Classes and Objects

Creation, accessing class members, Private Vs Public, Constructor and Destructor Objects.

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 myclass
int a;
public:
// public members to myclass
void set_a(intnum);
int get_a( );
};
```

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(int num); //member function declaration
int get_a( ); //member function declaration
};
//member function definition outside class using scope resolution operator
void myclass :: set_a(int num)
{
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 function
void show() {cout << i;}};
int static_type :: i; // static data member definition
int main(){
static_type::init(100); //Accessing static function
static_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 declaration
int a;
public:
myclass( ); //default constructor
void show( );
};
myclass :: myclass( ) {
cout <<"In constructor\n";
a=10;
}
myclass :: show( ) {
cout<< a;
}
```

```
int main( ) {
int ob; // automatic call to constructor
ob.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 constructor
ob.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(){ //Constructor
x=10; }
~Myclass(){ //Destructor
cout<<"Destructing…." ;
}
int main(){
Myclass ob1, ob2;
cout<<ob1.x<<" "<<ob2.x;
return 0; }
```

Output:

```
10 10
Destructing……..
Destructing……..
```