Bitwise Operators in C

In C, the following 6 operators are bitwise operators (also known as bit operators as they work at the bit-level). They are used to perform bitwise operations in C.

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, the left shifts the bits of the first operand, and 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, and 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.

Bitwise operators allow precise manipulation of bits, giving you control over hardware operations.

Let's look at the truth table of the bitwise operators.

X	Y	X & Y	X | Y	X ^ Y
0	**0**	0	0	0
0	**1**	0	1	1
1	**0**	0	1	1
1	**1**	1	1	0

Example of Bitwise Operators in C

The following program uses bitwise operators to perform bit operations in C.

```
// C Program to demonstrate use of bitwise operators


#include <stdio.h>

int main()
```

```
{
    // a = 5 (00000101 in 8-bit binary), b = 9 (00001001 in
    // 8-bit binary)
    unsigned int a = 5, b = 9;

    // The result is 00000001
    printf("a = %u, b = %u\n", a, b);
    printf("a&b = %u\n", a & b);

    // The result is 00001101
    printf("a|b = %u\n", a | b);

    // The result is 00001100
    printf("a^b = %u\n", a ^ b);

    // The result is 11111111111111111111111111111010
    // (assuming 32-bit unsigned int)
    printf("~a = %u\n", a = ~a);

    // The result is 00010010
    printf("b<<1 = %u\n", b << 1);

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

    return 0;
}
```

Output

```
a = 5, b = 9
a&b = 1
```

```
a|b = 13

a^b = 12

~a = 4294967290

b<<1 = 18

b>>1 = 4
```

Time Complexity: *O(1)*
Auxiliary Space: *O(1)*

Interesting Facts About Bitwise Operators

1. The left-shift and right-shift operators should not be used for negative numbers.

If the second operand(which decides the number of shifts) is a negative number, it results in undefined behavior in C. For example, results of both 1 <<- 1 and 1 >> -1 are undefined. Also, if the number is shifted more than the size of the integer, the behavior is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits. Another thing is NO shift operation is performed if the additive expression (operand that decides no of shifts) is 0. See this for more details.

2. The bitwise OR of two numbers is simply the sum of those two numbers if there is no carry involved; otherwise, you add their bitwise AND.

Let’s say, we have a=5(101) and b=2(010), since there is no carry involved, their sum is just a|b. Now, if we change ‘b’ to 6 which is 110 in binary, their sum would change to a|b + a&b since there is a carry involved.

3. The bitwise XOR operator is the most useful operator from a **technical interview perspective.**

It is used in many problems. A simple example could be “Given a set of numbers where all elements occur an even number of times except one number, find the odd occurring number” This problem can be efficiently solved by doing XOR to all numbers.

Example

Below program demonstrates the use XOR operator to find odd occcuring elements in an array.

```
// C program to find odd occcuring elements in an array


#include <stdio.h>


// Function to return the only odd

// occurring element

int findOdd(int arr[], int n)

{

   int res = 0, i;
```

```
    for (i = 0; i < n; i++)

        res ^= arr[i];

    return res;

}


int main(void)

{

    int arr[] = { 12, 12, 14, 90, 14, 14, 14 };

    int n = sizeof(arr) / sizeof(arr[0]);

    printf("The odd occurring element is %d ",

           findOdd(arr, n));

    return 0;

}
```

Output

```
The odd occurring element is 90
```

4. The Bitwise operators should not be used in place of logical operators.

The result of logical operators (&&, || and !) is either 0 or 1, but bitwise operators return an integer value. Also, the logical operators consider any non-zero operand as 1. For example, consider the following program, the results of & & && are different for the same operands.

Example

The below program demonstrates the difference between & and && operators.

```
// C program to Demonstrate the difference between & and &&

// operator


#include <stdio.h>


int main()

{

    int x = 2, y = 5;

    (x & y) ? printf("True ") : printf("False ");

    (x && y) ? printf("True ") : printf("False ");
    return 0;
}
```

Output

```
False True
```

Time Complexity: *O(1)*
Auxiliary Space: *O(1)*

5. The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively.

As mentioned in point 1, it works only if numbers are positive.

Example:

The below example demonstrates the use of left-shift and right-shift operators.

```
// program to demonstrate the use of left-shift and
// right-shift operators.
#include <stdio.h>

int main()
{
    int x = 19;
    printf("x << 1 = %d\n", x << 1);
    printf("x >> 1 = %d\n", x >> 1);
    return 0;
}
```

Output

```
x << 1 = 38
x >> 1 = 9
```

Time Complexity: *O(1)*
Auxiliary Space: *O(1)*

6. The & operator can be used to quickly check if a number is odd or even.

The value of the expression (x & 1) would be non-zero only if x is odd, otherwise, the value would be zero.

Example

The below example demonstrates the use bitwise & operator to find if the given number is even or odd.

```
#include <stdio.h>


int main()

{

   int x = 19;

   (x & 1) ? printf("Odd") : printf("Even");

   return 0;

}
```

Output

```
Odd
```

Time Complexity: *O(1)*
Auxiliary Space: *O(1)*

7. The ~ operator should be used carefully.

The result of the ~ operator on a small number can be a big number if the result is stored in an unsigned variable. The result may be a negative number if the result is stored in a signed variable (assuming that the negative numbers are stored in 2's complement form where the leftmost bit is the sign bit).

Example

The below example demonstrates the use of bitwise NOT operator.

```
// C program to demonstrate the use of bitwise NOT operator.


#include <stdio.h>


int main()

{

   unsigned int x = 1;

   printf("Signed Result %d \n", ~x);

   printf("Unsigned Result %u \n", ~x);

   return 0;

}
```

Output

```
Signed Result -2

Unsigned Result 4294967294
```

Time Complexity: *O(1)*
Auxiliary Space: *O(1)*