UNIT-II

P. Madhuravani

UNIT - II

Object and Classes: Specifying a Class, Member function, Nesting of Member functions, Arrays in a Class, Memory Allocation for Objects, Static Members and Member Functions, Array of Objects, Friend Functions, Pointers to Members.

Constructors and Destructors: Parameterized Constructor, Copy Constructor, Dynamic Constructor, Destructors.

Object and Classes

Specifying a Class

- The class is basis for the OOP. The class is used to define the nature of an object, and also it is basic unit for the encapsulation.

- The class combines the data and its associated functions together. It allows the data to be hidden.

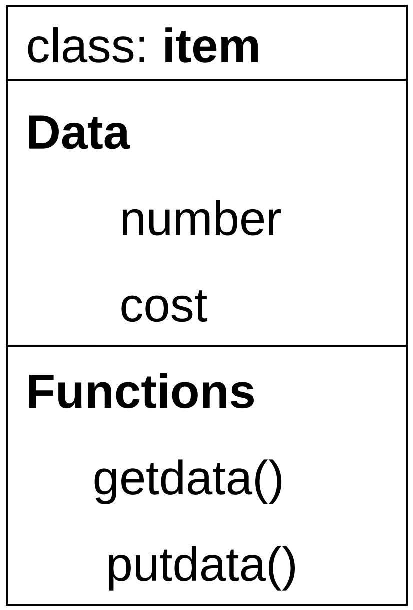
- The keyword class is used to create a class.

Class has two parts:

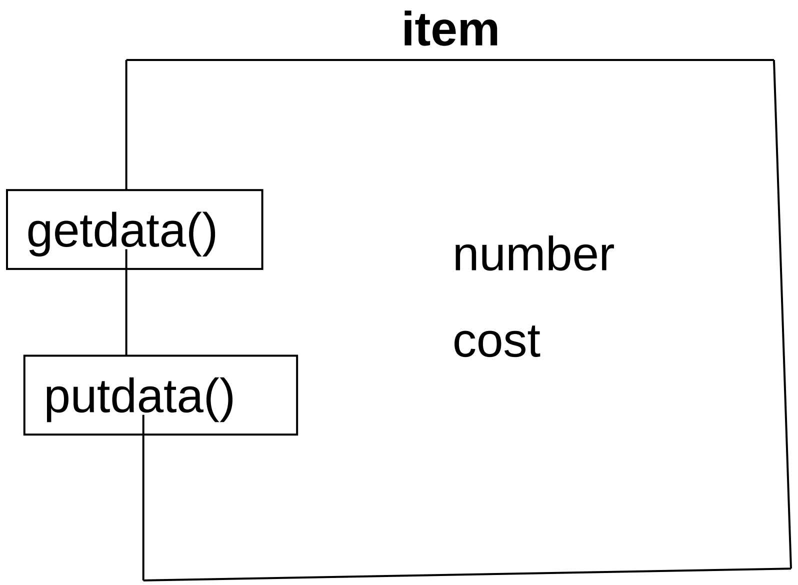
1. Data members declaration
2. Prototype of member function declarations

Note: Data members can't be initialized with in the class. They can be initialized using member functions of that class.

Representation of class



(a)



(b)

Syntax of declaring a class:

```
class class-name {
        private data and functions;
        access-specifier:
                data and functions;
        access-specifier:
                data and functions;
        // ...
        access-specifier:
                data and functions;
}object-list;
```

- Class should enclose both data declaration and function declaration part between curly parenthesis and class definition should be ended with semicolon.
- Members (data members and member functions) are grouped under access specifiers, namely **private**, **public** and **protected**, which define the visibility of members.
- The **object-list** is optional. If present, it declares objects of the class

Creating objects

- The process of creating objects of a class is called class instantiation.
- Object is instance of the class.
- Once the class is created we can create any number of objects.

Syntax1:

```
class class-name
{
        ------;
        ------;
}object-list;
```

Example1:

```
class student
{
        ------;
        ------;
}s1,s2,s3;
```

Syntax2:

```
class_name   object-name1,object-name2,object-name3;
```

Example2:

```
student s1, s2, s3;
```

Accessing class members:

- Once an object of a class has been created, then we can access the members of the class. This is achieved by using the member access operator, dot(.).

Syntax:

```
Object-Name.DataMember;

Object-Name.Memberfunction(arguments);
```

Example:

```
x.getdata(10,20);
x.number;
```

Here x.number accessing is illegal, if number is private data member of the class. Private data members are accessed only through the member functions not directly by objects.

Class vs Structure:

- Members of a class are private by default and members of struct are public by default.

```
// Program 1
#include <stdio.h>

class Test {
    int x; // x is private
};
int main()
{
  Test t;
  t.x = 20; // compiler error because x is private
  getchar();
  return 0;
}
```

```
// Program 2
#include <stdio.h>

struct Test {
    int x; // x is public
};
int main()
{
  Test t;
  t.x = 20; // works fine because x is
public
  getchar();
  return 0;
}
```

Member Functions:

We can define the function in two ways:

1. Outside the class definition 2. Inside the class definition

Outside the class definition: Use the **Scope Resolution Operator ::** to define a member function outside of the class.

:: Scope Resolution Operator tells the compiler that to which class the specified function belongs to.

Syntax:

```
Return_type class-name:: function-name(argmlist)
{
        ----
        ----
}
```

Inside the class definition: We can directly define a member function inside of the class without using Scope Resolution Operator.

//Program for implementing member functions inside of a class

```
#include<iostream.h>
#include<string.h>
class student
{
	private:
		int rollno;
		char name[30];
	public:
		void setdata(int rn, char *n)
		{
			rollno=rn;
			strcpy(name, n);
		}
		void putdata()
		{
			cout<<"RollNumber="<<rollno<<" ";
			cout<<"Name="<<name<<"\n";
		}
};
```

```
void main()
{
	student s1,s2;
	s1.setdata(2361,"RAMU");
	s2.setdata(2362,"VENKAT");
	s1.putdata();
	s2.putdata();
}
```

```
RollNumber=2361      Name=RAMU
RollNumber=2362 Name=VENKAT
```

//Program for implementing member functions outside of a class using Scope Resolution Operator.

```
#include<iostream.h>        #include<string.h>
class student
{
    private:
        int rollno;
        char name[30];
    public:
        void setdata(int rn, char *n)
        void putdata()
};
void  student :: setdata(int rn, char *n)
{
    rollno=rn;
    strcpy(name, n);
}
void student :: putdata()
{
    cout<<“RollNumber=“<<rollno<<“  ”;
    cout<<“Name=“<<name<<“\n”;
}
```

```
void main()
{
    student s1,s2;
    s1.setdata(2361,"RAMU");
    s2.setdata(2362,"VENKAT");
    s1.putdata();
    s2.putdata();
}
```

```
RollNumber=2361      Name=RAMU
RollNumber=2362      Name=VENKAT
```

B. Madhuravani

Access Specifiers

The following are the access specifiers:

1. Private 2. Public 3. Protected

Default access specifier is private.

Private:

- Private members of a class have strict access control.
- Only the member functions of the same class can access these members.
- They prevents the accidental modifications from the outside world.

Example:

```
class person
{
    private:
        int age;
        int getage();
};
```

```
int person::getage(){
  age=22;      //correct
}
```

```
void main(){
       person p1;
       p1.age=5;        // error
       p1.getage();     // error
}
```

- A private member functions is only called by the member function of the same class . Even an object cannot invoke a private function using the dot operator.

Public:

- All members declared with public can have access to outside of the class without any restrictions.

Example:

```
class person
{
  public:
        int age;
        int getage();
};

int person :: getage(){
  age=10; //correct
}
```

```
void main(){
      person p1;
      p1.age=20;//correct
      cout<<p1.age;
      p1.getage();     // correct
}
```

Protected:

- Similar to the private members and used in inheritance.
- The members which are under protected, can have access to the members of its derived class.

Example:

```
class A
{
    private:
        //members of A
    protected:
        //members of A
    public:
        //members of A
};
class B: public A        //Here class B can have access on
{                        protected data of its parent class as
    //members of B       well as public data.
};
```

Nesting of Member Functions:

- Member function of a class can be called only by an object of that class using a dot operator.
- A member function can also be called by using its name inside another member function of the same class. This is known as nesting of member function.

```
#include <iostream.h>
class set
{
int m,n;
public:
        void input(void);
        void display(void);
        void largest(void);
};
int set :: largest(void)
{
if(m >= n)
       return(m);
else
       return(n);
}
```

```
void set :: input(void)
{
cout << "Input value of m and n"<<"\n";
cin >> m>>n;
}
void set :: display(void)
{
cout << "largest value=" << largest() <<"\n";
}

int main()
{
set A;
A.input();
A.display();

return 0;
}
```

The output of program would be:

```
Input value of m and n
25 18
Largest value=25
```

Arrays within class:

- Arrays can be declared as the members of a class. The arrays can be declared as private, public or protected members of the class.

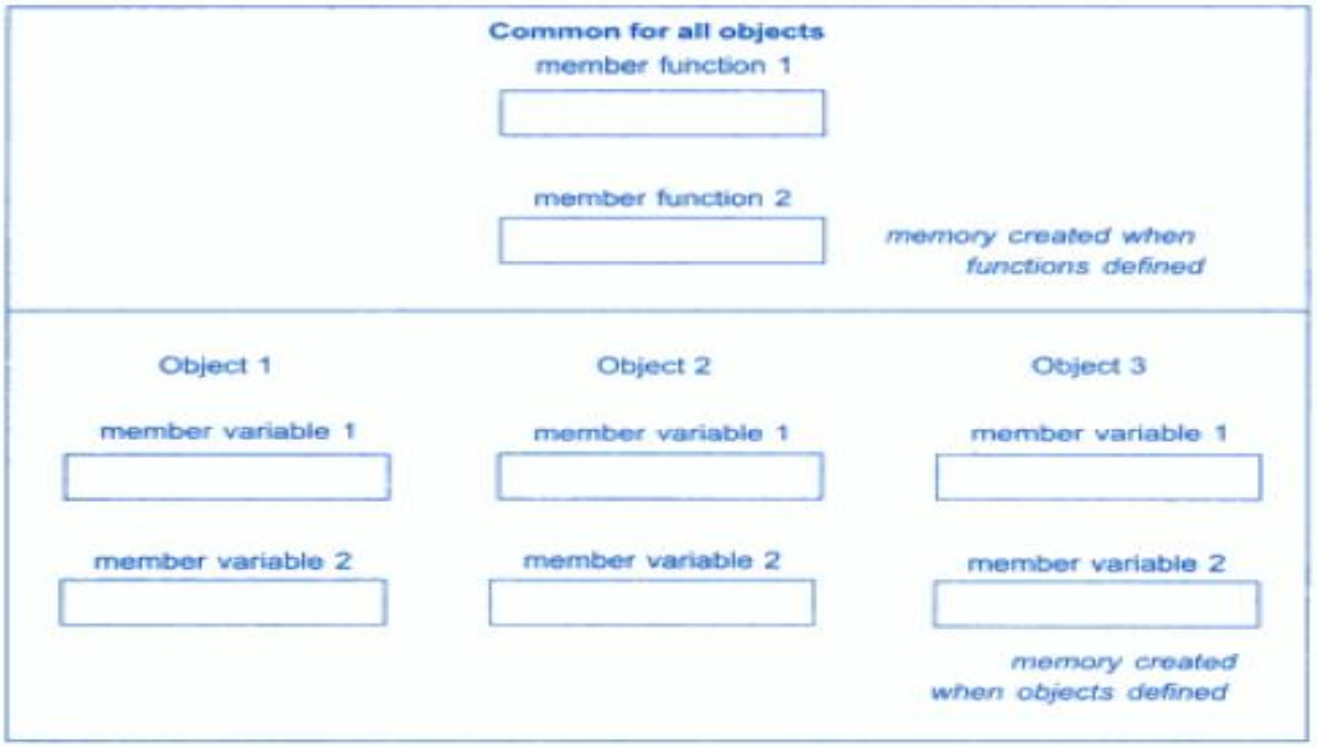
```
#include<iostream>
using namespace std;
const int size=5;
class student
{
            int roll_no;
            int marks[size];
            public:
            void getdata ();
            void tot_marks ();
} ;
void student :: getdata ()
{
            cout<<"\nEnter roll no: ";
            Cin>>roll_no;
for(int i=0; i<size; i++)
      {
            cout<<"Enter marks in
                  subject"<<(i+1)<<": ";
            cin>>marks[i] ;
      }
}
void student :: tot_marks() //calculating total
{
        int total=0;
        for(int i=0; i<size; i++)
        total+ = marks[i];
        cout<<"\n\nTotal marks "<<total;
}
int main()
{
        student stu;
        stu.getdata() ;
        stu.tot_marks() ;
        return 0;
}
```

The output of program would be:

```
Enter roll no: 101
Enter marks in subject 1: 67
Enter marks in subject 2 : 54
Enter marks in subject 3 : 68
Enter marks in subject 4 : 72
Enter marks in subject 5 : 82
Total marks = 343
```

Memory Allocation for Objects:

- Memory space for objects is allocated when they are declared and not when the class is specified.
- Member functions are created and placed in the memory space only once when they are defined as a part of a class specification.
- All the objects belonging to the same class uses the member function, no separate space is allocated for member functions when the objects are created.
- Only space for member variables is allocated separately for each object.



Static Class Members

Characteristics of static data members:

- ➢ Static class variable is efficient when a single copy of data is enough.
- ➢ It is initialized to zero when the first object of its class is created.
- ➢ Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created
- ➢ They exist when the program starts to execute and continue to exist throughout the program's entire lifetime.
- ➢ It can be public, private or protected.

Accessing static class variables:

- – **public static** variables
  - • Can also be accessed using scope resolution operator(**::**)

    Employee::count;
- – **private static** variables
  - • Can only be accessed via **public static** member function.

    Employee::getCount();

Static member function

Characteristics of static member function:

- A static function can be have access to only other static members (functions or variables) declared in the same class.
- That is, it cannot access non-static data or functions.
- A static member function can be called using the class name (instead of its objects) as follows:

  class-name :: function-name;

- No this pointer for static functions.

Example program for Static Class Variable

```
#include<iostream.h>
#include<conio.h>
class SE{
      public:
        int i;
        static int count;
};
int SE::count;
```

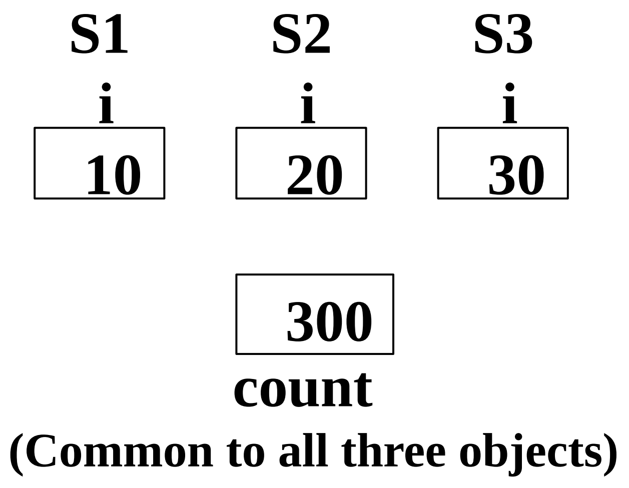
```
void main(){
        SE s1,s2,s3;
        clrscr();

        s1.i=10;
        s2.i=20;
        s3.i=30;

        s1.count=100;
        s2.count=200;
        s3.count=300;

        cout<<s1.i<<"  "<<s2.i<<"  "<<s3.i<<"\n";
        cout<<s1.count<<" "<<s2.count<<" "<<s3.count;

        getch();
}
```



```
Output:
10          20          30
300         300         300
```

Example program for Static member function

```
#include<iostream.h>
#include<conio.h>
class test {
    int code;
    static int count;
    public:
     void setcode(void) {
          code=++count;
     }
     void showcode(void) {
          cout<<"object number:"<<code<<"\n";
     }
     static void showcount(void) {
          cout<<"count:"<<count<<"\n";
     }
};
int test :: count;
```

```
count:2
count:3
object number:1
object number:2
object number:3
```

```
int main()
{
     test t1,t2;
     t1.setcode();
     t2.setcode();

     test :: showcount();

     test t3;
     t3.setcode();

     test ::showcount();

     t1.showcode();
     t2.showcode();
     t3.showcode();
     getch();
     return 0;
}
```

“this” Pointer

- Within a member function, the **this** keyword is a pointer to the current object, i.e. the object through which the function was called.
- C++ passes a **hidden this** pointer whenever a member function is called.
- Within a member function definition, there is an **implicit use of this pointer for references to data members.**
- The this pointer is a **constant** pointer.
- Every object has access to its own address through a pointer called **this.**
- A **static** member function **does not** have a **this** pointer.
- **this** pointer is not counted for calculating the size of the object. (**sizeof**)
- Member functions use the **this** pointer implicitly or explicitly
  - Implicitly when accessing members directly
  - Explicitly when using keyword **this**

```
/*Example program for this keyword*/
#include<iostream.h>
#include<conio.h>
class  basic {
        int i;
        float f;
        public:
                basic getdata(int i,float f) {
                        this->i=i;
                        this->f=f;
                        return *this;
                }
                void display() {
                        cout<<"\ni value is: "<<i;
                        cout<<"\nf value is: "<<f<<"\n\n";
                }
};
```

```
void main()
{
        basic b,b1;
        clrscr();
        b1=b.getdata(100,1.6734);

        b.display();
        b1.display();

        getch();
}
```

```
i value is: 100
f value is: 1.6734

i value is: 100
f value is: 1.6734
```

//Application of this pointer: return the object it points to

```
#include<iostream.h>
#include<string.h>
class person
{
        char name[20];
        float age;
        public:
                person(char *s,float a)
                {
                        strcpy(name,s);
                        age=a;
                }
                person & person::greater(person &x)
                {
                        if(x.age>=age)
                                return x;
                        else
                                return *this;
                }
                void display()
                {
                        cout<<"Name:" <<name<<endl;
                        cout<<"Age:" <<age<<endl;
                }
};
```

```
int main()
{
        person p1("John",37.50),p2("Ahmed",29.0),p3("Hebber",40.2
        person p=p1.greater(p3);
        cout<<"Elder person is\n";
        p.display();
        p=p1.greater(p2);
        cout<<"Elder person is\n";
        p.display();
        return 0;
}
```

```
Elder person is:
Name: Hebber
Age: 40.25
Elder person is:
Name: John
Age: 37.5
```

Array of Objects

- Like array of other user-defined data types, an array of type class can also be created. The array of type class contains the objects of the class as its individual elements. Thus, an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type.
- The syntax for declaring an array of objects is

```
class_name array_name [size] ;
```

```
/*Example */
#include<iostream>
using namespace std;
class books
{
char tit1e [30];
float price ;
public:
void getdata ();
void putdata ();
} ;
void books :: getdata ()
{
cout<<"Title:";
Cin>>title;
cout<<"Price:";
cin>>price;

}
```

```
void books :: putdata ()
{
cout<<"Title:"<<title<< "\n";
cout<<"Price:"<<price<< "\n";
const int size=3 ;
int main ()
{
books book[size] ;
for(int i=0;i<size;i++)
{
cout<<"Enter details o£ book "<<(i+1)<<"\n";
book[i].getdata();
}
for(int i=0;i<size;i++)
{
cout<<"\nBook "<<(i+l)<<"\n";
book[i].putdata() ;
}
return 0;
}
```

```
Enter detail
book 1
Title: c++
Price: 325
Enter detail
book 2
Title: DBM
Price:. 455
Enter detail
book 3
Title: Java
Price: 255
Book 1
Title: c++
Price: 325
Book 2
Title: DBM
Price: 455
Book 3
Title: Java
Price: 255
```

Friend Functions

- The functions that are declared with the keyword **friend** are known as friend functions.
- The function **declaration** should be **preceded by the keyword friend**.
- A friend function, although **not a member function of a class**, has full **access rights to the private members** of the class.
- The function **definition** should not use **friend** keyword.

Friend types:

- Friend non-member function
- Friend member function
- Friend class

- **Friend non-member function:** It is a function which does not belong to any class. Non-member function can access private members of the class by **making this function as friend** to the class.

- **Member function** of one class say X can access the private members of another class say Y by making **member function of class X as friend to class Y**.

- **Friend class: Member functions (all)** of one class say X can access the private members of another class say Y by **making class X as friend to class Y**.

To declare a function as a friend function, prefix its prototype with the keyword **friend**.

Syntax:

```
class MyClass {
        ………… . .
        friend void func1(MyClass c1, . . . );    //non-member function
        friend  void MyClass :: memfunc(. . .);       //member function
        friend  class OtherClass;                     //class as a friend class
};
```

```
//non-member function as a friend
function
class ABC{
      ……
      ……
      public:
            ……
            ……
            friend void xyz(void);
            //declaration
};
```

```
//class as a friend class
class Z{
      ……
      ……
      friend class X;
      //all member functions of X
      //are friends to Z.
            ……..
};
```

```
//member function as a
//friend function
class X{
      ……
      ……
      public:
            ……
            ……
            int fun1();
            ……..
};
class Y
      ……
      ……
      friend int X :: fun1();
      //fun1 of X is friend of Y
      ……..
};
```

Characteristics of friend function:

- It is not in the **scope of the class** to which it has been declared as **friend.**
- Since it is not in the scope of the class, it **cannot be called using the object** of that class.
- It can be **invoked like a normal function** without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to **use an object name and dot membership operator** with each member name.
- It can be declared either in **the public or the private** part of a class without affecting its meaning.
- Usually, it has the **objects as arguments.**

```
//Example program for friend
//function.
#include<conio.h>
#include<iostream.h>
class record {
    int   marks;
    public:
    void readmarks(){
            cout<<endl<<"Enter marks:";
            cin>>marks;
    }
    void writemarks() {
     cout<<"Original marks "<<marks;
    }
    friend void modify(record);
     //friend function declaration
     // under public specifier
};

void modify(record r) {
      // friend function definition
      // friend function takes object as a
      //parameter
       r.marks=r.marks+20;
   cout<<"Modified marks "<<r.marks;
}

void main()
 {
       record r;
       clrscr();

       r.readmarks();
       r.writemarks();

       modify(r);

       getch();
}
```

```
Enter marks:40

Original marks 40
Modified marks 60
```

Pointers to Members

Defining a pointer of class type

We can define pointer of class type, which can be used to point to class objects.

```
class Simple
{
        public: int a;
};
int main()
{
                Simple obj;
                Simple* ptr; // Pointer of class type
                ptr = &obj;
                cout << obj.a;
                cout << ptr->a; // Accessing member with pointer
}
```

Pointers to Members

Pointer to Data Members of class

We can use pointer to point to class's data members (Member variables).

Syntax for Declaration :

datatype class_name :: *pointer_name ;

Syntax for Assignment :

pointer_name = &class_name :: datamember_name ;

Both declaration and assignment can be done in a single statement too.

datatype class_name::*pointer_name = &class_name::datamember_name ;

Pointers to Members

Example:

```
Class A
{
        private:
                int m;
        public:
                void show();
};
```

Pointer to the member m is defined as follows:

int A::* ip = &A::m   **(**datatype class_name::*pointer_name = &class_name::datamember_name ;

where A::* means "pointer-to-member of A class"

&A:m means the "address of the m member of A class"

int *ip = &m;   //won't work

Pointers to Members

Let us assume that a is an object of A declared in a member function. We access m using the poiner ip as :

```
cout<<a.*ip;

cout<<a.m;  //same as above



ap=&a;  //ap is pointer to object a
cout<<ap->*ip; // display m
cout<<ap->m;  //same as above
```

The differencing operator ->* is used to access a member when we use pointers to both the object and the member. The dereferencing operator .* is used when the object itself is used with the member pointer.

Pointers to Members

Using with Objects

For accessing normal data members we use the dot . operator with object

and -> qith pointer to object.

But when we have a pointer to data member, we have to dereference that pointer to get what its pointing to, hence it becomes,

Object.*pointerToMember

and with pointer to object, it can be accessed by writing,

ObjectPointer->*pointerToMember

Pointers to Members

```
class Data
{
        public:
                int a;
                void print()
                {
                        cout << "a is "<< a;
                }
};
int main()
{
        Data d, *dp;
        dp = &d; // pointer to object
        int Data::*ptr=&Data::a; // pointer to data member 'a'
        d.*ptr=10;
        d.print();
        dp->*ptr=20;
        dp->print();
}
```

Output :

```
a is 10 a is 20
```

Pointers to Members

Pointer to Member Functions

Pointers can be used to point to class's Member functions.

Syntax :

return_type (class_name::*ptr_name) (argument_type) = &class_name::function_name ;

(object-name .* pointer-to-member function) (10);

(pointer-to-object ->* pointer-to-member function) (10)

Example:

```
class M
{
    int x,y;
    public:
        void set_xy(int a, int b)
        {
            x=a;
            y=b;
        }
        friend int sum(M m);
};
int sum(M m)
{
    int M ::* px = &M :: x;
    int M :: *py = &M :: y;
    M *pm = &m;
    int S = m.*px + pm->*py;
    return S;
}
```

```
int main()
{
    M n;
    void (M :: *pf) (int,int) = &M :: se_xy;
    (n.*pf) (10,20);
    cout<<"Sum="<<sum(n) <<"\n";

    M *op = &n;
    (op->*pf) (30,40);
    cout<<"Sum="<<sum(n) <<"\n";
    return 0;
}
```

OUTPUT:

```
Sum = 30

Sum = 70
```

Constructors:

- A **constructor** is a special member function whose task is to **initialize the objects of its class**.
- It is special because its **name is the same as the class name**.
- The constructor is **invoked** whenever an **object of its associated class** is created.

Syntax:

```
class Test
{
    int m, n;
    public:
          Test();
          //constructor declaration
          ----------
          ----------
};
```

```
Test :: Test()    //constructor definition
{
    m=0;
    n=0;
}

void main(){
    Test t;     //constructor is invoked
                //after creation of object t
}
```

Characteristics of Constructors:

- Constructors should be declared in the **public section**.
- They are invoked automatically when the objects are created.
- They do not have **return types**, not even void and therefore, they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- We **cannot** refer to their **address**.
- Constructors / Destructors cannot be **const**.
- They make ‘implicit calls’ to the operators new and delete when memory allocation/de-allocation is required.

Types of Constructors:

1) Default Constructor 2) Parameterized Constructor

3) Copy Constructor

Default constructor:

- Constructor without parameters is called default constructor.

```
#include<iostream.h>
class  point {
    int   a;
    public:
     point( ) {        //Default constructor
          a=1000;
     }
     void display( ){
          cout<<"a value is "<<a;
     }
};
void main( )  {
    point  p;             //constructor is invoked after creation of object p
    p.display();
}
```

Parameterized Constructor:

- Constructor which takes parameters is called Parameterized Constructor.

Example:

```
#include<iostream.h>
class A
{
        int m, n;
        public:
                A (int x, int y);        // parameterized constructor
};
A :: A (int x, int y)
{
        m=x; n=y;
        cout<<m<<n;
}
void main(){
        A obj(10,20);     or        A(10,20);
}
```

Copy constructor:

- Copy Constructor is used to create an object that is a copy of an existing object.
- Copy constructor is a member function which is used to initialize an object from another object.
- By default, the compiler generates a copy constructor for each class.

Syntax:

```
class-name (class-name &variable)
{
        -----;
        -----;
};
```

You can call or invoke copy constructor in the following way:

class obj2(obj1);

or

class obj2 = obj1;

//Example for Copy Constructor

```
#include<iostream.h>
class  point {
     int   a;
     public:
          point( )  {          //Default Constructor
               a=1000;
           }
           point(int x) {   //Parameterized Constructor
               a = x;
           }
           point(point  &p) {     //Copy Constructor
              a = p.a;
           }
          void display( )
          {
              cout<<"a value is "<<a;
          }
 };
```

```
void main( )
{
     point  p1;
     point  p2(500);
     point  p3(p1);

     p1.display();
     p2.display();
     p3.display();
}
```

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.

```
class DynamicCon
{
        int * ptr;
        public:
        DynamicCon()
        {
          ptr=new int;
         *ptr=100;
        }
        DynamicCon(int v)
        {
          ptr=new int;
         *ptr=v;
        }

         int display()
        {
          return(*ptr);
        }

 };

 void main()
 {

         DynamicCon obj1, obj2(90);
         cout<<"\nThe value of obj1's ptr  is:";
         cout<<obj1.display();
         cout<<"\nThe value of obj2's ptr is:"<<endl;
         cout<<obj2.display();
 }
```

Output:

```
The value of obj1's ptr is:100
The value of obj2's ptr is: 90
```

Destructors

- Destructors are used to de-allocate memory for a class object and its class members when the object is destroyed.
- A destructor is called for a class object when that object passes out of scope or is explicitly deleted.
- A destructor is a member function with the same name as its class prefixed by a **~** (tilde).
- A destructor takes no arguments and has no return type.
- Destructors cannot be declared const or static.
- A destructor can be declared virtual or pure virtual.
- If no user-defined destructor exists for a class, the compiler implicitly declares a destructor.

Syntax:

```
class X
{
        public:           // Constructor for class X
                X();
                ~X();     // Destructor for class X
};
```

//Example program for Default Constructor

```
include<iostream.h>
class  des {
      int   a;
      public:
        des( ) {              //Default constructor
              a=1000;
        }
        ~des( ) {             //Destructor
              cout<<”Destructor called”;
        }
        void display( ) {
              cout<<endl<<”a value is “<<a;
        }
 };
void main( ) {
        des  p;          //Constructor is invoked after creation of object p
        p.display();
}
```