OPERATORS, CONTROLS STRUCTURES, FUNCTIONS

Operators in C++

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

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

There are following arithmetic operators supported by C++ language −

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0

++	**Increment operator**, increases integer value by one	A++ will give 11
--	**Decrement operator**, decreases integer value by one	A-- will give 9

Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then 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 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 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 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 condition becomes true.	(A <= B) is true.

Logical Operators

There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then −

Show Examples

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

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are 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 if A = 60; and B = 13; now 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 Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the	(~A) will give -61 which is 1100 0011 in 2's complement form due

	effect of 'flipping' bits.	to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 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 will give 15 which is 0000 1111

Assignment Operators

There are following assignment operators supported by C++ language −

Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A

/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to 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

Misc Operators

The following table lists some other operators that C++ supports.

Sr.No	Operator & Description
1	**sizeof** **sizeof operator** returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
2	**Condition ? X : Y** **Conditional operator (?)**. If Condition is true then it returns value of X otherwise returns value of Y.

3	**,** **Comma operator** causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
4	**. (dot) and -> (arrow)** **Member operators** are used to reference individual members of classes, structures, and unions.
5	**Cast** **Casting operators** convert one data type to another. For example, int(2.2000) would return 2.
6	**&** **Pointer operator &** returns the address of a variable. For example &a; will give actual address of the variable.
7	***** **Pointer operator *** is pointer to a variable. For example *var; will pointer to a variable var.

Operators Precedence in C++

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

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has 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

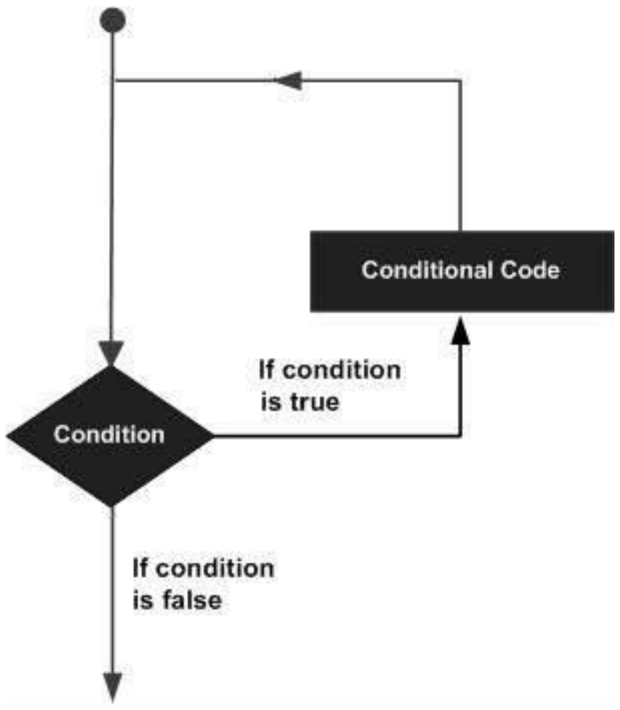
Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
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

C++ Loop Types

There may be a situation, when you need to execute a block of code several numbers of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages −



C++ programming language provides the following type of loops to handle looping requirements.

Sr.No	Loop Type & Description
1	**while loop** Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	**for loop**

	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**do...while loop** Like a 'while' statement, except that it tests the condition at the end of the loop body.
4	**nested loops** You can use one or more loop inside any another 'while', 'for' or 'do..while' loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements.

Sr.No	Control Statement & Description
1	**break statement** Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	**continue statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	**goto statement** Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the ‘for’ loop are required, you can make an endless loop by leaving the conditional expression empty.

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


int main () {
   for( ; ; ) {
      printf("This loop will run forever.\n");
   }


   return 0;
}
```

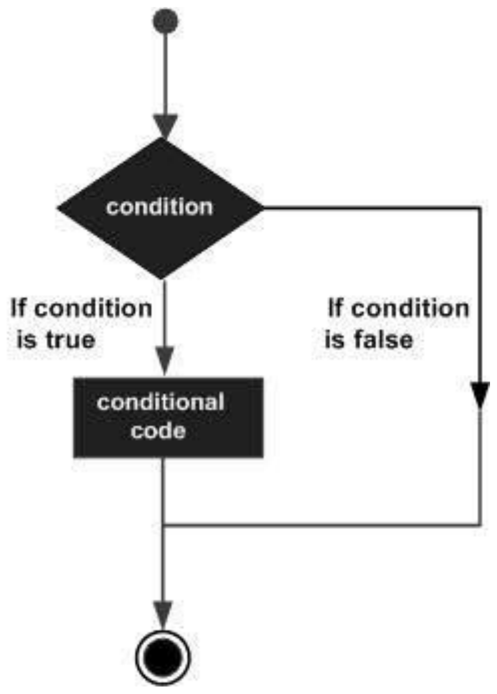
When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the ‘for (;;)’ construct to signify an infinite loop.

NOTE − You can terminate an infinite loop by pressing Ctrl + C keys.

C++ decision making statements

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



C++ programming language provides following types of decision making statements.

Sr.No	Statement & Description
1	**if statement** An 'if' statement consists of a boolean expression followed by one or more statements.
2	**if...else statement** An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.
3	**switch statement** A 'switch' statement allows a variable to be tested for equality against a list of values.
4	**nested if statements** You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).

5	**nested switch statements** You can use one 'switch' statement inside another 'switch' statement(s).

The ? : Operator

We have covered conditional operator "? :" in previous chapter which can be used to replace **if...else** statements. It has the following general form −

```
Exp1 ? Exp2 : Exp3;
```

Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a '?' expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire '?' expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

C++ Functions

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list ) {
   body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both −

```
// function returning the max between two numbers

int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration −

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration −

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example −

```
#include <iostream>

using namespace std;


// function declaration

int max(int num1, int num2);


int main () {

   // local variable declaration:

   int a = 100;

   int b = 200;

   int ret;


   // calling a function to get max value.
   ret = max(a, b);
   cout << "Max value is : " << ret << endl;

   return 0;
}

// function returning the max between two numbers
int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result −

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function −

Sr.No	Call Type & Description
1	**Call by Value** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	**Call by Pointer** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	**Call by Reference** This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example −

```
#include <iostream>

using namespace std;
```

```
int sum(int a, int b = 20) {

   int result;

   result = a + b;

   return (result);

}

int main () {

   // local variable declaration:

   int a = 100;

   int b = 200;

   int result;


   // calling a function to add the values.

   result = sum(a, b);

   cout << "Total value is :" << result << endl;


   // calling a function again as follows.

   result = sum(a);

   cout << "Total value is :" << result << endl;


   return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Total value is :300
Total value is :120
```

Numbers in C++

Normally, when we work with Numbers, we use primitive data types such as int, short, long, float and double, etc. The number data types, their possible values and number ranges have been explained while discussing C++ Data Types.

Defining Numbers in C++

You have already defined numbers in various examples given in previous chapters. Here is another consolidated example to define various types of numbers in C++ −

```
#include <iostream>

using namespace std;


int main () {

   // number definition:

   short  s;

   int    i;

   long   l;

   float  f;

   double d;


   // number assignments;

   s = 10;

   i = 1000;

   l = 1000000;

   f = 230.47;

   d = 30949.374;


   // number printing;

   cout << "short  s :" << s << endl;

   cout << "int    i :" << i << endl;

   cout << "long   l :" << l << endl;

   cout << "float  f :" << f << endl;

   cout << "double d :" << d << endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
short  s :10
int    i :1000
long   l :1000000
float  f :230.47
double d :30949.4
```

Math Operations in C++

In addition to the various functions you can create, C++ also includes some useful functions you can use. These functions are available in standard C and C++ libraries and called **built-in** functions. These are functions that can be included in your program and then use.

C++ has a rich set of mathematical operations, which can be performed on various numbers. Following table lists down some useful built-in mathematical functions available in C++.

To utilize these functions you need to include the math header file **<cmath>**.

Sr.No	Function & Purpose
1	**double cos(double);** This function takes an angle (as a double) and returns the cosine.
2	**double sin(double);** This function takes an angle (as a double) and returns the sine.
3	**double tan(double);** This function takes an angle (as a double) and returns the tangent.
4	**double log(double);** This function takes a number and returns the natural log of that number.

5	**double pow(double, double);** The first is a number you wish to raise and the second is the power you wish to raise it t
6	**double hypot(double, double);** If you pass this function the length of two sides of a right triangle, it will return you the length of the hypotenuse.
7	**double sqrt(double);** You pass this function a number and it gives you the square root.
8	**int abs(int);** This function returns the absolute value of an integer that is passed to it.
9	**double fabs(double);** This function returns the absolute value of any decimal number passed to it.
10	**double floor(double);** Finds the integer which is less than or equal to the argument passed to it.

Following is a simple example to show few of the mathematical operations −

```
#include <iostream>

#include <cmath>

using namespace std;


int main () {

   // number definition:

   short  s = 10;

   int    i = -1000;
   long   l = 100000;

   float  f = 230.47;

   double d = 200.374;


   // mathematical operations;

   cout << "sin(d) :" << sin(d) << endl;

   cout << "abs(i)  :" << abs(i) << endl;

   cout << "floor(d) :" << floor(d) << endl;

   cout << "sqrt(f) :" << sqrt(f) << endl;

   cout << "pow( d, 2) :" << pow(d, 2) << endl;


   return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
sign(d)     :-0.634939
abs(i)      :1000
floor(d)    :200
sqrt(f)     :15.1812
pow( d, 2 ) :40149.7
```

Random Numbers in C++

There are many cases where you will wish to generate a random number. There are actually two functions you will need to know about random number generation. The first is **rand()**, this function will only return a pseudo random number. The way to fix this is to first call the **srand()** function.

Following is a simple example to generate few random numbers. This example makes use of **time()** function to get the number of seconds on your system time, to randomly seed the rand() function −

```
#include <iostream>

#include <ctime>

#include <cstdlib>

```

```
using namespace std;


int main () {

   int i,j;


   // set the seed

   srand( (unsigned)time( NULL ) );


   /* generate 10  random numbers. */

   for( i = 0; i < 10; i++ ) {

      // generate actual random number

      j = rand();

      cout <<" Random Number : " << j << endl;

   }


   return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Random Number : 1748144778
Random Number : 630873888
Random Number : 2134540646
Random Number : 219404170
Random Number : 902129458
Random Number : 920445370
Random Number : 1319072661
Random Number : 257938873
Random Number : 1256201101
Random Number : 580322989
```