

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 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

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.

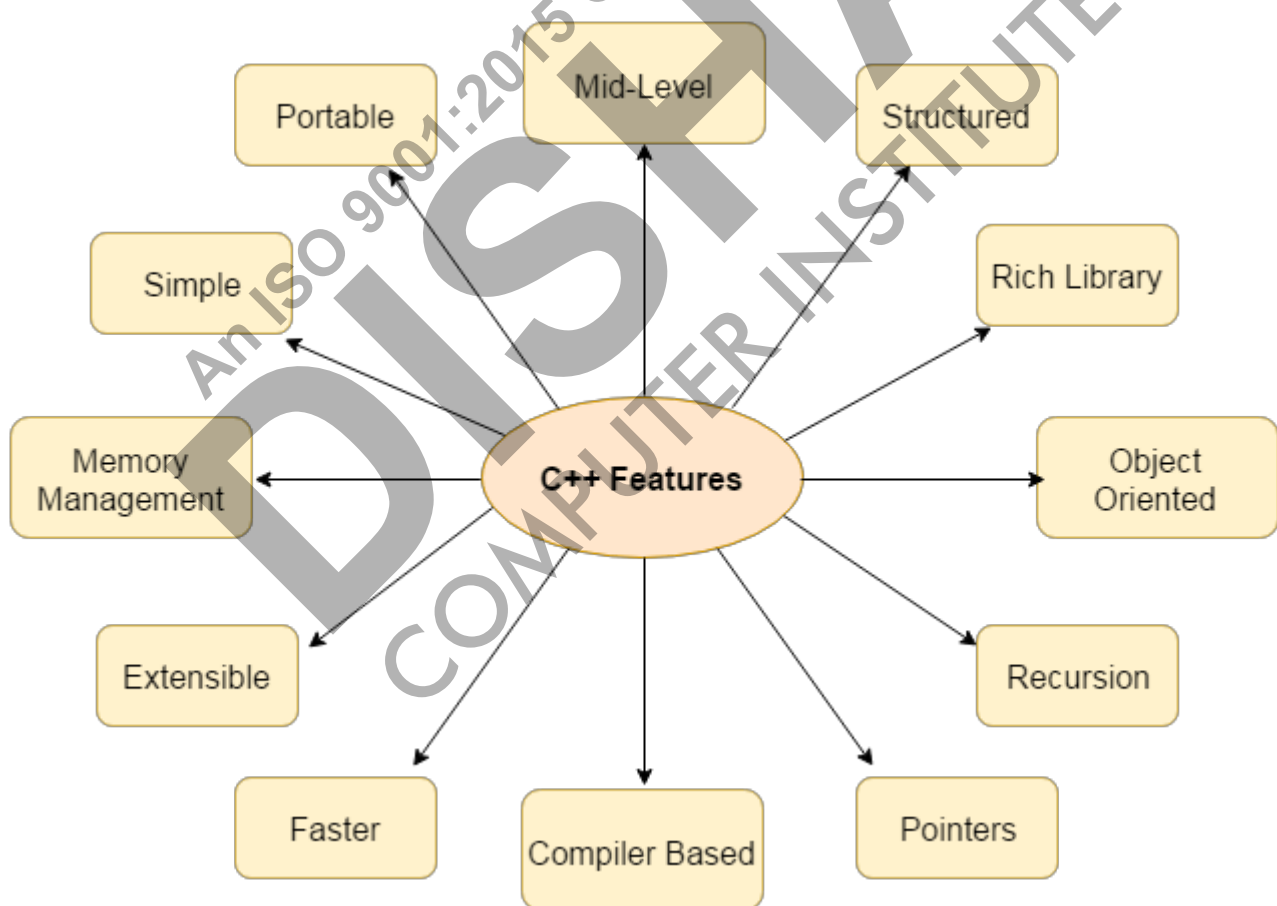
C++ programming language was developed in 1980 by Bjarne Stroustrup at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A.

Bjarne Stroustrup is known as the **founder of C++ language**.

It was develop for adding a feature of **OOP (Object Oriented Programming)** in C without significantly changing the C component.

C++ Features

C++ is object oriented programming language. It provides a lot of **features** that are given below.



1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. Structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible
11. Object Oriented
12. Compiler based

1) Simple

C++ is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Machine Independent or Portable

c programs can be executed in many machines with little bit or no change.

3) Mid-level programming language

C++ is also used to do low level programming. It is used to develop system applications such as kernel, driver etc. It also supports the feature of high level language. That is why it is known as mid-level language.

4) Structured programming language

C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

5) Rich Library

C++ provides a lot of inbuilt functions that makes the development fast.

6) Memory Management

It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.

7) Speed

The compilation and execution time of C++ language is fast.

8) Pointer

C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.

9) Recursion

In C++, we can call the function within the function. It provides code reusability for every function.

10) Extensible

C++ language is extensible because it can easily adopt new features.

11) Object Oriented

C++ is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

12) Compiler based

C++ is a compiler based programming language, it means without compilation no C++ program can be executed. First we need to compile our program using compiler and then we can execute our program.

C++ Program

Before starting the abcd of C++ language, you need to learn how to write, compile and run the first C++ program.

To write the first C++ program, open the C++ console and write the following code:

```
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.

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';`

C++ Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character etc.

There are 4 types of data types in C++ language.

Types	Data Types
Basic Data Type	int, char, float, double, etc
Derived Data Type	array, pointer, etc
Enumeration Data Type	enum
User Defined Data Type	structure

C++ Keywords

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. **A list of 32 Keywords in C++ Language which are also available in C language are given below.**

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

C++ Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.

There are following types of operators to perform different types of operations in C language.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operator
- Unary operator
- Ternary or Conditional Operator

	Operator	Type
Binary Operator	+, -, *, /, %	Arithmetic Operators
	<, <=, >, >=, ==, !=	Relational Operators
	&&, , !	Logical Operators
	&, , <<, >>, ~, ^	Bitwise Operators
	=, +=, -=, *=, /=, %=	Assignment Operators
Unary Operator	→ ++, --	Unary Operator
Ternary Operator	→ ?	Ternary or Conditional Operator

C++ Identifiers

C++ identifiers in a program are used to refer to the name of the variables, functions, arrays, or other user-defined data types created by the programmer. They are the basic requirement of any language. Every language has its own rules for naming the identifiers.

In short, we can say that the C++ identifiers represent the essential elements in a program which are given below:

- **Constants**
- **Variables**
- **Functions**
- **Labels**
- **Defined data types**

Some naming rules are common in both C and C++. They are as follows:

- Only alphabetic characters, digits, and underscores are allowed.

- The identifier name cannot start with a digit, i.e., the first letter should be alphabetical. After the first letter, we can use letters, digits, or underscores.
- In C++, uppercase and lowercase letters are distinct. Therefore, we can say that C++ identifiers are case-sensitive.
- A declared keyword cannot be used as a variable name.

For example, suppose we have two identifiers, named as 'FirstName', and 'Firstname'. Both the identifiers will be different as the letter 'N' in the first case is in uppercase while lowercase in second. Therefore, it proves that identifiers are case-sensitive.

Valid Identifiers

The following are the examples of valid identifiers are:

1. Result
2. Test2
3. _sum
4. power

Invalid Identifiers

The following are the examples of invalid identifiers:

1. Sum-1 // containing special character '-'.
2. 2data // the first letter is a digit.
3. **break** // use of a keyword.

C++ if-else

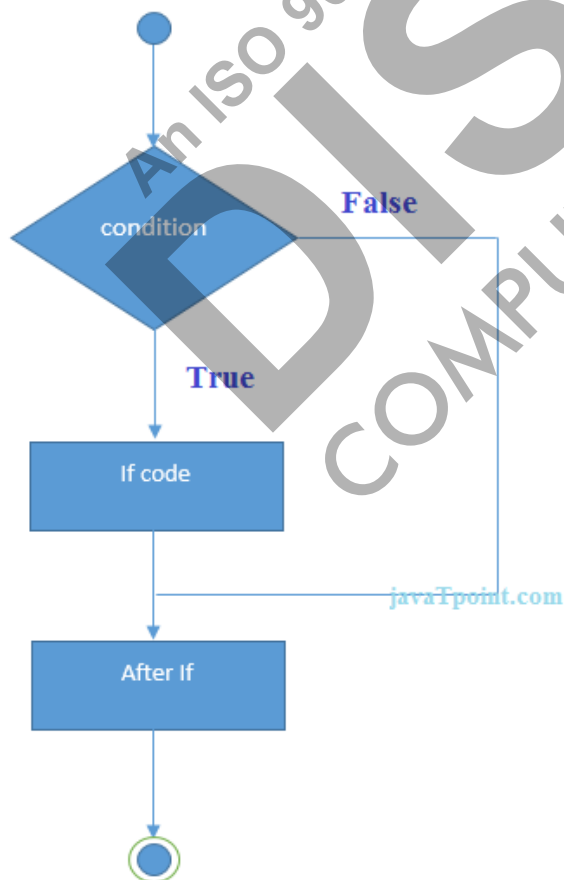
In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.

- 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

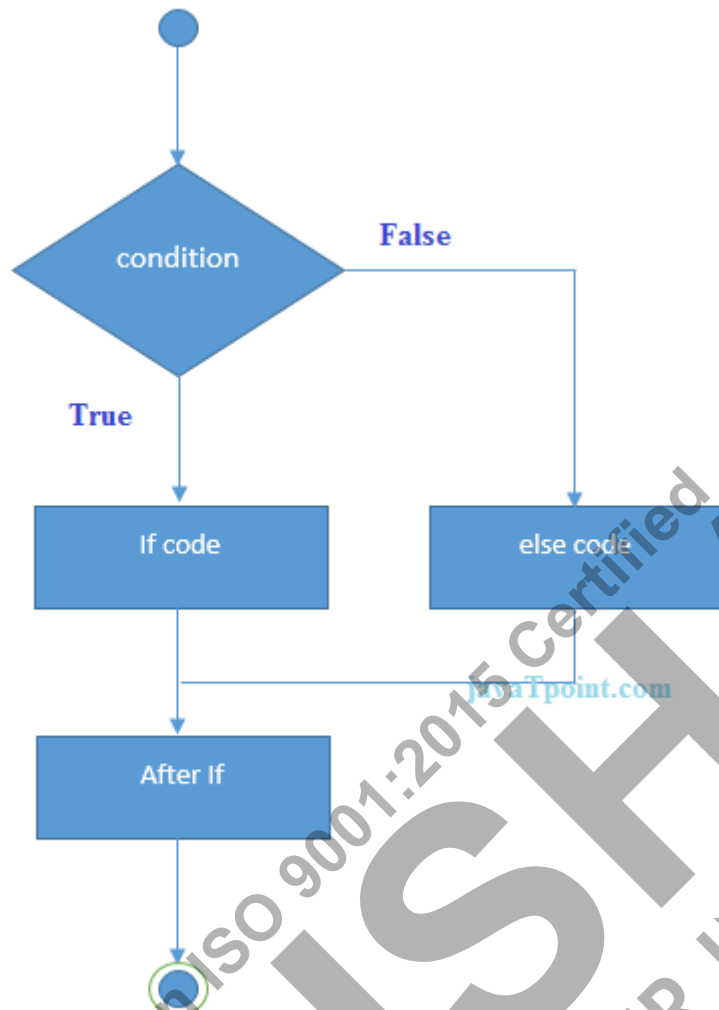
```
1. #include <iostream>
2. using namespace std;
3.
4. int main () {
5.     int num = 10;
6.     if (num % 2 == 0)
7.     {
8.         cout<<"It is even number";
9.     }
10.    return 0;
11. }
```

```
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:

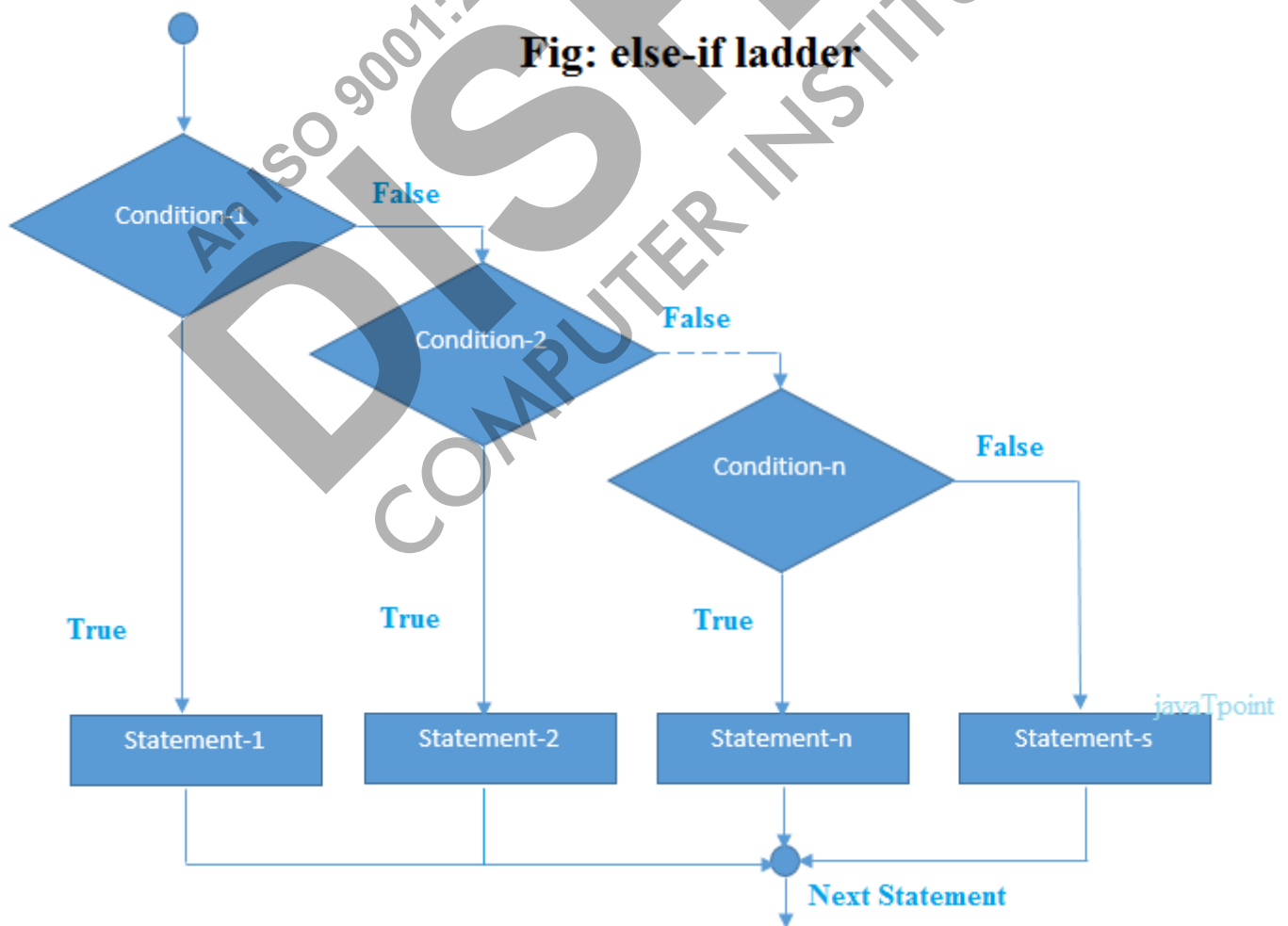
```
Enter a number:12
It is even number
```

C++ IF-else-if ladder Statement

The C++ if-else-if ladder statement executes one condition from multiple statements.

```
1. if(condition1){  
2. //code to be executed if condition1 is true  
3. }else if(condition2){  
4. //code to be executed if condition2 is true  
5. }  
6. else if(condition3){  
7. //code to be executed if condition3 is true  
8. }  
9. ...  
10. else{  
11. //code to be executed if all the conditions are false  
12. }
```

Fig: else-if ladder



C++ If else-if 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.     if (num <0 || num >100)
8.     {
9.         cout<<"wrong number";
10.    }
11.    else if (num >= 0 && num < 50){
12.        cout<<"Fail";
13.    }
14.    else if (num >= 50 && num < 60)
15.    {
16.        cout<<"D Grade";
17.    }
18.    else if (num >= 60 && num < 70)
19.    {
20.        cout<<"C Grade";
21.    }
22.    else if (num >= 70 && num < 80)
23.    {
24.        cout<<"B Grade";
25.    }
26.    else if (num >= 80 && num < 90)
27.    {
28.        cout<<"A Grade";
29.    }
30.    else if (num >= 90 && num <= 100)
31.    {
32.        cout<<"A+ Grade";
33.    }
34. }
```


Output:

```
Enter a number to check grade:66
C Grade
```

Output:

```
Enter a number to check grade:-2
wrong number
```

C++ switch

The C++ switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement in C++.

1. **switch**(expression){
2. **case**value1:
3. *//code to be executed;*
4. **break**;
5. **case**value2:
6. *//code to be executed;*
7. **break**;
8.
- 9.
10. **default**:
11. *//code to be executed if all cases are not matched;*
12. **break**;
13. }

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

```

Output:

```

Enter a number:
55
Not 10, 20 or 30

```

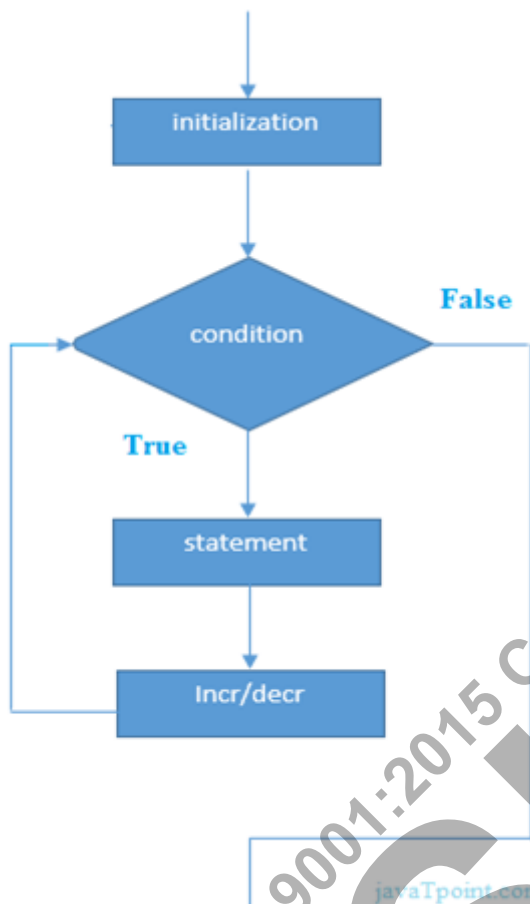
C++ For Loop

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:

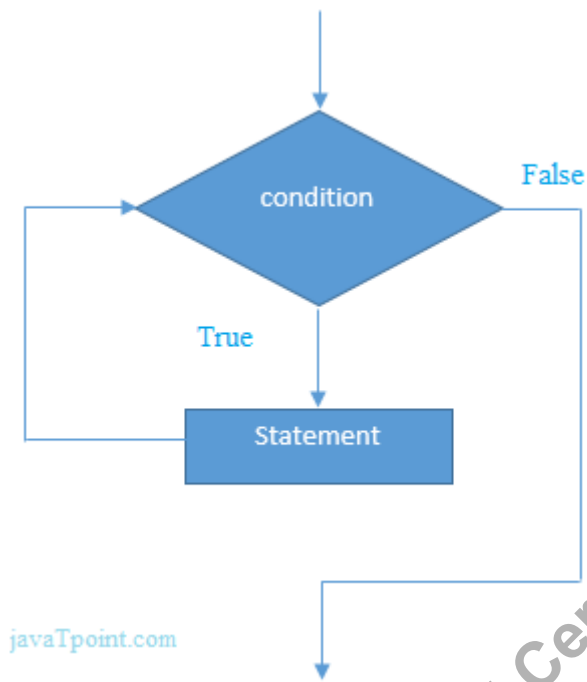
```
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
ctrl+c
```

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++ Nested While Loop Example

In C++, we can use while loop inside another while loop, it is known as nested while loop. The nested while loop is executed fully when outer loop is executed once.

Let's see a simple example of nested while loop in C++ programming language.

```
1. #include <iostream>
2. using namespace std;
3. int main () {
4.     int i=1;
5.     while(i<=3)
6.     {
7.         int j = 1;
8.         while (j <= 3)
9.         {
10.            cout<<i<<" "<<j<<"\n";
11.            j++;
12.        }
13.        i++;
14.    }
15. }
```

Output:

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

C++ Infinite While Loop Example:

We can also create infinite while loop by passing true as the test condition.

```
1. #include <iostream>
2. using namespace std;
3. int main () {
4.     while(true)
5.     {
6.         cout<<"Infinitive While Loop";
7.     }
8. }
```

Output:

```
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
ctrl+c
```

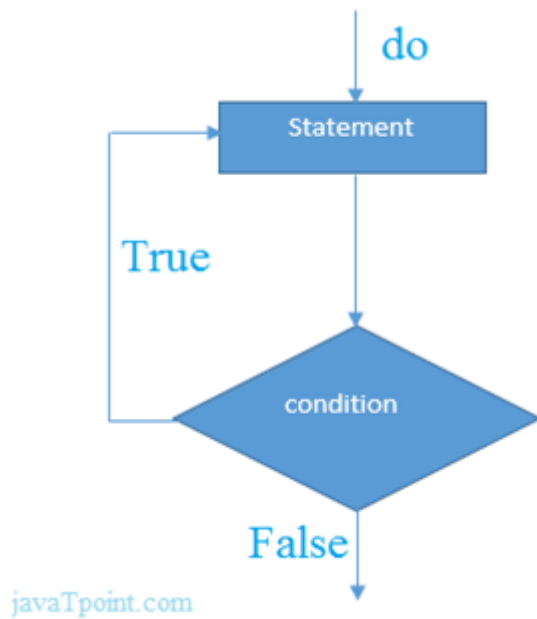
C++ Do-While Loop

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++ Infinitive do-while Loop

In C++, if you pass **true** in the do-while loop, it will be infinitive do-while loop.

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

C++ Infinitive do-while Loop Example

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

Output:

```
Infinitive do -while Loop
Infinitive do -while Loop
Infinitive do -while Loop
Infinitive do -while Loop
Infinitive do -while Loop
ctrl+c
```

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:

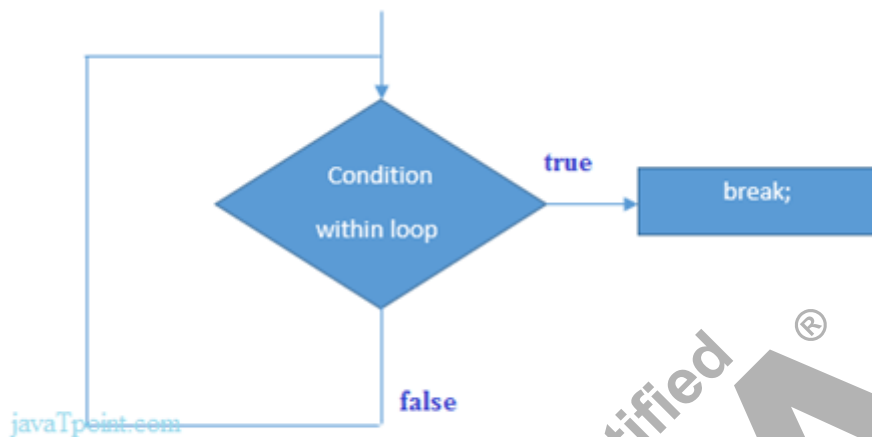


Figure: Flowchart of break statement

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
```

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.

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

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.

It can be used to transfer control from deeply nested loop or switch case 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++ Comments

The C++ comments are statements that are not executed by the compiler. The comments in C++ programming can be used to provide explanation of the code, variable, method or class. By the help of comments, you can hide the program code also.

There are two types of comments in C++.

- Single Line comment
- Multi Line comment

C++ Single Line Comment

The single line comment starts with // (double slash). Let's see an example of single line comment in C++.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int x = 11; // x is a variable
6.     cout << x << "\n";
7. }
```

Output:

```
11
```

C++ Multi Line Comment

The C++ multi line comment is used to comment multiple lines of code. It is surrounded by slash and asterisk (/* */). Let's see an example of multi line comment in C++.

```
1. #include <ostream>
2. using namespace std;
```

```
3. int main()
4. {
5.     /* declare and
6.     print variable in C++. */
7.     int x = 35;
8.     cout<<x<<"\n";
9. }
```

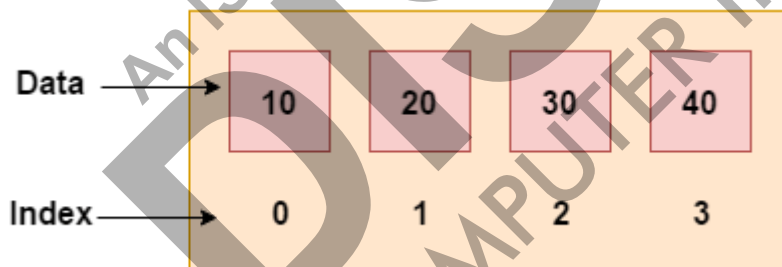
Output:

```
35
```

C++ Arrays

Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location.

In C++ **std::array** is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.



Advantages of C++ Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

Disadvantages of C++ Array

- Fixed size

C++ Array Types

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array
2. Multidimensional Array

C++ Single Dimensional Array

Let's see a simple example of C++ array, where we are going to create, initialize and traverse array.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
6.     //traversing array
7.     for (int i = 0; i < 5; i++)
8.     {
9.         cout<<arr[i]<<"\n";
10.    }
11. }
```

Output:/p>


```
10
0
20
0
30
```

C++ Array Example: Traversal using foreach loop

We can also traverse the array elements using foreach loop. It returns array element one by one.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
6.     //traversing array
7.     for (int i: arr)
8.     {
9.         cout<<i<<"\n";
10.    }
11. }
```

Output:



```
10
20
30
40
50
```

C++ Passing Array to Function

In C++, to reuse the array logic, we can create function. To pass array to function in C++, we need to provide only array name.

```
1. functionname(arrayname); //passing array to function
```

C++ Passing Array to Function Example: print array elements

Let's see an example of C++ function which prints the array elements.

```
1. #include <iostream>
2. using namespace std;
```

```
3. void printArray(int arr[5]);
4. int main()
5. {
6.     int arr1[5] = { 10, 20, 30, 40, 50 };
7.     int arr2[5] = { 5, 15, 25, 35, 45 };
8.     printArray(arr1); //passing array to function
9.     printArray(arr2);
10. }
11. void printArray(int arr[5])
12. {
13.     cout << "Printing array elements:" << endl;
14.     for (int i = 0; i < 5; i++)
15.     {
16.         cout << arr[i] << "\n";
17.     }
18. }
```

Output:

```
Printing array elements:
10
20
30
40
50
Printing array elements:
5
15
25
35
45
```

C++ Multidimensional Arrays

The multidimensional array is also known as rectangular arrays in C++. It can be two dimensional or three dimensional. The data is stored in tabular form (row * column) which is also known as matrix.

C++ Multidimensional Array Example

Let's see a simple example of multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int test[3][3]; //declaration of 2D array
6.     test[0][0]=5; //initialization
7.     test[0][1]=10;
8.     test[1][1]=15;
9.     test[1][2]=20;
10.    test[2][0]=30;
11.    test[2][2]=10;
12.    //traversal
13.    for(int i = 0; i < 3; ++i)
14.    {
15.        for(int j = 0; j < 3; ++j)
16.        {
17.            cout<< test[i][j]<<" ";
18.        }
19.        cout<<"\n"; //new line at each row
20.    }
21.    return 0;
22. }
```

Output:

```
5 10 0
0 15 20
30 0 10
```

C++ Multidimensional Array Example: Declaration and initialization at same time

Let's see a simple example of multidimensional array which initializes array at the time of declaration.

```

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int test[3][3] =
6.     {
7.         {2, 5, 5},
8.         {4, 0, 3},
9.         {9, 1, 8} }; //declaration and initialization
10.    //traversal
11.    for(int i = 0; i < 3; ++i)
12.    {
13.        for(int j = 0; j < 3; ++j)
14.        {
15.            cout<< test[i][j]<<" ";
16.        }
17.        cout<<"\n"; //new line at each row
18.    }
19.    return 0;
20. }

```

Output:"

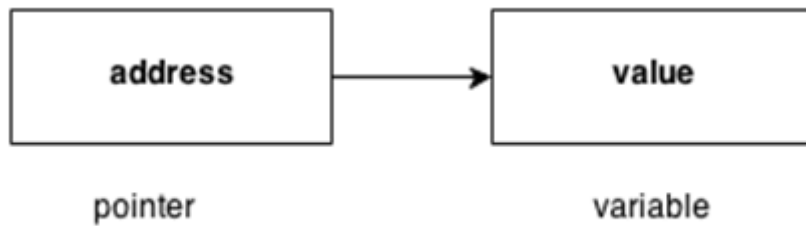
```

2 5 5
4 0 3
9 1 8

```

C++ Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



Advantage of pointer

- 1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.
- 2) We can return multiple values from function using pointer.

- 3) It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.

* (asterisk sign)	Indirection operator	Access the value of an address.
-------------------	----------------------	---------------------------------

Declaring a pointer

The pointer in C++ language can be declared using * (asterisk symbol).

1. `int * a; //pointer to int`
2. `char * c; //pointer to char`

Pointer Example

Let's see the simple example of using pointers printing the address and value.

1. `#include <iostream>`
2. `using namespace std;`
3. `int main()`
4. `{`
5. `int number=30;`
6. `int * p;`
7. `p=&number; //stores the address of number variable`
8. `cout<<"Address of number variable is:"<<&number<<endl;`
9. `cout<<"Address of p variable is:"<<p<<endl;`
10. `cout<<"Value of p variable is:"<<*p<<endl;`
11. `return 0;`
12. `}`

Output:

```
Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30
```

Pointer Program to swap 2 numbers without using 3rd variable

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

```
4. {  
5.  int a=20,b=10,*p1=&a,*p2=&b;  
6.  cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;  
7.  *p1=*p1+*p2;  
8.  *p2=*p1-*p2;  
9.  *p1=*p1-*p2;  
10. cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;  
11.  return 0;  
12. }
```

Output:

```
Before swap: *p1=20 *p2=10  
After swap: *p1=10 *p2=20
```

What is Memory Management?

Memory management is a process of managing computer memory, assigning the memory space to the programs to improve the overall system performance.

Why is memory management required?

As we know that arrays store the homogeneous data, so most of the time, memory is allocated to the array at the declaration time. Sometimes the situation arises when the exact memory is not determined until runtime. To avoid such a situation, we declare an array with a maximum size, but some memory will be unused. To avoid the wastage of memory, we use the new operator to allocate the memory dynamically at the run time.

New operator

A **new** operator is used to create the object while a **delete** operator is used to delete the object. When the object is created by using the new operator, then the object will exist until we explicitly use the delete operator to delete the object. Therefore, we can say that the lifetime of the object is not related to the block structure of the program.

Syntax

1. `pointer_variable = new data-type`

The above syntax is used to create the object using the new operator. In the above syntax, '**pointer_variable**' is the name of the pointer variable, '**new**' is the operator, and '**data-type**' defines the type of the data.

Example 1:

1. `int *p;`
2. `p = new int;`

In the above example, 'p' is a pointer of type int.

Example 2:

1. `float *q;`
2. `q = new float;`

In the above example, 'q' is a pointer of type float.

In the above case, the declaration of pointers and their assignments are done separately. We can also combine these two statements as follows:

1. `int *p = new int;`
2. `float *q = new float;`

Assigning a value to the newly created object

Two ways of assigning values to the newly created object:

- We can assign the value to the newly created object by simply using the assignment operator. In the above case, we have created two pointers 'p' and 'q' of type int and float, respectively. Now, we assign the values as follows:

1. `*p = 45;`
2. `*q = 9.8;`

We assign 45 to the newly created int object and 9.8 to the newly created float object.

- We can also assign the values by using new operator which can be done as follows:

1. `pointer_variable = new data-type(value);`

Let's look at some examples.

1. `int *p = new int(45);`
2. `float *p = new float(9.8);`

How to create a single dimensional array

As we know that new operator is used to create memory space for any data-type or even user-defined data type such as an array, structures, unions, etc., so the syntax for creating a one-dimensional array is given below:

1. `pointer-variable = new data-type[size];`

Examples:

1. `int *a1 = new int[8];`

In the above statement, we have created an array of type int having a size equal to 8 where p[0] refers first element, p[1] refers the first element, and so on.

Delete operator

When memory is no longer required, then it needs to be deallocated so that the memory can be used for another purpose. This can be achieved by using the delete operator, as shown below:

1. `delete pointer_variable;`

In the above statement, '**delete**' is the operator used to delete the existing object, and '**pointer_variable**' is the name of the pointer variable.

In the previous case, we have created two pointers 'p' and 'q' by using the new operator, and can be deleted by using the following statements:

1. `delete p;`
2. `delete q;`

The dynamically allocated array can also be removed from the memory space by using the following syntax:

1. `delete [size] pointer_variable;`

In the above statement, we need to specify the size that defines the number of elements that are required to be freed. The drawback of this syntax is that we need to remember the size of the array. But, in recent versions of C++, we do not need to mention the size as follows:

1. **delete** [] pointer_variable;

Let's understand through a simple example:

```
1. #include <iostream>
2. using namespace std
3. int main()
4. {
5.     int size; // variable declaration
6.     int *arr = new int[size]; // creating an array
7.     cout<<"Enter the size of the array : ";
8.     std::cin >> size; //
9.     cout<<"\nEnter the element : ";
10.    for(int i=0;i<size;i++) // for loop
11.    {
12.        cin>>arr[i];
13.    }
14.    cout<<"\nThe elements that you have entered are :";
15.    for(int i=0;i<size;i++) // for loop
16.    {
17.        cout<<arr[i]<<" ";
18.    }
19.    delete arr; // deleting an existing array.
20.    return 0;
21. }
```

Output

```
Enter the size of the array : 5

Enter the element : 1
2
3
4
5

The elements that you have entered are :1,2,3,4,5,

...Program finished with exit code 0
Press ENTER to exit console.
```

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.

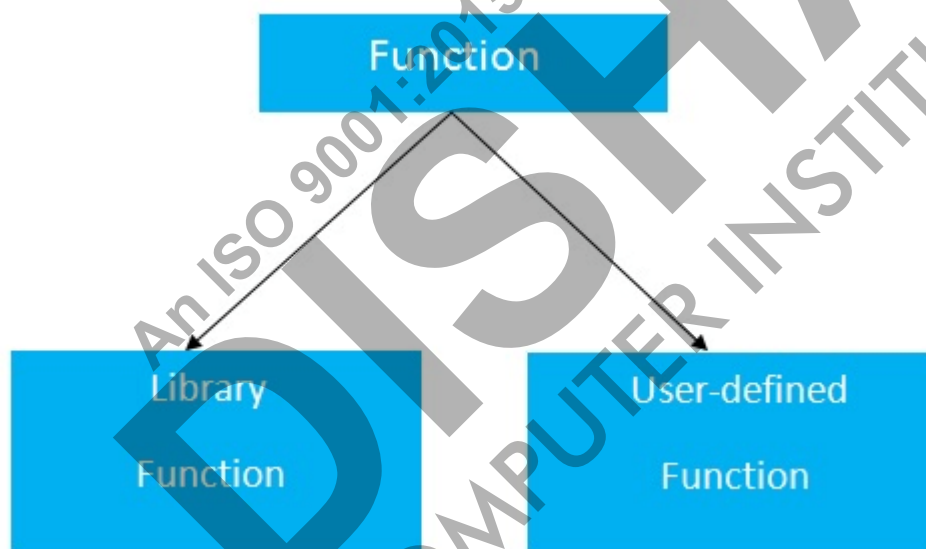
Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

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

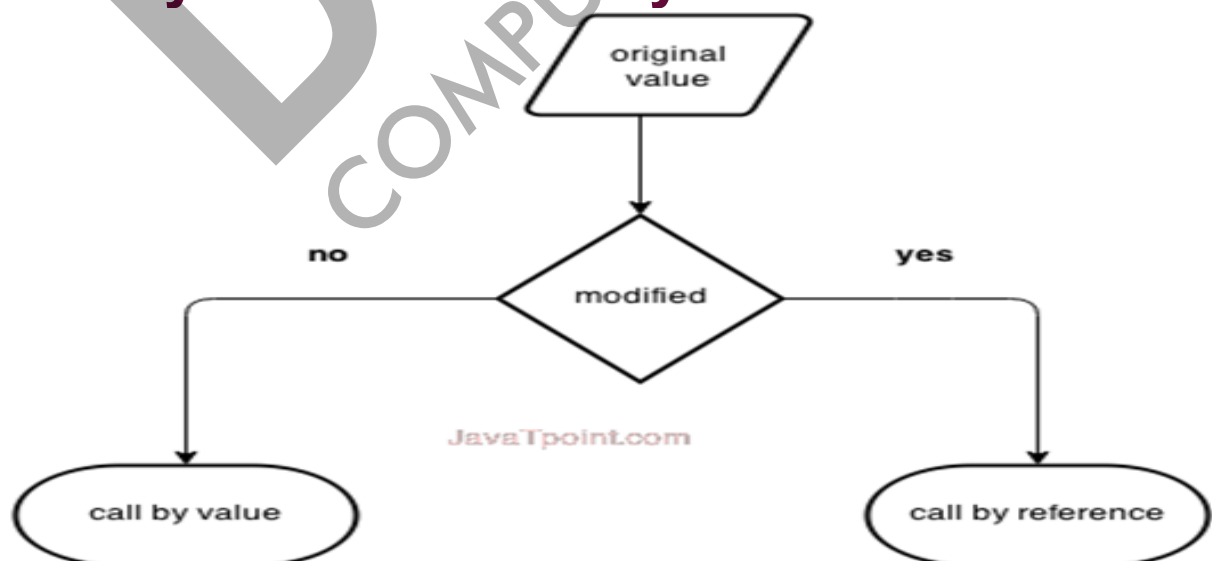
Let's see the simple example of C++ function.

```
1. #include <iostream>
2. using namespace std;
3. void func() {
4.     static int i=0; //static variable
5.     int j=0; //local variable
6.     i++;
7.     j++;
8.     cout<<"i=" << i<<" and j=" <<j<<endl;
9. }
10. int main()
11. {
12.     func();
13.     func();
14.     func();
15. }
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

Call by value and call by reference in C++



Let's understand call by value and call by reference in C++ language one by one.

Call by value in C++

In call by value, **original value is not modified**.

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Let's try to understand the concept of call by value in C++ language by the example given below:

```
1. #include <iostream>
2. using namespace std;
3. void change(int data);
4. int main()
5. {
6.     int data = 3;
7.     change(data);
8.     cout << "Value of the data is: " << data << endl;
9.     return 0;
10. }
11. void change(int data)
12. {
13.     data = 5;
14. }
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in C++ language by the example given below:

```
1. #include<iostream>
2. using namespace std;
3. void swap(int *x, int *y)
4. {
5.     int swap;
6.     swap=*x;
7.     *x=*y;
8.     *y=swap;
9. }
10. int main()
11. {
12.     int x=500, y=100;
13.     swap(&x, &y); // passing value to function
14.     cout<<"Value of x is: "<<x<<endl;
15.     cout<<"Value of y is: "<<y<<endl;
16.     return 0;
17. }
```

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

C++ Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.

A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.

Let's see a simple example of recursion.

```
1. recursionfunction(){  
2. recursionfunction(); //calling self function  
3. }
```

C++ Recursion Example

Let's see an example to print factorial number using recursion in C++ language.

```
1. #include<iostream>  
2. using namespace std;  
3. int main()  
4. {  
5. int factorial(int);
```



```

6. int fact,value;
7. cout<<"Enter any number: ";
8. cin>>value;
9. fact=factorial(value);
10. cout<<"Factorial of a number is: "<<fact<<endl;
11. return 0;
12. }
13. int factorial(int n)
14. {
15. if(n<0)
16. return (-1); /*Wrong value*/
17. if(n==0)
18. return (1); /*Terminating condition*/
19. else
20. {
21. return (n*factorial(n-1));
22. }
23. }

```

Output:

```

Enter any number: 5
Factorial of a number is: 120

```

We can understand the above program of recursive method call by the figure given below:

```

return 5 * factorial(4) = 120
└─ return 4 * factorial(3) = 24
    └─ return 3 * factorial(2) = 6
        └─ return 2 * factorial(1) = 2
            └─ return 1 * factorial(0) = 1

```

javaTpoint.com

$1 * 2 * 3 * 4 * 5 = 120$

Fig: Recursion

sizeof() operator in C++

The sizeof() is an operator that evaluates the size of data type, constants, variable. It is a compile-time operator as it returns the size of any variable or a constant at the compilation time.

1. **sizeof**(data_type);

In the above syntax, the data_type can be the data type of the data, variables, constants, unions, structures, or any other user-defined data type.

The sizeof () operator can be applied to the following operand types:

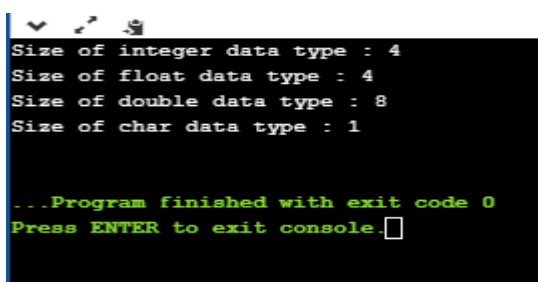
- **When an operand is of data type**

If the parameter of a **sizeof()** operator contains the data type of a variable, then the **sizeof()** operator will return the size of the data type.

Let's understand this scenario through an example.

1. `#include <iostream>`
2. `using namespace std;`
3. `int main()`
4. `{`
5. `// Determining the space in bytes occupied by each data type.`
6. `std::cout << "Size of integer data type : " << sizeof(int) << std::endl;`
7. `std::cout << "Size of float data type : " << sizeof(float) << std::endl;`
8. `std::cout << "Size of double data type : " << sizeof(double) << std::endl;`
9. `std::cout << "Size of char data type : " << sizeof(char) << std::endl;`
10. `return 0;`
11. `}`

Output



```
Size of integer data type : 4
Size of float data type : 4
Size of double data type : 8
Size of char data type : 1

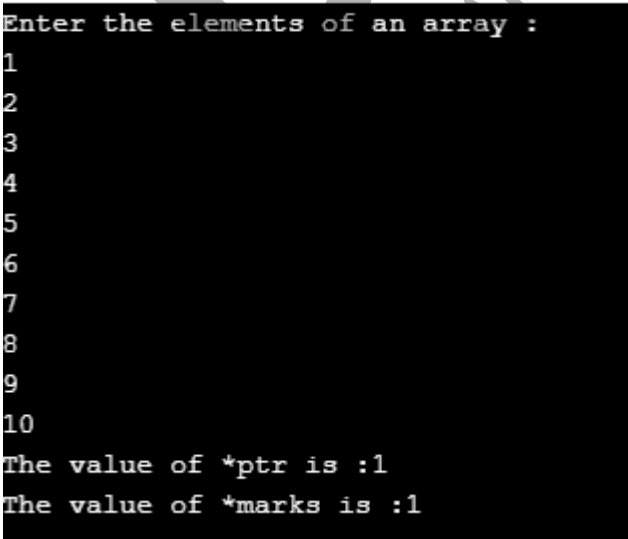
...Program finished with exit code 0
Press ENTER to exit console.
```

C++ Array of Pointers

Let's understand this scenario through an example.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int *ptr; // integer pointer declaration
6.     int marks[10]; // marks array declaration
7.     std::cout << "Enter the elements of an array : " << std::endl;
8.     for(int i=0;i<10;i++)
9.     {
10.         cin>>marks[i];
11.     }
12.     ptr=marks; // both marks and ptr pointing to the same element..
13.     std::cout << "The value of *ptr is : " << *ptr << std::endl;
14.     std::cout << "The value of *marks is : " << *marks << std::endl;
15. }
```

Output



```
Enter the elements of an array :
1
2
3
4
5
6
7
8
9
10
The value of *ptr is :1
The value of *marks is :1
```

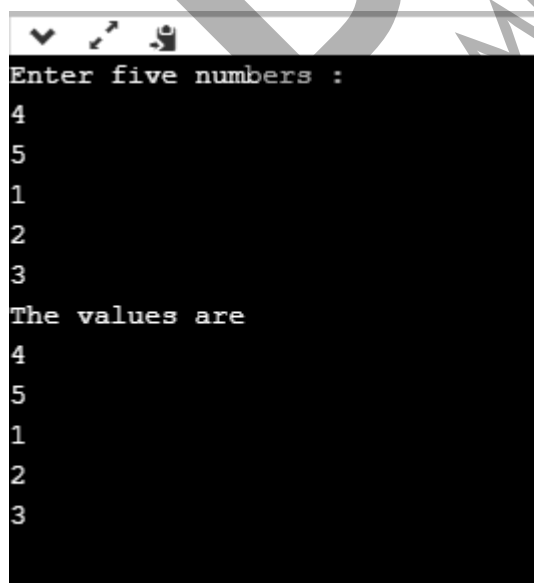
Let's understand through an example.

```

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int ptr1[5]; // integer array declaration
6.     int *ptr2[5]; // integer array of pointer declaration
7.     std::cout << "Enter five numbers : " << std::endl;
8.     for(int i=0;i<5;i++)
9.     {
10.         std::cin >> ptr1[i];
11.     }
12.     for(int i=0;i<5;i++)
13.     {
14.         ptr2[i]=&ptr1[i];
15.     }
16.     // printing the values of ptr1 array
17.     std::cout << "The values are" << std::endl;
18.     for(int i=0;i<5;i++)
19.     {
20.         std::cout << *ptr2[i] << std::endl;
21.     }
22. }

```

Output



```

Enter five numbers :
4
5
1
2
3
The values are
4
5
1
2
3

```

C++ OOPs Concepts

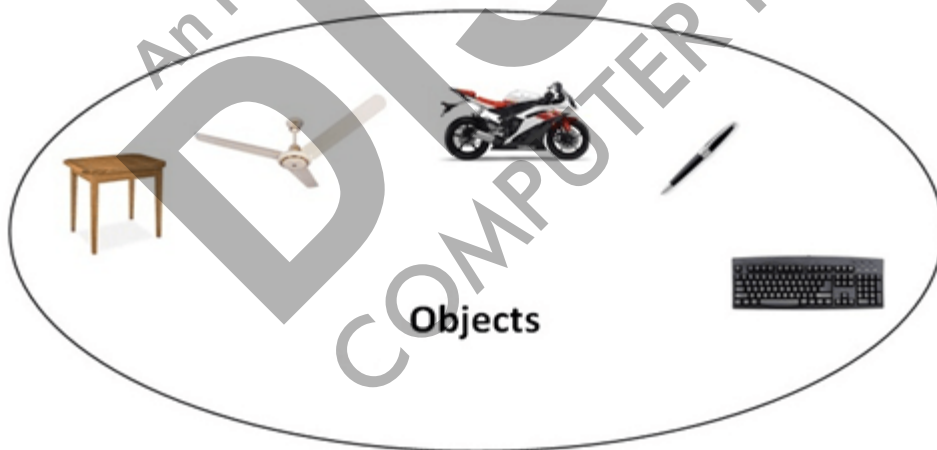
The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

Object Oriented Programming is a paradigm that provides many concepts such as **inheritance**, **data binding**, **polymorphism etc.**

The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language. ®

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:



- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In C++, we use Function overloading and Function overriding to achieve polymorphism.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

C++ Object and Class

Since C++ is an object-oriented language, program is designed using objects and classes in C++.

C++ Object

In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object of student class using s1 as the reference variable.

1. Student s1; *//creating an object of Student*

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

C++ Class

In C++, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

Let's see an example of C++ class that has three fields only.

```
1. class Student
2. {
3.     public:
4.     int id; //field or data member
5.     float salary; //field or data member
6.     String name; //field or data member
7. }
```

C++ Object and Class Example

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

```
1. #include <iostream>
2. using namespace std;
3. class Student {
4.     public:
5.     int id; //data member (also instance variable)
6.     string name; //data member (also instance variable)
7. };
8. int main() {
9.     Student s1; //creating an object of Student
10.    s1.id = 201;
11.    s1.name = "Sonoo Jaiswal";
12.    cout<<s1.id<<endl;
13.    cout<<s1.name<<endl;
14.    return 0;
15. }
```

Output:

```
201
Sonoo Jaiswal
```

C++ Class Example: Initialize and Display data through method

Let's see another example of C++ class where we are initializing and displaying object through method.

```
1. #include <iostream>
2. using namespace std;
3. class Student {
4.     public:
5.         int id; //data member (also instance variable)
6.         string name; //data member (also instance variable)
7.         void insert(int i, string n)
8.         {
9.             id = i;
10.            name = n;
11.        }
12.        void display()
13.        {
14.            cout << id << " " << name << endl;
15.        }
16. };
17. int main(void) {
18.     Student s1; //creating an object of Student
19.     Student s2; //creating an object of Student
20.     s1.insert(201, "Sonoo");
21.     s2.insert(202, "Nakul");
22.     s1.display();
23.     s2.display();
24.     return 0;
25. }
```

Output:

```
201 Sonoo
202 Nakul
```

C++ Class Example: Store and Display Employee Information

Let's see another example of C++ class where we are storing and displaying employee information using method.

```
1. #include <iostream>
2. using namespace std;
3. class Employee {
4.     public:
5.         int id; //data member (also instance variable)
6.         string name; //data member (also instance variable)
7.         float salary;
8.         void insert(int i, string n, float s)
9.         {
10.            id = i;
11.            name = n;
12.            salary = s;
13.        }
14.        void display()
15.        {
16.            cout << id << " " << name << " " << salary << endl;
17.        }
18. };
19. int main(void) {
20.     Employee e1; //creating an object of Employee
21.     Employee e2; //creating an object of Employee
22.     e1.insert(201, "Sonoo", 990000);
23.     e2.insert(202, "Nakul", 29000);
24.     e1.display();
25.     e2.display();
26.     return 0;
27. }
```

Output:

```
201 Sonoo 990000
202 Nakul 29000
```

C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

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

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream>
using namespace std;
class Employee {
public:
int id;//data member (also instance variable)
string name;//data member(also instance variable)
float salary;
Employee(int i, string n, float s)
{
id = i;
name = n;
salary = s;
}
void display()
{
cout<<id<<" "<<name<<" "<<salary<<endl;
}
};
int main(void) {
Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
Employee e2=Employee(102, "Nakul", 59000);
e1.display();
e2.display();
return 0;
}
```

Output:

```
101 Sonoo 890000
102 Nakul 59000
```

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
```

C++ this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

C++ this Pointer Example

Let's see the example of this keyword in C++ that refers to the fields of current class.

```
1. #include <iostream>  
2. using namespace std;  
3. class Employee {  
4.     public:  
5.         int id; //data member (also instance variable)  
6.         string name; //data member(also instance variable)  
7.         float salary;  
8.         Employee(int id, string name, float salary)  
9.         {  
10.            this->id = id;  
11.            this->name = name;  
12.            this->salary = salary;  
13.        }  
14.        void display()  
15.        {  
16.            cout<<id<<" "<<name<<" "<<salary<<endl;  
17.        }  
18. };
```

```
19. int main(void) {
20.     Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
21.     Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
22.     e1.display();
23.     e2.display();
24.     return 0;
25. }
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

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

```
1. #include <iostream>
2. using namespace std;
3. class Box
4. {
5.     private:
6.         int length;
7.     public:
8.         Box()
9.         {
10.            Length=0;
11.        }
12.        friend int printLength(Box); //friend function
13. };
14. int printLength(Box b)
15. {
16.     b.length += 10;
17.     return b.length;
18. }
19. int main()
20. {
21.     Box b;
22.     cout<<"Length of box: "<< printLength(b)<<endl;
23.     return 0;
24. }
```

Output:

```
Length of box: 10
```


Let's see a simple example when the function is friendly to two classes.

```
1. #include <iostream>
2. using namespace std;
3. class B;      // forward declaration.
4. class A
5. {
6.     int x;
7.     public:
8.     void setdata(int i)
9.     {
10.         x=i;
11.     }
12.     friend void min(A,B);    // friend function.
13. };
14. class B
15. {
16.     int y;
17.     public:
18.     void setdata(int i)
19.     {
20.         y=i;
21.     }
22.     friend void min(A,B);    // friend function
23. };
24. void min(A a,B b)
25. {
26.     if(a.x<=b.y)
27.         std::cout << a.x << std::endl;
28.     else
29.         std::cout << b.y << std::endl;
30. }
31. int main()
32. {
33.     A a;
34.     B b;
35.     a.setdata(10);
```

```
36. b.setdata(20);
37. min(a,b);
38. return 0;
39. }
```

Output:

```
10
```

C++ Friend class

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. class A
6. {
7.     int x = 5;
8.     friend class B;    // friend class.
9. };
10. class B
11. {
12. public:
13.     void display(A &a)
14.     {
15.         cout << "value of x is : " << a.x;
16.     }
17. };
18. int main()
19. {
20.     A a;
21.     B b;
22.     b.display(a);
23.     return 0;
24. }
```

Output:

```
value of x is : 5
```

C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

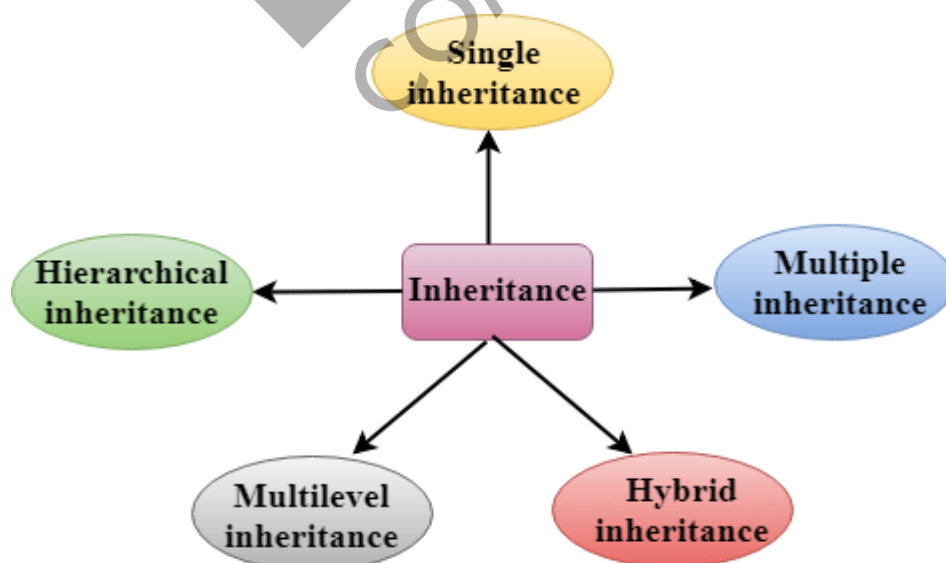
Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types Of Inheritance

C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

1. **class** derived_class_name :: visibility-mode base_class_name
2. {
3. // body of the derived class.
4. }

Where,

derived_class_name is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name is the name of the base class.

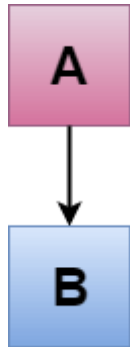
- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

C++ Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
1. #include <iostream>
2. using namespace std;
3. class Account {
4.     public:
5.         float salary = 60000;
6. };
7. class Programmer: public Account {
8.     public:
9.         float bonus = 5000;
10. };
11. int main(void) {
12.     Programmer p1;
13.     cout << "Salary: " << p1.salary << endl;
14.     cout << "Bonus: " << p1.bonus << endl;
15.     return 0;
16. }
```

Output:

```
Salary: 60000  
Bonus: 5000
```

In the above example, Employee is the **base** class and Programmer is the **derived** class.

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

```
1. #include <iostream>  
2. using namespace std;  
3. class Animal {  
4.     public:  
5.     void eat() {  
6.         cout<<"Eating..."<<endl;  
7.     }  
8. };  
9. class Dog: public Animal  
10. {  
11.     public:  
12.     void bark(){  
13.         cout<<"Barking...";  
14.     }  
15. };  
16. int main(void) {  
17.     Dog d1;  
18.     d1.eat();  
19.     d1.bark();  
20.     return 0;  
21. }
```

Output:

```
Eating...  
Barking...
```

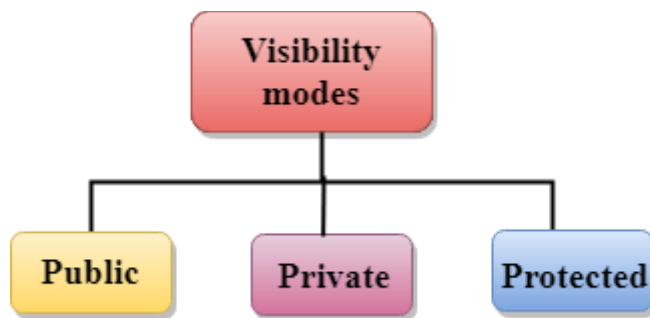
Let's see a simple example.

```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5.     int a = 4;
6.     int b = 5;
7.     public:
8.     int mul()
9.     {
10.         int c = a*b;
11.         return c;
12.     }
13. };
14.
15. class B : private A
16. {
17.     public:
18.     void display()
19.     {
20.         int result = mul();
21.         std::cout << "Multiplication of a and b is : " << result << std::endl;
22.     }
23. };
24. int main()
25. {
26.     B b;
27.     b.display();
28.
29.     return 0;
30. }
```

Output:

```
Multiplication of a and b is : 20
```

Visibility modes can be classified into three categories:



- **Public:** When the member is declared as public, it is accessible to all the functions of the program.
- **Private:** When the member is declared as private, it is accessible within the class only.
- **Protected:** When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Visibility of Inherited Members

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.



C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```
1. #include <iostream>
2. using namespace std;
3. class Animal {
4.     public:
5.     void eat() {
6.         cout<<"Eating..."<<endl;
7.     }
8. };
9. class Dog: public Animal
10. {
11.     public:
12.     void bark(){
13.         cout<<"Barking..."<<endl;
14.     }
```

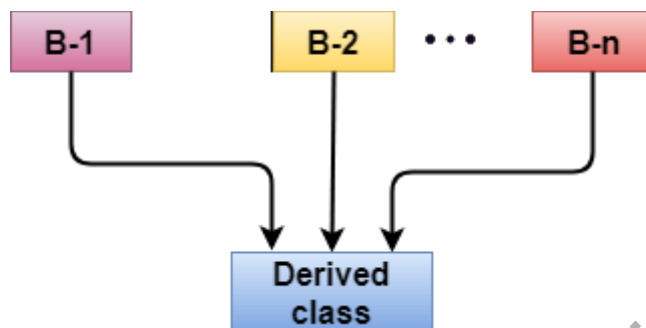
```
15. };
16. class BabyDog: public Dog
17. {
18.     public:
19.     void weep() {
20.         cout<<"Weeping...";
21.     }
22. };
23. int main(void) {
24.     BabyDog d1;
25.     d1.eat();
26.     d1.bark();
27.     d1.weep();
28.     return 0;
29. }
```

Output:

```
Eating...
Barking...
Weeping...
```

C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class:

1. `classD` : visibility B-1, visibility B-2, ?
2. {
3. // Body of the class;
4. }

Let's see a simple example of multiple inheritance.

- ```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5. protected:
6. int a;
7. public:
8. void get_a(int n)
9. {
10. a = n;
11. }
12. };
13.
14. class B
15. {
16. protected:
17. int b;
```

```

18. public:
19. void get_b(int n)
20. {
21. b = n;
22. }
23. };
24. class C : public A, public B
25. {
26. public:
27. void display()
28. {
29. std::cout << "The value of a is : " << a << std::endl;
30. std::cout << "The value of b is : " << b << std::endl;
31. cout << "Addition of a and b is : " << a + b;
32. }
33. };
34. int main()
35. {
36. C c;
37. c.get_a(10);
38. c.get_b(20);
39. c.display();
40.
41. return 0;
42. }

```

Output:

```

The value of a is : 10
The value of b is : 20
Addition of a and b is : 30

```

In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

## Ambiguity Resolution in Inheritance

```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5. public:
6. void display()
7. {
8. std::cout << "Class A" << std::endl;
9. }
10. };
11. class B
12. {
13. public:
14. void display()
15. {
16. std::cout << "Class B" << std::endl;
17. }
18. };
19. class C : public A, public B
20. {
21. void view()
22. {
23. display();
24. }
25. };
26. int main()
27. {
28. C c;
29. c.display();
30. return 0;
31. }
```

Output:

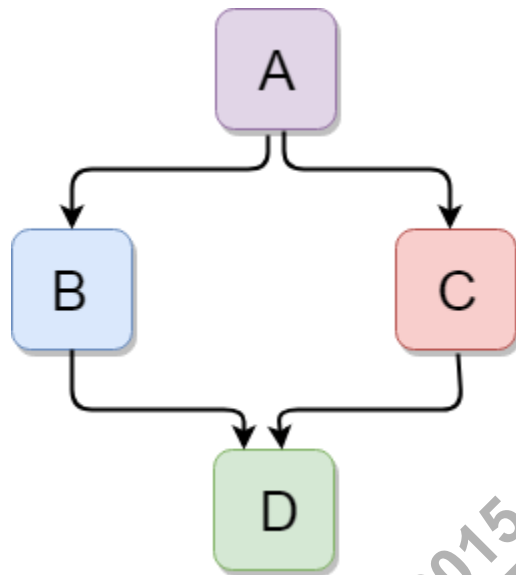
```
error: reference to 'display' is ambiguous
 display();
```

```
1. classC : public A, public B
2. {
3. void view()
4. {
5. A :: display(); // Calling the display() function of class A.
6. B :: display(); // Calling the display() function of class B.
7.
8. }
9. };
```

An ISO 9001:2015 Certified  
**DISHA**<sup>®</sup>  
COMPUTER INSTITUTE

# C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



Let's see a simple example:

```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5. protected:
6. int a;
7. public:
8. void get_a()
9. {
10. std::cout << "Enter the value of 'a' : " << std::endl;
11. cin >> a;
12. }
13. };
14.
15. class B : public A
16. {
17. protected:
18. int b;
19. public:
```

```

20. void get_b()
21. {
22. std::cout << "Enter the value of 'b' : " << std::endl;
23. cin>>b;
24. }
25. };
26. class C
27. {
28. protected:
29. int c;
30. public:
31. void get_c()
32. {
33. std::cout << "Enter the value of c is : " << std::endl;
34. cin>>c;
35. }
36. };
37.
38. class D : public B, public C
39. {
40. protected:
41. int d;
42. public:
43. void mul()
44. {
45. get_a();
46. get_b();
47. get_c();
48. std::cout << "Multiplication of a,b,c is : " << a*b*c << std::endl;
49. }
50. };
51. int main()
52. {
53. D d;
54. d.mul();
55. return 0;
56. }

```

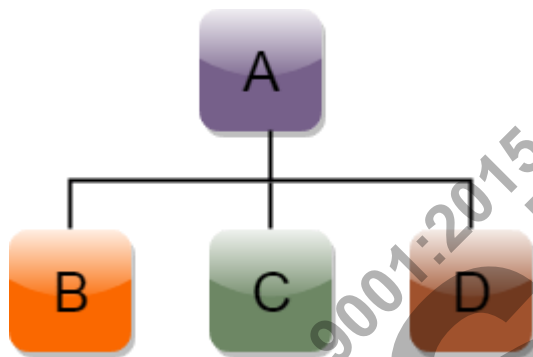


Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

## C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



**Syntax of Hierarchical inheritance:**

```
1. class A
2. {
3. // body of the class A.
4. }
5. class B : public A
6. {
7. // body of class B.
8. }
9. class C : public A
10. {
11. // body of class C.
12. }
13. class D : public A
14. {
15. // body of class D.
16. }
```

Let's see a simple example:

```
1. #include <iostream>
2. using namespace std;
3. class Shape // Declaration of base class.
4. {
5. public:
6. int a;
7. int b;
8. void get_data(int n,int m)
9. {
10. a = n;
11. b = m;
12. }
13. };
14. class Rectangle : public Shape // inheriting Shape class
15. {
16. public:
17. int rect_area()
18. {
19. int result = a*b;
20. return result;
21. }
22. };
23. class Triangle : public Shape // inheriting Shape class
24. {
25. public:
26. int triangle_area()
27. {
28. float result = 0.5*a*b;
29. return result;
30. }
31. };
32. int main()
33. {
34. Rectangle r;
35. Triangle t;
```

```
36. int length,breadth,base,height;
37. std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
38. cin>>length>>breadth;
39. r.get_data(length,breadth);
40. int m = r.rect_area();
41. std::cout << "Area of the rectangle is : " <<m<< std::endl;
42. std::cout << "Enter the base and height of the triangle: " << std::endl;
43. cin>>base>>height;
44. t.get_data(base,height);
45. float n = t.triangle_area();
46. std::cout <<"Area of the triangle is : " <<n<<std::endl;
47. return 0;
48.}
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

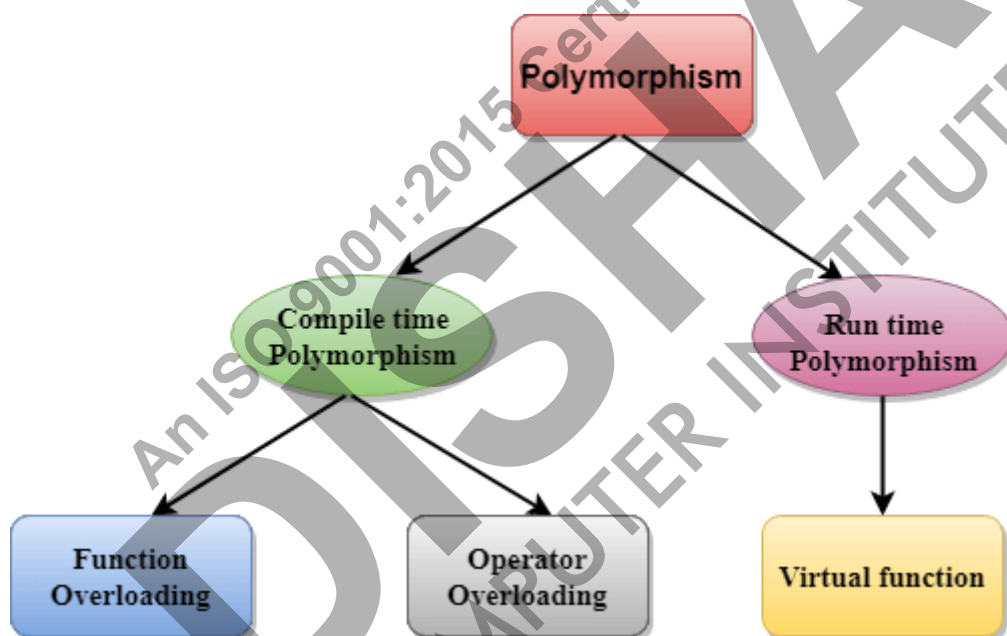
# C++ Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

## Real Life Example Of Polymorphism

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++:



- **Compile time polymorphism:** The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
1. class A // base class declaration.
2. {
3. int a;
4. public:
5. void display()
```

```

6. {
7. cout<< "Class A ";
8. }
9. };
10. class B : public A // derived class declaration.
11. {
12. int b;
13. public:
14. void display()
15. {
16. cout<<"Class B";
17. }
18. };

```

In the above case, the prototype of display() function is the same in both the **base and derived class**. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

- **Run time polymorphism:** Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

## Differences b/w compile time and run time polymorphism.

| Compile time polymorphism                                                                                                                                                | Run time polymorphism                                                                                                                          |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| The function to be invoked is known at the compile time.                                                                                                                 | The function to be invoked is known at the run time.                                                                                           |
| It is also known as overloading, early binding and static binding.                                                                                                       | It is also known as overriding, Dynamic binding and late binding.                                                                              |
| Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters. | Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters. |

|                                                                           |                                                                |
|---------------------------------------------------------------------------|----------------------------------------------------------------|
| It is achieved by function overloading and operator overloading.          | It is achieved by virtual functions and pointers.              |
| It provides fast execution as it is known at the compile time.            | It provides slow execution as it is known at the run time.     |
| It is less flexible as mainly all the things execute at the compile time. | It is more flexible as all the things execute at the run time. |

## C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.

// an example without the virtual keyword.

```

1. #include <iostream>
2. using namespace std;
3. class Animal {
4. public:
5. void eat(){
6. cout<<"Eating...";
7. }
8. };
9. class Dog: public Animal
10. {
11. public:
12. void eat()
13. { cout<<"Eating bread...";
14. }
15. };
16. int main(void) {
17. Dog d = Dog();
18. d.eat();
19. return 0;
20. }
```

**Output:**

```
Eating bread...
```

## C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.

// an example with virtual keyword.

```
1. #include <iostream>
2. using namespace std;
3. class Shape { // base class
4. public:
5. virtual void draw() { // virtual function
6. cout << "drawing..." << endl;
7. }
8. };
9. class Rectangle: public Shape // inheriting Shape class.
10. {
11. public:
12. void draw()
13. {
14. cout << "drawing rectangle..." << endl;
15. }
16. };
17. class Circle: public Shape // inheriting Shape class.
18. {
19. public:
20. void draw()
21. {
22. cout << "drawing circle..." << endl;
23. }
24. };
25. };
26. int main(void) {
27. Shape *s; // base class pointer.
28. Shape sh; // base class object.
29. Rectangle rec;
30. Circle cir;
```

```

31. s=&sh;
32. s->draw();
33. s=&rec;
34. s->draw();
35. s=○
36. s->draw();
37. }

```

### Output:

```

drawing...
drawing rectangle...
drawing circle...

```

## Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```

1. #include <iostream>
2. using namespace std;
3. class Animal { // base class declaration.
4. public:
5. string color = "Black";
6. };
7. class Dog: public Animal // inheriting Animal class.
8. {
9. public:
10. string color = "Grey";
11. };
12. int main(void) {
13. Animal d= Dog();
14. cout<<d.color;
15. }

```

### Output:

```

Black

```



# C++ Overloading (Function and Operator)

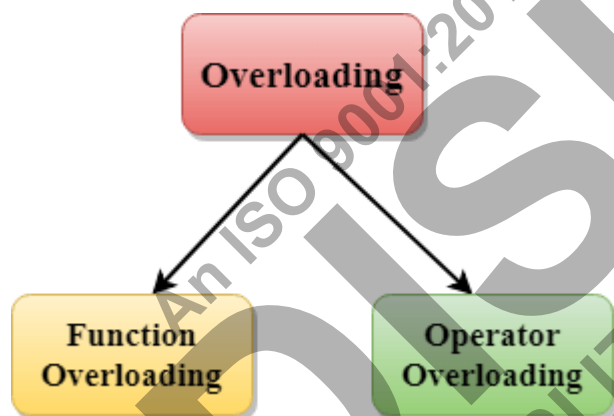
If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- Function overloading
- Operator overloading



## C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

## C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

```
1. #include <iostream>
2. using namespace std;
3. class Cal {
4. public:
5. static int add(int a,int b){
6. return a + b;
7. }
8. static int add(int a, int b, int c)
9. {
10. return a + b + c;
11. }
12. };
13. int main(void) {
14. Cal C; // class object declaration.
15. cout<<C.add(10, 20)<<endl;
16. cout<<C.add(12, 20, 23);
17. return 0;
18. }
```

**Output:**

```
30
55
```

Let's see the simple example when the type of the arguments vary.

// Program of function overloading with different types of arguments.

```
1. #include<iostream>
2. using namespace std;
3. int mul(int,int);
4. float mul(float,int);
5.
6.
7. int mul(int a,int b)
```

```

8. {
9. return a*b;
10. }
11. float mul(double x, int y)
12. {
13. return x*y;
14. }
15. int main()
16. {
17. int r1 = mul(6,7);
18. float r2 = mul(0.2,3);
19. std::cout << "r1 is : " << r1 << std::endl;
20. std::cout << "r2 is : " << r2 << std::endl;
21. return 0;
22. }

```

#### Output:

```

r1 is : 42
r2 is : 0.6

```

## C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

#### Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(\*)
- ternary operator(?:)

## Syntax of Operator Overloading

1. `return_type class_name :: operator op(argument_list)`
2. `{`
3. `// body of the function.`
4. `}`

Where the **return type** is the type of value returned by the function.

**class\_name** is the name of the class.

**operator op** is an operator function where `op` is the operator being overloaded, and the operator is the keyword.

## Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains at least one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

## C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

1. `#include <iostream>`
2. `using namespace std;`
3. `class Test`
4. `{`
5. `private:`
6. `int num;`

```

7. public:
8. Test()
9. {
10. Num=8;
11. }
12. void operator ++() {
13. num = num+2;
14. }
15. void Print() {
16. cout<<"The Count is: "<<num;
17. }
18. };
19. int main()
20. {
21. Test tt;
22. ++tt; // calling of a function "void operator ++()"
23. tt.Print();
24. return 0;
25. }

```

### Output:

```
The Count is: 10
```

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

```

1. #include <iostream>
2. using namespace std;
3. class A
4. {
5.
6. int x;
7. public:
8. A(){}
9. A(int i)
10. {
11. x=i;

```

```
12. }
13. void operator+(A);
14. void display();
15. };
16.
17. void A :: operator+(A a)
18. {
19.
20. int m = x+a.x;
21. cout<<"The result of the addition of two objects is : "<<m;
22.
23. }
24. int main()
25. {
26. A a1(5);
27. A a2(4);
28. a1+a2;
29. return 0;
30. }
```

**Output:**

```
The result of the addition of two objects is : 9
```

# C++ Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

## C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

```
1. #include <iostream>
2. using namespace std;
3. class Animal {
4. public:
5. void eat(){
6. cout<<"Eating...";
7. }
8. };
9. class Dog: public Animal
10. {
11. public:
12. void eat()
13. {
14. cout<<"Eating bread...";
15. }
16. };
17. int main(void) {
18. Dog d = Dog();
19. d.eat();
20. return 0;
21. }
```

Output:

```
Eating bread...
```

# C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
  - It is used to tell the compiler to perform dynamic linkage or late binding on the function.
  - There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
  - A 'virtual' is a keyword preceding the normal declaration of a function.
  - When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.
- 

## Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

---

### Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.



```

1. #include <iostream>
2. using namespace std;
3. class A
4. {
5. int x=5;
6. public:
7. void display()
8. {
9. std::cout << "Value of x is : " << x<<std::endl;
10. }
11. };
12. class B: public A
13. {
14. int y = 10;
15. public:
16. void display()
17. {
18. std::cout << "Value of y is : " << y<< std::endl;
19. }
20. };
21. int main()
22. {
23. A *a;
24. B b;
25. a = &b;
26. a->display();
27. return 0;
28. }

```

### Output:

```
Value of x is : 5
```

In the above example, \* a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

## C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoke the derived class in a program.

```
1. #include <iostream>
2. {
3. public:
4. virtual void display()
5. {
6. cout << "Base class is invoked" << endl;
7. }
8. };
9. class B:public A
10. {
11. public:
12. void display()
13. {
14. cout << "Derived Class is invoked" << endl;
15. }
16. };
17. int main()
18. {
19. A* a; //pointer of base class
20. B b; //object of derived class
21. a = &b;
22. a->display(); //Late Binding occurs
23. }
```

**Output:**

```
Derived Class is invoked
```

## Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

**Pure virtual function can be defined as:**

1. **virtual void** display() = 0;

**Let's see a simple example:**

```
1. #include <iostream>
2. using namespace std;
3. class Base
4. {
5. public:
6. virtual void show() = 0;
7. };
8. class Derived : public Base
9. {
10. public:
11. void show()
12. {
13. std::cout << "Derived class is derived from the base class." << std::endl;
14. }
15. };
16. int main()
17. {
18. Base *bptr;
19. //Base b;
20. Derived d;
```

```
21. bptr = &d;
22. bptr->show();
23. return 0;
24. }
```

### Output:

```
Derived class is derived from the base class.
```

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

## C++ Abstract class

In C++ class is made abstract by declaring at least one of its functions as `<>strong>` pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.

Let's see an example of abstract class in C++ which has one abstract method `draw()`. Its implementation is provided by derived classes: `Rectangle` and `Circle`. Both classes have different implementation.

```
1. #include <iostream>
2. using namespace std;
3. class Shape
4. {
5. public:
6. virtual void draw()=0;
7. };
8. class Rectangle : Shape
9. {
10. public:
11. void draw()
12. {
13. cout << "drawing rectangle..." << endl;
14. }
15.};
16. class Circle : Shape
17.{
18. public:
```

```

19. void draw()
20. {
21. cout <<"drawing circle..." << endl;
22. }
23. };
24. int main() {
25. Rectangle rec;
26. Circle cir;
27. rec.draw();
28. cir.draw();
29. return 0;
30. }

```

Output:

```

drawing rectangle...
drawing circle...

```

## Data Abstraction in C++

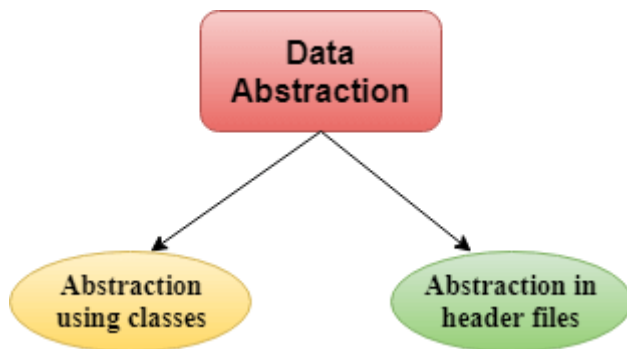
- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.
- Data Abstraction is a programming technique that depends on the separation of the interface and implementation details of the program.
- Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC separates the implementation details from the external interface.
- C++ provides a great level of abstraction. For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.

In C++ program if we implement class with private and public members then it is an example of data abstraction.

**Data Abstraction can be achieved in two ways:**

- Abstraction using classes

- Abstraction in header files.



**Abstraction using classes:** An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

**Abstraction in header files:** Another type of abstraction is header file. For example, `pow()` function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hide all the implementation details from the user.

#### Access Specifiers Implement Abstraction:

- **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
- **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

**// program to calculate the power of a number.**

```

1. #include <iostream>
2. #include <math.h>
3. using namespace std;
4. int main()
5. {
6. int n = 4;
7. int power = 3;
8. int result = pow(n,power); // pow(n,power) is the power function

```

```

9. std::cout << "Cube of n is : " << result << std::endl;
10. return 0;
11. }

```

### Output:

```
Cube of n is : 64
```

In the above example, `pow()` function is used to calculate 4 raised to the power 3. The `pow()` function is present in the `math.h` header file in which all the implementation details of the `pow()` function is hidden.

**Let's see a simple example of data abstraction using classes.**

```

1. #include <iostream>
2. using namespace std;
3. class Sum
4. {
5. private: int x, y, z; // private variables
6. public:
7. void add()
8. {
9. cout << "Enter two numbers: ";
10. cin >> x >> y;
11. z = x + y;
12. cout << "Sum of two number is: " << z << endl;
13. }
14. };
15. int main()
16. {
17. Sum sm;
18. sm.add();
19. return 0;
20. }

```

### Output:

```

Enter two numbers:
3
6
Sum of two number is: 9

```

In the above example, abstraction is achieved using classes. A class 'Sum' contains the private members x, y and z are only accessible by the member functions of the class.

## Advantages Of Abstraction:

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low level code.
- Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.

## C++ Namespaces

Namespaces in C++ are used to organize too many classes so that it can be easy to handle the application.

For accessing the class of a namespace, we need to use namespace::classname. We can use **using** keyword so that we don't have to use complete name all the time.

In C++, global namespace is the root namespace. The global::std will always refer to the namespace "std" of C++ Framework.

---

## C++ namespace Example

Let's see the simple example of namespace which include variable and functions.

```
1. #include <iostream>
2. using namespace std;
3. namespace First {
4. void sayHello() {
5. cout<<"Hello First Namespace"<<endl;
6. }
7. }
8. namespace Second {
9. void sayHello() {
10. cout<<"Hello Second Namespace"<<endl;
```



```
11. }
12. }
13. int main()
14. {
15. First::sayHello();
16. Second::sayHello();
17. return 0;
18. }
```

Output:

```
Hello First Namespace
Hello Second Namespace
```

## C++ Strings

In C++, string is an object of **std::string** class that represents sequence of characters. We can perform many operations on strings such as concatenation, comparison, conversion etc.

### C++ String Example

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

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4. string s1 = "Hello";
5. char ch[] = { 'C', '+', '+' };
6. string s2 = string(ch);
7. cout<<s1<<endl;
8. cout<<s2<<endl;
9. }
```

Output:

```
Hello
C++
```

---

## C++ String Compare Example

Let's see the simple example of string comparison using strcmp() function.

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4. int main ()
5. {
6. char key[] = "mango";
7. char buffer[50];
8. do {
9. cout << "What is my favourite fruit? ";
10. cin >> buffer;
11. } while (strcmp (key,buffer) != 0);
12. cout << "Answer is correct!!" << endl;
13. return 0;
14. }
```

Output:

```
What is my favourite fruit? apple
What is my favourite fruit? banana
What is my favourite fruit? mango
Answer is correct!!
```

---

## C++ String Concat Example

Let's see the simple example of string concatenation using strcat() function.

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4. int main()
5. {
6. char key[25], buffer[25];
7. cout << "Enter the key string: ";
```

```
8. cin.getline(key, 25);
9. cout << "Enter the buffer string: ";
10. cin.getline(buffer, 25);
11. strcat(key, buffer);
12. cout << "Key = " << key << endl;
13. cout << "Buffer = " << buffer<<endl;
14. return 0;
15. }
```

Output:

```
Enter the key string: Welcome to
Enter the buffer string: C++ Programming.
Key = Welcome to C++ Programming.
Buffer = C++ Programming.
```

## C++ String Copy Example

Let's see the simple example of copy the string using strcpy() function.

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4. int main()
5. {
6. char key[25], buffer[25];
7. cout << "Enter the key string: ";
8. cin.getline(key, 25);
9. strcpy(buffer, key);
10. cout << "Key = " << key << endl;
11. cout << "Buffer = " << buffer<<endl;
12. return 0;
13. }
```

Output:

```
Enter the key string: C++ Tutorial
Key = C++ Tutorial
Buffer = C++ Tutorial
```

## C++ String Length Example

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4. int main()
5. {
6. char ary[] = "Welcome to C++ Programming";
7. cout << "Length of String = " << strlen(ary)<<endl;
8. return 0;
9. }
```

```
Length of String = 26
```

An ISO 9001:2015 Certified  
**DISHA**  
COMPUTER INSTITUTE

# C++ Exception Handling

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from `std::exception` class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

---

## Advantage

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

---

## C++ Exception Classes

In C++ standard exceptions are defined in `<exception>` class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:

---

## C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

- try
- catch, and
- throw

Moreover, we can create user-defined exception which we will learn in next chapters.

# C++ try/catch

In C++ programming, exception handling is performed using try/catch statement. The C++ **try block** is used to place the code that may occur exception. The **catch block** is used to handle the exception.

---

## C++ example without try/catch

```
1. #include <iostream>
2. using namespace std;
3. float division(int x, int y) {
4. return (x/y);
5. }
6. int main () {
7. int i = 50;
8. int j = 0;
9. float k = 0;
10. k = division(i, j);
11. cout << k << endl;
12. return 0;
13. }
```

Output:

```
Floating point exception (core dumped)
```

---

## C++ try/catch example

```
1. #include <iostream>
2. using namespace std;
3. float division(int x, int y) {
4. if(y == 0) {
5. throw "Attempted to divide by zero!";
6. }
7. return (x/y);
8. }
9. int main () {
```

```

10. int i = 25;
11. int j = 0;
12. float k = 0;
13. try {
14. k = division(i, j);
15. cout << k << endl;
16. } catch (const char* e) {
17. cerr << e << endl;
18. }
19. return 0;
20. }

```

Output:

```

Attempted to divide by zero!

```

## C++ User-Defined Exceptions

The new exception can be defined by overriding and inheriting **exception** class functionality.

### C++ user-defined exception example

Let's see the simple example of user-defined exception in which **std::exception** class is used to define the exception.

```

1. #include <iostream>
2. #include <exception>
3. using namespace std;
4. class MyException : public exception{
5. public:
6. const char* what() const throw()
7. {
8. return "Attempted to divide by zero!\n";
9. }
10. };
11. int main()

```

```

12. {
13. try
14. {
15. int x, y;
16. cout << "Enter the two numbers : \n";
17. cin >> x >> y;
18. if (y == 0)
19. {
20. MyException z;
21. throw z;
22. }
23. else
24. {
25. cout << "x / y = " << x/y << endl;
26. }
27. }
28. catch(exception& e)
29. {
30. cout << e.what();
31. }
32. }

```

Output:

```

Enter the two numbers :
10
2
x / y = 5

```

Output:

```

Enter the two numbers :
10
0
Attempted to divide by zero!
-->

```

**Note:** In above example what() is a public method provided by the exception class. It is used to return the cause of an exception.



# C++ Files and Streams

In [C++ programming](#) we are using the **iostream** standard library, it provides **cin** and **cout** methods for reading from input and writing to output respectively.

To read and write from a file we are using the standard C++ library called **fstream**. Let us see the data types define in **fstream** library is:

| Data Type | Description                                                                              |
|-----------|------------------------------------------------------------------------------------------|
| fstream   | It is used to create files, write information to files, and read information from files. |
| ifstream  | It is used to read information from files.                                               |
| ofstream  | It is used to create files and write information to the files.                           |

## C++ FileStream example: writing to a file

Let's see the simple example of writing to a text file **testout.txt** using C++ FileStream programming.

```
1. #include <iostream>
2. #include <fstream>
3. using namespace std;
4. int main () {
5. ofstream filestream("testout.txt");
6. if (filestream.is_open())
7. {
8. filestream << "Welcome to javaTpoint.\n";
9. filestream << "C++ Tutorial.\n";
10. filestream.close();
11. }
12. else cout << "File opening is fail.";
13. return 0;
14. }
```

## Output:

```
The content of a text file testout.txt is set with the data:
Welcome to javaTpoint.
C++ Tutorial.
```

## C++ FileStream example: reading from a file

Let's see the simple example of reading from a text file **testout.txt** using C++ FileStream programming.

```
1. #include <iostream>
2. #include <fstream>
3. using namespace std;
4. int main () {
5. string srg;
6. ifstream filestream("testout.txt");
7. if (filestream.is_open())
8. {
9. while (getline (filestream,srg))
10. {
11. cout << srg << endl;
12. }
13. filestream.close();
14. }
15. else {
16. cout << "File opening is fail." << endl;
17. }
18. return 0;
19. }
```

**Note:** Before running the code a text file named as "testout.txt" is need to be created and the content of a text file is given below:  
Welcome to javaTpoint.  
C++ Tutorial.

## Output:

```
Welcome to javaTpoint.
C++ Tutorial.
```

# Templates in c++

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

## Function Template

The general form of a template function definition is shown here –

```
template <class type> ret-type func-name(parameter list) {
 // body of function
}
```

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b) {
 return a < b ? b : a;
}

int main () {
 int i = 39;
 int j = 20;
 cout << "Max(i, j): " << Max(i, j) << endl;

 double f1 = 13.5;
 double f2 = 20.7;
 cout << "Max(f1, f2): " << Max(f1, f2) << endl;

 string s1 = "Hello";
 string s2 = "World";
 cout << "Max(s1, s2): " << Max(s1, s2) << endl;

 return 0;
}
```

If we compile and run above code, this would produce the following result –

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

**A static member** is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator `::` to identify which class it belongs to.

Let us try the following example to understand the concept of static data members –

[Live Demo](#)

```
#include <iostream>

using namespace std;

class Box {
public:
 static int objectCount;

 // Constructor definition
 Box(double l = 2.0, double b = 2.0, double h = 2.0) {
 cout << "Constructor called." << endl;
 length = l;
 breadth = b;
 height = h;

 // Increase every time object is created
 objectCount++;
 }
 double Volume() {
 return length * breadth * height;
 }

private:
 double length; // Length of a box
 double breadth; // Breadth of a box
 double height; // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void) {
 Box Box1(3.3, 1.2, 1.5); // Declare box1
 Box Box2(8.5, 6.0, 2.0); // Declare box2

 // Print total number of objects.
 cout << "Total objects: " << Box::objectCount << endl;

 return 0;
}
```

```
Constructor called.
Constructor called.
Total objects: 2
```

# Static Function Members

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator ::.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members -

```
#include <iostream>

using namespace std;

class Box {
public:
 static int objectCount;

 // Constructor definition
 Box(double l = 2.0, double b = 2.0, double h = 2.0) {
 cout << "Constructor called." << endl;
 length = l;
 breadth = b;
 height = h;

 // Increase every time object is created
 objectCount++;
 }

 double Volume() {
 return length * breadth * height;
 }

 static int getCount() {
 return objectCount;
 }

private:
 double length; // Length of a box
 double breadth; // Breadth of a box
 double height; // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void) {
```

```
// Print total number of objects before creating object.
cout << "Initial Stage Count: " << Box::getCount() << endl;

Box Box1(3.3, 1.2, 1.5); // Declare box1
Box Box2(8.5, 6.0, 2.0); // Declare box2

// Print total number of objects after creating object.
cout << "Final Stage Count: " << Box::getCount() << endl;

return 0;
}
```

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

```
Initial Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```