

What is C++

C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming.

C++ is a middle-level language, as it encapsulates both high and low level language features.

Object-Oriented Programming (OOPs)

C++ supports the object-oriented programming, the four major pillar of object-oriented programming (OOPs) used in C++ are:

1. Inheritance
2. Polymorphism
3. Encapsulation
4. Abstraction

C++ Standard Libraries

Standard C++ programming is divided into three important parts:

- The core library includes the data types, variables and literals, etc.
- The standard library includes the set of functions manipulating strings, files, etc.
- The Standard Template Library (STL) includes the set of methods manipulating a data structure.

Usage of C++

By the help of C++ programming language, we can develop different types of secured and robust applications:

- Window application
- Client-Server application
- Device drivers
- Embedded firmware etc

C++ Program

```

1. #include <iostream>
2. using namespace std;
3. int main() {
4.     cout << "Hello C++ Programming";
5.     return 0;
6. }

```

C vs. C++

What is C?

C is a structural or procedural oriented programming language which is machine-independent and extensively used in various applications.

C is the basic programming language that can be used to develop from the operating systems (like Windows) to complex programs like Oracle database, Git, Python interpreter, and many more. C programming language can be called a god's programming language as it forms the base for other programming languages. If we know the C language, then we can easily learn other programming languages. C language was developed by the great computer scientist Dennis Ritchie at the Bell Laboratories. It contains some additional features that make it unique from other programming languages.

What is C++?

C++ is a special-purpose programming language developed by **Bjarne Stroustrup** at Bell Labs circa 1980. C++ language is very similar to C language, and it is so compatible with C that it can run 99% of C programs without changing any source of code though C++ is an object-oriented programming language, so it is safer and well-structured programming language than C.

C vs. C++

Let's understand the differences between C and C

No.	C	C++
1)	C follows the procedural style programming.	C++ is multi-paradigm. It supports both procedural and object oriented.
2)	Data is less secured in C.	In C++, you can use modifiers for class

		members to make it inaccessible for outside users.
3)	C follows the top-down approach .	C++ follows the bottom-up approach .
4)	C does not support function overloading.	C++ supports function overloading.
5)	In C, you can't use functions in structure.	In C++, you can use functions in structure.
6)	C does not support reference variables.	C++ supports reference variables.
7)	In C, scanf() and printf() are mainly used for input/output.	C++ mainly uses stream cin and cout to perform input and output operations.
8)	Operator overloading is not possible in C.	Operator overloading is possible in C++.
9)	C programs are divided into procedures and modules	C++ programs are divided into functions and classes .
10)	C does not provide the feature of namespace.	C++ supports the feature of namespace.
11)	Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.
12)	C does not support the inheritance.	C++ supports inheritance.

C++ Program

1. `#include <iostream.h>`
2. `#include<conio.h>`
3. `void main() {`
4. `clrscr();`
5. `cout << "Welcome to C++ Programming.";`
6. `getch();`
7. `}`

#include<iostream.h> includes the **standard input output** library functions. It provides **cin** and **cout** methods for reading from input and writing to output respectively.

#include <conio.h> includes the **console input output** library functions. The `getch()` function is defined in `conio.h` file.

void main() The **main() function is the entry point of every program** in C++ language. The `void` keyword specifies that it returns no value.

cout << "Welcome to C++ Programming." is used to print the data **"Welcome to C++ Programming."** on the console.

getch() The `getch()` function **asks for a single character**. Until you press any key, it blocks the screen.

Namespace in C++

Consider following C++ program.

```
// A program to demonstrate need of namespace
```

```
int main()
{
    int value;
    value = 0;
    double value; // Error here
    value = 0.0;
}
```

Output :

Compiler Error:

'value' has a previous declaration as 'int value'

In each scope, a name can only represent one entity. So, there cannot be two variables with the same name in the same scope. Using namespaces, we can create two variables or member functions having the same name.

```
// Here we can see that more than one variables
```

```
// are being used without reporting any error.
```

```
// That is because they are declared in the
```

```
// different namespaces and scopes.
```

```
#include <iostream>
```

```
using namespace std;
```

```
// Variable created inside namespace
```

```
namespace first
```

```
{
```

```
    int val = 500;
```

```
}
```

```
// Global variable
```

```
int val = 100;
```

```
int main()
```

```
{
```

```
    // Local variable
```

```
    int val = 200;
```

```
// These variables can be accessed from
// outside the namespace using the scope
// operator ::
cout << first::val << '\n';

return 0;
}
```

Output:

500

C++ Basic Input/Output

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as **output operation**.

If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as **input operation**.

Standard output stream (cout)

It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console

Let's see the simple example of standard output stream (cout):

1. #include <iostream>
2. **using namespace** std;
3. **int** main() {
4. **char** ary[] = "Welcome to C++ tutorial";
5. cout << "Value of ary is: " << ary << endl;
6. }

Output:

```
Value of ary is: Welcome to C++ tutorial
```

Standard input stream (cin)

It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

Let's see the simple example of standard input stream (cin):

```
1. #include <iostream>
2. using namespace std;
3. int main( ) {
4.     int age;
5.     cout << "Enter your age: ";
6.     cin >> age;
7.     cout << "Your age is: " << age << endl;
8. }
```

Output:

```
Enter your age: 22
Your age is: 22
```

Standard end line (endl)

It is used to insert a new line characters and flushes the stream.

Let's see the simple example of standard end line (endl):

```
1. #include <iostream>
2. using namespace std;
3. int main( ) {
4.     cout << "C++ Tutorial";
5.     cout << " Javatpoint"<<endl;
6.     cout << "End of line"<<endl;
7. }
```

Output:

```
C++ Tutorial Javatpoint
```

End of line

C++ Identifiers

All C++ **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code:

Example

```
#include <iostream>

using namespace std;

int main() {

    // Good name
    int minutesPerHour = 60;

    // OK, but not so easy to understand what m actually is
    int m = 60;

    cout << minutesPerHour << "\n";

    cout << m;

    return 0;

}
```

OUTPUT:

60

6

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits and underscores
- Names must begin with a letter or an underscore (`_`)
- Names are case sensitive (`myVar` and `myvar` are different variables)
- Names cannot contain whitespaces or special characters like `!`, `#`, `%`, etc.
- Reserved words (like C++ keywords, such as `int`) cannot be used as names

C++ Data Types

As explained in the [Variables](#) chapter, a variable in C++ must be a specified data type:

Example

```
int myNum = 5;           // Integer (whole number)
float myFloatNum = 5.99; // Floating point number
double myDoubleNum = 9.98; // Floating point number
char myLetter = 'D';    // Character
bool myBoolean = true;  // Boolean
string myText = "Hello"; // String
```

Basic Data Types

The data type specifies the size and type of information the variable will store:

Data Type	Size	Description
<code>int</code>	4 bytes	Stores whole numbers, without decimals

float	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 7 decimal digits
double	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits
boolean	1 byte	Stores true or false values
char	1 byte	Stores a single character/letter/number, or ASCII values

C++ Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

1. type variable_list;

The example of declaring variable is given below:

1. **int** x;
2. **float** y;
3. **char** z;

Here, x, y, z are variables and int, float, char are data types.

We can also provide values while declaring the variables as given below:

1. **int** x=5,b=10; //declaring 2 variable of integer type
2. **float** f=30.8;
3. **char** c='A';

Rules for defining variables

A variable can have alphabets, digits and underscore.

A variable name can start with alphabet and underscore only. It can't start with digit.

No white space is allowed within variable name.

A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

1. **int** a;
2. **int** _ab;
3. **int** a30;

Invalid variable names:

1. **int** 4;
2. **int** x y;
3. **int double**;

Constants

When you do not want others (or yourself) to override existing variable values, use the const keyword (this will declare the variable as "constant", which means **unchangeable and read-only**):

Example

1. **const** int myNum = 15; // myNum will always be 15
myNum = 10; // error: assignment of read-only variable 'myNum'

2. #include <iostream>

using namespace std;

```
int main() {
```

```
const int minutesPerHour = 60;

const float PI = 3.14;

cout << minutesPerHour << "\n";

cout << PI;

return 0;

}
```

Output:

```
60

3.14
```

Operators

Once introduced to variables and constants, we can begin to operate with them by using *operators*. What follows is a complete list of operators. At this point, it is likely not necessary to know all of them, but they are all listed here to also serve as reference.

Assignment operator (=)

The assignment operator assigns a value to a variable.

```
x = 5;
```

This statement assigns the integer value 5 to the variable x. The assignment operation always takes place from right to left, and never the other way around:

```
x = y;
```

This statement assigns to variable x the value contained in variable y. The value of x at the moment this statement is executed is lost and replaced by the value of y.

Consider also that we are only assigning the value of y to x at the moment of the assignment operation. Therefore, if y changes at a later moment, it will not affect the new value taken by x.

For example, let's have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator
#include <iostream>
using namespace std;

int main ()
{
    int a, b;      // a:?, b:?
    a = 10;       // a:10, b:?
    b = 4;        // a:10, b:4
    a = b;        // a:4, b:4
    b = 7;        // a:4, b:7

    cout << "a:";
    cout << a;
    cout << " b:";
    cout << b;
}
```

This program prints on screen the final values of a and b (4 and 7, respectively). Notice how a was not affected by the final modification of b, even though we declared `a = b` earlier.

Assignment operations are expressions that can be evaluated. That means that the assignment itself has a value, and -for fundamental types- this value is the one assigned in the operation. For example:

```
y = 2 + (x = 5);
```

In this expression, y is assigned the result of adding 2 and the value of another assignment expression (which has itself a value of 5). It is roughly equivalent to:

```
1 x = 5;
2 y = 2 + x;
```

With the final result of assigning 7 to y.

The following expression is also valid in C++:

```
x = y = z = 5;
```

It assigns 5 to the all three variables: x, y and z; always from right-to-left.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by C++ are:

operator	description
+	addition
-	subtraction
*	multiplication
/	division
%	modulo

Operations of addition, subtraction, multiplication and division correspond literally to their respective mathematical operators. The last one, *modulo operator*, represented by a percentage sign (%), gives the remainder of a division of two values. For example:

```
x = 11 % 3;
```

results in variable x containing the value 2, since dividing 11 by 3 results in 3, with a remainder of 2.

Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

Compound assignment operators modify the current value of a variable by performing an operation on it. They are equivalent to assigning the result of an operation to the first operand:

expression	equivalent to...
y += x;	y = y + x;

<code>x -= 5;</code>	<code>x = x - 5;</code>
<code>x /= y;</code>	<code>x = x / y;</code>
<code>price *= units + 1;</code>	<code>price = price * (units+1);</code>

and the same for all other compound assignment operators. For example:

```
// compound assignment operators
#include <iostream>
using namespace std;

int main ()
{
    int a, b=3;
    a = b;
    a+=2;      // equivalent to a=a+2
    cout << a;
}
```

Increment and decrement (++ , --)

Some expression can be shortened even more: the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

```
1 ++x;
2 x+=1;
3 x=x+1;
```

are all equivalent in its functionality; the three of them increase by one the value of x.

In the early C compilers, the three previous expressions may have produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally performed automatically by the compiler, thus the three expressions should produce exactly the same executable code.

A peculiarity of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable name (++x) or after it (x++). Although in simple expressions like x++ or ++x, both have exactly the same meaning; in other expressions in which

the result of the increment or decrement operation is evaluated, they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++x) of the value, the expression evaluates to the final value of x, once it is already increased. On the other hand, in case that it is used as a suffix (x++), the value is also increased, but the expression evaluates to the value that x had before being increased. Notice the difference:

Example 1	Example 2
<pre>x = 3; y = ++x; // x contains 4, y contains 4</pre>	<pre>x = 3; y = x++; // x contains 4, y contains 3</pre>

In *Example 1*, the value assigned to y is the value of x after being increased. While in *Example 2*, it is the value x had before being increased.

Relational and comparison operators (==, !=, >, <, >=, <=)

Two expressions can be compared using relational and equality operators. For example, to know if two values are equal or if one is greater than the other.

The result of such an operation is either true or false (i.e., a Boolean value).

The relational operators in C++ are:

operator	description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Here there are some examples:


```
1 (7 == 5) // evaluates to false
2 (5 > 4) // evaluates to true
3 (3 != 2) // evaluates to true
4 (6 >= 6) // evaluates to true
5 (5 < 5) // evaluates to false
```

Of course, it's not just numeric constants that can be compared, but just any value, including, of course, variables. Suppose that a=2, b=3 and c=6, then:

```
1 (a == 5) // evaluates to false, since a is not equal to 5
2 (a*b >= c) // evaluates to true, since (2*3 >= 6) is true
3 (b+4 > a*c) // evaluates to false, since (3+4 > 2*6) is false
4 ((b=2) == a) // evaluates to true
```

Be careful! The assignment operator (operator =, with one equal sign) is not the same as the equality comparison operator (operator ==, with two equal signs); the first one (=) assigns the value on the right-hand to the variable on its left, while the other (==) compares whether the values on both sides of the operator are equal. Therefore, in the last expression ((b=2) == a), we first assigned the value 2 to b and then we compared it to a (that also stores the value 2), yielding true.

Logical operators (!, &&, ||)

The operator ! is the C++ operator for the Boolean operation NOT. It has only one operand, to its right, and inverts it, producing false if its operand is true, and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

```
1 !(5 == 5) // evaluates to false because the expression at its right (5 == 5) is true
2 !(6 <= 4) // evaluates to true because (6 <= 4) would be false
3 !true // evaluates to false
4 !false // evaluates to true
```

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds to the Boolean logical operation AND, which yields true if both its operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a&&b:

&& OPERATOR (and)

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

The operator `&&` corresponds to the Boolean logical operation OR, which yields true if either of its operands is true, thus being false only when both operands are false. Here are the possible results of `a||b`:

OPERATOR (or)		
a	b	a b
true	true	true
true	false	true
false	true	true
false	false	false

For example:

- 1 `((5 == 5) && (3 > 6)) // evaluates to false (true && false)`
- 2 `((5 == 5) || (3 > 6)) // evaluates to true (true || false)`

When using the logical operators, C++ only evaluates what is necessary from left to right to come up with the combined relational result, ignoring the rest. Therefore, in the last example `((5==5)||(3>6))`, C++ evaluates first whether `5==5` is true, and if so, it never checks whether `3>6` is true or not. This is known as *short-circuit evaluation*, and works like this for these operators:

operator	short-circuit
<code>&&</code>	if the left-hand side expression is false, the combined result is false (the right-hand side expression is never evaluated).

	if the left-hand side expression is true, the combined result is true (the right-hand side expression is never evaluated).
--	--

This is mostly important when the right-hand expression has side effects, such as altering values:

```
if ( (i<10) && (++i<n) ) { /*...*/ } // note that the condition increments i
```

Here, the combined conditional expression would increase *i* by one, but only if the condition on the left of `&&` is true, because otherwise, the condition on the right-hand side (`++i<n`) is never evaluated.

Conditional ternary operator (?)

The conditional operator evaluates an expression, returning one value if that expression evaluates to true, and a different one if the expression evaluates as false. Its syntax is:

```
condition ? result1 : result2
```

If condition is true, the entire expression evaluates to result1, and otherwise to result2.

```
1 7==5 ? 4 : 3 // evaluates to 3, since 7 is not equal to 5.
2 7==5+2 ? 4 : 3 // evaluates to 4, since 7 is equal to 5+2.
3 5>3 ? a : b // evaluates to the value of a, since 5 is greater than 3.
4 a>b ? a : b // evaluates to whichever is greater, a or b.
```

For example:

```
// conditional operator
#include <iostream>
using namespace std;

int main ()
{
    int a,b,c;

    a=2;
    b=7;
    c = (a>b) ? a : b;
```

```
cout << c << '\n';  
}
```

In this example, a was 2, and b was 7, so the expression being evaluated ($a > b$) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was b (with a value of 7).

Comma operator (,)

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the right-most expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```

would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

Bitwise operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

operator	asm equivalent	description
&	AND	Bitwise AND
	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift bits left
>>	SHR	Shift bits right

Explicit type casting operator

Type casting operators allow to convert a value of a given type to another type. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
1 int i;  
2 float f = 3.14;  
3 i = (int) f;
```

The previous code converts the floating-point number 3.14 to an integer value (3); the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is to use the functional notation preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int (f);
```

Both ways of casting types are valid in C++.

sizeof

This operator accepts one parameter, which can be either a type or a variable, and returns the size in bytes of that type or object:

```
x = sizeof (char);
```

Here, x is assigned the value 1, because char is a type with a size of one byte.

The value returned by sizeof is a compile-time constant, so it is always determined before program execution.

Scope resolution operator

The :: (scope resolution) operator is used to get hidden names due to variable scopes so that you can still use them. The scope resolution operator can be used as both unary and binary. You can

use the unary scope operator if a namespace scope or global scope name is hidden by a particular declaration of an equivalent name during a block or class. For example, if you have a global variable of name `my_var` and a local variable of name `my_var`, to access global `my_var`, you'll need to use the scope resolution operator.

example

```
#include <iostream>

using namespace std;

int my_var = 0;

int main(void) {
    int my_var = 0;
    ::my_var = 1; // set global my_var to 1
    my_var = 2;  // set local my_var to 2
    cout << ::my_var << ", " << my_var;
    return 0;
}
```

Output

This will give the output –

```
1, 2
```

Type Conversion in C++

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. Implicit Type Conversion

Also known as ‘automatic type conversion’.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.

- All the data types of the variables are upgraded to the data type of the variable with largest data type.

```
bool -> char -> short int -> int ->
```

```
unsigned int -> long -> unsigned ->
```

```
long long -> float -> double -> long double
```

Example of Type Implicit Conversion:

```
// An example of implicit conversion
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int x = 10; // integer x
```

```
    char y = 'a'; // character c
```

```
    // y implicitly converted to int. ASCII
```

```
    // value of 'a' is 97
```

```
    x = x + y;
```

```
    // x is implicitly converted to float
```

```
    float z = x + 1.0;
```

```
cout << "x = " << x << endl
    << "y = " << y << endl
    << "z = " << z << endl;

return 0;
}
```

Output:

```
x = 107
y = a
z = 108
```

2. **Explicit Type Conversion:** This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

Syntax:

```
(type) expression
```

where *type* indicates the data type to which the final result is converted.

Example:

```
// C++ program to demonstrate
// explicit type casting

#include <iostream>

using namespace std;

int main()
```



```
{  
    double x = 1.2;  
  
    // Explicit conversion from double to int  
    int sum = (int)x + 1;  
  
    cout << "Sum = " << sum;  
  
    return 0;  
}
```

Output:

```
Sum = 2
```

Statements and flow control

C++ if-else

In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.

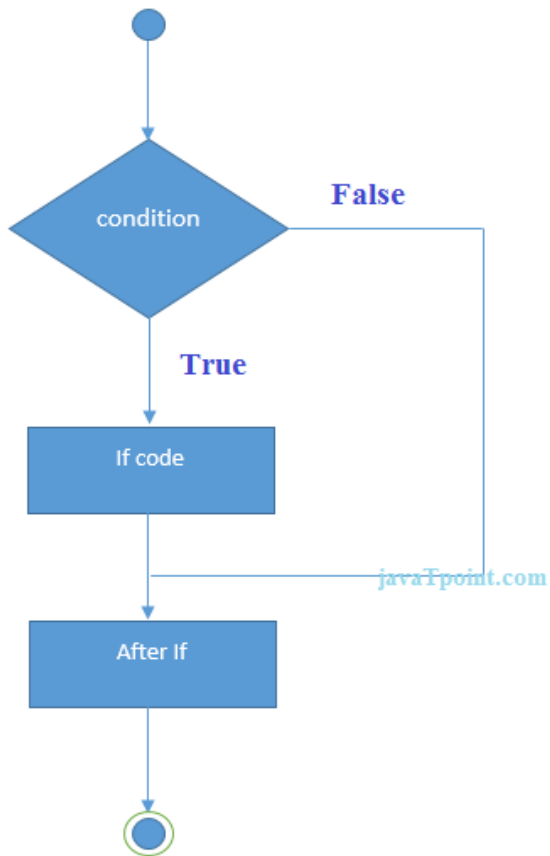
- if statement
- if-else statement
- nested if statement
- if-else-if ladder

C++ IF Statement

The C++ if statement tests the condition. It is executed if condition is true.

1. **if**(condition){

2. //code to be executed
3. }



C++ If Example

1. #include <iostream>
2. using namespace std;
- 3.
4. int main () {
5. int num = 10;
6. if (num % 2 == 0)
7. {
8. cout<<"It is even number";
9. }
10. return 0;
11. }

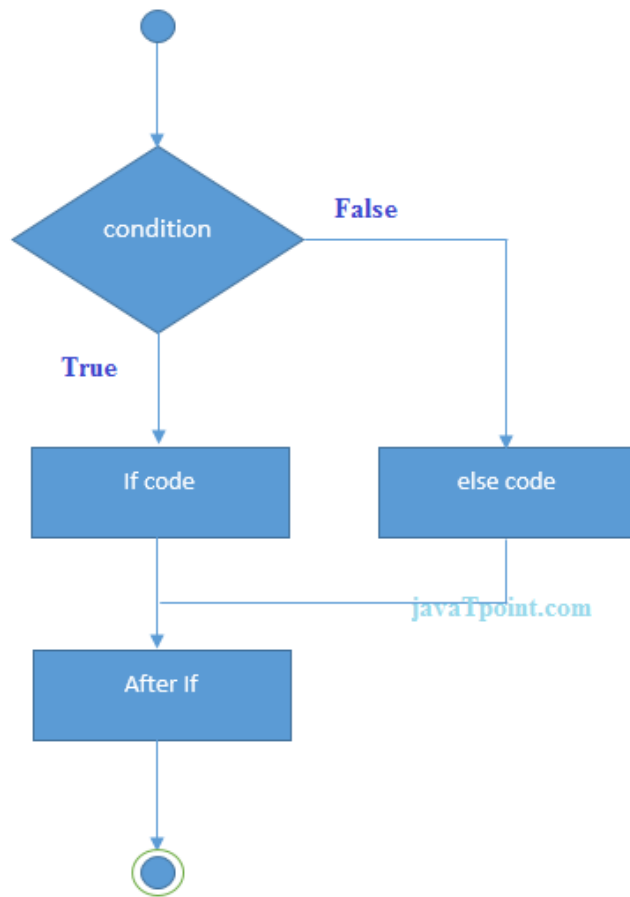
Output:

It is even number

C++ IF-else Statement

The C++ if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

1. **if**(condition){
2. //code if condition is true
3. }**else**{
4. //code if condition is false
5. }



C++ If-else Example

1. `#include <iostream>`
2. `using namespace std;`
3. `int main () {`

```
4.  int num = 11;
5.      if (num % 2 == 0)
6.      {
7.          cout<<"It is even number";
8.      }
9.      else
10.     {
11.         cout<<"It is odd number";
12.     }
13.  return 0;
14. }
```

Output:

```
It is odd number
```

C++ If-else Example: with input from user

```
1. #include <iostream>
2.  using namespace std;
3.  int main () {
4.      int num;
5.      cout<<"Enter a Number: ";
6.      cin>>num;
7.          if (num % 2 == 0)
8.          {
9.              cout<<"It is even number"<<endl;
10.         }
11.         else
12.         {
13.             cout<<"It is odd number"<<endl;
14.         }
15.  return 0;
16. }
```

Output:

```
Enter a number:11
It is odd number
```

Output:

Enter a number:12
It is even number

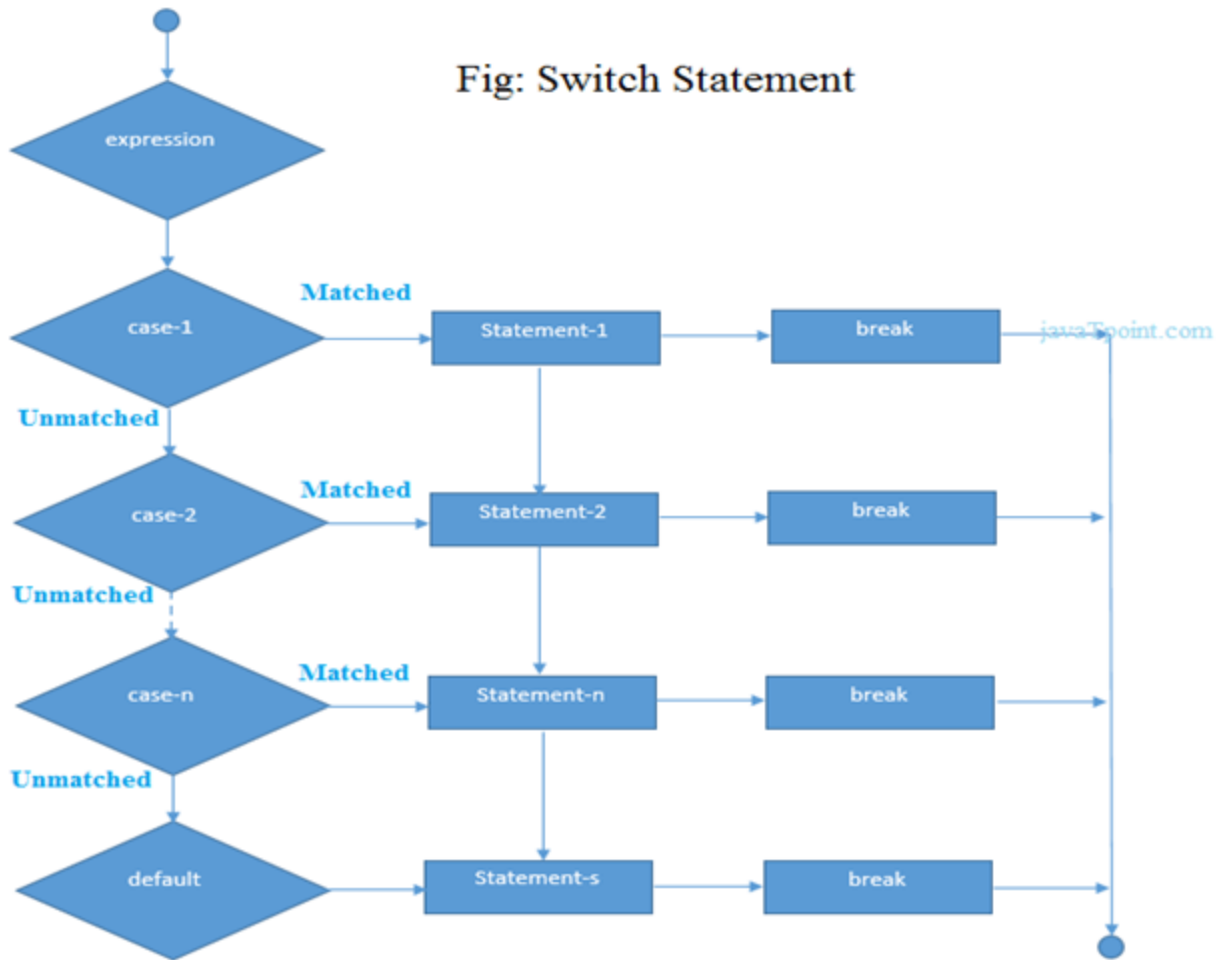
C++ switch

The C++ switch statement executes one statement from multiple conditions.

```
switch(expression){
```

1. **case** value1:
2. //code to be executed;
3. **break**;
4. **case** value2:
5. //code to be executed;
6. **break**;
7.
- 8.
9. **default**:
10. //code to be executed if all cases are not matched;
11. **break**;
12. }

Fig: Switch Statement



C++ Switch Example

```
1. #include <iostream>
2. using namespace std;
3. int main () {
4.     int num;
5.     cout<<"Enter a number to check grade:";
6.     cin>>num;
7.     switch (num)
8.     {
9.         case 10: cout<<"It is 10"; break;
10.        case 20: cout<<"It is 20"; break;
11.        case 30: cout<<"It is 30"; break;
12.        default: cout<<"Not 10, 20 or 30"; break;
13.    }
```

14. }

Output:

```
Enter a number:  
10  
It is 10
```

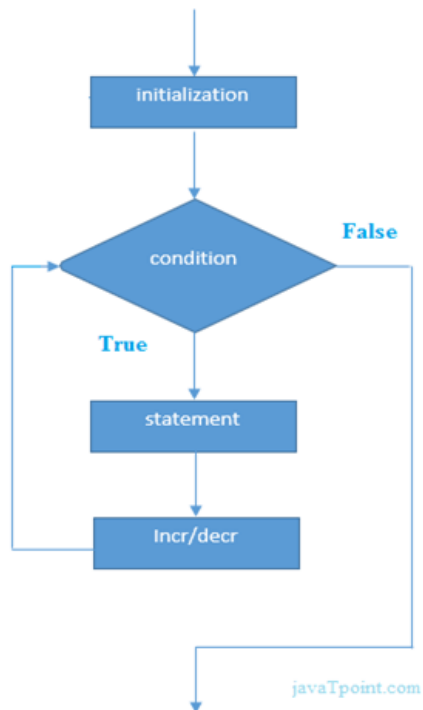
C++ For Loop

The C++ for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

The C++ for loop is same as C/C#. We can initialize variable, check condition and increment/decrement value.

1. **for**(initialization; condition; incr/decr){
2. //code to be executed
3. }

Flowchart:



C++ For Loop Example

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.     for(int i=1;i<=10;i++){
5.         cout<<i <<"\n";
6.     }
7. }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

C++ Nested For Loop

In C++, we can use for loop inside another for loop, it is known as nested for loop. The inner loop is executed fully when outer loop is executed one time. So if outer loop and inner loop are executed 4 times, inner loop will be executed 4 times for each outer loop i.e. total 16 times.

C++ Nested For Loop Example

Let's see a simple example of nested for loop in C++.

```
1. #include <iostream>
2. using namespace std;
3.
4. int main () {
5.     for(int i=1;i<=3;i++){
6.         for(int j=1;j<=3;j++){
7.             cout<<i<<" "<<j<<"\n";
8.         }
```


9. }
10. }

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

C++ Infinite For Loop

If we use double semicolon in for loop, it will be executed infinite times. Let's see a simple example of infinite for loop in C++.

1. #include <iostream>
2. **using namespace** std;
- 3.
4. **int** main () {
5. **for** (; ;)
6. {
7. cout<<"Infinitive For Loop";
8. }
9. }

Output:

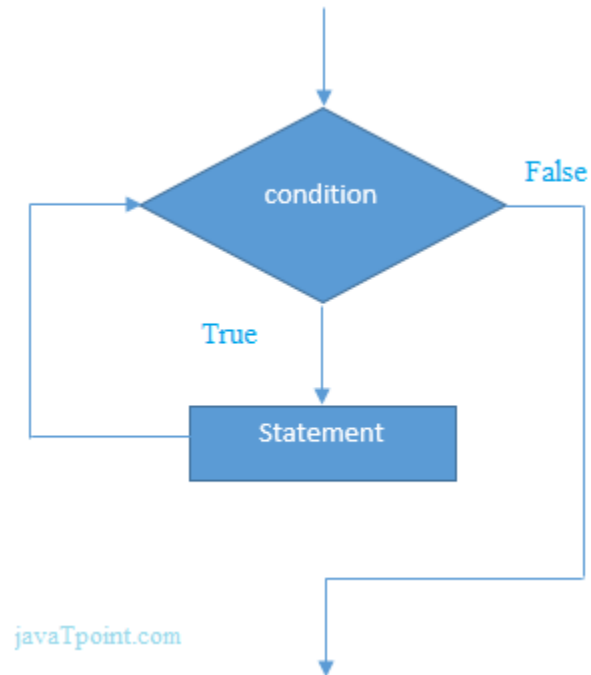
```
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
```

C++ While loop

In C++, while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop.

1. **while**(condition){
2. //code to be executed
3. }

Flowchart:



C++ While Loop Example

Let's see a simple example of while loop to print table of 1.

1. #include <iostream>
2. **using namespace** std;
3. **int** main() {
4. **int** i=1;
5. **while**(i<=10)
6. {
7. cout<<i <<"\n";
8. i++;
9. }
10. }

Output:

1
2
3
4
5
6
7
8
9
10

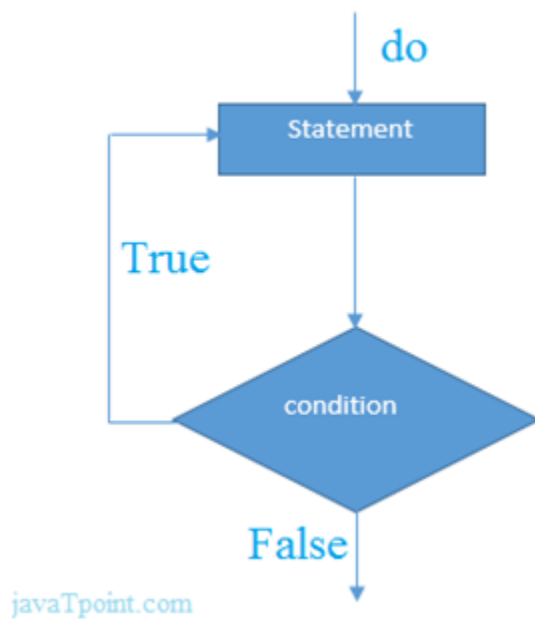
C++ Do-While Loop

The C++ do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The C++ do-while loop is executed at least once because condition is checked after loop body.

1. **do**{
2. //code to be executed
3. }**while**(condition);

Flowchart:



C++ do-while Loop Example

Let's see a simple example of C++ do-while loop to print the table of 1.

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.     int i = 1;
5.     do{
6.         cout<<i<<"\n";
7.         i++;
8.     } while (i <= 10);
9. }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

C++ Break Statement

The C++ break is used to break loop or switch statement. It breaks the current flow of the program at the given condition. In case of inner loop, it breaks only inner loop.

1. jump-statement;
2. **break**;

Flowchart:

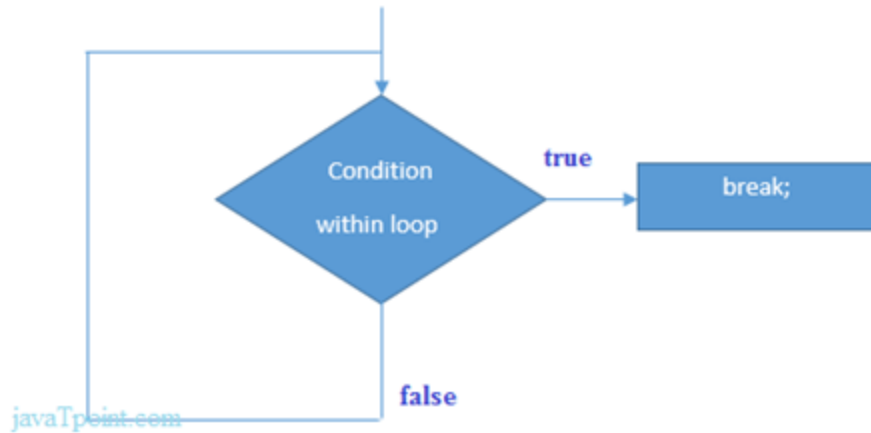


Figure: Flowchart of break statement

C++ Break Statement Example

Let's see a simple example of C++ break statement which is used inside the loop.

```

1. #include <iostream>
2. using namespace std;
3. int main() {
4.     for (int i = 1; i <= 10; i++)
5.     {
6.         if (i == 5)
7.         {
8.             break;
9.         }
10.    cout<<i<<"\n";
11.    }
12. }
  
```

Output:

```

1
2
3
4
  
```

C++ Continue Statement

The C++ continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

C++ Continue Statement Example

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     for(int i=1;i<=10;i++){
6.         if(i==5){
7.             continue;
8.         }
9.         cout<<i<<<"\n";
10.    }
11. }
```

Output:

```
1
2
3
4
6
7
8
9
10
```

C++ Goto Statement

The C++ goto statement is also known as jump statement. It is used to transfer control to the other part of the program. It unconditionally jumps to the specified label.

C++ Goto Statement Example

Let's see the simple example of goto statement in C++.

```
1. #include <iostream>
2. using namespace std;
3. int main()
```

```

4. {
5.   ineligible:
6.     cout<<"You are not eligible to vote!\n";
7.     cout<<"Enter your age:\n";
8.     int age;
9.     cin>>age;
10.    if (age < 18){
11.      goto ineligible;
12.    }
13.    else
14.    {
15.      cout<<"You are eligible to vote!";
16.    }
17. }

```

Output:

```

You are not eligible to vote!
Enter your age:
16
You are not eligible to vote!
Enter your age:
7
You are not eligible to vote!
Enter your age:
22
You are eligible to vote!

```

C++ Functions

The function in C++ language is also known as procedure or subroutine in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability.

Advantage of functions in C

There are many advantages of functions.

1) Code Reusability

By creating functions in C++, you can call it many times. So we don't need to write the same code again and again.

2) Code optimization

It makes the code optimized, we don't need to write much code.

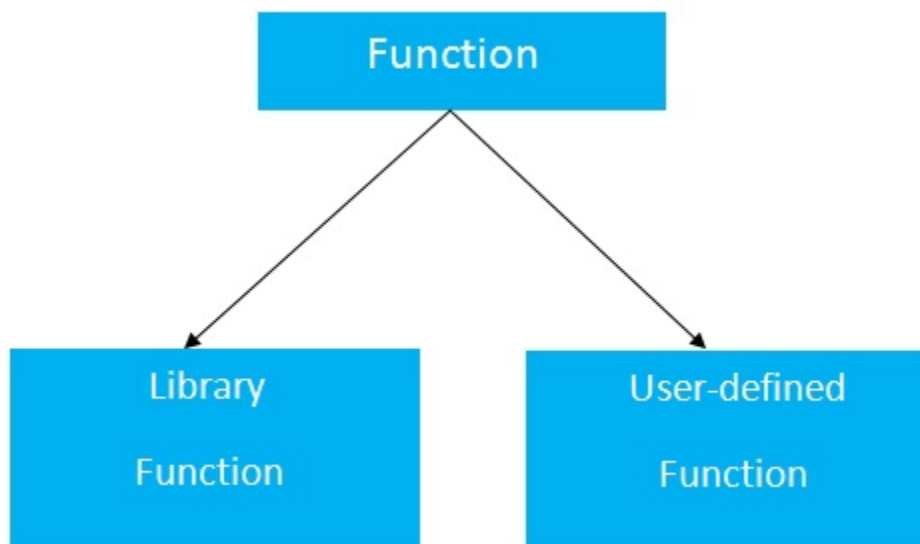
Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

- 1. Library Functions:** are the functions which are declared in the C++ header files such as `ceil(x)`, `cos(x)`, `exp(x)`, etc.
- 2. User-defined functions:** are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.



Declaration of a function

The syntax of creating function in C++ language is given below:

1. return_type function_name(data_type parameter...)
2. {
3. //code to be executed
4. }

C++ Function Example

Let's see the simple example of C++ function.

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

int addition (int a, int b)
{
    int r;
    r=a+b;
    return (r);
}

int main ()
{
    int z;
    z = addition (5,3);
    cout << "The result is " << z;
    return 0;
}
```

Functions with no type. The use of void.

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the void type specifier for the function. This is a special specifier that indicates absence of type.

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
    cout << "I'm a function!";
}
```

```
int main ()
{
    printmessage ();
    return 0;
}
```

Arguments passed by value and by reference.


Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition using the following code:

```
1 int x=5, y=3, z;
2 z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.

```
int addition (int a, int b)

z = addition ( 5 , 3 );
```



This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

```
1 // passing parameters by reference
2 #include <iostream>
3 using namespace std;
4
```

```
x=2, y=6, z=14
```

```


5 void duplicate (int& a, int& b, int& c)
6 {
7   a*=2;
8   b*=2;
9   c*=2;
10 }
11
12 int main ()
13 {
14   int x=1, y=3, z=7;
15   duplicate (x, y, z);
16   cout << "x=" << x << ", y=" << y << ", z=" << z;
17   return 0;
18 }

```

The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

```

void duplicate (int& a,int& b,int& c)
           
  

duplicate ( x , y , z );

```

To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z.

That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled.

If when declaring the following function:

```
void duplicate (int& a, int& b, int& c)
```

we had declared it this way:

```
void duplicate (int a, int b, int c)
```

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and z without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
1 // more than one returning value
2 #include <iostream>
3 using namespace std;
4
5 void prevnext (int x, int& prev, int& next)
6 {
7     prev = x-1;
8     next = x+1;
9 }
10
11 int main ()
12 {
13     int x=100, y, z;
14     prevnext (x, y, z);
15     cout << "Previous=" << y << ", Next=" << z;
16     return 0;
17 }
```

Previous=99, Next=101

Default values in parameters.

When declaring a function we 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. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```

1 // default values in functions
2 #include <iostream>
3 using namespace std;
4
5 int divide (int a, int b=2)
6 {
7     int r;
8     r=a/b;
9     return (r);
10 }
11
12 int main ()
13 {
14     cout << divide (12);
15     cout << endl;
16     cout << divide (20,4);
17     return 0;
18 }

```

```

6
5

```

As we can see in the body of the program there are two calls to function divide. In the first one:

```
divide (12)
```

we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with `int b=2`, not just `int b`). Therefore the result of this function call is 6 ($12/2$).

In the second call:

```
divide (20,4)
```

there are two parameters, so the default value for b (`int b=2`) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 ($20/4$).

Overloaded functions.

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:

```
1 // overloaded function
2 #include <iostream>
3 using namespace std;
4
5 int operate (int a, int b)
6 {
7     return (a*b);
8 }
9
10 float operate (float a, float b)
11 {
12     return (a/b);
13 }
14
15 int main ()
16 {
17     int x=5,y=2;
18     float n=5.0,m=2.0;
19     cout << operate (x,y);
20     cout << "\n";
21     cout << operate (n,m);
22     cout << "\n";
23     return 0;
24 }
```

10
2.5

In this case we have defined two functions with the same name, `operate`, but one of them accepts two parameters of type `int` and the other one accepts them of type `float`. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two `ints` as its arguments it calls to the function that has two `int` parameters in its prototype and if it is called with two `floats` it will call to the one which has two `float` parameters in its prototype.

In the first call to `operate` the two arguments passed are of type `int`, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type `float`, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to `operate` depends on the type of the arguments passed because the function has been *overloaded*.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Inline functions.

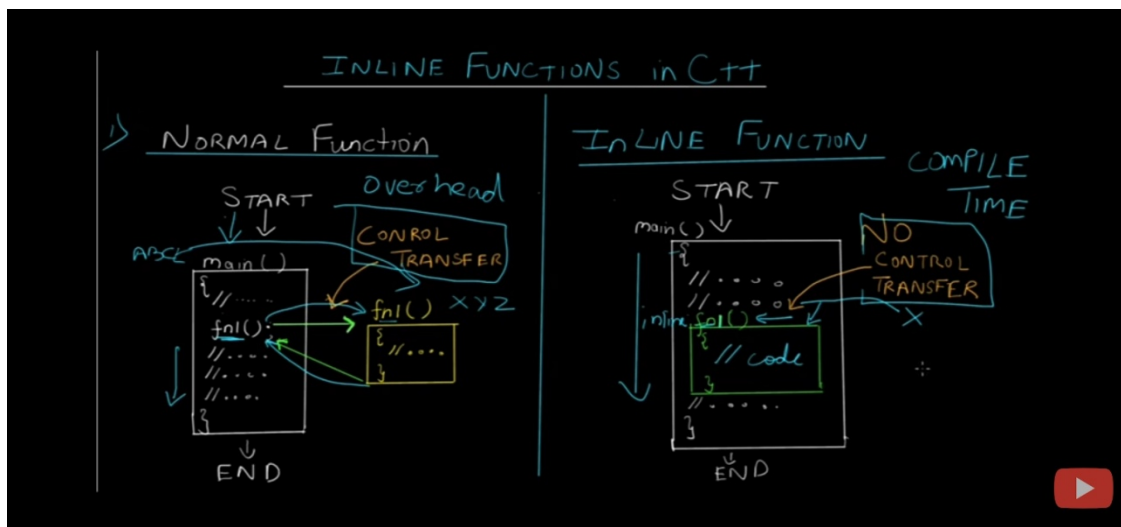
The inline specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

```
inline type name ( arguments ... ) { instructions ... }
```

and the call is just like the call to any other function. You do not have to include the inline keyword when calling the function, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.



Macro

Macro is a "preprocessors directive". Before compilation, the program is examined by the preprocessor and where ever it finds the macro in the program, it replaces that macro by its

definition. Hence, the macro is considered as the “text replacement”. Let us study macro with an example.

```
1. #include <stdio.h>
2. #define GREATER(a, b) ((a < b) ? b : a)
3. int main( void)
4. {
5.     cout << "Greater of 10 and 20 is " << GREATER("20", "10") << "\n";
6.     return 0;
7. }
```

In above code, we declared a macro function GREATER(), which compares and find the greater number of both the parameters. You can observe that there is no semicolon to terminate the macro as the macro is terminated only by the new line. As a macro is a just a text replacement, it will expand the code of macro where it is invoked.

- The macros are always defined in the capital letters just to make it easy for the programmers to identify all the macros in the program while reading.
- The macro can never be a class’s member function, nor it can access the data members of any class.
- The macro function evaluates the argument each time it appears in its definition, which results in an unexpected result.
- Macro must be of a smaller size as the larger macros will unnecessarily increase the size of the code.

Key Differences Between Inline and Macro

1. The basic difference between inline and macro is that an inline functions are parsed by the compiler whereas, the macros in a program are expanded by preprocessor.
2. The keyword used to define an inline function is “**inline**” whereas, the keyword used to define a macro is “**#define**”.
3. Once the inline function is declared inside a class, it can be defined either inside a class or outside a class. On the other hand, a macro is always defined at the start of the program.
4. The argument passed to the inline functions are evaluated only once while compilation whereas, the macros argument are evaluated each time a macro is used in the code.
5. The compiler may not inline and expand all the functions defined inside a class. On the other hand, macros are always expanded.
6. The short function that are defined inside a class without inline keyword are automatically made inline functions. On the other hand, Macro should be defined specifically.
7. A function that is inline can access the members of the class, whereas, a macro can never access the members of the class.

8. To terminate an inline function, a closing curly brace is required whereas, a macro is terminated with the start of a new line.
9. Debugging become easy for inline function as it is checked during compilation for any error. On the other hand, a macro is not checked while compilation so, debugging a macro becomes difficult.
10. Being a function an inline function bind its members within a start and closing curly braces. On the other hand, macro does not have any termination symbol so; binding becomes difficult when macro contains more that one statement.

Access Specifiers

Access Modifiers or Access Specifiers in a class are used to assign the accessibility to the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

Let us now look at each one these access modifiers in details:

1. Public: All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

Example:

```
// C++ program to demonstrate public
```

```
// access modifier
```

```
#include<iostream>
```

```
using namespace std;
```

```
// class definition
class Circle
{
    public:
        double radius;

        double compute_area()
        {
            return 3.14*radius*radius;
        }

};

// main function
int main()
{
    Circle obj;

    // accessing public datamember outside class
    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

Output:

```
Radius is: 5.5
```

```
Area is: 94.985
```

In the above program the data member *radius* is declared as public so it could be accessed outside the class and thus was allowed access from inside main().

2. Private: The class members declared as *private* can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

Example:

```
// C++ program to demonstrate private
```

```
// access modifier
```

```
#include<iostream>
```

```
using namespace std;
```

```
class Circle
```

```
{
```

```
    // private data member
```

```
    private:
```

```
        double radius;
```

```
    // public member function
```

```
    public:
```

```
        double compute_area()
```

```
    { // member function can access private
      // data member radius
      return 3.14*radius*radius;
    }
};
```

```
// main function
int main()
{
  // creating object of the class
  Circle obj;

  // trying to access private data member
  // directly outside the class
  obj.radius = 1.5;

  cout << "Area is:" << obj.compute_area();
  return 0;
}
```

Output:

```
In function 'int main()':
11:16: error: 'double Circle::radius' is private
      double radius;
      ^
```

```
31:9: error: within this context
```

```
obj.radius = 1.5;
```

^

The output of above program is a compile time error because we are not allowed to access the private data members of a class directly outside the class. Yet an access to obj.radius is attempted, radius being a private data member we obtain a compilation error.

However, we can access the private data members of a class indirectly using the public member functions of the class.

Example:

```
// C++ program to demonstrate private
```

```
// access modifier
```

```
#include<iostream>
```

```
using namespace std;
```

```
class Circle
```

```
{
```

```
    // private data member
```

```
    private:
```

```
        double radius;
```

```
    // public member function
```

```
    public:
```

```
        void compute_area(double r)
```

```
        { // member function can access private
```

```
            // data member radius
```

```
            radius = r;
```

```
            double area = 3.14*radius*radius;
```

```
        cout << "Radius is: " << radius << endl;
        cout << "Area is: " << area;
    }

};

// main function
int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.compute_area(1.5);

    return 0;
}
```

Output:

Radius is: 1.5

Area is: 7.065

3. Protected: Protected access modifier is similar to private access modifier in the sense that it can't be accessed outside of it's class unless with the help of friend class, the difference is that the

class members declared as Protected can be accessed by any subclass(derived class) of that class as well.

Note: This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the modes of Inheritance.

Example:

```
// C++ program to demonstrate
```

```
// protected access modifier
```

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
// base class
```

```
class Parent
```

```
{
```

```
    // protected data members
```

```
    protected:
```

```
    int id_protected;
```

```
};
```

```
// sub class or derived class from public base class
```

```
class Child : public Parent
```

```
{
```

```
    public:
```

```
    void setId(int id)
```

```
    {
```

```
        // Child class is able to access the inherited
```

```
// protected data members of base class

    id_protected = id;

}

void displayId()
{
    cout << "id_protected is: " << id_protected << endl;
}
};

// main function
int main() {

    Child obj1;

    // member function of the derived class can
    // access the protected data members of the base class

    obj1.setId(81);
    obj1.displayId();

    return 0;
}
```

Output:

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

1. **class** class_name
2. {
3. **friend** data_type function_name(argument/s); // syntax of friend function.
4. };

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or **scope resolution operator**.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ friend function Example

```
include <iostream>

using namespace std;

class Box {
    double width;

    public:
```

```

    friend void printWidth( Box box );
    void setWidth( double wid );
};

// Member function definition
void Box::setWidth( double wid ) {
    width = wid;
}

// Note: printWidth() is not a member function of any class.
void printWidth( Box box ) {
    /* Because printWidth() is a friend of Box, it can
    directly access any member of this class */
    cout << "Width of box : " << box.width << endl;
}

// Main function for the program
int main() {
    Box box;

    // set box width without member function
    box.setWidth(10.0);

    // Use friend function to print the width.
    printWidth( box );

    return 0;
}

```

When the above code is compiled and executed, it produces the following result –
Width of box : 10

Constructors in C++

What is constructor?

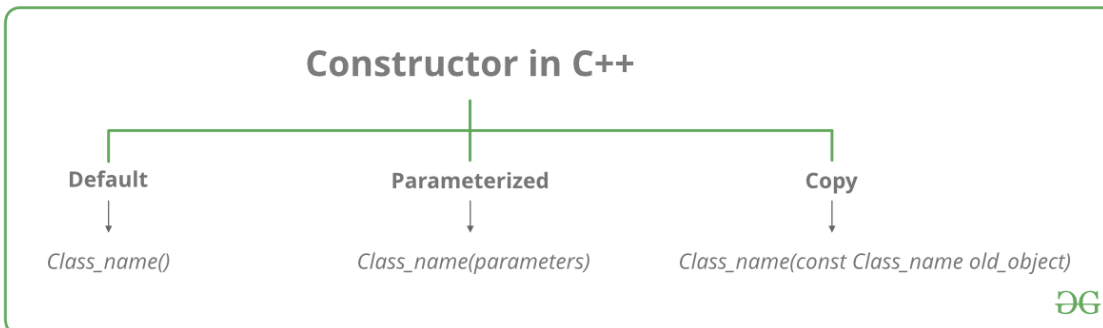
A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class.

How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.

- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).



Types of Constructors

1. **Default Constructors:** Default constructor is the constructor which doesn't take any argument. It has no parameters.

// Cpp program to illustrate the

// concept of Constructors

```
#include <iostream>
```

```
using namespace std;
```

```
class construct
```

```
{
```

```
public:
```

```
    int a, b;
```

```
    // Default Constructor
```

```
    construct()
```

```
{
```

```
    a = 10;
```

```
    b = 20;
```

```

    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl
         << "b: " << c.b;
    return 1;
}

```

Output:

```

a: 10
b: 20

```

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

2. **Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```

// CPP program to illustrate
// parameterized constructors
#include <iostream>

```

```
using namespace std;
```

```
class Point
```

```
{
```

```
private:
```

```
    int x, y;
```

```
public:
```

```
    // Parameterized Constructor
```

```
    Point(int x1, int y1)
```

```
    {
```

```
        x = x1;
```

```
        y = y1;
```

```
    }
```

```
    int getX()
```

```
    {
```

```
        return x;
```

```
    }
```

```
    int getY()
```

```
    {
```

```
        return y;
```

```
    }
```

```
};
```

```

int main()
{
    // Constructor called

    Point p1(10, 15);

    // Access values assigned by constructor

    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}

```

Output:

```
p1.x = 10, p1.y = 15
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
Example e = Example(0, 50); // Explicit call
```

```
Example e(0, 50); // Implicit call
```

1. Uses of Parameterized constructor:

1. It is used to initialize the various data elements of different objects with different values when they are created.
2. It is used to overload constructors.
- 3.

Can we have more than one constructor in a class?

Yes, It is called Constructor Overloading.

3. Copy Constructor: A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```

Following is a simple example of copy constructor.

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

class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()      { return x; }
    int getY()      { return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
```

```
p2.x = 10, p2.y = 15
```

C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically.

```
1. #include <iostream>
2. using namespace std;
3. class Employee
4. {
5.     public:
6.         Employee()
7.         {
8.             cout<<"Constructor Invoked"<<endl;
9.         }
10.        ~Employee()
11.        {
12.            cout<<"Destructor Invoked"<<endl;
13.        }
14. };
15. int main(void)
16. {
17.     Employee e1; //creating an object of Employee
18.     Employee e2; //creating an object of Employee
19.     return 0;
20. }
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```