# 逻辑移位与算术移位

## 1. 先从段子看起

话说有这样的一段代码：

```
1.  #include <stdlib.h>
2.  #include <stdio.h>
3.
4.
5.  static void divide_by_two(int num)
6.  {
7.      while (num) {
8.          printf("%d\n", num);
9.          num /= 2;
10.     }
11. }
12.
13. int main()
14. {
15.     int num;
16.     scanf("%d", &num);
17.
18.     divide_by_two(num);
19.
20.     return 0;
21. }
```

某天，一个刚毕业的朋友——没有贬低应届生的意思哦~~~，开始负责维护这段代码。他呢，看到 num /= 2，想起课程上讲过，整数右移一位，就等于除于二，并且右移操作比除法运算要高效的多。于是将 num /=2 改为了 num = num>>1。

代码变为

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


static void divide_by_two(int num)
{
    while (num) {
        printf("%d\n", num);
        num = num>>1;
    }
}

int main()
{
    int num;
    scanf("%d", &num);

    divide_by_two(num);

    return 0;
}
```

编译成功后，当然不能忘了测试：

```
[xxx@xxx-vm-fc13 test]$ ./a.out
10
10
5
2
1
[xxx@xxx-vm-fc13 test]$ ./a.out
3
3
1
```

这位朋友对于结果很满意，于是将改动提交到了服务器，下班回家~~~

结果第二天就有同事来找他，说他的程序陷入了死循环：

```
[xxx@xxx-vm-fc13 test]$ ./a.out
-5
-5
-2
-1
-1
-1
……..
```

这位朋友刚参加工作，第一次提交改动，就造成了这样的结果，自然很紧张。不是右移一位就等于除以 2 吗？究竟是怎么回事呢？虽然暂时不知道答案，也只能将改动 rollback 回去。

那么到底是什么原因呢？没错，右移一位就等于除以 2，但是这里需要加一个条件，这里指的是正数。而对于有符号整数，且其值为负数时，在 C99 标准中对于其右移操作的结果的规定是 implementation-defined.

在 Linux 上的 GCC 实现中，有符号数的右移操作的实现为使用符号位作为补充位。因此-1 的右移操作仍然为 0xFFFFFFFF。这导致了死循环。

## 2. 符号位

计算机根据不同的需求把数据类型分为有符号数和无符号数。即 signed 和 unsigned。signed 和 unsigned 的区别在于数据是否包含符号位。

在 signed 类型数据中，数据的最高位是符号位，对于 signed short 类型的数： 范围时-32768~+32767 ；最高位是 0 时，数据是正数，最高位是 1 时，数据是负数。

在 unsigned 类型中，数据无符号位,所有 bit 都是数据位，对于 unsigned short 类型的数，范围是 0~65535。

## 3. 补码

计算机里，数据以补码的形式保存。

正数的补码是其本身，在无符号（unsigned）数中，补码都是器本身（无符号数都是正数）。在有符号数（signed）的正数部分中，补码也是其本身，由于最高位的符号位也是 0,所以和无符号数情况一致。

负数的补码计算规则：对应正数的反码加一。比如：-35 。

正数 35 的二进制码 ： 00100011 ；

正数 35 的二进制反码： 11011100 ；

正数 35 的反码加一 ： 11011101 ；

即 -35 的补码为： 11011101 ；

-1 的补码：11111111 ；

-32768 的补码为 1000 0000 0000 0000 。

# 4. 算术右移

算术右移时，符号位不变，数据位（注意是数据位，不包括符号位）向右移动一位，数据位最左边空出的一位补充符号位的数据。

如： - 23735 ，

补码为： 1010001101001001

符号位不变，数据位向右移动一位后： 1_01000110100100

数据位最左边空出的一位补充符号位的数据后 1101000110100100

化成十进制数为 -11868 。

右移相当于原来的数据除以 2，这一条在大多数情况下都是对的。

但对于-1 ： 11111111，算术右移之后还是-1 。开篇的段子就是这个原因。

对于正数，算术右移时，最左边空出位补充的是 0（符号位是 0），所以和逻辑右移一致。

# 5. 逻辑右移

逻辑右移时，不区分数据位和符号位。所有的 bit 都参与移位，最高位补充 0 。

对于 -23735 ：1010001101001001

右移后为 ：0101000110100100

移位后值为 ：20900

# 6. 算术左移和逻辑左移

算术左移和逻辑左移都是一样的，不区分数据位和符号位。所有 bit 都参与移位。最低位补充 0 。

如 -23735 ：1010001101001001

左移后 ：0100011010010010

值为：20900

对于无符号数： 41801 ：1010001101001001

左移后 ：0100011010010010

值为：20900

上例可以看出，对负数左移时，当符号为溢出时，数据可能变成正数。

下面是具体 VS2010C++的程序运行结果：

```
cout<<"算术移位"<<endl ;
short i=-23735 ;

cout<<"十进制数 : "<<dec <<i <<endl ;
cout<<"二进制数 : "<<bitset<sizeof(short)*8>(i)<<endl;

i = i<<1;
cout<<"左移一位 : "<<bitset<sizeof(short)*8>(i)<<endl;
cout<<"十进制数 : "<<dec <<i <<endl ;
i=-23735 ;
i = i>>1;
cout<<"右移一位 : "<<bitset<sizeof(short)*8>(i)<<endl;
cout<<"十进制数 : "<<dec <<i <<endl ;

cout<<endl;
cout<<"逻辑移位"<<endl ;
unsigned short k= 41801 ;
cout<<"十进制数 : "<<dec <<k <<endl ;
cout<<"二进制数 : "<<bitset<sizeof(unsigned short)*8>(k)<<endl;

k = k<<1;
cout<<"左移一位 : "<<bitset<sizeof(unsigned short)*8>(k)<<endl;
cout<<"十进制数 : "<<dec <<k <<endl ;
k=41801 ;
k = k>>1;
cout<<"右移一位 : "<<bitset<sizeof(unsigned short)*8>(k)<<endl;
cout<<"十进制数 : "<<dec <<k <<endl ;
```