COMP 215: OBJECT ORIENTED PROGRAMMING WITH C++

C++ programming language was developed in 1980 by Bjarne Stroustrup at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A. **C++** is an object-oriented programming language. It is an extension to C programming.

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.

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 behaviour 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.

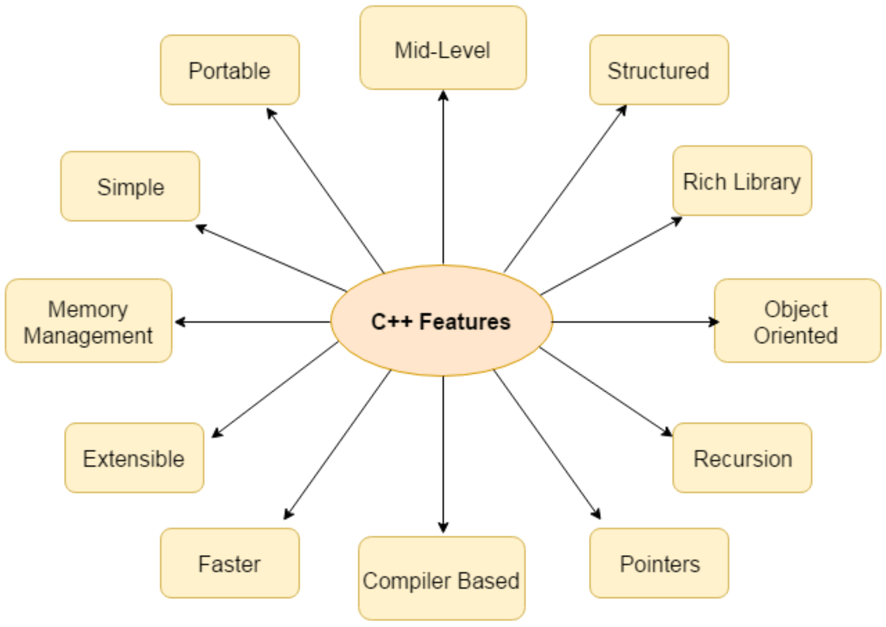
Advantage of OOPs over Procedure-oriented programming language

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++ 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

C++ - Download & Installation

There are many compilers available for C++. You need to download any one – freely available online.

C++ Basic Input/Output

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

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

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

I/O Library Header Files

Let us see the common header files used in C++ programming are:

HeaderFile	Function and Description
<iostream>	It is used to define the **cout, cin and cerr** objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as **setprecision and setw.**
<fstream>	It is used to declare services for user-controlled file processing.

Standard output stream (cout)
The **cout** is a predefined object of **ostream** class. It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console.
Standard input stream (cin)
The **cin** is a predefined object of **istream** class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.
Standard end line (endl)
The **endl** is a predefined object of **ostream** class. It is used to insert a new line characters and flushes the stream.
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:

```
#include <iostream.h>
#include<conio.h>
void main() {
  clrscr();
  cout << "Welcome to C++ Programming.";
  getch();
}
```

#include<iostream.h> includes the **standard input output** library functions. It provides **cin** and **cout** methods for reading from input and writing to output respectively.
#include <conio.h> includes the **console input output** library functions. The getch() function is defined in conio.h file.
void main() The **main() function is the entry point of every program** in C++ language. The void keyword specifies that it returns no value.
cout << "Welcome to C++ Programming." is **used to print the data "Welcome to C++ Programming."** on the console.
getch() The getch() function **asks for a single character**. Until you press any key, it blocks the screen.

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:

type variable_list;

The example of declaring variable is given below:

```
int x;
float y;
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:

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

Rules for defining variables

- A variable can have alphabets, digits and underscore.
- A variable name can start with alphabet and underscore only. It can't start with digit.
- No white space is allowed within variable name.
- A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

```
int a;
int _ab;
int a30;
```

Invalid variable names:

```
int 4;
int x y;
int double;
```

C++ Data Types

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
- Misc Operator

C++ Control Statements

A **control statement** is a **statement** that determines whether other **statements** will be executed.

C++ if-else

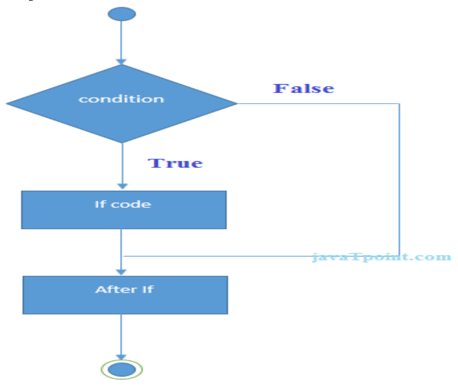
In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.

- 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.

```
if(condition){
//code to be executed
}
```



C++ If Example

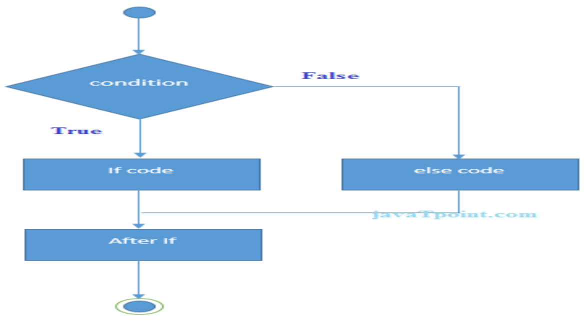
```
#include <iostream>
using namespace std;

int main () {
  int num = 10;
        if (num % 2 == 0)
        {
           cout<<"It is even number";
        }
  return 0;
}
```

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.

```
if(condition){
```

```
//code if condition is true
}else{
//code if condition is false
}
```



C++ If-else Example

```
#include <iostream>
using namespace std;
int main () {
  int num = 11;
        if (num % 2 == 0)
        {
           cout<<"It is even number";
        }
        else
        {
           cout<<"It is odd number";
        }
  return 0;
}
```

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

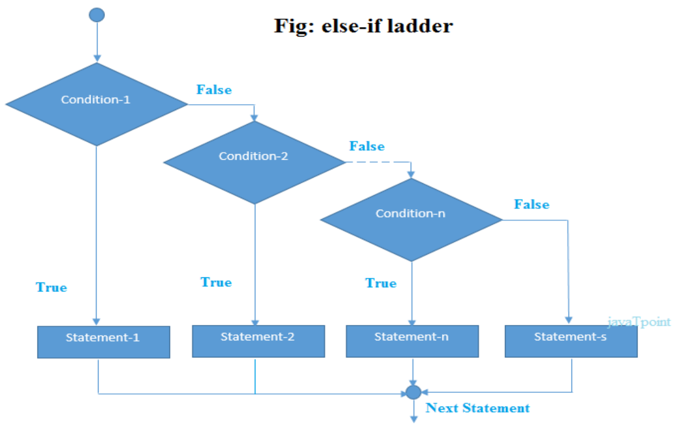
```
#include <iostream>
using namespace std;
int main () {
   int num;
   cout<<"Enter a Number: ";
   cin>>num;
        if (num % 2 == 0)
        {
           cout<<"It is even number"<<endl;
        }
        else
        {
           cout<<"It is odd number"<<endl;
        }
  return 0;
}
```

C++ IF-else-if ladder Statement

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

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

Fig: else-if ladder



C++ If else-if Example

```
#include <iostream>
using namespace std;
int main () {
       int num;
       cout<<"Enter a number to check grade:";
       cin>>num;
            if (num <0 || num >100)
            {
                cout<<"wrong number";
            }
            else if(num >= 0 && num < 50){
                cout<<"Fail";
            }
            else if (num >= 50 && num < 60)
            {
                cout<<"D Grade";
            }
            else if (num >= 60 && num < 70)
            {
                cout<<"C Grade";
            }
            else if (num >= 70 && num < 80)
            {
                cout<<"B Grade";
            }
            else if (num >= 80 && num < 90)
            {
               cout<<"A Grade";
            }
            else if (num >= 90 && num <= 100)
            {
               cout<<"A+ Grade";
            }
    }
```

C++ switch

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

```
switch(expression){
case value1:
 //code to be executed;
 break;
case value2:
 //code to be executed;
 break;
......

default:
 //code to be executed if all cases are not matched;
 break;
}
```

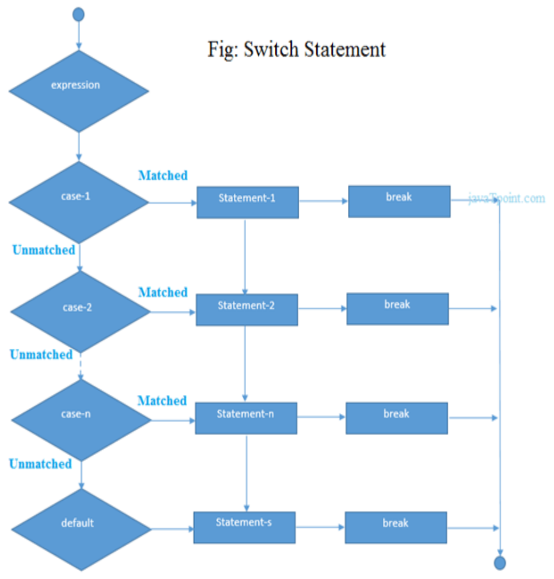


Fig: Switch Statement

C++ Switch Example

```
#include <iostream>
using namespace std;
int main () {
       int num;
       cout<<"Enter a number to check grade:";
       cin>>num;
```

```
 switch (num)
{
   case 10: cout<<"It is 10"; break;
   case 20: cout<<"It is 20"; break;
   case 30: cout<<"It is 30"; break;
   default: cout<<"Not 10, 20 or 30"; break;
}
}
```

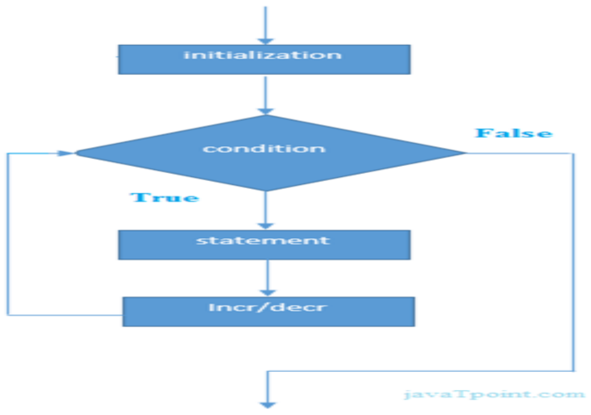
C++ For Loop

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.

```
for(initialization; condition; incr/decr){
//code to be executed
}
```

Flowchart:



C++ For Loop Example

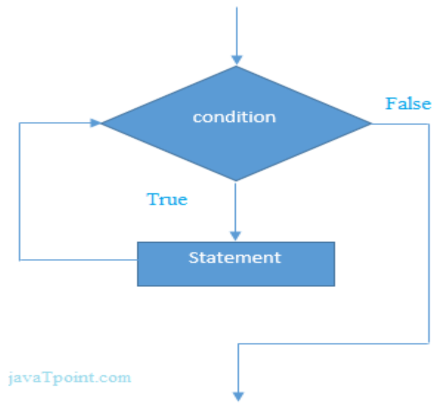
```
#include <iostream>
using namespace std;
int main() {
      for(int i=1;i<=10;i++){
        cout<<i <<"\n";
      }
  }
```

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.

```
#include <iostream>
using namespace std;
int main() {
 int i=1;
        while(i<=10)
      {
          cout<<i <<"\n";
          i++;
        }
   }
```

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.

```
#include <iostream>
using namespace std;
int main () {
        int i=1;
          while(i<=3)
          {
              int j = 1;
              while (j <= 3)
{
```

```
                 cout<<i<<" "<<j<<"\n";
                 j++;
                }
                i++;
              }
          }
```
Output:

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

C++ Infinitive While Loop Example:

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

```
    #include <iostream>
    using namespace std;
    int main () {
            while(true)
              {
                    cout<<"Infinitive While Loop";
              }
          }
```

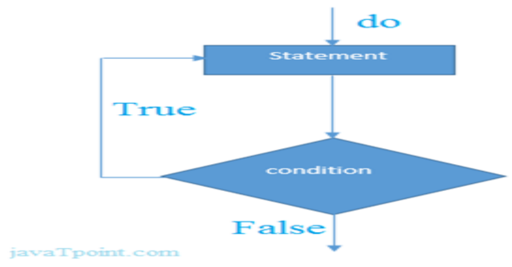
C++ Do-While Loop

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.

```
    do{
    //code to be executed
    }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.

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

```
int main() {
     int i = 1;
          do{
              cout<<i<<"\n";
              i++;
          } while (i <= 10) ;
}
```

Output:

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

C++ Nested do-while Loop

In C++, if you use do-while loop inside another do-while loop, it is known as nested do-while loop. The nested do-while loop is executed fully for each outer do-while loop.

Let's see a simple example of nested do-while loop in C++.

```
#include <iostream>
using namespace std;
int main() {
     int i = 1;
         do{
              int j = 1;
              do{
                cout<<i<<"\n";
                  j++;
              } while (j <= 3) ;
              i++;
          } while (i <= 3) ;
}
```

Output:

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

C++ Infinitive do-while Loop

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

```
do{
//code to be executed
}while(true);
```

C++ Infinitive do-while Loop Example

```
#include <iostream>
using namespace std;
int main() {
      do{
              cout<<"Infinitive do-while Loop";
          } while(true);
}
```

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:

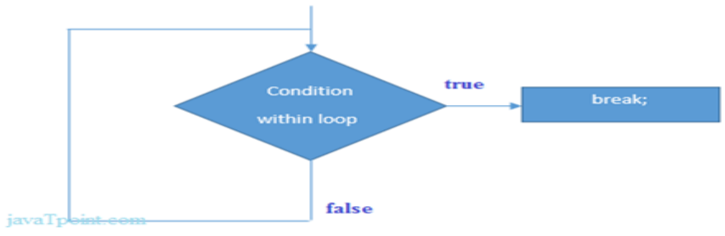


Figure: Flowchart of break statement

C++ Break Statement Example

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

```
#include <iostream>
using namespace std;
int main() {
      for (int i = 1; i <= 10; i++)
          {
              if (i == 5)
              {
                  break;
              }
            cout<<i<<"\n";
          }
}
```

Output:

```
1
```

```
2
3
4
```

C++ Break Statement with Inner Loop

The C++ break statement breaks inner loop only if you use break statement inside the inner loop.

Let's see the example code:

```
#include <iostream>
using namespace std;
int main()
{
   for(int i=1;i<=3;i++){
         for(int j=1;j<=3;j++){
            if(i==2&&j==2){
               break;
                  }
               cout<<i<<" "<<j<<"\n";
               }
         }
}
```

Output:

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

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=10;i++){
         if(i==5){
            continue;
         }
         cout<<i<<"\n";
       }
}
```

Output:

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

C++ Continue Statement with Inner Loop

C++ Continue Statement continues inner loop only if you use continue statement inside the inner loop.

```
#include <iostream>
using namespace std;
int main()
{
 for(int i=1;i<=3;i++){
            for(int j=1;j<=3;j++){
             if(i==2&&j==2){
                continue;
                        }
                cout<<i<<" "<<j<<"\n";
                    }
            }
}
```

Output:

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

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++.

```
#include <iostream>
using namespace std;
int main()
{
ineligible:
         cout<<"You are not eligible to vote!\n";
      cout<<"Enter your age:\n";
```

```
            int age;
            cin>>age;
            if (age < 18){
                    goto ineligible;
            }
            else
            {
                    cout<<"You are eligible to vote!";
            }
      }
```

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++.

```
      #include <iostream>
      using namespace std;
      int main()
      {
       int x = 11; // x is a variable
       cout<<x<<"\n";
      }
```

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++.

```
      #include <ostream>
      using namespace std;
      int main()
```

```
{
/* declare and
print variable in C++. */
 int x = 35;
 cout<<x<<"\n";
}
```

Output:

```
35
```

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.

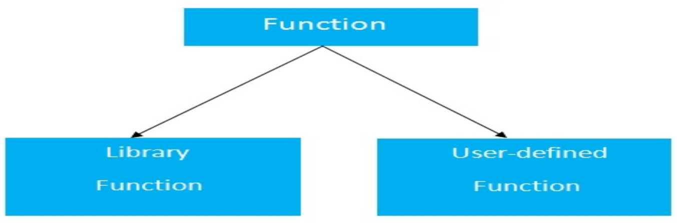
But if you use functions, you need to write the logic only once and you can reuse it several times.

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:

```
return_type function_name(data_type parameter...)
{
//code to be executed
}
```

C++ Function Example

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

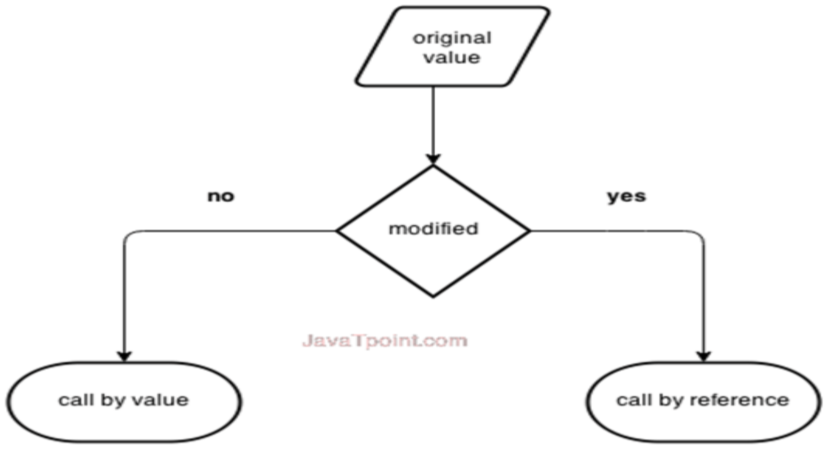
```
#include <iostream>
using namespace std;
void func() {
   static int i=0; //static variable
   int j=0; //local variable
   i++;
   j++;
   cout<<"i=" << i<<" and j=" <<j<<endl;
}
int main()
{
 func();
 func();
 func();
}
```

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++

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.



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:

```
#include <iostream>
using namespace std;
void change(int data);
```

```
int main()
{
int data = 3;
change(data);
cout << "Value of the data is: " << data<< endl;
return 0;
}
void change(int data)
{
data = 5;
}
```

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:

```
#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
 int swap;
 swap=*x;
 *x=*y;
 *y=swap;
}
int main()
{
 int x=500, y=100;
 swap(&x, &y);  // passing value to function
 cout<<"Value of x is: "<<x<<endl;
 cout<<"Value of y is: "<<y<<endl;
 return 0;
}
```

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.

```
recursionfunction(){
recursionfunction(); //calling self function
}
```

C++ Recursion Example

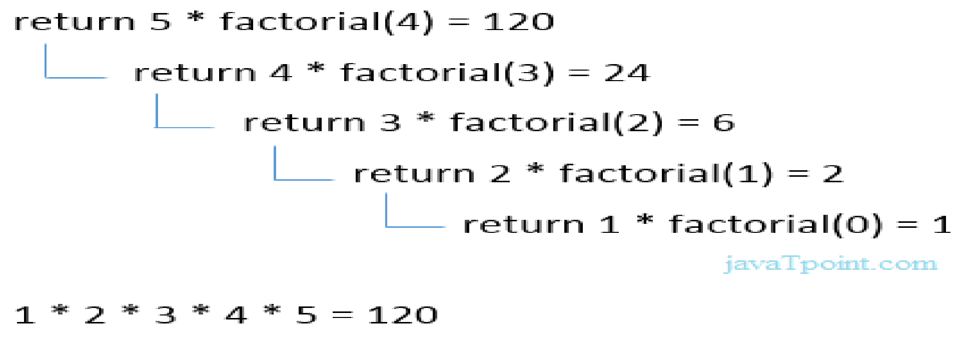
Let's see an example to print factorial number using recursion in C++ language.

```
#include<iostream>
using namespace std;
int main()
{
int factorial(int);
int fact,value;
cout<<"Enter any number: ";
cin>>value;
fact=factorial(value);
cout<<"Factorial of a number is: "<<fact<<endl;
return 0;
}
int factorial(int n)
{
if(n<0)
return(-1); /*Wrong value*/
if(n==0)
return(1);  /*Terminating condition*/
else
{
return(n*factorial(n-1));
}
}
```

Output:

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

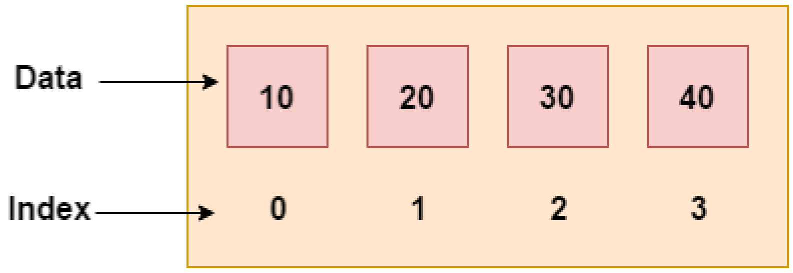
We can understand the above program of recursive method call by the figure given below:



```
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

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

Fig: Recursion

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.

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5]={10, 0, 20, 0, 30};  //creating and initializing array
        //traversing array
        for (int i = 0; i < 5; i++)
        {
            cout<<arr[i]<<"\n";
        }
}
```

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.

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
        //traversing array
       for (int i: arr)
        {
            cout<<i<<"\n";
        }
}
```

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.

```
#include <iostream>
using namespace std;
void printArray(int arr[5]);
int main()
{
        int arr1[5] = { 10, 20, 30, 40, 50 };
        int arr2[5] = { 5, 15, 25, 35, 45 };
        printArray(arr1); //passing array to function
        printArray(arr2);
}
void printArray(int arr[5])
{
  cout << "Printing array elements:"<< endl;
  for (int i = 0; i < 5; i++)
  {
          cout<<arr[i]<<"\n";
  }
}
```

Output:

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

C++ Passing Array to Function Example: Print minimum number

Let's see an example of C++ array which prints minimum number in an array using function.

```
    #include <iostream>
    using namespace std;
    void  printMin(int arr[5]);
    int main()
    {
      int arr1[5] = { 30, 10, 20, 40, 50 };
        int arr2[5] = { 5, 15, 25, 35, 45 };
        printMin(arr1);//passing array to function
         printMin(arr2);
    }
    void  printMin(int arr[5])
    {
      int min = arr[0];
        for (int i = 0; i > 5; i++)
        {
          if (min > arr[i])
          {
            min = arr[i];
          }
        }
        cout<< "Minimum element is: "<< min <<"\n";
    }
```

Output:

```
Minimum element is: 10
Minimum element is: 5
```

C++ Passing Array to Function Example: Print maximum number

Let's see an example of C++ array which prints maximum number in an array using function.

```
#include <iostream>
using namespace std;
void  printMax(int arr[5]);
int main()
{
        int arr1[5] = { 25, 10, 54, 15, 40 };
        int arr2[5] = { 12, 23, 44, 67, 54 };
        printMax(arr1); //Passing array to function
         printMax(arr2);
}
void  printMax(int arr[5])
{
    int max = arr[0];
        for (int i = 0; i < 5; i++)
        {
            if (max < arr[i])
            {
                max = arr[i];
            }
        }
        cout<< "Maximum element is: "<< max <<"\n";
}
```

Output:

```
Maximum element is: 54
Maximum element is: 67
```

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.

```
#include <iostream>
using namespace std;
int main()
{
  int test[3][3];  //declaration of 2D array
    test[0][0]=5;  //initialization
    test[0][1]=10;
    test[1][1]=15;
    test[1][2]=20;
    test[2][0]=30;
```

```
     test[2][2]=10;
     //traversal
     for(int i = 0; i < 3; ++i)
     {
        for(int j = 0; j < 3; ++j)
        {
           cout<< test[i][j]<<" ";
        }
        cout<<"\n"; //new line at each row
     }
     return 0;
  }
```

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.

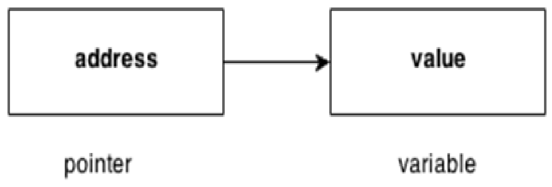
```
  #include <iostream>
  using namespace std;
  int main()
  {
    int test[3][3] =
     {
        {2, 5, 5},
        {4, 0, 3},
        {9, 1, 8}  };  //declaration and initialization
     //traversal
     for(int i = 0; i < 3; ++i)
     {
        for(int j = 0; j < 3; ++j)
        {
           cout<< test[i][j]<<" ";
        }
        cout<<"\n"; //new line at each row
     }
     return 0;
  }
```

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.

```
#include <iostream>

using namespace std;

int main()

{

int number=30;

int ∗   p;

p=&number;//stores the address of number variable

cout<<"Address of number variable is:"<<&number<<endl;

cout<<"Address of p variable is:"<<p<<endl;

cout<<"Value of p variable is:"<<*p<<endl;

   return 0;

}
```

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

```
#include <iostream>

using namespace std;

int main()

{

int a=20,b=10,∗p1=&a,∗p2=&b;

cout<<"Before swap: ∗p1="<<∗p1<<" ∗p2="<<∗p2<<endl;

∗p1=∗p1+∗p2;

∗p2=∗p1-∗p2;

∗p1=∗p1-∗p2;
cout<<"After swap: ∗p1="<<∗p1<<" ∗p2="<<∗p2<<endl;

  return 0;

}
```

Output

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

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.

```
#include <iostream>

using namespace std;

class Employee

 {

  public:

      Employee()

      {

         cout<<"Default Constructor Invoked"<<endl;

      }

};

int main(void)
```

```
    {

        Employee e1; //creating an object of Employee

        Employee e2;

        return 0;

    }
```

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.

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
        Employee()
        {
           cout<<"Constructor Invoked"<<endl;
        }
        ~Employee()
        {
           cout<<"Destructor Invoked"<<endl;
        }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    return 0;
}
```

Output:

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

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.

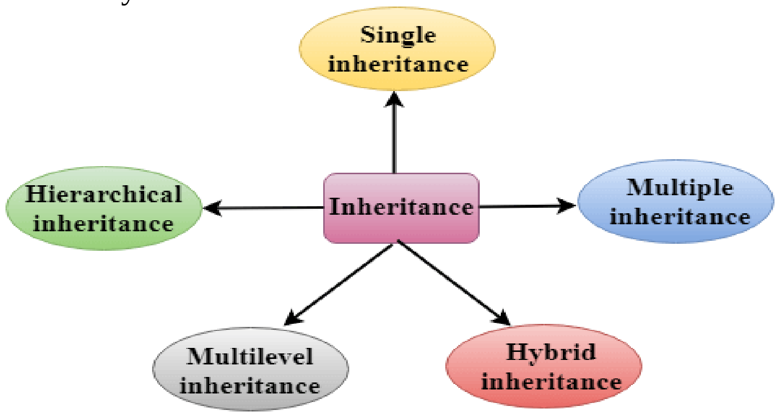
Advantage of C++ Inheritance

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:

```
class derived_class_name :: visibility-mode base_class_name
{
   // body of the derived class.
}
```

Where, derived_class_name: It 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: It is the name of the base class.

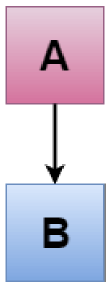
- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.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

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
     Programmer p1;
     cout<<"Salary: "<<p1.salary<<endl;
     cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
}
```

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.

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking...";
      }
   };
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```
Output:

```
Eating...
Barking...
```
Let's see a simple example.

```
    #include <iostream>
    using namespace std;
    class A
    {
        int a = 4;
        int b = 5;
        public:
        int mul()
        {
            int c = a*b;
            return c;
        }
    };

    class B : private A
    {
        public:
        void display()
        {
            int result = mul();
            std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
        }
    };
    int main()
    {
       B b;
       b.display();

        return 0;
    }
```
Output:

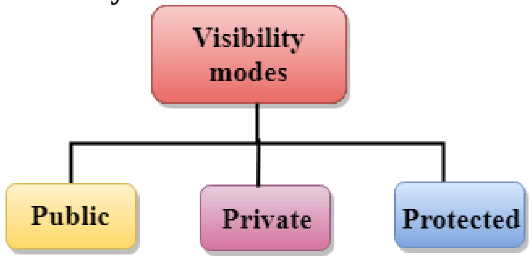
```
Multiplication of a and b is : 20
```
In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

Visibility modes can be classified into three categories:



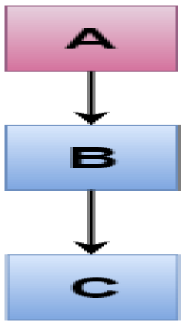
- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as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

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
```

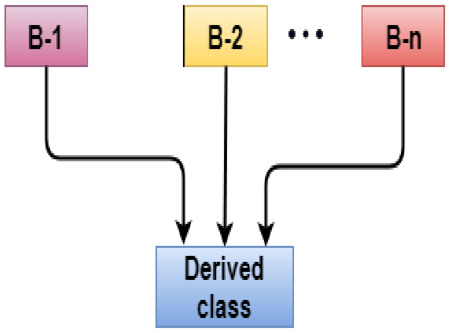
```
   void bark(){
  cout<<"Barking..."<<endl;
   }
  };
  class BabyDog: public Dog
  {
      public:
   void weep() {
  cout<<"Weeping...";
   }
  };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

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

Let's see a simple example of multiple inheritance.

```
#include <iostream>
using namespace std;
class A
{
    protected:
     int a;
    public:
    void get_a(int n)
    {
        a = n;
    }
};

class B
{
    protected:
    int b;
    public:
    void get_b(int n)
    {
        b = n;
    }
};
class C : public A,public B
{
   public:
    void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

    return 0;
}
```

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.

Ambiquity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class A
{
    public:
    void display()
    {
        std::cout << "Class A" << std::endl;
    }
};
class B
{
   public:
   void display()
   {
      std::cout << "Class B" << std::endl;
   }
};
class C : public A, public B
{
   void view()
   {
      display();
   }
};
int main()
{
   C c;
   c.display();
   return 0;
}
```

Output:

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

- The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
{
   void view()
   {
      A :: display();        // Calling the display() function of class A.
      B :: display();        // Calling the display() function of class B.

   }
};
```

An ambiguity can also occur in single inheritance.
Consider the following situation:

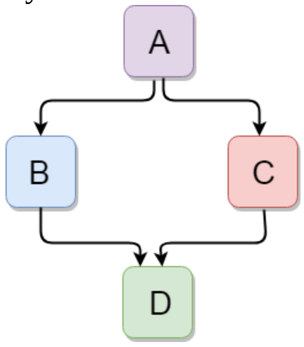
```
class A
{
  public:
void display()
{
  cout<<?Class A?;
}
} ;
class B
{
 public:
 void display()
{
 cout<<?Class B?;
}
} ;
```

In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

```
int main()
{
    B b;
   b.display();               // Calling the display() function of B class.
   b.B :: display();       // Calling the display() function defined in B class.
}
```

C++ Hybrid Inheritance

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



Let's see a simple example:

```
#include <iostream>
using namespace std;
class A
{
    protected:
    int a;
    public:
    void get_a()
    {
       std::cout << "Enter the value of 'a' : " << std::endl;
       cin>>a;
    }
};

class B : public A
{
    protected:
    int b;
    public:
    void get_b()
    {
        std::cout << "Enter the value of 'b' : " << std::endl;
       cin>>b;
    }
};
```

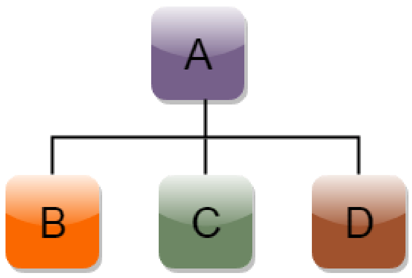
```
    class C
    {
        protected:
        int c;
        public:
        void get_c()
        {
            std::cout << "Enter the value of c is : " << std::endl;
            cin>>c;
        }
    };

    class D : public B, public C
    {
        protected:
        int d;
        public:
        void mul()
        {
             get_a();
             get_b();
             get_c();
             std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
        }
    };
    int main()
    {
        D d;
        d.mul();
        return 0;
    }
```

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:

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Shape                  // Declaration of base class.
{
   public:
   int a;
   int b;
   void get_data(int n,int m)
   {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape  // inheriting Shape class
{
   public:
   int rect_area()
   {
      int result = a*b;
      return result;
   }
};
class Triangle : public Shape    // inheriting Shape class
{
   public:
   int triangle_area()
   {
      float result = 0.5*a*b;
      return result;
   }
};
int main()
{
   Rectangle r;
   Triangle t;
   int length,breadth,base,height;
   std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
   cin>>length>>breadth;
   r.get_data(length,breadth);
   int m = r.rect_area();
   std::cout << "Area of the rectangle is : " <<m<< std::endl;
   std::cout << "Enter the base and height of the triangle: " << std::endl;
   cin>>base>>height;
   t.get_data(base,height);
   float n = t.triangle_area();
   std::cout <<"Area of the triangle is : "  << n<<std::endl;
   return 0;
}
```

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
```

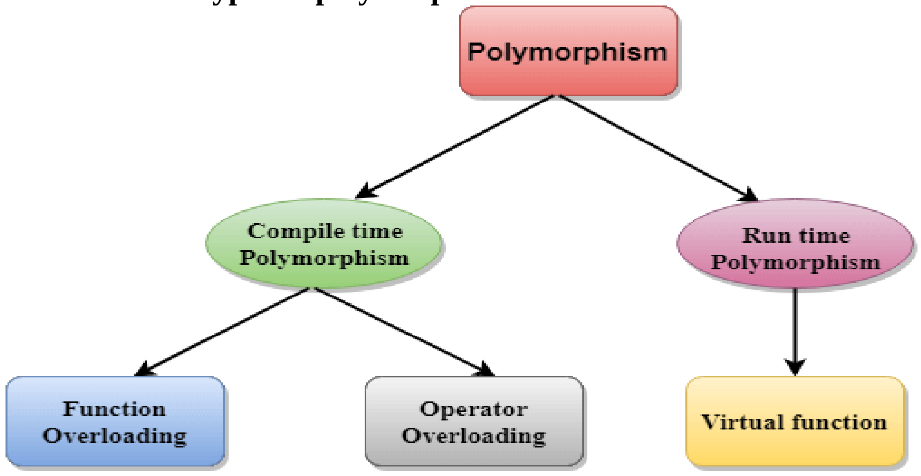
C++ Polymorphism

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.

```
  class A                          //  base class declaration.
 {
     int a;
     public:
     void display()
     {
          cout<< "Class A ";
      }
 };
class B : public A                 //  derived class declaration.
{
    int b;
    public:
   void display()
  {
       cout<<"Class B";
  }
};
```

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++ Overloading (Function and Operator)

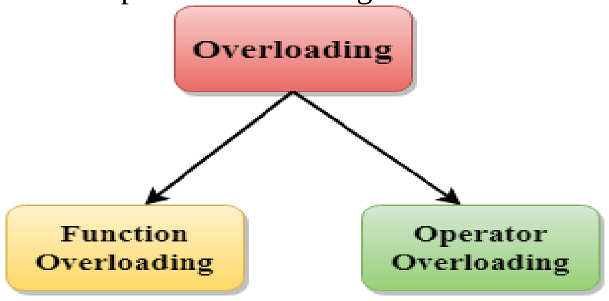
If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- 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.

```
#include <iostream>
using namespace std;
class Cal {
   public:
static int add(int a,int b){
      return a + b;
   }
static int add(int a, int b, int c)
   {
      return a + b + c;
   }
};
int main(void) {
   Cal C;                              //    class object declaration.
   cout<<C.add(10, 20)<<endl;
   cout<<C.add(12, 20, 23);
  return 0;
}
```

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.
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);


int mul(int a,int b)
{
   return a*b;
}
float mul(double x, int y)
{
   return x*y;
}
int main()
{
   int r1 = mul(6,7);
   float r2 = mul(0.2,3);
   std::cout << "r1 is : " <<r1<< std::endl;
   std::cout <<"r2 is : "  <<r2<< std::endl;
   return 0;
}
```
Output:

```
r1 is : 42
r2 is : 0.6
```

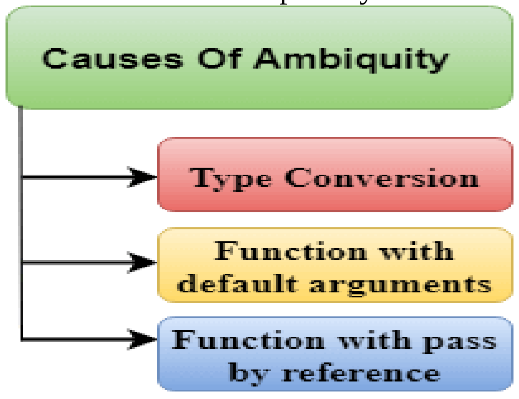
Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.


Causes Of Ambiquity
Type Conversion
Function with default arguments
Function with pass by reference

- Type Conversion:

Let's see a simple example.

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(float j)
{
   std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{
   fun(12);
   fun(1.2);
   return 0;
}
```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

Function with Default Arguments

Let's see a simple example.

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
   std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
   fun(12);

   return 0;
}
```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

Function with pass by reference

Let's see a simple example.

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
}
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

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

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

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 atleast 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 ++.
#include <iostream>
using namespace std;
class Test
{
   private:
      int num;
   public:
       Test(): num(8){}
       void operator ++()         {
          num = num+2;
       }
       void Print() {
           cout<<"The Count is: "<<num;
       }
};
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output:

```
The Count is: 10
```

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

```
// program to overload the binary operators.
#include <iostream>
using namespace std;
class A
{

    int x;
      public:
      A(){}
    A(int i)
    {
      x=i;
    }
    void operator+(A);
    void display();
};

void A :: operator+(A a)
{

    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;

}
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

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.

```
#include <iostream>
using namespace std;
class Animal {
    public:
void eat(){
cout<<"Eating...";
    }
};
class Dog: public Animal
{
 public:
 void eat()
    {
      cout<<"Eating bread...";
    }
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
}
```

```
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.

```
#include <iostream>
using namespace std;
class A
{
  int x=5;
   public:
   void display()
   {
       std::cout << "Value of x is : " << x<<std::endl;
   }
};
class B: public A
{
   int y = 10;
   public:
   void display()
   {
       std::cout << "Value of y is : " <<y<< std::endl;
   }
};
```

```
int main()
{
    A *a;
    B b;
    a = &b;
   a->display();
    return 0;
}
```

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 invoked the derived class in a program.

```
#include <iostream>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
 a = &b;
 a->display();   //Late Binding occurs
}
```

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:

```
virtual void display() = 0;
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Base
{
   public:
   virtual void show() = 0;
};
class Derived : public Base
{
   public:
   void show()
   {
      std::cout << "Derived class is derived from the base class." << std::endl;
   }
};
int main()
{
   Base *bptr;
   //Base b;
   Derived d;
   bptr = &d;
   bptr->show();
   return 0;
}
```

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.

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 seperation 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 seperates the implementation details from the external interface.

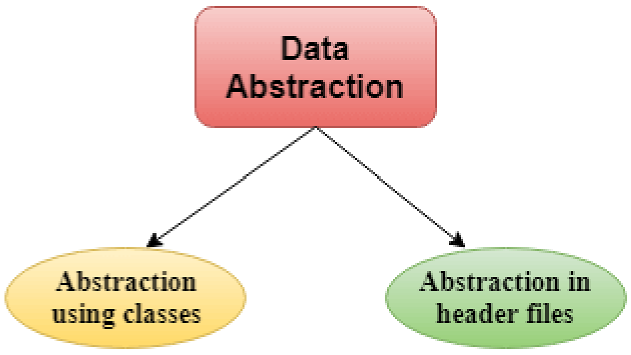
C++ provides a great level of abstraction. For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.

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: An 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 hides 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.

```
#include <iostream>
#include<math.h>
using namespace std;
int main()
{
 int n = 4;
   int power = 3;
   int result = pow(n,power);         // pow(n,power) is the  power function
   std::cout << "Cube of n is : " <<result<< std::endl;
   return 0;
}
```

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.

```
#include <iostream>
using namespace std;
 class Sum
{
private: int x, y, z; // private variables
public:
void add()
{
cout<<"Enter two numbers: ";
cin>>x>>y;
z= x+y;
cout<<"Sum of two number is: "<<z<<endl;
}
};
int main()
{
Sum sm;
sm.add();
return 0;
}
```

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.