Corurse Code-22316
Lesson no: 1

Course Name: OOP with C++
Lesson Name: Introduction of OOP

Unit Structure:

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

1, 0

Machine Language

Assembly Language

Procedure- Oriented

Object Oriented Programming

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 material*s". To build today's complex software it is just not enough to put together a sequence of programming statements and sets of procedures and modules; we need to incorporate sound construction techniques and program structures that are easy to comprehend implement and modify.

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

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

1.3 Procedure-Oriented Programming

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

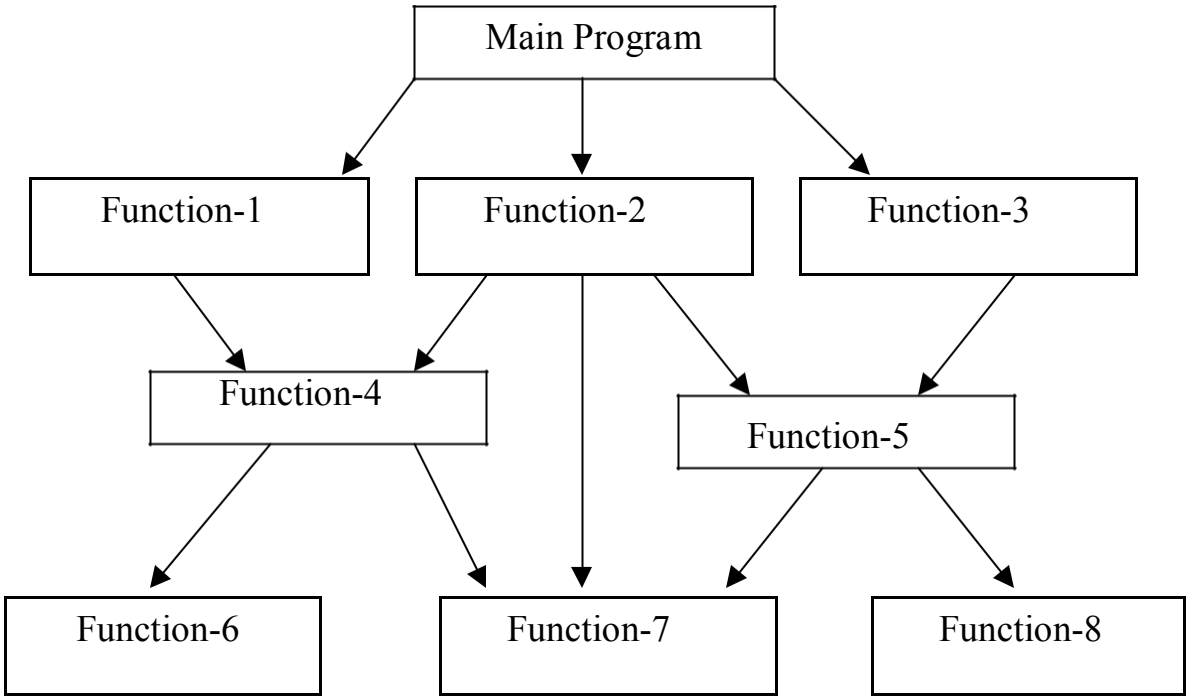


Fig. 1.2 Typical structure of procedural oriented programs

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

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

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

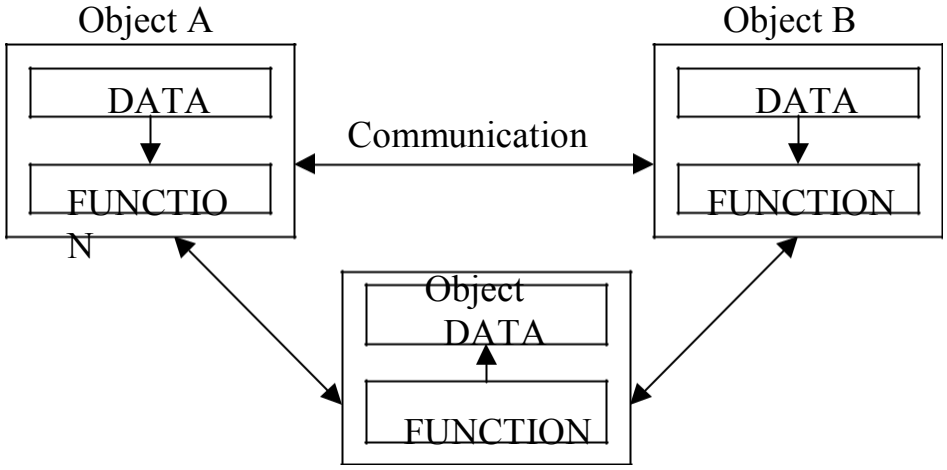
Some Characteristics exhibited by procedure-oriented programming are:

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

1.4 Object Oriented Paradigm

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

Organization of data and function in OOP



Some of the features of object oriented programming are:

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

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

1.5 Basic Concepts of Object Oriented Programming

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

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

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

1.5.1 Objects

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

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

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

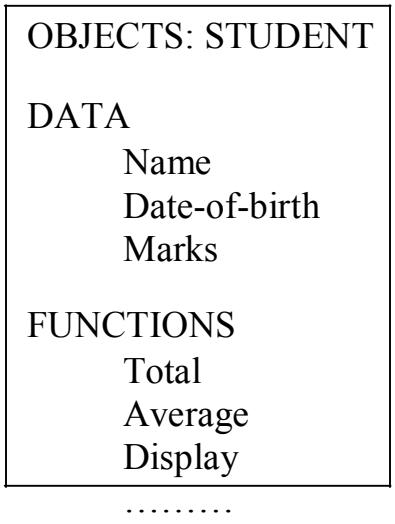


Fig. 1.5 representing an object

1.5.2 Classes

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

Fruit Mango;

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

1.5.3 Data Abstraction and Encapsulation

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

insulation of the data from direct access by the program is called *data hiding or information hiding*.

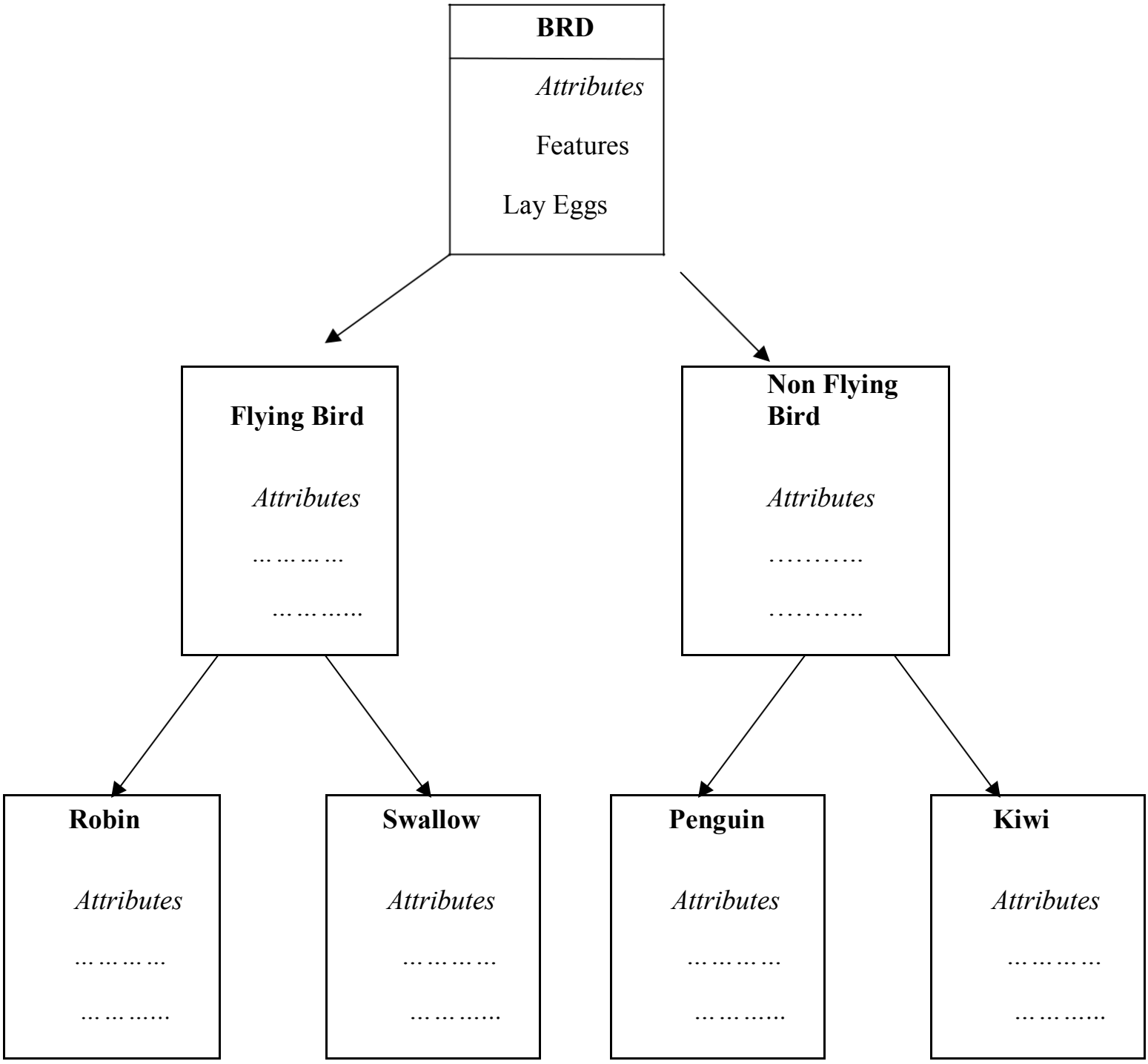
Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all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

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

1.5.4 Inheritance

Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of *hierarchical classification.* For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in fig 1.6.
In OOP, the concept of inheritance provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes. The real appeal and power of the inheritance mechanism is that it

Fig. 1.6 Property inheritances



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

1.5.5 Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than on form. An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as *operator overloading*.

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

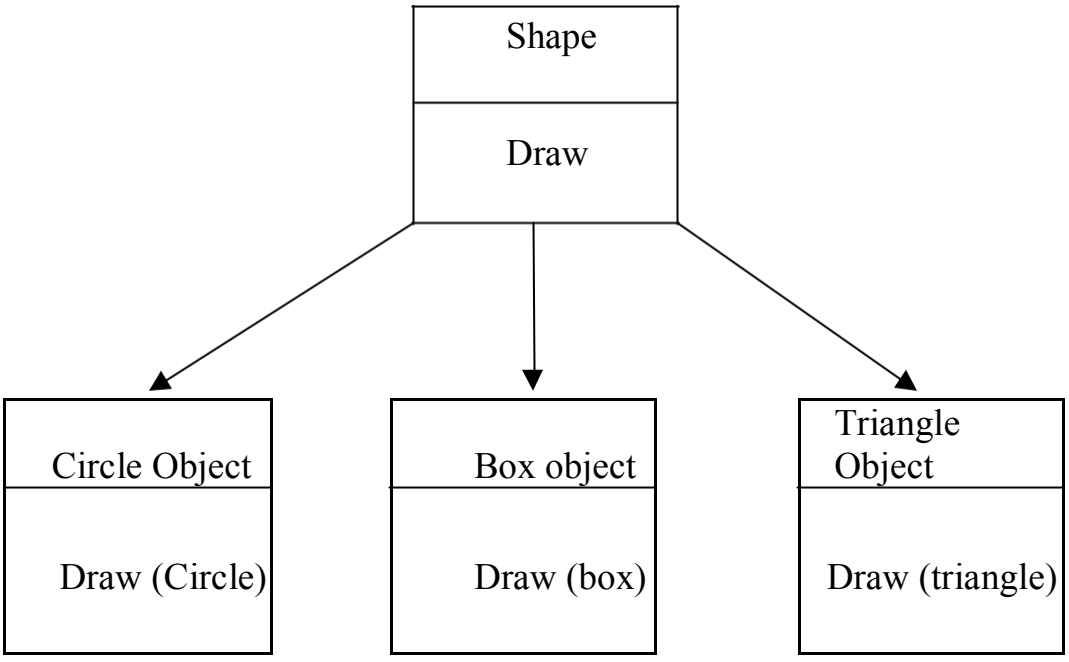


Fig. 1.7 Polymorphism

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

1.5.6 Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

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

1.5.7 Message Passing

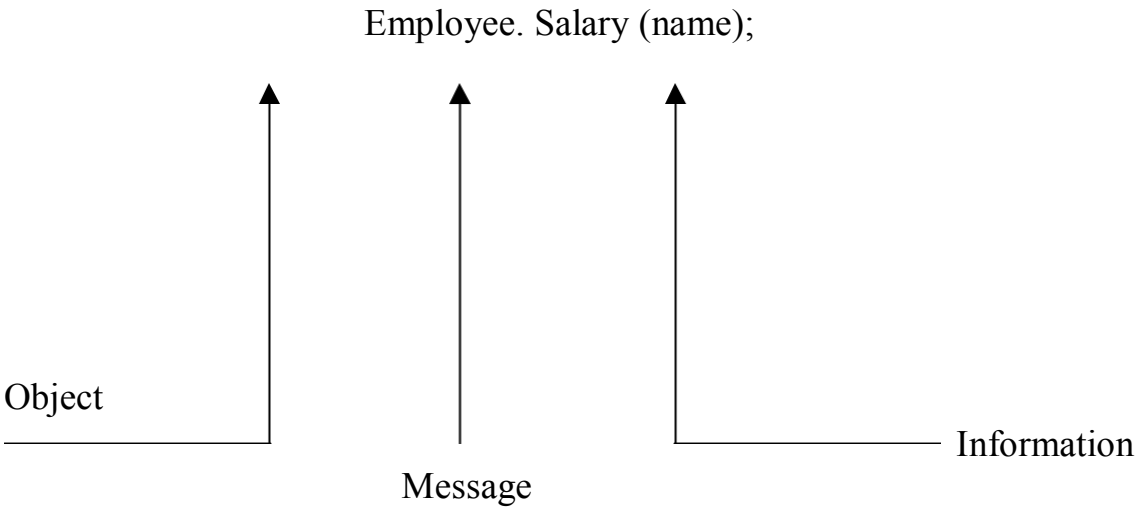
An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

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

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

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

Example:



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

1.6 Benefits of OOP

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

- Through inheritance, we can eliminate redundant code extend the use of existing
- Classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure program that can not be invaded by code in other parts of a programs.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map object in the problem domain to those in the program.

- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more detail of a model can implemental form.
- Object-oriented system can be easily upgraded from small to large system.
- Message passing techniques for communication between objects makes to interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

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

1.7 Object Oriented Language

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

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.

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

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.

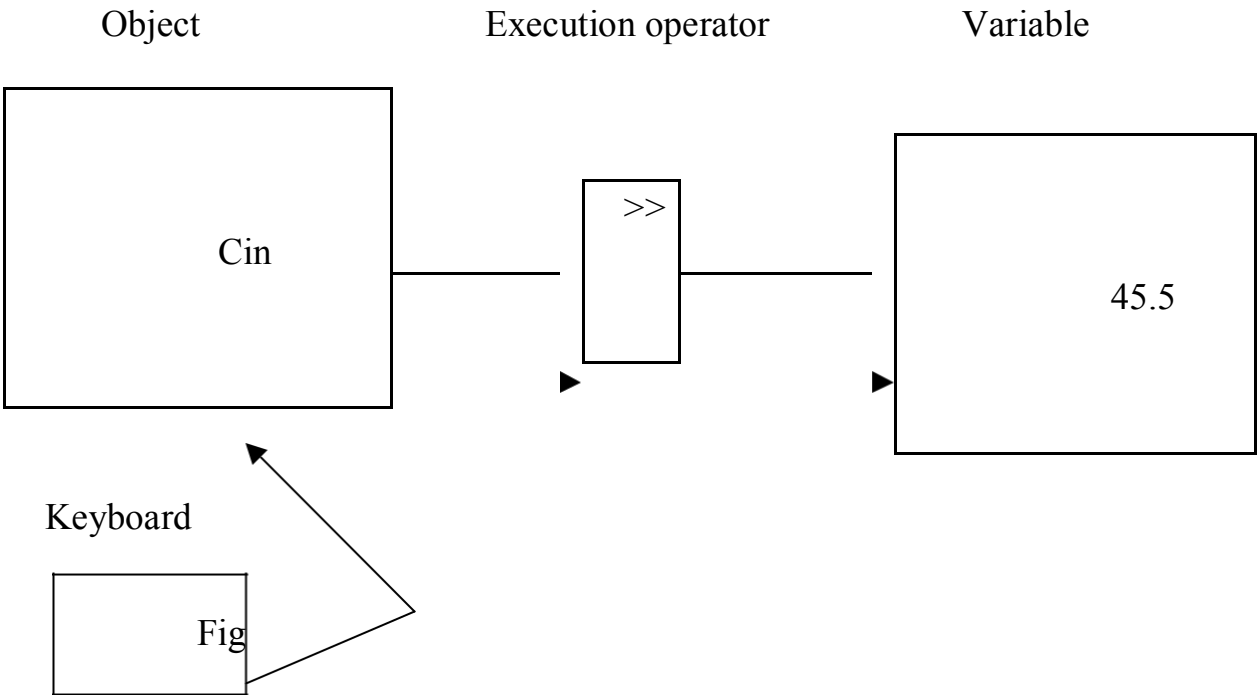
1.11.2 Input Operator

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.

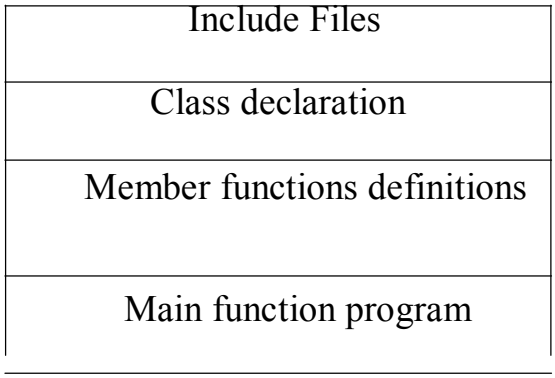
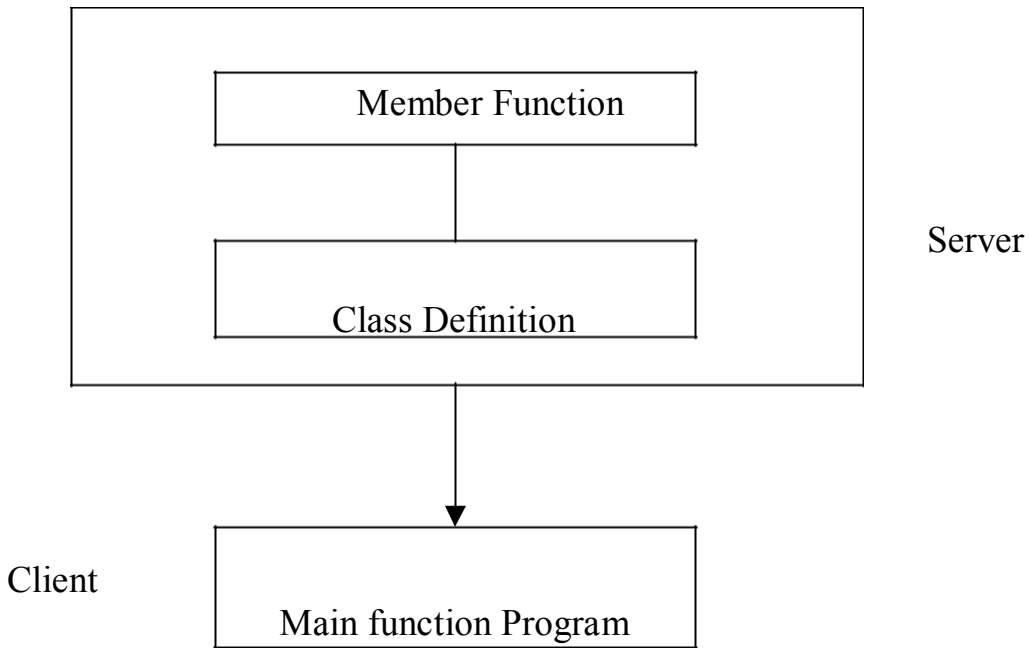


Fig 1.9 Structure of a C++ program

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.

Summary

- Software technology has evolved through a series of phases during the last five decades.
- POP follows top-down approach where problem is viewed as sequence of task to be performed and functions are written for implementing these tasks.
- POP has two major drawbacks:
- Data can move freely around the program.
- It does not model very well the real-world problems.
- OOP was inventing to overcome the drawbacks of POP. It follows down -up approach.
- In OOP, problem is considered as a collection of objects and objects are instance of classes.
- Data abstraction refers to putting together essential features without including background details.
- Inheritance is the process by which objects of one class acquire properties of objects of another class.
- Polymorphism means one name, multiple forms. It allows us to have more than one function with the same name in a program.
- Dynamic binding means that the code associated with a given procedure is not known until the time of the run time.
- Message passing involves specifying the name of the object, the name of the function and the information to be sent.
- C++ is a superset of C language.
- C++ ads a number of features such as objects, inheritance, function overloading and operator overloading to C.

- C++ supports interactive input and output features and introduces anew comment symbol // that can be used for single line comment.
- Like C programs, execution of all C++ program begins at main() function.

Keywords:

- Assembly Language
- Bottom up Programming
- C++
- Classes
- Data Abstraction
- Data Encapsulation
- Data Hiding
- Data Member
- Dynamic Binding
- Early Binding
- Function overloading
- Functions
- Global Data
- Hierarchical Classification
- Inheritance
- Late Binding
- #include
- Main( )
- Local data
- Machine Language
- Member Function
- Message Passing
- Methods
- Modular Programming
- Multiple Inheritances
- Object Based Programming
- Objective C
- Object Oriented Language
  Object Oriented
- Programming
- Objects
- Operator Overloading
- Polymorphism
  Procedure Oriented
- Programming
- Reusability
- Top down Programming
- Extraction Operator

- Cascading
- Namespace
- Class
- Object
- Operator overloading
- Comments
- Output operator
- cout
- edlin
- return ()
- Float
- Get from
- Operator
- Input operator
- Turbo c++
- iostream
- int
- using
- iostream.h
- windows
- Keyboard

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.

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.

Subject: Object Oriented Programming using C++

Corurse Code-22316

Lesson: Function in c++ &Object and classes

STRUCTURE

2.1 Introduction

4.2 Function Definition and Declaration

4.3 Arguments to a Function

4.3.1 Passing Arguments to a Function

4.3.2 Default Arguments

4.3.3 Constant Arguments

4.4 Calling Functions

4.5 Inline Functions

4.6 Scope Rules of Functions and Variables

4.7 Definition and Declaration of a Class

4.8 Member Function Definition

4.8.1 Inside Class Definition

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

4.9 Declaration of Objects as Instances of a Class

4.10 Accessing Members From Object(S)

4.11 Static Class Members

4.11.1 **Static Data Member**

4.11.2 **Static Member Function**

4.12 Friend Classes

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.

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.

```
// 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.

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;
}
```

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)
          {
              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 :

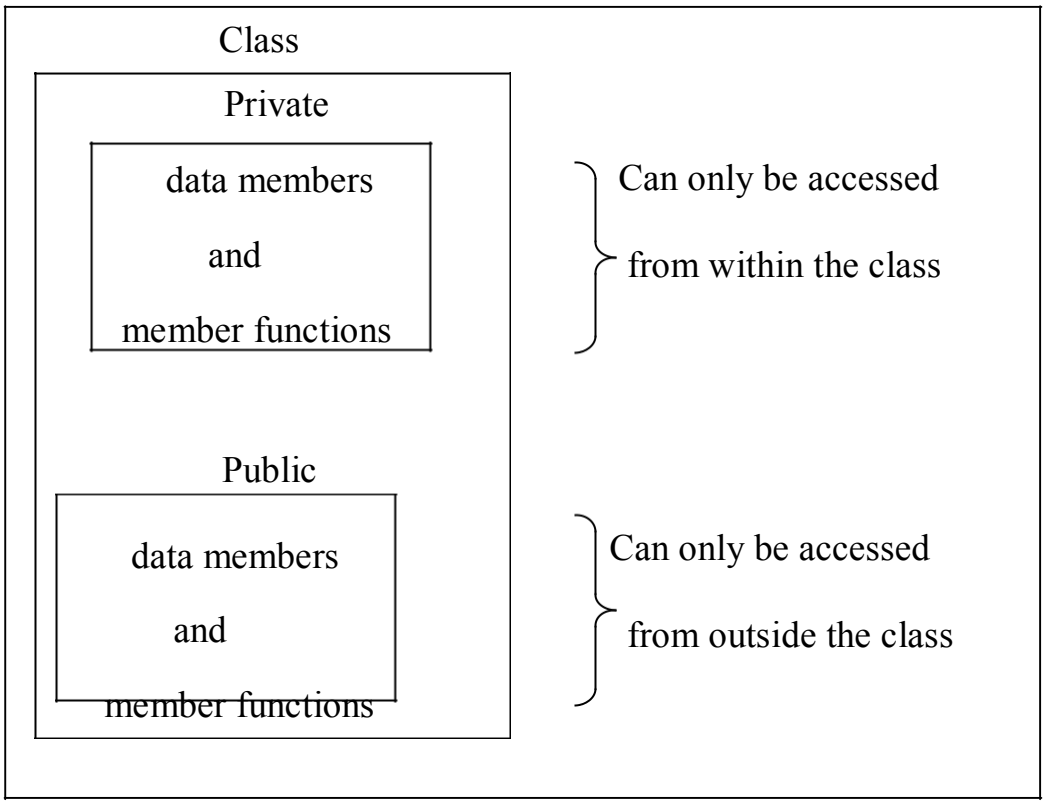
```
Class name_of _class

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

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

The body of the class has two keywords namely :

(i) *private* *(ii) public*

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



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

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

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

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,

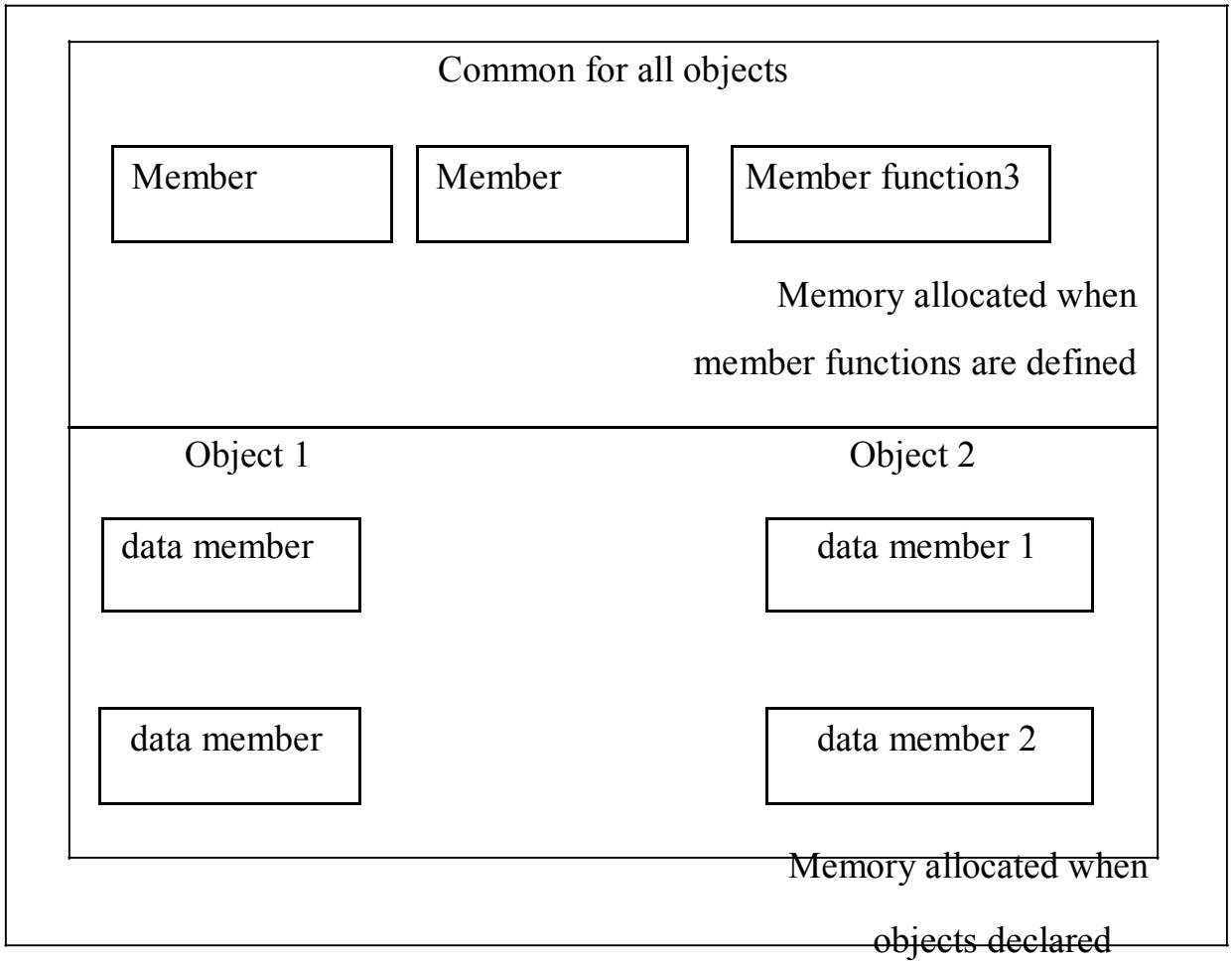
Largest ob1,ob2; //object declaration

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



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,

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.

4.13 Summary

In this Unit, we have discussed the concept of function in c++, its declaration and definition. we have also discussed the concept of class, its declaration and definition. It also explained the ways for creating objects, accessing the data members of the class. We have seen the way to pass objects as arguments to the functions with call by value and call by reference.

4.14 Keywords

Inline Functions:- An inline function is expanded in the line where it is invoked.

Member Function:- Private means that they can be accessed only by the functions within the class.

Classes:- When you create the definition of a class you are defining the attributes and behavior of a new type.

Objects:- Declaring a variable of a class type creates an object. You can have many variables of the same type (class).

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. 9. Define a class student with the following specifications:

Adm_no	integer
Sname	20 characters
Eng, math, science	float (marks in three subjects)
Total	float
Ctotal()	a function to calculate eng + math + science marks

Public member functions of class student

Takedata() function to accept values for adm_no , sname, marks in eng, math, science and invoke ctotal() to calculate total.

Showdata() function to display all the data members on the screen.

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.

3.

Subject: Object Oriented Programming using C++

Corurse Code-22316
Lesson: Constructors and Destructors,Operator Overloading and Type Conversions

Lesson No. : 3

STRUCTURE

3.1 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 de-initialize objects when they are destroyed. A destructor is called when an object of the class goes out of scope, or when the memory space used by it is de allocated with the help of **delete** operator.

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

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
            {
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.

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()
{
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 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.**

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"; }
//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 non-static member functions. Due to this, their first operands will be lvalues.

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

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

Some examples of Global Operator Functions are given below:

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

We can declare these Global Operator Functions as being friends of any other class.
Examples of operator overloading:

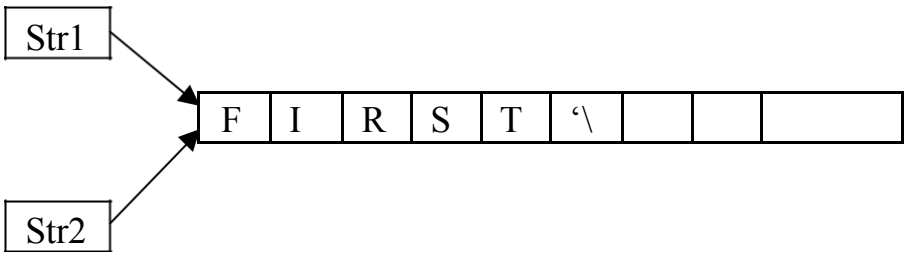
Operator overloading using friend.

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

 return z;
}

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

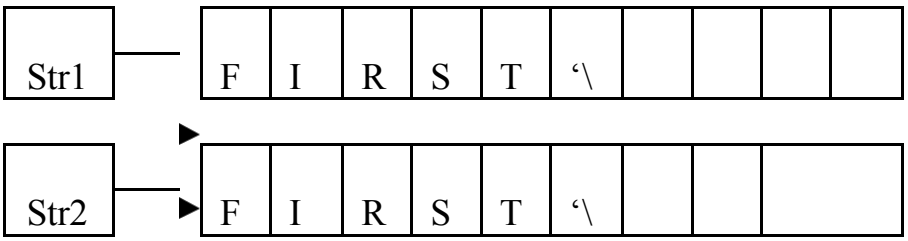
Operator overloading using member function:

```
Class abc
{
char * str;
int len ; // Present length of the string
int max_length; // (maximum space allocated to string)
public:
abc ( );  // 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:

```
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.

5.10 Summary

In this lesson , we discussed the concept and type of constructor and destructor. All the operators that can be overloaded. Even after writing operator overloaded functions, the precedence of operators remains unchanged. The '++' & '--' operators can be used as Postfix or Prefix operators. So, separate functions overloading them for both the different applications have been shown. we are of a view that Private data of a class can be accessed only in member functions of that class.

5.11 Keywords

Constructor: Constructors is special member functions of classes that are used to construct class objects.

Destructor: destructors are special member functions of classes that are used to destroy class objects.

Operator Overloading: Overloaded operators are implemented as functions and can be member functions or global functions.

3.12 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.

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”.

5.13 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.

Course: Object Oriented Programming using C++

Corurse Code-22316

Lesson: Inheritance(Extending Classes),Pointers,

Virtual Functions and
Polymorphism

Lesson No. : 4

STRUCTURE

4.1 Introduction

Inheritance allows a class to include the members of other classes without repetition of members. There were three ways to inheritance means, "public parts of super class remain public and protected parts of super class remain protected." Private Inheritance means "Public and Protected Parts of Super Class remain Private in Sub-Class".

Protected Inheritance means "Public and Protected Parts of Superclass remain protected in Subclass.

A pointer is a variable which holds a memory address. Any variable declared in a program has two components:

(i) Address of the variable

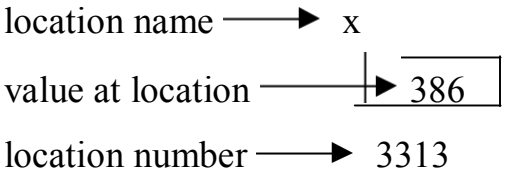
(ii) Value stored in the

variable. For example,

int x = 386;

The above declaration tells the C++ compiler for :

(a) Reservation of space in memory for storing the value.
(b) Associating the name x with his memory location.
(c) Storing the value 386 at this location.
It can be represented with the following figure :



Here, the address 3313 is assumed one, it may be some other address also.

The pointers are one of the most useful and strongest features of C++. There are three useful reason for proper utilization of pointer :

(i) The memory location can be directly accessed and manipulated.
(ii) Dynamic memory allocation is possible.
(iii) Efficiency of some particular routines can be improved.

4.2 CONCEPT OF INHERITANCE

Inheritance is a concept which is the result of commonality between classes. Due to this mechanism, we need not repeat the declaration as well as member functions in a class if they are already present in another class. For example, consider the classes namely "minister" and "prime minister". Whatever information is present in minister, the same will be present in prime minister also. Apart from that there will be some extra information in class prime minister due to the extra privileges enjoyed by him. Now, due to the mechanism of inheritance, it is enough only to indicate that information which is a specific to prime minister in its class. In addition, the class prime minister will inherit the information of class minister.

4.3 BASE CLASS AND DERIVED CLASS

Let us take the classes, Employee and Manager. A Manager is an Employee with some additional information. when we are declaring the classes Employee and Manager without applying the concept of inheritance, they will look as follows:

```
class Employee
{ public: char*
 name; int age;
 char* address;
 int salary;
 char*departmen
 t; int id;
};
```

Now, the class Manager is as follows:

```
Class Manager
{ public: char*
 name; int age;
 char* address;
 int salary;
```

```
 char*departmen
 t; int id;
  employee* team_members; //He heads a group of employees
  int level;   // his position in hierarchy of the organisation
 .
 .
 .
 .
};
```

Now, without repeating the entire information of class Employee in class Manager, we can declare the Manager class as follows:

```
class Manager: Public Employee
{ public:
 Employee*Team_members;
  int level;
 .
 .
 .
 .
};
```

The latest declaration of class Manager is the same as that of its previous one, with the exception that we did not repeat the information of class Employee explicitly. This is what is meant by the Application of inheritance mechanism. Please note that in the above example, Employee is called Base Class and Manager is called Derived Class.

4.4 SINGLE INHERITANCE

In this Section, you will learn the ways of deriving a class from single class. So, there will be only one base class for the derived class.

4.4.1 Private Inheritance

Consider the following classes:

```
class A { /*.....*/);
class C: private A
```

```
{/*
        .
        .
       .
       .
        */
}
```

All the public parts of class A and all the protected parts of class A, become private members/parts of the derived class C in class C. No private member of class A can be accessed by class C. To do so, you need to write public or private functions in the Base class. A public function can be accessed by any object, however, private function can be used only within the class hierarchy that is class A and class C and friends of these classes in the above cases.

6.4.2 Public Inheritance

Consider the following classes:

```
class A{/*.....*/};
class E: public A
{/*
        :
        :
        :
};
```

Now, all the public parts of class A become public in class E and protected part of
A become protected in E

4.6.3 Protected Inheritance

Consider the following classes:

```
 class E: protected A
{ /*
    .
  .
  .
  */
 };
```

Now, all the public and protected parts of class A become protected in class E.

No private member of class A can be accessed by class E. Let us take a single example to demonstrate the inheritance of public and private type in more details. Let
us assume a class close_shape as follows:

```
class closed_shape
{
 public:
 .
 .
 .
}
class circle: public closed_shape
  // circle is derived in public access mode from class
  // closed-shape
{
  float x, y; // Co-ordinates of the centre of the
 circle float radius;
 public:
 .
 .
 .
 .
}

class semi-circle : public circle
{  private:
      .
 .
 .
    public:
      .
      .
      .
      .
}

class rectangle: private closed_shape
```

```
      {
       float x y ;
                 1, 1
             float x ,y ;
                   2 2
       public:
            .
            .
            .
            .
      };
class rounded_rectangle : public rectangle
{
private:
public :
          .
          .
          .
}
```

4.5 MULTIPLE INHERITANCE

A class can have more than one direct base classes.

Consider the following classes:

```
Class A {/* .....*/};
Class B {/* .....*/};

 {   /*

              .
            .
            .
            .

 */
 };
```

This is called Multiple Inheritance. If a class is having only one base class, then it is known as single inheritance. In the case of Class C, other than the operations

specified in it, the union of operations of classes A and B can also be applied.

4.6 Nestesd Classes

A class may be declared as a member of another class. Consider the following:

```
Class M1
{
  int n;
 public:
  int m;
};

class M2
{
  int n;
 public:
  int m;
};

class M3
{ M1 N1;
 public:
   M2  N2;
};
```

Now, N1 and N2 are nested classes of M3. M3 can access only public members of N1
and N2. A nested class is hidden in the lexically enclosing class.

4.7 Dynamic Memory Allocation/ Deallocation Operators

Using New, Delete:-

New Operator

In C++, the pointer support dynamic memory allocation (allocation of memory during runtime). While studying arrays we declared the array size approximately. In this case if the array is less than the amount of data we cannot

increase it at runtime. So, if we wish to allocate memory as and when required **new** operator helps in this context.

The syntax of the new operator is given below :

pinter_variable = **new** data_type;Where the data type is any allowed C++ data type and the pointer_variable is a pointer of the same data type. For example,

```
char * cptr

cptr = new char;
```

The above statements allocate 1 byte and assigns the address to cptr.

The following statement allocates 21 bytes of memory and assigns the starting address to cptr :

```
char * cptr;

cptro = new char [21];
```

We can also allocate and initialize the memory in the following way :

Pointer_variable = **new** data_type (value);

Where value is the value to be stored in the newly allocated memory space and it must also be of the type of specified data_type. For example,

```
Char *cptr = new char ('j');

Int *empno = new int [size]; //size must be specified
```

Delete Operator

It is used to release or deallocate memory. The syntax of **delete** operator is :

delete_pointer_variable;

For example,

delete cptr;

delete [ ] empno; //some versions of C++ may require size

4.8 The This Pointer

We know that while defining a class the space is allocated for member functions only once and separate memory space is allocated for each object, as shown in figure

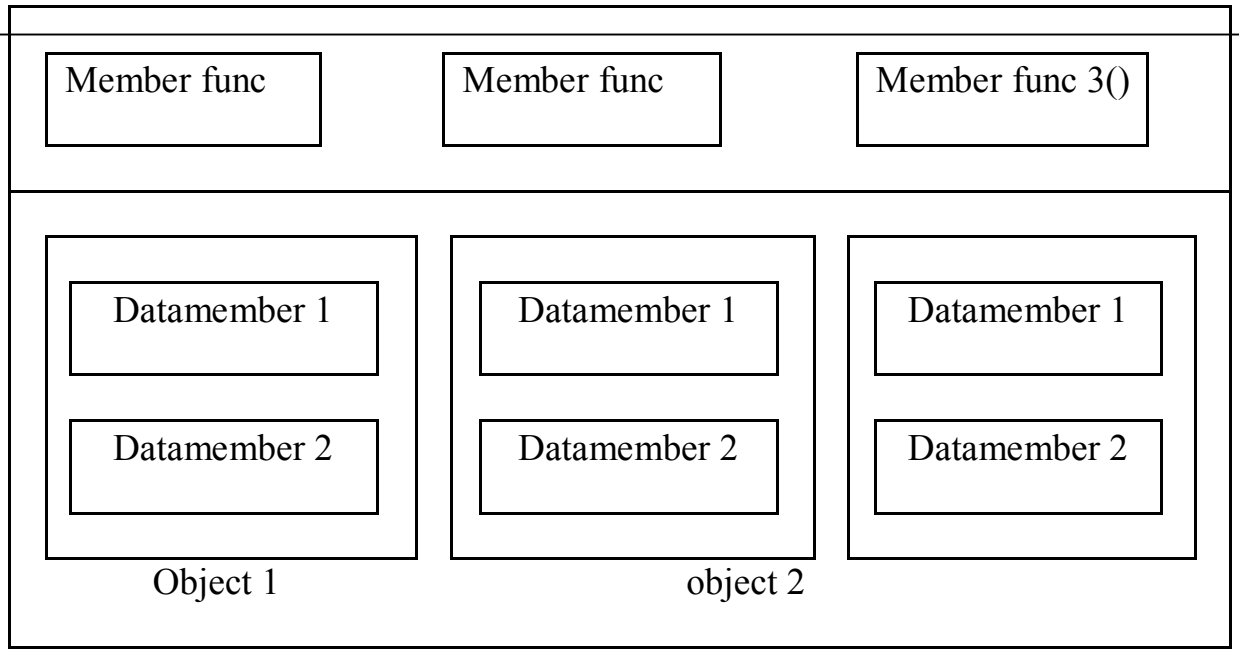


Fig. Allocation of memory for functions and class objects

With the above shown allocation there exists a serious problem that is which object's data member is to be manipulated by any member function. For

example, if **memberfunc2( )** is responsible for modifying the value of datamember1 and we are interested in modifying the value of **datamember1** of **object3**. In the situation like it, how to decide the manipulation of which object's **datamember1**? The **this** pointer is an answer to this problem. The **this** is a pointer that points to that object using which the function is called. The **This** pointer is automatically passed to a member function when it is called. The following program illustrates the above mentioned concept :

```
#include<iostream.h>
#include<string.h>
class per
{
        char name[20];
        float saralry;
        public :
        per (char *s,float a)
        {strcpy(name,s); salaru =a'}
        per GR(per & x)
        { if (x.salary> =salary)
                return &x;
        else
            return this;
        }
        void display()
        {
        cout<<"name : "<<name<<'\n';
```

```
        cout<<"salar :"<<salary<<'\n';
        }
};
        Void main ()
        {
                Per p1("REEMA:, 10000), p2("KRISHANAN",20000),
p3 ("GEORGE", 50000);
```

The output of the Program would be :

Name : REEMA

Salary : 10000

Name : KRISHANAN

Salary : 20000

Here, the first call to the function **GR** returns reference to the object **P1** and the second call returns reference to the object **P2.**

4.9 VIRTUAL FUNCTIONS

Polymorphism is a mechanism that enables same interface functions to work with the whole class hierarchy. Polymorphism mechanism is supported in C++ by the use of virtual functions. The concept of virtual function is related to the concept of dynamic binding. The term Binding refers to binding of actual code to a function call. Dynamic binding also called late binding is a binding mechanism in which the actual function call is bound at run-time and it is dependent on the contents of function pointer at run time. It meant that by altering the content of function pointers, we may be able to call different functions having a same name but different code, that is demonstrating polymorphic behaviour.

Let us look into an example for the above concept:

```
#include <iostream.h>
class employee
{
```

```
public:
 char *name;
  char *department;

  employee (char *n, char *d)
 {
  name = n;
  department = d;
 }
  virtual void print ( );
};

void employee:: print ( )
{
  cout << "name:"<<name;
  cout << "department:" << department;
}

class manager : public employee
{
public:
  short position;
  manager (char *n, char *d, short p) : employee (n, d)
 {
    name = n;
  department = d;
  position = p;
 }
void print ( )
{
  cout << name << "\n" << department << "\n" << position;
}
};
void main ( )
{
  employee* e ("john", "sales");
   manager* m ("james", "marketing", 3);
 e    print   (
 ) m print (
 ) ;
```

```
}
```

The output will be:
John
Sales
James
marketing
3

4.10 Polymorphim

Polymorphism means **'one name multiple forms'.** Runtime polymorphism can

be achieved by using virtual functions. The polymorphism implementation in C++ can be shown as in figure.

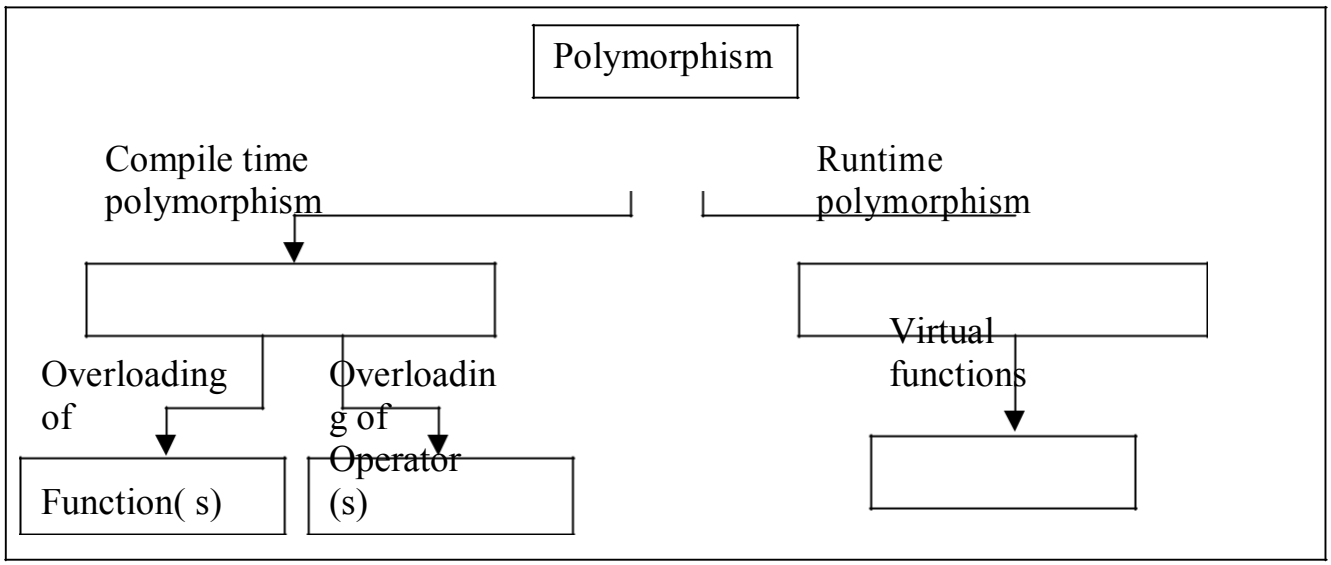


Fig. Implementation of polymorphism.

4.10.1 STATIC POLYMORPHISM OR COMPILE TIME POLYMORPHISM

It means existence of an entity in various physical forms simultaneously. Static polymorphism refers to the binding of functions on the basis of their **signature** (number, type and sequence of parameters). It is also called **early**

binding because the calls are type and sequence of parameters). It is also called **early binding** because the calls are already bound to the proper type of functions during the compilation of the program. For example,

```
Void volume (int);                    //prototype
Void volume (int,int,int);          //prototype
```

When the function volume ( ) is invoked, the passed parameters determine which one to be executed. This resolution takes place at compile time.

4.10.2 DYNAMIC POLYMORPHISM

It means change of form by entity depending on the situation. A function is said to exhibit dynamic polymorphism if it exists in various forms, and the resolution to different function calls are made dyanamically during execution time. This feature makes the program more **flexible** as a function can be called, depending on the context.

4.10.3 STATIC AND DYNAMIC BINDING

As stated earlier the dynamic binding is more flexible, and the static binding is more efficient in certain cases.

Statically bound functions do not require run-time search, while the dynamic function calls need it. But in case of dynamic binding, the function calls are resolved at execution time and the user has the flexibility to alter the call without modifying the source code.

For a programmer, efficiency and performance are more important, but to the user, flexibility and maintainability are of primary concern. So a trade-off between the efficiency and flexibility can be made.

.

4.11 Summary

In this Unit, you have been exposed to the concepts of base class and derived classes. A derived class is a class which includes the member of another class. This concept is also known as inheritance. When a derived class has more than one direct base class, then it is called Multiple Inheritance. There were three types of inheritance. We can also declare classes as members of another class. We have also touched on the concept of polymorphism.

4.12 Keywords

Inheritance:- *Inheritance* is a mechanism of reusing and extending existing classes without modifying them.

Polymorphism:- Polymorphism is a mechanism that enables same interface functions to work with the whole class hierarchy.

6.13 Review Questions

Q.1. Illustrate the concept of inheritance with the help of an example.
Q.2. What is a virtual base class ? When do we make it?
Q.3. Write a program in c++ which demonstrate the use of inheritance.
Q.4. What do you understand by function returning a pointer ? Give any suitable example to support your answer.
Q.5. Differentiate between compile time polymorphism and run time polymorphism.

6.14 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.

Course Code-22316 **Paper Name:OOPS using C++**

Lesson no:5 **Lesson name: Managing console input /output operations**

Unit structure

5.1 Introduction

5.2 C++ streams

5.3 C++ streams classes

5.4 Unformatted I/O Operations

5.5 Formatted console I/O Operations

5.6 Managing output with manipulators

5.7 Design Our Own Manipulators

5.1 Introduction:

C++ supports two complete I/O systems: the one inherited from C and the object-oriented I/O system defined by C++ (hereafter called simply the C++ I/O system). Like C-based I/O, C++'s I/O system is fully integrated. The different aspects of C++'s I/O system, such as console I/O and disk I/O, are actually just different perspectives on the same mechanism.Every program takes some data as input and generates processed data as output following the input -process-output cycle.C++ supports all of C's rich set of I/O functions that can be used in the C++ programs.But these are restrained from using due to two reasons ,first I/O methods in C++ supports the concept of OOP and secondly I/O methods in c can not handle the user defined data types such as class objects.C++ uses the concept of streams and stream classes to implement its I/O operation with the console and disk fils.

5.2 C++ streams:-

A stream is a logical device that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same way even though the actual physical devices they are connected to may differ substantially. Because all streams behave the same, the same I/O functions can operate on virtually any type of physical device. For example, one can use the same function that writes to a file to write to the printer or to the screen. The advantage to this approach is that you need learn only one I/O system.

A stream act like a source or destination.The source stream that provide data to the program is called the input stream and the destination stream that receive output from
the program is called the output stream.C++ containes cin and cout predefined streamsthat opens automatically when a program begins its execution.cin represents the input stream connected to the standard input device and cout represents the output stream connected to standard output device.

5.3 C++ Stream Classes:

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes.Figure 5.1 shows the hierarchy of the stream classes used for input and output operstions with the console unit. These classes are declared in the header file iostrem. The file should be included in all programs that communicate with the console unit.

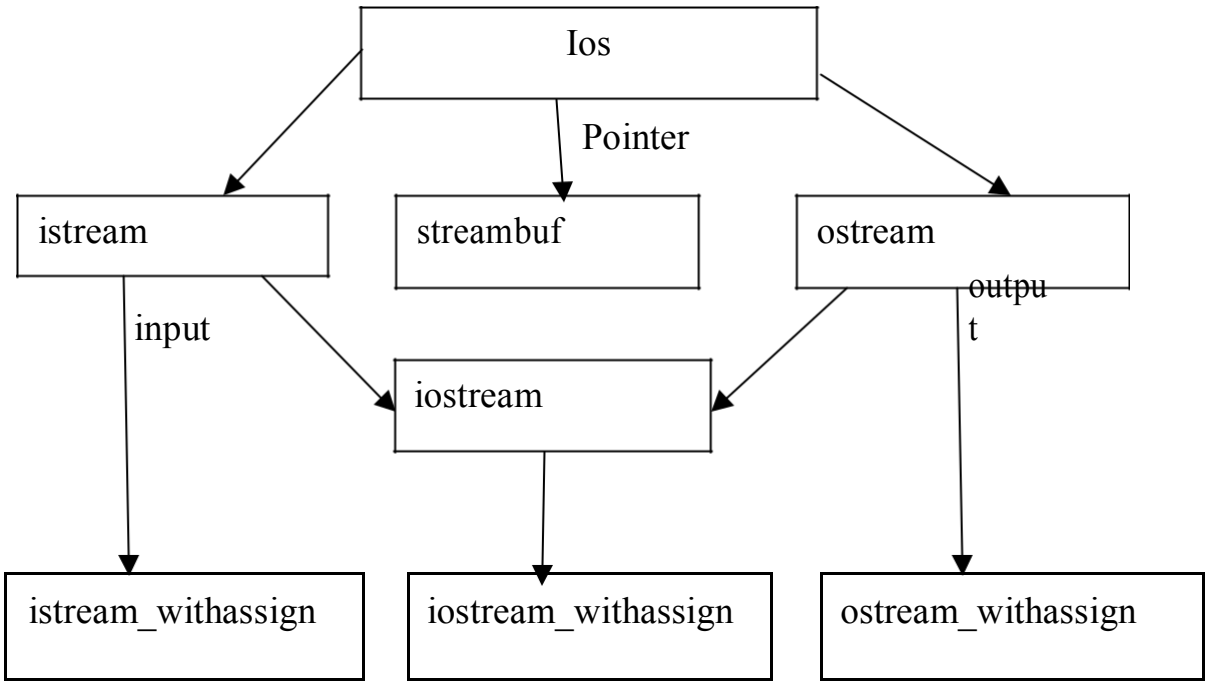


Figure 5.1 Stream classes for console I/O operations

As in figure 5.1 ios is the base class for istream(input stream) and ostream(output stream) which are base classes for iostream(input/output stream).The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.

The class ios provides the basic support for formatted and unformatted input/output operations.The class istream provides the facilities for formatted and unformatted input while the class ostream(through inheritance) provides the facilities for formatted output.The class iostream provides the facilities for handling both input output streams.Three classes namely istream_withassign, ostream_withassign and iostream_withassign add assignment operators to these classes.

Table 5.1 Stream classes for console operations

Class name	Contents
ios(General input/output stream class)	Contains basic facilities that are ued by all other input and output classes Also contains a pointer to buffer object(streambuf object) Declares constants and functions that are necessary for handling formatted input and output operations
istream(input stream)	Inherits the properties of ios Declares input functions such as get(),getline() and read() Contains overloaded extraction operator>>
ostream(output stream)	Inherits the property of ios Declares output functions put() and write() Contains overloaded insertion operator <<
iostream (input/output	Inherits the properties of ios stream and ostream through

stream)	multiple inheritance and thus contains all the input and output Functions
streanbuf	Provides an interface to physical devices through buffer Acts as a base for filebuf class used ios files

5.4 Unformatted input/output Operations:-

5.4.1 Overloaded operators >> and<<

Objects cin and cout are used for input and output of data by using the overloading of >> and << operators.The >> operator is overloaded in the istream class and << is overloaded in the ostream class. The following is the format for reading data from keyboard:

cin>>variable1>>variable2>>……..>>variable n

where variable 1 to variable n are valid C++ variable names that have declared already.This statement will cause the computer to stop the execution and look for the input data from the keyboard.the input data for this statement would be

data1 data2……..data n

The input data are separated by white spaces and should match the type of variable in the cin list spaces, newlines and tabs will be skipped.

The operator >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type. For example consider the code

int code;

cin>> code;

Suppose the following data is entered as input

42580

the operator will read the characters upto 8 and the value 4258 is assigned to code.The character D remains in the input streams and will be input to the next cin statement.The general form of displaying data on the screen is

cout <<item1<<item2<<……<<item n

The item item1 through item n may be variables or constants of any basic type.

5.4.2 put() and get() functions:-

The classes istream and ostream define two member functions get(),put() respectively to handle the single character input/output operations. There are two types of get() functions.Both get(char *) and get(void) prototype can be used to fetch a character including the blank space,tab and newline character. The get(char *) version assigns the input character to its argument and the get(void) version returns the input character.

Since these functions are members of input/output Stream classes,these must be invoked using appropriate objects.

Example

```
Char c;
cin.get( c )  //get a character from the keyboard and assigns it to c
while( c!='\n')
{   cout<< c; //display the character on screen
cin.get( c )          //get another character
}
```

this code reads and display a line of text. The operator >> can be used to read a character but it will skip the white spaces and newline character.The above while loop will not work properly if the statement

cin >> c;

is used in place of cin.get (c);

The get(void) version is used as follows:

……..

char c;

c= cin.get();

……

The value returned by the function get() is assigned to the variable c.

The function put(), a member of ostream class can be used to output a line of text, character by character. For example

cout.put('x');

displays the character x and

cout.put(ch);

displays the value of variable ch.

The variable ch must contain a character value.A number can be used as an argument to function put().For example,

cout.put(68);

displays the character D.This statement will convert the numeric value 68 to a char value and displays character whose ASCII value is 68.

The following segment of a program reads a line of text from keyboard and displays it on the screen

```
char c;
cin.get ( c );
while( c!= '\n')
{   cout.put(c);
      cin.get ( c);
}
```

The program 5.1 illustrate the use of two character handling functions.

Program 5.1

```
Character I/O with get() and put()
#include <iostream>
using namespace std;
int main()
{
  int count=0;
char c;
cout<<"INPUT TEXT \n";
cin.get( c );
while ( c 1='\n' )
{ cout.put(
c);
count++;
cin.get( c );
}
 cout<< "\n Number of characters =" <<count
<<"\n"; return 0;
}
```

Input

Object oriented programming

Output

Object oriented programming

Number of characters=27

5.4.3 getline() and write() functions:-

A line of text can be read or display effectively using the line oriented input/output functions getline() and write() .The getline() function reads a whole line of text that ends with a newline character.This function can be invokedby using the object cin as follows:

cin.getline(line,size);

This function call invokes the function getline() which reads character input into the variable line.The reading is terminated as soon as either the newline character '\n' is encountered or size-1 characters are read(whichever occurs first) .The newline character is read but not saved.instead it is replaced by the null character.For example consider the following code:

char name[20];

cin.getline(name,20);

Assume that we have given the following input through key board:

Bjarne Stroustrup<press Return>

This input will be read correctly and assigned to the character array name.Let us suppose the input is as follows:

Object Oriented Programming<press Return>

In this case ,the input will be terminated after reading the following 19 characters

Object Oriented Pro

Remember ,the two blank spaces contained in the string are also taken into account.Strings cen be read using the operator >> as follows

cin>>name;

But remember cin can read strings that do not contain white spaces.This means that cin can read just one word and not a series of words such as "Bjarne Stroustrup".But it can read the following string correctly:

Bjarne_Stroustrup

After reading the string ,cin automatically adds the terminating null character to the character array.

The program 5.2 demonstrates the use of >> and getline() for reading the strings.

Program 5.2

Reading Strings With getline()

```
#include <iostream>
```

```
using namespace std;
int main()
{ int size=20; char
city[20]; cout<<"enter
city name:\n ";
cin>>city;

cout<<"city
name:"<<city<<"\n\n";
cout<<"enter city name again:
\n"; cin.getline(city,size);
cout<<"city name now:"<<city<<"\n\n";

cout<<"enter another city name: \n";
cin.getline(city,size);
cout <<"New city name:"<<city<<"\n\n';
return 0;
}
```

output would be:

first run

enter city name:

Delhi

Enter city name again:

City name now:

Enter another city name:

Chennai

New city name: Chennai

During fist run the newline character '\n\ at the end of "Delhi" which is waiting in the input queue is read by the getline() that follows immediately and therefore it dos not wait for any response to the prompt 'enter city name again'.The character'\n' is read as an empty line.

The write() function displays an entire line and has the following form:

cout.write(line,size)

The first argument line represents the name of the string to be displayed and the second argument size indicates the number of characters automatically when the null character is encountered.If the size is greater than the length of line, then it displays beyond the bound of line.Program5.3 illustrates how write() method displays a string

Program 5.3

Displaying String With write()

```
#include <iostream>
#include<string>
using namespace std;
int main(0
{
 char * string1="C++";
char * string2 ="Programming";
int m=strlen(string1);
int n =strlen(string2);
for (int i=1;i<n;i++)
```

```
{
 cout.write(string2,i);
cout<<"\n";
}
 for (i<n;i>0;i--)
{
 cout.write(string2,i);
cout<<"\n";}
//concatenating strings
cout.write(string1,m).write(string2,n);
cout<<"\n";
//crossing the boundary
cout.write(string1,10);
return 0;
}
```

output

P

Pr

Pro

Prog

Progr

Progra

Program

Programm

Programmi

Programmin

Programming

Programmin

Programmi

Programm

Program

Progra

Progr

Prog

Pro

Pr

P

C++ Programming

C++ Progr

The last line of the output indicates that the statement

cout.write(string1,10);

displays more character than what is contained in string1.

It is possible to concatenate two strings using the write() function.The statement

cout.write(string1,m).write(string2,n);

is equivalent to the following two statements:

cout.write(string1,m);

cout.write(string2,n);

5.5 Formatted Console I/O Operations:-

C++ supports a number of features that could be used for formatting the output.These features include:

--ios class function and flags.

--manipulators.

--User-defined output functions.

The ios class contains a large number of member functions that would help us to format the output in a number of ways.The most importany ones among them are listed in table 5.2

Table 5.2 ios format functions

Function	**Task**
Width()	To specify the required field size for displaying an output value
Precision()	To specify the number of digits to be displayed after the decimal point of float value
Fill()	To specify a character that is used to fill the unused portion of a field
	a field
Setf()	To specify format flags that can control the form of output display(such as left-justification and right-justification)
Unsetf()	To clear the flags specified

Manipulators are special functions that can be included in the I/O statements to alter the format parameter of stream .Table 5.3 shows some important manipulator functions that are frequently used. To access these manipulators, the file iomanip should be included in the program.

.

Table 5.3 Manipulators

Manipuators	**Equivalent ios function**
setw()	width()
setprecision()	precision()
setfill()	fill()
setiosflags()	setf()
resetiosflags()	unsetf()

In addition to these standard library manipulators we can create our own manipulator functions to provide any special output formats.

5.5.1 Defining Field Width:width()

The width() function is used to define the width of a field necessary for the output of an item.As it is a member function object is required to invoke it like

cout.width(w);

here w is the field width.The output will be printed in a field of w character wide at the right end of field.The width() function can specify the field width for only one item(the item that follows immediately).After printing one item(as per the specification) it will revert back the default.for example,the statements

cout.width(5);

cout<<543<<12<<"\n";

will produce the following output:

		5	4	3	1	2

The value 543 is printed right justified in the first five columns.The specification width(5) does not retain the setting for printing the number 12.this can be improved as follows:

```
cout.width(5);
cout<<543;
cout.width(5);
cout<<12<<"\n";
```

This produces the following output:

		5	4	3				1	2

The field width should be specified for each item.C++ never truncate the values and therefore,if the specified field width is smaller than the size of the value to be printed,C++ expands the field to fit the value.program 5.4 demonstrates how the function width() works.

Program 5.4

Specifying field size with width()

```
#include <iostream>
using namespace std;
int main()
{
 int item[4] ={ 10,8,12,15};
int cost[4]={75,100,60,99};
cout.width(5);
cout<<"Items";
cout.width(8);cout<<"Cost";
cout.width(15);
```

```
cout<<"Total Value"<<"\n";
int sum=0;
for(int i=0;i<4 ;i++)
{
  cout.width(5);
cout<<items[i];
cout.width(8);
cout<<cost[i];
int value = items[i] * cost[i];
cout.width(15);
cout<<value<<"\n";
sum= sum + value;

}
cout<<"\n Grand total = ";
cout.width(2);
cout<<sum<<"\n";
return 0;
}
```

The output of program 5.4 would be

ITEMS	COST	TOTAL VALUE
10	75	750
8	100	800
12	60	720
15	99	1485

Grand total

=3755

5.5.2 Setting Precision: precision():-

By default ,the floating numbers are printed with six digits after the decimal points. However ,we can specify the number of digits to be displayed after the decimal point while printing the floating point numbers.

This can be done by using the precision () member function as follows:

cout.precision(d);

where d is the number of digits to the right of decimal point.for example the statements

cout.precision(3);

cout<<sqrt(2)<<"\n";

cout<<3.14159<<"\n";

cout<<2.50032<<"\n";

will produce the following output:

1.141 (truncated)
3.142 (rounded to nearest cent)
2.5 (no trailing zeros)

Unlike the function width(),precision() retains the setting in effect until it is reset.That is why we have declared only one statement for precision setting which is used by all the three outputs.We can set different valus to different precision as follows:

cout.precision(3);

cout<<sqrt(2)<<"\n";

cout.precision(5);

cout<<3.14159<<"\n";

We can also combine the field specification with the precision setting.example:

cout.precision(2);

cout.width(5);

cout<<1.2345;

The first two statement instruct :"print two digits after the decimal point in a field of five character width".Thus the output will be:

	1		2	3

Program 5.5 shows how the function width() and precision() are jointly used to control the output format.

Program 5.5

PRECISION SETTING WITH precision()

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
cout<<"precision set to 3 digits\n\n";
cout.precision(3);
cout.width(10);
cout<<"value";
cout.width(15);
cout<<"sqrt_of _value"<<"\n";
for (int n=1;n<=5;n++)
{
   cout.width(8);
cout<<n;
cout.width(13);
cout<<sqrt(n)<<"\n";
}
cout<<"\n precision set to 5 digits\n\n";
cout.precision(5);
cout<<"sqrt(10) = " <<sqrt(10)<<"\n\n";
cout.precision(0);
cout<<"sqrt(10) = " <<sqrt(10)<<"(default setting)\n";
```

return 0;

}

The output is

Precision set to 3 digits

VALUE	SQRT OF VALUE
1	1
2	1.41
3	1.73
4	2
5	2.24

Precision set to 5 digits

Sqrt(10)=3.1623
Sqrt(10)=3.1622
78 (Default setting)

5.5.3 FILLING AND PADDING :fill()

The unused portion of field width are filled with white spaces, by default. The fill() function can be used to fill the unused positions by any desired character.It is used in the following form:

cout.fill(ch);

Where ch represents the character which is used for filling the unused positions.Example:

cout.fill('*');

cout.width(10);

cout<<5250<<"\n";

The output would be:

*	*	*	*	*	*	5	2	5	0

Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily.Like precision (),fill()

Stays in effect till we change it.As shown in following program

Program 5.6

```
#include<iostream>
using namespace std;
int main()
{ cout.fill('<');
cout.precision(3);
for(int
n=1;n<=6;n++)
{
 cout.width(5);
cout<<n;
cout.width(10);
cout<<1.0/float(n)<<"\n";
if(n==3)
cout.fill('>');
```

```
}
cout<<"\nPadding changed
\n\n"; cout.fill('#'); //fill()
reset cout.width(15);
cout<<12.345678<<"\n";
return 0;
}
```

The output will be
<<<<1<<<<<<<<<<1

<<<<2<<<<<<<<<0.5

<<<<3<<<<<<0.333

>>>>4>>>>>>0.25

>>>>5>>>>>>>>0.2

PADDING CHANGED

#########12.346

5.5.4 FORMATTING FLAGS,Bit Fields and setf():-

The setf() a member function of the ios class, can provide answers left justified.The setf()
function can be used as follows:

cout.setf(arg1.arg2)

The arg1 is one of the formatting flags defined in the class ios.The formatting flag specifies the format action required for the output.Another ios constant,arg2,known as bit field specifies the group to which the formatting flag belongs. for example:

cout.setf(ios::left,ios::adjustfield);

cout.setf(ios::scientific,ios::floatfield);

Note that the first argument should be one of the group member of second argument.

Consider the following segment of code:

```
cout.fill('*');

cout.setf(ios::left,ios::adjustfield);

cout.width(15);

cout<<"table1"<<"\n";
```

This will produce the following output:

T	A	B	L	E		1	*	*	*	*	*	*	*	*

The statements

```
cout.fill('*');

cout.precision(3);
cout.setf(ios::internal,ios::adjustfield);

cout.setf(ios::scientific,ios::floatfield);

cout.width(15);

cout<<-12.34567<<"\n";
```

Will produce the following output:

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1

5.6 Managing Output with Manipulators:-

The header file iomanip provides a set of functions called manipulators which can be used to manipulate the output format. They provide the same features as that of the ios member functions and flags.For example,two or more manipulators can be used as a chain in one statement as follows

```
cout<<manip1<<manip2<<manip3<<item;
```

cout<<manip1<<item1<<manip2<<item2;

This kind of concatenation is useful when we want to display several columns of output. The most commonly used manipulators are shown below

In the table 5.4

Manipulator	**Meaning**	**Equivalent**
setw(int w)	Set the field width to w	width()
setprecision(int d)	Set the floating point precision to d	precision()
setfill(int c)	Set the fill character to c	fill()
setiosflags(long f)	Set the format flag f	setf()
resetiosflags(long f)	Clear the flag specified by f	unsetf()
endl	Insert new line and flush stream	"\n"

Examples of manipulators are given below:

cout<<setw(10)<<12345;

This statement prints the value 12345 right-justified in a field of 10 characters. The output can be made left-justified by modifying the statement

follows:

cout<<setw(10)<<setiosflags(ios::left)<<12345;

One statement can be used to format output for two or more values.For example, the statement

cout<<setw(5)<<setprecision(2)<<1.2345

<<setw(10)<<setprecision(4)<<sqrt(2)<<setw(15)<<setiosflags(ios::scientific)<<sqrt(3);

<<endl;

will print all the three values in one line with the field sizes of 5,10,15 respectively.

The following program illustrates the formatting of the output values using both manipulators and ios functions.

Program 5.7

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
  cout.setf(ios::showpoint);
cout<<setw(5)<<"n"<<setw(15)<<"inverse of n"<<setw(15)<<"sum of terms";
double term,sum=0;
for (int n=1;n<=10;n++)
 {
```

```
    term=1.0/float(n);
sum=sum + term;
cout<<setw(5)<<n<<setw(14)<<setprecision(4)
<<setiosflags(ios::scientific)<<term
<<setw(13)<<resetioflags(ios::scientific) <<sum<<endl;
```

```
}
 return 0;
}
```

5.7 Designing our own manipulators:-

The general form for creating a manipulator without any argument is ostream & manipulator (ostream & output)

{

…….

…….(code)

……..

return output;

}

The following program illustrate the creation and use of user defined manipulators.

Program 5.8

```
#include <iostream>
#include <iomanip>
using namespace std;
ostream &currency (ostream & output)
{
    output<< "Rs";
return output;
}
ostream & form (ostream &output)
{
```

```
  output.set(ios::showpos);
  output.setf(ios::showpoint);
output.fill('*');
output.precision(2);
output<<setiosflags(ios::fixed)
<<setw(10);
return output;
}
int main()
{
  cout<<currency<<form<<78
  64.5;
return 0;
}
```

the output of program is

Rs**+7864.50

In the program form represents a complex set of format functions and manipulators.

Summary

5.1.A stream is a sequence of bytes and serves as a source or destination for an I/O data.

5.2.The source stream that provides data to the program is called as input stream and the destination stream that receives output from the program is called the output stream.

5.3.The C++ I/O system contains a hierarchy of stream classes used for input and output operations.These classes are declared in the header file 'iostream'.

5.4.cin represents the input stream connected to standard input device and cout represents the output stream connected to standard output device.

5.5.The >> operator is overloaded in the istream class as an extraction operator and the << operator is overloaded in the ostream class as an insertion operator.

5.6.We can read and write a line of text more efficiently using the line oriented I/O functions getline() and write() respectively.

5.7.The header file iomanip provides a set of manipulator functions to manipulate output formats.

Key Terms

adjustfield

console I/O operations

output stream

precision()

fill()	put()
flags	setfill()
get()	setiosflags()
getline()	setw()
ios	write()
iostream	
ostream	

Exercises

5.1.What is a stream?

5.2.Describe briefly the features of I/O system supported by C++.

5.3.How is cout able to display various types of data without any special instructions?

5.4.Why it is necessary to include the file iostream in all our programs?

5.5.What is the role of iomanip file?

5.6. What is the basic difference between manipulators and ios member functions in implementation?Give examples.

Corurse Code-22316 Paper Name:OOPS using C++
Lesson no:6 Lesson name: Working with C++ files

Unit Structure

6.1Introduction

6.2File stream classes

6.3 Steps of file operations

6.4Finding end of file

6.5File opening modes

6.6File pointers and manipulators

6.7Sequential input and output operations

6.8 Error handling functions

6.9 Command Line argument

6.1 Introduction :-

When a large amount of data is to be handled in such situations floppy disk or hard disk are needed to store the data .The data is stored in these devices using the concept of files. A file is a collection of related data stored in a particular area on a disk. Programs can be designed to perform the read and write operations on these files .The I/O system of C++ handles file operations which are very much similar to the console input and output operations .It uses file streams as an interface between the programs and files. The
stream that supplies data to the program is called input stream and the one that receives data from the program is called output stream. In other words input stream extracts data
from the file and output stream inserts data to the file. The input operation involves the creation of an input stream and linking it with the program and input file. Similarly, the output operation involves establishing an output stream with the necessary links with the program and output file.

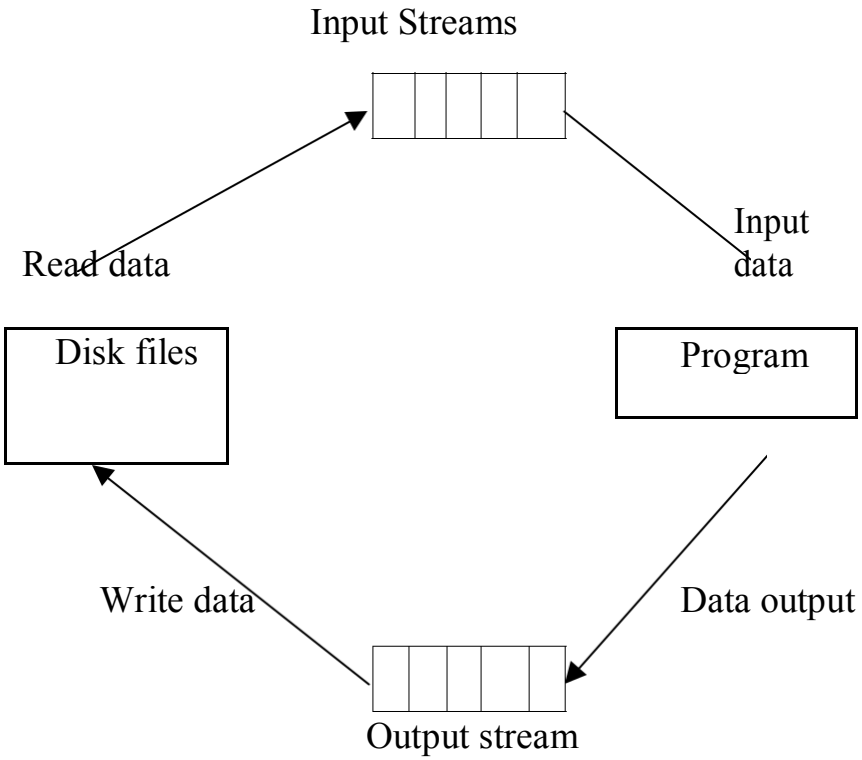


Figure 6.1 File Input and output stream

6.2 File Stream Classes:-

The I/O system of C++ contains a set of classes that defines the file handling methods. These include ifstream, ofstream and fstream.These classes are derived from fstreambase and form the corresponding iostream class. These classes ,designed to manage the disk files are declared in fstream and therefore this file is included in any program that uses files.

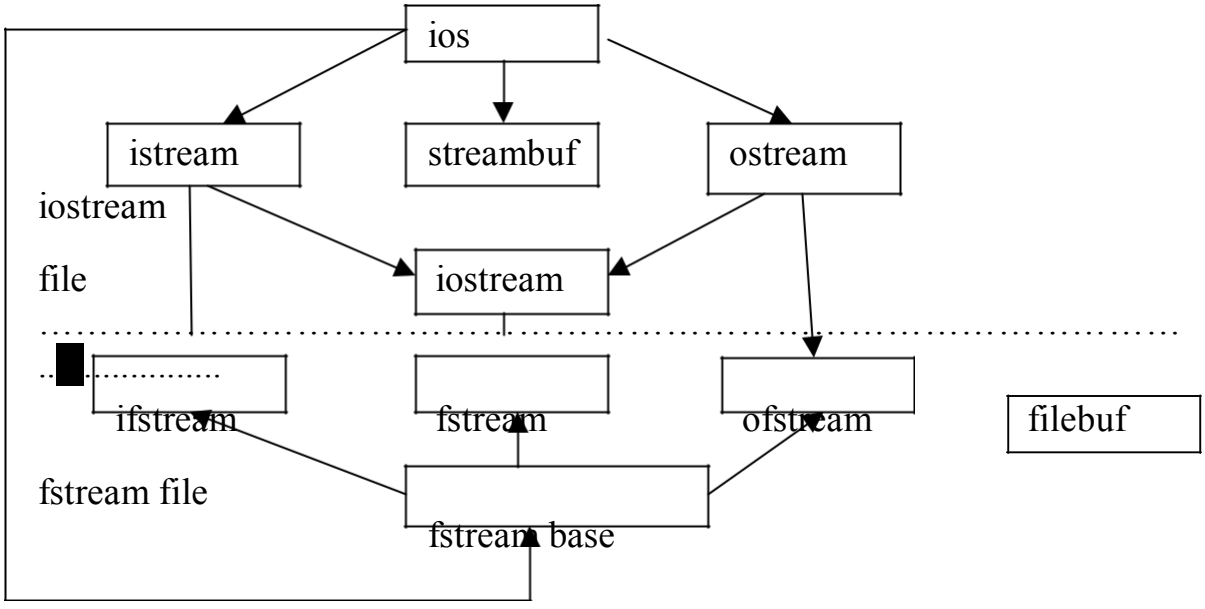


Figure 6.2 Stream classes for file operations

6.3 Steps of File Operations:-

For using a disk file the following things are necessary

1. Suitable name of file
2. Data type and structure
3. Purpose
4. Opening Method

Table 6.1 Detail of file stream classes

Class	**Contents**
filebuf	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes.Also contain **close()** and **open()** as members.
fstreambase	Provides operations common to file streams.Serves as a base for **fstream,ifstream** and **ofstream** class.Contains **open()** and **close()** functions.
ifstream	Provides input operations.Contains **open()** with default input mode.Inherits the functions **get(),getline(),read(),seekg(),tellg()** functions from istream.
ofstream	Provides output operations.Contains **open()** with default output mode.Inherits **put(),seekp(),tellp()** and **write()** functions from **ostream**.
fstream	Provides support for simultaneous input and output operations.Contains open with default input mode.Inherits all the functions from **istream** and **ostream** classes through **iostream**.

The filename is a string of characters that makeup a valid filename for the operating system. It may contain two parts ,primary name and optional period with extension.
Examples are Input.data, Test.doc etc. For opening a file firstly a file stream is created and then it is linked to the filename.A file stream can be defined using the classes ifstream**, ofstream** and fstream that contained in the header file **fstream**.The class to be used depends upon the purpose whether the write data or read data operation is to be performed on the file.A file can be opened in two ways:

(a) Using the constructor function of class.

(b) Using the member function **open()** of the class.
The first method is useful only when one file is used in the stream.The second method is used when multiple files are to be managed using one stream.

6.3.1Opening Files using Constructor:

While using constructor for opening files,filename is used to initialize the file stream object.This involves the following steps

(i) Create a file stream object to manage the stream using the appropriate class

i.e the class ofstream is used to create the output stream **ifstream** to and the class create the input stream.

(ii) Initialize the file object using desired file name.

For example, the following statement opens a file named "results" for output:

ofstream outfile("results"); //output only

This create outfile as an **ofstream** object that manages the output stream. Similarly ,the following statement declares infile as an **ifstream** object and attaches it to the file data for reading (input).

ifstream infile ("data"); //input only

The same file name can be used for both reading and writing data.For example

Program1

……..

…….

```
ofstream outfile          //creates outfile and connects salary to it
("salary");

……….

……..
```

Program 2

```
……….

……
```

```
ifstream infile
("salary");
//creates infile and
connects salary to it
```

……..

…….

The connection with a file is closed automatically when the stream object expires i.e when a program terminates.In the above statement ,when the program 1 is terminated,the salary file is disconnected from the outfile stream.The same thing happens when program 2 terminates.

Instead of using two programs,one for writing data and another for reading data ,a single program can be used to do both operations on a file.

……

…….

outfile.close(); //disconnect salary from outfile and

connect to infile ifstream infile ("salary");

…….

…….

infile.close();

The following program uses a single file for both reading and writing the data .First it take data from the keyboard and writes it to file.After the writing is completed the file is closed.The program again opens the same file read the information already written to it and displays the same on the screen.

PROGRAM 6.1

WORKING WITH SINGLE FILE

```
//Creating files with constructor function
#include <iostream.h>
#include <fstream.h>
int main()
{
```

```
 ofstream outf("ITEM");
 cout <<"enter item name: ";
char name[30];
cin >>name;
```

```
  outf <<name <<"\n";
  cout <<"enter item cost :";
 float cost;
 cin >>cost;
outf <<cost <<"\n";
outf.close();
ifstream inf("item");
inf >>name;
inf >>cost;
cout <<"\n";
cout <<"item name : " << name <<"\n";
cout <<"item cost: " << cost <<"\n";
inf.close();
return 0;
}
```

6.3.2 Opening Files using open()

The function **open()** can be used to open multiple files that uses the same stream object. For example to process a set of files sequentially,in such case a single stream object can be created and can be used to open each file in turn. This can be done as follows;

File-stream-class stream-object;

stream-object.**open** ("filename");

The following example shows how to work simultaneously with multiple files

PROGRAM 6.2

```
WORKING WITH MULTIPLE FILES
//Creating files with open() function
```

```
#include <iostream.h>
#include<fstream.h>
int main()
```

```
{
  ofstream fout;
 fout.open("country");
fout<<"United states of America \n";
fout<<"United Kingdom";
fout<<"South korea";
fout.close();
fout.open("capital");
fout<<"Washington\n";
fout<<"London\n";
fout<<"Seoul \n";
fout.close();
const int N =80;
char line[N];
ifstream fin;
fin.open("country");
cout<<"contents of country file \n";
while (fin)
{
fin.getline(line,N);
cout<<line;
}
fin.close();
fin.open("capital");
cout<<"contents of capital file";
```

```
while(fin)
{
```

```
fin.getline(line,N);
cout<<line;
}
fin.close();
return 0;
}
```

6.4 Finding End of File:

While reading a data from a file,it is necessary to find where the file ends i.e end of file.The programmer cannot predict the end of file,if the program does not detect end of file,the program drops in an infinite loop.To avoid this,it is necessary to provide correct instruction to the program that detects the end of file.Thus when end of file of file is detected,the process of reading data can be easily terminated. An **ifstream** object such as **fin** returns a value of 0 if any error occurs in the file operation including the end-of –file condition.Thus the while loop terminates when f**in** returns a value of zero on reaching the end-of –file condition.There is an another approach to detect the end of file condition.The statement

```
if(fin1.eof() !=0 )
 {
    exit(1);
 }
```

returns a non zero value if end of file condition is encountered and zero otherwise.Therefore the above statement terminates the program on reaching the end of file.

6.5 File Opening Modes:

The **ifstream** and **ofstream** constructors and the function **open()** are used to open the files.Upto now a single arguments a single argument is used that is filename.However,these functions can take two arguments, the second

one for specifying the file mode.The general form of function open() with two arguments is:

stream-object.open("filename",mode);

The second argument mode specifies the purpose for which the file is opened.The prototype of these class member functions contain default values for second argument and therefore they use the default values in the absence of actual values.The default values are as follows :
ios::in for **ifstream** functions meaning open for reading only.

ios::out for **ofstream** functions meaning open for writing only.

The file mode parameter can take one of such constants defined in class **ios**.The following table lists the file mode parameter and their meanings.

Table 6.2 File Mode Operation

Parameter	**Meaning**
ios::app	Append to end-of-file
ios::ate	Go to end-of-file on opening
ios::binary	Binary file
ios::in	Open file for reading only
ios::nocreate	Open fails if file the file does not exist
ios::noreplace	Open fails if the file already exists
ios::out	Open file for writing only
ios::trunk	Delete the contents of the file if it exists

6.6 File Pointers and Manipulators:

Each file has two pointers known as file pointers,one is called the input pointer and the other is called output pointer..The input pointer is used for reading the contents of of a given file location and the output pointer is used for writing to a given file location.Each time an input or output operation takes place,the appropriate pointer is automatically advanced.

6.6.1 Default actions:

When a file is opened in read-only mode,the input pointer is automatically set at the beginning so that file can be read from the start.Similarly when a file is opened in write-only mode the existing contents are deleted and the output pointer is set at the beginning.This enables us to write the file from start.In case an existing file is to be opened in order to add more data,the file is opened in 'append' mode.This moves the pointer to the end of file.

6.6.2 Functions for Manipulations of File pointer:

All the actions on the file pointers takes place by default.For controlling the movement of file pointers file stream classes support the following functions

- **seekg()** Moves get pointer (input)to a specified location.
- **seekp()** Moves put pointer (output) to a specified location.
- **tellg()** Give the current position of the get pointer.
- **tellp()** Give the current position of the put pointer.

For example, the statement

```
infile.seekg(10);
```

moves the pointer to the byte number 10.The bytes in a file are numbered beginning from zero.Therefore ,the pointer to the 11^{th} byte in the file.Consider the following statements:

```
ofstream fileout;

fileout.open("hello",ios::app);

int p=fileout.tellp();
```

On execution of these statements,the output pointer is moved to the end of file "hello"

And the value of p will represent the number of bytes in the file.

6.6.3 Specifying the Offset:

'Seek' functions **seekg()** and **seekp()** can also be used with two arguments as follows:

seekg (offset,refposition);

seekp (offset,refposition);

The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition.The refposition takes one of the following three constants defined in the ios class:

- **ios::beg** Start of file
- **ios::cur**Current position of the pointer
- **ios::end** End of file

The **seekg()** function moves the associated file's 'get' pointer while the **seekp()** function moves the associated file's 'put 'pointer.The following table shows some sample pointer offset calls and their actions.**fout** is an **ofstream** object.

Table 6.3 Pointer offset calls

Seek call	Action
fout.seekg(o,ios::beg)	Go to start
fout.seekg(o,ios::cur)	Stay at the current position
fout.seekg(o,ios::end)	Go to the end of file
fout.seekg(m,ios::beg)	Move to (m+1)th byte in the file
fout.seekg(m,ios::cur)	Go forward by m byte from current position
fout.seekg(-m,ios::cur)	Go backward by m bytes from current position.
fout.seekg(-m,ios::end)	Go backward by m bytes from the end

6.7 Sequential Input and Output Operations:

The file stream classes support a number of member functions for performing the input and output operations on files.One pairs of functions,**put()** and **get()** are designed for handling a single character at a time.Another pair of functions, **write(),read()** are designed to write and read blocks of binary data.

6.7.1 put() and get() Functions:

The function **put()** writes a single character to the associated stream.Similarly,the function **get()** reads a single character from the associated stream.The following program illustrates how the functions work on a file.The program requests for a string.On receiving the string,the program writes it,character by character,to the file using the **put()** function in a for loop.The length of string is used to terminate the for loop.

The program then displays the contents of file on the screen.It uses the function **get()** to fetch a character from the file and continues to do so until

the end –of –file condition is reached.The character read from the files is displayed on screen using the operator <<.

PROGRAM 6.3

I/O OPERATIONS ON CHARACTERS

```
#include <iostream.h>
#include <fstream.h>
#onclude<string.h>
int main()
```

```
{
  char string[80];
  cout<<"enter a string \n";
   cin>>string;
  int len =strlen(string);
  fstream file;
   file.open("TEXT". Ios::in | ios::out);
   for (int i=o;i<len;i++)
   file.put(string[i]);
    file .seekg(0);
    char ch;
     while(file)
     {
         file.get(ch);
          cout<<ch;
     }
      return 0;
 }
```

6.7.2 write() and read () functions:

The functions **write()** and **read(),**unlike the functions **put()** and **get()** ,handle the data in binary form.This means that the values are stored in the disk file in same format in which they are stored in the internal memory.An int character takes two bytes to store its value in the binary form,irrespective of its size.But a 4 digit int will take four bytes to store it in the character form.The binary input and output functions takes the following form:

infile.read ((char *) & V,sizeof (V));

outfile.write ((char *) & V ,sizeof (V));

These functions take two arguments.The first is the address of the variable V, and the second is the length of that variable in bytes.The address of the variable must be cast to type char*(i.e pointer to character type).The following program illustrates how these two functions are used to save an array of floats numbers and then recover them for display on the screen.

PROGRAM 6.4

```
// I/O OPERATIONS ON BINARY FILES
 #include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
const char * filename =”Binary”;
int main()
{
 float height[4] ={ 175.5,153.0,167.25,160.70};
ofstream outfile;
outfile.open(filename);
outfile.write((char *) & height,sizeof(height));
outfile.close();
 for (int i=0;i<4;i++)
height[i]=0;
ifstream infile;
infile.open(filename);
infile.read ((char *) & height,sizeof (height));
for (i=0;i<4;i++)
{
   cout.setf(ios::showpoint);
cout<<setw(10)<<setprecision(2)<<height[i];
```

```
}
infile.close();
return 0;
}
```

6.8 Error Handling during File Operations:

There are many problems encounterd while dealing with files like

- a file which we are attempting to open for reading does not exist.
- The file name used for a new file may already exist.
- We are attempting an invalid operation such as reading past the end of file.
- There may not be any space in the disk for storing more data.
- We may use invalid file name.
- We may attempt to perform an operation when the file is not opened for that purpose.The C++ file stream inherits a 'stream-state 'member from the class ios.This member records information on the status of a file that is being currently used.The stream state member uses bit fields to store the status of error conditions stated above.The class ios support several member functions that can be used to read the status recorded in a file stream.

Table 6.4 Error Handling Functions

Function	**Return value and meaning**
eof()	Returns true(non zero value) if end of file is encountered while reading otherwise returns false(zero).
fail()	Returns true when an input or output operation has failed .
bad()	Returns true if an invalid operation is attempted or any unrecoverable error has occurred.However,if it is false,it may be possible to recover from any other error reported and continues operation.
good()	Returns true if no error has occurred.This means all the above functions are false.For instance,if file.good() is true.all is well

	with the stream file and we can proceed to perform I/O operations.When it returns false,no further operations is carried out.

These functions can be used at the appropriate places in a program to locate the status of a file stream and thereby to take the necessary corrective measures.Example:

```
…….

……..

ifstream infile;
infile.open(“ABC”);

while(!infile.fail())

{

……

…….. (process the file)

…….

}

if (infile.eof())

{

…….(terminate the program normally)

}

else

  if (infile.bad())

{

    …….(report fatal error)

}

else
```

```
{
 infile.clear(); //clear error state
…….
…….
}
……..
……..
```

The function **clear()** resets the error state so that further operations can be attempted.

6.9 Command Line Arguments:-

Like C,C++ also support the feature of command line argument i.e passing the arguments at the time of invoking the program.They are typically used to pass the names of data files.Example:

C>exam data results

Here exam is the name of file containing the program to be executed and data and results are the filenames passed to program as command line arguments.The command line arguments are typed by the user and are delimited by a space.The first argument is alwayas the filename and contains the program to be executed.The **main()** functions which have been using upto now without any argument can take two arguments as shown below:

```
main(int argc,char * argv[])
```

The first argument **argc** repesents the number of argumnts in commandline.The second argument **argv** is an array of character type pointers that points to the the command line arguments.The size of this array will be equal to the value of argc.For instance,for command line

C>exam data results

The value of **argc** would be 3 and the **argv** would be an array of three pointers to string as shown:

argv[0] exam

argv[1] data

argv[2] results

The argv[0] alwayas represents the command name that invokes the program.The character pointer argv[1], and argv[2] can be used as file names in the file opening statements as shown:

……

……

inline.open(argv[1]); //open data file for reading

……..

……..

outfile.open(argv[2]); //open result file for writing

……..

……..

Summary

1.Stream is nothing but flow of data .In object oriented programming the streams are controlled using classes.

2.The istream and ostream classes control input and output functions respectively.

3.The iostream class is also a derived class .It is derived from istream and ostream classes.There are three more derived classes istream_withassign,ostream_withassign and iostream_withassign.They are derived from istream,ostream and iostream respectively.

4.There are two methods constructor of class and member function open() of the class for opening the file.

5.The class ostream creates output stream objects and ifstream creates input stream objects.

6.The close() member function closes the file.

7.When end of file is detected the process of readind data can be easily terminated.The eof() function is used for this purpose.The eof() stands for end of file.The eof() function returns 1 when end of file is detected.

8.The seekg () functions shifts the associated file's input file pointer and output file pointer.

9.The put() and get() functions are used for reading and writing a single character whereas write() and read() are used to read or write block of binary data.

Key Terms

argv

clear()

eof()

fail()

ios::in

ios::out

iostream

ofstream

filemode

filebuf

get()

seekg()

open()

put()

read()

seekp(
)

Exercises

6.1.What are input and output streams?

6.2.What are the various classes available for file operations.

6.3. What is a file mode ?describe the various file mode options available.

6.4.Describes the various approaches by which we can detect the end of file condition.

6.5.What do you mean by command line arguments?