Copy #1

Unit 1

Copy #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

Copy #1

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.

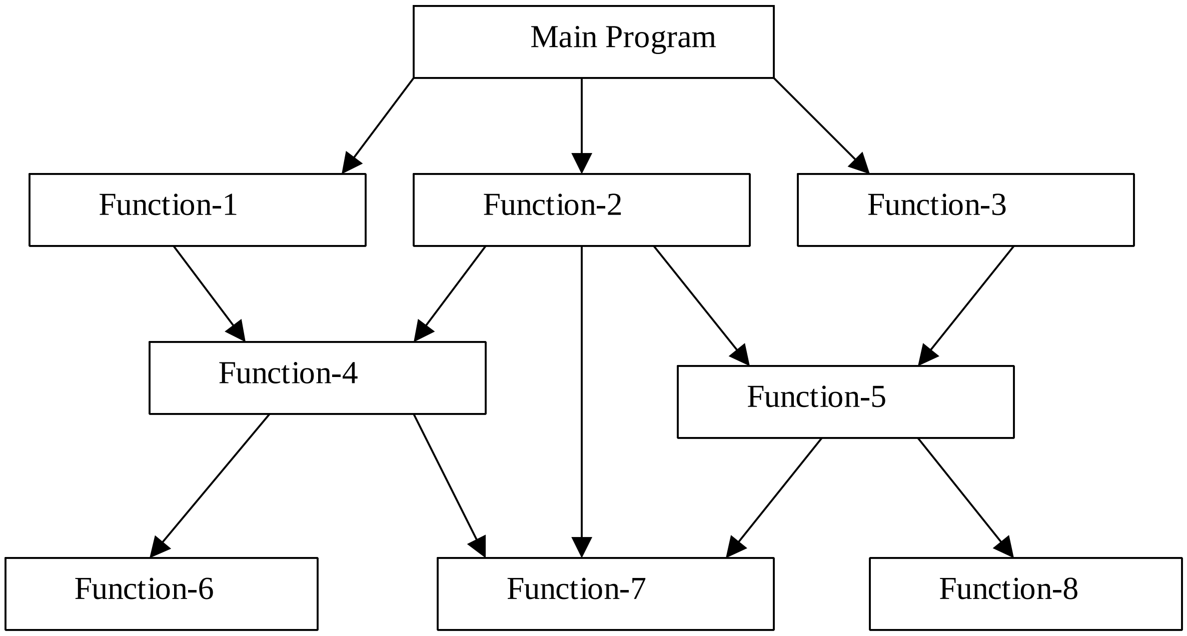


Fig. 1.2 Typical structure of procedural oriented programs

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.

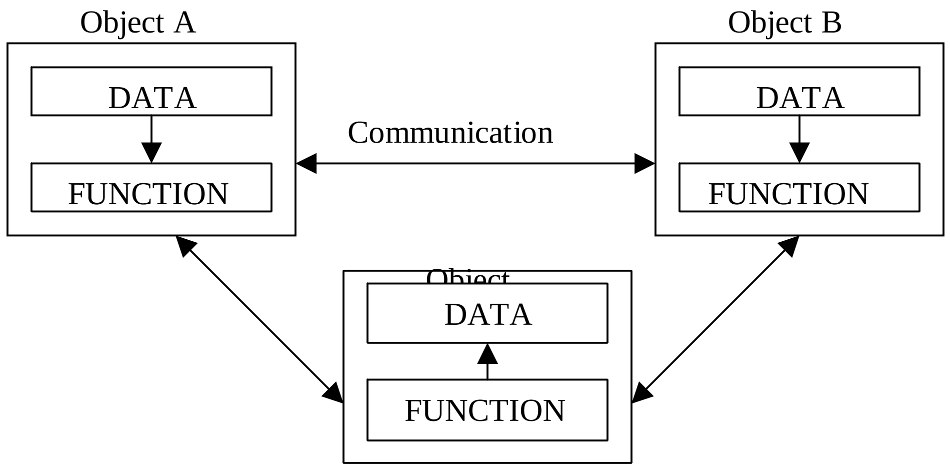
Some Characteristics exhibited by procedure-oriented programming are:

- 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

Copy #1

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 objectoriented analysis and design.

```
OBJECTS: STUDENT

DATA
        Name
        Date-of-birth
        Marks

FUNCTIONS
        Total
        Average
        Display
        .........
```

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

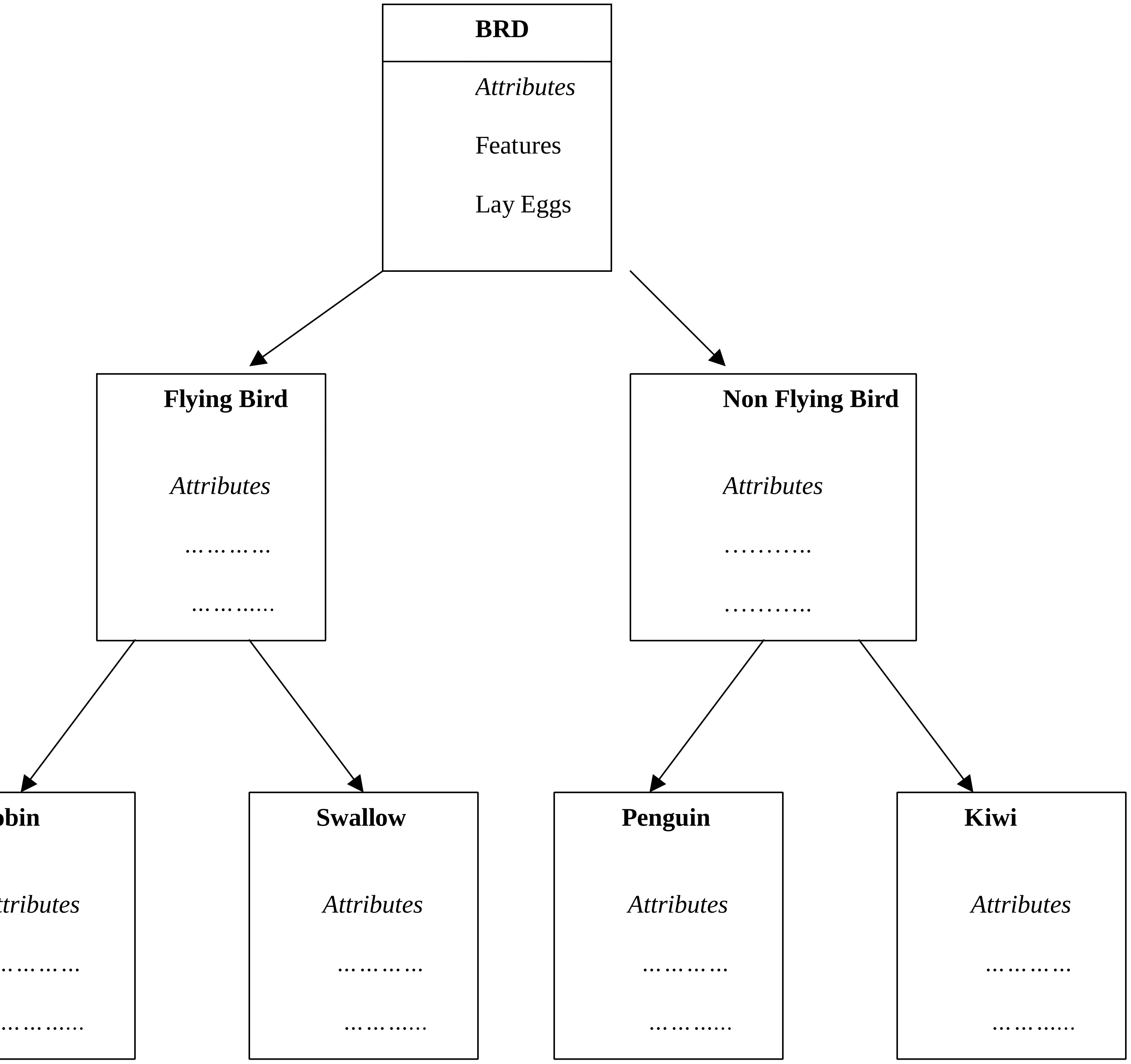
The wrapping up of data and function into a single unit (called class) is known as *encapsulation*. Data and encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide 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

Copy #1



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.*

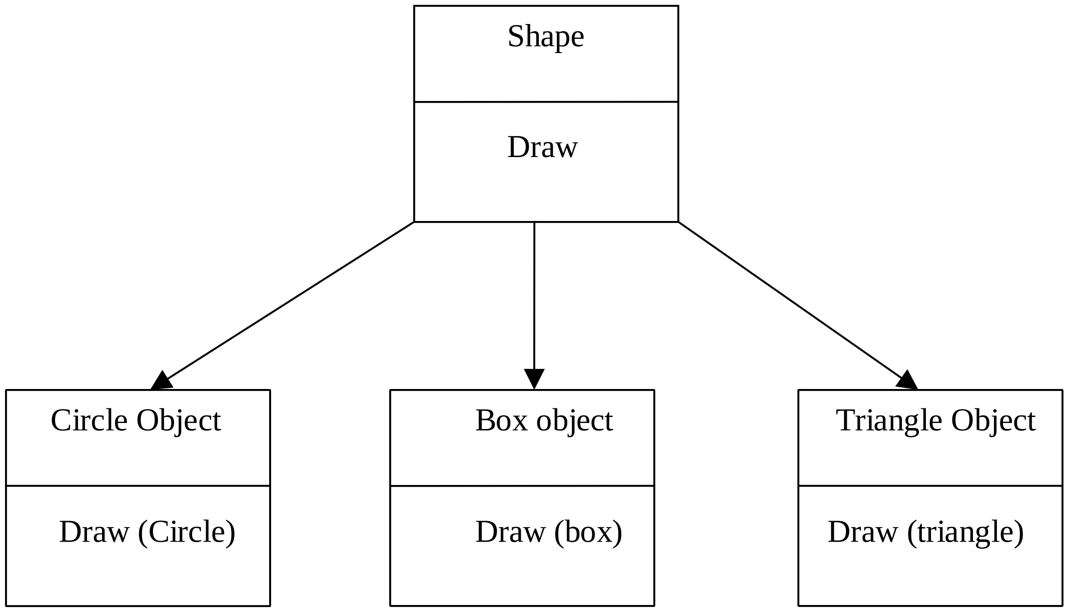


Fig. 1.7 Polymorphism

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

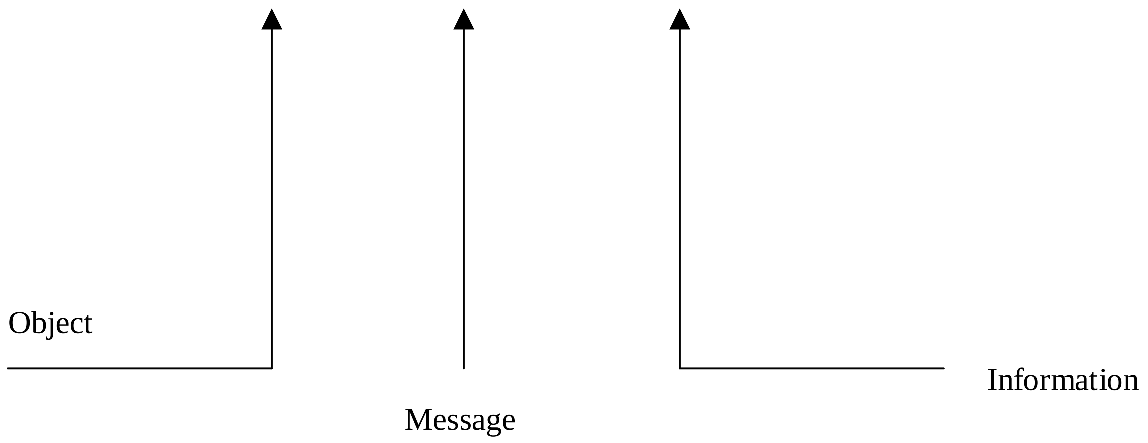
An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

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 realworld 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:

Employee. Salary (name);



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. ObjectOrientation 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.

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 feature that are required for object based programming are:

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

Languages that support programming with objects are said to the objects-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

Object-oriented programming language incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statements:

Object-based features + inheritance + dynamic binding

1.8 Application of OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as window. Hundreds of windowing systems have been developed, using the OOP techniques.

Real-business system are often much more complex and contain many more objects with complicated attributes and method. OOP is useful in these types of application because it can simplify a complex problem. The promising areas of application of OOP include:

- Real-time system
- Simulation and modeling

- Object-oriented data bases
- Hypertext, Hypermedia, and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

The object-oriented paradigm sprang from the language, has matured into design, and has recently moved into analysis. It is believed that the richness of OOP environment will enable the software industry to improve not only the quality of software system but also its productivity. Object-oriented technology is certainly going to change the way the software engineers think, analyze, design and implement future system.

1.9 Introduction of C++

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented version 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++ complier. We shall see these differences later as and when they are encountered.

The most important facilities that C++ adds on to C care 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.

1.9.1 Application of C++

C++ is a versatile language for handling very large programs; it is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real life applications systems.

- 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.

Copy #1

1.10 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;
}
```

Program 1.10.1

This simple program demonstrates several C++ features.

1.10.1 Program feature

Like C, the C++ program is a collection of function. The above example contain only one function **main().** As usual execution begins at main(). Every C++ program must have a **main()**. C++ is a free form language. With a few exception, the compiler ignore carriage return and white spaces. Like C, the C++ statements terminate with semicolons.

1.10.2 Comments

C++ introduces a new comment symbol // (double slash). Comment start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line, and whatever follows till the end of the line is ignored. Note that there is no closing symbol.

The double slash comment is basically a single line comment. Multiline comments can be written as follows:

```
    // This is an example of
    // C++ program to illustrate
    // some of its features
```

The C comment symbols /*,*/ are still valid and are more suitable for multiline comments. The following comment is allowed:

```
    /* This is an example of        C++
program to illustrate
      some of its features
    */
```

1.10.3 Output operator

The only statement in program 1.10.1 is an output statement. The statement

```
Cout<<"C++ is better than C.";
```

Causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, cout and <<. The identifier cout(pronounced as C out) is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices. The operator << is called the insertion or put to operator.

1.10.4 The iostream File

We have used the following #include directive in the program:

```
#include <iostream>
```

The #include directive instructs the compiler to include the contents of the file enclosed within angular brackets into the source file. The header file **iostream.h** should be included at the beginning of all programs that use input/output statements.

1.10.5 Namespace

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifier defined in the **namespace** scope we must include the using directive, like

```
Using namespace std;
```

Here, std is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope. **Using** and **namespace** are the new keyword of C++.

1.10.6 Return Type of main()

In C++, main () returns an integer value to the operating system. Therefore, every main () in C++ should end with a return (0) statement; otherwise a warning an error might occur. Since main () returns an integer type for main () is explicitly specified as **int.** Note that the default return type for all function in C++ is **int.** The following main without type and return will run with a warning:

```
main () {
 …………..
 ………….
}
```

1.11 More C++ Statements

Copy #1

Let us consider a slightly more complex C++ program. Assume that we should like to read two numbers from the keyboard and display their average on the screen. C++ statements to accomplish this is shown in program 1.11.1

AVERAGE OF TWO NUMBERS

```
#include<iostream.h> // include header file

Using namespace std;

Int main()

{

        Float number1, number2,sum, average;
        Cin >> number1;        // Read Numbers
        Cin >> number2;        // from keyboard
        Sum = number1 + number2;
        Average = sum/2;
        Cout << ”Sum = “ << sum << “\n”;
        Cout << “Average = “ << average << “\n”;

        Return 0;

}       //end of example
```

The output would be:

```
Enter two numbers: 6.5  7.5
Sum = 14
Average = 7
```

Program 1.11.1

1.11.1 Variables

The program uses four variables number1, number2, sum and average. They are declared as type float by the statement.

```
float number1, number2, sum, average;
```

All variable must be declared before they are used in the program.

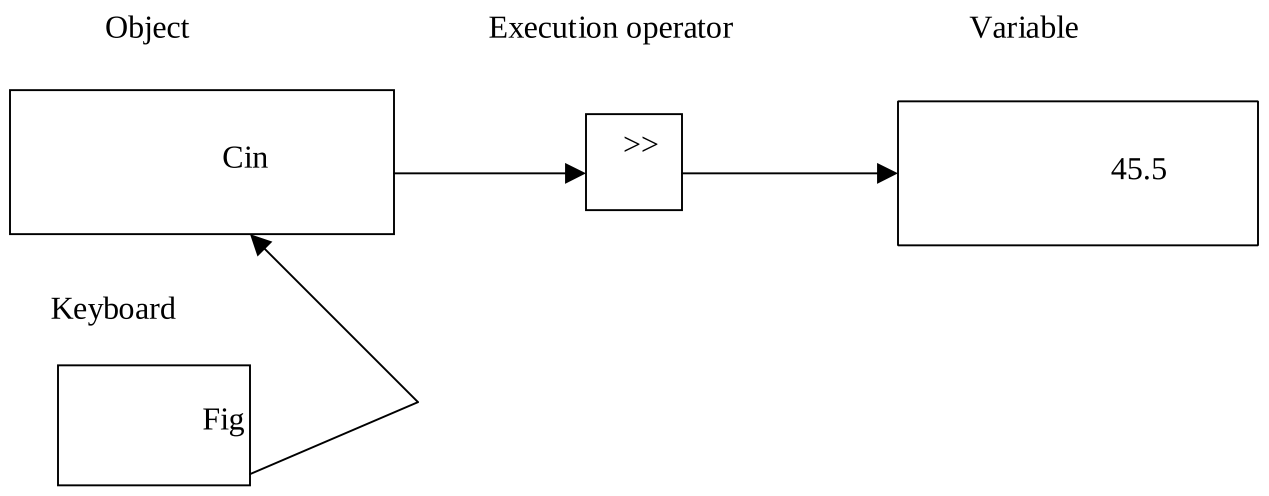
1.11.2 Input Operator

The statement

```
cin >> number1;
```

Is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier cin (pronounced ‘C in’) is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator >> is known as extraction or get from operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right fig 1.8. This corresponds to a familiar scanf() operation. Like <<, the operator >> can also be overloaded.



1.8 Input using extraction operator

1.11.3 Cascading of I/O Operators

We have used the insertion operator << repeatedly in the last two statements for printing results.

The statement

```
Cout << “Sum = “ << sum << “\n”;
```

First sends the string “Sum = “ to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << in one statement is called cascading. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

```
Cout << “Sum = “ << sum << “\n”
     << “Average = “ << average << “\n”;
```

This is one statement but provides two line of output. If you want only one line of output, the statement will be:

```
Cout << “Sum = “ << sum << “,”
     << “Average = “ << average << “\n”;
```

The output will be:

```
Sum  = 14, average = 7
```

We can also cascade input iperator >> as shown below:

Cin >> number1 >> number2;

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to munber1 and 20 to number2.

1.12 An Example with Class

- One of the major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Like structures in C, classes are user-defined data types.

Program 1.12.1 shows the use of class in a C++ program.

USE OF CLASS

```
        #include<iostream.h> // include header file

        using namespace std;
        class person
        {

                char name[30];
                Int age;

                public:
         void getdata(void);      void
display(void);
        };
        void person :: getdata(void)
        {
                cout << "Enter name: ";
        cin >> name;
                cout << "Enter age: ";
                cin >> age;
        }
        Void person : : display(void)
        {
                cout << "\nNameame: " << name;
                cout << "\nAge: " << age;
        }

        Int main()
        {               person p;
                p.getdata();
                p.display();
                Return 0;

        }       //end of example
```

PROGRAM 1.12.1

The output of program is:

```
Enter Name: Ravinder
Enter age:30
Name:Ravinder
Age: 30
```

The program define **person** as a new data of type class. The class person includes two basic data type items and two function to operate on that data. These functions are called **member function**. The main program uses **person** to declare variables of its type. As pointed out earlier, class variables are known as objects. Here, p is an object of type **person**. Class object are used to invoke the function defined in that class.

1.13 Structure of C++ Program

As it can be seen from program 1.12.1, a typical C++ program would contain four sections as shown in fig. 1.9. This section may be placed in separate code files and then compiled independently or jointly.
It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification of the interface from the implementation details (member function definition).
Finally, the main program that uses the class is places in a third file which “includes: the previous two files as well as any other file required.

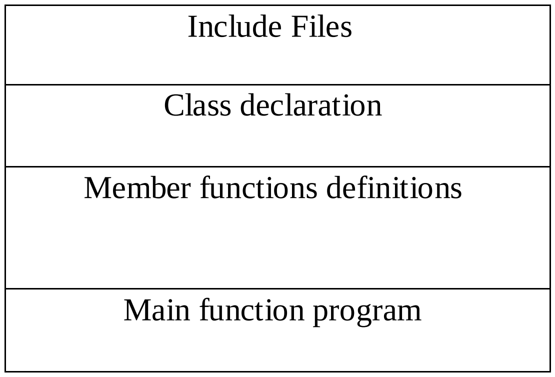
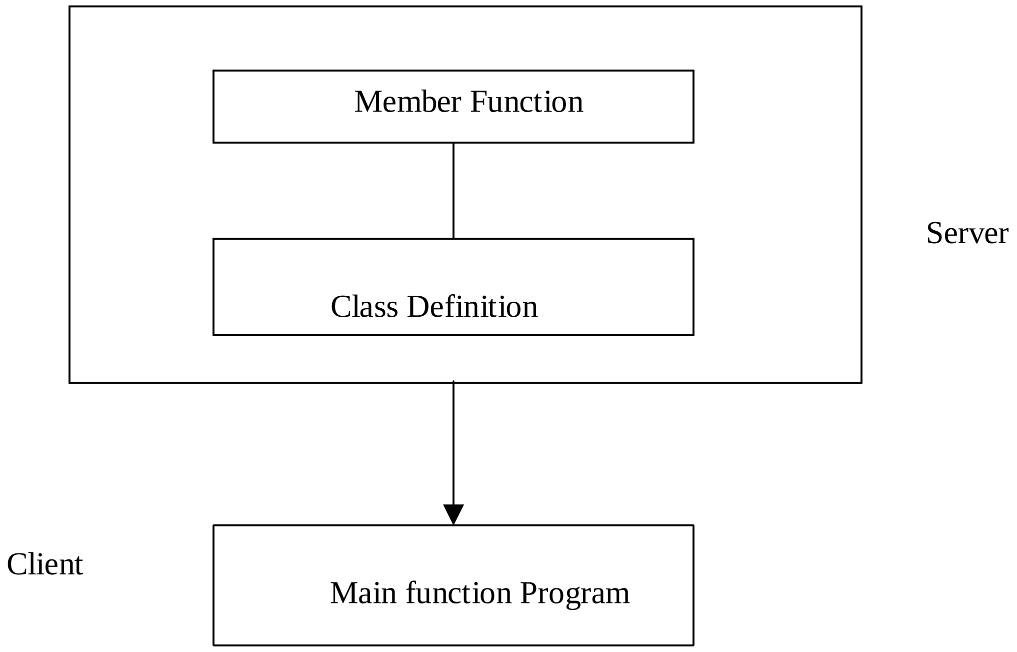


Fig 1.9 Structure of a C++ program

Copy #1

This approach is based on the concept of client-server model as shown in fig. 1.10. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

Fig. 1.10 *The client-server model*



1.14 Creating the Source File

Like C programs can be created using any text editor. Foe example, on the UNIX, we can use vi or ed text editor for creating using any text editor for creating and editing the source code. On the DOS system, we can use endlin or any other editor available or a word processor system under non-document mode.

Some systems such as Turboc C++ provide an integrated environment for developing and editing programs

The file name should have a proper file extension to indicate that it is a C++ implementations use extensions such as .c,.C, .cc, .cpp and .cxx. Turboc C++ and Borland C++ use .c for C programs and .cpp(C plus plus) for C++ programs. Zortech C++ system use .cxx while UNIX AT&T version uses .C (capital C) and .cc. The operating system manuals should be consulted to determine the proper file name extension to be used.

1.15 Compiling and Linking

The process of compiling and linking again depends upon the operating system. A few popular systems are discussed in this section.

Unix AT&T C++

This process of implementation of a C++ program under UNIX is similar to that of a C program. We should use the “cc” (uppercase) command to compile the program. Remember, we use lowercase “cc” for compiling C programs. The command

```
CC example.C
```

At the UNIX prompt would compile the C++ program source code contained in the file **example.C**. The compiler would produce an object file **example.o** and then automatically link with the library functions to produce an executable file. The default executable filename is **a. out**.

A program spread over multiple files can be compiled as follows:

CC file1.C file2.o

The statement compiles only the file **file1.C** and links it with the previously compiled **file2.o** file. This is useful when only one of the files needs to be modified. The files that are not modified need not be compiled again.

Turbo C++ and Borland C++

Turbo C++ and Borland C++ provide an integrated program development environment under MS DOS. They provide a built-in editor and a menu bar includes options such as File, Edit, Compile and Run.
We can create and save the source files under the **File option,** and edit them under the **Edit option.** We can then compile the program under the **compile** option and execute it under the **Run option.** The **Run option** can be used without compiling the source code.

Questions

1. What are the major issues facing the software industry today?
2. What is POP? Discuss its features.
3. Describe how data are shared by functions in procedure-oriented programs?
4. What is OOP? What are the difference between POP and OOP?
5. How are data and functions organized in an object-oriented program?
6. What are the unique advantages of an object-oriented programming paradigm?
7. Distinguish between the following terms:
   (a) Object and classes
   (b) Data abstraction and data encapsulation
   (c) Inheritance and polymorphism (d) Dynamic binding and message passing
8. Describe inheritance as applied to OOP.
9. What do you mean by dynamic binding? How it is useful in OOP?
10. What is the use of preprocessor directive #include<iostream>?
11. How does a main () function in c++ differ from main () in c?
12. Describe the major parts of a c++ program.
13. Write a program to read two numbers from the keyboard and display the larger value on the screen.
14. Write a program to input an integer value from keyboard and display on screen “WELL DONE” that many times.

References:

1. Object –Oriented –Programming in C++ by E Balagurusamy.

Copy #1

2. Object –Oriented –Programming with ANSI & Turbo C++ by Ashok N. Kamthane.
3. OO Programming in C++ by Robert Lafore, Galgotia Publications Pvt. Ltd. 4. Mastering C++ By K R Venugopal, Rajkumar Buyya, T Ravishankar.
5. Object Oriented Programming and C++ By R. Rajaram.
6. Object –Oriented –Programming in C++ by Robert Lafore.

Copy #1

4.1 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

Monolethic 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.

4.2 FUNCTION DEFINITION AND DECLARATION

In C++, a function must be defined prior to it's 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;
          }
```

4.3 ARGUMENTS TO A FUNCTION

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

Copy #1

4.3.1 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;
                                                       formal parameters
                                                       Semicolon here
    float SUMFUN(float x, int N); //function declaration
return type
     ..............................
    ..............................
          result = SUMFUN(X,N); //function declaration

         }
         //function SUMFUN() definition
                                                   No semicolon here
         float SUMFUN(float x,int N) //function declaration
         {
         ..............................
           ..............................     Body of the function
           ..............................

      }
                           No semicolon here
```

4.3.2 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.

Copy #1

```
// 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);
    }
```

4.3.3 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.

Copy #1

4.4 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 yeaers) :";
  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;
}
```

Copy #1

4.5 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<<"\Enter 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.

4.6 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)
```

Copy #1

```
    {
  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.

4.7 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
```

Copy #1

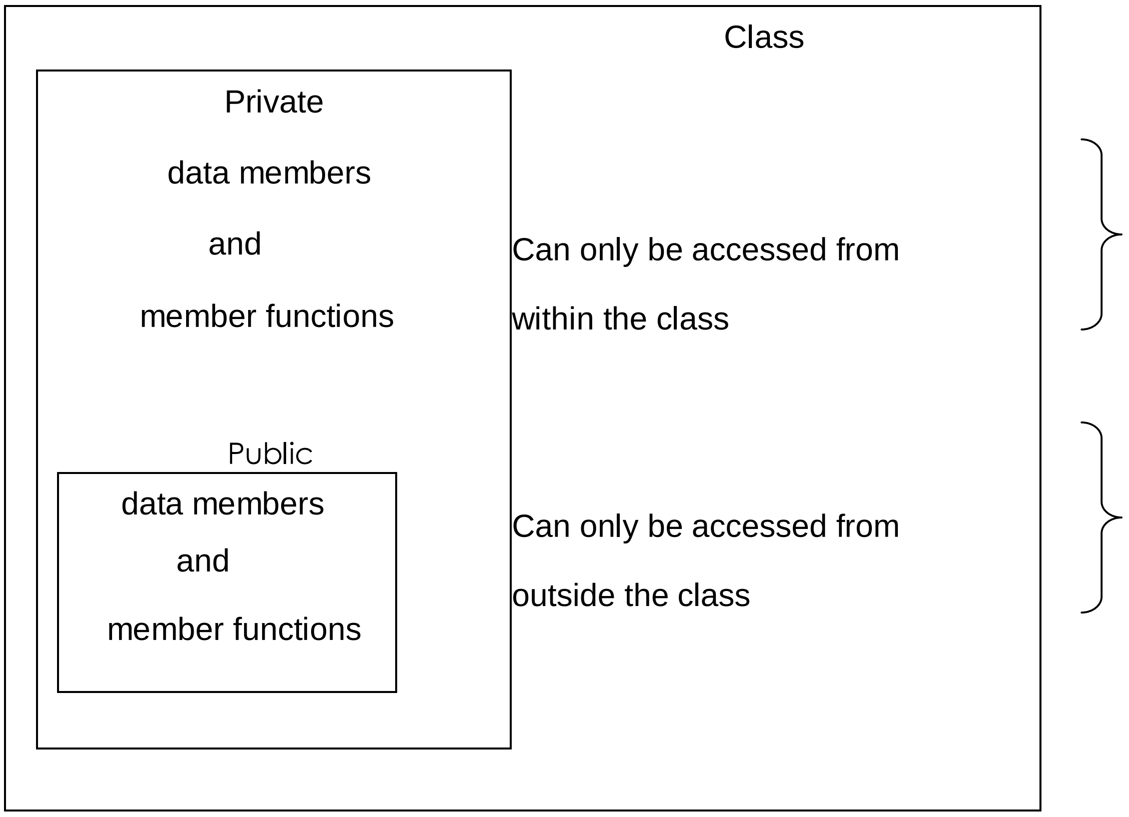
```
          Function declaration; // Member Function (Method)
     protected: Variable declaration;          Function
     declaration; public  : variable declaration;
     Function declaration;
     };
```

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.

4.8 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 :

4.8.1 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.

4.8.2 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 ised om situations where a global variable exists with the same name as a local variable and it identifies the global variable.

4.9 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

Common for all objects

Member Member Member function3 f ti 1 f ti 2

Memory allocated when

member functions are defined

Object 1

Object 2

data member

data member 1

data member   data member 2

Memory allocated when

objects declared

A class, its member functions and objects in memory.

4.10 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.

4.11 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 : sudent 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 fifers 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,

Copy #1

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

4.12 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.

4.15 Review Questions

Q. 1. what is a function ? How will you define a function in C++ ?

Q. 2. How are the argument data types specified for a C++ function? Explain with Suitable example.

Q. 3. What types of functions are available in C++ ? Explain.

Q. 4. What is recursion? While writing any recursive function what thing(s) must be taken care of ?

Q. 5. What is inline function? When will you make a function inline and why ?

Q.6. What is a class? How objects of a class are created ?

Q. 7. What is the significance of scope resolution operator (::) ?

Q. 8. Define data members , member function, private and public members with example.

Q.10. Define a string data type with the following functionality:

- A constructor having no parameters,
- Constructors which initialize strings as follows:
  - A constructor that creates a string of specific size

  - Constructor that initializes using a pointer string
  - A copy constructor
- Define the destructor for the class
- It has overloaded operators. (This part of question will be taken up in the later units).
- There is operation for finding length of the string.

4.16 Further Readings

1. Rambagh J. , " Object Oriented Modeling and Design" , Prentice Hall of India , New Delhi.
2. E. Balagrusamy, "Object Oriented Programming with C++", Tata McGraw Hill.

Copy #1

5.1 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 deinitialize 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++.

5.2 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 <iostram.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();
    }
```

5.3 Type Of Constructor

There are different type of constructors in C++.

5.3.1 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
```

Copy #1

```
                 {
          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
        }
```

5.3.2 Copy Constructor

It is of the form classname (classname &) and used for the initialization of an object form 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<<end1;
        }
};
```

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.

Copy #1

5.3.3 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<<end1;

}

};

Void main()

{
```

Copy #1

```
    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();
     }
```

5.3.4 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
```

5.3.5 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
```

Copy #1

```
        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<<end1;
     }
```

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.

5.4 SPECIAL CHARACTERISTICS OF CONSTRUCTORS

These have some special characteristics. These are given below:

(i) These are called automatically when the objects are created.

(ii) All objects of the class having a constructor are initialized before some use.

(iii) These should be declared in the public section for availability to all the functions.

(iv) Return type (not even **void**) cannot be specified for constructors.

(v) These cannot be inherited, but a **derived** class can call the base class constructor.

(vi) These cannot be static.

(vii) Default and copy constructors are generated by the compiler wherever required. Generated constructors are public.

(viii) These can have default arguments as other C++ functions.

(ix) A constructor can call member functions of its class.

(x) An object of a class with a constructor cannot be used as a member of a **union.**

(xi) A constructor can call member functions of its class.

(xii) We can use a constructor to create new objects of its class type by using the syntax.

```
                Name_of_the_class (expresson_list)
For example,
                Employee obj3 = obj2; // see program 10.5
            Or even
        Employee obj3 = employee (1002, 35000); //explicit call
```

(xiii) The make **implicit** calls to the memory allocation and deallocation operators **new** and **delete.**

(xiv) These cannot be **virtual.**

Copy #1

5. 5 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";
}
```

Copy #1

```
        //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<<end1;
        } 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();
        }
```

5.6 Special Characteristics of Destructors

Some of the characteristics associated with destructors are :

(i) These are called automatically when the objects are destroyed.

(ii) Destructor functions follow the usual access rules as other member functions.

(iii) These **de-initialize** each object before the object goes out of scope.

(iv) No argument and return type (even void) permitted with destructors.

(v) These cannot be inherited.

(vi) **Static** destructors are not allowed.

(vii) Address of a destructor cannot be taken.

(viii) A destructor can call member functions of its class.

(ix) An object of a class having a destructor cannot be a member of a union.

5.7 DECLARATION AND DEFINITION OF A OVERLOADING

For defining an additional task to an operator, we must mention what is 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 nonstatic 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
{
```

Copy #1

```
 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.

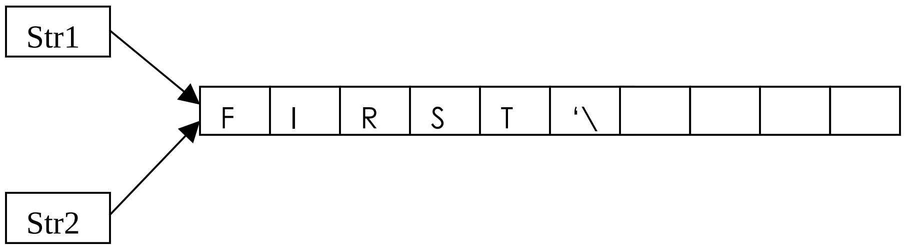
```
Class time
{   int r;
int i;
public:
        friend time operator + (const time &x, const time &y );
                        // operator overloading using friend
  time ( ) { r = i = 0;}
  time (int x, int y) {r = x; i = y;}
};
 time operator + (const time &x, const time &y)
 {
time z;
   z.r = x.r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

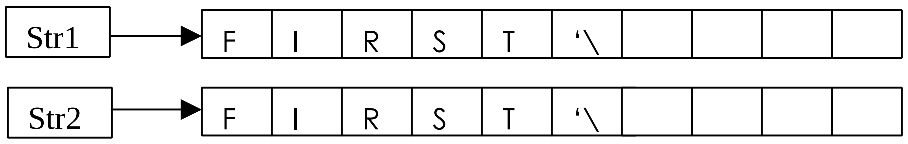
```
 Class abc
{   char *
str;
 int len ; // Present length of the string     int max_length; // (maximum
space allocated to string)   public:     abc ( );   // black string of length 0 of
maximum allowed length of size 10.    abc (const abc &s ) ;// copy
constructor
```

```
  ~ 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.

5.8 ASSIGNMENT AND INITIALISATION

Consider the following class:

Copy #1

```
class student
{  char
name;   int
rollno;
public:
student ( ) {name = new char [20];}
~ student ( ) {delete [ ] name;}
};
int f ( )
{ student S1,
S2;    cin >> S1;
cin >> S2;
  S1 = S2;
}
```

Now, the problem is that after the execution of f ( ), destructors for S1& S2 will be executed. Since both S1 & S2 point to the same storage, execution of destructor twice will lead to error as the storage being pointed by S1 & S2 were disposed off during the execution of destructor for S1 itself.

Defining assignment of strings as follows can solve this problem,

```
class student
{
Public:
  char name;
  int rollno;
  student ( ) {name = new char [20];}
~ student ( ) {delete [ ] name ;}
student & operator = (const student & )
}
student & student :: Operator = (const student &e)
{
  if (this ! =&e)
 delete       []       name;
name = new char [20];
 strcpy(name, name);
  }
 return *this;
}
```

5.9 TYPE CONVERSIONS

We have overloaded several kinds of operators but we haven't considered the assignment operator (=). It is a very special operator having complex properties. We know that = operator assigns values form one variable to another or assigns the value of user defined object to another of the same type. For example, `int  x, y ;`

```
x = 100;

y = x;
```

Here, first 100 is assigned to x and then x to y.

Consider another statement, 13 = t1 + t2;

This statement used in program 11.2 earlier, assigns the result of addition, which is of type time to object t3 also of type time.

So the assignments between basic types or user defined types are taken care by the compiler provided the data type on both sides of = are of same type.

But what to do in case the variables are of different types on both sides of the = operator? In this case we need to tell to the compiler for the solution.

Three types of situations might arise for data conversion between different types :

(i) Conversion form basic type to class type. (ii)
Conversion from class type to basic type.
(iii) Conversion from one class type to another class type.

Now let us discuss the above three cases :

(i) Basic Type to Class Type

This type of conversion is very easy. For example, the following code segment converts an int type to a class type.

```
class distance
        {
              int feet;
              int inches;
              public:
                .....
                .....
        distance (int dist) //constructor
           {
              feet = dist/12;
      inches = dist%12;
        }
};
```

The following conversion statements can be coded in a function :

```
distance dist1; //object dist1 created
int length = 20;
```

```
dist1=length; //int to class type
```

After the execution of above statements, the **feet** member of **dist1** will have a value of 1 and **inches** member a value of 8, meaning 1 feet and 8 inches.

A class object has been used as the left hand operand of = operator, so the type conversion can also be done by using an overloaded = operator in C++.

(ii) Class Type to Basic Type

For conversion from a basic type to class type, the constructors can be used. But for conversion from a class type to basic type constructors do not help at all. In C++, we have to define an overloaded **casting operator** that helps in converting a class type to a basic type. The syntax of the **conversion function** is given below:

```
Operator typename()
        {
                .......
                ....... //statements
        }
```

Here, the function converts a class type data to typename. For example, the **operator float ( )** converts a class type to type **float, the operator int ( )** converts a class type object to type int. For example,

```
matrix :: operator float ()
        {
              float sum = 0.0;
      for(int i=0;i<m;i++)
        {
              for (int j=0; j<n; j++)
sum=sum+a[i][j]*a[i][j];
                }
        Return sqrt(sum); //norm of the matrix
    }
```

Here, the function finds the norm of the matrix (Norm is the square root of the sum of the squares of the matrix elements). We can use the operator float ( ) as given below : `float norm = float (arr);`

```
                or
        float norm = arr;
```

where **arr** is an object of type matrix. When a class type to a basic type conversion is required, the compiler will call the casting operator function for performing this task.

The following conditions should be satisfied by the casting operator function :

(a) It must not have any argument
(b) It must be a class member
(c) It must not specify a return type.

(i) One Class Type to Another Class Type

There may be some situations when we want to convert one class type data to another class type data. For example,

```
Obj2 = obj1; //different type of objects
```

Suppose **obj1** is an object of class **studdata** and **obj2** is that of class **result.** We are converting the class **studdata** data type to class **result** type data and the value is assigned to obj2. Here **studdata** is known as **source class** and **result** is known as the **destination class.**

The above conversion can be performed in two ways :

(a) Using a constructor.
(b) Using a conversion function.

When we need to convert a class, a casting operator function can be used i.e. source class. The source class performs the conversion and result is given to the object of destination class.

If we take a single-argument constructor function for converting the argument's type to the class type (whose member it is). So the argument is of the source class and being passed to the destination class for the purpose of conversion. Therefore it is compulsory that the conversion constructor be kept in the destination class.

.

5.12 Review Questions

Q. 1. What is the use of a constructor function in a class? Give a suitable example of a constructor function in a class.

Q. 2. Design a class having the constructor and destructor functions that shiukd display the number of object being created or destroyed of this class type.

Copy #1

Q. 3. Write a C++ program, to find the factorial of a number using a constructor and a destructor (generating the message “you have done it”)

Q. 4. Define a class “string” with members to initialize and determine the length of the string. Overload the operators ‘+’ and ‘+=’ for the class “string”.

10CS36: Object Oriented Programming in C++

Unit 2

Classes and Objects I

AGENDA

- Class Specification
- Class Objects
- Access members
- Defining member functions
- Scope resolution operator ::
- Static data members and Functions
- Data hiding
- Constructors, Destructors
- Parameterized constructors

Objectives

- To know about class specification and objects.
- To know how to access members and member functions.
- Different way of defining functions
- To know about constructors and destructors
- To understand static members and member functions.

Class

Concepts of C++

Class

- A class is a user defined data type which can store many values of **different data types**.
- A class combines data and the associated functions in a single unit.

Class Definition

Syntax :

```
class       class_name
{
private :
            data members
            member functions
protected :
            data members
            member functions
public :
            data members
            member functions
};
```

- **class** is a keyword.
- **class_name** is any valid identifier name.

Access specifiers :

private, protected, public are the keywords known as access specifiers.

Data members :

Variables declared inside the class .

Member functions (Methods) :

Functions declared inside the class.

Class Definition

Data Members :

- Variables declared inside the class are called as data members.
- Data members may be of different data types.
- Data members are used by the associated functions – member functions.

Member Functions :

- Also known as **methods.**
- Member functions are associated functions used to process data members.

Access Specifiers

private	protected	public
Can be used by only member functions of the same class.	Can be used by only member functions of the same class.	Can be used by member functions and non member functions.
Other class member functions and non member functions cannot access them.	Other class member functions and non member functions cannot access them.	Other class member functions and non member functions can access them.
Class variables (objects) also cannot access them.	Class variables (objects) also cannot access them.	Class variables (objects) can access them.
Data is more secured.	Data is more secured.	Data may not be secured.
Private members cannot be inherited.	Protected members can be inherited.	Public members can be inherited.

Access Specifiers - Example

Identify valid and invalid statements in main()

```
class student
{
        int rollno;
        char name[21];
        float marks;
public:
        void in();
        void out();
};
```

```
int main()
{
        student s;
        s.in();          // valid
        s.rollno=10;     // invalid – rollno is private
        s.out();         // valid
        marks=56.5;      // invalid – no object
        return 0;
}
```

Access Specifiers - Example

```
class A
{
        int a, b;
protected :
        void init();
        char c;
public:
        int z;
        void in();
        void out();
};
```

```
void main()
{
        A x, y;
        x.in();
        x.init();   //invalid
        x.z=45;
        y.c='A';    //invalid
        y.out();
        y.b=555;   //invalid
}
```

Points to Remember

- The default access specifier of the class is **private.**
- Any of the access specifiers may be omitted in the class definition.
- The access specifiers may not be in the order
- Creation of a class does not create memory for data members.
- Memory will be created for the data members only after creating variable of class type **(object).**
- Generally all data members are made as private to achieve data security and all member functions are made as public

Class Definition - Example

```
class student
{
private :
        int admno;
        char name[21];
        float marks;
public :
        void input();
        void output();
} ;
```

```
class product
{
        char product_ID[21], name[31];
        int qty;
        float unit_price;
protected :
        float total;
        float calculate();
public:
        void getdata();
        void printdata();
} ;
```

Object

Object

- It is an **instance** of a class.
- In simple words, a **variable** of type class is called as object.

Example of a class definition:

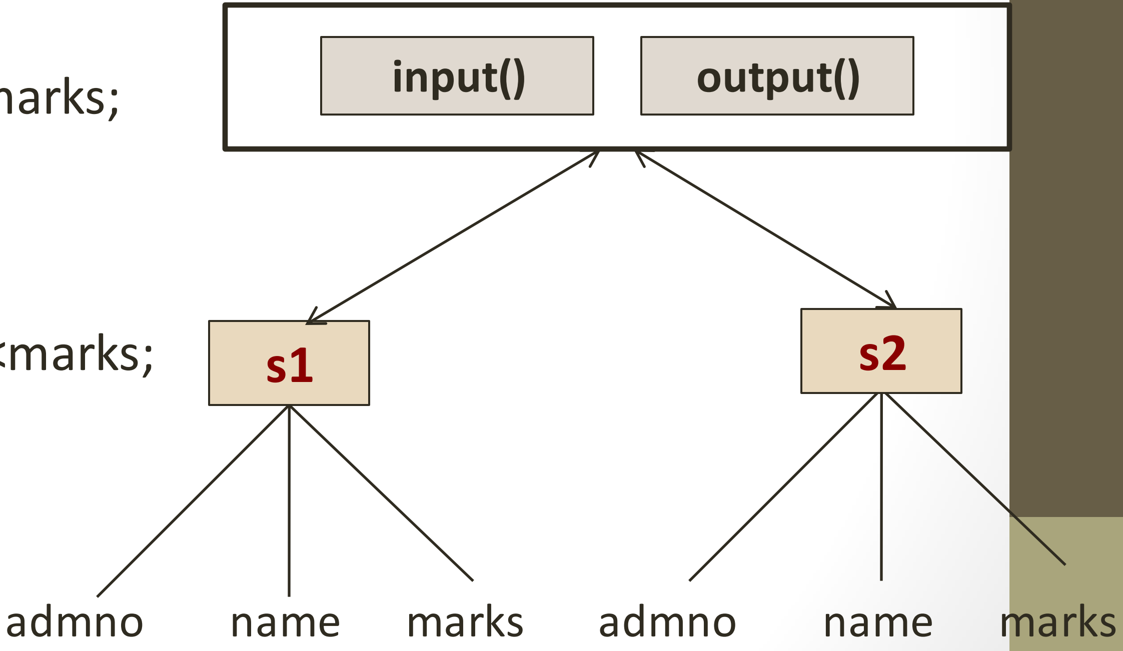
```
class student
{
private :
            int admno;
            char name[21];
            float marks;
public :
            void input();
            void output();
} ;
```

```
int main()
{
            student s;
            s.input();
            s.output();
}
```

Creation of Object - Example

```
class student
{
        private :
        int admno;
        char name[21];
        float marks;
        public :
        void input()
        {
                cin>>admno>>name>>marks;
        }
        void output()
        {
                cout<<admno<<name<<marks;
        }
} ;
```

```
int main()
{
                student s1, s2;
                s1.input(); s2.input();
                s1.output(; s2.output();
                return(0):
}
```



Differences between structure and class

When they are used in C language	
Structure	**Class**
Only variables are the members of the structure.	Both variables and functions are the members of the class.
Keyword used for definition is struct.	Keyword used for definition is class.
Access specifiers(private, protected, public) cannot be used with variables.	Any of the access specifiers can be used.
Default access specifier is public.	Default access specifier is private.
Structure variables can access all the elements using dot(.) operator.	Class objects can access only public members using dot(.) operator.
No friend functions, no inheritance concept in structures.	Friend functions and OOPs concepts are implemented in classes.

Differences between structure and class

When they are used in C++ language	
Structure	**Class**
Keyword used for definition is struct.	Keyword used for definition is class.
Default access specifier is public.	Default access specifier is private.

Methods of a Class

Definition of Member Functions

- Inside the class (Inline functions)
- Outside the class (Outline functions)

Inline functions

Brief idea about inline functions …….

- It is a function that is expanded in line when it is invoked.
- Inline function runs little faster than a normal outline function.
- Avoids the function call overhead
- The compiler replaces function call with the corresponding function code.
- Only once need to be changed

Example of Inline functions

```
inline int small(int a, int b)
{   int s;
    s=a<b?a:b;
    return (s);
}
```

```
inline int square(int a)
{
    return (a*a);
}
```

Limitations of inline function:

A function cannot be made inline when

- It has a big instruction code.
- It has for, while, do-while, goto and switch statements.
- It has static variables.
- It is recursive.

Example of Inline functions

To write a program to find the multiplication values and the cubic values using inline function.

```
#include<iostream>
Using namespace std
#include<conio.h>

class line
{
   public:
            inline float mul(float x,float y)
            {
                        return(x*y);
            }
            float cube(float x)
            {
                        return(x*x*x);
            }
};
```

```
void main()
{
          line obj;
          float val1,val2;
          clrscr();
          cout<<"Enter two values:";
          cin>>val1>>val2;
          cout<<obj.mul(val1,val2);
          cout<<obj.cube(val2);
          getch();
}
```

Output:

```
     Enter two values: 5  7
     Multiplication Value is: 35
     Cube Value is: 125
```

Class Methods

Defining a Member Function inside the class

- By default it becomes ***inline*** function.
- The keyword inline is not necessary in declaration and definition of member functions.
- Since inline is only a request to the compiler, it can ignore it and can compile as normal outline function if the function does not meet the requirements of an inline function.

Example of a class with M.F defined inside the class

```
class flight
{
            int flightno;
            char flightname[21];
            float fare;
            public:
            // M.F definition inside the class
            void input()
            {
                        cout<<“Enter flightno, name and fare\n”;
                        cin>>flightno>>name>>fare;
            }
            void output()
            {
                        cout<<“Flightno:”<<flightno<<endl;
                        cout<<“Flight Name:”<<flightname<<endl;
                        cout<<“Fare:”<<fare<<endl;
            }
};
```

```
void main()
{
            flight f;
            f.input();
            f.output();
}
```

Class Methods

Defining a Member Function outside the class

- **Scope Resolution Operator ::** is used.
- :: is used to resolve the identity and scope of the function.

- **Syntax :**

```
return_type class_name :: function_name(arguments)
{
        statements
}
```

Example of a class with M.F defined outside the class

Write a program to read and print flight's information which includes flight no, name and fare

```
class flight
{
        int flightno;
        char flightname[21];
        float fare;
        public:
        void input();
        void output(;
};

void flight :: input()
{
  cout<<"Enter flightno, name and fare\n";
  cin>>flightno>>name>>fare;
}
```

```
void flight :: output()
{
  cout<<"Flightno:"<<flightno<<endl;
  cout<<"Flight Name:"<<flightname<<endl;
  cout<<"Fare:"<<fare<<endl;
}
void main()
{
        flight f;
        f.input();
        f.output();
}
```

Example of two classes with M.F defined outside the classes

```
class A
{           int a;
            float b;
            public:
            void init();
            void output();
};
class B
{           char ch;
            double d;
            public:
            void init();
            void display();
};
void A :: init()
{
            a=16;
            b=5.6
}
```

```
void A :: output()
{
            cout<<“a=“<<a<<“   b=“<<b;
}
void B :: init()
{
            ch=‘H’;
            d=67.89;
}
void B :: display()
{
            cout<<“ch=“<<ch<<“   d=“<<d;
}
void main()
{
            A x;  B y;
            x.init();        y.init();
            x.output();    y.display();
}
```

Nesting of Member Function

- A member function can be called inside another member function of the same class.
- It can be called without using object name and dot operator.

Nesting of Method

Nesting of Member Function - Example

```
class set
{
        int m, n;
        public:
        void input();
        void display();
        private :
        int big(int, int);
        int small(int, int);
};
void set :: input()
{
        cin>>m>>n;
}
void set :: display()
{
        int s,l;
        s=small(m,n);
        l=big(m,n);
        cout<<s<<endl<<b;
}
```

```
void set :: big(int x, int y)
{
        if(x>y)
        return x;
        else return y;
}

void set :: small(int x, int y)
{
        if(x<y)
        return x;
        else return y;
}
void main()
{
        set s;
        s.input();
        s.display();
}
```

Example

Write a C++ program to define a class called box with length, breadth and height as data members and input(), print() and volume() as member functions.

```
#include<iostream>
using namespace std;
class box
{
     float length, breadth,height,vol;
     public:
     void input();
     void print();
     float volume();
};
void box::input()
{
    cout<<"Enter length, breadth and
               height\n";
    cin>>length>>breadth>>height;
    vol=volume();
}
```

```
void box::print()
{
    cout<<"Length="<<length<<endl;
    cout<<"Breadth="<<breadth<<endl;
    cout<<"Height="<<height<<endl;
    cout<<"Volume="<<vol<<endl;
}
float box::volume()
{
     return(length*breadth*height);
}
int main()
{
   box b;
   b.input(); b.print();
   return(0);
}
```

Static Data Members

Static Data Members

Characteristics

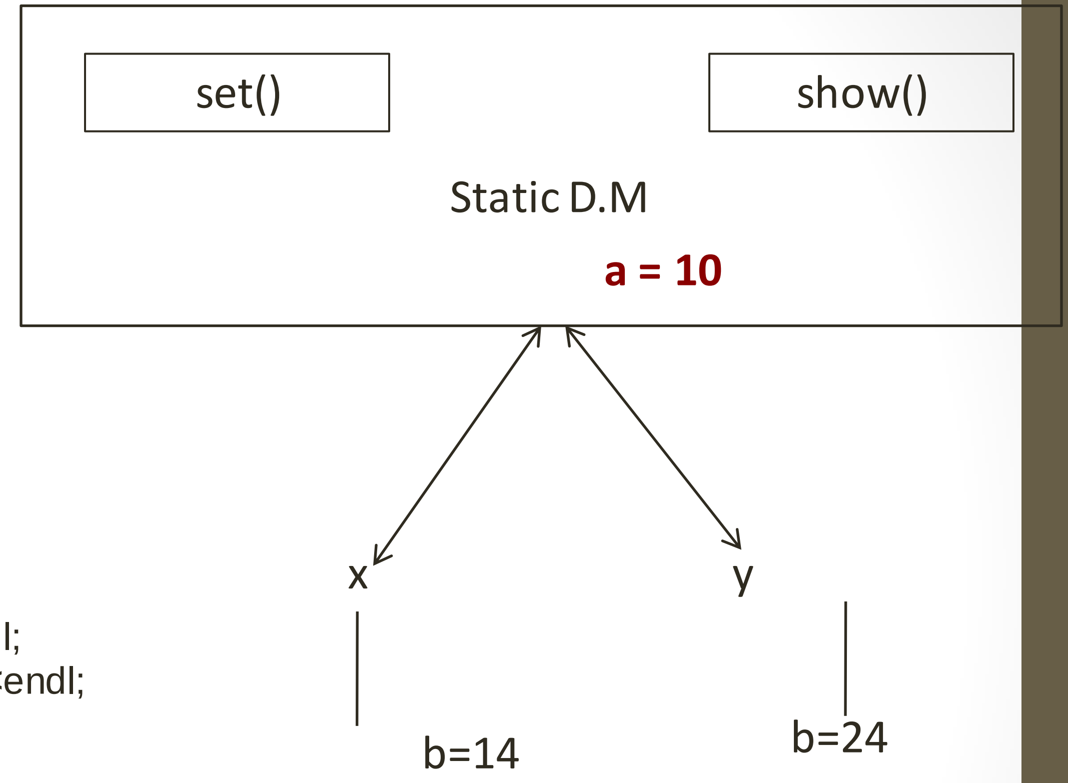
- Default initial value is 0.
- Only one copy of the static data members is created for the entire class
- It is visible only within the function but its lifetime for the entire class.
- It is declared inside the class and defined outside the class (::)
- Accessed using class name not with object

Example Program

- REFER NOTES

Static Data Members - Example

```
#include <iostream>

class shared
{
        static int a;     // Declare a
        int b;
        public:
        void set(int i)
        {
                b=i;
               a+=5;
        }
        void show()
        {
        cout<<"This is static a : "<<a<<endl;
        cout<<"This is non static b :"<<b<<endl;
        }
} ;
int shared :: a;      // Define a
int main()
{      shared x, y;
        x.set(14);
        x.show();
        y.set(24);
        y.show();
        return 0;
}
```



Output :

```
This is static a : 5
This is non static b : 14
This is static a : 10
This is non static b : 24
```

Static Data Members - Example

```
#include <iostream>
using namespace std;
class A
{       int number;
        static int  count;
        public:
        void getdata(int x, int y)
        {       number=x;
                  count=y;
                count++;
        }
        void print()
        {
              cout<<number<<“  “<<count<<endl;
        }
} ;
```

```
int A :: count=2;
int main()
{               A a, b, c;
                b.print();
                a.getdata(5,6);
                b.getdata(10,3);
                a.print();
                b.print();
                c.print();
                c.getdata(4, 50);
                a.print();
                return 0;
}
```

Output :

```
Junk    2
5       4
10      4
Junk    4
5       51
```

Static Data Members - Example

```
#include <iostream>
using namespace std;
class A
{       int a;
        float b;
        static int  count;
        public:
        void init()
        {       a=1;
                b=2.2;
                count++;
        }
        void print()
        {
            cout<<a<<“  “<<b<<“  “<<count<<endl;
        }
} ;
```

```
int A :: count=1;
int main()
{           A  x,y;;
            x.print();
            x.init();
            y.init();
            x.print();
            x.init();
            y.print();
            return 0;
}
```

Output :

```
Junk      Junk      1
1         2.2       3
1         2.2       4
```

Static Data Members - Example

```
#include <iostream>
using namespace std;
class A
{     int a;
      float b;
      public:
      static int  count;
      void init()
      {     a=1;
            b=2.2;
            count++;
      }
      void print()
      {
          cout<<a<<“  “<<b<<“  “<<count<<endl;
      }
} ;
```

```
int A :: count;
int main()
{
      A::count=99;
      A  x,y;
      x.init();
      y.init();
      x.print();
      cout<<x.count;
      return 0;
}
```

Output :

```
1       2.2     101
101
```

Static Data Members - Example

```
#include <iostream>
using namespace std;
class A
{
        public:
        static int a;
};

int A::a;
int main()
{

        // initializing a before creating object
         A::a=3;
        cout<<"This is initial value of a :"<<A::a<<endl;
        A x;
        cout<<"This is x.a :"<<x.a<<endl;
        A::a+=10;
        cout<<"A::a="<<A::a<<"    x.a="<<x.a;
        return 0;
}
```

Output :

```
This is initial value of a :3
This is x.a :3
A::a=13    x.a=13
```

Static Data Members

Uses of Static Data Members

- to provide access control to some shared resource used by all objects of a class.
- to keep track of the number of objects of a particular class type that are in existence.

Static Methods

Class Members

Static Member Function

Characteristics

- It can access static members of the same class.
- It is called using class name along with :: operator.
- It is used to pre-initialize private static data before any object is actually created.

Static Member Function - Example

```
// Example of a static Member Function
#include <iostream>
using namespace std;
class item
{
    int number;
    static int count;
    public:
    void getdata(int a, int b)
    {
        number=a;
        count=b;
        number++;
        count++;
    }
    void print()
    {
        cout<<number<<"   "<<count<<endl;
    }
    static void showcount();
};
```

```
void item::showcount()
{
    // cout<<number;  INVALID
    cout<<count<<endl;
}
int item::count;

int main()
{

   item a;
   item::showcount();
   a.getdata(11,7);
   a.print();
   item::showcount();
   return 0;
}
```

Output :

```
0
12  8
8
```

this

OOPs concepts

this Pointer:

- The **this** pointer is always a constant pointer.
- **this** is an internal pointer which points to the current object.
- accessible only within the non static member functions of a **class**
- It points to the object for which the member function is called.
- Returning *this will return a reference to the object that was implicitly passed to the function by C++.

```
#include <iostream>
using namespace std;

struct X {
private:
  int a;
public:
  void Set_a(int a) {

this->a = a;
  }
   void Print_a() { cout << "a = " << a << endl; }
};

int main() {
  X xobj;
  int a = 5;
  xobj.Set_a(a);
  xobj.Print_a();
}
```

Constructors

Class Special Member Functions

Constructors

- C++ allows objects to initialize themselves when they are created.
- This initialization has to be done using a member function.
- This special member function is called **constructor.**

Definition

A constructor is a special member function having the ***same name as its class*** used to initialize objects of that class.

Properties of Constructors

- **Constructor has same name as its class because**
  - The compiler can automatically call this special member function without any explicit call.
  - Hence the responsibility of calling constructor is reduced to the programmer.
- If the programmer does not define a constructor, the compiler automatically creates one and by default it is **public.**

Properties of Constructors

- A constructor is called automatically when an object is created.
- It obeys the rules of access specifiers.
- **A constructor has no return type not even void but can take arguments.**
- It is not possible to take the address of a constructor.
- A Class can have multiple constructor
- Define in public area

TYPES & WAYS TO CALL CONSTRUCTOR

FOR Example Programs on:

- Default Constructor
- Parameterized Constructor
- Copy Constructor
- Dynamic Constructor

********* REFER NOTES *********

Constructor - Example

```
// constructor
class A
{
        int a;
        float b;
        public:
        A();        // default constructor
        void print();
};
A::A()
{
        a=45;
        b=67.7;
        cout<<"Constructor working\n";
}
void A::print()
{
        cout<<a<<"  "<<b<<endl;
}
```

```
int main()
{
        A x;        // constructor called
        x.print();
        A y;        // constructor called
        y.print();
}
```

Output :

```
Constructor working
45   67.7
Constructor working
45   67.7
```

Parameterized Constructors

```
class A
{
        int a;
        float b;
        public:
        A(int, float);
        void print();
};
A::A(int n1, float n2)
{
        a=n1;
        b=n2;
        cout<<“Parameterized constructor\n”;
}
void A::print()
{
        cout<<a<<“  “<<b<<endl;
}
```

```
int main()
{
        //   A x;   ERROR !
        A x(3,6.7), y(9,6.89);
        x.print();
        y.print();
}
```

Parameterized Constructors

```
class A
{
        int a;
        float b;
        public:
        A();                //  function 1
        A(int, float);      //  function 2
        void print();       //  function 3
};
A::A()
{
        a=10;
        b=20.5;
        cout<<"Default constructor\n";
}
A::A(int n1, float n2)
{
        a=n1;
        b=n2;
        cout<<"Parameterized constructor\n";
}
```

```
void A::print()
{
        cout<<a<<"  "<<b<<endl;
}

int main()
{

        A x(3,6.7)          // 2
        A y;                // 1
        x.print();          // 3
        y.print();          // 3
        return(0);

}
```

Output:

```
Parameterized constructor
Default Constructor
3    6.7
10   20.5
```

Constructors with One Parameter

```
// Constructors with One Parameter
#include <iostream>
using namespace std;
class X
{
        int a;
        public:
        X(int j)
        { a = j; }
        int geta()
        { return a; }
};
```

```
int main()
{
        X p = 99;
        // passes 99 to j
        cout << ob.geta();
        // outputs 99
        return 0;
}
```

Write a C++ program to count the no. of objects created.

```
#include<iostream>
using namespace std;
class A
{
     static int count;
     public:
     A()
     {
           count++;
     }
     static void print()
     {
         cout<<"No. of objects created = "<<count<<endl;
     }
};
int A::count;
int main()
{
   A x, y, z;
   A::print();
   return(0);
}
```

- **Copy Constructor**
- The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

The copy constructor is used to:

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

Destructors

Class Special Member Functions

Destructors

- C++ allows objects to deinitialize themselves when they got out of the scope.
- This deinitialization has to be done using a member function.
- C++ creates a special member function for this purpose.

Definition

A destructor is a special member function having the ***same name as its class preceded by tilde (~) sign*** used to deinitialize objects of that class.

Properties of Destructors

- If the programmer does not define a destructor, the compiler automatically creates one and by default it is **public.**
- A destructor is called automatically when an object goes out of the scope.
- It obeys the rules of access specifiers.
- **A destructor has no return type not even void . Also it cannot take any arguments.**
- It is not possible to take the address of a destructor.

Destructor - Example

```
class A
{
        int a;
        public:
        A();
        ~A();
        void print();
};
A::A()
{
        a=10;
        cout<<“Constructor working\n”;
}
A::~A()
{
        cout<<“Destructor working\n”;
}
A::print()
{
        cout<<“a=“<<a<<endl;
}
```

```
int main()
{
        A  x, y;
        x.print();
        y.print();
        return(0);
}
```

Output :

```
Constructor working

Constructor working

a=10

a=10

Destructor working

Destructor working
```

Constructor & Destructor - Example

```
class stack
{
    int s[10], top;
    public:
    stack(); // constructor
    ~stack(); // destructor
    void push(int i); int pop();
};
stack::stack()
{
    top = -1;
    cout << "Stack Initialized\n";
}
stack::~stack()
{
    cout << "Stack Destroyed\n";
}
void stack::push(int i)
{   if(tos==10-1)
    cout << "Stack is full.\n";
else
    s[++top] = i;      }
```

```
    int stack::pop()
{
    if(top==-1)
    { cout << "Stack underflow.\n";
          return 0;
    }
    top--;
    return s[top];
}
int main()
{
    stack a, b;
    a.push(1);
    b.push(2);
    a.push(3);
    b.push(4);
    cout << a.pop() << " ";
    cout << a.pop() << " ";
    cout << b.pop() << " ";
    cout << b.pop() << "\n";
    return 0;
}
```

Invocation of Constructors & Destructors

When Constructors and Destructors are executed

- **Constructor** is called when the object comes into existence and an object's
- **Destructor** is called when the object is destroyed.
- A local object's constructor is executed when the object's declaration statement is encountered. The destructors for local objects are executed in the reverse order of the constructor functions.
- **Global objects** have their constructors execute **before main ( ) begins execution.** Global constructors are executed in order of their declaration, within the same file.
- Global destructors execute in reverse order after main ( ) has terminated.

Assignment

1. Write a C++ program to define a class called box with length, breadth and height as data members and input(),print(),and volume() as member functions.
2. Explain a general form of a class specification .Explain class objects.
3. Explain scope resolution operator with example.
4. Compare struct and class keyword in c++.
5. What is data hiding ?
6. What are constructor? Explain different types of constructors with example.
7. Explain the two different ways of defining member function with an example
8. When constructors and destructors are called? Explain with C++ program.
9. Explain static members with example program.
10. Explain static member function with example program.

UNIT-3: Classes and Objects II

Topics covered

- **Classes & Objects –II:** Friend functions, Passing objects as arguments,
- Returning objects, Arrays of objects, Dynamic objects, Pointers to objects,
- Copy constructors, Generic functions and classes, Applications
- Operator overloading using friend functions such as +, - , pre-increment,post-increment,[ ] etc., overloading <<, >>.

Friend Functions

- **A friend function,** although not a member function, has full access rights to the private members of the class.
- <u>Special characteristics of friend functions are</u>
  - It is not in the scope of the class to which it has been declared as friend.
  - It can be invoked like a normal functions without the help of any object
  - It can be declared either in the public or private part of a class
  - Usually it has the objects as arguments.
  - Often used in operator overloading.

```
#include <iostream.h>
class myclass {
int a, b;
public:
friend int sum(myclass x);
void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j)
{    a = i;
     b = j;
}
// Note: sum() is not a member function of any class.
int sum(myclass x)
{ /* Because sum() is a friend of myclass, it can directly access a and b. */
     return x.a + x.b;
}
int main()
{
     myclass n;
     n.set_ab(3, 4);
     cout << sum(n);
     return 0;
}
```

Friend Classes

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

Ex:

```
// Using a friend class.
#include <iostream.h>
class TwoValues {
    int a;
    int b;
    public:
    TwoValues(int i, int j) { a = i; b = j; }
    friend class Min;
};
class Min {
    public:
    int min(TwoValues x);
};
int Min::min(TwoValues x) {  return x.a < x.b ? x.a : x.b; }
int main()
{
    TwoValues ob(10, 20);
    Min m;
    cout << m.min(ob);
    return 0;
}
```

Passing Objects to Functions

- Objects may be passed to functions in just the same way that any other type of variable can.
- This can be done in two ways
  - A copy of the entire object is passed to the function(call by value)
  - The address of the object is transferred to the function
  - Example program /* Refer Notes*/

Returning Objects

/*Refer Notes*/

- **A function may return an object to the caller.**

```
#include <iostream.h>
class myclass {
    int i;
    public:
    void set_i(int n) { i=n; }
    int get_i() { return i; }
};
myclass f(); // return object of type myclass
int main()
{
    myclass o;
    o = f();
    cout << o.get_i() << "\n";
    return 0;
}
myclass f()
{
    myclass x;
    x.set_i(1);
    return x;
}
```

Object assignment

- If both objects are of the same type, you can assign one object to another.
- This causes the data of the object on the right side to be copied into the data of the object on the left.

```
// Assigning objects.
#include <iostream>
using namespace std;
class myclass {
    int i;
    public:
    void set_i(int n) { i=n; }
    int get_i() { return i; }
};
int main()
{
    myclass ob1, ob2;
    ob1.set_i(99);
    ob2 = ob1; // assign data from ob1 to ob2
    cout << "This is ob2's i: " << ob2.get_i();
    return 0;
}
```

Arrays of Objects

- Arrays of variables of type class are known as arrays of objects.

```
#include <iostream>
using namespace std;
class cl {
    int i;
    public:
    void set_i(int j) { i=j; }
    int get_i() { return i; }
};
int main()
{
    cl ob[3];
    int i;
    for(i=0; i<3; i++) ob[i].set_i(i+1);
    for(i=0; i<3; i++)
    cout << ob[i].get_i() << "\n";
    return 0;
}
```

```
#include <iostream>
using namespace std;
class cl {
      int i;
      public:
      cl(int j) { i=j; } // constructor
      int get_i() { return i; }
};
int main()
{
      cl ob[3] = {1, 2, 3}; // initializers
      int i;
      for(i=0; i<3; i++)
      cout << ob[i].get_i() << "\n";
      return 0;
}
```

Dynamic objects

- C++ provides two dynamic allocation operators: **new** and **delete**. These operators are used to allocate and free memory at run time.
- The **new** operator allocates memory and returns a pointer to the start of it.
- The **delete** operator frees memory previously allocated using **new**.

- The general forms of **new** and **delete** are shown here:
- *p_var* = new *type*;
- delete *p_var*;
- Here, *p_var* is a pointer variable that receives a pointer to memory that is large enough to hold an item of type *type*.

Program that allocates memory to hold an integer:

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
      int *p;
      try {
            p = new int; // allocate space for an int
            } catch (bad_alloc xa) {
      cout << "Allocation Failure\n";
      return 1;
      }
      *p = 100;
      cout << "At " << p << " ";
      cout << "is the value " << *p << "\n";
      delete p;
      return 0;
}
```

Initializing Allocated Memory

- General form of **new** when an initialization is included:
- *p_var* = new *var_type (initializer);*
- *Ex:*
- **Allocating Arrays**
- We can allocate arrays using **new** by using this general form:
- *p_var* = new *array_type [size];*
- Here, *size* specifies the number of elements in the array.
- To free an array, use this form of **delete:**
- delete [ ] *p_var*;

Program allocates a 10-element integer array.

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
    int *p, i;
    p = new int [10]; // allocate 10 integer array
                                 return 1;
    for(i=0; i<10; i++ )
    p[i] = i;
    for(i=0; i<10; i++)
    cout << p[i] << " ";
    delete [] p; // release the array
    return 0;
}
```

Allocating Objects

- We can allocate objects dynamically by using **new**
- The dynamically created object acts just like any other object. When it is created, its constructor (if it has one) is called. When the object is freed, its destructor is executed.
- EX:Program that creates a class called **balance** that links a person's name with his or her account balance.

The *this* pointer

- When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object.

  This pointer is called ***this*****.**
- **The *this* pointer is always constant pointer. The *this* pointer always points at the object with respect to which the function was called.**
- **Important application of the pointer *this* is to return the object it points to.**

Copy constructors

- Important forms of an overloaded constructor is the copy constructor.
- Copy constructor is a special type of parameterized constructor.As its name implies it copies one object to another.
- General form of a copy constructor is

```
classname(const classname &o)
{
      //body of function;
}
```

- The copy constructors applies only to initialization.

- C++ defines two types of situations in which the value of one object is given to another.

1)Assignment

2)Initialization.Which can occur any of 3 forms.

- When one object explicitly initializes another
- When a copy of an object is made to passed to a function
- When a temporary object is generated.

Ex: Program

```
#include<iostream.h>
 class code{
      int id;
      public:code(){id=0;}
                    code(int a){id=a;}
                    code(code &x){ id=x.id;}
                    void display(){cout<<id;}
      };
      int main()
      {
            code a(100);
            code b(a);
            code c=a;
            code d;
            d=a;
            code e;
            cout<<"\n id of a is:"; a.display();
             cout<<"\n id of b is:"; b.display();
             cout<<"\n id of c is:"; c.display();
             cout<<"\n id of d is:"; d.display();
             cout<<"\n id of e is:"; e.display();
            return 0;
      }
```