第3章

C语言选择结构的程序设计

第2章我们学习了C语言中的多种基础运算，输入、输出语句以及顺序结构的程序设计等。顺序结构只能解决简单的问题，不能解决选择性的问题。比如，要计算一元二次方程 $ax^2+bx+c=0$ 的实数根，需要对 $\Delta=b^2-4ac$ 进行判断；根据3条边求三角形的面积，需要对三边关系进行判断等，必须用选择结构来解决。

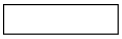
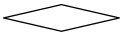
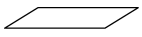
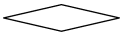
选择结构有3种形式：一是一个条件两项选择的形式，二是多个条件多项选择的形式，三是一个条件多种情况选择的形式。选择结构形式比较复杂，需要用流程图来描述。

3.1 流程图算法介绍

顺序结构的工作流程是一进一出，选择结构的工作流程是一进两出或者一进多出。

要解决问题，首先要有一个基本思路或者方法，可以把这种方法广义地称为算法。同一个问题，算法有很多，流程图就是常用的一种算法。

在顺序结构中有变量的定义、输入、公式计算以及结果的输出等语句，在选择结构中又遇到了分支判断语句，后面还会有循环结构等语句，这些语句都可以用图形来表示。

- 程序的开始或结束：用一个椭圆或圆角矩形来表示； 
- 执行语句：用一个矩形来表示； 
- 判断语句：用一个菱形来表示； 
- 输入或者输出语句：用一个平行四边形来表示； 
- 循环语句：用一个扁平的六边形或者菱形来表示； 
- 流程图各模块之间的连接：用一个箭头来表示； 
- 当流程图很大，在一个页面画不下时，可以用一个标注有顺序号的小圆圈来表示前后流程图之间的连接关系；圆圈中的数字编号前后要一致。○

我们把这种流程图的画法简称为“五个模块一条线”。

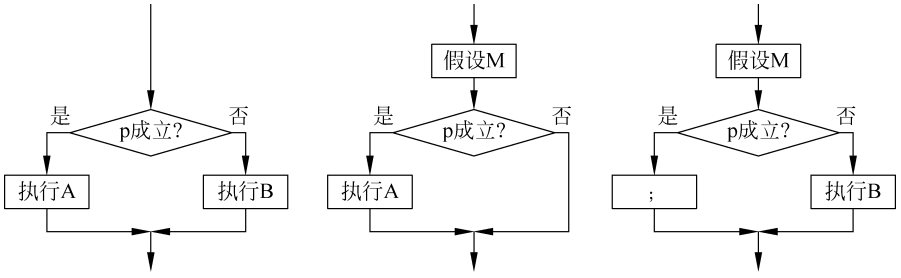
有了这“五个模块一条线”，对于绝大多数程序设计的结构流程图就可以很直观地描述出来。

3.2 单一选择结构的用法

1. 单一选择结构流程图介绍

单一选择结构就是一个条件两项选择的结构形式，它有一个条件，满足条件的是一种结果，不满足的是另一种结果。这个条件可能简单，也可能复杂，但有两种结果是必然的。

图 3-1 所示就是单一选择结构流程图，它有 3 种不同形式。



(a) 直接判断流程图　(b) 假设判断流程图之一　(c) 假设判断流程图之二

图 3-1　选择结构的 3 种基本模式

3 个图均由菱形和矩形模块构成，菱形代表条件判断，p 是判断的条件，矩形代表执行语句。A 和 B 是两种不同的执行语句。“是”表示条件成立的分支，“否”表示条件不成立的分支。

其中图 3-1(a)是对条件 p 的直接判断流程图；图 3-1(b)和图 3-1(c)是对条件 p 的假设判断流程图，也就是在条件 p 之前先做一个假设前提 M。图 3-1(b)和图 3-1(c)的不同分支输出，主要是因为条件 p 的判断方向发生了改变。在图 3-1(c)中，“;”也是一个执行语句，它没有具体的执行内容，也叫作空语句，只起到语句支撑的作用。如果没有它，那么程序会出现逻辑上的错误。

虽然图 3-1 中的 3 个流程图模式不同，但是输出的结果完全相同。所以，在实际应用中，任选哪一种都行。

根据箭头所示，菱形判断模块的执行路径都是“一进两出”。所以，画流程图时不要出现路径上的漏洞。不过，实际的运算过程还是一进一出的形式。

2. 单一选择结构的语句命令介绍

用一个菱形和两个分支来表示选择结构流程图很直观，用语句命令来描述也很简单，就是 if()和 else 两条语句。使用时判断条件要放在 if()语句的圆括号内，不能留空。

if(判断的条件)语句的作用就是判断圆括号内的条件是否成立？其结果是“逻辑值”，只有两个：要么成立，要么不成立。如果成立，逻辑值就为 1；如果不成立，逻辑值就为 0。

图 3-1(a)的语句如下：

```
if( p )
   执行 A;
else
   执行 B;
```

图 3-1(b)的语句如下：

```
执行假设 M;
if( p )
   执行 A;
执行 C;
```

图 3-1(c)的语句如下：

```
执行假设 M;
 if( p )
    ;
 else
    执行 B;
```

图 3-1(a)的功能是：如果条件 p 的逻辑值为 1，则说明条件成立，流程就去执行紧跟在 if()语句后面的执行语句 A；如果条件 p 的逻辑值为 0，则说明条件不成立，流程就去执行紧跟在 else 语句后面的执行语句 B。所以，在编写判断语句时，必须把条件成立的执行语句 A 紧放在 if()语句的后面，而把条件不成立的执行语句 B 紧放在 else 语句的后面，不能放错位置。

图 3-1(b)的功能是：根据假设条件 M，如果条件 p 成立，则执行语句 A，接着执行公共语句 C；若不成立则直接执行公共语句 C。

图 3-1(c)的功能是：根据假设的条件 M，如果条件 p 成立，就执行空语句“;”，否则执行语句 B。

不论是哪种流程图，选择结构的第一句命令一定是由 if()开始的，if(p)语句之后都要有“是”(成立)的执行语句。当“否”的分支上有执行语句时，必须要用 else 语句形式；没有语句时，就不需要 else 语句了。记住：在 if(p)的语句之后不能加“;”，否则就是条件 p 成立时什么都不做，是一个空语句。

例 3-1　从键盘上输入任意两个整数，找出其中的大数。

按照图 3-1(a)所示流程图设计的程序如下：

```
1  #include <stdio.h>
2  void main()
3  {
4      int a, b, max;
5      printf("a=?b=?");
6      scanf("%d%d",&a,&b);
7      if(a>b)
8         max=a;
9      else
10        max=b;
11     printf("a=%d\tb=%d\n",a, b);
12     printf("max=%d",max);
13 }
```

程序第 4 行是给 a、b、max 变量定义；第 5 行和第 6 行是给变量 a、b 从键盘上输入赋值；第 7～10 行就是按照图 3-1(a)所示流程图的模式通过 if()语句直接对变量 a、b 的值进行判断；如果 a＞b，把大数 a 赋值给 max，否则把大数 b 赋值给 max；第 11 行和第 12 行分别输出 a、b 和 max 的值。

如果按照图 3-1(b)所示流程图设计程序，那么只需要将图 3-1(a)对应程序中的第 7～10 行改为下面的语句形式，其他程序都不变：

```
7      max=a;                          //先假设 a 是大数
8      if(max<b)                       //或 b>max
9          max=b;
```

“max=a;”先假设 a 是大数。如果假设错，if(max<b)成立，则执行“max=b;”，否则不变。

如果按照图 3-1(c)所示流程图设计程序，只需要将图 3-1(a)对应程序中的第 7～10 行改为下面的语句形式，其他程序都不变：

```
7       max = a;
8       if(max > b)
9           ;                                   // 这个空语句形式必须要有，否则会出现逻辑错误
10      else
11          max = b;
```

“max=a;”先假设 a 是大数。如果假设对，if(max<b)成立，则执行“; ”，否则执行“max=b;”。

图 3-1(c)与图 3-1(b)程序不同的是判断语句 if(max>b)中的条件方向相反，所以，后续的执行不同。

以上 3 种判断方式虽然不同，但是执行结果是一样的，大家可以自己去验证。

在选择结构中，判断条件表达式的构成最重要，不同的判断表达式有着不同的判断结果。

3.3 选择结构判断条件的构成形式介绍

选择结构的判断条件有很多构成形式，比如，可以由关系表达式构成、由逻辑表达式构成、由算术表达式构成、由逗号表达式构成，还可以由独立的数字构成等。

1. 由关系表达式构成判断条件

C 语言中包含 6 种关系运算符号，分别是>、<、>=、<=、!=、==。

应注意它们与数学关系运算符号在写法上的区别。≥是数学中的大于或等于符号，而>=是 C 语言中的大于或等于符号，小于或等于也不同。尤其要注意不等于和等于符号的写法区别：≠是数学中的不等于，而“!=”是 C 语言中的不等于；=是数学中的等于，而==是 C 语言中的等于。在关系运算符号的写法上，后两种运算符号有很多人会写错，一定要牢牢地记住。

例 3-2 a>b，c>=d，a!=b，h==b 等都是正确的关系表达式写法。

关系运算的结果有两种：要么成立，要么不成立。成立时逻辑值为 1，不成立时为 0。

关系运算的优先顺序为：>、<、>=、<=，这 4 种运算级别相同，要优先算；当多个相连时，要从左到右运算。而!=、==这两种运算级别低一点，要后运算。

比如，y==a<b，因为“<”运算比“==”运算优先，所以，要先判断 a<b 的结果，然后再判断 y 与 a<b 的结果是否相等。

再比如，a>b!=c，要先判断 a>b 的结果，然后再判断该结果与 c 是否不相等。

再比如，假设 a=5，b=4，c=3，d=2，f=1；问 a>b>c<d==f 的逻辑值是多少？

这种判断题千万不要想当然，必须按照逻辑运算的规则进行。首先是“==”左边的关系运算要先运算，而且要从左到右运算，最后才能与 f 进行相等==的判断运算。

所以，先要对 a>b 进行判断运算，因为 a=5，b=4，所以，a>b 的判断运算结果等于 1。

然后要用这个判断结果1与c进行判断运算，因为c=3，所以1>3是错的，它的逻辑值为0，也就是a>b>c的逻辑值为0。

接着再进行0和d的判断运算，也就是0<d，因为d=2，所以0<2是成立的，它的逻辑值为1，也就是a>b>c<d的逻辑值为1。

最后再进行该逻辑值与f的相等判断运算，即1==f的判断运算。因为f=1，也就是1==1，这个逻辑判断是成立的，它的逻辑值为1，也就是a>b>c<d==f整式的逻辑值是1。大家看明白了吗？如果不是很明白，可以多做一些练习，加深理解。

我们也可以用一个简单的程序来验证a>b>c<d==f的逻辑运算值，因为逻辑运算的结果不是0就是1，就是最简单的整型数，所以，我们可以用下面的简单程序来验证：

```
#include <stdio.h>
void main()
{
    int a = 5, b = 4, c = 3, d = 2, f = 1;
    printf(" (a>b>c<d==f) = %d\n",a>b>c<d==f);
}
```

在该程序中，我们给5个变量定义时直接赋初值，然后用输出语句就可以直接输出逻辑运算的结果。程序的运行结果如下：

```
(a>b>c<d==f) = 1
Press any key to continue
```

可见，程序的运行结果与我们分析得出的结果1是完全一致的。希望大家学会这种验证方法。需要注意：if(1==1)永远是真的，if(0==0)也永远是真的。

2. 由逻辑表达式构成判断条件

选择结构的判断条件有相当一部分是由逻辑表达式构成的，C语言包含3种逻辑运算符号，分别是!、&&、||。! 表示逻辑“非”，“&&”表示逻辑“与”，||表示逻辑“或”。逻辑运算也有1和0两种结果。

例3-3　!A、!0、a&&b、c||d等都是正确的逻辑表达式写法。

逻辑运算的优先顺序为：!、&&、||。逻辑! 比较独立，它是单目运算，只需要一个数据即可运算，其运算级别最高，什么时候都可以优先运算。&& 和|| 都是双目运算，都需要两个数据参与运算，其运算级别依次降低。

凡是非0数字的! 非运算其结果都是0。比如，!5的逻辑值为0，!(-1)的逻辑值也是0。而0的! 非运算结果为1。

a&&b的“与”运算，只要a或者b中有一个是0，其“与”的逻辑运算结果就是0；只有a和b两个都不是0时，其“与”的逻辑运算结果才是1。如果 && 左边的逻辑值是0，右边的运算将不再进行。

比如a=1，b=2，c=3，d=4，m=5，n=6，经过(m=a>b)&&(n=c<d)判断运算后，a、b、c、d、m、n各是多少？因为该例 && 左边的逻辑值为0，所以右边不需要再算，最后的结果是：a=1，b=2，c=3，d=4，m=0，n=6，这也可以用程序进行计算验证。

a||b的“或”运算，只要a或者b中有一个不是0，其“或”的逻辑运算结果就是1；只有a和b两个都是0时，其“或”的逻辑运算结果才是0。如果|| 左边的逻辑值是1，右边的运算也将不再进行。

例 3-4 a=1,b=2,c=3,d=4,m=5,n=6,经过(m=a<b)||(n=c>d)判断运算后,a、b、c、d、m、n 各是多少?大家自己去验证练习。

3. 由算术表达式构成判断条件

在选择结构中,有时候会用算术表达式作为判断的条件。独立的数字也属于算术表达式,所以,独立的数字也可以作为判断的条件。比如,2+3,a-2,1,35,0 等都可以构成判断的条件。只要表达式的值不为 0,它的判断结果一定是真的,逻辑值一定是 1;如果表达式的值是 0,一定是假的,其逻辑值一定是 0。比如,if(1)永远是真的,而 if(0)永远是假的。

4. 由逗号表达式构成判断条件

逗号表达式也可以构成判断条件。

比如,if(i=2,m=5)和 if(a=2,c=a+3,d-c)就是由逗号表达式构成的判断条件。前一个判断的逻辑值为 1,因为 m=5 是真的。后一个判断中先需要计算 c=a+3,而判断的逻辑值要看 d-c 的结果,如果结果不为 0 就是真的,否则就是假的。

请看下面程序的例子:

```
1  #include <stdio.h>
2  void main()
3  {
4      int a,b,c,d;
5      a=2;c=a+3;b=2;
6      if(d=(a+c,c-b,b,a-1))
7          printf("\n\n  a=%d,b=%d,c=%d,d=%d\n",a,b,c,d);
8      else
9          printf("\n\n  d=%d,c=%d,b=%d,a=%d\n",d,c,b,a);
10 }
```

程序的第 6 行就是用逗号表达式作为判断的条件,而逗号表达式中每一个表达式都要进行计算,然后把逗号表达式的值赋给变量 d,再根据 d 的值判断条件是否成立?

根据逗号表达式的定义可知,该逗号表达式的值就是 a-1 的值,也就是 2-1=1,所以,把该表达式的值赋给变量 d,d=1,条件判断为真。程序的输出结果如下:

```
  a=2,b=2,c=5,d=1
Press any key to continue
```

输出结果是按照第 7 行 a、b、c、d 条件成立时的顺序输出的。

如果我们把第 6 行逗号表达式的判断条件修改成 if(d=(a+c,c-b,b,a-2))的形式,那么程序的输出结果如下:

```
  d=0,c=5,b=2,a=2
Press any key to continue
```

输出结果是按照第 9 行 d、c、b、a 条件不成立时的顺序输出的。

5. 选择条件的构成举例

选择条件的构成有简有繁,关键是要描述正确。

例 3-5 将 a>b>c 写成 C 语言要求的条件判断格式。

写法可能有 a>b>c、a>b&&a>c、a>b||a>c、a||b>c 等,其实这些写法都是错的,而正确的描述只有一种,即 a>b&&b>c。

再比如，a 是数字，一定要写成 a>=0&&a<=9 才对。

还有，b 是正数且能被 3 整除，正确的描述是：b>0&&b%3==0。

还有 year=2022，该年是闰年吗？正确的描述是：year/4==0&&year/400==0。

还有，ch 是字母，正确的描述是：ch>='a'&&ch<='z'||ch>='A'&&ch<='Z'，要包括小写字母和大写字母两种情况。

6. 综合算式的优先级与运算结果的归属类别

1）综合算式的优先级别与运算结果的归属类别

当一个 C 语言程序中同时包含()、+、-、*、/和%等综合算术运算时，优先级别为：先算括号，再算乘、除和求余，最后算加、减。其中，乘、除和求余属于相同级别的运算，要按照从左到右的先后顺序进行计算。

例 3-6　假设“s=3+(10-3)-6+6*4/8%2-3;”，则 s 等于多少？s 是什么类别？

第一步，计算(10-3)=7；

第二步，从左到右计算 6*4/8%2=1；

第三步，算加减，s=3+7-6+1-3=2。

这个例子的计算结果是一个整数，也就是变量 s 要定义为“int s;”。

不过，在 C 语言的计算中，不仅有 int 整型数运算，还有 char 字符型数运算，也有 float 单精度的浮点数运算，甚至还有 double 双精度的浮点数运算等。这些混合运算的优先级又怎么确定？计算的结果归属类别又怎么确定？

首先，算术运算优先级由高到低是：先算括号，再算乘、除和求余，最后算加、减。

其次，字符的运算都是按照字符的 ASCII 值进行运算，而字符的 ASCII 值就是一个整数，所以，字符的运算可以直接按照整数的相关运算方法进行处理。

最后运算结果的归属类别规律为：char→int→float→double。

对 char→int→float→double 归类的说明如下：

(1) 如果一个综合算式中只有字符型和整型数的运算，那么结果就是 int 整型数的类型；

(2) 如果一个综合算式中有单精度浮点数的运算，那么结果就是 float 单精度浮点数的类型；

(3) 如果一个综合算式中有双精度浮点数的运算，那么结果就是 double 双精度浮点数的类型。

2）逻辑运算综合算式的优先级别与运算结果的归属类别

如果在一个综合算式构成的条件表达式中，既包含算数表达式，又包含关系表达式，还包含逻辑表达式，那么其运算优先级别为：逻辑非或者括号→算数→关系→逻辑。也就是说，逻辑非和括号的运算优先级别最高，要先计算，然后是算术表达式运算，其次是关系表达式运算，最后是逻辑表达式运算。其中，

算数表达式运算的顺序规则依然为：先算乘、除、求余，后算加、减；

关系表达式运算的顺序为：>、<、>=、<=级别相同且优先，!=、==级别相同且延后；

逻辑表达式运算的顺序为：! →&&→||，而! 运算的级别最高，要先运算，||运算的级别最低，最后才能运算。

对于一个条件运算的式子而言，不论简单、复杂，其运算结果要么是 1，要么是 0，它们都

是 int 整型数。

例 3-7 对于综合条件表达式：5＜3＋7－!2||9－3＞＝4&&6＋4＞7，求其逻辑值。

其运算过程为：

```
5 < 10 - 0||6 > = 4&&10 > 7
5 < 10||6 > = 4&&10 > 7
      0 || 1 && 1
      0 || 1
```

该综合条件表达式的逻辑值为 1。

在选择结构中，一旦条件描述正确了，程序也就成功了一半。

3.4 位逻辑运算及判断条件的构成

位逻辑运算也是 C 语言中比较重要的运算之一，它也是数字控制的基础。采用位逻辑运算也可以构成逻辑判断条件。下面我们就学习与“位”相关的有关概念和逻辑运算。

大家都知道，在不少城市节日的夜空，会看到绚丽多姿、栩栩如生的无人机灯光秀，这些无人机灯光的亮、灭都可以用数字来控制，而这些数字的控制就离不开位逻辑运算。要学习位逻辑运算先要了解有关“位”的概念。

1. “位”与二进制数字的概念

在 2.1.3 节中对“位”的概念做过简单的介绍，“位”是二进制数的最小单位，是二进制数逻辑运算的基础。二进制数的运算规律就是：1＋1＝10。

二进制数的单位有“位”“字节”“字”“双字”等。英文对应的“位”“字节”“字”“双字”的单词分别为：Bit、Byte、Word、Double Word。它们之间的关系如下：

一个双字＝字＋字，也就是高字＋低字；

一个字＝字节＋字节，也就是高字节＋低字节；

一字节＝8 个位，也就相当于 8 个并列的开关。

2. 二进制数与十进制数字的对应关系

一个二进制数的字节排列规律与十进制数的对应关系如下：

“位”的编号为 7 6 5 4 3 2 1 0，总共 8 个位；

对应二进制数为 1 1 1 1 1 1 1 1，总共 8 个 1；

对应十进制数为 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0，每一“位”的十进制数取 2 对应位号的幂值；

一字节 8 个 1 的二进制数对应的十进制数依次为：

128 64 32 16 8 4 2 1

把这 8 个十进制数全部加起来就是一个字节 8 个 1 的二进制数对应的十进制总数：128＋64＋32＋16＋8＋4＋2＋1＝255。可见，十进制数 255 对应的就是二进制数字 8 个 1。

根据以上规律，可以把一个十进制整数转换为二进制数，例如，十进制数 9 的二进制数是 0000 1001；十进制数 13 的二进制数是 0000 1101。

3. 二进制的正负数相关规定

二进制数也有正、负之分，带符号的二进制数一般用“字”来表示，一个“字”有 16 个位，

最低“位”编号是0，最高“位”编号是15。

二进制数对符号位有规定：最高“位”就是符号位，也就是“字”的第15位是符号位，或者“双字”的第31位也是符号位，其余各位均为数字位。

二进制数规定：如果一个二进制数是“负数”，那么其最高“位”要设定为1；如果一个二进制数是“正数”，那么其最高“位”要设定为0。

下面举例说明正、负十进制数与正、负二进制数的对应关系：

十进制数9的二进制数是0000 0000 0000 1001；

十进制数－9的二进制数是1000 0000 0000 1001。

在位运算中，有时把符号位单列在数字位的最左边。

3.4.1　二进制数的位逻辑运算方法

二进制数的位逻辑运算总共有位“与”、位“或”、位“非”、位“异或”、位“左移”和位“右移”6种方式，其对应的逻辑运算符号依次为：&、|、～、^、<<、>>。

在位逻辑表达式运算中，位或(|)运算的级别最低要最后运算，位非(～)运算级别最高要先算，位左移(<<)和位右移(>>)运算的级别次于位非(～)，但高于位与(&)和位或(|)运算，也就是介于位非(～)和位与(&)及位或(|)运算之间。位异或(^)运算比较灵活，多种运算组合时级别不易确定，可高可低。

每一种二进制的位逻辑运算，可能涉及二进制的正数，也可能涉及二进制的负数，即便是同样的逻辑运算，它们的计算方法却大不相同，尤其是对二进制负数的运算。

1. 负数的补码运算

凡是涉及二进制负数的“位”运算很多都需要进行补码运算，加上计算机对二进制负数的存放也是采用补码的方式，所以，负数的补码运算还是比较重要的。正数也有补码，正数和负数的补码是不同的。

正数的补码就是正数本身，比如，9的二进制数和补码都是0000 0000 0000 1001。

负数的补码与正数完全不同，是要进行运算的。

负数的补码运算方法：负数的符号位1保持不变，对其余的数字位先逐位取反，然后再加1，结果就是负数的补码。

比如，十进制数－9的二进制数是：

```
1000  0000  0000  1001
```

符号位1不变，对数字位取反：

```
1111  1111  1111  0110
```

然后再进行加1运算：

```
0000  0000  0000  0001
```

所计算的结果就是－9的补码值：

```
1111  1111  1111  0111
```

再比如－5的补码变换用一字节简化表示如下：

－5的二进制数为：

```
1000  0101
```

对数字位取反为：

```
1111  1010;
```

再进行加 1 运算：

```
0000  0001
```

计算结果即补码：

```
1111  1011
```

有了负数补码运算的基础，二进制数的逻辑运算方法就容易理解和掌握了。

2. 变量的类型对位运算的影响

变量的基本类型有整型 int、浮点型 float 和字符型 char，位运算只适用于整型和字符型，浮点型不适用。整型有正整型、负整型或者 0。同样，字符型也有正、负之分。

C 语言中还有无符号整型和无符号字符型，其定义的关键词分别为 unsigned int 和 unsigned char。因为无符号数据所对应的二进制数最高位不再是符号位，而是数据位，所以，无符号的数据范围要比有符号的数据范围大。

无论是哪种类型的数据，正数的位运算规则都是一样的。而负数位运算的规则却差异很大，这一点一定要特别注意。

3. 二进制数的位“与”运算方法

位“与”运算的符号是 &，它是一个双目运算，其左、右两侧需要两个数，两个二进制数按对应位进行 & 运算。

& 运算的基本规则是：1&0=0，0&1=0，0&0=0，1&1=1。只有两个位都为 1 时，& 运算的结果才是 1，其余的运算结果都为 0。如果 & 运算的数是十进制数，要先将其转换为二进制数，然后逐位进行 & 运算，但正负数有差别。

1）正数的位“与”运算规则

正数的位“与”运算就是按对应位进行“与”运算即可。

例 3-8 *采用整型、无符号整型、字符型和无符号字符型分别计算 9&3 的值。*

采用整型变量计算的程序如下：

```
1  #include <stdio.h>
2  void main()
3  {
4      int a,b,c;  //也可用 unsigned int、char 和 unsigned char
5      a = 9; b = 3;
6      c = a&b;
7      printf("\n\n  a = %d\tb = %d\tc = %d\n",a,b,c);
8  }
```

程序的第 4 行是采用整型 int 定义了 3 个变量 a、b、c；

第 5 行是给整型变量 a、b 分别赋值为 9 和 3；

第 6 行是把变量 a&b 运算结果赋值给变量 c；

第 7 行是对 3 个变量的结果输出。

输出结果为：

```
  a = 9   b = 3   c = 1
Press any key to continue
```

当变量采用整型、无符号整型、字符型和无符号字符型时，运算结果均为 c＝1。

下面用二进制的字节形式进行简化验证。

9 的二进制数为：

```
0000  1001
```

3 的二进制数为：

```
0000  0011
```

逐位进行“与”运算：

```
0000  0001
```

所得的结果为 0000　0001，其对应的就是十进制数 1。所以，9&3＝1。

2）负数的位“与”运算规则

负数的位“与”运算先要对负数求补码，然后再对补码按位进行“与”运算。如果位“与”运算之后的结果中高数位仍然是 4 个 1 时，还需要对位“与”运算的结果再求一次补码运算，使高数位由 1 变为 0 即可。

例 3-9　采用整型、无符号整型、字符型和无符号字符型分别计算－9&3 的值。

将例 3-8 程序中的变量 a 改为 a＝－9，其他程序均不变，其输出结果为：

```
  a = -9   b = 3   c = 3
Press any key to continue
```

当变量为－9 和 3 时，采用整型、无符号整型、字符型时，其运算结果均为 c＝3。

如果将前面程序第 4 行的变量定义为无符号字符型“unsigned char a,b,c;”，其他程序均不变，其运行结果如下：

```
  a = 247   b = 3   c = 3
Press any key to continue
```

运算结果中 b、c 没有变。只是 a 并不是－9，而是 247，它是－9 的补码形式。

下面用二进制的字节形式进行简化验证。

－9 的二进制数为：

```
1000  1001
```

先对－9 求补码，然后再“与”运算。

对－9 数字位取反：

```
1111  0110
```

再加 1：

```
0000  0001
```

－9 的补码为：

```
1111  0111
```

3 的二进制数为：

0000 0011

逐位进行“与”运算：

0000 0011

所得的结果为：0000 0011，其对应的就是十进制数 3。所以，－9&3＝3。

例 3-10 用二进制数计算－13&－5 的值。

其运算方法如下：

－13 的二进制数为：

1000 1101

对－13 数字位取反：

1111 0010

再加 1：

0000 0001

－13 的补码为：

1111 0011

－5 的二进制数为：

1000 0101

对－5 数字位取反：

1111 1010

再加 1：

0000 0001

－5 的补码为：

1111 1011

然后对－13 的补码和－5 的补码进行“与”运算。

－13 的补码为：

1111 0011

－5 的补码为：

1111 1011

逐位进行“与”运算：

1111 0011

因为计算结果中的高数位仍然是 4 个 1，所以还需要对该计算结果再进行一次补码运算就是所求的结果。

前“与”结果为：

1111 0011

再进行取反为：

```
1000  1100
```

再加 1：

```
0000  0001
```

进行补码运算后的结果为：

```
1000  1101
```

可见，所得的结果为：1000　1101，其对应的就是十进制数－13。所以，－13&－5＝－13。

如果采用程序用 int、unsigned int 或者 char 对变量 a、b、c 进行定义，并赋值为：a＝－13，b＝－5，然后计算 c＝a&b 的值，其运算结果为：－13&－5＝－13。

如果用 unsigned char 对变量 a、b、c 进行定义，并赋值为：a＝－13，b＝－5，然后计算 c＝a&b 的值，其运算结果为：a＝243，b＝251，c＝243，这个结果实际上就是 3 个变量负数值的补码形式。虽然输出的结果形式不同，但是，负数的位“与”运算规则和执行过程是完全一致的。

3）位“与”运算应用举例

例 3-11　将 243 的 4 个高数位清 0，保留低数位。

十进制数 243 对应的二进制数用字节表示为：1111 0011，要将 4 个高数位清 0，保留低数位，我们可以用 243 跟 15 进行位“与”运算就可以达到预期的目的。用字节简化二进制数表示如下：

```
1111 0011(243)
0000 1111(&15)
0000 0011( = )
```

所以，243&15＝3，实现了高数位清 0。

例 3-12　保留 243 对应二进制数 1111 0011 的右边第 2 位。

243 对应二进制数右边的第 2 位数就是十进制的 2，所以可以用 243 跟 2 的位“与”运算来解决。用二进制数表示如下：

```
1111 0011(243)
0000 0010(&2)
0000 0010( = )
```

所以，243&2＝2，这样就保留了原数右边的第 2 位不变，其他位全为 0。

4. 二进制数的位“或”运算方法

位“或”运算的符号是“|”，它也是一个双目运算，其左、右两侧需要两个数，两个二进制数按对应位进行|运算。

位“或”运算的基本规则是：1|0＝1，0|1＝1，1|1＝1，0|0＝0。只有两个位都为 0 时运算结果才是 0，其余的运算结果都为 1。如果位“或”运算的数是十进制数，那么要先将其转换为二进制数，然后进行逐位“或”运算。

1）正数的位“或”运算规则

正数的位“或”运算就是按对应位进行“或”运算即可。

例 3-13　采用整型、无符号整型、字符型和无符号字符型分别计算 9|3 的值。

采用整型变量计算的程序如下：

```
1  #include <stdio.h>
2  void main()
3  {
4      int a,b,c; //也可用 unsigned int、char 和 unsigned char
5      a = 9; b = 3;
6      c = a|b;
7      printf("\n\n  a = %d\tb = %d\tc = %d\n",a,b,c);
8  }
```

程序的第 4 行定义了 3 个整型变量 a、b、c；

第 5 行是给整型变量 a、b 分别赋值为 9 和 3；

第 6 行是把变量 a|b 运算结果赋值给变量 c；

第 7 行是对 3 个变量的结果输出。

输出结果为：

```
  a = 9   b = 3   c = 11
Press any key to continue
```

当变量采用整型、无符号整型、字符型和无符号字符型时，其运算结果均相同。

下面用二进制的字节形式进行简化验证。

9 的二进制数为：

```
0000  1001
```

3 的二进制数为：

```
0000  0011
```

逐位进行|运算结果为：

```
0000  1011
```

所得的结果 0000 1011 对应的就是十进制数 11。所以，9|3＝11。

2）负数的位“或”运算规则

负数的位“或”运算先要对负数求补码，然后再对补码按位进行“或”运算。如果位“或”运算之后的结果中高数位仍然是 4 个 1 时，还需要对位“或”运算的结果再求一次补码运算，使高数位由 1 变为 0 即可。

例 3-14　采用整型、无符号整型、字符型和无符号字符型分别计算－9|3 的值。

将例 3-13 程序中变量 a 改为 a＝－9，其他程序均不变，其输出结果为：

```
  a = -9   b = 3   c = -9
Press any key to continue
```

可以验证，当变量的类型为无符号整型和字符型时，运算的结果也是 c＝－9。

当变量的类型为无符号字符型时（unsigned char a,b,c;），运算的结果如下：

```
  a = 247   b = 3   c = 247
Press any key to continue
```

这个输出结果实际上就是负数值的补码形式，其中 247 是－9 的补码。

下面用字节形式进行简化验证。

－9 的二进制数为：

1000　1001

对－9数字位取反：

1111　0110

再加1：

0000　0001

－9的补码为：

1111　0111

3的二进制数为：

0000　0011

逐位求“或”为：

1111　0111

可见，所得的结果为：

1111　0111

因为计算结果的高数位仍然是4个1，还需要对位“或”运算的结果再进行一次“补码”运算即为所得。

原来的位或运算结果为：

1111　0111

再取反为：

1000　1000

再加1：

0000　0001

进行补码运算后的结果为：

1000　1001

所得的结果1000　1001对应的就是十进制数－9。所以，－9|3＝－9。

例3-15　用二进制的方法计算－13|－5的值。

用二进制方法验证为：－13|－5＝－5。

如果用程序验证也是：a＝－13，b＝－5，c＝－5。

如果用unsigned char对变量a、b、c进行定义，其运算结果为：a＝243，b＝251，c＝251。其中，243是－13的补码，251是－5的补码。虽然结果形式不同，但是，负数的位“或”运算规则和执行过程也是完全一致的。

3）位“或”运算应用举例

例3-16　将3的4个低数位置1，高数位不变。

十进制数3对应的二进制数用字节表示为：0000 0011，要将4个低数位置1，高数位不变，用3跟15进行位“或”运算就可以达到预期的目的。用二进制数表示如下：

0000 0011(3)

```
0000 1111(|15)
0000 1111( = )
```

所以，3|15＝15，实现了低数位置 1。

例 3-17 将 3 对应的二进制数 0000 0011 左边的第 2 位置 1。

3 对应二进制数左边的第 2 位数就是十进制的 64，所以，我们可以用 3 跟 64 的位“或”运算来解决。用二进制数表示如下：

```
0000 0011(3)
0100 0000(|64)
0100 0011( = )
```

所以，3|64＝67，这样就将原数左边的第 2 位置 1，其他位均不变。

5. 二进制数的位“非”运算方法

位“非”运算的符号是“～”，它是一个单目运算，其右边需要跟一个数。

位“非”运算的基本规则是：按位取反，当位值为 1 时取 0，为 0 时取 1。

不少教科书上将位“非”运算笼统地称为“取反”运算，这是不恰当的，可能会误导学习者。因为位“非”运算并不是简单的“取反”运算，不要理解为：～5＝－5；也不要简单地理解为按位“取反”就是：～1＝0、～0＝1。其实，这些数据的“非”运算都是错误的。因为，这些运算数据都是十进制的数据，与二进制的数据是不同的。所以，二进制数的位“非”运算另有玄机。

如果位“非”运算的数是十进制数，那么要先将其转换为二进制数，然后再做位“非”运算。

正数和负数的位“非”运算规则不同，而且与正数或者负数定义的变量类型密切相关。

1）正数的位“非”运算规则

当变量定义的类型为整型、无符号整型或者字符型时，正数的位“非”运算要先对正数按位“取反”，然后再求“补码”运算即为所得。

当变量定义的类型为无符号字符型时，正数的位“非”运算就等于按位“取反”后所得的十进制数，不再需要求“补码”运算了。

例 3-18 采用整型、无符号整型、字符型和无符号字符型分别计算～1 的值。

采用整型变量计算的程序如下：

```
1  #include <stdio.h>
2  void main()
3  {
4      int a,c;
5      a = 1;
6      c = ~a;
7      printf("\n\n  a = %d\tc = %d\n",a,c);
8  }
```

程序的第 4 行定义了两个整型变量 a、c；

第 5 行是给整型变量 a 赋值为 1；

第 6 行是把变量～a 运算的结果赋值给变量 c；

第 7 行是结果输出。

输出结果为：

```
  a = 1    c = -2
Press any key to continue
```

可以验证，当变量的类型为无符号整型和字符型时，运算结果都是 c＝－2。

如果将例 3-18 程序中的变量改为“unsigned char a, c;”，其他部分均不变，其输出结果为：

```
  a = 1    c = 254
Press any key to continue
```

c＝254 是对 1 的二进制数逐位取反后累加所得的十进制数。这个结果与前面的差异很大，实际上它就是－2 的补码。可见，当变量定义为无符号字符型时，直接逐位取“反”即可，不需要再进行“补码”运算。

下面用字节形式进行简化验证。

1 的二进制数为：

```
0000  0001
```

先按位取反为：

```
1111  1110
```

再求补码取反为：

```
1000  0001
```

加 1 为：

```
0000  0001
```

求补码所得为：

```
1000  0010
```

可见，所得的结果为：1000　0010，其对应的就是十进制数－2。所以，～1＝－2。这就是整型、无符号整型和字符型的位“非”运算结果。

对 1 逐位取“反”所得的结果 1111　1110 对应的就是十进制数 254，这就是无符号字符型的位“非”运算结果。可见，无符号字符型的位“非”运算要少一个求补码的运算环节。按照类似的程序和方法计算～9 的值，所得的结果为－10，或者是 246。大家可以自己去验证。

2）负数的位“非”运算规则

不论变量是什么类型，负数的位“非”运算都是先对负数求补码，然后再按位进行“取反”运算即为所得。

例 3-19　采用整型、无符号整型、字符型和无符号字符型分别计算～(－1)的值。

将例 3-18 程序中的变量 a 改为 a＝－1，其他程序均不变，其输出结果为：

```
  a = -1    c = 0
Press any key to continue
```

可以验证，当变量的类型为无符号整型和字符型时，运算结果也是 c＝0。

当变量的类型为无符号字符型时，其运算结果为：

```
  a = 255    c = 0
Press any key to continue
```

其中，a＝255 是－1 的补码。而 c＝0 就是～(－1)＝0，与其他类型的位“非”运算结果完全相同。

下面用字节形式进行简化验证。

－1 的二进制数为：

```
1000  0001
```

对－1 求补取反为：

```
1111  1110
```

再加 1：

```
0000  0001
```

－1 的补码为：1111 1111，其对应的就是十进制数 255。

逐位取反为：

```
0000  0000
```

可见，所得的结果 0000 0000 对应的就是十进制数 0。所以，～(－1)＝ 0。

根据以上对正数和负数位“非”的计算结果，对于整型、无符号整型和字符型的变量，其位“非”的运算也可以得出简便算法：

位“非”运算等于原数加 1 后符号取反即可。

比如，～5＝ －6；～25＝ －26；～(－7)＝6；～(－25)＝24 等，大家可以自己去验证。

6. 二进制数的位“异或”运算方法

位“异或”运算的符号是“^”，也是一个双目运算，其左、右两侧需要两个数，两个二进制数按对应位进行“异或”运算。

位“异或”运算的基本规则是：1^0＝1，0^1＝1，1^1＝0，0^0＝0。当两个位的状态值不同时，其运算结果是 1，相同时为 0。如果位“异或”运算的数是十进制数，要先将其转换为二进制数，然后再进行逐位“异或”运算。

1）正数的位“异或”运算规则

正数的位“异或”运算就是按对应位进行“异或”运算即可。

例 3-20　采用整型、无符号整型、字符型和无符号字符型分别计算 9^3 的值。

采用整型变量计算的程序如下：

```
1  #include <stdio.h>
2  void main()
3  {
4      int a,b,c;
5      a = 9; b = 3;
6      c = a^b;
7      printf("\n\n  a = %d\tb = %d\tc = %d\n",a,b,c);
8  }
```

第 7 行就是把变量 a^b 异或运算的结果赋值给变量 c。

输出结果为：

```
  a = 9    b = 3    c = 10
Press any key to continue
```

当变量的类型为无符号整型、字符型、无符号字符型时，运算结果也是 c＝10。

下面用字节形式进行简化验证。

9 的二进制数为：

```
0000  1001
```

3 的二进制数为：

```
0000  0011
```

逐位进行“异或”运算的结果为：

```
0000  1010
```

可见，所得的结果 0000　1010 对应的就是十进制数 10。所以，9^3＝10。

2）负数的位“异或”运算规则

负数的位“异或”运算要对负数先求补码后再按对应位进行“异或”运算。如果位“异或”运算之后的结果中高数位仍然是 4 个 1 时，则需要对位“异或”运算的结果再求一次“补码”运算，使高数位由 1 变为 0 即可。

例 3-21　采用整型、无符号整型、字符型和无符号字符型分别计算(－9)^3 的值。

将例 3-20 程序中的 a 改为 a＝－9，其他程序均不变，其输出结果为：

```
  a = -9   b = 3   c = -12
Press any key to continue
```

可以验证，当变量的类型为无符号整型和字符型时，运算的结果也是 c＝－12。

当变量的类型为无符号字符型时，运算的结果如下：

```
  a = 247   b = 3   c = 244
Press any key to continue
```

其中，a＝247 是－9 的补码，c＝244 是－12 的补码，这是无符号字符型运算用补码的特点。

下面用字节形式进行简化验证。

－9 的二进制数为：

```
1000  1001
```

对－9 数字位取反：

```
1111  0110
```

再加 1：

```
0000  0001
```

－9 的补码为：

```
1111  0111
```

3 的二进制数为：

```
0000  0011
```

逐位进行“异或”运算的结果为：

```
1111  0100
```

可见，所得的结果为：

```
1111  0100
```

因为计算结果的高数位仍然是 4 个 1，所以还需要对位“异或”运算的结果再计算一次“补码”运算即为所得。

原来的运算结果为：

```
1111  0100
```

再取反为：

```
1000  1011
```

再加 1：

```
0000  0001
```

进行补码运算后的结果为：

```
1000  1100
```

可见，所得的结果 1000 1100 对应的就是十进制数－12。所以，(－9)^3＝－12。如果采用无符号字符型运算，其结果为 1111 0100，也就是十进制数 244。

例 3-22　计算(－13)^(－5)的值。

验证结果为：(－13)^(－5)＝8，大家可以自己用程序去验证。

3）位“异或”运算应用举例

例 3-23　采用位“异或”运算对－9 清 0。

采用整型变量计算的程序如下：

```
1  #include <stdio.h>
2  void main()
3  {
4      int a = -9;
5      a = a^a;
6      printf("\n\n  a = %d\n",a);
7  }
```

第 5 行是把变量 a^a 异或运算的结果再赋值给变量 a。

输出结果为：

```
  a = 0
Press any key to continue
```

可见，输出变量 a 的值已经由－9 变为 0。

用二进制的运算验证也是：(－9)^(－9)＝0。这种方法对所有的数据都是如此。

例 3-24　假如有两个整数 a＝3，b＝－7，请将 a、b 两个数对调。

在 C 语言程序中要对调两个数是需要第 3 个中间变量的，比如，“int c; { c＝a; a＝b; b＝c;}”，这样就可以将 a、b 两个数对调。而利用位的“异或”运算，可直接对两个变量进行对调，不需要第 3 个中间变量。

请看下面的程序：

```
1  #include <stdio.h>
2  void main()
3  {
```

```
4       int a = -9, b = 3;
5       printf("\n\n  原来的 a = %d\tb = %d\n",a,b);
6       a = a^b;
7       b = b^a;
8       a = a^b;
9       printf("  对调后 a = %d\tb = %d\n",a,b);
10 }
```

第 6～8 行 3 次采用位“异或”运算，对变量 a、b 的值进行对调。

输出结果为：

```
  原来的:a = -9    b = 3
  对调后:a = 3     b = -9
Press any key to continue
```

可见，对调的结果正确。

用二进制的位“异或”运算验证结果完全一致，大家可以自己去验证。

7. 二进制数的位“左移”运算方法

位“左移”运算的符号是“<<”，它也是一个双目运算，左侧是要移位的二进制数，右侧是要移位的位数。位“左移”的运算符号“<<”看起来像是一个双箭头指向左侧，所以“<<”是位的左移运算。位“左移”是对整个二进制数一起向左移位。

十进制数的位“左移”要先将其转换为二进制数，然后再“左移”运算。

1）正数位“左移”运算规则

正数位“左移”时移出的“位数”丢弃，在右侧补 0，移出几位就补几个 0。

例 3-25　采用整型、无符号整型、字符型和无符号字符型分别计算 9 << 1 的值。

采用整型变量计算的程序如下：

```
1  #include <stdio.h>
2  void main()
3  {
4       int a = 9, b;
5       b = a << 1;
6       printf("\n\n  a = %d\tb = %d\n",a,b);
7  }
```

第 5 行是把变量 a << 1 左移运算的结果赋值给变量 b。

输出结果为：

```
  a = 9    b = 18
Press any key to continue
```

可见，正数 9 左移 1 位后的运算结果是 18，是原值 9 的 2 倍。如果采用其他 3 种类型定义变量 a、b，其运算结果是一样的。

用二进制字节简化验证也是 9 << 1＝18。如果对 9 左移 2 位，所得的结果为 9 << 2＝36。

2）负数位“左移”运算规则

负数位“左移”时符号位 1 保持不变，只移左侧的数据位，移出的“位数”丢弃，在右侧补 0，移出几位就补几个 0。

例 3-26　采用整型、无符号整型、字符型和无符号字符型分别计算(－9)<< 1 的值。

将例 3-25 程序中的变量 a 改为 a＝－9，其他程序均不变，其输出结果为：

```
  a = -9    b = -18
Press any key to continue
```

可见，负数－9左移1位后的运算结果是－18，也是原值－9的2倍。如果采用无符号整型和字符型定义变量a、b，程序的运算结果与整型的结果也是一样的。

如果采用无符号字符型定义变量a、b，那么程序运算的结果如下：

```
  a = 247   b = 238
Press any key to continue
```

a＝247是a＝－9的补码形式，b＝238也是b＝－18的补码形式，运算实质并没有改变。

用二进制数字节简化验证为－9 << 2＝－36。

根据以上对正数和负数位“左移”的计算结果，可以得出简便算法：一个数的位“左移”运算等于原数乘以2的移位次数幂的倍数。

比如，6 << 2＝ $6\times2^2=24$，$-7<<2=-7\times2^2=-28$，大家可以自己去验证。

8. 二进制数的位“右移”运算方法

位“右移”运算的符号是“>>”，与位的“左移”运算类似，也是一个双目运算，左侧是要移位的二进制数，右侧是要移位的位数。位“右移”的运算符号“>>”看起来就像是一个双箭头指向右侧，所以“>>”是位的右移运算。位“右移”是对整个二进制数一起向右移位。

十进制数的位“右移”运算要先将其转换为二进制数，然后再“右移”运算。

1）正数位“右移”运算规则

正数位“右移”时，移出“位”丢弃，在左侧高数位补0，移出几位就补几个0。

例3-27 采用整型、字符型和无符号字符型、无符号整型分别计算正数9 >> 1的值。

采用整型变量计算的程序如下：

```
1  #include <stdio.h>
2  void main()
3  {
4      int a = 9, b;
5      b = a >> 1;
6      printf("\n\n  a = %d\tb = %d\n",a,b);
7  }
```

第5行是把变量a >> 1右移运算的结果赋值给变量b。

输出结果为：

```
  a = 9    b = 4
Press any key to continue
```

可见，正数9右移1位后的运算结果是4。若是其他3种类型，其运算结果也是b＝4。

用二进制字节形式简化验证为9 >> 1＝4。如果对9右移2位，所得的结果为9 >> 2＝2。

正数的位“右移”运算可以简单表示为：$a/2^x$ 取整，x代表移位数。比如，25 >> 1＝25/2＝12，25 >> 3＝$25/2^3$＝3。大家也可以自己去验证一下。

2）负数位“右移”运算规则

负数的位“右移”运算规则比较复杂，与变量的类型有很大关系。

（1）整型负数的位“右移”运算规则。

先对负数求补码，再对补码右移，移出“位”丢弃，在左侧高数位补1，移出几位补几个1，然后对右移后的数再求补码就是所求结果。

例 3-28　采用整型计算－9 >> 1 的值。

将例 3-27 程序中的变量 a 改为 a＝－9，其他程序均不变，其输出结果为：

```
  a = -9   b = -5
Press any key to continue
```

可见，－9 右移 1 位后的运算结果是－5。

用二进制字节简化验证为(－9)>> 1＝－5。如果对－9 右移 2 位，所得结果为(－9)>> 2＝－3。

根据整型变量负数的位“右移”运算结果，可以总结得到下面的简便算法。

如果 a 是一个负整数，当 a 右移 x 位时，其右移结果分两种情况：

若能被整除，其结果就是：$a/2^x$。比如，(－8)>> 1＝(－8)/2＝－4，(－8)>> 2＝(－8)/2^2＝－2。

若不能被整除，其结果就是：$a/2^x-1$。比如，(－9)>> 1＝(－9)/2－1＝－5，(－9)>> 2＝(－9)/2^2－1＝－3。

（2）无符号整型负数的位“右移”运算规则。

先对负数逐位取反码，再右移，移出“位”丢弃，在左侧高数位补 0，移出几位就补几个 0，右移后对应的十进制数就是所求的结果。

例 3-29　采用无符号整型计算－9 >> 1 的值。

将例 3-27 程序中的变量采用无符号整型，其他程序均不变，其输出结果为：

```
  a = -9   b = 2147483643
Press any key to continue
```

可见，无符号整数－9 右移 1 位后的运算结果是 2147483643，这个数字实际上是－9 取反码的数 4294967286/2 所得。

下面用二进制字节形式进行验证。

－9 的二进制数为：

```
1  0000  0000  0000  0000  0000  0000  0000  1001
```

－9 取反码为：

```
1  1111  1111  1111  1111  1111  1111  1111  0110
```

其对应的十进制数为 4294967286。

右移 1 位为：

```
1  0111  1111  1111  1111  1111  1111  1111  1011
```

其对应的十进制数为 2147483643，也就是 4294967286/2＝2147483643。这也就是无符号整数对应负数位“右移”的运算结果。按照同样的方法，可计算－9 >> 2。

取反右移 2 位为：

```
1  0011  1111  1111  1111  1111  1111  1111  1101
```

其对应的十进制为1073741821，也就是4294967286/2^2＝1073741821。

当定义变量为无符号整型时，其他负整数的位“右移”运算方法与上述相同，大家可以自己去验证。

(3) 字符型负数的位“右移”运算规则。

先对负数求补码，再对补码右移，移出“位”丢弃，在左侧高数位补1，移出几位就补几个1，然后对右移后的数再求补码就是所求结果。

例3-30 采用字符型计算(－9)>>1的值。

将例3-27程序中的变量采用字符型，其他程序均不变，－9右移1位的输出结果为：

```
  a = -9    b = -5
Press any key to continue
```

可见，－9右移1位后的运算结果是－5。

用二进制字节简化验证(－9)>>1＝－5。

根据字符型变量负数的位“右移”运算结果，也可以总结得到下面的简便算法。

假设a是一个负整数，当a右移x位时，其右移结果分两种情况：

① 若能被整除，其结果就是：a/2^x。比如，(－8)>>1＝(－8)/2＝－4，(－8)>>2＝(－8)/2^2＝－2。

② 若不能被整除，其结果就是：a/2^x－1。比如，(－9)>>1＝(－9)/2－1＝－5，(－9)>>2＝(－9)/2^2－1＝－3。

可见，字符型负数的位“右移”运算方法与整型负数的位“右移”运算方法相同，负数的位“右移”极限是－1。

(4) 无符号字符型负数的位“右移”运算规则。

无符号字符型的负数位“右移”时，先对负数求补码，然后对补码之后的数进行位“右移”；右移时移出“位”丢弃，在左侧的高数位补0；右侧移出几位就在左侧高数位补几个0；移位后所得的十进制数就是运算的结果。

例3-31 采用无符号字符型计算－9>>1的值。

将例3-27程序中的变量采用无符号字符型，其他程序均不变，其输出结果为：

```
  a = 247    b = 123
Press any key to continue
```

其中，a＝247是－9的补码，变量b是对变量a的补码位“右移”后的十进制数123。

用二进制字节简化验证b＝a>>1＝123。如果在无符号字符型变量的情况下，将a＝－9位“右移”2位，即a>>2，结果为b＝61，也就是247/2^2＝61。

3.4.2 位逻辑判断条件的应用

1. 位逻辑判断条件的构成方法

位逻辑运算共有6种，每一种都可以构成判断条件，也可以组合构成判断条件。

请看下面的程序：

```
1  #include <stdio.h>
2  void main()
3  {
```

```
4       unsigned char a,b,c;
5       a = 5;
6       printf("\n\n  请输入 b 的值(1～3):");
7       scanf(" %d",&b);
8       c = a&b;
9       if(c == 1)
10           printf("  a = %d\tb = %d\tc = %d\n",a,b,c);
11      else
12           printf("  条件不具备!\n");
13 }
```

程序的第 4 行定义了 3 个无符号整型变量 a、b、c；

第 5 行给变量 a 赋值为 5；

第 6 行和第 7 行提示给变量 b 从键盘赋值，赋值范围为 1～3；

第 8 行把 a&b 的值赋给变量 c；

第 9 行对变量 c 进行 c＝＝1 的逻辑判断；

如果条件成立，那么第 10 行就输出 3 个变量的值；

如果不成立，那么第 12 行就输出提示信息“条件不具备!”。

当程序运行后给变量 b 输入 1，条件成立，输出结果为：

```
  请输入 b 的值(1～3): 1
  a = 5   b = 1   c = 1
Press any key to continue
```

当程序运行后给变量 b 输入 2，条件不成立，输出结果为：

```
  请输入 b 的值(1～3): 2
  条件不具备!
Press any key to continue
```

当程序运行后给变量 b 输入 3，条件成立，输出结果为：

```
  请输入 b 的值(1～3): 3
  a = 5   b = 3   c = 1
Press any key to continue
```

实际上，采用位逻辑运算还可以构成其他的判断条件，大家可以自己去练习。

2. 位逻辑运算综合应用举例

请看下面的程序：

```
1  #include <stdio.h>
2  void main()
3  {
4       unsigned char a = 1,b = 2,c = 3,d,e,f;
5       d = ～a&b|c;
6       e = a|b<<1;
7       f = ～a<<1;
8       printf("\n\n  a = %d\tb = %d\tc = %d\td = %d\te = %d\tf = %d\n",a,b,c,d,e,f);
9  }
```

程序的第 4 行定义了 a、b、c、d、e、f 五个无符号字符型变量，并给 a、b、c 三个变量分别赋了初值 1、2、3；

第 5 行将位逻辑混合运算～a&b|c 的结果赋给变量 d；

第 6 行将位逻辑混合运算 a|b<<1 的结果赋给变量 e；

第 7 行将位逻辑混合运算～a<<1 的结果赋给变量 f；

第 8 行输出所有变量的值。

输出结果为：

```
a=1   b=2   c=3   d=3   e=5   f=252
```

从程序的运行结果 d=3、e=5、f=252 可以验证，第 5 行的逻辑运算顺序位为：非→与→或；第 6 行的运算顺序为：位左移→位或；第 7 行的运算顺序为：位非→位左移。大家也可以自己去验证一下。

归纳总结：正数的位运算比较简单，尤其是无符号字符型的位运算更简单，它只有一字节，数字范围最小。负数的位运算比较复杂，变量的定义类型不同，位运算的规则就不同，尤其是位“右移”的运算规则差异很大。所以，位运算通常把变量的类型多定义为无符号字符型。

3.5 选择结构的程序设计应用举例

例 3-32 任意给定 3 条边，计算三角形的面积。

编程思路：首先，要判断所给定的 3 条边能否构成一个三角形，如果能，计算才有意义。分析的依据就是任意两边之和要大于第三边。其次，要知道根据 3 条边计算三角形的面积公式有什么特殊之处，以便在设计程序时考虑周到一些。

根据数学概念，已知 3 条边计算三角形的面积公式为：$s=\sqrt{t(t-a)(t-b)(t-c)}$，其中 $t=(a+b+c)/2$。很明显，这个面积计算要用到开方的运算，这是数学的专有运算。在 C 语言中是不能直接进行开方运算的，需要调用数学的开方运算函数 sqrt()来完成相关的开方计算任务，还需要在程序前面加入数学运算的函数平台 #include <math.h>。

具体的程序如下：

```
#include <stdio.h>
#include <math.h>
void main()
{
    float a, b, c, t, s;
    printf("a=? b=? c=?\n");
    scanf("%f%f%f",&a, &b, &c);
    if(a+b>c&&a+c>b&&b+c>a)
    {
        t=(a+b+c)/2;
        s=sqrt(t*(t-a)*(t-b)*(t-c));
        printf("a=%1.2f\tb=%1.2f\tc=%1.2f\n",a, b, c);
        printf("s=%5.2f\n\n",s);
    }
    else
        printf("三条边不能构成三角形!退出.\n\n");
}
```

程序说明：在判断条件中，对 3 条边的判断必须用逻辑与“&&”，不能用逻辑或“||”。if(…)语句后面的花括号{ }所包含的 4 个语句是一个复合语句，要一同执行完毕。

程序运行如下：

```
a=? b=? c=? 3  3  2↙
a=3.00   b=3.00   c=2.00
s=2.83
```

说明输入正确，程序设计也正确。如果输入“1 2 3”，那么程序会发出错误提示信息。

3.6　选择结构的嵌套

单一选择结构是一个条件两项选择，它不能解决多个条件多项选择的问题。选择结构的嵌套可以解决此类问题，这就是多个条件多项选择的结构形式。

例 3-33　从键盘上任意输入一个字符，判断它是什么字符。

算法分析：首先输入的字符可能是字母、数字、空格或者是别的字符等。而每一种字符的判断条件是不同的，选项结果也是不同的。可见，此例用单一的选择结构无法实现，必须用选择结构的嵌套来实现。图 3-2 所示就是选择结构的嵌套流程图。

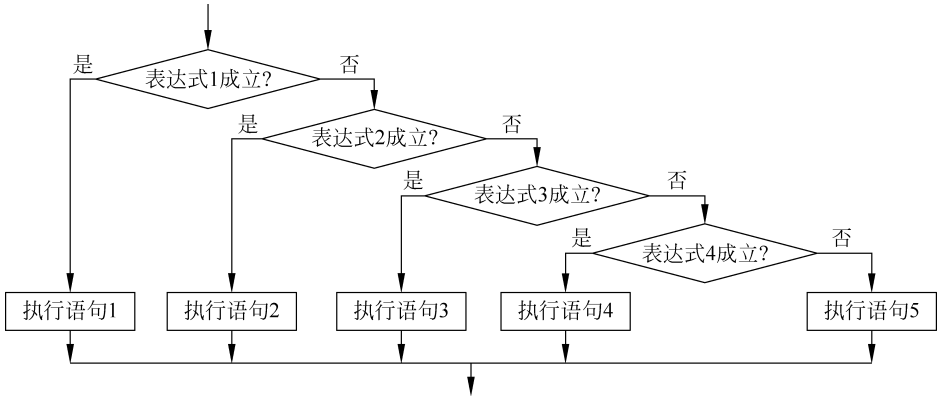


图 3-2　选择结构的嵌套流程图

如图 3-2 所示的选择结构对应的判断语句格式为：

```
if (表达式 1)
    语句 1;
else if (表达式 2)
        语句 2;
     else if (表达式 3)
             语句 3;
          else if (表达式 4)
                  语句 4;
              else
                  语句 5;
```

从流程图和选择语句中可以看出，它有 4 个条件判断表达式，还有 5 个执行选项结果，这就是多个条件多项选择的结构形式。

在选择语句的嵌套中多次出现 if()和 else 语句，前后很容易混淆。所以，书写时同级的 if()与 else 语句要对齐布局，形成倒阶梯形状，这样也有利于逻辑分析和查找问题。

对例 3-33 怎样编程？对分段函数计算怎样编程？大家试做一下这两个练习。

例 3-34 下面是一个选择结构嵌套程序的例子，该程序的输出结果是什么？

```
#include <stdio.h>
void main()
{
    int a, b, c, d, x;
    a = b = c = 0;
    d = 20;
    if ( a )
        d = d - 10;
    else
        if ( d + 2 )
          if (!c)
              x = 15;
          else
              x = 25;
    printf("d = %d, x = %d\n", d, x);
}
```

程序的运行结果是什么？大家猜得出来吗？

在这个程序中有两点需要说明：

一是给变量等价赋值“a＝b＝c＝0;”，这是对的；如果是“int a＝b＝c＝d＝0;”，则是错的。因为赋值号“＝”左边是左值，左值必须是变量，而变量名中是不能包含“＝”的，所以是错的。

二是判断条件表达式用的是算术表达式 a 和 d＋2，即 if(a)以及 f(d＋2)，这种形式是可行的。用算术表达式作为判断条件时，只要表达式的值不为 0，条件就成立，否则，就不成立。

理解了上述两点，再判断上面程序的结果就容易多了。

该程序的运行结果为：d＝20，x＝15，你猜对了吗？

3.7 条件表达式和条件语句介绍

C 语言有一个独门绝技，这个独门绝技就是条件运算。它的运算符号就是“？：”。之所以说成独门绝技，是因为在其他的计算机语言中没有这种运算方式，只有 C 语言才有。

运算符“？：”叫作条件运算符，它是 C 语言中唯一一个三目运算符，需要 3 个数据。条件运算符怎么用？其第一个数据是一个条件或者是一个逻辑值，第二个数据是条件成立时的结果，第三个数据是条件不成立时的结果。第一个数要放在“?”之前；第二个数要放在“?”与“:”之间；第三个数要放在“:”之后。

例如，“a＞b？a：b”或者“X？方案一：方案二”都是条件语句的用法，这也叫作条件表达式。

如果把条件表达式的运算结果赋给一个变量，就可以构成条件表达式的赋值语句。

例如，“max＝a＞b？a：b;”，这样 max 就可以获得两个数据中的大者。

不论条件判断的结果如何，只要是指向同一个目标，都可以用条件语句来实现。

例 3-35 求两个整数中的大者。用条件表达式来实现的程序如下：

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

```
2  void main()
3  {
4      int a, b, max;
5      printf("a = ? b = ?");
6      scanf(" % d % d",&a, &b);
7      max = a > b ?a: b;
8      printf("a = % d\tb = % d\tmax = % d\n",a, b, max);
9  }
```

程序第 7 行用条件表达式实现判断功能，简化了程序设计。

3.8　switch()多分支语句介绍

虽然选择结构的嵌套能解决多个条件多项选择的问题，但是它不能解决一个条件多种情况选择的问题，而采用 switch()多分支语句解决此类问题正合适。

1. switch()多分支语句介绍及应用方法

switch()多分支语句的格式如下：

```
switch(表达式)
{
    case 常量表达式 1:语句组 1;
    case 常量表达式 2:语句组 2;
    case 常量表达式 3:语句组 3;
    …
    case 常量表达式 n: 语句组 n;
    [default: 语句组 n + 1;]
}
```

switch()多分支语句是由首部和 case 分支表组成的。它的首部括号中的“表达式”就是唯一的一个判断条件，但是它有多种情况与 case 分支表中的常量表达式相对应，而每个常量表达式后面对应着不同的分支选项。所以，它是一个多种情况多项选择的结构形式。

switch()语句就像一个多路开关，它的分支表中 case 可多可少，具体由所控制的对象来决定。如果没有特殊要求，那么分支表的前后顺序可以调换，原则上不影响语句的执行。

在 switch()多分支语句中需要注意以下几点：

(1) [default：语句组 n+1；]是一个默认选项，可有可无。

(2) case 中的常量表达式 1：后面的符号必须是冒号“：”。

(3) switch()括号中的表达式不能用浮点型数据或者表达式，比如，switch(a+3.5)或者 switch(2.3)等都是错误的。其他类型的表达式均可以，可以是整型、字符型表达式或者数据，也可以是关系表达式或者逻辑表达式。

(4) 分支表 case 中的常量表达式只能是字符常量或者整型常量，比如，0，1，2，3，…或者 A，B，C，…，a，b，c 等。不能用浮点数，更不能用变量。每个常量表达式的值不能重复，必须是唯一的。

switch()语句的执行过程是：当首部表达式的值与 case 分支表中常量表达式的值相等时，程序就去执行该分支表之后所列语句组的内容。分支表中常量表达式的值不同，执行的语句组内容也就不同。

例 3-36 下面 switch()语句的输出结果是什么？

```
#include <stdio.h>
void main()
{
    int a = 0, i = 1;
    switch( i )
    {
        case 0:
        case 1: a = 2; printf("a = %d\n",a);
        case 2:
        case 3: a = 3; printf("a = %d\n",a);
        default: a = 7; printf("a = %d\n",a);
    }
}
```

在这个例子中，因为 i=1，所以 switch(i)就是 switch(1)，程序就去执行分支表中“case 1：a=2; printf("a=%d\n",a);”的语句，输出的结果应该是 a=2，但是程序执行的结果有 3 行输出：

```
a = 2
a = 3
a = 7
```

这是因为每个分支语句之后没有中断跳转语句，所以后面的情况都被一起执行了。假如 i=0，后面所有的 case 情况也都会被执行，这样的 case 分支表选项是没有选择性的。

2. switch()语句中“break;”语句的功能

要想使 switch()语句的分支表选择更清晰，就要在每个 case 分支语句组之后加上一条“break;”语句。其流程图如图 3-3 所示。

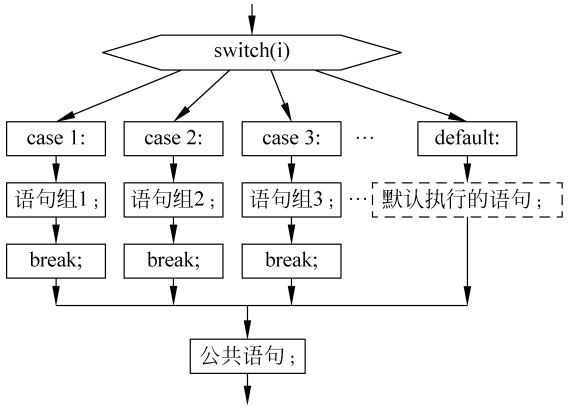


图 3-3 switch()多分支语句流程图

图 3-3 中的六边形表示 switch()多分支语句的流程图模块。

例 3-37 已知某公司员工的保底工资为 1000 元，某月的销售利润 profit(整数)与利润的提成关系如下(计量单位：元)：

profit≤1000　　没有提成；

1000＜profit≤2000　提成 10%；

2000＜profit≤5000　提成 15%；
5000＜profit≤10000　提成 20%；
10000＜profit　提成 25%。

根据利润求出员工的工资数额。

设计分析：利润数很大，若直接用于 case 分支表中的常量表达式会很麻烦，所以要对利润进行简化处理，使 case 中的常量表达式变得简单，这也是处理同类问题的好方法。

由于利润提成的变化点都是 1000 的整数倍（1000、2000、5000、……），可以将利润 profit 整除以 1000，那么：

profit≤1000　对应 0、1
1000＜profit≤2000　对应 1、2
2000＜profit≤5000　对应 2、3、4、5
5000＜profit≤10000　对应 5、6、7、8、9、10
profit＞10000　对应 10、11、12、……

从上面对应的数字来看，利润 1000 以下有一个 1，1000 以上也有一个 1；同样，利润在 5000＜profit≤10000 以下的有 10，利润在 profit＞10000 以上的也有 10。两个相邻级别之间出现了重叠问题，这不利于利润的分配。

解决重叠问题的方法很简单：采用最小增量法，就是将利润 profit 先减 1，然后再整除以 1000 就可以了。请看结果：

profit≤1000　对应 0
1000＜profit≤2000　对应 1
2000＜profit≤5000　对应 2、3、4
5000＜profit≤10000　对应 5、6、7、8、9
profit＞10000　对应 10、11、12、……

经过这样的分级后所设计的程序如下：

```
#include <stdio.h>
void main()
{
    long profit;                    //定义 profit 为长整型变量
    int grade;
    float salary = 1000;            //工资底薪为 1000 元
    printf("Input profit: ");
    scanf("%ld", &profit);          // %ld 中 l 表示长整型格式
    grade = (profit - 1) / 1000;    /* 最小增量法的应用,这是编程的关键 */
    switch(grade)
    {
        case 0:  break;             //profit≤1000 没有提成
        case 1:  salary += profit * 0.1;  break;
        case 2:
        case 3:
        case 4:  salary += profit * 0.15;  break;
        case 5:
        case 6:
        case 7:
        case 8:
```

```
            case  9:  salary += profit * 0.2;  break;
            default:  salary += profit * 0.25;
        }
        printf("salary = %.2f\n", salary);
    }
```

此例中同一个等级之间不使用“break;”语句，只在不同等级中使用，这样输出的结果符合题目要求。这样的多分支程序对确定利润提成很方便，大家可以按照自己心目中的薪资确定规则，感受一下该程序的乐趣。

3. 复合赋值运算符号的介绍

在例 3-37 的程序中，我们看到了一种全新的运算符号“+=”，这种运算符号叫作复合赋值运算符。C 语言中共有 5 种类型的复合赋值运算符，即+=、-=、*=、/=、%=，它们所对应的运算功能就是求累加、累减、累乘、累除和累求余。

例如，赋值表达式 a=a+2，就是用变量 a 原来的值加上 2 之后将所得到的新值再赋值给变量 a，即 a 现在的值已经被新的值刷新了。这种赋值表达式可以用复合赋值运算符写成 a+=2。同样，b=b-3 可以写成 b-=3；c =c *4 可以写成 c*=4；d =d%5 可以写成 d%=5 等。

4. 自增自减运算符号的介绍

当增量值≥2 时都可以写成复合赋值运算符的表达方式，如果增量值为 1，即 a=a+1 可以写成 a+=1；b=b-1 可以写成 b-=1。

不过，在 C 语言中把增量为 1 的累加和累减运算用一种更简单的运算符号来描述，这就是++和--。也就是可以把 a=a+1 直接写成 a++，把 b=b-1 直接写成 b--。

++或者--运算就叫作自增或者自减运算。

自增和自减运算各有两种运算形式，比如，a++和++a，b--和--b。自增、自减运算符号在变量后面的叫后置运算，在变量前面的叫前置运算。前置运算与后置运算的规则是不一样的。

前置运算的规则是：先增减后运算；后置运算的规则是：先运算后增减。

先增减后运算就是先给变量加 1 或者减 1，然后才用变量的值进行计算；先运算后增减就是先用变量原来的值进行计算，之后再给变量的值加 1 或者减 1。

虽说++i、--i 与 i++、i--都对应于 i=i+1 和 i=i-1 的形式，但是，运算之后的值要么加 1 要么减 1。

例 3-38 下面例子的结果是多少？

如果 i=3，j=++i，则 i=__ 、j=__ ；

如果 i=3，k=i++，则 i=__ 、k=__ ；

如果 i=3，x=i--，则 i=__ 、x=__ ；

如果 i=3，y=--i，则 i=__ 、y=__ 。

答案是：i=4、j=4；

i=4、k=3；

i=2、x=3；

i=2、y=2。

你的答案对吗？

5. switch()语句的嵌套应用

switch()语句还可以嵌套应用，方法与单一的 switch()语句相同，但是，要掌握好各自 case 语句的对应关系，还要确定好“break;”语句跳出后的准确位置，否则就会出错。

例 3-39　下面例子的输出结果是什么？

```
#include <stdio.h>
void main()
{
    int a = 2, b = 7, c = 5;
    switch(a > 0)
    {
        case 1: switch(b < 0)
        {
            case 0: printf("@"); break;
            case 1: printf("!"); break;
        }
        case 0: switch(c == 5)
        {
            case 0: printf("*"); break;
            case 1: printf("#"); break;
            default: printf("$"); break;
        }
    default: printf("&");
    }
}
```

程序的输出结果如下：

```
@#&,你答对了吗?
```

6. switch()语句的菜单功能

用 switch()多分支语句可以直接计算任意直角三角形的面积、矩形的面积、圆形的面积或者梯形的面积等，而不是一个单一的计算程序，这就要用到 switch()多分支语句菜单功能。怎样实现这个菜单功能？

我们需要借助 4 个语句来完成这个任务：一是 printf()输出语句，二是 scanf()输入语句，三是 switch()多分支语句，四是 goto 转向语句(该语句是一种无条件的循环语句，会在第 4 章中学习)。其中，由 printf()输出语句构成菜单选项；由 scanf()输入语句对菜单选项进行输入确认；由 switch()多分支语句对选项内容进行具体的计算操作和执行输出；由 goto 转向语句确定是否需要返回菜单。请看下面的例子。

例 3-40　设计一个实用菜单，要求如下：

```
    选择计算有关面积
======================
    1、求矩形的面积
    2、求直角三角形的面积
    3、求菱形的面积
```

```
    4、退出计算
=======================
      你的选择是:
```

按照上面的书写形式,先用 printf()输出语句设计一个显示菜单;根据菜单的不同选项编号,就可以计算不同图形的面积。这个选项编号要通过 scanf()输入语句输入;根据输入的编号,由 switch()多分支语句选择执行计算;执行完之后,如果还要进行其他图形的面积计算,则不用退出程序,可以再返回菜单进行新的选项计算,这个功能就由 goto 转向语句来完成,以此类推。直到从菜单上选择退出后,程序运行才真正结束。

下面就是所设计的程序:

```
#include <stdio.h>
#include <stdlib.h>                    //这是清屏函数的头函数库声明
void main()
{
    int a, b, s, m;
  w: m = 0;                            //将选择序号先清 0,避免出现重复的选择
    system("cls");                     //使用清屏命令,使屏幕的显示内容更整洁
    printf("选择计算菜单\n");
    printf("==============\n");
    printf("1、计算矩形面积\n");
    printf("2、计算直角三角形面积\n");
    printf("3、计算菱形面积\n");
    printf("4、退出计算\n");
    printf("================\n");
    printf("   你的选择是:");
    scanf("%d",&m);
    switch(m)
    {
        case 1: system("cls");    // 清屏语句
                printf("\n\n 请输入矩形的长和宽\n");
                scanf("%d%d",&a,&b);
                s = a * b;
                printf("\n 矩形的长 = %d\t 宽 = %d\t 面积 = %d\n",a,b,s);
                break;
        case 2: system("cls");    // 清屏语句
                printf("\n\n 请输入直角三角形的底和高\n");
                scanf("%d%d",&a,&b);
                s = a * b/2;
                printf("\n 直角三角形的底 = %d\t 高 = %d\t 面积 = %d\n",a,b,s);
                break;
        case 3: system("cls");    // 清屏语句
                printf("\n\n 请输入菱形的边长和高\n");
                scanf("%d%d",&a,&b);
                s = a * b;
                printf("\n 菱形的边长 = %d\t 高 = %d\t 面积 = %d\n",a,b,s);
                break;
        case 4: break;
    }
    if (m!= 4)
        { getchar();  getchar();  goto w; }
    printf("\n\n     已退出运算!\n\n\n");
}
```

程序总共有 42 行语句命令，其中：

第 1 行和第 2 行是两个文件包含声明；

第 5 行是变量的定义；

第 6 行是菜单循环的入口，“w：”是循环入口的标号；

第 7 行是清屏语句命令；

第 8～16 行是菜单与选项输入语句；

第 17～38 行是 switch()多分支选择语句；

第 39 行和第 40 行是判断以及菜单的循环语句；

第 41 行是退出提示语句。

其中，“case 1：”对应的是矩形面积的计算程序，包括变量的输入、面积计算以及结果输出等，最后是“break;”跳转语句。

“case 2：”对应的是直角三角形面积的计算程序，也包括变量的输入、面积计算以及结果输出等，最后也是“break;”跳转语句。

“case 3：”对应的是菱形面积的计算程序，也包括变量的输入、面积计算以及结果输出等，最后也是“break;”跳转语句。

“case 4：”是退出菜单的功能，结束程序的运行。

程序中 case 语句的标号必须与菜单中的选项编号类型一致，要么都用整型数据，要么都用字符型数据。“goto w;”语句中的 w 是转向语句的标号，w 后面必须用分号(;)；第 6 行的“w：m=0;”语句前面的 w 就是 goto 要跳转的语句位置标号，w 的位置标号后面必须用冒号“：”，前后的标号字符必须一致。

程序中的“getchar();”语句是一个字符输入语句，它可以从键盘上获得任意一个字符。在该程序中连续用了两个“getchar();”语句，其中第一个“getchar();”语句是用来吸收输入选项后的回车键，第二个“getchar();”语句是起“暂停”的作用，以便能够对程序的计算结果看得清楚一些，等看清楚之后，没有问题，就在键盘上按一个任意键或者单击一下鼠标，程序就会返回到选择菜单。

“getchar();”语句是程序应用中的又一个技巧，同样，“system("cls");”语句的应用也是一个技巧，它叫作清屏语句，可以使屏幕显示的内容更干净。

这个程序主要介绍了把菜单与不同的面积计算程序相结合的方法，如果要设计其他类型的菜单，比如，卡拉 OK 菜单、材料存放菜单、工资管理菜单等都可以照此设计。

本章小结

本章重点介绍了 C 语言的 3 种选择结构形式，即一个条件两项选择结构形式、多个条件多项选择的嵌套形式和一个条件多种情况选择的 switch()分支语句形式；介绍了条件表达式的多种构成方式和条件语句的用法，包括 switch()语句中“break;”语句的功能、复合赋值运算、自增自减运算、switch()语句的嵌套以及由 switch()语句构成菜单功能的方法及其相关的应用等。

本章首先简要介绍了算法的概念以及流程图的模块及画法。算法就是解决问题的步骤和方法，而流程图主要由“五个模块一条线”构成。怎样画流程图是由程序运行的主体内容

决定的，画流程图时要根据功能要求选择相应的图形模块，原则上一个流程图模块对应一个编程语句。

选择结构主要是由 if()和 else 语句来实现的。流程图不同，语句命令的写法也不一样。选择语句命令都是由 if()开头的，在 if()语句之后一定要接条件成立的语句；在不成立的分支上只要有语句输出，就要用 else 的语句引导形式，之后再接条件不成立的语句，否则就不需要 else 语句了。

由关系表达式、逻辑表达式、算数表达式以及位逻辑表达式等多种形式可以构成选择判断的条件。在关系表达式中要注意不等于、等于符号的写法，等于“==”、不等于“!=”的优先级别比其他的关系运算级别低，要后运算。在逻辑表达式运算中，逻辑或“||”的运算级别最低，一定要后算。在位逻辑表达式运算中，位或“|”运算的级别最低，要最后运算，位非“~”运算级别最高要先算，位左移“<<”或右移“>>”运算级别次于位非“~”，但高于位与“&”和位或“|”运算。

作为 if()语句的判断条件，关系表达式、逻辑表达式和位逻辑表达式的值只有成立和不成立两种形式，成立的用 1 表示，不成立的用 0 表示。

位逻辑运算是数字控制的基础。正数的位运算比较简单，尤其是无符号字符型的位运算更简单，它只有一字节，数字范围最小。负数的位逻辑运算比较复杂，变量的定义类型不同，位逻辑运算的规则就不同，尤其是位“右移”的运算规则差异很大。所以，位逻辑运算通常把变量的类型多定义为无符号字符型。

在 if()语句的嵌套格式中，书写时 else 语句要与所对应 if()语句对齐布局，前后的 if()语句要错位布局，形成倒阶梯形状。

条件表达式是由条件运算符“? :”构成的式子，它是一种三目运算。利用条件表达式可以简化选择结构的程序。

switch()多分支语句是由其首部和分支表构成的，switch()语句就像一个多路开关，它的分支表 case 项可多可少，具体由所控制的对象决定。若没有特殊要求，则分支表的前后顺序可以调换，原则上不影响语句的执行。switch()括号里面的表达式不能用浮点型数据形式，分支表 case 中的常量表达式只能是字符常量或者是整型常量，分支表的标号之后只能用冒号“:”。

利用“break;”语句可以实现 switch()语句的准确分支，而不会出现前后语句功能的混淆。

对于 switch()语句的嵌套程序，一定要明白“break;”语句退出后的层次位置，“break;”语句只能退出一层，不能退出多层。

复合赋值运算主要用于增量值大于或等于 2 以上的情况，自增自减运算只能用于增量值为 1 的情况。前置运算的规则是：先增减后运算；后置运算的规则是：先运算后增减。

设计菜单功能需要借助 4 个语句来完成：一是 printf()输出语句，二是 scanf()输入语句，三是 switch()多分支语句，四是 goto 转向语句。其中，由 printf()输出语句构成菜单的选项内容；由 scanf()输入语句对菜单选项进行输入确认；由 switch()多分支语句对选项内容进行具体的分项计算操作和执行输出；由 goto 转向语句确定是否需要返回菜单。