Defining Member Function:

Member Functions can be defined in two places

(i) Inside the class definition and

(ii) Outside the class definition

(i) Member function inside the class:-

- Member function inside the class can be declared in public (or) private section.
- The member function defined inside the class are treated as inline function.
- The member functions are defined inside the class when it is small otherwise it should be defined outside the class.

The following program illustrate the use of member function inside the class in public section.

Example:

```
//Member function within the class

#include<iostream.h>
#include<conio.h>
class item
{
 private:                          //member data
            int codeno;
            float price;
            int qty;
public:                           //member function
             void getdata()
             {
               codeno=123;
               Price=250.50;
               qty=100;
             }
            void display()
            {
                cout<<"Codeno="<<codeno<<endl;
                cout<<"Price="<<price<<endl;
                cout<<"Quantity="<<qty<<endl;
            }
};

void main()
{
   item one;          //object declartion
   clrscr();
   one.getdata();    //calling member function
   one.display();
   getch();
}
```

Output:-

Codeno=123

Price=250.5

Quantity=100

Explanation:

In the above program, the member function getdata() and display() is defined inside the class in public section. In main() object one is declared. We know that an object has permission to access the public member of the class. The object one is invokes the public member function getdata() and initializes their values and display() invokes to display the result as an output.

(ii) Member function outside the class:-

Member function outside the class is defined when the function is large. To define a function outside the class, the following steps should be followed

1. The prototype of a function must be declared inside the class.

2. The function name must be preceded by class name and its return type separated by scope resolution operator(::).

Syntax:-

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

The following example illustrates the function defined outside the class

```
//Member function outside the class

#include<iostream.h>
#include<conio.h>
class item
{
 private:                          //member data
            int codeno;
            float price;
            int qty;
public:
            void getdata();    //function prototype
            void display();
};
 void item :: getdata()
  {
      codeno=123;
      price=250.50;
      qty=100;
  }
  void item  :: display()
  {
       cout<<"Codeno="<<codeno<<endl;
       cout<<"Price="<<price<<endl;
       cout<<"Quantity="<<qty<<endl;
   }

void main()
{
   item one;          //object declartion
   clrscr();
   one.getdata();   //calling member function
   one.display();
   getch();
}
```

Output:-

```
Codeno=123

Price=250.5

Quantity=100
```

Explanation:-

In the above program, the prototype of a function getdata() and display() is declared inside the class.

The function definition is defined outside the class as void item :: getdata() and void item :: display(). The Scope access operator seperates the class name and function name .

Here return type is void, so there is no any return value from function after calling from main().

Characteristics of member function:-

1. The difference between member and normal function is that the normal function can be invoked freely whereas the member function only using an object of the same class.

2. The same function can be used in any number of classes. This is possible because the scope of the function is limited to their classes and cannot overlap one another.

3. The private data or private function can be accessed by public member function. Other function have no access permission.

4. The member function can invoke one another without using an object or dot operator.

Array of Objects:-

An object of class represents a single record in memory, if we want more than one record of class type, we have to create an array of class or object. As we know, an array is a collection of similar type, therefore an array can be a collection of class type.

Syntax:-

```
class class-name
{
  private:
            member of datas;
  public:
            member of functions;
};

Void main()
{
   Class-name object-name[size];
    -------
}
```

Example:-

```
class student
{
  private:
            int idno;
            Char name[30];
  public:
            void getdata();
            void display();
};
void main()
{
   student stats[5];
   student elec[5];
   student phy[5];
    --------------------
}
```

Memory Representation of above example,

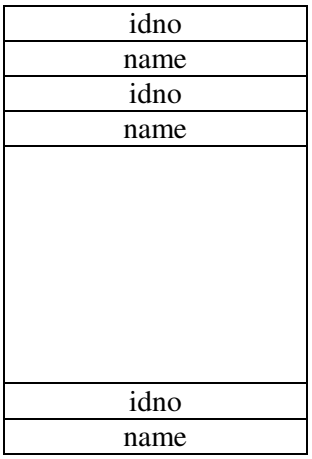

Fig. Array of object for stats[5]

As show above figure, array of objects of type student are created. The array stats[5] contains idno and name for five objects. The next two declarations can maintain the same information of other student elec[5] and phy[5]. These arrays can be initialized (or) accessed like an ordinary array.

Example:-

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

class student
{
 private:
        int idno;
        char name[20];
 public:
        void getdata();
        void display();
};

void student::getdata()
{
 cout<<"Enter student id:";
 cin>>idno;
 cout<<"Enter name:";
 cin>>name;
}
void student::display()
{
 cout<<"Student Id no:"<<idno<<endl;
 cout<<"Name:"<<name<<endl;
}

void main()
{
 student s[5];
 int i;
 clrscr();
 cout<<"Enter studenet details..."<<endl;
 for(i=0;i<5;i++)
 {
  s[i].getdata();
 }
 cout<<"Student details are..."<<endl;
 for(i=0;i<5;i++)
 {
  s[i].display();
 }
 getch();
}
```

Explanation:-

In the above program, the member function getdata() is used to read the information and display() is used to display the output on screen.

In main(), array of object is created as s[5]. By using object, we invoke(call) the two member function declared in the class definition.

Member function with object as arguments:-

Similar to variables, object can be passed to functions. The following are the three methods to pass argument to a function,

a) Pass–by–Value:- A copy of object (actual object) is sent to function and assigned to the object of called function (formal object). Both actual and formal copies of objects are stored at different memory locations. Hence, changes made in formal object are not reflected to actual object.

b) Pass–by–Reference:- Address of object is implicitly sent to function.

c) Pass–by–Address:- Address of object is explicitly sent to function.

In pass-by-reference and pass-by-address methods, an address of actual object is passed to the function. The formal argument is reference/pointer to the actual object.

Hence, changes made in the object are reflected to actual object.

Examples:-

a) Pass-by-Value:-

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

class addition
{
  private:
            int a,b,c;
  public:
           void getdata();
           void display();
};

void addition :: getdata()
{
  cout << ”Enter two values: “ << endl;
  cin >> a >> b;
}

void addition :: display(addition k)
{
  c = k.a + k.b;
  cout << “Addition of two number:” << c;
}

void main()
{
   addition a1;
   clrscr();
   a1.getdata();
   a1.display(a1);
   getch();
}
```

b) Pass-by-Address:-

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

class addition
{
  private:
            int a,b,c;
  public:
           void getdata();
           void display();
};

void addition :: getdata()
{
  cout << ”Enter two values: “ << endl;
  cin >> a >> b;
}

void addition :: display(addition *k)
{
  c = k - > a + k - > b;
  cout << “Addition of two number:” << c;
}

void main()
{
   addition a1;
   clrscr();
   a1.getdata();
   a1.display(&a1);
   getch();
}
```

Argument as Return type:-

A function can also return objects either by value or by reference. When an object is returned by value from a function, a temporary object is created within the function, which holds the return value. This value is further assigned to another object in the calling function.

Syntax:-

class_name function_name (parameter_list)

{

// body of the function

}

Example:-

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

class addition
{
  private:
            int a,b;
  public:
            void getdata();
            addition sum(addition s1,addition s2);
            void display();
};

void addition :: getdata()
{
  cout << "Enter two values: " << endl;
  cin >> a >> b;
}

addition  addition :: sum(addition s1, addition s2)
{
    addition temp;
    temp.a = s1.a + s2.a;
    temp.b = s1.b + s2.b;
    return temp; //return object to calling function
}

void addition :: display()
{
  cout << "Sum of A = " << a << endl;
  cout<< "Sum of B =" << b;
}
```

```
void main()
{
   addition a1,a2,a3;
   clrscr();
   cout << "First Object:" << endl;
   a1.getdata();
   cout << "Second Object:" << endl;
   a2.getdata();
   a3 = a3.sum(a1,a2); //passing object as argument
   cout << "Third Object Values:" << endl;
   a3.display();
   getch();
}
```

Output:-

```
First Object:

Enter two values: 10 100

Second Object:

Enter two values: 20 200

Third Object Values:

Sum of A = 30

Sum of B = 300
```

Explanation:- Write down by yourself….

Static Member Data:-

- 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.
- We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

i.e. **data-type class-name :: variable-name = 0;**

Example:-

```
class number
{
   private:
            static int a;  // static member data
            int b;         // normal member data
  public:
            void getdata()
           {
              a = 10;
              b = 20;
              a++;
              b++;
           }
           void display()
          {
             cout<< a << endl << b;
          }
};
int number :: a = 0;   // initialization of static member data

void main()
{
   number n1,n2;
   clrscr();
   n1.getdata();
  n2.getdata();
  n1.display();
  n2.display();
  getch();
}
```

Output:-

12

21

12

21

Explanation:- In the above program, a is static data and b is normal data. When n1,n2 objects are created, 'a' value is common for all the object.

While invoking the getdata() for n1, a is 11 and b is 21. For n2, again a value is increased by 1 ,a is 12 and b is 21(which is individual memory for object).

Finally calling display() , the result is printed(as shown above).

Note:- Here 'a' value increased two time

Static Member Function:-

- Like member data, you can also use static keyword preceded to member function, known as static member function.

  i.e,. static return-type function-name( ); Ex:- static void display( );

- By declaring a function member as static, you make it independent of any particular object of the class.
- A static member function can be **called even if no objects of the class** exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.
- A static member function can only **access static data member**, other static member functions and any other functions from outside the class.
- Static member functions have a class scope and they do not have access to the **this** pointer of the class.
- When one of the object changes the value of member data variables, the effect is visible to all the objects of the class.
- You could use a static member function to determine whether some objects of the class have been created or not.

Example:-

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

class number
{
  private:
            static int a;   // static data
                   int b;   // normal data
   public:
            static void print1() //static member function
            {
               cout<<"A="<<a<<endl;
            }
             void print2()      //normal member function
             {
                 b=100;
                 cout<<"B="<<b<<endl;
             }
};

int number :: a=10;
```

```
 void main()
 {
    number n1;
    clrscr();

     //calling by object
    n1.print1();

    //calling by class (because a is static)
    number :: print1();

   //here b is normal function
    n1.print2();
    getch();
}
```

Output:-

10

10

100

Explanation:

In class definition we declared one static member data and one normal member data in private section. In public section, declared one static member function and normal member function.

In main( ), we created n1 as object and called the static member function using object and class

i.e. n1.print1( ) and number :: print1() for access static member data value 'a' . Finally we invoked non-static member function to display b value. i.e. n1.print2( ).

Note: You can access both member data and static member data is member function. But in static member function you can access static member data only.

Friend Function and Friend Classes:-

Introduction:-

- One of the important concepts of OOP 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.**

Friend Function:-

(Write introduction)

- 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 className
{
   ... .. ...
   friend return_type functionName(argument/s);
   ... .. ...
};
  return_type functionName(argument/s)
{
   ... .. ...   // Private and protected data of className can be accessed from
```

```
    // this function because it is a friend function of className.
    ... .. ...
}
```

Properties of Friend function:-

- There is no scope restriction for the friend function. Hence, they can be called directly without using objects.
- Unlike member functions of class, friend function cannot access the member directly.
- By default, friendship is not shared (mutual). For example, if class X is declared as friend of Y, this does not mean that Y has privileges(rights) to access private members of class X.
- Use of friend function is rarely done, because it violates(breaks) the rule of encapsulation and data hiding.
- The function can be declared either in public or private sections.

Example:-

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

class data
{
  private:
            int a,b;
  public:
            void getdata()
            {
               cout<<"Enter a and b value:";
                cin >> a >> b;
            }
            friend void sum(data);
};

void sum(data d)
{
  cout << " SUM = " << d.a + d.b ;
}

void main()
{
  clrscr();
  data d;
  d.getdata();
  sum(d);
  getch();
}
```

Output:-

Enter a and b value: 10 20

SUM = 30

Friend Classes:-

(Write introduction)

- Similar to friend function, a class can also be declared to be the friend of some other class.
- When we create a friend class then all the member functions of the friend class also become the friend of the other class.
- This requires the condition that the friend becoming class must be first declared or defined (forward declaration).

Example:-

```
... .. ...
class B;
class A
{
  // class B is a friend class of class A
  friend class B;
  ... .. ...
};

class B
{
  ... .. ...
};
```

In this example, all member functions of class B will be friend functions of class A. Thus, any member function of class B can access the private and protected data of class A. But, member functions of class A cannot access the data of class B.

- Friend relation in C++ is **only granted, not taken.**

Example:-

```
#include<iostream.h>
#include<conio.h>
class two;
class one
{
   private:
            int a,b;
   public:
            void getdata()
            {
               cout << "Enter a and b value: ";
               cin >> a >> b;
             }
            friend class two;
};
```

```
class two
{
   private:
            int c;
    public:
           void add(one d)
           {
             c = d.a + d.b;
             cout << " SUM = " << c;
            }
};

void main()
{
    clrscr();
    one k;
    two t;
    k.getdata();
    t.add(k);
    getch();
}
```

Output:-

Enter a and b value: 10 20

SUM = 30

Constructor:-

The process of creating and deleting objects in c++ is vital(necessary) task.

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors initialize values to object members after storage is allocated to the object.

The constructor is declared and defined as follows

```
class value
{
    private:
                int a,b;
    public:
                value( );    //constructor declaration
};
value :: value( )           //constructor definition
{
      a = 10;
      b = 20;
}
```

In the above example, whenever object created to class, it automatically invokes the function at compile time.

Characteristics of Constructor:-

1. They should be declared in the public section.
2. They are invoked directly when an object is created.
3. They don't have return type, not even void and hence can't return any values.
4. Like other C++ functions, they can have default arguments.
5. Constructors can be inside the class definition or outside the class definition.
6. Constructor can't be friend function.
7. Constructors can't be virtual.
8. They can't be inherited; through a derived class, can call the base class constructor.
9. They can't be used in union.
10. They make implicit calls to the operators new and delete when memory allocation is required.
11. When a constructor is declared for a class, initialization of the class objects becomes necessary after declaring the constructor.

Types of Constructor:-

There are five type of constructor listed below

1.Default Constructor / Empty Constructor
2.Parameterized Constructor
3.Copy Constructor
4.Constructor Overloading / Multiple Constructor and
5.Dynamic Constructor

1.Default Constructor :-

- Default Constructor is the constructor which does not take any argument
- It has no parameter, but programmer can write some initialization statement there.
- Even we do not define a constructor explicitly, the compiler automatically provides a default constructor implicitly.

Syntax:-

```
class class-name
{
    private:
            member data;
    public:
        class-name( ) //default constructor( without argument)
        {
          ------------
         }
};
```

Example: //Default Argument

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

class value
{
    private:
            int a,b;
    public:
            value( )
            {
               a = 10;
               b = 20;
                cout << a << endl << b;
            }
};

void main()
{
     clrscr();
     value v1; //constructor executed;
     getch();
 }
```

Output:-
10 20

2. Parameterized Constructor:-

The constructor with arguments are called parameterized constructor. Using this constructor, you can provide different values to data members of different objects, by passing the appropriate values as arguments.

Syntax:-

```
class class-name
{
    private:
            member data;
    public:

        class-name(argument list) //parameter constructor
        {
          --------
         }
};
```

```
void main()
{
     clrscr();
     value v1(100,200);
//passing arguments
     getch();
 }
```

Output:-
100 200

Example: //Parameter Constructor

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

class value
{
    private:
            int a,b;
    public:
            value(int x, int y )
            {
               a = x;
               b = y;
                cout << x << endl << y;
            }
};
```

3. Copy Constructor:-

The Copy Constructor is a constructor which creates an object by initializing it with another object of the same class, which has been created previously. The copy constructor is used to

- Initialize on object from another object 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.

Syntax:-

```
class class-name
{
    private:
              member data;
    public:
              class-name(class-name &obj )
              {
                a = 10;
                b = 20;
              }
};
```

Example:

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

class value
{
    private:
               int a,b;
    public:
              value(int x, int y)
              {
                a = x;
                b = y;
              }

              value(value &k)
              {
                   a = k.a;
                   b = k.b;
              }

             void display( )
             {
               cout << a << endl;
               cout << b << endl;
             }
};
```

```
void main()
{
     clrscr();
     value v1(50,100);
     value v2(v1);      // by argument
     value v3 = v1;    // by assignment
     cout<< "First Object values:" << endl;
     v1.display( );
     cout<< "Second Object values:" << endl;
     v2.display( );
     cout<< "Third Object values:" << endl;
     v3.display( );


     getch();
 }
```

Output:-

```
First Object values:
50
100
Second Object values:
50
100
Third Object values
50
100
```

4. Constructor Overloading / Multiple Constructor:

Constructor can be overloaded in a similar way as function overloading. Overloaded constructors have the same name (name of the class) but different number of arguments. Depending upon the number and type of arguments passed, specific constructor is called.

Since, there are multiple constructors present, argument to the constructor should also be passed while creating the object.

Syntax:

```
class class-name
{
      private:
              member data;
      public:
             class-name( );
             class-name(argument);
             class-name(argument,argument);
};
```

Example:-

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

class value
{
   private:
             int a,b;
   public:
             value()
             {
                a = 10;
                b = 20;
             }

              value(int x)
              {
                 a = x;
                 b = NULL;
              }

              value(int x, int y)
              {
                 a = x;
                 b = y;
              }
          void dislay()
          {
              cout << "A = " << endl;
              cout << "B = " << endl;
          }
};

void main()
{
     clrscr();
     value v1,v2(34),v3(59,95);
     cout << "First Object" << endl;
     v1.display();
     cout << "First Object" << endl;
     v2.display();
     cout << "First Object" << endl;
     v3.display();
     getch();
}
```

Output:-

```
First Object:
A = 10
B = 20

Second Object:
A = 34
B = 0

Third Object:
A = 59
B = 95
```

5. Dynamic Constructor:-

Dynamic constructor is used to allocate the memory to the objects at the run time.Memory is allocated at run time with the help of 'new' operator.

By using this constructor, we can dynamically initialize the objects.

Example:-

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

class dynamiccons
{
 private:
          int *p;
 public:
          dynamiccons()
          {
            p = new int;
            *p = 100;
          }

          dynamiccons(int v)
          {
            p = new int;
            *p = v;
          }

          void display()
          {
           cout<<*p<<endl;
          }
};
void main()
{
 clrscr();
 dynamiccons d1,d2(500);
 cout<<"First pointer object value: ";
 d1.display();
 cout<<"Second Pointer object value: ";
 d2.display();
 getch();
}
```

Output:

First pointer object value: 100
Second pointer object value: 500

Destructor:-

- 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.

Syntax:

```
class A
{
 Public:
        ~A();   //destructor
};
```

Characteristics of Destructor:-

1. Destructor have the same name as that of the class they belong to preceded by ~ (tilde).
2. It do not have any return type and not even void
3. It can be virtual
4. Only one destructor can be defined in the destructor.
5. The destructor does not have any arguments
6. Destructors neither have default values nor can be overloaded.
7. Programmers cannot access addresses of constructors and destructors
8. They cannot be used in union.
9. Destructors cannot be inherited.
10. It make implicit calls to operators new and delete if memory allocation/de-allocation is needed for an object.

Example:-

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

class test
{
   public:
                test()
                {
                   cout<< " Constructor is created…" << endl;
                }

               ~test()
               {
                   Cout << "Constructor is destroyed…" <<
endl;
               }
};

void main()
{
        clrscr();
        test t;
        getch();
}
```

Output:-

Constructor is created…

Constructor is destroyed…

Difference between Constructor and Destructor:-

Constructor	Destructor
1. Constructor has the same name as class name.	1. Destructor also has the same name as class name but with tilde(~) operator.
2. Constructor is used to creates an object.	2. Destructor destroys the objects when they are no longer needed.

3. Syntax:
Class-name(arguments)
{
//body of constructor
}
	3. Syntax:
	~ class-name( )
	{
	
	}

4. Constructor is called when object (new instance) of a class is created.	4. Destructor is called object (when instance of a class) is deleted or released or de-allocated.
5. Constructor allocates the memory.	5. Destructor releases the memory.
6. Constructors can have arguments.	6. Destructor cannot have any argument.
7. Overloading of constructor is possible.	7.Overloading of destructor is not possible.
8. It cannot be virtual	8. It can be virtual.
9. It cannot be union.	9.It cannot be union.
10. Constructors cannot be inherited, though a derived class can call the constructor of base class.	10. Destructor cannot be inherited, though a derived class can call the destructor of base class.