Chapter 1
Thinking Object Oriented

Since the invention of computer, many programming approaches have been tried. These techniques include modular programming, top-down programming, bottom-up programming and structured programming. The primary motivation in each has been the concern to handle the increasing complexity of programs that are reliable and maintainable.

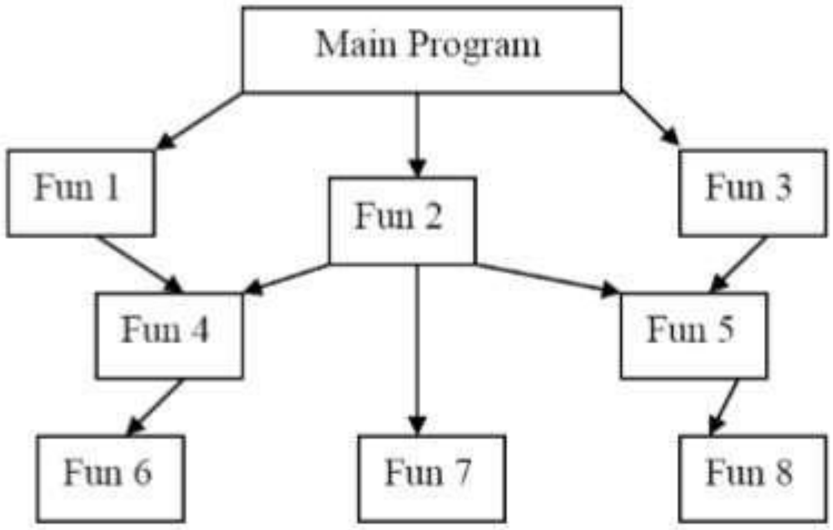
Procedural Oriented Language

A procedural language is a type of computer programming language that specifies a series of well-structured steps and procedures within its programming context to compose a program. It contains a systematic order of statements, functions and commands to complete a computational task or program. The procedural language divides a program within variables, functions, statements and conditional operators.

In procedural approach,

- A program is a list of instruction.
- When program in PL become larger, they are divided into functions (subroutines, sub-programs, procedures).
- Functions are grouped into modules.

A typical program structure of Procedural programming is shown in figure below



Characteristics of POP

- Emphasis is on doing thing (algorithm).
- Large programs are divided into smaller programs known as functions.
- Data move openly around the system from function to function.
- Employee **top-down approach** in program design.

Disadvantages of POP

- It emphasis on doing things.
- Since every function has complete access to the global variables, the new programmer can corrupt the data accidentally by creating function. Similarly, if new data is to be added, all the function needed to be modified to access the data.
- It is difficult to create new data types with procedural languages.

Difference between OOP and POP

OOP	POP
OOP takes a bottom-up approach in designing a program.	POP follows a top-down approach.
Program is divided into objects depending on the problem.	Program is divided into small chunks based on the functions.
Each object controls its own data.	Each function contains different data.
Focuses on security of the data irrespective of the algorithm.	Follows a systematic approach to solve the problem.
The main priority is data rather than functions in a program.	Functions are more important than data in a program.
The functions of the objects are linked via message passing.	Different parts of a program are interconnected via parameter passing.
Data hiding is possible in OOP.	No easy way for data hiding.
Inheritance is allowed in OOP.	No such concept of inheritance in POP.
Example: C++ , JAVA	Example: C,FORTRAN

Object-oriented Programming (OOP)

OOP is an approach to programming paradigm that attempts to eliminate some of the drawbacks of conventional programming methods with several powerful new concepts.

The fundamental idea behind object-oriented programming is to combine or encapsulate both data (or instance variables) and functions (or methods) that operate on that data into a single unit. This unit is called an object. The data is hidden, so it is safe from accidental alteration. An object's functions typically provide the only way to access its data. In order to access the data in an object, we should know exactly what functions interact with it. No other functions can access the data. Hence OOP focuses on data portion rather than the process of solving the problem.

The general structure of OOP is shown in the figure below.

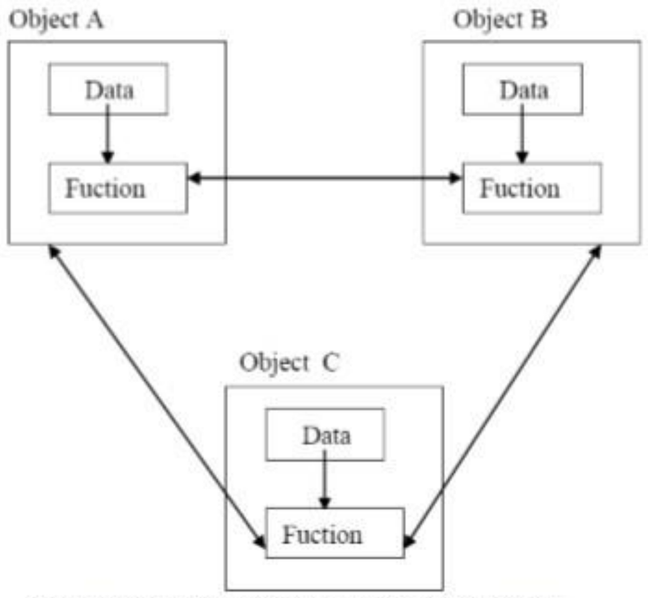


Fig: Organization of data and function in OOP

Characteristics of OOP

- Emphasis is on data rather than procedures.
- Programs are divided into objects.
- Data structures are designed such that they characterize the objects.
- Functions & data are tied together in the data structures so that data abstraction is introduced in addition to procedural abstraction.
- Data is hidden & can't be accessed by external functions.
- Object can communicate with each other through function.
- New data & functions can be easily added.
- Follows Bottom up approach.

Benefits of OOP

- Making the use of inheritance, redundant code is eliminated and the existing class is extended.
- Through data hiding, programmer can build secure program.
- It is possible to have multiple instances of an object to co-exist without any interference.

Basic concept / Principles of Object Oriented Programming

Basically there are 6 basic principles of OOP. They are as follows.

1. Object and Class
2. Data Abstraction and Encapsulation
3. Inheritance
4. Polymorphism
5. Dynamic Binding
6. Message Passing

Object and Class

Object is an instance of a class i.e. variable of class type. Object are the basic runtime entities in an object oriented system. Objects contain data and code to manipulate the data. When a program is executed, the objects interact by sending message to one another. Generally, program objects are chosen such that they match closely with real world objects.

A class is a framework that specifies what data and what functions will be included in objects of that class. It serves as a plan or blueprint from which individual objects are created. A Class is the collection of objects of similar type.
For example: mango, apple and orange are members of class Fruit. so if we create a class Fruit then mango, apple, orange can be the object of Fruit.

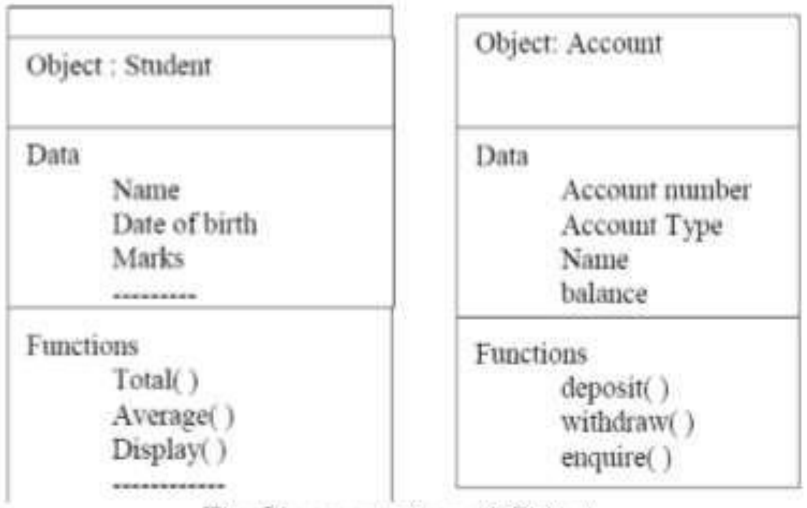


Fig: Representation of Object

Data Abstraction and Encapsulation

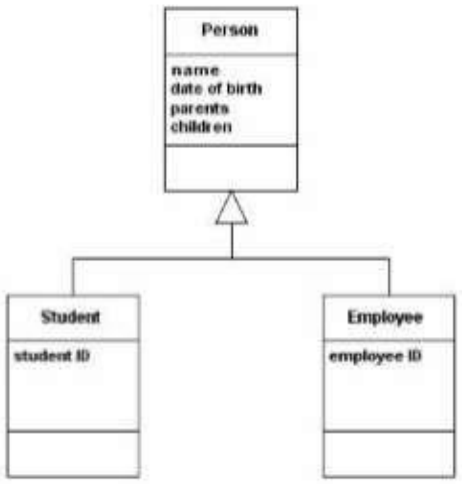
Encapsulation is the process of combining the data (called fields or attributes) and functions (called methods or behaviors) into a single framework called class. Encapsulation helps preventing the modification of data from outside the class by properly assigning the access privilege to the data inside the class. So **the term data hiding** is possible due to the concept of encapsulation, since the data are hidden from the outside world, so that it is safe from accidental alteration.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes uses the concept of abstraction and are defined as a list of abstract attributes and functions. Since classes uses the concept of data abstraction, they are known as abstract data types (ADT).

Inheritance

Inheritance is the process by which objects of one class acquire the characteristics of object of another class. In OOP, the concept of inheritance provides the idea of reusability. We can use additional features to an existing class without modifying it. This is possible by deriving a new class (derived class) from the existing one (base class). This process of deriving a new class from the existing base class is called inheritance.

It supports the concept of hierarchical classification. It allows the extension and reuse of existing code without having to rewrite the code.



In the above figure, Person is base class while Student and Employee are derived form Person so they are referred to as derived class

Polymorphism

Polymorphism means the quality of having more than one form. The representation of different behaviors using the same name is called polymorphism. However, the behavior depends upon the attribute the name holds at particular moment. Example of polymorphism in OOP is operator overloading, function overloading.

Example1:
Operator symbol '+' is used for arithmetic operation between two numbers, however by overloading same operator '+' it can be used for different purpose like concatenation of strings. So the process of making an operator to exhibits different behaviors in different instances is called **operator overloading**.

Example 2:

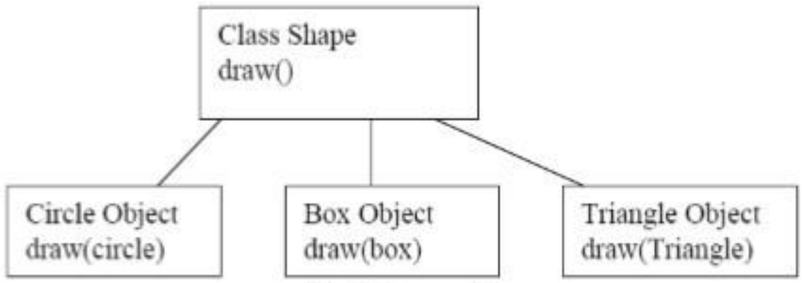


Fig: Polymorphism

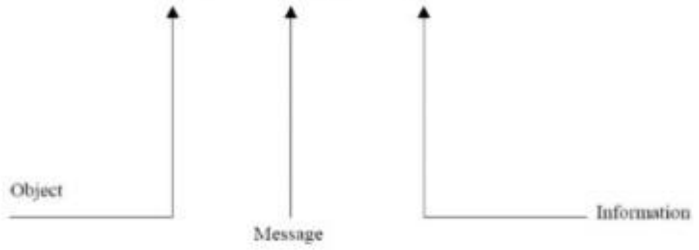
So in the above figure, a single function name is used to handle different number and different type of arguments. So using a single function name to perform different types of task is known as **function overloading**.

Message Passing

An object-oriented program consists of set of objects that communicate with each other. Object communicates with one another by sending and receiving information. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results. Message passing involves specifying the name of object, the name of the function (message) and the information to be sent.
Example:

Employee.Salary(name);



Dynamic Binding

Binding means link between procedure call and code to be execute. Dynamic binding means link exist between procedure call and code to be execute at run time when that procedure is call. It is also known late binding. It is generally use with polymorphism and inheritance. For example, complier comes to know at runtime that which function of sum will be call either with two arguments or with three arguments.

Chapter 2
Classes and Methods

Review of structures

Structure is a convenient tool for handling a group of logically related heterogeneous data types. Structure helps to organize data especially in large programs, because they provide group of variables of different data type to be treated as a single unit. It is most convenient way to keep related data under one roof.

- A structure is usually used when we need to store dissimilar or heterogeneous data together.
- The structure elements are stored in contiguous memory location as array.
- Structure elements can be accessed through a structure variable using a dot (.) operator.
- Structure elements can be accessed through a pointer to a structure using the arrow (->) operator.
- All the elements of one structure variable can be assigned to another structure variable using the assignment (=) operator.
- It is possible to pass a structure variable to a function either by value or by reference.
- It is possible to create an array of structure i.e. similar type of structure is placed in a common variable name. For example: we need to store the detail information of individual student in a class.

Example:

```
struct book
{
   char name[20];
   int pages;
   float price;
};
void main()
{
   struct book b1;
   b1.pages=500;
   b1.price=815.5;
   strcpy(b1.name, "C Programming");
   printf("\nName=%s, pages=%d and price=%f",b1.name,b1.pages,b1.price);
   getch();
   clrscr();
}
```

Limitation of Structure / Structure vs Class

- The standard C does not allow the struct data type to be treated like built-in data types. For example

```
struct complex
{
        float real, imag;
};
struct complex c1,c2,c3;
```

  The complex numbers c1,c2,c3 can easily be assigned values using the dot operator. But we cannot add two complex numbers or subtract one from the other. That is c3=c1 + c2; is illegal in C.

Also data hiding is not permitted in C, that is structure members can be directly accessed by the structure variable by any function anywhere in their scope.
In c structure, it contains only variable or data as member but In C++, a class can have both variables and functions as members. It can also declare some of its members as private so that cannot be accessed directly by the external functions.

The data member in c++ structure are public by default which may lead to security issue but in Class the default access specifier is private.

Structure in C are only object based but Class support all features of OOP.

Class

A class is user-defined data type that includes different member data and associated member functions to access on those data. Object-oriented programming (OOP) encapsulates data (attributes) and functions (behavior) into packages called classes. The data components of the class are called data members and the function components are called member functions.
Since class is a collection of logically related data items and the associated functions which operate and manipulate those data. This entire collection can be called a new **user defined data-type.** So, classes are user-defined data types and behave like the built-in types of a programming language.

Generally a class specification has two parts.

i. Variable declaration

ii. Function definition

The general form of a class declarations is

```
class class-name
{
        private:
                Variable declarations;
                Function declarations;
        public:
                Variable declarations;
                Function declarations;
};
```

```
#include<conio.h>
#include<iostream>
using namespace std;
class student          // class name student, starting of class
```

```
{
private:                // private data members name roll;
      char name[20];
      int roll;
public:                          //public methods input() and output()
      void input()
      {
              cin>>name>>roll;
      }
      void output()
      {
              cout<<"Name : "<<name<<endl;
          cout<<" Roll No :"<<roll<endl;
     }
};
int main()
{
     student s1;        // object (s1) declaration inside main()
     s1.input();        //calling methods syntax: object.method();
     s1.output();       //calling methods syntax: object.method();
     getch();
     return 0;
}
```

Difference between class and structure

Class	Structure
Class is a reference type and its object is created on the heap memory	Structure is a value type and its object is created on the stack memory.
Class can create a subclass that will inherit parent's properties and methods,	Structure does not support the inheritance.
A class has all members private by default.	A struct is a class where members are public by default.
Sizeof empty class is 1 Byte	Size of empty structure is 0 Bytes

Access Specifiers

Access specifiers are the keyword that controls access to data member and member functions within user-defined class. The access restriction to the class members is specified by the labeled public, private, and protected sections within the class body. The keywords public, private, and protected are called access specifiers.

```
class Demo
{
public:
// public members go here
protected:
// protected members go here
private:
// private members go here
};
```

Public

All the class members declared under public will be available to anywhere in the program. 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.

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.

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.

Creating object

Once a class has been specified (or declared), we can create variables of that type (by using the class name as datatype). General syntax for declaration of object is

Class_name   Object_name

Example:

```
Student s1; //memory for s1 is allocated.
```

The above statement creates a variable s1 of type Student. The class variables are known as objects. So, s1 is called object of class Test.

Accessing Class Member

The private data members of a class can be accessed only through the member functions of that class. However, the public members can be accessed from outside the class using object name and dot operator(.). The following is the format for calling public membersof a class.

```
Object_name.Function _name(actual argument)
Object_name.Data_name
```

Example:

If e1 is the object of Class Employee then we can access the public member function getname() with the following statement

```
E1.getname();
```

Defining a member Function

Member functions can be defined in two ways.

- Outside the class
- Inside the class

The code for the function body would be identical in both the cases. Irrespective of the place of definition, the function should perform the same task.

Outside the class

In this approach, the member functions are only declared inside the class, whereas its definition is written outside the class.

General form:

```
return-type class-name::function-name(argument-list)
{
- - - - - - - - -
- - - - - - - -- - //function body
- - - -- - -- --
}
```

Example:

```
void Employee::getdata()
{
- - - - - - - - -
- - - - - - - -- - //function body
- - - -- - -- --
}
```

Inside the class

Function body can be included in the class itself by replacing function declaration by function definition. If it is done, the function is treated as **an inline function**. Hence, all the restrictions that apply to inline function, will also apply here.

Example:

```
class Demo
{
        int a,b;
        public:
        void getdata()
        {
                cin>>a>>b;
        }
};
```

Here, getdata() is defined inside the class. So, it will act like an inline function.

A function defined outside the class can also be made 'inline' simply by using the qualifier 'inline' in the header line of a function definition.

Example:

```
class Demo
{
- - - - - - - - -
- - - - - - - - - -
public:
        void getdata(); // function declaration inside the class
};

void A::getdata()
{
        //function body
}
```

Function in C++

Dividing a program into function is one of the major principles of top-down structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

Syntax of function:

```
void show( ) ; / * Function declaration * /
main ( )
{
- - - - - -
- - - - - -
show( ) ; / * Function call * /
}
void show( ) / * Function definition * /
{
- - - - - -
------ / * Function body * /
------
}
```

Function Prototyping

The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself.

Function prototype is a declaration statement in the calling program and is of of the following form:

type function_name (argument-list) ;

Example:

float volume(int x, int y, float z) ; // legal

float volume (int x, int y, z) ; // illegal

Passing Arguments to Function

An argument is a data passed from a program to the function. In function, we can pass a variable by three ways:

1. Passing by value
2. Passing by reference
3. Passing by address or pointer

Passing by value:

In this the value of actual parameter is passed to formal parameter when we call the function. But actual parameter are not changed.

```
#include <iostream>
using namespace std;
void swap(int, int) ; // declaration prototype
int main ( )
{
        int x,y ;
        x=10 ;
        y=20 ;
        swap(x,y) ;
        cout <<x<<endl;
        cout<<y<<endl ;
        return 0;
}
```

```
void swap(int a, int b) // function definition
{
        int t ;
        t=a ;
        a=b ;
        b=t ;
}
Output:
x=10
y=20
```

Passing by reference:

Passing argument by reference uses a different approach. In this, the reference of original variable is passed to function. But in call by value, the value of variable is passed.

```
#include <iostream>
using namespace std;
void swap(int &, int &) ;
int main( )
{

        int x,y ;
        x=10 ;
        y=20 ;
        swap(x,y);
        cout<<x;
        cout <<y;
        return 0;
}
void swap(int &a, int &b)
{
        int t ;
        t=a ;
        a=b ;
        b=t ;
}
```

Passing by Address or Pointer:

This is similar to passing by reference but only difference is in this case, we can pass the address of a variable.

```
#include <iostream>
using namespace std;
void swap(int *, int *) ;
int main( )
{
        int x, y ;
        x=10 ;
        y=20 ;
        swap(&x,&y);
        cout<<x;
        cout<<y;
        return 0;
}

void swap(int *a, int *b)
{
        int t;
        t=*a ;
        *a=*b ;
        *b=t ;
}
```

Inline Function:

The functions which are expanded inline by the compiler each time it's call is appeared instead of jumping to the called function as usual is called inline function When a function is defined as inline, compiler copies it's body where the function call is made, instead of transferring control to that function. A function is made inline by using a keyword "inline" before the function definition.

Advantage: Increases execution speed

Disadvantages: More memory required because of copying

Example1:

```
#include<conio.h>
```

```
#include<iostream>
using namespace std;
//While declartion no need for keyword inline
int Findcube(int x);
int main()
{
        cout<<Findcube(3);
        getch();
        return 0;
}
inline int Findcube(int x)
{
        return x*x*x;
}
```

Example 2:
WAP to display hello world by using class and inline member function

```
        #include<iostream>
        using namespace std;
        class Demo
        {
                public:
                inline void display();
        };
        int main()
        {
                Demo d;
                 d.display();
        }
        inline void Demo:: display()
        {
                cout<<"hello world";
        }
```

Example of C++ Program with Class

Q. Create a class Rectangle with two data members (length and breadth), a function called readdata() to take detail of sides and a function called displaydata() to display area of rectangle. In main create two objects of a class Rectangle and for each object, call both of the function.

Q. Create a class Rectangle with two data members (length and breadth), a function called readdata() to take detail of sides and a function called displaydata() to display area of rectangle. The two functions are defined outside the class. In main create two objects of a class Rectangle and for each object, call both of the function.

```
#include<iostream>
 using namespace std;
 class Rectangle
 {
         float length,breadth;
         public:
                 void readdata();
                 void displaydata();
 };
 void Rectangle::readdata()
 {
         cout<<"Enter lengtha and breadth of rectangle"<<endl;
         cin>>length>>breadth;
 }
 void Rectangle::displaydata()
 {
         float area;
         area=length*breadth;
         cout<<"Area of rectangle= "<<area<<endl;
 }
 int main()
 {
         Rectangle rect1,rect2;
         rect1.readdata();
         rect1.displaydata();
         rect2.readdata();
         rect2.displaydata();
 }
```

Q. Define a class to represent a bank account. Include the following members

Data Member:

i. Name of account holder

i. Account Number

ii. Account Type

iii. Balance Amount

Member Functions:

- to take detail of the fields from user
- To deposit an amount
- To withdraw an amount
- To display name and balance

```
#include<iostream>
using namespace std;
class Account
{
        char name[50],acctype[20];

    int accno;
    float wdraw,bal,dep;
    public:
            void setdata()
            {
                    cout<<"Enter name of account holder"<<endl;
                    cin>>name;
                    cout<<"Enter the type of account"<<endl;
                    cin>>acctype;
                    cout<<"Enter the account number"<<endl;
                    cin>>accno;
                    cout<<"Enter the balance"<<endl;
                    cin>>bal;
            }
            void deposit()
            {
                    cout<<"Enter amount to be deposited"<<endl;
                    cin>>dep;
                    bal=bal+dep;
            }
            void withdraw()
            {
                    cout<<"Enter amount to be withdraw"<<endl;
                    cin>>wdraw;
                    bal=bal-wdraw;
            }
            void display()
            {
                    cout<<"Name= "<<name<<endl;
                    cout<<"Balance= "<<bal;
            }

};
```

```
int main()
{
            Account a;
            a.setdata();
            a.deposit();
            a.withdraw();
            a.display();
}
```

Private Member Function

As we have seen, member functions are, in general, made public. But in some cases, we may need a private function to hide them from outside world. Private member functions can only be called by another function that is a member of its class. Object of the class cannot invoke private member functions using dot operator.

Example:

```
 class Demo
 {
        int a,b;

       void func1(); //private member by default
       public:
       void func2();
};

void Demo::func2()
{
       Func1(); //func1() is private member so cant be accessed by using object of class Demo
}
```

Q. Define a class Student with the following specification

Private members:

Data member: Id, name, eng, math, science

Member function: float calculateTotal() to return the total obtained mark.

Public members:

Member functions:

getData() to take input(id, name, eng, math, science) from user showData() to display all the data member along with total on the screen.

Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Student
{
        int id;
        char name[50];
        float eng,math,science;
        float calculateTotal()
        {
                return(eng+math+science);
        }
        public:
        void getData()
        {
                cout<<"Enter id"<<endl;
                cin>>id;
                cout<<"Enter name"<<endl;
                cin>>name;
                cout<<"Enter marks in maths, science and english"<<endl;
                cin>>math>>science>>eng;
        }

        void showData()
        {
        float t;
        cout<<"ID= "<<id<<endl;
        cout<<"Name ="<<name<<endl;
        cout<<"Marks in maths, science and english ="<<math<<science<<eng<<endl;


            t=calculateTotal(); // Within the class scope, we can call members directly
with their name
            cout<<"Total= "<<t;
        }

};
int main()
{
        Student s;
        s.getData();
        s.showData();
}
```

Function Overloading

Two or more functions can share the same name as long as either the type of their arguments differs or the number of their arguments differs – or both. When two more functions share the same name, they are said overloaded. Overloaded functions can help reduce the complexity of a program by allowing related operations to be referred to by the same name.

```
#include <iostream>
using namespace std;
int area(int) ; // function area with int return type
double area(double, int) ; // function area with double return type
long area(long, int, int) ; // function area with long return type

int main( )
{

cout<<area(10);
cout<<area(2.5,8);
cout<<area(100L,75,15);
return 0 ;

}
int area(int s) // square
{
return(s*s) ;
}
double area(double r, int h) // Surface area of cylinder ;
{
return(2*3.14*r*h) ;
}
long area(long l, int b, int h) //area of parallelopiped
{
return(2*(l*b+b*h+l*h)) ;
}
```

Default Arguments

When declaring a function, we can specify a default value for each parameter. This value will be used if the corresponding argument is left blank when calling to the function.

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

```
#include <iostream>
using namespace std;
int main( )
{
int divide(int a, int b=2) ; //prototype and b=2 default value
cout<<divide(12);
cout<<endl;
cout<<divide(20,4);
return 0;
}
int divide(int x, int y)
{
int r ;
r = x / y ;
return(r) ;
}
```

O/P

```
6
5
```

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 function 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.

```
#include<iostream>
usingnamespace std;
classDemo
{
private:
int num;
char ch;
public:
void setMyValues(int num, char ch)
        {
        this->num =num;
        this->ch=ch;
        }
void displayMyValues()
        {
cout<<num<<endl;
cout<<ch;
}
};
int main()
{
Demo obj;
obj.setMyValues(100, 'A');
obj.displayMyValues();
return0;
}
```

O/P:

```
        100
        A
```

Static data Members

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. We can define class members static using static keyword.
Unlike normal member variables, static member variables are shared by all objects of the class.

```
class Something
{
public:
        static int s_value;
};

int Something :: s_value = 1;

int main()
{
        Something first;
        Something second;
        first.s_value = 2;
        cout << first.s_value <<'\n';
        cout << second.s_value <<'\n';
        return 0;
}
```

O/p

```
2
2
```

Static member function

Like static member variables, static member functions are not attached to any particular object. Because static member functions are not attached to a particular object, they can be called directly by using the class name and the scope resolution operator. Static member functions have no *this pointer.

We can access a static member function with class name, by using following syntax:

class_name:: function_name(parameter);

```
#include <iostream>
usingnamespacestd;
class Demo
{
private:
//static data members
static int X;
static int Y;
public:
//static member function
static void  Print()
{
cout<<"Value of X: "<< X <<endl;
cout<<"Value of Y: "<< Y <<endl;
}
};
//static data members initializations
int Demo :: X =10;
int Demo :: Y =20;
intmain()
{
Demo OB;
//accessing class name with object name
cout<<"Printing through object name:"<<endl;
OB.Print();
//accessing class name with class name
cout<<"Printing through class name:"<<endl;
Demo::Print();
return0;
}
```

Output

```
Printing through object name:
Value of X: 10
Value of Y: 20
Printing through class name:
Value of X: 10
Value of Y: 20
```

Friend Function

One of the important concepts of is data hiding, i.e., a nonmember function cannot access an object's private or protected data.

But, sometimes this restriction may force programmer to write long and complex codes. So, there is mechanism built in C++ programming to access private or protected data from non-member functions.

This is done using a friend function or/and a friend class.

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

The complier knows a given function is a friend function by the use of the keyword **friend**.

For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

Syntax:

```
class class_name
{
... .. ...
friend return_type function_name(arguments);
... .. ...
}
return_type functionName(argument/s)
{
... .. ...
// Private and protected data of class_Name can be accessed from
// this function because it is a friend function of className.
```

```
... .. ...
}
```

How can we make two classes friendly?

i. Common friend function on both classes (see above examples)

ii. Friend class technique

Like friend function, a class can also be a friend of another class. A friend class can access all the private and protected members of other class. In order to access the private and protected members of a class into friend class we must pass on object of a class to the member functions of friend class.

Example:

```
#include<iostream.h>
 class Rectangle
 {
         int L,B;
         public:
         Rectangle()
         {
                 L=10;
                 B=20;
         }
         friend class Square;       //Statement 1
 };
 class Square
 {
         int S;
         public:
         Square()
         {
                 S=5;
         }
         void Display(Rectangle Rect)
         {

                 cout<<"\n\n\tLength : "<<Rect.L;
                 cout<<"\n\n\tBreadth : "<<Rect.B;
                 cout<<"\n\n\tSide : "<<S;
          }
  };
  void main()
  {
          Rectangle R;
          Square S;
          S.Display(R);     //Statement 2
  }
```

Output :

```
           Length : 10

           Breadth : 20

           Side : 5
```

Q. WAP to find the sum of two number using the concept of friend function

```
#include<iostream>
using namespace std;
class Demo
{
        int fn,sn;
        public:
                void setdata()
                {
                        cout<<"Enter two number"<<endl;
                        cin>>fn>>sn;
                }
                friend void sum(Demo d);
};

void sum(Demo d)
{
        int sum;
        sum=d.fn+d.sn;
        cout<<"Sum= "<<sum;
}

int main()
{

        Demo a;
        a.setdata();
        sum(a);
}
```

Q. Create classes called class1 and class2 with each of having one private member. Add member function to set a value (say setvalue) on each class. Add one more function max () that is friendly to both classes. max() function should compare two private member of two classes and show maximum among them. Create one-one object of each class then set a value on them. Display the maximum number among them.

```
#include<iostream>
using namespace std;
class Class2;// forward declaration
class Class1
{
        int a;
        public:
                void setdata()
                {
                        cout<<"Enter a number"<<endl;
                        cin>>a;
                }
                friend void max(Class1,Class2);
};
class Class2
{
        int a;
        public:
                void setdata()
                {
                        cout<<"Enter a number"<<endl;
                        cin>>a;
                }
                friend void max(Class1,Class2);
};
```

```
void max(Class1 c1, Class2 c2)
{
        if(c1.a>c2.a)
        {
                cout<<"Larger= "<<c1.a;
        }
        else
        {
                cout<<"Larger= "<<c2.a;
        }
}
int main()
{
        Class1 x;
        Class2 y;

         x.setdata();
         y.setdata();
         max(x,y);
 }
```

Reference Variable

Reference operator (&) is used to define referencing variable Reference variable prepares an alternative name (alias name) for previously defined variable

Syntax of referencing variable

Here same variable can be used with help of 2 names

```
        <return-type>&reference-variable =variable;
```

Example

```
        int y=9;
        int &x=y;
```

Now value of y is 9 and after initialization x also becomes 9

State and Behaviour of Object

State

An objects state is defined by the attributes (i.e. data members or variables) of the object
In OOP state is defined as data members
It is determined by the values of its attributes
What the objects have, Example: Student have a first name, last name, age, etc...

Behaviour

An objects behaviour is defined by the methods or action (i.e. Member functions) of the object
In OOP behaviours are defined as member function
It determines the actions of a object
What the objects do, Example Student attend a course "OOP", "C programming" etc…

Example: Consider Lamp as an Object

Its states are on/off and behaviour are turn on/ turn of

Chapter 3
Message, Instance and Initialization

Message Passing

Message Passing is sending and receiving of information by the objects same as people exchange information. So this helps in building systems that simulate real life. Following are the basic steps in message passing.

- Creating classes that define objects and its behavior.
- Creating objects from class definitions
- Establishing communication among objects

In OOPs, Message Passing involves specifying the name of objects, the name of the function, and the information to be sent.

Message passing syntax

object.methodname(arguments);

Where, object is a receiver.

Method name is a message

Arguments are information to be passed.

Object as function Argument

Unlike other argument, we can also pass object as function argument using following two ways.

Pass by value:

A copy of entire object is passed to the function. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function.

Pass by reference:

Only the address of the object is transferred to the function. Since the actual address of object is passed, any changes made to the object inside the function will reflect in the actual object used to call the function. This method is more efficient than previous one.

Q. WAP to perform the addition of time in hour and minute format using the concept of object as function arguments (Pass by value)

```
#include<iostream>
using namespace std;
class Time
{
        int hour,min;
        public:
                void gettime()
                {
                        cout<<"Enter hour"<<endl;
                        cin>>hour;
                        cout<<"Enter minute"<<endl;
                        cin>>min;
                }
                void puttime()
                {
                        cout<<"Hour= "<<hour<<"\t";
                        cout<<"Minute= "<<min<<endl;
                }
                void add(Time a, Time b)
                {
                        min=a.min +b.min;
                        hour=min/60;
                        min=min%60;
                        hour=hour +a.hour+b.hour;
                }
};
int main()
{
        Time t1,t2,t3;
        t1.gettime();
        t2.gettime();
        t3.add(t1,t2);
        t1.puttime();
        t2.puttime();
        t3.puttime();
}
```

Q. WAP to perform the addition of time in hour and minute format using the concept of object as function arguments(Pass by reference)

```
#include<iostream>

using namespace std;

class Time

{

        int hour,min;
        public:
                void gettime()
                {
                        cout<<"Enter hour"<<endl;
                        cin>>hour;
                        cout<<"Enter minute"<<endl;
                        cin>>min;
                }
                void puttime()
                {
                        cout<<"Hour= "<<hour<<"\t";
                        cout<<"Minute= "<<min<<endl;
                }
                void add(Time &a, Time &b) // object 'a' work on the same address that of object t1
                {
                        min=a.min +b.min;
                        hour=min/60;
                        min=min%60;
                        hour=hour +a.hour+b.hour;
                }
};
 int main()
 {
        Time t1,t2,t3;
        t1.gettime();
        t2.ge1ttime();
        t3.add(t1,t2);
        t1.puttime();
        t2.puttime();
        t3.puttime();
 }
```

Returning Objects

A function can take object as argument as well as return object.

Q. WAP to find the square of a given number using the concept of object as argument(call by reference) and returning object.

```
#include<iostream>
using namespace std;
class Demo
{
        int a;
        public:
                void setdata(int x)
                {
                        a=x;
                }

                Demo square(Demo *p)
                {
                        Demo x;
                        x.a=p->a * p->a;
                        return x;

                }

                void putdata()
                {
                        cout<<a;
                }
};
int main()
{
        Demo o1,o2,o3;
        o1.setdata(10);
        o3=o2.square(&o1);
        o3.putdata();
}
```

Q. Create a class Complex with two data member (x and y) for storing real and imaginary part of a complex number and three member functions named void input(int ,int) to initialize x and y, Complex sum(Complex, Complex) to return the Complex object with sum, and void show() to display the sum of two complex number. Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Complex
{
        int x,y;
        public:
        void input(int real,int img)
        {
                x=real;
                y=img;
        }

        Complex sum (Complex c1, Complex c2)
        {
                Complex c3;
                c3.x=c1.x+c2.x;
                c3.y=c1.y+c2.y;
                return c3;
        }
        void show()
        {
                cout<<"Real Part= "<<x<<endl;
                cout<<"Imaginary part= "<<y<<endl;
        }
};
int main()
{
        Complex a,b,c;

        a.input(5,6);
        b.input(4,5);
        c=c.sum(a,b);
        c.show();
}
```

Initialization class Object

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as setdata() and diplay() to provide initial values to the private member variables.
An OOP also has a provision of initializing objects of a class during their definition itself. A class in C++ may contain two special member functions dealing with the internal working of a class. These functions are the constructors and the destructors. A constructor enables an object to initialize itself during creation and the destructor destroys the object when it is no longer required, by releasing all the resources allocated to it.

Constructor

A constructor is a special member function having the same name as that of the class which is used to automatically initialize the objects of the class type with legal initial values. The constructor is invoked automatically whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

Properties:

- They are also used to allocate memory for a class object.

- They execute automatically when an object of a class is created.
- Constructor's name is same as that of class name.
- They should be declared in the "public" section.
- They do not have return types, not even void and therefore, and they cannot return any values.
- Like C++ functions, they can have default arguments.
- Constructor is NOT called when a pointer of a class is created.

A constructor is declared and defined as below

```
class Demo
{
        private:
        --------------
        --------------
        public:
        Demo(); //constructor
};
Demo ::Demo() //note no return type required
{
        //Body of constructor if defined outside the class
}
```

Types of constructor

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

Default Constructor

A constructor that does not take any parameter is called default constructor. This constructor is always called by compiler if no user-defined constructor is provided.
The following Class definition shows a default constructor.

```
class Demo
{
        public:
        Demo()  //Default Constructor
        {

        }
};
```

This constructor is also called as implicit constructor because if we do not provide any constructor with a class, the compiler provided one would be the default constructor. And it does not do anything other than allocating memory for the class object.

Example:
Create a class Demo with two data member (id, name), a default constructor to initialize these fields and a member function display() to show student details. Write a main program to test your class.

```
#include<iostream.>
#include<string.h>
using namespace std;
class Demo
{
        int id;
        char name[10];
        public:
                Demo() // Default constructor
                {
                        id=5;
                        strcpy(name,"ram");
                }
                void display()
                {
                        cout<<"ID= "<<id<<endl;
                        cout<<"Name= "<<name<<endl;
                }
};
int main()
{
        Demo d;
        d.display();
}
```

Parameterized constructor

The constructors that can take arguments or parameters are called parameterized constructor. In this, we pass the initial value as arguments to the constructor function when the object is declared.
The following Class definition shows a default constructor.

```
class Demo
{

         public:
         Demo(int x, int y)  //Parameterized Constructor
         {
                 // constructor body
         }
 };
```

Q. WAP to find the area of rectangle using the concept of parameterized constructor

```
#include<iostream>
using namespace std;
class Demo
{
        int l,b;
        public:
                Demo(int length, int breadth)
                {
                        l=length;
                        b=breadth;
                }
                void area()
                {
                        int a;
                        a=l*b;
                        cout<<"Area of rectangle= "<<a<<endl;
                }
};
int main()
{
        Demo d(5,6);
        d.area();
}
```

Q. Create a class Person with data member Name, age, address and citizenship_number. Write a constructor to initialize the value of a person. Assign citizenship number if the age of the person is greater than 16 otherwise assign values zero to citizenship number. Also, create a function to display the values. Write a main program to test your class.

```
#include<iostream.>
#include<string.h>
using namespace std;
class Demo
{
        char name[50],address[50];
        int citizenship_number,age;
        public:
                Demo(char n[],char ad[],int a, int cn)
                {
                        strcpy(name,n);
                        strcpy(address,ad);
                        citizenship_number=cn;
                        age=a;
                }
                void display()
                {
                        cout<<"Name= "<<name<<endl;
                        cout<<"Address= "<<address<<endl;
                        cout<<"Age= "<<age<<endl;
                        cout<<"CitizenshipNumber="<<citizenship_number<<endl;
                }

};
int main()
{
        char nm[50],ad[50];
        int cn,a;
        cout<<"Enter name"<<endl;
        cin>>nm;
        cout<<"Enter address"<<endl;
        cin>>ad;
        cout<<"Enter age"<<endl;
        cin>>a;
        cn=0;
        if(a>16)
        {
                cout<<"Enter Citizenship Number"<<endl;
                cin>>cn;
        }
        Demo d(nm,ad,a,cn);
        d.display();

}
```

Copy Constructor

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. Parameterized constructors having object of the same class as parameter or argument is called copy constructor. The copy constructor is used to:

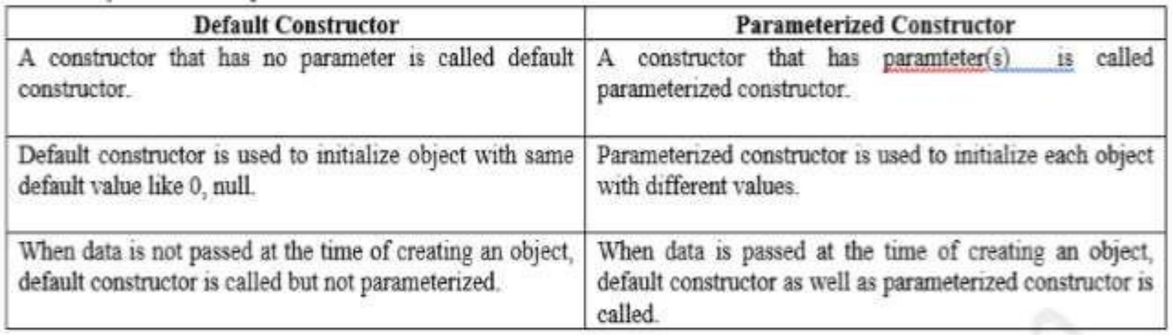
Initialize one object from another of the same type.

- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

Q. A program to demonstrate the concept of copy constructor

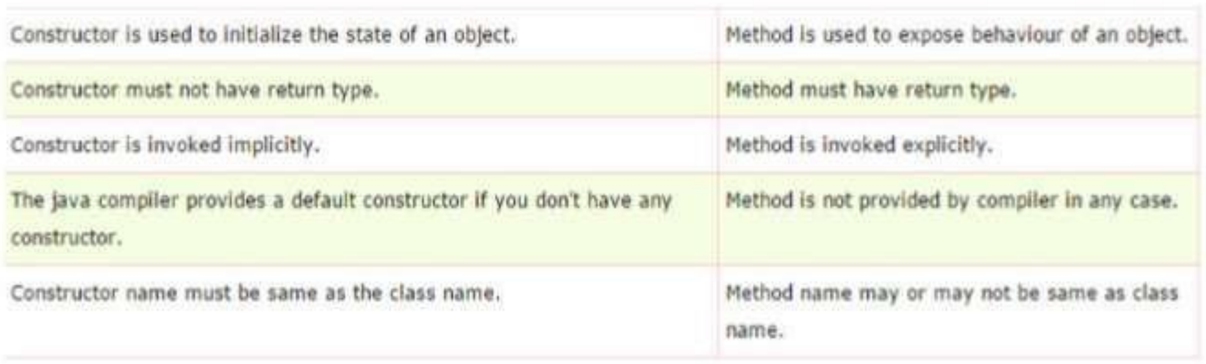
```
#include<iostream>
using namespace std;
class Demo
{
        int a;
        public:
                Demo()//default constructor
                {

                }
                Demo (int b)
                {
                        a=b;
                }
                void display()
                {
                        cout<<a<<endl;
                }
                //the below definition of copy constructor is optional
                Demo (Demo &x)
                {
                        a=x.a;
                }
};
int main()
{

        Demo d(5);

        Demo x(d); // copy constructor will called and if no copy constructor available the compiler will create one

        Demo y=d; //copy constructor will called and if no copy constructor available the compiler will create one

        Demo z;
        z=d; // It will not call copy constructor but assigns value member by member
        d.display();
        x.display();
        y.display();
        z.display();
}
```

Default vs parameterized constructor



Default Constructor	Parameterized Constructor
A constructor that has no parameter is called default constructor.	A constructor that has paramteter(s) is called parameterized constructor.
Default constructor is used to initialize object with same default value like 0, null.	Parameterized constructor is used to initialize each object with different values.
When data is not passed at the time of creating an object, default constructor is called but not parameterized.	When data is passed at the time of creating an object, default constructor as well as parameterized constructor is called.

Constructor vs Method



Constructor	Methods
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Constructor Overloading

When more than one constructor functions are defined in the same class then we say the constructor is overloaded. All the constructors have the same name as the corresponding class, and they differ in terms of number of arguments, data types of argument or both. This makes the creation of object flexible.

Example

```
class Demo
{
        public:
                Demo()  //Default Constructor
                {

                }
                Demo(parameter1) // Parameterized constructor with one parameter
                {

                }
                Demo(parameter1,parameter2)//Parameterized constructor with two parameters
                {

                }
}
```

In the above example, the constructor is overloaded because we have three different definition for same constructor name with different signature.

Q. WAP to find the area of circle and rectangle using the concept of constructor overloading

```
#include<iostream.>
using namespace std;
class Demo
{
        public:
                Demo(float radius)
                {
                        cout<<"Area of circle= "<<(3.1416*radius*radius)<<endl;
                }
                Demo(float length, float breadth)
                {
                        cout<<"Area of rectangle= "<<(length * breadth)<<endl;
                }
};
int main()
{
        Demo d(4);// invokes constructor with one parameter
        Demo d1(5,6); //invokes constructor with two parameter
}
```

Constructor with default argument

We can define constructor with default argument unlike function with default argument. The following program demonstrates the concept of default constructor.

```
#include<iostream>
using namespace std;
class Demo
{
        int l,b;
        public:
                Demo(int length, int breadth=20)
                {
                        l=length;
                        b=breadth;
                }
                void area()
                {
                        cout<<"Area of rectangle= "<<(l*b)<<endl;
                }

};
int main()
{
        Demo d(4);

      Demo d1(4,5);
      d.area();
      d1.area();
 }
```

Destructors

Destructors are the special function that destroys the object that has been created by a constructor. In other words, they are used to release dynamically allocated memory and to perform other "cleanup" activities. Destructors, too, have special name, a class name preceded by a tilde sign (~).

Example:

```
~Demo ()
{

}
```

Destructor gets invoked, automatically, when an object goes out of scope (i.e. exit from the program, or block or function). They are also defined in the public section. Destructor never takes any argument, nor does it return any value. So, they cannot be overloaded.

Note: whenever 'new' is used to allocate memory in the constructors, we should use 'delete' to free the memory.

For example:

```
Demo:: ~Demo()
{
        delete obj;
}
```

Also destructor can be defined inside class

```
#include<iostream>
using namespace std;
class Demo
{
        int *p;
        public:
                Demo()
                {
                        p=new int();
                        *p=7;
                }
        ~Demo()
        {
                delete(p);
        }
};
```

For example:

Note: Destructor also can be invoked manually but basically it is useless because the destructor is automatically invoked when the scope of created object ends. So this lead to invoking of destructor twice.

```
#include<iostream>
using namespace std;
class Demo
{

};
int main()
{
        Demo d;
        d.~Demo();

}
```

Characteristic of Destructor

- It can't take any argument so can't be overloaded.
- It has same name as the class name but preceded by a tilde symbol.
- It can't be declared statics.
- It can't return a value.
- It should have public or protected access specifiers.
- Only one destructor exists in a class.

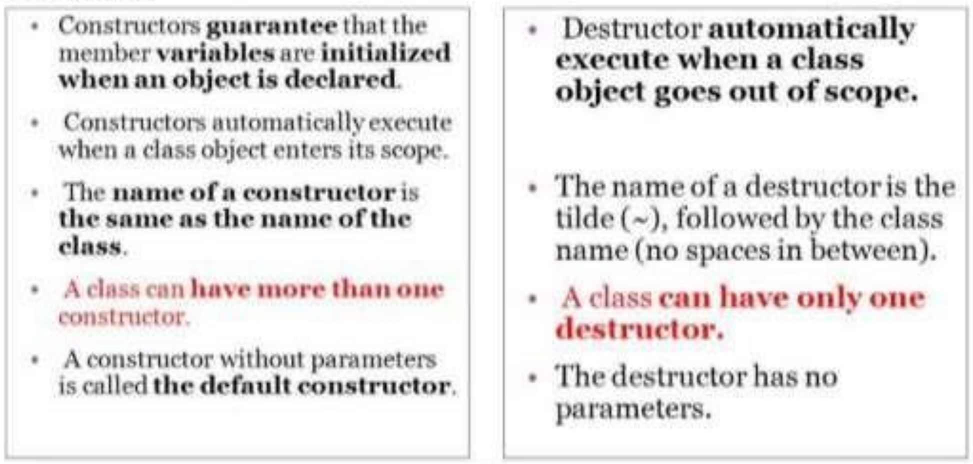
Example:

The below program demonstrates the concept of constructor and destructor

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

```
class Demo
{
static int count;
        public:
        Demo()
        {
                count++;
                cout<<"Object created= "<<count<<endl;
        }
        ~Demo()
        {
                cout<<"Object Destroyed= "<<count<<endl;
                count--;
        }
};
int Demo::count;
int main()
{
Demo a1,a2,a3;
        {
                Demo a4;
        }
}
```

Constructor vs Destructor



- Constructors **guarantee** that the member **variables** are **initialized when an object is declared**.
- Constructors automatically execute when a class object enters its scope.
- The **name of a constructor** is **the same as the name of the class**.
- A class can **have more than one** constructor.
- A constructor without parameters is called **the default constructor**.

- Destructor **automatically execute when a class object goes out of scope.**
- The name of a destructor is the tilde (~), followed by the class name (no spaces in between).
- A class **can have only one destructor.**
- The destructor has no parameters.

Q. "Can there be more than one destructor in the same class? If no, explain it with suitable example."