Bitwise Operators in C

To perform operations on available data at the bit level, we use the bitwise operators in C. As a result, carrying out a bitwise operation is also referred to as bit-level programming. Because it consists of only two digits - 1 or 0 - it is primarily used in numerical computations for faster calculation.

The bitwise operators in C work with integers, that is, they take integer inputs, manipulate with their bits, and return an integer value. The bitwise AND and OR operators are '&' and '|', while the logical AND and OR operators are '&&' and '||'.

Types of Bitwise Operators in C

Some of these operator's symbols and names may appear similar to logical operators, but they are different from them. There are various types of bitwise operators in all programming languages. There are basically six different types of bitwise operators in C that are as follows:

- Bitwise OR Operator
- Bitwise AND Operator
- Unary Operator (Binary One’s complement operator)
- Bitwise XOR Operator
- Binary Right Shift Operator
- Binary Left Shift Operator

Bitwise OR Operator

The bitwise OR operator is represented by the vertical bar or pipe symbol, |. The bitwise OR operator is a binary operator which requires two operands (two integers) to operate on. It takes the binary values of both the left and right operands and performs the logical OR operation on the bit level over them, i.e. if both operands have a 0 in the specified position, the result will also have a 0 in the corresponding position, otherwise it will be 1.

The truth table of the bitwise OR operator is as follows:

Num1	Num2	Result = Num1 | Num2
0	0	0
1	0	1

0	1	1
1	1	1

Bitwise AND Operator

The bitwise AND operator is represented by a single ampersand symbol, &. The bitwise AND operator is a binary operator which requires two operands (two integers) to operate on. It takes the binary values of both the left and right operands and performs the logical AND operation on the bit level over them, i.e. if both operands have a 1 in the specified position, the result will also have a 1 in the corresponding position, otherwise it will be 0.

The truth table of the bitwise AND operator is as follows:

Num1	Num2	Result = Num1 & Num2
0	0	0
1	0	0
0	1	0
1	1	1

Unary Operator (Binary One's Complement Operator)

We have above seen two bitwise operators which require two operands. But this is different as it requires only one operand, whereas all the others require two operands. The one's complement of a single value is returned by the bitwise complement operator. A number's one's complement is obtained by changing all of its 0's to 1's and all of its 1's to 0's.

This is denoted by the tilde symbol '~'. The truth table of the unary operator is as follows:

Num1	Result = ~Num1
0	1
1	0

Bitwise XOR Operator

The bitwise XOR operator is represented by '^'. The bitwise XOR operator is a binary operator which requires two operands (two integers) to operate on. It takes the binary values of both the left and right operands and performs the logical XOR operation on the bit level over them, i.e. if one operand has a 1 and the other has a 0, the result will have a 1 in the corresponding position, and a 0 if they have the same bits, such as both 0s or both 1s.

The truth table of the bitwise XOR operator is as follows:

Num1	Num2	Result = Num1 ^ Num2
0	0	0
1	0	1
0	1	1
1	1	0

Binary Right Shift Operator

Binary Shift right is represented by two consecutive greater than operators, i.e. >>. This is equivalent to dividing the number by 2 power n, where n is the operand to the operator's right.

The value of 'x' can be negative, but not that of 'n'; if 'n' is negative, the compiler will throw an error stating 'negative shift count.' When the value of 'x' is negative, the right Shift operation is applied to the number's two's complements. As a result, the sign of the number may or may not be the same as the right shift operation.

Binary Left Shift Operator

Binary Shift Left is represented by two consecutive greater than operators, i.e. <<. This is equivalent to multiplying the number by 2 power n, where n is the operand to the operator's right.

The value of 'x' can be negative, but not that of 'n'; if 'n' is negative, the compiler will throw an error stating 'negative shift count.' When the value of 'x' is negative, the Left Shift operation is applied to the number's two's complements. As a result, the sign of the number may or may not be the same as the left shift operation.

Examples of Bitwise Operators in C

Example 1: We will use this programme to demonstrate how to perform a left-shift operation in C. The example of the Binary Left Shift () Operator in C.

```
#include <stdio.h>

int main()

{

unsigned int num = 0xff;

printf("\nValue of num = %04X before the left shift.", num);

/*shifting 2 bytes left*/

num = (num << 2);

printf("\nValue of num = %04X after the left shift.", num);

return 0;

}
```

Output:

Value of num = 00FF before the left shift.
Value of num = 03FC after the left shift.

Explanation:

The Left Shift Operator (<<) is a bitwise operator that operates on bits. It shifts a given number of bytes to the left and inserts 0's to the right.

0000 0000 1111 1111 is the binary of 0xFF (in 4 bytes format).

After a two-byte left shift (in four-byte format), the result is 0000 0011 1111 1100, which is equivalent to 0x03FC.

Example 2: We will use this programme to demonstrate how to perform a right shift operation in C. The example of the Binary right Shift () Operator in C.

```
#include <stdio.h>

int main()
```

```
{

unsigned int num = 0xff;

printf("\nValue of num = %04X before the left shift.", num);

/*shifting 2 bytes left*/

num = (num >> 2);

printf("\nValue of num = %04X after the left shift.", num);

return 0;

}
```

Output:

Value of num = 00FF before the left shift.
Value of num = 003F after the left shift.

Explanation:

The Right Shift Operator (>>) is a bitwise operator that operates on bits. It shifts a given number of bytes to the right and inserts 0's to the left.

0000 0000 1111 1111 is the binary of 0xFF (in 4 bytes format).

After a two-byte left shift (in four-byte format), the result is 0000 0000 0011 1111, which is equivalent to 0x003F.

Example 3: We will use this programme to demonstrate how to perform a Bitwise AND operators in C. The example of the Bitwise AND operators in C.

```
#include <stdio.h>

int main()

{

int num1 = 20;

int num2 = 15

printf("\n Value of Bitwise AND operator = %d", num1 & num2);
```

```
return 0;

}
```

Output:

Value of Bitwise AND operator = 4

Explanation:

In binary 20 = 00010100

In binary 15 = 00001111

Bitwise AND operators of 20 and 15

```
  00010100
& 00001111
  00000100   4(In decimal)
```

Example 4: We will use this programme to demonstrate how to perform a Bitwise OR operators in C. The example of the Bitwise OR operators in C.

```
#include <stdio.h>

int main()

{

int num1 = 20;

int num2 = 15

printf("\n Value of Bitwise AND operator = %d", num1 | num2);

return 0;

}
```

Output:

Value of Bitwise OR operator = 31

Explanation:

In binary 20 = 00010100

In binary 15 = 00001111

Bitwise OR operators of 20 and 15

  00010100

| 00001111

  00011111 31(In decimal)

Example 5: We will use this programme to demonstrate how to perform a Bitwise XOR operators in C. The example of the Bitwise XOR operators in C.

```
#include <stdio.h>

int main()

{

int num1 = 20;

int num2 = 15

printf("\n Value of Bitwise AND operator = %d", num1 ^ num2);

return 0;

}
```

Output:

Value of Bitwise XOR operator = 27

Explanation:

In binary 20 = 00010100

In binary 15 = 00001111

Bitwise XOR operators of 20 and 15

   00010100

^ 00001111

  00011011 27(In decimal)