OPERATION ON BITS

As mentioned earlier, C language is a systems programming language. Hence it supports bit operations. For people involved in systems programming and machine interfacing, the bitwise operators provide the memory access needed without forcing them to write machine language code. However in day to day business usage, these operators are rarely used. Before dwelling into bit operations and the operators available in C, let us discuss some basics.

The computer can understand the binary number system. Binary means base two number system. The binary number system consists of 0 and 1. I am assuming the student is familiar with different number systems and the conversion of decimal to binary, and vice-versa.

On most computer systems a byte represents 8 bits , each bit can be a 0 or 1. So a byte is a string of zeros and ones. The right most bit of a byte is known as least significant bit and the left most bit is known as the most significant bit. A negative number is represented in a slightly different manner. (Remember that interger can be signed or unsigned.)If the left most bit is the sign bit and if it is 1 then the number is negative, if it is 0 then the number is positive.

Bit operators as the name suggests are used to perform operations on binary digits. The different bit operators supported by C are as follows :

&	Bitwise AND
|	Bitwise inclusive OR
^	Bitwise EXOR
~	Ones complement
<<	Left shift
>>	Right shift

Table 11.1

Bitwise logical operators

Logical operations have two results, it is either true or false. In binary notation 1 denotes true and 0 a false. An int stored in memory can be seen as a string of true and false values.

The bitwise logical operators work on integer operands, which are evaluated bit by bit to an integer result.

Bitwise AND

The logical AND evaluates to true (1), only if both operands are 1.

0 & 0	0
0 & 1	0
1 & 0	0
1 & 1	1

Table 11.2

Program 11.1

```
/* Program to demonstrate logical AND operator */

#include <stdio.h>
main ()
{
    int a = 25;
    int b = 77;
    int c;
    c = a & b;
    printf("Value of c is %d\n", c);
}
```

The program results in the following output :

```
Value of c is 9.
```

The AND operation that occurs is shown below :

0000000000011001	25
0000000001001101	77
0000000000001001	9

Table 11.3

The bitwise AND operation is used for *masking* operations.This operator can be used to set specific bits of a number to 0. Also you can use bitwise AND to test whether an integer is odd or even. When you do a bitwise AND of an integer with 1, the result will be true if the rightmost bit of the integer is 1. This is the case with an odd integer, whereas an even integer will have 0 as the rightmost bit.

Bitwise inclusive OR

In this again, the binary representation of the two operands involved are compared bit by bit. Each bit that is a 1 in the first operand or a 1 in the second operand will produce a 1 in the corresponding bit of the result.

0 | 0	0

0 | 1	1
1 | 0	1
1 | 1	1

Table 11.3

The bitwise inclusive OR operation is used when you want to set some specified bits of a number to 1.

Bitwise Exclusive OR

The bitwise exclusive OR or XOR gives 1 if either bit is 1 but not both.

0 ^ 0	0
0 ^ 1	1
1 ^ 0	1
1 ^ 1	0

Table 11.4

One important point to remember is that any value which is XORed with itself results in 0. This is used by assembly language programmers to test for equality of two values.

One's complement operator

The one's complement operator in C is represented as a tilde **~**. This converts a value into its one's complement, in other words, all the zeros become ones and the ones become zeros. This is a unary operator that operates on an integer constant or expression.

For example

```
~ 12
~ (Total - 2)
```

Here, Total must be an integer.

One's complement is used to make a program more portable. For example, if we want to set the rightmost bit of an integer to zero on both 16 bit and 32 bit machines, it is wiser to AND it with a one's complement

```
Total & = ~1;
```

This will result in a one's complement of 1 with as many leftmost 1 bits as required to fill the size of an integer. It will be 15 leftmost bits on a 16-bit integer machine, and 31 on a 32-bit integer machine.

Shift operators

Shift operations that shift one bit respectively to the left or to the right are analogous to respectively multiplication by 2 or division by 2. When we do a left shift of one bit on an operand, we obtain the operand's value multiplied by 2. When we do a right shift of one bit on an operand, we obtain the operand's value divided by 2.

Left shift operator <<

When a left shift operation is performed on an operand, the bits of the operand are shifted left by the number of bits specified by the right operand. Bits that are shifted out of the high order bit of the data item are lost and 0s are added through the low order bit of the operand.

If *Salary* is a variable with the octal value of 6, then a left shift operation results in the octal value 14.

Program 11.2

```
/* Program to demonstrate left shift operator */

#include <stdio.h>
main ()
{
    unsigned int salary = 06;
    salary <<=1;
    printf("Value of salary is %o\n", salary);
}
```

Right shift operator >>

When a right shift operation is performed on an operand, the bits of the operand are shifted right by the number of bits specified by the right operand. In a right shift operation, the bit on the right which is the low-order bit is lost and depending on the type of machine, either a 1 or 0 will be shifted into the leftmost bit. The shift operation is also sometimes referred to as rotating left and right.

Source : http://www.peoi.org/Courses/Coursesen/cprog/frame11.html