CSC 313 - Part 1

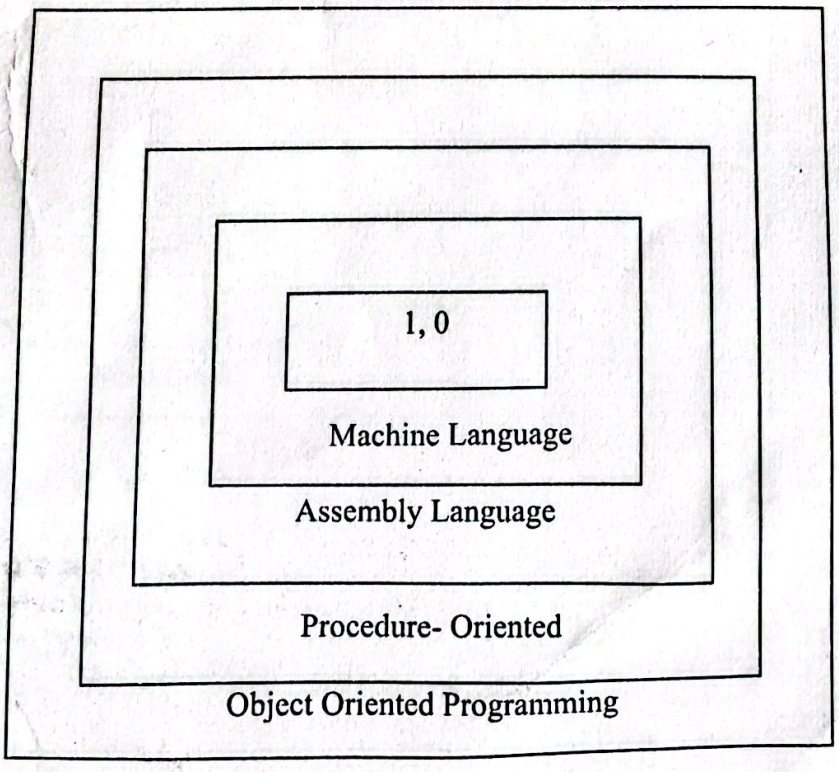
1.1 Software Crisis

Developments in software technology continue to be dynamic. New tools and techniques are announced in quick succession. This has forced the software engineers and industry to continuously look for new approaches to software design and development, and they are becoming more and more critical in view of the increasing complexity of software systems as well as the highly competitive nature of the industry. These rapid advances appear to have created a situation of crisis within the industry. The following issued need to be addressed to face the crisis:

- How to represent real-life entities of problems in system design?
- How to design system with open interfaces?
- How to ensure reusability and extensibility of modules?
- How to develop modules that are tolerant of any changes in future?
- How to improve software productivity and decrease software cost?
- How to improve the quality of software?
- How to manage time schedules?

1.2 Software Evaluation

Ernest Tello, A well known writer in the field of artificial intelligence, compared the evolution of software technology to the growth of the tree. Like a tree, the software evolution has had distinct phases "layers" of growth. These layers were building up one by one over the last five decades as shown in fig. 1.1, with each layer representing and improvement over the previous one. However, the analogy fails if we consider the life of these layers. In software system each of the layers continues to be functional, whereas in the case of trees, only the uppermost layer is functional



Alan Kay, one of the promoters of the object-oriented paradigm and the principal designer of Smalltalk, has said: "*As complexity increases, architecture dominates the basic materials*". To build today's complex software it is just not enough to put together a sequence of programming statements and sets of procedures and modules; we need to incorporate sound construction techniques and program structures that are easy to comprehend implement and modify.

With the advent of languages such as c, structured programming became very popular and was the main technique of the 1980's. Structured programming was a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired result in terms of bug-free, easy-to- maintain, and reusable programs.

Object Oriented Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily.

1.3 Procedure-Oriented Programming

In the procedure oriented approach, the problem is viewed as the sequence of things to be done such as reading, calculating and printing such as cobol, fortran and c. The primary focus is on functions. A typical structure for procedural programming is shown in fig.1.2. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

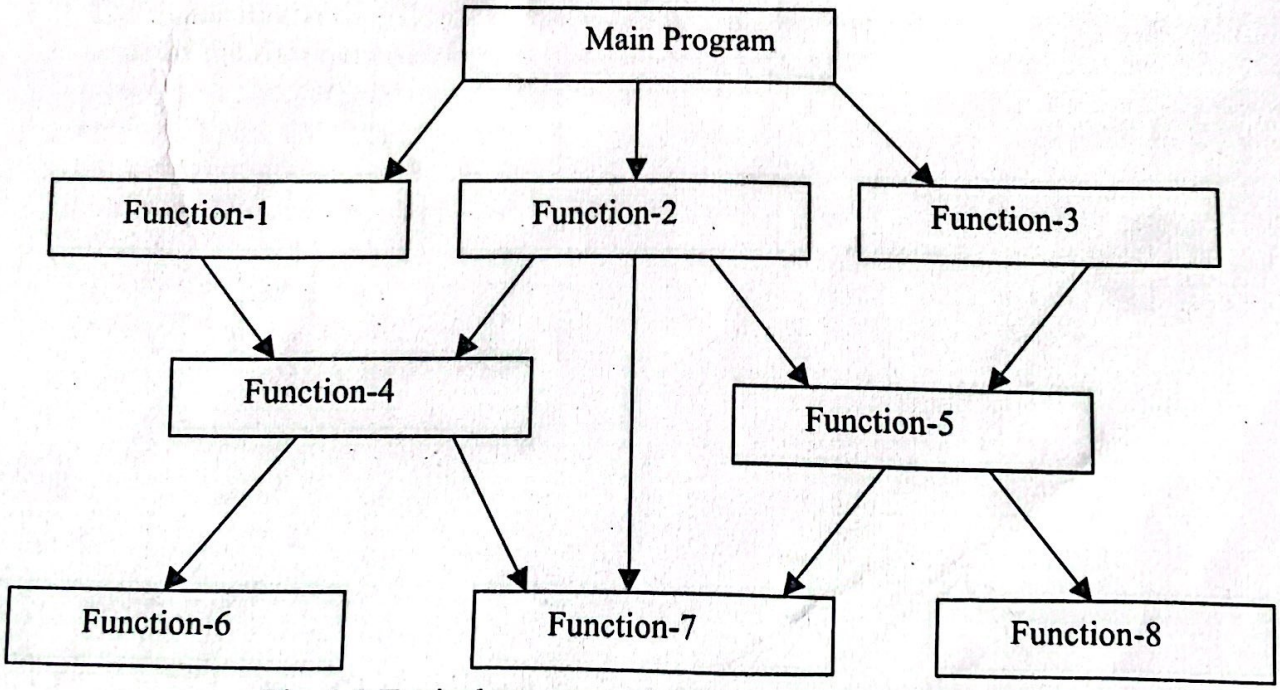


Fig. 1.2 Typical structure of procedural oriented programs

Procedure oriented programming basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as *functions*. We normally use flowcharts to organize these actions and represent the flow of control from one action to another.

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.

Another serious drawback with the procedural approach is that we do not model real world problems very well. This is because functions are action-oriented and do not really corresponding to the element of the problem.

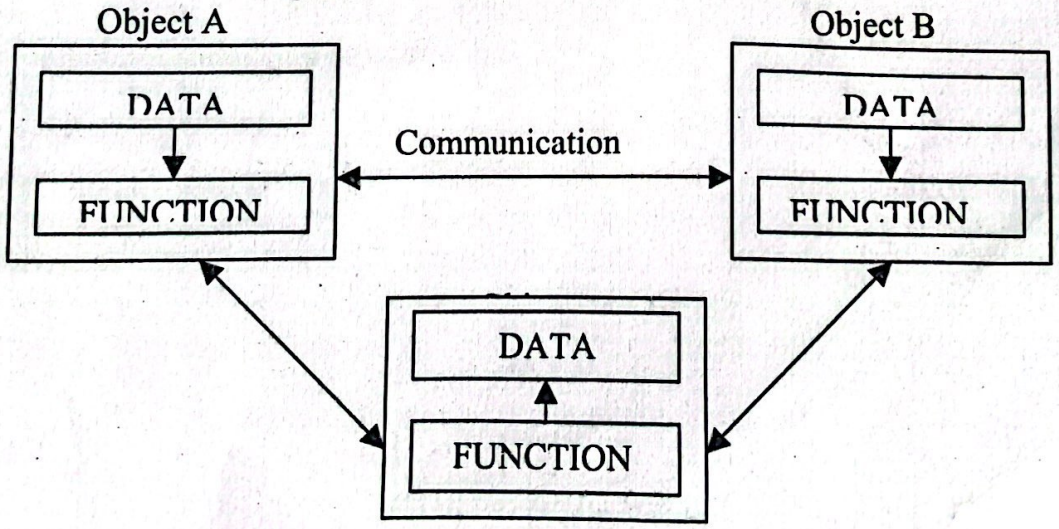
Some Characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

1.4 Object Oriented Paradigm

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The organization of data and function in object-oriented programs is shown in fig.1.3. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.

Organization of data and function in OOP



Some of the features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are ties together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people.

1.5 Basic Concepts of Object Oriented Programming

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

We shall discuss these concepts in some detail in this section.

1.5.1 Objects

Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in term of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in c.

When a program is executed, the objects interact by sending messages to one another. Foe example, if "customer" and "account" are to object in a program, then the customer object may send a message to the count object requesting for the bank balance. Each object contain data, and code to manipulate data. Objects can interact without having to know details of each other's data or code. It is a sufficient to know the type of message accepted, and the type of response returned by the objects. Although different author

represent them differently fig 1.5 shows two notations that are popularly used in object-oriented analysis and design.

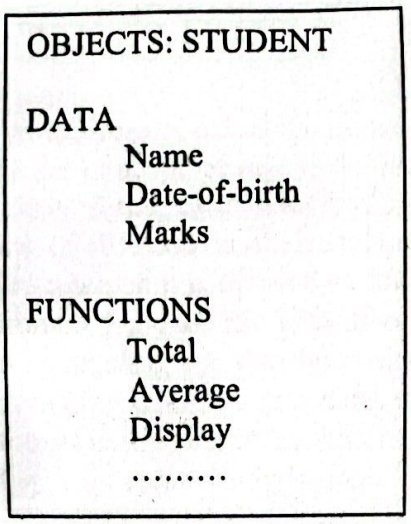


Fig. 1.5 representing an object

1.5.2 Classes

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects similar types. For examples, Mango, Apple and orange members of class fruit. Classes are user-defined that types and behave like the built-in types of a programming language. The syntax used to create an object is not different then the syntax used to create an integer object in C. If fruit has been defines as a class, then the statement

```
Fruit Mango;
```

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

1.5.3 Data Abstraction and Encapsulation

The wrapping up of data and function into a single unit (called class) is known as *encapsulation.* Data and encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding.*

Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all the essential properties of the object that are to be created.

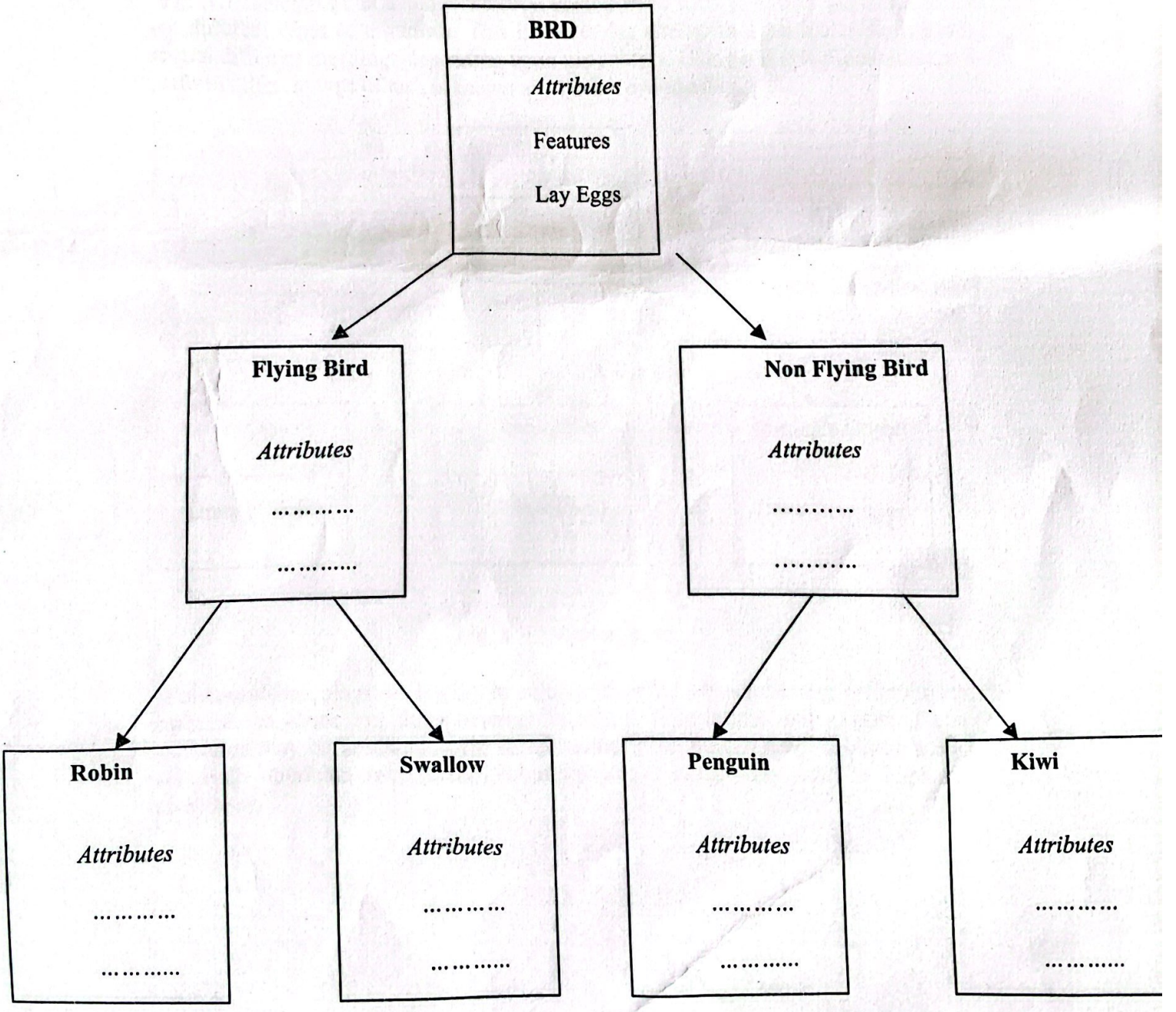
The attributes are some time called *data members* because they hold information. The functions that operate on these data are sometimes called *methods or member function.*

1.5.4 Inheritance

Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of *hierarchical classification.* For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in fig 1.6.

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 feature of both the classes. The real appeal and power of the inheritance mechanism is that it

Fig. 1.6 Property inheritances



Allows the programmer to reuse a class i.e almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduced any undesirable side-effects into the rest of classes.

1.5.5 Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than on form. An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of 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. The process of making an operator to exhibit different behaviors in different instances is known as *operator overloading*.

Fig. 1.7 illustrates that a single function name can be used to handle different number and different types of argument. This is something similar to a particular word having several different meanings depending upon the context. Using a single function name to perform different type of task is known as *function overloading*.

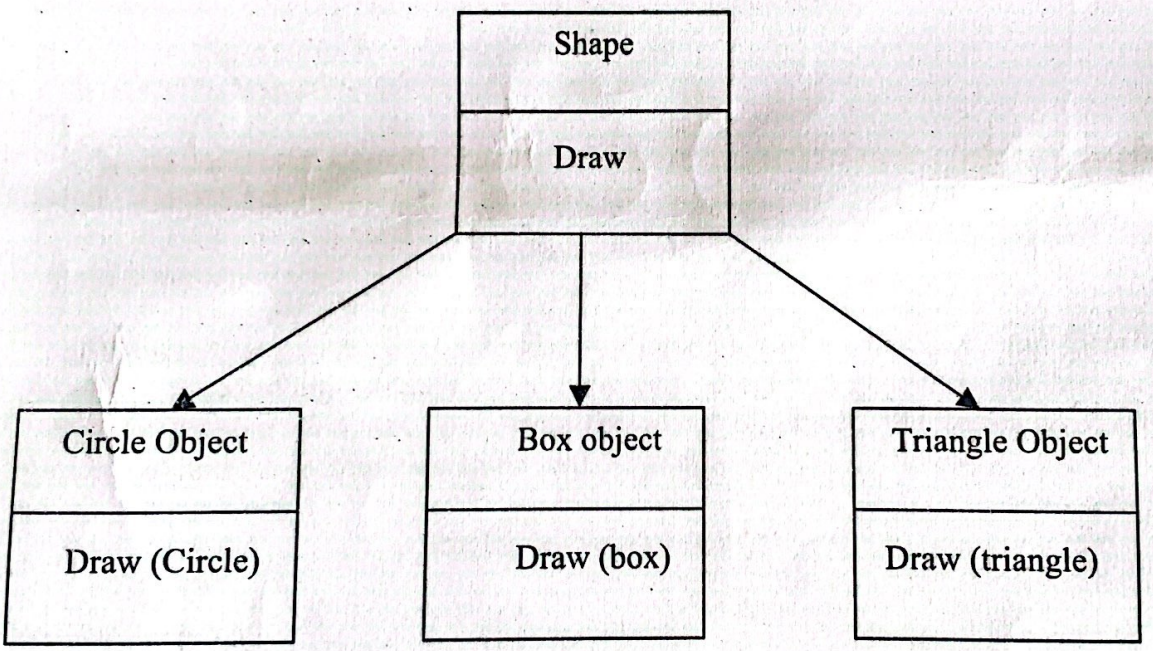


Fig. 1.7 Polymorphism

Polymorphism plays an important role in allowing objects having different 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 action associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

1.5.6 Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. 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 associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure "draw" in fig. 1.7. by inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

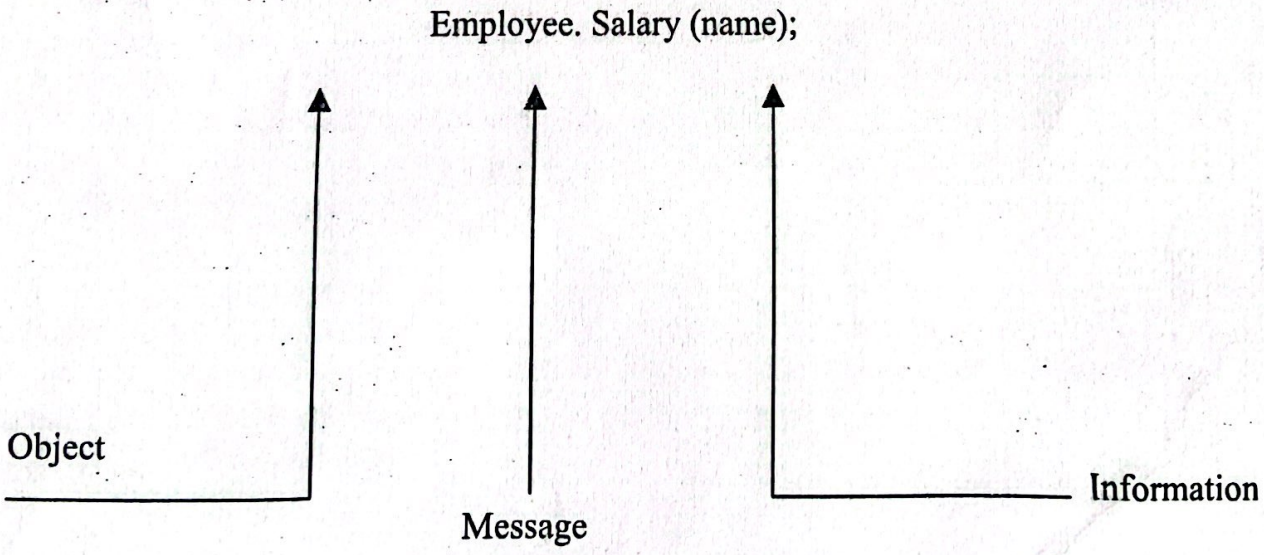
1.5.7 Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

1. Creating classes that define object and their behavior,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results. *Message passing* involves specifying the name of object, the name of the function (message) and the information to be sent. Example:



Object has a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

1.6 Benefits of OOP

OOP offers several benefits to both the program designer and the user. Object-Orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- Through inheritance, we can eliminate redundant code extend the use of existing
- Classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure program that can not be invaded by code in other parts of a programs.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map object in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more detail of a model can implemental form.
- Object-oriented system can be easily upgraded from small to large system.
- Message passing techniques for communication between objects makes to interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. There are a number of issues that need to be tackled to reap some of the benefits stated above. For instance, object libraries must be available for reuse. The technology is still developing and current product may be superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.

1.7 Object Oriented Language

Object-oriented programming is not the right of any particular languages. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially id designed to support the OOP concepts makes it easier to implement them.

Part 2

C++ Classes and Objects

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

For example, we defined the Box data type using the keyword class as follows:

```
class Box {
   public:
      double length;    // Length of a box
      double breadth;   // Breadth of a box
      double height;    // Height of a box
};
```

The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected**.

Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

The public data members of objects of a class can be accessed using the direct member access operator (.)

LET US MAKE THE EXAMPLE CLEAR

```
#include <iostream>

using namespace std;

class Box {
   public:
      double length;    // Length of a box
      double breadth;   // Breadth of a box
      double height;    // Height of a box
};

int main( ) {
   Box Box1;          // Declare Box1 of type Box
   Box Box2;          // Declare Box2 of type Box
```

```
    double volume = 0.0;      // Store the volume of a box here

    // box 1 specification
    Box1.height = 5.0;
    Box1.length = 6.0;
    Box1.breadth = 7.0;

    // box 2 specification
    Box2.height = 10.0;
    Box2.length = 12.0;
    Box2.breadth = 13.0;

    // volume of box 1
    volume = Box1.height * Box1.length * Box1.breadth;
    cout << "Volume of Box1 : " << volume <<endl;

    // volume of box 2
    volume = Box2.height * Box2.length * Box2.breadth;
    cout << "Volume of Box2 : " << volume <<endl;

    return 0;
}
```

.....

C++ Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base & Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

```
#include <iostream>

using namespace std;

// Base class
class Shape  {
   public:
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

int main(void) {
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
```

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Types of Inheritance

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

- **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Part 3

To write a program to find out the payroll system using single inheritance.

ALGORITHM:

Step 1: Start the program.
Step 2: Declare the base class emp.
Step 3: Define and declare the function get() to get the employee details.
Step 4: Declare the derived class salary.
Step 5: Declare and define the function get1() to get the salary details.
Step 6: Define the function calculate() to find the net pay.
Step 7: Define the function display().
Step 8: Create the derived class object.
Step 9: Read the number of employees.
Step 10: Call the function get(),get1() and calculate() to each employees.
Step 11: Call the display().
Step 12: Stop the program.

PROGRAM:PAYROLL SYSTEM USING SINGLE INHERITANCE

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

class emp
{
   public:
     int eno;
     char name[20],des[20];
     void get()
     {
              cout<<"Enter the employee number:";
              cin>>eno;
              cout<<"Enter the employee name:";
              cin>>name;
              cout<<"Enter the designation:";
              cin>>des;
     }
};

class salary:public emp
{
     float bp,hra,da,pf,np;
   public:
     void get1()
     {
              cout<<"Enter the basic pay:";
              cin>>bp;
```

```
                cout<<"Enter the Humen Resource Allowance:";
                cin>>hra;
                cout<<"Enter the Dearness Allowance :";
                cin>>da;
                cout<<"Enter the Profitablity Fund:";
                cin>>pf;
        }
        void calculate()
        {
                np=bp+hra+da-pf;
        }
        void display()
        {
                cout<<eno<<"\t"<<name<<"\t"<<des<<"\t"<<bp<<"\t"<<
hra<<"\t"<<da<<"\t"<<pf<<"\t"<<np<<"\n";
        }
};

void main()
{
     int i,n;
     char ch;
     salary s[10];
     clrscr();
     cout<<"Enter the number of employee:";
     cin>>n;
     for(i=0;i<n;i++)
     {
                s[i].get();
                s[i].get1();
                s[i].calculate();
     }
     cout<<"\ne_no \t e_name\t des \t bp \t hra \t da \t pf \t np
\n";
     for(i=0;i<n;i++)
     {
                s[i].display();
     }
     getch();
}
```

Output

Enter the Number of employee:1
Enter the employee No: 150

Enter the employee Name: ram
Enter the designation: Manager
Enter the basic pay: 5000
Enter the HR allowance: 1000
Enter the Dearness allowance: 500
Enter the profitability Fund: 300

E.No	E.name	des	BP	HRA	DA	PF	NP
150	ram	Manager	5000	1000	500	300	6200

Part A

To find out the student details using multiple inheritance.

ALGORITHM:

Step 1: Start the program.
Step 2: Declare the base class student.
Step 3: Declare and define the function get() to get the student details.
Step 4: Declare the other class sports.
Step 5: Declare and define the function getsm() to read the sports mark.
Step 6: Create the class statement derived from student and sports.
Step 7: Declare and define the function display() to find out the total and average.
Step 8: Declare the derived class object,call the functions get(),getsm() and display().
Step 9: Stop the program.

Program: student details to show Multiple Inheritance

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

class student
{
    protected:
       int rno,m1,m2;
    public:
                 void get()
               {
                              cout<<"Enter the Roll no :";
                              cin>>rno;
                              cout<<"Enter the two marks   :";
                              cin>>m1>>m2;
               }
};
class sports
{
    protected:
       int sm;                        // sm = Sports mark
    public:
                void getsm()
              {
                 cout<<"\nEnter the sports mark :";
                 cin>>sm;

              }
};
class statement:public student,public sports
{
```

```
    int tot,avg;
    public:
    void display()
              {
                 tot=(m1+m2+sm);
                 avg=tot/3;
                                        cout<<"\n\n\tRoll   No        :
"<<rno<<"\n\tTotal      : "<<tot;
                cout<<"\n\tAverage    : "<<avg;
              }
};
void main()
{
   clrscr();
   statement obj;
   obj.get();
   obj.getsm();
   obj.display();
   getch();
}
```

Output:

Enter the Roll no: 100

Enter two marks

90
80

Enter the Sports Mark: 90

Roll No: 100
Total : 260
Average: 86.66

Part 5

Data abstraction

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

Now, if we talk in terms of C++ Programming, C++ classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this:

```
#include<iostream>

usingnamespace std;

int main(){

cout <<"Hello C++"<<endl;

return0;

}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout is free to change.

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:

Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.

Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction

Data abstraction provides two important advantages:

Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.

The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

Data Abstraction Example

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example:

```
#include<iostream>

usingnamespace std;

classAdder{

        public: // constructor

        Adder(int i =0){

        total = i;

        }

        // interface to outside world
```

```
   void addNum(int number){
   total += number;
   }
   // interface to outside world
   int getTotal(){
   return total;
   };
   private:
   // hidden data from outside world
    int total;
};
int main(){
Adder a;
a.addNum(10);
a.addNum(20);
a.addNum(30);
cout <<"Total "<< a.getTotal()<<endl;
return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.