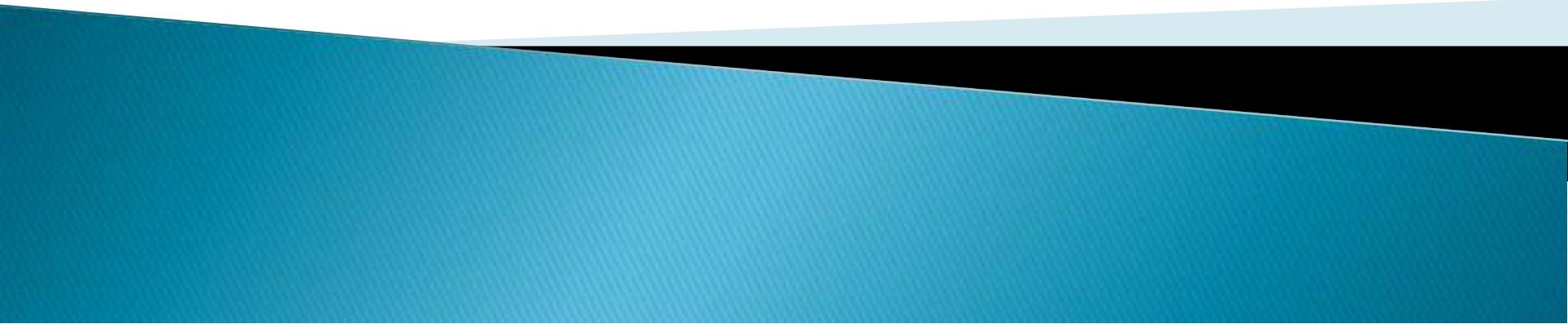


Unit_2 Class Object Constructor Desrutcor



**Classes and Objects,
Constructor and
Destructor**

- C structures revisited
- Specifying a class
- Local Classes
- Nested Classes
- Defining member functions, nesting of Member functions, private member function, making outside function inline
- Arrays within a class
- Memory allocation for objects
- Static data member
- Static member functions
- Arrays of objects
- Objects as function arguments
- Friendly functions
- Returning objects
- Const member function
- Pointer to members

- Characteristics of constructor
- Explicit constructor
- Parameterized constructor
- Multiple constructor in a class
- Constructor with default argument
- Copy constructor
- Dynamic initialization of objects
- Constructing two dimensional array
- Dynamic constructor
- MIL, Advantage of MIL
- Destructors

c structure revisited

- ← Structure is a User Defined Data Type.
- ← A Structure contains a number of data types group together.
- ← These data types may or may not be of same type.
- ← For **example**, an entity Student may have its name (string), roll number (int), marks (float).

- **Syntax of creating a structure :**

- ▶ struct [structure tag]
- ▶ {
 - ▶ member definition/declaration ; member definition/declaration ;
 - ▶ ...
 - ▶ member definition/declaration ;
- ▶ } [one or more structure variables];

C structure revisited

- **How to declare structure variables/object**

- ▶ **variable?**

- ← Before semicolon at structure terminates .
- ← At global declaration section (Global Scope) .
- ← Inside the main function (Local Scope) .

- **Syntax :**

- ▶ `struct <tag_name> <object_name>,[obj2,3,4....];`

- ▶ **Accessing Structure Members**

- ← To access any member of a structure, we use the member access operator (.).
- ← The member access operator is coded as a period between the structure variable name and the structure member that we wish to access.
- ← You would use struct keyword to define variables of structure type.

- **Syntax :**

- ▶ `structure_variable_name.structure_member_name`

C structure revisited

- ▶ struct Point
- ▶ {
 - ▶ int x, y;
- ▶ }p1; //before semicolon
- ▶ Struct Point p2 //global declaration
- ▶ void main()
- ▶ {
- ▶ struct Point p3;
- ▶ // Local Variable -> The variable p3 is declared like a normal variable
- ▶ p3.x=43;
- ▶ p3.y=65;
- ▶ struct Point p4={10,20};
- ▶ cout<<p3.x<<endl;
- ▶ cout<<p3.y<<endl;
- ▶ cout<<p4.x<<endl; cout<<p4.y<<endl;
- ▶ }

Specifying a class :

- **A class is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.**
- ◀ A C++ class is like a blueprint for an object.
 - ▶ For example: in real life, a car is an **object**.
The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.
- ◀ Attributes and methods are basically variables and functions that belongs to the class.
- ◀ These are often referred to as "class members"

- ← A Class is a user-defined data type that has data members and member functions.
- ← Data members are the data variables and member functions are the functions used to manipulate these variables together,
 - ▶ these data members and member functions define the properties and behaviour of the objects in a Class.
- ← But we cannot use the class as it is.
- ← We first have to create an object of the class to use its features.
- ← An **Object** is an instance of a Class.
- **Note:** *When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.*

- **Defining Class in C++**

- **syntax:**

```
class ClassName  
{
```

```
    access_specifier:
```

```
    // Body of the class
```

```
    //Data Members;
```

```
    //Member Functions();
```

```
};
```

- **Example :**

```
class student  
{
```

```
    public:
```

```
        int age;
```

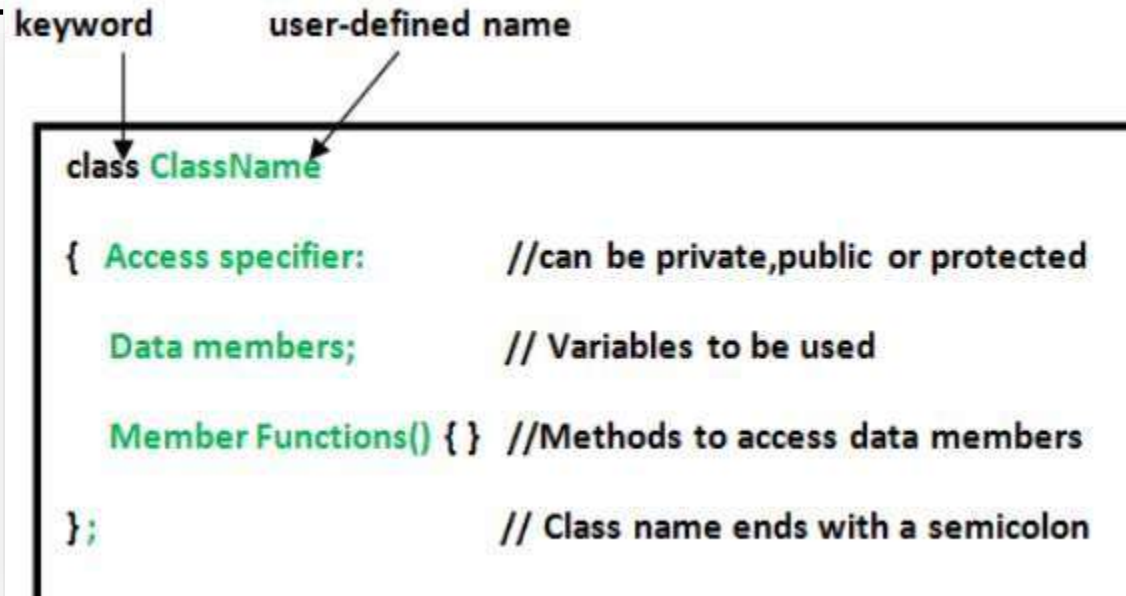
```
        void print( )
```

```
        {
```

```
            cout << "Hello";
```

```
        }
```

```
};
```



Access Specifiers :

- ← Classes have the same format as plain data structures, except that they can also include functions and have these new things called access specifiers.
- ← Access specifiers are one of the following three keywords:
 - ▶ private, public or protected.
- **The public members:**
 - ← A public member is accessible from anywhere outside the class but within a program.
- **The private members:**
 - ← A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.
 - ← By default all the members of a class would be private.
- **The protected members:**
 - ← A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

Creating object

- ← In C++, Object is a real world entity.
 - ▶ for example, chair, car, pen, mobile, laptop etc.
- ← In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.
- ← Object is a runtime entity, it is created at runtime.
- ← Object is an instance of a class. All the members of the class
 - ▶ can be accessed through object.
- ← An object is simply a variable of its type (class). Therefore
 - ▶ creating an object is much similar to declaring a variable.
- ← For example, if you want to create a variable of **int type, you would write:** `int i;`
- ← Same as you can create object of class **student we created earlier now create object of class student.**
- **Syntax :**
 - ▶ `ClassName ObjectName;`
- **Example :**
 - ▶ `students1, or student s1, s2, s3;`

Accessing Data Members and Member Functions

- ✦ The data members and member functions of the class can be accessed using the dot('.') operator with the object.
- ✦ For example, if the name of the object is *obj* and you want to access the member function with the name *printName()* then you will have to write:
 - *obj.printName()*

Example :

```
#include<iostream.h>
#include<conio.h>
class demo
{
    public:
    int age;
    void print()
    {
        cout<<age;
    }
};
void main()
{
    string a;
    clrscr();
    demo d;
    d.age=22;
    d.print();
    getch();
}
```

Local classes

- ◀ A class which is declared inside a function or block is called local class.
- A local class name can only be used in its *function and not outside it*.
- the methods of a local class must be defined *inside the class* only.
- A local class cannot have static data members but it can have static functions.

- **Syntax :**

```
➤ return_type function_name()
➤ {
    ➤ class cls_name
    ➤ {
        ➤ .....
        ➤ .....
    ➤ };

    ➤ class_name object_name; object_name.data_members_name;
      //member function call
➤ }
➤ main()
➤ {
    ➤ function_call();
➤ }
```

```
▶ #include<iostream.h>
▶ #include<conio.h>
▶ void function()
▶ {
▶ cout<<"UDF";
▶ class demo
▶ {
▶     ▶ public:
▶     ▶ void cls_fun()
▶     ▶ {
▶         ▶ cout<<"\nThis is local class";
▶     ▶ };
▶ demo d;
▶ d.cls_fun();
▶ }
▶ void main()
▶ {
▶ clrscr();
▶ function();
▶ getch();
▶ }
```

Nested class:

- ▶ A nested class is a class that is declared in another class.
- ▶ The class defined inside the class is known as inner class and the class in which a class is defined is known as outer class.
- ▶ The nested class is also a member variable of the enclosing class and has the same access rights as the other members.
- ▶ However, the member functions of the enclosing class have no special access to the members of a nested class.

- **Syntax :**

- ▶ class OuterClass
- ▶ {
 - ▶ class InnerClass
 - ▶ {
 - ▶ //Code
 - ▶ };
- ▶ };

Example :

```
class inner
{
};

#include<iostream.h>
#include<conio.h>
class outer
{
    public:
    void out_fun()
    {
        cout<<"\nouter";
    }
    class inner
    {
        public:
        void in_fun()
        {
            cout<<"\ninner class";
        }
    };
};

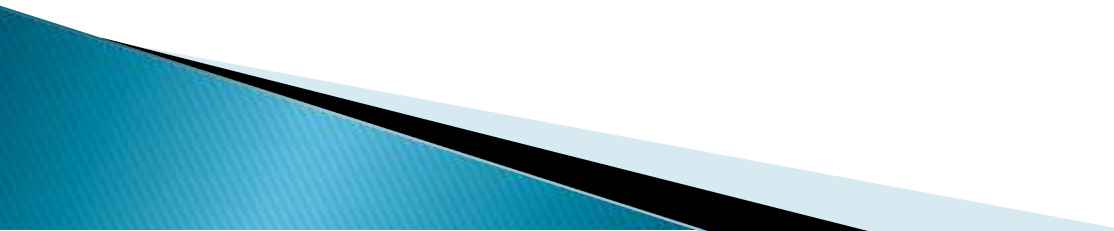
void main()
{
    clrscr();
    outer o;
    outer::inner i;
    o.out_fun();
    i.in_fun();
    getch();
}
```

Defining member function

- ← A Member function is a function that is declared as a member of a class. It is declared inside the class in any of the visibility modes like : public, private, and protected, and it can access all the data members of the class.
- ← The functions can be defined at two places:
 1. Inside the class
 2. Outside the class
- If the member function is defined inside the class definition, it can be defined directly inside the class.
- If we want to defined outside the class definition , we need to use the scope resolution operator (::) to declare the member function in C++ outside the class.
- The main aim of using the member function is to provide modularity to a program, which is generally used to improve code reusability and to make code maintainable.


Defining member function

▶ Member Function Inside the Class :

- ▶ If you want to declare the function body Inside the class.
 - ▶ There is no need to function Prototype.
 - ▶ It will take automatically from function definition.
 - ▶ A member function will be called using a dot operator (.) on a object where it will manipulate data related to that object .
- 

Syntax for Member Function inside the Class :

```
class className
{
public:
    // Member function 1
    returnType1 functionName1(arguments1,..)
    {
        /* Function Definition
           .....
           .....
        */
    }
    // Member function 2
    returnType2 functionName2(arguments1,..)
    {
        /* Function Definition
           .....
           .....
        */
    }
};
```



Example for Member Function inside the Class :

```
class data
{
    int x;
    int y;
public:
    void assign(int a,int b)
    {
        x=a;
        y=b;
    }
    void display()
    {
        cout<<x<<endl;
        cout<<y<<endl;
    }
};

void main()
{
    data d;
    d.assign(10,43);
    d.display();
}
```

Syntax for Member Function Outside the Class :

```
class className{  
public/private:  
    returnType memberFunctionName (arguments); //prototype only  
};
```

```
returnType className :: memberFunctionName (arguments)  
{  
    /* Statements  
        .....  
        .....  
        .....  
    */  
}
```

```
void main(){ className  
    object;  
  
    object.memberFunctionName(arguments);  
}
```

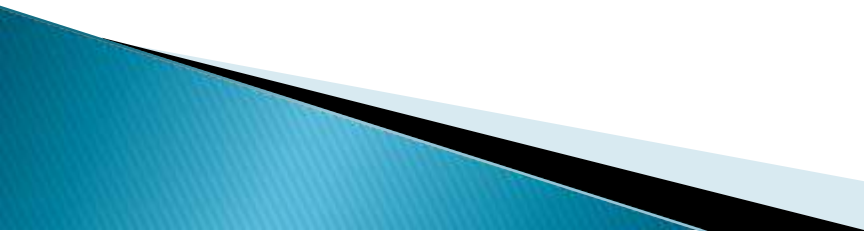
Example for Member Function

Outside the Class :

```
#include<iostream.h>
#include<conio.h>
class data {
    int x;
    int y;
public:
    void assign(int,int);
    void display();
};
void data::assign(int a,int b)
{
    x=a;
    y=b;
}
```

```
void
data::display()
{
    cout<<x<<endl;
    cout<<y<<endl;
}
void main()
{
    clrscr();
    data d;
    d.assign(10,43);
    d.display();
    getch();
}
```


Nesting of member functions :

- ← A member function of a class can be called only by an object of that class using a dot operator.
 - ← If a member function calls another member function of its class, it is known as nesting of member functions.
 - ← A member function can be called by using its name inside another member function of the same class.
 - ← When a function calls another member function of its own class, it does not need to use dot (.) operator to call it.
- 

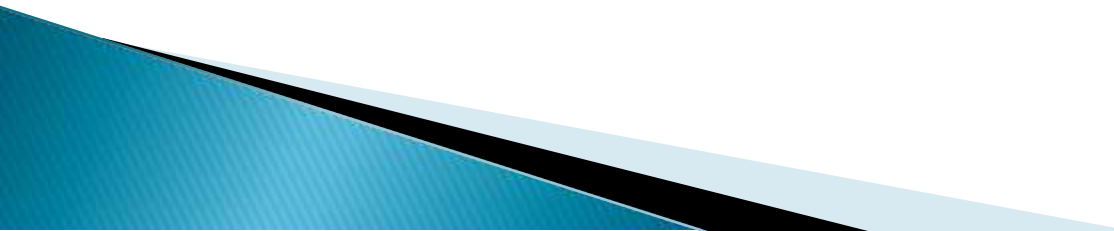
Example :

```
#include<iostream.h>
#include<conio.h>
class data
{
    void fun()
    {
        cout<<"hello function"<<endl;
    }
public:
    void display()
    {
        fun();           //this is nesting of function
    }

};

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

Private Member Functions :

- ← Generally the member variables are kept private and the functions are kept public so that the object cannot access the variables but can call the functions.
 - ← If we make a member function private, it cannot be called by its object.
 - ← So we can restrict access to a member function if we don't want to allow objects to directly call it.
 - ← The private member function can be called by its member function without using objects.
- 

Example private member function :

```
#include<iostream.h>
#include<conio.h>
class data
{
    private:
        int x;
        int y;
        void assign(int a,int b)
        {
            x=a;
            y=b;
        }
    public:
        void display()
        {
            assign(32,45);
            cout<<x<<endl;
            cout<<y<<endl;
        }
};

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

Making Outside Function inline :

- ← C++ also allows you to declare the inline functions within a class.
- ← These functions need not be explicitly defined as inline as they are, by default, treated as inline functions.
- ← All the features of inline functions also apply to these functions.
- ← However, if you want to explicitly define the function as inline, you can easily do that too.
- ← You just have to declare the function inside the class as a normal function and define it outside the class as an inline function using the inline keyword.

Example :

```
class demo_cls
{
public:
    int func(int n); // function declaration inside the
    class as inline
};
inline int demo_cls::func(int n) // defining the
    function as inline using inline keyword
    return n+100;
}
void main()
{
    demo_cls d;
    cout<<"sum is "<<d.func(101);
}
```

Arrays within a 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.

- **Syntax :**

- ▶ class class_name
- ▶ {
 - ▶ access modifier:
 - ▶ data_type array[size];
- ▶ }


- **Example :**

```
#include<iostream.h>
#include<conio.h>
class demo
{
    int arr[5];
    public :
        void value();
        void show();
};
void demo::value()
{
    cout<<"enter Value for Array
    ";
    for(int i=0;i<5;i++)
    {
        cin>>arr[i];
    }
}
```

```
void
demo::show()
{
    cout<<"values of array is ";
    for(int i=0;i<5;i++)
    {
        cout<<arr[i]<<endl;
    }
}
```

```
void main()
{
    clrscr();
    demo d;
    d.value();
    d.show();
    getch();
}
```

Memory Allocation of Objects :

- ← When the object of the class is created, memory is allocated to the object according to the member variable of the class.
 - ← But the memory space for the member function is allocated when they are defined.
 - ← So the complete memory allocation is done when an object is created.
 - ← Individual memory is allocated for each object created.
 - ← But the common memory is allocated for the member functions means no separate memory space is allocated for member function.
- 

3 Objects of Cube Class with their individual private datamember side

Cube c1
side = 5

Cube c2
side = 8

Cube c3
side = 10

**Static Data member
objectCount of Class Cube
Common to all 3 Objects**

objectCount = 3

```
graph TD; c1["Cube c1<br/>side = 5"] --> oc(["objectCount = 3"]); c2["Cube c2<br/>side = 8"] --> oc; c3["Cube c3<br/>side = 10"] --> oc; oc --- text["Static Data member<br/>objectCount of Class Cube<br/>Common to all 3 Objects"]
```

Static Data Member

- ← Static data members are class members that are declared using **static** keywords.
- ← A static member has certain special characteristics which are as follows:
- ← Static variable was initialized with *zero* value when object is created first time.
- ← 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.
- ← It is initialized before any object of this class is created, even before the main starts outside the class itself.
- ← It is visible can be controlled with the class access specifiers.
- ← Its lifetime is the entire program.
- ← The static variable is connected with all the object of class that why we can say the static variable is the *CLASS VARIABLE* in OOP.

- **Syntax**

- ▶ className
 - ▶ {
 - ▶ **static** data_type data_member_name;
 - ▶ }

Example :

```
#include<iostream.h>
#include<conio.h>
class demo
{
    static int a;
    public:
    void fun()
    {
        a++;
        cout<<"\nvalue of a is \t"<<a;
    }
};
int demo::a;
void main()
{
    demo d1,d2,d3;
    clrscr();
    d1.fun();
    d2.fun();
    d3.fun();
    getch();
}
```

Static Member Function :

- ← Static Member Function in a class is the function that is declared as static
- ← A static member function is independent of any object of the class.
- ← A static member function can be called even if no objects of the class exist.
- ← A static member function can also be accessed using the class name through the scope resolution operator.
- ← A static member function can access static data members and static member functions inside or outside of the class.
- ← Static member functions have a scope inside the class and cannot access the current object pointer.
- ← You can also use a static member function to determine how many objects of the class have been created.
- **The reason we need Static member function:**
- ← Static members are frequently used to store information that is shared by all objects in a class.
- ← For instance, you may keep track of the quantity of newly generated objects of a specific class type using a static data member as a counter.
- ← This static data member can be increased each time an object is generated to keep track of the overall number of objects.

Static Member Function :

```
#include<iostream.h>
#include<conio.h>
class demo
{
    public:
    static int a;
    int b;
    void fun()
    {
        cout<<"\nvalue of static a is \t"<<a;
        cout<<"\nvalue of normal b is \t"<<b;
    }
    static void f()
    {
        demo ds;
        cout<<"\nf()\nvalue of static a : "<<a
        <<endl<<"value of normal b : "<<ds.b
        <<endl;
        //a is static member
        //b is non-member,
        //to use it we need to use object of class }
    }
};
int demo::a;
//int demo::a=10;
```

```
void main()
{
    demo obj;
    clrscr();
    //assign value of data member
    obj.b=20;//normal data member
    //obj.a=29;//static data member
    //demo::a=10;//static data member

    //member function calling
    cout<<"\ncall normal member FUN";
    obj.fun();

    cout<<"\ncall static membern fun without
    object\n";
    demo::f();

    cout<<"\nstatic member call with object\n";
    obj.f();

    getch();
}
```


Arrays of Object :

- ◀ In C++, an **array of objects** is a collection of objects of the same class type that are stored in contiguous memory locations.
- ◀ Since each item in the array is an instance of the class, each one's member variables can have a unique value.
- ◀ This makes it possible to manage and handle numerous objects by storing them in a single data structure and giving them similar properties and behaviours.
- ◀ We can think of array of objects as a single variable that can hold multiple values.
- ◀ Each value is stored in a separate element of the array, and each element can be accessed by its index.
- ◀ Arrays in C++ are typically defined using square brackets [] after the type.
- ◀ The index of the array, which ranges from 0 to $n - 1$, can be used to access each element.

- ▶ `class className`
- ▶ `{`
 - ▶ `//variables and functions`
- ▶ `};`
- ▶ `className arrayObjectName[arraySize];`

- **className** is the name of the class that the object belong to.
- **arrayName** is the name of the array of objects.
- **arraySize** is the number of objects in the array or the size of array, specified as a constant expression

Example :

```
#include<iostream
```

```
.h>
```

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

```
class stud
```

```
{
```

```
    int roll;
```

```
    char name[30];
```

```
    public:
```

```
    void get_data()
```

```
    {
```

```
        cout<<"Enter Roll Number : ";
```

```
        cin>>roll;
```

```
        cout<<"Enter Name : ";
```

```
        cin>>name;
```

```
    }
```

```
    void show_data()
```

```
    {
```

```
        cout<<endl<<"Roll number : "
```

```
        <<roll;
```

```
        cout<<endl<<"Name : "<<name;
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    clrscr();
```

```
    stud obj[2];
```

```
    for(int i=0;i<2;i++)
```

```
    {
```

```
        obj[i].get_data();
```

```
    }
```

```
    for(int j=0;j<2;j++)
```

```
    {
```

```
        obj[j].show_data();
```

```
    }
```

```
    getch();
```

```
}
```

Object as Function Argument :

- ← We have seen examples of member functions having arguments. Just like any other normal variables, we can also pass object as function arguments.
 - ▶ ▫ *A copy of the entire object is passed to the function.*
 - ▶ ▫ *Only the address of the object is transferred to the function.*
- ← As the objects are the variables of type class, you have to specify the class name as the type of the object arguments.
- ← In previous chapter, we discussed about call by value and call by reference functions.
- ← The same concept applies to the functions having objects as arguments.
- ← If we pass address of the object to the function it is called by reference. So any changes made on the object will also affect the passing object values.
- ← But if you pass object normally it is called by value. So the changes made on the object will not reflect to the original object.

Example :

```
#include<iostream
.h>
#include<conio.h>
class demo_cls
{
    int a;
    public:
    void data(int);
    void sum(demo_cls,demo_cls);
};

void demo_cls::sum(demo_cls
    a_obj1,demo_cls a_obj2)
{
    cout<<"Sum of 2 object is :
    "<<a_obj1.a+a_obj2.a;
}
```

```
void demo_cls::data(int x)
{
    a=x;
}

void main()
{
    clrscr();
    demo_cls obj1,obj2,obj3;
    obj1.data(10);
    obj2.data(20);

    obj3.sum(obj1,obj2);
    getch();
}
```

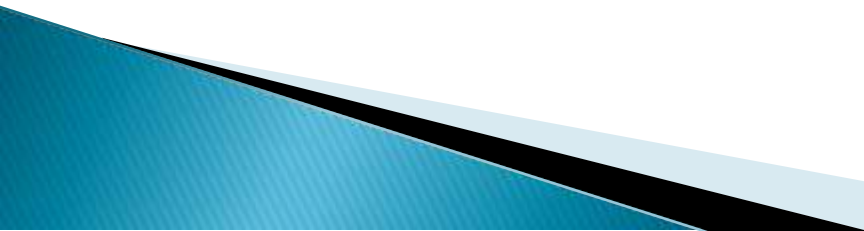
Friendly function in C++ :

- ✦ Normally, the private members cannot be accessed by external functions.
- ✦ Means a function which is not a member function of the class
- ✦ cannot have access to the private member (variable and function) of the class.
- ✦ C++ introduces a kind of functions known as friend functions which behaves like
 - ▶ friend of the class.
- ✦ We can define a function friendly to one or more classes allowing the function to access the public as well as *private* / *protected* member of all the class to which it is declared as friend.

- By using the keyword **friend** compiler knows the given function is a friend
- ▶ For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.
- **Declaration of friend function :**
 - ▶ **class** class_name
 - ▶ {
 - ▶ **friend** data_type function_name(argument/s) //
syntax of friend function.
 - ▶ };

- In the above declaration, the friend function is preceded by the keyword friend.
- The function can be defined anywhere in the program like a normal C++
- ▶ The function definition *does not use* either the keyword **friend** or **scope resolution operator**.

Characteristics of friend function

- ← The function is not in the scope of the class to which it has been declared as a friend.
 - ← It cannot be called using the object as it is not in the scope of that class.
 - ← It can be invoked like a normal function without using the object(UDF).
 - ← It cannot access the member names directly and has to use an object name and **dot** membership operator with the member name.
(obj_name.datamember)
 - ← It can be declared either in the private or the public part.
- 

Example :

```
▶ #include<iostream.h>
#include<conio.h> class
Point
▶ {
▶ int x;
  int y;
  public:
▶ friend void sum_fun(Point);
▶ void add_data(int x1 = 0, int y1 = 0)
▶ {
▶ x = x1;
▶ y = y1;
▶ }
▶ void display()
▶ {
▶ cout<<"x = "<< x <<"\n";
  cout<<"y = "<< y <<"\n";
▶ }
▶ };
```

```
void sum_fun(Point obj1)
{
    int s;
    s=obj1.x+obj1.y;
    cout<<"Sum of 2 numbers using friend
        function : "<<S<<endl;
}
void main()
{
  clrscr();
  Point p1;

  p1.add_data(5,3);

  cout<<"Point 1\n";
  p1.display();

  cout<<"The sum of the two points is:\n";
  p1.display();

  sum_fun(p1);
  getch();
}
```

Const member function

- Constant member functions are those functions that are *denied permission to change / modify* the values of the data members of their class.
- To make a member function constant, the keyword `const` is appended to the function prototype and also to the function definition header.
- Like member functions and member function arguments, the objects of a class can also be declared as `const`.
- An object declared as `const` cannot be modified and hence,
 - can invoke only `const` member functions as these functions ensure not to modify the object.
- A `const` object can be created by prefixing the `const` keyword to the object declaration.
- Any attempt to change the data member of `const` objects results in a compile-time error.

- *The const member function can be defined in three ways:*

- 1. For function definition within the class declaration :**

- **Syntax :**

- ▶ return_type function_name() const
- ▶ {
 - ▶ //function body
- ▶ }

- **Example:**

- ▶ int get_data() const
- ▶ {
 - ▶ //function body

- 2. For function definition outside the class.**


- **Syntax :**

- ▶ return_type class_name::function_name() const
- ▶ {
 - ▶ //function body
- ▶ }

- **Example:**

- ▶ int Demo :: get_data() const
- ▶ {
- ▶ }

- **Important Points :**

- ← When a function is declared as const, it can be called on any type of object, const object as well as non-const objects.
 - ← Whenever an object is declared as const, it needs to be initialized at the time of declaration. however, the object initialization while declaring is possible only with the help of constructors.
 - ← A function becomes const when the const keyword is used in the function's declaration. The idea of const functions is not to allow them to modify the object on which they are called.
 - ← It is recommended practice to make as many functions const as possible so that accidental changes to objects are avoided.
- 

Example const member as outside of the class :

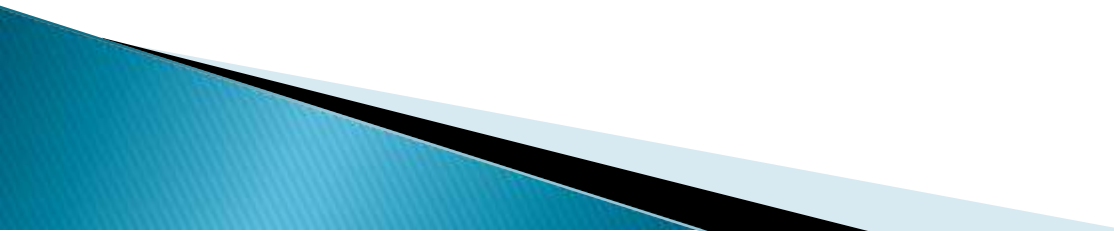
```
#include<iostream.h>
#include<conio.h>
class demo
{
    int a;
public:
    void data(int x)
    {
        a=x;
    }
    void fun() const;
};
void demo::fun() const
{
```

```
int y=10;
y=230;
```

```
    cout<<"data member"<<a;
    cout<<endl<<y;
}

void main()
{
    clrscr();
    demo obj;
    obj.data(10);
    obj.fun();
    getch();
}
```

Pointer to members :

- ← Like ***normal variables***, we can also create pointer to member variable also.
 - ← For normal variables, we can create pointer using *operator and the address of a variable can be obtained by applying &operator.
 - ← In case of ***member variables***, we can create pointer using ::*operator and the address of a variable can be got using &operator after the variable name followed by class name and scope resolution operator.
- 

- **Example :** *Pointer and variable address*
 - ▶ `int a;`//Normal variable **a**
 - ▶ `int *p=&a;`//Pointer **p stores address of a**
- **Example :** *The same code for member variable:*
 - ▶ `class test`
 - ▶ `{`
 - ▶ `int a;`
 - ▶ `};`
 - ▶ `main(){`
 - ▶ `int test::*p=&test::a; //(Normal)int *p=&a;`
 - ▶ `}`

- ← `int test::*p=&test::a; //(Normal)int *p=&a;`
- ← In The above example we have to use class name and scope resolution operator before ***sign**
- ← To specify pointer variable name (**test::*p**) and same to specify the address between **&sing** and member variable name.
- ← If we have created pointer to object, we can access the member by using dot pointer sing (**.***) instead of dot (**.**) operator.

Function Pointer to Member Function :

- function pointers enable users to treat functions as objects.
- They provide a way to pass functions as arguments to other functions.
- A function pointer to a member function is a pointer that points to a non-static member function of a class.
- A function pointer to a member function is a pointer that can refer to a member function of a specific class.
- **Syntax :**
- `return_type (ClassName::*pointer_name)(argument_types) = &ClassName::member_function;`
- **return_type:** is the return type of the member function of the class.
- **ClassName:** is the name of the class to which the member function belongs.
- ***pointer_name:** is the name of the function pointer variable.
- **argument_types:** are the types of the arguments accepted by the member function.
- **&ClassName::member_function:** is the address of the member function being assigned to the function pointer.

- **Example of function pointer to member function & pointer to member :**

```
#include<iostream.h>
#include<conio.h>
class demo
{
public :
    int x;
    void fun(int a)
    {
        cout<<"a : "<<a;
    }
};
void main()
{ //::*
  //pointer to member
  int demo::*p=&demo::x;
  //it is similar to int *p=&x;
```

```
//int for which type of variable
//demo for scope of *p
void (demo::*ptr_var_fun)
(int)=&demo::fun;
```

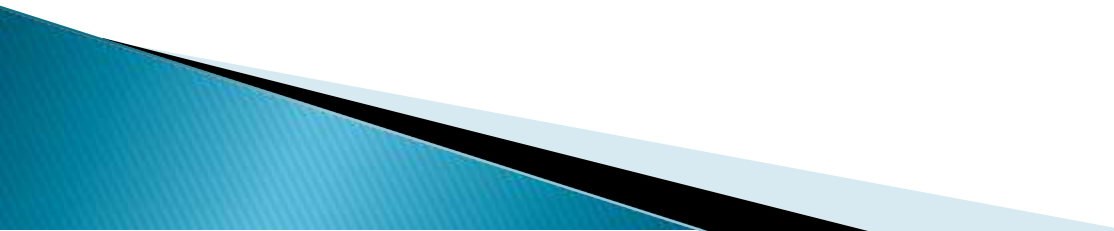
demo obj;//object of class

```
//assign value of data member x from *p
obj.*p=10;
cout<<"\nvalue of x data member :
"<<obj.*p;
```

```
//to call member function by using
*variable ptr_var_fun
(obj.*ptr_var_fun)(23);
//it call function fun with value of
parameter a is 23
```


getch();

Assignment Questions

1. What is member function? Explain private member function in detail.
 2. Explain way's to define a member functions in c++ (inside/outside).
 3. Write a note on making a outside function inline.
 4. Write a sort note on memory allocation for objects.
 5. Explain static data member & static member function
▶ in detail.
 6. Explain array of objects.
 7. Explain friend function in detail.
 8. Explain const member function in detail.
- 

UNIT 2 PART 2

Constructor & destructor

- ← Characteristics of constructor
 - ← Explicit constructor
 - ← Parameterized constructor
 - ← Multiple constructor in a class
 - ← Constructor with default argument
 - ← Copy constructor
 - ← Dynamic initialization of objects
 - ← Constructing two dimensional array
 - ← Dynamic constructor
 - ← MIL, Advantage of MIL
 - ← Destructors
- 

Constructor :

- ✦ In C++, constructor is a special method which is invoked **automatically at the time of object creation.**
- ✦ It is used to initialize the data members of new object.
- ✦ The constructor in C++ has the **same name as class name.**
- ✦ A particular procedure called a constructor is called automatically when an object is created in C++.
- ✦ It is employed to create the data members of new things.
- ✦ The class name also serves as the constructor name.
- ✦ When an object is completed, the constructor is called.
- ✦ Because it creates the values or gives data for the thing, it is known as a constructor.

• **Syntax :**

▶ `<class-name> () {`

▶ `...`

▶ `}`

Characteristics of Constructor

1. The constructor has the same name as the class it belongs to.
2. Although it is possible, constructors are typically declared in the class's public section. However, this is not a must.
3. Because constructors don't return values, they lack a return type.
4. When we create a class object, the constructor is immediately invoked.
5. Overloaded constructors are possible.
6. Declaring a constructor virtual is not permitted.
7. One cannot inherit a constructor.
8. Constructor addresses cannot be referenced to.
9. When allocating memory, the constructor makes implicit calls to the new and delete operators.

Types of Constructor :

1. Default constructor
2. Parameterized constructor

- **1. Default Constructor :**

- A constructor which has no argument is known as default constructor.
- It is invoked at the time of creating object.

- **Syntax :**

- ▶ `<class-name> () {`
- ▶ `...`
- ▶ `}`

▶ **Example of Default constructor :**

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

```
▶ #include<conio.h>
```

```
▶ class demo
```

```
▶ {
```

```
▶ int a;
```

```
▶ public: demo()
```

```
▶ {
```

```
▶     ▶ a=10;
```

```
▶ }
```

```
▶ void show()
```

```
▶ {
```

```
▶     ▶ cout<<a;
```

```
▶ }
```

```
▶ };
```

```
▶ void main()
```

```
▶ {
```

```
▶ demo d;
```

```
▶ d.show();
```

```
▶ }
```

Parameterized Constructor :

- ← A constructor which has parameters is called parameterized constructor.
- ← It is used to provide different values to distinct objects.
- ← It is invoked at the time of creating object.

- **Syntax :**

- ▶ `<class-name> (arguments...){`
- ▶ `...`
- ▶ `}`

```
▶ #include<iostream.h>
▶ #include<conio.h>
▶ class demo
▶ {
▶ int a;
▶ public: demo(int x)
▶ {
▶     ▶ a=x;
▶ }
▶ void show()
▶ {
▶     ▶ cout<<endl<<a;
▶ }
▶ };
▶ void main()
▶ {
▶ demo d(10);
▶ demo d1(20);
▶ d.show();
▶ d1.show();
▶ }
```

Explicit Constructor :

- ◀ When a constructor has been parameterized we must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways.
- ◀ by calling the constructor explicitly by calling.
- **Example :**
 - ▶ `class_nm obj1(10,20); //implicit call`
 - ▶ `class_nm obj1 = class_nm(10,20); //explicit call`

```
▶ class demo
▶ {
▶ int a;
▶ public:
▶ demo()
▶ {a=0}
▶ demo(int x)
▶ {
▶     ▶ a=x;
▶ }
▶ void show()
▶ {
▶     ▶ cout<<endl<<a;
▶ }
▶ };
```

```
void main()
```

```
{
    clrscr();
    int data;
    demo d;
    d.show();
}
```

```
d=demo(100);
//explicit calling
d.show();
```

```
demo d1(10);
//implicit calling
d1.show();
    getch();
}
```

Constructor with Default Argument:

- ◀ A constructor with default argument is a one type of parameterized but the difference is the parameter has a default value assigned.
- ◀ When the object is created at that time if we don't pass any value then it will consider a default value of that parameter.

- **Syntax :**

- ▶ `<class-name> (data_type arg_nm = default_value){`
- ▶ `...`
- ▶ `}`

Example constructor with default argument

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

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

```
class demo
```

```
{
```

```
    int a;
```

```
    public:
```

```
    demo(int x=0)
```

```
    {
```

```
        a=x;
```

```
    }
```

```
    void show()
```

```
    {
```

```
        cout<<endl<<a;
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    demo d;
```

```
    demo d1(78);
```

```
    d.show();
```

```
    d1.show();
```

```
}
```

Multiple constructor in a class (Constructor Overloading) :

- ← A class can have multiple constructors that assign the fields in different ways.
- ← Sometimes it's beneficial to specify every aspect of an object's data by assigning parameters to the fields, but other times it might be appropriate to define only one or a few.
- ← Constructors Overloading are used to increase the flexibility of a class by having more number of constructor for a single class.
- ← By have more than one way of initializing objects can be done using overloading constructors.
- ← Compiler differentiates which constructor is to called depending upon number of parameters and their sequence of data type.

- **Syntax :**

- ▶ `class name()`

- ▶ `{`

- ▶ `.....`

- ▶ `}`

- ▶ `Class name(argument1)`

- ▶ `{`

- ▶ `.....`

- ▶ `}`

- ▶ `Class name(argument1,argument2)`

- ▶ `{`

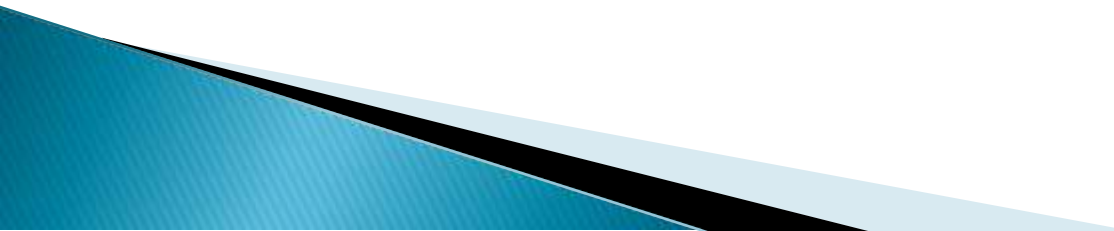
- `.....`

- ▶ `}`

Example of Multiple constructor :

```
#include<iostream.h>
#include<conio.h> class demo
{
    int a; public:
    demo(int x)
    {
        a=x;
    }
    demo(int x,int y)
    {
        a=x+y;
    }
    void show()
    {
        cout<<endl<<a
    }
};
void main()
{
    demo d(10); demo
    d1(20,34); d.show();
    d1.show();
}
```

Copy Constructor :

- ← A copy constructor is a constructor which is used to create a new object from an existing object.
 - ← This type of constructor takes reference to an object as argument and initializes the member variables of its class with the values of the specified object.
- 


Example for copy constructor :

```
class sum
{
    int a;int b;
    public:
    sum(int x, int y)
    {
        a=x;b=y;
    }
    sum(sum&s)
    {
        a=s.a;
        b=s.b;
    }
}
```

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

```
void main( )
{
    sum a(5,55);
    sum b(a);
    a.display();
    b.display();
}
```

Dynamic initialization of objects :

- ◀ We can provide the initial values for an object dynamically at runtime.
 - ◀ This is known as the dynamic initialization of objects.
 - ◀ In this concept we can get the values from the user at runtime and these values can be passed to the constructor to build the object.
 - ◀ It can be achieved by using constructors and by passing parameters to the constructors.
 - ◀ This comes in really handy when there are multiple constructors of the same class with different inputs.
- 

Dynamic initialization of objects :

```
#include<iostream.h>
#include<conio.h>
class demo
{
    int a;
    public:
    demo(){}
    demo(int x)
    {
        a=x;
    }
    void show()
    {
        cout<<endl<<a;
    }
};
void main()
```

```
clrscr();
int data;
demo o1,o2;
cout<<"Enter integer value :";
cin>>data;
o1=demo(data);
cout<<"Enter integer value :";
cin>>data;
o2=demo(data);
o1.show();
o2.show();
getch();
```

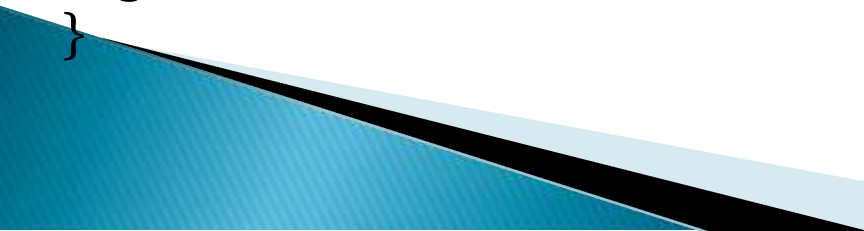
MIL : (Member Initialization List)

- ◀ MIL is stands for Member Initialization List
- ◀ The MIL is a way by which we can initialize member
 - ▶ variables of class before the constructor body executes.
- ◀ Initializer List is used in initializing the data members of a
 - ▶ class.
- ◀ The list of members to be initialized is indicated with
 - ▶ constructor as a comma-separated list followed by a colon.
- **Syntax :**
 - ▶ Constructor(Argument List) : Data_member1(argument1) ,
 - ▶ Data_member2(argument2)
 - ▶ {
 - ▶ Body of constructor
 - ▶ }

Example of MIL :

```
class demo
{
    int a,b;
    public:
    demo(int x,int y):a(x),b(y)
    {
    }
    void show()
    {
        cout<<endl<<a<<endl<<b;
    }
};

void main()
{
    clrscr();
    demo d(10,20);
    d.show();
    getch();
}
```



Advantage of MIL :

1. **Initialization Order:** They allow you to control the initialization order of class members, which can be crucial when members depend on each other.
2. **Const and Reference Members:** They enable the initialization of const and reference members, which must be initialized upon object creation.
3. **Avoiding Default Constructors:** You can directly initialize members with specific values, bypassing the need for default constructors.
4. **Clarity:** They enhance code readability by making the initialization explicit, showing how each member is set up right at the point of declaration.
5. **Performance:** They can lead to better performance by avoiding unnecessary default constructions followed by assignments.
6. **Consistency:** Using member initializers can help ensure that members are always initialized in a consistent manner, reducing the chances of uninitialized members.
7. **Complex Types:** They simplify the initialization of complex types or objects, allowing for more straightforward syntax and potentially reducing the need for multiple constructors.

Constructing Two Dimensional Array

- ← A class can have multidimensional arrays as data members.
- ← Their size can be either statically defined or dynamically varied during runtime.



Example of 2D array :

```
class demo
```

```
{
    int r,c,arr;
    public:
    demo(int row,int col);
    void show( );
    void data( );
};
demo::demo(int row,int col) {
    r=row; c=col;
    //assign memory for array
    arr=new int[r,c];
}
```

```
void demo::data( ) {
    for(int i=0;i<r;i++)
    {
        for(int j=0;j<c;j++)
        {
            cin>>arr[i][j];
        }
        cout<<endl;
    }
}
```

```
void demo::show()
```

```
{
    for(int i=0;i<r;i++)
    {
        for(int j=0;j<c;j++)
        {
            cout<<" "<<arr[i][j];
        }
        cout<<endl;
    }
}

void main()
{
    clrscr();
    int d1,d2;
    cout<<"Enter Size for dimension 1(row)";
    cin>>d1;
    cout<<"Enter Size for dimension 2(col)";
    cin>>d2;

    demo d(d1,d2);//d1 row , d2 col
    d.data();
    d.show();
    getch();
}
```

Destructor

- ◀ A destructor is also a special kind of member function which is used to **destroy the object** created by constructor.
- ◀ It is special because like constructor, it has also same name as the class name.
- ◀ The destructor is written by specifying a tilde(~) sign before its name.

- **Example :**


- ▶ ~Test()
 - ▶ {
 - ▶ // Code...
 - ▶ }

- **Note:** *Objects are destroyed in the reverse order of their creation.*

Characteristics of a Destructor :

1. A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.
2. Destructor has the same name as their class name preceded by a tilde (~) symbol.
3. It is not possible to define more than one destructor.
4. The destructor is only one way to destroy the object created by the constructor. Hence, destructor cannot be overloaded.
5. It cannot be declared static or const.
6. Destructor neither requires any argument nor returns any value.
7. It is automatically called when an object goes out of scope.
8. Destructor release memory space occupied by the objects created by the constructor.
9. In destructor, objects are destroyed in the reverse of an object creation.

❖ Importance of Destructor :

- When we use constructors to create object, it allocates memory to those objects.
 - Now as the **new objects are created more and more memory is allocated to the object.**
 - At some point these constructors may not be in use, but still they have occupied some memory. But in some case memory is very important so we have to take care about memory management.
 - In C++, destructors are the solution to this problem.
 - The destructor destroys the object by releasing the memory allocated by the constructor.
 - If in the constructor, the memory is allocated by **new** keyword, it should be deleted by **delete** keyword in destructor.
- 

```
class demo
{
    int a,b;
public:
    demo(int x,int y)
    {
        a=x;
        b=y;
    }
    void show()
    {
        cout<<endl<<a<<endl<<b;
    }
    ~demo()
    {
        cout<<"\ndestructor";
    }
};

void main()
{
    clrscr();
    demo d(77,7);
    d.show();
    getch();
}
```