UNIT-II: Classes and Objects &Constructors and Destructor

Classes in C++-Declaring Objects- Access Specifiers and their Scope- Defining Member Function-Overloading Member Function- Nested class, Constructors and Destructors , Introduction-Constructors and Destructor- Characteristics of Constructor and Destructor-Application with Constructor- Constructor with Arguments (parameterized Constructor- Destructor's) - Anonymous Objects.

LIMITATION OF C STRUCTURES

Struct data type can't be treated as the built in type.

Example:

```
struct complex
{
int real;
int img;
} c1, c2, c3;
```

c3= c1+c2; //Illegal in C but it is legal in C++ using operator overloading concept.

Structures do not permit data hiding. Structure members can be directly accessed by structure variables by any function any where.

Example:

```
complex c1;
c1.real= 20; c1.img=70;
```

C++ incorporates all the features of structure while avoiding the drawbacks in new user defined data types called Class. Classes can hold both data and the functions. Class members are private by default.

Classes in C++

A class is a way to bind the data and its associated functions together. It makes a data type. Classes represent real world entities that have both data type properties (characteristics) and associated operations (behavior). The basic mechanism is to provide data encapsulation. The class has a mechanism to prevent direct access to its members, which is the central idea of OOP.

The syntax of a class declaration is shown below :

Class name_of _class
{
private :
variable declaration; // data member
Function declaration; // Member Function
protected:
Variable declaration;
Function declaration;
public:
variable declaration;
Function declaration;
};
	Private members can be accessed only from within the class.
	
	Protected members can be accessed by own class and its derived classes.
	
	Public members can be accessed from outside the class also.

Class is a keyword. Declaration of a class is enclosed with curly braces and terminated with a semicolon. The variables and functions can be declared in three sections such as private, public and protected. These keywords are terminated with colon (:).

By default the data members and member function of a class are private. There are two types of class members

1. Data Members
2. Member functions

The variables which are declared inside the class are data members. The functions which are declared or defined inside the class are member functions. Normally Member functions are used to access the data members of a class.

Example for class declaration:

```
class item
{
private:
    int number; float cost;
public:
void getdata(int a, float b); void putdata(void);
};
```

Class Scope/ Member function definition:

Member Functions can be defined in two places:

1. Inside the Class
2. Outside the Class

1. Defining member function inside the class:

Replace the function declaration by the actual function definition inside the class. When a function is defined inside the class, is treated as an **inline function**. Normally only small functions are defined inside the class definition.

Example:-

```
class item
{
int number;
float cost;
public:
void getdata(int a,float b)
{
number=a;
cost=b;
}

void putdata()
{
cout<<number; cout<<cost;
}

};
```

2. Defining member function outside the class (using Scope resolution operator ::)

Member functions that are declared inside the class have to be defined outside the class. The difference between member function and a normal function is that member function incorporates a membership identity label in the header. This label tells the compiler which class the function belongs to.

Syntax:

```
returnType className::functionName(arguments)
{
        function body;
}
```

Example:

```
class item
{
int number; float cost;
public:
void getdata(int a,float b); void putdata();
};
```

```
void item::getdata(int a, float b)
{
number=a; cost=b;
}
void item::putdata()
{
cout<<number; cout<<cost;
}
```

Creating Objects:

Once a class has been declared, we can create any number of variables of that type using that class name.

Example:

```
class item
{
int number; //data member float cost;
public:
void getdata(int a , float b); //member function
void putdata();
};

item x; //single object
item x[5]; // array of objects
// At the time of definition also we can create objects.
class item
{
.....
.....
}x, z[20];
```

Accessing Class Members:

To access member of a class, dot operator is used.

Syntax:

```
Object_name.data-member
ObjectName.functionName(arguments);
```

Example:

```
x.getdata(100,75.5);    x.putdata();
x.number=20;                //will not work. Because number private member.
```

Memory Allocation for Objects:

• Each object has its own separate data items and memories are allocated when the objects are created.

• Member function are created and put in the memory only once when class are defined.

//C++ program to read and display the details of an item using class

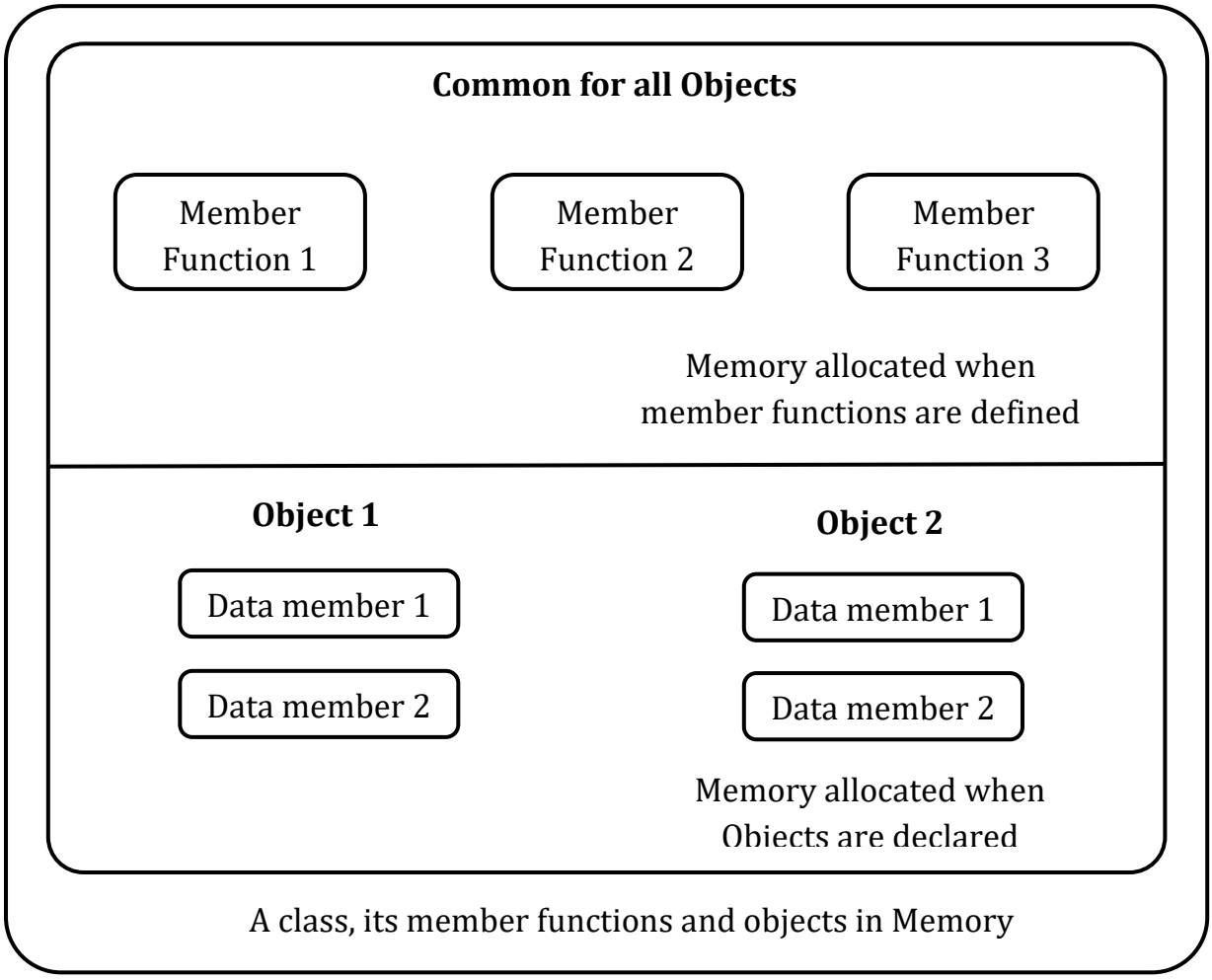
```
#include<iostream.h>
#include<conio.h>
class item
{
int number; float cost;
public:
void getdata(int a, float b); void putdata()
{
cout<<"Number:"<<number<<endl;
cout<<"Cost:"<<cost<<endl;
}
};

void item::getdata(int a, float b)
{
number=a;
cost=b;
}
```

```
int main()
{
item x; clrscr();
cout<<"Object x"<<endl; x.getdata(10,2.9);
x.putdata();
item y;
cout<<"Object y"<<endl; y.getdata(12,4.7);
y.putdata();
getch();
return 0;
}
```

Output:

```
Object x
number:10
Cost: 2.9
Object y
Number:12
Cost: 4.7
```



A class, its member functions and objects in Memory

// Write a C++ program to read and print the name and age of a person

```
#include<iostream.h> // include header file

class person
{
char name[30];
Int age;
public:
void getdata(void);
void display(void);
};
void person :: getdata(void)
{
cout << "Enter name: ";
cin >> name;
cout << "Enter age: ";
cin >> age;
}
```

```
void person : : display(void)
{
cout << "\nNameame: " << name;
cout << "\nAge: " << age;
}

int main()
{
person p;
p.getdata();
p.display();
return 0;
}
```

The output of program is:

Enter Name: Ravinder
Enter age:30
Name:Ravinder
Age: 30

Array of objects:

Collection of similar types of object is known as array of objects.

//Write a program to input name and age of 5 employees and display them.

```
#include<iostream.h>
class Employee
{
char name[30];
int age;
public:
void getdata(void);
void putdata(void);
};

void Employee:: getdata(void)
{
cout<<"Enter Name and Age:";
cin>>name>>age;
}
```

```
void Employee:: putdata(void)
{
cout<<name<<"\t"<<age<<endl;
}

int main()
{
Employee e[5];
int i;
for(i=0; i<5; i++)
{          e[i].getdata();          }

for(i=0; i<5; i++)
 {          e[i].putdata();          }

}
```

Output:	Rajib 25
Enter Name and Age: Rajib 25	Sunil 27
Enter Name and Age: Sunil 27	Ram 23
Enter Name and Age: Ram 23	Bibhuti 26
Enter Name and Age: Bibhuti 26	Ramani 32
Enter Name and Age: Ramani 32	

Inline Functions:-

In C++, we can create short functions that are not actually called, rather their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro. To cause a function to be expanded in line rather than called, precede its definition with the inline keyword.

A function which is expanded in a line when it is called is called inline function.
1. It executes faster than other member function.
2. It can be recursive.
3. Its body does not contain if else, switch, loop, goto statement.
4. The inline keyword is preceded by function definition.

Why inline function is used?
Whenever a function is called, control jumps to definition part of the function. During this jumping of control, a significant amount of time is required. For functions having short definition if it is called several time, huge amount of time will be lost. Therefore we declare such function as inline so that when the function is called, rather than jumping to the definition of function, function definition is expanded in a line wherever it is called.

//Write a program to find area of a circle using inline function.

```
#include<iostream.h>
inline float area(int);
void main()
{
int r;
cout<<"Enter the value of radius:"; cin>>r;
cout<<" Area is: "<<area(r);
}
```

```
inline float area( int r)
{
return 3.14*r*r;
}
```

Output:
Enter the value of radius: 7
Area is : 153.86

//Making Outside Function Inline

Class item
{
.....
public:
.....……..
void getdata(int a, float b);
};
	inline void item :: getdata(int a, float b)
	{
	function body;
	}

//Write a C++ program for finding the area of a triangle using inline functions.

#include<iostream.h>
inline float area(float,float);
int main()
{
int r;
cout<<"Enter the breadth:";
cin>>b;
cout<<"Enter the height:";
cin>>h;
cout<<" Area is: "<<area(b,h);
return 0;
}
	inline float area(float b, float h)
	{
	return (0.5*b*h);
	}
	
	Output:
	Enter the breadth:2
	Enter the height: 3
	Area is : 3.0

Nesting of member functions:

A member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

```
#include<iostream.h>
#include<conio.h>
class item
{
int n,m;
public:
void getdata(int a, int b);
int largest();
void putdata();
};
void item::getdata(int a, int b)
{
n=a;
m=b;
}
void item::putdata()
{
cout<<"The largest number is "
<<largest();
}
int item::largest()
{
if(n>m)
return n;
else
return m;
}
void main()
{
item x;
clrscr();
x.getdata(20,70);
x.putdata();
getch();
}
```

Output:

The largest number is 70

Access Control (Access Specifiers) in Classes:

Access specifiers in C++ class define the access control rules. C++ has 3 new keywords introduced, namely,

1. public
2. private
3. protected

These access specifiers are used to set boundaries for availability of members of a class. Access specifiers in the program, are followed by a colon. You can use one, two or all 3 specifiers in the same class to set different boundaries for different class members. They change the boundary for all the declarations that follow them.

Public

Public, means all the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. Hence there are chances that they might change them. So the key members must not be declared public.

```
class publicAccess
{
public:                     // public access specifier
        int x;              // Data Member Declaration
        void display();     // Member Function decaration
}
```

Private

Private keyword, means that no one can access the class members declared private outside that class. If someone tries to access the private member, they will get a compile time error. By default class variables and member functions are private.

```
class PrivateAccess
{
private:                        // private access specifier
            int x;              // Data Member Declaration
            void display();     // Member Function decaration
}
```

Protected

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class. (If class A is inherited by class B, then class B is subclass of class A. We will learn this later.)

```
class ProtectedAccess
{
protected:                      // protected access specifier
            int x;              // Data Member Declaration
            void display();     // Member Function declaration
}
```

Function overloading:

Function overloading is the process of using the same name for two or more Functions. The secret to overloading is that each redefinition of the function must use either

• different types of parameters (or) • different number of parameters.

You cannot overload function declarations that differ only by return type.

Principles of Function Overloading

1. If two functions have the similar type and number of arguments (data type), the function cannot be overloaded. The return type may be similar or void, but argument data type or number of arguments must be different. For example,

a. sum(int, int, int); sum(int, int);

Here, the above function can be overloaded. Though the data type of arguments in both the functions are similar, number of arguments are different.

b. sum(int, int, int); sum(float, float, float);

In the above example, number of arguments in both the functions are same, but data types are different. Hence, the above function can be overloaded.

2. Passing constant values directly instead of variables also results in ambiguity. For example,

```
int sum(int, int);
float sum(float, float, float);
```

Here, sum() is an overloaded function for integer and float values. Values are passed as follows:

```
sum(2,3);
sum(1.1, 2.3, 4.3);
```

The compiler will flag an error because the compiler cannot distinguish between these two functions. Here, internal conversion of float to int is possible. Hence, in both the above calls integer version of function sum() is executed.

To overcome this problem, the user needs to do the following things.

(1) Declare prototypes of all the overloaded functions before function main().

(2) Pass argument using variables as follows:

```
sum(a, b);          // a and b are integer variables
sum(e, r, t, y);    // e,r,t, and y are float variables
```

Refer following program for confirmation. Also try this program with direct values and by declaring function prototype inside main().

3. The compiler attempts to find an accurate function definition that matches in types and number of arguments and invokes that function. The arguments passed are checked with all declared function. If matching is found then that function gets executed.

4. If there are no accurate matches found, the compiler makes the implicit conversion of actual argument. For example, char is converted to int, and float is converted to double. If all the above steps have failed, then compiler performs user defined functions.

// C++ program to swap two variables using function overloading

```
#include<iostream.h>
#include<conio.h>
void swap(int &a,int &b)
{
int temp;
temp=a; a=b; b=temp;
}
void swap(float &a, float &b)
{
float temp;
temp=a; a=b; b=temp;
}
void swap(char &a, char &b){ char temp;
temp=a; a=b; b=temp;
}

int main()
{
int ix,iy;
float fx,fy;
char cx,cy;
clrscr();
cout<<"Enter 2 integers:";
cin>>ix>>iy;
cout<<"Enter 2 floating point no:s:";
cin>>fx>>fy;
cout<<"Enter 2 characters:";
cin>>cx>>cy;
cout<<"\nIntegers:";
cout<<"\nix="<<ix<<"\niy="<<iy;
swap(ix,iy);
cout<<"\nAfter swapping";
cout<<"\nix="<<ix<<"\niy="<<iy;

cout<<"\nFloating Point numbers:";
cout<<"\nfx="<<fx<<"\nfy="<<fy;
swap(fx,fy);
cout<<"\nAfter swapping";
cout<<"\nfx="<<fx<<"\nfy="<<fy;

cout<<"\nCharacters:";
cout<<"\ncx="<<cx<<"\ncy="<<cy;
swap(cx,cy);
cout<<"\nAfter swapping";
cout<<"\ncx="<<cx<<"\ncy="<<cy;
 return 0;
}
```

Output:

```
Enter 2 integers: 2 4
Enter 2 floating point no:s: 2.5 4.5
Enter 2 characters: A C
Integers:              ix=2      iy= 4
After swapping:        ix=4      iy= 2

floating point nos:    fx= 2.5   fy= 4.5
After swapping         fx= 4.5   fy= 2.5

Characters:            cx= A     cy= C
After swapping         fx= C     fy= A
```

/* Write a c++ program to find the area of circle, rectangle and triangle using function overloading. */

```
#include <iostream.h>
float area(float r)
{
return (3.14*r*r);
}

float area(float l,float b)
{
return (l*b);
}

float area(float t, float l,float b)
{
return (05*l*b);
}
```

```
int main( )
{
float r, l, b;
cout << "Enter the Value of r, l & b: ";
cin>>r>>l>>b;

cout<< "Area of circle is "<<area(r)<<endl;
cout<< "Area of rectangle is "<<area(l,b);
cout<< "Area of triangle is "<<area(0.5,l,b);
return 0;
}
```

Output :

```
Enter the Value of r, l & b: 7 8 6
Area of circle is 153.86
Area of rectangle is 54.0
Area of triangle is 24.0
```

Constructors

A constructor is a special member function which is automatically used to initialize the objects of the class type with legal initial values. The name of the constructor is same the class name. The Compiler calls the Constructor whenever an object is created. Constructors initialize values to object members after storage is allocated to the object. While defining a constructor you must remember that the name of constructor will be same as the name of the class, and constructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator. Constructor always defined in the public section only.

```
class A
{
        int i;
public:
        A();            //Constructor declaration
};

A::A()          // Constructor definition
{
}
```

Use of Constructor in C++:
Suppose you are working on 100's of Person objects and the default value of a data member age is 0. Initializing all objects manually will be a very tedious task.

Instead, you can define a constructor that initializes age to 0. Then, all you have to do is create a Person object and the constructor will automatically initialize the age. These situations arise frequently while handling array of objects.

Also, if you want to execute some code immediately after an object is created, you can place the code inside the body of the constructor.

SPECIAL CHARACTERISTICS OF CONSTRUCTORS:

The following are the some special characteristics of Constructors.

(i) These are called automatically when the objects are created.

(ii) All objects of the class having a constructor are initialized before some use.

(iii) These should be declared in the public section for availability to all the functions.

(iv) Return type (not even void) cannot be specified for constructors.

(v) These cannot be inherited, but a derived class can call the base class constructor.

(vi) These cannot be static.

(vii) A constructor can call member functions of its class.

(viii) These can have default arguments as other C++ functions.

(ix) A constructor can call member functions of its class.

(x) An object of a class with a constructor cannot be used as a member of a union.

Types of Constructors
Constructors are three types :
1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

Default Constructor

Default constructor is the constructor which doesn't take any argument. It has no parameter.

Syntax :

```
class_name ( )
{ Constructor Definition }
```

Example :

```
class Cube
{
int side;
public: Cube( )  {
                        side=10;          }
};

int main()
{
Cube c;
}
```

Output : 10

In this case, as soon as the object is created the constructor is called which initializes its data members. A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
int side;
}

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

Output : 0

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

Parameterized Constructor

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

Example :

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

int main()
{
Cube c1(10);
Cube c2(20);
Cube c3(30);
cout << c1.side;
cout << c2.side;
cout << c3.side;
return 0;
}
```

Output : 10 20 30

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

Copy Constructor

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

Syntax of Copy Constructor

```
class-name (class-name &)
{
. . . .
}
```

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

/* Example Program For Simple Example Program Of Copy Constructor */

```
 #include<iostream.h>
#include<conio.h>
class Example {
                int a,b;                        // VariableDeclaration
        public:
                Example(int x,int y)        //Constructor with Argument
                {
                a=x;                            // Assign Values in Constructor
                b=y;
                cout<<"\n I'm Constructor";
                }

                void Display()  {
                cout<<"\nValues :"<<a<<"\t"<<b;
                }
        };

int main()
{
Example Object(10,20);
Example Object2=Object;         //Copy Constructor (explicit)
Example Object3(Object);        //Copy Constructor (implicit)
Object.Display();
Object2.Display();
Object2.Display();
getch();
return 0;
}
```

Overloaded constructors:

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to function overloading. Overloaded constructors essentially have the same name (name of the class) and different number of arguments. A constructor is called depending upon the number and type of arguments passed. While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

// C++ program to illustrate Constructor overloading

```
#include <iostream.h>
class construct
{
public:
        float area;

construct()  // Constructor with no parameters
{
area = 0;
}

construct(int a, int b) // Constructor with two
parameters

{
area = a * b;
}

void disp()
{
cout<< area<< endl;
}
};
```

```
int main()
{
// Constructor Overloading
// with two different constructors
// of class name

construct obj1;
construct obj2( 10, 20);
obj1.disp();
obj2.disp();
return 1;
}
```

Output :

```
0
200
```

Destructors

Destructors are the functions that are complimentary to constructors. These are used to de- initialize objects when they are destroyed. The destructor is called automatically by the compiler when the object goes out of scope, or when the memory space used by it is de allocated with the help of delete operator.

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

```
class   A
{
public:
~A();
};
```

Destructors will never have any arguments.

class A
{
Public:
A()
{
cout << "Constructor called";
}
~A()
{
cout<<"\n Destructor Called, Object Destroyed “;
}
};
	int main()
	{
	Aobj1; // ConstructorCalled
	int x=1 ;
	if(x)
	{
	A obj2; // Constructor Called
	} // Destructor Called for obj2

Anonymous Objects

It is possible to declare objects without a name, such objects are known as **anonymous objects**. In **anonymous objects**, without an object declaration, a user can initialize and destroy the contents of a class i.e., originally an object exists but it is hidden.

The anonymous object can access the data members by using this pointer. this ptr can store the address of the object with which the member function is called. Anonymous objects are **destroyed** automatically after their use is over.

Anonymous types are *handy* when you create types for “use and throw” purposes.

Let us consider an example:

```
#include <iostream.h>
class Cents {
        int cents;
        public:
        Cents(int value) {
                this-> cents = value; // this pointer refers to the address of the object
                        }
                int getCents() const {
                        return this-> cents;
                }
                ~Cents() {
                        cout << "Destructor is called" << endl;
                }
};
```

```
Cents add(const Cents &c1, const Cents &c2) {
        return Cents(c1.getCents() + c2.getCents()); // return anonymous Cents value
}

int main()
{
cout << "I have " << add(Cents(6), Cents(8)).getCents() << " cents." << endl;
                                        // print anonymous Cents value
}
```

In the above code:

- Cents(6) creates an anonymous object and Cents(8) also creates another anonymous object.

- add() function is called with two anonymous objects and which will return another anonymous object contains the addition of two object data members.

Static Members of a class

1. Static Data Members

It is initialized to zero when first object is created. Only one copy is created for entire class. It is visible within the class but its life time is the entire program. Type and scope of each static member variable must be defined outside the class definition. They are stored separately rather than as a part of the object.

//Program to show the static data members

```
#include<iostream.h>
#include<conio.h>
class item
{
static int count;
int number;
public:
void getdata(int a)
{
number=a;
count++;
}
void getcount()
{
```

```
int main()
{
item a,b,c;
clrscr();
cout<<"Before Reading Data" <<endl;
a.getcount();
b.getcount();
c.getcount();
a.getdata(10);
b.getdata(20);
c.getdata(30);
a.getcount();
b.getcount();
c.getcount();
```

```
cout<<"Count: "<<count<<endl;
}
};
int item::count; // This is mandatory for Static data members of a class, because we know that static members are not stored with class data members.
```

```
getch();
}
```

2. Static Member Function

It has access to only other static members(function or variables) declared in the same class. To declare static function just put the keyword "**static**" in front of the function definition.

```
static returnType functionName(arguments)
{
function body;
}
```

It can be called using the class name as follows:

```
className :: functionName(arguments);
```

// Program to show the work of static function:

```
#include<iostream.h>
#include<conio.h>
class item
{
static int count; int code;
public:
void setcode()
{
code=++count;
}
void showcode()
{
cout<<"Object Number:  "<<code<<endl;
}
static void showcount()
{
cout<<"Count: "<<count<<endl;
}
};
int item::count;
```

```
void main()
{
item t1,t2;
clrscr();
t1.setcode();
t2.setcode();
item::showcount();
item t3;
t3.setcode();
item::showcount();
t1.showcode();
t2.showcode();
t3.showcode();
getch();
}
```

Remember the following Won't Work because code is not static.

```
static void showcount()
{
cout<<code; //code  variable is not static
}
```

Friend Functions

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in

which case the entire class and all of its members are friends. To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows:

```
class Box
{
double width;
public:
double length;
friend void printWidth( Box box );
void setWidth( double wid );
};
```

To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne:

```
class ClassOne{
.....
friend class ClassTwo;
};
```

Consider the following program:

```
#include <iostream.h>
class Box
{
double width;
public:
friend void printWidth( Box bx );
void setWidth( double wid );
};

void Box::setWidth( double wid )
{
width = wid;
}

// Note: printWidth() is not a member
function of any class.
```

```
void printWidth( Box box )
{
/* Because printWidth() is a friend of Box, it
can directly access any member of this class */

cout << "Width of box : " << box.width <<endl;
}
int main( )
{
Box box;
box.setWidth(10.0);
printWidth( box );
return 0;
}
```

Output:

```
Width of box : 10
```

//Program to add two complex numbers using friend function

```
#include<iostream.h>
class complex
{
int real, imag;
public:
void set( )
{
cout<<"enter real and imag part";
cin>>real>>imag;
}
friend complex sum(complex,complex);
void display( );
};

complex sum(complex a,complex b)
{
complex t;
t.real=a.real+b.real;
t.imag=a.imag+b.imag;
return t;
}
```

```
void complex::display( )
{
cout<<"the sum of complex num is:";
cout<<real<<"+i"<<imag;
}

int main( )
{
complex a,b,c;
a.set();
b.set();
c=sum(a,b);
c.display();
return(0);
}
```

Output:

```
enter real and imag part: 2 3
enter real and imag part: 4 5
The sum of complex num is: 6+i8
```

NESTED CLASSES

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

Nested class is a class defined inside a class, that can be used within the scope of the class in which it is defined. In C++ nested classes are not given importance because of the strong and flexible usage of inheritance. Its objects are accessed using "Nest::Display".

```
#include <iostream.h>
class Nest
{
public:
        class Display
        {
                private:
                int s;
```

```
        public:

        void sum( int a, int b)
        {
        s =a+b;
        }

        void show( )
        {
        cout << "\nSum of a and b is:: " << s;
        }
    };
};

void main()
{
Nest::Display x;
x.sum(12, 10);
x.show();
}
```

Output

Sum of a and b is::22

Important Questions

1. What is copy constructor? Explain with an example.
   Ans : Refer topic Copy Constructor (Page no : 16 & 17)

2. Define inline function. Write a C++ program for finding the area of a triangle using inline functions.
   Ans : Refer topic Inline Functions (Page no : 7 & 8)

3. What is function overloading? What are the principles of function overloading?
   Ans : Refer topic Function overloading (Page no : 10 & 11)

4. Write C++ Program that demonstrates the usage of static data member and static member function.
   Ans : Refer topic Static Members of a class (Page no : 20 & 21)

5. Write C++ program to find the area of a circle, rectangle and triangle using function overloading.

Ans : Refer Example program (Page no : 13)

6. What is a constructor? Write different rules associated with declaring constructors.
   Ans : Refer topic Constructors (Page no : 13 & 14)

7. Explain about default and parameterized constructors with suitable examples.
   Ans : Refer topic Default & Parameterized Constructors (Page no : 15 & 16)

8. Write C++ program to add two complex numbers using friend functions.
   Ans : Refer Example program (Page no : 23)

Assignment Questions:

1. What is function overloading? Explain function overloading with an example Program?

2. Explain about static members of a class with an example program ?

3. What is Constructor? What are the different types of constructors ? Explain about them with examples?

4. Define inline function. Explain about inline functions with an example program ?

5. What is friend function? Explain about friend function with an example program?