UNIT 1 OBJECT ORIENTED PROGRAMMING

Structure **Page Nos.**

1.0	Introduction	5
1.1	Objectives	5
1.2	Program and Programming	6
1.3	Programming Languages	8
1.4	Structured Programming Paradigm	9
1.5	Object-Oriented Programming Paradigm	11
1.6	Structured Vs. Object-Oriented Programming	12
1.7	Object-Oriented Programming Concepts	13
1.8	Benefits of OOPs	20
1.9	Summary	21
1.10	Answers to Check Your Progress	22
1.11	Further Readings	23

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.

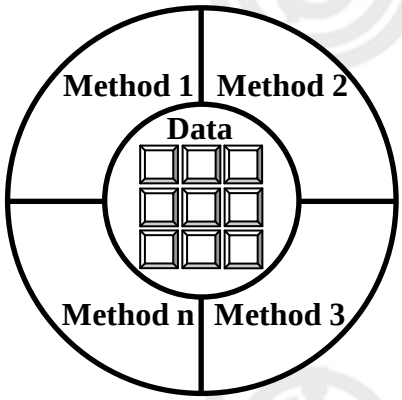


Figure 1.1: Encapsulation & Information Hiding Concept

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.

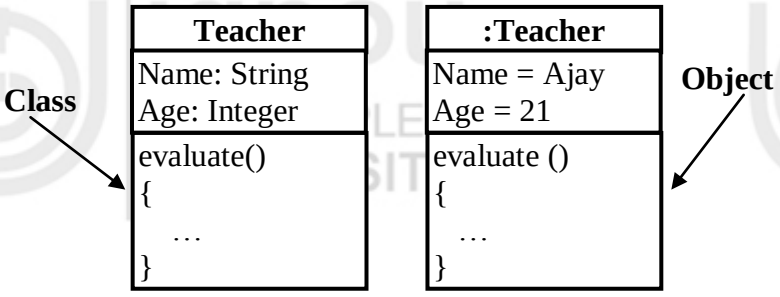


Figure 1.2: Concept of a Class and a Corresponding Object

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'.

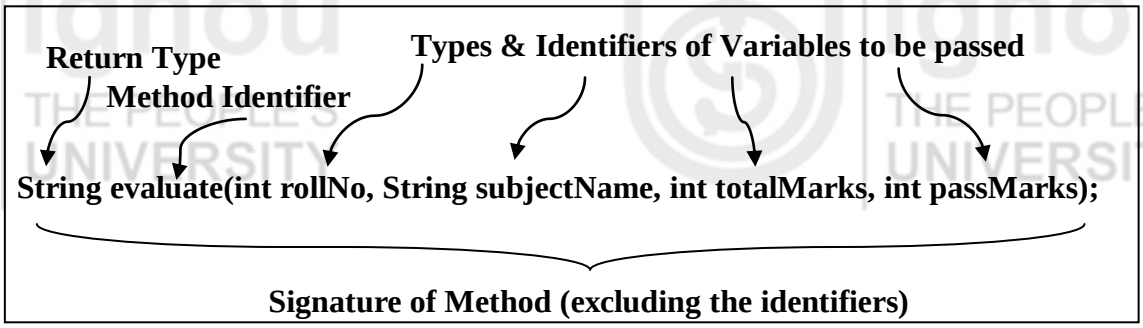


Figure 1.3: Signature of a Method

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.

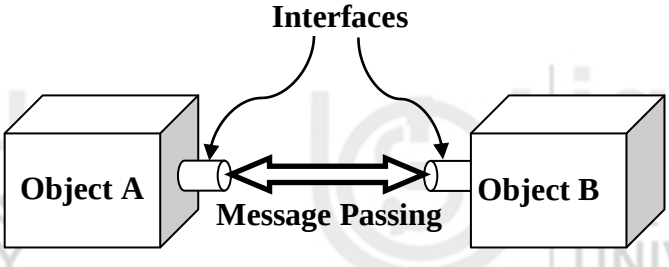


Figure 1.4: Objects Interacting through Interfaces

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.

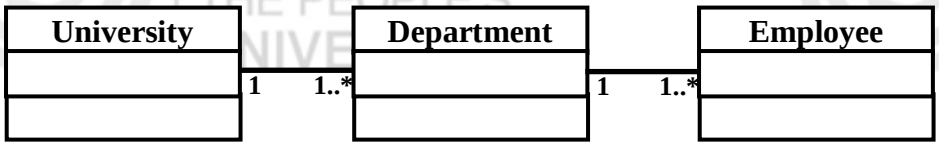


Figure 1.5: Association & Multiplicity among Classes

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.

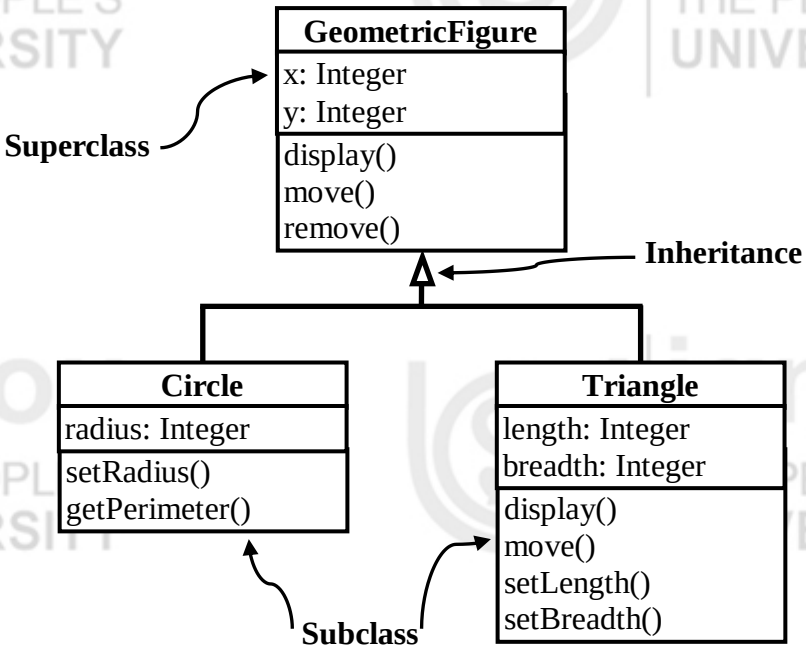


Figure 1.6: Inheritance among Classes

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.

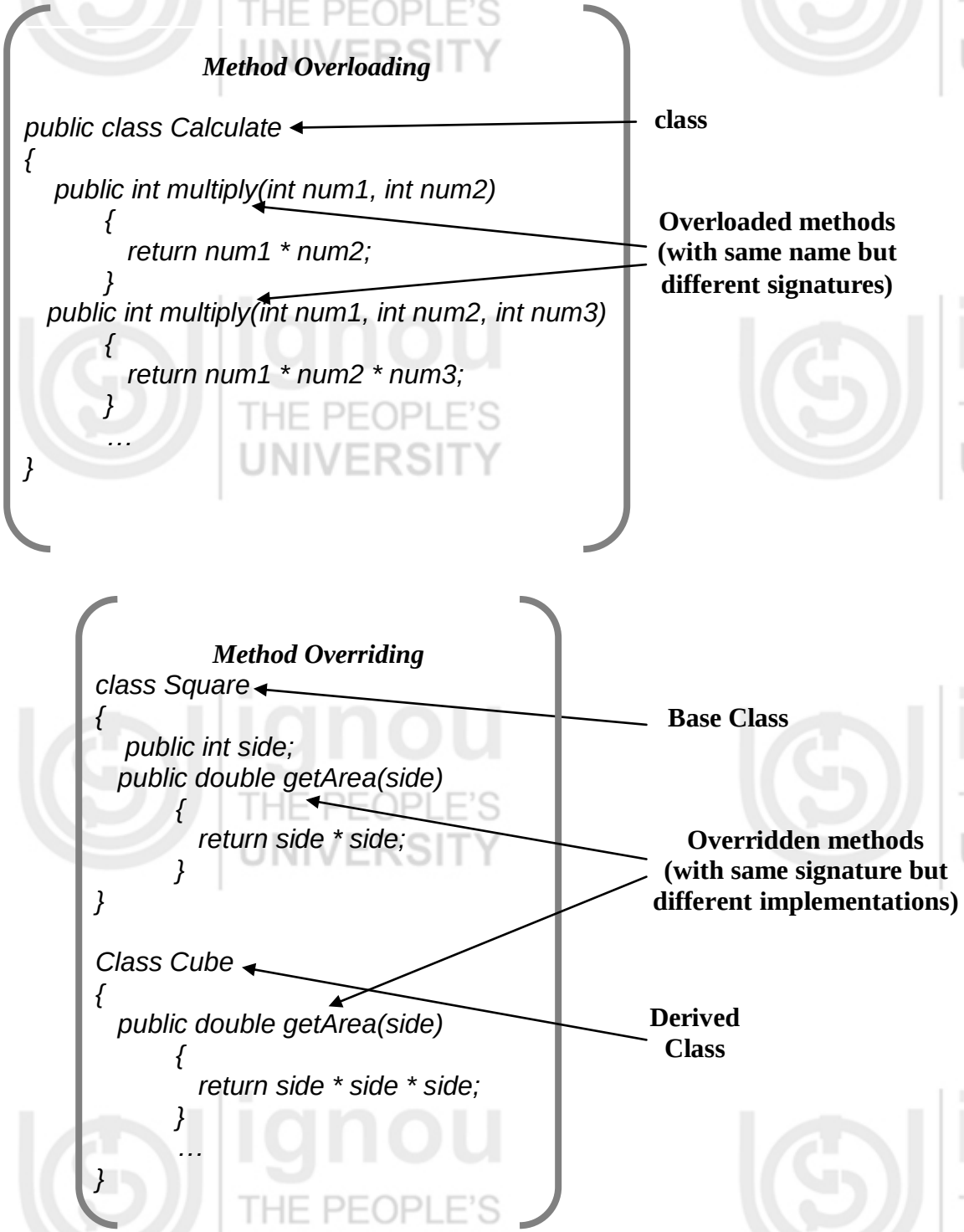


Figure 1.7: Two Different Types of Polymorphism

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+ +

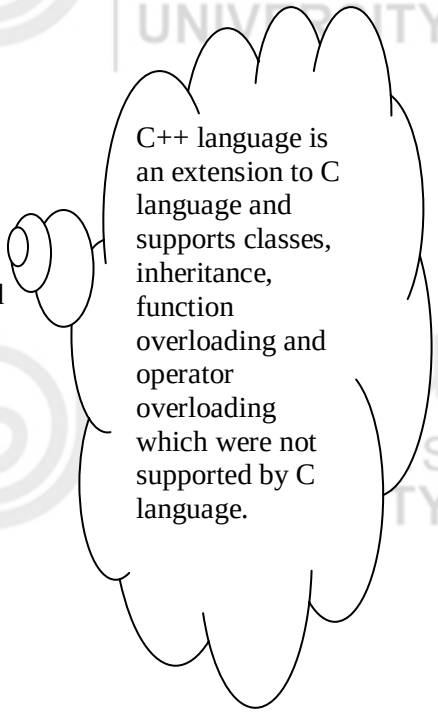
Structure **Page Nos.**

2.0 Introduction 24
2.1 Objectives 26
2.2 Basics of C++ 26
2.2.1 C++ Character Set
2.2.2 Identifiers
2.2.3 Keywords
2.3 A Simple C++ Program 28
2.4 Some Simple C++ Programs 31
2.5 Difference between C and C++ 34
2.6 Data Types in C++ 34
2.6.1 Built in Data Types
2.6.2 Derived Data Types
2.6.3 User- Defined Data Types
2.7 Type Conversion 37
2.8 Variables 38
2.9 Literals or Constants 39
2.10 Operators in C++ 40
2.10.1 Arithmetic Operators
2.10.2 Relational Operators
2.10.3 Logical Operators
2.10.4 Bitwise Operators
2.10.5 Precedence of Operators
2.10.6 Special Operators
2.10.7 Escape Sequence
2.11 Control Structure in C++ 45
2.11.1 Selection or conditional statements
2.11.2 Iterative or looping statement
2.11.3 Breaking Statement
2.12 I/O Formatting 56
2.12.1 Comments in C++
2.12.2 Unformatted console I/O formats
2.12.3 setw()
2.12.4 inline()
2.12.5 setprecision()
2.12.6 showpoint bit format flags
2.12.7 Input and output stream flags
2.13 Summary 60
2.14 Answers to Check Your Progress 61
2.15 Further Readings and References 64

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:

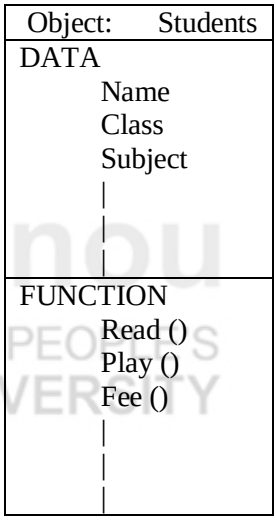


Figure 2.1: Representation of Object

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.

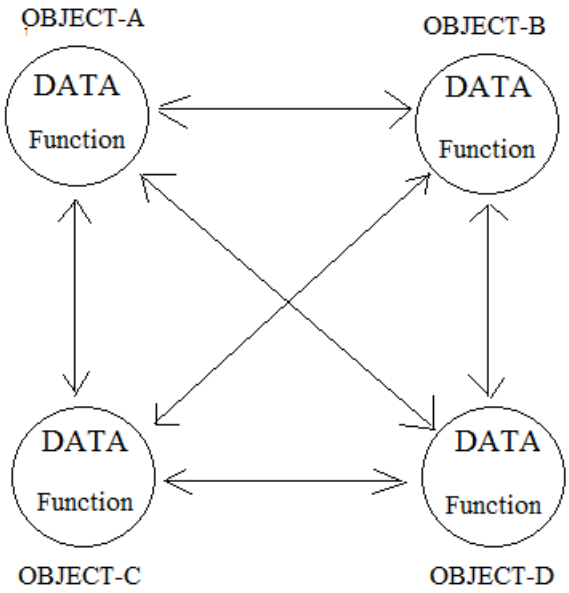


Figure 2.2: The Object-oriented approach

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:

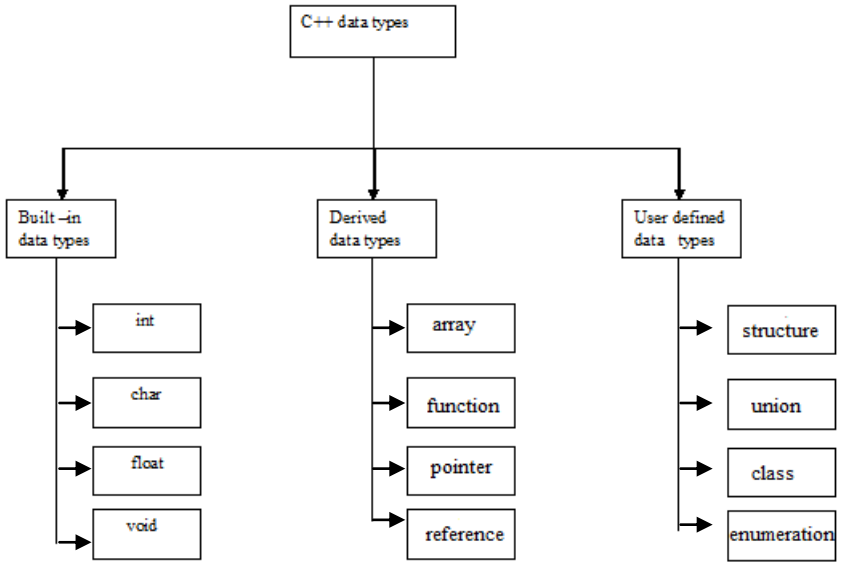


Figure 2.3 Hierarchy of C++ Data types

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.

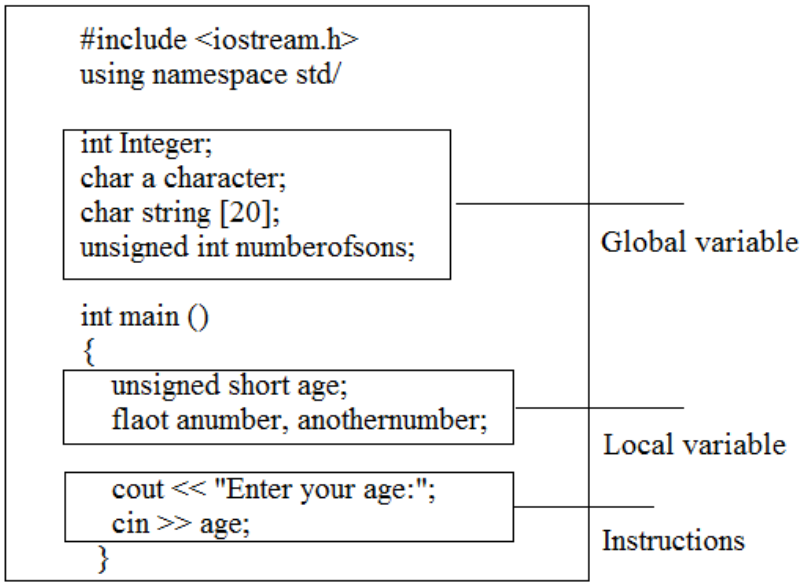


Figure 2.4: Scope of variables in C++ program

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	65
3.1	Objectives	65
3.2	Classification	66
3.3	Class	67
	3.3.1 Defining a Class	
	3.3.2 Encapsulation	
	3.3.3 Accessibility Rules/Labels	
3.4	Objects	70
	3.4.1 Instantiating Object	
3.5	Member Functions	74
	3.5.1 Nesting of Member Function	
	3.5.2 Passing Objects as Arguments:	
3.6	Friend Function	81
3.7	Static Members	84
3.8	Summary	87
3.9	Answers to Check Your Progress	87
3.10	Further Readings	92

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 93
4.1 Objectives 94
4.2 Constructors and Destructors 94
4.2.1 Characteristics of Constructors
4.2.2 Declaration of Constructors
4.2.3 Application of Constructors
4.2.4 Constructors with Arguments
4.3 Types of Constructor 100
4.3.1 Default Constructor
4.3.2 Parameterized Constructor
4.3.3 Copy Constructor
4.3.4 Constructor Overloading
4.3.5 Constructing two Dimensional Constructor
4.4 Destructors 107
4.5 Declaration of Destructor 107
4.6 Application of Destructor 109
4.7 Private Constructors and Destructors 109
4.8 Programs on Constructors and Destructors 110
4.9 Memory Management 114
4.10 Summary 116
4.11 Answers to Check your Progress 116
4.12 Further Readings and References 119

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