数据。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-25590506

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

广州地址：广州市天河区中山大道 268 号天河广场 3 层，电话：020-28916067