成惠资料订购链接

第12章 位 运 算

位运算是C语言的重要特色，是其他计算机高级语言所没有的。

所谓位运算是指以二进制位为对象的运算。在系统软件中，常要处理二进制位的问题。例如，将一个存储单元中的各二进制位左移或右移一位、两个数按位相加等。C语言提供位运算的功能，与其他高级语言相比，它显然具有很大的优越性。

指针运算和位运算往往是编写系统软件所需要的。在计算机用于检测和控制领域中也要用到位运算的知识，因此要真正掌握和使用好C语言，应当学习位运算。

12.1 位运算和位运算符

C语言提供表12.1所列出的位运算符。

表 12.1

运算符	含 义	运算符	含 义
&	按位与	~	按位取反
|	按位或	<<	左移
^	按位异或	>>	右移

说明：

(1) 位运算符中除“~”以外，均为二目(元)运算符，即要求两侧各有一个运算量。如a&b。

(2) 参加位运算的对象只能是整型或字符型的数据，不能为实型数据。

下面对各种位运算分别介绍。

12.1.1 “按位与”运算

参加运算的两个数据按二进制位进行“与”运算。如果两个相应的二进制位都为1，则该位的结果值为1；否则为0。即

0&0=0， 0&1=0， 1&0=0， 1&1=1。

例如，7 &5并不等于12，应该是进行“按位与”的运算：

```
  00000111(7)     (7用二进制表示为111)
(&) 00000101(5)   (5用二进制表示为101)
-------------------
  00000101(5)     (二进制数101等于十进制数5)
```

因此，7 &5的值为5。如果参加“&”运算的是负数(如-7 & -5)，则以补码形式表示为二进制数，然后按位进行“与”运算。

注意：&& 是逻辑与运算符，7 && 5 的值为 1，因为非 0 的数值按“真”处理，逻辑与的结果是“真”，以 1 表示。而 & 是按位与，7 & 5 的结果是 5。

按位与有一些特殊的用途：

(1) **清零**。如果想将一个单元清零，即使其全部二进制位为 0，只要找一个二进制数，其中各个位符合以下条件：原来的数中为 1 的位，新数中相应位为 0。然后使二者进行 & 运算，即可达到清零目的。

例如，原有数为 00101011，另找一个数，设它为 10010100，它符合以上条件，即在原数为 1 的位置上，它的位值均为 0。将两个数进行 & 运算：

```
  00101011
(&) 10010100
------------
  00000000
```

其道理是显然的。当然也可以不用 10010100 这个数，而用其他数(如 01000100)也可以，只要符合上述条件即可。

(2) **取一个数中某些指定位**。如有一个整数 a(为方便起见，以 2 个字节表示)，想要其中的低字节。只须将 a 与$(377)_8$按位与即可，见图 12.1。

图中表示 c=a&b 的运算。b 为八进制数的 377，运算后 c 只保留 a 的低字节，高字节为 0。

如果想取两个字节中的高字节，只须用 c=a & 0177400 (0177499 表示八进制数的 177400)，见图 12.2。

a	00 10 11 00	10 10 11 00
b	00 00 00 00	11 11 11 11
c	00 00 00 00	10 10 11 00

图 12.1

a	00 10 11 00	10 10 11 00
b	11 11 11 11	00 00 00 00
c	00 10 11 00	00 00 00 00

图 12.2

(3) **要想将哪一位保留下来，就与一个数进行 & 运算，此数在该位取 1**。例如，有一数 01010100，想把其中左面第 3,4,5,7,8 位保留下来，可以这样运算：

```
  01010100  (十进制数 84)
(&) 00111011  (十进制数 59)
------------
  00010000  (十进制数 16)
```

可以看到得到的结果的数(16) 其中第 3,4,5,7,8 位就是 01010100 的第 3,4,5,7,8 位，其他各位均为 0。以上运算表示为：a=84,b=59,c=a & b,结果是 16。

12.1.2 “按位或”运算

按位或运算的规则是：两个对应的二进制位中只要有一个为 1，该位的结果值为 1。即

0|0=0, 0|1=1, 1|0=1, 1|1=1。

例如：

将八进制数 60 与八进制数 17 进行按位或运算。

```
060 | 017
```

```
    00110000    (八进制数 60)
(|) 00001111    (八进制数 17)
    ----------
    00111111
```

低 4 位全为 1。如果想使一个数 a 的低 4 位改为 1，只须将 a 与 017 进行按位或运算即可。

按位或运算常用来对一个数据的某些位定值为 1。例如，a 是一个整数，有表达式：a|0377，则低 8 位全置为 1，高 8 位保留原样。

12.1.3　“异或”运算

异或运算符“∧”也称 XOR 运算符。它的规则是：若参加运算的两个二进制位异号，则得到 1(真)，若同号，则结果为 0(假)。即

0∧0=0，　0∧1=1，　1∧0=1，　1∧1=0。

例如：

```
    00111001  (十进制数 57,八进制数 071)
(∧) 00101010  (十进制数 42,八进制数 052)
    ----------
    00010011  (十进制数 19,八进制数 023)
```

即 071∧052，结果为 023(八进制数)。

“异或”的意思是判断两个相应的位值是否为“异”。为“异”(值不同)就取真(1)；否则为假(0)。

下面举例说明异或运算的应用。

(1) **使特定位翻转。**

假设有 01111010，想使其低 4 位翻转，即 1 变为 0，0 变为 1。可以将它与 00001111 进行异或(∧)运算，即

```
    01111010
(∧) 00001111
    ----------
    01110101
```

结果值的低 4 位正好是原数低 4 位的翻转。要使哪几位翻转就将与其进行∧运算的数在该几位置为 1(其他位置为 0)即可。这是因为原数中值为 1 的位与 1 进行∧运算得 0，原数中的位值 0 与 1 进行∧运算的结果得 1。

(2) **与 0 相∧，保留原值。**

例如：012∧00=012

```
    00001010
(∧) 00000000
    ----------
    00001010
```

因为原数中的 1 与 0 进行∧运算得 1，0∧0 得 0，故保留原数。

(3) **交换两个值，不用临时变量。**

假如 a=3，b=4。想将 a 和 b 的值互换，可以用以下赋值语句实现：

```
a=a∧b;
b=b∧a;
a=a∧b;
```

可以用下面的竖式来说明：

```
        a=0 1 1
(∧)     b=1 0 0
-------------------
        a=1 1 1     (a∧b 的结果,a 已变成 7)
(∧)     b=1 0 0
-------------------
        b=0 1 1     (b∧a 的结果,b 已变成 3)
(∧)     a=1 1 1
-------------------
        a=1 0 0     (a∧b 的结果,a 已变成 4)
```

即等效于以下两步：

① 执行前面两个赋值语句："a=a∧b;"和"b=b∧a;"相当于 b=b∧(a∧b)。而 b∧a∧b 等于 a∧b∧b。b∧b 的结果为 0,因为同一个数与本身相∧,结果必为 0。因此 b 的值等于 a∧0,即 a,其值为 3。

② 再执行第 3 个赋值语句：a=a∧b。由于 a 的值等于(a∧b),b 的值等于(b∧a∧b),因此,相当于 a=a∧b∧b∧a∧b,即 a 的值等于 a∧a∧b∧b∧b,等于 b。

a 得到 b 原来的值。

12.1.4 "取反"运算

"～"是一个单目(元)运算符,用来对一个二进制数按位取反,即将 0 变 1,将 1 变 0。例如,～025 是对八进制数 25(即二进制数 00010101)按位求反。

```
  0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1     (八进制数 25)
(～)
-------------------------------------  (按位求反)
  1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0     (八进制数 177752)
```

即八进制数 177752。因此,～025 的值为八进制数 177752。不要误认为～025 的值是－025。

下面举一例说明～运算符的应用。

若一个整数 a 为 16 位,想使最低一位为 0,可以用

```
a=a & 0177776
```

177776 即二进制数 1111111111111110,如果 a 的值为八进制数 75,a & 0177776 的运算可以表示如下：

```
    0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1
(&) 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
-------------------------------------
    0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0
```

a 的最后面一个二进制位变成 0。但如果一个整数 a 为 32 位,想将最后一位变成 0 就不能用 a & 0177776 了。应改用

```
a & 037777777776
```

这种方法不易记忆,可以改用

```
a=a & ～1
```

它对以 16 位和以 32 位存放一个整数的情况都适用,不必作修改。因为在以两个字节存储一个整数时,1 的二进制形式为 0000000000000001,～1 是 1111111111111110(注意～1 不等于－1,要弄清～运算符和负号运算符的不同)。在以 4 个字节存储一个整数时,～1

是 11111111111111111111111111111110。

～运算符的优先级别比算术运算符、关系运算符、逻辑运算符和其他位运算符都高，例如：

```
~a&b
```

先进行～a 运算，然后进行 & 运算。

12.1.5　左移运算

“<<”用来将一个数的各二进制位全部左移若干位。例如：

```
a=a<<2
```

将 a 的二进制数左移 2 位，右补 0。若 a=15，即二进制数 00001111，左移 2 位得 00111100，即得到十进制数 60。高位左移后溢出，舍弃。

```
                00001111
(<<2,左移 2 位)
-------------------------
              00|00111100
                ↑       ↑
          溢出舍弃      右补 0
```

为简单起见，用 8 位二进制数表示十进制数 15，如果用 16 位或 32 位二进制数表示，结果是一样的。

左移 1 位相当于该数乘以 2，左移 2 位相当于该数乘以 $2^2=4$。上面举的例子 15<<2，结果是 60，即乘了 4。但此结论只适用于该数左移时被溢出舍弃的高位中不包含 1 的情况。例如，假设以一个字节(8 位)存一个整数，若 a 为无符号整型变量，则 a=64 时，左移一位时溢出的是 0，而左移 2 位时，溢出的高位中包含 1。

由表 12.2 可以看出，若 a 的值为 64，在左移 1 位后相当于乘 2，左移 2 位后，值等于 0。

表　12.2

a 的值	a 的二进制形式	a<<1		a<<2	
64	01000000	0	10000000	01	00000000
127	01111111	0	11111110	01	11111100

左移比乘法运算快得多，有些 C 编译程序自动将乘 2 的运算用左移一位来实现，将乘 2^n 的幂运算处理为左移 n 位。

12.1.6　右移运算

a>>2 表示将 a 的各二进制位右移 2 位，移到右端的低位被舍弃，对无符号数，高位补 0。例如，a=017 时，a 的值用二进制形式表示为 00001111。

```
                00001111
(a>>2)
-------------------------
              00000011|11
                ↑       ↑
            左补 0    此 2 位舍弃
```

右移一位相当于除以2,右移 n 位相当于除以 2^n。

在右移时,需要注意符号位问题。对无符号数,右移时左边高位补0;对于有符号的数,如果原来符号位为0(该数为正),则左边也补0,如同上例表示的那样。如果符号位原来为1(即负数),则左边移入0还是1,要取决于所用的计算机系统。有的系统补0,有的系统补1。补0的称为"逻辑右移",即简单右移,不考虑数的符号问题,补1的称为"算术右移"(保持原有的符号)。

例如,a是short型变量,用两个字节存放数,若a值为八进制数113755,即最高位为1,对它进行右移两位的运算:a>>1。请看结果:

```
a:      1001011111101101    (用二进制形式表示)
a>>1:   0100101111110110    (逻辑右移时)
a>>1:   1100101111110110    (算术右移时)
```

在有些系统上,a>>1得八进制数045766(逻辑右移时),而在另一些系统上可能得到的是145766(算术右移时)。Visual C++和其他一些C编译采用的是算术右移,即对有符号数右移时,如果符号位原来为1,左面补入高位的是1。

12.1.7 位运算赋值运算符

位运算符与赋值运算符可以组成复合赋值运算符,如:&=,|=,>>=,<<=,^=等。

例如,a &=b相当于a=a&b,a<<=2相当于a=a<<2。

12.1.8 不同长度的数据进行位运算

如果两个数据长度不同(例如short和int型)进行位运算时(如a&b,而a为short型,b为int型),系统会将二者按右端对齐。如果a为正数,则左侧16位补满0;若a为负数,左端应补满1;如果a为无符号整数型,则左侧添满0。

12.2 位运算举例

【例12.1】 从一个整数a中把从右端开始的4~7位取出来。

可以这样考虑:

① a右移4位,见图12.3。图12.3(a)是未右移时的情况,(b)图是右移4位后的情况。目的是使要取出的那几位移到最右端。右移到右端可以用下面方法实现:

```
a>>4
```

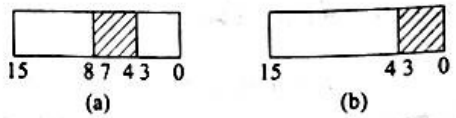

图 12.3

② 设置一个低4位全为1,其余全为0的数。可用下面方法实现:

```
~(~0<<4)
```

~0的全部二进制位为1,左移4位,这样右端低4位为0,见下面所示:

```
0:              0 0 0 0…0 0 0 0 0 0
~0:             1 1 1 1…1 1 1 1 1 1
~0<<4:          1 1 1 1…1 1 0 0 0 0
~(~0<<4):       0 0 0 0…0 0 1 1 1 1
```

③ 将上面①、②进行 & 运算。即

(a>>4)&~(~0<<4)

根据 2.1.1 节介绍的方法，与低 4 位为 1 的数进行 & 运算，就能将这 4 位保留下来。

程序如下：

```
#include <stdio.h>
int main()
  {unsigned a,b,c,d;
   printf("please enter a:");
   scanf("%o",&a);
   b=a>>4;
   c=~(~0<<4);
   d=b&c;
   printf("%o,%d\n%o,%d\n",a,a,d,d);
   return 0;
  }
```

运行结果：

```
please enter a:331
331,217
15,13
```

输入 a 的值为八进制数 331，即十进制数 217，其二进制形式为 11011001，经运算最后得到的 d 为 00001101，即八进制数 15，十进制数 13。

可以任意指定从右面第 m 位开始取其右面 n 位。只须将程序中的“b=a>>4”改成“b=a>>(m-n+1)”以及将“c=~(~0<<4)”改成“c=~(~0<<n)”即可。

【例 12.2】 循环移位。要求将 a 进行右循环移位，见图 12.4。图 12.4 表示将 a 右循环移 n 位，即将 a 中原来左面 16-n 位右移 n 位，原来右端 n 位移到最左面 n 位。今假设用两个字节存放一个短整数(short int 型)。

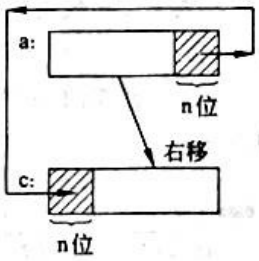

图　12.4

解题思路：

为实现以上目的可以用以下步骤：

① 将 a 的右端 n 位先放到 b 中的高 n 位中，可以用下面语句实现：

```
b=a<<(16-n);
```

② 将 a 右移 n 位，其左面高位 n 位补 0，可以用下面语句实现：

```
c=a>>n;
```

③ 将 c 与 b 进行按位或运算，即

c=c|b;

程序如下:

```
#include <stdio.h>
int main()
  {unsigned a,b,c,
   int n;
   printf("please enter a & n"\n);
   scanf("a=%o,n=%d",&a,&n);
   b=a<<(16-n);
   c=a>>n;
   c=c|b;
   printf("a:\nc:",a,c);
   return 0;
  }
```

运行结果:

```
please enter a & n:
a=157653,n=3
a:157653
c:75765
```

运行开始时输入八进制数 157653,即二进制数 1101111110101011,循环右移 3 位后得二进制数 0111101111110101,即八进制数 75765。

同样可以进行左循环位移。

12.3 位 段

以前曾介绍过对内存中信息的存取一般以字节为单位。实际上,有时存储一个信息不必用一个或多个字节,例如,"真"或"假"用 0 或 1 表示,只需 1 个二进位即可。在计算机用于过程控制、参数检测或数据通信领域时,控制信息往往只占一个字节中的一个或几个二进制位,常常在一个字节中放几个信息。

那么,怎样向一个字节中的一个或几个二进制位赋值和改变它的值呢?可以用以下两种方法。

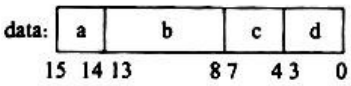

图 12.5

(1) 人为地将一个整型变量 data 分为几段。例如,a,b,c,d 分别占 2 位、6 位、4 位、4 位(见图 12.5)。如果想将 c 段的值变为 12(设 c 原来为 0)。

可以这样:

① 将整数 12 左移 4 位(执行 12<<4),使 1100 成为右面起第 4~7 位。

② 将 data 与"12<<4"进行"按位或"运算,即可使 c 的值变成 12。

如果 c 的原值不为 0,应先使之为 0。可以用下面方法:

data=data & 0177417 (0177417 的最左边的 0 表示 177417 是八进制数)

(177417)$_8$ 的二进制表示为

$$\underbrace{11}_{a}\ \underbrace{111111}_{b}\ \underbrace{0000}_{c}\ \underbrace{1111}_{d}$$

也就是使第 4～7 位全为 0，其他位全为 1。它与 data 进行 & 运算，使第 4～7 位为 0，其余各位保留 data 的原状。

这个 177417 称为“屏蔽字”，即把 c 以外的信息屏蔽起来，不受影响，只使 c 改变为 0。但要找出和记住 177417 这个数比较麻烦。可以用

```
data=data &~(15<<4);
```

15 是 c 的最大值(c 共占 4 位，最大值为 1111，即 15)。15<<4 是将 1111 左移到以右侧开始 4～7 位，即 c 段的位置，再取反，就使 4～7 位变成 0，其余位全是 1，以上可以示意为

```
15:              0000000000001111
15<<4:           0000000011110000
~(15<<4):        1111111100001111
```

这样可以实现对 c 清零，而不必计算屏蔽码。

将上面几步结合起来，可以得到

```
data=data &~(15<<4)|(n&15)<<4;
     (赋给 4～7 位，使之为 0)
```

n 是应赋给 c 的值(例如 12)。n & 15 的作用是只取 n 的右端 4 位的值，其余各位置 0，即把 n 放到最后 4 位上，(n & 15) << 4 就是将 n 置在 4～7 位上，见下面：

```
data & ~(15<<4):   11011011|0000|1010
(n & 15)<<4:       00000000|1100|0000
(按位或运算)       11011011|1100|1010
```

可见，data 的其他位保留原状未改变，而第 4～7 位改变为 12(即 1100)了。

但是用以上方法给一个字节中某几位赋值太麻烦了。可以用下面介绍的位段结构体的方法。

(2) 使用位段

C 语言允许在一个结构体中以位为单位来指定其成员所占内存长度，这种以位为单位的成员称为“位段”或称“位域”(bit field)。利用位段能够用较少的位数存储数据。例如：

```
struct Packed_data
  {unsigned a:2;
   unsigned b:2;
   unsigned c:2;
   unsigned d:2;
   short i;
  }data;
```

见图 12.6。其中 a,b,c,d 段分别占 2 位、6 位、4 位、4 位，i 为 short 型，以上共占 4 个字节。也可以使各个位段不恰好占满一个字节。例如：

```
struct Packed_data
  {unsigned a:2;
```

```
    unsigned b:3;
    unsigned c:4;
    short i;
  };
struct Packed_data dats;
```

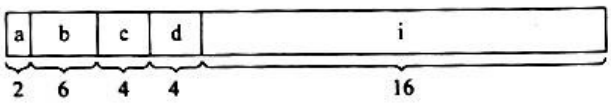

图 12.6

见图 12.7。

图 12.7

其中 a,b,c 共占 9 位,占 1 个字节多,不到 2 个字节,它的后面为 short 型,占两个字节。在 a,b,c 之后 7 位空间闲置不用,i 从另一字节开头起存放。

注意:位段的空间分配方向因机器而异。一般是由右到左进行分配的,如图 12.8 所示。但用户可以不必过问这种细节。

可以直接对位段进行操作。例如可以直接对位段赋值:

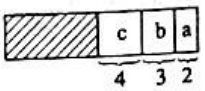

图 12.8

```
data.a=2;
data.b=7;
data.c=9;
```

请注意位段允许的最大值范围。如果写成

```
data.a=8;
```

就错了。因为 data.a 只占两位,最大值为 3。在此情况下,系统会自动取赋予它的数的低位。例如,8 的二进制数形式为 1000,而 data.a 只有 2 位,取 1000 的低 2 位,故 data.a 得值 0。

关于位段的定义和引用,有几点要说明:

(1) 声明位段的一般格式为

类型名 [成员名]:宽度

位段成员的类型可以指定为 unsigned int 或 int 型。"宽度"应是一个整型常量表达式,其值应是非负的,且必须小于或等于指令类型的位长。

(2) 对位段组(例如上面的结构体变量 data 在内存中存放时,至少占一个存储单元(即一个机器字,4 个字节),即使实际长度只占一个字节,但也分配 4 个字节。如果想指定某一位段从下一个存储单元(字)存放,可以用以下形式定义:

```
unsigned a: 1;
unsigned b: 2;
```

(一个存储单元)

```
unsigned:  0;      (表示本存储单元不再存放数据)
unsigned c: 3;     (另一存储单元)
```

本来 a,b,c 应连续存放在一个存储单元(字)中,由于用了长度为 0 的位段,其作用是使下一个位段从下一个存储单元开始存放。因此,现在只将 a 和 b 存储在一个存储单元中,c 另存放在下一个单元(上述"存储单元"可能是一个字节,也可能是两个字节,视不同的编译系统而异)。

(3) 一个位段必须存储在同一存储单元中,不能跨两个单元。如果第 1 个单元空间不能容纳下一个位段,则该空间不用,而从下一个单元起存放该位段。

(4) 可以定义无名位段。例如:

```
unsigned a : 1;
unsigned   : 2;(这两位空间不用)
unsigned b : 3;
unsigned c : 4;
```

见图 12.9。在 a 后面的是无名位段,该空间不用。

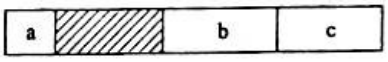

图　12.9

(5) 位段的长度不能大于存储单元的长度,也不能定义位段数组。

(6) 位段中的数可以用整型格式符输出。例如:

```
printf("%d,%d,%d",data.a,data.b,data.c);
```

当然,也可以用%u、%o、%x 等格式符输出。

(7) 位段可以在数值表达式中引用,它会被系统自动地转换成整型数。例如:

```
data.a+5/data.b
```

是合法的。

在本章中简要地介绍了有关位运算的知识,读者可以从中了解为什么说 C 语言是贴近机器的语言,以及 C 语言是怎样对二进位操作的。这些知识对有些读者是很需要的。