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, Destructors, Anonymous Objects.

Class

- Object Oriented Programming encapsulates data (attributes) and functions (behavior) into packages called classes.
- The class combines data and methods for manipulating that data into one package. An object is said to be an instance of a class.
- A class is a way to bind the data and its associated functions together. It allows the data to be hidden from external use.
- Generally, a class specification has two parts:
  1. **Class declaration-** The class declaration describes the type and scope of its members.
  2. Class function definitions - The class function definitions describe how the class functions are implemented.
- The general form of a class declaration is:

```
class   class_name
{
        private:
                variable declarations;
                function declarations;
        public:
                variable declarations;
                function declarations;

};
```

- The keyword class is used to declare a class. The body of a class is enclosed within braces and terminated by a semicolon.
- The class body contains the declaration of variables and functions.
- The variables declared inside the class are known as data members and the functions are known as member functions.By default, the members are private.
- A Simple Class Example:

```
        class item
        {

                int no; // variables declaration

                float cost; // private by default

                public:
                        void getdata(int a, float b); // functions declaration
                        void putdata(void); // using prototype
        };
```

Object :

- An object is an instance of a class. It is an entity with characteristics and behaviour that are used in the object oriented programming. An object is the entity that is created to allocate memory.

Declaring Objects:

- The declaration of objects is same as declaration of variables of basic data types.
- Defining objects of class data type is known as class instantiation. Only when objects are created, memory is allocated to them.

Syntax:

ClassName ObjectName;

How to declare objects to a class

- Once a class has been declared, we can create variables of that type by using the class name. For example

  item x; // memory for x is created Creates a variable x of type item.

  In C++, the class variables are known as objects. Therefore, x is called an object of type item.
- We may also declare more than one object in one statement.

  Example:

  item x, y, *z;

  The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification provides only a template and does not create any memory space for the objects.

How to access the class members

- The object can access the public class members of a class by using dot operator or arrow operator.

  Syntax

  Objectname operator membername;

  Example:

  x.show();

  z->show(); {where z is a pointer to class item}

Access Modifiers in C++

- Access modifiers are used to implement an important feature of Object-Oriented Programming known as Data Hiding.
- Access Modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.
- There are 3 types of access modifiers available in C++:

1. Public
2. Private
3. Protected

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be Private.

1. **Public:** The data members and member functions declared public can be accessed by other classes too. The keyword public followed by colon (:) means to indicate the data member and member function that visible outside the class. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

   Program

```
#include<iostream>
using namespace std;
class Test
{
        public:
                int x, y; // variables declaration
        void show()
        {
                cout<<x<<y;
        };
        int main()
        {
                Test t; //Object creation
                t.x = 10;
                t.y = 20;
                t.show();

        };
```

   Output

   10 and 20

2. **Private :** Private keyword is used to prevent direct access to member variables 4 or functions by the object. It is the default access. The keyword private followed by colon (:) is used to make data member and member function visible with in the class only.

Example program

```
#include<iostream>
using namespace std;
class Circle
{
        private:
                double radius;   // private data member
        public:
                double  compute_area() // public member function
                {
                        // member function can access private data member radius
                        return 3.14*radius*radius;
                }
};
int main() // main function
{
        Circle obj;      // creating object of the class
        obj.radius = 1.5;
        cout<< "Area is:" <<obj.compute_area();
        return 0;
}
```

3. **Protected** -Protected access is the mechanism same as private. It is frequently used in inheritance. Private members are not inherited at any case whereas protected members are inherited. The keyword private followed by colon (:) is used to make data member and member function visible with in the class and its child classes.

Defining Member Functions:

- Member functions are defined in two places. They are;

  1. Member Function Inside the class:

  2. Member Function Outside the class:

1. Member Function Inside the class:

- Member function inside the class can be declared in public or private section. The member functions defined inside the class are treated as inline function. If the member function is small then it should be defined inside the class. Otherwise, it should be defined outside the class.

Program

```
#include<iostream>
using namespace std;
class student
{
        private:
                string name;
                int rollnumber;
        public:
                void display()//member function
                {
                        cout<<"enter name";
                        cin>>name;
                        cout<<"enter roll number";
                        cin>>rollnumber;
                        cout<<name;
                        cout<<rollnumber;
                }
};
main()
{
        student s1;
        s1.display();
}
```

2. Member Function Outside the class:

- If function is defined outside the class, its prototype declaration must be done inside the class. While defining the function, scope access operator and class name should precede the function name.
- To define a function outside the class, following care must be taken.
  1. The prototype of function must be declared inside the class.
  2. The function name must be preceded by class name and its return type separated by scope access operator.

Example

```
#include<iostream>

using namespace std;

class book

{

        private:

                string name;

                int price;

                int pages;

        public:

                void show(); // prototype declaration

};

void book::show() // definition outside the class

{

                name="c++ programming";

                price=200;

                pages=100;

                cout<<"\n Codeno ="<<name;

                cout<<"\n Price ="<<price;

                cout<<"\n Quantity="<<pages;

}

int  main()

{
```

```
    book book1; // object declaration

    book1.show(); // call to public member function

    return 0;

}
```

Scope resolution operator

- As the name suggests, Scope resolution operator is used to get the hidden names due to variable scopes so that you can still use them
- In C++, scope resolution operator is ::. Scope resolution operator in C++ can be used for:
  1. To access a global variable when there is a local variable with same name
  2. To define a function outside a class
  3. To access a class's static variables
  4. Refer to a class inside another class
  5. For namespace
  6. In case of multiple Inheritance

1. **To access a global variable when there is a local variable with same name**
   - C++ program to show that we can access a global variable using scope resolution operator :: when there is a local .

   Program

```
#include<iostream>

using namespace std;

int x;  // Global x

int main()

{

        int x = 10; // Local x

        cout<< "Value of global x is " << ::x;

        cout<< "\nValue of local x is " << x;

        return 0;

}
```

2. **To define a function outside a class.**

```
#include<iostream>
using namespace std;
class A
{
	public:
		void fun();
};
void A::fun()
{
	cout<< "fun() called";
}
int main()
{
	A a;
	a.fun();
	return 0;
}
```

3. **For namespace**

```
#include<iostream>
int main()
{
	std::cout<< "Hello" <<std::endl;
}
```

4. Refer a class inside another class

```
#include<iostream>
using namespace std;
class outside
{
        public:
                int x;
         class inside
         {
                public:
                        int x;
                        static int y;
                        int fun();
         };
  };
int outside::inside::y = 5;
int main()
{
        outside A;
        outside::inside B;
}
```

5. To access a class's static variables

```
#include<iostream>
using namespace std;
class Test
{
        Static int x;
    public:
        static int y;
        void func(int x)
        {
            cout<< "Value of static x is " << Test::x;
            cout<< "\nValue of local x is " <<x;
        }
};
int Test::x = 1;
int Test::y = 2;
int main()
{
    Test obj;
    int x = 3 ;
    obj.func(x);
    cout<< "\nTest::y = " << Test::y;
    return 0;
}
```

Function Overloading:

- Member functions can be overloaded like any other normal functions. Overloading means one function is defined with multiple definitions with same functions name in the same scope.
- Defining multiple functions with same name is known as function overloading or function polymorphism.
- Polymorphism means one function having many forms. The overloaded function must be different in their argument list and with different data types.

Principles of Function Overloading:

- 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,

  sum(int,int,int);

  sum(int,int);

- Here, the above function can be overloaded. Though the data types of arguments in both the functions are similar, numbers of arguments are different.

  sum(int,int,int);

  sum(float,float,float);

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

Example program

```
#include<iostream>

Using namespace std;

#define pi 3.142857142857142857;

Int calcarea(int length,int breadth);

Float calcarea(double base,double height);

Float calcarea(double radius);

int main()

{

    int area1;

    float area2;
```

```
                double area3;
                area1=calcarea(10,20);
                area2=calcarea(4.5,2.1);
                area3=calcarea(3.12145);
                cout<<"Area of rectangle is: "<<area1><<endl;
                cout<<"Area of triangle is: "<<area2><<endl;
                cout<<"Surface Area of sphere is: "<<area3><<endl;
                return 0;
}
Int calcarea(int length,int breadth)
{
        return (length*breadth);
}
Float calcarea(double base,double height)
{
        return ((0.5)*base*height);
}
Float calcarea(double radius)
{
        return (4*pi*radius*radius);
}
```

Overloading Member Functions:

- Member functions are also overloaded in the same fashion as other ordinary functions.
- We learned that overloading is nothing but a function is defined with multiple definitions with same function name in the same scope.

Example program-1

```
#include <iostream>
using namespace std;
class printvalues
{
    public:
        void print(int i)
        {
            cout<< "Printing integer number: " << i <<endl;
        }
        void print(double  f)
        {
            cout<< "Printing floating number: " << f <<endl;
        }
        void print(string c)
        {
            cout<< "Printing string : " << c <<endl;
        }
};
int main(void)
{
        printvalues p1;
        p1.print(5);
        p1.print(500.263); // Call print to print float
        p1.print("Hello C++");   // Call print to print strings
        return 0;
}
```

Example Program -2

```
#include<iostream>
using namespace std;
#define pi 3.14
Class printvalues
{
        public:
                int area1;
                float area2;
                double area3;
                int calcarea(int length,int breadth)
                {
                        return (length*breadth);
                }
                Float calcarea(double base,double height)
                {
                        return ((0.5)*base*height);
                }
                Float calcarea(double radius)
                {
                                return (4*pi*radius*radius);

                }
};
main()
{
        printvalues p1;
        p1.area1=p1.calcarea(10,20);
```

```
        p1.area2=p1.calcarea(4.5,2.1);
        p1.area3=p1.calcarea(3.12145);
        cout<<"area of reactangle is"<<p1.area1<<endl;
        cout<<"area of triangle is"<<p1.area2<<endl;
        cout<<"area of sphere is "<<p1.area3<<endl;
    }
```

Nested Class:

- When a class is defined in another class, it is known as nesting of classes.
- In nested class the scope of inner class is restricted by outer class.

Write a program to display some message using nested class.

```
#include<iostream>
using namespace std;
class one
{
public:
    class two
    {
        public:
            void display()
            {
                cout<< “c++ programming\n”;
            }
        };
};
int main()
{
    one::two x;
    x .display();
}
```

Constructors and Destructors:

We defined a separate member function for reading input values for data members. Using object, member function is invoked and data members are initialized. The programmer needs to call the function. C++ provides a pair of in-built special member functions called constructor and destructor.

The constructor constructs the objects and destructor destroys the objects. In operation, they are opposite to each other. The compiler automatically executes these functions. The programmer does not need to make any effort for invoking these functions.

When an object is created, constructor is executed. The programmer can also pass values to the constructor to initialize member variables with different values. The destructor destroys the object. The destructor is executed at the end of the function when objects are of no use or go out of scope. Constructors and destructors are special member functions. The only difference is that destructor is preceded by a ~ (tilde) operator.

Constructor

- A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class.

Characteristics of constructors

- They should be declared in the public section
- They do not have any return type, not even void
- They get automatically invoked when the objects are created
- They cannot be inherited though derived class can call the base class constructor.
- Like other functions, they can have default arguments.
- You cannot refer to their address.
- Constructors cannot be virtual

How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself.
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).
- They cannot be inherited though derived class can call the base class constructor
- You cannot refer to their address.

Types of constructors

1. Default constructor
2. Parameterized constructor

1. Default constructor

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

Example program-1

```
#include<iostream>
using namespace std;
class numbers
{
    public:
        int a,b;
    public:
        numbers()//function definition
        {
            a=10;
            b=20;
            cout<<a<<endl<<b<<endl;
        }
};
int main()
{
        numbers n1;
}
```

Example program-2

```
#include<iostream>
Using namespace std;
class num
{
        int a, b;
    public:
        num()
        {
            a =5; b=2;
        }
        void show()
        {
            cout<<a<<b;
        }
    };
    Void main()
    {
        num n;
        n.show();
    }
```

Example program-3

```
#include <iostream>
using namespace std;
// declare a class
class  Wall
{
        private:
                double length;
        public:
                Wall()
                {
                        length = 5.5;
                        cout<< "Creating a wall." <<endl;
                        cout<< "Length = " << length <<endl;
                }
};
int main()
{
        Wall wall1;
        return 0;
}
```

Example program-4

```
#include <iostream>
using namespace std;
class print
{
        public:
                int a;
                float b;
                string c;
                print()
                {
                        a=10;
                        b=20.4;
                        c="c++ programming";
                }
};
int main()
{
        print p1;
        cout<<"integer number is:"<<p1.a<<endl;
        cout<<"floating point number is:"<<p1.b<<endl;
        cout<<"string is:"<<p1.c;
}
```

2. Parameterized Constructors:

- It is also possible to create constructor with arguments, and such constructors are called parameterized constructors.
- For such constructors, it is necessary to pass values to the constructor when an object is created.
- Typically, these arguments help initialize an object when it is created.
- When you define the constructor's body, use the parameters to initialize the object.

- When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function.

Example program

```
#include <iostream>

using namespace std;

class Wall

{

    private:

            double length;

            double height;

    public:

            Wall(double len, double hgt)    // create parameterized constructor

            {

                    // initialize private variables

                    length = len;

                    height = hgt;

            }

            Double calculateArea()

            {

                    return length * height;

            }

        };

        main()

        {

                    Wall wall1(10.5, 8.6);    // create object and initialize data members

                    Wall wall2(8.5, 6.3);

                    cout<< "Area of Wall 1: " << wall1.calculateArea() <<endl;

                    cout<< "Area of Wall 2: " << wall2.calculateArea() <<endl;

        }
```

OVERLOADING CONSTRUCTORS (MULTIPLE CONSTRUCTORS):

- Similar to functions, it is also possible to overload constructors.
- In the previous examples, we declared single constructors without arguments and with all arguments.
- A class can contain more than one constructor. This is known as **constructor overloading**.
- All constructors are defined with the same name as the class they belong to.
- All the constructors contain different number of arguments.
- Depending upon the number of arguments, the compiler executes appropriate constructor.

Example program-1

```
#include <iostream>

#include <string>

using namespace std;

class student

{

        public:

                int number;

                string name;

                student()

                {

                        number=1;

                        name="aditya";

                        cout<<"number is "<<number<<endl;

                        cout<<"name is:"<<name<<endl;

                }

                student(int i,string s)

                {

                        number=i;

                        name=s;

                        cout<<"name is"<<number<<endl;

                        cout<<"number is"<<name<<endl;

                }
```

```
};
main()
{
        student s1;
        student s2(1,"aditya");
}
```

Example program-2

```
#include <iostream>
#include <string>
using namespace std;
class student
{
        public:
                int rollnumber;
                string name;
                string branch;
                student(int r,string n,string b)
                {
                        rollnumber=r;
                        name=n;
                        branch=b;
                }
                void display()
                {

                        cout<<"name:"<<rollnumber<<endl;
                        cout<<"number:"<<name<<endl;
                        cout<<"branch:"<<branch<<endl;
```

```
        }
};
int main()
{
    student s1(1,"aditya","IT");
    student s2(2,"pragati","CSE");
    s1.display();
    s2.display();
}
```

Destructors:

- The destructor is also a special member function like constructor. Destructors destroy the class objects created by constructors.
- The destructors have the same name as their class, preceded by a tilde (~).
- It is automatically called when object goes out of scope.
- Destructor releases memory space occupied by the objects.
- The program given below explains the use of destructor.

Characteristics of destructor

1) Name should begin with tilde sign(~) and must match class name.
2) There cannot be more than one destructor in a class.
3) Unlike constructors that can have parameters, destructors do not allow any parameter.
4) They do not have any return type, just like constructors.
5) When you do not specify any destructor in a class, compiler generates a default destructor and inserts it into your code.

6) Destructors cannot be overloaded.

Example program-1

```
#include <iostream>
using namespace std;
class student
{
    public:
        student()
```

```
        {
                cout<<"Constructor called"<<endl;
        }
        ~student()
        {
                cout<<"Destructor called"<<endl;
        }
};
int main(void)
{
   student e1; //creating an object of student
   student e2; //creating an object of student
   return 0;
}
```

Example program-2

```
#include <iostream>
using namespace std;
class sample
{
        public:
                sample()
                {
                        cout<<"Constructor is called"<<endl;
                }
                ~sample()
                {
                        cout<<"Destructor is called"<<endl;
                }
```

```
        void display()
        {
            cout<<"Hello World!"<<endl;
        }
};
int main()
{
        Sample obj;
        obj.display();
        return 0;
}
```

Anonymous objects

- It is possible to declare objects without any name.
- These objects are said to be anonymous objects.
- Constructors and destructors are called automatically whenever an object is created and destroyed respectively.
- The anonymous objects are used to carry out these operations without object.

Program

```
#include <iostream>
using namespace std;
class noname
{
        int x;
public:
        noname()
        {
            cout<<"\n In default constructor";
            x=10;
            cout<<x;
        }
        noname(int i)
        {
            cout<<"\n Parameterized constructor";
            x=i;
            cout<<x;
```

```
                    }
                    ~noname()
                    {
                              Cout<<"\n  In destructor";
                    }
};
Void main()
{
          noname();
          noname(12);
}
```