Introduction to Class and Object

The classes are the most important feature of C++ that leads to Object Oriented programming. Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class which is also called object . The variables inside class definition are called as data members and the functions are called member functions.

Object

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. In other words, Any entity that has state and behavior is known as an object. An object is an instance of a class. Objects take space in memory.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

A class can also be defined as a blueprint from which you can create an individual object. It is a collection of objects of similar type. Unlike object, class doesn’t consume any space in memory. A class can have any number of objects associated with it.

An example of class can be student, smart phone, animal , vehicle etc.

Example: Basic structure of Class and Object

```
class Abc
{
   int x;
   void display()
   {
      // some statement
   }
};

int main()
{
   Abc obj;   // Object of class Abc created
}
```

Access Modifiers

Access modifiers are used to implement an important feature of Object Oriented Programming known as **Data Hiding**. Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

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

- Public
- Private
- Protected

Public:

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

Example: C++ program to demonstrate the use of public access specifier

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

class Circle
{
   public:
      double radius;

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

};

int main()
{
   Circle obj;
   obj.radius = 5.5;
```

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

Output:

```
Radius is:5.5
Area is:94.985
```

Private:

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

Example: C++ program to demonstrate the use of private access specifier

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

class Circle
{

   private:
      double radius;
   public:
      void compute_area(double r)
      {
         radius = r;
```

```
            double area = 3.14*radius*radius;
            cout << "Radius is:" << radius << endl;
            cout << "Area is: " << area;
        }

};


 int main()
 {

    Circle obj;
    obj.compute_area(1.5);
    return 0;
 }
```

Output:

```
Radius is:1.5
Area is: 7.065
```

Protected:

Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

Example: C++ program to demonstrate the use of protected access specifier

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

// Base class

class Parent
{

    protected:
    int id_protected;

};

// sub class or derived class
class Child : public Parent
{



    public:
    void setId(int id)
    {
        id_protected = id;
    }

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

```
 int main() {

    Child obj1;
    obj1.setId(81);
    obj1.displayId();
    return 0;
 }
```

<u>Output</u>:

```
id_protected is:81
```

Accessing Class Members

Accessing a data member depends solely on the access control of that data member. If its public, then the data member can be easily accessed using the direct member access (.) operator with the object of that class.

If, the data member is defined as private or protected, then we cannot access the data variables directly. Then we will have to create special public member functions to access, use or initialize the private and protected data members. These member functions are also called **Accessors** and **Mutator** methods or **getter** and **setter** functions.

Accessing Public Data Members

Following is an example to show you how to initialize and use the public data members using the dot (.) operator and the respective object of class.

```
class Student
{
   public:
   int rollno;
   string name;
};

int main()
{
   Student A;
   Student B;

   // setting values for A object
   A.rollno=1;
   A.name="Adam";

   // setting values for B object
   B.rollno=2;
   B.name="Bella";

   cout <<"Name and Roll no of A is: "<< A.name << "-" << A.rollno;
   cout <<"Name and Roll no of B is: "<< B.name << "-" << B.rollno;
}
```

Output:

```
Name and Roll no of A is: Adam-1
Name and Roll no of B is: Bella-2
```

Accessing Private Data Members

To access, use and initialize the private data member you need to create getter and setter functions, to get and set the value of the data member.

The setter function will set the value passed as argument to the private data member, and the getter function will return the value of the private data member to be used. Both getter and setter function must be defined public.

Example :

```
class Student
{
   private:
            int rollno;

   public:
   // public function to get value of rollno - getter
   int getRollno()
   {
      return rollno;
   }

   // public function to set value for rollno - setter
   void setRollno(int i)
   {
      rollno=i;
   }
};

int main()
{
   Student A;
   A.rollono=1;  //Compile time error
   cout<< A.rollno; //Compile time error
```

```
    A.setRollno(1);  //Rollno initialized to 1
    cout<< A.getRollno(); //Output will be 1
    return 0;
}
```

Accessing Protected Data Members

Protected data members, can be accessed directly using dot (.) operator inside the **subclass** of the current class, for non-subclass we will have to follow the steps same as to access private data member.

Defining Member Functions

A member function of a class is a function that has its definition within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member functions can be defined in two places:

- Outside the class definition
- Inside the class definition

Outside the class definition:

The member functions of a class can be defied outside the class definitions. It is only declared inside the class but defined outside the class. The general form of member function definition outside the class definition is:

```
 return_type class_name:: function_name (argument list)
{
/        /function body
}
```

Where symbol **::** is a scope resolution operator.

Example: A program to add two numbers defining a member getdata () and display () inside a class named sum and displaying the result.

```
#include<iostream.h>
#include<conio.h>
class sum
{
        Int A, B, total;
        public:
             void getdata ();
             void display ();
};
void sum:: getdata ()
{
        cout<<" \n Enter the value of A and B:"<<endl;
        cin>>A>>B;
}
void sum:: display ()
{
        total =A+B;
        cout<<"\n The sum of A and B="<<total;
}
int main ()
{
        Sum a;
        a.getdata ();
        a.display ();
        return 0;
}
```

Output:

```
Enter value of A and B:
5
6
The sum of A and B=11
```

Inside the class definition

The member functions of a class can be declared and defined inside the class definition. When a function is defined inside the class, it is treated as an inline function.

Example:

```
#include<iostream.h>
#include<conio.h>
class sum
{
private:
        Int A, B, total;
public:
        void getdata ()
        {
                cout<,"\n enter the value of A and B";
                cin>>A>>B;
        }
        void display ()
        {
                total = A+B;
                cout<<"\n the sum of A and B="<<total;
        }
};
int main ()
{
        sum a;
        a.getdata ();
        a.display ();
        return 0;
}
```

<u>Output:</u>

same as above

Constructor and Destructor

A class constructor is a special member function of a class that is executed whenever we create new objects of that class.
A constructor will have the exact same name as the class and it does not have any return type, not even void. Constructors can be very useful for setting initial values for certain member variables.
Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

A constructor is different from normal functions in following ways:

- Constructor has the same name as the class itself
- Constructors don’t have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Characteristics of Constructor

- Constructor should be defined or declared in public section.
- They do not have return types.
- They cannot be inherited but a derived class can call the base class constructor.
- Like functions, they can have default arguments.

- Constructors cannot be virtual.
- We cannot refer to their addresses.
- An object with a constructor cannot be used as a member of a union.

Types of Constructors in C++

Constructors are of three types:

1. Default Constructor
2. Parameterized Constructor
3. Copy COnstructor

Default Constructors

Default constructor is a constructor which doesn't take any argument. It has no parameters.

Syntax:

```
class_name()
{
   // constructor Definition
}
```

For example:

```
#include<iostream>
using namespace std;
class Cube
{
    public:
        int side;
    Cube()
    {
        side = 10;
    }
};

int main()
{
    Cube c;
    cout << c.side;
    return 0;
}
```

Output:

```
10
```

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
    public:
    int side;
};

int main()
{
    Cube c;
    cout << c.side;
}
```

Output:

```
0 or any random value
```

In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 or any random integer value in this case.

Parameterized Constructors

These are the constructors with parameters. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

For example:

```
class Cube
{
   public:
      int side;
   Cube(int x)
   {
      side=x;
   }
};

int main()
{
   Cube c1(10);
   Cube c2(20);
   Cube c3(30);
   cout << c1.side;
   cout << c2.side;
   cout << c3.side;
}
```

Output

```
10
20
30
```

By using parameterized constructor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

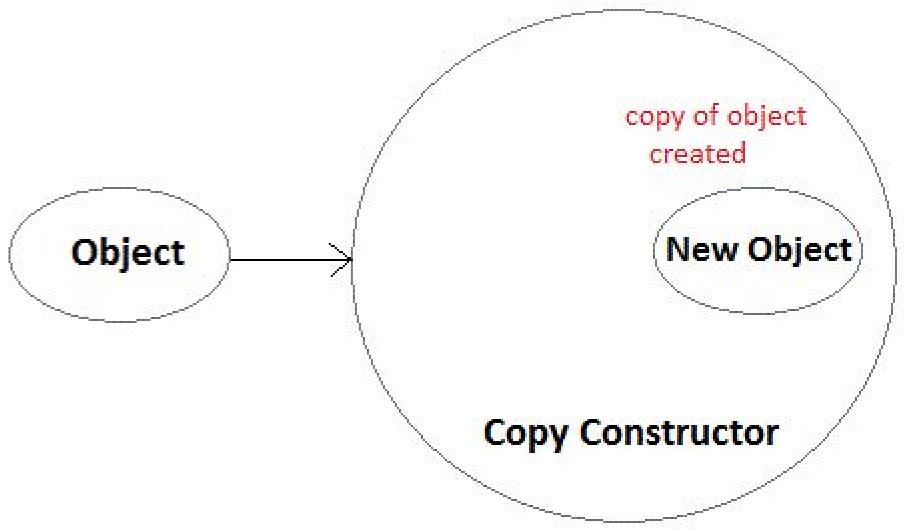
Copy Constructor:

Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form **X (X&)**, where X is the class name. The compiler provides a default Copy Constructor to all the classes.

Syntax of Copy Constructor

```
Classname(const classname & objectname)
{
  . . . .
}
```

As it is used to create an object, hence it is called a constructor. And, it creates a new object, which is exact copy of the existing copy, hence it is called **copy constructor**.



Below is a sample program on Copy Constructor:

```
#include<iostream>
using namespace std;
class copy
{
   private:
   int x, y;   //data members

   public:
   copy(int x1, int y1)
   {
      x = x1;
      y = y1;
   }
```

```
    /* Copy constructor */
    copy(const copy &sam)
    {
        x = sam.x;
        y = sam.y;
    }

    void display()
    {
        cout<<x<<" "<<y<<endl;
    }
};

int main()
{
    copy obj1(10, 15);     // Normal constructor
    copy obj2 = obj1;      // Copy constructor
    cout<<"Normal constructor : ";
    obj1.display();
    cout<<"Copy constructor : ";
    obj2.display();
    return 0;
}
```

Output:

```
Normal constructor : 10 15
Copy constructor : 10 15
```

Destructors in C++

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** ~ sign as prefix to it.

```
class A
{
   public:
   // defining destructor for class
   ~A()
   {
      // statement
   }
};
```

Destructors will never have any arguments.

Characteristics of Destructor

- A destructor has the same name as the constructor but is preceded by a tilde (~).
- Like constructors, destructors do not have a return value.
- Destructors take no arguments.
- Destructors can be virtual.
- Destructor cannot be inherited.

Example to see how Constructor and Destructor are called

Below we have a simple class A with a constructor and destructor. We will create object of the class and see when a constructor is called and when a destructor gets called.

```
#include<iostream>
using namespace std;
class A
{
   // constructor
   A()
   {
      cout << "Constructor called";
   }
```

```
    // destructor
    ~A()
    {
        cout << "Destructor called";
    }
};

int main()
{
    A obj1;   // Constructor Called
    int x = 1
    if(x)
    {
        A obj2;  // Constructor Called
    }   // Destructor Called for obj2
} //  Destructor called for obj1
```

<u>Output</u>

```
Constructor called
Constructor called
Destructor called
Destructor called
```

When an object is created the constructor of that class is called. The object reference is destroyed when its scope ends, which is generally after the closing curly bracket } for the code block in which it is created.

The object obj2 is destroyed when the if block ends because it was created inside the if block. And the object obj1 is destroyed when the main() function ends

Constructor Overloading in C++

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other parameters.

You can have any number of Constructors in a class that differ in parameter list.

```
#include<iostream>
using namespace std;
class Student
{
   public:
   int rollno;
   string name;
   // first constructor
   Student(int x)
   {
         rollno = x;
      name = "None";
         }
   // second constructor
   Student(int x, string str)
   {
          rollno = x;
      name = str;
         }

         void display()
         {
                  cout<<"Roll no:"<<rollno<<"\t"<<"Name:"<<name<<endl;
         }



};

int main()
{
   // student A initialized with roll no 10 and name None
   Student A(10);

   // student B initialized with roll no 11 and name John
   Student B(11, "John");
   A.display();
   B.display();
}
```

Output

```
Roll no:10     Name:None

Roll no:11     Name:John
```

In the above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write Student S; in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and the compiler will not provide its default constructor because we have defined other parameterized constructors.

This Pointer

In C++, this pointer is used to represent the address of an object inside a member function. For example, consider an object *obj* calling one of its member functions say *method()* as *obj.method()*. Then, this pointer will hold the address of object *obj* inside the member function *method()*. The this pointer acts as an implicit argument to all the member functions.

Example 1: C++ program using this pointer to distinguish local members from parameters.

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

```
class sample
{
   int a,b;
   public:
      void input(int a,int b)
      {
         this->a=a+b;
         this->b=a-b;
      }
      void output()
      {
         cout<<"a = "<<a<<endl<<"b = "<<b;
      }
};

int main()
{
   sample x;
   x.input(5,8);
   x.output();
   getch();
   return 0;
}
```

Example 2: C++ program to display the record of student with highest percentage.

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

class student
{
    char name[100];
```

```
int age,roll;
float percent;
public:
   void getdata()
   {
      cout<<"Enter data"<<endl;
      cout<<"Name:";
      cin>>name;
      cout<<"Age:";
      cin>>age;
      cout<<"Roll:";
      cin>>roll;
      cout<<"Percent:";
      cin>>percent;
      cout<<endl;
   }
   student  max(student s1,student s2)
   {
      if(percent>s1.percent && percent>s2.percent)
         return *this;
      else if(s1.percent>percent && s1.percent>s2.percent)
         return s1;
      else if(s2.percent>percent && s2.percent>s1.percent)
         return s2;
   }
   void display()
   {
```

```
            cout<<"Name:"<<name<<endl;
            cout<<"Age:"<<age<<endl;
            cout<<"Roll:"<<roll<<endl;
            cout<<"Percent:"<<percent;
        }
};

int main()
{
    student s,s1,s2,s3;
    s1.getdata();
    s2.getdata();
    s3.getdata();
    s=s3.max(s1,s2);
    cout<<"Student with highest percentage"<<endl;
    s.display();
    return 0;
}
```

Output:

```
Enter data
Name:Paul
Age:24
Roll:11
Percent:79
```

```
Enter data
Name:Reem
Age:21
Roll:9
Percent:87

Enter data
Name:Philip
Age:23
Roll:5
Percent:81

Student with highest percentage
Name:Reem
Age:21
Roll:9
Percent:87
```

Application of this pointer

- **Return Object**

  One of the important applications of using *this* pointer is to return the object it points. For example, the statement

  ```
  return *this;
  ```

  inside a member function will return the object that calls the function.

- **Method Chaining**

  After returning the object from a function, a very useful application would be to chain the methods for ease and a cleaner code.

  For example,

  ```
  positionObj->setX(15)->setY(15)->setZ(15);
  ```

  Here, the methods *setX*, *setY*, *setZ* are chained to the object, *positionObj*. This is possible because each method return *this pointer.

  This is equivalent to

  ```
  positionObj->setX(15);
  positionObj->setY(15);
  positionObj->setZ(15);
  ```

- **Distinguish Data Members**

  Another application of *this* pointer is distinguishing data members from local variables of member functions if they have the same name. For example **: see example 1**

Friend Function and Friend Class

Friend Class

A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of another class.

Example:

```
#include <iostream>
using namespace std;
class A {
      private:
        int a;
      public:
        A()
        {
              a=10;
       }
        friend class B;    // Friend Class
};

class B {
      private:
        int b;
      public:
        void showA(A x) {
           // Since B is friend of A, it can access
           // private members of A
```

```
            cout << "Value of A from class A =" << x.a;
        }
};

int main() {
    A a;
    B b;
    b.showA(a);
    return 0;
}
```

Output:

```
Value of A from class A =10
```

Friend Function

The function which are declared with the keyword “friend” are known as friend function. The friend function doesn’t belong to particular class.Function can be defined elsewhere in the program like a normal program. The function definition doesn’t use either the keyword friend or scope resolution (::) operation because it’s not the function of any particular class.

Characteristics

1. It isn’t in the scope of the class.
2. It cannot be called by using object or class name.
3. It can be invoked like a normal function in C++.
4. It can be declared either in the public or private part of the class.

5. Usually, it has object as an argument.

Example:

A program to determine whether the family is happy or not based on father salary and son fee.

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

class son;

class father
{
        int salary;

        public:

                void getSalary()
                {
                        cout<<"Enter salary of father:"<<endl;
                        cin>>salary;
                }

                friend void comparison(father,son);
};

class son
{
        int fee;

        public:
```

```
		void getFee()
		{
			cout<<"Enter fee of son:"<<endl;
			cin>>fee;
		}

		friend void comparison(father,son);

};

void comparison(father f, son s)
{
	if(f.salary>s.fee)
		cout<<"family is happy!";
	else
		cout<<"Family is unhappy!";
}

int main()
{
	father f;
	son s;
	f.getSalary();
	s.getFee();
	comparison(f,s);
	return 0;
}
```

Output

```
Enter salary of father:
12321
Enter fee of son:
```

```
23123
Family is unhappy!
```

Example 2: Addition of members of two different classes using friend Function

```
#include <iostream>
using namespace std;
// forward declaration
class B;
class A {
   private:
     int numA;
   public:
     A()
          {
            numA=12;
          }
     // friend function declaration
     friend int add(A, B);
};
class B {
   private:
      int numB;
   public:
      B()
           {
             numB=12;
           }

      friend int add(A , B);
};
```

```
int add(A objectA, B objectB)
{
  return (objectA.numA + objectB.numB);
}
int main()
{
   A objectA;
   B objectB;
   cout<<"Sum: "<< add(objectA, objectB);
   return 0;
}
```

Output

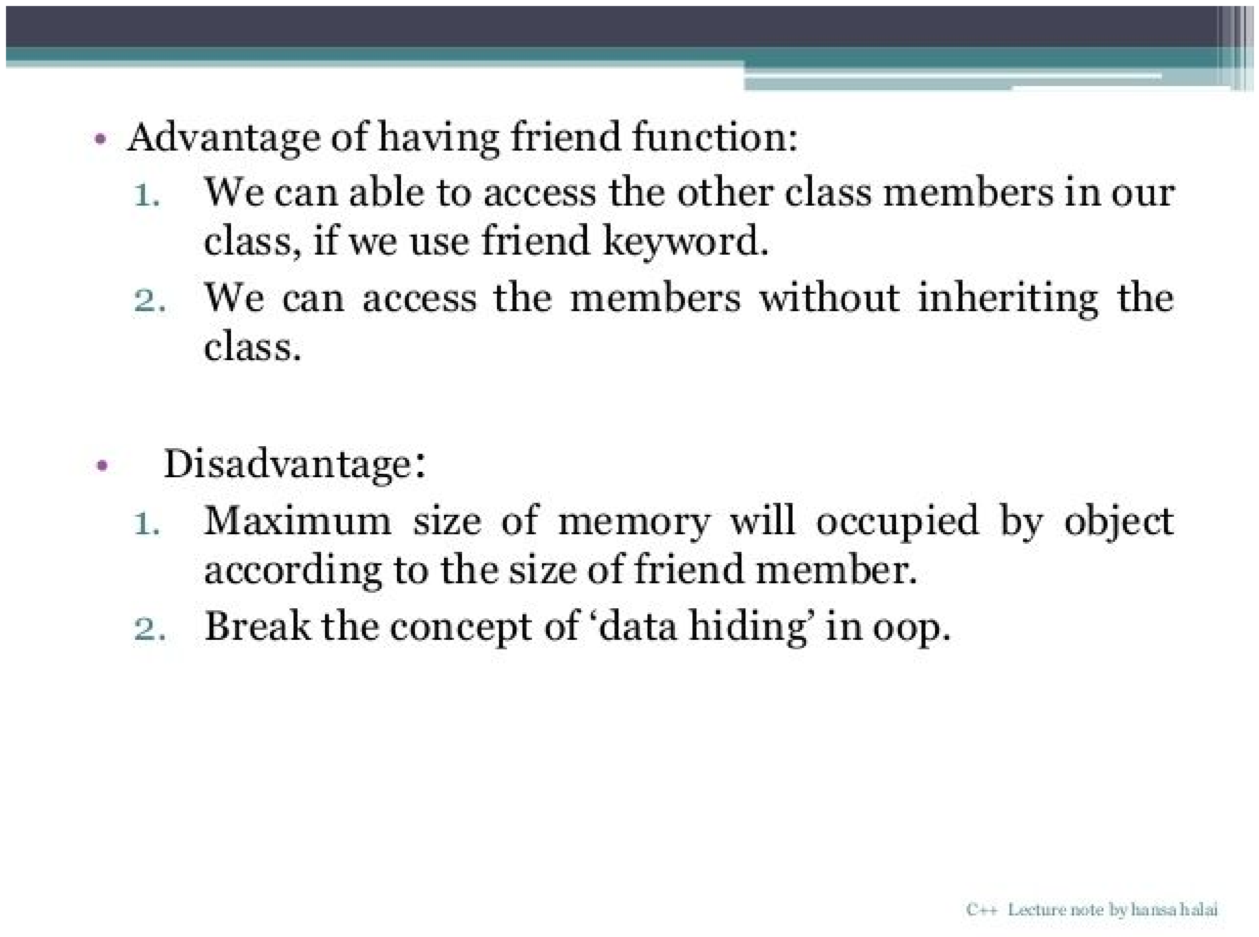
```
Sum: 24
```

Advantages of friend function

- It allows us to access other class member in our class.
- We can access the member without inheriting.

Disadvantages of friend function

- Maximum memory will be occupied by an object.
- It violates the concept of data hiding in OOP.



- Advantage of having friend function:
  1. We can able to access the other class members in our class, if we use friend keyword.
  2. We can access the members without inheriting the class.

- Disadvantage:
  1. Maximum size of memory will occupied by object according to the size of friend member.
  2. Break the concept of 'data hiding' in oop.

C++ Lecture note by hansa halai

Static Data Members

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member. 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.

Static data member has certain characteristics which are as follows;

- It is initialized to zero when the first object of its class is created.
- Static data members are always defined outside the class.
- Only one copy of the member is created for the entire class and it is shared by all objects of that class.
- It is visible only within the class but its lifetime is in the entire program.

Syntax for Declaration:

```
static data_type variable_name;
```

Syntax for Definition:

```
data_type class_name::variable_name=value;
```

Note: Static data members are always defined outside the class

Example:

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

class statTest
{

   public:

         static int count;

         void calculate(int l, int b){
                  cout<<"Area of rectangle:"<<l*b<<endl;
```

```
                count++;
        }


};


int statTest::count=0;

int main(){
   statTest s1,s2;
   s1.calculate(5,8);
   s2.calculate(9,7);
   cout<<"Total Count:"<<statTest::count;
   return 0;
}
```

Output:

```
Area of rectangle:40
Area of rectangle:63
Total Count:2
```

```
};
```

Static Member Function

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 **::**.

Example:

```
#include <iostream>
using namespace std;
class statTest
{
   private:
          static int count;

   public:

          void calculate(int l, int b){
                 cout<<"Area of rectangle:"<<l*b<<endl;

                 count++;
          }

          static int displayCount()
          {
                 return count;
          }
};
```

```
int statTest::count;

int main(){
   statTest s1,s2;
   s1.calculate(5,8);
   s2.calculate(9,7);
   cout<<"Total Count:"<<statTest::displayCount();
   return 0;
}
```

Output:

same as above