C++ 位运算：

|  |  |  |  |
| --- | --- | --- | --- |
| **位运算操作符** | **对应的位运算** | **用法** | **功能描述** |
| ~ | 按位非 | ~expr | 翻转expr的每一个位：1变0，0变1 |
| << | 左移 | expr<<n | 将expr向左移动n位，移到外面的被丢弃，右边的位补0，因此左移n位相当于乘以2n |
| >> | 右移 | expr>>n | 将expr向右移n位，移到外面的被丢弃，如果expr是无符号类型，则左边补0，否则，左边插入符号位的拷贝或者0(视具体实现而定) |
| & | 按位与 | expr1&expr2 | 在每个位所在处，如果expr1和expr2都含有1，那么结果该位为1，否则为0 |
| | | 按位或 | Expr1 | expr2 | 在每个位所在处，如果expr1和expr2都含有0，那么结果该位为0，否则为1 |
| ^ | 按位异或 | Expr1 ^ expr2 | 在每个位所在处，如果expr1和expr2不相同，那么结果该位为1，否则为0 |

除了上面的基本位运算操作符外，还有&=，^=，|=，<<=，>>=等组合符号，它们分别是：按位与赋值，按位异或赋值，按位或赋值，左移赋值，右移赋值。

常用位操作：

1.将expr的第n（n从0开始）位设置为1： expr |= (1<<n);

2.将expr的第n（n从0开始）位设置为0： expr &= (~(1<<n));

3.判断expr的第n（n从0开始）位是否为1： bool b =expr & (1<<n);

4.翻转expr的第n（n从0开始）位： expr ^=(1<<n)；

5．    用法：掩码

如只显示第二、三位

107 = 0110 1011

6     = 0000 0110

&

2   = 0000 0010

6．    用法：打开位

如只打开第二、三位

107 = 0110 1011

6    = 0000 0110

|

111 = 0110 1111

7．    用法：关闭位

如关闭第二、三位

107 = 0110 1011

~6   = 1111 1001

&

105 = 0110 1001

8．    用法：转置位

如转置第二、三位

107 = 0110 1011

6 = 0000 0110

^

105 = 0110 1101

C++中的bitset容器

1.头文件：

  #include <bitset>

2.声明一个容器：

bitset<int> bits; (所有位设为0)

bitset<n> bits(unsigned long);

bitdet<int> bits(string&)

3.bitset的基本用法：

|  |  |  |
| --- | --- | --- |
| 操作 | 功能 | 用法 |
| test(pos) | pos位是否为1? | a.test(4) |
| any() | 任意位是否为1? | a.any() |
| all() | 是否全部位都为1? | a.all() |
| none() | 是否没有位为1? | a.none() |
| count() | 值是1的位的小数 | count() |
| size() | 位元素的个数 | size() |
| [pos] | 访问pos位 | a[4] |
| flip() | 翻转所有位 | a.flip() |
| flip(pos) | 翻转pos位 | a.flip(4) |
| set() | 将所有位置1 | a.set() |
| set(pos) | 将pos位置1 | a.set(4) |
| reset() | 将所有位置0 | a.reset() |
| reset(pos) | 将pos位置0 | a.reset(4) |

4.bitset与传统C位操作及字符串的转换

    unsigned long bits = bits.to\_ulong();

  sting str(bits.to\_string());

位运算只能用于操作有整数类型的数，比如说char,short,int,long等(包含signed 和unsigned)，不能操作浮点数，比如float，double！std：：bitset的构造函数的参数是unsigned long int，尽量不要对负数进行为操作，因为可移植性差，不同的系统平台对负数的右移操作定义不一样（大多数平台规定高位补符号位，有些平台规定高位补0）。

用法：

**1.不用任何中间变量，交换两个整数**

这个问题是比较经典的了，你可以很容易地在网上找到多种答案，我在这里给出两个方案：

方法1：该方案的思路简单，实现代码很短，如下：

1. Template<**class** T>
2. Void mySwap\_1(T& a,T& b)
3. {
4. a=a+b;
5. b=a-b;
6. a=a-b;
7. }

简单吧，但是我还要简单说一下：第一句a=a+b是用a保存原来的a跟原的b的和；第二句b = a-b使得原来的a的值被保存到了b里面；最后一句a=a-b使得原来的b的值保存到了a里面。

我们说这个方法是不那么完美的，原因在于算术运算可能会出现结果溢出的问题，假如a，b都非常大，那么第一句a=a+b就会导致结果溢出，比如说原来的a = 2147483647,b = 2,那么a+b就为2147483649，这个数大于了最大的无符号整数2147483648，因此发生溢出，a中保存的结果实际上是：-2147483647，**但是让人惊讶的是：虽然第一句程序得到的结果为-2147483647，后面两句得到的结果却是正确的，即能实现交换原始a,b的值，也就是说：只有第一句的结果是错误的，但最后的结果却是正确的，这一点让我很迷惑，至今还没弄清楚缘由，再次向各位求教！**

最后，谈谈这种方法相对于后面的方案2的优点：该方法可以用于交换两个非整数(浮点数)，而方案2基于位运算，而对浮点数不能直接使用位运算，因此方案2不能用于交换两个浮点数！

方法2：该方案代码与方案1及其相似，思路也不难，先看代码，然后再看我啰嗦的剖析：

1. **template**<**class** T>
2. **void** mySwap\_2(T& a,T& b)
3. {
4. a=a^b;
5. b=b^a;
6. a=a^b;
7. }

对于编程老手来说，这个交换函数并不陌生，但我相信这些编程老手之中有一部分人只记得这么写代码，而不知道三句代码为何这么写，事实上我最初也是这样，因此一开始我就觉得短短3行代码，让我花费时间去理解分析，还不如直接记忆来得划算。事实上，直到今天我写这篇文章时，我舍得消耗一点脑细胞来理解它，下面我尝试着对上述三句代码进行阐述，为了方便，假设数据类型为char，并且a = 5，b=3，那么在内存中a,b存储如下：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a: | 0 | 0 | 0 | 0 | 0 | **1** | **0** | **1** |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| b: | 0 | 0 | 0 | 0 | 0 | **0** | **1** | **1** |

**接下来详细分析每一句：**

首先来看第一句：a=a^b，执行该语句后**a中保存了a与b的差异位**，也就是说如果原来的a和b的某一位不同，那么就将a的该位置为1，因此a在内存中成了如下图的样子，它说明a与b的第2,3个bit有差异：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a: | 0 | 0 | 0 | 0 | 0 | **1** | **1** | 0 |

接着我们来看第二句：b=b^a，其意思是，将b中有差异的位**翻转**，如此一来b中保存的值其实就等于原来a中的值，记住当第二个语句执行完之后a仍然保存了原来的a,b的差异信息，而b则变成了原来的a！

最后我们来看第三句：a=a^b，由于异或运算满足交换律，因此这一句等价于：a=b^a，记住这个语句赋值号右边的b中已经保存了原始的a值，而a中保存了原始的a，b的差异，因此这一句的最终作用是将原始a中有差异的位翻转(变成b)然后赋值给a，如此一来a中就保存了原始的b值。

总结：上述三句中：第一句是记录差异，第2,3句是翻转，最终实现了不用任何中间变量就交换两个变量的值。

**分析：位运算不考虑进位问题，因此不会有结果溢出的问题！但是由于不能对浮点数进行直接位运算，因此该方法不能实现交换两个浮点数！当然原题题目是交换两个整数。**

**备注：还有其他实现两个数交换的方法，比如采用内存拷贝！由于不属于位运算范畴，这里就不赘述了。**

**2.进制转换**

要求：分别实现十进制整数按二进制、十六进制输出。

**两种方法实现按二进制输出：**

方法1：由于整数在计算机中是按二进制存储的，我们只需要将其每个bit按顺序打印出来即可，如果某位为1，则打印字符‘1’，否则打印字符‘0’。我给出的代码如下：

1. **void** printBinary(**int** num)
2. {
3. **for**(**int** i=31;i>=0;i--)
4. {
5. cout<<((num>>i)&1);
6. }
7. }

这个函数的思路很简单，就是从高到底逐位打印每个bit。我上面的代码有一点不好的地方，那就是语句太复杂，一个cout语句干了太多的事情，如果影响您的理解，那么你可以增加几个临时变量，然后把它拆分成多个简单语句。我这么写主要是考虑到篇幅的原因，因此程序段太占篇幅了。随便说一句，编程时，语句力求简单明了：一行只写一条语句，一条语句只干一件事情！

方法2：bitset是标准库提供的一个类(不是容器)，利用它就可以很方便地操作位，下面是用bitset来实现的程序：

1. **void** printBinary(**int** num)
2. {
3. bitset<32> bits=bitset<32>((unsigned **long**)(num));
4. **for**(**int** i=31;i>=0;i--)
5. {
6. cout<<(bits[i]==**true**? '1':'0');
7. }
8. }

备注：关于bitset重载了多个运算符，其中包含下标运算符：[]，可以方便地取得某一个bit，看它是否为1。关于bitset的更多信息请查阅msdn或者其他资料，你只要记住bitset是标准库提供的，你可以随时使用，不要忘记添加相应的头文件。

**实现按16进制输出：**

同样由于数据在内存中是按二进制存储的，因此将整数按照16进制输出我们可以如下做：从左向右，每4位bit一组，组合成一个十六进制数，一次输出即可，其程序如下：

1. **void** printHex(**int** num)
2. {
3. **for**(**int** i=28;i>=0;i-=4)
4. {
5. **int** temp=num>>i;
6. temp=temp&15;
7. **char** ch;
8. temp>9?(ch='A'+temp-10):(ch='0'+temp);
9. cout<<ch;
10. }
11. }

该程序与上面的printBinary函数非常相似，要注意的是i每次变化4，最关键点在于语句temp= temp&15；由于是16进制，因此这里用15做掩码。我想有了printBinary做铺垫，理解这个printHex并不难，这里不赘述了。接下来我将对这两个函数进行个小小的扩展：实现整数按2n (2的n次方)进制输出！比如按8进制，32进制等。为了方便描述，我们限制1<=n<=6；并用字符’0’到’9’表示数字0到9，用字符A,B,……Z,a,b,……表示数字10到63。程序如下：

1. **void** print2powerN(**int** num,**int** N)
2. {
3. **for**(inti=32-N;i>=0;i-=N)
4. {
5. **int** temp=num>>i;
6. temp=temp&((1<<N)-1);
7. **char** ch;
8. **if**(temp<=9)
9. {
10. ch='0'+temp;
11. }
12. **else** **if**(temp<=35)
13. {
14. ch='A'+temp-10;
15. }
16. **else**
17. {
18. ch='a'+temp-36;
19. }
20. cout<<ch;
21. }
22. }

备注：用位运算也能实现十进制到任意进制的转换，这个问题比较难，我暂时还没弄透彻！

**3.求整数的二进制表示中1的个数**

**问题描述：**输入一个整数N要求输出其二进制表示中1的个数M，比如N=13，则M=3；

**分析：**该问题的求解方法不止一种，可以对二进制表示的每一位逐位扫描来实现，这种方法的复杂度是o(n)其中n是N的二进制表示的总位数。这里介绍如何用位操作来求解，并且保证其复杂度低于o(n)，事实上该方法的复杂度为o(m)，其中m是N的二进制标识中1的个数！

**思路：**在讲述具体实现时，来看这样一个事实：**n&(n-1)能实现将最低位的1翻转**！比如说n=108，其二进制表示为01101100,则n&(n-1)的结果是01101000。因此只要不停地翻转n的二进制的最低位的1，每翻转一次让计数器+1，直到n等于0时，计数器中就记录了n的二进制中1的位数，程序如下：

1. **int** countBits(**long** n)
2. {
3. **int** count=0;
4. **while**(n)
5. {
6. count++;
7. n&=(n-1);
8. }
9. printf("%d",count);
10. }

**4.将最右侧的1翻转成0：x&(x-1)**

假设x=01110100，那么x，x-1和x&(x-1)的二进制表示分别如下：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x: | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| x-1: | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| X&(x-1): | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

可以看出，x-1的功能是：将最右侧的1翻转了，并让其后的0变成了1，而保持了其他位不变。然后x&(x-1)就使得x的最右侧的1翻转成0。

该位运算在上一篇中被用来计算二进制中有多少个1，其思路是：不停的翻转右侧的1，知道将该数翻转成0为止。

**5向右连续传播最右侧的1位：x|(x-1)**

比如说x=0010**1**000，那么x，x-1和x|(x-1)的二进制表示分别如下：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x: | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| x-1: | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| X|(x-1): | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |

从上面的过程可以看出x|(x-1)使得x中最右边的1的右边的0都被1填充了，**注意这不能理解成：翻转最右侧的连续的0，而应该理解成用最右侧的1填充了它右边的所有0位。或许你要问为什么，那么你假设x =00101001，那么x|(x-1)仍然等于00101001，它并没有将最右侧的两个0翻转！**

http://www.jb51.net/article/41461.htm