UNIT 1 OBJECT ORIENTED PROGRAMMING

Structure **Page Nos.**

1.0 INTRODUCTION

The computer programs are the means used by human beings for communication with the machines especially the computers. As you all know, programs or the software contain instructions for the hardware to accept inputs, to process those inputs according to the instructions and to produce information as per the instructions contained in the program. The term 'programming' today is used to define the solution to a specific problem to be solved with the help of programs and it is said to define its solutions in terms of its design, creation, testing, debugging, implementation and its maintenance. Throughout the history of programming, ever increasing complexity of the problems to be solved using computers has encouraged the researchers and developers to evolve better means to manage this complexity. Complexity and intended areas of application coupled with some other factors have led to the evolution of a number of programming paradigms. Various programming languages are in use today depending upon various existing programming paradigms.

The structured programming and the Object-Oriented Programming (OOP) paradigms are the two paradigms that have been drawing attention of programmers for last so many years. The term OOP was used by Xerox PARC for the first time in its programming language, Smalltalk referring to the usage of objects as computational units for processing. The language Smalltalk itself got its inspiration from another OOP language called Simula 67 developed under the aegis of Simula Project in late 60s. The feature of inheritance introduced for the first time in Smalltalk allowed it to surpass both Simula 67 as well as other analog programming systems. Simula 67 and Smalltalk paved the way for many other OOP languages including C++ by 1980s.

This unit starts with a discussion on what a program is and what the programming is all about. It further highlights various important programming paradigms focussing basically on the structured and OOP paradigms. Subsequently, you will learn the main concepts involved in the OOP have been presented along with the benefits of OOP.

1.1 OBJECTIVES

After going through this Unit, you will be able to:

- understand the concepts of program and programming;

- know about various major computer programming paradigms;
- explain the structured and OOP paradigms and to appreciate the differences between these two;
- gain insight into various concepts that support the OOP; and
- describe the benefits of OOP.

1.2 PROGRAM AND PROGRAMMING

As you might be aware, the two essential components of any computer system are hardware and software. Both hardware and software have their own sets of functionalities which can be interdependent or independent of each other. A computer system is designed to produce the desired results by making the functionalities of both the hardware and the software to converge. The hardware is what we can see, touch and feel e.g. keyboard, mouse, visual display units like monitors, printers etc. Once it has been designed and manufactured to provide a certain set of functionalities, it can not be modified easily. Any modification in the hardware requires lot of effort, time and money. That is why we don't change our hardware very frequently. If the computers are required to carry out only a few predefined operations, these can be very easily embedded in the design of its hardware. But this kind of a computer completely lacks flexibility. In order to provide flexibility to perform some different operation in a computer, most of the existing hardware requires to be replaced with a newer one; whenever a new operation is added or older operations are to be abandoned or modified. Therefore, a computer system always contains a minimum basic hardware which is used by the software to provide lot of flexibility of operations. The software can be modified / replaced with lesser effort, time and money.

As such, a computer is essentially a data processing machine which requires two kinds of inputs for its operations and these are: data and instructions. The hardware of a computer can not produce the desired results unless it is given the requisite instructions and data by the user(s). The data is what needs to be processed by the hardware and the instructions (from within the set of its functionalities) tell this (minimum basic) hardware how to process that data step by step within the realms of set of functionalities so that expected results are achieved. Do you know what is software all about? The software deals with the instructions. The examples of software are Microsoft Office, Microsoft Windows 7, Red Hat Linux, Railways Reservation System, Microsoft Internet Explorer, Google Search Engine etc.

A software is an integrated set of interrelated programs which instruct the hardware as to what to do and how; and is responsible for getting the desired jobs executed by the computer to produce the predetermined outputs.

A program as an independent entity or as part of a software is intended to instruct the hardware to carry out specific task(s) to the satisfaction of the user(s) by providing specific outcomes. So how do you define a program? A program can be defined to be a set of instructions written in a programming language which are given in a fixed sequence to the hardware of a specific computer and executed by its hardware to produce predetermined and expected outcomes. The instructions in a program are written mostly in natural languages (e.g. English, Hindi, French, German, and Chinese etc.) following the syntax (form) and semantics (meaning) of the programming language chosen for writing the program. There are a variety of programming languages available for writing the programs e.g. BASIC, C, C++, Java, Prolog, Lisp, HTML, PHP etc. The sequence of instructions is very important because if the sequence is not correct, the expected results cannot be achieved by the program.

Now, let us see what does programming mean to us? The meaning of the term programming (or computer programming) has been changing rapidly since the idea of a first program was envisaged. Initially the computers were used to solve the mathematical problems with the help of calculations. Hartee in 1950 suggested that

the process of preparing a calculation for a machine can be broken down into two parts, 'programming' and 'coding'. He described programming as the process of drawing up the schedule of the sequence of individual operations required to carry out the calculation. Before the availability of assemblers, coding was in fact, a very tedious and time consuming task. Soon, programming became the major activity in this process.

In 1958, Booth proposed that the process of organizing a calculation can be divided into two parts, a) the mathematical formulation, and b) the actual programming. With the passage of time, the definition of programming has kept on evolving and at present programming is considered to be the process of writing programs and may include activities as diverse as designing, writing, testing, debugging and maintaining the code of a program. In normal conversation, programming is described as the process of instructing the computer to do something desired and useful for the user with the help of a programming language.

☞ Check Your Progress 1

Objective type Questions:

1) How are hardware and software related in a general purpose computer?

   a) They are independent of each other
   b) Hardware has to be dependent on software
   c) Software (System) has to be dependent on hardware
   d) None of these

2) Which is of the following is not a part of the definition of the program?

   a) Instructions
   b) Sequence
   c) Data
   d) Desired output

3) Which one of the following is not a programming language?

   a) C++
   b) HTML
   c) BASIC
   d) English

Answers to short type questions:

1) Why can hardware of a computer not produce the desired results in the absence of instructions (program / software)?

   ……

   ……

   ……

2) How is a program related to software?

   ……

   ……

   ……

3) Why does hardware not provide flexibility of operations to the user(s)?

   ……

   ……

   ……

1.3 PROGRAMMING LANGUAGES

You must appreciate the fact that the programming languages are created by, we, human beings. These languages are used to communicate instructions to the machines especially computers so that the programs can control the behaviour of the hardware of the machines to get desired results. Basically, the hardware of the computers understands only the language of the hardware which is called the machine language. The hardware is unable to understand and decipher any program written in any other programming language. Moreover, every type of a CPU has its own machine language. Therefore, in order to make the hardware of a computer understand the instructions contained in a program written in any other programming language, a mechanism called 'translator' is required. This translator converts the program written in programming languages other than the native machine language of the CPU (hardware) into the native machine language of a particular CPU on which this program is intended to be executed. Every programming language must have its own translator for the programs written in it to be executed or run on the computer hardware. Various types of translators available can be categorized into assemblers, interpreters or compilers.

The primitive or the first generation of programming languages were called machine languages and the symbols like '0' and '1' were used to write programs under this category of programming languages. The second generation of programming languages were called the assembly languages and mainly used mnemonics to construct a program. Both of these generations of programming languages were CPU dependent i.e., every type of a CPU will have its own machine and assembly language. The third generation of programming languages was called high level languages as these programming languages were independent of the CPU of the hardware being used and the instructions written in the programs were just like the instructions given in natural languages. The third generation languages are known as 3GL languages. The current generation of the programming languages are called the fourth generation languages or the 4GLs. These languages represent the class of programming languages that are closest to the human (natural) languages.

Based on the intended use of domain of use, the programming languages are broadly classified as imperative programming languages where imperative sentences are used in a program to issue commands in terms of instructions; and declarative programming languages where declarative instructions are used in a program to assert the desired result. But a more common paradigm classifies these languages into imperative, functional, logic programming and object-oriented languages. Table 1.1 presents the summary of main features of these programming paradigms.

Table 1.1: Summary of Main Features of Programming Paradigms

Paradigm	Key Concepts	Program	Program Execution	Result
Imperative	Command (instruction)	Sequence of commands	Execution of commands	Final state of computer memory
Functional	Function	Collection of functions	Evaluation of functions	Value of the main function
Logic	Predicate	Logic formulas: axioms & a theorem	Logic proving of the theorem	Failure or Success of proving
Object-oriented	Object	Collection of classes of objects	Exchange of messages between the objects	Final state of the objects

Table 1.2 shows some of the examples of programming Paradigms.

Table 1.2: Programming Languages under Programming Paradigms

Paradigm	Example
Imperative	Algol, Pascal, C, Ada
Functional	Lisp, Refal, Planner, Scheme
Logic	Prolog
Object-oriented	Smalltalk, Eiffel, C++, Java

There exist certain programming languages that inherit the features of more than one paradigms and the examples of some modern programming languages are presented in Table 1.3.

Table 1.3: Programming Languages under Two Programming Paradigms

Paradigms	Example
Imperative + Object-oriented	Object Pascal, C++, Java, Ada-95
Functional + Object-oriented	Clos
Logic + Object-oriented	Object Prolog

1.4 STRUCTURED PROGRAMMING PARADIGM

The structured programming paradigm is a sub discipline of procedural programming under the category of imperative programming paradigm. Most of the present day procedural programming languages include the features that encourage structured programming. Do you know who proposed this paradigm in the first instance? It was first proposed by two mathematicians Corrado Bohm and Guiseppe Jacopini who proposed and demonstrated that a computer program may contain just three structures namely decisions, sequences, and loops.

Structured paradigm is based on the principle of building a program from logical structures.

Any program can be created by breaking large programs into smaller modular routines and imposing these logical structures. That is why structured programming is also sometimes known to follow the concepts of modular programming. This paradigm generally follows the top down approach where the complex programming blocks are broken down into smaller blocks maintaining a well defined structure and organization of the overall program. It discourages the use of global variables and instead encourages to use variables that are local to each of the blocks. Moreover, the use of GOTO statement is completely forbidden in the languages supporting this paradigm. Some of the languages that follow this paradigm are Pascal, C, C++, Java, Ada etc. in contrast to non-structured programming languages like BASIC, COBOL, FORTRAN etc.

The structured programming follows the principle of divide and conquer. A solution of a problem can be said to consist of (or include) a set of tasks, on the same lines, a program can also be designed to perform a set of tasks by dividing it into the task performing blocks.

Under unstructured programming paradigm, a) mostly, all the program code is written in a single continuous main program, b) logic is difficult to follow within the program,

c) code from other programs is hard to incorporate, d) it is difficult to test specific portions of a program, and e) program is difficult to debug and maintain. In contrast, structured programming is defined as a programming paradigm which follows certain set of quality standards to create programs that are more reliable and readable; and easier to maintain. Under this paradigm, the aim is that before the code is written, the structure of a program is required to be defined clearly and a decision to attach other programs and libraries be taken.

Some of the major advantages and disadvantages of structured programming are given below:

1.4.1 Advantages of Structured Programming

a) Complexity can be reduced using the concepts of divide and conquer.

b) Logical structures ensure clear flow of control.

c) Increase in productivity by allowing multiple programmers to work on different parts of the project independently at the same time.

d) Modules can be re-used many times, thus it saves time, reduces complexity, and increases reliability.

e) Easier to update/fix the program by replacing individual modules rather than larger amounts of code.

f) Ability to either eliminate or at least reduce the necessity of employing GOTO statement.

1.4.2 Disadvantages of Structured Programming

a) Since GOTO statement is not used, the structure of the program needs to be planned meticulously.

b) Lack of encapsulation.

c) Same code repetition.

d) Lack of information hiding.

e) Change of even a single data structure in a program necessitates changes at many places throughout it, and hence, the changes become very difficult to track even in a reasonably sized program.

f) Not much reusability of code.

g) Can support the software development projects easily up to a certain level of complexity. If complexity of the project goes beyond a limit, it becomes difficult to manage.

1.5 OBJECT-ORIENTED PROGRAMMING PARADIGM

Simula was the first programming language developed in the mid-1960s to support the object-oriented programming paradigm followed by Smalltalk in the mid-1970s that is known to be the first ‘pure’ object-oriented language. Eiffel, Java, C++, Object Pascal, Visual Basic, C# etc are the other OOP languages that came into existence later on, all having different complexities of syntax and dynamic semantics.

The main motive of the developers of programming languages over the years has always been to create such programming languages that are close to human (i.e. natural) languages. The way we perceive and interact with the things in our day-to-day lives, the representation of programming constructs should closely match the same. Hence came into existence the concept of 'objects' and 'object-oriented programming paradigm'.

The real world can be considered to be made up of various objects with which the human beings regularly interact e.g. a ball pen, a tooth brush, a vehicle, a foot bear or a house etc.

Every object has certain defining properties which distinguish it not only from different types of other objects but from the similar types of objects too. If we take an example of an object like a ball pen, its defining property may be the colour in which it can write, length, shape, unique manufacturing code etc. Some of these properties not only distinguish the ball pen object from the tooth brush object but also distinguish individual ball pens objects. It must be understood clearly that no two objects in the physical world, even of same type, are identical since no two objects can have the value of all its defining properties same.

Likewise, every object has certain functions associated with it and all similar types of objects are supposed to support these. Although, Some of these functions can be the same as associated with different types of objects. In case of a ball pen, one of the functions associated with each type of ball pen object, is to write and another associated function is the provision to hold it in hands conveniently. All the ball pen objects support both these functions. Incidentally, all the tooth brush objects also support the function of holding them in the hands conveniently but, in addition, support other functions like brushing the teeth too.

This concept of objects borrowed from the real world has been the basis of the object-oriented programming (OOP) paradigm and this paradigm is a direct consequence of an effort to have a programming language closely matching the human behaviour. This OOP paradigm is all about creating program(s) dealing with objects where these objects interact with one another to achieve the overall objectives of the program. Every object in the programs has certain defining properties called attributes (or instance variables) possessing supporting values for each of the attributes and some associated functions (normally called methods or operations). As in the real world objects, no two objects in a program can have the same values of all the attributes.

At times, instead of dealing with individual objects, it is convenient to talk collectively about a group of similar objects where all the objects of this group will have the same set of attributes and methods. In the object-oriented programming paradigm parlance, this collection of objects corresponding to a particular group is known as a class. All programs under this paradigm contain a description of the structure (corresponding to attributes) and behaviour (corresponding to methods) of so called classes. In a program, various objects are created from these classes. The process of creation of an object from a class is called 'instantiation' and the object created is known as an instance of the class. Every object created will have a 'state' associated with the description of the structure in the class from which it has been instantiated. The state of an object is defined by the set of values assigned to its corresponding attributes (and stored in the memory) of the object.

Therefore, we can say that a class is used to represent a set of objects having same structure and behaviour. So, how do we define a class? A class is defined to be a template or a prototype so that a collection of attributes and methods can be described within it and this definition can be used for creating different objects within a program. It is this concept of encapsulating the data and methods within the objects that provides the programmers with flexibility within the OOP paradigm because an object can be extended or modified without making changes to its external interface or other classes/objects in the program. Various classes may exhibit features like inheritance and polymorphism of methods.

When a program is executed, various objects are created with their corresponding states and these objects interact with one another by exchanging messages, the messages thereby causing the modification of their states. Modification in the state of an object is said to have occurred when the values of one or more of its attributes (instance variables) change due to the interaction. All the objects created are made to exhibit a behaviour through their corresponding methods such that the program produces the desired results once its execution is over. A program in this paradigm therefore, becomes a collection of cooperating and interacting objects instead of just a list of objects.

For the sake of an example, a Maruti Wagon-R could be an object. It would have a state depend upon whether its engine is running or not. Also it would have a behaviour, like starting ignition of its engine or stopping the ignition of its engine and this behaviour is responsible for changing its state from 'engine is not running' to 'engine is running' or from 'engine is running' to 'engine is not running'.

In a different example, we can take object ABC representing a student having a state defined by its attribute named RESULT. A part of the behaviour of this object could be reflected through 'compute result'. If this student has not appeared in the examination yet, the state of this object defined by the attribute RESULT may be NOT DECLARED. But once this student has appeared in the examination and the marks obtained by this student are available, the behaviour of this object changes the state of the object ABC by using the method 'compute result' from NOT DECLARED to either FAIL or PASS with percentage of marks.

1.6 STRUCTURED Vs OBJECT-ORIENTED PROGRAMMING

The OOP, can be considered to be a type of structured programming which uses structured programming techniques for program flow, but adds more structure for data to be modelled. We have highlighted some of the basic differences between the two as under:

1) OOP is closer to the phenomena in real world due to the use of objects whereas structured programming is a bit away from the natural way of thinking of human beings.
2) Structured programming is a subset of object-oriented programming. Therefore, OOP can help in developing much larger and complex programs than structured programming.
3) Under structured programming, the focus of a program is on procedures or functions (behaviour) and the data is considered separately (data structures are not well organized within the program) whereas in OOP, data (structure) and methods (behaviour) both are in collective focus.
4) In OOP, the basic units of manipulation are the objects whereas in case of structured programming, functions or procedures are the basic units of manipulation.
5) The focus of a program is on manipulation of data in structured programming whereas the focus of OOP is on both data (structures and states of objects) as well as on its manipulation (behaviour of objects).
6) In OOP, data is hidden within the objects and its manipulation can be strictly controlled whereas in structure programming the data in the form of variables is exposed to unintended manipulation too.

7) The OOP promotes the reuse of classes and their parts of the code too. In addition, it also supports the inheritance of state and behaviour. This feature is missing in structured programming.
8) The OOP supports polymorphism of operations.
9) The concepts of OOP have got integrated with most of the prominent object-oriented methodologies of software development in a better way and help in reducing the development effort and time as compared with that of structured methodologies based on structured programming.
10) The structured programming normally emphasized on single exit point in their constituent functions or procedures. Since each function/procedure allocates some memory to itself for storing the values of its variables and code, before the exit, there must be a provision for deallocating this memory. Otherwise, more and more of the memory gets occupied by each function/procedure and in large programs, there may occur a shortage of memory for use by other functions/procedures and the processing can get slower or even completely halted. When in any function/procedure, memory is allocated, but not deallocated, a memory leak is said to have occurred (the memory has leaked out of the computer) in it.

1.7 OBJECT-ORIENTED PROGRAMMING CONCEPTS

The Object-Oriented Programming is based on sound principles and provides the developers of various object-oriented programming languages with a variety of new concepts to be incorporated in those languages. Some of the commonly found important concepts in most of the OOP languages are as given below:

Abstraction (data)

Abstraction is a technique which allows the hiding or elimination of the irrelevant; and focussing on the essential. According to IEEE 1983 definition, abstraction is defined as a view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information. There are different levels of abstraction. As in real life, for an example of a car, a buyer would see the car at a different level of abstraction, designer has different level of abstraction to look at it, a mechanic sees it at another level of abstraction, a junk yard owner sees the car at an altogether different level of abstraction. The buyer is interested in the colour, mileage, shape, manufacturer etc; the designers are concerned about the minute details like designing the fuel tank, ignition system, electrical parts and their wiring, breaking system etc; a mechanic may be concerned about how to test the battery, how to open and reconnect various parts etc, spare parts etc; and the junk yard owner is interested only in how much reusable metallic and plastic parts are there in the car. If we, therefore, apply the same concept of abstraction to a program in a given context, a programmer hides or eliminates the irrelevant attributes and methods and use only the attributes and methods that are relevant for a given class or an object.

If we take another example of a class ‘student’, the class may have plenty of attributes like, name, roll number, father’s name, mother’s name, date of birth, class, year or semester, course, marks obtained, address for correspondence, height, weight, colour of hair, colour of eyes, size of the shoes they wear, no. of teeth, finger prints, IQ level etc. The list can be very long. But, when we use this ‘student’ class in some program, not many of these attribute would be required to be used. Only those attributes which are of interest (i.e. the attributes that are required to define the state of the program at any point of time during the execution of the program) shall be included in the definition of the class and rest of these attributes shall never be used and hence be not

included in the program. Same is true for behaviour depicted by the methods of the class 'student'.

Information Hiding

Information or data hiding in an object is characterised by its knowledge of a design decision which it hides from all other objects. The interface of an object is chosen to reveal only the desired data or working of the object. According to the definition of information/data hiding given by Booch in 1991, it is the process of hiding all details of an object that do not contribute to its essential characteristics; typically, the structure of an object as well as the implementation of its methods is hidden from other objects. There can be two types of information hiding: functional information hiding related to the hiding of implementation details of methods (behavioural information of a particular object) and data hiding (structural information of a particular object). As in the case of abstraction, there are varying degrees of information hiding. Some languages like C++ also allow varying degree of visibility of objects like public, private and protected. So the mechanism of information hiding is said to provide a strictly controlled access to the information enclosed within the capsule.

Encapsulation

In 1991, Rumbaugh and others defined encapsulation as consisting of separating the external aspects of an object which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects. According to Booch, it is also known as information hiding and it prevents clients from seeing the object's inside view, where the behaviour of the abstraction is implemented. In reference to classes and objects in OOP, encapsulation is the process of enclosing within these classes and objects the attributes and the methods. It is the programmers who specify what information in an object can be shared with other objects. Figure 1.1 illustrates the concepts of encapsulation and information hiding.

Encapsulation refers to how the implementation details of a particular class are hidden from all objects outside of the class.

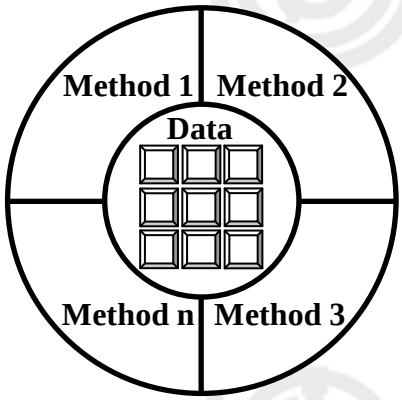


Figure 1.1: Encapsulation & Information Hiding Concept

Do you think information hiding and encapsulation mean the same thing? No, information hiding cannot be treated as encapsulation, both are related but altogether different aspects e.g. an array or a record structure also encloses the information but this information cannot be said to be hidden. It is true that the encapsulation mechanism like classes and objects hide information but these also provide visibility of some of their information through well defined interfaces.

Classes and Objects

A class is a collection of similar entities which have same structure and exhibit same behaviour. These are used to describe something in the real world like places,

organizations, roles, things, occurrences etc. A class is said to describe the structure and behaviour of these sets of similar entities called objects. As opposed to actual objects, the class gives a general description of these objects like a template, blueprint or a pattern; and contains the definitions of all the attributes and methods which will become the part of each object created from the class. Only after a class has been defined, specific instances of the class can be created and these instances are called the instances of that class. The process of creation of these instances as objects of the class is called instantiation. Table 1.4 cites certain examples of classes and their objects for your ready reference.

Table 1.4: Examples of Classes & Corresponding Objects

Type	Example of Class	Example of Object
Place	Hill station	Shimla
Organization	University Department	Computer Science
Occurrence	Alarm	Fire alarm
Role	Teacher	Manoj Kumar
Thing	Car	Maruti Wagon R

Figure 1.2 shows an example of a class having an identifier as 'Teacher', its structure defined by attributes 'Name' of type 'String' and 'Age' of type 'Integer' depending upon the particular context. The behavior of this class in a using abstraction is depicted by a single method 'evaluate'. In this particular context, the programmer is required to focus only on name and age of the teachers as part of the structure of this class and in the evaluation method as part of behavior of this class.

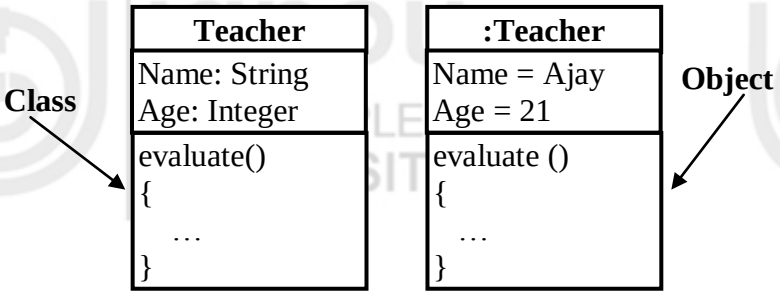


Figure 1.2: Concept of a Class and a Corresponding Object

In different contexts, there may be different sets of attributes and methods of interest for the programmer. An object (instance) created from this class is also shown in this figure having values of attributes 'Name' and 'Age', these values of attributes at any point of time also describe the state of this object. The behaviour of an object is defined by the set of methods which can be applied on it.

Message Passing

In object oriented languages, you can consider a running program under execution as a pool of objects where objects are created for 'interaction' and later destroyed when their job is over. This interaction is based on 'messages' which are sent from one object to another asking the recipient object to apply one of its own methods on itself and hence, forcing a change in its state. The objects are made to communicate or interact with each other with the help of a mechanism called message passing. The methods of any object may communicate with each other by sending and receiving messages in order to change the state of the object. An Object may communicate with other objects by sending and receiving messages to and from their methods in order to

change either its own state or the state of other objects taking part in this communication or that of both. An object can both send and receive messages.

The messages are sent and received by passing various variables among specific methods using the signatures (a term that is not prevalent in common parlance) of the methods. Every method has a well defined and structured signature. The signature of a method is composed of: a) a type, associated with the variable whose value after execution of the method, would be returned to the object that would invoke the method; b) the types of a specific number of variables and the order associated with these variables whose values would be passed to the method before execution of the method starts. All these variables have a well defined format and corresponding values at any instant of time available for communication during the execution of the program. Figure 1.3 shows an example of a signature for a method 'evaluate'.

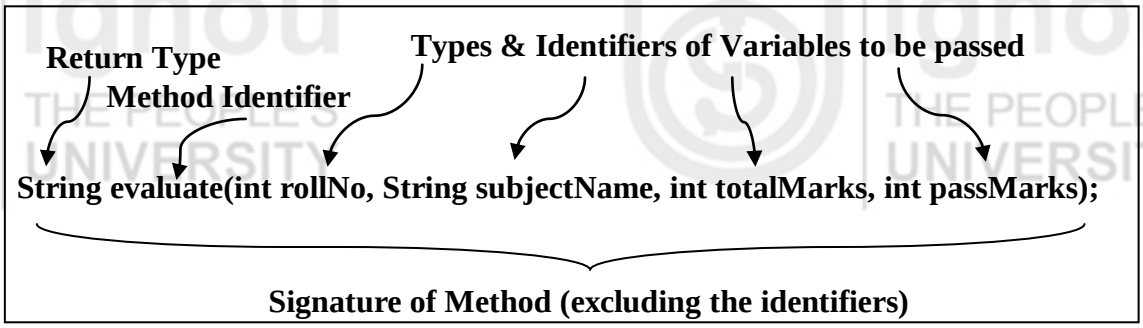


Figure 1.3: Signature of a Method

Interface

Every class defines an interface for itself and its objects use only this interface for all types of communications. We may say that an interface of a class or an object is the collection of signatures of all the methods contained in the class or the object. It is through this interface, the objects communicate with themselves or with other objects by passing value of variables to and fro and hence, in the process, changing their own state or hat of other objects or that of both. As the signatures of various methods of an object are well structured and precisely defined, therefore, the interface of an object also has a well defined precise structure. As you can see, figure 1.4 shows the mechanism of message passing between two objects through their interfaces. Various OOP languages have different mechanisms to implement the interfaces.

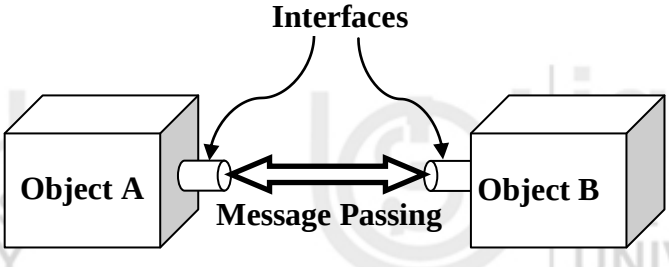


Figure 1.4: Objects Interacting through Interfaces

Association

The classes and hence the corresponding objects in OOP languages are in relationship with one another. Various kinds of vital relationships are association, aggregation, inheritance etc. An association is the term used to represent the relationships among various objects of one or more classes. An illustration of association is given in figure

1.5. The association here has been represented by a simple straight line whereas the classes have been represented by rectangles.

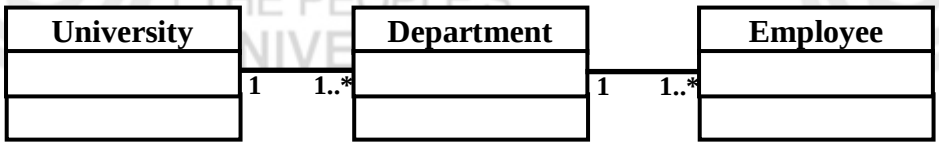


Figure 1.5: Association & Multiplicity among Classes

The term 'multiplicity' represented by a numerical specification, is used to indicate how many objects of one side of an association are connected with how many objects on the other side. Common categories of multiplicities have been enlisted in Table 1.5.

Table 1.5: Various Categories of Multiplicity

0..1	No instance, or at the most one instance
1	Exactly one instance
0..*, *	Zero or more instances
1..*	One or more instances

Aggregation is a special form of association. It is the composition of an object out of a set of its parts. A university, for example, is an aggregation of departments, employees, students, class rooms, faculty rooms, laboratories and so on. Some of these parts may further be the aggregations of some other parts. Sometimes, aggregation is also known as a 'whole-part' hierarchy (university is 'whole' and department is a 'part') and represented as 'has-a' relationship (a university has-a department).

Inheritance

Can you guess what inheritance is? Inheritance is a mechanism by virtue of which the classes inherit the attributes and methods of some other class(s). In a sense, we can say, inheritance is a way to reuse the code of some existing or already defined classes. The classes, in this case, are said to have 'a-kind-of' and 'is-a' relationship with the other classes. Using this property of OOP, one class can extend other classes by including additional methods and/or attributes (variable). The original class is called the 'superclass' of the extending class and the extended class is called the 'subclass' of the class that is being extended. The subclass is sometimes known with the name of 'derived' class and the superclass with the name of 'base' class. The derived or subclasses classes can further be used to have their own derived subclasses. This kind of a relationship of classes through inheritance gives rise to an inheritance hierarchy of the classes. Can you explain, why?

As an example, if you take 'car' as a superclass; then you can treat SUV, sedan, sports car, roadster as its subclasses. All or some of these subclass cars have some common attributes (structure) and methods (behaviour). But the structure and behaviour of these subclass cars is not restricted to those of the superclass 'car'. A subclass car can contain methods and attributes in addition to those inherited from the superclass 'car' e.g. a sports car will generally have only two doors whereas a sedan will have four. The subclasses can also contain methods that override the methods they have inherited to have unique implementation of these methods. Another example of inheritance can be that of a 'geometric figure' as a superclass and a 'circle' and a 'triangle' as its subclasses as shown in figure 1.6.

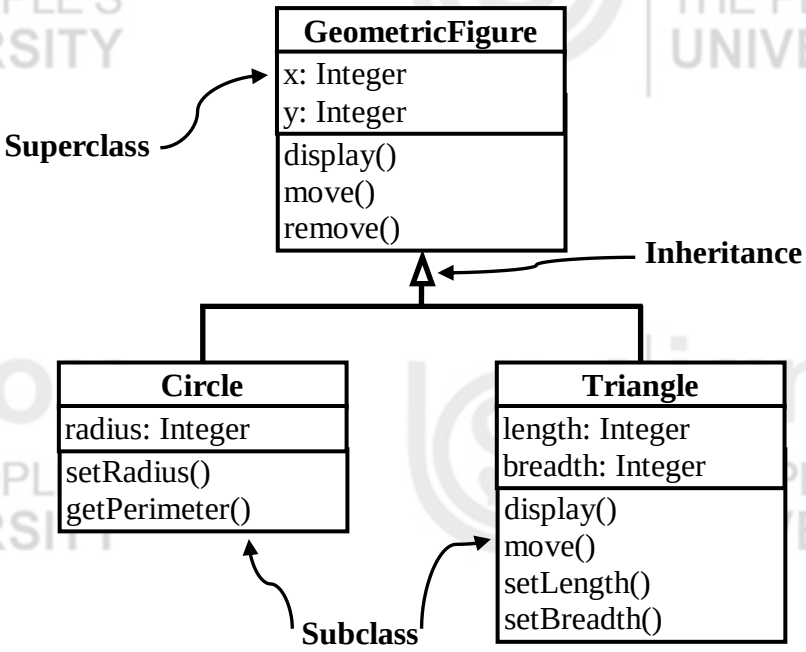


Figure 1.6: Inheritance among Classes

Various categories of Inheritance which are often used in OOP are single level, multi-level, multiple inheritance etc.

Polymorphism

Polymorphism or the ability to appear in many forms, is one of the vital primary characteristic concepts of OOP. 'Poly' means 'many' and 'morph' means 'form'. In reference to OOP, it is an ability of assigning different meanings to entities such as variable, methods or objects so that these can be made to exhibit more than one form. It provides the programmers with the flexibility of processing any object differently depending upon their data types. Using this concept, a programmer can redefine various methods of the classes derived from their base classes. Objects of different types can receive the same message and respond in different ways provided these objects have the same method definition (i.e. interface). The calling object, also sometimes known as the client, need not know what type of object it is calling, the only thing that it needs to know or ensure is that the called object has a method of a specific name with defined arguments. Polymorphism is more often than not applied to derived classes, where the methods of the parent class are replaced with those having different behaviours. It is the concepts like inheritance and polymorphism that together make OOP flexible and easy to extend.

Do you know how many categories of polymorphisms exist? There are two categories of polymorphisms; static or compile-time and dynamic or run-time. In static polymorphism, which form of the method (from among the various available forms) is to be called and executed is decided during the time of compilation, the example being 'Method Overloading'. In method overloading, same method name having different parameters is used more than once in the same class. Which method is to be called and executed depends upon the parameters passed by the calling method and is decided during the compilation of the program. The dynamic polymorphism is applied in the form of method overriding which means there can exist two or more methods in a program which have the same signature (name; return type; type, number and order

of arguments to be passed) having different implementations. In figure 1.7 these two types of polymorphisms are explained.

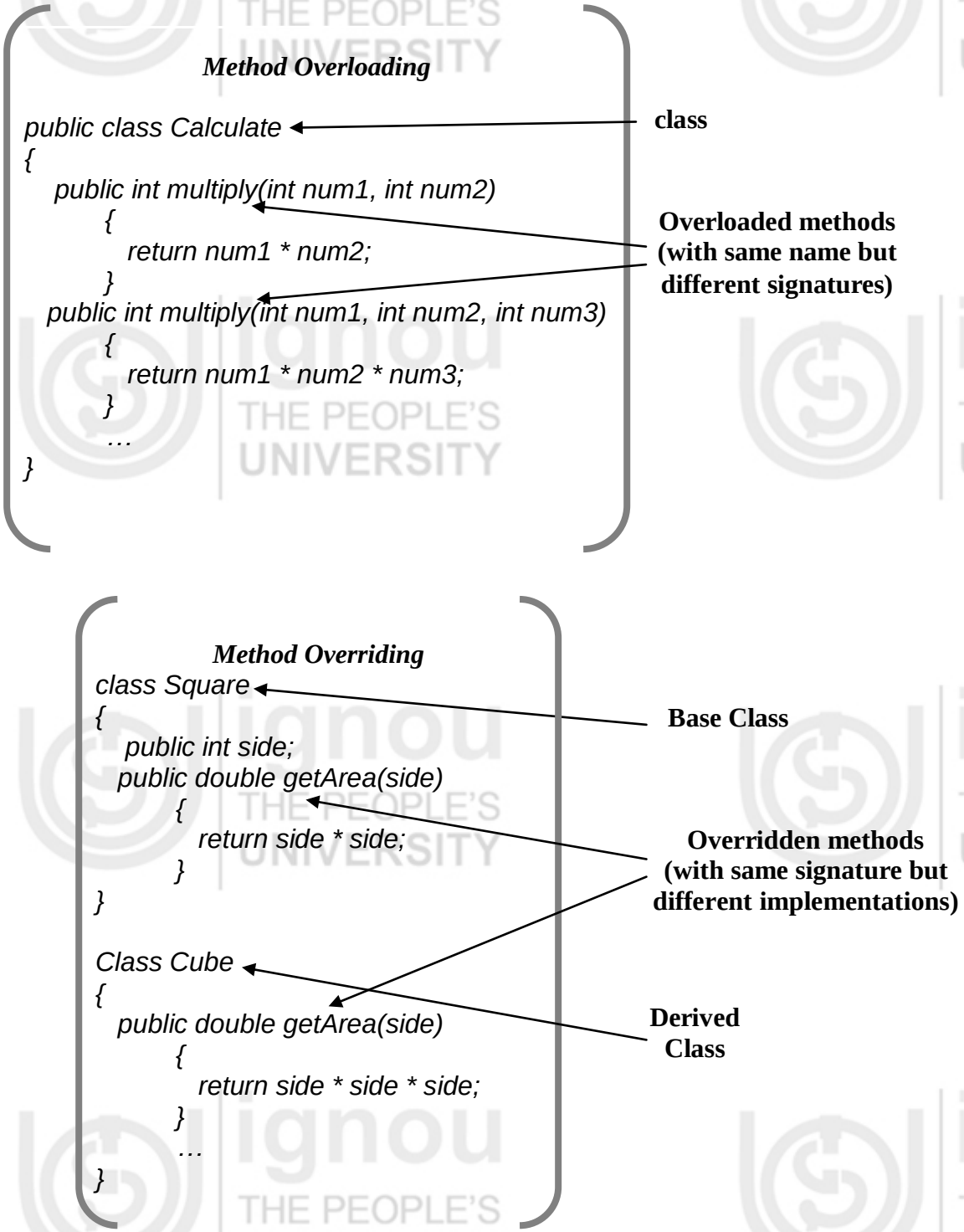


Figure 1.7: Two Different Types of Polymorphism

In addition to object-oriented programming, the programmers sometimes also use object-based programming languages. So, how is object-based programming different from object-oriented programming? According to Rumbaugh, object-oriented programming should be supported by the languages which have at least the following four features:

a) Identity which means the quantization of data in terms of entities called the objects that are discrete and distinguishable;
b) Classification into classes i.e. grouping of objects of same structure and behaviour;
c) Polymorphism i.e. depiction of different behaviour of same operations on different classes; and
d) Inheritance in terms of sharing of structure and behaviour among classes in a hierarchical relationship.

But, what about the languages that support similar kinds of features? As such, there are certain languages that may support some of these feature but not all. The languages that support only some of these features like identity and classification (and may be polymorphism too) do not qualify to be called object-oriented programming languages. These programming languages are called object-based programming languages. Visual Basic is one such programming language which supports objects and classes but not inheritance and that is why it is called an object-based programming language. In contrast, VB.NET is an object-oriented programming language. Fortran 90 is another example of object-based programming language that does not support inheritance. Another example is JavaScript, a language that does not have classes. In this language, the objects can be made to inherit code and data directly from the template objects.

1.8 BENEFITS OF OOP

By now, you might have understood the basic concepts of object-oriented programming. Therefore, you are in a better position to appreciate the following as some of the major benefits of OOP:

1) As OOP is closer to the real world phenomena, hence, it is easier to map real world problems onto a solution in OOP.
2) The objects in OOP have the state and behaviour that is similar to the real world objects.
3) It is more suitable for large projects.
4) The projects executed using OOP techniques are more reliable.
5) It provides the bases for increased testability (automated testing) and hence higher quality.
6) Abstraction techniques are used to hide the unnecessary details and focus is only on the relevant part of the problem and solution.
7) Encapsulation helps in concentrating the structure as well as the behaviour of various objects in OOP in a single enclosure.
8) The enclosure is also used to hide the information and to allow strictly controlled access to the structure as well as the behaviour of the objects.
9) OOP divides the problems into collection of objects to provide services for solving a particular problem.
10) Object oriented systems are easier to upgrade/modify.
11) The concepts like inheritance and polymorphism provide the extensibility of the OOP languages.
12) The concepts of OOP also enhance the reusability of the code written.
13) Software complexity can be better managed.
14) The use of the concept of message passing for communication among the objects makes the interface description with external system much simpler.
15) The maintainability of the programs or the software is increased manifold. If designed correctly, any tier of the application can be replaced by another provided

the replaced tier implements the correct interface(s). The application will still work properly.

☞Check Your Progress 2

Objective type Questions:

1) Which of the following is not a disadvantage of structured programming?
   a) Availability of information hiding b) Lack of encapsulation
   e) Non-availability of GOTO statement d) None of these

2) Which is of the following is not a concept associated with OOP?
   b) Information hiding b) Clauses
   d) Abstraction d) Message Passing

3) Which one of the following describe method overloading the best?
   a) Same signature, different implementation
   b) Same name, different signatures
   c) Different name, same signatures
   d) Different name, different signatures

Short Answer type Questions:

1) What is the signature of a method?

   ……

   ……

   ……

2) Differentiate between information hiding and encapsulation.

   ……

   ……

   ……

3) What is the difference between object-oriented and object-based programming languages?

   ……

   ……

   ……

1.9 SUMMARY

The programs are the means through which we can make the computers produce the useful desired outputs. Out of a variety of programming paradigms being used by practitioners as well as the researchers, the structured and the object-oriented programming paradigm and corresponding structured and object-oriented programming have been in focus for quite some time now. In this unit, you studied that the structured programming languages initially helped in coping with the inherent complexity of the softwares of those times but later on, were found wanting in the handing the same as far as the software of present days are concerned. In the backdrop

of this, you also studied the advantages and the disadvantages of the structured programming.

Next, you were introduced to the concepts of object-oriented programming paradigm and it was illustrated as to how this paradigm is closer to natural human thinking. Subsequently, an overview of the differences between the structured and object-oriented programming paradigms was presented to you so that you can clearly understand these intricate differences. After that, the basic concepts of object-oriented programming well supported by relevant illustrations were introduced to you. Here we first defined abstraction, encapsulation and information hiding elucidating the difference between the last two. It was followed by the illustrations of some more concepts of object-oriented programming like classes, objects, message passing, interface, associations, inheritance and polymorphism. In the end, you saw what the various benefits of OOPs are and how can these help in producing good quality software.

1.10 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

Answers to Objective type Questions:

1) c 2) c 3) d

Answers to Short Answer type Questions:

1) The hardware of a computer does not do anything on its own unless it is provided with instructions in the form of a program or software. These instructions are decoded and executed by the hardware to produce the desired result after getting the instructions from the user for doing so. Therefore, if no instructions are given to the hardware by the user, the hardware will not be able to do anything and hence the desired result will not be produced.

2) Software is a set of related programs which provide specific instructions to the hardware so that the intended results are achieved when the software is used by the computer hardware. A software may consist of a number of programs that are related to each other in such a way that these programs shall be interacting with each other while in execution. It is this interaction among various related programs which helps users to get their intended goals achieved through the execution of a particular software.

3) To provide any functionality through operations to the users, various hardware components are needed to be interconnected based upon the circuit diagram (design) and their operations need to be strictly controlled and synchronized especially with reference to time. These components along with the defined interconnections are then hardwired. Once the components are hardwired, they can not be changed. Even if a small change in functionality through operations is required to be carried out, a new circuit diagram needs to be designed and a new set of components needs to be interconnected all over again. This process is very expensive, wasteful and time consuming. In contrast, the software can be changed any numbers of times without much hassles.

Check Your Progress 2

Answers to Objective type Questions:

1) a 2) c 3) b

Answers to Short Answer type Questions:

1) The signature of a method consists of:
   a) Return type of the method
   b) Number of arguments to be passed
   c) Types of each of these arguments
   d) The sequence of these arguments

   It is through the signature of a method, the mechanism of message passing is supported for interaction within an object or among various objects of a program. Whenever any method of an object intends to communicate with another method of the same object or some other object, the message to be sent from the transmitting object will have to adhere to the format of the signature of the receiving method.

2) The mechanism of information hiding is said to provide a strictly controlled access to the information enclosed within the capsule. Encapsulation is the process of enclosing within classes and objects the attributes and the methods. But information hiding cannot be treated as encapsulation, it is different e.g. an array or a record structure also encloses the information but this information cannot be said to be hidden. It is true that the encapsulation mechanism like classes and objects hide information but these also provide visibility of some of their information through well defined interfaces.

3) Object-oriented programming languages are characterised by four essential properties: identity in terms of objects, classification in terms of classes, polymorphism and inheritance. Any other programming language that uses objects and in addition, supports at least one or more of these essential characteristics (may also support features other than these four, in addition) is called as an object-based programming language.

1.11 FURTHER READINGS

1) Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

2) Bolshakova E., Programming Paradigms in Computer Science Education, International Journal: *Information Theory & Applications*, 12(3), 2005.

3) http://en.wikipedia.org/wiki/Object-oriented_programming_language

4) http://en.wikipedia.org/wiki/Object-based_language

5) http://140.134.26.20/wbem/eng/ch3.html

6) http://www.desy.de/gna/html/cc/Tutorial/node2.html#SECTION00200000000000000000

7) The *C++ Programming Language* by Bjarne Stroustrup, Addison-Wesley, 3rd edition, 1997.

8) *C++ Programming Today* by Johbston Barbara Johnston 2nd Edition, PHI

9) *C++: The Complete Reference*, Herbert Schildt, 4th Edition, Mc Graw Hill.

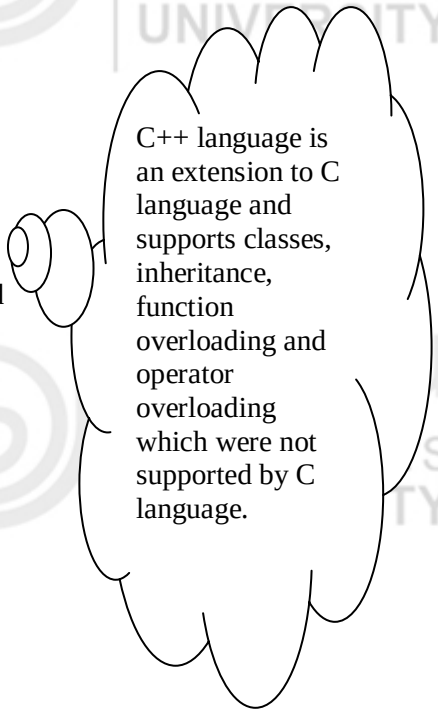
UNIT 2 INTRODUCTION TO C+ +

Structure **Page Nos.**

2.0 INTRODUCTION

In the previous unit, we have discussed concept of Objects Oriented Programming and benefit from this Object Oriented Language. In this unit we shall discuss something about Data Types, Operators and control structures used in C++. Data Type in C++ is used to define the type of data that identifiers accepts in programming and operators are used to perform a special task such as addition, multiplication, subtraction, and division etc of two or more operands during programming.

C++ is regarded as an intermediate-level language, as it comprises a combination of both high-level and low-level language features. C++ is an extension to C Programming language. C++ is one of the most popular programming languages and is used in the development of system software such as Microsoft Windows and Application Software such as device drivers, embedded software, high performance servers and client applications.



It was developed at AT&T Bell Laboratories in the early 1979s by Bjarne Stroustrup. Its initially name was C with classes, but later on in 1983 it was renamed as C++. It is a deviation from traditional procedural languages in the sense that it follows object oriented programming (OOP) approach which is quite suitable for managing large and complex programs.

An object oriented language combines the data to its function or code in such a way that access to data is allowed only through its function or code. Such combination of data and code is called an object. For example, an object called Student may contain data and function or code as shown in Figure 2.1:

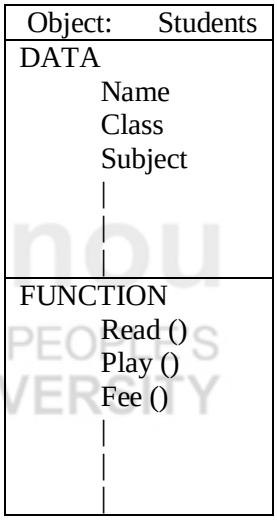


Figure 2.1: Representation of Object

The data part contains the Name, Class and Subject and function part contains three functions such as: read ( ), Play ( ) and Fee ( ). Thus, the various objects in the object-oriented language interact with each other through their respective codes or functions as shown in Figure 2.2.

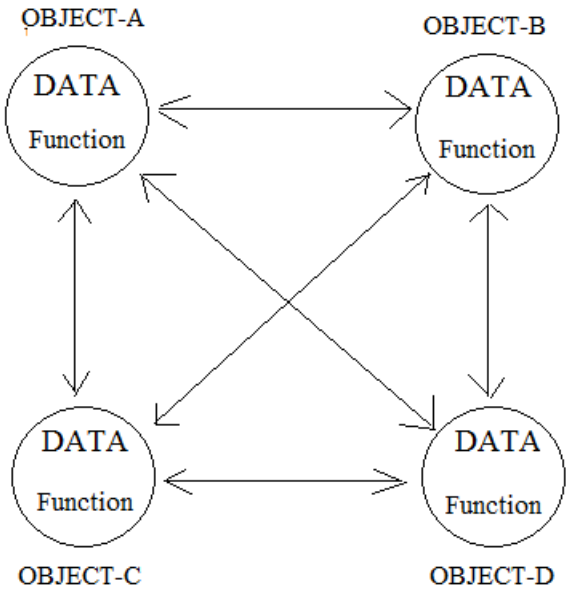


Figure 2.2: The Object-oriented approach

It may be noted here that the data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

2.1 OBJECTIVES

After studying this unit, you should be able to do the following:

- explain basic concepts of Object Oriented Programming Language;
- explain operators and their syntax in C++;
- learn about C++ character set, tokens and basic data types;
- identify the difference between implicit and explicit conversions;
- explain about Input/Output streams supported by C++;
- explain the structure of a C++ program;
- write a simple program in C++; and
- understand keywords used in C++ and control structure in C++.

2.2 BASICS OF C++

C++ is an object oriented programming (OOP) language. It was developed at AT&T Bell Laboratories in the early 1979s by Bjarne Stroustrup. Its initial name was C with classes, but later on in 1983 it was renamed as C++.

It is a deviation from traditional procedural languages in the sense that it follows object oriented programming (OOP) approach which is quite suitable for managing large and complex programs. C++ language is an extension to C language and supports classes, inheritance, function overloading and operator overloading which were not supported by C language.

In any language, there are some fundamentals you need to learn before you begin to write even the most elementary programs. This chapter includes these fundamentals;

basic program constraints, variables, and Input/output formats. C++ is a superset of C language. It contains the syntax and features of C language. It contains the same control statements; the same scope and storage class rules; and even the arithmetic, logical, bitwise operators and the data types are identical. C and C++ both the languages start with main function.

The object oriented feature in C++ is helpful in developing the large programs with clarity, extensibility and easy to maintain the software after sale to customers. It is helpful to map the real-world problem properly. C++ has replaced C programming language and is the basic building block of current programming languages such as Java, C# and Dot.Net etc.

2.2.1 C++ Character Set

Character set is a set of valid characters that a language can recognise. The character set of C++ is consisting of letters, digits, and special characters. The C++ has the following character set:

Letters (Alphabets)	A------Z, a-------z
Digits	0------9
Special Characters	+, -, *, /, ^, \, ( ), [ ], { }, =, !, < >. ‘, ", $, ;, :, % , &, ?, _, #, <=, >=, @

There are 62 letters and digits character set in C++ (26 Capital Letters + 26 Small Letters + 10 Digits) as shown above. Further, C++ is a case sensitive language, i.e. the letter A and a, are distinct in C++ object oriented programming language. There are 29, punctuation and special character set in C++ and is used for various purposes during programming.

White Spaces Characters:

A character that is used to produce blank space when printed in C++ is called white space character. These are spaces, tabs, new-lines, and comments.

Tokens:

A token is a group of characters that logically combine together. The programmer can write a program by using tokens. C++ uses the following types of tokens:

- Keywords
- Identifiers
- Literals
- Punctuators
- Operators

The identifier is a sequence of characters taken from C++ character set.

2.2.2 Identifiers

A symbolic name is generally known as an identifier. Valid identifiers are a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid.

In addition, variable identifiers always have to begin with a letter. In no case can they begin with a digit. Another rule for declaring identifiers is that they cannot match any keyword of the C++ programming language. The rules for the formation of identifiers can be summarised as:

An identifier may include of alphabets, digits and/or underscores.
It must not start with a digit.

C++ is case sensitive, i.e., upper case and lower case letters are considered different form each other. It may be noted that TOTAL and total are two different identifier names.

It should not be a reserved word.

A member function with the same name as its class is called constructor and it is used to initialize the objects of that class type with an initial value. Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, to avoid unexpected results in the example given below we have initialized the value of rollno as 0 and marks as 0.0.

2.2.3 Keywords

There are some reserved words in C++ which have predefined meaning to complier called keywords. These are also known as reserved words and are always written or typed in lower cases. There are following keywords in C++ object oriented language:

List of Keywords:

asm	**double**	**new**	**switch**
auto	**else**	**operator**	**template**
break	**enum**	**private**	**this**
case	**extern**	**protected**	**try**
catch	**float**	**public**	**typedef**
char	**for**	**register**	**union**
class	**friend**	**return**	**unsigned**
const	**goto**	**short**	**virtual**
continue	**if**	**signed**	**void**
default	**inline**	**sizeof**	**volatile**
delete	**int**	**static**	**while**
do	**long**	**struct**	

2.3 A SIMPLE C++ PROGRAM

The best way to start learning a programming language is by writing a program. A simple C++ program has four sections and these are shown in following C++ program

Simple C++ Program:

```
#include <iostream.h>                    // Section: 1- The include Directive
using namespace std;                  // Section :2 - Class declaration and member
functions
int main ()                           // Section: 3 - Main function definition
{                                     // Section: 4 - Declaration of an object
  cout << "Hello World!";
  return 0;
}
```

Output:

Hello World!

This is one of the simplest programs that can be written in C+ programming language. It contains all the fundamental components which every C++ program can have. Line by line explanation of the codes of this program and its sections is given below:

Section: 1 – The include Directive

#include <iostream.h>

Lines beginning with a hash sign (#) are directives for the pre-processor. They are not regular code lines with expressions but indications for the compiler's pre-processor. In this case the directive #include <iostream> tells the pre-processor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

Section: 2 – Class declaration and member functions

using namespace std;

All the elements of the standard ANSI C++ library are declared within namespace std;. The syntax of this command is: using namespace std;. In order to access its functionality we declare all the entities inside namespace std;. This line is very often used in C++ programs that use the standard library and defines a scope for the identifiers that are used in a program.

Section: 3 - Main function definition

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions is these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within-them.

Section: 4 - Declaration of an object

Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

cout << "Hello World!";

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement is used to display output on the

screen of the computer. cout is the name of the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters.
cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs.

return 0;

The return statement causes the main function to finish.

☞ Check Your Progress 1

Fill in the appropriate words from following:

a) An int data type requires _____
   2 bytes
   4 bytes
   1 bytes
   8 bytes

b) iostream.h _____
   is a header file
   pre-processor directives
   user-defined function
   both a and b

c) cout in C++ is a _____
   object
   class
   function
   command

d) The standard C++ comment is _____
   /
   //
   /* and */
   None of the above

Short Answer type questions:

1) What do you mean by keyword in C++?

   ……

   ……

   ……

2) What do you mean by identifier in C++?

   ……

   ……

   ……

3) What do you mean by data types in C++?

……

……

……

4) What is the difference between variable and constant in C++ programming language?

……

……

……

2.4 SOME SIMPLE C+ + PROGRAMS

In this section, some basic C++ program are given. You practice it and may write some more program like these.

Program: 1

```
// Printing a message
#include <iostream.h>
int main(void)
{
  cout << "Hello, this is my first C++ program" << endl;
  return 0;
}
```

Output:

Hello, this is my first C++ program

Program: 2

```
// Printing name
#include <iostream.h>
 # include<conio.h>
 main()
{
char  name [15];
clrscr();
cout << "Enter your name:";
cin >> name;
cout<<"Your name is: " <<name;
return0;
}
```

Output:

Enter your name: Ram
Your name is: Ram

Program: 3

```
// operating with variables
#include <iostream.h>
using namespace std;
int main ()
{
  // declaring variables:
  int a, b;
  int result;
  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;
  // print out the result:
  cout << result;
  // terminate the program:
  return 0;
}
```

Output:

Result = 4

Program: 4

```
// initialization of variables
#include <iostream.h>
using namespace std;
int main ()
{
  int a=5;               // initial value = 5
  int b(2);              // initial value = 2
  int result;            // initial value undetermined
  a = a + 3;
  result = a - b;
  cout << result;
  return 0;
}
```

Output:

Result = 6

Program: 5

```
// my first string
#include <iostream.h>
#include <string>
using namespace std;
int main ()
{
  string mystring;
  mystring = "This is the initial string content";
  cout << mystring << endl;
  mystring = "This is a different string content";
  cout << mystring << endl;
  return 0;
}
```

Output:

This is the initial string content
This is a different string content

Program: 6

```
// defined constants: calculate circumference
#include <iostream.h>
using namespace std;
#define PI 3.14159
#define NEWLINE '\n'
int main ()
{
  double r = 5.0;               // radius
  double circle;
  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;
  return 0;
}
```

Output:

Circle = 31.4258714

Program: 7

```
#include <iostream.h>
 # include<conio.h>
 main()
{
int num, num1;
clrscr();
cout << "Enter two numbers:”;
cin >> name>>num1;
cout<<”Entered numbers are : “ ;
cout <<num<<”\t”<<num1;
return0;
}
```

Output:

```
Enter two numbers: 9, 15
Entered numbers are 9, 15
```

2.5 DIFFERENCE BETWEEN C AND C++

Following are some differences between C and C ++ :

- C++ is regarded as an intermediate-level language. It comprises a combination of both high-level and low-level language features. C++ is an extension to C Programming language. The difference between the two languages can be summarised as follows:
- The variable declaration in C, must occur at the top of the function block and it must be declared before any executable statement. In C++ variables can be declared anywhere in the program.
- In C++ we can change the scope of a variable by using scope resolution operator. There is no such facility in C language.
- C Language follows the top-down approach while C++ follows both top-down and bottom-up design approach.
- C is a procedure language and C++ is an object oriented language.
- C allows a maximum of 32 characters in an identifier name whereas C++ allows no limit on identifier length.
- C++ is an extension to C language and allows declaration of class, while C language does not allow this feature.
- C++ allows inheritance and polymorphism while C language does not.

2.6 DATA TYPES IN C++

In C++ programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them. The amount of memory required to store a single number is not the same as required by a single letter or a large number. Further, interpretation of different data is different inside computers memory.

The memory in computer system is organized in bits and bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer. In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Data Type in C++ is used to define the type of data that identifiers accepts in programming and operators are used to perform a special task such as addition, multiplication, subtraction, and division etc of two or more operands during programming.

C++ supports a large number of data types. The built in or basic data types supported by C++ are integer, floating point and character type. A brief discussion on these types is shown in Figure 2.3 which are shown below:

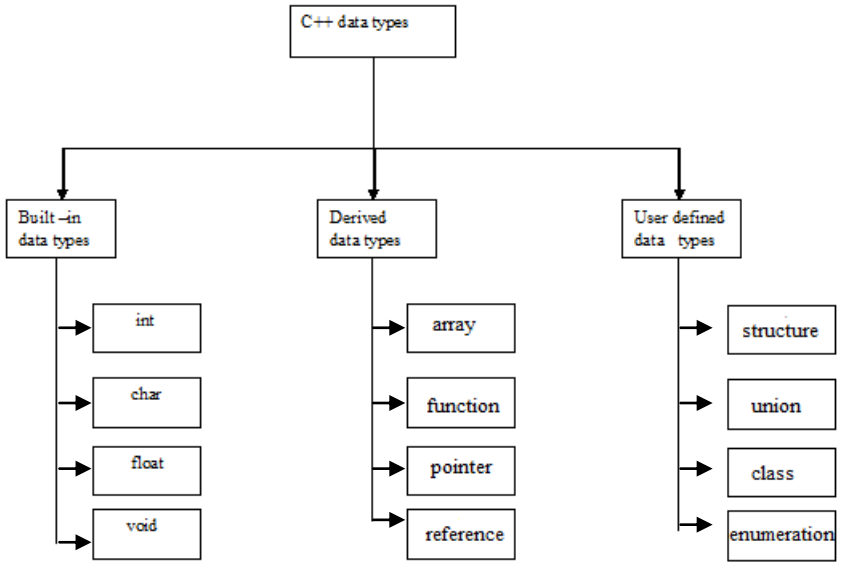


Figure 2.3 Hierarchy of C++ Data types

2.6.1 Built-in Data Types

There are four types of built-in data types as shown in the fig: 2. Let us discuss each of these and the range of values accepted by them one by one.

Integer Data type (int)

An integer is an integral whole number without a decimal point. These numbers are used for counting. For example 26, 373, -1729 are valid integers. Normally an integer can hold numbers from -32768 to 32767.

The int data type can be further categorized into following:

- Short
- Long
- Unsigned

The short int data type is used to store integer with a range of – 32768 to 32767, However, if the need be, a long integer (long int) can also be used to hold integers from -2, 147, 483, 648 to 2, 147, 483, 648. The unsigned int can have only positive integers and its range lies up to 65536.

Floating point data type (float)

A floating point number has a decimal point. Even if it has an integral value, it must include a decimal point at the end. These numbers are used for measuring quantities. Examples of valid floating point numbers are: 27.4, -92.7, and 40.03.

A float type data can be used to hold numbers from 3.4*10-38 to 3.4*10+38 with six or seven digits of precision. However, for more precision a double precision type (double) can be used to hold numbers from 1.7*10-308 to 1.7*10+308 with about 15 digits of precision.

Summary of Basic fundamental data types as well as the range of values accepted by each data type is shown in the following table.

Void data type

It is used for following purposes:

- It specifies the return type of a function when the function is not returning any value.
- It indicates an empty parameter list on a function when no arguments are passed.
- A void pointer can be assigned a pointer value of any basic data type.

Char data type

It is used to store character values in the identifier. Its size and range of values is given in Table 2.1.

Table: 1 Basic Fundamental Data Type

Name	Description	Size*	Range
Char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
Int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
Bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
Float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
Double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 or 4 bytes	1 wide character

Note: The values of the column Size and Range given in the table above, depends on the computer system on which the program is compiled. The values shown above are those found on 32-bit computer systems. But for other systems, the general specification is that int has the natural size suggested by the system architecture (one "word") and the four integer type's char, short, int and long must each one be at least as large as the one preceding it, with char being always one byte in size. The same applies to the floating point types float, double and long double, where each one must provide at least as much precision as the preceding one.

2.6.2 Derived Data types

C++ also permits four types of derived data types. As the name suggests, derived data types are basically derived from the built-in data types. There are four derived data types. These are:

- Array
- Function
- Pointer, and
- Reference

We will discuss these data types subsequently in this unit.

2.6.3 User Defined Data Types

C++ also permits four types of user defined data types. As the name suggests, user defined data types are defined by the programmers during the coding of software development. There are four user defined data types. These are:

- Structure
- Union
- Class, and
- Enumerator

We will discuss these data types in the later units of this course.

2.7 TYPE CONVERSION

In C++ object oriented language smaller memory data type variable can be converted to large data type by the compiler. It is required to make the language robust. When a variable of int type is multiplied by a variable of float type then the output is saved inside the computer system memory as double data type. Thus C++ permits mixed expressions. Type conversion can be done by following two ways:

a) Automatic

When an expression consists of more than one type of data elements in an expression, the C++ compiler converts the smaller data type element in larger data type element. This process is known as implicit or automatic conversion.

b) Typecasting

This statement allows the programmer to convert one data type into another data type by writing the following syntax:

aCharVar = static_cast<char>(an IntVar);

Here in the above syntax char variable will be converted into int Variable after execution of the syntax in the C++ program.

2.8 VARIABLES

A variable is the most fundamental aspect of any computer language. It is a location in the computer memory which can store data and is given a symbolic name for easy reference. The variables can be used to hold different values at different values at different times during the execution of a program.

To understand more clearly, let us take following example:

Total = 20.00 (i)
Net = Total - 12.00 (ii)

In equation (i), a value 20.00 has been stored in a memory location Total. The variable Total is used in statement (ii) for the calculation of another variable Net. The point worth noting is that the variable Total is used in statement (ii) by its name not by its value. Before a variable is used in a program, it has to be defined. This activity enables the compiler to make available the appropriate type of location in the memory. The definition of a variable consists of the type name followed by the name of the variable.

Declaration of variables:

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float, etc.) followed by a valid variable identifier. For example:

```
int a;
float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type int with the identifier a. The second one declares a variable of type float with the identifier mynumber. Once declared, the variables a and mynumber can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
int a,b,c;
```

This declares three variables (a, b and c), all of them of type int, and has exactly the same meaning as:

```
int a;
int b;
int c;
```

Similarly, a variable Total of type float can be declared as shown below:

```
float Total;
```

Similarly the variable Net can also be defined as shown below:

```
float Net;
```

Examples of some valid variable declarations are:

(i) int count;
(ii) int i, j, k;
(iii) char ch, first;
(iv) float total, Net;
(v) long int sal;

2.9 LITERALS OR CONSTANTS

A number which does not charge its value during execution of a program is known as a constant or literals. Any attempt to change the value of a constant will result in an error message. A keyword const is added to the declaration of an identifier to make that identifier constant. A constant in C++ can be of any of the basic data types. Let us consider the following C++ expression:

Once an identifier is declared as constant at the time of declaration, its value can't be changed during the execution of the program.

const float Pi = 3.1215;

The above declaration means that Pi is a constant of float types having a value: 3.1415.

Examples of some valid constant declarations are:

const int rate = 50;
const float Pi = 3.1415;
const char ch = 'A';
Scope of variables:

Let us now discuss scope of variables in C++ programming. A variable can be either of global or local scope. A global variable is a variable declared in the main body of the C++ source code, outside all the functions. Global variables can be called from anywhere in the code, even inside functions, whenever it is after its declaration.

The local variable is one declared within the body of a function or a block. To illustrate the scope of global variable and local variable, let us look at the figure 4. The scope of local variables is limited to the block enclosed in braces ({}) where they are declared. For example, if they are declared at the beginning of the body of a function (like in function main), their scope is between its declaration point and the end of that function.

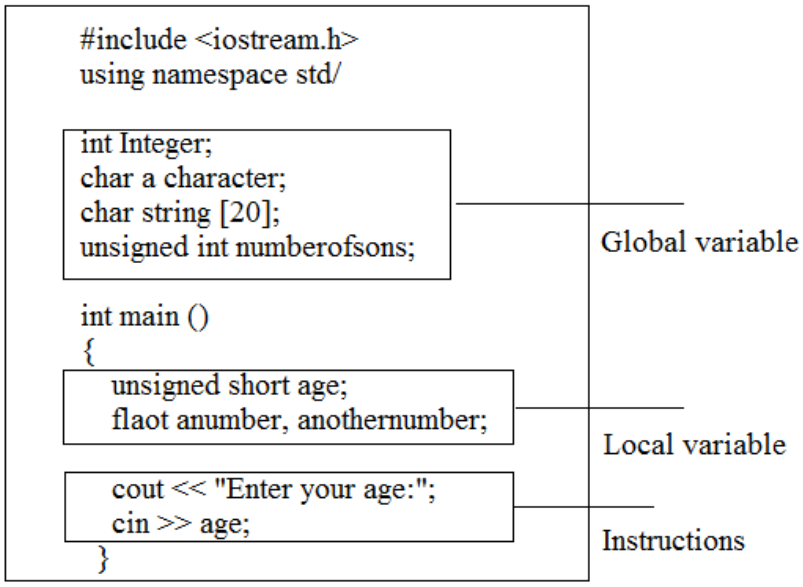


Figure 2.4: Scope of variables in C++ program

In the example above, this means that if another function existed in addition to main, the local variables, declared in main could not be accessed from the other function and *vice versa*.

☞ Check Your Progress 2

1) What do you mean by global variable and local variable in C++?

……

……

……

2) What do you mean string literals in C++ programming language?

……

……

……

3) Explain scope of a variable?

……

……

……

4) Explain the difference between C and C++?

……

……

……

2.10 OPERATORS IN C++

C++ has a rich set of operators. Operators are the term used to describe the action to be taken between two data operands. Expressions are made by combining operators between operands. C++ supports six types of operators:

- Arithmetical operators
- Relational operators
- Logical operators
- Bitwise operators
- Precedence of operators
- Special operators
- Escape sequence

2.10.1 Arithmetical operators

An operator that performs an arithmetic (numeric) operation such as +, -, *, /, or % is called arithmetic operator. Arithmetic operation requires two or more operands. Therefore these operators are called binary operators. The Table 2.2 shows the arithmetic operators:

Table 2.2:Operators Meaning with Example

Operator	Meaning	Example	Answer
+	addition	8+5	13
-	subtraction	8-5	3
*	multiplication	8*5	40
/	division	10/2	5
%	modulo	5%2	1

2.10.2 Relational operators

The relational operators shown in Table 2.3 are used to test the relation between two values. All relational operators are binary operators and therefore require two operands. A relational expression returns zero when the relation is false and a non-zero when it is true.

Table 2.3: Relational operators with Example

Operator	Meaning	Example
==	Equal to	5==5
!=	Not equal to	5!=7
>	Greater than	7>5
<	Less than	8<9
>=	Greater than or equal to	8>=8
<=	Less than or equal to	9<=9

2.10.3 Logical operators

The (!) operator is the C++ operator to perform the Boolean operation NOT. It has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. Logical operators of C++ are given in Table 2.4.

Table 2.4: Logical operators

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

To understand the use of these operators in C++, let us take following example:

Example:

```
!(5 == 5)    // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4)    // evaluates to true because (6 <= 4) would be false.
!true        // evaluates to false
!false       // evaluates to true.
```

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true and false otherwise. The Table 2.5 shows the result of operator && by evaluating the expression a && b:

Table 2.5: Use of && Operator

Operand (a)	Operand (b)	Result
true	True	True
true	False	False
false	True	False
false	False	False

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. To understand the use || OR operator, let us take the possible results of a || b in Table 2.6.

Table 2.6: Use of || Operator

Operand (a)	Operand (b)	Result (a || b)
True	True	True
True	False	True
False	True	True
False	False	False

Example:

((5 == 5) && (3 > 6)) // evaluates to false (true && false).
((5 == 5) || (3 > 6)) // evaluates to true (true || false).

2.10.4 Bitwise Operators

In C++ programming language, bitwise operators are used to modify the bits of the binary pattern of the variables. Table 2.7 gives use of some bitwise operators:

Table 2.7: Use of Bitwise Operator

operator	Asm equivalent	Description
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

2.10.5 Precedence of Operators

In case of several operators in an expression, we may have some doubt about which operand is evaluated first and which later. For example, let us take following expression:

a = 5 + 7 % 2
Here we may doubt if it really means:
A = 5 + (7 % 2) // with a result of 6, or
a = (5 + 7) % 2 // with a result of 0

The correct answer is the first of the two expressions, with a result of 6. Precedence order of some operators in C++ programming language is given in the Table 2.8.

Table 2.8: Precedence of operators in descending order

Level of Precedence	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	?:	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^= |=	assignment	Right-to-left
18	,	comma	Left-to-right

Grouping defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression. Thus if you want to write complicated expressions and you are not completely sure of the precedence levels, always include parentheses. It will also make your code easier to read.

2.10.6 Special Operators

Apart from the above operators that we have discussed above so far, C++ programming language supports some special operators. Some of them are: increment and decrement operator; size of operator; comma operator etc.

Increment and Decrement Operator

In C++ programming language, Increment and decrement operators can be used in two ways: they may either precede or follow the operand. The prefix version before the operand and postfix version comes after the operand. The two versions have the same effect on the operand, but they differ when they are applied in an expression. The prefix increment operator follows "change then use" rule and post fix operator follows "use then change" rule.

The size of operator

We know that different types of variables, constant, etc. require different amount of memory to store them. The sizeof operator can be used to find how many bytes are required for an object to store in memory.

Example:

sizeof (char) returns 1
sizeof (int) returns 2
sizeof (float) returns 4
if k is integer variable, the sizeof (k) returns 2.

The sizeof operator determines the amount of memory required for an object at compile time rather than at run time.

The comma operator

The comma operator gives left to right evaluation of expressions. It enables to put more than one expression separated by comma on a single line.

Example:

int i = 20, j = 25;

In the above statements, comma is used as a separator between the two statements.

2.10.7 Escape Sequence

There are some characters which can't be typed by keyboard in C++ programming language. These are called non-graphic characters. An escape sequence is represented by backslash (\) followed by one or more characters. The Table 2.9 gives a listing of common escape sequences.

Table 2.9: Escape Sequence

Sequence	Task
\a	Bell (beep)
\b	Backspace
\f	Formatted
\n	Newline or line feed
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\?	Question mark
\ \	Backslash
\ '	Single quote
\ "	Double quote
\ xhh	Hexadecimal number (hh represents the number in hexadecimal)
\ 000	Octal number (00 represents the number in octal)
\0	Null

Punctuators

In C++ programming language, following characters are used as punctuators for enhancing the readability and maintainability of programs.

Brackets [ ]	opening and closing brackets indicate single and multidimensional array subscript.
Parentheses ( )	opening and closing brackets indicate functions calls, function parameters for grouping expressions etc.
Braces { }	opening and closing braces indicate the start and end of a compound statement.
Comma ,	it is used as a separator in a function argument list.
Semicolon ;	it is used as a statement terminator.
Colon :	it indicates a labelled statement or conditional operator symbol.
Asterisk *	it is used in pointer declaration or as multiplication operator
Equal sign =	it is used as an assignment operator.
Pound sign #	it is used as pre-processor directive

2.11 CONTROL STRUCTURE IN C++

C++ program is usually not limited to a linear sequence of instructions but it may bifurcate, repeat code or may have to take decisions during the process of coding. For that purpose, C++ provides control structures which are used to control the flow of program.

Before we discuss control structures, let us first discuss a new concept: the compound-statement or block, which is very much needed to understand well the flow of control in a program.

A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }: for example:

```
{
statement1;
statement2;
statement3;
...
}
```

represents a compound statement or block.

In C++ object oriented programming, the control structure can be classified into following three categories:

Selection or conditional statement;
Iterating or looping statement;
Breaking statement;

Let us discuss the above control statement and their types in the following section.

2.11.1 Selection or conditional statement

In this type of statement, the execution of a block depends on the next condition. If the condition evaluates to true, then one set of statement is executed, otherwise another set of statements is executed. C++ provides following types of selection statements:

If;
If-else;
Nested if;
Switch
conditional

a) if statement:

The syntax of if statement is

```
If (expression)
        {
(Body of if)
Statements;
}
```

Where, expression is the condition that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

Program: 1

```
# include <iostream.h>
main()
{       int a, b;
a=10;
b=20;
if (a<b)
cout <<"a is less than b";
}
```

Output:

a is less than b.

Since here in the program value of a is less than the value of b, so the output of the program 1 is "a is less than b"

b) if-else statement:

The syntax of if –else statement is

```
If (expression)
        {
          (Body of if)
         Statements 1;
         }
        else
        {
          (Body of else}
           Statement 2
        }
```

Where, expression is the condition that is being evaluated. If this condition is true, statement - 1 is executed. If it is false, then if statement is skipped and the body of else statement is executed.

Program: 2

```
# include <iostream.h>
main()
{        int a, b;
a=10;
b=20;
if (a<b)
cout <<"a is less than b";
}
else
{
         cout<< "b is less than a"
}
```

Output:

a is less than b.

Since here in the program value of a is less than the value of b so the output of the program 1 is "a is less than b"

c) Switch statement:

Switch statement is used for multiple branch selection. The syntax of switch statement is

```
switch (expression)
        {
 case exp 1:
First case body;
Break;
case exp 2:
Second case body;
Break;
case exp 3:
Third case body;
Break;
        default:
                default case body;
}
```

Here, expression is the condition that is being evaluated. If the case 1 condition is true, First case body is executed, otherwise case exp 2 is checked and so on....If none of case expressions is true then the value of default case body is executed.

Program: 3

```
# include <iostream.h>
# include <conio.h>
 int main()
{
        clrscr();
                int d_o_w;
cout <<"Enter number of week's day (1-7)";
cin>>d_o_w;
        switch(d_o_w)
{

case 1: cout<<"/n Sunday";
break;
case 2: cout<<"/n Monday";
break;
case 3: cout<<"/n Tuesday";
break;
case 4: cout<<"/n Wednesday";
break;
case 5: cout<<"/n Thursday";
break;
case 6: cout<<"/n Friday";
break;
case 7: cout<<"/n Saturday";
break;
default: cout<<"/n Wrong number of day";
}
return 0;
}
```

Output:

Enter number of week's dat (1-7): 4

Wednesday

d) Nested if statement:

A nested if statement is a statement that has another if in its if's body or in its else's body. The syntax of switch statement is

```
if (expression 1)
                statement 1;
else if (expression 2)
statement 2;
else if (expression 3)
statement 3;
.
.
.
else
statement;
```

Here, expression is the condition that is being evaluated. If the case 1 condition is true, First case body is executed, otherwise it is skipped and next else expression is evaluated and so on.....

Program Segment: 4

```
# include <iostream.h>
# include <conio.h>
void main(void)
{
   float a,b,c,d:
         cout<< “Enter any four numbers \n”;
cin >>a >>b >>c >>d;
if (a > b) {
         if (a > c) {
                  if (a > d)
  cout << "largest = “ << a << end1;
  else
  cout << “largest = “ << d << end1;
   }
else
      {
         if (c > d)
                cout << " largest = “ << c << end1;
            else
                cout << “largest = “ << d << end1;
     }
  } // end of outer if part
else
  if (b > c) {
               if (b > d)
        cout << "largest = “ << b << end1;
else
        cout << largest = “ << d << end1;
              }
else {
          if (c > d)
            cout<< “largest = “ << c << end1;
        else
            cout << “largest = “ << d << end1;
       }
} // end of main program
```

Output:

Enter any four numbers
10 20 30 35

largest = 35

2.11.2 Iterative or looping statement

In C++ , programming language looping statement is used to repeat a set of instructions until certain condition is fulfilled. The iteration statements are also called loops or looping statement. C++ allows following four kinds of iterative loops:

- for loop
- while loop
- do-while loop and
- nested loops

a) for loop

This loop is easiest amongst all loops in C++ programming. The syntax of this loop is:

```
for (initialization; condition; increase)
{
        (body of the loop) statements;
}
```

This loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration

It works in the following way:

initialization is executed. Generally, it is an initial value setting for a counter variable. This is executed only once.
condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Program: 5

```
// Program by using a for loop
#include <iostream.h>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--)
          {
          cout << n << ", ";
          }
  cout << "FIRE!\n";
  return 0;
}
```

Output:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

b) while loop

If we do not know the number of iterations before starting the loop then while loop is used. Its syntax is as follows:

The functionality of this loop is simply to repeat statement while the condition set in expression is true.

```
initialization;
while (expression)
{
statement;
increment;
}
```

Program: 6

```
// Program by using while loop
#include <iostream.h>
using namespace std;
int main ()
{
  int n;
  cout << "Enter the starting number : ";
  cin >> n;
            while (n>0)
            {
            cout << n << ", ";
            -n;
            }
            cout << "FIRE!\n";
  return 0;
}
```

Output:

Enter the starting number: 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!

When execution of program starts, the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfils the condition n>0 (that n is greater than zero) the block that follows the condition will be executed and repeated while the condition (n>0) remains being true.

c) The do-while loop

Unlike for and while loops, the do-while is an exit-controlled loop i.e., it evaluates its test – expression at the bottom of the loop after executing its loop-body statement. This means that a do-while loop always executes at least once, even when the test – expression evaluates to false initially.

Its syntax is given as:

```
do
{
statement
}
while (test condition);
```

Program: 7

```
// number echoer
#include <iostream.h>
using namespace std;
int main ()
{
          unsigned long n;
        do
        {
                cout << "Enter number (0 to end): ";
                cin >> n;
                cout << "You entered: " << n << "\n";
        }
        while (n != 0);
 return 0;
}
```

Output:

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end.

d) Nested for loop

If a loop is placed inside the same loop then it is called nested for loop in C++ programming language. To understand, it let us take the following program:

Program: 8

```
// Program to print the pyramid of numbers by using a nested for loop
#include <iostream.h>
#include <conio.h>
#include <math.h>
void main ()
{
clrscr();
int n, i, j, k;
cout << "Enter the number of rows in the pyramid:";
cin>> n;
for (i=1; i<=n-i; i++)
{
        for (j=1; j<=n-i; j++)
        {
        cout <<  " ";
```

```
}
        for (k=1; k<+i; k++)
        {
        cout<< k;
        }
        cout << end;
}
getch();
}
```

Output:

```
Enter the number of rows in the pyramid: 5
     1
    12
   123
  1234
 12345
```

2.11.3 Breaking statement

Using break, we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. In this section, we will discuss following breaking statements:

- break statement
- continue statement
- goto statement and
- exit statement

a) break statement

The break statement is used to terminate the execution of the loop program. It terminates the loop in which it is written and transfers the control to the immediate next statement outside the loop. The break statement is normally used in the switch conditional statement. To understand, it let us take the following C++ program:

Program: 9

```
// break loop example
#include <iostream.h>
using namespace std;
int main ()
{
  int n;
  for (n=10; n>0; n--)
        {
        cout << n << ", ";
        if (n==3)
                {
```

```
                cout << "countdown aborted!";
                break;
                }
        }
 return 0;
}
```

Output:

10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

b) continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

Program: 10

```
// continue loop example
#include <iostream.h>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

Output:

10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!

c) goto statement

The goto statement is used to transfer control to some other parts of the program. It is used to alter the execution sequence of the program. To illustrate goto statement, let us take the following C++ program:

Program: 11

```
// goto loop example
#include <iostream.h>
using namespace std;
int main ()
{
  int n=10;
  loop:
```

```
cout << n << ", ";
  n--;
  if (n>0) goto loop;
  cout << "FIRE!\n";
  return 0;
}
```

Output:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

d) exit () statement

The exit statement is used to terminate the execution of the program. It is used when we want to stop the execution of the program depending on some condition. When we use exit () statement, we have to include other library functions such as process.h, or stdio.h file. To illustrate exit () statement, let us take the following C++ program:

Program: 12

```
// Program for exit statement
#include <iostream.h>
#include <conio.h>
void main ()
{

clrscr();
int i, number;
i = 1;
while(i<5)
{
 cout <<"Enter the number:";
cin>>number;
if number>5
{
cout <<"The number is greater than five or equal to" <<end1;
exit();
 }
 cout << "The number is: "<<number<<end1;
i++
}
getch();
}
```

Output:

Enter the number: 2
The number is: 2
Enter the number: 3
The number is: 3
Enter the number: 9
The number is greater than or equal to 9

2.12 I/O FORMATTING

In this section, we will discuss those functions which are used to format the Input and output of a C++ program. These functions are helpful in managing the I/O operations in C++ programming.

C + + supports input/output statements which can be used to feed new data into the computer or obtain output on an output device such as: VDU, printer etc. It provides both formatted and unformatted stream I/O statements. The following C + + streams can be used for the input/output purpose. In this section, we will discuss following I/O formatting functions:

Comments in C++
Unformatted Console I/O Functions
Setw I/O Formatting in C++
Inline Functions

Input/Output

Output is accomplished by using cout, which opens a "stream" to the standard output device (the screen). Data is inserted into the output stream using the << (insertion) operator. Input is accomplished by using cin, which opens a "stream" from the standard input device (keyboard). Data is retrieved from the stream by using the >> (extraction) operator.
Note: Every cin should be prefaced by a cout to prompt the user.

(i) Include <iostream.h>

The lines in the above program that start with symbol '#' are called directives and are instructions to the compiler. The word include with '#' tells the compiler to include the file iostream.h into the file of the above program. File iostream.h is a header file needed for input/output requirements of the program. Therefore, this file has been included at the top of the program.

(ii) void main ( )

The word main is a function name. The brackets ( ) with main tells that main ( ) is a function. The word void before main ( ) indicates that no value is being returned by the function main ( ). Every C++ program consists of one or more functions. However, when program is loaded in the memory, the control is handed over to function main ( ) and it is the first function to be executed.

(iii) The curly brackets and body of the function main ( )

Each, C ++ program starts with function called main ( ). The body of the function is enclosed between curly braces. These braces are equivalent to Pascal's BEGIN and END keywords. The program statements are written within the brackets. Each statement must end by a semicolon, without which an error message is generated.

2.12.1 Comments in C++

A comment is a statement in the program body to enhance the reading and understanding of the program. Comments are included in a program to make it more readable. If a comment is short and can be accommodated in a single line, then it is started with double slash sequence in the first line of the program. The syntax of short and one line comment is:

// Comment line...

However, if there are multiple lines in a comment, it is enclosed between the two symbols /* and */
Everything between /* and */ is ignored by the complier. The syntax of multiple line comment is

/* Start of multiple line comment
.....
.....
.....End of multiple line comment */

2.12.2 Unformatted Console I/O Functions

The I/O functions such as getch(), putchar(), get(), and put() etc. are called unformatted console I/O functions. The header file for these functions is <stdio.h> and should be included in the beginning of the program. The meaning and use of these functions can be illustrated as follows:

getch() This function is used to accept the input character which is typed by the keyboard during the execution of C++ program.

putchar() It displays the character on the screen at the current location of cursor.

Get (), and put () are the string functions in C++ programming language.

2.12.3 Setw - I/O Formatting in C++

In C++ programming language, stew () function is used to set the number of characters to be used as the field width for the next insertion operation. The field width determines the minimum number of characters to be written in some output representations.

This manipulator is declared in header <iomanip.h>, along with the other parameterized manipulators. This header file declares the implementation-specific requirement for setw (). To understand the use of sew () function in C++ programming; let us look at the following programs:

Program: 1

A program to insert a tab character between two variables by using setw ()

```
// setw example
#include <iostream.h>
#include <iomanip.h>
void main (void)
{
  int x, y, z;
  x = 400;
  y = 500;
  z = 600;
cout << x <<  '\t' << y << '\t' << z << end1;
}
```

Output:

400 500 600

Here in the above program 1 setw function has been used to insert a tab between three variables while displaying the content on the screen of computer.

2.12.4 Inline Functions

An inline function is written in one line when they are invoked. These functions are very short, and contain one or two statements. Inline functions are functions where the call is made to inline functions. The actual code then gets placed in the calling program.

Normally, a function call transfers the control from the calling program to the function and after the execution of the program returns the control back to the calling program after the function call. These concepts of function save program space and memory space and are used because the function is stored only in one place and is only executed when it is called. This execution may be time consuming since the registers and other processes must be saved before the function gets called.

The extra time needed and the process of saving is valid for larger functions. If the function is short, the programmer may wish to place the code of the function in the calling program in order for it to be executed. This type of function is best handled by the inline function.

The inline function takes the format as a normal function but when it is compiled it is compiled as inline code. The function is placed separately as inline function, thus adding readability to the source program. When the program is compiled, the code present in function body is replaced in the place of function call.

General Format of inline Function:

The general format of inline function is as follows:

```
inline datatype function_name(arguments)
```

The keyword inline specified in the above example, designates the function as inline function. For example, if a programmer wishes to have a function named exforsys with return value as integer and with no arguments as inline it is written as follows:
inline int exforsys ( )

Program 2

```
#include <iostream.h>
using namespace std;
int exforsys (into);
void main ( )
{
      int x;
      cout << "n Enter the Input Value: ";
      cin>>x;
      cout << "n The Output is: " << exforsys(x);
}

inline int exforsys(int x1)
{
      return 5*x1;
}
```

Output:

Enter the input value: 10
The output is> 50
Press any key to continue

2.12.5 Setprecision ()- I/O Formatting in C++

This function is used to control the number of digits of an output stream to be displayed on the screen in floating point value. This function is included in <iomanip.h> the header file and is included in the beginning of any C++ program. The syntax of this function is given as follows:

setprecision (int p);

To understand the use of this function, let us take the following C++ program:

Program 3

```
# include <iostream.h>
# include <iomanip.h>
void main (void)
{
float x,y,z;
x = 11;
y = 7;
z = x/y;
cout << setprecision(1) << z << end1;
cout << setprecision(2) << z << end1;
cout << setprecision(3) << z << end1;
cout << setprecision(4) << z << end1;
cout << setprecision(5) << z << end1;
}
```

Output:

1.6
1.57
1.571
1.5714
1.57142

2.12.6 Showpoint bit format flag- I/O Formatting in C++

This flag is used to show the decimal point for all floating point values. By default, it takes six decimal point values in C++ programming. The syntax of this flag is given as follows:

```
cout.setf(ios::shpowpoint);
```

To understand the use of this flag, let us take the following C++ program:

Program 4

```
# include <iostream.h>
void main ()
{
float w,x,y,z;
w = 2.34567845612
x = 11.34567653433;
y = 7.2345458765432;
z =  - 2345.5677225844;
cout.setf (ios::showpoint);
cout << "w = "  << w << "\n";
cout << "x = "  << x << "\n";
cout << "y = "  << y << "\n";
cout << "z = "  << z << "\n";
}
```

Output:

w = 2.345678
x= 11.345677
y = 7.234546
z = -2345.567723

2.12.7 Input and output stream flags - I/O Formatting in C++

To use many of the (I/O) manipulators, I/O streams have a flag field that specifies the current setting of decimal places and upper and lower case of alphabets in the output of a C++ program.

Flag Name	Meaning
skipws	skip white space during input
right	left justification of output
internal	pad after sign or base indicator
dec	decimal base
oct	octal base
hex	hexa decimal base
showbase	show base for octal and hexadecimal numbers
showpoint	show the decimal points for all floating numbers
uppercase	show upper case hex numbers
showpos	show '+' to positive numbers
scientific	use e for floating notations
fixed	use floating notations
unitbuf	flush all streams after insertions
stdio	flush out, stderr after insertion

Some of the I/O flag name and their meaning are given in the table above.

☞Check Your Progress 3

1) Explain the function of operators in C++?

……

……

……

2) Explain the term control structure in C++?

……

……

……

3) What do you mean by I/O formatting in C++ programming language?

……

……

……

4) Write a program in C++ to demonstrate the use of switch statement?

……

……

……

2.13 SUMMARY

In this unit you have learnt the features of object oriented programming, particularly that of C++ language. We have explained the C+ character set, tokens which include variables, constants and operators and data types used. In C++, the interchanging of data takes place automatically or by the programmer. The concept of input/output statement has been explained. The concept of comment statement which makes the program more readable is also given. Finally, we have also discussed control structure in C++ which is used to control execution sequence during the compilation and execution of program. At the end, you should be able to write a C++ program which will take input from the user, manipulate and print it on the screen by reading this unit.

2.14 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

Answers to fill in the blanks type questions

a) - 1, b) - 1 c) - 2 d) – 3

Answers to short answer type questions

1) There are some reserved words in C++ which have predefined meaning to complier called keywords. These are also known as reserved words and are always written or typed in lower cases.
2) A symbolic name is generally known as an identifier. The identifier is a sequence of characters taken from C++ character set. Valid identifiers are a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier.

3) Data Type in C++ is used to define the type of data that identifiers accepts in programming and operators are used to perform a special task such as addition, multiplication, subtraction, and division etc of two or more operands during programming.

4) A variable is the most fundamental aspect of any computer language. It is a location in the computer memory which can store data and is given a symbolic name for easy reference. The variables can be used to hold different values at different values at different times during the execution of a program.

   A number which does not *change* its value during execution of a program is known as a constant or literals. Any attempt to change the value of a constant will result in an error message. A keyword const is added to the declaration of an identifier to make that identifier constant. A constant in C++ can be of any of the basic data types.

Check Your Progress 2

1) A global variable is a variable declared in the main body of the C++ source code, outside all the functions. Global variables can be called from anywhere in the code, even inside functions, whenever it is after its declaration.

   The local variable is one declared within the body of a function or a block.

2) A string literal consists of zero or more characters from the source character set surrounded by double quotation marks ("). A string literal represents a sequence of characters that, taken together, form a null-terminated string.
   String literals may contain any graphic character from the source character set except the double quotation mark ("), backslash (\), or newline character. They may contain the same escape sequences supported by C++ language. C++ strings have these types:

   Array of char[n], where n is the length of the string (in characters) plus 1 for the terminating '\0' that marks the end of the string.
   Array of wchar_t, for wide-character strings.

3) Scope of a variable in C++ can be defined as: a variable can be either of global or local scope. A global variable is a variable declared in the main body of the C++ source code, outside all the functions. Global variables can be called from anywhere in the code, even inside functions, whenever it is after its declaration.

4) The difference between the two languages can be summarised as follows:

   The variable declaration in C must occur at the top of the function block and it must be declared before any executable statement. In C++ variables can be declared anywhere in the program.

   In C++ we can change the scope of a variable by using scope resolution operator. There is no such facility in C language.

C Language follows the top-down approach while C++ follows both top-down and bottom-up design approach.

C is a procedure language and C++ is an object oriented language.

C allows a maximum of 32 characters in an identifier name whereas C++ allows no limit on identifier length.

C++ is an extension to C language and allows declaration of class, while C language does not allow this feature.

Check Your Progress 3

1) C++ has a rich set of operators. Operators is the term used to describe the action to be taken between two data operands. Expressions are made by combining operators between operands. C++ supports six types of operators:

- Arithmetical operators
- Relational operators
- Logical operators
- Bitwise operators
- Precedence of operators
- Special operators

2) C++ program is usually not limited to a linear sequence of instructions but it may bifurcate, repeat code or may have to take decisions during the process of coding. For that purpose, C++ provides control structures which are used to control the flow of program.

In C++ object oriented programming, the control structure can be classified into following three categories:

- Selection or conditional statement;
- Iterating or looping statement;
- Breaking statement;

3) I/O functions are those functions which are used to format the Input and output of a C++ program. These functions are helpful in managing the I/O operations in C++ programming.

C + + supports input/output statements which can be used to feed new data into the computer or obtain output on an output device such as: VDU, printer etc. It provides both formatted and unformatted stream I/O statements. The following C + + streams can be used for the input/output purpose. C++ supports following I/O formatting functions:

Comments in C++
Unformatted Console I/O Functions
Setw I/O Formatting in C++
Inline Functions

```
1)  # include <iostream.h>
    # include <conio.h>
     int main()
    {
         clrscr();
                  int d_o_w;
    cout <<"Enter number of week's day (1-7)";
    cin>>d_o_w;
         switch(d_o_w)
    {
    case 1: cout<<"/n Sunday";
```

```
break;
case 2: cout<<"/n Monday";
break;
case 3: cout<<"/n Tuesday";
break;
case 4: cout<<"/n Wednesday";
break;
case 5: cout<<"/n Thursday";
break;
case 6: cout<<"/n Friday";
break;
case 7: cout<<"/n Saturday";
break;
default: cout<<"/n Wrong number of day";
}
return 0;
}
```

Output:

Enter number of week's dat (1-7) : 4

Wednesday

2.15 FURTHER READINGS AND REFERENCES

1) E Balagurusamy, *Object Oriented Programming with C++*, Tata McGraw-Hill Publishing Company Ltd, New Delhi , 2001.
2) Er V. K. Jain, *Object Oriented Programming with C++*, Cyber Tech Publication, Daryaganj N Delhi-110002
3) Robert Lafore, *Object Oriented Programming in C++*, Galgotia Publications Pvt. Ltd. Daryaganj N Delhi-11002
4) Rajesh K Shukla, *Object Oriented Programming in C++*, Wiley India Publishing Pvt. Ltd. Daryaganj, N delhi-110002
5) Bjarne AT&T Labs Murray Hill, New Jersey Stroustrup, Basics of C++ Programming, Special Edition, Publisher: Addison-Wesley Professional.
6) D Ravichandran, Programming with C++, Tata McGraw-Hill Publishing Company Ltd, New Delhi - 110008

Reference Websites:

(1) www.sciencedirect.com
(2) www.ieee.org
(3) www.webpedia.com
(4) www.microsoft.com
(5) www.freetechbooks.com
(6) www.computer basics.com
(7) www.youtube.com

UNIT 3 OBJECT AND CLASSES

Structure **Page Nos.**

3.0 INTRODUCTION

Objects and Classes are key to understand the Object-Oriented Programming Language (OOPL)/Object-Oriented Technique. Object-Oriented Programming Languages are based on the concept of abstraction modeled by classes and objects. OOPL is based on the concept of Object-Oriented technique. Object-Oriented technique (approach) for software development has become defacto standard for software development in software industry. Object-Oriented technique is a natural way of thinking or visualizing the real world problem in terms of the objects involved in the system. In Unit 1 of Block 1, we have learnt the basic concepts and characteristics of Object-Oriented technique.

In this Unit, we can begin with the concept of classification which is foundation in Object-Oriented programming. This is followed by what and how C++ supports classes, objects and how objects are used in problem solving. The classes are the crucial components for developing applications in C++. The accessibility rule that controls the visibility of class members (data and methods) is discussed. Moreover, member functions and friend functions are also discussed. This unit also covers a special kind of member function known as static function. Illustrative examples that facilitate you in understanding of the concept are presented with adequate emphasis.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- understand and define own classes;
- understand objects and use classes to create objects;
- use created objects;
- access members of a class;
- explain the need of friend function;
- describe the need of use static member; and
- write simple C++ program.

3.2 CLASSIFICATION

Classification is one of the important concepts of Object-Oriented philosophy including identity, abstraction, encapsulation, inheritance, polymorphism, and persistence. Let us take an example to understand the concept of classification. In the real world environment in which we operate, we need and identify hundreds or thousands of objects to solve the problem. It is very difficult to manage this large number of objects. Could you tell how to handle them easily, that too in such large numbers? Well, Object-Oriented uses classification to group objects that have attributes and behaviours in common and classify them into bigger entities. The bigger entity is called a 'class'. Thus a class defines a group of objects with similar attributes, common operations, a common relationship to the other objects in the class, and common semantics.

Suppose you have several objects like dog, cat, cow, elephant, car, airplane, fighter-plane, rocket, cup, mountain bike, racing bike, tandem bike. There are four animal objects grouped together in the animal class, three plane objects are grouped together into a plane class, and three bike objects group in bike class. The single car is in separate class as is a cup.

Can you tell how did we classify the objects? Yes, of course we can classify, based on the similar properties, common operations, a common relationship to the other objects in the class, and common semantics. But how will we classify the objects based on the concept of common semantics? The term common semantics in classifying objects is best described by an example. Consider two classes, Bus and House given as follows:

Name of the class	: Bus
Attributes	: Cost
	Colour
	Model
Services	: Maintenance

Name of the class	: House
Attributes	: Cost
	Colour
	Model
Services	: Maintenance

If you look at the above classes, you can observe that both have same set of attributes and service functions. Naturally, one can raise the questions. Whether both of these objects be considered of the same class or should they belong to different classes? These questions can be answered by looking at the semantics of two classes. If the underlying semantics is based on object usage, then two classes should not be combined even if their attributes and services are the same but if the underlying semantics is based on the object's asset, then both Bus and House belong to a common class Asset.

From the above discussion, we have understood the concept of classification and how it is used to reduce hundreds or thousands of objects into smaller groups/class, based on their characteristics.

3.3 CLASS

In this section, we will discuss what is class? What is importance of class in C++? We will define a class and also discuss encapsulation which is one of the striking features of a class.

A class is set of objects that shares a common definition described by data and methods.

According to the Webster's New World dictionary, a class is defined as " *A number of people or things grouped together because of certain likeness; kind; sort".* In other words, we can say that *class is an object template.* Every object under that class has the same data format, definition and responds in the same manner to an operation.

A class is a user defined data type like any other built-in data type for example int, char, float, .. etc. It is most important feature of C++. It makes C++ an Object-Oriented language. Can you tell the difference between structure and class ? Yes, of course, the only difference between a structure and a class in C++ is that *by default, the members of a class are private, while by default the members of a structure are public.* In the next section, we will see as to how we will define and use a class in the program.

3.3.1 Defining a Class

A class has a class name, a set of attributes (data members/characteristics) and a set of actions or services(function members/methods). Now, let us see how a class is defined in C++. The common syntax of a class declaration/specification/definition is given as follows:

```
class Class_name
{
        private :
                variable declaration;
                function declaration;
        public :
                variable declaration;
                function declaration;
        protected :
                variable declaration;
                function declaration;
};
```

It can be easily seen that a class is defined by the keyword class. The class specifies the type and scope of variables and functions declared inside the class. The variables(data) declared inside the class are known as data members and the functions(methods) are known as member functions. Being the part of the class, data and function are called members of the class. The body of the class is enclosed within braces and terminated by the colon. The keywords private, public and protected are known as visibility labels, which defines the visibility of members. We will discuss the visibility of members in the next section. It is common practice to declare data members as private and member functions as public. Let us see an example to understand how the class is defined.

Before defining a class, you should decide about the members of class i.e. who will be the members of a class. Actually, it depends on the problem domain i.e which type of data problems needs to keep in the class and also which type of operations will be needed to manipulate the data.

Based on the above discussion, suppose we want to define a Employee class. We are interested to display the basic information like ID, name, department etc. For this, first we have to decide the data members and their types required to represent the basic information, then we need a member functions to display basic information. We also need the member function to take information from the real world.

Let us see, how Employee class is defined.

```
class Employee
{
int id;
char name[25];
char deptt[25];
public:
void get_data(void)
{
        cout<< "Enter Employee ID:" <<endl;
        cin>>id;
        cout<<"Enter name:"<<endl;
        cin>>name;
        cout<< "Enter department:"<<endl;
        cin>>deptt;
}
void display_information(void)
{
cout<< "Employee ID=" <<id<<endl;
cout<< "Employee Name=" <<name<<endl;
cout<< "Employee Department=" <<deptt<<endl;
}
};
```

We, generally, give a class some meaningful name by reflecting information it holds. In the above declaration of the class, the name of the class is Employee. Now, Employee becomes a new data type. It is used to define instances of class data type.

3.3.2 Encapsulation

In this section, we will discuss about Encapsulation.

Encapsulation is one of the important characteristics of Object Oriented Programming Language. As we have discussed, a class can be described as a collection of data members and member functions. This property of C++ which allow wrapping up of data and functions into a single unit is called encapsulation. Data encapsulation is most striking feature of a class. Could you tell, what are the advantages of encapsulation? Well, the advantages of encapsulation are data hiding, information hiding and implementation independence. Let us see what these terms mean.

If the implementation details are not known to the user, it is called information hiding. Restrictions of external access to features of a class results in data hiding. The user's interface is not affected by changing the implementation mechanism. A change in the implementation is done easily without affecting the interface. This leads to implementation independence.

3.3.3 Accessibility Rules/Labels

You can see, in class declaration, we have used three terms private, public and protected. Can you tell what is the purpose of these terms in the program? Right, private, public and protected are known as visibility labels, used to control the access to members (data members and member functions) of a class.

Why do we need to control the access of members of a class? Well, as we know that purpose of data encapsulation is to prevent accidental modification of information of a class. Data can be protected from external tempering by users and access to specific methods can also be controlled. However, an entire program cannot be hidden. A part of the program needs to be accessed by users. For this and many other reasons, we need to access the members of a class in controlled way. It is achieved by imposing a set of rules-the manner in which a class is to be manipulated and data and functions of the class can be accessed. The set of rules is known as accessibility rules. This accessibility rules are achieved by using three keywords private, public and protected. These keywords are called access-control specifiers (visibility mode). The visibility of class members is summarized in Table 3.1.

Table 3.1: Visibility of class members

Access-control specifiers	Accessible to	
	Own class members	Objects of a class
private	yes	no
protected	yes	no
public	yes	yes

From the above table, you can observe that private and protected members of a class are accessible only from within other members of the same class. They are not accessible by the objects of a class. Further, public members are accessible both from members of the same class and objects of a class. They are accessible from anywhere where the object is visible. Members of a class without any access specifier are private by default. A class which is totally private is hidden from the external world and will not serve any useful purpose. Can you access the private member of a class from outside a class? Yes, we can have a mechanism to access private data using friends, pointers to members etc. from outside the class.

A class can use all of three visibility/accessibility labels as illustrated below:

```
Class A
{
private:
int x;
void fun1()
{
// This function can refer to data members x, y, z and functions fun1(), fun2() and
fun3()
}
protected:
int y;
void fun2()
{
//This function can also refer to data members x, y, z and functions fun1(), fun2()
and fun3()
}
public :
int z;
void fun3()
```

```
{
//This function can also refer to data members x, y, z and functions fun1(), fun2()
and fun3()
}
};
```

Now, consider the statements

A obja; //obja is an object of class A
int b; // b is an integer variable
The above statements define an object obja and an integer variable b. The accessibility of members of the class A is illustrated through the obja as follows:

1. Accessing private members of the class A:

b=obja.x; //Won't Work: object can not access private data member 'x'
obja.fun1(); //Won't Work: object can not access private member function fun1()
Both the statements are illegal because the private members of the class are not accessible.

2. Accessing protected members of the class A:

b=obja.y; // Won't Work: object can not access protected data member 'y'
obja.fun2(); // Won't Work: object can not access member function fun2()
Both the statements are also illegal because the protected members of the class are not accessible.

3. Accessing public members of the class A:

b=obja.c; //OK
obja.fun3(); //OK
Both the statements are valid because the public members of the class are accessible.

3.4 OBJECTS

In this section, we will study about object. What is object? What is the relationship between object and class? How will we create object? We shall make an attempt to all such address this question.

Webster's New World Dictionary defines object as *"A thing that can be seen or touched; material thing; a person or thing to which action, thought or feeling is directed".* In Object-Oriented domain an object is one of the many things that together constitute the problem domain. You can look around and find many real world examples of objects like person, customer, student, employee, car, dog, table, bike …etc. An object may stand alone or it may belong to a class of similar objects. Examples of objects that may stand alone are knife, frame … etc. Examples of objects that belong to a class of similar objects are: your car, your table … etc.

An object may have a name, a set of attributes, and a set of actions or services.

Can you tell the difference between object and class? Well, objects are the basic run-time entities in Object-Oriented programming language. They occupy space in memory that keeps its state and is operated on by the defined operations on the object, while a class defines a possible set of objects. What is the relationship between object and class? The relationship between a class and objects of that class is similar to the relationship between a type and elements of that type. *A class represents a set of objects that share a common structure and a common behavior, whereas an object is an instance of a class.* In the next section we will see as to how object is created.

3.4.1 Instantiating Object

The declaration of a class does not define any object but only specify the structure of objects i.e. what they will contain. A class must be instantiated in order to make use of the services provided by it. This process of creating objects (variables) of the class is called class instantiation or instantiating of objects. Thus, an object is an instantiation of a class. The common syntax of instantiating object of a class / declaration of a object is as follows:

class className objectName1, objectName2 …;
or
className objectName1, objectName2 …;

For example, let us see, how will we create the object of the Employee class discussed in class section?
class Employee emp1;
or
Employee emp1;

You can see that the declaration of an object is similar to that of any basic type. You can also notice that keyword class is optional. Employee class creates object emp1. More than one object can also be created with a single statement as follows.
Employee emp1, emp2,emp3;

Objects can also be created by placing their names immediately after the closing brace as we do in the case of structure. Thus, the definition
class Employee
{
.
.
.
}emp1,emp2,emp3;
would create objects emp1, emp2 and emp3 of the class Employee.

Now let us see, how will we assess the members of a class? We can assess the member of a class using the member assess operator, dot(.). The syntax for assessing data member of a class is given as follows:
objectName.dataMember;
The syntax for assessing member functions of a class is given as follows.
objectName.functionName(actualArguments);
Let us consider the snapshot of a program which illustrates use of dot (.) operator:

```
class ABC
{
        int x;
        int y;
        void f1(void);
        public:
        int z;
        void f2(void);
};
.
.
void main()
```

```
{
ABC a;

a.x=10;          //error, x is private data
a.z=10;          // OK, z is public data
.
.
a.f1();          //error, f1 is private function
a.f2();          //OK, f2 is public function
.
.
}
```

If you look at the above snapshot of C++ program, you can observe that private data like x and private member function like f1() cannot be accessed through object while public data like z and public member function like f2() are accessed through object. Furthermore, this program shows that you can access the members of a class through dot (.) operator.

Let us consider the complete program employee.cpp which illustrates the declaration of the class Employee with the operations on its object. Further, it is also seen that how we will access the members of a class. This program reads the basic information of employee like id, name, age etc. from the keyboard and display on the screen.

```
//program:employee.cpp
#include<iostream.h>
#include<string.h>
class Employee
{
int id;
int age;
char name[25];
public:
int salary;
void get_data(void)
{
cout<<"Enter ID :"<<endl;
cin>>id;
cout<<"Enter Name:"<<endl;
cin>>name;
cout<<"Enter Age:"<<endl;
cin>>age;
cout<<"Enter Salary :"<<endl;
cin>>salary;
}
void display_info(void)
{
cout<<"\nID     :" <<id<<endl;
cout<<"Name   :"<<name<<endl;
cout<<"Age    :"<<age<<endl;
cout<<"Salary :"<<salary<<endl;
}
};
```

```
void main()
{

Employee e1;                    // first object/variable of a Employee class
Employee e2;                    // second object/variable of a Employee class

cout<<"\nEnter 1st Employee Basic Information:"<<endl;
e1.get_data();                  // object e1 calls member get_data()
cout<<"\nEnter 2nd  Employee Basic Information:"<<endl;
e2.get_data();                  // object e2 calls member get_data()
cout<<"\n1st Employee Basic Information:"<<endl;
e1.display_info();              // object e1 calls member display_info()
cout<<"\n 2nd Employee Basic Information:"<<endl;

e2.display_info();              // object e2 calls member display_info()
}
```

The output of the above program is given below.

```
Enter 1st Employee Basic Information:
Enter ID:
100
Enter Name:
A.K. Malviya
Enter Age
39
Enter Salary
20000
Enter 2nd Employee Basic Information:
Enter ID:
101
Enter Name:
N. Badal
Enter Age
35
Enter Salary
22000

1st Employee Basic Information:
ID      : 100
Name    : A.K. Malviya
Age     : 39
Salary  : 20000
2nd Employee Basic Information:
ID      : 101
Name    : N. Badal
Age     : 35
Salary  : 22000
```

This program shows how we can access the members of a class with the help of the object. Furthermore, it also illustrates that the various operation on the objects of the class Employee. In main(), the statements Employee e1;
Employee e2;
create two objects called e1 and e2 of the employee class. The statements
e1.get_data();
e2.getdata();
initialize the data members of the objects e1 and e2. The statements
e1.display_info();
e2.display_info();
call their display_info() to display the contents of data members namely id, name, age and salary of the employee objects e1 and e2.

☞Check Your Progress 1

1) What are the differences between structures and classes in C++?

……

……

……

2) What is the concept of classification in Object-Oriented Programming Languages?

……

……

……

3) What are empty classes? Explain purpose of empty classes?

……

……

……

4) Write a C+ + program to find out the sum of n numbers.

……

……

……

3.5 MEMBER FUNCTIONS

In this section, we will discuss what is member function? How will we define a member function?

A member function performs an operation required by the class. It is the actual interface to initiate an action of an object belonging to a class. It may be used to read, manipulate, or display the data member. The data member of a class must be declared within the body of the class, while the member functions of a class can be defined in two places:

(i) inside the class definition
(ii) outside the class definition

The syntax of a member function definition changes depending on whether it is defined inside or outside the class declaration/definition. However, irrespective of the location of their definition, the member function must perform the same operation. Thus, the code inside the function body would be identical in both the cases. The compiler treats these two definitions in a different manner. Let us see, how we can define the member function inside the class definition.

The syntax for specifying a member function declaration is similar to a normal function definition except that is enclosed within the body of a class. Foe example, we could define the class as follows:

```
class Number
{
int x, y, z;
public:
void get_data(void);              //declaration
void maximum(void);               //declaration
void minimum(void)                //definition
{
int min;
min=x;
if (min>y)
min=y;
if (min>z)
min=z;
cout<<"\n Minimum value is ="<<min<<endl;
}
};
```

if you look at the above declaration of class number you can observe that the member function get_data() and maiximun() are declared, but they are not defined. The only member function which is defined in the class body is minimum(). When a function is defined inside a class, it is treated as a inline function. Thus, member function minimum is a inline function. Generally, only small functions are defined inside the class.

Now let us see how we can define the function outside the class body. Member functions that are declared inside a class have to be defined outside the class. Their definition is very much like the normal function. Can you tell how does a compiler know to which class outside defined function belong? Yes, there should be a mechanism of binding the functions to the class to which they belong. This is done by the scope resolution operator (**::**). It acts as an identity-label. This label tells the compiler which class the function belongs to. The common syntax for member function definition outside the class is as follows:

```
return_type class_name :: function_name(argument declaration)
{
functionbody
}
```

The scope resolution :: tells the compiler that the function_name belongs to the class class_name. Let us again consider the class Number.

```
class Number
{
int x, y, z;

public:
void get_data(void);              //declaration
```

```
void maximum(void);              //declaration.
.
.
};
void Number :: get_data(void)
{
cout<< "\n Enter the value of fist number(x):"<<endl;
cin>>x;
cout<< "\n Enter the value of second number(y):"<<endl;
cin>>y;
cout<< "\n Enter the value of third number(z):"<<endl;
cin>>z;
}
void Number :: maximum(void)
{
int max;
max=x;
if (max<y)
max=y;
if (max<z)
max=z;
cout<<"\n Maximun value is ="<<max<<endl;
}
```

if you look at the above declaration of class Number, you can easily see that the member function get_data() and maiximun() are declared in the class. Thus, it is necessary that you have to define this function. You can also observe in the above snapshot of C++ program identity label (::) which are used in void Number :: get_data(void) and void Number ::maximum(void) tell the compiler the functionget_data() and maximum() belong to the class Number.

Now, let us see the complete C++ program to find out the minimum and maximum of three given integer numbers:

```
#include<iostream.h>
class Number
{
int x, y, z;

public:
void get_data(void);             //declaration
void maximum(void);              //declaration
void minimum(void)               //definition
{
int min;
min=x;
if (min>y)
min=y;
if (min>z)
min=z;
cout<<"\n Minimum value is ="<<min<<endl;
}
};
void Number :: get_data(void)
{
```

```
cout<< "\n Enter the value of fist number(x):"<<endl;
cin>>x;
cout<< "\n Enter the value of second number(y):"<<endl;
cin>>y;
cout<< "\n Enter the value of third number(z):"<<endl;
cin>>z;
}
void Number :: maximum(void)
{
int max;
max=x;
if (max<y)
max=y;
if (max<z)
max=z;
cout<<"\n Maximun value is ="<<max<<endl;
}

void main()
{
Number num;

num.get_data();
num.minimum();
num.maximum();
}
```

The output of the above program is given as follows:

Enter the value of the first number (x):
10
Enter the value of the second number (y):
20
Enter the value of the third number (z):
5
Minimum value is=5
Maximum value is=20

3.5.1 Nesting of Member Functions

In this section, we will discuss the nesting of member functions.

A member function of a class can call any other member function of its own class irrespective of its privilege. This is known as nesting of member functions.

Consider the problem of finding the largest of n given number which illustrates the above concept.

```
#include<iostream.h>
#define MAX_SIZE 100

class Data
{
int num[MAX_SIZE];
int n;
```

```
public:
void get_data(void);              //declaration
int largest(void);                //declaration
void display(void);               //declaration
};
void Data :: get_data(void)
{
cout<< "\n Enter the total numbers(n):"<<endl;
cin>>n;
cout<< "\n Enter the number:"<<endl;
for (int i=0;i<n; i++)
{
cout<< "\n Enter the numer"<<i+1<<": ";
cin>>num[i];
}
}
int Data :: largest(void)
{
int max;
max=num[0];
for(int i=1; i<n; i++)
{
if (max<num[i])
max=num[i];
}
return max;
}
void Data :: display(void)
{
cout<<"The largest number:"<<largest()<<endl;
}
void main()
{
Data num;

num.get_data();
num.display();
}
```

The class Data has the member function display having the statement cout<<"The largest number:"<<largest()<<endl;. It calls the member function largest () to compute the largest of n given numbers.

3.5.2 Passing Objects as Arguments

We can pass objects as arguments to a function like any other data type. This can be done by a pass-by-value and a pass-by-reference. In pass-by-value, a copy of the object is passed to the function and any modifications made to the object inside the function are not reflected in the object used to call the function. While, in pass-by-reference, an address of the object is passed to the function and any changes made to the object inside the function is reflected in the actual object. Furthermore, we can also return object from the function like any other data type.

Consider the following C++ program for addition and multiplication of two square matrices which illustrates the above concepts.

```
#include<iostream.h>
#define MAX_SIZE 10

int n;
class Matrix
{
int item[MAX_SIZE][MAX_SIZE];

public:
void get_matrix(void);
void display_matrix(void);
Matrix add(Matrix m);// Matrix object as argumentand as return: pass by value
void mul(Matrix &mat, Matrix m);// Matrix object as argument: pass by
reference and pass by value
};
void Matrix :: get_matrix(void)
{
cout<< "\n Enter the order of square matrix(nXn):"<<endl;
cin>>n;
cout<< "\n Enter the element of matrix:"<<endl;
for (int i=0;i<n; i++)
for (int j=0;j<n; j++)
cin>>item[i][j];
}
void Matrix :: display_matrix(void)
{
cout<<"\n The element of matrix is :"<<endl;
for (int i=0;i<n; i++)
{
for (int j=0;j<n; j++)
cout<<item[i][j]<<"\t";
cout<<endl;
}
}
Matrix Matrix :: add(Matrix m)
{
Matrix temp;     // object temp of Matrix class

for (int i=0;i<n; i++)
for (int j=0;j<n; j++)
temp.item[i][j]=item[i][j]+m.item[i][j];
return (temp);    // returm matrix object
}
void Matrix :: mul(Matrix &rm, Matrix m)
{
for (int i=0;i<n; i++)
for (int j=0;j<n; j++)
{
rm.item[i][j]=0;
for(int k=0; k<n; k++)
rm.item[i][j]=rm.item[i][j]+item[i][k]*m.item[k][j];
}
}
void main()
{
```

```
Matrix X, Y, Result;
cout<<"Matrix X :"<<endl;
X.get_matrix();
cout<<"Matrix Y :"<<endl;
Y.get_matrix();
cout<<"\n Addition of X & Y :"<<endl;
Result=X.add(Y);
Result.display_matrix();
cout<<"\n Multiplication of X & Y :"<<endl;
X.mul(Result,Y);          //result=X*Y
Result.display_matrix();
}
```

If you look at the above program, you can observe that in main(), the statement
Result=X.add(Y);
in which X.add(Y) invokes the member function add() of the class matrix by the object X with the object Y as arguments. The members of Y can be accessed only by using the dot operator (like m.item[i][j]) within the add() member. Any modification made to the data members of the object Y is not visible to the caller's actual parameter. The data members of X are accessed without the use of the class member access operator. The statement in the function add(),
return(temp);
returns the object temp as a return object. The result has become the return object temp which stores the sum of X and Y. This illustrates that function also return object.
Further, in main(), the statement
X.mul(Result, Y);
transfers the object result by reference and Y by value to the member function mul(). When mul() is invoked with result and Y the objects parameters, the data members of X are accessed without the use of the class member access operator, while the data members of result and Y are accessed by using their names in association with the name of the object to which they belong. Modifications which are carried out on result object in the called function will also be reflected in the calling function.

☞ Check Your Progress 2

1) What is the scope resolution operator?

……

……

……

2) When would we define a member function inside or outside of the class?

……

……

……

3) What is the purpose of the member function?

……

……

……

4) Define a class to represent a bank account. Include the following members:

Data Members:

a. Name of the depositor
b. Account Number
c. Type of Account
d. Balance amount in the account

……

……

……

Member Functions:

a. To assign initial value
b. To deposit an amount
c. To withdraw an amount after checking the balance
d. To display name and balance and account Number

……

……

……

5) Write a interactive program in C++ for the above problem. Assume a bank has maintained two types of account: Saving account and Current account. You should open a saving account with minimum Rs 500 and also keep minimum balance of Rs. 500 and current account with Rs. 5000 and also keep minimum balance of Rs. 5000.

……

……

……

3.6 FRIEND FUNCTION

As we have discussed that the private members cannot be accessed from outside the class. It implies that a non-member function cannot have an access to the private data of a class. Let us suppose, we want a function operate on objects of two different classes. In such situations, C++ provides the friend function which is used to access the private members of a class. Friend function is not a member of any class. So, it is defined without scope resolution operator. The syntax of declaring friend function is given below:

```
class class_name
{
…
…
public:
…
…
friend return type function_name(arguments);
}
```

Let us consider the complete C++ program to find out sum of n given numbers to understand the concept of friend function.

```
#include<iostream.h>
#define MAX_SIZE 100
class Sum
{
int num[MAX_SIZE];
int n;
public:
void get_number(void);
friend int add(void);
};
void Sum :: get_number(void)
{
cout<< "\n Enter the total number(n):"<<endl;
cin>>n;
cout<< "\n Enter the number:"<<endl;
for (int i=0;i<n; i++)
cin>>num[i];
}
int add(void)
{
Sum s;
int temp=0;
s.get_number();
for (int i=0;i<s.n; i++)
temp+=s.num[i];
return temp;
}

void main()
{
int res;
res=add();
cout<<"The sum of n value is="<<res<<endl;
}
```

If you look at the above program, you can easily see that the function add is declared as a friend function of class Sum. The add function accesses the private data, adds the numbers of array and returns value to the main function where it is called upon. Furthermore, you can also see that friend function add() is defined without scope resolution operator(::), because it does not belong to a class.

Now, let us consider a situation in which we want to operate on objects of two different classes. In such a situation, friend functions can be used to bridge the two classes.

```
#include<iostream.h>
class Two;        //forward declaration like function prototype
class One
{
int a;
public:
void get_a(void);
```

```
friend int min(One, Two);
};
class Two
{
int b;
public:
void get_b(void);
friend int min(One, Two);
};
void One :: get_a(void)
{
cout<<"Enter the value of a:"<<endl;
cin>>a;
}
void Two :: get_b(void)
{
cout<<"Enter the value of b:"<<endl;
cin>>b;
}
int min (One o, Two t)
{
if(o.a<t.b)
return o.a;
else
return t.b;
}
void main()
{
One one;
Two two;
int minvalue;

one.get_a();
two.get_b();
minvalue=min(one,two);
cout<<"Minimum="<<minvalue<<endl;
}
```

You can observe that the above program contains two classes named one and two. The function min() is declared in the both the classes with the keyword friend. An object of each class has been passed as an argument to the function min (). Being a friend function, it can access the private members of both classes through these arguments. Now, let us note some special properties possessed by friend function:

(i) A friend function is not in the scope of the class to which it has been declared as friend.
(ii) A friend function cannot be called using the object of that class. It can be invoked like a normal function without the use of any object.
(iii) Unlike member functions, it can not access the members directly. However, it can use the object and dot membership operator with each member name to access both private and public members.
(iv) It can be declared either in the public or the private part of a class without affecting its meaning.
(v) Generally, it has got objects as arguments.

3.7 STATIC MEMBERS

In this section, we will discuss the static members.

The members of a class can be made static (data member and function member both). Can you tell, what is static? Well, static is a storage class specifier/qualifier that provides information about locality and visibility of variables. Let us discuss the static data member. Sometimes, we need to have one or more common data member, which are accessible to all objects of a class. For example, we need to keep the status as to how many objects of a class are created and how many of them are currently active in the program. In such situation, and many others, C++ provides static data member. Static data member is initialized to zero when the first object of its class is created. No other initialization is permitted.

A variable that is part of a class, yet is not the part of an object of that class, is called a static data member. There is exactly one copy of static data member instead of one copy per object, as for ordinary non-static data members. Similarly, a function that needs access to members of a class, yet doesn't need to be invoked for a particular object is called a static member function. The common syntax of defining static data member is given as follows:

```
class class_name
{
…..
….
static data_type data member;
….
….
};
data_type class_name :: data member=initial_value;
```

Static data member must be defined outside the class. Initialization of static data member is optional. Let us consider the following C++ program to understand the concept of static data members.

```
#include<iostream.h>
class Counter
{
static int count;  // static member: count is static
static int n;        // static member: number is static
public:
void get_data(int num)   // initializes object's member
{
n=num;
count++;
}
void show_count(void)
{
cout<<"Number of times calls made to function 'get_data()' through any object:";
cout<<count<<endl;
cout<<"n:"<<n<<endl;
}
};
int Counter :: count=0;   // definition and initialization(optional) of a static data member
```

```
int Counter :: n;              // definition of static data member
void main()
{
Counter x,y,z;
x.show_count();
y.show_count();
z.show_count();

x.get_data(25);
y.get_data(50);
z.get_data(75);
cout<<"After reading data :"<<endl;
x.show_count();
y.show_count();
z.show_count();
}
```

The above program shows the concept of static data member. There are two static variables **count** and **n**. The variables **count** and **n** are initialized zero when their object is created. The static data member **count** is incremented and data **n** is assigned with parameter value. Whenever the get_data() function is called. Since get_data() member function is called three times by the object **x**, **y**, and **z**, the data member count is incremented three times. Furthermore, the variable n is assigned with values 25, 50 and 75 respectively by each function call get_data(). All the statements print the value of **count** as 3 and **n** as a 75 because there is only one copy of **count** and one copy of **n** which are shared by all three objects.

Now, let us discuss about static member function. Like static data member variable, we can also have a static member function. The static function can only access the static member(data or function) declared in the same class. There is one important difference between static member function and member function. A static member function is called using the class name instead of its object. The following program illustrates the concept of the static member function:

```
#include<iostream.h>
class Counter
{
static int count;  // static member: count is static
 int n;    // static member: number is static
public:
void set_data(void)        // initializes object's member
{
count++;
n=count;
}
void show_value_of_n(void)
{
cout<< "The value of n is ="<<n<<endl;
}
void static show_count(void)
{
cout<<"Count :"<<count<<endl;
}
};
int Counter :: count=0;   // definition and initialization(optional) of a data
member
```

```
void main()
{
Counter c1,c2;
Counter::show_count();
c1.set_data();
c2.set_data();
Counter::show_count();
Counter c3;
c3.set_data();
Counter::show_count();
c1.show_value_of_n();
c2.show_value_of_n();
c3.show_value_of_n();
}
```

If you look at the above program, you can easily see that the static function show count() displays the number of objects created till that moment. A count on the number of objects is maintained by static variable count. The function show_value() displays the value of the non-static variable n.

☞ Check Your Progress 3

1) What is friend class? Write a program to illustrate the concept of friend class.

…..

……

……

2) Why is friend function needed?

……

……

……

3) Discuss memory requirements for classes, objects, data members and member functions.

……

……

……

4) Bring out the differences between the memory requirements of static data members and non-static data members.

……

……

……

3.8 SUMMARY

In this unit, we have seen and discussed the concept of class as well as object which are the basic components used in C++ programming. Class declaration and object creation have been discussed and illustrated with examples. The members viz., data members and function members of a class, defining member functions were explained and elaborated. We have seen the way to pass objects as arguments to the functions with call by value and call by reference. Furthermore, static members, Friend function and Friend class are also discussed with examples.

3.9 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1. Structures and classes in C++ are given same set of features. A class in C++ is identical to structure in C++. In C++, the difference between structures and classes is that, by default, structure members have public accessibility while class members have private access control unless otherwise explicitly stated.

2. Classification is a concept/technique which is used to make group of objects or partition objects by some logic and classify them into bigger entities i.e. class.

3. Though the main reason for using a class is to encapsulate data and code. It is, however, possible to have a class that has neither data nor code. In other words, it is possible to have empty classes. The declaration of empty classes is as follows:

```
    class ABC{};
    class Employee{};
class xyz
{
};
```

During the initial stage of development of a project, some of the class are either not fully identified, or not fully implemented. In such cases, they are implemented as empty classes.

4.

```
#include<iostream.h>
#define MAX_SIZE 25
class Sum
{
int number[MAX_SIZE];
int n;
int total;
public:

void get_data(void)
{
int i;
cout<<"Enter Total Number :"<<endl;
cin>>n;
cout<<"Enter Number One by One:"<<endl;
for(i=0; i<n; i++)
{
cout<<"Enter Number"<<i+1<<":"<<endl;
cin>>number[i];
```

```
}
}
void cal_sum(void)
{
total=0;
int i;
for (i=0; i<n; i++)
total=total+number[i];
}
void display_sum(void)
{
cout<<"\nSum :"<<total<<endl;
}
};
void main()
{
Sum s;
s.get_data();
s.cal_sum();
s.display_sum();
}
```

Check Your Progress 2

1. The scope resolution operator (::) especially defined in C++. They are used in two ways: (1) for resolving the class member function (2) for resolving the global variable.

2. If your function is very small and make this inline, then you can define your function in the class because all defined function within class are, by default, inline functions. If you are using much control statements and looping, you have to define your function outside the class.

3. A member function performs an operation required by the class. It is the actual interface to initiate an action of an object belonging to a class. It may be used to read, manipulate, or display the data member. The data member of a class must be declared within the body of the class, while the member functions of a class can be defined in two places:
   (i) inside the class definition
   (ii) outside the class definition

4.

```
#include<iostream.h>
#include<stdlib.h>
#define MAX_SIZE 25

// unit3e3.cpp

class Account
{
char depositor_name[25];
int account_no;
int type_of_account;
int balance;
public:
```

```
void assign_initial_value(void);
void deposit(void);
void withdraw(void);
void account_info(void);
};
void Account :: assign_initial_value(void)
{
cout<< "\n Enter the depositor name:"<<endl;
cin>>depositor_name;
cout<< "\n Enter the account no.:"<<endl;
cin>>account_no;
cout<< "\n Enter the type of account(1-for saving, 2-for current):"<<endl;
cin>>type_of_account;
cout<< "\n Enter the amount to be deposited(For saving min:500 & current min:5000"<<endl;
cin>>balance;
}
void Account :: deposit(void)
{
int da;
cout<< "\n Enter the amount to be deposited:"<<endl;
cin>>da;
balance=balance+da;
}
void Account :: withdraw(void)
{
int aw;
int d;

cout<< "\n Enter the amount to be withdraw:"<<endl;
cin>>aw;
d=balance-aw;
if((type_of_account==1) && (d>=500))
{
balance=balance-aw;
cout<<"\n Withdraw Successful:"<<endl;
}
if((type_of_account==1) && (d<500))
{
cout<<"\n Withdraw UnSuccessful, check your balance:"<<endl;
}
if((type_of_account==2) && (d>=5000))
{
balance=balance-aw;
cout<<"\n Withdraw Successful:"<<endl;
}
if((type_of_account==2) && (d<5000))
{
cout<<"\n Withdraw UnSuccessful, check your balance:"<<endl;
}
}
void Account :: account_info(void)
{

cout<< "\n depositor name:"<<depositor_name<<endl;
cout<< "\n Account no.:"<<account_no<<endl;
cout<< "\n Balance:"<<balance<<endl;
```

```
}
int main()
{
Account a;
int choice;
while(1)
{
cout<< "\n Account operation:..."<<endl;
cout<< "\n 1. Assign initial value";
cout<< "\n2. Deposit Ammount";
cout<< "\n3. Withdraw Amount";
cout<< "\n4. Balance Enquiry";
cout<< "\n5. Quit";
cout<< "\n Enter choice :"<<endl;
cin>>choice;
switch(choice)
{
case 1: a.assign_initial_value();
break;
case 2: a.deposit();
break;
case 3: a.withdraw();
break;
case 4: a.account_info();
break;
case 5: exit(1);
break;
default: cout<<"Bad option selected";
 break;
}
}
}
```

Check Your Progress 3

1. We can define a class as friend of another class, granting that second class access to the protected and private members of the first one. A class declared as a friend of another class, it possesses the rights of access to the private member of this class using objects.

```
#include<iostream.h>
class X
{
int x;
public:
void get_x(void)
{
cout <<" Enter the value of x:"<<endl;
cin>>x;
}
friend class Y;
};
class Y
{
int y;
public:
```

```
void get_y(void)
{
cout <<" Enter the value of y:"<<endl;
cin>>y;
}
void add(void)
{
X objx;
objx.get_x();
get_y();
y=y+objx.x;
}
void display()
{
cout<<"The Sum="<<y;
}
};

void main()
{
Y objy;
objy.add();
objy.display();
}
```

2. There are the following situations in which the friend function needed.
   - Function operating on objects of two different classes.
   - Friend functions can be used to increase the versatility of overloaded operators.
   - Sometimes, a friend allows a more obvious syntax for calling a function rather than what a member function can do.

3. When a class is declared, memory is not allocated to data members but allocated to only member functions. When an object of a particular class is created, memory is allocated only to its data members. Thus, all objects of that class have access to the same area in the memory where the member functions are stored. It is logically also true as the member functions are same for all objects and there is no need to allocate a separate copy for each and every object created. However, storage space for data member is allocated for every object's data members. This is essential because data member will hold different data values for different objects.

4. Whenever a class is instantiated, the memory is allocated for the created objects. Memory space for static data members is allocated only once during class declaration while memory space of on-static data members is allocated when objects of a class are created. Therefore, all objects of the class have access the same memory area allocated to static data members. When one of them modifies the static data member, the effect is visible to all the instance of the class i.e. objects.

3.10 FURTHER READINGS

1) B. Stroustrup, *The C++ Programming Language*, third edition, Pearson/Addison-wesley Publication,1997.

2) K. R. Venu Gopal, Raj Kumar Buyya, T Ravishankar, *Mastering C++*, Tata-McGraw-Hill Publishing Company Limited, New Delhi.

3) E. Balagurusamy, *Object Oriented Programming with C++*, Tata Mc-Graw-Hill Publishing Company Limited, New Delhi.

4) N. Barkakati, *Object Oriented Programming in C++*, Prentice-Hall of India.

UNIT 4 CONSTRUCTORS AND DESTRUCTORS

Structure **Page Nos.**

4.0 INTRODUCTION

In the previous unit, we have discussed concept of Objects and Class and their use in object oriented programming. In this unit we shall discuss about constructors and destructors and their use in memory management for C++ programming.

C++ allows different categories of data types, that is, built-in data types (e.g., int, float, etc.) and user defined data types (e.g., class). We can initialize and destroy the variable of user defined data type (class), by using constructor and destructor in object oriented programming.

Constructor is used to create the object in object oriented programming language while destructor is used to destroy the object. The constructor is a function whose name is same as the object with no return type. The name of the destructor is the name of the class preceded by tilde **(~).** First we will discuss briefly about constructor and destructor and then move on to the types of constructor and memory management.

Whenever an object is created, the special member function, that is, constructor will be executed automatically. Constructors are used to allocate the memory for the newly created object and they can be overloaded so that different form of initialization can be accommodated. If a class has constructor, each object of that class will be initialized. It is called constructor because it constructs the value of data members of the class.

Let us take an example to illustrate the syntax and declaration of constructor and destructor, inside a class in object oriented programming.

Example: A constructor is declared and defined as follows:

```
// class with constructor

        class integer
        {
                int m, n;
            public:
                integer (void);             // constructor declared
                ……….
                ……….
        };

        integer :: integer (void)           // constructor defined
        {
                m = 0, n = 0;
        }
```

In the above example class integer contains a constructor, and the object created by the class is initialized automatically.

4.1 OBJECTIVES

After studying this unit, you should be able to:

- explain the basic concepts of constructor and destructor;
- describe purpose of Constructor and Destructor;
- explain types of constructor and destructor;
- use Automatic Initialization, and
- memory management by using constructor and destructor in programming.

4.2 CONSTRUCTOR AND DESTRUCTOR

C++ is an object oriented programming (OOP) language which provides a special member function called constructor for initializing an object when it is created. This is known as automatic initialization of objects. It also provides another member function called destructor which is used to destroy the objects when it is no longer required. Constructor has the same name as that of the class's name and its main job is to initialize the value of the class. Constructors and destructors have no return type, not even void.

Declaration of destructor function is similar to that of constructor function. The destructor's name should be exactly the same as the name of constructor; however it should be preceded by a tilde (~). For example if the class name is student then the prototype of the destructor would be ~ student ()

To illustrate the use of constructor, let us take the following C++ program:

C++ Program to illustrate constructor:

```
# include<iostream.h>
# include<conio.h>
class student
{
public:
student()                    // user defined constructor
{
cout<< "object is initialized"<<end1;
}
};
void main ()
{
student x,y,z;
getch();
}
```

Output:

Object is initialized

Here in the above program, one default constructor student has been created which is similar to the class name student. When objects of the student class are created, then default constructor is automatically executed, and three times it displays the output "Object is initialized". Since here, three objects, x, y and z have been defined for every execution student constructor.

4.2.1 Characteristics of Constructors

A constructor for a class is needed so that the compiler automatically initializes an object as soon as it is created. A class constructor if defined is called whenever a program creates an object of that class. The constructor functions have some special characteristics which are as follows:

- They should be declared in the public section.
- They are invoked directly when an object is created.
- They don't have return type, not even void and hence can't return any values.
- They can't be inherited; through a derived class, can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors can't be virtual.
- Constructor can be inside the class definition or outside the class definition.
- Constructor can't be friend function.
- They can't be used in union.
- They make implicit calls to the operators new and delete when memory allocation is required.

When a constructor is declared for a class, initialization of the class objects becomes necessary after declaring the constructor

Limitations of Constructor:

C++ constructors have the following limitations:

- **No return type**
  A constructor cannot return a result, which means that we cannot signal an error, during the object initialization. The only way of doing it is to throw an exception from a constructor.
- **Naming**
  A constructor should have the same name as the class, which means we cannot have two constructors that both take a single argument.
- **Compile time bound**
  At the time when we create an object, we must specify the name of a concrete class which is known at compile time. There is no way of dynamic binding constructors at run time.
- **There is no virtual constructor**
  We cannot declare a virtual constructor. We should specify the exact type of the object at compile time, so that the compiler can allocate memory for that specific type. If we are constructing derived object, the compiler calls the base class constructor first and the derived class hasn't been initialized yet. This is the reason why we cannot call virtual methods from the constructor.

4.2.2 Declaration of Constructors

A member function with the same name as its class is called constructor and it is used to initialize the objects of that class type with an initial value. Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, to avoid unexpected results in the example given below we have initialized the value of rollno as 0 and marks as 0.0.

In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even void.

A constructor is declared and defined as follows:

```
// class with constructor

class student
{
        private:
                int rollno;
                float marks;
        public:
              .
              .
              student ()
             {
        // constructor student declared having same name as
                        // that of its class
                        rollno = 0;
        // constructor value initialized
                        marks = 0.0;
                }
                .
                        .
}
```

In the above example class student has the constructor function defined with the same name and its values are initialized after creating it. When a class contains a constructor like one defined above, it is granted and understood that an object created by the class will be initialized automatically.

The output of the above C++ program is: rollno and marks of the student class.

4.2.3 Application of Constructors

A constructor is a special method that is created when the object is created or defined. This particular method holds the same name as that of the object and it initializes the instance of the object whenever that object is created. The constructor also usually holds the initializations of the different declared member variables of its object. Unlike some of the other methods, the constructor does not return a value, not even void.

When you create an object, if you do not declare a constructor, the compiler would create one for your program; this is useful because it lets all other objects and functions of the program know that this object exists. This compiler created constructor is called the default constructor. If you want to declare your own constructor, simply add a method with the same name as the object in the public section of the object. When you declare an instance of an object, whether you use that object or not, a constructor for the object is created and signals itself.

Generally, a constructor should be defined under the public section of a class, so that its objects can be created in any function.

A class can have multiple constructors for various situations. Constructors overloading are used to increase the flexibility of a class by having more number of constructor for a single class. To illustrate this let us take the following C++ program

```
# include <iostream.h>
  class Overclass
    {
          public:
          int x;
          int y;
     Overclass() { x = y = 0; }
          Overclass(int a) { x = y = a; }
          Overclass(int a, int b) { x = a; y = b; }
    };
  int main()
    {
           Overclass A;
      Overclass A1(4);
      Overclass A2(8, 12);
      cout << "Overclass A's x, y value:: " << A.x  << " , "<< A.y << "\n";
      cout << "Overclass A1's x,y value:: "<< A1.x << " ,"<< A1.y << "\n";
      cout << "Overclass A2's x,y value:; "<< A2.x << " , "<< A2.y << "\n";
      return 0;
  }
```

Here in the above example, the constructor "Overclass" is overloaded thrice with different initialized values.

4.2.4 Constructor with Arguments

Parameters and arguments are sometimes loosely used interchangeably; in particular, "argument" is sometimes used in place of "parameter". Nevertheless, there is a

difference. Properly, parameters appear in procedure definitions; arguments appear in procedure calls.

A parameter is an intrinsic property of the procedure, included in its definition. For example, in many languages, a minimal procedure to add two supplied integers together and calculate the sum total would need two parameters, one for each expected integer. In general, a procedure may be defined with any number of parameters or no parameters at all. If a procedure has parameters, the part of its definition that specifies the parameters is called its parameter list.

By contrast, the arguments are the values actually supplied to the procedure when it is called. Unlike the parameters, which form an unchanging part of the procedure's definition, the arguments can, and often do, vary from call to call. Each time a procedure is called, the part of the procedure call that specifies the arguments is called the argument list. Many programmers use parameter and argument interchangeably, depending on context to distinguish the meaning.

To better understand the difference, let us consider the following function:

```
int sum(int addend1, int addend2)
{
   return addend1 + addend2;
}
```

The function sum has two parameters, named addend1 and addend2. It adds the values passed into the parameters, and returns the result

The code which calls the *sum* function might look like this:

```
int sumvalue;
int value1 = 40;
int value2 = 2;
sumvalue = sum(value1, value2);
```

The variables value1 and value2 are initialized with values. Value1 and value2 are both arguments to the sum function in this context.

At runtime, the values assigned to these variables are passed to the function sum as arguments. In the sum function, the parameters addend1 and addend2 are evaluated, yielding the arguments 40 and 2, respectively.

The values of the arguments are added, and the result is returned to the caller, where it is assigned to the variable sumvalue.

☞ Check Your Progress 1

Multiple choice questions:

a) Default constructor takes _____

1) one parameter
2) two parameters
3) no parameters
4) character type parameter

b) A constructor is called when ever _____

1) An object is declared
2) An object is used
3) A class is declared
4) A class is used

c) The difference between constructor and destructor are_____

1) Constructor can take arguments but destructor didn't
2) Constructor can be overloaded but destructor didn't
3) Both a & b
4) None of these

d) A class having no name_____
1) is not allowed
2) can't have a constructor
3) can't have a destructor
4) can't be passed as an argument

Short answer type questions:

1: What do you mean by constructor in C++ programming?

……

……

……

2: Explain the application of constructors and destructors in memory management with reference to C++ programming.

……

……

……

3: What is the difference between parameter and argument with respect to constructor and destructor in C++ programming?

……

……

……

4: In what ways a constructor is different from an automatic initialization?

……

……

……

4.3 TYPES OF CONSTRUCTOR

In object oriented programming (OOP's), a constructor in a class is used to initialize the value of the object. It prepares the new object for use by initializing its legal value. We will discuss main types of constructors, their features and applications in the following section:

4.3.1 Default Constructor

A default constructor is a constructor that either has no parameters, or if it has parameters, all the parameters have default values.

If no user-defined constructor exists for a class A and one is needed, the compiler implicitly declares a default parameter less constructor A::A(). This constructor is an inline public member of its class. The compiler will implicitly define A::A() when the compiler uses this constructor to create an object of type A. The constructor will have no constructor initializer and a null body. For instance, consider the following example

```
class A {
                int i;
                public:
                void getval(void);
                void prnval(void);
                // member function definitions
        }

        A 0b 1;
// uses default constructor for creating   0b1. Since user can use it,
// that means, this implicitly defined default constructor is public
// member of the class
        0b1. Getval();
        0b1. Getval();
```

Having a default constructor simply means that an application can declare instances of the class. The compiler first implicitly defines the implicitly declared constructors of the base classes and non-static data members of a class A before defining the implicitly declared constructor of A. No default constructor is created for a class that has any constant or reference type members.
A constructor of a class A is trivial if all the following statements are true:

- It is implicitly defined
- A has no virtual functions and no virtual base classes
- All the direct base classes of A have trivial constructors
- The classes of all the non-static data members of A have trivial constructors
- The default constructor provided by the compiler does not do anything specific. It simply allocates memory to data members of the object.

4.3.2 Parameterized Constructor

We can write a constructor in C++ which can accept parameters for its invocation. Such constructor that can take the arguments are called parameterized constructor.

In other words; "Declaring a constructor with arguments hides the default constructor". This means that we can always specify the arguments whenever we

declare an instance of the class. To illustrate parameterized constructor well, let us write the syntax and take one example:

Syntax:

```
class <cname>
{
   //data
   public: cname(arguments); // parameterized constructor
         .....
         .....
};
```

Examples:

```
class student
{
     char name[20];
     int rno;
     public: student(char,int);       //parameterized constructor
};
student :: student(char n,int r)
{
    name=n;
    rno=r;
}
```

In the above example parameterized constructor has been shown by comment line. When the constructor is parameterized, we must provide appropriate arguments for the constructor.

4.3.3 Copy Constructor

A copy constructor is used to declare and initialize an object from another object. For example, the statement

```
integer 12(11);
```

would define the object 12 and at the same time initialize it to the value of 11. Another form of this statement is

```
integer 12 = 11;
```

Thus the process of initializing through a copy constructor is known as copy initialization. A copy constructor is always used when the compiler has to create a temporary object of a class object. The copy constructors are used in the following situations

- The initialization of an object by another object of the same class.
- Return of objects as a function value.
- Stating the object as by value parameters of a function.

The syntax of copy constructor is:

```
class_name :: class_name(class_name &ptr)
```

Another Example of copy constructor is:

X :: X(X &ptr)

Here, X is user defined class name and ptr is pintor to a class object X.

- Normally, the copy constructor takes an object of their own class as arguments and produces such an object.

- The copy constructors usually do not return a function value. Constructors cannot return any function values.

Program to illustrate the use of Copy Constructor:

```
#include<iostream>
#include<conio.h>
using namespace std;
class student
{

     int roll;
     char name[30];
   public:
       student()                          // default constructor
         {
               roll =10;
               strcpy(name,"x");
         }
       student( student &O)               // copy constrcutor
         {
               roll =O.roll;
               strcpy(name, O.name);
         }

         void input_data()
          {
                    cout<<"\n Enter roll no :"; cin>>roll;
                    cout<<"\n Enter name :"; cin>>name;
           }
         void show_data()
          {
             cout<<"\n Roll no :"<<roll;
             cout<<"\n Name    :"<<name;
          }
};
int main()
 {
        student s;                          // default construtor
        s.show_data();
        student A(s);                       // copy constructor
        A.show_data();
        getch();
        return 0;
        }
```

Here in the above C++ program we have taken a user defined class student and used copy constructor to initialize another object student rollno and name from the student constructor.

4.3.4 Constructor Overloading

When more than one constructor function is defined in a class, then it is called constructor overloading or use of multiple constructor in a class. It is used to increase the flexibility of a class by having more number of constructors for a single class. Overloading constructors in C++ programming gives us more than one way to initialize objects in a class.

To illustrate the constructor overloading, let us take following examples:

```
#include<iostream>
#include<conio.h>
using namespace std;
class student
{
int roll;
     char name[30];
   public:
      student(int x, char y[])                  // parameterized constructor
        {
              roll =x;
              strcpy(name,y);
        }
       student()                                 // normal constructor
        {
              roll =100;
              strcpy(name,"y");
        }

        void input_data()
         {
           cout<<"\n Enter roll no :"; cin>>roll;
           cout<<"\n Enter name :"; cin>>name;
         }
        void show_data()
         {
            cout<<"\n Roll no :"<<roll;
            cout<<"\n Name    :"<<name;
         }
};

int main()
 {
        student s(10,"z");
        s.show_data();
        getch();
        return 0;
        }
```

The above C++ program is used to create a constructor student inside the class student. Here in this program, the student constructor has been defined in many ways and the output of the program is student name and his roll no.

```
// overloading class constructors

#include <iostream>
using namespace std;
class CRectangle
{
   int width, height;
  public:
   CRectangle ();
   CRectangle (int,int);
   int area (void)
            {
                     return (width*height);
            }
};

CRectangle::CRectangle ()
            {
              width = 5;
              height = 5;
            }

CRectangle::CRectangle (int a, int b)
            {
              width = a;
              height = b;
            }

int main ()
{
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
return 0;
}
```

Output:

rect area: 12
rectb area: 25

In this case, rectb was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both width and height with a value of 5.

4.3.5 Constructing two Dimensional Constructor

A typical two-dimensional array is like a time-table. To locate a piece of information, you determine the required row and column and then read the location where they meet.

In the same way, a two dimensional array is a grid containing rows and columns in which each element is uniquely specified by means of its row and column coordinates. Two-dimensional character arrays hold an array of strings wherein a row represents a string and a column represents a single character in each of the strings.

Declaration of Two Dimensional Arrays:

The general form of declaration of a two-dimensional character array is:

```
char arrayname[x][y];
```

where 'x' is the number of rows and 'y' is the number of columns.

The following guidelines need to be followed while declaring two-dimensional character arrays.

- The number of rows should be equal to the number of strings in the array.
- Column specification should be greater than or equal to the length of the longest string in the array plus one.

For example, if there are seven strings in an array and the length of the longest string is nine, the array can be declared in the following manner:

char cWeekdays[7][10];

Initializing of Two-dimensional Arrays:

The rules for initializing two-dimensional arrays are the same as for one-dimensional arrays.

For example, to declare and initialize an array that would hold the days of the week, the array definition would be as follows:

```
char cWeekdays[7][10] = {"Sunday","Monday","Tuesday","Wednesday",
"Thursday","Friday","Saturday"};
```

Explanation of the code:

In the above example, cWeekdays[0] will refer to the string "Sunday" and cWeekdays[4][1] would refer to the character 'h' of the string "Thursday".

We have discussed the constructor and their types in the above paragraphs and we have come to the end of constructor with the discussion of two dimensional array in object oriented programming.

We will discuss destructor and memory management in coming pages of this unit. The function of destructor is opposite to that of constructor and is used to free the memory which was allocated for the creation of constructor. Let us now discuss the destructor in detail in the next paragraph.

☞ Check Your Progress 2

Multiple choice questions:

1: A constructor is called whenever

a) An object is declared
b) An object is used
c) A class is created
d) A class is used

2: A destructor takes

a) One argument
b) Two argument
c) Three argument
d) Zero argument

3: The difference between constructor and destructor is

a) Constructor can take argument but destructor can not
b) Constructor can be overloaded but destructor can not
c) Both a and b
d) None of the above

Short Answer type questions:

1: What do you mean by default constructor? Explain by taking example.

……
……
……

2: What do you mean by copy constructor? Explain it with a C ++ Program.

……
……
……

3: Define parameterized constructor by taking a C++ program.

……
……
……

4: What do you mean by constructor overloading in C ++. Explain by taking one example.

……
……
……

4.4 DESTRUCTORS

Destructors are functions that are complimentary to constructors. A destructor, as the name implies, is used to destroy the objects that have been created by using constructor. They de-initialize objects when they are destroyed. A destructor is invoked when an object of the class goes out of scope, or when the memory occupied by it is de allocated using the delete operator. A destructor is a function that has the same name as that of the class but is prefixed with a ~ (tilde).

For example the destructor for the class students will bear the name ~student(). Destructor takes no arguments or specifies a return value, or explicitly returns a value not even void. It is called automatically by the compiler when an object is destroyed. A destructor cleans up the memory that is no longer required or accessible. To understand the syntax of destructor let us take following example.

4.5 DECLARATION OF DESTRUCTOR

Let us take the following C++ program to illustrate the declaration of destructor

Program: 1

```
#include<iostream.h>
#include<conio.h>
class student                          // student is a class
{
public :
~student()                             // destructor declaration
{
cout <<"Thanx for using this program" <<endl;
}
};
main()
{
clrscr();
student s1; // s1 is an object
getch();
}
```

Here in the above program ~student () function has been used for destroying the use of memory which was allocated to constructor class by the operating system.

Program: 2

```
#include<iostream>
#include<conio.h>
using namespace std;
class xyz
  {
     public:
          xyz()
            {
               cout<<"\n Constructor ";
            }
          ~xyz()              // use of destructor with  -- tilde
            {
               cout<<"\n Destructor ";
            }
  };
int main()
  {
     {
       xyz B;
      }
     getch();
     return 0;
  }
```

Here in the above program ~xyz() function has been used for destroying the use of memory which was allocated during the initialization of the constructor to constructor class.

Destructors are less complicated than constructors. You don't call them explicitly

(they are called automatically for you), and there's only one destructor for each object. The name of the destructor is the name of the class, preceded by a tilde (~).

Only the function having access to a constructor and destructor of a class can define objects of this class types otherwise compiler reports error at the time of compilation of the program. Generally, a destructor should be defined under the public section of the class, so that its objects can be destroyed in any function.

Further, destructors are usually used to de-allocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted. A destructor is a member function with the same name as its class prefixed by a ~ (tilde). For example:

```
class X
 {
  public:
  // Constructor for class X
  X();
  // Destructor for class X
  ~X();
};
```

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared const, volatile, const volatile or static. A destructor can be declared virtual or pure virtual.

4.6 APPLICATION OF DESTRUCTOR

During construction of an object by the constructor, a resource may be allocated for use. For example, a constructor may have opened a file and a memory area may be allocated to it. Similarly, a constructor may have allocated memory to some other objects. These allocated resources must be deallocated before the object is destroyed. In object oriented programming such as C++, this work is done by using destructor which performs all clean-up tasks and is equally useful as constructor is in managing the memory and resources of the computer system.

The main application and characteristics of destructors are given as follows:

1. Destructor functions are invoked automatically when the objects are destroyed.
2. We can have only one destructor for a class, i.e. destructors can't be overloaded.
3. If a class have destructor, each object of that class will be de initialized before the object goes out of the scope
4. Destructor function, also obey the usual access rules of as other member function do.
5. No argument can be provided to a destructor, neither does it return any value.
6. Destructor can't be inherited.
7. A destructor may not be static.
8. It is not possible to take the address of destructor.
9. Member function may be called from within a destructor.
10. An object of a class with a destructor can't be a member of a union.
11. We may make an object const if it does not change any of its data value.

Limitations of destructors:

C++ destructors have mainly two disadvantages:

1) They are case sensitive.
2) They are no good for big programs having thousand lines of code.

4.7 PRIVATE CONSTRUCTOR AND DESTRUCOTR

In object oriented programming such as C++, private constructor is used to create object of class only one time. It means that your class does not have more than one instance of your class. Private constructor is same as any other constructor but it is declared as private. Due to this, you can only access object of that class and not in other class. Most of the people use private constructor which do all the work desired in the program by constructor.

Let us take an example to understand the use of private constructor:

```
class student()
        {
        private:
```

```
Student ()  {}
        public:
        student  & instances()
                {
                Static student * aGlobalInsts = new student ();
                Return * student;
                }
        }
```

In the above example, private constructor student has been defined in the private section of the program and can be accessed within the private section only. The output of the above C++ program is details of student.

In object oriented programming such as C++, destructor are declared in private section to prevent the object of one class from automatic deleting by the destructor of any other class. If a class is in singleton then it is possible to declare constructor and destructor as private. However, in standard programming of C++ destructors are declared as public and not private.

4.8 PROGRAMS ON CONSTRUCTOR & DESTRUCTOR

A program to print student details using constructor and destructor:

```
#include<iostream.h>
  #include<conio.h>
  class stu
  {
            private: char name[20],add[20];
                     int roll,zip;
                public: stu ( );//Constructor
                     ~stu( );//Destructor
                     void read( );
                     void disp( );
  };
            stu :: stu( )
  {
            cout<<"This is Student Details"<<endl;
  }
  void stu :: read( )
  {
  cout<<"Enter the student Name";
  cin>>name;
  cout<<"Enter the student roll no ";
  cin>>roll;
  cout<<"Enter the student address";
  cin>>add;
  cout<<"Enter the Zipcode";
  cin>>zip;
  }
  void stu :: disp( )
  {
```

```
        cout<<"Student Name :"<<name<<endl;
                cout<<"Roll no   is      :"<<roll<<endl;
                cout<<"Address is       :"<<add<<endl;
                cout<<"Zipcode is       :"<<zip;
        }
        stu : : ~stu( )
        {
                cout<<"Student Detail is Closed";
        }

        void main( )
        {
        stu s;
        clrscr( );
        s.read ( );
        s.disp ( );
        getch( );
        }
```

Output:

Enter the student Name
Sushil
Enter the student roll no
01
Enter the student address
Delhi
Enter the Zipcode
110092
Student Name : Sushil
Roll no is : 01
Address is : Delhi
Zipcode is :110092

A program to calculate factorial of a given number using copy constructor

```
#include<iostream.h>
#include<conio.h>
class copy
{
        int var,fact;
        public:

         copy(int temp)
         {
          var = temp;
         }

         double calculate()
         {
                fact=1;
                for(int i=1;i<=var;i++)
                {
                fact = fact * i;
```

```
                    }
                    return fact;
                }
};
void main()
{
   clrscr();
   int n;
   cout<<"\n\tEnter the Number : ";
   cin>>n;
   copy obj(n);
   copy cpy=obj;
   cout<<"\n\t"<<n<<" Factorial is:"<<obj.calculate();
   cout<<"\n\t"<<n<<" Factorial is:"<<cpy.calculate();
   getch();
}
```

Output:

Enter the Number: 5
Factorial is: 120
Factorial is: 120

C++ Program for Add two time variables using constructor and destructor

```
#include<iostream.h>
#include<conio.h>
class Time
{
int minutes,hours,a;
static int i;
public:
Time(int a)
{
this->a=hours=a;
this->a+=5;
minutes=i++;
cout<<"\nObj address : "<<this;
cout<<"\nAddress of i : "<<&i;
cout<<"\na= "<<this->a<<"\t\t"<<a;
getch();
}
~Time()
{
cout<<endl<<"\t\t"<<hours<<" : "<<minutes;
getch();
}
};
int Time ::i;
void main()
{
clrscr();

Time t3(10),t2(1);
}
```

A C++ program by taking a "patient" class and by using constructor and destructor

```
#include <iostream>
#include <string>

using namespace std;

class Patient {
public:
   Patient(string name, int heartRate = 0, double moneyOwed = 0.0);

   Patient()
   {
      //default constructor for string, _name, implicitly called
      _heartRate = 0; //Bad for business
      _moneyOwed = 0.0;
   }

   int getHeartRate() {return _heartRate;}
   void setHeartRate(int heartRate) {_heartRate = heartRate;}
   double getMoneyOwed() {return _moneyOwed;}
   void setMoneyOwed(double moneyOwed) {_moneyOwed = moneyOwed;}
   string getName() {return _name;}
   void setName(string name) {_name = name;}
   ~Patient() {} // This could be defined outside
                 // the class definition instead

private:
   int _heartRate;
   double _moneyOwed;
   string _name;
};

// This could be defined within the class definition instead.
// Always initialize member classes within the member initialization list.
Patient::Patient(string name, int heartRate, double moneyOwed) : _name(name)
{
   _heartRate = heartRate;
   _moneyOwed = moneyOwed;
}

int main()
{

   Patient jk("John",70,0.0);

   //Before visit to doctor
   cout << "Patient " << jk.getName() << " owes " << jk.getMoneyOwed() << endl;
   cout << "His heart rate is " << jk.getHeartRate() << endl;

   //After visit to doctor
```

```
    jk.setMoneyOwed(1000.0);
    jk.setHeartRate(140);
    cout << "Patient " << jk.getName() << " owes " << jk.getMoneyOwed()
<< endl;
    cout << "His heart rate is " << jk.getHeartRate() << endl;

    return 0;
}
```

Finally, let us sum up the advantages and disadvantages of constructor and destructors:

- Constructors and destructors do not have return types nor can they return values.
- References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
- Constructors cannot be declared with the keyword virtual.
- Constructors and destructors cannot be declared static, const, or volatile.
- Unions cannot contain class objects that have constructors or destructors.

4.9 MEMORY MANAGEMENT

Memory management is a large subject area and will be discussed length other courses of BCA. However, we will have a brief discussion on memory management in this unit of constructor and destructor for the learners.

C++ offers a wide range of choices for the management of memory. There are various techniques to manage the memory in object oriented programming such as C++. Some of the techniques for memory management are given as follows:

Manual Memory Management: Up until the mid 1990s, the majority of programming languages used in industry supported manual memory management. Today, C++ is the main manually memory management languages. Manual memory management refers to the usage of manual instructions by the programmer to identify and de-allocate unused objects, or garbage.

All programming languages use manual techniques to determine when to *allocate* a new object from the free store. C++ language uses the new operator to determine when an object ought to be created and memory is allocated to it. There are two types of memory management operators in C++. These are:

- **new**
- **delete**

These two memory management operators are used for allocating and freeing memory blocks in efficient and convenient way.

The fundamental issue is the determination of when an object is no longer needed (i.e. is garbage), and arranging for its underlying storage to be returned to the free store so that it may be re-used to satisfy future memory requests.

Constructors and destructor are used in C++ programming language to initialize and destroy the memory blocks, when it is no longer required by the program. In manual memory allocation, this is also specified manually by the programmer; at the time of writing the codes by programmer.

Manual Memory Management and Correctness

- Manual memory management is known to enable several major classes of bugs into a program, when used incorrectly.
- When an unused object is never released back to the free store, this is known as a memory leak in C++ programming. In some cases, memory leaks may be tolerable, such as a program which "leaks" a bounded amount of memory over its lifetime, or a short-running program which relies on an operating system to de-allocate its resources when it terminates.
- Programmers are expected to invoke dispose() manually as appropriate; to prevent "leaking" of scarce graphics resources. However, in many cases memory leaks occur in long-running programs, and in such cases an unbounded amount of memory is leaked. Whenever this occurs, the size of the available free store continues to decrease over time; when it finally exhaust then the program crashes/terminates abnormally.
- When an object is deleted more than once, or when the programmer attempts to release a pointer to an object not allocated from the free store, catastrophic failure of the dynamic memory management system can result and cause bugs in the program.

☞ Check Your Progress 3

Multiple choice questions:

1. What is the only function all C++ programs must contain?
   A. start()
   B. system()
   C. main()
   D. program()

2. What punctuation is used to signal the beginning and end of code blocks?
   A. { }
   B. -> and <-
   C. BEGIN and END
   D. (and)

3. What punctuation ends most lines of C++ code?
   A. . (dot)
   B. ; (semi-colon)
   C. : (colon)
   D. ' (single quote)

Short Answer type questions

1: What do you mean by destructor? Explain by taking example.

……

……

……

2: What do you mean by private constructor and destructor?

……

……

……

3: Explain the role of destructor in C++ in memory management.

……

……

……

4: Write a program in C++ to demonstrate the use of destructor.

……

……

……

4.10 SUMMARY

A class constructor is a class method having the same name as the class name. The constructor does not have a return type. A constructor may take zero or more parameters. If we don't apply a constructor, the compiler provides a no-argument default constructor. It is important to understand that if we write our own constructor, the compiler does not provide the default constructor. A class may define one or more constructor. It is up to us to decide which constructor to call during object creation by passing an appropriate parameter list to the constructor. We may set the default value for the constructor parameter.

A class destructor is a class method having the same name as the class name and is prefixed with tilde (~) sign. The destructor does not return anything and does not accept any argument. A class definition may contain one and only one destructor. The runtime calls the destructor, if available, during the object creation. A destructor is typically used for freeing the resources allocated in the class constructor.

4.11 ANSWERS TO CHECK YOUR PROGRESS

Multiple choice questions

a) - 3, b)-1 c)-1 d)-3

Answers to short type questions

1: A member function with the same name as its class is called constructor and it is used to initialize the objects of that class type with an initial value. Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution.

2: There are various techniques to manage the memory in C++ programming. These are:

1) By using constructor and destructor
2) By using New and Delete operator
3) By using manual garbage collection feature of C++ programming
4) By using the function dispose() in C++ programming

3: Parameters and arguments are sometimes loosely used interchangeably; in particular, "argument" is sometimes used in place of "parameter". Nevertheless, there is a difference. Properly, parameters appear in procedure definitions; arguments appear in procedure calls.

4: Constructor needs to initialize before it is used and it is not automatically incremented or decremented during the execution of program, while automatic initialization takes place automatically, without any interaction of the user.

Check Your Progress 2

Multiple choice questions

1) a 2) d 3) c

Answers to short type answer questions

1: A default constructor is a constructor that either has no parameters, or if it has parameters, all the parameters have default values. Following program depicts the use of default constructor in C++ programming:

```
class A {       int i;
                        public:
                          void getval(void);
                          void prnval(void);
                           .
                           .                    // member function definitions
                }

        A 0b 1;         // uses default constructor for creating 0b1. Since user
can use it,
                        // that means, this implicitly defined default constructor
is public
                        // member of the class
        0b1. Getval();
        0b1. Getval();
```

2: A copy constructor is used to declare and initialize an object from another object. For example, the statement

```
integer 12(11);
```

would define the object 12 and at the same time initialize it to the value of 11. Another form of this statement is

```
integer 12 = 11;
```

3: We can write a constructor in C++ which can accept parameters for its invocation. Such constructor that can take the arguments are called parameterized constructor.

4: When more than one constructor function is defined in a class, then it is called constructor overloading or use of multiple constructor in a class. It is used to increase the flexibility of a class by having more number of constructors for a single class. Overloading constructors in C++ programming gives us more than one way to initialize objects in a class.

Check Your Progress 3

Multiple choice questions

1) c 2) a 3) b

Answers to short type answer questions

1: A destructor, as the name implies, is used to destroy the objects that have been created by using constructor. They de-initialize objects when they are destroyed. A destructor is invoked when an object of the class goes out of scope, or when the memory occupied by it is de allocated using the delete operator. A destructor is a function that has the same name as that of the class but is prefixed with a ~ (tilde). For example the syntax of class student could be written as follows

```
~student()
```

2: In object oriented programming such as C++ private constructor is used to create object of class only one time. It means that your class does not have more than one instance of your class. Private constructor is same as any other constructor but it is declared as private. Due to this you can only access object of that class in only that class and not in other class.

3: The role of destructors in C++ programming is given as follows:

1. Destructor functions are invoked automatically when the objects are destroyed.
2. We can have only one destructor for a class, i.e. destructors can't be overloaded.
3. If a class have destructor, each object of that class will be de initialized before the object goes out of the scope
4. Destructor function, also obey the usual access rules of as other member function do.
5. No argument can be provided to a destructor, neither does it return any value.
6. Destructor can't be inherited.
7. A destructor may not be static.
8. It is not possible to take the address of destructor.
9. Member function may be called from within a destructor.

4: A C++ program by using destructor is given as follows

```
#include<iostream.h>
#include<conio.h>
class student                    // student is a class
{
public :
~student()                       // destructor declaration
{
cout <<"Thanx for using this program" <<endl;
}
};
main()
{
clrscr();
student s1; // s1 is an object
getch();
}
```

4.12 FURTHER READINGS AND REFERENCES

1) E Balagurusamy, *Object Oriented Programming with C++*, Tata McGraw-Hill Publishing Company Ltd, 2004, New Delhi - 110008

2) Er V. K. Jain, *Object Oriented Programming with C++*, Cyber Tech Publication, Daryaganj N Delhi-110002

3) Robert Lafore, *Object Oriented Programming in C++*, Galgotia Publications Pvt. Ltd. Daryaganj N Delhi-11002

4) Rajesh K Shukla, *Object Oriented Programming in C++*, Wiley India Publishing Pvt. Ltd. Daryaganj, June 2008, New delhi-110002

5) Bjarne AT&T Labs Murray Hill, New Jersey Stroustrup, *Basics of C++ Programming*, Special Edition, Publisher: Addison-Wesley Professional.

Reference Websites:

(1) www.sciencedirect.com
(2) www.ieee.org
(3) www.webpedia.com
(4) www.microsoft.com
(5) www.freetechbooks.com
(6) www.computer basics.com
(7) www.youtube.com

UNIT 1 INHERITANCE

Structure **Page No.**

1.0 INTRODUCTION

There is a great surge of interest today in Object Oriented Programming (OOP) due to obvious reasons. One of the most fundamental concepts of the object-oriented paradigm is inheritance that has profound consequences on the development process. In OOP, it is possible to define a class as inheriting from another. Object Oriented Software Development involves a large number of classes. Many of the classes extension of others. Object Oriented Programming has been widely acclaimed as a technology that will support the creation of re-usable software, particularly because of the inheritance facility. In OOP, inheritance is a re-usability technique. We can show similarities between classes by means of inheritance and describe their similarities in a class which other classes can inherit. Thus, we can re-use common descriptions. Therefore, inheritance is often promoted as a core idea for reuse in the software industry. The abilities of inheritance, if properly used, is a very useful mechanism in many contexts including reuse. This unit starts with a discussion on the re-usability concept. This is followed by the inheritance which is prime feature of object oriented paradigm. It focuses on the standard form of inheritance by extension of a base class with a derived class. Moreover, in this unit, different types of inheritance, the time of its use and methods of its implementation are also discussed. Base classes, derived classes, visibility of class members, and constructors and destructors in derived classes are introduced. Illustrative examples that facilitate understanding of the concept are presented.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- define and understand why we need to study inheritance;
- define and understand the concept of reusability;
- describe an inheritance relationship;
- identify the cases where inheritance is suitable;
- define base classes and derived classes;
- implement different types of inheritance in C ++;

- explain the different types of inheritance;
- understand the need of virtual base class, and
- understand and implement the constructors in derived classes.

1.2 CONCEPT OF RE-USABILITY

Software re-usability is primary attribute of software quality. C++ strongly supports the concept of reusability. C++ features such as classes, virtual functions, and templates allow designs to be expressed so that re-use is made easier (and thus more likely), but in themselves such features do not ensure re-usability. Do we really need re-use? This question can best be answered by an analogy from automobile industry. Consider the design and creation of a new car model. The automotive engineer does not design a new car from scratch. Rather, the engineer borrows from the design of existing cars. For example, the engine design from an existing car may be used in a new model. If the engine design has been used in a previous model, design problems have likely been resolved. Thus, development costs are reduced because a new engine does not need to be designed and tested. Finally, consumer maintenance costs are reduced because machines and others who must maintain the car are already familiar with the operation of the engine.

The American Heritage Dictionary defines quality as "a characteristic or attribute of something". Re-usability is the degree to which a thing can be reused. Software re-usability represents the ability to use part or the whole system in other systems which are related to the packaging and scope of the functions that programs perform. Can you tell why do we need to study re-usability? Well, the need for re-usability comes from the observation that software systems often follow similar patterns; it should be possible to exploit this commonality and avoid reinventing solutions to problems that have been encountered before. Do you know what the advantage of re-usability is? There are many advantages of re-usability. They can be applied to reduce cost, effort and time of software development. It also increases the productivity, portability and reliability of the software product.

We can say that re-usability is concerned as to how we can use a system or its part in other systems.

Now, can you tell how the re-use is achieved in program? Re-use in OOP language can be achieved by two ways basically: The first is through class definition-every time a new object of class is defined, we reuse all the code and declarations of that class. This type of re-use can be supported in the function-oriented approach also. The other type of re-use, which is particular to OOP language, can be supported by inheritance. Inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possibly by deriving a new class from the existing one. You have already seen the first type of reusability through class. In the next section, we shall see, how will we achieve the reusability through inheritance?

1.3 INHERITANCE

Inheritance is a prime feature of object oriented programming language. It is process by which new classes called derived classes(sub classes, extended classes, or child classes) are created from existing classes called base classes(super classes, or parent classes). The derived class inherits all the features (capabilities) of the base class and can add new features specific to the newly created derived class. The base class remains unchanged.

Inheritance is a technique of organizing information in a hierarchical form. It is a relation between classes that allows for definition and implementation of one class based on the definition of existing classes.

Inheritance is often referred to as an "is-a" relationship because every object of the class being defined "is" carries an object of the inherited class also.

You can look around and find many real world examples of inheritance like Inheritance between parent and child, employee and manager, person and student, vehicle and light motor vehicle, and animal and mammal etc. Why are we interested in inheritance and how will we use this concept? Well, Let us take the example to understand this concept. Suppose we want to use the classes Employee and Manager in the C++ program. For the time being, let us assume that we do not know the concept of the inheritance. Now, we define the Employee and Manager classes as follows:

```
class Employee
{
public:
int id_number;
char Name_of_Employee[25];
int age;
char Department_Name[25];
int salary;
char Address[25];
// Member functions
void display_name(void);
void display_id(void);
void raise_salary(float percent);
.
.
.
};
class Manager
{
public:
int id_number;
char Name_of_Employee[25];
int age;
char Department_Name[25];
int salary;
char Address[25];
char  Name_of_secretary[25];
Employee *team_members;
void display_name(void);
void display_id(void);
void display_secretary_name(void);
void raise_salary(float percent);
.
.
.
};
```

If you look at the above declarations of the classes Employee and Manager, you can make the observation that there are some common attributes and behavior in Employee and Manager class. We have shown the common attributes and behavior in Manager class again. Now, we introduce the concept of the inheritance. As we know that generally Managers are treated differently from other employees in the organization. Their salary raises are computed differently, they have access to a secretary, they have a group of employee under them and so on. There are some

common data members as well as member functions like name, age, salary, address, display_name(),display_id() etc . This is a kind of situation in which we use the concept of inheritance. Why? Well, we can retain some of what we have already laid down and defined in the Employee class in terms of data members and member functions. Employee and Manager classes are declared as follows:

```
class Employee
{
public:
int id_number;
char Name_of_Employee[25];
int age;
char Department_Name[25];
int salary;
char Address[25];
// Member functions
void display_name(void);
void display_id(void);
void raise_salary(float percent);
.
.
.
};
class Manager :: public Employee
{
public:
char  name_of_secretary[25];
Employee *team_members;
void display_secretary_name(void);
void raise_salary(float percent);
.
.
.
};
```

You can look at the above declaration and observe that we did not declare the common attributes and functions again in the Manager class. Thus, we have reused the previous declarations of data members and functions. We can also observe that we have redefined raise_salary function in the Manager class due to different way to compute salary of the Manager. From the above discussion, we can conclude that what is meant by the application of inheritance and how it is supporting the concept of re-usability by adding additional feature to an existing classes without modifying it.

1.3.1 Derived and Base Class

As we know, when Class A inherits the feature from class B, then Class A is called the derived class and B is called Base class. A derived class extends its features by inheriting the properties (features) from another class called the base class while adding features of its own. In next section, we will see as to how we will define a derived class.

1.3.2 Declaration of Derived Class

The declaration of a derived class shows its relationship with the base class in addition to its own details. The common syntax of declaring a derived class is given as follows:

```
class DerivedClassName : [VisibilityMode] BaseClassName
{
// members of derived class
};
```

The derivation of DerivedClassName from the BaseClassName is indicated by colon (:). The VisibilityMode enclosed within the square bracket is optional. If the VisibilityMode is specified, it must be either public or private or protected. It specifies the features of the base class that are privately derived or publicly derived. There are four possible ways of derivation of derived class which is given below:

```
class DerivedClassName : public BaseClassName          //public derivation
{
//members of derived class
};
class DerivedClassName : private BaseClassName         //private
derivation
{
//members of derived class
};
class DerivedClassName : protected BaseClassName       //protected
derivation
{
//members of derived class
};
class DerivedClassName : BaseClassName                 //private
derivation
{
//members of derived class
 };
```

1.3.3 Visibility of Class Members

There are three visibility modes (visibility modifier). They are private, public and protected. We have already learnt about private and public visibility mode in Unit 3 of Block 1 of this course. Could you tell what the role of these terms in the programs exists? They are actually controllers, used to control the access to members (data members and functions members) of a class.

Why is the different type of visibility mode needed in derivation of a derived class? Well, a class may contain some secret information which we are not interested to share by the derived classes and non-secret information which we are interested to share by the derived class. In nutshell, we can say that visibility mode promotes encapsulation. The visibility of the base class members undergoes modification in a derived class as summarized in Table 1.1.

Table 1.1: Visibility Mode

Base class Visibility	**Derived class Visibility**		
	Private derivation	**Public derivation**	**Protected derivation**
private public protected	Not inherited private private	Not inherited public protected	Not inherited protected protected

From the Table 1.1, you can observe that in derived class declaration, if the visibility mode is private then both 'public members' of the base class as well as 'protected members' of the base class will become private members of the derived class. Therefore, both public and protected member of base class can only be accessed by the member functions of the derived class. They can not be accessed by the objects of the derived class. And private members of the base class will not be inherited. On the other hand, if visibility mode is public, public members of the base class will become public members of the derived class and protected members of the base class will become protected members of the derived class whereas private member of the base class will never become the members of the declared class i.e. it will not be inherited. If the visibility mode is protected then the public and protected members of the base class will become the protected members of the derived class. In this case also, the private members of the base class will not become the member of its derived class. As we have demonstrated that the private members of base class will remain private to the base class whether the base class is inherited publicly or privately or protected by any means. They add to the items of the derived class and they are not directly accessible to the member of a derived class. Derived class can access them through the base class member functions. Consider the following declarations of a base class A and a derived classes B, C, and D to illustrate private and public inheritance.

```
class A
{
private:
int privateA;      // private member of base class A
protected:
int protectedA;  // protected member of base class A
public:
int publicA;       // public member of base class A
int getPrivateA() //public function of base class A
{
return privateA;
}
};

class B: private A          // privately derived class
{
private:
int privateB;
protected:
int protectedB;
public:
int publicB;
void fun1()
{
int b;
b=privateA;       //Won't work: privateA is not accessible
b=getPrivateA(); //OK: inherited member access private data
b=protectedA;   // OK
b=publicA;        // OK
}
};
```

```
class C: public A          // publically derived class
{
private:
int privateC;
protected:
int protectedC;
public:
int publicC;
void fun2()
{
int c;
c=privateA;       //Won't work :privateA is not accessible
c=getPrivateA(); //OK: inherited member access private data
c=protectedA;   // OK
c=publicA;        // OK
}
};
```

Consider the following statements:

```
B objb; // objb is a object of class B
C objc; // objc is a object of class C
int x;    // temporary variable x
```

The above statements define the object objb, objc and the integer variable x. Let us consider the statements as follows:

```
x=objb.protectedA;        //Won't work : protectedA is not accessible
x=objb.publicA;           //Won't work : publicA is not accessible
x=objb.getPrivateA();     // Won't work: getPrivateA() is not accessible
```

The above all statements are illegal. Because protectedA, publicA and getPrivateA() each have private accessibility status in the derived class B. However, fun1() of derived class B accesses getPrivateA(), protectedA and publicA. Let us again consider the statements as follows:

```
x=objc.protectedA;        //Won't work : protectedA is not accessible
x=objc.publicA;           //Valid
x=objc.getPrivateA();     // Valid
```

The above first statement is illegal but second and third statements are valid. It is so because in the first statement protected member protectedA of the base class A has protected visibility status in class C. However in the second and third statements both publicA and getPrivateA() have their public visibility status in class C, so they are accessible.

☞ Check Your Progress 1

1) What are the benefits of inheritance? Explain in brief.

……

……

……

2) When do we need inheritance?

……

……

……

3) Why do we need to study access specifiers?

……

……

……

4) What are the advantages of re-usability?

……

……

……

5) Why do we need re-use? Give an analogy to explain your answer.

……

……

……

1.4 TYPES OF INHERITANCE

In previous section, we discussed inheritance. In this section, we will discuss the types of inheritance. As we know, the derived class inherits some or all of the features from the base class depending on the visibility mode. A derived class can also inherit properties from more than one class or from more than one level. We can classify inheritance into the following type accordingly:

- **Single Inheritance:** Derivation of a class from only one base class is called a single inheritance.
- **Multiple Inheritance:** Derivation of a class from several (two or more) base classes is called multiple inheritance.
- **Multi-level Inheritance:** Derivation of a class from another derived class is called multilevel inheritance.
- **Hierarchical Inheritance:** Derivation of several classes from a single base class is called hierarchical inheritance.
- **Hybird Inheritance:** Derivation of a class involving more than one form of inheritance is called hybrid inheritance.
- **Multi-path Inheritance:** Derivation of a class from other derived classes, which are derived from the same base class is called multi-path inheritance.

In next sections, we shall discuss in detail single inheritance, multiple inheritance and multi-level inheritance.

1.5 SINGLE INHERITANCE

Now, let us discuss about single inheritance. In Single Inheritance, derived class inherits the feature of one base class. If a class is derived from one base class, it is called Single Inheritance.

Figure 1.1 depicts single inheritance:

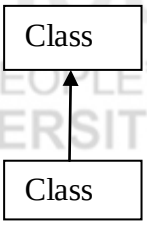


Figure 1.1: Single Inheritance

Class A is the base class and class B is the derived class. The following are the common steps to implement an inheritance. First, declare a base class and second declare a derived class. The syntax of single inheritance of the above figure is given as follows:

A derived class can be declared if its base class is already declared.

```
class A
{
// members of class A
};
class B :[public/private/protected] A
{
// members of class B
};
```

Let us understand the concept of single inheritance with Example1. In the example 1 given below, it is seen that how a single inheritance is implemented:

Example 1: Single Inheritance

```
#include <iostream.h>
class A
{
int a;
public :
int b;
void input_ab(void);
void output_a(void);
int get_a(void);
};
class B : public A
{
int c,d;
public :
void input_c(void);
void display(void);
void sum(void);
};
void A :: input_ab()
{
cout<< "\n Enter the value of a and b :"<<endl;
cin>>a>>b;
}
void A :: output_a()
{
cout<<"\\n The Value of a is :"<<a<<endl;
}
```

```
int A :: get_a()
{
return a;
}
void B :: input_c()
{
cout<< "\n Enter the value of c :"<< endl;
cin>>c;
}
void B :: sum()
{
d=get_a()+b+c;
}
void B :: display()
{
cout<< "\n The value of b is :"<<b<< endl;
cout<< "\n The value of c is :"<<c<< endl;
cout<< "\n The value of d(sum of a,b and c) is :"<<d<< endl;
}
void main()
{
B objb;

objb.input_ab();          //base class member function
objb.input_c();           //derived class member function
objb.output_a();          //base class member function
objb.sum();               //derived class member function
objb.display();           //derived class member function
objb.b=0;                 //objb.a would not work
objb.sum();               //derived class member function
objb.display();           //derived class member function
}
```

The output of the example1 is given below.

```
Enter the value of a and b:
10
20
Enter the value of c:
30
The value of a is : 10
The value of b is : 20
The value of c is : 30
The value of d(sum of a, b and c) is : 60
The value of a is : 10
The value of b is : 0
The value of c is : 30
The value of d(sum of a, b and c) is : 40
```

The above example shows a base class A and a derived class B. The base class A contains one private data member **a**, one public data member **b**, and three public member functions input_ab(), output_a() and get_a(). The class B contains two private data **c** and **d** and three public functions input_c(), display() and sum(). The class B is a derived publicly by class A. Therefore, B inherits all the public members (data and functions) of class A and retains their visibility. Hence, the public members of class A is also a public members of class B. But the private members of class A cannot be inherited by class B. Thus, the derived class B will have more members than what it contains at the time of declaration.

In the above example, we can see that the member functions sum() and display() are not able to access the private data member of class A because it cannot be inherited. However, the data member functions sum() and display() of derived class are able to access the private data of class A through an inherited member function get_a() of class A. In the main part of the program, we can also observe that the object of B can directly access the data member **b** of class A, because data member **b** is publically defined in A.

1.6 MULTIPLE INHERITANCE

Now, let us discuss the multiple inheritance. In multiple inheritance, derived class inherits features from more than one parent classes (base classes). In other way we can say that if a class is derived from more than one parent class (base classes), then it is called multiple inheritance. Figure 2.2 depicts multiple inheritance.

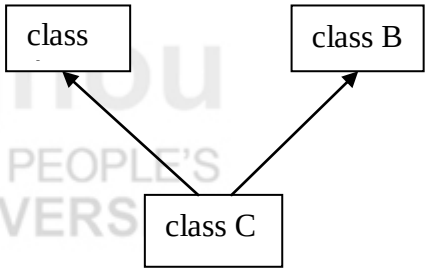


Figure 2.2: Multiple Inheritance

The syntax of declaration of Multiple Inheritance is given below:

```
class A
{
// members of class A
};

class B
{
// members of class B
};

class C :[public/private/protected] A, [public/private/protected] B
{
// members of class C
};
```

In the example 2 given below, it is seen that how multiple inheritance is implemented?

Example 2: Multiple Inheritance

```
#include <iostream.h>
class A
{
int a;
public :
void input_a(void);
void output_a(void);
int get_a(void);
};
class B
{
int b;
public :
void input_b(void);
void output_b(void);
int get_b(void);
};
class C : public A, public B
{
int c,d;
public :
void input_c(void);
void display(void);
void sum(void);
};
void A :: input_a()
{
cout<< "\n Enter the value of a :"<< endl;
cin>>a;
}
void A :: output_a()
{
cout<< "\n The value of a is :"<<a <<endl;
}
int A :: get_a()
{
return a;
}
void B :: input_b()
{
cout<< "\n Enter the value of b :"<< endl;
cin>>b;
}
void B :: output_b()
{
cout<< "\n The value of b is :"<<b<<endl;
}
int B :: get_b()
{
return b;
}
void C :: input_c()
```

```
{
cout<< "\n Enter the value of c :"<< endl;
cin>>c;
}
void C :: sum()
{
d=get_a()+get_b()+c;
}
void C ::display()
{
cout<< "\n The value of c is :"<<c<<endl;
cout<< "\n The value of d (sum of a, b and c) is :"<<d<<endl;
}
void main()
{
C objc;

objc.input_a();          //base class member function
objc.input_b();          //base class member function
objc.input_c();          //derived class member function
objc.output_a();         //base class member function
objc.output_b();         //base class member function
objc.sum();              //derived class member function
objc.display();          //derived class member function
}
```

The output of the example 2 is given below.

```
Enter the value of a:
10
Enter the value of b:
20
Enter the value of c:
30
The value of a is : 10
The value of b is : 20
The value of c is : 30
The value of d(sum of a, b and c) is : 60
```

The above example shows multiple inheritance. It contains three classes A, B and C. The class A and class B are parent classes (base classes) and class C is derived class. This class inherits the feature of class A and class B.

1.7 MULTI-LEVEL INHERITANCE

Now, let us discuss the Multi-level inheritance. In multi-level inheritance, the class inherits the feature of another derived class. If a class C is derived from class B which in turn is derived from class A and so on. It is called multi-level inheritance. Figure 1.3 depicts the multi-level inheritance.

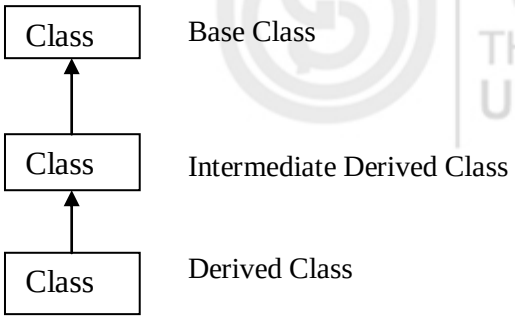


Figure 1.3: Multi-level Inheritance

The syntax of the multi-level inheritance of the above figure is given as follows:

```
class A
{
// members of class A
};

class B : [public/private/protected] A
{
// members of class B
};

class C :[public/private/protected] B
{
// members of class C
};
```

In example 3 given below, it is shown that how multilevel inheritance is implemented.

Example 3: Multi-level Inheritance

```
#include <iostream.h>
class A
{
int a;
public :
void input_a(void);
void output_a(void);
int get_a(void);
};
class B : public A
{
int b;
public :
void input_b(void);
void output_b(void);
int get_b(void);
};
class C : public B
{
int c,d;
public :
```

```
void input_c(void);
void display(void);
void sum(void);
};
void A :: input_a()
{
cout<< "\n Enter the value of a :"<< endl;
cin>>a;
}
void A :: output_a()
{
cout<< "\n The value of a is :"<<a<<endl;
}
int A :: get_a()
{
return a;
}
void B :: input_b()
{
cout<< "\n Enter the value of b :"<< endl;
cin>>b;
}
void B :: output_b()
{
cout<< "\n The Value of b is :"<<b<<endl;
}
int B :: get_b()
{
return b;
}
void C :: input_c()
{
cout<< "\n Enter the value of c :"<< endl;
cin>>c;
}
void C:: sum()
{
d=get_a()+get_b()+c;
}
void C ::display()
{
cout<< "\n The value of c is :"<<c<<endl;
cout<< "\n The value of d (sum of a, b and c) is :"<<d<<endl;
}
void main()
{
C objc;
objc.input_a();                    // member function of class A
objc.input_b();                    // member function of class B
objc.input_c();                    // member function of class C
objc.output_a();                   // member function of class A
objc.output_b();                   // member function of class B
objc.sum();                        // member function of class C
objc.display();                    // member function of class C
}
```

The output of the example3 is given below.

```
Enter the value of a:
10
Enter the value of b:
20
Enter the value of c:
30
The value of a is : 10
The value of b is : 20
The value of c is : 30
The value of d(sum of a, b and c) is : 60
```

The above example shows multi-level inheritance. It contains three classes A, B, and C. The class A is the base class. The class B is derived class. It inherits the features of A. The class C is derived from intermediate derived class B. The class C, after inheritance from A through B, would contain the following members:

```
private :
int c,d;            //own member
public :
void input_a(void);          // inherited from A via B
void output_a(void);         // inherited from A via B
int get_a(void);             // inherited from A via B

void input_b(void);          // inherited from B
void output_b(void);         // inherited from B
int get_b(void);             // inherited from B
void input_c(void);          // own
void display(void);          //own
void sum(void);              //own
```

☞ Check Your Progress 2

1) What are the different forms of inheritance supported by C++?

……

……

……

2) Write a interactive program in c++ which reads the two integer number a and b then performs the following operation:
   (i) a+b
   (ii) a-b
   (iii) a*b
   (iv) a/b

   Design the function a+b and a-b in a base class and design a*b and a/b in a derived class.

……

……

……

3) What is multi-level inheritance?

……

……

……

4) What are the ways to inherit properties of one class into another class?

……

……

……

5) What is containership? How does it differ from inheritance?

……

……

……

1.8 CONSTRUCTORS AND DESTRUCTORS IN DERIVED CLASSES

You have already learnt about the constructors and destructors in Unit 4 of Block 1 of this course. As we know, constructors and destructors play an important role in object initialization and remove the resources allocated to the object. Now, we will discuss as to how the constructors and destructors are used in derived classes.

The constructors are used to initialize object's data members and to allocate the required resources such as memory.

Constructors in Derived Classes:

Can you tell when and why we need a constructor in derived class? Well, the derived class need not have a constructor as long as base class has a no-argument constructor. However, if any base class contains a constructor with arguments (one or more), it is necessary for the derived class to have a constructor and pass the arguments to the base class constructors. In inheritance, generally derived class objects are created instead of the base class. Thus, it makes sense for the derived class to have a constructor and pass arguments to the constructor of the base class. When an object of a derived class is created, the constructor of the base class is executed first and later on the constructor of the derived class. Let us consider example 4 given below in which both base and derived class have constructor with parameter.

Example 4: Parametric constructors in Base and Derived classes

```
#include<iostream.h>
class A
{
private:
int a;
 protected:
 int b;
public:
A(int i, int j)
{
a=i;
b=j;
cout<< "A initialized"<<endl;
```

```
}
void display_ab(void)
{
cout<< "\nThe value of a is : "<<a;
cout<< "\nThe value of b is : "<<b;
}
int get_a(void)
{
return a;
}
};
class B
{
private:
int c;
protected:
int d;
public:
B(int i, int j)
{
c=i;
d=j;
cout<< "\nB initialized"<<endl;
}
void display_cd(void)
{
cout<< "\nThe value of c is : "<<c;
cout<< "\nThe value of d is : "<<d;
}
int get_c(void)
{
return c;
}
};

class C : public B, public A
{
int e,f, total;

public:
void C(int m, int n, int o, int p, int q, int r): A(m,n), B(o,p)
{
e=q;
f=r;
cout<< " \nC initialized";
}
void sum(void)
{

total=get_a()+b+get_c()+d+e+f;
}
void display(void)
{
cout<< "\nThe value of e is : "<<e;
cout<< "\nThe value of f is : "<<f;
cout<< "\nThe sum of a,b,c,d,e and f is : "<<total;
```

```
}
};
void main()
{
C objc(10,20,30,40,50,60);

objc.display_ab();
objc.display_cd();
objc.sum();
objc.display();
}
```

The output of the programme given above is:

```
B initialized
A initialized
C initialized
The value of a is :10
The value of b is :20
The value of c is :30
The value of d is :40
The value of e is :50
The value of f is :60
The sum of a,b,c,d,e,f is :210
```

The above example shows the three classes A, B and C. The class A have one parametric constructor, class B have also one parametric constructor and class C are derived class and inherits the features of class A and class B. The class C also have a parametric constructor. It is mandatory to have a parametric constructor in class C. Here you can observe that the class b is initialized first, albeit it appears second in the derived constructor because the class B has been declared first in the derived class header before the class A. You can also see that sum() member function of derived class C which is not able to use data members a and c of the base class A and B due to private members of their respective classes. However, it is able to receive b and d due to protected members of their respective classes. Table 1.2 depicts the order of execution of constructors:

Table 1.2: Order of Execution of Constructors

Method of Inheritance	Order of Execution
class B : public A {};	A() : base constructor B() : derived constructor
class C : public B, public A {};	B() : base constructor A() : base constructor C() : derived constructor
class C : public B, virtual A {};	A() : virtual base constructor B() : base constructor C() : derived constructor
class B : public A {}; class C : public B{};	A() : super base constructor B() : base constructor C() : derived constructor

Destructors in Derived Classes:

Unlike constructor in class hierarchy, destructors are invoked in the reverse order of the constructor invocation. Whenever object goes out of scope, the destructor of that

class, whose constructor was executed last while building object of that class, will be executed first. Let us take example 5 given below which shows the order of calling constructors and destructors in Inheritance:

Example 5: Order of calling of Constructors and Destructors in Inheritance

```
#include<iostream.h>
class A
{
protected:
int a,b;
public:
A(int i, int j)
{
a=i;
b=j;
cout<< "A initialized"<<endl;
}
~A()
{
cout<< "\Destructor in base class A"<<endl;
}
void display_ab()
{
cout<< "\nThe value of a is : "<<a;
cout<< "\nThe value of b is : "<<b;
}
};
class B
{
protected:
int c,d;
public:
B(int i, int j)
{
c=i;
d=j;
cout<< "\nB initialized"<<endl;
}
~B()
{
cout<< "\Destructor in base class B"<<endl;
}
void display_cd()
{
cout<< "\nThe value of c is : "<<c;
cout<< "\nThe value of d is : "<<d;
}
};
class C : public B, public A
{
Int e,f, total;

public:
C(int m, int n, int o, int p, int q, int r): A(m,n), B(o,p)
{
```

```
e=q;
f=r;
cout<< " \nC initialized";
}
~C()
{
cout<< "\Destructor in derived class C"<<endl;
}
void sum(void)
{
total=a+b+c+d+e+f;
}
void display(void)
{
cout<< "\nThe value of e is : "<<e;
cout<< "\nThe value of f is : "<<f;
cout<< "\nThe sum of a,b,c,d,e and f is : "<<total<<endl;
}
};
void main()
{
C objc(10,20,30,40,50,60);
objc.display_ab();
objc.display_cd();
objc.sum();
objc.display();
}
```

The output of the program given above is:

```
B initialized
A initialized
C initialized
The value of a is :10
The value of b is :20
The value of c is :30
The value of d is :40
The value of e is :50
The value of f is :60
The sum of a,b,c,d,e,f is :210
Destructor in derived class C
Destructor in base class A
Destructor in base class B
```

The above example shows the three classes A, B and C. The class A have one parametric constructor, class B have also one parametric constructor and C is derived class that inherits the feature of class A and class B. The class C also have a parametric constructor. It is mandatory to have a parametric constructor in class C. Here you can observe that the class B is initialized first, because the class B has been declared first in the derived class header before the class A. You can also see that the constructors are invoked in the order of B(), A() and C() whereas the destructors are invoked in the order of C(), A() and B() which is in reverse order.

☞Check Your Progress 3

1) Explain as to how ambiguity in member access is resolved.

……

……

……

2) What are base and derived classes? Create a base class called Stack and a derived class called MyStack. Write an interactive program to show the operations of stack.

……

……

……

3) Discuss the cost of inheritance.

……

……

……

4) Consider an example of declaring the examination result of BCA students of Indira Gandhi National Open University. Design three classes: Student, Exam, and Result. The Student class has data members such as those representing roll number, name, etc. Create the class Exam by inheriting Student class. The Exam class adds fields representing the marks scored in six subjects. Derive the Result from the Exam class, and it has its own fields such as total-marks. Write an interactive program to model this relationship.

……

……

……

1.9 SUMMARY

Inheritance is one of the prime features of Object Oriented programming language that helps to represent hierarchical relationship between classes. It is technique of building new classes from the existing classes. It facilitates code re-use and extensibility. It helps organize software components into categories and subcategories resulting in classification of software. Classification is the widely accepted use of inheritance of course other mechanisms may also be used for classification. Use of inheritance helps to generate software systems more quickly and easily using reusable components. The syntax of implementing inheritance through base and derived class is discussed. The concept of reusability, constructors and destructors in derived class are also discussed with examples. In this unit, we studied six different forms of inheritance: simple inheritance, multiple inheritance, multilevel inheritance, hybrid inheritance, hierarchical inheritance and multipath inheritance.

1.10 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) There are many benefits that can be derived from the proper use of inheritance. They are code reuse, ease of code maintenance and extension, and reduction in the time to market. The following situations explain benefits of inheritance:

- When inherited from another class, the code that provides a behavior required in the derived class does not need to be rewritten.
- Code sharing can occur at several levels.
- When multiple classes inherit from the same super class, there is a sufficient guarantee that the behavior they inherit will be same in all cases.

2) Inheritance is suitable where the following situation arises:

- Whenever there are similarities between two or more classes, you can apply inheritance.
- If a new class to be defined has certain features in addition to the features of an existing class, you can use inheritance and code only the additional features of this new class.
- If the relationship between the classes is Is_a, Is_a_Kind_Of, or Is_Link or is Type_Of, you can select inheritance.
- The hierarchical relationship creates a relationship tree with specialized type branching off from more generalized types. Inheritance is advisable when generalization is fixed and does not require any modification or change.
- The most common use of inheritance is for specialization.

3) Access specifiers are used to control the accessibility of data members and member functions of class. It helps classes to prevent unwanted exposure of members (data and functions) to outside world.

4) There are many advantages of re-usability. They can be applied to reduce cost, effort and time of software development. It also increases the productivity, portability and reliability of the software product.

5) Because of the high development costs, software should be reusable. If many millions of dollars are to be spent to develop a software system, it makes sense to make its component flexible so they can be re-used when developing a new software system. Re-use can improve reliability, reduce development costs, and improve maintainability. Let us take an analogy of Automobile industry to understand the need of re-usability. Consider the design and creation of a new car model. The automotive engineer does not design a new car from scratch. Rather, the engineer borrows from the design of existing cars. For example, the engine design from an existing car may be used in a new model. If the engine design has been used in a previous model, design problems have likely been resolved. Thus development costs are reduced because a new engine does not need to be designed and tested. Finally, consumer maintenance costs are reduced because machines and

others who must maintain the car are already familiar with the operation of the engine.

Check Your Progress 2

1) There are six forms of inheritance supported by c+ +, namely
   (i) Single Inheritance
   (ii) Multiple Inheritance
   (iii) Multi-level Inheritance
   (iv) Hybrid Inheritance
   (v) Multi-path Inheritance
   (vi) Hierarchical Inheritance

2)

```
#include <iostream.h>
    #include<stdlib.h>
    class A
    {
    int a,b;
    public :
    void input_ab(void);
    int get_a(void);
    int get_b(void);
    int add(void);
    int sub(void);
    };
    class B : public A
    {
    public :
    int mul(void);
    int div(void);
    void display(int opt, int res);
    };
    void A :: input_ab()
    {
    cout<< "\n Enter the value of a and b :"<<endl;
    cin>>a>>b;
    }
    int A :: get_a()
    {
    return a;
    }
    int A :: get_b()
    {
    return b;
    }
    int A :: Add()
    {
    return (a+b);
    }
    int A :: Sub()
    {
    return (a-b);
    }

    void B :: Mul()
    {
```

```
return(get_a()*get_b());
}
void B :: Div()
{
return(get_a()/get_b());
}
void B :: display(int choice, int result)
{
cout<< "\n The value of a is :"<<a<< endl;
cout<< "\n The value of b is :"<<b<< "endl;
switch(choice)
{
case 1 : cout << "\n The sum of a and b is :" <<result<<endl;
 break;
case 2 : cout << "\n The subtraction of a and b is :" <<result<<endl;
 break;
case 3 : cout << "\n The multiplication of a and b is :" <<result<<endl;
 break;
case 4 : cout << "\n The division of a and b is :" <<result<<endl;
 break;
}
}
void main()
{
B objb;
int choice;
int result;
objb.input_ab();
while (1)
{
cout <<" Operations on two numbers …"<<endl;
cout <<" 1. Addition"<<endl;
cout <<" 2. Subtraction"<<endl;
cout <<" 3. Multiplication"<<endl;
cout <<" 4. Division"<<endl;
cout <<" 5. Quit"<<endl;
cout <<" Enter choice:"<<endl;
cin>>choice;
switch(choice)
{
case 1 : result=objb.add();
objb.display(choice,result);
 break;
case 2 : result=objb.sub();
objb.display(choice,result);
 break;
case 3 : result=objb.mul();
objb.display(choice,result);
 break;
case 4 : if (b!=0)
{
result=objb.div();
objb.display(choice,result);
}
else
cout<< "\nDivide by zero error:"<<endl;
 break;
```

```
case 5 : exit(1);
 break;
default :
cout <<" Bad option selected"<<endl;
continue;
}
}
}
```

3) Derivation of a class from another derived class is called multi-level inheritance. The multi-level inheritance mechanism can be extended to any levels.

4) There are two ways to inherit properties of one class into another class as follows:
(i) Inheritance
(ii) Object Composition

5) The use of objects in a class as data members is referred to as object composition. Thus, we can say that an object can be collection of many other objects. This relationship is called has-a relationship or containership. This relationship is also called nesting of objects. In many situations, inheritance and containership relationships can serve the same purpose. Containership does not provide flexibility of ownership. Inheritance relationship is simpler to implement and offers a clearer conceptual framework.

Check Your Progress 3

1) Ambiguity is a problem that surfaces in certain situations involving multiple inheritance. Consider the following cases:

- Base classes having functions with the same name.
- The class derived from these base classes is not having a function with the name as those of its base classes.
- Members of a derived class or its objects referring to a member whose name is the same as those of base classes.

These situations create ambiguity in deciding which of the base class's function has to be referred. This problem is resolved by using the scope resolution operator which is given as follows:

ObjectName.BaseClassName::MemberName(...).

2) Inheritance is a property by which one class inherits the feature of another class. The class which inherits the feature from another class is called derived class and class from which another class takes feature is called base class.

```
#include <iostream.h>
#include<stdlib.h>
#define Max_Size 5          //Maximum stack size
class Stack
{
protected :
int stack[Max_Size];
int top;
public :
Stack (void)
```

```
void push (int item);
void pop (int &item);
};
class MyStack : public Stack
{
public :
int push(int item);
int pop(int &item);
void stackContent(void);
};
Stack::Stack()
{
top=-1;              //Stack empty
}
void Stack::push(int item)
{
top++;
stack[top]=item;
}
void Stack::pop(int &item)
{
item=stack[top];
top--;
}
int MyStack :: push(int item)
{
If (top<Max_Size-1)
{
Stack::push(item);
return 1;                              //push operation successful
}
cout<< "\n Stack Overflow :"<< endl;
return 0;
}
int MyStack :: pop(int &item)
{
If (top>=0)
{
Stack::pop(item);
return 1;                              //push operation successful
}
cout<< "\n Stack Underflow :"<< endl;
return 0;
}
void MyStack :: stackContent(void)
{
int stop;
stop=top;
for (int i=0; i<=stop;i++)
cout<< ":"<<stack[i];
}
void main()
{
MyStack stack;
int choice;
int item;
while (1)
```

```
{
cout << "\nStack Operation …"<<endl;
cout << "\n1. Item to push?"<<endl;
cout << "2. Item to pop"<<endl;
cout << "3. Quit"<<endl;
cout << "Enter choice:"<<endl;
cin>>choice;
switch(choice)
{
case 1 :cout<< "Enter the item:"<<endl;
cin>>item;
cout<< "\n Stack content before push operation:";
stack.stackContent();
if ((stack.push(item))==1)
{
cout<< "\n Stack content after push operation:";
stack.stackContent();
}
 break;
case 2 : cout<< "\n Stack content before pop operation:";
stack.stackContent();
if ((stack.pop(item))==1)
{
cout<< "\n Stack content after pop operation:";
stack.stackContent();
cout<< "popped item:" <<item;
}
 break;
case 3 : exit(1);
 break;
default :
cout <<" Bad option selected"<<endl;
continue;
}
}
}
```

3) Despite the advantages of inheritance, it incurs compiler overhead. In inheritance relationship, there are certain members in the base class that are not at all used; however, data space is allocated to them. This necessitates the need for specialized inheritance which is complex to develop. The following are some of the perceived costs of inheritance:

- Inherited methods, which must be prepared to deal with arbitrary subclasses, are often slower than specialized codes.
- Message passing by its very nature is a more costly than the invocation of simple procedures.
- Albeit object oriented programming is often touted as a solution to the problem of software complexity, overuse or improper use of inheritance can simply transfer one form of complexity to another form.

4)

```
#include<iostream.h>
    #include<string.h>
    class Student
    {
    int roll_no;
    char name[25];
    public:
    void ReadStudentData(void);
    void DisplayStudentData(void);
    };
    class Exam :public Student
    {
    protected:
    int marks[6];
    public :
    void ReadExamMarks(void);
    void DisplayExamMarks(void);
    };
    class Result : public Exam
    {
    int total_marks;
    public :
    void Display(void);
    };
    void Student :: ReadStudentData()
    {
    cout<<"\n Enetr the Name:"<<endl;
    cin>>name;
    cout<< "\n Enter the Roll No.:"<<endl;
    cin>>roll_no;
    }
    void Student :: DisplayStudentData()
    {
    cout<<"\n Name :"<<name<<endl;
    cout<<"\n Roll No. :"<<roll_no;
    }
    void Exam::ReadExamMarks()
    {
    cout <<"\nEnter Marks :"<<endl;
    for (int i=0; i<6; i++)
    {
    cout<<"\n Marks scored in subject"<<i+1<<"<Max:100>"<<endl;
    cin>>marks[i];
    }
    }
    void Exam::DisplayExamMarks()
    {
    for (int i=0; i<6; i++)
    cout<<"\n Marks scored in subject"<<i+1<<":"<<marks[i];
    }
    void Result::Display()
    {
total_marks=0;
for (int i=0; i<6; i++)
total_marks=total_marks+marks[i];
cout<<"\n Total Marks scored in six subjects :"<<total_marks;
}
```

```
void main()
{
Result objr;
objr.ReadStudentData();
objr.ReadExamMarks();
objr.DisplayExamMarks();
objr.Display();
}
```

1.11 FURTHER READINGS

1) B. Stroustrup, *The C++ Programming Language*, Third Edition, Pearson/Addison-wesley Publication, 1997.

2) K. R. Venu Gopal, Raj Kumar Buyya, T Ravishankar, *Mastering C++*, Tata-McGraw-Hill Publishing Company Limited, New Delhi, 2004.

3) E. Balagurusamy, *Object Oriented Programming with C++*, Tata Mc-Graw-Hill Publishing Company Limited, New Delhi, 2001.

4) N. Barkakati, *Object Oriented Programming in C++*, Prentice-Hall of India.

UNIT 2 OPERATOR OVERLOADING

Structure **Page Nos.**

2.0 INTRODUCTION

When you create an object (a variable), you give a name to a region of storage. A function is a name for an action. By making up names to describe the system at hand, you create a program that is easier for people to understand and change. Function overloading is one of the defining aspects of the C++ programming language. Not only does it provide support for compile time polymorphism, it also adds flexibility and convenience. In addition, function overloading means that if you have two libraries that contain functions of the same name, they won't conflict as long as the argument lists are different. We'll look at all these factors in detail across this unit.

In C++, you can overload most operators so that they perform special operations relative to classes that you create. When an operator is overloaded, none of its original meanings are lost. Operator overloading allows the full integration of new class type into programming environment. After overloading the appropriate operators, you can use objects in expressions in just the same way that you use C++'s built-in data types. Operator overloading also forms the basis of C++'s approach to I/O. Function overloading and operator overloading really aren't very complicated. By the time you reach the end of this unit, you shall learn when to use them and also underlying mechanisms that implement them during compiling and linking.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- explain the need of overloading;
- describe mechanism of function overloading;
- learn the use and application of default argument;

- understand how to overload (redefine) operators to work with new types;
- understand how to convert objects from one class to another;
- learn when to, and when not to, overload operators; and
- learn about several cases of overloaded operators.

2.2 FUNCTION OVERLOADING

Function overloading refers to using the same thing for different purposes. C++ permits the use of different functions with the same name. However such functions essentially have different argument list. The difference can be in terms of number or type of arguments or both. These functions can perform a variety of different tasks. This process of using two or more functions with the same name but differing in the signature is called function overloading. It is only through these differences that the compiler knows which function to call in any given situations.

Overloading of functions with different return types are not allowed.

Consider the following function:

```
int add (int a, int b)
{
return a + b;
}
```

This trivial function adds two integers. However, what if we also need to add two floating point numbers? This function is not at all suitable, as any floating point parameters would be converted to integers, causing the floating point arguments to lose their fractional values.

One way to work around this issue is to define multiple functions with slightly different names:

```
int add (int a, int b)
{
return a + b;
}
double addition (double p, double q)
{
return p + q;
}
```

However, for best effect, this requires that you define a consistent naming standard, remember the name of all the different flavors of the function, and call the correct one (calling add() for addition of double type numbers with integer parameters may produce the wrong result due to precision issues).

Function overloading provides a better solution. Using function overloading, we can declare another add () function that takes double parameters:

double add (double p, double q)

We now have two version of add ():

```
{
return p + q;
}
```

int add (int A, int B); // integer version

double add (double P, double Q); // floating point version

Which version of add () gets called depends on the arguments used in the call — if we provide two ints, C++ will know we mean to call add(int, int). If we provide two floating point numbers, C++ will know we mean to call add (double, double). In fact, we can define as many overloaded add () functions as we want, so long as each add () function has unique parameters.

Consequently, it's also possible to define add () functions with a differing number of parameters:

```
int add (int a, int b, int c)
{
return a + b + c;
}
```

Even though this add () function has 3 parameters instead of 2, because the parameters are different than any other version of add (), this is valid.

Function overloading is one of the most powerful features of C++ programming language. It forms the basis of polymorphism (compile-time polymorphism). Most of the time you'll be overloading the constructor function of a class.

2.2.1 How Function Overloading is Achieved

One thing that might be coming to your mind is, how will the compiler know when to call which function, if there are more than one function of the same name. The answer is, you have to declare functions in such a way that they differ either in terms of the number of parameters or in terms of the type of parameters they take. What that means is, nothing special needs to be done, you just need to declare two or more functions having the same name but either having different number of parameters or having parameters of different types. In overloaded functions, the function call determines which function definition will be executed. The biggest advantage of overloading is that it helps us to perform same operations on different datatypes without having the need to use separate names for each version. For example, an overloaded test() function handles different types of data as shown below:

```
// Declarations
int test (int x);                    //prototype 1
int test (int x, int y);             //prototype 2
double test (int a, double b);       //prototype 3
// Function call
Cout<< test (23);                    //uses prototype 1
Cout<< test (45, 55);                //uses prototype 2
Cout<< test (12, 3.25);          //uses prototype 3
```

Following example illustrates the function overloading:

```
// Example: Function overloading to find the absolute value of any number int, long, float,
double
#include<iostream>
using namespace std;
int abslt(int );
long abslt(long );
float abslt(float );
double abslt(double );
int main()
{
int intgr=-5;
long lng=34225;
float flt=-5.56;
double dbl=-45.6768;
cout<<\" absoulte value of \"<<intgr<<\" = \"<<abslt(intgr)<<endl;
cout<<\" absoulte value of \"<<lng<<\" = \"<<abslt(lng)<<endl;
cout<<\" absoulte value of \"<<flt<<\" = \"<<abslt(flt)<<endl;
cout<<\" absoulte value of \"<<dbl<<\" = \"<<abslt(dbl)<<endl;
}
int abslt(int num)
{
if(num>=0)
return num;
else
 return (-num);
}
long abslt(long num)
{
if(num>=0)
return num;
else return (-num);
}
float abslt(float  num)
{
if(num>=0)
return num;
else return (-num);
}
double abslt(double  num)
{
if(num>=0)
return num;
else return (-num);
}
```

Output

- absoulte value of -5 = 5
- absoulte value of 34225 = 34225
- absoulte value of -5.56 = 5.56
- absoulte value of -45.6768 = 45.6768

The use of overloading may not have reduced the code complexity /size but has definitely made it easier to understand and avoided the necessity of remembering different names for each version function which perform identically the same task.

Example: overloading functions that differ in terms of number of parameters

```
#include<iostream.h>
// function prototype
int func(int i);
int func(int i, int j);
void main(void)
  {
   cout<<func(15);                 //func(int i)is called

   cout<<func(15,15);              //func(int i, int j) is called
  }
  int func(int i)
  {
   return i;
  }
  int func(int i, int j)
  {
   return i+j;
  }
```

Example: overloading functions that differ in terms of types of parameters

```
#include<iostream.h>
 //function prototypes
  int func(int i);
  double func(double i);
  void main(void)
  {
   cout<<func(15);                 //func(int i) is called
   cout<<func(15, 155);            //func(double i) is called
  }
  int func(int i)
  {
   return i;
  }
double func(double i)
  {
   return i;
  }
```

2.2.2 How Function Calls are Matched with Overloaded Functions

Making a call to an overloaded function results in one of three possible outcomes:

1) A match is found. The call is resolved to a particular overloaded function.
2) No match is found. The arguments can not be matched to any overloaded function.
3) An ambiguous match is found. The arguments matched more than one overloaded function.

When an overloaded function is called, C++ goes through the following process to determine which version of the function will be called:

1) First, C++ tries to find an exact match. This is the case where the actual argument exactly matches the parameter type of one of the overloaded functions. For example:

```
void Print(char *szValue);
void Print(int nValue);
Print(10); // exact match with Print(int)
```

Although 10 could technically match Print(char*), it exactly matches Print(int). Thus Print(int) is the best match available.

2) If no exact match is found, C++ tries to find a match through promotion. In the lesson on type conversion and casting, we covered how certain types can be automatically promoted via internal type conversion to other types.

To summarize,

a) Char, unsigned char, and short is promoted to an int.
b) Unsigned short can be promoted to int or unsigned int, depending on the size of an int
c) Float is promoted to double
d) Enum is promoted to int

For example:

```
void Print(char *szValue);
void Print(int nValue);
Print('a'); // promoted to match Print(int)
```

In this case, because there is no Print(char), the char ‘a’ is promoted to an integer, which then matches Print(int).

3) If no promotion is found, C++ tries to find a match through standard conversion. Standard conversions include:

   a) Any numeric type will match any other numeric type, including unsigned (eg. int to float)
   b) Enum will match the formal type of a numeric type (eg. enum to float)
   c) Zero will match a pointer type and numeric type (eg. 0 to char*, or 0 to float)
   d) A pointer will match a void pointer

For example:

```
void Print(float fValue);
void Print(struct sValue);
Print('a'); // promoted to match Print(float)
```

In this case, because there is no Print(char), and no Print(int), the 'a' is converted to a float and matched with Print(float).

Note that all standard conversions are considered equal. No standard conversion is considered better than any of the others.

4) Finally, C++ tries to find a match through user-defined conversion. Although we have not covered classes yet, classes (which are similar to structs) can define conversions to other types that can be implicitly applied to objects of that class.

For example, we might define a class X and a user-defined conversion to int.

```
class X; // with user-defined conversion to int
void Print(float fValue);
void Print(int nValue);
X cValue; // declare a variable named cValue of type class X
Print(cValue); // cValue will be converted to an int and matched to Print(int)
```

Although cValue is of type class X, because this particular class has a user-defined conversion to int, the function call Print(cValue) will resolve to the Print(int) version of the function.

2.3 FUNCTION OVERLOADING AND AMBIGUITY

You can create a situation in which the compiler is unable to choose between two (or more) overloaded functions. When this happens, the situation is said to be ambiguous. Ambiguous statements are errors, and programs containing ambiguity will not compile.

2.3.1 Ambiguous Matches

If every overloaded function has to have unique parameters, how is it possible that a call could result in more than one match? Because all standard conversions are considered equal, and all user-defined conversions are considered equal, if a function call matches multiple candidates via standard conversion or user-defined conversion, an ambiguous match will result.

For example:

```
void Print(unsigned int nValue);
void Print(float fValue);
Print('p');
Print(10);
Print(1.14);
```

In the case of Print('p'), C++ can not find an exact match. It tries promoting 'a' to an int, but there is no Print (int) either. Using a standard conversion, it can convert 'a' to both an unsigned int and a floating point value. Because all standard conversions are considered equal, this is an ambiguous match.

Print(10) is similar. 10 is an int, and there is no Print(int). It matches both calls via standard conversion.

Print(1.14) might be a little surprising, as most programmers would assume it matches Print(float). But remember that all literal floating point values are doubles unless they have the 'f' suffix. 1.14 is a double, and there is no Print (double). Consequently, it matches both calls via standard conversion.

Ambiguous matches are considered a compile-time error. Consequently, an ambiguous match needs to be disambiguated before your program will compile. There are two ways to resolve ambiguous matches:

1) Often, the best way is simply to define a new overloaded function that takes parameters of exactly the type you are trying to call the function with. Then C++ will be able to find an exact match for the function call.

2) Alternatively, explicitly cast the ambiguous parameter(s) to the type of the function you want to call. For example, to have Print(10) call the Print(unsigned int),

2.4 MULTIPLE ARGUMENTS

If there are multiple arguments, C++ applies the matching rules to each argument in turn. The function chosen is the one for which each argument matches at least as well as all the other functions, with at least one argument matching better than all the other functions. In other words, the function chosen must provide a better match than all the other candidate functions for at least one parameter, and no worse for all of the other parameters.

In the case that such a function is found, it is clearly and unambiguously the best choice. If no such function can be found, the call will be considered ambiguous.

☞ **Check Your Progress 1**

1) What is function overloading?

……

……

……

2) How function calls are matched with overloaded functions?

……

……

……

3) What are the main applications of function overloading?

……

……

……

4) Multiple choice questions:

i) In C++, dynamic memory allocation is accomplished with the operator ____

a) new

b) this

c) malloc( )

d) delete

ii) The operator that cannot be overloaded is

a) ++

b) ::

c) ( )

d) ~

iii) The operator << when overloaded in a class

a) must be a member function

b) must be a non member function

c) can be both (A) & (B) above

d) cannot be overloaded

iv) Identify the operator that is NOT used with pointers

a) ->

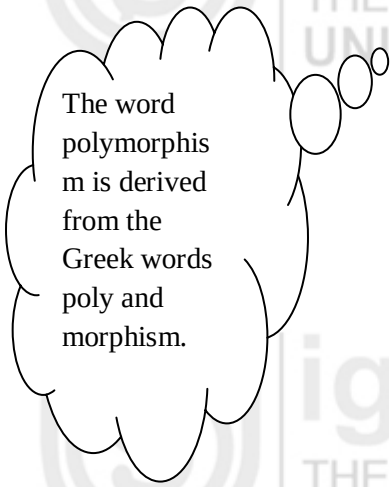
b) &

c) *

d) >>

2.5 OPERATOR OVERLOADING

We already know that a function can be overloaded (same function name having multiple bodies). The concept of overloading a function can be applied to operators as well. For example, in C++ we can multiply two variables of user-defined data type with the same syntax that is applied to the basic data type. This means that C++ has the ability to provide the operators with a special meaning for data type. The



mechanism which provides this special meaning to operators is called operator overloading. The operator overloading feature of C++ is one of the methods of realizing polymorphism. Here, poly refers to many or multiple and morphism refers to actions, i.e. performing many actions with a single operator. Thus operator overloading enables us to make the standard operators, like +, -, * etc, to work with the objects of our own data types. So what we do is, write a function which redefines a particular operator so that it performs a specific operation when it is used with the object of a class. Operator overloading is very exciting feature of C++. The concept of operator overloading can also be applied to data conversion. It enhances the power of extensibility of C++. Thus operator overloading concepts are applied to the following two principle areas:

- Extending capability of operators to operate on user defined data, and
- Data conversion

This session deals with overloading of operators to make Abstract Data Types (ADTS) more natural, and closer to fundamental data types.

2.5.1 Why to Overload Operators

Most fundamental data types have pre-defined operators associated with them. For example, the C++ data type float, together with the operators +, -, *, and /, provides an implementation of the mathematical concepts of an integer. This is purely a convenience to the user of a class. Operator overloading isn't strictly necessary unless other classes or functions expect operators to be defined.

To make a user-defined data type as natural as a fundamental data type, the user-defined data type must be associated with the appropriate set of operators. Operators are defined as either member functions or friend functions. The purpose of operator overloading is to make programs clearer by using conventional meanings for ==, [], +, etc. Operators can be overloaded in any way from those available like globally or on the basis of class by class. While implementing the operator overloading this can be achieved by implementing them as functions. Whether it improves program readability or causes confusion depends on how well you use it. In any case, C++ programmers are expected to be able to use it.

The user can understand the operator notation more easily as compared to a function call because it is closer to the real-life implementation. Thus, by associating a set of meaningful operators, manipulation of an ADT can be done in a conventional and simpler form. Associating operators with an ADT involves overloading them. Although the semantics of any operator can be extend, we can't change its syntax, associativity, precedence etc.

2.5.2 Member vs. Non-member Operators

There are two groups of operators in terms of how they are implemented: member operators and non-member operators. The distinction between the two is the same as it is with methods and functions.

Member operators are operators that are implemented as member functions (methods) of a class.

Non-member operators are operators that are implemented as regular, non-member functions.

Some operators are required to be member operators, others must be non-member operators, and some can be both.

Note: Operator when overloaded is called operator function. Operator function is declared with operator keyword that precedes the operator that has to be overloaded. Again overloaded operator and functions are not the same, but the same way as overloaded functions are distinguished by the number and type of operands, the mechanism is applicable to the operators.

2.5.3 General rules for Operator Overloading

The following rules constrain how overloaded operators are implemented. However, they do not apply to the new and delete operators.

(i) You cannot define new operators, such as **.

(ii) You cannot redefine the meaning of operators when applied to built-in data types.

(ii) Overloaded operators must either be a non static class member function or a global function.

(iii) Obey the precedence, grouping, and number of operands dictated by their typical use with built-in types. Therefore, there is no way to express the concept "add 2 and 3 to an object of type Point," expecting 2 to be added to the x coordinate and 3 to be added to the y coordinate.

(iv) Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.

(v) Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.

(vi) If an operator can be used as either a unary or a binary operator (&, *, +, and -), you can overload each use separately.

(viii) Overloaded operators cannot have default arguments.

(ix) All overloaded operators except assignment (operator=) are inherited by derived classes.

(x) The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked (the class in which the operator is declared, or a class derived from that class). No conversions are supplied for the first argument.

(xi) Overloading + doesn't overload +=, and similarly for the other extended assignment operators.

(xii) You may not redefine ::, sizeof, ?:, or . (dot).

(xiii) =, [], and -> must be member functions if they are overloaded.

(xiv) ++ and -- need special treatment because they are prefix and postfix operators.

(xv) There are special issues with overloading assignment (=). Assignment should always be overloaded if an object dynamically allocates memory.

Table 2.1 shows the operators which can be overloaded

Table 2.1: List of Operators that can be overloaded

+	-	*	/	%	^	&	|	~	!
=	<	>	+=	-=	*=	/=	%=	^=	&=
|=	<<	>>	<<=	>>=	==	!=	<=	>=	&&
||	++	--	,	->*	->	( )	[ ]	new	delete
new[]	delete[]								

Following list shows the operators which can't be overloaded

.	.*	::	sizeof	?:

2.5.4 Why Some Operators can't be Overload

Generally the operators that can't be overloaded are like that because overloading them could and probably would cause serious program errors or it is syntactically not possible. Following section describe the reason for not overloading some of the operators defined in C++ language.

a) **:: (scope resolution)), . (member selection), and .* (member selection through pointer to function)**

For the operators that cannot be overloaded like :: (scope resolution), . (member selection), and .* (member selection through pointer to function), we are quoting from Stroustrup's 3rd edition of 'The C++ Programming Language', section 11.2 (page 263), these operators 'take a name, rather than a value, as their second operand and provide the primary means of referring to members. C++ has no syntax for writing code that works on names rather than values so syntactically these operators can not be overload.

The right hand side of operators **.** (member selection/access) , .* (member selection through pointer to function) and :: (scope resolution) are names of things, e.g. name of a class member. In other words: above three unloaded operators use name instead of operand, so we can`t pass any name (either of variable, class) to any function. We must have to pass the operand for that.

b) **size of operator**

The size of operator returns the size of the object or type passed as an operand. It is evaluated by the compiler not at runtime so you can not overload it with your own runtime code. It is syntactically not possible to do.

c) **? : (conditional operator)**

All operators that can be overloaded must have at least one argument that is a user-defined type. That means you can't overload that operator which has no arguments.

But it does not suite for ? : (conditional operator) as it does not take name as parameter. The reason we cannot overload ? : is that it takes 3 argument rather than 2 or 1. There is no mechanism available by which we can pass 3 parameters during operator overloading.

2.6 DEFINING OPERATOR OVERLOADING

You can overload or redefine the most of built-in operators in C++. These operators can be overloaded globally or on a class-by-class basis. Overloaded operators are implemented as functions and can be member functions or global functions.
An overloaded operator is called an operator function. You declare an operator function with the keyword operator preceding the operator. Overloaded operators are distinct from overloaded functions, but like overloaded functions, they are distinguished by the number and types of operands used with the operator.

2.6.1 Syntax

Defining an overloaded operator is like defining a function, but the name of that function is operator #, in which # represents the operator that's being overloaded. The number of arguments in the overloaded operator's argument list depends on two factors:

1. Whether it's a unary operator (one argument) or a binary operator (two arguments).
2. Whether the operator is defined as a global function (one argument for unary, two for binary) or a member function (zero arguments for unary, one for binary – the object becomes the left-hand argument).

It may look like following way:

```
Return_type  class_name :: operator op (op_argument_list)
{
Body of function
}
```

Here return_type is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator. Operator op is the name of function. They may be also friend functions. Member function has no argument for unary operator and one argument for binary operator. This is because the object used to invoke the member function is passed implicitly and so it available to member function. This case is not with the friend function. Friend function will have one argument for unary operator and two arguments for binary operator. All the arguments may be passed either by value or by reference. Following lines define the steps in overloading the operators

a. Build a class that defines the data type that is going to use in operation of overloading.
b. Declare the operator function *operator op()* in public area of class.
c. Now define the operator function to implement the required operations.

Invoking Operator Function

(i) For member Function

a. For unary operator: **op m** or **m op**
b. For binary operator: **m op n** or **m.operator op (n)**

(ii) For friend Function

a. For unary operator: **operator op (m)**
b. For binary operator: **operator op (m, n)**

Overloading unary operators

To declare a unary operator function as a non static member, you must declare it in the form:
return_type **operator** *op***();**
where *op* is one of the operators listed in the preceding table.
To declare a unary operator function as a global function, you must declare it in the form:
return_type **operator** *op***(** *arg* **);**
Where *op* is described for member operator functions and the *arg* is an argument of class type on which to operate. An overloaded unary operator may return any type.

Some examples of operator overloading:

Unary operator overloading
Example:

```
#include<iostream.h>
#include<conio.h>
Class complex
{
int real, imaginary;
Public:
complex()
{
}
complex(int a, int b)
{
real = a;
imaginary = b;
}
void operator-();
void display()
{
cout<<"Real value"<<real<<endl;
cout<<"imaginary value"<<imaginary<<endl;
}
};
void complex::operator-()
{
real = -real;
imaginary = -imaginary;
}
void main()
{
Clrscr();
complex c1(10,12);
cout<< "real and imaginary value befor operation"<<endl;
c1.display();
c1;        //c1- /*It will give error*/
cout<< "real and imaginary' value after operation"<<endl;
c1.display();
getch();
}
```

Output

Real and imaginary value before operation
10 20
Real and imaginary value after operation
-10 -20

Explanation: In the above example operator, overloading is of unary operator declared in class complex. Outline function make changes values of real and imaginary part by negation. When this is called, it has to be remembered that operand must precede operator otherwise there will be an error. The reason is simple, 'operator' is the function name that is called with operand.

Unary operator overloading using friend function

Example:

```
#include<iostream.h>
#include<conio.h>
class complex
{
int real;
int imaginary;
public:
  complex()                    //default constructor
{
}
complex(int a, int b)
{
        real = a;
        imaginary = b;
}
friend void operator -(); //operator overloading prototype
void display()
 {
cout<<"real value is"<<real<<endl;
cout <<"imaginary value is:"<<imaginary<<endl;
 }
};
//definition of operator overloading function
friend void complex :: operator - (complex &c)
{
        c.real = - c.real;
        c.imaginary = - c.imaginary;
}
void main()
{
clrscr();
complex c1(50,100);
cout<<"real and imaginary value before operation"<<endl;
c1.diaplay();     //calling operator overloading function
-c1;
//c1-;              It will give you an error.
cout<<"real and imaginary value after operator"<<endl;
 c1.display();
 getch();
}
```

Output:

Real and imaginary value before operation

real value is 50 and imaginary value 100

real and imaginary value after operation

real value is -50 and imaginary value -100

2.7.2 Binary Operator Overloading

Example :

```
#include<iostream.h>
#include<conio.h>
class complex
{
int real, imaginary;
public:
complex()
{
}
complex(int a,int b)
{
real = a;
imaginary = b;
}
void operator +(complex c);
};
void complex::operator+(complex c)
{
complex temp;
temp.real = real + c.real;
temp.imaginary = imaginary + c.imaginary;
cout<<"real sum is"<<temp.real<<endl;

}
void main()
{
clrscr();
complex c1(10,20);
complex c2(20,30);
c1+c2;
getch();
}
```

Output

Real sum is 30

Imaginary sum is 50

Explanation: Output is easily predictable, and most of the program is same as above program. Difference lies in void complex :: operator +(complex c) if we compare this statement with our conventional outline statement

<return type > <classname>::<function name> <argument list>

We find that return type is void. Class name is complex. Function name is 'operator+' and it takes one argument which is of class complex type object.

When this function is called in main with statement c1+c2, c1 is object that invokes function 'Operator+' and c2 is passed as argument. Now with in the function real and imaginary parts are c1 will be directly accessible (as it invokes function) while real and imaginary of c2 are taken using formal argument 'complex c'. Here we can find that keyword 'operator' is inserted automatically whenever invoking object is user defined type with operator.

Binary operator overloading using friend function

Example

```
#include<iostream.h>
#include<conio.h>
class complex
{
int real;
int imaginary;
public:
  complex(){ }                    //default constructor
complex(int a, int b)
{
        real = a;
        imaginary = b;
}
friend complex operator +(comlex c1, complex c2);
void display()
 {
cout<<"real value is"<<real<<endl;
cout <<"imaginary value is:"<<imaginary<<endl;
 }
};
complex  operator + (comlex c1, complex c2)
{
        complex tmp;
        tmp.real =  c1.real + c2.real
        tmp.imaginary =  c1.imaginary + c2.imaginary;
        return(tmp);
}
void main()
{
clrscr();
complex c1(10,20);
complex c1(30,50);
ccomlex c3 = c1 + c2;
c3.display();
 getch();
}
```

Output:

real value is : 40

imaginary value is : 70

Unary Operators (Increment and decrement)

The overloaded ++ and – – operators present a dilemma because you want to be able to call different functions depending on whether they appear before (prefix) or after (postfix) the object they're acting upon. The solution is simple, but people sometimes find it a bit complex and confusing at first. When the compiler sees, for example, ++a (a pre-increment), it generates a call to operator ++(a); but when it sees a++, it generates a call to operator ++(a, int.) That is, the compiler differentiates between the two forms by making calls to different overloaded functions.

The following example overloads the unary operators:

Example

```
#include<iostream>
using namespace std;
//Increment and decrement overloading
class Inc {
        private:
                int count ;
        public:
                Inc() {
                        //Default constructor
                        count = 0 ;       }
                Inc(int C) {
                        // Constructor with Argument
                        count = C ;
                }
                Inc operator ++ () {
                        // Operator Function Definition
                        return Inc(++count);
                }
                Inc operator -- () {
                        // Operator Function Definition
                        return Inc(--count);
                }
                void display(void) {
                        cout << count << endl ;
                }
};
```

```
void  main(void)
 {
        Inc a, b(10), c, d, e(5), f(10);
        cout << "Before using the operator ++()\n";
        cout << "a = ";
        a.display();
        cout << "b = ";
        b.display();
        ++a;
        b++;
        cout << "After using the operator ++()\n";
        cout << "a = ";
        a.display();
        cout << "b = ";
        b.display();
        c = ++a;
        d = b++;
        cout << "Result prefix (on a) and postfix (on b)\n";
        cout << "c = ";
        c.display();
        cout << "d = ";
        d.display();
        cout << "Before using the operator --()\n";
        cout << "e = ";
        e.display();
        cout << "f = ";
        f.display();
        --e;
        f--;
        cout << "After using the operator --()\n";
        cout << "e = ";
        e.display();
        cout << "f = ";
        f.display();
}
```

Output:

Before using the operator ++()

a = 0

b = 10

After using the operator ++()

a = 1

b = 11

Result prefix (on a) and postfix (on b)

c = 2

d = 12

Before using the operator --()

e = 5

f = 10

After using the operator - -()

e = 4

f = 9

Note :

When specifying an overloaded operator for the postfix form of the increment or decrement operator, the additional argument must be of type int; specifying any other type generates an error.

Overloading assignment operator ('=')

```
class Complex
{
private:
        double real, imag;
public:
        Complex(){
                real = 0;
                imag = 0;
        }
        Complex(double r, double i) {
                real = r;
                imag = i;
        }
        double getReal() const {
                return real;
        }
        double getImag() const {
                return imag;
        }
        Complex & operator=(const Complex &);
};
Complex & Complex::operator=(const Complex& c) {
        real = c.real;
        imag = c.imag;
        return *this;
}
#include <iostream>
int main()
{
        using namespace std;

        Complex c1(5,10);
Complex c2(50,100);
        cout << "c1= " << c1.getReal() << "+" << c1.getImag() << "i" << endl;
```

```
        cout << "c2= " << c2.getReal() << "+" << c2.getImag() << "i" << endl;
        c2 = c1;
        cout << "assign c1 to c2:" << endl;
        cout << "c2= " << c2.getReal() << "+" << c2.getImag() << "i" << endl;
}
```

Output:

```
c1= 5+10i
c2= 50+100i
assign c1 to c2:
c2= 5+10i
```

Overloading the << Operator

Output streams use the << operator for standard types. We can also overload the << operator for our own classes.

Actually, the << is left shift bit manipulation operator. But the ostream class overloads the operator, converting it into an output tool. The cout is an ostream object and that it is smart enough to recognize all the basic C++ types. That's because the ostream class declaration includes an overloaded operator<<() definition for each of the basic types.

Example

```
#include <iostream>
using namespace std;
class Complex
{
private:
        double real, imag;
public:
        Complex(){
                real = 0;
imag = 0;
        }
        Complex(double r, double i) {
                real = r;
                imag = i;
        }
        double getReal() const {
                return real;
        }
        double getImag() const {
                return imag;
        }
        Complex & operator=(const Complex &);
        const Complex operator+(const Complex & );
        Complex & operator++(void);
        Complex  operator++(int);
        /*friend const
                Complex operator+(const Complex&, const Complex&); */
        friend ostream& operator<<(ostream& os, const Complex& c);
};
```

```
Complex & Complex::operator=(const Complex& c) {
        real = c.real;
        imag = c.imag;
        return *this;
}
const Complex Complex::operator+(const Complex& c) {

Complex temp;
        temp.real = this->real + c.real;
        temp.imag = this->imag + c.imag;
        return temp;
}
//pre-increment
Complex & Complex::operator++() {
        real++;
        imag++;
        return *this;
}
//post-increment
Complex Complex::operator++(int) {
          Complex temp = *this;
          real++;
          imag++;
          return temp;
}
/* This is not a member function of Complex class */
/*const Complex operator+(const Complex& c1, const Complex& c2) {
          Complex temp;
          temp.real = c1.real + c2.real;
          temp.imag = c1.imag + c2.imag;
          return temp;
}*/
ostream& operator<<(ostream &os, const Complex& c) {
        os << c.real << '+' << c.imag << 'i' << endl;
        return os;
}
int main()
{
        Complex c1(5,10);
        cout << "c1 = " << c1.getReal() << "+" << c1.getImag() << "i" << endl;
        cout << "Using overloaded << " << endl;
        cout << "c1 = " << c1 << endl;
}
```

Output:

c1 = 5+10i

Using overloaded <<

c1 = 5+10i

Note that we just used:

cout << "c1 = " << c1 << endl;

Note that when we do

cout << c1;

it becomes the following function call:

operator<<(cout, c1);

operator overloading for strings

```
#include<iostream.h>
 #include<string.h>
 #include<conio.h>
class string
   {
      private:
       char str[80];
      public:
       string()  { strcpy(str,"ttt"); }
string(char s[])  { strcpy(str,s); }
       void display()  { cout<<str<<endl; }
           string operator+(string );
   };
string string::operator+(string ss){
      string temp;
      if(strlen(str)+strlen(ss.str)<80)
     {
       strcpy(temp.str,str);
       strcat(temp.str,ss.str);
     }
      else
     {
       cout<<"string overflow"<<endl;
       temp=0;
     }
      return temp;     }
 main()
   {
      clrscr();
      string s1(" Operator");
      string s2(" Overloading");
      string s3;
      s1.display();
      s2.display();
      s3=s1+s2;
      s3.display();
      getch();
      return 0;
   }
```

Output:

Operator

Overloading

Operator Overloading

Operator overloading from String object to basic string

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
class string
{
  char *p;
  int len;
  public:
   string()
   {}
   string(char *a)
   {
     len=strlen(a);
     p=new char[len+1];
     strcpy(p,a);
   }
   operatorchar*()
   {
     return(p);
   }
   void display()
   {
     cout<<p;
   }
};
void main()
{
 clrscr();
 string o1="IGNOU";
 cout<<"String of Class type : ";
 o1.display();
 cout<<endl;

 char *str=o1;
 cout<<"String of Basic type : "<<str;
 getch();
}
```

Output:

String of Class type : IGNOU

String of Blass type : IGNOU

Instream and outstream operator overloading

Example

```
Class Complex
{
Public:
Friend istream & operator >>(istream &is, Complex &c2);
Friend ostream & operator <<(ostream &os, Complex &c2);
Private:
Int real,imaginary;
};
Istream& operator >>(istream &is, Complex &c2);
{
Cout<<"enter real and imaginary"<<endl;
Is>>c2.real>>c2.imaginary;
Return(is);
}
Istream& operator <<(ostream &os, Complex &c2)

{
Os<<"the complex number is "<<endl;
Os<<c2.real<<"I"<<c2.imaginary;
Return(os);
}
Void main()
{
Complex c1,c2;
Cin>>c1;
Cout<<c1;
Cin>>c2;
Cout<<c2;
}
```

This operator function have to be declared friends since they have to access the user class and the objects of stream and ostream classes that are sytem defined. Since these operators functions are friend functions, the two objects cin and cout are passed as arguments, along with the objects of the user class. They return the isteam and ostream objects so that the operator can be chained. That is the above two input statements can also be written as,

Cin>>c1>>c2;

Cout<<c1<<c2;

☞ Check Your Progress 2

1) What is concept of operator overloading?

……

……

……

2) What are the basic rules for operator overloading in C++?

……

……

……

3) What are the limitations of Operator overloading and Functional overloading?

……

……

……

4) Multiple choice questions:

i) Which of the following statements is NOT valid about operator overloading?

a) Only existing operators can be overloaded.

b) Overloaded operator must have at least one operand of its class type.

c) The overloaded operators follow the syntax rules of the original operator.

d) none of the above.

ii) The new operator

a) returns a pointer to the variable

b) creates a variable called new

c) obtains memory for a new variable

d) tells how much memory is available

iii) Which of the following operator can be overloaded through friend function?

a) ->

b) =

c) ( )

d) *

iv) Overloading a postfix increment operator by means of a member function takes

a) no argument

b) one argument

c) two arguments

d) three arguments

2.7 SUMMARY

In this unit, we have discussed two important concepts about overloading namely function overloading and operator overloading. As a part of function overloading, you learnt that you can create multiple functions of the same name that work differently depending on parameter type. Function overloading can lower a programs complexity significantly while introducing very little additional risk. Although this particular lesson is long and may seem somewhat complex (particularly the matching rules), in reality function overloading typically works transparently and without any issues. The compiler will flag all ambiguous cases, and they can generally be easily resolved. Operator overloading is common-place among many efficient C++ programmers. Operating overloading allows you to pass different variable types to the same function and produce different results. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type. Operator overloading allows the programmer to define how operators should interact with various data types. It is so because operators in C++ are implemented as functions, operator overloading works very analogously to function overloading. Operator overloading is the ability to tell the compiler how to perform a certain operation when its corresponding operator is used on one or more variables.

2.8 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as their types are concerned). This capability is called function overloading. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types.

2) a) A match is found. The call is resolved to a particular overloaded function.
b) No match is found. The arguments can not be matched to any overloaded function.
c) An ambiguous match is found. The arguments matched more than one overloaded function.

When an overloaded function is called, C++ goes through the following process to determine which version of the function will be called:

i) First, C++ tries to find an exact match. This is the case where the actual argument exactly matches the parameter type of one of the overloaded functions. For example:

```
void Print(char *szValue);

void Print(int nValue);
```

Print(10); // exact match with Print(int)

Although 10 could technically match Print(char*), it exactly matches Print(int). Thus Print(int) is the best match available.

ii) If no exact match is found, C++ tries to find a match through promotion. In the lesson on type conversion and casting, we covered how certain types can be automatically promoted via internal type conversion to other types.

3) a) The use of function overloading is to increase consistency and readability.

b) Overloading provides multiple behaviour to same object with respect to attributes of object.

c) By using function overloading, we can call specific behaviour of that object attributes we set at compiled time. Further, we don't need to write different function name for different action.

d) can develop more than one function with the same name.

e) function overloading exhibits the behavior of polymorphism which helps to get different behaviour, although there will be some link using same name of function. Another powerful use is constructor overloading, which helps to create objects differently and it also helps a lot in inheritance.

4) Multiple Choice Questions

i) (A)

ii) (B)

iii) (C)

iv) (D)

Check Your Progress 2

1) Operator overloading allows existing C++ operators to be redefined so that they work on objects of user-defined classes. They form a pleasant facade that doesn't add anything fundamental to the language (but they can improve understandability and reduce maintenance costs). When an operator is overloaded, it takes on an additional meaning relative to a certain class. But it can still retain all of its old meanings.

Examples:
a) The operators >> and << may be used for I/O operations because in the header, they are overloaded.

b) In a stack class, it is possible to overload the + operator so that it appends the contents of one stack to the contents of another. But the + operator still retains its original meaning relative to other types of data.

2) The following rules constrain how overloaded operators are implemented. However, they do not apply to the new and delete operators, which are covered separately.

You cannot define new operators, such as **.

You cannot redefine the meaning of operators when applied to built-in data types.

Overloaded operators must either be a non-static class member function or a global function.

Operators obey the precedence, grouping, and number of operands dictated by their typical use with built-in types.

Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.

Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.

If an operator can be used as either a unary or a binary operator (&, *, +, and -), you can overload each use separately.

Overloaded operators cannot have default arguments.

All overloaded operators except assignment (operator=) are inherited by derived classes.

The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked (the class in which the operator is declared, or a class derived from that class). No conversions are supplied for the first argument.

3) Function overloading is like you have different functions with the same name but different signatures working differently. So, the compiler can differentiate and find out which function to call depending on the context. In case of operator overloading, you try to create your own functions which are called when the corresponding operator is invoked for the operands.

One important thing to understand is that you can create as many functions as you want with the same name and different signatures so that they can work differently but for a particular class, you cannot overload the operator function based on number of arguments. There is a fundamental reason behind this.

According to the rules, you can not create your own operators and you have to use already available operators. Another thing is, since the operators are already defined for use with built-in types, you can't change their characteristics. For example, the binary operator '+' always takes two parameters, so for this you cannot create a function that takes three parameters. But you can always overload them based on the type of the parameters.

4) Multiple Choice Questions

i) (D)

ii) (C)

iii) (D)

iv) (A)

2.9 FURTHER READINGS

1) *Object-Oriented Programming with C++*, E. Balagurusamy, 2nd edition, TMH, New Delhi, 2001

2) *The C++ Programming Language*, Bjarne Stroustroup, 3rd edition, Addison Wesley, 1997

3) *C++: The Complete Reference*, H. Shildt, 4th edition, TMH, New Delhi, 2004

4) *Mastering C++*, K. R. Venugopal, Rajkumar and T. Ravishankar, TMH, New Delhi, 2004

5) www.learncpp.com/cpp-tutorial

6) http://www.bogotobogo.com/cplusplus

UNIT 3 POLYMORPHISM AND VIRTUAL FUNCTION

Structure **Page Nos.**

3.0 INTRODUCTION

Polymorphism is one of the important features of object-oriented programming. It simply means 'one name multiple forms'. We have already seen the polymorphism concept used in function overloading and operator overloading in unit -2. For example, an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data used in the operation. Polymorphism is the ability to use an operator or function in different ways. The word poly means many, signifies the many uses of these operators and function. A single function usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts. C++ supports polymorphism both at run-time and at compile-time. Function overloading & operator overloading belongs to compile time polymorphism where as run-time polymorphism can be achieved by the use of both derived classes and virtual functions. In this unit, you will learn about the concept of run time (dynamic) binding, virtual function and pure virtual function in detail.

3.1 OBJECTIVES

After going through this unit, you will be able to:

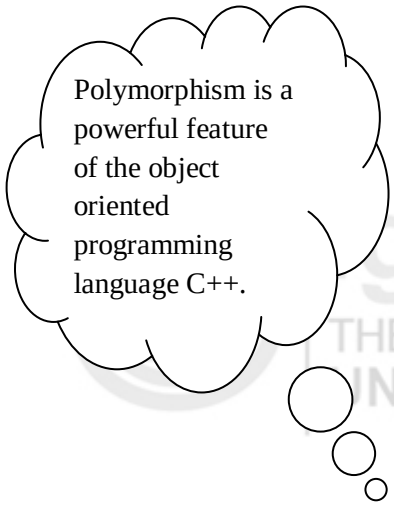
- explain the concept of polymorphism;
- explain the Concepts of dynamic binding;

- learn the concept and application of virtual function;
- use pointers to object;
- explain how to use pointers to derived class;
- understand the concept of pure virtual function, and
- describe the Concepts of Abstract Classes.

3.2 POLYMORPHISM

Polymorphism is the ability to use an operator or method in different ways. Polymorphism gives different meanings or functions to the operators or methods. Poly refers many that signify the many uses of these operators and methods. A single method usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts. "Polymorphism is a mechanism that allows you to implement a function in different ways."

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 actions associated with each operation may differ.



Example of the concept of polymorphism:

- 6 + 10 //The above refers to integer addition.
- The same + operator can be used with different meanings with strings:
- "Technical" + "Training"
- The same + operator can be also used for floating point addition: 7.15 + 3.78

We saw above that a single operator '+' behaves differently in different contexts such as integer, string or float referring the concept of polymorphism. The above concept leads to operator overloading. When the exiting operator or function operates on new data type it is overloaded. C++ also permits the use of different functions with the same name. Such functions have different argument list. The difference can be in terms of number or type of arguments or both. It refers as function overloading. So, we conclude that the concept of operator overloading and function overloading is a branch of polymorphism. Both the concepts have been discussed in unit 2 in detail.

3.2.1 Advantages of Polymorphism

- The biggest advantage of polymorphism is creation of reusable code by programmer's classes once written, tested and implemented can be easily reused without caring about what's written in the case.
- Polymorphic variables help with memory use, in that a single variable can be used to store multiple data types (integers, strings, etc.) rather than declaring a different variable for each data format to be used.

- Applications are Easily Extendable: Once an application is written using the concept of polymorphism, it can easily be extended, providing new objects that conform to the original interface. It is unnecessary to recompile original programs by adding new types. Only re-linking is necessary to exhibit the new changes along with the old application. This is the greatest achievement of C++ object-oriented programming. In programming language, there has always been a need for adding and customizing. By utilizing the concept of polymorphism, time and work effort is reduced in addition to making future maintenance easier.

- It provides easier maintenance of applications.

- It helps in achieving robustness in applications.

3.2.2 Types of Polymorphism

C++ provides three different types of polymorphism:

- Function overloading (for definition and example, see block 2, unit 2)
- Operator overloading (for definition and example, see block 2, unit 2)
- Virtual functions (Defined in section 3.4 of this unit)

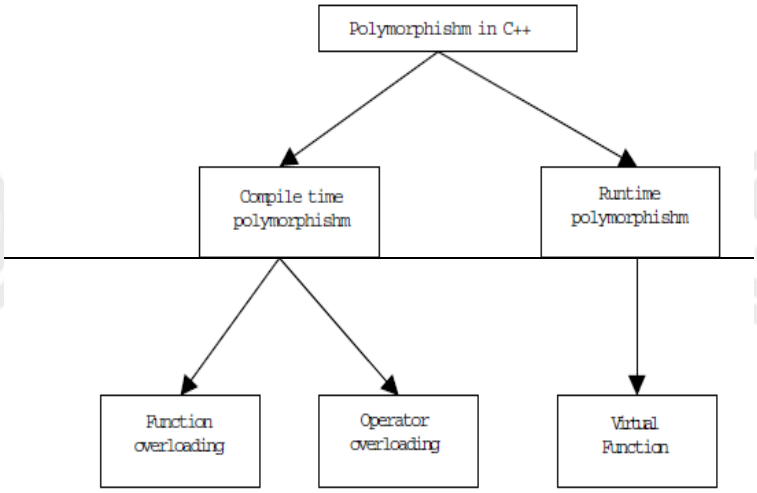


Figure 3.1: Achieving polymorphism

3.3 DYNAMIC BINDING

You know that polymorphism can be implemented using operator and function overloading, where the same operator and function works differently on different arguments producing different results. The overloaded member functions are selected for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called *early binding*, *static binding*, *static linking* or *compile time polymorphism*.

However, ambiguity creeps in when the base class and the derived class both have a function with same name. For instance, let us consider the following code snippet.

```
Class P
{
int a;
public:
void display() {……} //display in base class
};

Class Q : public P
{
int b;
public:
void display() {……} //display in derived class
};
```

It is also known as *dynamic binding* because the selection of the appropriate function is dynamically at run time.

Since, both the display() functions are same but at in different classes, there is no overloading, and hence early binding does not apply. We have seen earlier that, in such situations, we may use the class resolution operator to specify the class while invoking the function with the derived class objects.

It would be better if the appropriate function is chosen at the run time. This is known as run time polymorphism. C++ supports *run-time polymorphism* by a mechanism called *virtual function.* It exhibits *late binding.*

As stated earlier, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. Therefore, an essential feature of polymorphism is the ability to refer to objects without any regard to their classes. It implies that a single pointer variable may refer to object of different classes. So, dynamic binding requires pointers to object for its implementation.

In the case of a compiled language, the compiler still doesn't know the actual object type, but it inserts code that finds out and calls the correct function body. When a language implements dynamic binding, there must be some mechanism to determine the type of the object at runtime and call the appropriate member function.

3.4 VIRTUAL FUNCTIONS

You know that polymorphism also refers to the ability to call different functions by using only one type of function call. Suppose a programmer wants to code vehicles of different shapes such as circles, squares, rectangles, etc. one way to define each of these classes is to have a member function for each that makes vehicles of each shape. Another convenient approach the programmer can take is to define a base class named Shape and

then create an instance of that class. The programmer can have array that hold pointers to all different objects of the vehicle followed by a simple loop structure to make the vehicle, as per the shape desired, by inserting pointers into the defined array. This approach leads to different functions executed by the same function call. Polymorphism is used to give different meanings to the same concept. This is the basis for virtual function implementation.

In polymorphism, a single function or an operator functioning in many ways depends upon the usage to function properly. In order for this to occur, the following conditions must apply:

1. All different classes must be derived from a single base class. In the above example, the shapes of vehicles (circle, triangle, rectangle) are from the single base class called Shape.

2. The member function must be declared virtual in the base class. In the above example, the member function for making the vehicle should be made as virtual to the base class.

Conclusion of above discussion is that the form of a member function can be changed at runtime. Such member functions are called virtual functions and the corresponding class is called polymorphic class. The objects of the polymorphic class, addressed by pointers, change at runtime and respond differently for the same message. The word 'virtual' means something that does not exist in reality, some sort of imaginary thing. The concept of virtual function is the same as a function, but it does not really exist although it appears in needed places in a program. The object-oriented programming language C++ implements the concept of virtual function as a simple member function, like all member functions of the class. The functionality of virtual functions can be *overridden* in its derived classes.

3.4.1 Function Overriding

To cause late binding to occur for a particular function, C++ requires that you use the **virtual** keyword when declaring the function in the base class. Late binding occurs only with **virtual** functions, and only when you're using an address of the base class where those **virtual** functions exist, although they may also be defined in an earlier base class. To create a member function as **virtual**, you simply precede the declaration of the function with the keyword **virtual**. Only the declaration needs the **virtual** keyword, not the definition. If a function is declared as **virtual** in the base class, it is **virtual** in all the derived classes. The redefinition of a **virtual** function in a derived class is usually called *function overriding*. Notice that you are only required to declare a function **virtual** in the base class. All derived-class functions that match the signature of the base-class declaration will be called using the virtual mechanism. You *can* use the **virtual** keyword in the derived-class declarations (it does no harm to do so), but it is redundant and can be confusing.

3.4.2 Properties of Virtual Functions

Dynamic Binding Property: Virtual Functions are resolved during run-time or dynamic binding. Virtual functions are also simple member functions. The main difference between a non-virtual C++ member function and a virtual member function is in the way

they are both resolved. A non-virtual C++ member function is resolved during compile time or static binding. Virtual Functions are resolved during run-time or dynamic binding.

- Virtual functions are member functions of a class.
- Virtual functions are declared with the keyword virtual.
- Virtual function takes a different functionality in the derived class.

3.4.3 Definition of Virtual Functions

C++ provides a solution to invoke the exact version of the member function, which has to be decided at runtime using virtual functions. They are the means by which functions of the base class can be overridden by the functions of the derived class. The keyword virtual provides a mechanism for defining virtual functions. When declaring the base class member function, the keyword virtual is used with those functions, which are to be bound dynamically.

The general syntax to declare a virtual function uses the following format:

```
class class_name //This denotes the base class of C++ virtual function
{
public:
virtual return_type member_function_name(arguments) //This denotes the C++ virtual function
{
...
...
}
};
```

Virtual functions should be define in the public section of a class to realize its full potential benefits. When such a declaration is made, it allows to decide which function to be used at runtime, based on the type of object, pointed to by the base pointer rather than the type of the pointer. The examples of virtual functions provided in this unit illustrate the use of base pointer to point to different objects for executing different implementations of the virtual functions.

Note: By default, C++ matches a function call with the correct function definition at compile time. This is called *static binding*. You can specify that the compiler match a function call with the correct function definition at run time; this is called *dynamic binding*. You declare a function with the keyword virtual if you want the compiler to use dynamic binding for that specific function.

3.4.4 Need of Virtual Function

The first and foremost question which is arises why do we need virtual function? Suppose we do have a list of pointer to objects of a super class in an inheritance hierarchy and we wish to invoke the functions of its derived classes with the help of single list of pointers provided that the functions in super class and sub classes have the same name and signature. That in turn means we want to achieve run time polymorphism. So, let we have brief concept about pointers to derived types.

3.4.4.1 Pointers to derived types

We know that pointer of one type may not point to an object of another type. You shall now learn about the one exception to this general rule: a pointer to an object of a base class can also point to any object derived from that base class. Similarly, a reference to a base class can also reference any object derived from the original base class. In other words, a base class reference parameter can receive an object of types derived from the base class, as well as objects within the base class itself. Let us try to work it out with the following example:

Example 1. //Program without virtual function

```
/*The case when we wish to invoke the child class function with the parent class pointer, but the output is not the same as we expected*/

#include<iostream.h>
#include<conio.h>
class Faculty
{
        int facultyId;
        char facultyName[25];
        char facultyType;
        public:
                void calculateSalary()  //Parent Class Function
                {
                cout<<"\n Calculating the salary of a faculty, no matter the faculty is regular or guest !!";
                }
};

class RegularFaculty : public Faculty
{
        public:
                void calculateSalary()  //Child Class Function
                {
                cout<<"\n Calculating the salary of a regular faculty !!";
                }
};

class GuestFaculty : public Faculty
{
        public:
                void calculateSalary()   //Child Class Function
                {
                cout<<"\n Calculating the salary of a guest faculty !!";
                }
};
```

```
void main()
{
        Faculty *pFaculty;
        RegularFaculty rFaculty;
        GuestFaculty gFaculty;
        clrscr();
        //Assigning the address of child class object into parent class pointer
        pFaculty=&rFaculty;
//Invocation of calculateSalary() function with parent class pointer
        pFaculty->calculateSalary();
        //Assigning the address of child class object into parent class pointer
        pFaculty=&gFaculty;
//Invocation of calculateSalary() function with parent class pointer
        pFaculty->calculateSalary();
        getch();
}
```

Output:

Calculating the salary of a faculty, no matter the faculty is regular or guest !!

Calculating the salary of a faculty, no matter the faculty is regular or guest !!

Note: But here the output does not come out to be as we expect it to be because here the super class 'Faculty' pointer is having reference of child class objects of RegularFaculty and GuestFaculty classes respectively but when we try to call the derived class function namely the calculateSalary() with the help of this pointer it does not do so. Both the time it is calling the function calculateSalary() of super class only!!

Now look at the following program:

Example 2. //Program with virtual function

```
/* The case when we wish to invoke the child class function
with the parent class pointer and the output comes as expected*/
#include<iostream.h>
#include<conio.h>
class Faculty
{
        int facultyId;
        char facultyName[25];
        char facultyType;
        public:
                virtual void calculateSalary()  //Parent Class Function
                {
                cout<<"\n Calculating the salary of a faculty, no matter the faculty
is regular or guest !!";
                }
};
```

```
class RegularFaculty : public Faculty
{
        public:
                void calculateSalary() //Child Class Function
                {
                cout<<"\n Calculating the salary of a regular faculty !!";
                }
};
class GuestFaculty : public Faculty
{
        public:
                void calculateSalary() //Child Class Function
                {
                cout<<"\n Calculating the salary of a guest faculty !!";
                }
};

void main()
{
        Faculty *pFaculty;
        RegularFaculty rFaculty;
        GuestFaculty gFaculty;
        clrscr();
        //Assigning the address of child class object into parent class pointer
        pFaculty=&rFaculty;
//Invocation of calculateSalary() function with parent class pointer
        pFaculty->calculateSalary();
//Assigning the address of child class object into parent class pointer
        pFaculty=&gFaculty;
//Invocation of calculateSalary() function with parent class pointer
        pFaculty->calculateSalary();
        getch();
}
```

Output:

Calculating the salary of a regular faculty !!

Calculating the salary of a guest faculty !!

Explanation

See the magic of virtual function. There is a slight change in the above program the function calculateSalary() in super class is declared to be virtual and the output is turned to be as per our expectations, we are having the pointer 'pFaculty' to super class but when the references of child class objects namely the rFaculty and gFaculty to this pointer we see that with the pointer having the reference of rFaculty the function of child class regularFaculty is called where as when it contains the referenc of gFaculty then the function of child class GuestFaculty is called.

Let we have one more example to clarify the concept:

The vital reason for having a virtual function is to implement a different functionality in the derived class.

For example: a Make function in a class Vehicle may have to make a Vehicle with red color. A class called FourWheeler, derived or inherited from Vehicle, may have to use a blue background and 4 tires as wheels. For this scenario, the Make function for FourWheeler should now have a different functionality from the one at the class called Vehicle. This concept is called Virtual Function.

Example 3.

```
class Vehicle //This denotes the base class of C++ virtual function
{
public:
virtual void Make() //This denotes the C++ virtual function
{
cout << "Member function of Base Class Vehicle Accessed" << endl;
}
};
```

After the virtual function is declared, the derived class is defined. In this derived class, the new definition of the virtual function takes place. When the class FourWheeler is derived or inherited from Vehicle and defined by the virtual function in the class FourWheeler, it is written as:

```
#include <iostream>
using namespace std;
class Vehicle   //This denotes the base class of C++ virtual function
{
public:
      virtual void Make()   //This denotes the C++ virtual function
      {
            cout << "Member function of Base Class Vehicle Accessed" << endl;
      }
};
class FourWheeler : public Vehicle
{
public:
      void Make()
      {
            cout << "Virtual Member function of Derived class FourWheeler Accessed"
<< endl;
      }
};
void main()
{
      Vehicle *a, *b;
      a = new Vehicle();
      a->Make();
      b = new FourWheeler();
      b->Make();
}
```

Explanation

In the above example, it is evident that after declaring the member functions Make() as virtual inside the base class Vehicle, class FourWheeler is derived from the base class Vehicle. In this derived class, the new implementation for virtual function Make() is placed.

In this example, the member function is declared virtual and the address is bounded only during run time, making it dynamic binding and thus the derived class member function is called.

To achieve the concept of dynamic binding in C++, the compiler creates a v-table each time a virtual function is declared. This v-table contains classes and pointers to the functions from each of the objects of the derived class. This is used by the compiler whenever a virtual function is needed.

3.4.5 Rules for Virtual Functions

- The virtual functions must be the members of some class.
- A class member function can be declared to be virtual by just specifying the keyword 'virtual' in front of the function declaration. The syntax of declaring a virtual function is as follows:

  virtual <return type> <function name><(argument list)>

  {//Function Body}
- Virtual Functions enables derived (sub) class to provide its own implementation for the function already defined in its base (super) class.
- Virtual Functions give power to the derived class functions to override the function in its base class with the same name and signature.
- Virtual Functions can't be static members.
- Only the functions that are members of some class can be declared as virtual that means we can't declare regular functions or friend functions as virtual.
- A virtual function can be a friend of another class.
- A virtual function in a base class must be defined, even though it may not be used.
- If one will call the virtual function with the pointer having the reference to the base class object then the function of the base class will be called for sure.
- The corresponding functions in the derived class must agree with the virtual function's name and signature that means both must have same name and signature.

3.4.6 Limitations of Virtual Functions

- The function call takes slightly longer due to the virtual mechanism, and it also makes it more difficult for the compiler to optimize because it doesn't know exactly which function is going to be called at compile time.
- In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.
- Virtual functions will usually not be inlined.
- Size of object increases due to virtual pointer.

3.5 PURE VIRTUAL FUNCTIONS

Remember that a class containing pure virtual functions can't be used to declare any objects of its own.

Pure Virtual Functions are the specific type of virtual functions. A virtual function with no function body is called pure virtual function i.e. a pure virtual function is a function declared in a base class that has no definition relative to base class. A pure virtual function purely exists in base class only to be overridden by the derived class functions. A base class only specifies that there is a function with this name and signature which will be implemented by any of derived class or by some other derived class in the same inheritance hierarchy. Such classes are also called *abstract base class*.

3.5.1 Syntax of pure virtual function

The syntax of declaring a pure virtual function is as follows:
virtual <return type> <function name><(argument list)> =0;

//Program with pure virtual function

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

class Faculty
{
        int facultyId;
        char facultyName[25];
        char facultyType;
        public:
virtual void calculateSalary()=0;  //Pure Virtual Function in parent class
};
class RegularFaculty:public Faculty
{
        public:
                void calculateSalary() //Child Class Function
        {
                        cout<<"\n Calculating the salary of a regular faculty !!";
                }
};
class GuestFaculty:public Faculty
{
        public:
                void calculateSalary() //Child Class Function
                {
                        cout<<"\n Calculating the salary of a guest faculty !!";
                }
};
void main()
{
        Faculty *pFaculty;
        RegularFaculty rFaculty;
        GuestFaculty gFaculty;
        clrscr();
```

```
        //Assigning the address of child class object into parent class pointer
        pFaculty=&rFaculty;
        pFaculty->calculateSalary(); //Invocation of calculateSalary() function with
        parent class  pointer
        //Assigning the address of child class object into parent class pointer

        pFaculty=&gFaculty;
        pFaculty->calculateSalary(); //Invocation of calculateSalary() function with
        parent class pointer
        getch();
}
```

Output:

```
Calculating the salary of a regular faculty !!
Calculating the salary of a guest faculty !!
```

This is the same faculty salary calculation program that is introduced in the discussion of virtual functions but there is a change here, the function calculateSalary() in base class 'Faculty' is declared to be pure virtual with no function definition. This function is implemented by two concrete children of Faculty class namely the 'ReguarFaculty' and 'GuestFaculty'.

Then in main, we have pointer to base class Faculty namely the 'pFaculty' that is assigned by the references of the child class objects 'rFaculty' and 'gFaculty' one by one and is used to call the calculateSalary() function and it is clear from the output that one time it calls up the function in RegularFaculty and the function in GuestFaculty the other time.

3.5.2 Characteristics of Pure Virtual Functions

- A class member function can be declared to be pure virtual by just specifying the keyword 'virtual' in front and putting '=0' at the end of the function declaration.
- Pure virtual function itself do nothing but acts as a prototype in the base class and gives the responsibility to a derived class to define this function.
- As pure virtual functions are not defined in the base class thus a base class can not have its direct instances or objects that means a class with pure virtual function acts as an abstract class that cannot be instantiated but its concrete derived classes can be.
- We cannot have objects of the class having pure virtual function but we can have pointers to it that can in turn hold the reference of its concrete derived classes.
- Pure virtual functions also implements run time polymorphism as the normal virtual functions do as binding of functions to the appropriate objects here is also delayed up to the run time, that means which function is to invoke is decided at the run time.
- Pure virtual functions are meant to be overridden.
- Only the functions that are members of some class can be declared as pure virtual that means we cannot declare regular functions or friend functions as pure virtual.

- The corresponding functions in the derived class must agree be compatible with the pure virtual function's name and signature that means both must have same name and signature.
- For abstract class, pure virtual function is must.
- The pure virtual functions in an abstract base class are never implemented. Because no objects of that type are ever created, there is no reason to provide implementations, and the ADT (Abstract Data Type) works purely as the definition of an interface to objects which derive from it.
- It is possible, however, to provide an implementation to a pure virtual function. The function can then be called by objects derived from the ADT, perhaps to provide common functionality to all the overridden functions.

3.5.3 Abstract Classes

Abstract classes act as a container of general concepts from which more specific classes can be inherited. Thus an abstract class is one that is not used to create any object of its own but it solely exists to act as a base class for the other classes that means the abstract class must be a part of some inheritance hierarchy.

An abstract class can further be illuminated through following points:

- An abstract class can not be instantiated that means abstract classes can not have their own instances but their child or derived classes may have their own instances provided the child class itself is not an abstract class.
- Though objects of an abstract class cannot be created, however, one can use pointers and references to abstract class types.
- A class should contain at least one pure virtual function to be called as abstract. Pure virtual functions can be declared with the keyword virtual and =0 syntax at the end of function declaration statement.
- If a class is made abstract by giving a pure virtual function then it must be inherited by a child class of it that provides the implementation of the pure virtual function.
- If a class inherits an abstract class and does not provide the implementation of the pure virtual function then the child class itself should declare the function as pure virtual that means the child class will be an abstract class as well.
- The signature of the function declared as pure virtual in base class must strictly agree with the signature of the function in child class that implements the pure virtual function.

Example

```
//Program with abstract class having pure virtual functions
/* Complete Faculty Salary Calculation program in action*/
#include<iostream.h>
#include<conio.h>
#include<string.h>
class Faculty  //Abstract class having pure virtual function
{
        protected:
                int facultyId;
                char facultyName[25];
char facultyType;
        public:
                //Pure Virtual Functions in parent class
                virtual float calculateSalary()=0;
                virtual void showDetails()=0;
};
class RegularFaculty:public Faculty
{
        float basic,da,hra,tax;
        public:
        RegularFaculty(int id,char name[])
                {
                        facultyId=id;
                        strcpy(facultyName,name);
                        facultyType='R';
                }
        setSalaryParameters(float b,float d,float h,float t)
                {
                        basic=b;
                        da=d;
                        hra=h;
                        tax=t;
                }

        float calculateSalary() //Child Class Function
{
                        return((basic+da+hra)-tax);
                }

        void showDetails()
                {
                        cout<<"\n Id:"<<facultyId;
                        cout<<"\n Name:"<<facultyName;
                        cout<<"\n FacultyType: Regular";
                }
};
class GuestFaculty:public Faculty
{
```

```
        int noOfLectures;
        float perLectureRemuneration;
        public:
        GuestFaculty(int id,char name[])
                {
                        facultyId=id;
                        strcpy(facultyName,name);
                        facultyType='G';
                }
        setSalaryParameters(int nol,float plr)
                {
                        noOfLectures=nol;
                        perLectureRemuneration=plr;
                }
        float calculateSalary() //Child Class Function
                {
                        return(noOfLectures*perLectureRemuneration);
                }
        void showDetails()
                {
                cout<<"\n Id:"<<facultyId;
                cout<<"\n Name:"<<facultyName;
                cout<<"\n FacultyType: Guest";
                }
};
void main()
{
        float sal;
        Faculty *pFaculty;
        RegularFaculty rFaculty(1,"Ram");
        GuestFaculty gFaculty(2,"Shyam");
        clrscr();
        rFaculty.setSalaryParameters(1500,550.65,250.5,120);
        //Assigning the address of child class object into parent class pointer
        pFaculty=&rFaculty;
        //Invocation of calculateSalary() function with parent class pointer
        sal=pFaculty->calculateSalary();
        //Invocation of showDetails() function with parent class pointer
        pFaculty->showDetails();
        cout<<"\n Salary:"<<sal<<endl;
        gFaculty.setSalaryParameters(20,150.50);
//Assigning the address of child class object into parent class pointer
        pFaculty=&gFaculty;
        //Invocation of calculateSalary() function with parent class pointer
        sal=pFaculty->calculateSalary();
        //Invocation of showDetails() function with parent class pointer
        pFaculty->showDetails();
        cout<<"\n Salary:"<<sal;
        getch();
}
```

Output:

Id:1

Name:Ram

FacultyType: Regular

Salary:2181.149902

Id:2

Name:Shyam

FacultyType: Guest

Salary:3010

Explanation

Here in this example the class 'Faculty' is an abstract class as it is having two pure virtual functions named calculateSalary and showDetails. The implementation of these pure virtual functions is provided by the concrete subclasses of the class Faculty namely the RegularFaculty and GuestFaculty.

In the main function we do have objects of the two concrete subclasses and we assign the address of these objects in the pointer variable of super class type i.e. Faculty (as we cannot have direct objects of base class but we can have pointer to the abstract class type that can contain the references to the objects of its concrete child class objects) and when the functions are invoked by this pointer variable, the invocation or calling of function is bound with the exact function definition with the help of the type of reference that the pointer variable is containing. When it contains the reference of object of RegularFaculty class then the calcuateSalary and showDetails of RegularFaculty class are invoked and when it contains the reference of object of GuestFaculty class then the calcuateSalary and showDetails of GuestFaculty class are invoked and this binding is delayed upto the run time that's why it is termed as late binding or dynamic binding.

☞ Check Your Progress 1

1) Describe the concept of polymorphism

……

……

……

2) What is dynamic binding?

……

……

……

3) Explain the pointers to object with the help of an example.

……

……

……

4) How Virtual functions call up is maintained?

……

……

……

5) Explain briefly the importance of pure virtual function in the software development paradigm.

……

……

……

6) Multiple choice questions:

i) RunTime Polymorphism is achieved by _____

a) friend function

b) virtual function

c) operator overloading

d) function overloading

ii) Pure virtual functions

a) have to be redefined in the inherited class.

b) cannot have public access specification.

c) are mandatory for a virtual class.

d) None of the above

iii) Use of virtual functions implies

a) overloading.

b) overriding.

c) static binding.

d) dynamic binding.

iv) A *virtual* class is the same as

a) an abstract class

b) a class with a virtual function

c) a base class

d) none of the above.

v) A pointer to the base class can hold address of

a) only base class object

b) only derived class object

c) base class object as well as derived class object

d) None of the above

vi) A pure virtual function is a virtual function that

a) has no body

b) returns nothing

c) is used in base class

d) both (A) and (C)

3.6 SUMMARY

Polymorphism simply defines the concept of one name having more than multiple forms. It has two types of time compile time and run time. In compile time, an object is bound to its function call at compile time. In run time polymorphism, an appropriate member function is selected while the program is running. C++ supports the run time polymorphism with the help of virtual function by using the concept of dynamic binding. Dynamic binding requires use of pointers to objects. Pointers to objects of a base class type are compatible with pointers to objects of a derived class. Run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. Such virtual functions (equated to zero) are called pure virtual functions. A class containing such pure function is called an abstract class.

3.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) Polymorphism in biology means the ability of an organism to assume a variety of forms.

Polymorphism is the ability to use an operator or function in different ways. Poly implies multiple that signify the many uses of these operators and methods. A single function usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts. Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. Polymorphism is two type, compile time polymorphism (operator & function overloading) and run-time polymorphism (virtual function)

2) Dynamic binding is a mechanism that is use to implement run-time polymorphism. Dynamic binding change the form of function at run time. In this binding, the objects of the class, addressed by pointers, change at run-time and respond differently for the same message. Such mechanism requires postponement of binding of a function call to the member function until run-time.

3) By pointers you can access the class members. Pointer can also point to object created by a class. Consider the following statement:

element el;

Where element is a class and el is an object of that class. IN similar way we can define a pointer *el_pointer* of type *element* as follows:

Element *el_pointer;

Pointers to objects are useful in crteating objects at run time. Let we explain it more broadly with the help of an example.

Example: //pointers to objects

```
#include <iostream.h>

Class element
{

int id;
float price;

public:

void input(int p, int q)
{
id=p;
price=q;
}

Void display(void)
{

Cout<< "ID :" << id<< "\n";
 Cout<< "PRICE :" << price<< "\n";

}
};

 Const int limit = 3;
```

```
main()

{
        int *a = new element[limit];
        int *b = a;
        int m, i;
        float n;
        for(i=0; i<limit; i++)
        {
        cout<< "Input ID and price of element" <<i+1;
                cin >> m  >> n;
                a->input(m, n);
                a++;
        }
        for(i=0; i<limit; i++)
        {
        cout<< "ELEMENT"  << i+1<<  "\n";
                b-> display();
                        b++;
                }
                }
```

Output:

Input ID and price of element 1 40 342.25

Input ID and price of element 2 11 250.50

Input ID and price of element 3 101 500

ELEMENT 1

ID : 40

PRICE : 342.25

ELEMENT 2

ID : 11

PRICE : 250.50

ELEMENT 3

ID : 101

PRICE : 500

Explanation

In above program we created space **dynamically** for three objects of equal size. The statement

int *a = new element[limit]

allocates enough memory for an array of 3 objects of **element** in the object structure and assign the address of the memory space to pointer **a.** The object pointer also use an object pointerto access the public members of an object.

4) Through Look up tables added by the compiler to every class image. This also leads to performance penalty.

5) Its normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serve as a placeholder. Such functions are called 'do-nothing' functions.

A 'do-nothing' function may be defined as follows:

virtual void show ( ) = 0;

Such functions are called pure virtual functions.

6) Multiple Choice Questions

I-(B)

II-(A)

III-(D)

IV-(D)

V-(C)

VI-(D)

3.8 FURTHER READINGS

1) *The C++ Programming Language*, Bjarne Stroustroup, 3rd edition, Addison Wesley, 1997

2) *C++: The Complete Reference*, H. Shildt, 4th edition, TMH, New Delhi, 2004

3) *Mastering C++*, K. R. Venugopal, Rajkumar and T. Ravishankar, TMH, New Delhi, 2004

4) *Object-Oriented Programming with C++*, E. Balagurusamy, 2nd edition, TMH, New Delhi, 2001

5) www.learncpp.com/cpp-tutorial

6) http://www.bogotobogo.com/cplusplus

7) http://www.gamespp.com/c/introductionToCppMetrowerksLesson11.html

UNIT 1 STREAMS AND FILES

Structure **Page Nos.**

1.0 INTRODUCTION

In the previous blocks, we have discussed about the basic approach of object oriented programming and its important themes such as classes, inheritance, overloading, polymorphism and virtual functions. You might have seen a number of programming examples where you used **cin** and **cout** with operators >> and << for the input and output operations. We have, however, not taken up the issue of input and output to C++ programs formally. This was deliberately deferred to this point since I/O functions in C++ makes use of features like classes, derived classes and virtual functions. As we have already covered these topics, this unit builds further upon the preliminary introduction of I/O statements in C++ programs.

Through this unit, we aim to present a systematic and formal description of input and output mechanisms employed by C++ programs. C++ provides a rich set I/O functions and operations through the concept of streams and stream classes to implement I/O operations: both console and disk. The C++ I/O system contains a hierarchy of classes (known as stream classes) that are used for reading and writing by programs. Like many popular high level languages C++ supports two types of I/O: unformatted and formatted. While unformatted I/O uses functions like put(), get(), getline(), write(); formatted I/O makes use of manipulators and user-defined functions in addition to ios class functions and flags. We have also described the file stream operations and use of buffer and pointers for manipulating files.

1.1 OBJECTIVES

At the end of the unit, you should be able to:

- understand the basic mechanism of I/O in C++ through streams;
- distinguish between unformatted and formatted I/O operations in C++;
- use functions for unformatted I/O;
- design manipulators and user-defined functions for formatted I/O;
- understand stream classes for file manipulation;
- open and close files for various I/O operations;
- manage buffer and pointers for I/O from files; and
- obtain a thorough understanding and practice of using I/O functions in C++.

1.2 C++ STREAMS AND STREAM CLASSES

In C++ I/O occurs in streams, which are sequence of bytes. A stream acts as an interface between the program and the I/O device and can acts either as a source from which the input data can be obtained or as a destination to which the output data may be sent. In input operations, thy bytes flow from a device (e.g. a keyboard, a disk drive, a network connection, etc.) to main memory. In output operations bytes flow from main memory to a device (e.g., display monitor, a printer, a disk drive, a network connection, etc.). The stream that acts as a source is called input stream whereas the stream acting as destination is called output stream. The figure below illustrates the flow of data while using streams"

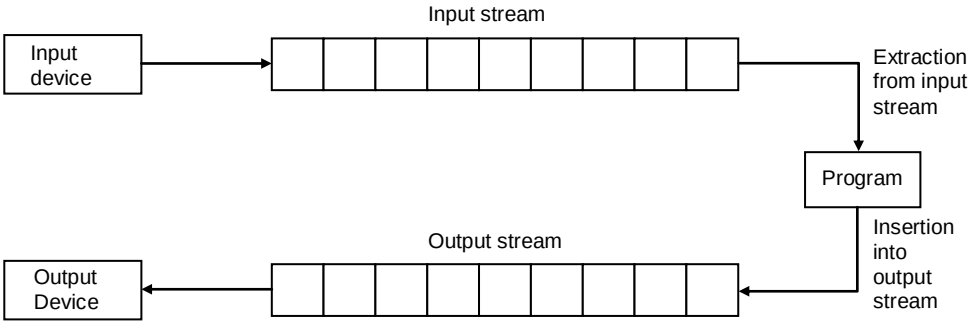


Figure 1.1 Data Streams

Though a program can take input and write output to a variety of devices (each of which may be quite different), the C++ streams provides a common interface for input and output operations irrespective of the device used. The bytes in the stream could represent characters, raw data, an image, video, speech or any other information that an application may require. It is the application (user program) that associates meaning with bytes.

In the past, the C++ used streams (often termed as classic streams) to enable input and output of characters. Since a character usually occupies one byte, it can represent only a limited set of characters (such as those in ASCII character set). Many languages however use alphabets that contain more alphabets than a single-byte character can represent. Hence C++ now includes the standard stream library that allows performing I/O operations with new character encoding systems such as Unicode. As we will shortly see, C++ now contains several pre-defined streams (e.g., cin, cout, cerr, etc.) that are automatically opened when a program begins its execution.

1.2.1 C++ Stream Classes

The C++ I/O system contains a hierarchy of classes which are used to define various streams to manage both console and disk I/O operations. These classes are called stream classes. The figure 1.2 given below shows the hierarchy of the stream classes used for input and output operations with the console. The stream class hierarchy for disk I/O is described in section 1.5

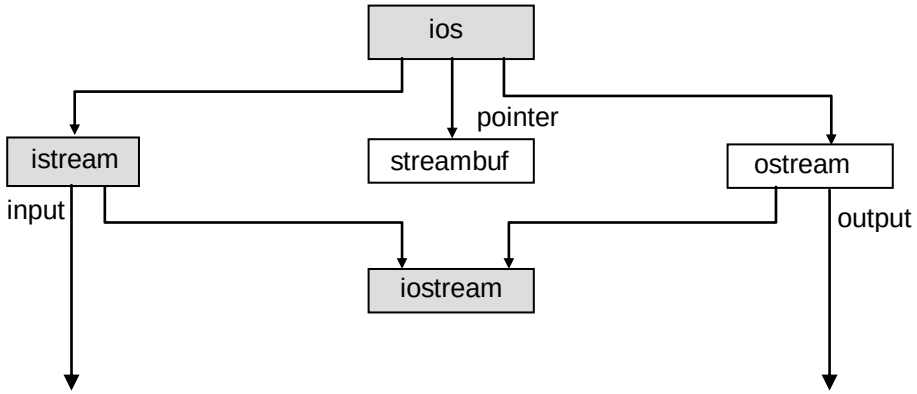


Figure 1.2 Stream Classes for Console I/O operations

The ios class is the base class for iostream (input stream) and ostream (output stream) which are in turn base classes for iostream class (input/ output stream). The ios class is declared as the virtual base class so that only one copy of its member are inherited by the iostream. The ios class thus provides the basic support for formatted and unformatted I/O operations. The class istream provides facilities for formatted and unformatted input while the class ostream provides facilities for formatted output. The class iostream provides facilities for handling both input and output operations. All these classes are declared in the header file iostream. Most of the C++ programs include the <iostream> header file, which declares all the basic services required for all stream-I/O operations. Relevant classes for formatted I/O and disk I/O are declared in <iomanip> and <fstream> header files respectively (discussed in later part of the chapter).

The standard streams- cin, cou, cerr and clog- are also defined in <iostream> header file.

1.2.2 Standard Stream Objects: cin, cout, cerr and clog

The cin, cout, cerr and clog objects correspond to the standard input stream, the standard output stream, the unbuffered standard error stream and the buffered standard error stream, respectively. These are predefined objects which are by default connected to certain devices. The extraction (>>) and insertion (<<) operators define the direction of data flow. For example, the first statement below is used to read a value from keyboard and pass it to the variable 'num'; whereas the second statement the value of variable 'avg' is written to the monitor (the standard output device).

```
cin >> num;  // reading the value of num from keyboard, data flow denotes input
cout << avg;  // displaying the value of avg on monitor, data flow denotes output
```

The number of characters read is determined by the type of variable. For example, if we type 3456X as the value for number, only '3456' is assigned to the variable 'num' and 'X' remains in the stream. Hence the operator >> reads the data character by character and assigns it to the indicated variable, unless a whitespace or a character that does not match with the destination variable type is encountered. At this point, extraction from stream is terminated and the remaining characters are consumed by subsequent cin statements. The nature of extraction (>>) and insertion (<<) operators therefore allows use of following kinds of cin and cout statements.

```
cin >> var1 >> var2 >> var3 >> ….>> varN
cout << var1 << var2 << var3 <<….<< varN
```

The variables in first statement may be variables of any type. The variables in second statement may be variables or constants of any type.

The predefined object **cerr** is an ostream instance and is connected by default to the standard error device (usually the monitor). Outputs to object **cerr** are *unbuffered*, implying that each stream insertion to **cerr** causes its output to appear immediately. (This is appropriate since it allows notifying a user promptly about errors). The predefined object **clog** is an instance of the ostream class also connected to the standard error device. However, outputs to **clog** are *buffered*. This means that each insertion to **clog** causes its ouput to be held in a buffer until the buffer is filled or it is flushed.

1.2.3 Types of I/O

The **ios** class provides the basic support for two kinds of input and output: formatted and unformatted. The class **istream** provides the facilities for formatted and unformatted input operations whereas the **ostream** class provides the facilities for formatted and unformatted output. The **istream** class declares input functions such as **get()**, **getline()**, **read()** etc., in addition to inheriting the properties of **ios** class. It also contains overloaded extraction operator >>. The **ostream** class declares output functions such as **put()** and **write()**, in addition to inheriting properties of **ios** class. It also contains overloaded insertion operator <<. The **iostream** class inherits the properties of **ios**, **istream** and **ostream** classes through multiple inheritance and hence contains all input and output functions. This is the reason why we include the iostream class only in our programs.

1.3 UNFORMATTED I/O

We have already seen use of cin and cout objects along with overloaded operators >> and << for input and output operations, in previous blocks. C++ provides many other functions for input and output operations. We will therefore briefly review their use through some examples. Functions put(), get(), getline(), putline(), read() and write() are other commonly used functions for unformatted I/O. The following sections describe, with appropriate examples, the use of these functions for unformatted I/O operations.

1.3.1 Overloaded Operators >> and <<

As stated in section 1.2.2 earlier, the overloaded **extraction (>>)** and **insertion (<<)** operators are used with **cin** and **cout** objects for stream input and output operations, respectively. The general syntax of its usage is as:

```
cin >> variable 1 >> variable 2 >> ……. >> variable N
cout << item 1 << item 2 << ……. << item N
```

where, variable 1 to N are any valid variable types in C++ and items 1 to N are any valid variable or constants in C++. C++ takes care of reading only appropriate number and type of variables (corresponding to the type of input variable) from the stream. The insertion operator puts the unformatted output into the output stream which is then displayed on the monitor. The use of these operators with **cin** and **cout** is further illustrated through following programming example:

```
# include <iostream.h>
int main()
{
        float num1, num2, num3, sum, avg;
        cout << "Enter the three numbers:";
        cin >> num1 >> num2 >> num3;
```

```
        sum = num1 + num2 +num3;
        avg = sum / 3;
        cout << "Sum of the numbers =" << sum;
        cout << "Average of the numbers =" << avg;
        return (0);
}
```

This program first displays a prompt for entering three numbers by using cout and then reads three numbers entered from the keyboard by using cin. The program then computes sum and average of the numbers and their values are displayed on the monitor through cout statements.

1.3.2 Using member functions get (), put (), getline (), ignore (), putback () and peek ()

Unformatted input and output operations can also be carried out using various member functions of cin and cout objects provided by the istream and ostream classes. The functions get() and put() can be used to handle single character input and output operations, respectively. The general usage syntax of get is as follows:

```
get (char *)
get (void)
```

The get (char *) function assigns the character read from the keyboard to its argument. Unlike extraction operator >>, it can read blanks spaces and newline characters. [Note that >> operator simply skips the blank spaces and newline character while extracting characters from the input stream.] The get (void) simply returns the character read from the keyboard without assigning it to any variable.

The put () member function can be used to write one character to the monitor. It may take a character constant or variable as argument. The put () function can also take an integer value as input (for ex. 65), however rather than displaying the integer value it displays its ASCII equivalent character, "A" for 65. It can be put in a loop to output a line of text character by character.

```
put ('char constant')
put (char variable)
```

The program below illustrates use of get () and put () functions. The program reads an input text and displays it on the output screen. It also counts the number of characters and displays it on output screen. For example, if the user enters "I love Programming", the output screen displays the entered text "I love Programming" and "Number of characters = 18" on separate lines on the output screen.

```
# include <iostream.h>
int main()
{
        int count = 0;
        char c;
        cout << "Enter some text:" ;
        cin.get(c);
        while (c!= '\n')
        {
                cout.put(c);
                count++;
                cin.get(c);
```

```
        }
        cout << "\n Number of characters = " << count << "\n";
        return (0);
}
```

The functions get() and put() can handle a single character at a time. Many practical situations however require us to read and display more than a single character, for example a line of text. The getline () member function can be used for this purpose. The general syntax of getline () function is as follows:

cin.getline (variable, size)

This function reads the character input into the variable. The reading is terminated as soon as size-1 characters are read or '\n' is encountered. The delimiter character '\n' however is discarded and not saved in the variable. For example, for the code segment:

```
char name [20];
cin.getline (name, 20)
```

if the text entered from the keyboard is "IGNOU <press RETURN>", it stores the line as "IGNOU" in the variable **name**. However, if we enter "Object Oriented Programming <press RETURN>", it stores only first 19 characters in the variable **name** as "Object Oriented Pro". Similar to get (), getline () can also read blank spaces.

The **ignore ()** member function reads and discards a designated number of characters (default = 1) or terminates upon encountering delimiter EOF. The **putback ()** member function places the character just read by get () back into the stream. The **peek ()** member function returns the next character from an input stream but does not remove the character from the stream.

1.3.3 Using member functions read (), write () and gcount ()

The other member functions used to perform unformatted I/O include read (), write () and gcount (). The member function read () inputs bytes to a character array in memory; member function write () write output bytes from character array; and member function gcount () reports the number of characters read by the last input operation. The general syntax of read () and write () functions are as follows:

```
cin.read (variable, size);
cout.write (variable, size)
```

The member function read () inputs a designated number of characters (max. = size) into a character array variable. It is different from getline () in terms of usage of array variable. The member function write () outputs the characters in the character array variable on the output screen. It however does not stops at delimiter (null char) boundary and continues to display characters even beyond the line (when size > length of array). The gcount () member function is used to report the number of characters actually read by the last read () operation. The program below demonstrates the use of read () and write () member functions. If the string entered from the keyboard is "I love Programming", it stores only "I love" in the input array variable. Function gcount () returns the value as 6.

```
# include <iostream.h>
int main()
{
        char buffer [80];
        cout << "Enter a line of text \n" ;
```

```
        cin.read(buffer, 20);
        cout.write (buffer, cin.gcount());
        return (0);
}
```

☞ Check Your Progress 1

1) Fill in the blanks:
   a) Input/ output in C++ occurs as ……of bytes.
   b) Most C++ program that do I/O should include the …… header file that contains the required for all stream I/O operations.
   c) The symbol for stream extraction operator is …….. .
   d) The four objects that correspond to the standard devices on the system include ……., ……., ……, and …… .
   e) Unformatted output member functions are provided in the class ……..

2) State whether following are *True* or *False*.
   a) The cin stream normally is connected to the display screen.
   b) Input with stream extraction operator >> always skips leading blank spaces in the input stream, by default.
   c) The ostream member function put () outputs the specified number of characters.

3) For each of the following, write a single statement that performs the indicated task:
   a) Output the string "Enter your name".

   ……

   ……

   ……

   ……

   ……

   b) Use istream member function read to input 50 characters into char array line.

   ……

   ……

   ……

   ……

   ……

   c) Get the value of next character to input without extracting it from the stream.

   ……

   ……

   ……

   ……

   ……

d) Use the istream member function gcount to determine the number of characters input into character array line by the last call to read and output that number of characters using write.

……

……

……

……

……

1.4 FORMATTING OUTPUTS

C++ provides a number of features that can be used display the outputs in a specified manner (formatted output). These features can be broadly categorized into following three types:

- **ios** class functions and flags
- Stream manipulators
- User-defined output functions

The **ios** class contains a large number of member functions that are used to format outputs in variety of ways. Most of the stream manipulators provide roughly the same features as that of **ios** class formatting functions, though they are at times convenient to use than their counterpart **ios** class functions. C++ allows users to design their own stream manipulators as well.

1.4.1 ios class format functions and flags

The **ios** class contains functions for defining field width, setting precision, filling and padding, displaying sign of numerical values etc. The Table 1.1 shows the lists of important **ios** class functions for formatting I/O:

Table1.1: List of important ios class functions for formatting I/O

Function	Purpose
width ()	To specify field size for displaying an output value
precision ()	To specify the number of digits to be displayed after the decimal sign
fill ()	To specify a character that fills the unused portion of an output data filed
setf()	To specify format flags such as left-justify, right-justify etc.
unsetf ()	To clear/ reset defined flags

Setting field width

The **width ()** function is used to define the width of a field required for displaying an output item. It is invoked by **cout** object as follows:

cout.width (w)

where **w** is the number of columns (field width). The output value is printed in a field of **w** characters wide at right end. This can specify field width for displaying only one

output value (the one that immediately follows it). The following statements demonstrate the use of width ():

```
cout.width (5);
cout << 123 << "\n";
cout.width (5);
cout << 35;
```

The output of the above statements would be displayed in a field width of 5 as follows:

		1	2	3
			3	5

However, C++ never truncates the display value if the specified width is smaller than required. In turn it expands field width to accommodate the output value. This feature can be used to print large number of numerical values in a predetermined manner (a situation that is often encountered in accounting and other commercial applications).

Setting Precision

In C++ the floating point numbers are by default printed with six digits after the decimal point. We can however change the number of digits to be displayed after the decimal sign by using precision () function. The syntax is as follows:

cout.precision (d)

where d is the number of digits to be displayed to the right of decimal point. For example, the statements:

```
cout.precision (3);
cout << sqrt (2) << "\n";
cout << 3.14159 << "\n";
cout << 1.50009 << "\n";
```

will produce the following output:

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

The **precision ()** function is different from **width ()** in its effect, as the effect of precision once set continues in all subsequent statements until it is reset. The **precision ()** function can be used with **width ()** function as illustrated in following example:

```
cout.precision(2);
cout.width(5);
cout << 5.6705;
```

produces following output:

	5		6	7

The statements print the output value with a precision of 2 digits after the decimal place within a field width of 5.

Filling and Padding

The **fill ()** function is used to fill unused portion of a display field with a desired character rather than the blank spaces printed by default. The syntax is:

cout.fill (ch);

where, ch is the character to be used to fill the unused portions of a display field. For example, the following statement:

```
cout.fill ('*');
cout.width(10);
cout << 1234 << "\n";
```

produces following output:

*	*	*	*	*	*	1	2	3	4

This kind of padding is very useful for institutions like banks which use it in their financial instruments (demand drafts etc) so that no one can change the figures. The **fill ()** function also stays in effect till it is reset.

Formatting Flags, Bit-fields and setf ()

The **setf ()** is another important **ios** class function that is commonly used for formatting outputs. The general syntax of **setf ()** is as follows:

cout.setf (arg1, arg2)

or

cout.setf (arg)

Here the **arg1** is one of the formatting flags defined in **ios** class and **arg2** is an **ios** constant that specifies the group to which the formatting flag belongs. These flags and bit-fields can be used for changing the alignment of display (left, right and center justify), displaying numbers in desired notation (scientific or fixed point), displaying numbers in different base systems (decimal, octal and hexadecimal). The setf () function can be used with single argument as well. In that case only flags are used without bit-fields.

The flags, bit-fields and their format actions are described in Table 1.2

Table1.2: Flags, bit-fields, formats

Format Required	Flag (arg1)	Bit-field (arg2)
Left justified output	ios::left	ios::adjustfield
Right justified output	ios::right	ios::adjustfield
Padding after sign or base indicator	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal base	ios::dec	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

The **ios** flags that do not have bit-fields are described in Table 1.3:

Table 1.3: ios Flags

Flag	Purpose
ios::showbase	Use base indicator on output
ios::showpos	Print '+' before positive numbers
ios::showpoint	Show trailing decimal point and zeros
ios::uppercase	Use uppercase letters for hex output
ios::skipus	Skip blank space on input
ios::unitbuf	Flush all streams after insertion
ios::stdio	Flush stdout and stderr after insertion

These flags are used with **setf ()** to produce desired outputs. The following example code segments demonstrate use of some of these flags:

Example Code Segment 1:

```
cout.fill ('*');
cout.setf (ios::left, ios::adjustfield);
cout.width (10);
cout << "OUTPUT 1" << "\n";
```

will produce the following output:

O	U	T	P	U	T		1	*	*

Example Code Segment 2:

```
cout.fill ('#');
cout.precision (2);
cout.setf (ios::internal, ios::adjustfield);
cout.setf (ios::scientific, ios::floatfield);
cout.width (12);
cout << -32.234 << "\n";
```

will produce the following output: (Sign is left justified and value is right-justified)

-	#	#	3	2	.	2	3	e	+	0	1

Example Code Segment 3:

```
cout.setf (ios::showpoint);
cout.setf (ios::showpos);
cout.precision (3);
cout.setf (ios::fixed, ios::floatfield);
cout.setf (ios::internal, ios::adjustfield);
cout.width (10);
cout << 123.4 << "\n";
```

will produce the following output: (output with sign and trailing zeros)

+			1	2	3	.	4	0	0

1.4.2 Stream Manipulators

C++ defines a number of functions called manipulators in header file **iomanip** that can be used to manipulate the output formats. Most of these manipulators are quite similar in function to the **ios** class functions and flags, though at times they are more convenient to use. The best thing is that two or more manipulators can be used in a single statement as:

cout << manip1 << manip2 << manip3 << manip4 << item;

Some of the commonly used stream manipulators and their effect is listed in Table 1.4.

Table1.4: List of Used Stream Manipulators

Manipulator	Purpose
setw (int w) setprecision (int d)	Set the field width to w. Set the floating point precision to d.
setfill (int c)	Set the fill character to c.
setiosflags (long f)	Set the format flag f.
resetiosflags (long f)	Clear the format flag f.
endif	Insert a new line and flush stream.

You can easily notice that in terms of functionality, **setw ()** has similar effect to ios function **width ()**, **setprecision ()** to ios function **precision ()**, **setfill ()** to ios function **fill ()**, **setiosflag ()** to ios function **setf ()**, **resetiosflags ()** to ios function **unsetf ()** and **endif** to **"\n"**. There is however a major difference between implementation of **ios** functions and stream manipulators. The **ios** member functions return the previous format state which can be used later but the manipulator does not return to previous format state.

Some example code segments illustrating use of various stream manipulators and their outputs are given below:

Example Code Segment 1:

```
cout << setw (10) << 12345;
```

will produce the following output: (12345 is displayed right-justified in a field width of 10)

					1	2	3	4	5

Example Code Segment 2:

```
cout << setw (5) << setprecision (2) << 1.2345;
cout <<  setw (10) << setprecision (4) << 3.1415435;
```

will produce the following output: (Sign is left justified and value is right-justified)

	1	.	2	3

				3	.	1	4	1	5

1.4.3 User Defined Stream Manipulators

C++ allows users to design their own customized stream manipulators. These user-defined manipulators may be non-parameterized or parameterized ones. The general syntax for creating a user-defined stream manipulator without any argument is as follows:

```
Ostream & manipulator (ostream & output)
{
        ……
        …… Statements for formatting
        ……
        return output;
}
```

Here, the manipulator is the name of user-defined manipulator to be created. For example, we can create the user defined manipulator unit that displays "INR" after each numerical value displayed.

```
ostream & unit (ostream & output)
  {
    output << "INR";
    return output;
  }
```

A more complex user-defined manipulator show can be defined as follows:

```
ostream & show (ostream & output)
{
output.setf (ios::showpoint);
output.setf (ios::showpos);
output << setw (10);
return output;
}
```

The following program presents a full blown example of designing user-defined stream manipulators. This program creates two user-defined manipulators currency and form, which are used to display output values in a specific manner.

```
# include <iostream.h>
#include <iomanip.h>
ostream & currency(ostream & output)
{
        output << "INR";
        return (output);
}
ostream & form(ostream & output)
{
        output.setf (ios::showpos);
        output.setf (ios::showpoint);
        output.fill ('*');
        output.precision (2);
        output << setiosflags (ios::fixed) << setw (10);
        return (output);
}
```

```
int main()
{
        cout << currency << form << 5465.4;
        return (0);
}
```

The output of the program would be INR **+5465.40.

Another example code which makes use of stream manipulators and other formatting techniques is given below. This program creates two user-defined manipulators area and volume, which are used to display output values in a specific manner. The value of area is displayed with SQ.MTS unit and the volume is displayed with CUBIC MTS unit.

```
# include <iostream.h>
#include <iomanip.h>
ostream & area(ostream & output)
{
        output << "SQ.MTS";
        return (output);
}
ostream & volume(ostream & output)
{
        output.precision (4);
        output << setiosflags (ios::fixed) << setw (15);
        output << "CUBIC MTS"
        return (output);
}
int main()
{
        cout << area << 3000;
        cout<< volume << 9000;
        return (0);
}
```

☞ Check Your Progress 2

1) Fill in the blanks:
   a) Member function ……. can be used to set and reset format state.
   b) The stream manipulators that format justification are ……., ……., and …….. .
   c) The ……. stream manipulator causes positive number to be displayed with a plus sign.
   d) The functionally equivalent ios function for stream manipulator setw () is …….

2) State whether True or False:
   a) The stream manipulators dec, oct and hex affect only the next integer output operations.
   b) The stream member function flags with a long argument sets the flags state variable to its argument and returns its previous value.
   c) By default, memory addresses are displayed in hexadecimal format.

3) For each of the following, show the output:
   a) cout << setw (10) << setfill ('$') << 10000;

   ……

   ……

   ……

   ……

   b) cout << showbase << oct << 99 << endl << hex << 99;

   ……

   ……

   ……

   ……

   c) cout << 10000 << endl << showpos << 10000;

   ……

   ……

   ……

   ……

4) For each of the following, write a single C++ statement that performs the desired task.
   a) Use a stream manipulator such that, when integer values are output, the integer base for octal and hexadecimal values is displayed.

   ……

   ……

   ……

   b) Print the current precision setting, using a member function of object cout.

   ……

   ……

   ……

   c) Print 1234 right justified in a 10- digit field.

   ……

   ……

   ……

   d) Print 1.92, 1.925 and 1.9258 separated by tabs and with 3 digits of precision, using a stream manipulator.

   ……

   ……

   ……

5) Create a user-defined manipulator *showcurr* that prints '#' sign before every numerical value displayed.

……

……

……

1.5 FILE STREAM OPERATIONS

Many real world applications require reading and writing large number of data items. Usually large volumes of data are stored as files on disk. Like many high level programming languages, C++ also provides mechanism to read and write data items from files. A C++ program can thus take input from a disk file and also write data to it. C++ file handling mechanism is quite similar to console input-output operations. It uses streams (called file streams) as an interface between programs and files. The Figure 1.3 illustrates the use of file streams for input and output:

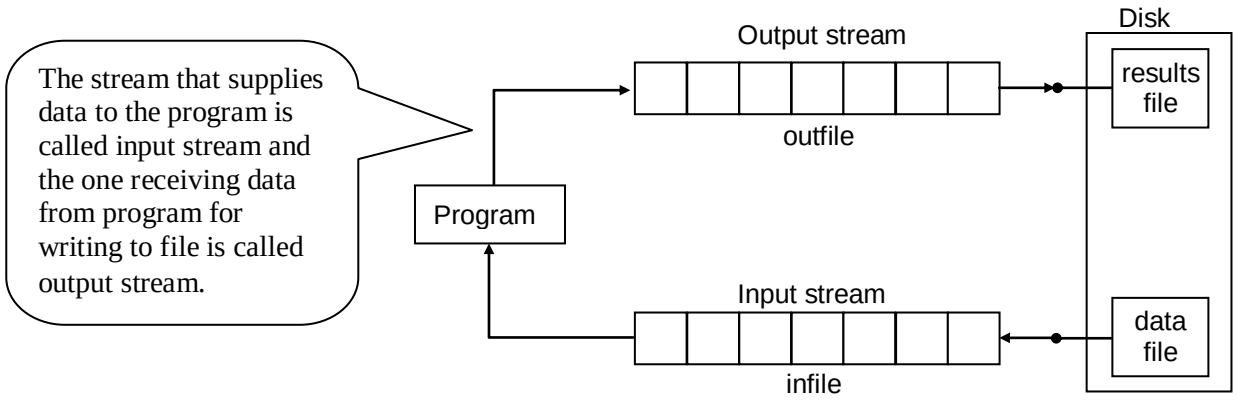


Figure 1.3 File Input and Output Steams

The Figure 1.3 shows a C++ program that reads data from one file and writes the output to another file (named results). The input stream is connected to data file used for input and output stream is connected to results file used for output. Programs can do reading and writing on the same file as well.

The I/O system of C++ contains a set of classes that provide methods for reading and writing from files. These classes are ifstream, ofstream and fstream. All these classes are derived from fstream base class and also inherits features from iostream class. The stream classes and their inheritance is shown in the Figure 1.4:

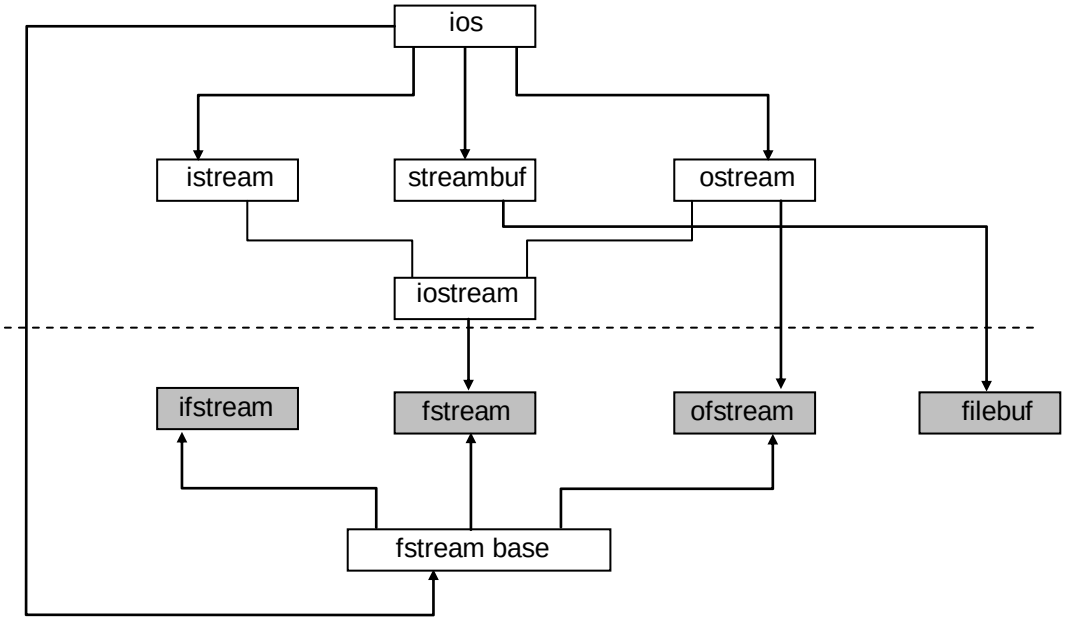


Figure 1.4 File Steam Classes

All these classes are declared in **fstream** and that is why this header file is included in all programs doing file processing. The **fstream** base class provides operations common to the file streams and contains **open()** and **close()** functions. It also serves as base class for the other three file stream classes. The **ifstream** class provides functions for input operations. This includes functions like **get()**, **getline()**, **read()**, **seekg()** etc. The **ofstream** class provides functions for output operations and include functions like **put()**, **write()** etc. The **fstream** class provides support for simultaneous input and output operations. It also inherits all functions from **istream** and **ostream** classes through **iostream**. The **filebuf** is used for setting the file buffers for read and write operations. This is required since volume of data is read and written to files.

Opening and Closing Files

Every disk file has a name (usually a string of valid characters). In order to read data from a file or to write data into it, we first need to open the file. This opened file has to be then connected to input-output stream so that the corresponding program becomes able read and write from it. A file stream can be defined using any of the three classes ifstream, ofstream or fstream, depending upon the purpose of the file.

A file can be opened either by using the **constructor** function of the class or by invoking member function **open ()** of the class. In order to open file using constructor function, we may use statements of the form:

```
ifstream infile (“data”); and
ofstream outfile (“results”);
```

Here, first statement opens a file called data and attaches it to the stream infile. This file can then be used for input operations. The second statement opens a file called results and attaches it to stream outfile. This file can only be used for output operations. Once the infile and outfile streams are created and connected to corresponding files, statements of the form:

```
outfile << "Sum";
outfile << "avg";
infile >> num;
infile >> name;
```

can be used to write data items to the output file results and to read data from input file data. After completing the input-output operations, the file may be closed by closing the corresponding stream. For example:

```
outfile.close();
infile.close();
```

Even if we fail to explicitly close a file, it gets closed automatically when the program terminates (and hence the corresponding stream expires). Nevertheless it is a good practice to close the open files once they are no longer required.

A file can also be opened using **open ()**. The general syntax of open () is as follows:

```
file-stream-class stream object;
stream-object.open("filename");
```

For example, to open a file data for input, we may use the statements:

```
ifstream infile;
infile.open("data");
```

Now, the stream infile can be used for reading data form the file "data" in a similar manner as that of console I/O. After the file's use is complete, it should be closed by closing the corresponding (by invoking close() function).

C++ allows the files to be opened in different modes. Hence, the open() function can specify the mode of opening as well. The statement invoking open can be written as:

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

The mode argument specifies the purpose for which the file is opened. The file mode parameter is defined in ios class and can take any of the following values:

Mode Parameter	Effect
ios::app	Append to end-of-file.
ios::ate	Go to end-of-file on opening.
ios::binary	Open a binary file.
os::in	Open file for input only.
ios::nocreate	Open fails if file does not exist.
ios::noreplace	Open already existing file.
ios::out	Open file for output only.
ios::trunk	Delete the contents of the file if it exists.

The function open() contains default values for these modes and hence even if the mode is not specified, the file is opened with default mode.

One must also be careful in reading from an input file so as to ensure that no read attempt is made once the file end is reached. We will see the use of EOF() member function of ios class for this purpose in next section.

An example program demonstrating how we can read from a disk file and write the results to the disk file is given below:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main()
{
        const int SIZE = 80;
        char line [SIZE];
        ifstream infile;
        ofstream outfile;
        infile.open("namelist");
        outfile.open("results");
        while(infile.eof()!=0)
        {
                infile.getline(line, SIZE);
                outfile << line;
        }
        return (0);
}
```

This program opens a file "namelist" for reading and "results" for writing. Then it continues to read the entire contents of file "namelist" and copies it to the file "results". The getline() stream member function is used for reading and the output content is simply directed to the stream "outfile" for writing it to the file "results". The reading (and subsequent copying) continues till the end of file "namelist" is reached. If one needs to display the contents while copying, a statement cout << line; may be added. This program can be made meaningful, if we copy the sorted namelist to the "results" file. This can be done by first sorting the contents of "namelist" file (temporarily storing it in an array of strings) and then writing the sorted values.

1.6 FILE POINTERS AND OPERATIONS

However, if we open a file in append mode, the output pointer is set to the end of file.

We can further control the reading and writing operations from file by manipulating the file pointers. Every file in C++ is marked by two pointers, one for input (called get pointer) and one used for output (called put pointer). The get pointer can be set to a specified location in order to reach from that desired location in the file. These pointers are automatically incremented every time after every read and write operation. You may be wondering how we have been able to read and write in our earlier programs without setting these pointers. Actually every time we open a file for input, the input pointer is automatically set to the beginning of the file.

We can manipulate these file pointers by invoking member functions of the file stream class. The commonly used functions for this purpose include **seekg()**, **seekp()**, **tellg()**, **tellp()** etc. The **seekg()** function moves the input (get) pointer to a specified location. The **seekp()** function moves to output (put) pointer to a specified location. The **tellg()** and **tellp()** functions tell the current position of get and put pointers, respectively. The general syntax for using **seekg()** and **seekp()** is given below:

```
seekg(offset, refposition);
seekp(offset, refposition);
```

Where, **offset** represents the number of bytes, the file pointer is to be moved from the location specified by the parameter refposition. The **refposition** may take one of the three positions defined in the **ios** class:

ios::beg – start of the file
ios::cur – current position of the pointer
ios::end – end of file.

The seekg() function moves the get pointer whereas the seekp() function moves the put pointer as per the parameters specified in the statement.

Functions to read from and write to files

Reading and writing operations in files are made simpler by C++ by providing some member functions in the file stream class. The functions **put()** and **get()** are used for handling a single character at a time. Functions **read()** and **write()** are designed to handle blocks of binary data. We also have functions like **getline().**

Two programming examples demonstrating use of **put()** – **get()** and **read() - write()** are given below:

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
int main()
{
        char name [80];
        cin >> name;
        int len = strlen(name);
        fstream file;
        file.open("text", ios::in I ios:out);
        for (int i =0; I < len; i++)
                file.put(name[i]);
        file.seekg(0);
        char c;
        while (file)
        {
                file.get(ch);
                cout << ch;
        }
        return (0);
}
```

The program above reads a string and puts it character by character in a file called "text". This file is opened for both reading and writing. Then the program goes to read the contents written in the file one character at a time by using get() function. Before doing that, the file pointer is set to the beginning of the file.

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
const char * filename = "BINARY";
int main()
{
        int num[5] = {12, 23, 34, 45, 56};
        ofstream outfile;
        outfile.open(filename);
        outfile.write((char *) & num, sizeof(num));
        outfile.close();
        for (int i=0; i<5; i++)
```

```
                num[i]= 0;
        ifstream infile;
        infile.open(filename);
        infile.read((char *) & num, sizeof(num));
        for (i=0; i<5; i++)
        {
                cout.setf(ios::showpos);
                cout << setw(5) << name[i];
        }
        infile.close();
        return (0);
}
```

The program above reads an integer array and then writes its contents to a binary file using write(). The file is then opened again this time for reading and the contents written in the file are read and displayed on the monitor. The read() function is used for reading.

☞ Check Your Progress 3

1) Fill in the blanks:

   a) Header file …… contains the declarations required for file processing.
   b) The ios mode …….. only opens a file that already exists, otherwise it fails.
   c) The function……. can be used to position the input file pointer at a desired location.

2) State whether the following statements are True or False.

   a) The file opening mode ios::ate positions the file pointer at end-of-file on opening.
   b) The function tellp() can be used to detect end of a file.
   c) A C++ program can do reading and writing on the same file.

3) For each of the following, write a single statement that performs the desired task:

   a) To open a file "text" in append mode and connect a stream infile to it.

   ……

   ……

   ……

   b) To read a single character in a file connected to stream file.

   ……

   ……

   ……

c) To read an entire line of text of *size* characters from a file connected to f1 file stream into a variable *line*.

……

……

……

d) To detect the end of a file connected to stream file.

……

……

……

1.7 SUMMARY

This unit has introduced you to the concept of streams and how they are used for console and disk I/O operations by a C++ program. The stream concept provides in C++ the flexibility of using the same mechanism for input-output operations from different devices. A stream is a sequence of bytes and it serves as both source and destination for an I/O data. C++ provides a hierarchy of stream classes that support different functions for managing I/O devices from console and disk. There are certain streams which are by default connected to standard devices. The console I/O operations may be unformatted or formatted. We use separate functions for both purposes. Output can be produced in a desired manner (formatted) by using one of the three mechanisms: ios class functions and flags, stream manipulators and user-defined stream manipulator functions. C++ also provides a large number of functions that can be used to read and write data from disk files. Files need to be opened and closed for I/O operations. Files may be opened in different modes depending upon the kind of operation to be performed. C++ provide different functions for manipulating file pointers and hence deciding where to read and write the data.

1.8 ANSWERS TO CHECK YOR PROGRESS

Check Your Progress 1

1. a) streams b) <iostream> c) >> d) cin, cout, cerr, clog
   e) ostream
2. a) False, It is connected to standard input, which is keyboard.
   b) True
   (b) False, It outputs its single character argument.
3. a) cout << “Enter your name”;
   b) cin.read (line, 50);
   c) cin.peek ()
   d) cout.write (line, cin.gcount());

Check Your Progress 2

1. a) flags b) left, right and internal
   c) showpos d) width ()

2. a) False, They have effect till it is reset or the program terminates.
   b) False, The stream member function flags with a fmtflags argument sets the flags state variable to its argument and returns the prior state setting.

c) True

3. a) $$$$$10000
   b) o143
      oX63
   c) 10000
      +10000

4. a) cout << showbase;
   b) cout << cout.precision ();
   c) cout << setw (10) <<1234;
   d) cout << setprecision (3) << 1.92 << '\t' << 1.925 << '\t' <<1.9258

5.
```
ostream & showcurr (ostream & output)
{
output << "#";
return output;
}
```

Check Your Progress 3

1. a) <fstream> b) ios::noreplace c) seekg()

2. a) True
   b) False, It is actually eof() function.
   c) True
3. a) ifstream infile; infile.open("text", "ios::app");
   b) file.get(ch);
   c) f1.getline(line, size);
   d) file.eof()

1.9 FURTHER READINGS

1) E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2) P. Deitel and H. Deitel, *C++: How to Program, PHI,* 7th edition, 2010.
3) B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.

UNIT 2 TEMPLATES AND STANDARD TEMPLATE LIBRARY

Structure **Page Nos.**

2.0 INTRODUCTION

In the previous units, we have discussed about different features and functionalities of C++ programming language for object oriented programming. C++ being an object oriented language also supports reuse. Templates are one of the most prominent examples of reuse concept in action. This feature has been added to the C++ design recently. It supports the idea of generic programming by providing facility for defining generic classes and functions. Thus a template class provides a broad architecture which can be used to create a number of new classes. Similarly, a template function can be used to write various versions of the function. This chapter describes the template mechanism in C++ based on the paper titled 'Parameterized types for C++' by Bjarne Stroustup (creator of C++) published in Proceedings of the USENIX C++ Conference held in Denver, Colorado in October 1988.

This unit aims to explain the basic idea and motivation for templates. The unit explains in detail (with appropriate examples) the design and use of both Class and Function Templates. It then proceeds further to explain the use of templates and how they are related to the concepts of overloading and inheritance. We then discuss the structure and utility of the Standard Template Library in C++. Further, the unit describes the three core components of the Standard Template Library: containers, algorithms and iterators. These features are used in many real world application designs. The unit concludes with a summary of the Template and Standard template Library framework followed by model answers to assist you in check your progress exercises.

2.1 OBJECTIVES

At the end of the unit, you should be able to:

- identify the concept of Templates in C++;
- understand the definition and usage of class and function templates;
- write your own class and function templates;
- understand the design of class and function templates with and without parameters;
- visualize the idea behind design of the Standard Template Library;
- describe the components of the Standard Template Library and understand their use; and

- appreciate the use of the Standard template Library in real world application designs.

2.2 CLASS TEMPLATES

You might have got at least an idea from the introduction that templates are like stencils out of which we trace shapes. Function-template specializations and class-template specializations are like the separate tracings that all have the same shape, but could, for example, be drawn in different colours. In other words, a template may be considered as a kind of macro. When the actual object of that type is to be defined, the template definition is substituted with required data type. For example, if we define a template Array of elements, then this same generic definition may be used to create Array of integers or of characters or float quantities. We need not make a new class definition every time. We define a generic class with a parameter that s replaced by a particular data type at the time of actual use of that class. This is the reason template classes are also known as parameterized classes.

Designing a template class thus involves a simple process of creating a generic class with an anonymous type. The general syntax for defining a class template is:

```
template <class T>
class classname
{
……..
…….class specification with anonymous type T
……..
};
```

For example, consider the following template definition for a **Vector** class:

```
template<class T>
class vector
{
  T * v;  // the vector is of type T
  int size;
Public:
  vector(int m)
  {
    v = new T [size = m];
    for (int i=0; i<size; i++)
        v[i]=0;
  }
  vector (T * a)
  {
    for(int i=0; i<size; i++)
        v[i] = a[i];
  }
  T operator * (vector &x)
  {
   T sum = 0;
   for(int i=0; i<size; i++)
       sum += this->v[i] *x- v[i];
   return (sum);
  }
};
```

This definition creates a template class *vector* of type T with variables, constructors and '*' operator. This class definition is similar to an ordinary class definition except that of the use of **template<class T>** and use of type **T** inside the class definition. The template<class T> tells the compiler that it is a template class with parameter T instead of a normal class definition. The declaration thus creates a parameterized class with T as the parameter, which can be substituted with any valid data type. This can be done by a statement of the following form:

- classname <type> objectname(argument list);

For example, following statements create classes of 20 element integer and float vectors, respectively.

- vector <int> v(20);
- vector <float> v(20);

This task of creating an actual object from a template class is instantiation. Here we have written only one class definition for class *vector* but we have been able to create more than one actual instantiations of the template class *vector*.

A class template can also be created by using multiple generic data types as arguments. The general syntax for such definition would be as below:

```
template <class T1, class T2, …….>
class classname
{
……..
…….class specification with anonymous type T
……..
};
```

A simple program demonstrating the declaration and use of class templates is given below. Please note that this program instantiates two objects from the class template Example. The program first declares a template class Example with two arguments and then declares a constructor to instantiate the class. The main function creates two objects test1 and test2 of different types and their values are then displayed using the function show.

```
#include <iostream>

template<class T1, class T2>
class Example
{
  T1 x;
  T2 y;
 Public:
  Example(T1 a, T2 b)
  {
    x = a;
    y = b;
  }
  void show ()
  {
    cout << x << "and" << y << "\n";
  }
};
```

```
int main()
{
  Example <float, int> test1 (3.45, 345);
  Example <int, char> test2 (100, 'm');
  test1.show();
  test2.show();
return(0);
};
```

The program creates two template classes test1 and test2 using the template class Example. The test1 class has two parameter values "3.45" and "345", whereas test2 class has two parameter values "100" and character "m". For creating test1 object, arguments are float and integer respectively, whereas in case of test2 object they are integer and character. The values displayed in invocation of show() function from main will be "3.45 and 345" for test1 and "100 and m" for test2 object.

2.3 FUNCTION TEMPLATES

Function templates are similar to class templates in the sense that they create a generic function type. This generic function can then be used to create a family of functions that may take different arguments. A function template can be defined as follows:

```
template <class T>
return_type function_name (arguments of type T)
{
……..
…….body of function with argument of type T
……..
};
```

Function templates are another way of handling overloaded function requirements. If overloaded functions perform identical operations for different type of data then they can be more appropriately and conveniently declared as function templates. The following example demonstrates creation of a function template swap:

```
Template <class T>
void swap(T & x, T& y)
{
T temp = x;
x = y;
y = temp:
};
```

The swap() function can now be invoked like any ordinary function. Any call to swap() with input arguments will exchange the values contained in those arguments. Hence if a and b are integer variables and p and q are float variables; we may invoke swap() function as swap(a,b) and swap(p,q), respectively. The same function definition can be used to swap values of two variables of different types.

As we have discussed earlier, we can define class templates with multiple arguments, we can also define function templates with multiple arguments. This can be done through a declaration of the form:

```
template <class T1, class T2, ……..>
return_type function_name (arguments of type T1, T2,….)
{
……..
…….body of function
……..
};
```

The function and class templates can be used to write programs which work correctly on different types of data. For example, we can write the following program to sort a list in desired order using function templates swap() and bsort() as shown in the program below:

```
 #include <iostream>

template<class T>
void bsort(T a[], int n)
{
  for (int i=0; i<n-1; i++)
    for (int j=n-1; i<j; j--)
        if (a[j] < a[j-1])
          swap(a[j], a[j-1]);
 }
template <class X>
void swap( X &a, X &b)
{
  X temp =a;
  a = b;
  b = temp;
}

int main()
{
  int x[5] = {10,50,30,60,40};
  float y[5] = {3.2, 71.5, 17.3, 45.9, 92.7};

bsort(x,5);
bsort(y,5);

cout << "Sorted X-Array:";
for (int i=0; i<5; i++)
   cout << x[i] << " ";
cout << endl;

cout << "Sorted Y-Array:";
for (int j=0; j<5; j++)
   cout << y[j] << " ";
cout << endl;
return(0);
};
```

This program uses two function templates swap() and bsort(). The function template swap() is invoked within the bsort() function and is hence said to be nested in it. This program can be used to sort different types of lists without the need of modifying the program. The program will produce following output:
Sorted X-Array: 10 30 40 50 60

Sorted Y-Array: 3.2 17.3 45.9 71.5 92.7

A template function may also be overloaded in a manner similar to other functions. In fact, function templates and overloading are intimately related. All the function-template specializations generated from a function template have the same name, so the compiler uses overloading resolution to invoke the proper function. A function template can be overloaded in several ways:

a) functions having same name but different parameters
b) providing a non-template functions with the same function name but different arguments

Whenever, the compiler has to perform the matching process to determine what function to call, it achieves the overloading resolution as follows:

a) call an ordinary function that has an exact match
b) call a template function that could be created with an exact match
c) try normal overloading resolution to ordinary functions and call the one that matches.

In case no match is found, the compiler generates an error. In case there are multiple matches for the function call, the compiler considers the call to be ambiguous and the compiler generates an error message. It would also be worth mentioning that no automatic conversions are applied to arguments on the template functions.

☞ Check Your Progress 1

1) Fill in the blanks:
   a) Templates enable us to specify, with a single code segment, an entire range of related functions called ……..……, or an entire range of related classes called…….…… .
   b) All function template definitions begin with the keyword…… followed by a list of template parameters to the function template enclosed in…….. ..
   c) Class templates are also called ……..…… types.
   d) The …….. operator is used with a class template name to tie each member function definition to the class template's scope.

2) State whether following are *True* or *False*.
   a) A function template can be overloaded by another function template with the same function name.
   b) Template parameter names along template definitions must be unique.
   c) Each member function definition outside a class template must begin with a template header.
   d) Keywords *typename* and *class* as used with a template type parameter specifically mean "any user-defined class type."

3) Write appropriate statements to create a function template *printarray* that can display the values contained in array passed as parameter to the function. The function must be able to accept integer, float and character arrays as arguments.

   ……

   ……

   ……

4) Write appropriate statements to create a template class item that can instantiate objects of at least following types: item name: shirt, measure: size (expressed as characters 'S', 'M', 'L' and 'X') and item name: trouser, measure: size (expressed as waist size of integers). The template class must also have a template function to display the details of the items.

……

……

……

2.4 USE OF TEMPLATES

The concept of class templates and function templates derives its motivation from the principle of reuse. We have seen in earlier sections how templates allow us to create generic data types and functions that can fit into various situations. Rather than defining multiple classes and functions, we define a generic type and depending on the kind of input data it may customize itself. Templates in this sense serve as a blueprint for defining classes and functions. This not only eliminates code duplication for handling different data types but also makes the program development easier and more manageable. In the previous section, we saw an example of program for sorting that can a sort lists of various types. We present below another example demonstrating use of templates for solving a quadratic equation:

```
 #include <iostream>
#include <iomanip>
#include <cmath>

template<class T>
void roots(T a, T b, T c)
{
  T d = b*b - 4*a*c;
  if (d==0)
   {
     cout << "R1 = R2 ="  << -b/(2*a) << endl;
   }
  else if (d > 0)
   {
      cout << "Roots are real \n";
      float R = sqrt (d);
      float R1 = (-b + R) / (2*a);
      float R2 = (-b - R) / (2*a);
      cout << "R1 =" << R1 << "and" ;
      cout << "R2 =" << R2 << endl;
    }
   else
   {
      cout << "Roots are complex \n";
      float R1 = -b /(2*a);
      float R2 = sqrt (-d) / (2*a);
      cout << "Real part =" << R1 << endl;
      cout << "Imaginary part =" <<R2 <<endl;
    }
 }

int main()
```

```
{
  cout << "Integer coefficients \n";
  roots (1, -5, 6);
  cout << "\n Float coefficients \n";
  roots (1.5, 3.6, 5.0);

  return(0);
};
```

The program will generate following output:

```
Integer coefficients
Roots are real
R1 = 3 and R2 = 2
Float coefficients
Roots are complex
Real part = -1.2
Imaginary part = 1.375985
```

As we can see, the program above can be used to compute roots of a quadratic equation having different kinds of coefficients. It calculates roots for an equation having integer coefficient and for another equation having float coefficients. The templates have become so popular that now we have a rich library of predefined templates in C++, known as the Standard Template Library. We will discuss the contents and use of the standard template library in next section.

2.5 THE STANDARD TEMPLATE LIBRARY

The STL was developed by Alexander Stepanov and Meng Lee at Hewlett-Packard while pursuing their research in generic programming, with significant contributions from David Musser.

Recognizing that many data structures and algorithms are commonly used, the C++ standards committee added the Standard Template Library (STL) to the C++ standard library. The STL defines powerful, template-based, reusable components that implement many common data structures, and algorithms used to process those data structures. STL is a large collection of generic classes and functions. This large collection can be grouped at its core into three categories:

- Containers
- Algorithms, and
- Iterators.

These components work in conjunction with each other to provide solution to complex programming problems. A statement summarising their relationship could be "*Algorithms employ iterators to perform operations stored in containers*". The figure 2.1 further elaborates this relationship:

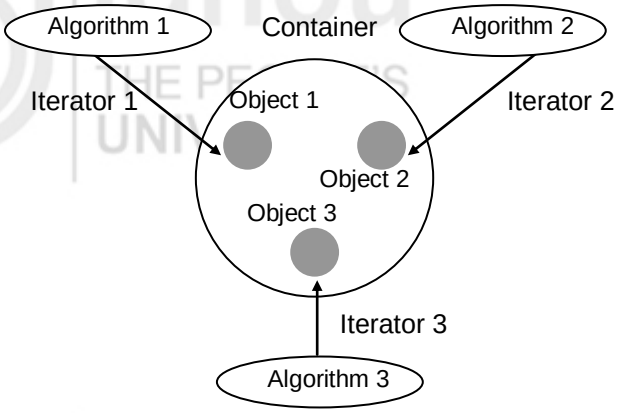


Figure 2.1 : Relationship between the three STL components

A *container* is an object that actually stores data. It is a way data is organized in memory. The STL containers are implemented by template classes and hence can be easily customized to hold different types of data. An *algorithm* is a procedure that is used to process the data stored in the containers. The STL include many different kinds of algorithms such as searching, sorting, copying, merging etc. Algorithms are implemented by template functions. An *iterator* is an object that works like a pointer. It is used to point to elements in a container. Iterator value may be incremented or decremented just like pointers. They play a key role in accessing and manipulating various data structures.

Containers

Containers are objects that hold data. These container classes are defined as class templates that can be customized to hold different kinds of data. The Figure 2.2 illustrates the three main types of container classes. The container classes contain definitions for commonly used data structures such as vector, list, queue, stack, set, map etc. Each container class also defines a set of functions that can be used to manipulate its contents. The STL defines a number of containers which can be grouped into mainly three types: sequence containers, associative containers and derived containers. The derived containers are sometime also referred to as container adapters.

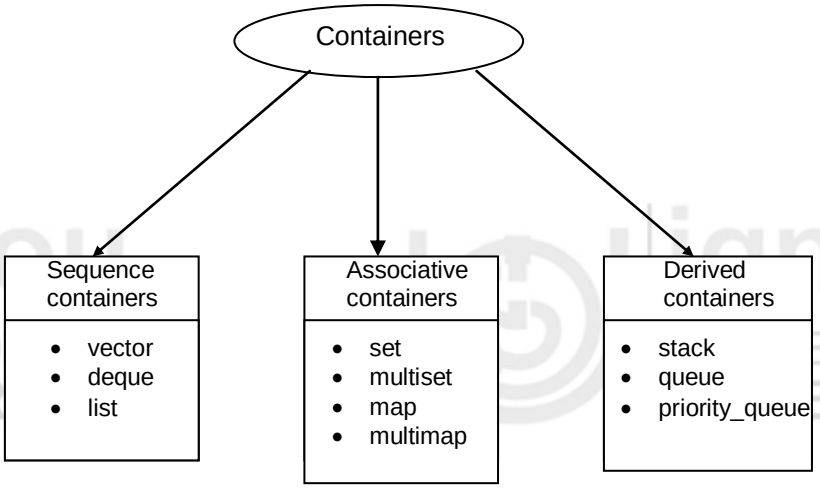


Figure 2.2 : Main Container types

The Table 2.1 lists some commonly used container classes available in the STL.

Table 2.1: List of commonly used Container

Container	Description	Header File	Iterator
vector	A dynamic array. Allows insertion and deletions at rear. Permits direct access.	<vector>	Random access
list	A bidirectional linear list. Allows insertion and deletions anywhere.	<list>	Bidirectional
stack	A standard stack, Last in First out operation.	<stack>	No iterator
queue	A standard Queue, First in First out operation.	<queue>	No iterator
priority queue	A priority queue, highest priority element as first out.	<queue>	No iterator
deque	A double ended queue, allows	<deque>	Random

	insertions and deletions at both ends.		access
set	An associative container for storing unique sets. Allows fast lookup.	<set>	Bidirectional
multiset	An associative container for storing non-unique sets.	<set>	Bidirectional
map	An associative container for storing unique key-value pairs.	<map>	Bidirectional
multimap	An associative container for storing key-value pairs that may use one-to-many mapping.	<map>	Bidirectional

The sequence containers represent linear data structures such as vectors and linked lists. The associative containers are non-linear container that typically store elements in a key-value pair fashion and support fast lookup. The sequence containers and associative containers are collectively referred to as *First Class containers*. Stacks and Queues are actually constrained versions of these first class containers and that is why often referred to as derived containers or container adapters. They enable a program to view a sequential container in a constrained manner. Sometimes we also hear about "near conatiners", which are similar to first class containers but do not support all functionalities of first class containers. Bitsets is one such example.

Hence, out of the various container classes listed in the table, vector, list and deque are sequential containers. Set, multiset, map and multimap are associative containers. Stack, queue and priority queue are derived containers.

Most STL containers provide similar functionality. Many generic operations, such as member function size, therefore apply to all containers. A good number of operations apply on subsets of container classes. Some of the common member functions that apply to most of container classes are listed in the Table 2.2:

Table 2.2: Some common member functions of Container Classes

Member Functions
default constructor, copy constructor, destructor, empty, insert, size, operator=, operator>, operator<, operator<=, operator>=, operator==, operator!=, swap
Functions found only in First class containers
max_size, begin, end, rbegin, rend, erase, clear.

Algorithms

Algorithms are functions that are used across a variety of container classes for processing their contents. As we have just learnt that each container class provides member functions for its basic operations, but STL further extends this by providing some standard algorithms for manipulating different containers. The STL contains approximately seventy standard algorithms to support more extended or complex operations. Standard algorithms have another advantage that they allow working with two different types of containers at the same time, unlike container member functions. The STL implement these algorithms as standalone function templates that can be customized to work with different kind of containers. Inserting, deleting, searching and sorting are some of the examples.

Unlike member functions, the algorithms operate on containers indirectly through the use of iterators. Many algorithms operate on sequences of elements defined by pair of iterators- one pointing to the first element of the sequence and other pointing to one

element past the last element. It is also possible to create new algorithms that operate in a similar fashion to that of STL algorithms. Algorithms often return iterators that indicate the results of algorithms (for example algorithm find). STL algorithms, based on the nature of operations they perform, may be categorized into following groups:

- Retrieve or non-modifying sequence algorithms
- Mutating-sequence algorithms
- Sorting Algorithms
- Set Algorithms
- Relational Algorithms

A list of some of these algorithms along with a description of their purpose is given in the Table 2.3:

Table 2.3: Mutating sequence Algorithm

Mutating-sequence algorithms	
copy()	Copies a sequence
copy_backward()	Copies a sequence from the end
fill()	Fills a sequence with a specified value
fill_n()	Fills first n elements with a specified value
generate()	Replaces all elements with the result of an operation
generate_n()	Replaces first n elements with the result of an operation
iter_swap()	Swaps elements pointed to by iterators
random_shuffle()	Places elements in random order
Remove()	Deletes elements of a specified value
remove_copy()	Copies a sequence after removing a specified value
remove_copy_if()	Copies a sequence after removing elements matching a predicate
remove_if()	Deletes elements matching matching a predicate
replace()	Replaces elements with a specified value
replace_copy()	Copies a sequence replacing elements with a given value
replace_copy_if()	Copies a sequence replacing elements matching a predicate

Non-modifying sequence algorithms	
adjacent_find()	Finds adjacent pair of objects that are equal
count()	Counts occurrence of a value in a sequence
count_if()	Counts number of elements that matches a predicate
equal()	True if two ranges are the same
find()	Finds first occurrence of a value in a sequence
find_end()	Finds last occurrence of a value in a sequence
find_first_of()	Finds a value from one sequence in another
find_if()	Finds first match of a predicate in a sequence
for_each()	Apply an operation to each element
mismatch()	Finds first elements for which two sequences differ
search()	Finds a subsequence within a sequence
search_n()	Finds a sequence of a specified number of similar elements

Sorting algorithms	
binary_search()	Conducts a binary search on an ordered sequence
equal_range()	Finds a sub range of elements with a given value
inplace_merge()	Merges two consecutive sorted sequences
lower_bound()	Finds the first occurrence of a specified value
make_heap()	Makes a heap from a sequence
merge()	Merges two sorted sequences
nth_element()	Puts a specified element in its proper place
partial_sort()	Sorts a part of a sequence
partial_sort_copy()	Sorts a part of a sequence and then copies
partition()	Places elements matching a predicate first
pop_heap()	Deletes the top element
push_heap()	Adds an element to heap
sort()	Sorts a sequence
sort_heap()	Sorts a heap
stable_partition()	Places elements matching a predicate first matching relative order
stable_sort()	Sorts maintaining order of equal elements
upper_bound()	Finds the last occurrence of a specified value

Set algorithms	
includes()	Finds whether a sequence is a subsequence of another
set_difference()	Constructs a sequence that is the difference of two ordered sets
set_intersection()	Constructs a sequence that contains the intersection of ordered sets
set_symmetric_difference()	Produces a set which is the symmetric difference between two ordered sets
set_union	Produces sorted union of two ordered sets

Relational algorithms	
equal()	Finds whether two sequences are the same
lexicographical_compare()	Compares alphabetically one sequence with other
max()	Gives minimum of two values
max_element()	Finds the maximum element within a sequence
min()	Gives minimum of two values
min_element()	Finds the minimum element within a sequence
Mismatch()	Finds the first mismatch between the elements in two sequence

Iterators

Iterators are used to access container class elements. They are called iterators because of their use in traversing the elements (from one to another) of a container class. In this sense they are quite similar to pointers. Iterators hold state information sensitive to the particular containers on which they operate, thus, iterators are implemented appropriately for each type of container. Certain iterator operations are uniform across containers. For example, ++ operation on an iterator moves it to the next element of the container. If iterator i points to a particular element, then, i++ points to the "next" element and *I refers to the element pointed by i.

There are five broad types of iterators supported by the STL. These are listed in Table 2.4:

Table 2.4: Types of Iterators

Iterator	Access method	Movement	I/O Capability
Input	Linear	Forward only	Read only
Output	Linear	Forward only	Write only
Forward	Linear	Forward only	Read/ Write
Bidirectional	Linear	Forward & Backward	Read/ Write
Random	Random	Forward & Backward	Read/ Write

Each type of iterator is used for performing a particular set of functions. The input and output iterators are used to traverse a container and have functionality limited to this use. The forward operator also supports input and output and at the same time retaining its position in the container. The bidirectional iterator provides ability to move backwards in addition to forward movements. The random access iterator allows random jumps to a particular location in addition to bidirectional operations.

Examples of use of List and Map containers

Now that we had a look at the organization and different components of the STL, we will go through some illustrative examples of use of the STL components. Since the STL is quite big and we can not cover examples on the entire STL, we will see here one example each of the use of List and Map container classes.

List is a commonly used container class that implements a standard bidirectional linked list. It supports insertion and deletion operations and can be accessed only in a sequential manner. The STL class list provides appropriate set of member functions to manipulate lists. We will see here an example of use of list container class for creating and processing a list:

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

void display(list<int> &lst)
{
  list<int> :: iterator p;
  for (p=lst.begin(); p!=lst.end(); ++p)
        cout << *p << " " ;
  cout << "\n";
}

int main()
{
 list<int> list1; // empty list of zero length
 list<int> list2(5); //empty list of 5 elements

 for (int i=0; i<3; i++)
    list1.push_back(rand()/100);

 list<int> :: iterator p;
 for (p=list2.begin(); p!=list2.end(); ++p)
   *p=rand()/100;
 cout << "list1 \n";
 display(list1);
```

```
 cout << "list2 \n";
 display(list2);

//Add elements at both the ends of list1
  list1.push_front(100);
  list1.push_back(200);

 //Remove an element at front of list2
  list2.pop_front();

  cout << "now list1 \n";
  display(list1);
  cout << "now list2 \n";
  display(list2);

  list<int> listA, listB;
  listA=list1;
  listB=list2;

  //Merging two lists
  list1.merge(list2);
  cout << "Merged unsorted list \n";
  display(list1);

  //Sorting and Merging
  listA.sort();
  listB.sort();
  listA.merge(listB);
  cout << "Merged sorted list \n";
  display(listA);

  return(0);
};
```

The program above creates various lists and performs different operations on them. It uses member functions like begin(), end(), push_back(), push_front() etc. It also sorts and merges two lists. The user-defined function display() makes use of iterator to display the elements of the lists.

Another example that we would consider is that of container class map. As we discussed earlier a map is a sequence of key:value pairs, where a single value is associated with each unique key. Its an associative container class. The entries in a map are specified as:

phone ["ashok"] = 123456;

Here, phone is a map object and the statement associates number 123456 with the key value "ashok". The map entries can be manipulated using various member functions and algorithms. The key operations in a map include operations like add, delete, modify, sort the entries in map etc. We will have a look at an example of use of map in the program below:

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

```
typedef map<string, int> phonemap;

int main()
{
 string name;
 int number;
 phonemap phone;

// Entering key:value pairs in map
 cout << "Enter three sets of name and numbers \n";
 for (int i=0, i<3; i++)
  {
    cin >> name;
    cin >> number;
    phone[name] = number;
  }

// inserting a new entry
phone["Ramesh"]= 621345;

 //inserting using insert() function
 phone.insert(pair<string, int> ("ajay", 234432));
 int n = phone.size();
 cout << "\n size of map:"  << n;

 // reading the entries in the map using iterator
 cout << "\n List of telephone numbers \n";
 phonemap::iterator p;
 for (p=phone.begin(); p!=phone.end(); p++)
    cout << (*p).first << "  " << (*p).second << "\n";

 cout << "\n";

 //looking up for an entry
 cout << "Enter name:";
 cin >> name;
 number = phone[name];
 cout << "Number:" << number << "\n";

 return(0);
};
```

This program first creates a map (phone) with three entries read from the keyboard. Then it moves to insert two new entries using two ways. It then prints the entire map using iterator. Finally it looks up the value of a given key stored in the map.

Similar to list and map container classes, the other container classes can also be used as the situation desires. We can use vector container class to create and manipulate various arrays; stack for handling LIFO memory operations; queue and priority queues for managing queue operations etc. We can manipulate these data structures not only by the member functions they contain, but also by using various algorithms that apply to them. Iterators help and guide the processing of elements contained in the class. The STL provides a rich set of template declarations along with different algorithms that are very useful in designing and implementing programming solutions for different real world problems. It supports and promotes reuse, a key theme of object oriented programming.

☞ Check Your Progress 2

1) Fill in the blanks:
   a) The two types of first-class STL containers are sequence containers and …….. containers.
   b) The five main iterator types are ……., ……..., ……., ……., and ……. .
   c) The three STL container adapters are ……., ……., and ……. .
   d) STL algorithms operate on container elements indirectly, using …… .
   e) The sort algorithm requires a (n) …… iterator.

2) State whether following are *True* or *False*.
   a) The STL makes abundant use of inheritance and virtual functions.
   b) An iterator acts like a pointer to an element.
   c) STL algorithms can operate on C-like pointer-based arrays.
   d) STL algorithms are encapsulated as member functions within each container class.
   e) Container member function end yields the position of the container's last element.

3) For each of the following, write a single statement that performs the indicated task:
   a) Name the member functions that are used to refer to beginning and end of the list class.

      ……

      ……

      ……

   b) Name the different type names used to categorize the algorithms.

      ……

      ……

      ……

   c) What is the purpose of push_back(), push_front(), pop_back() and pop_front() functions of a list.

      ……

      ……

      ……

2.6 SUMMARY

This unit has introduced the basic idea of template classes and functions. Templates are mechanisms supported by C++ for generic programming. Templates allow us to generate a family of classes or a family of functions to handle different data types. Template classes and functions promote reuse and avoid code duplication. The member functions of a class template are also defined using the parameters of the class templates.

C++ now contains a rich set of template classes and functions packaged as a library, known as the standard template library. The STL consists of three main components: containers, algorithms, and iterators. Containers are objects that hold data of some type and are usually grouped into three types: sequential, associative and derived. Container classes contain a large number of member functions that make manipulating them simple. In addition to member functions, we also have a large number of algorithms (such as sorting, searching, copying, and merging) that are used to manipulate the container classes and perform various operations on them. Iterators, which are similar to pointers allow manipulation of elements of container classes indirectly by algorithms.

2.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1. (a) function-template specialization, class-template specialization
   (b) template<…….>
   (c) parameterized
   (d) binary scope resolution

2. (a) True
   (b) False, it need not be unique.
   (c) True
   (d) False, This also allows for a type parameter of a fundamental type

3.
```
template<typename T>
        void printarray(T a[], int n)
        {
         for (int i=0; i<n-1; i++)
            cout << a[i] << " ";
         }
```

4.
```
template<class T1, class T2>
        class item
        {
          T1 x;
          T2 y;
        Public:
         item(T1 a, T2 b)
         {
           x = a;
           y = b;
         }
        template<typename T>
        void display(T a, T b)
        {
           cout << "item name" << a;
          cout << "item measure" << b ;
        }
```

Check Your Progress 2

1. (a) Associative
   (b) input, output, forward, bidirectional and random access
   (c) stack, queue, priority queue
   (d) iterators
   (e) random access

2. (a) False, These are avoided for performance reasons.
   (b) True
   (c) True
   (d) False, STL algorithms are not member functions. They operate indirectly on container classes through iterators.
   (e) False, it actually yields the position just after the end of the container.

3. (a) begin() and end()
   (b) non-modifying, mutating, sort, set and relational
   (c) push_bac() – is used to insert an element at the back of a list
   push_front() – is used to insert an element at the front of the list
   pop_back() – deleting an element from the back of the list
   pop_front() – deleting an element from the front of the list

2.8 FURTHER READINGS

1) E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2) P. Deitel and H. Deitel, C++: *How to Program, PHI*, 7th ed, 2010.
3) B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.
4) B. Stroustrup, *"Parameterized types for C++" Proceedings of the USENIX C++* Conference, Denver, Colorado, October 1988.

UNIT 3 EXCEPTION HANDLING

Structure **Page Nos.**

3.0 INTRODUCTION

In the units covered so far, we have talked about various features provided by C++ to design different programs. The programs which are written correctly produce the desired output when input data is supplied to them. However, in some cases programs may behave in an unexpected manner. One of the reasons that may lead to this situation is when we provide inappropriate input data (something that the programmer never expected or anticipated). These situations are unexpected and that is why they are termed exceptions. Exceptions are different from syntactical and logical errors but they also cause the program to misbehave. Exceptions are usually encountered at run time. In order to ensure that programs work correctly under all conditions, we have to incorporate mechanisms to identify and handle different exceptions that may occur in a program.

This unit introduces the nature and type of exceptions that may occur in a C++ program. It then describes the steps in exception handling. Then the syntax and use of try, throw and catch expressions are explained with appropriate examples. This unit tries to present a comprehensive picture of the exception handling mechanisms in C++ and their use in designing correct and robust programs.

3.1 OBJECTIVES

At the end of the unit, you should be able to:

- know what is exceptions and how to handle them;
- use try, catch and throw expressions to identify and handle exceptions;
- describe the standard exception hierarchy;
- appreciate the usefulness of exception handling mechanisms in C++; and
- write robust and fault tolerant C++ programs.

3.2 EXCEPTIONS IN C++ PROGRAMS

The name ‘exception’ implies that the problem is one which occurs infrequently- if the “rule” is that a statement normally executes correctly, then the “exception to the rule” is that a problem occurs.

The term exception itself implies an unusual condition. Exceptions are anomalies that may occur during execution of a program. Exceptions are not errors (syntactical or logical) but they still cause the program to misbehave. They are unusual conditions which are generally not expected by the programmer to occur. An exception in this sense is an indication of a problem that occurs during a programs’s execution. The

typical exceptions may include conditions like divide by zero, access to an array outside its range, running out of memory etc.

For example: Consider the following code segment:

```
# include <iostream>
                    int main()
                    {
                      int x,y;
                      cout << "enter values of x & y \n";
                      cin >> x;
                      cin >> y;
                      cout << "result of x divided by y is:" <<
x/y;
                     }
```

This program reads values of variables x and y from standard input and produces output of x divided by y, which is then displayed on the standard output. However, please note that if the value of y read from the keyboard is '0', then the program witnesses a divide by zero situation. In this case, the result will be infinite and program terminates.

To deal with these unexpected conditions, C++ provides an exception handling mechanism that detects and handles exceptions in a predefined way. Exception handling mechanism was not part of the original C++ specifications. It was added later and now almost all compilers support this feature. C++ exception handling mechanism provide with a scheme to identify and handle predictable exceptions that may occur during program execution. It may kindly be noted that exception handling mechanism needs to be incorporated explicitly in the program to handle the exceptions and that it may be able to handle only those exceptions that are provisioned in exception handling statements.

Exceptions can be of two kinds: *asynchronous and synchronous*. The exceptions that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions. The other unusual conditions (such as overflow, out of range array index) that are part of the program are called synchronous exceptions. The C++ exception handling mechanism is designed to handle synchronous type of exceptions only. It provides a means to detect, report and act on an unusual condition. The exception handling mechanism C++ deals with exceptions by performing following tasks:

a. Identify the problem (**hit** the exception)
b. Inform that an exception has occurred (**throw** the exception)
c. Exception handler catches the exception information (**catch** the exception)
d. Exception handler takes corrective actions (**handle** the exception)

These tasks are incorporated in C++ exception handling mechanism in two segments, one to detect (**try**) and inform (**throw)** about the exception, and the other to catch the exception and take appropriate actions to handle it.

As indicated earlier, the C++ exception handling mechanism is built upon following three expressions:

- Try
- Throw
- Catch

The expression **try** is used to preface (enclose in braces) that block which may generate exceptions. This block is often termed as try block. The **throw** expression is invoked when an exception is detected. It informs the catch block that an exception has occurred. The exception handling code is enclosed in **catch** block. The catch block catches the exception thrown by the throw expression and handles it in a predefined manner. The Figure 3.1 further shows the use of these three expressions and their relationship:

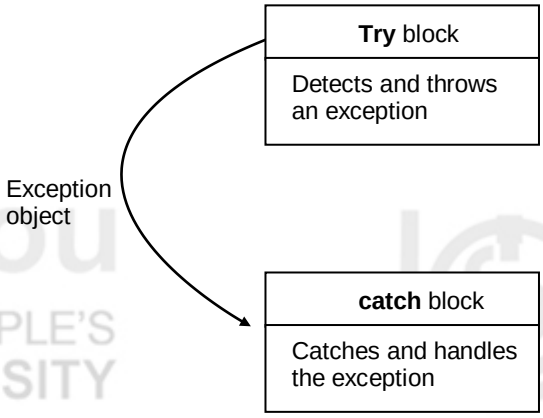


Figure 3.1 : Exception Handling

3.3 TRY, THROW AND CATCH EXPRESSIONS

The keyword 'try' is used to enclose the statements that may generate an exception. These statements begin with a try keyword and are enclosed in braces. When the try block encounters an exception, it uses the throw keyword to pass the information to the exception handling block. The exception handling block is enclosed within a block preceding the catch keyword. This catch block should immediately follow the try block that throws the exception. The general syntax of these statements and blocks is as follows:

```
……..
……..
try
{
  ……
  ……
  throw exception;
  ……
  ……
 }
catch (type arg)
{
  ……..
  ……..
  ……..
 }
……
……
```

As some statement within the try block of the program generates an exception, it is thrown using the throw statement. At this time, the program control transfers to the

catch block. The throw contains an argument named exception which is passed to the catch block as an argument. However, the catch block handles this exception only if the arguments passed from throw matches with the argument specified in the catch block. In case the exceptions that may be generated could be of different types, then multiple catch blocks will be required. This can be done by writing these catch blocks one after another. When an exception is thrown, the exception object is compared with the argument in the catch blocks written in succession. The first catch block matching the exception object is executed. We will see use of multiple catch blocks in the next part of the chapter. If the exception object thrown does not match with the argument specified in catch block, then the program gets terminated by automatic invocation of abort() function. Sometimes exceptions are thrown from functions that are invoked within the try block. The point at which this throw is executed is called throw point. After an exception is thrown to the catch block, control cannot return back to the throw point. The Figure 3.2 demonstrates this situation when a function invoked from within the try block throws an exception.

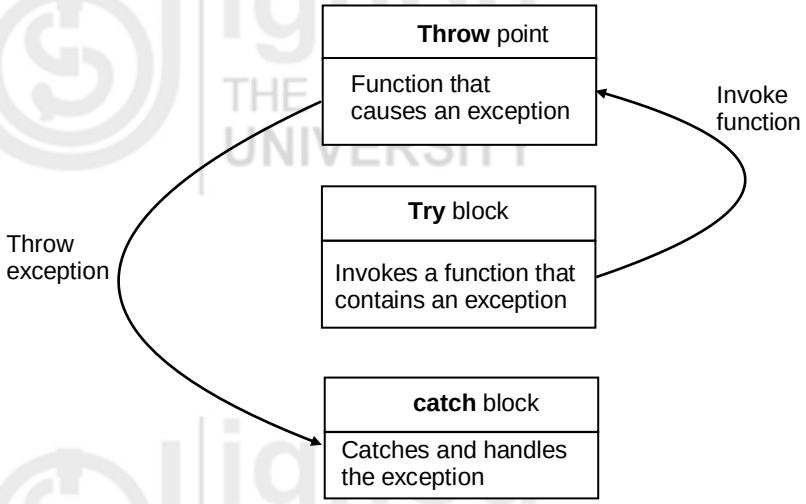


Figure 3.2 Throw, try and catch blocks for exception handling

We present following two code segments that demonstrate use of try, throw and catch expressions for exception handling. The first program presents a simple example of use of a single try and catch block. The second program presents an example where the exception is thrown from a function invoked from within the try block.

```
# include <iostream>
int main()
{
        int x,y;
        cout << "enter values of x & y \n";
        cin >> x;
        cin >> y;
        int a = x-y;
        try
        {
         if (a!=0)
         {
          cout << "Result (x/a) =" << x/a;
         }
         else
         {
          throw (a);
         }
        }
```

```
        catch (int i)
        {
          cout << "Exception caught a=" << a;
        }
        return(0);
        }
```

```
# include <iostream>
void divide (int x, int y, int z)
{
        cout << "inside function \n";
        if (x-y)!=0)
        {
          int r = z/ (x-y);
          cout << "result =" << r << "\n";
        }
        else
        {
          throw (x-y);
        }
 }

int main()
{
        Try
        {
         cout << inside try block \n";
         divide (10,20,30);
         divide(10,10,20);
        }
        catch (int i)
        {
          cout << "Caught the exception";
        }
        return(0);
}
```

You would see that in the first program, if input values are 20 and 15 then the program runs correctly without any exception with result being 4. On the other hand, if input given is 10 and 10, the program would detect an exception and display it as "0" value. Similarly, in the second program for the first invocation of divide (), the result will be -3, whereas the second invocation will result into an exception.

The throw statement can take multiple forms:

```
throw (exception);
throw exception;
throw;
```

The first two throw statements pass the exception object to the catch handler, whereas the third throw statement without any exception object is used to rethrow an exception from within a catch block.

A catch handler may rethrow the exception, without handling it, to the next catch block. This is equivalent to passing the exception to the next enclosing catch block within the scope of try/catch sequence. Please note that a rethrown exception is not

caught by the same catch handler or any other catch handler in that group, but by an appropriate catch handler in the outer try/catch sequence only. This also applies to those exceptions that may be detected within a catch handler block itself.

Multiple Catch Statements

We have seen earlier that a catch block looks like a function definition. It has a type argument which specifies the type of exception that this catch block may handle followed by statements (enclosed within braces) to process the exception. After executing these statements, the control goes to the statement immediately following the catch block. It is also possible that a program may have more than one exception condition. In such cases multiple catch statements are needed to handle the different types of exceptions. Now when an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that matches the thrown exception object type is executed. Rest of the catch blocks are then skipped and the control goes to first statement following the last catch block. This indicates that in case of multiple catch blocks matching the thrown exception, only the first matching catch block is executed. The following program demonstrates the use of multiple catch blocks:

```
#include<iostream>
void test(int x)
{
        try
        {
          if (x==1) throw x;
          else
            if (x==0) throw 'x';
          else
            if(x==-1) throw 1.0;
        }
        catch (char c)
        {
          cout << "caught a character \n";
        }
        catch(int m)
        {
          cout << "caught an integer \n";
        }

        catch(double d)
        {
          cout << "caught a double \n";
        }
}

int main()
{
        cout << "testing multiple catches \n";
        cout << "x==1 \n";
        test(1);
        cout << "x==0 \n";
        test(0);
        cout << "x==-1 \n";
        test(-1);
        cout << "x==2 \n";
        test(2);
```

```
        return(0);
}
```

As you can see this program has multiple catch blocks, each handling exception objects of different types. One integer, other character and the third one a double exception argument. The last invocation of test with argument 2 does not throw any exception and hence no catch block is invoked.

Catch all exceptions

There are many situations where it may be very difficult to anticipate in advance all possible types of exceptions that may occur in a program. C++ provides with a mechanism to cope with this problem in form of a catch (…) expression. This type of catch statement catches all exceptions, unlike only one exception being caught. The general syntax of use of this kind of catch expression is as follows:

```
catch(…)
{
//statements for processing
//all exceptions
}
```

The following example presents a case of use of this kind of catch expression:

```
# include <iostream>
void test (int x)
{
        try
        {
          if (x==0) throw x;
          if (x==-1) throw 'x';
          if (x==1) throw 1.0;
        }
        catch (…)
        {
         cout << "caught an exception \n";
        }
}
int main()
{
        cout << "testing generic catch \n";
        test (-1);
        test (0);
        test(1);
        return(0);
}
```

You may see that this program prints the line "caught an exception" three times as follows:

Output:

caught an exception
caught an exception
caught an exception

This is because all the exceptions (x getting values -1, 0 and 1) are caught by the same catch handler block. This is the reason why this kind of catch block is termed as *catch all expression.* This property can make this kind of catch all expression to be placed as default catch handler. This default handler may then be used to catch all those exceptions which are not handled explicitly. However, one must be careful to place this catch all expression in the last place of catch handlers.

☞ Check Your Progress 1

1) List five common examples of exceptions.

……

……

……

2) If no exceptions are thrown in a try block, where does control proceed to after the try block completes the execution?

……

……

……

3) What happens if an exception is thrown outside a try block?

……

……

……

4) What does the statement throw do?

……

……

……

5) What happens if several handlers match the type of thrown object?

……

……

……

3.4 SPECIFYING EXCEPTION TYPES

We have seen earlier that the exception type is reported by the throw statement to the catch handler, which then takes appropriate action on it. It is also possible to restrict a function to throw only certain specified exceptions. This can be done by adding a throw list clause to the function definition. The general syntax of doing this exception type specification is as follows:

```
type function (arg-list) throw (type-list)
{
……..
……..
}
```

The type-list after throw specifies the type of exceptions that may be thrown. An attempt to throw another type of exception will cause abnormal program termination. In case we want to prevent a function from throwing any exception at all, we may do so by making the type-list empty, i.e., simply writing throw ( ) in the function header line. However, these specifications operate only when the function is called back from a try block and it reports back the exceptions to that try block. It does not apply if exception is thrown within the function code itself. An example to demonstrate this scenario is as follows:

```
#include <iostream>
void test(int x) throw (int, double)
{
        if (x==0) throw 'x';
        else
          if (x==1) throw x;
          else
            if (x==-1) throw 1.0;
}
int main()
{
        try
        {
          cout << :testing throw specifications \n";
          cout << "x==0 \n";
          test(0);
          cout << "x==1 \n";
          test(1);
          cout << "x==-1 \n";
          test(-1);
          cout << "x==2 \n";
          test(2);
        }
        catch(cahr c)
        {
           cout << "caught a character \n";
        }
        catch (int m)
        {
          cout << "caught an integer \n";
        }
        catch (double d)
        {
          cout << caught a double \n";
        }
        return (0);
}
```

You may note that this program tries to throw a char exception object in the very first invocation of test(). This results in an abnormal termination of the program, since the test can throw only exceptions of type int and double.

Program Output:

```
testing throw specifications
x==0
caught a character
```

Throwing an exception that has not been declared in a function's exception specification causes a call to function unexpected. The compiler will not generate a compilation error if a function contains a throw expression for an exception not listed in the function's specification. An error occurs only when that function attempts to throw that exception at execution time. To avoid surprises at execution time, its better to carefully check the code to ensure that functions do not throw exceptions not listed in their specifications.

The *function unexpected* calls, the function registered with the function set_unexpected (defined in header file <exception>). If no function has been registered in this manner, function terminate is called by default. The function set_terminate can specify the function to invoke when terminate is called. Otherwise, terminate calls abort, which terminates the program without calling the destructors of any remaining objects of automatic or static storage class. This could lead to resource leaks when a program terminates prematurely.

Exceptions and Stack Unwinding

When an exception is thrown but not caught in a particular scope, the function call stack is "unwound" and an attempt is made to catch the exception in the next outer try….catch block. Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function. If a try block encloses that statement, an attempt is made to catch the exception. If a try block does not enclose the statement, stack unwinding occurs again. If no catch handler ever catches this exception, function terminate is called to terminate the program. The following programming example demonstrates stack unwinding:

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

//function3 throws runtime error
void function3() throw (runtime_error)
{
  cout << "in function3" << endl;
// no try block, stack unwinding occurs, return control to function2
  throw runtime_error ("runtime_error in function3");
}

//function2 invokes function3
void function2() throw (runtime_error)
{
  cout << "function3 is called inside function2" << endl;
  function3(); // stack unwinding occurs, return control to function1
}

//function1 invokes function2
void function1() throw (runtime_error)
{
  cout << "function2 is called inside function1" << endl;
  function2(); //stack unwinding occurs, return control to main
}

int main()
{
  //invoke function1
```

```
  try
   {
     cout << "function1 is called inside main" << endl;
     function1();
    }
  catch (runtime_error & error)
   {
     cout << "exception occurred:"  << error.what() << endl;
     cout << " exception handled in main" << endl;
   }
}
```

The program will produce the following output:

function1 is called inside main
function2 is called inside function1
function3 is called inside function2
in function3
exception occurred: runtime_error in function3
exception handled in main

Exception handling and Constructors and Destructors

It is worth discussing that what happens if an exception is thrown while executing a constructor? Since the object is not yet fully constructed, its destructor would not be called once the program control goes out of the object's context. And, if the constructor had reserved some memory before the exception was raised, then there would be no mechanism to prevent such memory leak. Hence. Appropriate exception handling mechanism must be implemented pertaining to the constructor routine to handle exceptions that occur during object construction.
Where to catch the exception is another important issue here. Whether it should be done inside the constructor block or inside the main. If we allow the exception to be handled inside main then we would not be able to prevent the memory leak situation. Therefore, we must catch the exception within the constructor block so that we get chance to free up any reserved memory spaces. However, we must simultaneously rethhrow the exception to be appropriately handled inside the main block.

Exceptions and Inheritance

Like the normal inheritance concept, various exception classes can be derived from a common base class. If a ctach handler catches a pointer or reference to an exception objectof a base-class type, it also can catch a pointer or reference to all objects of class publicly derived from that base class- this allows for polymorphic processing of related errors. Using inheritance with exceptions enables an exception handler to catch related errors with concise notation. One approach is to catch each type of pointer or reference to a derived-class exception object individually, but a more concise approach is to catch pointers or references to base-class exception objects nstead. Also, catching pointers or references to derived-class exception object individually is error prone, especially if you forget to test explicitly for one or more of the derived-class pointer or reference types.

Standard Library Exception Hierarchy

As we know that exceptions fall nicely into a number of categories. The C++ standard library includes a hierarchy of exception classes. The hierarchy is headed by base-class exception (defined in header file <exception>), which contains virtual function what, which derived base classes can override to issue appropriate error messages. Immediate derived classes of base class exception include runtime_error and logic_error (both defined in header <stdexcept>), each of which has several derived classes. Also derived from exception are the exceptions thrown by C++ operators – for example, bad_alloc is thrown by new, bad_cast is thrown by dynamic_cast and

bad_typeid is thrown by typeid. Including bad_exception in the throw list of a function means that, if an unexpected exception occurs, function unexpected can throw bad_exception rather than terminating the program's execution or calling another function specified by set_unexpected. The class logic_error is the base class of several standard exception classes that indicate errors in program logic, whereas class runtime_error is the base class of several other standard exception classes that indicate execution-time errors. The Figure 3.3 below presents an overview of standard library exception classes.

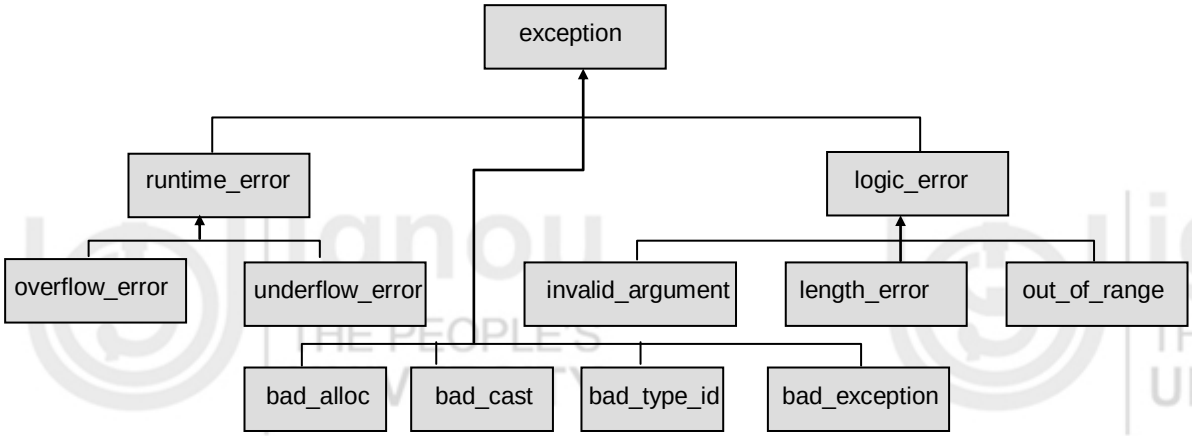


Figure 3.3 Some of the Standard Library Exception Classes

☞ Check Your Progress 2

1) What happens if no catch handler matches the type of a thrown object?

……

……

……

2) Must throwing an exception cause program termination?

……

……

……

3) What happens when a catch handler throws an exception?

……

……

……

4) How do you restrict the exception type that a function can throw?

……

……

……

5) What happens if a function throws an exception of a type not allowed by the exception specification for the function?

……

……

……

3.5 SUMMARY

Exceptions are a kind of problem that may occur when a program is executed. C++ provides an exception handling mechanism to handle synchronous exceptions. An exception is handled by a group of try, throw and catch expressions. The block that may cause a possible exception is enclosed within a try block. The exceptional situation occurring in the try block is reported by throw expression to an exception handling block, called catch block. When an exception is not caught the program is terminated. The throw expression passes the exception object to an appropriate catch block. A program may have more than one catch block to handle different kinds of exceptions. We can also have a catch all expression that can be used as a default handler. A catch block may also rethrow an exception without processing it. We may also restrict the type of exception objects that a function may throw. The try-throw-catch mechanism in C++ is quite useful to provide for dealing with unexpected situations that may occur at runtime in a program.

3.6 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) Insufficient memory to satisfy a new request, array subscript out of bounds, arithmetic overflow, division by zero, invalid function parameters.
2) The exception handlers (in the catch handlers) for that try block are skipped, and the program resumes execution after the last catch handler.
3) An exception thrown outside a try block causes a call to terminate.
4) It passes the exception object to the catch handler. If it occurs within a catch block, it rethrows the exception.
5) The first matching exception handler after the try block is executed.

Check Your Progress 2

1) This causes the search for a match to continue in the next enclosing try block if there is one. As this process continues, it might eventually be determined that there is no handler in the program that matches the type of thrown object. In this case, program is aborted.
2) No, but it does terminate the block in which the exception is thrown.
3) The exception will be processed by a catch handler (if one exists) associated with the try block (if one exists) enclosing the catch handler that caused the exception.
4) Provide an exception specification listing the exception type that the function can throw.
5) Function unexpected is called to terminate the program.

3.7 FURTHER READINGS

1) E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2) P. Deitel and H. Deitel, C++: *How to Program, PHI*, 7th edition, 2010.
3) B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.

UNIT 4 A CASE STUDY

Structure **Page Nos.**

4.0 INTRODUCTION

In the units covered so far, we have discussed various features of C++ language that help in designing and implementing useful programs to solve various real world problems. The bottom up approach adopted by C++ language provides a better way to capture the details of real world problems and design efficient and adaptable solutions. Whether it is the mechanism of constructors and destructors, inheritance, or polymorphism; all taken collectively provide suitable design methodology and tool for solving various real world problems through C++ programming formulations. In order to solve a real world problem, the first and foremost requirement is to identify the various objects in the system along with their general attributes. This is then followed by the more involved process of identifying the methods/ functions that can be applied on the different objects. Once this is done, the effort gets more focused towards design, which involves designing various classes and implementing different functions.

This unit presents a case study of designing and implementing a transaction-processing system for banking domain. Unlike a database design for bank accounts, we have adopted a file processing approach to emphasize the C++ features and capabilities. The design involves creating necessary data files to store accounts and customer information and then accessing them through suitable code for writing data, appending data, performing credit operations and displaying the results. Our focus in the case study is on demonstrating the design methodology and the steps required to design a small real world application. The steps involved in design and implementation of the transaction-processing system are described with relevant explanations at various places. The program design also makes use of certain features of C++ which are used for larger programs. After a careful study of the unit, you would be able to clearly identify the broad guidelines and general steps involved in solving a real world problem and using C++ for solving variety of problems.

4.1 OBJECTIVES

At the end of the unit, you should be able to:

- explain the steps involved in solving real world problems;
- describe the overall framework of such designs;
- appreciate the usefulness of various features of C++ for solving different real problems;
- design C++ programming solutions for many other problems; and
- design C++ programs involving multiple files.

4.2 DESIGNING A TRANSACTION-PROCESSING SYSTEM

A transaction processing system is one where certain activities are carried out as part of some bigger goal. The kind of transactions performed in a system depends on the domain of the problem. For example, in a ticket reservation domain the key transactions may be booking a ticket, realizing payment for a ticket, cancelling a ticket, modifying or amending a ticket, refund of a cancelled ticket etc. Similarly in a banking domain, the transactions could be opening a customer account, updating the account records, processing withdrawal from an account, deposit into an account, printing summary of accounts etc. Every transaction processing system, irrespective of the domain, has to complete certain activities (referred to as transactions). Nowadays when a large number of transaction processing activities are automated to be performed on a computer system, it is very important to know and understand how can we design such a system. Moreover, with the increased use of Internet and web-based services many of these transactions are initiated and realized at different physical machines. The data related to transactions is sent over communication lines. One common thing however in all kind of transaction processing systems is the need to store and manipulate associated data. The data of the system may be stored either in a database format or in terms of a collection of various files. The general practice in most of the real world transaction processing systems is to go for a database design approach. However, in the example described below we have used a file processing approach.

Most of the transaction processing systems are quite big and result into large programs. The large programs often comprise of various modules. In order to have better understanding and design convenience, these transaction processing systems are designed as a collection of multiple programs. In the previous units we have largely seen example programs consisting of a single program file. Whenever we have to design programming solution for a larger problem, it is often required to organize the large code into multiple files. Using a multiple file organization not only helps in clarity of the design but also in appropriate use of class libraries and better coordination of programmers working on the large project. In fact, large programs are usually divided into separate files, where different files have code for different functionalities, such as one file for mathematical analysis, another for graphics display and a separate one for I/O etc. Large applications sometimes also involve multiple designers who coordinate their effort to design the final solution. Most of the C++ IDEs provide a feature called PROJECT which helps in designing and organizing larger programs comprising multiple files.

We will now see how we can use the various features and capabilities of C++ that we have learned so far to design a transaction-processing system. The proposed system involves some fixed-length account records for a company having certain number of customers. Each record consists of an account number that acts as the record key, a last name, a first name and a balance. The transaction-processing program is to be designed in such a manner that it can provide overall management functions for the accounts and transactions. It should be able to perform functions like update an account, insert a new account, delete an account and insert all the account records into a formatted text file for printing.

The key components in this application design can be understood through following abstract diagram representation.

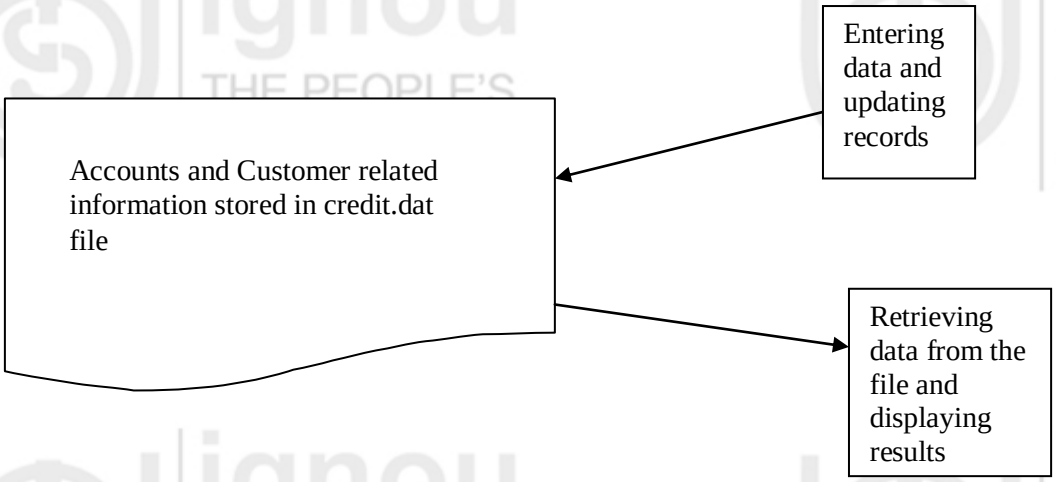


Figure 4.1 : Abstract Diagram of transaction processing system

As we can see, the entire data corresponding to accounts and customer information is stored in credit.dat file. We will design program to enter, add and update the data in this file. The data entered may also be retrieved and displayed as a formatted text output through the use of various stream manipulators. We first create a ClientData class header file that defines the format of the data and then define the constructor and certain basic methods. The following two program segments (Program 4.1 and 4.2) are written for this purpose.

```
 1  #ifndef CLIENTDATA_H
 2  #define CLIENTDATA_H
 3  #include <string>
 4  using namespace std;
 5  class ClientData
 6  {
 7  public:
 8  ClientData(int =0, string = " ", string = " ", double =
0.0);
 9
10  void setAccountNumber (int);
11  int getAccountNumber() const;
12  void setLastname(string);
13  string getLastName() const;
14  void setFirstName(string);
15  string getFirstName() const;
16  void setBalance(double);
17  double getBalance() const;
18
19  private:
20  int accountNumber;
21  char lastName[15];
22  char firstName[10];
23  double balance;
24  };
25  #endif
```

Program 4.1: ClientData class header file

The program 4.1 above defines client data header file which specifies the data format to be used in the application. The main data items are account number (an integer

value), last name of customer (a string), first name of customer (a string) and balance (a float value denoting the account balance). These data items are declared as private. The methods to access and modify this data are setAccountNumber(), getAccountNumber(), setLastName(), getLastName(), setFirstName(), getFirstName(), setBalance() and getBalance(). All these functions are used to define and retrieve values for various fields, and are defined in next program (Program 4.2). The client data represent a customer's credit information and the methods described in program 4.2 provide the code for manipulating the data. The exact code is described in in the program 4.2 given below:

```
 1  #include <string>
 2  #include "ClientData.h"
 3  using namespace std;
 4
 5  // default ClientData constructor
 6  ClientData::ClientData(int accountNumberValue, string
lastNameValue,
 7     string firstNameValue, double balanceValue)
 8 {
 9  setAccountNumber(accountNumberValue);
10  setLastName(lastNameValue);
11  setFirstName(firstNameValue);
12  setBalance(balanceValue);
13 } // end ClientData constructor
14  //get account number value
15  int ClientData::getAccountNumber() const
16 {
17  return accountNumber;
18 }
19  //set account number value
20  void ClientData::setAccountNumber(int accountNumberValue)
21 {
22  accountNumber=accountNumberValue;
23 }
24 // get last name value
25  string ClientData::getLastName() const
26  {
27   return lastName;
28  }
29  // set last name value
30  void ClientData::setLastName(string lastnameString)
31  {
32   int length=lastNameString.size();
33   length = (length<15? length:14); \\for copying at most 15
chars
34   lastNameString.copy(lastName, length);
35   lastName[length]='\0'; \\appending null character
36  }
37
38   //get first-name value
39   string ClientData::getfirstName() const
40   {
41    return firstName;
42   }
43
44   // set first-name value
45   void ClientData::setfirstName(string firstNameString)
46   {
47    // copy at most 10 chars
48    int length =firstNameString.size();
49    length = (length < 10? length:9);
```

```
50    firstNameString.copy(firstName, length);
51    firstName[length]='\0'; \\ appending null char
52   }
53
54    // get balance value
55    double ClientData::getBalance() const
56    {
57     return balance;
58    }
59
60    //set balance value
61    void ClientData::setBalance(double lalanceValue)
62    {
63     balance=balanceValue;
64    }
65
```

Program 4.2: ClientData class representing customer credit information

As you may easily notice, the program defines the ClientData constructor comprising of various functions, each of which have a specified code. The functions setLastName() and setFirstName() limit the number of characters read from input that are finally written to actual data.

We now look at the code (Program 4.3) for creating a file credit.dat with some data entries to be used in our transaction processing system. The program creates a binary file credit.dat for output. It then writes 100 blank records into the credit.dat data file. The next program (program 4.4) then uses various functions to actually write data into this file.

```
 1  // creating randomly accessible file credit.dat
 2  #include <iostream>
 3  #include <fstream>
 4  #include <cstdlib>
 5  #include "ClientData.h"
 6  using namespace std;
 7
 8  int main()
 9 {
10  ofstream outCredit ("credit.dat", ios::out | ios::binary);
11  if (!outCredit)
12  {
13    cerr << "File could not be opened." << endl;
14    exit(1);
15  }
16
17  ClientData blankClient; // constructor zeros out each data
member
18  // output 100 blank records to file
19  for (int i=0; i<100, i++)
20    outCredit.write(reinterpret_cast < const char * >
(&blankClient),
21     sizeof(ClientData));
22  }
```

Program 4.3: Creating the credit.dat file with 100 blank records

Once the file credit.dat is created we can use this file to store the desired data corresponding to the accounts and customers. After entering the basic data, actual

transaction-processing may be performed. The program below reads the data from user entered values through keyboard and then uses fstream functions to store data at desired locations in the file credit.dat. Note that the file is opened in out mode for writing. An example run of the program 4.4 is presented after the program code. The run shows how different data values can be entered into the data file. Note that line 19 includes the header file ClientData.h defined in Program 4.1 so the program can use ClientData objects.

```
 1
 2  // Writing to a random-access file.
 3  #include <iostream>
 4  using std::cerr;
 5  using std::cin;
 6  using std::cout;
 7  using std::endl;
 8  using std::ios;
 9
10  #include <iomanip>
11  using std::setw;
12
13  #include <fstream>
14  using std::fstream;
15
16  #include <cstdlib>
17  using std::exit; // exit function prototype
18
19  #include "ClientData.h" // ClientData class definition
20
21  int main()
22  {
23     int accountNumber;
24     char lastName[ 15 ];
25     char firstName[ 10 ];
26     double balance;
27
28     fstream outCredit( "credit.dat", ios::in | ios::out |
ios::binary );
29
30     // exit program if fstream cannot open file
31     if ( !outCredit )
32     {
33        cerr << "File could not be opened." << endl;
34        exit( 1 );
35     } // end if
36
37     cout << "Enter account number (1 to 100, 0 to end
input)\n? ";
38
39     // require user to specify account number
40     ClientData client;
41     cin >> accountNumber;
42
43     // user enters information, which is copied into file
44     while ( accountNumber > 0 && accountNumber <= 100 )
45     {
46        // user enters last name, first name and balance
47        cout << "Enter lastname, firstname, balance\n? ";
48        cin >> setw( 15 ) >> lastName;
49        cin >> setw( 10 ) >> firstName;
50        cin >> balance;
51
```

```
52         // set record accountNumber, lastName, firstName and
balance values
53         client.setAccountNumber( accountNumber );
54         client.setLastName( lastName );
55         client.setFirstName( firstName );
56         client.setBalance( balance );
57
58         // seek position in file of user-specified record
59         outCredit.seekp( ( client.getAccountNumber() - 1 ) *
60            sizeof( ClientData ) );
61
62         // write user-specified information in file
63         outCredit.write( reinterpret_cast< const char * >(
&client ),
64            sizeof( ClientData ) );
65
66         // enable user to enter another account
67         cout << "Enter account number\n? ";
68         cin >> accountNumber;
69      } // end while
70
71      return 0;
72   } // end main
```

Program 4.4: Writing data to credit.dat file

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Singh Shweta 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Tiwari Nisha -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Jolly Stellina 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Sen Ajay 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Ghosh Soumitra 314.33
Enter account number
? 0
```

The transaction processing system that we are designing can now be visualized in figure 4.2:

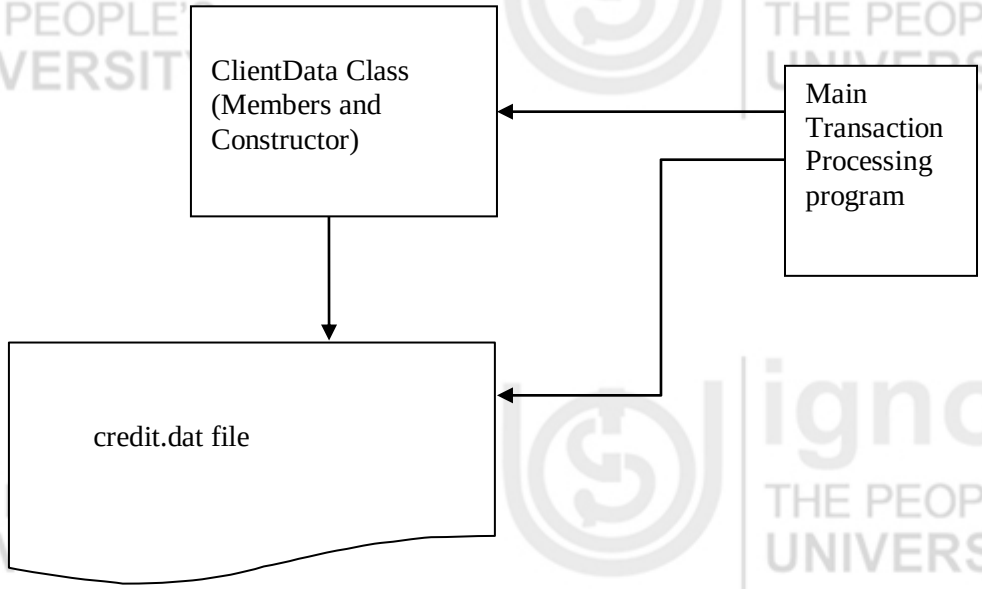


Figure 4.2: Transaction Processing System Program Structure

We now present our main transaction-processing program (Program 4.5) which uses the ClientData.h and credit.dat files to achieve "instant" -access processing. As we discussed earlier, the program manages a bank's account information. The program can perform all functions of accounts processing. It can update existing accounts, adds new accounts, deletes accounts and stores a formatted listing of all current accounts in a text file. We assume that the program 4.3 has been executed to create the file credit.dat and that the program of Program 4.4 has been executed to insert the initial data, before this program can be used for transaction-processing operations.

```
 1
 2   // This program reads a random-access file sequentially,
updates
 3   // data previously written to the file, creates data to be
placed
 4   // in the file, and deletes data previously stored in the
file.
 5   #include <iostream>
 6   using std::cerr;
 7   using std::cin;
 8   using std::cout;
 9   using std::endl;
10   using std::fixed;
11   using std::ios;
12   using std::left;
13   using std::right;
14   using std::showpoint;
15
16   #include <fstream>
17   using std::ofstream;
18   using std::ostream;
19   using std::fstream;
20
21   #include <iomanip>
22   using std::setw;
```

```
23   using std::setprecision;
24
25   #include <cstdlib>
26   using std::exit; // exit function prototype
27
28   #include "ClientData.h" // ClientData class definition
29
30   int enterChoice();
31   void createTextFile( fstream& );
32   void updateRecord( fstream& );
33   void newRecord( fstream& );
34   void deleteRecord( fstream& );
35   void outputLine( ostream&, const ClientData & );
36   int getAccount( const char * const );
37
38   enum Choices { PRINT = 1, UPDATE, NEW, DELETE, END };
39
40   int main()
41   {
42      // open file for reading and writing
43      fstream inOutCredit( "credit.dat", ios::in | ios::out |
ios::binary );
44
45      // exit program if fstream cannot open file
46      if ( !inOutCredit )
47      {
48         cerr << "File could not be opened." << endl;
49         exit ( 1 );
50      } // end if
51
52      int choice; // store user choice
53
54      // enable user to specify action
55      while ( ( choice = enterChoice() ) != END )
56      {
57         switch ( choice )
58         {
59            case PRINT: // create text file from record file
60               createTextFile( inOutCredit );
61               break;
62            case UPDATE: // update record
63               updateRecord( inOutCredit );
64               break;
65            case NEW: // create record
66               newRecord( inOutCredit );
67               break;
68            case DELETE: // delete existing record
69               deleteRecord( inOutCredit );
70               break;
71            default: // display error if user does not select
valid choice
72               cerr << "Incorrect choice" << endl;
73               break;
74         } // end switch
75
76         inOutCredit.clear(); // reset end-of-file indicator
77      } // end while
78
79      return 0;
80   } // end main
81
82   // enable user to input menu choice
83   int enterChoice()
```

```
84    {
85       // display available options
86       cout << "\nEnter your choice" << endl
87          << "1 - store a formatted text file of accounts" <<
endl
88          << " called \"print.txt\" for printing" << endl
89          << "2 - update an account" << endl
90          << "3 - add a new account" << endl
91          << "4 - delete an account" << endl
92          << "5 - end program\n? ";
93
94       int menuChoice;
95       cin >> menuChoice; // input menu selection from user
96       return menuChoice;
97    } // end function enterChoice
98
99    // create formatted text file for printing
100   void createTextFile( fstream &readFromFile )
101   {
102      // create text file
103      ofstream outPrintFile( "print.txt", ios::out );
104
105      // exit program if ofstream cannot create file
106      if ( !outPrintFile )
107      {
108         cerr << "File could not be created." << endl;
109         exit( 1 );
110      } // end if
111
112      outPrintFile << left << setw( 10 ) << "Account" <<
setw( 16 )
113         << "Last Name" << setw( 11 ) << "First Name" <<
right
114         << setw( 10 ) << "Balance" << endl;
115
116      // set file-position pointer to beginning of
readFromFile
117      readFromFile.seekg( 0 );
118
119      // read first record from record file
120      ClientData client;
121      readFromFile.read( reinterpret_cast< char * >( &client
),
122         sizeof( ClientData ) );
123
124      // copy all records from record file into text file
125      while ( !readFromFile.eof() )
126      {
127         // write single record to text file
128 if ( client.getAccountNumber() != 0 ) // skip empty records
129            outputLine( outPrintFile, client );
130
131         // read next record from record file
132    readFromFile.read( reinterpret_cast< char * >( &client ),
133            sizeof( ClientData ) );
134      } // end while
135   } // end function createTextFile
136
137   // update balance in record
138   void updateRecord( fstream &updateFile )
139   {
140      // obtain number of account to update
141      int accountNumber = getAccount( "Enter account to
```

```
update" );
142
143     // move file-position pointer to correct record in file
144     updateFile.seekg( ( accountNumber - 1 ) * sizeof(
ClientData ) );
145
146     // read first record from file
147     ClientData client;
148     updateFile.read( reinterpret_cast< char * >( &client ),
149        sizeof( ClientData ) );
150
151     // update record
152     if ( client.getAccountNumber() != 0 )
153     {
154        outputLine( cout, client ); // display the record
155
156        // request user to specify transaction
157        cout << "\nEnter charge (+) or payment (-): ";
158        double transaction; // charge or payment
159        cin >> transaction;
160
161        // update record balance
162        double oldBalance = client.getBalance();
163        client.setBalance( oldBalance + transaction );
164        outputLine( cout, client ); // display the record
165
166        // move file-position pointer to correct record in
file
167        updateFile.seekp( ( accountNumber - 1 ) * sizeof(
ClientData ) );
168
169        // write updated record over old record in file
170        updateFile.write( reinterpret_cast< const char * >(
&client ),
171           sizeof( ClientData ) );
172     } // end if
173     else // display error if account does not exist
174        cerr << "Account #" << accountNumber
175           << " has no information." << endl;
176  } // end function updateRecord
177
178  // create and insert record
179  void newRecord( fstream &insertInFile )
180  {
181     // obtain number of account to create
182     int accountNumber = getAccount( "Enter new account
number" );
183
184     // move file-position pointer to correct record in file
185     insertInFile.seekg( ( accountNumber - 1 ) * sizeof(
ClientData ) );
186
187     // read record from file
188     ClientData client;
189     insertInFile.read( reinterpret_cast< char * >( &client
),
190        sizeof( ClientData ) );
191
192     // create record, if record does not previously exist
193     if ( client.getAccountNumber() == 0 )
194     {
195        char lastName[ 15 ];
196        char firstName[ 10 ];
```

```
197         double balance;
198
199         // user enters last name, first name and balance
200         cout << "Enter lastname, firstname, balance\n? ";
201         cin >> setw( 15 ) >> lastName;
202         cin >> setw( 10 ) >> firstName;
203         cin >> balance;
204
205         // use values to populate account values
206         client.setLastName( lastName );
207         client.setFirstName( firstName );
208         client.setBalance( balance );
209         client.setAccountNumber( accountNumber );
210
211         // move file-position pointer to correct record in
file
212         insertInFile.seekp( ( accountNumber - 1 ) * sizeof(
ClientData ) );
213
214         // insert record in file
215         insertInFile.write( reinterpret_cast< const char *
>( &client ),
216            sizeof( ClientData ) );
217      } // end if
218      else // display error if account already exists
219         cerr << "Account #" << accountNumber
220            << " already contains information." << endl;
221   } // end function newRecord
222
223   // delete an existing record
224   void deleteRecord( fstream &deleteFromFile )
225   {
226      // obtain number of account to delete
227      int accountNumber = getAccount( "Enter account to
delete" );
228
229      // move file-position pointer to correct record in file
230      deleteFromFile.seekg( ( accountNumber - 1 ) * sizeof(
ClientData ) );
231
232      // read record from file
233      ClientData client;
234      deleteFromFile.read( reinterpret_cast< char * >(
&client ),
235         sizeof( ClientData ) );
236
237      // delete record, if record exists in file
238      if ( client.getAccountNumber() != 0 )
239      {
240         ClientData blankClient; // create blank record
241
242         // move file-position pointer to correct record in
file
243         deleteFromFile.seekp( ( accountNumber - 1 ) *
244            sizeof( ClientData ) );
245
246         // replace existing record with blank record
247         deleteFromFile.write(
248            reinterpret_cast< const char * >( &blankClient ),
249            sizeof( ClientData ) );
250
251         cout << "Account #" << accountNumber << "
deleted.\n";
```

```
252     } // end if
253     else // display error if record does not exist
254        cerr << "Account #" << accountNumber << " is
empty.\n";
255  } // end deleteRecord
256
257  // display single record
258  void outputLine( ostream &output, const ClientData &record
)
259  {
260     output << left << setw( 10 ) <<
record.getAccountNumber()
261        << setw( 16 ) << record.getLastName()
262        << setw( 11 ) << record.getFirstName()
263        << setw( 10 ) << setprecision( 2 ) << right << fixed
264        << showpoint << record.getBalance() << endl;
265  } // end function outputLine
266
267  // obtain account-number value from user
268  int getAccount( const char * const prompt )
269  {
270     int accountNumber;
271
272     // obtain account-number value
273     do
274     {
275        cout << prompt << " (1 - 100): ";
276        cin >> accountNumber;
277     } while ( accountNumber < 1 || accountNumber > 100 );
278
279     return accountNumber;
280  } // end function getAccount
```

Program 4.5: Main transaction-processing program

The program presents a menu driven interface to the user. The choices available to the user are 1-Print, 2-Update, 3-New account, 4- Delete account and 5-End processing. These menu choices are realized through following five options:

Option1: calls function createtextFile to store a formatted list of all account information in a text file called print.txt that may be printed. The function createTextFile takes an fstream object as an argument to be used to input data from the credit.dat file. It invokes istream member function read and uses sequential access to input data from credit.dat. The function outputLine is used to output the data to file print.txt. Note that the createTextFile uses istream member function seekg to ensure that the file-position pointer is at the beginning of the file.

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

1

Account     Last Name      First Name    Balance
29          Tiwari         Nisha         -24.54
33          Ghosh          Soumitra      314.33
37          Singh          Shweta        0.0
88          Sen            Ajay          258.34
96          Jolly          Stellina      34.98
```

Option2 calls updateRecord to update an account. This function updates only an existing record, so the function first determines whether the specified record is empty. Lines 128-129 read data into object client, using istream member function read. Then line 132 compares the value returned by getAccountNumber of the client object to zero to determine whether the record contains information. If this value is zero, lines 154-155 print an error message indicating that the record is empty. If the record contains information, line 134 displays the record, using function outputLine, line 139 inputs the transaction amount and lines 142-151 calculate the new balance and rewrite the record to the file. A typical output for option 2 is:

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

2

Enter account to update (1 - 100) : 37
37          Singh          Shweta        0.0
Enter charge (+) or payment (-): +87.9988
37          Singh          Shweta        87.99
```

Option3 calls function newrecord (lines 159-201) to add a new account to the file. If the user enters an account number for an existing account, newRecord displays an error message indicating that the account exists (lines 199-200). A typical output for option 3 is:

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

3


Enter new account number (1 - 100) : 22

Enter lastname, firstname, balance

?          Popli        Sukanya      247.45
```

Option4 calls function deleteRecord (lines 204-235) to delete a record from the file. Line 207 prompts the user to enter the account number. Only an existing record may be deleted, so if the specified account is empty, line 234 displays an error message. If the account exists, lines 227-229 reinitialize that account by copying an empty record (blank-Client) to the file. Line 231 displays a message to inform the user that the record has been deleted. A typical output for option 4 is:

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

4

Enter account to delete (1 - 100) : 29

Account #29 deleted.
```

Option5 terminating the program.

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

5
```

This transaction-processing system presents an example of a large multi-file program. The program demonstrates use of various features of C++ ranging from classes and methods, constructors, polymorphism, templates and stream I/O capabilities. The program illustrates how C++ language features can be used to solve real world applications by designing and deploying C++ programs.

It would also be in order to discuss here the fact that the sequential files are inappropriate for instant-access applications, in which a particular record must be located immediately. Common instant-access applications are airline reservation systems, banking systems, point-of-sale systems, automated teller machines and other kinds of transaction-processing systems that require rapid access to specific data. A bank might have hundreds of thousands (or even millions) of other customers, yet, when a customer uses an automated teller machine, the program checks that customer's account in a few seconds or less for sufficient funds. This kind of instant access is made possible with random-access files. Individual records of a random-access file can be accessed directly (and quickly) without having to search other records. As we have said, C++ does not impose structure on a file. So the application that wants to use random-access files must create them. A variety of techniques can be used. Perhaps the easiest method is to require that all records in a file be of the same fixed length. Using same-size, fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record relative to the beginning of the file. Using a database based approach is a common choice for many of these applications.

C++ provides rich set of features for designing various applications to solve various real world problems. A number of useful applications in different domain cab be designed using C++. Applications like passenger reservation systems, automated plant control, restaurant management, library management are some possible applications which can be implemented using C++ features.

4.5 SUMMARY

In the previous chapters, we have discussed various features of C++. C++ provides a rich set of features and capabilities that can be used to write useful programs to solve a number of real world problems. This unit presents a case study of designing and implementing a transaction- processing system in banking domain. The design involves creating necessary data files to store accounts and customer information and then accessing them through suitable code for writing data, appending data, performing credit operations and displaying the results. The program design makes use of various features of C++, including its capability to design multi-file programs. The file processing capability of C++ coupled with the rich I/O capability through stream classes can be used to design many interesting applications. This unit demonstrated use and application of various C++ features for solving one real world large scale problem. C++ can be used to solve many other simple and sophisticated real world problems.

4.6 FURTHER READINGS

1. E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2. P. Deitel and H. Deitel, *C++: How to Program*, PHI, 7th ed, 2010.
3. B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.
4. R. Lafore, *Object Oriented Programming in TURBO C++*, Galgotia Publications, 1994.