

左移只需要将全部数据往左移动对应位数，并将高位舍去，低位补0

对于右移，有俩种情况：

1~逻辑右移

将现有的数据往右移，并在高位补0

2~算术右移

将现有的数据往右移，并在高位处补符号位

这俩种右移情况，对于符号为为0的情况均相同，如第一个情况

然鹅对于符号位为1的情况（即数字为-的情况）就会出现错误了

如图

对于逻辑右移，主要用于无符号整数，因为无符号整数没有符号位

对于算术右移，主要用于有符号整数

有符号整数主要使用二进制补码表示，至于补码，正数补码为它本身，且符号位又为0，此时能正确表达，对于负数补码为原码取反加一

比如：

补码：   
 +5 0000 0101

-5 1111 1011

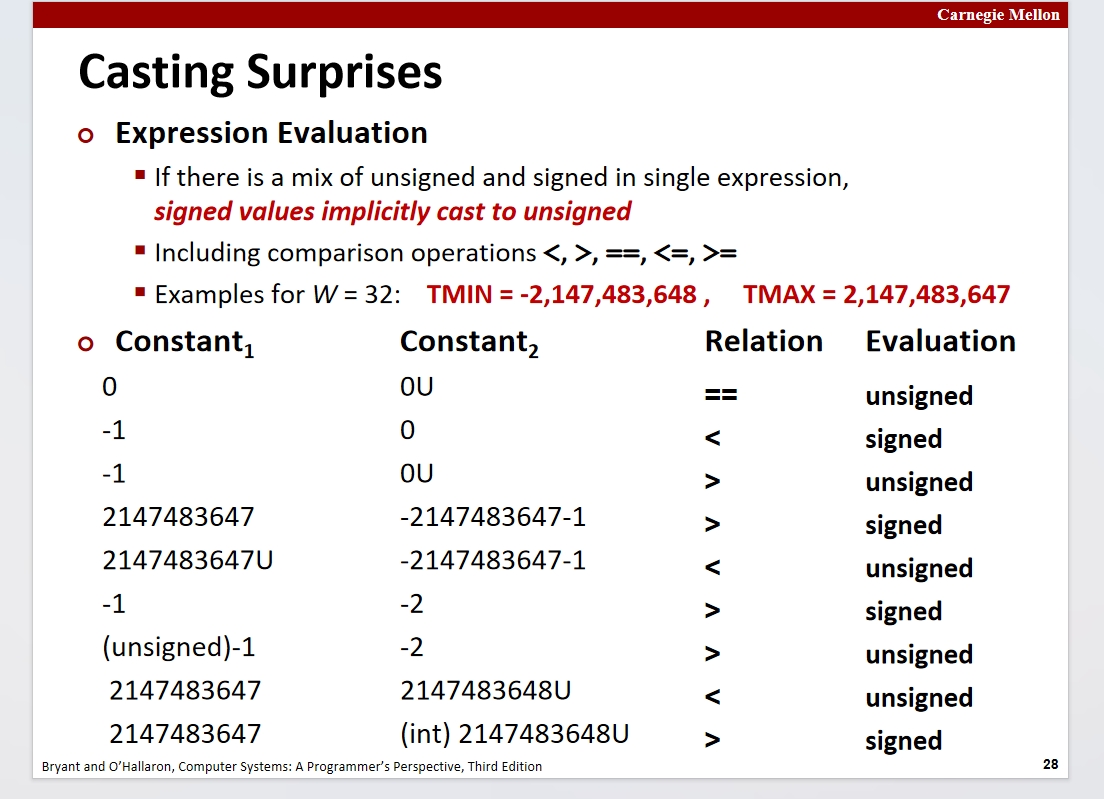
对这俩进行算术右移>>2

+5 0000 0001 ~ 1

-5 1111 1110 ~ -2

补充：

左移右移这个位运算可以视作为高数的乘除运算，右移可视为将数除以2的右移幂数（前提是能正常得出结果）同时左移可视为乘以2的左移幂数。效率高，用于优化



对于C中整形数据的表示，本质上都是使用补码来进行标识的

在这里，我们用U来修饰无符号数，没有U表示是有符号数

当我们看前俩个例子时，这能够得到正确的结果，但是当第三个时得到了与逻辑相违背的结果，而这个，涉及到了C中的类型转换，C中默认的是将有符号整形转换为无符号整形

对于-1>0U的解释:

1~第一点需要特别注意的是，在c程序中（其他应该也是），所有整型数据都是通过补码表示的，有无符号的区别就是为程序解释时进行解释方法的区别，有符号的程序在解释这一串数据是会将第一位解释为符号位，无符号则不会

2~好，第一点就已经解释了这里面的核心了，就是这几点：

1~程序在内存中用补码表示整形数据

2~程序在内存中对整形数据的强制类型装换并不是对该数据对应的内存中的数据进行 修改，它修改的知识对于这一串数据的访问方式

Eg:

(有符号)-1: 11111111 11111111 11111111 11111111 // 二进制补码表示的 -1

当将-1与0U进行比较时，会对-1进行类型装换，而这个过程并不会对-1所对应的这一串二进制数进行修改，修改的只是对于这一串数据的解释程序，会将本来的二进制有符号数解释程序修改为二进制无符号数的解释程序。这样的操作会导致我们上面的逻辑错误。

为什么？程序对于有符号数的识别，存在符号位，对于无符号数的识别则不存在这一位，这就意味在强制类型转换中对于有符号数中的负数，其符号位1会被解释，这就会导致奇诡的结果，有符号数的负数会变成无符号数中的大数。如下

(unsigned int)a = 11111111 11111111 11111111 11111111 // 二进制仍然是全1

此时程序对于a的解释为2^32-1，即4294967295

这样就会导致-1>0U的结果，而这是我们不希望看到的，不利于维护程序的健壮性

再举一个例子 2147483647 2147483648U

程序中的二进制补码表示：

2147483647: 01111111 11111111 11111111 11111111 // 最高位为符号位，0表示正数

2147483648U:10000000 00000000 00000000 00000000 // 无符号，最高位是数值位

进行隐式类型装换对有符号数进行重新解释

01111111 11111111 11111111 11111111：2147483647

它数值并没有发生变化(因为它符号位是0)，但是底层的一个解释程序已经在这个过程发生了变化（不持续）

故其转换后比较的结果为2147483647 < 2147483648U

这与逻辑相符，但是底层上进行了改变，故在使用有符号数无无符号数进行比较时要十分注意

for(int i=n-1;i>=0;i--);(n>1)

这样的程序总是能正确运行，但运行多少次呢

那么如果将int 换为unsigned int 呢

for(unsigned int i=n-1;i-sizeof(char)>=0;i--);呢

**一点十分需要注意的:类型转换不会改变其的二进制位**

对于TMAX和UMAX的一种转换，可以将UMAX姑且视为TMAX俩倍+1

Eg:

TMAX：0111 1111 1111 1111

2\*TMAX(左移一位补0):1111 1111 1110

+1:1111 1111 1111

此时就是UMAX的二进制表示（姑且可以看做是一种转换方法）

对于将一定位数的二进制数进行扩展的方法：

最朴素的事项就是用符号位来填充前面的位

Eg: 1010 -8+2=6 扩展到5位

用符号位进行填充后的结果 11010：-16+8+2=-6

可以看到结果是一样的，这是因为往前填充符号位会让其多间一个负数，但同时会让原本的最高位负数变为整数，计算这俩者间的差值，会发现其其实是原本最高位的值，这里好推，不给出证明，可以用数学方法

对于将一定位数的数字截断为更短数字的方法呢

**非常不建议这么做**

需要注意，程序中底层对截断的处理就是简单的截断，位数从末位进行计算

当我们截断一个无符号数时，我们可以通过对这个无符号数对应的二进制补码进行运算来计算截断后的结果，比如

70000

**00000000 00000001 00010001 01100000**

截断为16位

**00010001 01100000 （低 16 位）**

转换后的值为4464

而程序底层对截断后的结果进行度结果会是4448

在这里，4464是更符合逻辑的结果，其更符合无符号整数在不同位宽下的处理逻辑，（这个取模运算可以用来识别一个数在不同位宽下的映射），但是程序直接截断是不符合这个数的（一般都是，小数也有的会符合），因此对于用户来说，应避免使用类型转换来进行截断，这样给出的结果一般都是不符合逻辑的