# Operator in C ++

An operator is an icon, which tells the compiler to perform specific mathematical and logical operations. C ++ provides many available operators.

An operator is an icon, which tells the compiler to perform specific mathematical and logical operations. C ++ provides many available operators, which are:

1. Arithmetic operator
2. Relational operator
3. Logical operators
4. Bit comparison operator
5. Assignment operator
6. Mixed operator

## Arithmetic operators in C ++

The table below lists the arithmetic operators supported by the C ++ language:

Suppose variable A holds value 10, variable B holds 20 then:

Operator Description Example + Add two operands A + B with the result 30 - Subtract the second operand from the first operand A - B resulting in -10 * Multiply the two operands A * B resulting in 200 / Division B / A result is 2% The balance taking B% A results in 0 ++ Increase operator (++), increase the operand value by adding an A ++ unit resulting in 11 - Reduce operator (- ), reduce the value of the outgoing operand by one unit A - the result is 9

### Increase operator (++)

The increment operator (++) adds an extra operand, and the operator decreases (-) minus one unit from the operand. Therefore:

```
x = x + 1;
```

```
la tuong tu
```

```
x ++;
```

Similarly:

```
x = x-1;
```

```
la tuong tu
```

```
x--;
```

Both the increment operator and the reduced operator have a prefixed or suffixed form after the operand. For example:

```
x = x + 1;

can be opened

++ x;  // This is a great time (prefix) available
```

or format:

```
x ++;  // is the big post (postfix) later
```

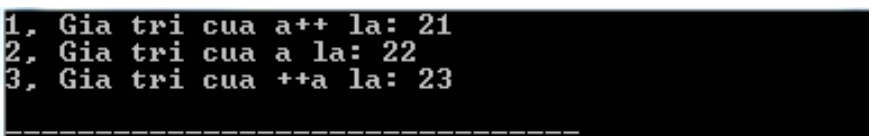
When an increase or decrease operator is used as part of an expression, there will be an important difference between the prefix and the suffix. If you use the prefix form, the increment operator or the reduction operator is performed before the expression, and if you use the suffix form, the increment or reduction operator is performed after the expression is evaluated.

**For example**

The following example explains this difference:

```
ch?a
using namespace std;

main ()
{
int a = 21;
int c;

// Family tri will not be able to use it when trying to learn.
c = a ++;
cout  "1, Gia tri a ++ la:"  c  endl;

// After the test is a test:
cout  "2, Gia tri is a la:"  a  endl;

// I will be able to learn more when I try to learn.
c = ++ a;
cout  "3, Tri tri ++ a la:"  c  endl;
return 0;
}
```

Running the above C ++ program will produce the following results:



## Operator decrease (-) in C ++

The increment operator (++) adds an extra operand, and the operator decreases (-) minus one unit from the operand. Therefore:

```
x = x + 1;

la tuong tu

x ++;
```

Similarly:

```
x = x-1;

la tuong tu

x--;
```

Both the increment operator and the reduced operator have a prefixed or suffixed form after the operand. For example:

```
x = x + 1;

can be opened

++ x;  // This is a great time (prefix) available
```

or format:

```
x ++;  // is the big post (postfix) later
```

When an increase or decrease operator is used as part of an expression, there will be an important difference between the prefix and the suffix. If you use the prefix form, the increment operator or the reduction operator is performed before the expression, and if you use the suffix form, the increment or reduction operator is performed after the expression is evaluated.

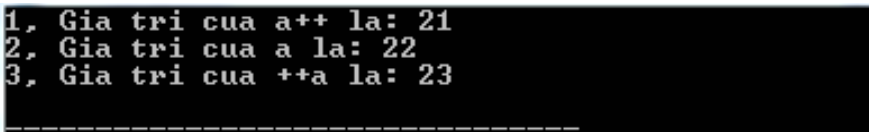
**For example**

The following example explains this difference:

```
ch?a
using namespace std;

main ()
{
int a = 21;
int c;

// Family tri will not be able to use it when trying to learn.
c = a ++;
cout  "1, Gia tri a ++ la:"  c  endl;
```

```
// After the test is a test:
cout  "2, Gia tri is a la:"  a  endl;

// I will be able to learn more when I try to learn.
c = ++ a;
cout  "3, Tri tri ++ a la:"  c  endl;
return 0;
}
```

Running the above C ++ program will produce the following results:




# Relational operator in C ++

The following table lists the relational operators supported by the C ++ language:

Suppose variable A holds value 10, variable B holds 20 then:

Operator Description Example == Check if 2 operands are equal or not. If equal, the condition is true. (A == B) is not correct! = Check for 2 different operands of different values. If not, the condition is true. (A! = B) is true> Check if the left operand is greater than the right operand. If larger, the condition is true. (A> B) is not correct If it is smaller then true. (A **= Check if the left operand has a value greater than or equal to the value of the right operand. If true, true. (A> = B) is incorrect = Check if the left operand is less than or equal to the right operand. If true, true. (A = B) is true**

# Boolean operators in C ++

The following table specifies all logical operators supported by the C language.

Suppose variable A has value 1 and variable B has value 0:

Operator Description Example && Called the logical AND operator (and). If both operators have values ??other than 0, the condition becomes true. (A && B) is false. || Called logical operators OR (or). If either operator is non-zero, then the condition is true. (A || B) is true. ! Called the NOT operator. Use to reverse the logic state of that operand. If the operand condition is true, the negative will be false. ! (A && B) is true.

# Bit comparison operator in C ++

The bit comparison operator works on bit units, calculates bitwise comparison expressions. The following table is about &, |, and ^ as follows:

pqp & qp | qp ^ q0 0 0 0 0 0 1 0 1 1 1 1 1 1 0 1 0 0 1 1
Suppose if A = 60; and B = 13; then 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

Bit comparison operators supported by the C ++ language are listed in the table below. The price we use has variable A having the value 60 and variable B having the value 13, we have:

Description Operator Example and binary AND (and) operator copy a bit to the result if it exists in both operands. (A & B) will result in 12, ie 0000 1100 | The binary OR operator (or) copies a bit to the result if it exists in one or two operands. (A | B) will result in 61, ie 0011 1101 ^ Binary copy XOR operator, which only exists in an operand and not both. (A ^ B) will result in 49, ie 0011 0001 ~ Bitwise operator (turn 1 bit into 0 bit and vice versa). (~ A) will result in -61, ie 1100 0011. Left shift operator. The left operand value is shifted left by the number of bits specified by the right operand. A 2 will give the result 240, ie 1111 0000 (left shift of two bits) >> Right shift operator. The left operand value is shifted right by the number of bits specified by the right operand. A >> 2 will result in 15, ie 0000 1111 (translate to two bits right)

# The assignment operator in C ++

Here are the assignment operators supported by the C ++ language:

Description OperatorExample = Simple assignment operator. Assign the right operand value to the left operand. C = A + B will assign the value of A + B to C + = Add the operand value to the left operand and assign that value to the left operand. C + = A is equivalent to C = C + A - = Subtract the right operand value from the left operand and assign this value to the left operand. C - = A is equivalent to C = C - A * = Multiply the right operand value by the left operand and assign this value to the left operand. C * = A is equivalent to C = C * A / = Divide the left operand for the right operand and assign this value to the left operand. C / = A is equivalent to C = C / A% = Take the remainder of the left operand division for the right operand and assign the left operand. C% = A is equivalent to C = C% A = Left shift of left-handed math to position number is the right operand value. C = 2 is equivalent to C = C 2 >> = Right shift of left operand to position number is the right operand value. C >> = 2 is equivalent to C = C >> 2 & = AND bit C & = 2 is equivalent to C = C & 2 ^ = OR operation excludes bit C ^ = 2 equivalent to C = C ^ 2 | = OR bit. C | = 2 is equivalent to C = C | 2

# Mixed operators in C ++

Here are some important mixed operators supported by the C ++ language.

Operator Description sizeof The sizeof operator in C ++ returns the size of a variable. For example, sizeof (a), with a being integer, will return 4 Conditions? X: Y Conditional operator in C ++. If Condition is true? then it returns an X value: otherwise it returns Y, the Comma operator in C ++ makes a sequence of operations performed. The value of the whole comma expression is the value of the last expression in the list separated by commas. (dot) and -> (arrow) The member operator in C ++ is used to reference single elements of classes, structures, and union Cast The cast operator (Casting) in C ++ transforms a data type other materials. For

example: int (2.2000) will return 2 & pointer operator & in C ++ returns the address of a variable. For example: & a; will return the actual address of this variable * The cursor operator * in C ++ is pointing to a variable. For example: * var will point to a var variable

## The sizeof operator in C ++

sizeof is a keyword in C ++, but it is a compile-time operator that determines the size, by byte value, of a variable or data type.

The sizeof operator can be used to get the size of the class, structure, union and any other self-defined data types (user-defined) in C ++.

The syntax for using the sizeof operator in C ++ is as follows:

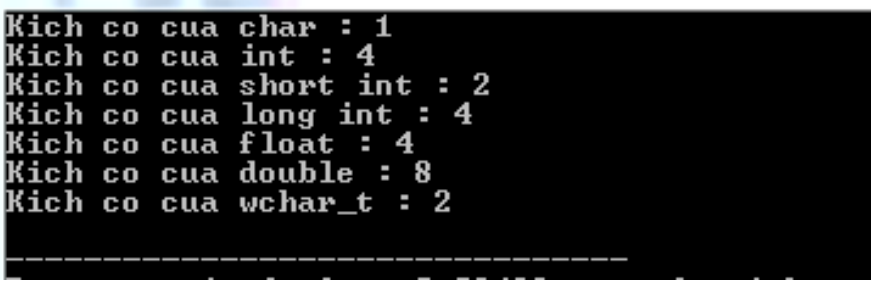
```
sizeof (travel guide)
```

Here, data type is a data type consisting of class, structure, union and any other self-defined data types (user-defined) in C ++.

Try the following example to understand how to use the sizeof operator in C ++. Copy and paste the following program in the test.cpp file, then compile and run the program.

```
ch?a
using namespace std;

int main ()
{
cout  "Kich co char:"  sizeof (char)  endl;
cout  "Kich has int:"  sizeof (int)  endl;
cout  "Short int:"  sizeof (short int)  endl;
cout  "Kich co int:"  sizeof (long int)  endl;
cout  "Kich has a float:"  sizeof (float)  endl;
cout  "Kich has double:"  sizeof (double)  endl;
cout  "Curious wchar_t:"  sizeof (wchar_t)  endl;
return 0;
}
```

Running the above C ++ program will produce the following results:




## Conditional operator? : in C ++

Syntax of conditional operator? : in C ++ is:

```
bieu_thuc_1?  bieu_thuc_2: bieu_thuc_3;
```

Here, bieu_thuc_1, bieu_thuc_2 and bieu_thuc_3 are expressions. Notice the use and position of the colon. Value of an expression? The decision is as follows: bieu_thuc_1 is estimated. If it is true, then bieu_thuc_2 is evaluated and becomes the value of the entire expression. If bieu_thuc_1 is false, then bieu_thuc_3 is evaluated and its value becomes the value of the expression?

Expression ? is treated as a trin operator because it requires three operands and can be used to replace the if-else statement, with the following form:

```
if (dieu_kien) {
var = X;
} else {
var = Y;
}
You consider the following code:

if (y 10) {
var = 30;
} else {
var = 40;
}
```

The above code can be rewritten like this:

```
var = (y 10)?  30: 40;
```

Here, x is assigned a value of 30 if y is less than 10 and is assigned 40 if not less than 10. You can try the following example:

```
#include
using namespace std;

int main ()
{
// How to open the package:
int x, y = 10;

x = (y 10)?  30: 40;

cout  "Family tri x la:"  x  endl;

return 0;
}
```

When the above code is compiled and executed, it gives the following result:

```
Family: 40
```

## The comma operator in C ++

The purpose of the comma operator in C ++ is to string some expressions together. The value of an expression list distinguished by commas is the value of the last right expression. Basically, the effect of commas is to make a sequence of operations to be performed.

The values ??of other expressions will be removed. That is, the last right expression will become the value of the entire expression distinguished by a comma. For example:

```
var = (biendem = 19, incr = 10, biendem + 1);
```

Here, first assign biendem value 19, assign increment value 10, then add 1 to biendem, and finally, assign var the value of the rightmost expression, the expression biendem + 1, is 20. **Parentheses** are necessary because the comma operator has a lower priority than the assignment operator.

To see the effect of the comma operator, try running the following example:

```
#include
using namespace std;

int main ()
{
int i, j;

j = 10;
i = (j ++, j + 100, 999 + j);

cout  i;

return 0;
}
```

When the above code is compiled and executed, it gives the following result:

```
1010
```

The procedure in which the value of i is estimated is: j starts with value 0. Then j is increased to 11. Then j is added to 100. Finally, j (still contains 11) is added to 999 , which will result in 1010.

## Member operator (. And ->) in C ++

Dot (.) And arrow operator (->) are used to reference individual members of the class, struct and union structures in C ++.

The dot operator is applied to the actual object. The arrow operator is used with a pointer to an object. For example, consider the following structure:

```
struct sinhvien {
char ten [16];
int diemthi;
} sv;
```

**The dot (.) Operator in C ++**

To assign a "wild" value to the **ten** member of the sinhvien object, write:

```
strcpy (sinhvien.ten, "hoang");
```

**Arrow operator (->) in C ++**

If p_sv is a pointer to an object of type sinhvien, then to assign a "wild" value to the ten member of the sinhvien object, write:

```
strcpy (p_sv-> ten, "wild");
```

It can be said simply: To access members of a structure, use the dot operator in C ++. To access the members of a structure through a pointer, use the arrow operator.

## Casting in C ++

The cast operator (a cast) in C ++ is a special operator that makes this type of data transform into another data type. The cast operator is a single-operator and has the same priority as any other unary operator in C ++.

The frequently used syntax of the cast operator in C ++ is:

```
(kieu_du_lieu) bieu_thuc
```

Here, kieu_du_lieu is the type of data you want. Here are some cast operators that are supported by C ++:

**const_cast (bieu_thuc)** : The const_cast operator is used to override const and / or volatile. The data type you want must be the same as the source data type except for the modification of the const or volatile properties in a cast. This casting type manipulates the const attribute of the transmitted object: either set or removed.

**dynamic_cast (bieu_thuc)** : The dynamic_cast operator in C ++ performs a casting at runtime that verifies the validity of cast. If the cast cannot be created, this cast fails and the estimated expression is null. A dynamic_cast operator performs castings on polymorphic types and can force an A * pointer to a B * pointer only if the object being pointed is actually an B object.

**reinterpret_cast (bieu_thuc)** : The reinterpret_cast operator in C ++ changes a pointer to any other pointer type. It also allows casting from pointers to an integer type and vice versa.

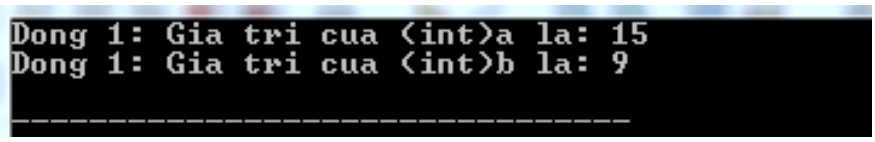
**static_cast (bieu_thuc)** : The static_cast operator in C ++ performs a non-polymorphic cast. For example, it can be used to cast a base class pointer to a legacy class pointer.

All of these cast operators will be used while working with classes and objects. Now, try the following example to understand a simple cast operator in C ++. Copy and paste the following C ++ program in the test.cpp file, then compile and run the program:

```
ch?a
using namespace std;

main ()
{
double a = 15.65653;
float b = 9.02;
```

```
int c;

c = (int) a;
cout  "Dong 1: Family member (int) a la:"  c  endl;

c = (int) b;
cout  "Dong 1: Family (int) b la:"  c  endl;

return 0;
}
```

Running the above C ++ program will produce the following results:




## The cursor operator in C ++

C ++ provides two pointer operators: the & operator (address of operator) and the * operator (indirection operator).

A pointer is a variable that contains the address of another variable or you can say that a variable containing the address of another variable is treated as a pointer to another variable. A variable can be any data type, including object, structure, or pointer itself.

Operator . (dot) and the -> (arrow) operator are used to refer to individual members of the class, structure, and union.

**Operator & in C ++**

The & operator is a unary operator in C ++ that returns the memory address of its operand. For example, if var is an integer variable, & var is its address. This operator has the same precedence and right-to-left order as other unary operators in C ++.

You should read the & operator as "address of", meaning that & var will be read as "var's address".

**Operator * in C ++**

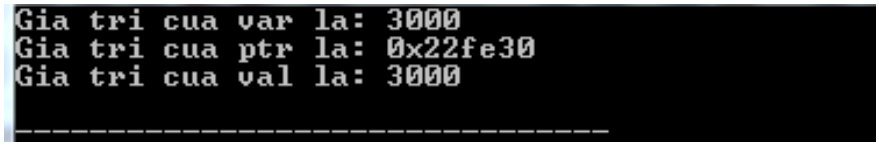
The second pointer operator is the * operator in C ++, and it is an addition to the & operator. This is a unary operator that returns the value of the variable located at the address specified by its operand.

The following is a program that illustrates two types of pointer operators in C ++:

```
#include

using namespace std;

int main ()
{
```

```
int var;
int * ptr;
int val;

var = 3000;

// divide the variable bien var
ptr = & var;

// Lay gia tri ptr
val = * ptr;
cout  "The list of var la:"  var  endl;
cout  "Family knowledge of ptr la:"  ptr  endl;
cout  "Family tri val la:"  val  endl;

return 0;
}
```

Running the above C ++ program will produce the following results:




## Priority order operator in C ++

The precedence of operator in C ++ determines how the expression is calculated. For example, the multiplication operator takes precedence over the addition operator, and it is done first.

For example, x = 7 + 3 * 2; Here, x is assigned a value of 13, not 20 because the * operator has a higher priority than the + operator, so first it performs multiplication 3 * 2 and then adds 7.

The following table lists the priority order of operators. Operators with the highest priority appear at the top of the table, and the operators with the lowest priority are at the bottom of the table. In an expression, the highest priority operators are first calculated.

Type EntriesPriority Order Postfix () [] ->. ++ - - Left to right Unary + -! ~ ++ - - (type) * & sizeof Right to left Multiplication * /% Left to right Addition + - Left to right Move >> Left to right Relationship =>> = Left to right Scales equals ==! = Left to right AND bit & Left to right XOR bit ^ Left to right OR bit | Left to right AND logic && Left to right Logical OR || Left to right Condition: Right to left Assign = + = - = * = / =% = >> = = & = ^ = | = Right to left Comma, Left to right
**According to Tutorialspoint**

Last lesson: Storage Class in C / C ++

Next article: Loop in C ++

you for reading and for following us regularly.