第5章

操作符和表达式

C提供了所有你希望编程语言应该拥有的操作符[1]，它甚至提供了一些你意想不到的操作符。事实上，C被许多人所诟病的一个缺点就是它品种繁多的操作符。C的这个特点使它很难精通。另一方面，C的许多操作符具有其他语言的操作符无可抗衡的价值，这也是C适用于开发范围极广的应用程序的原因之一。

在介绍完操作符之后，我将讨论表达式求值的规则，包括操作符优先级和算术转换。

5.1 操作符

为了便于解释，我将按照操作符的功能或它们的使用方式对它们进行分类。为了便于参考，按照优点级对它们进行分组会更方便一些。本章后面的表5.1就是按照这种方式组织的。

5.1.1 算术操作符

C提供了所有常用的算术操作符：

```
+    -    *    /    %
```

除了%操作符，其余几个操作符都是既适用于浮点类型又适用于整数类型。当/操作符的两个操作数都是整数时，它执行整除运算，在其他情况下则执行浮点数除法[2]。%为取模操作符，它接受两个整型操作数，把左操作数除以右操作数，但它返回的值是余数而不是商。

5.1.2 移位操作符

汇编语言程序员对于移位操作已经是非常熟悉了。对于那些适应能力强的读者，这里作一简单介绍。移位操作只是简单地把一个值的位向左或向右移动。在左移位中，值最左边的几位被丢弃，右边多出来的几个空位则由0补齐。图5.1是一个左移位的例子，它在一个8位的值上进行左移3位的操作，以二进制形式显示。这个值所有的位均向左移3个位置，移出左边界的那几个位丢失，

---

1 译注：operator有时也译为运算符，但本书为统一起见，一律译为操作符。

2 如果整除运算的任一操作数为负值，运算的结果是由编译器定义的。详情请参见第16章介绍的div函数。

右边空出来的几个位则用 0 补齐。

右移位操作存在一个左移位操作不曾面临的问题：从左边移入新位时，可以选择两种方案。一种是逻辑移位，左边移入的位用 0 填充；另一种是算术移位，左边移入的位由原先该值的符号位决定，符号位为 1 则移入的位均为 1，符号位为 0 则移入的位均为 0，这样能够保持原数的正负形式不变。如果值 10010110 右移两位，逻辑移位的结果是 00100101，但算术移位的结果是 11100101。算术左移和逻辑左移是相同的，它们只在右移时不同，而且只有当操作数是负值时才不一样。

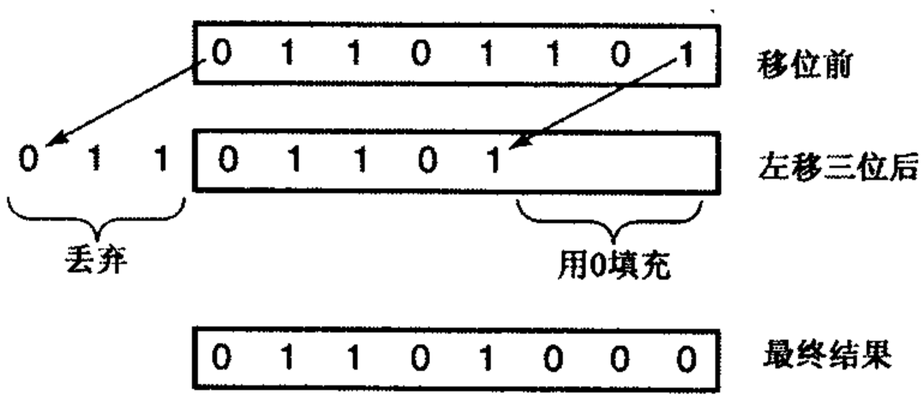

图 5.1　左移 3 位

左移位操作符为<<，右移位操作符为>>。左操作数的值将移动由右操作数指定的位数。两个操作数都必须是整型类型。

警告：

标准说明无符号值执行的所有移位操作都是逻辑移位，但对于有符号值，到底是采用逻辑移位还是算术移位取决于编译器。你可以编写一个简单的测试程序，看看你的编译器使用哪种移位方式。但你的测试并不能保证其他的编译器也会使用同样的方式。因此，一个程序如果使用了有符号数的右移位操作，它就是不可移植的。

警告：

注意类似这种形式的移位：

```
a << -5
```

左移-5 位表示什么呢？是表示右移 5 位吗？还是根本不移位？在某台机器上，这个表达式实际执行左移 27 位的操作——你怎么也想不出来吧！如果移位的位数比操作数的位数还要多，会发生什么情况呢？

标准说明这类移位的行为是未定义的，所以它是由编译器决定的。然而，很少有编译器设计者会清楚地说明如果发生这种情况将会怎样，所以它的结果很可能没有什么意义。因此，你应该避免使用这种类型的移位，因为它们的效果是不可预测的，使用这类移位的程序是不可移植的。

程序 5.1 的函数使用右移位操作来计数一个值中值为 1 的位的个数。它接受一个无符号参数（这是为了避免右移位的歧义），并使用%操作符判断最右边的一位最否非零。在学习完&、<<=和+=操作符之后，我们将进一步完善这个函数。

```
/*
** 这个函数返回参数值中值为 1 的位的个数。
*/
int
count_one_bits( unsigned value )
{
```

```
    int ones;

    /*
    ** 当这个值还有一些值为 1 的位时...
    */
    for( ones = 0; value != 0; value = value >> 1 )
         /*
         ** 如果最低位的值为 1，计数增 1。
         */
         if( value % 2 != 0 )
              ones = ones + 1;

    return ones;
}
```

程序 5.1 计数一个值中值为 1 的位的个数：初级版本　　count_1a.c

5.1.3　位操作符

位操作符对它们的操作数的各个位执行 AND、OR 和 XOR（异或）等逻辑操作。同样，汇编语言程序员对于这类操作已是非常熟悉了，但为了照顾其他人，这里还是作一简单介绍。当两个位进行 AND 操作时，如果两个位都是 1，结果为 1，否则结果为 0。当两个位进行 OR 操作时，如果两个位都是 0，结果为 0，否则结果为 1。最后，当两个位进行 XOR 操作时，如果两个位不同，结果为 1，如果两个位相同，结果为 0。这些操作以图表的形式总结如下。

A \ B	0	1
0	0	0
1	0	1

A AND B

A \ B	0	1
0	0	1
1	1	1

A OR B

A \ B	0	1
0	0	1
1	1	0

A XOR B

位操作符有：

```
&      |      ^
```

它们分别执行 AND、OR 和 XOR 操作。它们要求操作数为整数类型，它们对操作数对应的位进行指定的操作，每次对左右操作数的各一位进行操作。举例说明，假定变量 a 的二进制值为 00101110，变量 b 的二进制值为 01011011。a & b 的结果是 00001010，a | b 的结果是 01111111，a ^ b 的结果是 011110101。

位的操纵

下面的表达式显示了你可以怎样使用移位操作符和位操作符来操纵一个整型值中的单个位。表达式假定变量 bit_number 为一整型值，它的范围是从 0 至整型值的位数减 1，并且整型值的位从右向左计数。第 1 个例子把指定的位设置为 1。

```
value = value | 1 << bit_number;
```

下一个例子把指定的位清 0[1]。

[1] 这里简单描述一下单目操作符~，它用于对其操作数进行求补运算，即 1 变为 0，0 变为 1。

```
value = value & ~ ( 1 << bit_number );
```

这些表达式常常写成|=和&=操作符的形式，它们将在下一节介绍。最后，下面这个表达式对指定的位进行测试，如果该位已被设置为 1，则表达式的结果为非零值。

```
value & 1 << bit_number
```

5.1.4 赋值

最后，我们讨论赋值操作符，它用一个等号表示。赋值是表达式的一种，而不是某种类型的语句。所以，只要是允许出现表达式的地方，都允许进行赋值。下面的语句

```
x = y + 3;
```

包含两个操作符，+和=。首先进行加法运算，所以=的操作数是变量 x 和表达式 y+3 的值。赋值操作符把右操作数的值存储于左操作数指定的位置。但赋值也是个表达式，表达式就具有一个值。赋值表达式的值就是左操作数的新值，它可以作为其他赋值操作符的操作数，如下面的语句所示：

```
a = x = y + 3;
```

赋值操作符的结合性（求值的顺序）是从右到左，所以这个表达式相当于：

```
a = ( x = y + 3 );
```

它的意思和下面的语句组合完全相同：

```
x = y + 3;
a = x;
```

下面是一个稍微复杂一些的例子。

```
r = s + ( t = u - v ) / 3;
```

这条语句把表达式 u-v 的值赋值给 t，然后把 t 的值除以 3，再把除法的结果和 s 相加，其结果再赋值给 r。尽管这种方法也是合法的，但改写成下面这种形式也具有同样的效果。

```
t = u - v;
r = s + t / 3;
```

事实上，后面这种写法更好一些，因为它们更易于阅读和调试。人们在编写内嵌赋值操作的表达式时很容易走极端，写出难于阅读的表达式。因此，在你使用这个“特性”之前，确信这种写法能带来一些实实在在的好处。

警告：

在下面的语句中，认为 a 和 x 被赋予相同的值的说法是不正确的：

```
a = x = y + 3;
```

如果 x 是一个字符型变量，那么 y+3 的值就会被截去一段，以便容纳于字符类型的变量中。那么 a 所赋的值就是这个被截短后的值。在下面这个常见的错误中，这种截短正是问题的根源所在：

```
char ch;
...
while( ( ch = getchar() ) != EOF ) ...
```

EOF 需要的位数比字符型值所能提供的位数要多，这也是 getchar 返回一个整型值而不是字符值的原因。然而，把 getchar 的返回值首先存储于 ch 中将导致它被截短。然后这个被截短的值被提升为整型并与 EOF 进行比较。当这段存在错误的代码在使用有符号字符集的机器上运行时，如果读取了一个值为\377 的字节时，循环将会终止，因为这个值截短再提升之后与 EOF 相等。当这段代码

在使用无符号字符集的机器上运行时，这个循环将永远不会终止！

复合赋值符

到目前为止所介绍的操作符都还有一种复合赋值的形式：

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

我们只讨论+=操作符，因为其余操作符与它非常相似，只是各自使用的操作符不同而已。+=操作符的用法如下：

```
a += expression
```

它读作“把 expression 加到 a”，它的功能相当于下面的表达式：

```
a = a + ( expression )
```

唯一的不同之处是+=操作符的左操作数（此例为 a）只求值一次。注意括号：它们确保表达式在执行加法运算前已被完整求值，即使它内部包含有优先级低于加法运算的操作符。

存在两种增加一个变量值的方法有何意义呢？K&R C 设计者认为复合赋值符可以让程序员把代码写得更清楚一些。另外，编译器可以产生更为紧凑的代码。现在，a=a+5 和 a+=5 之间的差别不再那么显著，而且现代的编译器为这两种表达式产生优化代码并无多大问题。但请考虑下面两条语句，如果函数 f 没有副作用，它们是等同的。

```
a[ 2 * (y - 6*f(x)) ] = a[ 2 * (y - 6*f(x) ) ] + 1;
a[ 2 * (y - 6*f(x)) ] += 1;
```

在第 1 种形式中，用于选择增值位置的表达式必须书写两次，一次在赋值号的左边，另一次在赋值号的右边。由于编译器无从知道函数 f 是否具有副作用，所以它必须两次计算下标表达式的值。第 2 种形式效率更高，因为下标只计算一次。

提示：

+=操作符更重要的优点是它使源代码更容易阅读和书写。读者如果想判断上例第 1 条语句的功能，他必须仔细检查这两个下标表达式，证实它们的确相同，然后还必须检查函数 f 是否具有副作用。但第 2 条语句则不存在这样的问题。而且它在书写方面也比第 1 条语句更方便，出现打字错误的可能性也小得多。基于这些理由，你应该尽量使用复合赋值符。

我们现在可以使用复合赋值符来改写程序 5.1，结果见程序 5.2。复合赋值符同时能简化用于设置和清除变量值中单个位的表达式：

```
value |= 1 << bit_number;
value &= ~ ( 1 << bit_number );

/*
** 这个函数返回参数值中值为 1 的位的个数。
*/
int
count_one_bits( unsigned value )
{
        int ones;

        /*
```

```
** 当这个值中还存在一些值为 1 的位时.. */
for( ones = 0; value != 0; value >>= 1 )
    /*
    ** 如果最低位为 1，增加计数器的值。
    */
    if( ( value & 1 ) != 0 )
            ones += 1;

return ones;
}
```

程序 5.2 计数一个值中值为 1 的位的个数：最终版本 count_1b.c

5.1.5 单目操作符

C 具有一些单目操作符，也就是只接受一个操作数的操作符。它们是

```
!       ++        -       &       sizeof
~       --        +       *       (类型)
```

让我们逐个来介绍这些操作符。

!操作符对它的操作数执行逻辑反操作；如果操作数为真，其结果为假，如果操作数为假，其结果为真。和关系操作符一样，这个操作符实际上产生一个整型结果，0 或 1。

~操作符对整型类型的操作数进行求补操作，操作数中所有原先为 1 的位变为 0，所有原先为 0 的位变为 1。

-操作符产生操作数的负值。

+操作符产生操作数的值；换句话说，它什么也不干。之所以提供这个操作符，是为了与-操作符组成对称的一对。

&操作符产生它的操作数的地址。例如，下面的语句声明了一个整型变量和一个指向整型变量的指针。接着，&操作符取变量 a 的地址，并把它赋值给指针变量。

```
int  a, *b;
...
b = &a;
```

这个例子说明了你如何把一个现有变量的地址赋值给一个指针变量。

*操作符是间接访问操作符，它与指针一起使用，用于访问指针所指向的值。在前面例子中的赋值操作完成之后，表达式 b 的值是变量 a 的地址，但表达式*b 的值则是变量 a 的值。

sizeof 操作符判断它的操作数的类型长度，以字节为单位表示。操作数既可以是个表达式（常常是单个变量），也可以是两边加上括号的类型名。这里有两个例子：

```
sizeof ( int )          sizeof x
```

第 1 个表达式返回整型变量的字节数，其结果自然取决于你所使用的环境。第 2 个表达式返回变量 x 所占据的字节数。注意，从定义上说，字符变量的长度为 1 个字节。当 sizeof 的操作数是个数组名时，它返回该数组的长度，以字节为单位。在表达式的操作数两边加上括号也是合法的，如下所示：

```
sizeof( x )
```

这是因为括号在表达式中总是合法的。判断表达式的长度并不需要对表达式进行求值，所以

sizeof(a = b + 1)并没有向 a 赋任何值。

（类型）操作符被称为**强制类型转换**(cast)，它用于显式地把表达式的值转换为另外的类型。例如，为了获得整型变量 a 对应的浮点数值，你可以这样写

```
(float)a
```

强制类型转换这个名字很容易记忆，它具有很高的优先级，所以把强制类型转换放在一个表达式前面只会改变表达式的第 1 个项目的类型。如果要对整个表达式的结果进行强制类型转换，你必须把整个表达式用括号括起来。

最后我们讨论增值操作符++和减值操作符--。如果说有哪种操作符能够捕捉到 C 编程的“感觉”，它必然是这两个操作符之一。这两个操作符都有两个变型，分别为前缀形式和后缀形式。两个操作符的任一变种都需要一个变量而不是表达式作为它的操作数。实际上，这个限制并非那么严格。这个操作符实际只要求操作数必须是一个“左值”，但目前我们还没有讨论这个话题。这个限制要求++或--操作符只能作用于可以位于赋值符号左边的表达式。

前缀形式的++操作符出现在操作数的前面。操作数的值被增加，而表达式的值就是操作数增加后的值。后缀形式的++操作符出现在操作数的后面。操作数的值仍被增加，但表达式的值是操作数增加前的值。如果你考虑一下操作符的位置，这个规则很容易记住——在操作数之前的操作符在变量值被使用之前增加它的值；在操作数之后的操作符在变量值被使用之后才增加它的值。--操作符的工作原理与此相同，只是它所执行的是减值操作而不是增值操作。

这里有一些例子。

```
int a, b, c, d;
...
a = b = 10;                    a 和 b 得到值 10
c = ++a;                       a 增加至 11，c 得到的值为 11
d = b++;                       b 增加至 11，但 d 得到的值仍为 10
```

上面的注释描述了这些操作符的结果，但并不说明这些结果是如何获得的。抽象地说，前缀和后缀形式的增值操作符都复制一份变量值的拷贝。用于周围表达式的值正是这份拷贝（在上面的例子中，“周围表示式”是指赋值操作）。前缀操作符在进行复制之前增加变量的值，后缀操作符在进行复制之后才增加变量的值。这些操作符的结果不是被它们所修改的变量，而是变量值的拷贝，认识这一点非常重要。它之所以重要是因为它解释了你为什么不能像下面这样使用这些操作符：

```
++a = 10;
```

++a 的结果是 a 值的拷贝，并不是变量本身，你无法向一个值进行赋值。

5.1.6　关系操作符

这类操作符用于测试操作数之间的各种关系。C 提供了所有常见的关系操作符。不过，这组操作符里面存在一个陷阱。这些操作符是：

```
>     >=          <     <=          !=     ==
```

前 4 个操作符的功能一看便知。!=操作符用于测试“不相等”，而==操作符用于测试“相等”。

尽管关系操作符所实现的功能和你预想的一样，但它们实现功能的方式则和你预想的稍有不同。这些操作符产生的结果都是一个整型值，而不是布尔值。如果两端的操作数符合操作符指定的关系，表达式的结果是 1，如果不符合，表达式的结果是 0。关系操作符的结果是整型值，所以它可以赋值

给整型变量，但通常它们用于 if 或 while 语句中，作为测值表达式。请记住这些语句的工作方式：表达式的结果如果是 0，它被认为是假；表达式的结果如果是任何非零值，它被认为是真。所有关系操作符的工作原理相同，如果操作符两端的操作数不符合它指定的关系，表达式的结果为 0。因此，单纯从功能上说，我们并不需要额外的布尔型数据类型。

C 用整数来表示布尔型值，这直接产生了一些简写方法，它们在表达式测值中极为常用。

```
if( expression != 0 ) ...
if( expression ) ...

if( expression == 0 ) ...
if( !expression ) ...
```

在每对语句中，两条语句的功能是相同的。测试“不等于 0”既可以用关系操作符来实现，也可以简单地通过测试表达式的值来完成。类似，测试“等于 0”也可以通过测试表达式的值，然后再取结果值的逻辑反来实现。你喜欢使用哪种形式纯属风格问题，但你在使用最后一种形式时必须多加小心。由于！操作符的优先级很高，所以如果表达式内包含了其他操作符，你最好把表达式放在一对括号内。

警告：

如果说下面这个错误不是 C 程序员新手最常见的错误，那么它至少也是最令人恼火的错误。绝大多数其他语言使用=操作符来比较相等性。在 C 中，你必须使用双等于号==来执行这个比较，单个=号用于赋值操作。

这里的陷阱在于：在测试相等性的地方出现赋值符是合法的，它并非是一个语法错误[1]。这个不幸的特点正是 C 不具备布尔类型的不利之处。这两个表达式都是合法的整型表达式，所以它们在这个上下文环境中都是合法的。

如果你使用了错误的操作符，会出现什么后果呢？考虑下面这个例子，对于 Pascal 和 Modula 程序员而言，它看上去并无不当之处：

```
x = get_some_value();
if( x = 5 )
        执行某些任务
```

x 从函数获得一个值，但接下来我们把 5 赋值给 x，而不是把 x 与字面值 5 进行比较，从而丢失了从函数获得的那个值[2]。这个结果显然不是程序员的意图所在。但是，这里还存在另外一个问题。由于表达式的值是 x 的新值（非零值），所以 if 语句将始终为真。

你应该养成一个习惯，当你进行相等性测试比较时，你要检查一下你所书写的确实是双等号符。当你发现程序运行不正常时，赶快检查一下比较操作符有没有写错，这可能给你节省大量的调试时间。

5.1.7 逻辑操作符

逻辑操作符有&&和||。这两个操作符看上去有点像位操作符，但它们的具体操作却大相径庭——它们用于对表达式求值，测试它们的值是真还是假。让我们先看一下&&操作符。

---

1 有些编译器对于这类可疑的表达式将产生警告信息。在极偶尔的情况下，你确实要在比较中出现赋值时，此时你应该把赋值操作放在括号里，以避免产生警告信息。

2 =操作符有时被开玩笑地称为“现在就是”操作符，“你问 x 是不是等于 5？对！它现在就等于 5。”

```
expression1  &&  expression2
```

如果 expression1 和 expression2 的值都是真的，那么整个表达式的值也是真的。如果两个表达式中的任何一个表达式的值为假，那么整个表达式的值便为假。到目前为止，一切都很正常。

这个操作符存在一个有趣之处，就是它会控制子表达式求值的顺序。例如，下面这个表达式：

```
a > 5 && a < 10
```

&&操作符的优先级比>和<操作符的优先级都要低，所以子表达式是按照下面这种方式进行组合的：

```
( a > 5 ) && ( a < 10 )
```

但是，尽管&&操作符的优先级较低，但它仍然会对两个关系表达式施加控制。下面是它的工作原理：&&操作符的左操作数总是首先进行求值，如果它的值为真，然后就紧接着对右操作数进行求值。如果左操作数的值为假，那么右操作数便不再进行求值，因为整个表达式的值肯定是假的，右操作数的值已无关紧要。||操作符也具有相同的特点，它首先对左操作数进行求值，如果它的值是真，右操作数便不再求值，因为整个表达式的值此时已经确定。这个行为常常被称为“**短路求值**(short-circuited evaluation)”。

表达式的顺序必须确保正确，这点非常有用。下面这个例子在标准 Pascal 中是非法的：

```
if ( x >= 0 && x < MAX && array[x] == 0 ) ...
```

在 C 中，这段代码首先检查 x 的值是否在数组下标的合法范围之内。如果不是，代码中的下标引用表达式便被忽略。由于 Pascal 将完整地对所有的子表达式进行求值，所以如果下标值错误，尽管程序员已经费尽心思对下标值进行范围检查，但程序仍会由于无效的下标引用而导致失败。

警告：

位操作符常常与逻辑操作符混淆，但它们是不可互换的。它们之间的第 1 个区别是||和&&操作符具有短路性质，如果表达式的值根据左操作数便可决定，它就不再对右操作数进行求值。与之相反，|和&操作符两边的操作数都需要进行求值。

其次，逻辑操作符用于测试零值和非零值，而位操作符用于比较它们的操作数中对应的位。这里有一个例子：

```
if( a < b && c > d ) ...
if( a < b & c > d ) ...
```

因为关系操作符产生的或者是 0，或者是 1，所以这两条语句的结果是一样的。但是，如果 a 是 1 而 b 是 2，下一对语句就不会产生相同的结果。

```
if( a && b ) ...
if( a & b ) ...
```

因为 a 和 b 都是非零值，所以第 1 条语句的值为真，但第 2 条语句的值却是假，因为在 a 和 b 的位模式中，没有一个位在两者中的值都是 1。

5.1.8　条件操作符

条件操作符接受三个操作数。它也会控制子表达式的求值顺序。下面是它的用法：

```
expression1 ? expression2 : expression3
```

条件操作符的优先级非常低，所以它的各个操作数即使不加括号，一般也不会有问题。但是，为了清楚起见，人们还是倾向于在它的各个子表达式两端加上括号。

首先计算的是 expression1，如果它的值为真（非零值），那么整个表达式的值就是 expression2 的值，expression3 不会进行求值。但是，如果 expression1 的值是假（零值），那么整个条件语句的值就是 expression3 的值，expression2 不会进行求值。

如果你觉得记住条件操作符的工作过程有点困难，你可以试一试以问题的形式对它进行解读。例如，

```
a > 5 ? b - 6 : c / 2
```

可以读作“a 是不是大于 5？如果是，就执行 b-6，否则执行 c/2”。语言设计者选择问号符来表示条件操作符决非一时心血来潮。

提示：

什么时候要用到条件操作符呢？这里有两个程序片段：

```
if( a > 5 )
        b = 3;
else
        b = -20;
```

```
b = a > 5 ? 3 : -20;
```

这两段代码所实现的功能完全相同，但左边的代码段要两次书写“b=”。当然，这并没什么大不了，在这种场合使用条件操作符并无优势可言。但是，请看下面这条语句：

```
if( a > 5 )
        b[ 2 * c + d( e / 5 ) ] = 3;
else
        b[ 2 * c + d( e / 5 ) ] = -20;
```

在这里，长长的下标表达式需要写两次，确实令人讨厌。如果使用条件操作符，看上去就清楚得多：

```
b[ 2 * c + d( e / 5 ) ] = a > 5 ? 3 : -20;
```

在这个例子里，使用条件操作符就相当不错，因为它的好处显而易见。在此例中，使用条件操作符出现打字错误的可能性也比前一种写法要低，而且条件操作符可能会产生较小的目标代码。当你习惯了条件操作符之后，你会像理解 if 语句那样轻松看懂这类语句。

5.1.9 逗号操作符

提起逗号操作符，你可能都有点听腻了。但在有些场合，它确实相当有用。它的用法如下：

*expression1, expression2, ... , expression*N

逗号操作符将两个或多个表达式分隔开来。这些表达式自左向右逐个进行求值，整个逗号表达式的值就是最后那个表达式的值。例如：

```
if( b + 1, c / 2, d > 0 )
```

如果 d 的值大于 0，那么整个表达式的值就为真。当然，没有人会这样编写代码，因为对前两个表达式的求值毫无意义，它们的值只是被简单地丢弃。但是，请看下面的代码：

```
a = get_value();
count_value( a );
while( a > 0 ){
        ...
        a = get_value();
        count_value( a );
}
```

在这个 while 循环的前面，有两条独立的语句，它们用于获得在循环表示式中进行测试的值。这样，在循环开始之前和循环体的最后必须各有一份这两条语句的拷贝。但是，如果使用逗号操作符，你可以把这个循环改写为：

```
while( a = get_value(), count_value( a ), a > 0 ) {
        ...
}
```

你也可以使用内嵌的赋值形式，如下所示：

```
while( count_value( a = get_value() ), a > 0 ) {
        ...
}
```

提示：

现在，循环中用于获得下一个值的语句只需要出现一次。逗号操作符使源程序更易于维护。如果用于获得下一个值的方法在将来需要改变，那么代码中只有一个地方需要修改。

但是，面对这个优点，我们很容易表现过头。所以在使用逗号操作符之前，你要问问自己它能不能让程序在某方面表现更出色。如果答案是否定的，你就不要使用它。顺便说一下，“更出色”并不包括“更炫”、“更酷”或“令人印象更深刻”。

这里有一个技巧，你偶尔可能会看到：

```
while( x < 10 )
        b += x,
        x += 1;
```

在这个例子中，逗号操作符把两条赋值语句整合成一条语句，从而避免了在它们的两端加上花括号。不过，这并不是个好做法，因为逗号和分号的区别过于细微，人们很难注意到第 1 个赋值后面是一个逗号而不是个分号。

5.1.10　下标引用、函数调用和结构成员

剩余的一些操作符我将在本书的其他章节详细讨论，但为了完整起见，我在这里顺便提一下它们。下标引用操作符是一对方括号。下标引用操作符接受两个操作数：一个数组名和一个索引值。事实上，下标引用并不仅限于数组名，不过我们将到第 6 章再讨论这个话题。C 的下标引用与其他语言的下标引用很相似，不过它们的实现方式稍有不同。C 的下标值总是从零开始，并且不会对下标值进行有效性检查。除了优先级不同之外，下标引用操作和间接访问表达式是等价的。这里是它们的映像关系：

```
array[ 下标 ]
*( array + ( 下标 ) )
```

下标引用实际上是以后面这种形式实现的，当你从第 6 章起越来越频繁地使用指针时，认识这一点将会越来越重要。

函数调用操作符接受一个或多个操作数。它的第 1 个操作数是你希望调用的函数名，剩余的操作数就是传递给函数的参数。把函数调用以操作符的方式实现意味着“表达式”可以代替“常量”作为函数名，事实也确实如此。第 7 章将详细讨论函数调用操作符。

.和->操作符用于访问一个结构的成员。如果 s 是个结构变量，那么 s.a 就访问 s 中名叫 a 的成员。当你拥有一个指向结构的指针而不是结构本身，且欲访问它的成员时，就需要使用->操作符而不是.

操作符。第 10 章将详细讨论结构、结构的成员以及这些操作符。

5.2 布尔值

C 并不具备显式的布尔类型，所以使用整数来代替。其规则是：

零是假，任何非零值皆为真。

然而，标准并没有说 1 这个值比其他任何非零值“更真”。考虑下面的代码段：

```
a = 25;
b = 15;
if( a ) ...
if( b ) ...
if( a == b ) ...
```

第 1 个测试检查 a 是否为非零值，结果为真。第 2 个测试检查 b 是否不等于 0，其结果也是真。但第 3 个测试并不是检查 a 和 b 的值是否都为“真”，而是测试两者是否相等。

当你在需要布尔值的上下文环境中使用整型变量时，便有可能出现这类问题。

```
nonzero_a = a != 0;
...
if( nonzero_a == ( b != 0 ) ) ...
```

当 a 和 b 的值或者都是零，或者都不是零时，这个测试的结果为真。这个测试如上所示并没有问题，但如果你把(b != 0)这个表达式换作“相同”的表达式 b：

```
if( nonzero_a == b ) ...
```

这个表达式不再用于测试 a 和 b 是否都为零或非零值，而是用于测试 b 是否为某个特定的整型值，即 0 或者 1。

警告：

尽管所有的非零值都被认为是真，但是当你在两个真值之间相互比较时必须小心，因为许多不同的值都可能代表真。

这里有一种程序员经常使用的简写手法，用于 if 语句中——此时就可能出现这种麻烦。假如你进行了下面这些#define 定义，它们后面的每对语句看上去似乎都是等价的。

```
#define FALSE    0
#define TRUE     1
...
if( flag == FALSE ) ...
if( !flag ) ...

if( flag == TRUE ) ...
if( flag ) ...
```

但是，如果 flag 设置为任意的整型值，那么第 2 对语句就不是等价的。只有当 flag 确实是 TRUE 或 FALSE，或者是关系表达式或逻辑表达式的结果值时，两者才是等价的。

提示：

解决所有这些问题的方法是避免混合使用整型值和布尔值。如果一个变量包含了一个任意的整型值，你应该显式地对它进行测试：

```
if( value != 0 ) ...
```

不要使用简写法来测试变量是零还是非零，因为这类形式错误地暗示该变量在本质上是布尔型的。

如果一个变量用于表示布尔值，你应该始终把它设置为 0 或者 1，例如：

```
positive_cash_flow = cash_balance >= 0;
```

不要通过把它与任何特定的值进行比较来测试这个变量是否为真值，哪怕是与 TRUE 或 FALSE 进行比较。相反，你应该像下面这样测试变量的值：

```
if( positive_cash_flow ) ...
if( !positive_cash_flow ) ...
```

如果你选择使用描述性的名字来表示布尔型变量，这个技巧更加管用，能够提高代码的可读性："如果现金流量为正，那么 ..."

5.3 左值和右值

为了理解有些操作符存在的限制，你必须理解**左值**(L-value)和**右值**(R-value)之间的区别。这两个术语是多年前由编译器设计者所创造并沿用至今，尽管它们的定义并不与 C 语言严格吻合。

左值就是那些能够出现在赋值符号左边的东西。右值就是那些可以出现在赋值符号右边的东西。这里有个例子：

```
a = b + 25;
```

a 是个左值，因为它标识了一个可以存储结果值的地点，b + 25 是个右值，因为它指定了一个值。

它们可以互换吗？

```
b + 25 = a;
```

原先用作左值的 a 此时也可以当作右值，因为每个位置都包含一个值。然而，b + 25 不能作为左值，因为它并未标识一个特定的位置。因此，这条赋值语句是非法的。

注意当计算机计算 b + 25 时，它的结果必然保存于机器的某个地方。但是，程序员并没有办法预测该结果会存储在什么地方，也无法保证这个表达式的值下次还会存储于那个地方。其结果是，这个表达式不是一个左值。基于同样的理由，字面值常量也都不是左值。

听上去似乎是变量可以作为左值而表达式不能作为左值，但这个推断并不准确。在下面的赋值语句中，左值便是一个表达式。

```
int     a[30];
...
a[ b + 10 ] = 0;
```

下标引用实际上是一个操作符，所以表达式的左边实际上是个表达式，但它却是一个合法的左值，因为它标识了一个特定的位置，我们以后可以在程序中引用它。这里有另外一个例子：

```
int     a, *pi;
...
pi = &a;
*pi = 20;
```

请看第 2 条赋值语句，它左边的那个值显然是一个表达式，但它却是一个合法的左值。为什么？指针 pi 的值是内存中某个特定位置的地址，*操作符使机器指向那个位置。当它作为左值使用时，

这个表达式指定需要进行修改的位置。当它作为右值使用时，它就提取当前存储于这个位置的值。

有些操作符，如间接访问和下标引用，它们的结果是个左值。其余操作符的结果则是个右值。为了便于参考，这些信息也包含于本章后面的表 5.1 所示的优先级表格中。

5.4 表达式求值

表达式的求值顺序一部分是由它所包含的操作符的优先级和结合性决定。同样，有些表达式的操作数在求值过程中可能需要转换为其他类型。

5.4.1 隐式类型转换

C 的整型算术运算总是至少以缺省整型类型的精度来进行的。为了获得这个精度，表达式中的字符型和短整型操作数在使用之前被转换为普通整型，这种转换称为整型提升（integral Promotion）。例如，在下面表达式的求值中，

```
char    a, b, c;
...
a = b + c;
```

b 和 c 的值被提升为普通整型，然后再执行加法运算。加法运算的结果将被截短，然后再存储于 a 中。这个例子的结果和使用 8 位算术的结果是一样的。但在下面这个例子中，它的结果就不再相同。这个例子用于计算一系列字符的简单检验和。

```
a = ( ~a ^ b << 1 ) >> 1;
```

由于存在求补和左移操作，所以 8 位的精度是不够的。标准要求进行完整的整型求值，所以对于这类表达式的结果，不会存在歧义性[1]。

5.4.2 算术转换

如果某个操作符的各个操作数属于不同的类型，那么除非其中一个操作数转换为另外一个操作数的类型，否则操作就无法进行。下面的层次体系称为**寻常算术转换**(usual arithmetic conversion)。

```
long double
double
float
unsigned long int
long int
unsigned int
int
```

如果某个操作数的类型在上面这个列表中排名较低，那么它首先将转换为另外一个操作数的类型然后执行操作。

警告：

下面这个代码段包含了一个潜在的问题。

```
int     a = 5000;
int     b = 25;
long    c = a * b;
```

[1] 事实上，标准说明结果应该通过完整的整型求值得到，编译器如果知道采用 8 位精度的求值不会影响最后的结果，它也允许编译器这样做。

问题在于表达式 a*b 是以整型进行计算，在 32 位整数的机器上，这段代码运行起来毫无问题，但在 16 位整数的机器上，这个乘法运算会产生溢出，这样 c 就会被初始化为错误的值。

解决方案是在执行乘法运算之前把其中一个（或两个）操作数转换为长整型。

```
long  c = ( long )a * b;
```

当整型值转换为 float 型值时，也有可能损失精度。float 型值仅要求 6 位数字的精度。如果将一个超过 6 位数字的整型值赋值给一个 float 型变量时，其结果可能只是该整型值的近似值。

当 float 型值转换为整型值时，小数部分被舍弃（并不进行四舍五入）。如果浮点数的值过于庞大，无法容纳于整型值中，那么其结果将是未定义的。

5.4.3　操作符的属性

复杂表达式的求值顺序是由 3 个因素决定的：操作符的优先级、操作符的结合性以及操作符是否控制执行的顺序。两个相邻的操作符哪个先执行取决于它们的优先级，如果两者的优先级相同，那么它们的执行顺序由它们的结合性决定。简单地说，结合性就是一串操作符是从左向右依次执行还是从右向左逐个执行。最后，有 4 个操作符，它们可以对整个表达式的求值顺序施加控制，它们或者保证某个子表达式能够在另一个子表达式的所有求值过程完成之前进行求值，或者可能使某个表达式被完全跳过不再求值。

每个操作符的所有属性都列在表 5.1 所示的优先级表中。表中各个列分别代表操作符、它的功能简述、用法示例、它的结果类型、它的结合性以及当它出现时是否会对表达式的求值顺序施加控制。用法示例提示它是否需要操作数为左值。术语 lexp 表示左值表达式，rexp 表示右值表达式。记住，左值意味着一个位置，而右值意味着一个值。所以，在使用右值的地方也可以使用左值，但是在需要左值的地方不能使用右值。

表 5.1　操作符优先级

操作符	描　　述	用 法 示 例	结 果 类 型	结 合 性	是否控制求值顺序
( )	聚组	(表达式)	与表达式同	N/A	否
( )	函数调用	rexp(rexp, ..., rexp)	rexp	L-R	否
[ ]	下标引用	rexp[rexp]	lexp	L-R	否
.	访问结构成员	lexp.member_name	lexp	L-R	否
->	访问结构指针成员	rexp->member_name	lexp	L-R	否
++	后缀自增	lexp++	rexp	L-R	否
--	后缀自减	lexp--	rexp	L-R	否
!	逻辑反	!rexp	rexp	R-L	否
~	按位取反	~ rexp	rexp	R-L	否
+	单目，表示正值	+ rexp	rexp	R-L	否
-	单目，表示负值	- rexp	rexp	R-L	否
++	前缀自增	++lexp	rexp	R-L	否
--	前缀自减	--lexp	rexp	R-L	否
*	间接访问	* rexp	lexp	R-L	否
&	取地址	&lexp	rexp	R-L	否
sizeof	取其长度，以字节表示	sizeof rexp sizeof(类型)	rexp	R-L	否
(类型)	类型转换	(类型)rexp	rexp	R-L	否

续表

操作符	描　　述	用 法 示 例	结 果 类 型	结 合 性	是否控制求值顺序
*	乘法	rexp * rexp	rexp	L-R	否
/	除法	rexp / rexp	rexp	L-R	否
%	整数取余	rexp % rexp	rexp	L-R	否
+	加法	rexp + rexp	rexp	L-R	否
-	减法	rexp – rexp	rexp	L-R	否
<<	左移位	rexp << rexp	rexp	L-R	否
>>	右移位	rexp >> rexp	rexp	L-R	否
>	大于	rexp > rexp	rexp	L-R	否
>=	大于等于	rexp >= rexp	rexp	L-R	否
<	小于	rexp < rexp	rexp	L-R	否
<=	小于等于	rexp <= rexp	rexp	L-R	否
==	等于	rexp == rexp	rexp	L-R	否
!=	不等于	rexp != rexp	rexp	L-R	否
&	位与	rexp & rexp	rexp	L-R	否
^	位异或	rexp ^ rexp	rexp	L-R	否
|	位或	rexp | rexp	rexp	L-R	否
&&	逻辑与	rexp && rexp	rexp	L-R	是
||	逻辑或	rexp || rexp	rexp	L-R	是
?:	条件操作符	rexp? rexp: rexp	rexp	N/A	是
=	赋值	lexp = rexp	rexp	R-L	否
+=	以...加	lexp += rexp	rexp	R-L	否
-=	以...减	lexp == rexp	rexp	R-L	否
*=	以...乘	lexp *= rexp	rexp	R-L	否
/=	以...除	lexp /= rexp	rexp	R-L	否
%=	以...取模	lexp %= rexp	rexp	R-L	否
<<=	以...左移	lexp <<= rexp	rexp	R-L	否
>>=	以...右移	lexp >>= rexp	rexp	R-L	否
&=	以...与	lexp &= rexp	rexp	R-L	否
^=	以...异或	lexp ^= rexp	rexp	R-L	否
|=	以...或	lexp |= rexp	rexp	R-L	否
,	逗号	rexp, rexp	rexp	L-R	是

5.4.4　优先级和求值的顺序

如果表达式中的操作符超过一个，是什么决定这些操作符的执行顺序呢？C 的每个操作符都具有优先级，用于确定它和表达式中其余操作符之间的关系。但仅凭优先级还不能确定求值的顺序。下面是它的规则：

两个相邻操作符的执行顺序由它们的优先级决定。如果它们的优先级相同，它们的执行顺序由它们的结合性决定。除此之外，编译器可以自由决定使用任何顺序对表达式进行求值，只要它不违背逗号、&&、||和?:操作符所施加的限制。

换句话说，表达式中操作符的优先级只决定表达式的各个组成部分在求值过程中如何进行聚组。

这里有一个例子：

```
a + b * c
```

在这个表达式中，乘法和加法操作符是两个相邻的操作符。由于*操作符的优先级比+操作符高，所以乘法运算先于加法运算执行。编译器在这里别无选择，它必须先执行乘法运算。

下面是一个更为有趣的表达式：

```
a * b + c * d + e * f
```

如果仅由优先级决定这个表达式的求值顺序，那么所有 3 个乘法运算将在所有加法运算之前进行。事实上，这个顺序并不是必需的。实际上只要保证每个乘法运算在它相邻的加法运算之前执行即可。例如，这个表达式可能会以下面的顺序进行，其中粗体的操作符表示在每个步骤中进行操作的操作符。

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

注意第 1 个加法运算在最后一个乘法运算之前执行。如果这个表达式按以下的顺序执行，其结果是一样的：

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

加法运算的结合性要求两个加法运算按照先左后右的顺序执行，但它对表达式剩余部分的执行顺序并未加以限制。尤其是，这里并没有任何规则要求所有的乘法运算首先进行，也没有规则规定这几个乘法运算之间谁先执行。优先级规则在这里起不到作用，优先级只对相邻操作符的执行顺序起作用。

警告：

由于表达式的求值顺序并非完全由操作符的优先级决定，所以像下面这样的语句是很危险的。

```
c + --c
```

操作符的优先级规则要求自减运算在加法运算之前进行，但我们并没有办法得知加法操作符的左操作数是在右操作数之前还是之后进行求值。它在这个表达式中将存在区别，因为自减操作符具有副作用。--c 在 c 之前或之后执行，表达式的结果在两种情况下将会不同。

标准说明类似这种表达式的值是未定义的。尽管每种编译器都会为这个表达式产生某个值，但到底哪个是正确的并无标准答案。因此，像这样的表达式是不可移植的，应该予以避免。程序 5.3 以相当戏剧化的结果说明了这个问题。表 5.2 列出了在各种编译器中这个程序所产生的值。许多编译器由于是否添加了优化措施而导致结果不同。例如，在 gcc 中使用了优化器后，程序的值从 -63 变成了 22。尽管每个编译器以不同的顺序计算这个表达式，**但你不能说任何一种方法是错误的！**这是由于表达式本身的缺陷引起的，由于它包含了许多具有副作用的操作符，因此它的求值顺序存在歧义。

```
/*
** 一个证明表达式的求值顺序只是部分由操作符的优先级决定的程序。
*/
```

```
main()
{
        int     i = 10;

        i = i-- - --i * ( i = -3 ) * i++ + ++i;
        printf( "i = %d\n", i );
}
```

程序 5.3 非法表达式 bad_exp.c

表 5.2 非法表达式程序的结果

值	编 译 器
−128	Tandy 6000 Xenix 3.2
−95	Think C 5.02(Macintosh)
−86	IBM PowerPC AIX 3.2.5
−85	Sun Sparc cc(K&C 编译器)
−63	gcc, HP_UX 9.0, Power C 2.0.0
4	Sun Sparc acc(K&C 编译器)
21	Turbo C/C++ 4.5
22	FreeBSD 2.1R
30	Dec Alpha OSF1 2.0
36	Dec VAX/VMS
42	Microsoft C 5.1

K&R C：

在 K&R C 中，编译器可以自由决定以任何顺序对类似下面这样的表达式进行求值。

```
a + b + c
x * y * z
```

之所以允许编译器这样做是因为 b+c（或 y*z）的值可能可以从前面的一些表达式中获得，所以直接复用这个值比重新求值效率更高。加法运算和乘法运算都具有结合性，这样做的缺点在什么地方呢？

考虑下面这个表达式，它使用了有符号整型变量：

```
x + y + 1
```

如果表达式 x+y 的结果大于整型所能容纳的值，它就会产生溢出。在有些机器上，下面这个测试

```
if( x + y + 1 > 0 )
```

的结果将取决于先计算 x+y 还是 y+1，因为在两种情况下溢出的地点不同。问题在于程序员无法肯定地预测编译器将按哪种顺序对这个表达式求值。经验显示，上面这种做法是个坏主意，所以 ANSI C 不允许这样做。

下面这个表达式说明了一个相关的问题。

```
f() + g() + h()
```

尽管左边那个加法运算必须在右边那个加法运算之前执行，但对于各个函数调用的顺序，并没有规则加以限制。如果它们的执行具有副作用，比如执行一些 I/O 任务或修改全局变量，那么函数

调用顺序的不同可能会产生不同的结果。因此，如果顺序会导致结果产生区别，你最好使用临时变量，让每个函数调用都在单独的语句中进行。

```
temp = f();
temp += g();
temp += h();
```

5.5 总结

C 具有丰富的操作符。算术操作符包括+（加）、–（减）、*（乘）、/（除）和%（取模）。除了%操作符之外，其余几个操作符不仅可以作用于整型值，还可以作用于浮点型值。

<<和>>操作符分别执行左移位和右移位操作。&、|和 ^ 操作符分别执行位的与、或和异或操作。这几个操作符都要求其操作数为整型。

=操作符执行赋值操作。而且，C 还存在复合赋值符，它把赋值符和前面那些操作符结合在一起：

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

复合赋值符在左右操作数之间执行指定的运算，然后把结果赋值给左操作数。

单目操作符包括!（逻辑非）、～（按位取反）、–（负值）和+（正值）。++和--操作符分别用于增加或减少操作数的值。这两个操作符都具有前缀和后缀形式。前缀形式在操作数的值被修改之后才返回这个值，而后缀形式在操作数的值被修改之前就返回这个值。&操作符返回一个指向它的操作数的指针（取地址），而*操作符对它的操作数（必须为指针）执行间接访问操作。sizeof 返回操作数的类型的长度，以字节为单位。最后，强制类型转换(cast)用于修改操作数的数据类型。

关系操作符有：

```
>           >=          <           <=          !=          ==
```

每个操作符根据它的操作数之间是否存在指定的关系，或者返回真，或者返回假。逻辑操作符用于计算复杂的布尔表达式。对于&&操作符，只有当它的两个操作数的值都为真时，它的值才是真；对于||操作符，只有当它的两个操作数的值都为假时，它的值才是假。这两个操作符会对包含它们的表达式的求值过程施加控制。如果整个表达式的值通过左操作数便可决定，那么右操作数便不再求值。

条件操作符?:接受 3 个参数，它也会对表达式的求值过程施加控制。如果第 1 个操作数的值为真，那么整个表达式的结果就是第 2 个操作数的值，第 3 个操作数不会执行。否则，整个表达式的结果就是第 3 个操作数的值，而第 2 个操作数将不会执行。逗号操作符把两个或更多个表达式连接在一起，从左向右依次进行求值，整个表达式的值就是最右边那个子表达式的值。

C 并不具备显式的布尔类型，布尔值是用整型表达式来表示的。然而，在表达式中混用布尔值和任意的整型值可能会产生错误。要避免这些错误，每个变量要么表示布尔型，要么表示整型，不可让它身兼两职。不要对整型变量进行布尔值测试，反之亦然。

左值是个表达式，它可以出现在赋值符的左边，它表示计算机内存中的一个位置。右值表示一个值，所以它只能出现在赋值符的右边。每个左值表达式同时也是个右值，但反过来就不是这样。

各个不同类型之间的值不能直接进行运算，除非其中之一的操作数转换为另一操作数的类型。寻常算术转换决定哪个操作数将被转换。操作符的优先级决定了相邻的操作符哪个先被执行。如果

它们的优先级相等，那么它们的结合性将决定它们执行的顺序。但是，这些并不能完全决定表达式的求值顺序。编译器只要不违背优先级和结合性规则，它可以自由决定复杂表达式的求值顺序。表达式的结果如果依赖于求值的顺序，那么它在本质上就是不可移植的，应该避免使用。

5.6 警告的总结

1. 有符号值的右移位操作是不可移植的。
2. 移位操作的位数是个负值。
3. 连续赋值中各个变量的长度不一。
4. 误用＝而不是＝＝进行比较。
5. 误用|替代||，误用&替代&&。
6. 在不同的用于表示布尔值的非零值之间进行比较。
7. 表达式赋值的位置并不决定表达式计算的精度。
8. 编写结果依赖于求值顺序的表达式。

5.7 编程提示的总结

1. 使用复合赋值符可以使程序更易于维护。
2. 使用条件操作符替代 if 语句以简化表达式。
3. 使用逗号操作符来消除多余的代码。
4. 不要混用整型和布尔型值。

5.8 问题

1. 下面这个表达式的类型和值分别是什么？

```
(float)( 25 / 10 )
```

2. 下面这个程序的结果是什么？

```
int
func( void )
{
        static  int     counter = 1;

        return ++counter;
}

int
main()
{
        int     answer;

        answer = func() - func() * func();
        printf( "%d\n", answer );
}
```

3. 你认为位操作符和移位操作符可以用在什么地方？
4. 条件操作符在运行时较之 if 语句是更快还是更慢？试比较下面两个代码段。

```
if( a > 3 )
        i = b + 1;
else
        i = c * 5;
```

```
i = a > 3 ? b + 1 : c * 5;
```

5. 可以被 4 整除的年份是闰年，但是其中能够被 100 整除的年份又不是闰年。但是，这其中能够被 400 整除的年份又是闰年。请用一条赋值语句，如果变量 year 的值是闰年，把变量 leap_year 设置为真。如果 year 的值不是闰年，把 leap_year 设置为假。
6. 哪些操作符具有副作用？它们具有什么副作用？
7. 下面这个代码段的结果是什么？

```
int     a = 20;
...
if( 1 <= a <= 10 )
        printf( "In range\n" );
else
        printf( "Out of range\n" );
```

8. 改写下面的代码段，消除多余的代码。

```
a = f1( x );
b = f2( x + a );
for( c = f3( a, b ); c > 0; c = f3( a, b ) ){
        statements
        a = f1( ++x );
        b = f2( x + a );
}
```

9. 下面的循环能够实现它的目的吗？

```
non_zero = 0;
for( i = 0; i < ARRAY_SIZE; i += 1 )
        non_zero += array[i];

if( !non_zero )
        printf( "Values are all zero\n" );

else
        printf( "There are nonzero values\n" );
```

10. 根据下面的变量声明和初始化，计算下列每个表达式的值。如果某个表达式具有副作用（也就是说它修改了一个或多个变量的值），注明它们。在计算每个表达式时，每个变量所使用的是开始时给出的初始值，而不是前一个表达式的结果。

```
int     a = 10, b = -25;
int     c = 0, d = 3;
int     e = 20;
```

```
a.b
b.b++
c.--a
d.a / 6
e.a % 6
f.b % 10
g.a << 2
h.b >> 3
i.a > b
j.b = a
k.b == a
l.a & b
m.a ^ b
n.a | b
o.~b
p.c && a
```

```
q.c || a
r.b ? a : c
s.a += 2
t.b &= 20
u.b >>= 3
v.a %= 6
w.d = a > b
x.a = b = c = d
y.e = d + ( c = a + b ) + c
z.a + b * 3
aa.b >> a - 4
bb.a != b != c
cc.a == b == c
dd.d < a < e
ee.e > a > d
ff.a - 10 > b + 10
gg.a & 0x1 == b & 0x1
hh.a | b << a & b
ii.a > c || ++a > b
jj.a > c && ++a > b
kk.! ~ b++
ll.b++ & a <= 30
mm.a - b, c += d, e - c
nn.a <<= 3 > 0
oo.a <<= d > 20 ? b && c++ : d--
```

11. 下面列出了几个表达式。请判断编译器是如何对各个表达式进行求值的，并在不改变求值顺序的情况下，尽可能去除多余的括号。

    a. `a + ( b / c )`

    b. `( a + b ) / c`

    c. `( a * b ) % 6`

    d. `a * ( b % 6 )`

    e. `( a + b ) == 6`

    f. `! ( ( a >= '0' ) && ( a <= '9' ) )`

    g. `( ( a & 0x2f ) == ( b | 1 ) ) && ( ( ~ c ) > 0 )`

    h. `( ( a << b ) - 3 ) < ( b << ( a + 3 ) )`

    i. `~ ( a ++ )`

    j. `((a == 2) || (a == 4)) && ((b == 2) || (b == 4))`

    k. `( a & b ) ^ ( a | b )`

    l. `( a + ( b + c ) )`

12. 如何判断在你的机器上对一个有符号值进行右移位操作时执行的是算术移位还是逻辑移位？

5.9 编程练习

★ 1. 编写一个程序，从标准输入读取字符，并把它们写到标准输出中。除了大写字母字符要转换为小写字母之外，所有字符的输出形式应该和它的输入形式完全相同。

★★ 2. 编写一个程序，从标准输入读取字符，并把它们写到标准输出中。所有非字母字符都完全按照它的输入形式输出，字母字符在输出前进行加密。

加密方法很简单：每个字母被修改为在字母表上距其 13 个位置（前或后）的字母。例如，A 被修改为 N，B 被修改为 O，Z 被修改为 M，以此类推。注意大小写字母都应该被转换。**提示**：记住字符实际上是一个较小的整型值这一点可能对你有所帮助。

★★★ 3. 请编写函数

```
unsigned int reverse_bits( unsigned int value );
```

这个函数的返回值是把 value 的二进制位模式从左到右变换一下后的值。例如，在 32 位机器上，25 这个值包含下列各个位：

```
00000000000000000000000000011001
```

函数的返回值应该是 2 550 136 832，它的二进制位模式是：

```
10011000000000000000000000000000
```

编写函数时要注意不要让它依赖于你的机器上整型值的长度。

★★★★ 4. 编写一组函数，实现位数组。函数的原型应该如下：

```
void set_bit( char bit_array[],
    unsigned bit_number );

void clear_bit( char bit_array[],
    unsigned bit_number );

void assign_bit( char bit_array[],
    unsigned bit_number, int value );

int test_bit( char bit_array[],
    unsigned bit_number );
```

每个函数的第 1 个参数是个字符数组，用于实际存储所有的位。第 2 个参数用于标识需要访问的位。函数的调用者必须确保这个值不要太大，以至于超出数组的边界。第 1 个函数把指定的位设置为 1，第 2 个函数则把指定的位清零。如果 value 的值为 0，第 3 个函数把指定的位清 0，否则设置为 1。至于最后一个函数，如果参数中指定的位不是 0，函数就返回真，否则返回假。

★★★★ 5. 编写一个函数，把一个给定的值存储到一个整数中指定的几个位。它的原型如下：

```
int store_bit_field(int original_value,
    int value_to_store,
    unsigned starting_bit,unsigned ending_bit);
```

假定整数中的位是从右向左进行编号。因此，起始位的位置不会小于结束位的位置。为了更清楚地说明，函数应该返回下列值：

原 始 值	需要存储的值	起 始 位	结 束 位	返 回 值
0x0	0x1	4	4	0x10
0xffff	0x123	15	4	0x123f
0xffff	0x123	13	9	0xc7ff

提示：把一个值存储到一个整数中指定的几个位分为 5 个步骤。以上表最后一行为例：

1. 创建一个掩码(mask)，它是一个值，其中需要存储的位置相对应的那几个位设置为 1。此时掩码为 001111100000000。
2. 用掩码的反码对原值执行 AND 操作，将那几个位设置为 0。原值 1111111111111111，

操作后变为 1100000111111111。

3. 将新值左移，使它与那几个需要存储的位对齐。新值 0000000100100011(0x123)，左移后变为 0100011000000000。
4. 把移位后的值与掩码进行位 AND 操作，确保除那几个需要存储的位之外的其余位都设置为 0。进行这个操作之后，值变为 0000011000000000。
5. 把结果值与原值进行位 OR 操作，结果为 1100011111111111（0xc7ff），也就是最终的返回值。

在所有任务中，最困难的是创建掩码。你一开始可以把~0 这个值强制转换为无符号值，然后再对它进行移位。