

Bitwise Operators

☰ Tags	
🕒 Created time	@November 7, 2024 12:04 PM
☑ Reviewed	<input type="checkbox"/>

Let's examine a group of operators that look something like the logical operators you have seen earlier in the course, but in fact are quite different. These are called the **bitwise operators**, because they operate on the *bits in integer values*. There are six bitwise operators, as shown in Table below:

Operator	Description
&	Bitwise AND operator
	Bitwise OR operator
^	Bitwise exclusive OR (XOR) operator
~	Bitwise NOT operator, also known as bitwise complement operator
>>	Bitwise shift right operator
<<	Bitwise shift left operator

All of these only operate on integer types. The **~** operator is a unary operator, i.e. it applies to one operand—and. The others are *binary operators*.



It's important not to confuse the bitwise operators and the logical operators. The expression

`x & y`

will produce quite a different result from

`x && y`

in general.

Bitwise **AND** operator

```
int x = 13;
int y = 6;
int z = z & y;

printf("%d", z);
---
```

\$./run

4

After the third statement, `z` will have the value 4 (binary 100). This is because the corresponding bits in `z` and `y` are combined as follows:

<code>x</code>	0	0	0	0	1	1	0	1
<code>y</code>	0	0	0	0	0	1	1	0
<code>x & y</code>	0	0	0	0	0	1	0	0

Bitwise **OR** operator

```
int x = 13;
int y = 6;
int z = z | y;

printf("%d", z);
---
```

\$./run

15

This results in z containing the value 15 (binary 1111), because the bits combine as follows:

x	0	0	0	0	1	1	0	1
y	0	0	0	0	0	1	1	0
x y	0	0	0	0	1	1	1	1

Bitwise **XOR** operator

The bitwise **XOR** operator ^, produces a 1 if both bits are different and 0 if they are the same.

```
int x = 13;
int y = 6;
int z = x ^ y; // XOR corresponding bits of x and y
```

```
---  
$ ./run  
11
```

This results in z containing the value 11 (binary 1011), because the bits combine as follows:

x	0	0	0	0	1	1	0	1
y	0	0	0	0	0	1	1	0
x ^ y	0	0	0	0	1	0	1	1

Bitwise **NOT** operator

The unary operator, `~`, flips the bits of its operand, so 1 becomes 0 and 0 becomes 1. You could apply this operator to `x` with the value 13 as before:

```
int x = 13;  
int z = ~x;
```

x	0	0	0	0	1	1	0	1
~x	1	1	1	1	0	0	1	0

The value 1111 0010 is -14, in two's complement representation of negative integers.

Bitwise shift operator

Left

The left shift operators shift the bits in the left operand by the number of positions specified by the right operand.

```
int value = 12;
int shiftcount = 3;           // Number of positions to shift
int result = value << shiftcount; // Shift left shiftcount positions
```

- The variable result will contain the value `96`
- The binary number in value is `0000 1100`
- The bits are shifted to the left three positions, and 0s are introduced on the right, so the value of `value << shiftcount` as a binary number will be `0110 0000`

Right

The right shift operator moves the bits to the right, but it is a little more complicated than a left shift. For unsigned values, the bits that are introduced on the left (in the vacated positions as the bits are shifted right) are filled with zeros.

For signed values that are negative, the leftmost bit will be 1, and the result of a right shift depends on your system. In most cases, the sign bit is propagated, so the bits introduced on the right are 1 bits, but on some systems zeros are introduced in this case too.



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 (e.g., `3 >> -2`), it results in undefined behavior in C. If the number is shifted more than the size of the integer (e.g., `1 << 33`), the behavior is undefined.

Examples of using Bitwise operators

Add two numbers:

The bitwise OR of two numbers is just the sum of those two numbers **if there is no carry involved** otherwise you just add their bitwise AND.

```
int a = 5; //101
int b = 2; //010

printf("a + b = %d\n", a + b); //There is NO carry involved here
printf("a|b = %d\n", a|b);

a = 5;
b = 7;

printf("a + b = %d\n", a + b);
printf("a|b + a&b = %d\n", (a|b) + (a&b)); //There is carry involved

----
Running bitwiseex1.c...
a + b = 7
a|b = 7
```

```
a + b = 12  
a|b + a&b = 12
```

Swap two integers: This known trick is used to swap two integers by using only two integer variables. The beauty of it is that we don't need a temporary variable.

The most common solution for swapping variables is:

```
int temp = x;  
x = y;  
y = tmp;
```

You can do this with bitwise operators:

```
x ^= y;  
y = x ^ y;  
x ^= y;
```

How this works?

At start:

x = 0000 1101 /*original x = 13 in decimal */

y = 0000 0110 /*original y = 6 in decimal */

x = x ^ y

```
x = 0000 1101  
y = 0000 0110  
x = 0000 1011
```

$x = x \wedge y$

```
x = 0000 1011 /*this is the new x from the expression above *  
y = 0000 0110  
-----  
y = 0000 1101 /* original x */
```

$x = x \wedge y$

```
x = 0000 1011  
y = 0000 1101 /*this is the new y from the expression above  
-----  
x = 0000 0110 /* original y */
```

Program to Check if int is odd or even

```
#include <stdio.h>  
  
int main(int argc, char const *argv[])  
{  
    int x;  
    printf("Enter a number:");  
    scanf("%d", &x);  
  
    if (x & 1){  
        printf("Odd\n");  
    } else{  
        printf("Even\n");  
    }  
    return 0;  
}
```



```
-----  
% ./odddoreven.out  
Enter a number:2  
Even  
% ./odddoreven.out  
Enter a number:1  
Odd
```

Why is this good?

It is much quicker for the computer to do this, as it is good at working with bits, but if you use the modulus, it has a lot more work to do. Think about how you would do it as a human.