Unit-5 (C Preprocessor & Bitwise Operators)
(Ram Gopal Gupta- http://ramgopalgupta.com/)

Part-2

BITWISE Operators:

In C Programming, the following 6 operators are bitwise operators (work at bit-level)

1- The **& (bitwise AND)** in C takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

2- The **| (bitwise OR)** in C takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

3- The ^ **(bitwise XOR)** in C takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

4- The << **(left shift)** in C takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.

5- The >> **(right shift)** in C takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.

6- The **~ (bitwise NOT)** in C takes one number and inverts all bits of it.

Program example7:

// C Program to demonstrate use of bitwise operators

```
Line 1-     #include <stdio.h>
Line 2-     int main()
Line 3-     {

            // a = 5(00000101), b = 9(00001001)
Line 4-     unsigned char a = 5, b = 9;

            // The result is 00000001
Line 5-     printf("a = %d, b = %d\n", a, b);
Line 6-     printf("a&b = %d\n", a & b);

            // The result is 00001101
Line 7-     printf("a|b = %d\n", a | b);

            // The result is 00001100
Line 8-     printf("a^b = %d\n", a ^ b);

            // The result is 11111010
Line 9-     printf("~a = %d\n", a = ~a);
// The result is 00010010
printf("b<<1 = %d\n", b << 1);

// The result is 00000100
printf("b>>1 = %d\n", b >> 1);

return 0;
}
```

Output:

```
a = 5, b = 9

a&b = 1

a|b = 13

a^b = 12

~a = 250

b<<1 = 18

b>>1 = 4
```

Interesting facts about bitwise operators

1- **The left shift and right shift operators should not be used for negative numbers.** If any of the operands is a negative number, it results in undefined behaviour. For example results of both -1 << 1 and 1 << -1 is undefined. Also, if the number is shifted more than the size of integer, the behaviour is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits. See this for more details.

2- **The bitwise XOR operator is the most useful operator from technical interview perspective.** It is used in many problems. A simple example could be “Given a set of numbers where all elements occur even number of times except one number, find the odd occurring number” This problem can be efficiently solved by just doing XOR of all numbers.

3- The bitwise operators should not be used in place of logical operators.

<u>BIT Fields in C</u>

In C, we can specify size (in bits) of structure and union members. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.

Program example 8:

```
Line 1-     #include <stdio.h>
            // Space optimized representation of the date
Line 2-     struct date {
            // d has value between 1 and 31, so 5 bits are sufficient
Line 3-     unsigned int d : 5;
Line 4-     // m has value between 1 and 12, so 4 bits are sufficient
Line 5-     unsigned int m : 4;
Line 6-     unsigned int y;
Line 7-     };

Line 8-     int main()
Line 9-     {
Line 10-    printf("Size of date is %lu bytes\n", sizeof(struct date));
Line 11-    struct date dt = { 31, 12, 2014 };
Line 12-    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
Line 13-    return 0;
Line 14-    }
```

Output:

```
Size of date is 8 bytes
Date is 31/12/2014
```

Explanation:

At line 3 & 5, variable d & m assigned 5 & 4 bits receptively to represent value 31 & 12 respectively.