C++ Bitwise Operators

In this tutorial, we will learn about bitwise operators in C++ with the help of examples.

In C++, bitwise operators perform operations on integer data at the individual bit-level. These operations include testing, setting, or shifting the actual bits. For example,

```
a & b;
a | b;
```

Here is a list of 6 bitwise operators included in C++.

Operator	Description
`&`	Bitwise AND Operator
`|`	Bitwise OR Operator
`^`	Bitwise XOR Operator
`~`	Bitwise Complement Operator
`<<`	Bitwise Shift Left Operator
`>>`	Bitwise Shift Right Operator

1. C++ Bitwise AND Operator

The **bitwise AND** `&` operator returns **1** if and only if both the operands are **1**. Otherwise, it returns **0**.

The following table demonstrates the working of the **bitwise AND** operator. Let **a** and **b** be two operands that can only take binary values i.e. **1 and 0**.

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

> **Note:** The table above is known as the "Truth Table" for the **bitwise AND** operator.

Let's take a look at the **bitwise AND** operation of two integers 12 and 25:

```
      00001100
&     00011001
      ----------
      00001000   = 8 (In decimal)
```

Example 1: Bitwise AND

```cpp
#include <iostream>
using namespace std;

int main() {
    // declare variables
    int a = 12, b = 25;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "a & b = " << (a & b) << endl;

    return 0;
}
```

Output

```
a = 12
b = 25
a & b = 8
```

In the above example, we have declared two variables `a` and `b`. Here, notice the line,

```cpp
cout << "a & b = " << (a & b) << endl;
```

a and **b** be two operands that can only take binary values i.e. **1 or 0.**

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Let us look at the **bitwise OR** operation of two integers **12** and **25:**

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
    00001100
|   00011001
    __________
    00011101  = 29 (In decimal)
```

Example 2: Bitwise OR

```
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "a | b = " << (a | b) << endl;

    return 0;
}
```

Output

```
a = 12
b = 25
a | b = 29
```

The **bitwise OR** of `a = 12` and `b = 25` gives `29`.

---

3. C++ Bitwise XOR Operator

The **bitwise XOR** `^` operator returns **1** if and only if one of the operands is **1**. However, if both the operands are **0**, or if both are **1**, then the result is **0**.

The following truth table demonstrates the working of the **bitwise XOR** operator. Let **a** and **b** be two operands that can only take binary values i.e. **1 or 0**.

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

```
    ----------
    00010101  = 21 (In decimal)
```

Example 3: Bitwise XOR

```cpp
#include <iostream>

int main() {
    int a = 12, b = 25;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "a ^ b = " << (a ^ b) << endl;

    return 0;
}
```

Output

```
a = 12
b = 25
a ^ b = 21
```

The **bitwise XOR** of `a = 12` and `b = 25` gives `21`.

4. C++ Bitwise Complement Operator

The bitwise complement operator is a unary operator (works on only one operand). It is denoted by `~` that changes binary digits **1** to **0** and **0** to **1**.

It is important to note that the **bitwise complement** of any integer **N** is equal to **-(N + 1)**. For example,

Consider an integer **35**. As per the rule, the bitwise complement of **35** should be **-(35 + 1) = -36**. Now, let's see if we get the correct answer or not.

```
35 = 00100011 (In Binary)

// Using bitwise complement operator
~ 00100011
 ___________
  11011100
```

In the above example, we get that the bitwise complement of **00100011** (**35**) is **11011100**. Here, if we convert the result into decimal we get **220**.

However, it is important to note that we cannot directly convert the result into decimal and get the desired output. This is because the binary result **11011100** is also equivalent to **-36**.

To understand this we first need to calculate the binary output of **-36**. We use 2's complement to calculate the binary of negative integers.

2's Complement

The 2's complement of a number **N** gives **-N**.

In binary arithmetic, 1's complement changes **0 to 1** and **1 to 0**.

For example,

```
36 = 00100100 (In Binary)

1's Complement = 11011011

2's Complement :
11011011
 +     1
_________
11011100
```

Here, we can see the 2's complement of **36** (i.e. **-36**) is **11011100.** This value is equivalent to the **bitwise complement of 35** that we have calculated in the previous section.

Hence, we can say that the bitwise complement of 35 = -36.

Example 4: Bitwise Complement

```
    cout << "~(" << num1 << ") = " << (~num1) << endl;
    cout << "~(" << num2 << ") = " << (~num2) << endl;

    return 0;
}
```

Output

```
~(35) = -36
~(-150) = 149
```

In the above example, we declared two integer variables `num1` and `num2`, and initialized them with the values of `35` and `-150` respectively.

We then computed their bitwise complement with the codes `(~num1)` and `(~num2)` respectively and displayed them on the screen.

```
The bitwise complement of 35 = - (35 + 1) = -36
i.e. ~35 = -36

The bitwise complement of -150 = - (-150 + 1) = - (-149) = 149
i.e. ~(-150) = 149
```

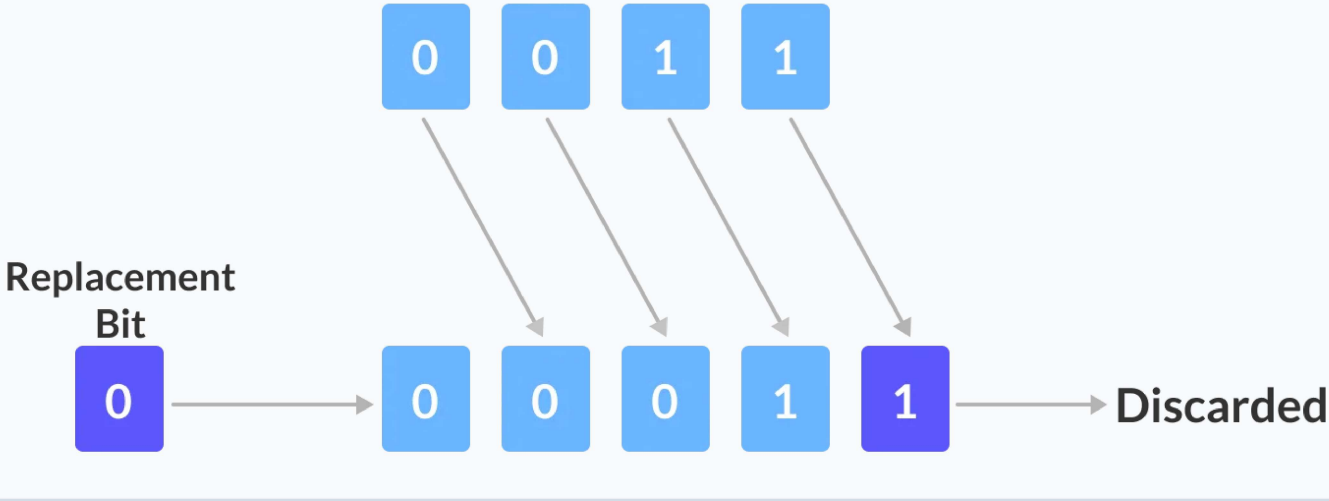
This is exactly what we got in the output.

C++ Shift Operators

There are two shift operators in C++ programming:

- Right shift operator `>>`
- Left shift operator `<<`

while the **most significant bits** are replaced by zeroes.



One bit Right Shift

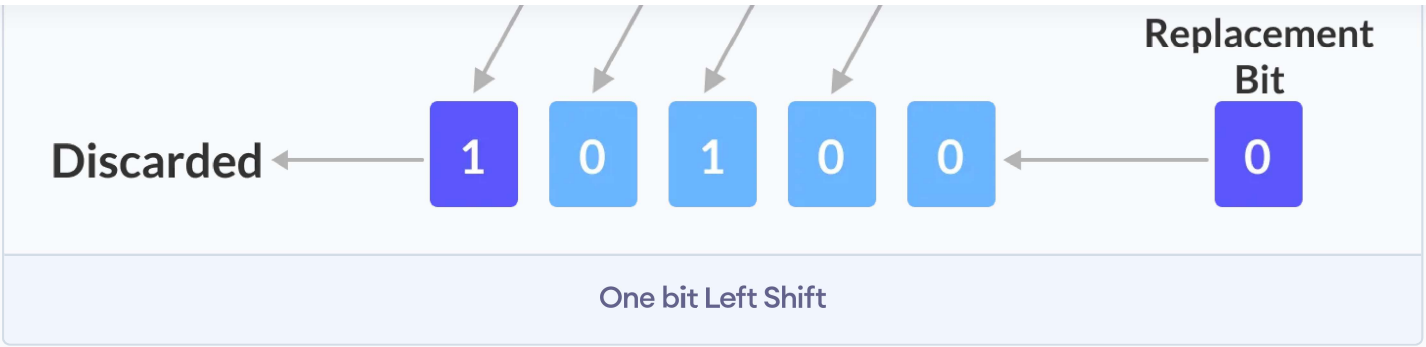
As we can see from the image above, we have a **4-bit number**. When we perform a **one-bit** right shift operation on it, each individual bit is shifted to the right by 1 bit.

As a result, the right-most bit is discarded, while the left-most bit remains vacant. This vacancy is replaced by a **0**.

---

6. C++ Left Shift Operator

The **left shift operator** shifts all bits towards the left by a certain number of **specified bits.** It is denoted by `<<`.



One bit Left Shift

As we can see from the image above, we have a **4-bit number.** When we perform a **1 bit** left shift operation on it, each individual bit is shifted to the left by 1 bit.

As a result, the left-most bit is discarded, while the right-most bit remains vacant. This vacancy is replaced by a **0**.

Example 5: Shift Operators

```
    int num = 212, i;

    // Shift Right Operation
    cout << "Shift Right:" << endl;

    // Using for loop for shifting num right from 0 bit to 3 bits
    for (i = 0; i < 4; i++) {
        cout << "212 >> " << i << " = " << (212 >> i) << endl;
    }

    // Shift Left Operation
    cout << "\nShift Left:" << endl;

    // Using for loop for shifting num left from 0 bit to 3 bits
    for (i = 0; i < 4; i++) {
        cout << "212 << " << i << " = " << (212 << i) << endl;
    }

    return 0;
}
```

Output

```
Shift Right:
212 >> 0 = 212
212 >> 1 = 106
212 >> 2 = 53
212 >> 3 = 26

Shift Left:
212 << 0 = 212
212 << 1 = 424
212 << 2 = 848
212 << 3 = 1696
```

From the output of the program above, we can infer that, for any number **N**, the results of the shift right operator are:

and so on.

Similarly, the results of the shift left operator are:

```
N << 0 = N
N << 1 = (N << 0) * 2
N << 2 = (N << 1) * 2
N << 3 = (N << 2) * 2
```

and so on.

Hence we can conclude that,

```
N >> m = [ N >> (m-1) ] / 2
N << m = [ N << (m-1) ] * 2
```

In the above example, note that the `int` data type stores numbers in **32-bits** i.e. an `int` value is represented by **32 binary digits.**

However, our explanation for the bitwise shift operators used numbers represented in **4-bits.**

For example, the base-10 number **13** can be represented in 4-bit and 32-bit as:

```
4-bit Representation of 13 = 1101
32-bit Representation of 13 = 00000000 00000000 00000000 00001101
```

As a result, the **bitwise left-shift** operation for **13** (and any other number) can be different depending on the number of bits they are represented by.

Because in **32-bit** representation, there are many more bits that can be shifted left

Did you find this article helpful?

Related Tutorials

C++ Tutorial

C++ Operator Precedence and Associativity

(/cpp-programming/operators-precedence-associativity)

C++ Tutorial

(/cpp-programming/relational-logical-operators)

C++ Tutorial

C++ Operator Overloading

(/cpp-programming/operator-overloading)