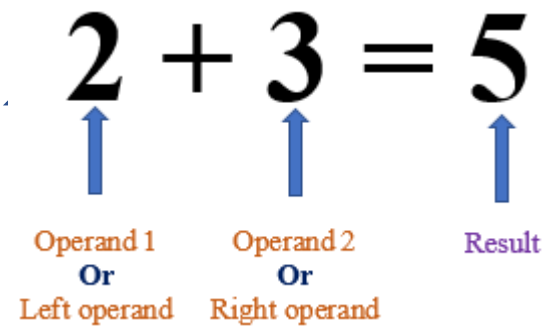
Operators and Expressions in C++

C++ provides operators for combining arithmetic, relational, logical, bitwise, and conditional expressions. Note that every operator must return some value. For example, + operator returns sum of two numbers, * operator return multiplication of two numbers etc.

Operator is a symbol that performs some operation. Simple operations may be addition, subtraction, multiplication, divison etc. So, operator is a symbol, which tells the compiler to do some operation or action.

For example,
2 + 3 = 5. Here the data 2 an 3 are known as operands. + is an operator, which perform summation of given numbers 2 and 3. The data 5 is the result of operation.



In above expression, the operator + operates on TWO operands. So, + is called binary operator. BI means 2. Rememeber bicycle has two wheels. Binacular has 2 eyes.

5 * 10 = 50. Here the data 5 an 10 are known as operands. * is an operator, which perform multiplication of given numbers 5 and 10. The data 50 is the result of operation.

z = x - y. Here the variables x an y are known as operands. - is an operator, which perform subtracton of given numbers x and y. The result of operation is stored in the variable, z.

List of some basic operators is given in below table:

Operator	Name	Example
+	Addition	12 + 4 // gives 16
-	Subtraction	3.98 – 2 // gives -1.98
*	Multiplication	5 * 3.2 // gives 16.0
/	Division	11 / 2 // gives 5.5
%	Remainder	10 % 4 //gives 2

Type of operators

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operators
- Pointer operators
- Special operators
- Bitwise operators

Other classification of operators

- Unary operators
- Binary operators
- Ternary operators

Unary operators: Some operators operate on single operand. They are called unary operators. Some examples are given below, which will be discussed later.

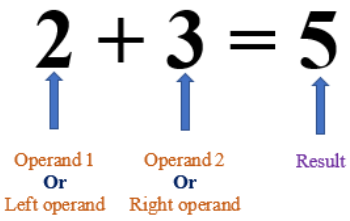
S. No.	Symbol	Meaning
1	!	Logical Not
2	~	One's complement
3	++	Increment
4	--	Decrement
5	-	Unary minus
6	*	Pointer operator
7	&	Address of operator
8	sizeof	Size of data type
9	(type)	Type conversion

Note:

For example, the symbol * can be used for two purposes. It can be used as multiplication operator. And also used as pointer operator. This is known as operator overloading. The process of making an operator to exhibit different behaviours in different instances (times) is known as operator overloading. Real life example of overloading is the use of word: "square". The word "square" has two meanings. When we say square of 5 = 25, the meaning is to find square of a number. Other meaning of a square is a geometric figure: SQUARE.

Binary operators: Some operators operate on two operands. They are called binary operators. Some examples are given below, which will be discussed later.

S. No.	Operator	Meaning
1	+	Addition
2	-	Subtraction
3	*	Multiplication
4	/	Division
5	%	Remainder

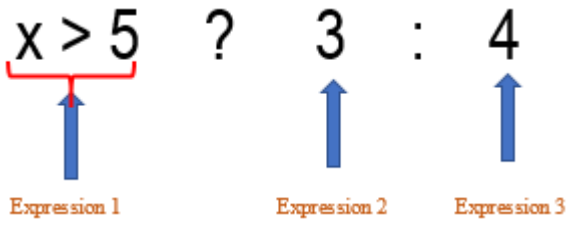


Ternary or conditional operators:

The conditional operators ? and : are called ternary operators since they take 3 arguments. Note that ternary operator takes 3 operands for its implementation.

Their general form is ⟶ expression 1 ? expression 2 : expression 3

See below statement for explanation.



What this expression says is: "if **expression 1** is true, then the value returned will be **expression 2**, otherwise the value returned will be **expression 3**".

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

int main()
{
    int x, y ;
    x = 50;
    y = ( x > 5 ? 3 : 4 ) ;

    cout<<y;

    return 0;
}
```

This program prints 3. Because x > 5 is true here.

Types of perators in c++

Arithmetic Operators

Arithmetic operators: +, -, *, /, %. These are binary operators as they operate on 2 operands at a time. Note that each of these operators can work with int, float or char.

Basic arithmetic operators provided by C++ are summarized in table below.

Operator	Name	Example
+	Addition	12 + 4 // gives 16
-	Subtraction	3.98 – 2 // gives -1.98
*	Multiplication	5 * 3.2 // gives 16.0
/	Division	11 / 2 // gives 5.5
%	Remainder	10 % 4 //gives 2

Operators +, -, *, / are self-explanatory.

Modulo division operator (%): This operator returns remainder after integer division. It is important to note that during modulo division, the sign of the result is always the sign of first operand (i.e., numerator). So, if you are interested to get remainder after division operation, we should use %.

Operation	Result	Remark
6 % 3	0	After dividing 6 with 3, we get remainder 0
3 % 3	0	After dividing 3 with 3, we get remainder 0
9 % 3	0	After dividing 9 with 3, we get remainder 0
5%3	2	After dividing 5 with 3, we get remainder 2
14 % 3	2	After dividing 14 with 3, we get remainder 2
-14% -3	-2	Numerator is -14. So, we get negative sign in result
3%5	3	We cannot divide 3 with 5. So, remainder is numerator itself.
-22% -6	-4	Numerator is -14. So, we get negative sign in result
10 % 3	1	After dividing 10 with 3, we get remainder 1

Points to remember

- Used with division function
- Modulo operator work with only int/char
- Use of % with float/double not allowed
- Sign of output is always same as the sign of numerator irrespective of the sign of the denominator

Logical Operators

Logical operators are used to combine two or more conditions in the program. There are 3 logical operators namely: NOT, AND, OR. Note that these operators yield BOOLEAN result. BOOLEAN means either true or False.

S. No.	Operator	Meaning	Example
1	!	Logical Negation	!(5 ==5) //returns FALSE
2	&&	Logical AND	5 < 6 && 6 < 6 // returns FALSE
3	||	Logical OR	5 < 6 || 6 < 5 // returns TRUE

Logical AND table:

Logical AND			
False	&&	False	False
False	&&	True	False
True	&&	False	False
True	&&	True	True

Logical OR table:

Logical OR			
False	||	False	False
False	||	True	True
True	||	False	True
True	||	True	True

Some examples are:

- !20 means 'not 20' ⟹ 'not True' ⟹ False. So, finally it returns FALSE
- 10 && 5 ⟹ True and True ⟹ TRUE
- 10 || 5.5 ⟹ True or True ⟹ TRUE
- 10 && -2 ⟹ True and False ⟹ FALSE

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

int main()
{
    int x = 125;
    int y;
    y = !x;

    cout<<y;

    return 0;
}
```

The ! negation operator converts a zero value to 1 and a non-zero value to zero.

!0 = 1
!25 = 0

Q. The marks obtained by a student in 5 different subjects are input through the keyboard. The student gets a division as per the following rules:

Percentage above or equal to 60 - First division
Percentage between 50 and 59 - Second division
Percentage between 40 and 49 - Third division
Percentage less than 40 – Fail

Write a program to calculate the division obtained by the student.

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

int main( )
{
int m1, m2, m3, m4, m5, percent ;

cout<<"Enter marks in 5 subjects"<<endl;
cin>>m1>>m2>>m3>>m4>>m5;

percent = ( m1 + m2 + m3 + m4 + m5 ) / 5 ;

if ( percent >= 60 )
cout<<"First division";

if ( ( percent >= 50 ) && ( percent < 60 ) )
cout<<"Second division";

if ( ( percent >= 40 ) && ( percent < 50 ) )
cout<<"Third division";

if ( percent < 40 )
cout<<"Fail";
}
```

Relational Operators

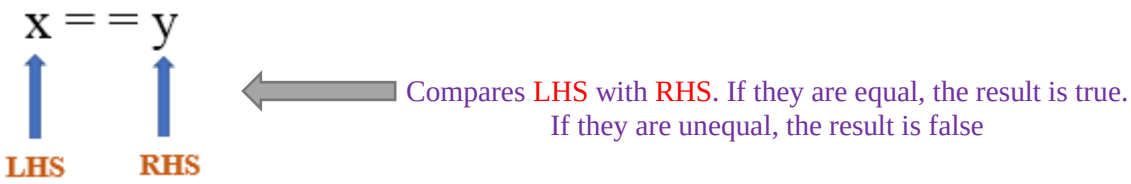
Relational operators are used to compare two quantities for big/small/equal etc. note that these quantities can be variables/constants/expressions. These operators compare two operands, so these are binary operators. Characters can also be compared as they are represented internally as ASCII codes (integers).

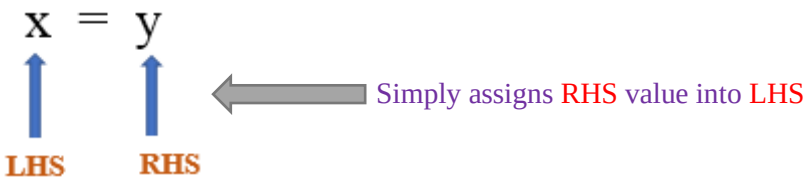
Relational operator returns either TRUE or FALSE. In C++, non-zero (+ve or –ve) value represents TRUE and zero represents FALSE. There are 6 relational operators, which are described in below table.

Operator	Meaning	Example
= =	Equality	5 == 5 // returns True
!=	Not Equal to	5 != 5 // returns False
<	Less Than	5 < 5.5 // returns True
<=	Less Than or Equal	5 <= 5 // returns True
>	Greater Than	5 > 5.5 // returns True
>=	Greater Than or Equal	6.3 >= 5 // returns True

For example, the statement x = = y; tests whether left-hand side (LHS) value and right-hand side (RHS) value are equal or not.

a == b; tests whether the value of a is equal to b
a = b; simply assigns b to a





Note:

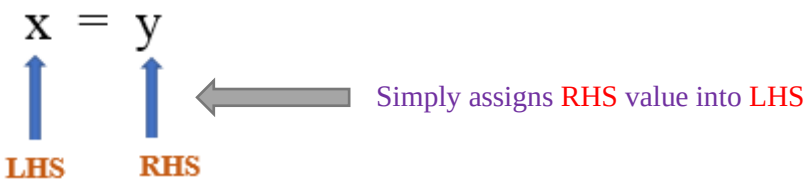
Compiler is ignorant to distinguish = and = = operators. So, this logical error should be taken care by user.

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

main()
{
 int i,j,k,m,n,t,r;
 i=8,j=13;
 k=i<j,  m=i+5>=j--,  n=++i==j%2+50,  t=i<=j-3;

 cout<<k<<m<<n<<t;
 }
```

```
F:\YOU-TUBE-TEACHING\web-layout\C++\W
1101
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Assignment Operator (=)

Used to assign right side value to the left side variable. For e.g. x =5; the value 5 is stored in variable x.

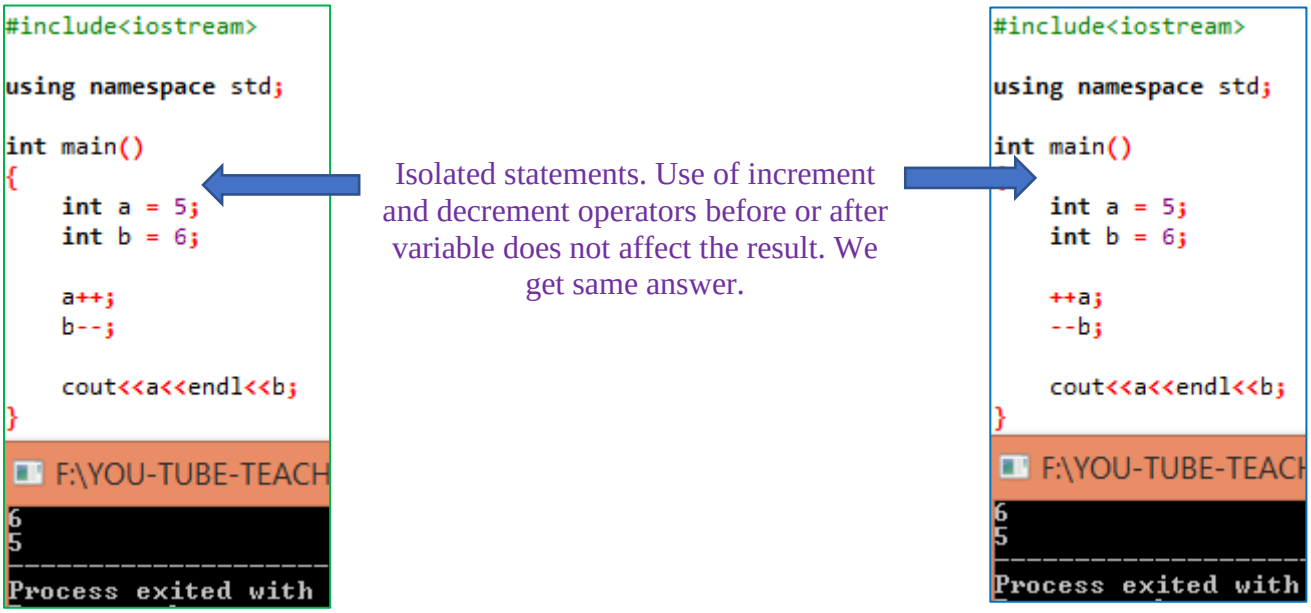


Increment/decrement Operators

The increment (++) and decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable. These are summarized in the following table. The examples assume the following variable definition:

```
int a = 5;
int b = 6
```

Operator	Meaning	Example
++	Increment	a++ // gives 6
--	Decrement	b-- // gives 5



Prefix & postfix operators have same effect if they are used in an isolated C++ statements. a++ and ++a are the same, if they are used in isolated statements.

Pre-increment: ++ x;	Increments the value of variable x by 1
Post-increment: x++;	Increments the value of variable x by 1
Pre-decrement: --x;	Decrements the value of variable x by 1
Post-decrement: x--;	Decrements the value of variable x by 1

In other words, the statements ++x or x++; is equivalent of the following statement: x = x + 1;
And --y or y--; is equivalent of the following statement: y = y -1.

Note: The increment and decrement operators can be used only with variables, not with constants.

Hence a++, x++ are legal. 6++, 10++ are invalid.

EXPRESSION	OUTPUTS
x=5 y=x++	x=6 y=5
x=5 y=++x	x=6 y=6
a=8 b=--a	a=7 b=7
a=8 b=a--	a=7 b=8
x=5 , y=8 z=x+y--	x=5 y=7 z=13
x=5 y=8 z=x+--y	x=5 y=7 z=12

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

int main( )
{
   int i,j,k;
   i = 3; j =7;
   k = i++ +  --j;

   cout<<i<<'\t'<<j<<'\t'<<k;
}
```

```
F:\YOU-TUBE-TEACHING\we
4       6       9
--------------------------
```

k = i++ + --j

initial value of i is used, because i++ is a post-increment. Value of j is decremented first, because –j is a pre-decrement operation. So, i = 4 j = 6 and k =9 get printed.

```
#include<iostream>

using namespace std;

int main()
{
    int a = 50;
    int b = 62;
    int x, y;

    x = a++;
    y = b--;

    cout<<a<<'\t'<<b<<endl;
    cout<<x<<'\t'<<y;
}
```

```
F:\YOU-TUBE-TEACHING\we
51      61
50      62
--------------------------
Process exited with return
```

These are compound statements. x = a++ means, initial value of a is assigned to x first. Then value of a will be incremented by 1. So, x = 50 and a = 51 will be printed. Similar explanation holds for statement: y = b--;

Prefix and postfix operators have different effects when used in association with some other operators in a C statement.

```
#include<iostream>

using namespace std;

int main()
{
    int a = 53;
    int b = 62;
    int x, y;

    x = ++a;
    y = --b;

    cout<<a<<'\t'<<b<<endl;
    cout<<x<<'\t'<<y;
}
```

```
F:\YOU-TUBE-TEACHING\we
54      61
54      61
--------------------------------
Process exited with return
```

These are compound statements. x = ++a means, initial value of a is incremented by 1 first and assigned to x. So, x = 54 and a = 54 will be printed. Similar explanation holds for statement: y = --b;

C++ offers increment (++) and decrement (- -) operators. ++ operator increments the value of the variable by 1. -- decrements the value of the variable by 1. These two are unary operators as they operate on only one operand.

Bitwise operators

These operators can operate upon **int**s and **char**s but not on **float**s and **double**s.

S. No.	Symbol	Meaning
1	~	1's complement
2	<<	Left shift
3	>>	Right shift
4	&	Bitwise AND
5	|	Bitwise OR
7	^	Exclusive OR

Bitwise AND: c=a & b: If the corresponding bits are both 1 then the resultant bit is 1 , otherwise 0.
Bitwise OR: c=a | b: If the corresponding bits are both 0 then the resultant bit is 0 , otherwise 1.
Exclusive OR: c=a ^ b: If the corresponding bits are both same then the resultant bit is 0 , otherwise 1.
1's Compliment: c=~a: 0 bit is converted to 1 and vice versa
Left Shift: c=a<<b: In the left shift operator(<<), b number of 0's are added on the right side.
Right Shift: c=a>>b: In the right shift operator(>>), the right most bit is removed and b number of 0's are added on the left side for right shift by 1.

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

#include<stdio.h>

int main()
{
    int a,b,c,d,e,f;
    a=197,b=153;

    c=a&b, d=a|b;

    e=a^b, f=~a;
    cout<<"c= "<<c<<'\t'<<"d= "<<d<<endl;

    cout<<"e= "<<e<<'\t'<<"f= "<<f;
}
```

```
F:\YOU-TUBE-TEACHING\web-layout\C
c= 129   d= 221
e= 92    f= -198
```

Some other special operators

Comma Operator

The comma operator is used between multiple expressions inside a set of parentheses. These expressions are then evaluated from left to right and the entire expression assumes the value of the last one evaluated.

For e.g. we can form an expression by separating two sub expressions with a comma. The result is as follows:

- Both expressions are evaluated, with the left expression being evaluated first.
- The entire expression evaluates to the value of the right most expression.

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

int main()
{
int i,j,k;

k=(i=4, j=5, i*j, i+j);

cout<<"k = "<<k;
```

```
F:\YOU-TUBE-TEACHI
k = 9
```

Address Operators

The address-of operator (&) returns the address of a given variable. For e.g. ptr = &x; &x is pronounced as ampersand x. &x return address of the variable x. The statement y = &x; stores address of variable x into ptr.

The indirection operator (*) returns the value of the memory location (variable) that it points to.
For e.g. z = *ptr; *ptr is pronounced as value at ptr memory location (variable). The statement z = *ptr; stores value at memory location ptr into variable z.

sizeof Operator

It returns the number of bytes of the given expression or operand or a data type.

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

int main()
{
    cout<<sizeof(int)<<endl;
    cout<<sizeof(float)<<endl;
    cout<<sizeof(char)<<endl;
    cout<<sizeof(double)<<endl;
    cout<<sizeof(long)<<endl;
}
```

```
F:\YOU-TUBE-TEACHING\web
4
4
1
8
4
```

The program is run on **Dev C++** compiler. Number of bytes occupied by various data types is:

integer: occupies 4 bytes
float: occupies 4 bytes
char: occupies 1 byte
double: occupies 8 bytes
long: occupies 4 bytes