Tutorial Link https://codequotient.com/tutorials/Bitwise Operators/5a1d998152795c1b16c0ac07

TUTORIAL

Bitwise Operators

Chapter

1. Bitwise Operators

Bitwise operators are similar to logical operators, but they work on individual bits and bytes of operands. Bitwise operators works on char and int variables only. Following are bitwise operators available:

Operator	Description
&	AND
|	OR
^	XOR
~	NOT
>>	Right Shift
<<	Left Shift

Following example will illustrate First 4 operators: **AND, OR, XOR, NOT.**

```c
#include <stdio.h>

int main()
{
  short int c = 5, d = 6;            // Assume short int is of 8-bits, then
  printf("c & d=%d\n", c & d);
      // 0000 0101 = 5
      // 0000 0110 = 6
  //AND= 0000 0100 = 4

  printf("c | d=%d\n", c | d);
      // 0000 0101 = 5
```

```
      // 0000 0110 = 6
  //OR = 0000 0111 = 7

  printf("c ^ d=%d\n", c ^ d);
      // 0000 0101 = 5
      // 0000 0110 = 6
  //XOR= 0000 0011 = 3

  printf("~c=%d\n", ~c);
      // 0000 0101 = 5
  //NOT= 1111 1010 = -6

  return 0;
}
```

C++

```
#include<iostream>
#include<cstdio>
#include<cmath>
using namespace std;
int main(){
    short int c = 5, d = 6;            // Assume short int is of 8-bits, then
    cout<<"c & d="<< (c & d) <<endl;
        // 0000 0101 = 5
        // 0000 0110 = 6
    //AND= 0000 0100 = 4

    cout<<"c | d="<< (c | d) <<endl;
        // 0000 0101 = 5
        // 0000 0110 = 6
    //OR = 0000 0111 = 7

    cout<<"c ^ d"<< (c ^ d) <<endl;
        // 0000 0101 = 5
        // 0000 0110 = 6
    //XOR= 0000 0011 = 3

    cout<<"~c="<< (~c) <<endl;
        // 0000 0101 = 5
    //NOT= 1111 1010 = -6
```

```
25
26    return 0;
27  }
28
```

One difference between logical and bitwise operators is logical operators return either 0 or 1, based on their result, whereas bitwise operators does not check for true or false, instead they compute a new value and return.

The bit-shift operators, >> and <<, move all bits in a variable to the right or left as specified. The general form of the shift-right statement is

```
variable >> number of bit positions
```

The general form of the shift-left statement is

```
variable << number of bit positions
```

As bits are shifted off one end, zeroes are brought in the other end. (In the case of a signed, negative integer, a right shift will cause a 1 to be brought in so that the sign bit is preserved.) Remember, a shift is not a rotate. That is, the bits shifted off one end do not come back around to the other. The bits shifted off are lost.

For example,

```c
1  #include <stdio.h>
```

```
2
3  int main()
4  {
5      short int c = 5, d = 6;            // Assume short
   int is of 8-bits, then
6      printf("c >> 1 = %d\n", c >> 1);
7          //  0000 0101 = 5
8      //C>>1= 0000 0010 = 2
9
10     printf("c >> 2 = %d\n", c >> 2);
11         //  0000 0101 = 5
12     //C>>2= 0000 0001 = 1
13
14     printf("d << 1 = %d\n", d << 1);
15         //  0000 0110 = 6
16     //d<<1= 0000 1100 = 12
17
18     printf("d << 2 = %d\n", d << 2);
19         //  0000 0110 = 6
20     //d<<2= 0001 1000 = 24
21
22   return 0;
23 }
24
```

C++

```
1  #include<iostream>
2  #include<cstdio>
3  #include<cmath>
4  using namespace std;
5  int main()
6  {
7      short int c = 5, d = 6;            // Assume short
   int is of 8-bits, then
8      cout<< "c >> 1 = " << (c >> 1) << endl;
9          //  0000 0101 = 5
10     //C>>1= 0000 0010 = 2
11
12     cout<< "c >> 2 = " << ( c >> 2) << endl;
13         //  0000 0101 = 5
14     //C>>2= 0000 0001 = 1
15
```

```
    cout<<"d << 1 = " << (d << 1) <<endl;
        //  0000 0110 = 6
    //d<<1= 0000 1100 = 12

    cout<< "d << 2 = " << (d << 2) <<endl;
        //  0000 0110 = 6
    //d<<2= 0001 1000 = 24

  return 0;
}

```

Eventually, if you shift a number k bit right, you are yielding quotient(number / (2^k)). And if you shift a number k bit left, you are yielding (number * (2^k)).

Bitwise operators are useful in many situations like masking and unmasking of bits. AND operator helps in making a specific bit to 0, and OR helps in making a specific bit to 1. For example, if

```
a = 126;   in binary a = 0111 1110
```

So if you want to make the 5th bit (from lsb) to 0, we can perform the following operation:

```
a & 239;   // it is evaluated as
         //    0111 1110 = 126 = a
         //    1110 1111 = 239
      // & =   0110 1110 = 110 (similar to a except 5th bit is
now 0)
```

Whereas if you want to check weather 5th bit from lsb of a number is 1 or not: -

```
a = 126;   in binary a = 0111 1110
```

we can perform the following operation:

```
a & 16;   // it is evaluated as
         //    0111 1110 = 126 = a
         //    0001 0000 = 16 = (2^(5-1))
      // & =   0001 0000 = 16 (if result is 0 then the bit 0,
otherwise bit is 1)
```

There are lot of ways to do these things with bitwise operators.