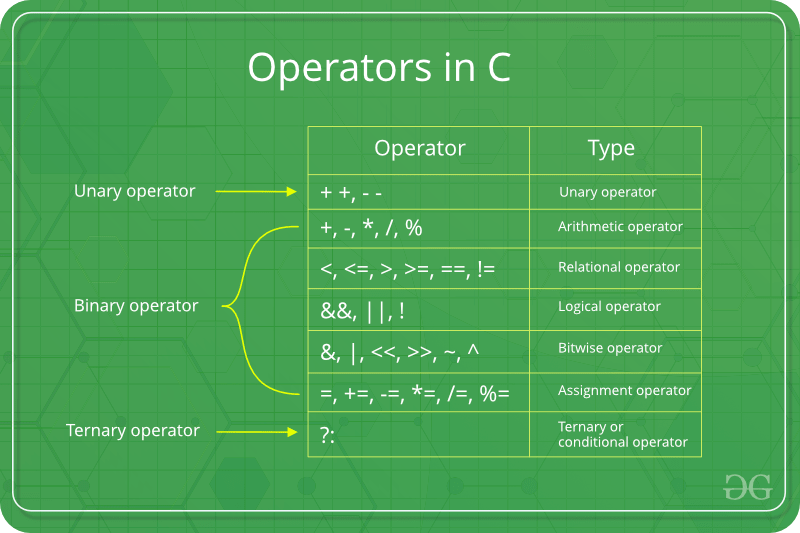
# **C - Operators**

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators −

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Bitwise Operators
* Assignment Operators
* Misc Operators

We will, in this chapter, look into the way each operator works.



## Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then −

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_arithmetic_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands. | A + B = 30 |
| − | Subtracts second operand from the first. | A − B = -10 |
| \* | Multiplies both operands. | A \* B = 200 |
| / | Divides numerator by de-numerator. | B / A = 2 |
| % | Modulus Operator and remainder of after an integer division. | B % A = 0 |
| ++ | Increment operator increases the integer value by one. | A++ = 11 |
| -- | Decrement operator decreases the integer value by one. | A-- = 9 |

// Working of arithmetic operators

#include <stdio.h>

int main()

{

int a = 9,b = 4, c;

c = a+b;

printf("a+b = %d \n",c);

c = a-b;

printf("a-b = %d \n",c);

c = a\*b;

printf("a\*b = %d \n",c);

c = a/b;

printf("a/b = %d \n",c);

c = a%b;

printf("Remainder when a divided by b = %d \n",c);

return 0;

}

**Output**

a+b = 13

a-b = 5

a\*b = 36

a/b = 2

Remainder when a divided by b=1

The operators +, - and \* computes addition, subtraction, and multiplication respectively as you might have expected.

In normal calculation, 9/4 = 2.25. However, the output is 2 in the program.

It is because both the variables *a* and b are integers. Hence, the output is also an integer. The compiler neglects the term after the decimal point and shows answer 2 instead of 2.25.

The modulo operator % computes the remainder. When a=9 is divided by b=4, the remainder is 1. The % operator can only be used with integers.

Suppose a = 5.0, b = 2.0, c = 5 and d = 2. Then in C programming,

// Either one of the operands is a floating-point number

a/b = 2.5

a/d = 2.5

c/b = 2.5

// Both operands are integers

c/d = 2

## C Increment and Decrement Operators:

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

### Example 2: Increment and Decrement Operators

// Working of increment and decrement operators

#include <stdio.h>

int main()

{

int a = 10, b = 100;

float c = 10.5, d = 100.5;

printf("++a = %d \n", ++a);

printf("--b = %d \n", --b);

printf("++c = %f \n", ++c);

printf("--d = %f \n", --d);

return 0;

}

**Output**

++a = 11

--b = 99

++c = 11.500000

++d = 99.500000

Here, the operators ++ and -- are used as prefixes. These two operators can also be used as postfixes like a++ and a--.

## Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 then −

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_relational_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the values of two operands are equal or not. If yes, then the condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true. | (A <= B) is true. |

### C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in [decision making](https://www.programiz.com/c-programming/c-if-else-statement) and [loops](https://www.programiz.com/c-programming/c-for-loop).

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning of Operator** | **Example** |
| == | Equal to | 5 == 3 is evaluated to 0 |
| > | Greater than | 5 > 3 is evaluated to 1 |
| < | Less than | 5 < 3 is evaluated to 0 |
| != | Not equal to | 5 != 3 is evaluated to 1 |
| >= | Greater than or equal to | 5 >= 3 is evaluated to 1 |
| <= | Less than or equal to | 5 <= 3 is evaluated to 0 |

### Example 4: Relational Operators

// Working of relational operators

#include <stdio.h>

int main()

{

int a = 5, b = 5, c = 10;

printf("%d == %d is %d \n", a, b, a == b);

printf("%d == %d is %d \n", a, c, a == c);

printf("%d > %d is %d \n", a, b, a > b);

printf("%d > %d is %d \n", a, c, a > c);

printf("%d < %d is %d \n", a, b, a < b);

printf("%d < %d is %d \n", a, c, a < c);

printf("%d != %d is %d \n", a, b, a != b);

printf("%d != %d is %d \n", a, c, a != c);

printf("%d >= %d is %d \n", a, b, a >= b);

printf("%d >= %d is %d \n", a, c, a >= c);

printf("%d <= %d is %d \n", a, b, a <= b);

printf("%d <= %d is %d \n", a, c, a <= c);

return 0;

}

**Output**

5 == 5 is 1

5 == 10 is 0

5 > 5 is 0

5 > 10 is 0

5 < 5 is 0

5 < 10 is 1

5 != 5 is 0

5 != 10 is 1

5 >= 5 is 1

5 >= 10 is 0

5 <= 5 is 1

5 <= 10 is 1

## Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then −

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_logical_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false. | !(A && B) is true. |

// Working of logical operators

#include <stdio.h>

int main()

{

int a = 5, b = 5, c = 10, result;

result = (a == b) && (c > b);

printf("(a == b) && (c > b) is %d \n", result);

result = (a == b) && (c < b);

printf("(a == b) && (c < b) is %d \n", result);

result = (a == b) || (c < b);

printf("(a == b) || (c < b) is %d \n", result);

result = (a != b) || (c < b);

printf("(a != b) || (c < b) is %d \n", result);

result = !(a != b);

printf("!(a == b) is %d \n", result);

result = !(a == b);

printf("!(a == b) is %d \n", result);

return 0;

}

**Output**

(a == b) && (c > b) is 1

(a == b) && (c < b) is 0

(a == b) || (c < b) is 1

(a != b) || (c < b) is 0

!(a != b) is 1

!(a == b) is 0

**Explanation of logical operator program**

* (a == b) && (c > 5) evaluates to 1 because both operands (a == b) and (c > b) is 1 (true).
* (a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
* (a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true).
* (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).
* !(a != b) evaluates to 1 because operand (a != b) is 0 (false). Hence, !(a != b) is 1 (true).
* !(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b) is 0 (false).

**Bitwise Operators**

Bitwise operators are used for manipulating a data at the bit level, also called as bit level programming. Bit-level programming mainly consists of 0 and 1. They are used in numerical computations to make the calculation process faster.

Following is the list of bitwise operators provided by 'C' programming language:

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| **&** | Bitwise AND operator |
| **|** | Bitwise OR operator |
| **^** | Bitwise exclusive OR operator |
| **~** | Binary One's Complement Operator is a unary operator |
| **<<** | Left shift operator |
| **>>** | Right shift operator |

Bitwise operators cannot be directly applied to primitive data types such as float, double, etc. Always remember one thing that bitwise operators are mostly used with the integer data type because of its compatibility.

The bitwise logical operators work on the data bit by bit, starting from the least significant bit, i.e. LSB bit which is the rightmost bit, working towards the MSB (Most Significant Bit) which is the leftmost bit.

The result of the computation of bitwise logical operators is shown in the table given below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

In this tutorial, you will learn-

* [What are Bitwise Operators?](https://www.guru99.com/c-bitwise-operators.html#1)
* [Bitwise AND](https://www.guru99.com/c-bitwise-operators.html#2)
* [Bitwise OR](https://www.guru99.com/c-bitwise-operators.html#3)
* [Bitwise Exclusive OR](https://www.guru99.com/c-bitwise-operators.html#4)
* [Bitwise shift operators](https://www.guru99.com/c-bitwise-operators.html#5)
* [Bitwise complement operator](https://www.guru99.com/c-bitwise-operators.html#6)

**Bitwise AND**

This is one of the most commonly used logical bitwise operators. It is represented by a single ampersand sign (&). Two integer expressions are written on each side of the (&) operator.

The result of the bitwise AND operation is 1 if both the bits have the value as 1; otherwise, the result is always 0.

Let us consider that we have 2 variables op1 and op2 with values as follows:

Op1 = 0000 1101

Op2 = 0001 1001

The result of the AND operation on variables op1 and op2 will be

Result = 0000 1001

As we can see, two variables are compared bit by bit. Whenever the value of a bit in both the variables is 1, then the result will be 1 or else 0.

**Bitwise OR**

It is represented by a single vertical bar sign (|). Two integer expressions are written on each side of the (|) operator.

The result of the bitwise OR operation is 1 if at least one of the expression has the value as 1; otherwise, the result is always 0.

Let us consider that we have 2 variables op1 and op2 with values as follows:

Op1 = 0000 1101

Op2 = 0001 1001

The result of the OR operation on variables op1 and op2 will be

Result = 0001 1101

As we can see, two variables are compared bit by bit. Whenever the value of a bit in one of the variables is 1, then the result will be 1 or else 0.

**Bitwise Exclusive OR**

It is represented by a symbol (^). Two integer expressions are written on each side of the (^) operator.

The result of the bitwise Exclusive-OR operation is 1 if only one of the expression has the value as 1; otherwise, the result is always 0.

Let us consider that we have 2 variables op1 and op2 with values as follows:

Op1 = 0000 1101

Op2 = 0001 1001

The result of the OR operation on variables op1 and op2 will be

Result = 0001 0100

As we can see, two variables are compared bit by bit. Whenever only one variable holds the value 1 then the result is 0 else 0 will be the result.

Let us write a simple program that demonstrates bitwise logical operators.

#include <stdio.h>

int main()

{

int a = 20; /\* 20 = 010100 \*/

int b = 21; /\* 21 = 010101 \*/

int c = 0;

c = a & b; /\* 20 = 010100 \*/

printf("AND - Value of c is %d\n", c );

c = a | b; /\* 21 = 010101 \*/

printf("OR - Value of c is %d\n", c );

c = a ^ b; /\* 1 = 0001 \*/

printf("Exclusive-OR - Value of c is %d\n", c );

getch();

}

Output:

AND - Value of c is 20

OR - Value of c is 21

Exclusive-OR - Value of c is 1

**Bitwise shift operators**

The bitwise shift operators are used to move/shift the bit patterns either to the left or right side. Left and right are two shift operators provided by 'C' which are represented as follows:

Operand << n (Left Shift)

Operand >> n (Right Shift)

Here,

* an operand is an integer expression on which we have to perform the shift operation.
* 'n' is the total number of bit positions that we have to shift in the integer expression.

The left shift operation will shift the 'n' number of bits to the left side. The leftmost bits in the expression will be popped out, and n bits with the value 0 will be filled on the right side.

The right shift operation will shift the 'n' number of bits to the right side. The rightmost 'n' bits in the expression will be popped out, and the value 0 will be filled on the left side.

Example: x is an integer expression with data 1111. After performing shift operation the result will be:

x << 2 (left shift) = 1111<<2 = 1100

x>>2 (right shift) = 1111>>2 = 0011

Shifts operators can be combined then it can be used to extract the data from the integer expression. Let us write a program to demonstrate the use of bitwise shift operators.

#include <stdio.h>

int main() {

int a = 20; /\* 20 = 010100 \*/

int c = 0;

c = a << 2; /\* 80 = 101000 \*/

printf("Left shift - Value of c is %d\n", c );

c = a >> 2; /\*05 = 000101 \*/

printf("Right shift - Value of c is %d\n", c );

return 0;

}

Output:

Left shift - Value of c is 80

Right shift - Value of c is 5

After performing the left shift operation the value will become 80 whose binary equivalent is 101000.

After performing the right shift operation, the value will become 5 whose binary equivalent is 000101.

**Bitwise complement operator**

The bitwise complement is also called as one's complement operator since it always takes only one value or an operand. It is a unary operator.

When we perform complement on any bits, all the 1's become 0's and vice versa.

If we have an integer expression that contains 0000 1111 then after performing bitwise complement operation the value will become 1111 0000.

Bitwise complement operator is denoted by symbol tilde (~).

Let us write a program that demonstrates the implementation of bitwise complement operator.

#include <stdio.h>

int main() {

int a = 10; /\* 10 = 1010 \*/

int c = 0;

c = ~(a);

printf("Complement - Value of c is %d\n", c );

return 0;

}

Output:

Complement - Value of c is -11

Here is another program, with an example of all the operatoes discussed so far:

#include <stdio.h>

main() {

unsigned int x = 48; /\* 48 = 0011 0000 \*/

unsigned int y = 13; /\* 13 = 0000 1101 \*/

int z = 0;

z =x & y; /\* 0 = 0000 0000 \*/

printf("Bitwise AND Operator - x & y = %d\n", z );

z = x | y; /\* 61 = 0011 1101 \*/

printf("Bitwise OR Operator - x | y = %d\n", z );

z= x^y; /\* 61 = 0011 1101 \*/

printf("Bitwise XOR Operator- x^y= %d\n", z);

z = ~x; /\*-61 = 1100 0011 \*/

printf("Bitwise One's Complement Operator - ~x = %d\n", z);

z = x << 2; /\* 192 = 1100 0000 \*/

printf("Bitwise Left Shift Operator x << 2= %d\n", z );

z= x >> 2; /\* 12 = 0000 1100 \*/

printf ("Bitwise Right Shift Operator x >> 2= %d\n", z );}

After we compile and run the program, it produces the following result:

Bitwise AND Operator - x & y = 0

Bitwise OR Operator - x | y = 61

Bitwise XOR Operator- x^y= 61

Bitwise One's Complement Operator - ~x = -49

Bitwise Left Shift Operator x << 2= 192

Bitwise Right Shift Operator x >> 2= 12

**Summary**

* Bitwise operators are special operator set provided by 'C.'
* They are used in bit level programming.
* These operators are used to manipulate bits of an integer expression.
* Logical, shift and complement are three types of bitwise operators.
* Bitwise complement operator is used to reverse the bits of an expression.

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Assume A = 60 and B = 13 in binary format, they will be as follows −

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_bitwise_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) = 12, i.e., 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) = 61, i.e., 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) = 49, i.e., 0011 0001 |
| ~ | Binary One's Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) = ~(60), i.e,. -0111101 |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 = 240 i.e., 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 = 15 i.e., 0000 1111 |

## Assignment Operators

The following table lists the assignment operators supported by the C language −

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_assignment_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator. Assigns values from right side operands to left side operand | C = A + B will assign the value of A + B to C |
| += | Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand. | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand. | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand. | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand. | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator. | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator. | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator. | C &= 2 is same as C = C & 2 |
| ^= | Bitwise exclusive OR and assignment operator. | C ^= 2 is same as C = C ^ 2 |
| |= | Bitwise inclusive OR and assignment operator. | C |= 2 is same as C = C | 2 |

// Working of assignment operators

#include <stdio.h>

int main()

{

int a = 5, c;

c = a; // c is 5

printf("c = %d\n", c);

c += a; // c is 10

printf("c = %d\n", c);

c -= a; // c is 5

printf("c = %d\n", c);

c \*= a; // c is 25

printf("c = %d\n", c);

c /= a; // c is 5

printf("c = %d\n", c);

c %= a; // c = 0

printf("c = %d\n", c);

return 0;

}

**Output**

c = 5

c = 10

c = 5

c = 25

c = 5

c = 0

## Misc Operators ↦ sizeof & ternary

Besides the operators discussed above, there are a few other important operators including **sizeof** and **? :** supported by the C Language.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| sizeof() | Returns the size of a variable. | sizeof(a), where a is integer, will return 4. |
| & | Returns the address of a variable. | &a; returns the actual address of the variable. |
| \* | Pointer to a variable. | \*a; |
| ? : | Conditional Expression. | If Condition is true ? then value X : otherwise value Y |

## sizeof Operator (Unary Operator):

The sizeof is a unary operator that returns the size of data (constants, variables, array, structure, etc).

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_sizeof_operator.htm)

#include <stdio.h>

int main()

{

int a;

float b;

double c;

char d;

printf("Size of int=%lu bytes\n",sizeof(a));

printf("Size of float=%lu bytes\n",sizeof(b));

printf("Size of double=%lu bytes\n",sizeof(c));

printf("Size of char=%lu byte\n",sizeof(d));

return 0;

}

**Output**

Size of int = 4 bytes

Size of float = 4 bytes

Size of double = 8 bytes

Size of char = 1 byte

## ? : Operator (Ternary Operator):

The ? : operator is just like an if ... else statement except that because it is an operator you can use it within expressions.

**? :** is a ternary operator in that it takes three values, this is the only ternary operator C has.

**? :** takes the following form:

|  |
| --- |
| if condition is true ? then X return value : otherwise Y |
| [Show Example](http://www.tutorialspoint.com/ansi_c/ternary_operator_examples.htm)   |  | | --- | | #include <stdio.h>  main()  {  int a , b;  a = 10;  printf( "Value of b is %d\n", (a == 1) ? 20: 30 );  printf( "Value of b is %d\n", (a == 10) ? 20: 30 );  } |   This will produce following result:   |  | | --- | | Value of b is 30  Value of b is 20 | |

## Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has a higher precedence than +, so it first gets multiplied with 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm)

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |