

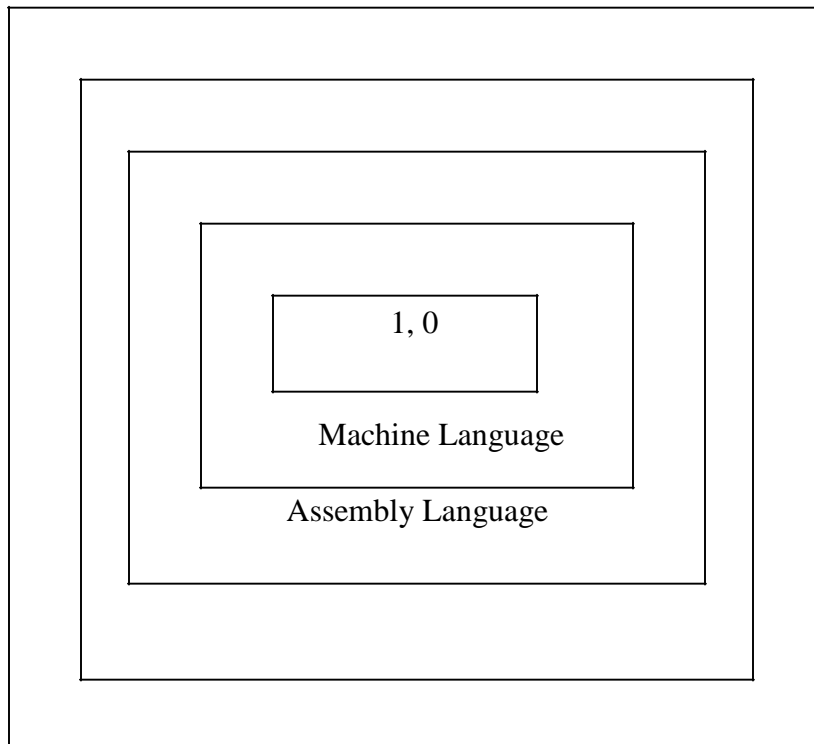
## Software Crisis

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 issues 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?

## 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.

## 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.

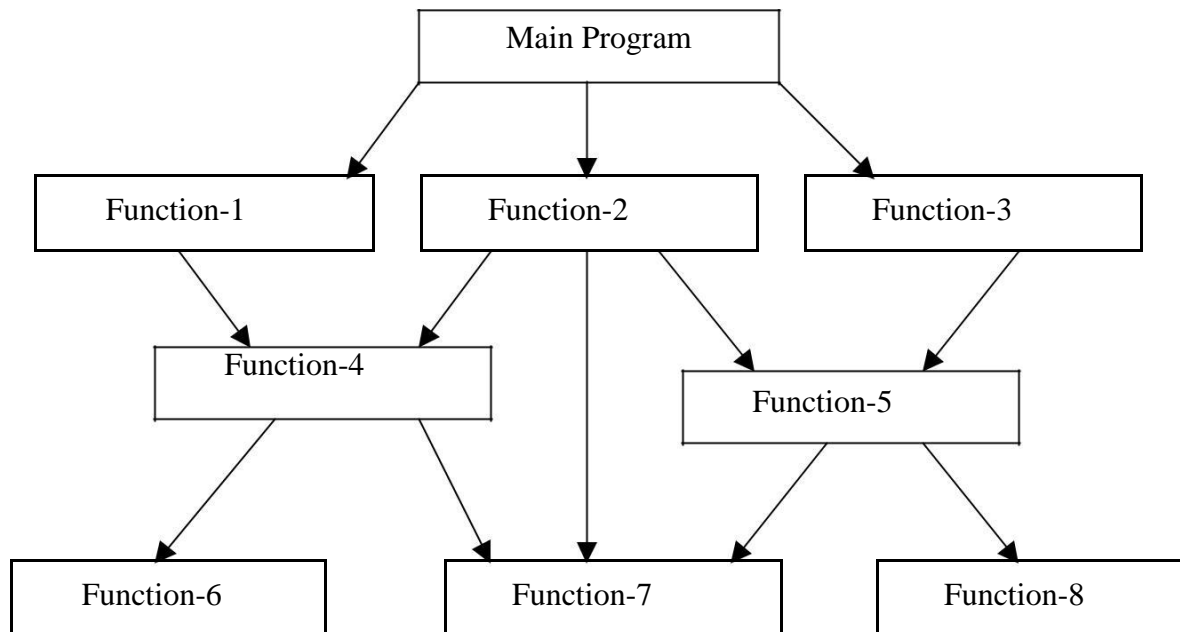


Fig. 1.2 Typical structure of procedural oriented programs

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.

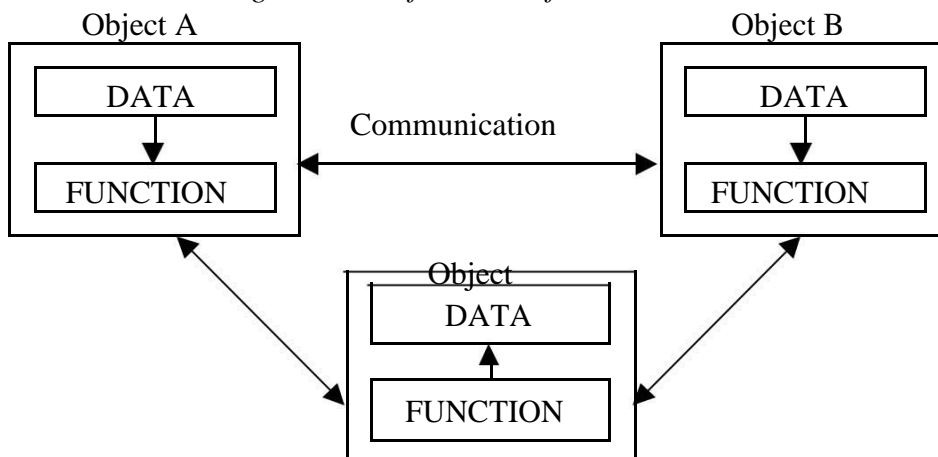
Some Characteristics exhibited by procedure-oriented programming are:

- 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.

## 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 tied 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.

## **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

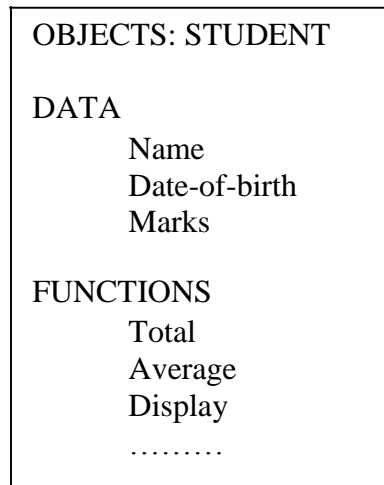
We shall discuss these concepts in some detail in this section.

## **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 terms 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. For example, if “customer” and “account” are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data, and code to manipulate 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. Although different authors

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*

## 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**.

## 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.

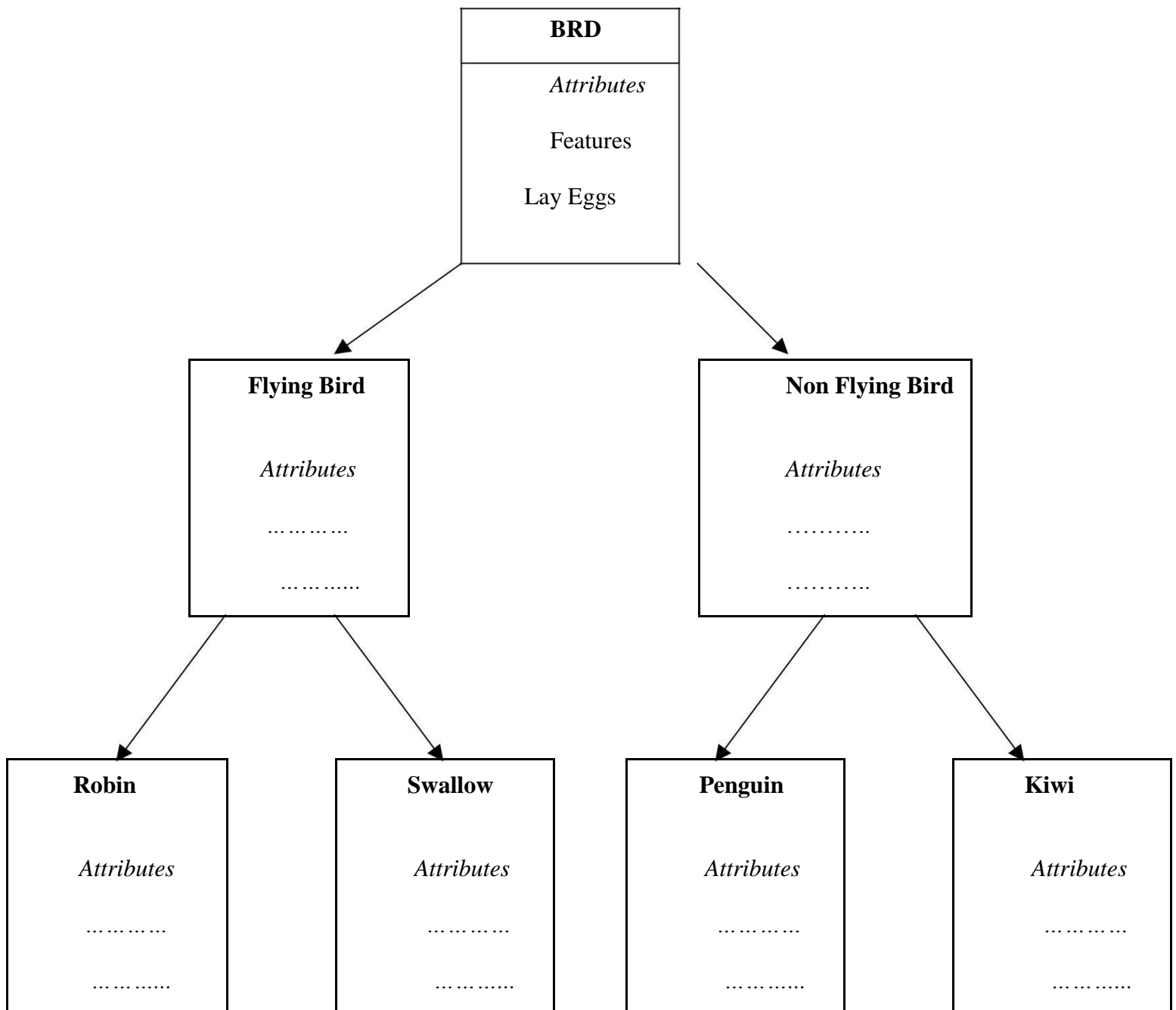
The attributes are some time called *data members* because they hold information. The functions that operate on these data are sometimes called *methods or member function*.

## 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.

## 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*.

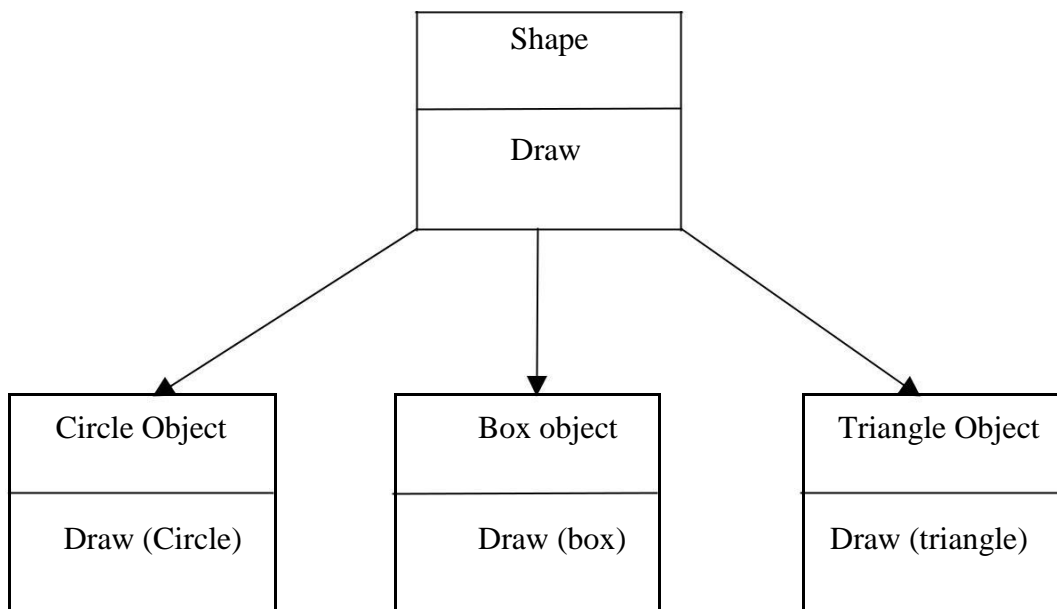


Fig. 1.7 Polymorphism

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.

## 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.

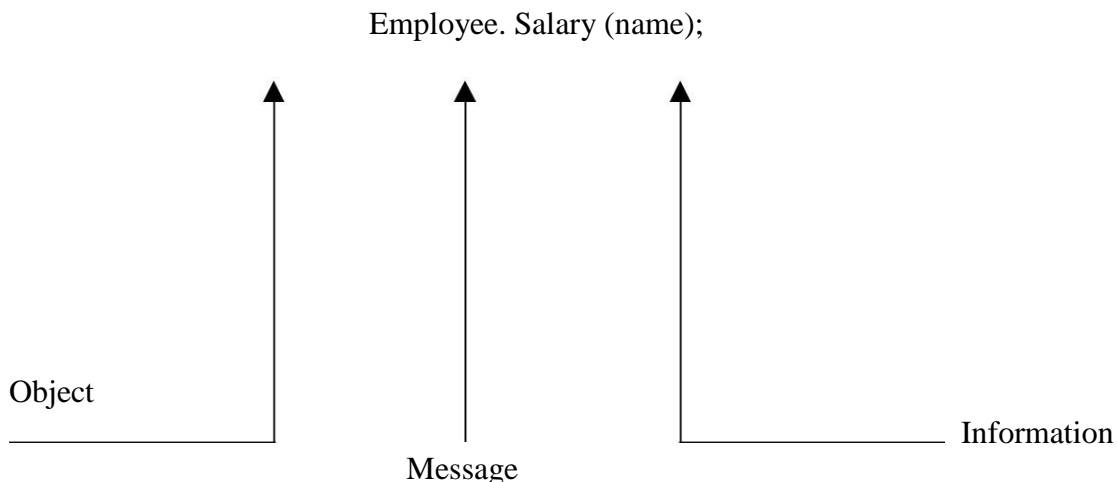
## Message Passing

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.

## **Benefits of OOP**

OOP offers several benefits to both the program designer and the user. Object-Oriented 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.

## **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.

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

1. Object-based programming languages, and
2. Object-oriented programming languages.

*Object-based programming* is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object based programming are:

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

## **Application of OOP**

- Real-time system
- Simulation and modeling
- Object-oriented data bases
- Hypertext, Hypermedia, and experttext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

## **Introduction of C++**

C++ is a superset of C. Almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ compiler. We shall see these differences later as and when they are encountered.

The most important facilities that C++ adds on to C are classes, inheritance, function overloading and operator overloading. These features enable creating of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

## **Application of C++**

- Since C++ allow us to create hierarchy related objects, we can build special object-oriented libraries which can be used later by many programmers.
- While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get closed to the machine-level details.
- C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.
- It is expected that C++ will replace C as a general-purpose language in the near future.

## **Simple C++ Program**

Let us begin with a simple example of a C++ program that prints a string on the screen.

### Printing A String

```
#include<iostream>
Using namespace std;
int main()
{
cout<<" c++ is better than c \n";
return 0;
}
```

This simple program demonstrates several C++ features.

## INTRODUCTION

Functions are the building blocks of C++ programs where all the program activity occurs. Function is a collection of declarations and statements.

### Need for a Function

Monoethic program (a large single list of instructions) becomes difficult to understand. For this reason functions are used. A function has a clearly defined objective (purpose) and a clearly defined interface with other functions in the program. Reduction in program size is another reason for using functions. The functions code is stored in only one place in memory, even though it may be executed as many times as a user needs.

The following program illustrates the use of a function :

```
//to display general message using function
#include<iostream.h>
include<conio.h>
void main()
{
    void disp(); //function prototype
    clrscr(); //clears the screen
    disp(); //function call
    getch(); //freeze the monitor
}
```

```

//function definition

void disp()

{

cout<<"Welcome to the GJU of S&T\n";

cout<<"Programming is nothing but logic implementation";

}

```

#### **PROGRAM 4.1**

In this Unit, we will also discuss Class, as important Data Structure of C++. A Class is the backbone of Object-Oriented Computing. It is an abstract data type. We can declare and define data as well as functions in a class. An object is a replica of the class to the exception that it has its own name. A class is a data type and an object is a variable of that type. Classes and objects are the most important features of C++. The class implements OOP features and ties them together.

### **FUNCTION DEFINITION AND DECLARATION**

In C++, a function must be defined prior to its use in the program. The function definition contains the code for the function. The function definition for display\_message () in program 6.1 is given below the main () function. The general syntax of a function definition in C++ is shown below:

```

Type name_of_the_function (argument list)

{

    //body of the function

}

```

Here, the type specifies the type of the value to be returned by the function. It may be any valid C++ data type. When no type is given, then the compiler returns an integer value from the function.

Name\_of\_the\_function is a valid C++ identifier (no reserved word allowed) defined by the user and it can be used by other functions for calling this function.

Argument list is a comma separated list of variables of a function through which the function may receive data or send data when called from other function. When no parameters,

the argument list is empty as you have already seen in program 6.1. The following function illustrates the concept of function definition :

```
//function definition add()
void add()
{
    int a,b,sum;
    cout<<"Enter two integers"<<endl;
    cin>>a>>b;
    sum=a+b;
    cout<<"\nThe sum of two numbers is "<<sum<<endl;
}
```

The above function add ( ) can also be coded with the help of arguments of parameters as shown below:

```
//function definition add()
void add(int a, int b) //variable names are must in definition
{
    int sum;
    sum=a+b;
    cout<<"\nThe sum of two numbers is "<<sum<<endl;
}
```

## ARGUMENTS TO A FUNCTION

Arguments(s) of a function is (are) the data that the function receives when called/invoked from another function.

### PASSING ARGUMENTS TO A FUNCTION

It is not always necessary for a function to have arguments or parameters. The functions **add ( )** and **divide ( )** in program 6.3 did not contain any arguments. The following example illustrates the concept of passing arguments to function **SUMFUN ( )**:

```
// demonstration of passing arguments to a function
```

```

#include<iostream.h>

void main ()
{
    float x,result; //local variables
    int N;

    float SUMFUN(float x, int N); //function declaration
    .....
    .....
    result = SUMFUN(X,N); //function declaration
    .....
}

//function SUMFUN() definition

float SUMFUN(float x,int N) //function declaration
{
    .....
    .....
    .....
}

```

Diagram annotations:

- Arrows from "formal parameters" point to `float x` and `int N` in the function declaration.
- An arrow from "Semicolon here" points to the semicolon in `float SUMFUN(float x, int N);`.
- An arrow from "return type" points to `float` in `float SUMFUN`.
- An arrow from "No semicolon here" points to the semicolon in `float SUMFUN(float x,int N);`.
- An arrow from "Body of the function" points to the curly braces of the function definition.
- An arrow from "No semicolon here" points to the closing brace of the `main` function.

## DEFAULT ARGUMENTS

**C++ allows a function to assign a parameter the default value in case no argument for that parameter is specified in the function call. For example.**

```
// demonstrate default arguments function
```



```

#include<iostream.h>

int calc(int U)
{
    If (U % 2 == 0)

        return U+10;
    Else
        return U+2
}

Void pattern (char M, int B=2)
{
    for (int CNT=0;CNT<B; CNT++)
        cout<<calc(CNT) <<M;
        cout<<endl;
}

Void main ()
{
    Pattern('*');
    Pattern('#',4)
    Pattern(';@',3);
}

```

## CONSTANT ARGUMENTS

A C++ function may have constant arguments(s). These arguments(s) is/are treated as constant(s). These values cannot be modified by the function.

For making the arguments(s) constant to a function, we should use the keyword **const** as given below in the function prototype :

```
Void max(const float x, const float y, const float z);
```

Here, the qualifier **const** informs the compiler that the arguments(s) having **const** should not be modified by the function `max()`. These are quite useful when call by reference method is used for passing arguments.

## CALLING FUNCTIONS

In C++ programs, functions with arguments can be invoked by :

- (a) *Value*
- (b) *Reference*

**Call by Value:** - In this method the values of the actual parameters (appearing in the function call) are copied into the formal parameters (appearing in the function definition), i.e., the function creates its own copy of argument values and operates on them. The following program illustrates this concept :

```
//calculation of compound interest using a function
#include<iostream.h>
#include<conio.h>
#include<math.h> //for pow() function
Void main()
{
    Float principal, rate, time; //local variables
    Void calculate (float, float, float); //function
prototype clrscr();
    Cout<<"\nEnter the following values:\n";
    Cout<<"\nPrincipal:";
    Cin>>principal;
    Cout<<"\nRate of interest:";
    Cin>>rate;
    Cout<<"\nTime period (in yeasers)
:"; Cin>>time;
    Calculate (principal, rate, time); //function call
```

```

    Getch ();
}

//function definition calculate()
Void calculate (float p, float r, float t)
{
    Float interest; //local variable Interest =
    p* (pow((1+r/100.0),t))-p; Cout<<"\nCompound
    interest is : "<<interest; }

```

**Call by Reference:** - A reference provides an alias – an alternate name – for the variable, i.e., the same variable's value can be used by two different names : the original name and the alias name.

In call by reference method, a reference to the actual arguments(s) in the calling program is passed (only variables). So the called function does not create its own copy of original value(s) but works with the original value(s) with different name. Any change in the original data in the called function gets reflected back to the calling function.

It is useful when you want to change the original variables in the calling function by the called function.

```

//Swapping of two numbers using function call by reference
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();
    int num1,num2;
    void swap (int &, int &); //function prototype
    cin>>num1>>num2;
    cout<<"\nBefore swapping:\nNum1: "<<num1;
    cout<<endl<<"num2: "<<num2;

```

```

        swap(num1,num2); //function call

        cout<<"\n\nAfter swapping : \Num1: "<<num1;

        cout<<endl<<"num2: "<<num2; getch();

    }

    //function fefinition swap()
    void swap (int & a, int & b)
    {

        Int temp=a;

        a=b;

        b=temp;

    }

```

## INLINE FUNCTIONS

These are the functions designed to speed up program execution. An inline function is expanded (i.e. the function code is replaced when a call to the inline function is made) in the line where it is invoked. You are familiar with the fact that in case of normal functions, the compiler have to jump to another location for the execution of the function and then the control is returned back to the instruction immediately after the function call statement. So execution time taken is more in case of normal functions. There is a memory penalty in the case of an inline function.

The system of inline function is as follows :

```

inline function_header

{
    body of the function
}

```

For example,

```

//function definition min()

inline void min (int x, int y)

    cout<< (x < Y? x : y);

}

```

```

Void main()
{
    int num1, num2;
    cout<<"\nEnter the two intergers\n";
    cin>>num1>>num2;
    min (num1,num2; //function code inserted here
    -----
    -----
}

```

An inline function definition must be defined before being invoked as shown in the above example. Here min ( ) being inline will not be called during execution, but its code would be inserted into main ( ) as shown and then it would be compiled.

If the size of the inline function is large then heavy memory pentaly makes it not so useful and in that case normal function use is more useful.

#### **The inlining does not work for the following situations :**

1. For functions returning values and having a *loop* or a *switch* or a *goto* statement.
2. For functions that do not return value and having a return statement.
3. For functions having static variable(s).
4. If the inline functions are recursive (i.e. a function defined in terms of itself).

#### **The benefits of inline functions are as follows :**

1. Better than a macro.
2. Function call overheads are eliminated.
3. Program becomes more readable.
4. Program executes more efficiently.

## **SCOPE RULES OF FUNCTIONS AND VARIABLES**

The scope of an identifier is that part of the C++ program in which it is accessible. Generally, users understand that the name of an identifier must be unique. It does not mean that a name can't be reused. We can reuse the name in a program provided that there is some scope by which it can be distinguished between different cases or instances.

In C++ there are four kinds of scope as given below :

1. Local Scope
2. Function Scope
3. File Scope
4. Class Scope

**Local Scope:-** A block in C++ is enclosed by a pair of curly braces i.e., '{' and '}'. The variables declared within the body of the block are called **local variables** and can be used only within the block. These come into existence when the control enters the block and get destroyed when the control leaves the closing brace. You should note the variable(s) is/are available to all the enclosed blocks within a block.

For example,

```
int x=100;

    { cout<<x<<endl;

        Int x=200;

        {

            cout<<x<<endl;

            int x=300;

            {

                cout<<x<<endl;

            }

        }

        cout<<x<<endl;

    }
```

**Function Scope :** It pertains to the labels declared in a function i.e., a label can be used inside the function in which it is declared. So we can use the same name labels in different functions.

For example,

```
//function definition add1()

void add1(int x,int y,int z)
{
    int sum = 0;
    sum = x+y+z;
    cout<<sum;
}

//function definition add2()
coid add2(float x,float y,float z)
{
    Float sum = 0.0;
    sum = x+y+z;
    cout<<sum;
}
```

Here the labels x, y, z and sum in two different functions add1 ( ) and add2 ( ) are declared and used locally.

**File Scope :** If the declaration of an identifier appears outside all functions, it is available to all the functions in the program and its scope becomes file scope. For Example,

```
int x;

void square (int n)
{
    cout<<n*n;
}

void main ()
{
    int num;
```

```

.....
cout<<x<<endl;

cin>>num;

squaer (num) ;

.....
}

```

Here the declarations of variable **x** and function **square ( )** are outside all the functions so these can be accessed from any place inside the program. Such variables/functions are called global.

**Class Scope :** In C++, every class maintains its won associated scope. The class members are said to have local scope within the class. If the name of a variable is reused by a class member, which already has a file scope, then the variable will be hidden inside the class. Member functions also have class scope.

## DEFINITION AND DECLARATION OF A CLASS

A class in C++ combines related data and functions together. It makes a data type which is used for creating objects of this type.

Classes represent real world entities that have both data type properties (characteristics) and associated operations (behavior).

The syntax of a class definition is shown below :

```

Class name_of _class
{
private    : variable declaration; // data member
           Function declaration; // Member Function (Method)

protected: Variable declaration;
           Function declaration;

public    : variable declaration;
           Function declaration;

};

```



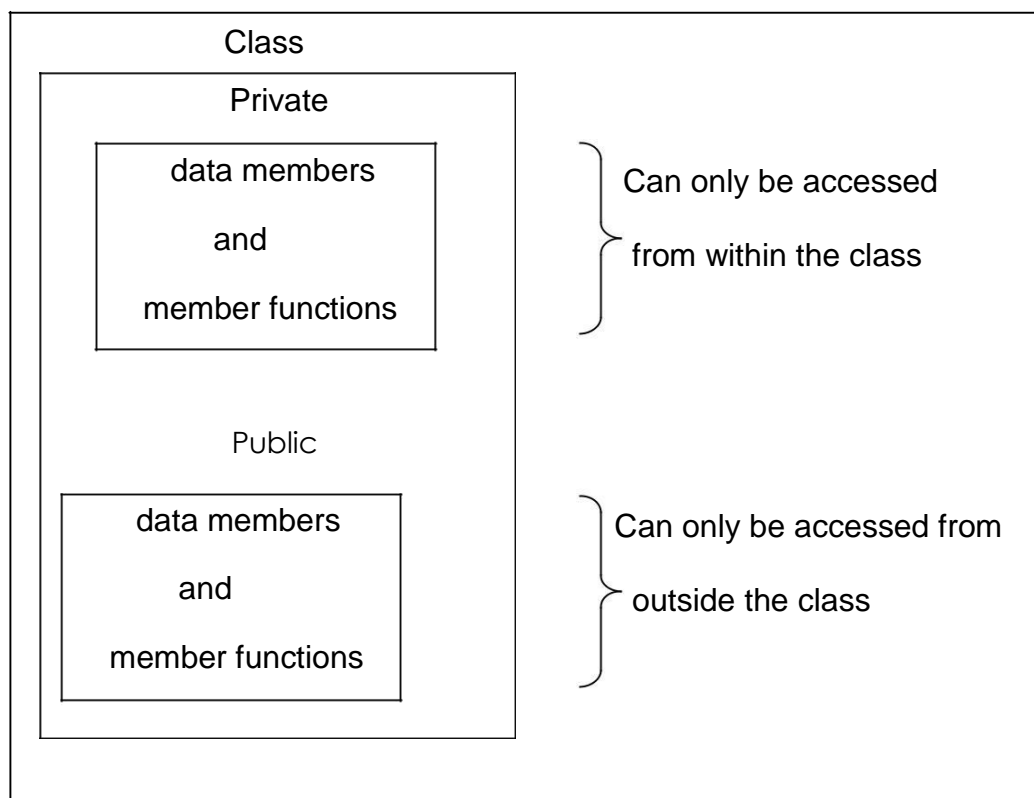
Here, the keyword `class` specifies that we are using a new data type and is followed by the class name.

The body of the class has two keywords namely :

(i) *private*

(ii) *public*

In C++, the keywords **private** and **public** are called access specifiers. The data hiding concept in C++ is achieved by using the keyword **private**. Private data and functions can only be accessed from within the class itself. Public data and functions are accessible outside the class also. This is shown below :



Data hiding not mean the security technique used for protecting computer databases. The security measure is used to protect unauthorized users from performing any operation (read/write or modify) on the data.

The data declared under **Private** section are hidden and safe from accidental manipulation. Though the user can use the private data but not by accident.

The functions that operate on the data are generally **public** so that they can be accessed from outside the class but this is not a rule that we must follow.

## MEMBER FUNCTION DEFINITION

The class specification can be done in two part :

- (i) **Class definition.** It describes both data members and member functions.
- (ii) **Class method definitions.** It describes how certain class member functions are coded.

We have already seen the class definition syntax as well as an example.

In C++, the member functions can be coded in two ways :

- (a) *Inside class definition*
- (b) *Outside class definition using scope resolution operator (::)*

The code of the function is same in both the cases, but the function header is different as explained below :

### Inside Class Definition:

When a member function is defined inside a class, we do not require to place a membership label along with the function name. We use only small functions inside the class definition and such functions are known as **inline** functions.

In case of inline function the compiler inserts the code of the body of the function at the place where it is invoked (called) and in doing so the program execution is faster but memory penalty is there.

### Outside Class Definition Using Scope Resolution Operator (::) :

In this case the function's full name (qualified\_name) is written as shown:

```
Name_of_the_class :: function_name
```

The syntax for a member function definition outside the class definition is :

```
return_type name_of_the_class::function_name (argument list)
{
    body of function
}
```

Here the operator::known as scope resolution operator helps in defining the member function outside the class. Earlier the scope resolution operator(::)was used in situations where a global variable exists with the same name as a local variable and it identifies the global variable.

## DECLARATION OF OBJECTS AS INSTANCES OF A CLASS

The objects of a class are declared after the class definition. One must remember that a class definition does not define any objects of its type, but it defines the properties of a class. For utilizing the defined class, we need variables of the class type. For example,

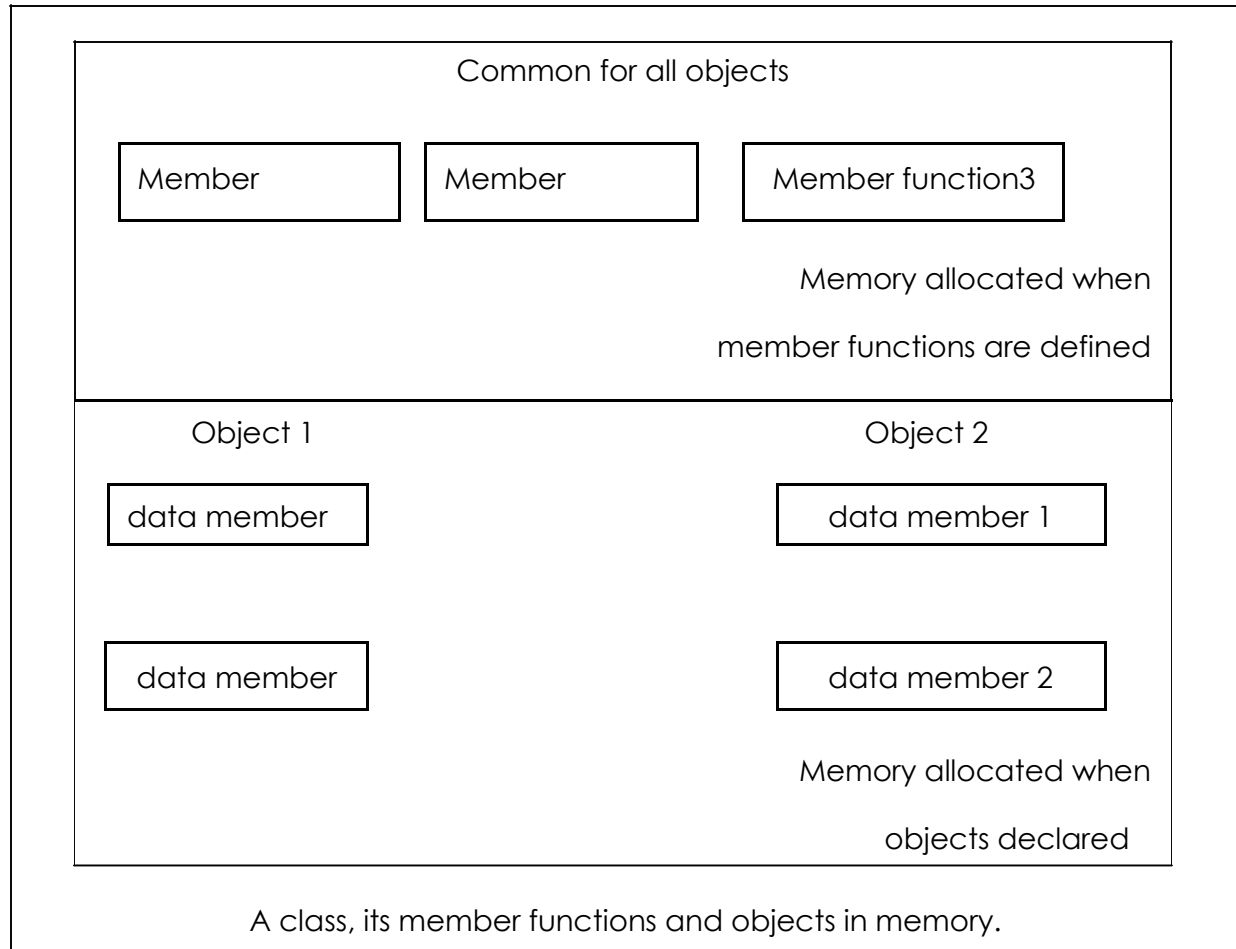
```
Largest ob1,ob2; //object declaration
```

will create two objects **ob1** and **ob2 of largest** class type. As mentioned earlier, in C++ the variables of a class are known as objects. These are declared like a simple variable i.e., like fundamental data types.

In C++, all the member functions of a class are created and stored when the class is defined and this memory space can be accessed by all the objects related to that class.

Memory space is allocated separately to each object for their data members. Member variables store different values for different objects of a class.

The figure shows this concept



## ACCESSING MEMBERS FROM OBJECT(S)

After defining a class and creating a class variable i.e., object we can access the data members and member functions of the class. Because the data members and member functions are parts of the class, we must access these using the variables we created. For functions are parts of the class, we must access these using the variable we created. For Example,

```
Class student
{
    private:
        char reg_no[10];
        char name[30];
        int age;
        char address[25];
```

```

public :
    void init_data()
    {
        - - - - - //body of function
        - - - - -
    }
    void display_data()
    {
};

student ob; //class variable (object) created
- - - - -
- - - - -

Ob.init_data(); //Access the member function
ob.display_data(); //Access the member
function - - - - -
- - - - -

```

Here, **the data members can be accessed in the member functions** as these have **private** scope, and the member functions can be accessed outside the class i.e., before or after the main() function.

## STATIC CLASS MEMBERS

Data members and member functions of a class in C++, may be qualified as static. We can have static data members and static member function in a class.

**4.11.1 Static Data Member:** It is generally used to store value common to the whole class. The **static** data member differs from an ordinary data member in the following ways :

- (i) Only a single copy of the static data member is used by all the objects.
- (ii) It can be used within the class but its lifetime is the whole program. For making a data member static, we require :
  - (a) Declare it within the class.
  - (b) Define it outside the class.

For example

```
Class student
{
    Static int count; //declaration within class
    -----
    -----
    -----
};
```

The static data member is defined outside the class as :

```
int student :: count; //definition outside class
```

**The definition outside the class is a must.**

We can also initialize the static data member at the time of its definition as:

```
int student :: count = 0;
```

If we define three objects as : student obj1, obj2, obj3;

**4.11.2 Static Member Function:** A static member function can access only the static members of a class. We can do so by putting the keyword static before the name of the function while declaring it for example,

```
Class student
{
    Static int count;
    -----

    public :
        -----
        -----

    static void showcount (void) //static member function
    {
        Cout<<"count="<<count<<"\n";
    }
}
```

```

    }

};

int student ::count=0;

```

Here we have put the keyword static before the name of the function shwocount ().

In C++, a static member function differs from the other member functions in the following ways:

- (i) Only static members (functions or variables) of the same class can be accessed by a static member function.
- (ii) It is called by using the **name of the class** rather than an object as given below:

```
Name_of_the_class :: function_name
```

For example,

```
student::showcount();
```

## FRIEND CLASSES

In C++ , a class can be made a friend to another class. For example,

```

class TWO; // forward declaration of the class TWO

class ONE
{
    .....

    .....

public:
    .....

    .....

    friend class TWO; // class TWO declared as friend of class ONE
};

```

Now from class TWO , all the member of class ONE can be accessed.

## INTRODUCTION

A **constructor** (having the same name as that of the class) is a member function which is automatically used to initialize the objects of the class type with legal initial values. Destructors are the functions that are complimentary to constructors. These are used to de-initialize objects when they are destroyed. A destructor is called when an object of the class goes out of scope, or when the memory space used by it is de allocated with the help of **delete** operator.

Operator overloading is one of the most exciting features of C++. It is helpful in enhancement of the power of extensibility of C++ language. Operator overloading redefines the C++ language. User defined data types are made to behave like built-in data types in C++. Operators +, \*, <=, += etc. can be given additional meanings when applied on user defined data types using operator overloading. The mechanism of providing such an additional meaning to an operator is known as operator overloading in C++.

### **Declaration and Definition of a Constructor:-**

It is defined like other member functions of the class, i.e., either inside the class definition or outside the class definition.

For example, the following program illustrates the concept of a constructor :

```
//To demonstrate a constructor
#include <iostream.h>
#include <conio.h>
Class rectangle
{
    private :
        float length, breadth;
    public:
        rectangle ()//constructor definition
        {
            //displayed whenever an object is created
            cout<<"I am in the constructor"; length-
            10.0;
```



```

        breadth=20.5;
    }

    float area()
    {
        return (length*breadth);
    }
};

void main()
{
    clrscr();

    rectangle rect; //object declared

    cout<<"\nThe area of the rectangle with default parameters
    is:"<<rect.area()<<"sq.units\n";
    getch();
}

```

## **Type Of Constructor**

There are different type of constructors in C++.

### **Overloaded Constructors**

Besides performing the role of member data initialization, constructors are no different from other functions. This included overloading also. In fact, it is very common to find overloaded constructors. For example, consider the following program with overloaded constructors for the figure class :

```

//Illustration of overloaded constructors

//construct a class for storage of dimensions of circles.

//triangle and rectangle and calculate their area

#include<iostream.h>

#include<conio.h>

#include<math.h>

#include<string.h> //for strcpy()

```

```

Class figure
{
Private:
    Float radius, side1,side2,side3; //data
    members Char shape[10];
Public:
    figure(float r) //constructor for circle
    {
radius=r;
strcpy (shape, "circle");
}
    figure (float s1,float s2) //constructor for rectangle
    strcpy
    {
        Side1=s1;
        Side2=s2;
        Side3=radius=0.0; //has no significance in
rectangle strcpy(shape,"rectangle");
    }
    Figure (float s1, floats2, float s3) //constructor for triangle
    {
        side1=s1;
        side2=s2;
        side3=s3;
        radius=0.0;
        strcpy(shape,"triangle");
    }
    void area() //calculate area
    {

```

```

float ar,s;
if(radius==0.0)
{
    if (side3==0.0)
        ar=side1*side2;
    else
        ar=3.14*radius*radius;
    cout<<"\n\nArea of the "<<shape<<"is :"<<ar<<"sq.units\n";
}
};
Void main()
{
    Clrscr();
    Figure circle(10.0); //objrct initialized using constructor
    Figure rectangle(15.0,20.6); //objrct initialized using onstructor
    Figure Triangle(3.0, 4.0, 5.0); //objrct initialized using constructor
    Rectangle.area();
    Triangle.area();
    Getch();//freeze the monitror
}

```

## Copy Constructor

It is of the form classname (classname &) and used for the initialization of an object from another object of same type. For example,

```

Class fun
{
    Float x,y;
Public:
    Fun (floata,float b)//constructor
{

```

```

        x = a;

        y = b;
    }

Fun (fun &f) //copy constructor
{
    cout<<"\ncopy constructor at work\n";

    X = f.x;

    Y = f.y;

}

Void display (void)
{
{
Cout<<" "<<y<<endl;

}

};

```

Here we have two constructors, one copy constructor for copying data value of a fun object to another and other one a parameterized constructor for assignment of initial values given.

## Dynamic Initialization of Objects

In C++, the class objects can be initialized at run time (dynamically). We have the flexibility of providing initial values at execution time. The following program illustrates this concept:

```

//Illustration of dynamic initialization of objects

#include <iostream.h>

#include <conio.h>

Class employee
{

Int empl_no;

Float salary;

```

```

Public:
Employee() //default constructor
{}
Employee(int empno,float s)//constructor with
arguments {
Empl_no=empno;
Salary=s;
}
Employee (employee &emp)//copy constructor
{
Cout<<"\ncopy constructor working\n";
Empl_no=emp.empl_no;
Salary=emp.salary;
}
Void display (void)
{
Cout<<"\nEmp.No:"<<empl_no<<"salary:"<<salary<<endl;
}
};
Void main()
{
int eno;
float sal;
clrscr();
cout<<"Enter the employee number and
salary\n"; cin>>eno>>sal;
employee obj1(eno,sal);//dynamic initialization of
object cout<<"\nEnter the employee number and salary\n";
cin>eno>>sal;

```

```

        employee obj2(eno,sal); //dynamic initialization of object
        obj1.display(); //function called
        employee obj3=obj2; //copy constructor called
        obj3.display();
        getch();
    }

```

## **Constructors and Primitive Types**

In C++, like derived type, i.e. class, primitive types (fundamental types) also have their constructors. Default constructor is used when no values are given but when we given initial values, the initialization take place for newly created instance. For example,

```

float x,y; //default constructor used

int a(10), b(20); //a,b initialized with values 10 and 20

float i(2.5), j(7.8); //I,j, initialized with valurs 2.5 and 7.8

```

## **Constructor with Default Arguments**

In C++, we can define constructor s with default arguments. For example,  
The following code segment shows a constructor with default arguments:

```

Class add
{
    Private:

    Int num1, num2,num3;

    Public:

    Add(int=0,int=0); //Default argument constructor
    //to reduce the number of constructors Void
    enter (int,int);
    Void sum();
    Void display();
};

```

```
//Default constructor definition
add::add(int n1, int n2)
{
    num1=n1;
    num2=n2;
    num3=n0;
}

Void add ::sum()
{
    Num3=num1+num2;
}

Void add::display ()
{
    Cout<<"\nThe sum of two numbers is "<<num3<<endl;
}

```

Now using the above code objects of type add can be created with no initial values, one initial values or two initial values. For Example,

```
Add obj1, obj2(5), obj3(10,20);
```

Here, obj1 will have values of data members num1=0, num2=0 and num3=0

Obj2 will have values of data members num1=5, num2=0 and num3=0 Obj3

will have values of data members num1=10, num2=20 and num3=0

**If two constructors for the above class add are**

```
Add::add() {} //default constructor
```

```
and add::add(int=0); //default argument constructor
```

Then the default argument constructor can be invoked with either two or one or no parameter(s).

Without argument, it is treated as a default constructor-using these two forms together causes ambiguity. For example,

The declaration `add obj;`

is ambiguous i.e., which one constructor to invoke i.e.,

```
add :: add()
```

```
or add :: add(int=0,int=0)
```

so be careful in such cases and avoid such mistakes.



## Declaration and Definition of a Destructor

The syntax for declaring a destructor is :

```
-name_of_the_class()  
  
{  
  
}
```

So the name of the class and destructor is same but it is prefixed with a ~

(tilde). It does not take any parameter nor does it return any value. Overloading a destructor is not possible and can be explicitly invoked. In other words, a class can have only one destructor. A destructor can be defined outside the class. The following program illustrates this concept :

```
//Illustration of the working of Destructor function  
  
#include<iostream.h>  
  
#include<conio.h>  
  
  
class add  
{  
  
    private :  
  
        int num1,num2,num3;  
  
    public :  
  
        add(int=0, int=0); //default argument constructor  
                                //to reduce the number of constructors  
  
        void sum();  
  
        void display();  
  
        ~ add(void); //Destructor  
  
};  
  
//Destructor definition ~add()  
  
Add:: ~add(void) //destructor called automatically at end of program
```

```

{
    Num1=num2=num3=0;
    Cout<<"\nAfter the final execution, me, the object has entered in
the"
    <<"\ndestructor to destroy
    myself\n"; }
//Constructor definition add()
Add::add(int n1,int n2)
{
    num1=n1;
    num2=n2;
    num3=0;
}
//function definition sum ()
Void add::sum()
{
    num3=num1+num2;
}
//function definition display ()
Void add::display ()
{
    Cout<<"\nThe sum of two numbers is "<<num3<<endl;
}
void main()
{
    Add obj1,obj2(5),obj3(10,20): //objects created and initialized
clrscr();
    Obj1.sum(); //function call
    Obj2.sum();
    Obj3.sum();
}

```

```
cout<<"\nUsing obj1 \n";  
obj1.display(); //function call  
cout<<"\nUsing obj2 \n";  
obj2.display();  
cout<<"\nUsing obj3 \n";  
obj3.display();  
}
```

## DECLARATION AND DEFINITION OF A OVERLOADING

For defining an additional task to an operator, we must mention what it means in relation to the class to which it (the operator) is applied. The **operator function** helps us in doing so.

The Syntax of declaration of an Operator function is as follows:

Operator Operator\_name

For example, suppose that we want to declare an Operator function for '='. We can do it as follows:

operator =

A Binary Operator can be defined either a member function taking one argument or a global function taking one arguments. For a Binary Operator X, a X b can be interpreted as either

an operator X (b) or operator X (a, b).

For a Prefix unary operator Y, Ya can be interpreted as either a.operator Y ( ) or Operator Y (a).  
For a Postfix unary operator Z, aZ can be interpreted as either a.operator Z(int) or Operator (Z(a),int).

The operator functions namely operator=, operator [ ], operator ( ) and operator? must be non-static member functions. Due to this, their first operands will be lvalues.

An operator function should be either a member or take at least one class object argument. The operators new and delete need not follow the rule. Also, an operator function, which needs to accept a basic type as its first argument, cannot be a member function. Some examples of declarations of operator functions are given below:

```
class P
{
    P operator ++ (int); //Postfix increment
    P operator ++ ( ); //Prefix increment
    P operator || (P); //Binary OR
}
```

Some examples of Global Operator Functions are given below:

```
P operator – (P); // Prefix Unary minus
P operator – (P, P); // Binary “minus”
P operator - - (P &, int); // Postfix Decrement
```

We can declare these Global Operator Functions as being friends of any other class.

Examples of operator overloading:

### **Operator overloading using friend.**

```
Class time
{
    int r;
    int i;
    public:
        friend time operator + (const time &x, const time &y );
        // operator overloading using
        friend time ( ) { r = i = 0;}
        time (int x, int y) {r = x; i = y;}
};
time operator + (const time &x, const time &y)
{
    time z;
```

```

        z.r = x.r + y.r;
        z.i = x.i + y.i;

    return z;
}

main ( )
{
    time x,y,z;
    x = time (5,6);
    y = time (7,8);
    z = time (9, 10);
    z = x+y; // addition using friend function +
}

```

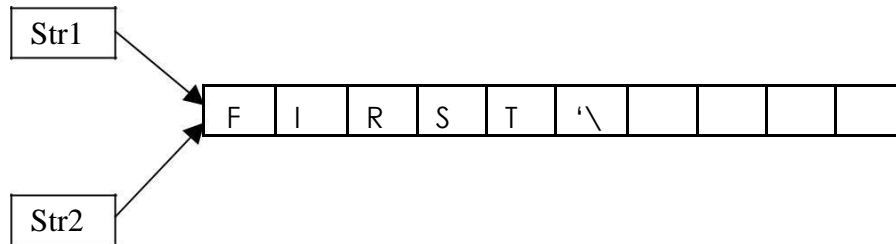
### **Operator overloading using member function:**

```

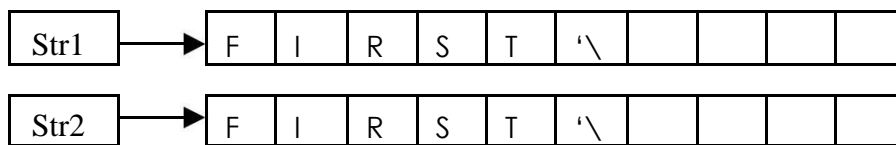
Class abc
{
    char * str;
    int len ; // Present length of the string
    int max_length; // (maximum space allocated to string)
public:
    abc ( ); // blank string of length 0 of maximum allowed length of size 10.
    abc (const abc &s ) ;// copy constructor
    ~ abc ( ) {delete str;}
    int operator == (const abc &s ) const; // check for
    equality abc & operator = (const abc &s ); // overloaded
    assignment operator
    friend abc operator + (const abc &s1, const abc &s2);
    } // string concatenation
abc:: abc ( )
{
    max_length = 10;
    str = new char [
    max_length]; len = 0;
    str [0] = '\0';
}
abc :: abc (const abc &s )
{
    len = s. len;
    max_length = s.max_length;
    str = new char [max_length];
    strcpy (str, s.str); // physical copying in the new location.
}

```

[ **Not:** Please note the need of explicit copy constructor as we are using pointers. For example, if a string object containing string “first” is to be used to initialise a new string and if we do not use copy constructor then will cause:



That is two pointers pointing to one instance of allocated memory, this will create problem if we just want to modify the current value of one of the string only. Even destruction of one string will create problem. That is why we need to create separate space for the pointed string as:



Thus, we have explicitly written the copy constructor. We have also written the explicit destructor for the class. This will not be a problem if we do not use pointers.

```
abc :: ~ abc ( )
{
    delete str;
}
abc & abc :: operator = (const abc &s )
{
    if (this != &s) // if the left and right hand variables are different
    {
        len = s.len;
        max_length = s.max-length;
        delete str; // get rid of old memory space allocated to this string
        str = new char [max_length]; // create new locations
        strcpy (str, s.str); // copy the content using string copy function
    }
    return *this;
}
```

// Please note the use of this operator which is a pointer to object that invokes the call

to this assignment operator function.

```
inline int abc :: operator == (const abc &s ) const
{
    // uses string comparison
    function return strcmp (str,s.str);
}
abc  abc:: operator + (const abc &s )
abc s3;
s3.len = len + s.len;
s3.max_length = s3.len;
char * newstr = new char [length + 1];
strcpy (newstr, s.str);
strcat (newstr,str);
s3.str = newstr;
return (s3);
}
```

Overloading << operator:

To overload << operator, the following function may be used:

```
Ostream & operator << (ostream &s, const abc &x )
{
    s<< "The String is:" <<x; }
    return s;
}
```

You can write appropriate main function and use the above overloaded operators as shown in the complex number example.