Unit-III

Class

A class is a user defined data type. A class is a logical abstraction. It is a template that defines the form of an object. A class specifies both code and data. It is not until an object of that class has been created that a physical representation of that class exists in memory. When you define a class, you declare the data that it contains and the code that operates on that data. Data is contained in instance variables defined by the class known as data members, and code is contained in functions known as member functions. The code and data that constitute a class are called members of the class. The general form of class declaration is:

class class-name {

access-specifier:

data and functions

access-specifier:

data and functions

// ...

access-specifier:

data and functions

} object-list;

The object-list is optional. If present, it declares objects of the class. Here, access-specifier is one of these three C++ keywords:

public private protected

By default, functions and data declared within a class are private to that class and may be accessed only by other members of the class. The public access_specifier allows functions or data to be accessible to other parts of your program. The protected access_specifier is needed only when inheritance is involved.

Example:

```
#include<iostream.h>

#include<conio.h>


Class myclass {              // class declaration

// private members to

myclassint a;


public:

// public members to

myclassvoid set_a(intnum);
```

```
int get_a( );

};
```

Object

An object is an identifiable entity with specific characteristics and behavior. An object is said to be an instance of a class. Defining an object is similar to defining a variable of any data type: Space is set aside for it in memory. Defining objects in this way means creating them. This is also called instantiating them. Once a Class has been declared, we can create objects of that Class by using the class Name like any other built-in type variable as shown:

```
className objectName
```

Example

```
void main( ) {

myclass ob1, ob2;                    //these are object of type myclass

// ... program code

}
```

Accessing Class Members

The main() cannot contain statements that access class members directly. Class members can be accessed only by an object of that class. To access class members, use the dot (.) operator. The dot operator links the name of an object with the name of a member. The general form of the dot operator is shown here:

```
object.member
```

Example

```
ob1.set_a(10);
```

The private members of a class cannot be accessed directly using the dot operator, but through the member functions of that class providing data hiding. A member function can call another member function directly, without using the dot operator.

C++ program to find sum of two numbers using classes

```
#include<iostream.h>


#include<conio.h>

class A{


 int a,b,c;

 public:


 void sum(){

 cout<<"enter two

 numbers";cin>>a>>b;
```

```
 c=a+b;
 cout<<"sum="<<c;

 }
};
int main(){
 A u;
 u.sum();

 getch();
 return(0)
 ;

}
```

Scope Resolution operator

Member functions can be defined within the class definition or separately using scope resolution operator (::). Defining a member function within the class definition declares the function inline, even if you do not use the inline specifier. Defining a member function using scope resolution operator uses following declaration

```
return-type class-name::func-name(parameter- list) {
// body of function
}
```

Here the class-name is the name of the class to which the function belongs. The scope resolution operator (::) tells the compiler that the function func-name belongs to the class class-name. That is, the scope of the function is restricted to the class-name specified.

```
Class myclass {
int a;
public:
void set_a(intnum);  //member function declaration
int get_a( );                //member function declaration
};

//member function definition outside class using scope resolution operator
void myclass :: set_a(intnum)
{
a=num;
```

```
}

int myclass::get_a( ) {

return a;

}
```

Another use of scope resolution operator is to allow access to the global version of a variable. In many situation, it happens that the name of global variable and the name of the local variable are same .In this while accessing the variable, the priority is given to the local variable by the compiler. If we want to access or use the global variable, then the scope resolution operator (::) is used. The syntax for accessing a global variable using scope resolution operator is as follows:-

:: Global-variable-name

Static Data Members

When you precede a member variable's declaration with static, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus, all objects of that class use that same variable. All static variables are initialized to zero before the first object is created. When you declare a static data member within a class, you are not defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated.

One use of a static member variable is to provide access control to some shared resource used by all objects of a class. Another interesting use of a static member variable is to keep track of the number of objects of a particular class type that are in existence.

Static Member Functions

Member functions may also be declared as static. They may only directly refer to other static members of the class. Actually, static member functions have limited applications, but one good use for them is to "preinitialize" private static data before any object is actually created. A static member function can be called using the class name instead of its objects as follows:

class name :: function name

```
//Program showing working of static class

members#include <iostream.h>


#include<conio.h>

class static_type {


static int i;                                   //static data member

public:
```

```
static void init(int x) {i = x;}            //static member
functionvoid show() {cout << i;}};

int static_type :: i;                              // static data member
definitionint main(){

static_type::init(100);                  //Accessing static
functionstatic_type x;

x. show();
return 0;
}
```

Constructor:

A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it construct the value data members of the class. The constructor functions have some special characteristics.

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.

Example:

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

class myclass {              // class
declarationint a;

public:
myclass( );                  //default
constructorvoid show( );

};
```

```
myclass :: myclass( ) {

cout <<"In

constructor\n";a=10;


}

myclass :: show( ) {

cout<< a;


}

int main( ) {

int ob; // automatic call to

constructorob.show( );


return0;

}
```

In this simple example the constructor is called when the object is created, and the constructor initializes the private variable a to10.

Default constructor

The default constructor for any class is the constructor with no arguments. When no arguments are passed, the constructor will assign the values specifically assigned in the body of the constructor. It can be zero or any other value. The default constructor can be identified by the name of the class followed by empty parentheses. Above program uses default constructor. If it is not defined explicitly, then it is automatically defined implicitly by the system.

Parameterized Constructor

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
#include <iostream.h>

#include<conio.h>

class myclass {


int a, b;

public:


myclass(int i, int j) //parameterized constructor

 {a=i; b=j;}
```

```
void show()  { cout << a << " " << b;}

};

int main() {

myclass ob(3, 5); //call to

constructorob.show();


return 0;

}
```

C++ supports constructor overloading. A constructor is said to be overloaded when the same constructor with different number of argument and types of arguments initializes an object.

Copy Constructors

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. If class definition does not explicitly include copy constructor, then the system automatically creates one by default. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

The most common form of copy constructor is shown here:

```
classname (const classname &obj) {

// body of constructor


}
```

Here, obj is a reference to an object that is being used to initialize another object. The keyword const is used because obj should not be changed.

Destructor

A destructor destroys an object after it is no longer in use. The destructor, like constructor, is a member function with the same name as the class name. But it will be preceded by the character Tilde (~).A destructor takes no arguments and has no return value. Each class has exactly one destructor. . If class definition does not explicitly include destructor, then the system automatically creates one by default. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible.

```
// A Program showing working of constructor and destructor

#include<iostream.h>
```

```
#include<conio.h>
class Myclass{
public:

int x;
Myclass(){
                                //Constructo
rx=10; }

~Myclass(){
                                //Destructo
rcout<<"Destructing...." ;

}
int main(){
Myclass ob1, ob2;

cout<<ob1.x<<" "<<ob2.x;
return 0; }
```

Output:

10 10

Destructing……..

Destructing……..

Friend function

In general, only other members of a class have access to the private members of the class. However, it is possible to allow a nonmember function access to the private members of a class by declaring it as a friend of the class. To make a function a friend of a class, you include its prototype in the class declaration and precede it with the friend keyword. The function is declared with friend keyword. But while defining friend function, it does not use either keyword friend or :: operator. A function can be a friend of more than one class. Member function of one class can be friend functions of another class. In such cases they are defined using the scope resolution operator.

A friend, function has following characteristics.

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

Program to illustrate use of friend function

```
#include<iostream.h>
#include<conio.h>
class A{

 int x, y;
 public:

 friend void display(A &obj);
 void  getdata()  { cin>>x>>y;

 }
};
void display(A &obj){
 cout<<obj.x<<obj.y;

}
int main(){
 A a;
 a.getdata()
 ;display(a);
 getch();
 return 0;

}
```

Operator overloading

There is another useful methodology in C++ called operator overloading. The language allows not only functions to be overloaded, but also most of the operators, such as +, -, *, /, etc. As the name suggests, here the conventional operators can be programmed to carry out more complex operations. This overloading concept is fundamentally the same i.e. the same operators can be made to perform

different operations depending on the context. Such operators have to be specifically defined and appropriate function programmed. When an operator is overloaded, none of its original meaning is lost. It is simply that a new operation, relative to a specific class, is defined. For example, a class that defines a linked list might use the + operator to add an object to the list. A class that implements a stack might use the + to push an object onto the stack.

An operator function defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword operator. The general form of an operator function is

type classname::operator#(arg-list) { // operations

}

Here, the operator that you are overloading is substituted for the #, and type is the type of value returned by the specified operation. Operator functions can be either members or nonmembers of a class. Nonmember operator functions are often friend functions of the class.

These operators cannot be overloaded:- ., : :, .*, ?

The process of overloading involves the following steps:

- Create a class that defines the data type that is to be used in the overloading operation.
- Declare the operator function operator op() in the public part of the class.
- Define the operator function to implement the required operations.

Overloading a unary operator using member function

Overloading a unary operator using a member function, the function takes no parameters. Since, there is only one operand, it is this operand that generates the call to the operator function. There is no need for another parameter.

Overloading unary minus operator

```
#include<iostream.h>

#include<conio.h>


class  A  {

 int  x,y,z;

 public:


  void getdata(int a,int b,int c) {

   x=a;
```

```
  y=b;
  z=c;
 }
 void display() {
  cout<<"\nx="<<x<<"\ny="<<y<<"\nz="<<z;
 }
 void operator -()  //unary minus overload function
 {
  x=-x;
  y=-y;
  z=-z;
 }
};
int main() {
 A a;
 a.getdata(2,3,4)
 ;a.display();
 -a;                    //activates operator –()
 functiona.display();
 getch();
 return
 0;
}
```

Overloading binary operator

When a member operator function overloads a binary operator, the function will have only one parameter. This parameter will receive the object that is on the right side of the operator. The object on the left side is the object that generates the call to the operator function and is passed implicitly by this pointer. 'this' can be used in overloading + operator .

```
#include<iostream.h>
#include<conio.h>
class A{
 int x,y;
 public:
 void input() {
  cin>>x>>y;
 }
 void display() {
 cout<<"\nx="<<x<<"\ny="<<y<<"\nx+y="<<x+y;
 }
 A operator+(A p );            //overload binary + operator
};

A A :: operator+(A p) {A
  t;
  t.x=x + p.x;
  t.y=y + p.y;
  return t;
}
int main(){ A
a1, a2, a3;
a1.input();
a2.input();
a3=a2+a1;                              //activates operator+()
functiona3.display();
```

```
getch();

return

0;


}
```

this Pointer

It is facilitated by another interesting concept of C++ called this pointer. 'this' is a C++ keyword. 'this' always refers to an object that has called the member function currently. We can say that 'this' is a pointer. It points to the object that has called this function this time. While overloading binary operators, we use two objects, one that called the operator function and the other, which is passed to the function. We referred to the data member of the calling object, without any prefix. However, the data member of the other object had a prefix. Always 'this' refers to the calling object place of the object name.

Inheritance

Inheritance is the mechanism by which one class can inherit the properties of another. It allows a hierarchy of classes to be build, moving from the most general to the most specific. When one class is inherited by another, the class that is inherited is called the base class. The inheriting class is called the derived class. In general, the process of inheritance begins with the definition of a base class. The base class defines all qualities that will be common to any derived class. In essence, the base class represent the most general description of a set of traits. The derived class inherits those general traits and adds properties that are specific to that class. When one class inherits another, it uses this general form:

```
class derived-class-name : access base-class-name{

// ...

}
```

Here access is one of the three keywords: public, private, or protected. The access specifier determines how elements of the base class are inherited by the derived class.

When the access specifier for the inherited base class is **public**, all public members of the base class become public members of the derived class. If the access specifier is **private**, all public members of the base class become private members of the derived class. In either case, any private members of the base class remain private to it and are inaccessible by the derived class.

It is important to understand that if the access specifier is **private**, public members of the base become private members of the derived class. If the access specifier is not present, it is private by default.

The **protected** access specifier is equivalent to the private specifier with the sole exception that protected members of a base class are accessible to members of any class derived from that base. Outside the base or derived classes, protected members are not accessible. When a protected member of a base class is inherited as public by the derived class, it becomes a protected member of the derived class. If the base class is inherited as private, a protected member of the base becomes a private member of the derived class. A base class can also be inherited as protected by a derived class. When this is the case, public and protected members of the base class become protected members of the derived class (of course, private members of the base remain private to it and are not accessible bythe derived class).

Program to illustrate concept of inheritance

```
#include<iostream.h>

#include<conio.h>

class base                          //base class

{

 int x,y;

 public:

  void show()  { cout<<"In

   base class";

  }

};

class derived : public base         //derived class

{

 int a,b;

 public:
```

```
 void show2()  { cout<<"\nIn
  derived class";

 }
};
int main() {
 derived d;

 d.show();          //uses base class's show() function
 d.show2();         //uses derived class's show2() function
 getch();

 return 0;
}
```

Types of Inheritances

Single Inheritance

The process in which a derived class inherits traits from only one base class, is called single inheritance. In single inheritance, there is only one base class and one derived class. The derived class inherits the behavior and attributes of the base class. However the vice versa is not true. The derived class can add its own properties i.e. data members (variables) and functions. It can extend or use properties of the base class without any modification to the base class. We declare the base class and derived class as given below:

```
class base_class {

};

class derived_ class : visibility-mode base_ class {

};
```

Program to illustrate concept of single inheritance

```
#include<iostream.h>

#include<conio.h>

class base                    //base class
```

```
{
 int x,y;
 public:
  void show()  { cout<<"In
   base class";
  }
};
class derived : public base //derived class
{
 int a,b;
 public:
  void show2()  { cout<<"\nIn
   derived class";
  }
};
 int main() {
  derived d;
  d.show();          //uses base class's show() function
  d.show2();         //uses derived class's show2() function
  getch();
  return 0;
 }
```

Ambiguity in single Inheritance

Whenever a data member and member functions are defined with the same name in both the base and derived class, ambiguity occurs. The scope resolution operator must be used to refer to particular class as: object name.class name :: class member

Multiple Inheritance

The process in which a derived class inherits traits from several base classes, is called multiple inheritance. In Multiple inheritance, there is only one derived class and several base classes. We declare the base classes and derived class as given below:

```
class base_class1{

};

class base_class2{

};

class derived_ class : visibility-mode base_ class1 , visibility-mode base_ class2 {

};
```

Multilevel Inheritance

The process in which a derived class inherits traits from another derived class, is called Multilevel Inheritance. A derived class with multilevel inheritance is declared as :

```
class base_class {

};

class derived_ class1 : visibility-mode base_ class {

};

class derived_ class 2: visibility-mode derived_ class1 {

};
```

Here, derived_ class 2 inherits traits from derived_ class 1 which itself inherits from base_class.

Hierarchical Inheritance

The process in which traits of one class can be inherited by more than one class is known as Hierarchical inheritance. The base class will include all the features that are common to the derived classes. A derived class can serve as a base class for lower level classes and so on.

Hybrid Inheritance

The inheritance hierarchy that reflects any legal combination of other types of inheritance is known as hybrid Inheritance.

Overriding

Overriding is defined as the ability to change the definition of an inherited method or attribute in a derived class. When multiple functions of the same name exist with different signatures it is called function overloading. When the signatures are the same, they are called function overriding. Function overriding allows a derived class to provide specific implementation of a function that is already provided by a base class. The implementation in the derived class overrides or replaces the implementation in the corresponding base class.

Virtual base classes

A potential problem exists when multiple base classes are directly inherited by a derived class. To understand what this problem is, consider the following class hierarchy:

Here the base class Base is inherited by both Derived1and Derived2. Derived3 directly inherits both Derived1and Derived2.However, this implies that Base is actually inherited twice by Derived3.First it is inherited through Derived1, and then again through Derived2.This causes ambiguity when a member of Base is used by Derived3. Since two copies of Base are included in Derived3, is a reference to a member of Base referring to the Base inherited indirectly through Derived1or to the Base inherited indirectly through Derived2? To resolve this ambiguity, C++ includes a mechanism by which only one copy of Base will be included in Derived3. This feature is called a virtual base class.

In situations like this, in which a derived class indirectly inherits the same base class more than once, it is possible to prevent multiple copies of the base from being present in the derived class by having that base class inherited as virtual by any derived classes. Doing this prevents two or more copies of the base from being present in any subsequent derived class that inherits the base class indirectly. The virtual keyword precedes the base class access specifier when it is inherited by a derived class.

```
// This program uses a virtual base class.

#include <iostream>

using namespace std;

class Base {

public:

int i;

};

// Inherit Base as virtual

classDerived1 : virtual publicBase {

public:

int j;

};

// Inherit Base as virtual here, too

classDerived2 : virtual publicBase {
```

```
public:
int k;
};
// Here Derived3 inherits both Derived1 and Derived2.
// However, only one copy of base is inherited.
class Derived3 : publicDerived1, publicDerived2 {
public:
int product( ) { return i*j*k; }
};
int main( ) {
Derived3 ob;
ob.i = 10; // unambiguous because virtual Base
ob.j = 3;
ob.k = 5;
cout << "Product is: " << ob.product( ) << "\n";
return 0;
}
```

If Derived1and Derived2 had not inherited Base as virtual, the statement ob.i=10 would have been ambiguous and a compile-time error would have resulted. It is important to understand that when a base class is inherited as virtual by a derived class, that base class still exists within that derived class.

For example, assuming the preceding program, this fragment is perfectly valid:

```
Derived1 ob;
ob.i = 100;
```

Friend Classes

It is possible for one class to be a friend of another class. When this is the case, the friend class and all of its member functions have access to the private members defined within the other class. For example,

```
// Using a friend class.
#include<iostream.h>
#include<conio.h>
class A{
```

```
 int x, y;
public:
 friend void display(A &obj);
 friend class B;
 void getdata() {
  cin>>x>>y;
 }
};
class    B{
 int    p,q;
 public:
 void get(A &obj) {
   p=obj.x;
   q=obj.y;
 }
};
void display(A &obj){
 cout<<obj.x<<obj.y;
}
int main(){A
 a;
 B b;
 b.get(a);
 a.getdata();
 display(a);
 getch();
```

```
return 0;

}
```

It is critical to understand that when one class is a friend of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become members of the friend class.