The assignment operators supported by the C language −

|  |  |  |
| --- | --- | --- |
| **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 |

Example

Try the following example to understand all the assignment operators available in C –

#include <stdio.h>

int main() {

int a = 21;

int c ;

c = a;

printf("Line 1 - = Operator Example, Value of c = %d\n", c );

c += a;

printf("Line 2 - += Operator Example, Value of c = %d\n", c );

c -= a;

printf("Line 3 - -= Operator Example, Value of c = %d\n", c );

c \*= a;

printf("Line 4 - \*= Operator Example, Value of c = %d\n", c );

c /= a;

printf("Line 5 - /= Operator Example, Value of c = %d\n", c );

c = 200;

c %= a;

printf("Line 6 - %= Operator Example, Value of c = %d\n", c );

c <<= 2;

printf("Line 7 - <<= Operator Example, Value of c = %d\n", c );

c >>= 2;

printf("Line 8 - >>= Operator Example, Value of c = %d\n", c );

c &= 2;

printf("Line 9 - &= Operator Example, Value of c = %d\n", c );

c ^= 2;

printf("Line 10 - ^= Operator Example, Value of c = %d\n", c );

c |= 2;

printf("Line 11 - |= Operator Example, Value of c = %d\n", c );

return 0;

}

When you compile and execute the above program, it produces the following result −

Line 1 - = Operator Example, Value of c = 21

Line 2 - += Operator Example, Value of c = 42

Line 3 - -= Operator Example, Value of c = 21

Line 4 - \*= Operator Example, Value of c = 441

Line 5 - /= Operator Example, Value of c = 21

Line 6 - %= Operator Example, Value of c = 11

Line 7 - <<= Operator Example, Value of c = 44

Line 8 - >>= Operator Example, Value of c = 11

Line 9 - &= Operator Example, Value of c = 2

Line 10 - ^= Operator Example, Value of c = 0

Line 11 - |= Operator Example, Value of c = 2

# **Operator Precedence**

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.

|  |  |  |
| --- | --- | --- |
| **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 |

## Example

example to understand operator precedence in C −

#include <stdio.h>

main() {

int a = 20;

int b = 10;

int c = 15;

int d = 5;

int e;

e = (a + b) \* c / d;

// ( 30 \* 15 ) / 5

printf("Value of (a + b) \* c / d is : %d\n", e );

e = ((a + b) \* c) / d; // (30 \* 15 ) / 5

printf("Value of ((a + b) \* c) / d is : %d\n" , e );

e = (a + b) \* (c / d); // (30) \* (15/5)

printf("Value of (a + b) \* (c / d) is : %d\n", e );

e = a + (b \* c) / d; // 20 + (150/5)

printf("Value of a + (b \* c) / d is : %d\n" , e );

return 0;

}

result −

Value of (a + b) \* c / d is : 90

Value of ((a + b) \* c) / d is : 90

Value of (a + b) \* (c / d) is : 90

Value of a + (b \* c) / d is : 50

## What are Unary Operators?

The operators which operates on single operand (i.e. to perform an operation through these operators, we need only one operand).

## List of Unary Operators in C programming language

Following are the unary operators in C and C++ programming language

|  |  |  |
| --- | --- | --- |
| **SrNo** | **Operators** | **Symbols** |
| 1 | Unary plus | + |
| 2 | Unary minus | - |
| 3 | Increment operator | ++ |
| 4 | Decrement operato | -- |
| 5 | Address of Operator | & |
| 6 | Size of Operator | sizeof() |
| 7 | Dereferencing Operator | \* |
| 8 | Logical NOT | ! |
| 9 | Bitwise NOT/ Bitwise Negation/ One's Compliment | ~ |

### **1) Unary plus (+) Operator**

This operator does not make any effect on the operand value, it just returns operands value.

**Consider the given example:**

#include <stdio.h>

**int** main()

{

**int** x= +4;

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

**return** 0;

}

Output

x= 4

*Here*, we assigned +4 to the variable x and the result is 4.

### **2) Unary minus (-) Operator**

This operator makes the value negative. It makes positive value to negative and negative value to positive.

**Consider the given example:**

#include <stdio.h>

**int** main()

{

**int** x=10;

**int** y=-20;

printf("value of -x: %d\n",-x);

printf("value of -y: %d\n",-y);

**return** 0;

}

Output

value of -x: -10

value of -y: 20

*Here*, we assigned 10 to variable x and -20 to variable y, when we print the value of both variables using **Unary minus operator**, the result is **-x= 10 and -y= 20**.

### **3) Increment (++) Operator**

This operator increases the value by 1; there are two varieties of increment operator 1) pre-increment operator and 2) post-increment operator.

**For example:** if we want to increase the value of variable x, the pre-increment operation will be written like ++x and post-increment operation will be written like x++.

### **4) Decrement (--) Operator**

This operator decreases the value by 1; there are two varieties of decrement operator 1) pre-decrement operator and 2) post-decrement operator.

**For example:** if we want to decrease the value of variable x, the pre-decrement operation will be written like --x and post-decrement operation will be written like x--.

### **5) Address of (&) Operator in C**

This operator returns address of any variable.

**Consider the given example**

#include <stdio.h>

**int** main()

{

**int** x=10;

printf("Value of x: %d\n",x);

printf("Address of x: %X\n",&x);

**return** 0;

}

Output

Value of x: 10

Address of x: F0C10E3C

### **6) sizeof() Operator**

This is a function like operator that returns the occupied size of any variable, object, constant etc, even this operator returns the size of any data type, literal etc.

#include <stdio.h>

**int** main()

{

**int** x=10;

printf("size of x : %d\n",**sizeof**(x));

printf("size of 10 : %d\n",**sizeof**(10));

printf("size of int : %d\n",**sizeof**(**int**));

printf("size of char : %d\n",**sizeof**(**char**));

printf("size of float: %d\n",**sizeof**(**float**));

**return** 0;

}

Output

size of x : 4

size of 10 : 4

size of int : 4

size of char : 1

size of float: 4

### **7) Dereferencing (\*) Operator**

This operator represents by character asterisk (\*), it is used to deference the value of any pointer variable.

Let suppose, there is a variable pointer variable ptr which has been initialised with the address of variable num and num holds value 10.

Then, to access the value of num using ptr, we use dereferencing operator (\*).

**Consider the given example**

#include <stdio.h>

**int** main()

{

**int** num=10;

**int** \*ptr=&num;

printf("Value of num is: %d\n",\*ptr);

**return** 0;

}

Output

Value of num is: 10

### **8) Logical NOT (!) operator**

This operator inverse the result of any expression, if expression's result is non zero it returns zero (0) and if expression's result is zero its returns 1.

**Consider the given example**

#include <stdio.h>

**int** main()

{

**int** x = !(0);

**int** y = !(1);

**int** z = !(10);

printf("x= %d, y= %d, z= %d\n",x,y,z);

**return** 0;

}

Output

x= 1, y= 0, z= 0

### **9) Bitwise NOT (~) Operator**

This operators also known as Bitwise negation and one’s compliment operator in C language, it is a Unary operator in C and C++, it converts (inverse) individual bits from 0 to 1 and 1 to 0.

**For example:** there is a variable x with value 0xAA (in binary 1010 1010), ~x will be 0x55 (in binary 0101 0101).

**Consider the given example**

#include <stdio.h>

**int** main()

{

**unsigned** **char** a=0xAA;

printf("Before negation value of a: %02X\n",a);

a=~a;

printf("After negation value of a: %02X\n",a);

**return** 0;

}

Output

Before negation value of a: AA

After negation value of a: 55