Bitwise operations in C: Part 1

Bitwise Operation: AND, OR, XOR, NOT

In computer, all the numbers and all the other data are stored using 2 based number system or in binary format. So, what we use to say a '5' in decimal (do we use decimal only because we have 10 figures? who knows...), a computer will represent it as '101', in fact, everything is represented as some sequence of 0s and 1s. We call this sequence a bit-string.

Bit-wise operations means working with the individual bits other than using the larger or default data types, like integers, floating points, characters, or some other complex types. C/C++ provides a programmer an efficient way to manipulate individual bits of a data with the help of commonly known logical operators like AND(&), OR(|), NOT(~), XOR(^), LEFT SHIFT(<<) and RIGHT SHIFT(>>) operators.

To be fluent with bit-wise operations, one needs to have a clear concepts about how data is stored in computer and what is a binary number system.

From a programming contest aspect, we'll explore bit-wise operations for only integer types (int in C/C++) and as this is almost 2010, we'll consider int as a 32 bit data type. But for the ease of writing, sometimes only the lowest 16 bits will be shown. The above operators works on bit-by-bit, i.e. during bit-wise operation, we have to align right the bits! (just like addition, multiplication,...) The shorter bits always have leading 0s at the front. And another important thing... we do not need to convert the integers to the binary form ourselves, when we use the operators, they will be automatically evaluated and the following examples shows the underlying activities... :P

& (AND) operator

This table will clearly explains the bit-wise operator & (AND):

```
0                   &                   0                   =                   0
0                   &                   1                   =                   0
1                   &                   0                   =                   0
1 & 1 = 1
```

Example:
What is the value of 24052 & 4978 ?
To solve this, we have to consider the operations in base 2.
So, we imagine both numbers in base 2 and align right the bits, as shown below using 16 bits. Now, AND the numbers : (shorter bits can have leading zeros at the front):

```
101110111110100                         ⇒                                   24052
001001101110010                          ⇒                                   4978
---------------                                                              &
001000101110000 ⇒ 4464
```

Each bit-column is AND-ed using the AND table above.

| (OR) operator

This table summarizes the OR operations :

```
0                   |                   0                   =                   0
0                   |                   1                   =                   1
1                   |                   0                   =                   1
1 | 1 = 1
```

Now, using the same example above, let's do it using | (OR) operator :

```
101110111110100                             ⇒                           24052
001001101110010                              ⇒                           4978
---------------                                                            |
101111111110110 ⇒ 24566
```

Each bit-column is OR-ed using the OR table above.

^ (XOR) operator

This table summarize the XOR operations :

```
0                   ^                   0                   =                   0
0                   ^                   1                   =                   1
1                   ^                   0                   =                   1
1 ^ 1 = 0
```

Now, using the same example above, let's do using ^ (XOR) operator :

```
101110111110100                             ⇒                           24052
001001101110010                              ⇒                           4978
---------------                                                            ^
100111010000110 ⇒ 20102
```

Easy, isn't it?

~ (NOT) operator

This operator is different from the above operators. This operator is called **unary** operator, since it only needs one operand. The operators &, |, ^ are **binary** operators, since it takes 2 operands/number to be operated.

This ~ (NOT) operator, inverts all the bits in a variable. That is, changes all zeros into ones, and changes all ones into zeros.
Remember! that the result of this ~ (NOT) operation highly depends on the **length** of the bit-string.

Example:

```
int                         a                       =                       10;
printf("%d\n", ~a);
```

Surprisingly, the output is -11. But actually this is normal as most of the cases computer represents negative numbers in the 2's complement form. Look at the operation shown below using 16 bits:

```
0000000000001010                    ⇒                   a(16                    bits)
```

```
----------------                                                    ~
1111111111110101 ⇒ -11
```

This is correct! Because computer stores -11 as 1111111111110101 in binary! (in the 2's complement form). Even if we use 32 bits representation, it is still the 2's complement form of -11;

```
-11(10) = 11111111111111111111111111110101(2)
```

Anyway, if we do the same operation for unsigned int:

```
unsigned int a = 10;
printf("%u\n", ~a);
```

Hah ha, don't be surprised if you get the output of the unsigned value of -11 is 4294967285.

If you use 32 bits, you actually will get -11 as 11111111111111111111111111110101 in signed binary representation:

```
00000000000000000000000000001010 ⇒ 10 (32 bits)
--------------------------------                                    ~
11111111111111111111111111110101 ⇒ -11 (for signed) and -> 4294967285 (for unsigned)
```

In the unsigned data type, all the bits are considered to be the part of the magnitude, there's no bit position reserved for sign bits.

Bitwise operations in C: Part 2

Bitwise Operation: LEFT SHIFT and RIGHT SHIFT

Back to Part 1

<< (SHIFT LEFT) operator

This operator also depends highly on the length of the bit-string. Actually this operator only "shift" or moves all the bits to the left. This operator is mostly used if we want to multiply a number by 2, or, some powers of 2. Example :

```
int a = 4978;
printf("%d\n", a<<1);
```

The output will be 9956. Do you see the relation between 4978 and 9956?

YES, 4978 * 2 = 9956. For more detailed explanation, we will see it in binary:

```
0001001101110010          ⇒          a          =          4978(16          bit)
----------------    <<    1    (SHIFT    LEFT    the    bits    by    one    bit)
0010011011100100 ⇒ 9956
```

Just move left all the bits by one. The left most bit is lost! and at the rightmost, a zero is added. Second example: count 4978 << 8 (count 4978 shifted left by 8 bits) Answer:

```
0001001101110010        ⇒        4978        (16        bit        representation)
----------------    <<    8    (SHIFT    LEFT    the    bits    by    8    bit)
0111001000000000 ⇒ 29184
```

So, using 16 bit compiler, 4978 << 8 is 29184, which is incorrect! Some of the left most bits are lost...It will not, however, if we use 32 bits data type.

```
00000000000000000001001101110010            ⇒            4978(32            bit)
--------------------------------  <<  8  (SHIFT  LEFT  the  bits  by  8  bit)
00000000000100110111001000000000 ⇒ 1274368
```

I've told you before that, this operator depends the length of bit-string. Don't be surprised if you got the output of 4978 << 8 is 1274368. (this is actually correct!) As we see, no bits are lost after shifting the bits left by 8 bits if we use a 32 bit data type.
Note that you don't have to have an int to store the value! In C, you can just print like this :

```
printf("%d", 4978 << 8);
```

The output will be 29184 for 16 bit compiler as Turbo C (16 bits; some bits will be lost)
The output will be 1274368 for 32 bit compiler as GNU C (32 bits; all bits are reserved since it has bigger capacity)
Now you know why shifting left 1 bit is the same as multiply the number by 2 right? And shifting left 8 bits is exactly the same as multiplying the number by 2^8.

4978 << 8 = 1274368 (in 32 bits compiler)
4978 * 2^8 = 4978 * 256 = 1274368. (exactly the same)

BTW, we need to be careful when using signed data type, as the left most bit is the sign bit. So, while left shifting, if you push a 1 at that position, the number will be negative.

>> (SHIFT RIGHT) operator

Not much different with << (SHIFT LEFT) operator. It just shifts all the bits to the right instead of shifting to the left. This operator is mostly used if we want to divide a number by 2, or, some powers of 2.

Example :

count 4978 >> 8

```
int a = 4978;
printf("%d\n", a>>8);
```

4978 >> 8 = 19 (in 16 bits compiler or 32 bits compiler, the >> has no difference, because >> will discard the right most bit, and increased capacity on the left side doesn't help here anyhow)

Detailed explanation :

```
0001001101110010 ⇒ 4978(16 bit)
---------------- >> 8 (SHIFT RIGHT the bits by 8 bit)
0000000000010011 ⇒ 19
```

and

```
00000000000000000001001101110010 ⇒ 4978(32 bit)
-------------------------------- >> 8 (SHIFT RIGHT the bits by 8 bit)
00000000000000000000000000010011 ⇒ 19
```

If you notice:

4978 >> 8 = 19 (in 32 bits compiler)

$4978 / 2^8 = 4978 / 256 = 19$. (They both the same, but using bit-wise operator >> (SHIFT RIGHT) is a LOT faster!)

The RIGHT SHIFT operator is machine dependent, it may work differently on different machines for **signed** data types. >> empties the left most bit and that is filled either by a 0 or by a 1, depending on the machine.
Filled by a 1 (RIGHT SHIFT ARITHMETIC)
Filled by a 0 (RIGHT SHIFT LOGICAL)
Example:

```
signed char x = -75 /* 1011 0101 */
signed char y = x >> 2

/* result of logical right shift */
y = 45 /* 0010 1101 */

/* result of arithmetic right shift */
y = -19 /* 1110 1101 */
```

So this behavior of RIGHT SHIFT leads to a non-portability of a program.

A few more words

The two shift operators are generally used with unsigned data type to avoid ambiguity. Result of Shifting Right or Left by a value which is larger than the size of the variable is undefined.

Result of shifting Right or Left by a negative value is also undefined.

Order precedence of the basic operators is:

```
NOT                 (                 ~                 )                 highest
AND                         (                         &                         )
XOR                         (                         ^                         )
 OR ( | ) lowest
```

Some basic operations:

```
Let   X   is   a   single   bit,   then   we   can   write   the   following:
X      &      1      =            X;      X      &      0      =            0
X      |      1      =            1;      X      |      0      =            X
X ^ 1 = ~X; X ^ 0 =  X
```

Bit-wise operations are quite fast and easy to use, sometimes they reduce the running time of your program heavily, so use bit-wise operations when-ever you can. But if you look on the software developing aspect, they are not much reliable because, they aren't applicable with any type of data, especially floating points, and signed types. Also, not many people understand them.

But, still, who cares? I want to give some scary looks to my code... lol :D Hopefully, on the next post, we'll see some simple ideas which will implement some bit-wise operations...

Bitwise operations in C: Part 3

Bitwise Operation: Some applications

Back to Part 2

Binary number system is closely related with the powers of 2, and these special numbers always have some amazing bit-wise applications. Along with this, some general aspects will be briefly shown here.

Is a number a power of 2?
How can we check this? of course write a loop and check by repeated division of 2. But with a simple bit-wise operation, this can be done fairly easy.
We, know, the binary representation of $p = 2^n$ is a bit string which has only 1 on the n^{th} position (0 based indexing, right most bit is LSB). And p-1 is a binary number which has 1 on 0 to n-1 th position and all the rest more significant bits are 0. So, by AND-ing p and (p-1) will result 0 in this case:

```
p   = ....01000 &rArr 8
p-1 = ....00111 &rArr 7
----------------------
AND = ....00000 &rArr 0
```

No other number will show this result, like AND-ing 6 and 5 will never be 0.

Swap two integers without using any third variable:

Well, as no 3rd variable is allowed, we must find another way to preserve the values, how about we some how combine two values on one variable and the other will then be used as the temporary one...

```
Let A = 5 and B = 6
A = A ^ B = 3 /* 101 XOR 110 = 011 */
B = A ^ B = 5 /* 011 XOR 110 = 101 */
A = A ^ B = 6 /* 011 XOR 101 = 110 */
So, A = 6 and B = 5
```

Cool!!!

Divisibility by power of 2
Use of % operation is very slow, also the * and / operations. But in case of the second operand is a power of 2, we can take the advantage of bit-wise operations.
Here are some equivalent operations:

Here, P is in the form 2^X and N is any integer (typically unsigned)

```
N % P = N & (P-1)
N / P = N >> X
N * P = N << X
```

A lot faster and smarter...
About the % operation : the above is possible only when P is a power of 2

Masking operation
What is a mask? Its a way that is used to reshape something. In bit-wise operation, a masking operation means to reshape the bit pattern of some variable with the help of a bit-mask using some sort of bit-wise operation. Some examples following here (we won't talk about actual values here, rather we'll look through the binary representation and using only 16 bits for our ease of understanding):

Grab a portion of bit string from an integer variable.
Suppose A has some value like A = ... 0100 1101 1010 1001
We need the number that is formed by the bit-string of A from 3rd to 9th position.

[Lets assume, we have positions 0 to 15, and 0th position is the LSB]
Obviously the result is B = ... 01 1010 1; [we simply cut the 3rd to 9th position of A by hand]. But how to do this in programming.

Lets assume we have a mask X which contains necessary bit pattern that will help us to cut the desired portion. So, lets have a look how this has to be done:

```
A = 0100 1101 1010 1001
X = 0000 0011 1111 1000
```

See, we have X which have 1s in 3rd to 9th position and all the other thing is 0. We know, AND-ing any bit b with 1 leaves b unchanged.

```
So, X = X & A
Now, we have,
X = 0000 0001 1010 1000
```

So, now we've cut the desired portion, but this is exactly not what we wanted. We have 3 extra 0s in the tail, So just right-shift 3 times:

```
X = X >> 3 = 0000 0000 0011 0101; // hurrah we've got it.
```

Well, I got everything fine, but how to get such a weird X?

```
int x = 0, i, p=9-3+1; // p is the number of 1s we need
for(i=0; i<p, i++)
    x = (x << 1) | 1;
x = x << 3; // as we need to align its lsb with the 3rd bit of A
```

```
Execution:
X = 0000 0000 0000 0000 (initially X=0)
X = 0000 0000 0000 0001 (begin loop i=0)
X = 0000 0000 0000 0011 (i=1)
X = 0000 0000 0000 0111 (i=2)
X = 0000 0000 0000 1111 (i=3)
X = 0000 0000 0001 1111 (i=4)
X = 0000 0000 0011 1111 (i=5)
X = 0000 0000 0111 1111 (i=6 loop ends)
X = 0000 0011 1111 1000 (X=X<<3)
```

So, following the same approach, we may invert some portion of a bit-string (using XOR), make all bits 1 (using OR), make all bits in the portion 0 (using AND) etc etc very easily...

So, these are some tasks for the learner,
You have an integer A with some value, print the integers which have:
1. same bit pattern as A except it has all bits 0 within 4th to 23rd position.
2. same bit pattern as A except it has all bits 1 within 9th and 30th position.
3. same bit pattern as A except it has all bits inverted within positions 2nd and 20th.
4. totally inverted bit pattern.

Subset pattern:
Binary numbers can be used to represent subset ordering and all possible combination of taking n items.
For example, a problem might ask you to determine the n'th value of a series when sorted, where each term is some power of 5 or sum of some powers of 5.
It is clear that, each bit in a binary representation correspondence to a specific power of two in increasing order from right to left. And if we write down the consecutive binary values, we get some sorted integers. Like:

```
3 2 1 0 ⇒ power of 2
0 0 0 0 = 0 // took no one
0 0 0 1 = 1 // took power 0
0 0 1 0 = 2 // took power 1
0 0 1 1 = 3 // took power 1 and 0
0 1 0 0 = 4 // took power 2
```

```
0 1 0 1 = 5 // took power 2 and 0
0 1 1 0 = 6 // took power 2 and 1
0 1 1 1 = 7 // took power 2, 1 and 0
...........
...........
...........
```

So, what we do here is, take a power of 2 or take the sum of some powers of 2 to get a sorted sequence. Well, if this work for 2.. this will also work for 5 in the same way... Instead of taking the power of 2, we'll take the power of 5.
So the n'th term will be the sum of those powers of 5 on which position n's binary representation has 1. So the 10th term is $5^3 + 5^1$; As, $10_{10} = 1010_2$, it has 1 in 3rd and 1st position.

Worse than worthless
A bitwise GCD algorithm (wikipedia), translated into C from its assembly routine.

Sieve of Eratosthenes (SOE)
This is the idea of compressing the space for flag variables. For example, when we generate prime table using SOE, we normally use 1 int / bool for 1 flag, so if we need to store 10^8 flags, we barely need 100MB of memory which is surely not available... and using such amount of memory will slow down the process. So instead of using 1 int for 1 flag, why don't we use 1 int for 32 flags on each of its 32 bits? This will reduce the memory by 1/32 which is less than 4MB :D

```
#define MAX 100000000
#define LMT 10000

unsigned flag[MAX>>6];

#define ifc(n) (flag[n>>6]&(1<<((n>>1)&31)))
#define isc(n) (flag[n>>6]|=(1<<((n>>1)&31)))

void sieve() {
    unsigned i, j, k;
    for(i=3; i<LMT; i+=2)
        if(!ifc(i))
            for(j=i*i, k=i<<1; j<MAX; j+=k)
                isc(j);
}
```

ifc(x) checks if x is a composite (if correspondent bit is 1)
isc(x) sets x as a composite (by making correspondent bit 1)

Other than this famous bit-wise sieve, we could use the technique in many places, to reduce memory for flagging.

Keep going on...

TANVER AHMED