TUESDAY, DECEMBER 22, 2009

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...

Continue to Part 3

Posted by Zobayer Hasan at 12:06 AM

4 comments:

Anonymous October 22, 2015 at 4:19 AM

X & 1 = X; X | 1 = 1; X ^ 1 = ~X; I think these statements are not always true unless X is a single bit boolean.

Reply

Replies

Zobayer Hasan October 22, 2015 at 11:54 PM

True. They are intended to be representing a single bit. I will update the post.

Reply

BIDDUT SARKER BIJOY October 9, 2017 at 12:19 AM

ভালই ছিল টিউটোরিয়ালটা.....

Reply

Anonymous May 7, 2018 at 11:25 PM

Can you clear the line "So, while left shifting, if you push a 1 at that position, the number will be negative." by giving an example please...

Reply