SCS1202_Object Oriented Programming

UNIT 2 CLASSES AND OBJECTS

Working with classes - Classes and objects - Class specification-Class objects-Accessing class members Defining class members-Inline functions-Accessing member functions within class-Data hiding-Class member accessibility-Empty classes,

Characteristics of OOPS

- Programs are divided into known objects
- Builds the data and functions around these objects or entities.

Organization of data and functions in OOP

- Hence object may communicate with each other through functions.
- Now data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

Definition of OOPS:

OOP is a method of implementation in which programs are organized as co-operative collections of objects, each of which represent an instance of some class and whose classes are all members of a hierarchy of classes united through the property called inheritance.

Concepts of OOPS

General concepts of OOPS comprises the following

1. Object
2. Class
3. Data abstraction
4. Inheritance
5. Polymorphism
6. Dynamic Binding
7. Message passing.

1. Object

Object is an entity that can store data and, send and receive messages. They are runtime entities, they may represent a person, a place a bank account, a table of data or any item that the program must handle. It is an instance of a class.

They may also represent user-defined data such as vectors, time and lists. When a program is executed, **the object interact by sending messages to one another**. Each object contain data and code to manipulate the data objects can interact without having to know details of each other’s data or code. It is sufficient to know the type of message accepted and the type of response returned by the objects.

OBJECT : STUDENT
data
functions

2. **classes**

A class is a collection of objects of similar type. Classes are user defined data types and behave like the built in types of a programming language. For example mango, apple and orange are members of the class fruit. Then the statement FRUIT MANGO; will create an object mango belonging to the class fruit. The syntax used to create an object is no different than the syntax used to create an integer object in C. if **fruit** has been defined as a class, then the statement

fruit mango;

will create an object **mango** belonging to the class **fruit**.

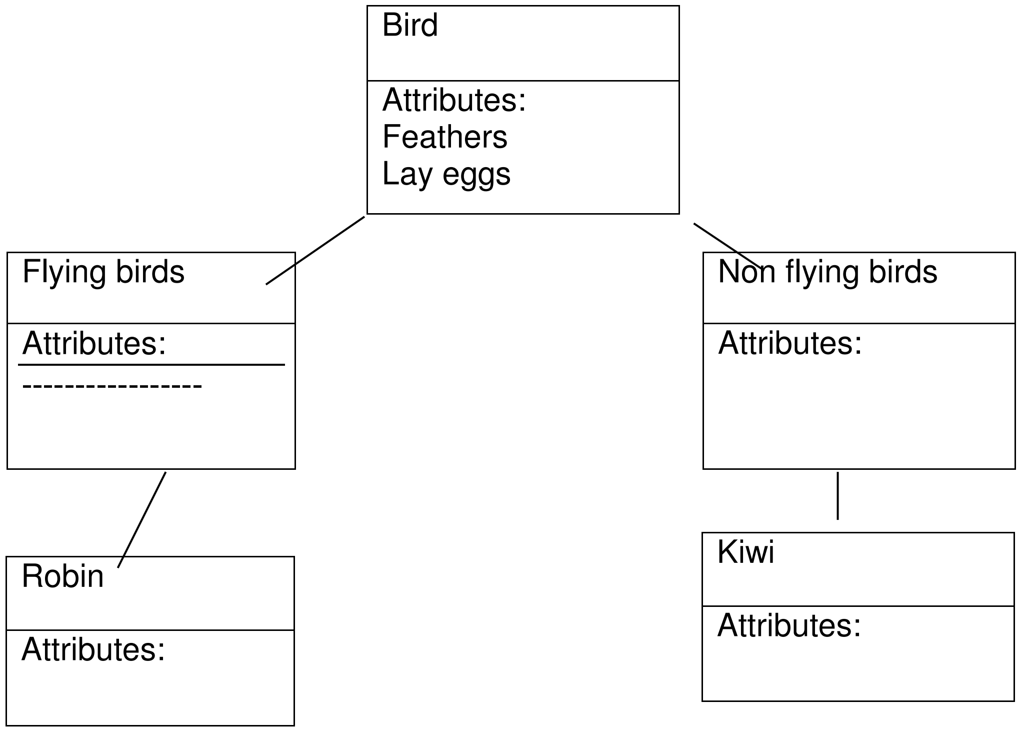
3. **Data abstraction and encapsulation:**

The wrapping up of data and its functions into a single unit (class) is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can assess it> these functions provide the interface between the objects data and the program> this insulation of data from direct access by the program is called **DATA HIDING (or data abstraction)**
Since the classes use the concept of data abstraction they are known as ABSTRACT DATA TYPES (ADT)

4. **Inheritance :**

In heritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example the bird **robin** is a part of the class **flying birds** which again a part of **bird**. As given in the diagram below each derived class shares common characteristics with the class from which it is derived.

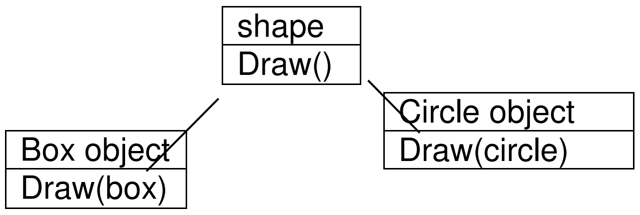
In **OOP**, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programer to reuse a class that is almost, but not exactly, what he wants.



5. Polymorphism:

Polymorphism means the ability to take more than one form. For example an operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example consider the operation addition. For two numbers, the operation will generate a sum . if the operands are strings, then the operation would produce a third string by concatenation.

Here in the below given diagram a single function draw () does different operation according to the behavior of the type derived. I.e. Draw () function works in different form.



Polymorphism plays an important role in allowing objects having internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensivel used in inheritance.

6. Dynamic binding

Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call is associated with a polymorphic reference depends on the dynamic type of that reference.

7. message communication

An object oriented program consists of a set of objects that communicate with each other. Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

Object

Object is an entity that can store data and send and receive messages. They are run time entities they may also represent user-defined data.

When a program is executed the object interacts by sending messages to one another.

Every object will have the data structure called attributes (or property or data) and behavior called operations.

Eg: Consider the object account

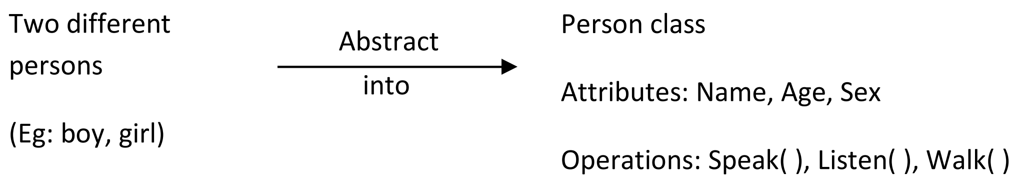
Structure	(General format)	Eg:	
Object Name		Account	
Attribute 1 Attribute 2 Attribute N	Structure	Account Number Account Type Name Balance	attribute
Operation 1	Behaviour	Deposit( )	Operation

Operation 2
Operation N
		Withdraw( )
		Enquire( )	

Classes:
The objects with the same data structure (attribute) and behaviour (operations) are grouped into a class. All these objects possessing similar properties and grouped into the same unit.

Eg: In the person class all person having similar attributes like Name, Age, Sex and the similar operations like speak, listen, walk. So, boy and girls objects are grouped into the person class.

This should be represented as person objects.


Two different persons
(Eg: boy, girl)
Abstract
into
Person class
Attributes: Name, Age, Sex
Operations: Speak( ), Listen( ), Walk( )

Representation of Class:

```
Class account
{
        private:
                char name[20];
                int accounttype;            Data members
                int accountnumber;
                float balance;

        public:
                Deposit( );
                Withdraw( );                Member functions
                Enquire( );
};
```

In this the account class groups the object such as saving account, current account, etc, Thus, objects having the same structural and behavioral propositions are grouped together to form a class.

The following points on classes can be noted:
1. A class is a template that unites data and operations.
2. A class is a abstraction of the real world entities with similar properties.

3. A class identifies a set of similar objects.

CLASSES AND OBJECTS:

Class Specification:

The class can be described as a collection of data member along with member functions. This property of C++, which allows association of data and functions into a single unit is called encapsulation. Sometimes classes may not contain any data members or member function called empty classes.

Syntax for class specification

Keyword

Userdefined name of the class

```
Class  classname
{
        //body of a class
};   end of class requires semicolon
```

More than one object can be created with a single statement as,

```
Class student s1,s2,s3;
        Or
Student s1,s2,s3.
```

Object can also be created by placing these names immediately after the closing brace. Thus the definition

```
Class  student
        {
                ---------------
                ---------------
        } s1,s2,s3;
```

Accessing class members:

Once an object of a class has been created, there must be a provision to access its members. This is achieved by using the member access operator, dot(.).

Syntax: for accessing datamember of a class

```
Object name . datamember
```

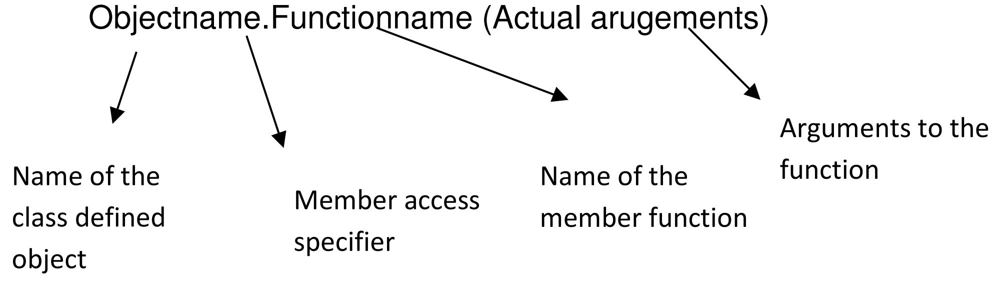
Datamember of a class

Name of the class defined object

Member access specifier

Syntax for accessing member function of a class

Objectname.Functionname (Actual arugements)


Name of the class defined object
Member access specifier
Name of the member function
Arguments to the function

Eg: s1.setdata(10, “Ram”);
    s1.outdata( );

The object s1 can be used to access the member functions setdata and outdata respectively.

Consider the following program

```
#include <iostream.h>
#include <string.h>
class student
{
    private:
        int roll_no;
        char name[20];
    public:
        void setdata(int roll_no_in, char name_in)
        {
                roll_no = roll_no_in;
                strcpy(name, name_in);
        }
        void outdata( )                 //display data members
        {
                cout<<”rollno = “<< roll_no <<endl;
                cout<<”name = “ << name << endl;
        }
};
void main( )
{
        student s1;
        s1.setdata(1, “Ram”);
        s2.setdata(10,”Kumar”);
        cout<<”Student details . . .”<<endl;
        s1.outdata( );
        s2.outdata( );
}
```

Output:

```
        Student details
        Rollno = 1
        Name = Ram
        Rollno = 10
        Name = Kumar
```

Client-server Model:

A client seeks service whereas, a server provides services requested by a client In the above example the class student resembles a server whereas, objects of the class student resemble clients. They make call to the server by sending messages.

In the statement

```
        s2.setdata(10, “kumar”);
```

The object s2 seeks the messages setdata to the server with the parameter 10 and kumar and server accept the messages and perform the operation and display the result whenver the client called the outdata( ) function.

Defining member function:

The data members of a class must be declared within the body of the class, whereas the member functions of the class can be defined in any one of the following ways.

- Inside the class specification
- Outside the class specification

The syntax of a member function definition changes depending on whether it is defined inside or outside the class specification, but it performs the same operation.

(a) **Member function inside the class body:**

All the member functions defined within the body of a class.
Eg:

```
        Class classname
        {
                private:
```

```
                int age;
                int setage(int agein);                  //member function
                {
                        age = agein;            //body of the function
                }
        ………..
        public:
                int b;
                void rect( )
                {
                    . . . .         // body of a function
                }
};
```

(b) **Member functions outside the classbody**

To declare function prototype within the body of a class and then define it outside the body of a class. This is done by using the 'scope resolution operator' (**::**). It acts as an identity-label to inform the compiler, the class to which the function belongs.

G.F or Syntax:

```
class classname
{
        . . . .
        Returntype memberfunction (arguments);                  //function declaration
        . . . .
};
returntype classname :: memberfunction (arguments)   //function definition
{
        //body of the function
}
```

Accessing member functions within the class

A member of a class is accessed by the objects of that class using the dot operator.

Ex:

```
#include <iostream.h>
class number
{
        int num1, num2;      //private by default
        public:
                void read( )
                {
                        cout<<"Enter first number: ");
                        cin>>num1;
                        cout<<"Enter second number: ");
                        cin>>num2;
                }
                int max( )
                {
                        if(num1>num2)
                                return num1;
                        else
                                return num2;
                }
                //Nesting of member function
                void showmax( )
                {
                        cout<<"maximum = "<<max( ) ;
                }
        };
        void main( )
        {
                number n1;
                n1.read( );
                n1.showmax( );
        }
```

Output:

```
Enter first number  : 5
Enter second number  : 10
Maximum = 10
```

This a member function of a class can call any other member function of its own class is called ‘nesting of member function’.

Inline function

One of the objective of usin g functions in a program is to save some memory space , which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack and returning to the calling function. When a function is small a substatial percentage of execution time may be spent in such overheads. C++ has a solution to this problem. To eliminate the cost of calls to small function C++ proposes a new feature called *inline function.*

An inline function is a function that is expanded in a line, this function keeps a request to compiler to give preference to execute it as a request, but it does not makes a commands to give that preference to execute.

An inline function can be defined as below

```
Inline function_header
{
function body
}
```

Example;

```
Inline int cube(int a)
{
return (a*a*a);
}
```

the above inline function can be invoked by the statement like;

```
c = cube(3);
d = cube(2+2);
```

the output of the above statements will be 27 and 64 for c and d respectively.
Remember that the **inline** keyword mearly sends a request , not a command to the compiler. The compiler may ignore this request if the the function definition is too long or too complicated and compile the function as a normal function.

Some of the situation where the **inline function may not work are:**

- For the function returning values, if a loop, a **switch**, or a **goto** exists.
- For function not returning values, if a return statement exists.
- If a function contains static variables.
- If **inline** function are recrusive.

Data Hiding:

Data is hidden inside a class, so the unauthorized access is not possible, which is the key feature of OOP.

All the data and functions defined in a class are private by default. Normally the data members are declared as private and member functions are declared as public.

Methods of Data hiding:

- Private
- Public
- Protected. These keywords are called access-control specifiers.

Private members:

In this only the member functions of the same class can access these members. The private members of a class are inaccessible outside the class.

```
Class person
{
        private:
                int age;                //private data
                int getage( ): //private function
                . . . .
}
person p1;
a = p1.age;             //cannot access private data //error
p1.getage( );           //Error access
```

ie., we can access the private members by using objects.

Protected member:

The access control of the protected members is similar to that of private members. The access control of protected members is shown below:

```
Class person
{
        protected:
        . . . .
        int age;                // protected data
        int getage( ): //protected function
                . . . .
};
person p1;      
a = p1.age;     }        Cannot access protected members
p1.getage( );   
```

Public Members

All data members and function declared in the public section of the class can be accessed without any restriction from anywhere in the program.

Eg:

```
Class person
```

```
{
        public:
                int age;                //public data
                int getage( );          //public function
}
person p1;
a =  p1.age;            // can access public data
p1.getage( );           // can access public function
```

Nesting of member function

A member function of a class can be called only by an object of that class using a dot operator. In nesting of member function, the member function can be called by using its name inside another member function of the same class is called nesting member function.

Consider the following example

```
#include <iostream.h>
class set
{
        int m,n;
        public:
                void input(void);
                void display(void);
                void largest(void);
};
int set :: largest(void)
{
if(m>=n)
        return(m);
else
        return(n);
}
void set:: input (void)
{
        cout<<"input values of m and n" << "\n";
}
void set:: display(void)
{
        cout<<"largest value" << largest( )<<"\n";
}
void main( )
{
        set A:
        A input( );
        A.set( );
}
```

Arrays within a class

The arrays can be used as member variables in a class. That is more than one related variable or data are grouped under the common name,

Eg:

```
Class abc
{
        int a[size];     //a is name of the array size – represents the size of the array
        public:
                void setdata(void);
                void display(void);
};
```

Empty classes (or Stubs)

Main reason for using a class is to encapsulate data and code, it is however, possible to have a class that has neither data nor code. In otherwords, it is possible to empty classes.

The declaration of empty classes is as follows:

```
Class xyz
{
};
Class abc
{
};
```

Such a empty classes are also called as stubs

Passing Objects arguments:

- It is possible to have functions which accept objects of a class as arguments, just as there are functions which accept other variables as arguments.
- An object can be passed as an argument to a function by the following ways:
    1.Passing object by value, a copy of the entire object is passed to the function
    2. Passing object by reference, only the address of the object is passed implicitly to the function.
    3. Passing object by pointer, the address of the object is passed explicitly to the function

- **Passing Object by value**

  In this case a copy of the object is passed to the function and any modifications made to the object inside the function are not reflected in the object used to call the function.

  Example:

```
#include<iostream.h>
class test
{
int m1,m2;
public:
void get()
{
cin>>m1>>m2;
}
void read(test t3)
{
m1=t3.m1;
m2=t3.m2;
}
void display()
{
cout<<m1<<m2;
}
};
void main()
{
        test t1;
        cout<<"Enter Ist object data";
        t1.get();
        cout<<"display Ist object data";
        t1.display();
        test t2;
        cout<<"copy of object1 to object2";
        t2. read(t1);
        cout<<"display 2nd object data";
        t2.display();
}
```

  Run:

```
Enter Ist object data
34
56
display Ist object data
34
56
copy of object1 to object2
```

```
display 2nd object data
34
56
```

The members of t1 are copied to t2. Any modification made to the data members of the objects t1 and t2 are not visible to the caller's actual parameter

- **Passing objects by Reference:**
  Accessibility of the objects passed reference is similar to those passed by value. Modifications carried out on such objects in the called function will also be reflected in the calling function.

Example:

```
#include<iostream.h>
class test
{
int m1,m2;
public:
void get()
{
cin>>m1>>m2;
}
void read(test &t3)
{
m1=t3.m1;
m2=t3.m2;
}
void display()
{
cout<<m1<<m2;
}
};
void main()
{
        test t1;
        cout<<"Enter Ist object data";
        t1.get();
        cout<<"display Ist object data";
        t1.display();
        test t2;
        cout<<"copy of object1 to object2";
        t2. read(t1);
        cout<<"display 2nd object data";
        t2.display();
}
```

Run:

```
Enter Ist object data
34
56
display Ist object data
34
56
copy of object1 to object2
display 2nd object data
34
56
```