

Object Oriented Programme with C++

Study Material for MCA-302/MS-17

Study Material Prepared by

Manoj Kumar

Copyright ©, Manoj Kumar

Published by Excel Books, A-45, Naraina, Phase-I, New Delhi-110 028

Published by Anurag Jain for Excel Books, A-45, Naraina, Phase-I, New Delhi-110 028 and printed by him at Excel Printers, C-206, Naraina, Phase-I, New Delhi-110 028.

CONTENTS

Unit 1	Object Oriented Approach	7
	1.1 Introduction	
	1.2 Software Crisis	
	1.3 Software Evolution	
	1.4 A Look at Procedure-oriented Programming	
	1.5 Object Oriented Programming Paradigm	
	1.6 Basic Concepts of Object Oriented Programming	
	1.7 Benefits of OOP	
	1.8 Object Oriented Languages	
	1.9 Applications of OOP	
	1.10 Summary	
	1.11 Keywords	
	1.12 Review Questions	
	1.13 Further Readings	
Unit 2	Objects and Classes	19
	2.1 Introduction	
	2.2 Features of OOP	
	2.3 The Concept of a Class	
	2.4 The Class Keyword	
	2.5 Class Relationship	
	2.6 Summary	
	2.7 Keyword	
	2.8 Review Questions	
	2.9 Further Readings	
Unit 3	Constructors and Destructors	25
	3.1 Introduction	
	3.2 Constructors	
	3.3 Constructors of the string Class	
	3.4 String Class Assignment	
	3.5 String Access Operator and Method	
	3.6 String Comparison Operators	
	3.7 Appending to a String Object	
	3.8 String Searching	
	3.9 Summary	
	3.10 Keyword	
	3.11 Review Questions	
	3.12 Further Readings	
Unit 4	Implementing OOPs Concepts	55
	4.1 Introduction	
	4.2 What is Object Oriented Programming?	
	4.3 Data Abstraction	
	4.4 Software Engineering Trends	
	4.5 Summary	
	4.6 Keywords	
	4.7 Review Questions	
	4.8 Further Readings	
Unit 5	Implementing OOPs Concepts	63
	5.1 Introduction	
	5.2 Tokens	
	5.3 Basic Data Types	
	5.4 User-Defined Data Types	
	5.5 Derived Data Types	
	5.6 Symbolic Constants	
	5.7 Type Compatibility	

- 5.8 Declaration of Variables
- 5.9 Dynamic Initialization of Variables
- 5.10 Reference Variables
- 5.11 Operations and Expressions
- 5.12 Unary Operators
- 5.13 Prefix and Postfix Notations
- 5.14 Comparison Operators
- 5.15 Shift Operators
- 5.16 Shifting Positive Numbers
- 5.17 Shifting Negative Numbers
- 5.18 Bit-Wise Operators
- 5.19 Short Circuit Logical Operators
- 5.20 Conditional Operators
- 5.21 Order of Precedence of Operators
- 5.22 The Const Keyword
- 5.23 Operator and Function Overloading
- 5.24 Manipulation of strings using operators
- 5.25 Rules for overloading Operators
- 5.26 Function Overloading
- 5.27 Polymorphism
- 5.28 Streams
- 5.29 Summary
- 5.30 Keywords
- 5.31 Review Questions
- 5.32 Further Readings

Unit 6 Conditions and Control Statements 109

- 6.1 Introduction
- 6.2 The if....else Construct
- 6.3 Branching: The if-else Statement
- 6.4 LOOPING: THE While STATEMENT
- 6.5 More Looping: The do-While Statement
- 6.6 Still More Looping: The for Statement
- 6.7 Identifier Syntax
- 6.8 Expressions
- 6.9 Statements
- 6.10 Summary
- 6.11 Keywords
- 6.12 Review Questions
- 6.13 Further Readings

Unit 7 Design Issues 127

- 7.1 Introduction
- 7.2 SAD (Software Analysis and Design)
- 7.3 Object-Oriented Analysis and Design Issues
- 7.4 Object-Oriented Approach
- 7.5 The Big Picture
- 7.6 System Users
- 7.7 Context of a system
- 7.8 The Environment of a system
- 7.9 Relationship between SDLC and Modelling Language
- 7.10 Depicting Elaboration
- 7.11 Depicting Construction
- 7.12 Diagrammatic Conventions
- 7.13 Depicting Implementation
- 7.14 Depicting Transition
- 7.15 Summary
- 7.16 Keywords
- 7.17 Review Questions
- 7.18 Further Readings

Unit 8	Arrays, Lists, Stacks and Queues	134
	8.1 Introduction	
	8.2 One Dimension Arrays	
	8.3 Int Array Sorting	
	8.4 Strings - One-Dimensional char Arrays	
	8.5 Two or more Dimensional Arrays	
	8.6 Introduction to Pointers	
	8.7 Lists	
	8.8 Introduction to Linked Lists	
	8.9 Advantages of Linked Lists Over Arrays	
	8.10 Types of Linked Lists	
	8.11 Creating a Linked List Class	
	8.12 Stacks	
	8.13 Queues	
	8.14 Trees	
	8.15 Binary Trees	
	8.16 Implementing Binary Tree	
	8.17 Summary	
	8.18 Keywords	
	8.19 Review Questions	
	8.20 Further Readings	
Unit 9	Templates and Error Handling	176
	9.1 Introduction	
	9.2 Function of Templates	
	9.3 Classes of Template	
	9.4 The typename Keyword	
	9.5 Template Specialization	
	9.6 Point of Instantiation	
	9.7 Name Resolution	
	9.8 Error Handling	
	9.9 Error Isolation	
	9.10 Watch Values	
	9.11 Breakpoints	
	9.12 Stepping	
	9.13 Concurrent Object-oriented Systems	
	9.14 Summary	
	9.15 Keyword	
	9.16 Review Question	
	9.17 Further Readings	

UNIT

1

OBJECT ORIENTED APPROACH

LEARNING OBJECTIVES

After studying this unit, you should be able to:

- Describe procedure-oriented programming
- Describe object oriented paradigms and metaphors
- Describe the advantage of OO paradigm
- Describe basic concepts of object oriented programming
- Understand the benefits of OOP

UNIT STRUCTURE

- 1.1 Introduction
- 1.2 Software Crisis
- 1.3 Software Evolution
- 1.4 A Look at Procedure-oriented Programming
- 1.5 Object Oriented Programming Paradigm
- 1.6 Basic Concepts of Object Oriented Programming
- 1.7 Benefits of OOP
- 1.8 Object Oriented Languages
- 1.9 Applications of OOP
- 1.10 Summary
- 1.11 Keywords
- 1.12 Review Questions
- 1.13 Further Readings

1.1 INTRODUCTION

Programming practices have evolved considerably over the past few decades. As more and more programmers gained experience problems unknown hitherto, began to surface. The programming community became evermore concerned about the philosophy that they adopt in programming and approaches they practice in program development.

Factors like productivity, reliability, cost effectiveness, reusability etc. started to become major concern. A lot of conscious efforts were put to understand these problems and to seek possible solutions. This is precisely the reason why more and more programming languages have been developed and still continue to develop. In addition to this, approaches to program development have also been under intense research thereby evolving different frame works. One such, and probably the most popular one is object oriented programming approach or simply OOP.

1.2 SOFTWARE CRISIS

Exactly what gave birth to object oriented approach in programming couldn't be stated clearly. However, it is almost certain that by the end of last decade millions and millions lines of codes have been designed and implemented all over the world. The colossal efforts that went into developing this large population of programs would be rendered useless if these codes could not be reused.

Each time, writing a program used to be a new project in itself having nothing to do with the old existing codes. Programmers were, most of time, busy creating perhaps the same thing again and again; writing same or similar codes repeatedly in each project implementation.

Software industry faced a set of problems that are encountered in the development of computer software other than programs not functioning properly. Surveys conducted time to time revealed that more and more software development projects were failing to come to successful completion. Schedule and cost overrun were more frequent than seemed to be. This situation has been dubbed by various industry experts as "software crisis".

In the wake of this, questions like - how to develop software, how to support a growing volume of existing software, and how to keep pace with a rapidly growing demand for more error-free and efficient software - became important concerns in connection with software development activities.

Soon software crisis became a fact of life. Engineers and Scientists of the computing community were forced to look into the possible causes and to suggest alternative ways to avoid the adverse affects of this software crisis looming large at software industry all over the world.

It is this immediate crisis that necessitated the development of a new approach to program designing that would enhance the reusability of the existing codes at the least. This approach has been aptly termed as object oriented approach.

1.3 SOFTWARE EVOLUTION

Unlike consumer products, software is not manufactured. In this sense, software is not a passive entity rather it behaves organically. It undergoes a series of evolutionary stages throughout its lifetime – starting from a problem terminating into a solution. That is why software is said to 'develop' or 'evolve' and not manufactured.

In other words, software is born, passes through various developmental phases, gets established, undergoes maintenance and finally grows old before being commissioned out of service.

Software engineers have developed a number of different 'life styles' through which software passes. In general these life styles are known as software development life cycle or SDLC in short.

Following are the developmental phases of software.

Study

The genesis of any software begins with the study of the problems, which it intends to solve. Software cannot be envisaged unless there is a problem that it must solve. Therefore, studying the problem in depth, understanding the true nature of the problem and representing the problem in comprehensible manner is what necessitates inclusion of this phase.

Analysis

It is a detailed study of the various operations performed by the proposed software. A key question that is considered in this phase of development is – What must be done to solve the problem? One aspect of analysis is defining the boundaries or interface of the software.

During analysis, data are collected in available files, decision points, and transactions handled by the present system. Bias in data collection and interpretation can be fatal to the developmental efforts. Training, experience and common sense are required for collection of the information needed to do the analysis.

Once analysis is completed the analyst has a firm understanding of what is to be done. The next step is to decide how the problem might be solved. Thus, in the software systems design, we move from the logical to the physical aspects of the life cycle.

Design

The most creative and challenging phase of software life cycle is design. The term design describes both a final software system and a process by which it is developed. It refers to the technical specifications (analogous to the engineer's blueprints) that will be applied in implementing the software system. It also includes testing the software. The key question around which this phase revolves is – How should the problem be solved? Design phase is itself subdivided into sub-phases discussed below.

The first step is to determine how the output is to be produced and in what format. Samples of the output (and input) are outlined.

Second, input data and master files (data base) have to be designed to generate the required output. The operational (processing) phase are handled through program construction and testing, including a list of the programs needed to meet the software objectives and complete documentation.

Finally, details related to justification of the system and an estimate of the impact of the software on the user are documented and evaluated before it is implemented.

Implementation

This phase is primarily concerned with coding the software design into an appropriate programming language; testing the programs and installing the software. User training, site preparation, and data migration are other important issues in this phase.

Maintenance

Change is inevitable. Software serving the users' needs in the past may become less useful or sometimes useless in the changed environment. Users' priorities, changes in organizational requirements, or environmental factors may call for software enhancements. A bank, for instance, may decided to increase its service charges for checking accounts from Rs. 3.00 to Rs. 450 for a minimum balance of Rs. 3,000. In this phase the software is continuously evaluated and modified to suit the changes as they occur.

1.4 A LOOK AT PROCEDURE-ORIENTED PROGRAMMING

Before you get into OOP, take a look at conventional procedure-oriented programming in a language such as C. Using the procedure-oriented approach; you view a problem as a sequence of things to do such as reading, calculating and printing. Conventional programming using high-level languages such as COBOL, FORTRAN and C is commonly known as procedure-oriented programming. You organize the related data items into C structures and write the necessary functions (procedures) to manipulate the data and, in the process, complete the sequence of tasks that solve your problem.

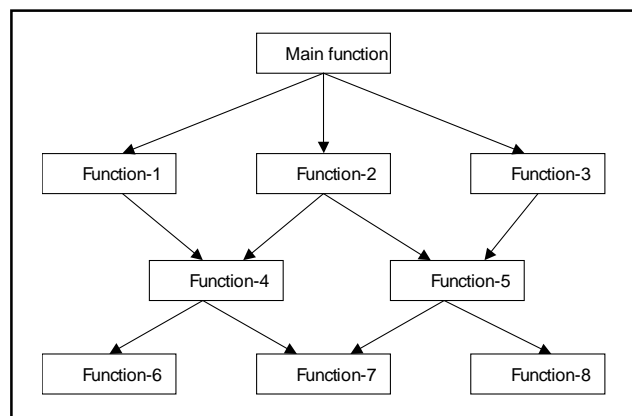


Figure 1.1: Typical Structure of Procedure-oriented Programs

Although data may be organized into structures, the primary focus is on functions. Each C function transforms data in some way. For example, you may have a function that calculates the average value of a set of numbers, another that computes the square root, and one that prints a string.

You do not have to look far to find examples of this kind of programming – C function libraries are implemented this way. Each function in a library performs a well-defined operation on its input arguments and returns the transformed data as a return value. Arguments may be pointers to data that the function directly alters or the function may have the effect of displaying graphics on a video monitor.

A typical program structure for procedural programming is shown in Figure 1.1. The technique of hierarchical decomposition has been used to specify the tasks to be completed in order to solve a problem.

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

While we concentrate on the development of functions, very little attention is given to the data that are being used by various functions. What happens to the data? How are they affected by the functions that work on them?

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Figure 1.2 shows the relationship of data and functions in a procedure-oriented program.

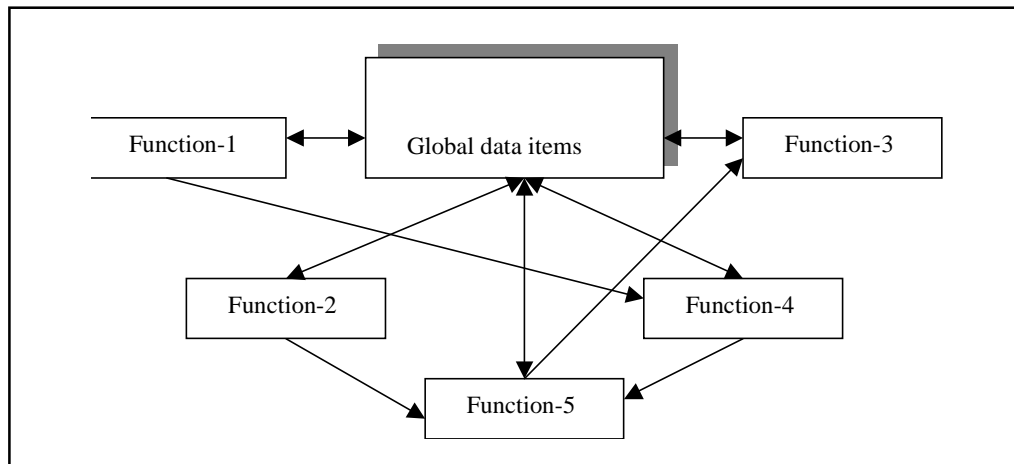


Figure 1.2: Relationship of Data and Functions in Procedural Programming

Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we should also revise all functions that access the data. This provides an opportunity for bugs to creep in.

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

Some characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.

1. Describe Various developmental phases of software.
2. What do you understand by procedure-oriented Programming?
3. What are the characteristics of procedure-Oriented Programming?

1.5 OBJECT ORIENTED PROGRAMMING PARADIGM

The term object-oriented programming (OOP) is widely used, but experts do not seem to agree on its exact definition. However, most experts agree that OOP involves defining abstract data types (ADT) representing complex real-world or abstract objects and organizing programs around the collection of ADTs with an eye toward exploiting their common features. The term data abstraction refers to the process of defining ADTs; inheritance and polymorphism refer to the mechanisms that enable you to take advantage of the common characteristics of the ADTs - the objects in OOP. This chapter further explores these terms later.

Before going any further into OOP, take note of two points. First, OOP is only a method of designing and implementing software. Use of object-oriented techniques does not impart anything to a finished software product that the user can see. However, as a programmer while implementing the software, you can gain significant advantages by using object-oriented methods, especially in large software projects. Because OOP enables you to remain close to the conceptual, higher-level model of the real world problem you are trying to solve, you can manage the complexity better than with approaches that force you to map the problem to fit the features of the language. You can take advantage of the modularity of objects and implement the program in relatively independent units that are easier to maintain and extend. You can also share code among objects through inheritance.

Secondly, OOP has nothing to do with any programming language, although a programming language that supports OOP makes it easier to implement the object-oriented techniques. As you will see shortly, with some discipline, you can use objects even in C programs.

1.6 BASIC CONCEPTS OF OBJECT ORIENTED PROGRAMMING

Procedural programming deals with functional parts of the problem. Programmers identify what actions must be taken to solve the problem at hand. Let us now, try to understand what aspect of problems are dealt with in object-oriented approach. We shall be discussing some essential concepts that make a programming approach object-oriented.

In object-oriented parlance, a problem is viewed in terms of the following concepts:

1. Objects
 2. Classes
 3. Data abstraction
 4. Data encapsulation
 5. Inheritance
 6. Polymorphism
 7. Dynamic binding
 8. Message passing
1. **Objects:** Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data; they may also represent user-defined data such as vectors, time and lists.

They occupy space in memory that keeps its state and is operated on by the defined operations on the object. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other data or code.

The figure shows the two notations popularly used.

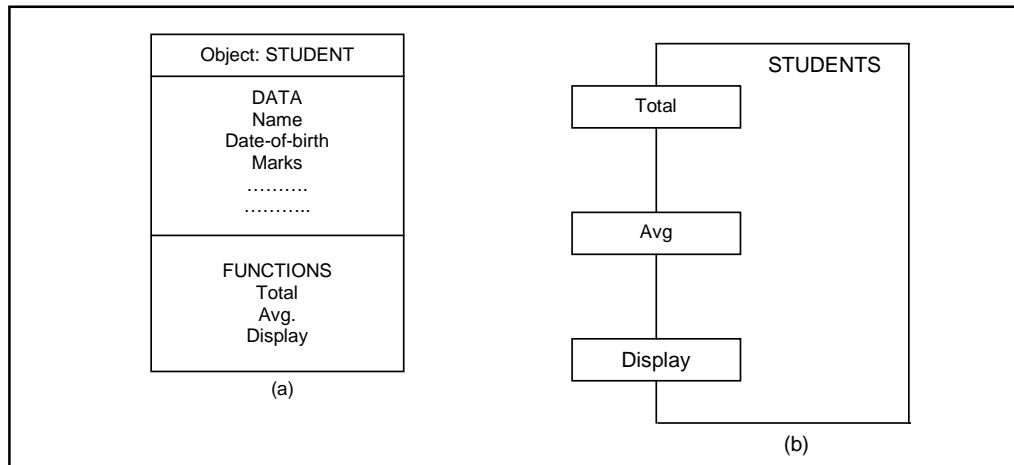


Figure 1.3

2. **Classes:** A class represents a set of related objects. The object has some attributes, whose value consist much of the state of an object. The class of an object defines what attributes an object has. The entire set of data and code of an object can be made a user-defined data type with the help of a class.

Classes are user defined data types that behave like the built-in types of a programming language. Classes have an interface that defines which part of an object of a class can be accessed from outside and how. A class body that implements the operations in the interface, and the instance variables that contain the state of an object of that class.

The data and the operation of a class can be declared as one of the three types:

- a. **Public:** These are declarations that are accessible from outside the class to anyone who can access an object of this class.
- b. **Protected:** These are declarations that are accessible from outside the class to anyone who can access an object of this class.
- c. **Private:** These are declarations that are accessible only from within the class itself.

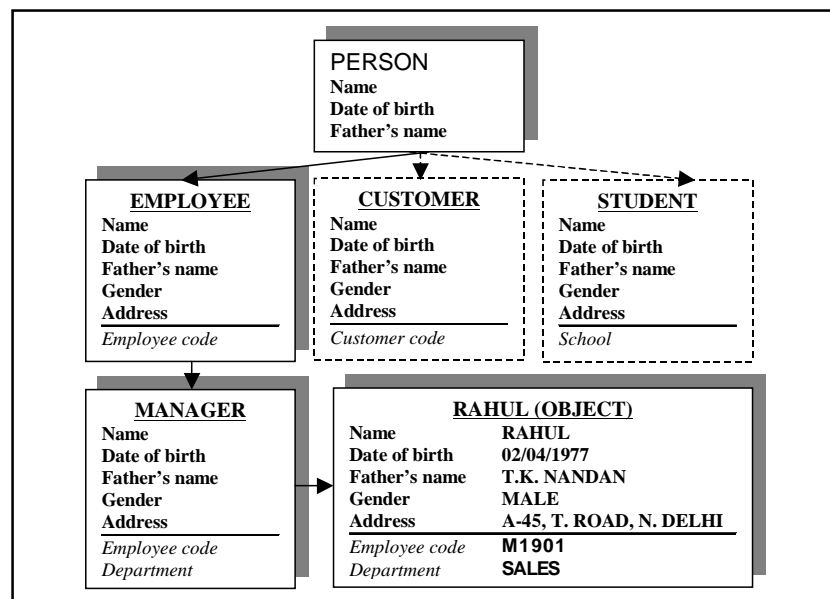


Figure 1.4: Property Inheritance

3. **Data Abstraction:** Abstraction refers to the act of representing essential-features without including the background details or explanations. Abstraction is indispensable part of the design process and is essential for problem partitioning. Partitioning essentially is the exercise in determining the components of the system. However, these components are not isolated from each other, they interact with each other. Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).
4. **Data Encapsulation:** The wrapping up to data and functions into a single unit (class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world and only those functions that are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding.
5. **Inheritance:** Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example, an manager class is a type of the class employee, which again is a type of the class person as illustrated below.

The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived. The power of inheritance lies in the fact that all common features of the subclasses can be accumulated in the super class. In other words, a feature is placed in the higher level of abstraction. Once this is done, such features can be inherited from the parent class and used in the subclass directly. This implies that if there are many abstract class definitions available, when a new class is needed, it is possible that the new class is a specialization of one or more of the existing classes.

In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. Each subclass defines only those features that are unique to it. In OOP, The concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. Each subclass defines only those features that are unique to it.

6. **Polymorphism:** Polymorphism means the ability to take more than one form. An operation may exhibit different behavior in different instances. The behavior depends upon the types of the data used in the operation. For example considering the operator plus (+).

$$16 + 4 = 20$$

$$\text{"xyz"} + \text{"sqr"} = \text{"xyzsqr"}$$

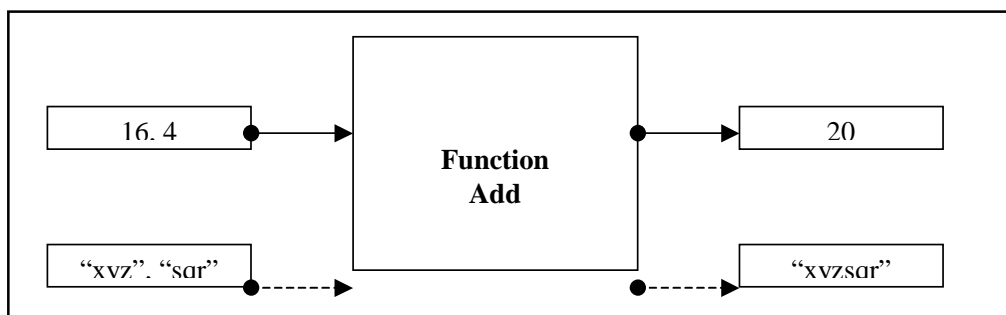


Figure 1.5

The process of making an operator to exhibit different behavior at different instance is called operator overloading. Another example of polymorphisms is function overloading, where a single function can perform various different types of task.

7. **Dynamic binding:** Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of call at run-time. This is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference. For example in the above figure, by inheritance, every object will have this procedure. Its algorithm is, however, unique to each object so the procedure will be redefined in each class that defines the objects. At run-time, the code matching the object under reference will be called.
8. **Message passing:** Message passing is another feature of object-oriented programming. An object-oriented program consists of a set of objects that communicate with each other. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counter-parts.

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

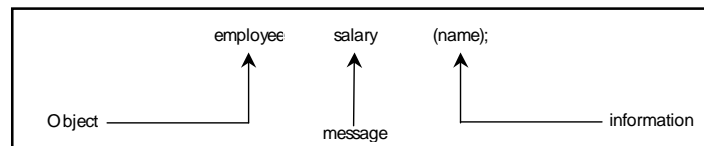


Figure 1.6

As is evident from the ongoing discussion, object-oriented approach deals with objects and classes as against series of functions and as actions adopted by procedural approach.

1.7 BENEFITS OF OOP

Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The technology provides greater programmer productivity, better quality of software and lesser maintenance cost. The principle advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding help the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those objects in the program.
- It is easy partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. There

are a number of issues that need to be tackled to reap some of the benefits stated above. For instance, object libraries must be available for reuse. The technology is still developing and current products may be superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.

Developing a software that is easy to sue makes it hard to build. It is hoped that the object-oriented programming tools would help manage this problem.

1.8 OBJECT ORIENTED LANGUAGES

Object-oriented programming is not the right of any particular language. Although languages like C and Pascal can be used but programming becomes clumsy and may generate confusion when program grow in size. A language that is specially designed to support the OOP concepts makes it easier to implement them.

To claim that they are object-oriented they should support several concepts of OOP.

Depending upon the features they support, they are classified into the following categories:

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

Major features required by object-based programming are:

1. Data encapsulation.
2. Data hiding and access mechanisms.
3. Automatic initialization and clear-up of objects.
4. Operator overloading.

Languages that support programming with objects are said to be object-based programming languages. These do not support inheritance and dynamic binding; ADA is a typical example. Object-Oriented programming incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Thus

Object-oriented programming = Object-based features + inheritance + dynamic binding.

Languages that support these features include C++, Small talk, Java among others.

Table 1.1: Characteristics of Some OOP Languages

Characteristics	Simula	Small talk 80	Objective C	C ++	ADA	Object Pascal	Eittel
Binding (early or late)	Both	Late	Both	Both	Early	Late	Early
polymorphism (operator overloading)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Data hiding	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Concurrency	Yes	Poor	Poor	Poor	Difficult	No	Promised
Inheritance	Yes	Yes	Yes	Yes	No	Yes	Yes
Multiple Inheritance	No	Yes	Yes	Yes	No	Yes	Yes
Garbage Collection	Yes	Yes	Yes	Yes	No	Yes	Yes
Persistence	No	Pormised	No	No	Like 3GL	No	Some support
Genericity	No	No	No	No	Yes	No	Yes
Object lib	Yes	Yes	Yes	No	Not much	Yes	Yes

1.9 APPLICATIONS OF OOP

Object Oriented approach offers perhaps the most logical description of the real world. Therefore, it can be applied in almost all the situations of problem solving. Because of its effectiveness in problem solving and programming, OOP has been adopted all over the software industry. The existing systems are being migrated to OOP.

One important aspect of OOP that is particularly beneficial is the framework, which encompasses all the phases of a system development. Thus, OOA (Object Oriented Analysis), OOD (Object Oriented Design), OOT (Object Oriented Testing) etc. are far more suitable tools than their non-object oriented counterparts.

Above all, the reusability feature of the Object Oriented approach has facilitated the rapid growth of software both in variety and scope.

Student Activity 2

1. What do you mean by abstraction?
2. What are the advantages of OOP over procedure oriented programming?
3. Define the following:
 - (i) Objects
 - (ii) Classes
 - (iii) Data Encapsulation
 - (iv) Inheritance
 - (v) Polymorphism
4. What are the benefits of OOP?
5. List some applications of OOP.

1.10 SUMMARY

Programming practices have evolved considerably over the past few decades. By the end of last decade, millions and millions lines of codes have been designed and implemented all over the world. The main objective is to reuse these lines of codes. More and more software development projects were software crisis. It is this immediate crisis that necessitated the development of object-oriented approach which supports reusability of the existing code.

Software is not manufactured. It is evolved or developed after passing through various developmental phases including study, analysis, design, implementation, and maintenance.

Conventional programming using high level languages such as COBOL, FORTRAN and C is commonly known as procedures-oriented programming. In order to solve a problem, a hierarchical decomposition has been used to specify the tasks to be completed.

OOP is a method of designing and implementing software. Since OOP enables you to remain close to the conceptual, higher-level model of the real world problem, you can manage the complexity better than with approaches that force you to map the problem to fit the features of the language. Some essential concepts that make a programming approach object-oriented are objects, classes, Data abstraction, Data encapsulation, Inheritance, Polymorphism, dynamic binding and message passing. The data and the operation of a class can be declared public, protected or private. OOP provides greater programmer productivity, better quality of software and lesser maintenance cost. Depending upon the features they support, they are classified as object based programming languages and object-oriented programming languages.

Object-oriented programming = Object-based features + inheritance + dynamic binding.

Object Oriented approach

Object Oriented approach offers the most logical description of the real world. The framework of OOP encompasses all the phases of system development. The reusability features of OOP has facilitated the rapid growth of software.

1.11 KEYWORDS

Analysis: A detailed study of the various operations performed by the proposed software.

Design: The term design describes both a final software system and a process by which is developed.

Implementation: This phase is primarily concerned with coding the software design into an appropriate programming language. Testing the programs and installing the software.

Maintenance: In this phase the software is continuously evaluated and modified to suit the changes as they occur.

Object: In the object-oriented model a combination of a small amount of data and instructions about what to do with that data when the object is selected or activated.

Object-oriented programming (OOP) Language: Programming language that encapsulates a small amount of data along with instructions about how to manipulate that data; inheritance and reusability features provide functional benefits.

Class: A class represents a set of related object.

Data Abstraction: The act of representing essential features without including the background details or exploitations.

Data encapsulation: The wrapping up of data and functions in a single unit (class).

Inheritance: The process by which objects of same class acquire the properties of object of another class.

Polymorphism: The ability to take more than one form.

Dynamic binding: The linking of a procedure call to the code to be executed in response to the call.

1.12 REVIEW QUESTIONS

1. Write a note on various problems faced by the software industry.
2. Write a note on software evolution.
3. What do you mean by object oriented programming? What are its benefits?
4. What do you mean by dynamic binding?
5. What is the difference between data abstraction and data encapsulation?
6. Name some language that support object-oriented paradigm.
7. Fill in the blanks:
 - (a) _____are the basic run-time entities in an object-oriented system.
 - (b) The ability of a function or operator to act in different ways on different data types is called_____.
 - (c) The process of linking a procedure call to the code to be executed in response to the call is called_____.
 - (d) A_____will invoke a function in the receiving object that generates the desired result.

- (e) It is possible to have —— of an object to co-exist without any interference.
- 8. State (T) rue or (F)alse:
 - (a) Object oriented programming languages permit reusability of the existing code.
 - (b) Languages earlier than procedural programming languages made use of only global variables.
 - (c) A class permits us to build user-defined data types.
 - (e) The data of a class cannot be declared public
 - (f) Objects are to classes as variables are to data types.

Answers to Review Answers

- 6. C++, Small talk, Java
- 7. (a) object (b) polymorphism (c) Dynamic binding (d) message (e) multiple instances
- 8. (a) True (b) True (c) True (d) False (e) True

1.13 FURTHER READINGS

Herbert Schildt, *The complete Reference-C++*, Tata Mc Graw Hill

Robert Lafore, *Object Oriented Programming in Turbo C++*, Galgotia Publications.

E. Balagurusamy, *Object Oriented Programming through C++*, Tata McGraw Hill.

UNIT

2

OBJECTS AND CLASSES

LEARNING OBJECTIVES

After studying this unit, you should be able to:

- Describe the features of OOP
- Describe a class
- Describe various class relationships
- Define base class and derived class

UNIT STRUCTURE

- 2.1 Introduction
- 2.2 Features of OOP
- 2.3 The Concept of a Class
- 2.4 The Class Keyword
- 2.5 Class Relationship
- 2.6 Summary
- 2.7 Keywords
- 2.8 Review Questions
- 2.9 Further Readings

2.1 INTRODUCTION

Object-Oriented Programs (OOPs) attempt to emulate real world in software system. The real world consists of objects, categorized in classes. For example, you're using an object categorized as a book to learn about programming. OOP describes a software system in terms of real world objects.

Consider a 14 inch television, an object. It has certain attributes: a 14 inch screen and controls for adjusting a volume, brightness and color. It also has certain behaviors: it shows moving pictures and allows a viewer to change channels. Finally, it has an identity: a serial number that distinguishes it from other televisions.

How was the TV made? The management of the TV company decided that it should have certain attributes and behave in a certain way. These decisions were documented as blueprint, which was used to make TV. The blueprint in OOP is called class. All objects belong to some class that defines their attributes and behaviors. An object is an instance of a class. In OOP, classes have attributes represented by data members. For example, a TV has an attribute called size, which may have value 14 inches and a state (on/off).

The attributes distinguish an object of the class. Classes have behaviors, which are represented as methods. The methods define how an object acts or reacts.

2.2 FEATURES OF OOP

The features of OOP are listed below:

1. **Real world:** The real world consists of objects like books, tables, chairs and TV. The same concept has been introduced into programming, bringing it closer to real life.
2. **Information encapsulation (Hiding):** Objects provide the benefit of information hiding. Electrical wiring in a TV should not be tampered with and should be hidden from the user. OOP allows you to encapsulate data that you do not want users of the object to access. Typically attributes of a class are encapsulated.
3. **Abstraction:** It allows us to focus only on those parts of an object that concern us. To continue the example given above, a person operating the TV does not need to know the intricacies of how it works. The person just need to know how to switch it on, change channels and adjust the volume. All the details that are unnecessary to the user are encapsulated, leaving only a simple interface to interact with. Providing only the users what they need to know is an abstraction. Abstraction lets us ignore the irrelevant details and concentrate on the essentials.
4. **Inheritance:** A TV has certain common attributes and functionality. For example, you expect a TV to have a screen. Companies add more features to the basic functionality of TVs to create their own brand or class of TVs. This means they've added new functionality to the existing class without creating a new one from the scratch. For example, a color TV has been derived from TV class. The child class, color TV, inherits the properties of the parent class, TV. Similarly, programmers can reuse code rather than rewrite it. You can use the existing features of the class and add more functionality to it. This is known as inheritance. Using inheritance you can extend the scope of classes and at the same time, reduce the amount of code to be written. Through inheritance you can provide the reusability of code.

2.3 THE CONCEPT OF A CLASS

The user defined data type, class, distinguishes C++ from traditional procedural language. A class is a new data type that is created to solve a particular kind of problem. Once a class is created anyone can use it, without knowing the specifics of how it works or even how a class is built.

Class is the core building block of a C++ program. It represents a real world object template concealing data and revealing member functions to be used by other classes. It resembles a C structure but has a number of extended features that make it an object oriented programming entity. A class may be defined as follows.

Class syntax	Class example
<pre>class < class_name> { private : data members; member functions; public : data members; member functions; }</pre>	<pre>class abc { private : int a; void addit(); public : int b; void printit(); }</pre>

Once the class is defined, an instance of this class (object) may be created as follows:

```
class   abc   pqr;

class   pqr = abc;
```

In either case an object of class abc type named pqr is created. Just what happens when an object of a class is created is discussed in what follows.

2.4 THE CLASS KEYWORD

The class keyword is used to declare a class. The braces are used to indicate the start and end of a class body. Member variables and member functions are declared inside the class body. A semicolon is used to end the declaration.

Example

```
#include <iostream.h>

class car
{
    ...
};

int main()
{
    car ford;
    ...
    return (0);
}
```

Student Activity 1

1. What is an object?
2. What are the features of OOP?
3. Define the following:
 - (a) Encapsulation
 - (b) Abstraction
 - (c) Inheritance
4. What is a class?
5. Give the syntax to declare a class.

2.5 CLASS RELATIONSHIP

Design and Construction

You need to be an expert carpenter to design a wooden bridge, but you need to know the fundamentals of wooden construction and the properties of the wood and the carpenters who will construct the bridge. Similarly, to design a piece of software successfully, you need fairly detailed knowledge of the chosen programming language. A good bridge designer considers and respects the properties of the material and resources. Similarly, a good programmer builds on the strengths of the implementation language.

Types of Relationships

1. **A “kind of” relationship:** Taking the example of a human being and an elephant, both are “kind of” mammals. Mammals have attributes - eyes and limbs. They also have behavior, walking. As human beings and elephants are “kind of” mammals, they share the attributes and behaviors of mammals. Human beings and elephants are subsets of mammal class. The following Figure 3.1 depicts the relationship between the mammal and human beings.

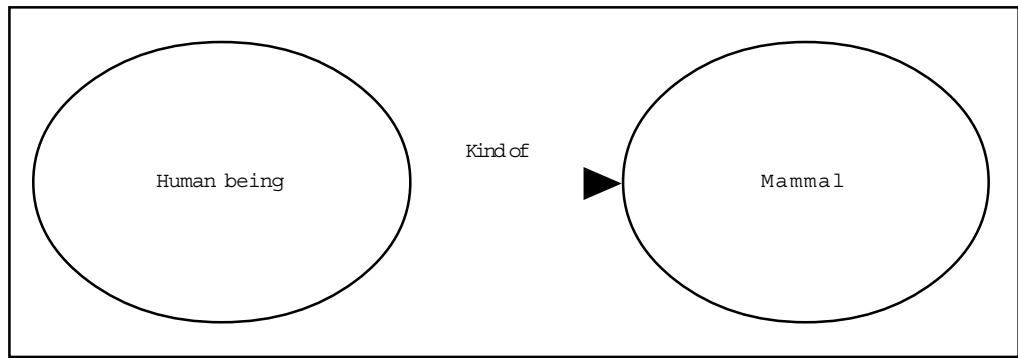


Figure 2.1

2. ***“Is a” relationship:*** The previous example depicts the relationship between human beings and mammals. Let's take an instance of a human being and therefore a mammal. The following figure depicts “is a” relationship:

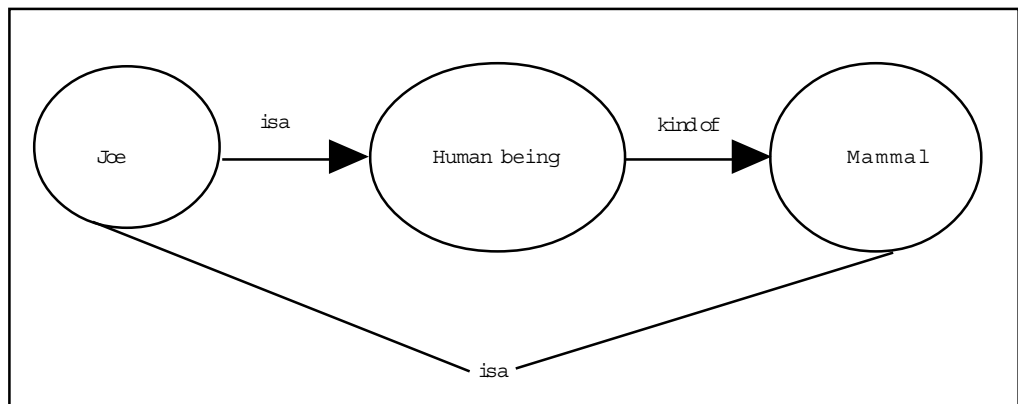


Figure 2.2

3. ***“Has a” relationship/part of relationship:*** A human being has a heart. This represents “has a” relationship. The same relationship can be represented as: A heart is a part of human being. The following figure depicts the relationship between a human being and heart.

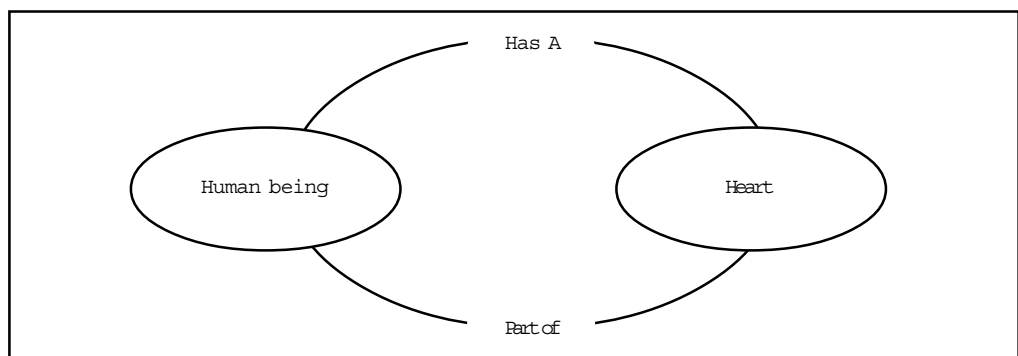
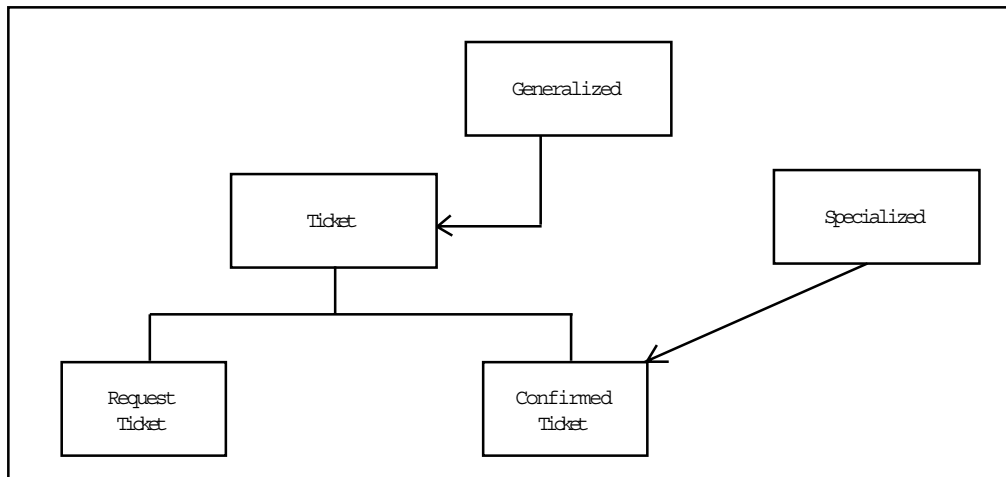


Figure 2.3

Base Class and Derived Class (Super class and Sub class)

Let us take the example of air tickets. Air tickets can be of two types – confirmed and request. Both these ticket have a lot of common attributes, for example, flight number, date, time and destination. However, a confirmed ticket would also have a seat number, while a request ticket would have status.

**Figure 2.4**

In the example above, we have incorporated the structure and behavior that is common to the two classes to form a superclass. The superclass represents generalized properties and the subclass represents specialization, in which attributes and behavior from the super class are added, modified or hidden.

A super class is a class from which another class inherit properties. A super class shares its properties with its child classes. A sub class is a class that inherits attributes and methods from super class.

Student Activity 2

1. Describe the “kind of” class relationship. Give an example
2. Describe “Is a” class relationship. Give an example.
3. What is a base class?
4. What is derived class?

2.6 SUMMARY

Object-oriented programs attempt to emulate real world in software system. The real world consists of objects, categorized in classes. The attributes distinguish an object of the class. Classes have behaviours, which are represented as methods. The methods define how an object acts or reacts.

Objects provide the benefit of information hiding abstraction and Inheritance. A class is a new data type that is created to solve a particular kind of problem. Once a class is created anyone can use it, without knowing the specifics of how it works or even how a class is built.

To design a piece of software successfully, you need fairly detailed knowledge of the chosen programming language. A good programmer, builds on the strength of the implementation language.

The superclass represents generalized properties and the subclass represents specialization, in which attributes and behaviours from the super class are added, modified or hidden.

2.7 KEYWORDS

Abstraction: The act of representing essential features without including the background details or exploitations.

Encapsulation: The wrapping up of data and functions in a single unit (class).

Inheritance: The process by which objects of some class acquire the properties of object of another class.

Class: A new data type that is created to solve a particular kind of problem.

Super class: A class from which another class inherit properties.

Subclass: A class that inherits attributes and methods from super class.

2.8 REVIEW QUESTIONS

1. What is object-oriented programming? List down its features and functions.
2. Diagrammatically explain different kinds of class relationships.
3. Define the following terms:
 - a. Object Oriented programming
 - b. Super class
 - c. Sub class
4. Distinguish between:
Base class and Derived class
5. Fill in the blanks:
 - (a) _____programs attempt to emulate real world in software system.
 - (b) Objective provide the benefit of _____, and _____.
 - (c) A _____is a new data type that is created to solve a particular kind of problem.
 - (d) A_____is used to end the declaration.
 - (e) A subclass inherits _____and_____from super class.
6. State (T) rue or (f)alse:
 - (a) The attributes distinguish an object of the class.
 - (b) Behaviours of classes are represented by methods.
 - (c) “Is a” relationship exists in human being and mammal.
 - (d) Super class inherits properties from its subclass.
7. What is the difference between superclass and subclass?
8. Give two examples showing:
 - (a) ‘Is a’ relationship
 - (b) kind of’ relationship
 - (c) ‘Has a’ relationship

Answers to Review Answers

1.

(a) Object oriented	(b) encapsulation, abstraction, Inheritance
(c) class	(d) semicolon
(e) attributes, methods	
2.

(a) True	(b) True
(c) False	(d) False

2.9 FURTHER READINGS

Herbert Schildt, *The complete Reference-C++*, Tata Mc Graw Hill

Robert Lafore, *Object Oriented Programming in Turbo C++*, Galgotia Publications.

E. Balagurusamy, *Object Oriented Programming through C++*, Tata McGraw Hill.

UNIT

3

CONSTRUCTORS AND DESTRUCTORS

L E A R N I N G O B J E C T I V E S

After studying this unit, you should be able to:

- Define constructors and destructors
- Describe the need of constructors
- Declare constructor
- Understand default and parameterized constructors
- Understand the dynamic initialization of constructors
- Describe the characteristics of constructors
- Describe various constructors of string class
- Describe string access operator and method.

U N I T S T R U C T U R E

- 3.1 Introduction
- 3.2 Constructors
- 3.3 Constructors of the string Class
- 3.4 String Class Assignment
- 3.5 String Access Operator and Method
- 3.6 String Comparison Operators
- 3.7 Appending to a String Object
- 3.8 String Searching
- 3.9 Summary
- 3.10 Keywords
- 3.11 Review Questions
- 3.12 Further Readings

3.1 INTRODUCTION

It is very common for some part of an object to require initialization before it can be used. This was performed by using the function in `print()`. Because the requirement for initialization is so common, C++ allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor functions.

The complement of constructor is the destructor. In many circumstances, an object will need to perform some action or actions when it is destroyed. Local objects are created when their block is entered, and destroyed when their block is left. Global objects are destroyed when the program terminates. When an object is destroyed, its destructor is automatically called.

3.2 CONSTRUCTORS

When an object of a class is created, its member variables are in uninitialized state, loaded with any arbitrary values. It is desirable to have the member variables initialized to some legal values at the time of object creation.

A constructor is a member function of a class, having the same name as its class and which is called automatically each time an object of that class is created. It is used for initializing the member variables with desired initial values.

Example:

```
class student    {

                                private:

                                int rollno;

                                float    marks;

                                public:

                                .

                                .

                                .

                                student()                //constructor of class student

                                {

                                rollno = 0;

                                marks = 0.0;

                                }

                                .

                                .

                                }
```

In this code snippet, the class student has a member function student() that will be called each time an object of this class is created, thereby initializing the member variables with appropriate values.

Need for constructor

A variable (including structure and array types) in C++ may be initialized with a value at the time of its declaration.

Example:

1.	int I = 4;	Integer I has been initialized with 4
2.	float p = 9.1;	Float p has been initialized with 9.1
3. with	int a[3] = {2, 7,9};	Array elements a[0], a[1] and a[2] have been initialized 2, 7 and 9 respectively
4.	<pre>struct student { int rollno; float marks; }; int main() { student s1 = {0, 0.0}; }</pre>	S1.rollno and s1.marks (Members of structure s1) have been initialized to 0 and 0.0 respectively

But no such initialization is possible with objects of a class because the private members are not accessible from outside the class.

<pre> 5. class student { private: int rollno; float marks; }; int main() { student s1 = {0, 0.0}; } </pre>	<p>Illegal!! Private data members not accessible from non-member function main()</p>
---	--

One can, however, define a member function (say doinit()) in the class and calling it explicitly, to provide initial values to the data members as shown below:

<pre> 6. class student{ private: int rollno; float marks; public: doinit() { rollno = 0; marks = 0.0; } }; int main() { student s1; s1.doinit(); } </pre>	<p>Public member function doinit will allow function main() to carry out the initialization</p> <p>Doinit() called explicitly to effect the initialization</p>
--	--

As is evident from the above examples that the programmer may carry out initialization by writing the member function and calling it explicitly. This approach rests lot of responsibilities on the programmer. The responsibility of initialization may be shifted, however, to the compiler by including a member function called constructor.

Therefore, a *constructor* for a class is needed so that the compiler automatically initializes an object as soon as it is created without explicitly calling any function for the same. A class constructor, if defined, is called whenever a program creates an object of that class.

Declaration and definition

A constructor is defined like any other member function. However,

- It has same name as the name of the class it belongs to
- It does not have any return type (not even void)

Example:

```

class abc      {
    private:
        int i;

    public:
        int j, k;

```

```
        abc()                //constructor (same name as class, i.e. abc)
        {                    //no return type
            i=0;j=0;k=0;
        }
        .....
        .....
    };
```

In this example, constructor `abc::abc()` was defined inline. It may also be defined outline.

Example:

```
class abc    {
    private:
        int i;
    public:
        int j, k;
        abc();                //constructor (same name as class, i.e. abc)
        ....
        ....
};

abc::abc()    {                    //outline definition
    i=0;j=0;k=0;
}
```

In the examples given above, the constructors have been defined as a public member so that any function of any object may create this object. However, it can be defined as private or protected as well, but in those cases not all functions can create this object. In later cases the object cannot be created in non-member functions but can be created only in member and friend functions.

Example:

```
class abc    {
    private:
        int i;
        abc(){ i=0;j=0;k=0; }    //constructor defined private inline
    public:
        int j, k;
        void aaa(void);
        friend void bbb(void);    //friend function
        :
};
```

```

void aaa(void)  {
    abc a; //create an instance a of class abc; valid here aaa being
    :      //member function
            }

void bbb(void)  {
    abc b; //create an instance b of class abc; valid here bbb being
    :      //friend function
            }

int main()      {
    abc c; //create an instance b of class abc; invalid here main being
    :      //non-member function abc::abc() not accessible from here
            }

```

Generally, therefore, constructors are defined as public member unless otherwise there is a good reason against.

Default constructors

A constructor may take argument(s). A constructor taking no argument(s) is known as *default constructor*. If the programmer does not provide any constructor to a class, the compiler automatically provides one with no argument(s). Default constructor provided by the compiler does nothing more than initializing the data members with dummy values. However, if a constructor is defined for the class the default constructor is no more available, it goes into hiding.

Example:

```

class abc      {
    private:
        int i;

    public:
        int j, k;          //no explicit constructor defined
        :
    };

int main()     {
    abc c;      //create an instance c of class abc; valid the constructor
    :          //abc() is provided by the compiler
    }

```

Parametrized constructors

A constructor may also have parameter(s) or argument(s), which can be provided at the time of creating an object of that class.

Example:

```
class abc    {
    private:
        int i;
    public:
        int j, k;
        abc(int aa, int bb, int cc);    //constructor with
                                        //parameters aa, bb and cc
        {
            i = aa; j = bb; k = cc;
        }
        :
};

int main()   {
    abc abc1(5, 8, 10);    //create an instance abc1 of class abc;
    :                    //initialize i, j, and k with 5, 8, and
                        //10 respectively
    abc abc2(100, 200, 300); //create an instance abc2 of class abc;
    :                    //initialize i, j, and k with 100, 200,
                        //and 3000 respectively
}
```

Recall that once an explicit constructor is defined, the default constructor is no more available, therefore, the constructor must be called with proper number of argument(s).

Example:

```
class abc    {
    private:
        int i;
    public:
        int j, k;
        abc(int aa, int bb, int cc); //constructor with
                                        //parameters aa, bb and cc
        {
            i = aa; j = bb; k = cc;
        }
        :
}
```

```

        };

int main()    {
    abc abc1(5, 8, 10);           //valid: create an instance abc1 of class abc;
    :                           //initialize i, j, and k with 5, 8, and 10
                                //respectively

    abc abc2(100, 200);          //invalid: less number of arguments

    abc abc3();                  //invalid: default constructor not available

}

```

Evidently, with parametrized constructor, the correct number and valid argument values must be passed to it at the time of object instantiation. This can be done in two different ways in C++.

Implicit call: calling the constructor without mentioning its name.

Explicit call: calling the constructor by mentioning its name explicitly.

Example:

```

class abc    {
    private:
        int i;

    public:
        int j, k;

        abc(int aa, int bb, int cc); //constructor with
                                    //parameters aa, bb and cc

        {
            i = aa; j = bb; k = cc;
        }

        :

        :

};

int main()    {
    abc abc1(5, 8, 10);           //implicit call: constructors has not
                                //been called by its name

    :

    abc abc2 = abc(100, 200, 300); //explicit call to the constructor of
                                //abc

    :

}

```

Implicit call to the constructor also allows one to create a temporary instance of a class that and the object thus created does not have any name. It is anonymous. It exists in the memory as long as its member(s) is(are) executing after which it is destroyed.

Example:

```
class abc      {
    private:
        int i;

    public:
        int j, k;
        abc(int aa, int bb, int cc); //constructor with
                                     //parameters aa, bb and cc

        {
            i = aa; j = bb; k = cc;
        }

        void show()
        {
            cout<<i<<j<<k;
        }

        :
        :

    };

int main()      {
    abc(5, 8, 10).show();           //create a temporary instance, initialize the
                                   //members

    :                               //with 5, 8 and 10 and execute the function show.

    :                               //when the execution is over destroy the instance

}
```

C++ classes are derived data types and so they have constructor(s). Similarly, the primitive data types also have constructors. If the programmer does not supply the argument, default constructor is called else the value supplied by the programmer is used to initialize the variables of that data type.

```
int aa, bb, cc;           //default constructor used

int aa(5), k(89);         //aa is initialized with 5 and k with 89

float xx(4.3)              //xx is initialized with 4.3
```

Copy constructors

A copy constructor is a constructor of the form `classname (classname &)`. The compiler will use the copy constructor whenever you initialize an instance using values of another instance of same type.

Example:

```
student st1;           // default constructor used

student st2 = st1;     // copy constructor used
```

Copy constructor is called whenever an instance of same type is assigned to another instance of the same class. The copy constructor copies the data contents of the instance being assigned to the other instance member by member. If a copy constructor is not defined explicitly, the compiler automatically creates it and it is public.

However, the programmer may define his/her own copy constructor for the class in which case the default copy constructor becomes unavailable. A copy constructor takes a reference to an object of the same class as argument.

Example:

```
class student {

int rollno;

float marks;

public :

    student (int a, float b)           // constructor

    { rollno = a; marks = b }

    student(student &s )               // copy constructor

    {

        rollno = s.rollno;

        marks = s.marks + 5;

    }

    :

    :

};
```

These constructors may be used as follows:

```
student s1(5, 78.5);           //constructor used to initialize s1.rollno to
                               //5 and s1.marks to 78.5

student s2(s1);                //copy constructor used to initialize
                               //s2.rollno to 5 and s2.marks to
                               //s1.marks+5, i.e. 78.5+5=83.5

student s3 = s1;               //copy constructor used to initialize
                               //s3.rollno to 5 and s3.marks to
                               //s1.marks+5, i.e. 78.5+5=83.5
```


Default argument in constructors

If a constructor is called with less number of arguments than required an error occurs. However, it is desired that when an argument is missing in a constructor call, then a default value should be assigned.

Example:

```
class student    {

                int rollno;

                float marks;

        public:

                student(int aa, int bb = 10.0)

                { rollno = aa; marks = bb;}

                :

        };

        student s1(10, 70.0);    //constructor will initialize s1.rollno with 10 and
                                //s1.marks with 70.0

        student s2(11); //constructor will initialize s2.rollno with
                        //11 and s2.marks with default 10.0
```

Order of invocation of constructors

Every time an object is created its constructor is invoked. In case where multiple objects are being created, then the constructors are invoked in a definite order. Consider the following statement.

```
student s1, s2, s3;
```

In this case, constructors of s1.student(), s2.student(), s3.student() are invoked in that order. But consider the following program.

Example:

```
#include <iostream.h>

#include <conio.h>

class subject    {

                int days;

                int subjectno;

        public:

                subject(int d = 111, int sn = 100); //function prototype

                void printsub(void)

                {

                        cout<<"Subject No:"<<subjectno;

                        cout<<"\n"<<"Days:"<<days<<"\n";

                }

        };

};
```

```

subject::subject(int d, int sn)    //constructor definition
{
    cout<<"Constructing Subject\n";
    days = d;
    subjectno = sn;
}

class student {
    int rollno;
    float marks;
public:
    student()
    {
        cout<<"Constructing Student\n";
        rollno = 0;
        marks = 0.0;
    }
};

class admission {
    subject sub;
    student stud;
    float fees;
public:
    admission()    //constructor
    {
        cout<<"Constructing admission\n";
        fees = 0.0;
    }
};

int main()
{
    clrscr();
    cout<<"start creating\n";
    admission adm;
    cout<<"end creating\n";
}

```

```
        return 0;  
    }
```

The above program creates an object of class admission that creates two objects of classes subject and student one each. The output of the program is:

```
Start creating  
Constructing subject  
Constructing student  
Constructing admission  
End creating
```

Evidently, the constructors have been called in the order of:

```
Subject  
Student  
Admission
```

Therefore, the constructors are called in such a way that the contained objects are created first and then the containing object.

Student Activity 1

1. Define Constructor?
2. What is the use of a constructor?
3. How will you declare a constructor?
4. What are default constructors?
5. What are parameterized constructors?
6. Define copy constructors? When does it called.
7. In which order are the constructors invoked.

Dynamic initialization of objects

When the initialization takes place at run time instead of compile time, it is known as dynamic initialization.

Example:

```
class student    {  
    int rollno;  
    float marks;  
public:  
    student(int a; float b) { rollno = a; marks= b }  
    :  
};  
  
int main()  
{  
    int i;  
    float j;
```

```

        cout<<"enter roll no.:";

        cin>>i;

        cout<<"\nenter marks:";

        cin>>j;

        student s1(i,j);          //object initialized at run time, i.e. dynamic
                                   //initialization

        :

    }

```

Dynamic initialization offers flexibility to the programmers in initializing the objects at run time rather than at compile time.

Characteristics of constructors

Constructors have certain specific characteristics. These are:

1. Construction functions are invoked automatically when the objects are created.
2. If a class has a constructor, each object of that class will be initialized before any use is made of the object.
3. Constructor functions obey the usual access rules. That is *private* and *protected* constructors are available only for member and friend functions, however, *public* constructors are available for all the functions. Only the functions that have access to the constructor of a class can create an object of the class.
4. No return type (not even void) can be specified for a constructor.
5. They cannot be inherited, though a derived class can call the base class constructor.
6. A constructor may not be *static*.
7. Default constructors and copy constructors are generated (by the compiler) where needed. Generated constructors are *public*.
8. Like other C++ functions, constructors can also have default arguments.
9. It is not possible to take the address of a constructor.
10. An object of a class with a constructor cannot be a member of a union.
11. Member functions may be called from within a constructor.
12. The default and copy constructors are generated by compiler only when not defined by the programmer.

Destructors

Just as objects are created, they are destroyed. The function that is automatically called when an object is no more required is known as *destructor*. It is also a member function very much like constructors but with an opposite intent. Its name is same as that of the class but is preceded by tilde (~). The destructor for class **student** will be **~student()**. A destructor takes no argument and returns no values.

Example:

```

class student    {

                int rollno;

                float marks;

        public:

                student()

```

```
        {  
            cout<<"Constructing Student\n";  
            rollno = 0;  
            marks = 0.0;  
        }  
        ~student()  
        {  
            cout<<"\nDestroying Student";  
        }  
    };  
  
int main()  
{  
    student s1;  
}
```

The output will be:

Constructing Student

Destroying Student

During construction of an object by the constructor, resources may be allocated for use. For example, a constructor may have opened a file and a memory 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. Whose responsibility is this? A destructor performs this cleaning-up task. Therefore, destructor is as useful as constructor.

Student Activity 2

1. What do you mean by dynamic initialization of an object? Give an example to explain it.
2. What are the characteristics of constructors?
3. Define destructors.
4. Write an example Program to explain the concept of destructor.

3.3 CONSTRUCTORS OF THE STRING CLASS

There are six overloaded constructors which are as follows:

- Default constructor
- Constructor using an array
- Constructor using part of an array
- Copy constructor
- Constructor using n copies of a character
- Constructor using range

Syntax

```
string ()
```

The function creates a default string object of zero size.

Example

```
#include <iostream.h>

#include <string.h>

int main()

{

string obj;

cout << "obj [0] contains :" << obj[0] << endl;

{if (obj[0] == '\0')

{

cout << "empty" << endl;

else

{

cout << "NOT empty" << endl;

return 0;

}

}
```

The output of the program is:

```
obj[0] contains:
empty
```

Constructor Using an Array

Syntax

```
String(const char *arr)
```

The function creates a string object and initializes it with the elements of an array. The array must terminate with a NULL character.

Constructor Using Part of an Array

Syntax

```
String(const char *arr, int size)
```

The function creates a string object and initializes it with the elements of an array. The array must terminate with a NUL character. Consider the syntax shown above. This constructor initializes the string object with the elements of the array, arr, and continues for size characters even if they exceed the size of the array, arr.

Example

```
#include <iostream.h>
```

```
#include <string.h>

int main()
{
    char arr[21] = "All decks Red Alert!";
    string obj1 (arr, 11);
    cout << "obj1 contains :" << obj1 << endl;
    string obj2(arr, 28);
    cout << "obj2 contains :" << obj2 << endl;
    return 0;
}
```

The output of the program shown above is:

```
obj1 contains : All decks R
obj2 contains : All decks Red Alert ! xÿÿ;3@
```

In the above program, the object, obj1, is initialized with the first eleven elements of the array, arr. The object, obj2, is initialized with the first 28 elements from the array, arr. Since the array does not have 28 elements, all the elements in the array are copied into the object, obj2. The remaining are junk characters.

Copy Constructor

Syntax

```
String (source_string_object, [begin_location], [maximum_number_of_
characters_to_copy])
```

The function creates a string object and initializes it with the contents of another string object called source_string_object.

The optimal second argument, begin_location, specifies a location in the source string object from which to begin copying. The position numbers begin with 0.

The optional third argument specifies the maximum number of characters to copy from the source string object. If there are fewer characters in the source string object than the number specified in the third argument, this constructor copies only till the end of the source string object.

Example

```
#include <iostream.h>
#include <string.h>

int main()
{
    string source ("wrap factor two");
    string obj1(source);
    cout << "obj1 contains:" << obj1 << endl;
    string obj2(source, 6);
```

```

        cout << "obj2 contains: " << obj2 << endl;

        string obj3(source, 6,3);

        cout << "obj3 contains:" << obj3 << endl;

        string obj4(source, 6, 20);

        cout << "obj4 contains: " << obj4 << endl;

        return 0;

    }

```

The output of the program shown above is:

```

Obj1 contains : wrap factor two

Obj2 contains : factor two

Obj3 contains : fact

Obj4 contains : factor two

```

In the above program, the object, obj1, is initialized with all the characters of the string object, source. The object, obj2, is initialized with all the characters from source, starting with the character at position six. The elements of source are numbered, starting with 0. The object, obj3, is initialized with three characters from source, starting with the character at position six. The object, obj4, is initialized with all the characters from source, starting with the character at position six.

Constructor Using n Copies of a Character

Syntax

```
string(int n, char)
```

The code creates a string object and initializes it with n copies of the character specified in the second argument.

Example

```

#include <iostream.h>

#include <string.h>

int main()
{
    string obj(5, 'G');

    cout << "obj contains :" << obj << endl;

    return 0;

}

```

The output of the program shown above is:

```
obj contains: GGGGG
```

Constructor Using a Range

Syntax

```
string(begin_location, end_location)
```


The code creates a string object and initializes it with the elements of a source array or a string object. The first parameter, `begin_location`, points to the element in the source at which copying begins. The second parameter, `end_location`, points to one past the last location to be copied.

Example

```
#include <iostream.h>

#include <string.h>

int main()

{

    char arr[] = "Space, the final frontier.";

    string obj1 (arr +2, arr + 9);

    cout << "obj1 contains :" << obj1 << endl;

    string source ("These are the voyages");

    string obj2(&source[3], &source [8]);

    cout << "obj2 contains:" << obj2 << endl;

    return 0;

}
```

The output of the program shown above is:

```
obj1 contains : ace, th
obj2 contains : se ar
```

3.4 STRING CLASS ASSIGNMENT

There are three overloaded assignment methods.

Syntax

```
string& operator = (const string & obj)
```

This code assigns the contents of one string object to another.

Example

```
string obj1 ("A Planet in Danger");

string obj2;

obj2 = obj1;

cout << obj2; // Displays: A Planet is Danger
```

Syntax

```
string& operator = (const char * ptr)
```

This assigns the contents of a constant array to a string object.

Example

```
string obj;
```

```
obj = "energize";
cout << obj; // Displays: Energize
```

Syntax

```
string& operator = (char var)
```

This assigns a single character to a string object.

Example

```
string obj;
obj = 'G';
cout << obj; // Displays:G
```

This string class also provides the overloaded method, assign().

Syntax

```
string& assign (const string& obj)
```

This assigns the contents of one string object to another.

Example

```
string obj1 ("Metamorphosis");
string obj2;
obj2.assign(obj1);
cout << obj2; // Displays: Metamorphosis
```

Syntax

```
string& assign (const string &obj, int from_position, int number_of_elements)
```

This assigns a part of the contents of the string object, obj.

Example

```
string obj1 ("phaser set to stun");
string obj2;
obj2.assign(obj, 7,5);
cout << obj2; // Displays:set t
```

Syntax

```
string& assign (const char *ptr, int number_of_elements)
```

This assigns a part of the contents of the const array ptr.

Example

```
string obj;
obj.assign("Activate the shields!", 7);
cout << obj; // Displays: Activate
```

Syntax

```
string& assign (const char *ptr)
```

This assigns the entire contents of the const array, ptr to the string object.

Example

```
string obj;

obj.assign ("We mean no harm!");

cout << obj; // Displays: We mean no harm!
```

Syntax

```
string& assign (int n, char var)
```

This assigns n identical characters, specified by the second parameter, to the string object.

Example

```
string obj;

obj.assign(5, 'H');

cout << obj; // Displays:HHHHH
```

Resizing an Object of the string Class

The member function, `resize()`, is used to change the size of a string object.

Syntax

```
void resize (int n, [char])
```

The increased size of the string object is n elements. If n is greater than the existing size, the function will pad the string with the NUL character. If the optional second parameter is specified and the new size of the string is greater than the existing size, the string will be padded with the character specified in the second parameter.

The member function, `size()`, returns the number of elements in the string object.

The member function, `erase()`, removes all the characters from a string object.

The member function, `empty()`, returns true if `size() == 0`.

Example

```
#include <iostream.h>

#include <string.h>

int main()

{

    string obj("Beam me up Scotty!");

    cout << "obj contains :" << obj << endl;

    cout << "Number of elements : " << obj.size();

    obj.resize(28, '!');

    cout << endl << "obj.resize(28, '!)" << endl;

    cout << "obj contains :" << obj << endl;

    cout << "Number of elements : " << obj.size();

    obj.resize(7, '!');
```

```

        cout << endl << "obj.resize(7, '!)" << endl;

        cout << "obj contains : " << obj << endl;

        cout << "Number of elements : " << obj.size();

        obj.resize(10);

        cout << endl << "obj.resize(10)" << endl;

        cout << obj << endl;

        cout << "Number of elements : " << obj.size();

        obj.erase();

        cout << endl << "obj.erase()" << endl;

        cout << "Number of elements : " << obj.size();

        if (obj.empty())

            cout << endl << "obj IS EMPTY" << endl;

        else

            cout << endl << "obj : " << obj << endl;

        return 0;

    }

```

The output of the program shown above is:

```

obj contains : Beam me up Scotty!

Number of elements : 18

obj.resize(28, '!)

obj contains : Beam me up Scotty!!!!!!!!!!!!

Number of elements : 28

obj.resize(7, '!)

obj contains : Beam me

Number of elements : 7

obj.resize (10)

obj contains : Beam me

Number of elements : 10

obj.erase()

Number of elements : 0

obj IS EMPTY

```

3.5 STRING ACCESS OPERATOR AND METHOD

The operator, [], of the string class can be used to access an individual element of a string using the array notation. It can be used to retrieve or alter the value.

Syntax

Operator[] (int pos)

Example

```
#include <iostream.h>

#include <string.h>

int main()

{

    string obj ("Aye Aye Sir!");

    cout << "obj [2] :" << obj[2] << endl;

    obj[0] = obj [4] = 'B';

    cout << "obj contains :" << obj << endl;

    cout << "Number of elements : "<<obj.size() << endl;

    cout << "obj[30] :" << obj[30] << endl;

    obj[30] = 'G';

    cout << "obj[30] :" << obj[30] << endl;

    cout << "Now obj is :" << obj << endl;

    cout << "Number of elements : "<<obj.size() << endl;

    return 0;

}
```

The output of the above program is:

```
obj[2] :e

obj contains : Bye Bye Sir!

Number of elements : 12

Obj[30] :

Obj[30] :G

Now obj is : Bye Bye Sir!

Number of elements :12
```

The operator, [], will return an undefined value, if the subscript is greater than the number of elements in the string object.

The function, substr(), returns the substring of a string object.

Syntax

string substr(int pos = 0, int n = string::npos)

It returns a string, that is, a copy of the contents of the string object, starting at the position, pos, and continuing for n characters or to the end of the string.

Example

```
#include <iostream.h>
```

```
#include <string.h>

int main()
{
    string obj("Doctor McCoy, report to Sick-bay");

    cout << "obj is      :" << obj << endl;

    cout << "obj.substr()  :"

        << obj.substr() << endl;

    cout << "obj.substr(7) << endl;

    cout << "obj.substr(7, 5) :"

        << obj.substr(7, 5) << endl;

    cout << "obj.substr(7, 50) :"

        << obj.substr(7,50) << endl;

    return 0;
}
```

The output of the above program is:

```
obj is      : Doctor McCoy, report to Sick-bay
obj.substr()    : Doctor McCoy, report to Sick-bay
obj.substr(7)    : McCoy,report to Sick-bay
obj.substr(7, 5) : McCoy
obj.substr(7, 50) : McCoy, report to Sick-bay
```

3.6 STRING COMPARISON OPERATORS

The non-member comparison functions are overloaded, relational operator is overloaded so that it can compare a string object to another, a string object to a character array, and a character array to a string object.

Syntax

```
operator==( )
operator<( )
operator<=( )
operator>( )
operator>=( )
operator!=( )
```

Example

```
#include <iostream.h>

#include <string.h>

int main()
```

```
{  
  
    string obj1 ("enterprise");  
    string obj2("Enterprise");  
    cout << endl << (obj1 == obj2); // Displays 1  
    obj2 = "enterprise";  
    cout << endl << (obj1 < obj2); // Displays 1  
    cout << endl << (obj1 > obj2); // Displays 0  
    cout << endl << (obj1 >= obj2); // Displays 0  
    cout << endl << (obj1 <= obj2); // Displays 1  
    cout << endl << (obj2 != obj1); // Displays 1  
    cout << endl << ("Enterprise" == obj1); // Displays 1  
    cout << endl << (obj2 == "enterprise"); // Displays 1  
    char arr[] = "enterprise";  
    cout << endl << (arr == obj2); // Displays 1  
    cout << endl << (obj1 == arr); // Displays 0  
    char * ptr = new char[15];  
    strcpy(ptr, "Enterprise");  
    cout << endl << (ptr == obj2); // Displays 0  
    cout << endl << (obj1 == ptr); // Displays 1  
    return 0;  
}
```

3.7 APPENDING TO A STRING OBJECT

The overloaded operator, +=, can be used to append a string object, a character array, or an individual character to another string object.

Syntax

string& operator += (const string& obj)

string& operator += (const char * ptr)

string& operator += (char var)

Example

```
#include <iostream.h>  
  
#include <string.h>  
  
int main()  
{  
  
    string obj1 ("These are");
```

```

    string obj2 ("the voyages");
    char arr[] = "of the spaceship ";
    char var = 'E';
    obj1 += obj2;
    obj1 += arr;
    obj1 += var;
    char *ptr = new char[20];
    strcpy(ptr, "enterprise");
    obj1 += ptr;
    cout << "obj1 :" << obj1 << endl;
    return 0;
}

```

The output of the program shown above is:

obj1 : These are the voyages of the spaceship Enterprise

The append() function can be used to append a string object, a character array, or an individual character to another string object.

Syntax

```
string& append(const string& obj)
```

```
string& append(const string& obj, int starting_pos, int number_of_characters)
```

Example

```

#include <iostream.h>
# include <string.h>
int main()
{
    string obj1 ("My name is Spock, ");
    string obj2("I am a Vulcan.");
    obj1.append(obj2);
    cout << obj1; // My name is Spock, I am a Vulcan.
    string obj3;
    obj3.append(obj1, 11, 5);
    cout << obj3; // Spock!!
    char arr[] = " fascinating!";
    obj3.append(arr, 8);
    cout << obj3; // spock!! Fascina return 0;
}

```

3.8 STRING SEARCHING

The find() functions in the string class allow you to find the position of a substring or a character in a string object.

The find() function

This overloaded function can be used to search for a single character or a string in a string object.

Syntax

```
int find(const string& obj, int position = 0) const;
int find(const char * arr, int position = 0) const;
int find(const char * arr, int position, int number_of_characters) const;
int find(char var, int position = 0) const;
```

Example

```
#include <iostream.h>
#include <string.h>

int main()
{
    string source ("Strange and new worlds");
    string sub("an");
    int pos1 = source.find(sub);
    cout << "The substring \"an\" is " << pos2;
    int pos3 = source.find(sub, pos2 + 1);
    if (pos3 == string::npos)
        cout << endl << "No other occurrences of the substring." << endl;
    pos3 = source.find("new", 2);
    cout << "The first occurrence of \"ne\" is at" << pos3 << endl;
    int loc1 = 0, loc2 = 0, count = 0;
    for (loc2 = source.find('r', loc1); loc2 != string::npos;)
    {
        cout << "The character \"r\" occurs " << count << " times" << endl;
        count++;
        loc1 = loc2 + 1;
        loc2 = source.find('r', loc1);
    }
    return 0;
}
```

The output of the above program is:

The substring "an" is at position 3

The next position of the substring "an" is 8

No other occurrences of the substring.

The first occurrence of "ne" is at 12

Constructors and Destructors

The character 'r' occurs 2 times

The find() Function

This overloaded function is similar to the find() function except that it finds the last occurrences of a string or a character that starts at or before position.

Syntax

```
int find(const string& obj, int position = 0) const;
```

```
int find(const char * arr, int position = 0) const;
```

```
int find(char var, int position = 0) const;
```

Example

```
#include <iostream.h>

#include <string.h>

int main()
{
    string source ("Strange and new worlds");
    string sub("an");
    int pos1 = source.find(sub);
    cout << "The substring \"an\" is at position " << pos1 << endl;
    int pos2 = source.find(sub, pos1 - 1);
    cout << "The previous position of the substring \"an\" is"
    << pos2;
    int pos3 = source.find(sub, pos2 - 1);
    if (pos3 <= 0)
        cout << endl << "No other occurrences of the substring." << endl;
    int loc1 = 0, loc2 = 0, count = 0;
    for (loc2 = source.find('r'); loc2 != string::npos && loc2 >= 0;)
    {
        ++count;
        loc1 = loc2 - 1;
        loc2 = source.find('r', loc1);
    }
    cout << "The character 'r' occurs " << count << "times" << endl;
    return 0;
}
```

The output of the previous program is:

The substring "an" is at position 8

The previous position of the substring "an" is 3

No other occurrences of the substring

The character 'r' occurs 2 times

The Replace() Function

This overloaded function replaces an occurrence of a search string with a replace string.

Syntax

```
string& replace(int position_in_source_str, int len_of_search_string, const string &
replace_str);
```

```
string& replace(int position_in_source_str, int len_of_search_string, const char *
replace_str);
```

The following program replaces all occurrences of the string, "at" with "rand".

Example

```
#include <iostream.h>
#include <string.h>

int main()
{
    string src = "THE FAT CAT SAT ON THE MAT";
    string findstr = "AT";
    string replace_str = "RAND";
    int position_in_source = src.find(findstr);
    cout << "The source string before replacement : " << src << endl;
    while (position_in_source != string::npos)
    {
        src.replace (position_in_source, findstr.length(), replace_str);
        position_in_source = src.find(findstr);
    }
    cout << "The source string after replacement : " << src << endl;
    return 0;
}
```

The output is:

The source string before replacement : THE FAT CAT SAT ON THE MAT

The source string after replacement : THE FRAND CRAND SRAND ON THE MRAND

The string member functions, c_str() and begin(), return a pointer to the first character in the string object. If the string object is modified after calling these functions, the pointer is rendered invalid.

Student-Activity 3

1. List various constructors of string class.
2. Give the syntax of copy constructor.
3. Give an example to describe constructor using n copies.
4. Give various overloaded assignment methods of string.
5. What is string access operator and method?
6. How will you append to a string object to another string object?
7. Describe find () and replace () functions.

3.9 SUMMARY

A constructor is a member function of a class, having the same name as its class and which is called automatically each time an object of that class is created. It is used for initializing the member variables with desired initial values. A variable (including structure and array type) in C++ may be initialized with a value at the time of its declaration. The responsibility of initialization may be shifted, however, to the compiler by including a member function called constructor.

A class constructor, if defined, is called whenever a program creates an object of that class. Constructors are public member functions unless otherwise there is a good reason against.

A constructor may take argument(s). A constructor taking no argument(s) is known as default constructor. A constructor may also have parameter(s) or argument(s), which can be provided at the time of creating an object of that class.

C++ classes are derived data types and so they have constructor(s). Copy constructor is called whenever an instance of same type is assigned to another instance of the same class. If a constructor is called with less number of arguments than required an error occurs. Every time an object is created its constructor is invoked.

The function that is automatically called when an object is no more required is known as destructor. It is also a member function very much like constructors but with an opposite intent.

3.10 KEYWORD

Constructor: A member function of a class, having the same name as its class and which is called automatically each time an object of that class is created.

Default constructors: A constructor may take arguments. A constructor taking no arguments is known as default constructor.

Implicit call: Calling the constructor without mentioning its name.

Explicit call: Calling the constructor by mentioning its name explicitly.

Copy constructors: A constructor of the form class name. The compiler will use the copy constructor. Whenever you initialize an instance using values of another instance of same type.

Dynamic initialization: When the initialization takes place at run time instead of compile time, it is known as dynamic initialization.

Destructors: The function that is automatically called when an object is no more required is known as destructor.

3.11 REVIEW QUESTIONS

1. Write down programs to show the working of following functions:
 - (a) Replace function
 - (b) Find function
2. Briefly explain the following constructors with the help of example:
 - a. Default constructor
 - b. Copy Constructor
 - c. Constructor
3. Describe various overloaded assignment methods of string class.
4. Describe the use of string access operator and method. Give an example to explain it.
5. Fill in the blanks:
 - (a) _____ is having same name as its class.
 - (b) A _____ is called whenever a program creates an object of that class.
 - (c) A constructor taking no argument is called _____.
 - (d) _____ initialization takes place at run time.
 - (e) _____ returns the number of elements in the string object

3.12 FURTHER READINGS

Herbert Schildt, *The complete Reference-C++*, Tata Mc Graw Hill

Robert Lafore, *Object Oriented Programming in Turbo C++*, Galgotia Publications.

E. Balagurusamy, *Object Oriented Programming through C++*, Tata McGraw Hill.

UNIT

4

IMPLEMENTING OOPs CONCEPTS

LEARNING OBJECTIVES

After studying this unit, you should be able to:

- Describe the meaning of object-oriented programming.
- Describe procedure oriented programming
- Define data abstraction, inheritance and polymorphism.
- Understand the trends of software engineering

UNIT STRUCTURE

- 4.1 Introduction
- 4.2 Objective
- 4.2 What is Object Oriented Programming?
- 4.3 Data Abstraction
- 4.4 Software Engineering Trends
- 4.5 Summary
- 4.6 Keywords
- 4.7 Review Questions
- 4.8 Further Readings

4.1 INTRODUCTION

Programming practices have evolved considerably over the past few decades. As more and more programmers gained experience problems unknown hitherto, began to surface. The programming community became evermore concerned about the philosophy that they adopt in programming and approaches they practice in program development.

Factors like productivity, reliability, cost effectiveness, reusability etc. started to become major concern. A lot of conscious efforts were put to understand these problems and to seek possible solutions. This is precisely the reason why more and more programming languages have been developed and still continue to develop. In addition to this, approaches to program development have also been under intense research thereby evolving different frame works. One such, and probably the most popular one is object oriented programming approach or simply OOP.

4.2 WHAT IS OBJECT-ORIENTED PROGRAMMING?

The term object-oriented programming (OOP) is widely used, but experts cannot seem to agree on its exact definition. However, most experts agree that OOP involves defining abstract data types (ADT) representing complex real-world or abstract objects and organizing your program around the collection of ADTs with an eye toward exploiting their common features. The term data abstraction refers to the process of defining ADTs; inheritance and polymorphism refer to the mechanisms that enable your to take advantage of the common characteristics of the ADTs - the objects in OOP. This chapter further explores these terms later.

Before you jump into OOP, take note of two points. First, OOP is only a method of designing and implementing software. Use of object-oriented techniques does not impart anything to a finished software product that the user can see. However, as a programmer while implementing the software, you can gain significant advantages by using object-oriented methods, especially in large software projects. Because OOP enables you to remain close to the conceptual, higher-level model of the real world problem you are trying to solve, you can manage the complexity better than with approaches that force you to map the problem to fit the features of the language. You can take advantage of the modularity of objects and implement the program in relatively independent units that are easier to maintain and extend. You can also share code among objects through inheritance.

The second point is that OOP has nothing to do with any programming language, although a programming language that supports OOP makes it easier to implement the object-oriented techniques. As you will see shortly, with some discipline, you can use objects in C programs.

Procedure-Oriented Programming

Before you get into OOP, take a look at conventional procedure-oriented programming in a language such as C. Using the procedure-oriented approach,

you view a problem as a sequence of things to do. You organize the related data items into C structs and write the necessary functions (procedures) to manipulate the data and, in the process, complete the sequence of tasks that solve your problem. Although the data may be organized into structures, the primary focus is on the functions. Each C function transforms data in some way. For example, you may have a function that calculates the average value of a set of numbers, another that computes the square root, and one that prints a string. You do not have to look far to find examples of this kind of programming – C function libraries are implemented this way. Each function in a library performs a well-defined operation on its input arguments and returns the transformed data as a return value. Arguments may be pointers to data that the function directly alters or the function may have the effect of displaying graphics on a video monitor.

4.3 DATA ABSTRACTION

To understand data abstraction, consider the file I/O routines in the C run-time library. These routines enable you to view the file as a stream of bytes and to perform various operations on this stream by calling the file I/O routines. For example, you can call `fopen` to open a file, `fclose` to close it, `fgetc` to read a character from it and `fputc` to write a character to it. This abstract model of a file is implemented by defining a data type named `FILE` to hold all relevant information about a file. The C constructs `struct` and `typedef` are used to define `FILE`. You will find the definition of `FILE` in the header file `stdio.h`. You can think of this definition of `FILE`, together with the functions that operate on it, as a new data type just like C's `int` or `char`.

To use the `FILE` data type, you do not have to know the C data structure that defines it. In fact, the underlying data structure of `FILE` can vary from one system to another. Yet, the C file I/O routines work in the same manner on all systems. This is possible because you never access the members of the `FILE` data structure directly. Instead, you rely on functions and macros that essentially hide the inner details of `FILE`. This is known as data hiding.

Data abstraction is the process of defining a data type, often called an abstract data type (ADT), together with the principle of data hiding. The definition of an ADT involves specifying the internal representation of the ADT's data as well as the functions to be used by others to manipulate the ADT. Data hiding ensures that the internal structure of the ADT can be altered without any fear of breaking the programs that call the functions provided for operations on that ADT. Thus, C's `FILE` data type is an example of an ADT.

Inheritance

Data abstraction does not cover an important characteristic of objects. Real world objects

do not exist in isolation. Each object is related to one or more other objects. In fact, you can often describe a new kind of object by pointing out how the new object's characteristics and behavior differ from that of a class of objects that already exists. This is what you do when you describe an object with a sentence such as: B is just like A, except that B has..., and B does... Here you are defining objects of type B in terms of those of type A.

This notion of defining a new object in terms of an old one is an integral part of OOP. The term inheritance is used for this concept, because you can think of one class of objects inheriting the data and behavior from another class. Inheritance imposes a hierarchical relationship among classes in which a child class inherits from its parent. In C++ terminology, the parent class is known as the base class, the child is the derived class.

Multiple Inheritance

A real world object often exhibits characteristics that it inherits from more than one type of object. For instance, on the basis of eating habits, an animal maybe classified as a carnivore; other ways of classification place it in a specific family, such as the bear family. When modeling a corporation, you may want to describe a technical manager as someone who is an engineer as well as a manager. An example from the programming world is a full-screen text editor. It displays a block of text on the screen and also stores the text in an internal buffer so that you can perform operations such as insert a character and delete a character. Thus, you may want to say that a text editor inherits its behavior from two classes: a text buffer class and a text display class that, for instance, rt~nages an 80-character by 25-line text display area.

These examples illustrate multiple inheritance – the idea that a class can be derived from more than one base class. Many object-oriented programming languages do not support multiple inheritance, but C++ does.

Polymorphism

In a literal sense, polymorphism means the quality of having more than one form. In the context of OOP, polymorphism refers to the fact that a single operation can have different behavior in different objects. In other words, different objects read

differently to the same message. For example, consider the operation of addition. For two numbers, addition should generate the sum. In a programming language that supports OOP, you should be able to express the operation of addition by a single operator, say, +. When this is possible, you can use the expression $x+y$ to denote the sum of x and y , for many different types of x and y – integers, floating-point numbers, and complex numbers, to name a few. You can even define the + operation for two strings to mean the concatenation of the string.

Similarly, suppose a number of geometrical shapes respond to the message, draw. Each object reacts to this message by displaying its shape on a display screen. Obviously, the actual mechanism for displaying the object differs from one shape to another, but all shapes perform this task in response to the same message.

Polymorphism helps by enabling you to simplify the syntax of performing the same operation on a collection of objects. For example, by exploiting polymorphism, you can compute the area of each geometrical shape in an array of shapes with a simple loop like this:

```
/* Assume "shapes" is an array of shapes (rectangles, circles,
 * and son on) and "compute_area) is a function that computes
 * the area of a shape
 */
for (i = 0; i < number_of_shapes; i++)
    area_of_shape = shapes[i].compute_area();
```


This is possible because regardless of the exact geometrical shape, each object supports the `compute_area` function and computes the area in a way appropriate for that shape.

Student Activity 1

1. What is object-oriented programming?
2. What is procedure-oriented programming?
3. What do you mean by data abstraction
4. Define Inheritance.
5. What is multiple Inheritance
6. Define Polymorphism.

4.4 SOFTWARE ENGINEERING TRENDS

Books on software engineering show the traditional lifecycle of software development in the form of a waterfall (see Figure 4.1), in which the development process follows a rigid sequence from analysis to design and to implementation and testing. Based on that waterfall process, here is an oversimplified view of the sequence of activities that constitute software development.

1. You begin development with the analysis phase by analyzing what the software must do and arriving at a complete, detailed description of the software's behavior in response to the possible set of inputs. You work with potential users of the software to find a definitive answer to the question: What does the software do? The result of this step is a set of requirements for the software.

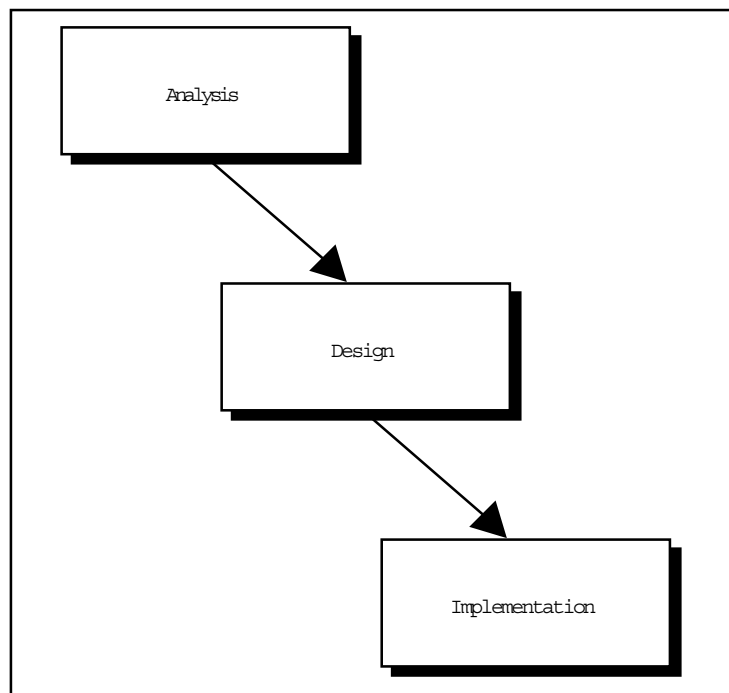


Figure 4.1

2. Next comes the design phase, during which you decide how to do what the user wants. Here your goal is to map the user's real world description of the software into algorithms and data that can be implemented in a programming language. Typically, you might follow a top-down approach and repeatedly decompose the software's functions into a sequence of progressively simpler functions that are eventually implemented in the implementation phase.
3. In the implementation phase you define the data structures and write the code to implement the functions that were identified in the design. Finally, you have to test the software to verify that it conforms to the original specification as much as possible.

Although there is no single, clear-cut, step-by-step approach to designing software, there are well-known methods and tools for accomplishing the analysis and design phases of the process.

Among the analysis methods, the most popular is structured analysis, attributed to Tom DeMarco, who built on prior work by Ed Yourdon and Larry Constantine. Structured analysis is concerned with the way data flows through the system. It generates a dataflow diagram (DFD) and two textual descriptions: a data dictionary and a minispecification. The DFD is a diagramming notation that depicts the flow of data through the system and identifies the processes that manipulate the data. The data dictionary describes the data shown in the DFD, whereas the minispecification describes, in plain English, how the data is processed by each process. As an example, consider the problem of querying a database and printing a sorted list of the items retrieved by the database. Figure 1.6 shows a grossly simplified data flow diagram for this task. The diagram shows the major processes (functions), with arrows indicating the data being passed between functions.

Another type of diagram for representing a software system is the structure chart introduced by Larry Constantine in the 1970s as a replacement for flowcharts of earlier years. The structure charts are quite similar to the flowcharts and they provide a notational means to represent the behavior of the system being implemented. As shown in Figure 5.2, a structure chart starts with a single top-level module represented by a rectangle. That module represents the overall function of the system. From that topmost module, arrows fan out to subordinate modules that the topmost module invokes. The subordinate modules, in turn, invoke other modules. Thus, the structure looks like an organization chart, reflecting the top-down nature of the design. Each arrow emanating from one module to another represents a function call as well as a data transfer. In an actual structure chart, a label next to each arrow identifies the data being transferred. There are other symbols as well that indicate a program's control flow, such as if statements and loops. Many commercially available computer-aided software engineering (CASE) tools support structure charts with symbols similar to those originally suggested by Larry Constantine.

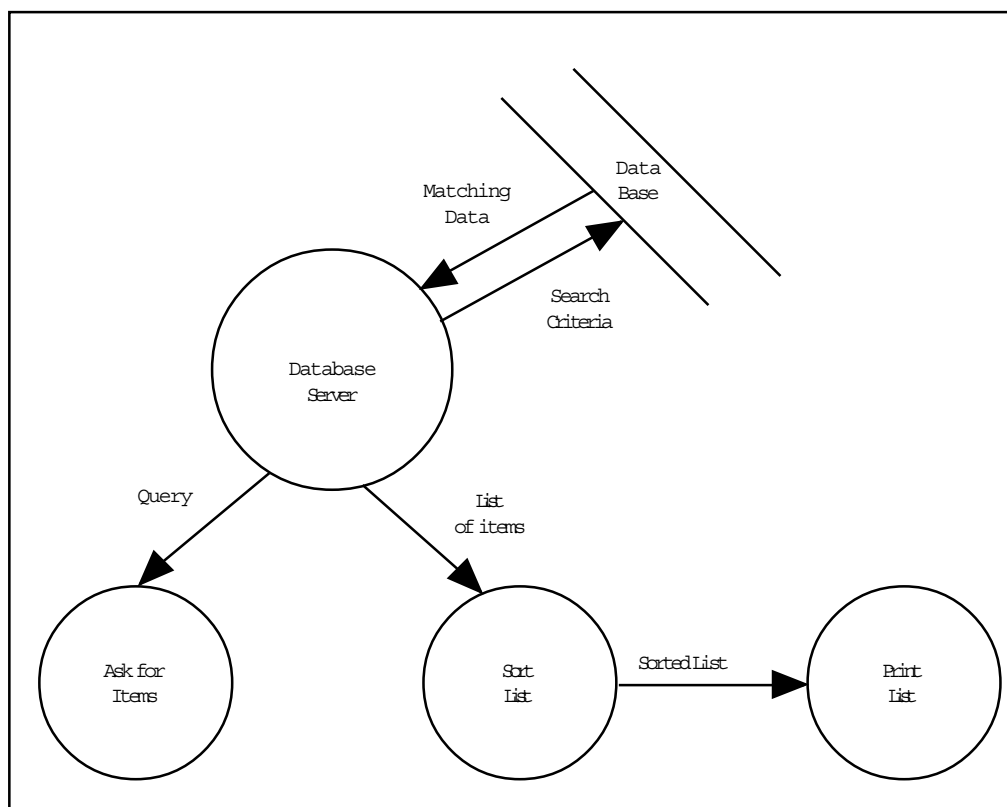


Figure 4.2

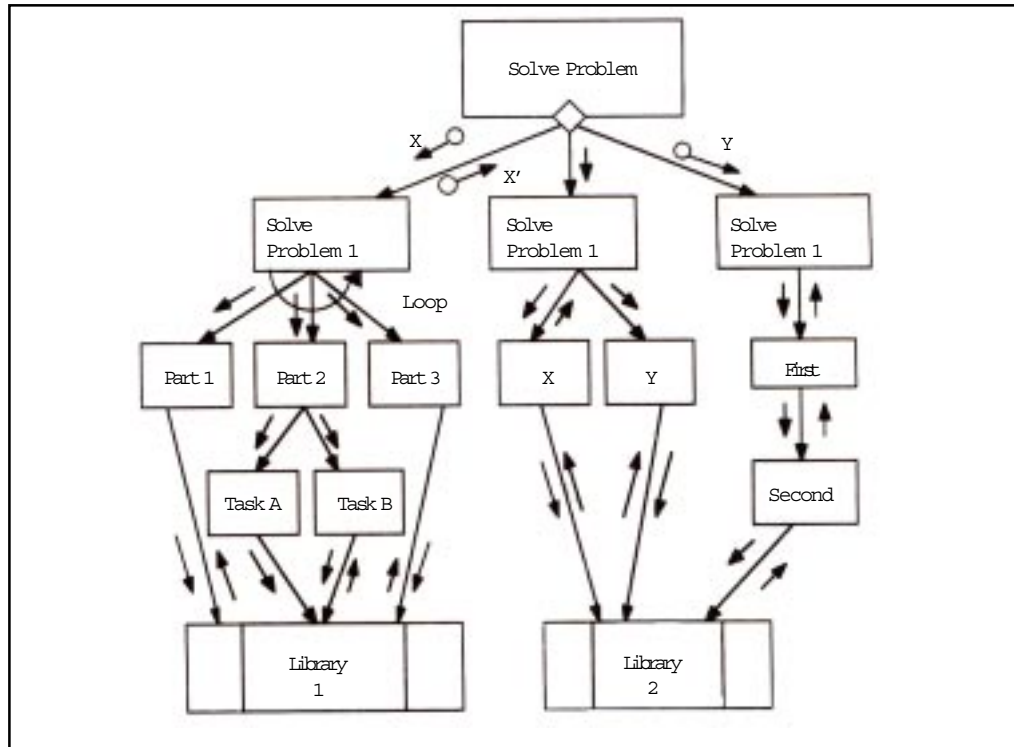


Figure 4.3: A Typical Structure Chart

The design phase of the traditional development cycle is rather loosely defined. The goal of design is to refine the data flow diagrams and map the data into data structures using facilities of the programming language (for example, struct in C) that you will use to implement the software. Procedures are then defined to implement the modules that manipulate the data. The distinction between the analysis and design phases are often blurred and, usually, there are several iterations before you arrive at a design for the software.

Student Activity 2

1. Describe waterfall model of software development.
2. Discuss the methods and tools used for accomplishing the analysis and design phases of the process.
3. What are DFDs?
4. What is the use of software engineering case tools?

4.5 SUMMARY

OOP is a method of designing and implementing software. Since OOP enables you to remain close to the conceptual, higher-level model of the real world problem, you can manage the complexity better than with approaches that force you to map the problem to fit the features of the language.

Some essential concepts that make a programming approach object-oriented are objects, classes, data abstraction, data encapsulation, Inheritance, Polymorphism, dynamic binding and message passing. The data and the operation of a class can be declared public, protected or private. OOP provides greater programmer productivity, better quality of software and lesser maintenance cost. Depending upon the features they support, they are classified as object based programming languages and object oriented programming languages.

OOP is method of designing and implementing software. Since OOP enables you to remain close to the conceptual, higher-level model of the real world problems. You can manage the complexity better than with approaches that force you to map the problem to fit the features of the language. Some essential concepts that make a programming approach object-oriented

are objects, classes, data obstruction, encapsulation, Inheritance, dynamic binding and message passing.

The traditional lifecycle of software development is a waterfall model in which the development process follows a rigid sequence from analysis to design and to implementation and testing. Although there is no single, clear-cut, step-by-step approach to designing software, there are well-known methods and tools for accomplishing the analysis and design phases of the process.

The design phase of the traditional development cycle is rather loosely defined. The goal of design is to define the data flow diagrams and map the data into data structures using facilities of the programming languages that you will use to implement the software.

4.6 KEYWORDS

Analysis: A detailed study of the various operations performed by the proposed software.

Design: The term design describes both a final software system and a process by which is developed.

Implementation: This phase is primarily concerned with coding the software design into an appropriate programming language. Testing the programs and installing the software.

Maintenance: In this phase the software is continuously evaluated and modified to suit the changes as they occur.

Object: In the object-oriented model a combination of a small amount of data and instructions about what to do with that data when the object is selected or activated.

Object-oriented programming (OOP) Language: Programming language that encapsulates a small amount of data along with instructions about how to manipulate that data; inheritance and resability features provide functional benefits.

Class: A class represents a set of related object.

Data Abstraction: The act of representing essential features without including the background details or exploitations.

Data encapsulation: The wrapping up of data and functions in a single unit (class).

Inheritance: The process by which objects of same class acquire the properties of object of another class.

Polymorphism: The ability to take more than one form.

Dynamic binding: The linking of a procedure call to the code to be executed in response to the call.

4.7 REVIEW QUESTIONS

1. Define the following terms:
 - (a) Multiple Inheritance
 - (b) Polymorphism.
2. Enumerate the benefits of OOPs.
3. Distinguish between:
 - (a) Data hiding and data abstraction
 - (b) Inheritance and Multiple Inheritance.
4. Describe activities that constitute software development.

5. Explain in brief the structure chart by Larry Constantine.
6. Fill in the blanks:
 - (a) Using_____approach, a problem can be viewed as a sequence of things to do.
 - (b) _____enable you to view file as a stream of bytes.
 - (c) _____objects do not exist in isolation.
 - (d) _____means the quality of having more than one form.
 - (e) _____model depicts the traditional life cycle of software development.

Answers to Review Questions

6.

(a) Procedure-oriented	(b) I/O routines
(c) Real world	(d) Polymorphism
(e) Waterfall	

4.8 FURTHER READINGS

Herbert Schildt, *The complete Reference-C++*, Tata Mc Graw Hill

Robert Lafore, *Object Oriented Programming in Turbo C++*, Galgotia Publications.

E. Balagurusamy *Object Oriented Programming through C++*, Tata McGraw Hill.

UNIT

5

IMPLEMENTING OOPs CONCEPTS

LEARNING OBJECTIVES

After studying this unit, you should be able to:

- Describe tokens
- Describe keywords
- Describe identifiers
- Describe basic data types of C++
- Describe user-defined data types
- Describe derived data types
- Describe symbolic constants
- Describe type compatibility
- Declare variable
- Describe operators and expressions
- Describe function overloading
- Describe streams

UNIT STRUCTURE

- 5.1 Introduction
- 5.2 Tokens
- 5.3 Basic Data Types
- 5.4 User-Defined Data Types
- 5.5 Derived Data Types
- 5.6 Symbolic Constants
- 5.7 Type Compatibility
- 5.8 Declaration of Variables
- 5.9 Dynamic Initialization Of Variables
- 5.10 Reference Variables
- 5.11 Operations and Expressions
- 5.12 Unary Operators
- 5.13 Prefix and Postfix Notations
- 5.14 Comparison Operators
- 5.15 Shift Operators
- 5.16 Shifting Positive Numbers
- 5.17 Shifting Negative Numbers
- 5.18 Bit-Wise Operators
- 5.19 Short Circuit Logical Operators
- 5.20 Conditional Operators
- 5.21 Order of Precedence of Operators

5.22	The Const Keyword
5.23	Operator and Function Overloading
5.24	Manipulation of strings using operators
5.25	Rules for overloading Operators
5.26	Function Overloading
5.27	Polymorphism
5.28	Streams
5.29	Summary
5.30	Keywords
5.31	Review Questions
5.32	Further Readings

5.1 INTRODUCTION

C++ is a language in essence. It is made up of letters, words, sentences, and constructs just like English language. This unit discusses these elements of the C++ language along with the operators applicable over them,

5.2 TOKENS

As we know, the smallest individual units in a program are known as tokens. C++ has the following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

A C++ program is written using these tokens, white spaces, and the syntax of the language. Most of the C++ tokens are basically similar to the C tokens with the exception of some additions and minor modifications.

Keywords

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements. Table 6.1 gives the complete set of C++ keywords. The keywords not found in ANSI C are shown boldface. These keywords have been added to the ANSI C keywords in order to enhance its features making it an object-oriented language.

Table 5.1: C++ Keywords

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

Identifiers refer to the names of variables, functions, arrays, classes, etc., created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

- Only alphabetic characters, digits and underscores are permitted.
- The name cannot start with a digit.
- Uppercase and lowercase letters are distinct.
- A declared keyword cannot be used as a variable name.

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, C++ places no limit on its length and, therefore, all the characters in a name are significant.

Care should be exercised while naming a variable that is being shared by more than one file containing C and C++ programs. Some operating systems impose a restriction on the length of such a variable name.

5.3 BASIC DATA TYPES

Data types in C++ can be classified under various categories as shown in Figure 5.1.

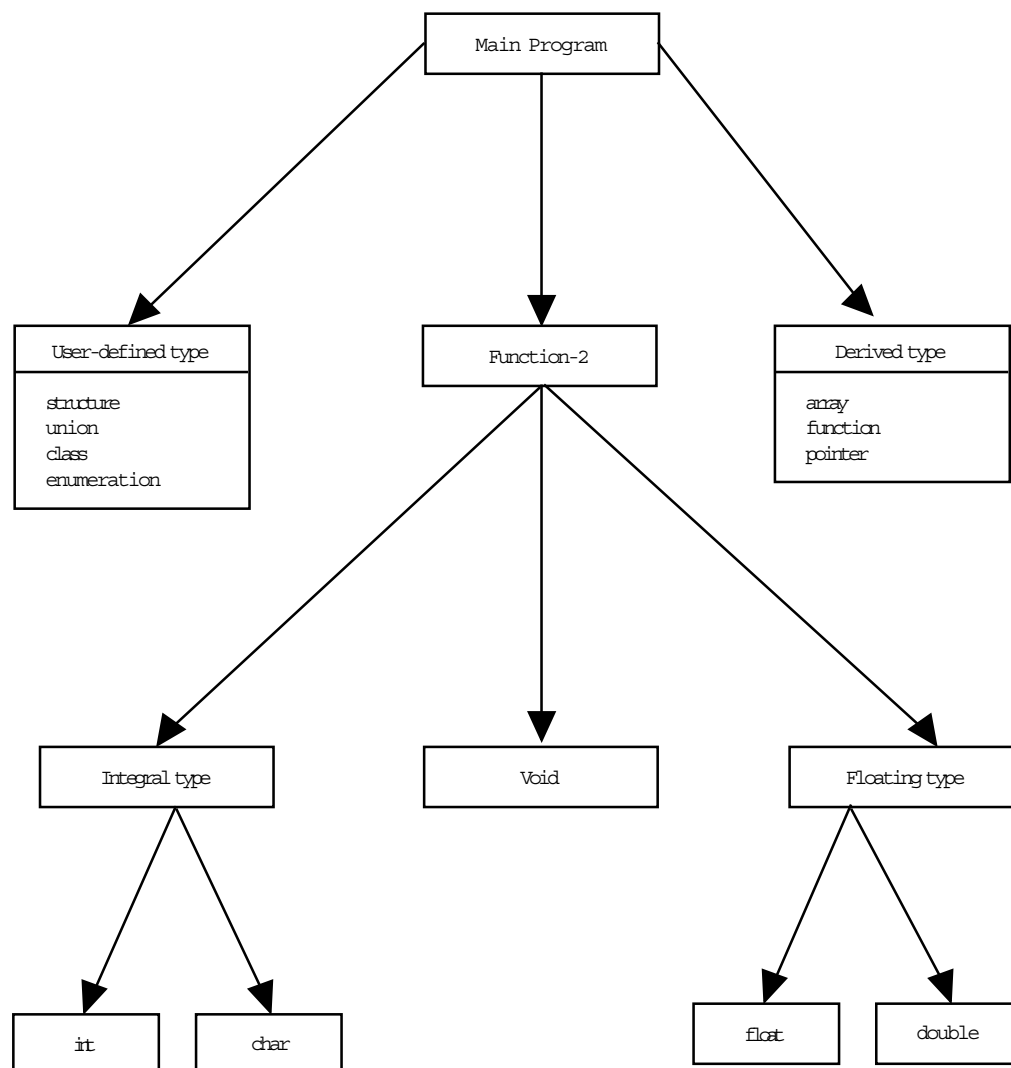


Figure 5.1: Hierarchy of C++ Data Types

Both C and C++ compilers support all the built-in (also known as basic-or fundamental) data types. With the exception of void, the basic data types may have several modifiers preceding them to serve the needs of various situations. The modifiers signed, unsigned, long, and short may be applied to character and integer basic data types. However, the modifier long may also be applied to double. Table 5.2 lists all combinations of the basic data types and modifiers along with their size and range.

Table 5.2: Size and Range of C++ Basic Data Types

Type	Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.AE - 38 to 3.4E + 38
double	8	1.7E - 308 to 1.7E + 308
long double	10	3.4E - 4932 to 1.1E + 4932

The type void was introduced in ANSI C. Two normal uses of void are (1) to specify the return type of a function when it is not returning any value, and (2) to indicate an empty argument list to a function. Example:

```
void funct1(void);
```

Another interesting use of void is in the declaration of generic pointers. Example:

```
void*gp; //gp becomes generic pointer
```

A generic pointer can be assigned a pointer value of any basic data type. But it may not be dereferenced. For example,

```
int *ip; // pointer
```

```
gp = ip; // assign int pointer to void pointer
```

are valid statements. But, the statement,

```
*ip = *gp;
```

is illegal. It would not make sense to dereference a pointer to a void value.

Assigning any pointer type to a void pointer without using a cast is allowed in both C++ and ANSI C. In ANSI C, we can also assign a void pointer to a non-pointer without using a cast to non-void pointer type. This is not allowed in C++. For example,

```
void * ptr1;
```

```
char * ptr2;
```

```
ptr2 = ptr1;
```

are all valid statements in ANSI C but not in C++. A void pointer cannot be directly assigned to other type pointers in C++.

5.4 USER-DEFINED DATA TYPES

Structures and Classes

We have used user-defined data types such as struct and union in C. While these data types are legal in C++, some more features have been added to make them suitable for object-oriented programming. C++ also permits us to define another user-defined data type known as class which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming.

Enumerated Data Type

An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The enum keyword (from C) automatically enumerates a list of words by assigning them values 0, 1, 2, and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an enum statement is similar to that of the struct statement. Examples:

```
enum shape {circle, square, triangle};

enum color {red, blue, green, yellow};

enum position {off, on};
```

The enumerated data types differ slightly in C++ when compared with those in ANSI C. In C++, the tag names shape, color, and position become new type names. That means we can declare new variables using these tag names. Examples:

```
shape ellipse; //ellipse is of type shape

color background; // background is of type color
```

ANSI C defines the types of enums to be ints. In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an int value to be automatically converted to an enum value. Examples:

```
color backgrounds blue; //allowed

color background = 7; // Error in C++

color background = (color) 7; // OK
```

However, an enumerated value can be used in place of an int value.

```
int c " red; // valid, color type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can override the default by explicitly assigning integer values to the enumerators. For example,

```
enum color {red, blue = 4, green = 8};

enum color {red s 5, blue, green};
```

are valid definitions. In the first case, red is 0 by default. In the second case, blue is 6 and green is 7. Note that the subsequent initialized enumerators are larger by one than their predecessors.

C++ also permits the creation of anonymous enums (i.e., enums without tag names). Example:

```
enum { off, on};
```

Here, off is 0 and on is 1. These constants may be referenced in the same manner as regular constants. Examples:

```
int switch_1 = off;
```

```
int switch_2 = on;
```

In practice, enumeration is used to define symbolic constants for a switch statement. Example:

```
enum shape
{
    circle,
    rectangle,
    triangle
};

main()
{
    cout << "Enter shape code:";
    int code;
    cin >> code;
    while (code >= circle && code <= triangle)
    {
        switch(code)
        {
            case circle:
                ...
                ...
                break;
            case rectangle
                ...
                ...
                break;
            case triangle:
                ...
                ...
                break;
        }
    }
}
```

```

        cout << "Enter shape code:"

        cin>> code;

    }

    cout << "BYE\n";

```

ANSI C permits an enum to be defined within a structure or a class, but the enum is globally visible. In C++, an enum defined within a class (or structure) is local to that class (or structure) only.

Student Activity 1

1. Define tokens.
2. What are keywords?
3. What are Identifiers?
4. List the basic data types available in C++.
5. What are user-defined data types?
6. What are structures and classes?
7. What are enumerated data types?

5.5 DERIVED DATA TYPES

Arrays

The application of arrays in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

```
char string[3] = "xyz";
```

is valid in ANSI C. It assumes that the programmer intends to leave out the null character `\0` in the definition. But in C++, the size should be one larger than the number of characters in the string.

```
char string[3] "xyz"; // O.K. for C++
```

Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these modifications and improvements were driven by the requirements of the object-oriented concept of C++. Some of these were introduced to make the C++ program more reliable and readable.

Pointer

Pointers are declared and initialized as in C. Examples:

```
int * ip; // Int pointer
```

```
ip = &x; // address of x assigned to ip
```

```
*ip = 10; // 10 assigned to x through indirection
```

C++ adds the concept of constant pointer and pointer to a constant.

```
char * const ptr1 = "GOOD"; // constant pointer
```

i.e., cannot modify the address that ptr1 is initialized to.

```
int const * ptr2 = &m; // pointer to a constant
```

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents what it points to cannot be changed.

We can also declare both the pointer and the variable as constants in the following way:

```
const char * const cp = "xyz";
```

This statement declares cp as a constant pointer to the string, which has been declared a constant. In this case, neither the address assigned to the pointer cp nor the contents it points to can be changed.

Pointers are extensively used in C++ for memory management and achieving polymorphism.

5.6 SYMBOLIC CONSTANTS

There are two ways of creating symbolic constants in C++:

1. Using the qualifier const.
2. Defining a set of integer constants using enum keyword.

In both C and C++, any value declared as const cannot be modified by the program in any way. However, there are/some differences in implementation. In C++, we can use const in a constant expression, such as

```
const int size = 10;  
char name[size];
```

This would be illegal in C. const allows us to create typed constants instead of having to use #define to create constants that have no type information.

As with long and short, if we use the const modifier alone, it defaults to int. For example,

```
const size = 10;
```

The named constants are just like variables except that their values cannot be changed.

C++ requires a const to be initialized. ANSI C does not require an initializer; if none is given, it initializes the const to 0.

The scoping of const values differs. A const in C++ defaults to the internal linkage and therefore it is local to the file where it is declared. In ANSI C, const values are global in nature. They are visible outside the file in which they are declared. However, they can be made local by declaring them as static. To give a const value external linkage so that it can be referenced from another file, we must explicitly define it as an extern in C++. Example:

```
extern const total = 100;
```

Another method of naming integer constants is as follows:

```
enum {X,Y,Z};
```

This defines X, Y and Z as integer constants with values 0, 1, and 2 respectively. This is equivalent to

```
const X = 0;  
const Y = 1;  
const Z = 2;
```

We can also assign values to X, Y, and Z explicitly.

```
enum {X = 100, Y = 50, Z = 200};
```

Such values can be any integer values.

5.7 TYPE COMPATIBILITY

C++ is very strict with regard to type compatibility as compared to C. For instance, C++ defines int, short int, and long int as three different types. They must be cast when their values are assigned to one another. Similarly, unsigned char, char, and signed char are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility. Otherwise, a cast must be applied.

These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Another notable difference is the way char constants are stored. In C, they are stored as ints. Therefore,

```
sizeof('x')
is equivalent to
sizeof(int)
```

in C. But, in C++, char is not promoted to the size of int and therefore,

```
sizeof('x')
equals
sizeof(char)
```

5.8 DECLARATION OF VARIABLES

We know that, in C, all variables must be declared before they are used in executable statements. This is true with C++ as well. However, there is a significant difference between C and C++ with regard to the place of their declaration in the program. C requires all the variables to be defined at the beginning of a scope. When we read a C program, we usually come across a group of variable declarations at the beginning of each scope level. Their actual use appears elsewhere in the scope, sometimes far away from the place of declaration. Before using a variable, we should go back to the beginning of the program to see whether it has been declared and, if so, of what type it is. C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use. The example below illustrates this point.

```
main ()
{
    float x;           // declaration
    float sum = 0;
    for (int i = 0; i < 5; i++) //declaration
    {
        cin >> x;
        sum = sum + x;
    }
    float average;     //declaration
    average = sum / i;
    cout << average;
}
```

The only disadvantage of this style of declaration is that we cannot see at a glance all the variables used in a scope.

Student Activity 2

1. What are Arrays?
2. Define functions?
3. Define pointer.
4. What are symbolic constants? How are they created in C++?
5. Give an example program to describe variable declaration.

5.9 DYNAMIC INITIALIZATION OF VARIABLES

One additional feature of C++ is that it permits initialization of the variables at run time. This is referred to as dynamic initialization. Remember that, in C, a variable must be initialized using a constant expression and the C compiler would fix the initialization code at the time of compilation: However, in C++, a variable can be initialized at run time using expressions at the place of declaration. For example, the following are valid initialization statements:

```
...
...
int n = strlen(string);
...
float area °= 3.14159 *rad *rad;
```

This means that both the declaration and initialization of a variable can be done simultaneously at the place where the variable is used for the first time. The following two statements in the example of the previous section

```
float average;    // declare where it is necessary
average = sum / i;
can be combined into a single statement:
float average = sum /i;  // initialize dynamically
// at run time
```

Dynamic initialization is extensively used in object-oriented programming. We can create exactly the type of object needed using information that is known only at the run time.

5.10 REFERENCE VARIABLES

C++ introduces a new kind of variable known as the reference variable. A reference variable provides an alias (alternative name) for a previously defined variable. For example, if we make the variable sum a reference to the variable total, then sum and total can be used interchangeably to represent that variable. A reference variable is created as follows:

`data-type & reference-name * variable-name`

Example:

```
float total" 100;
float & sum = total;
```

total is a float type variable that has already been declared, sum is the alternative name declared to represent the variable total. Both the variables refer to the same data object in the memory. Now, the statements

```
count << total;

and
```

```
count << sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both total and sum to 110. Likewise, the assignment

```
sum = 0;
```

will change the value of both the variables to zero.

A reference variable must be initialized at the time of declaration, This establishes the correspondence between the reference and the data object that it names. Note that the initialization of a reference variable is completely different from assignment to k.

Note that C++ assigns additional meaning to the symbol &. Here, & is not an address operator.

The notation float & means reference to float. Other examples are:

```
int n[10];
```

```
int& x = n[10]; //x is alias for n[10]
```

```
char & a = n; // initialize reference to a literal
```

The variable x is an alternative to the array element n[10]. The variable a is initialized to the new line constant. This creates a reference to the otherwise unknown location where the new line constant \n is stored.

The following references are also allowed:


- i.

```
int x;
int *p=&x;
int &m = *p;
```
- ii.

```
int & n = 50;
```

The first set of declarations causes m to refer to x which is pointed to by the pointer p and the statement in (ii) creates an int object with value 50 and name n.

A major application of reference variables is in passing arguments to functions. Consider the following:



```
void f(int & x) // uses reference
{
    x = x+10;    // x is incremented; so also m
}
main ()
{
    int m = 10;
    f(m);       // function call
    ...
    ...
}
```

When the function call f(m) is executed, the following initialization occurs:

```
Int & x s m;
```


Thus x becomes an alias of m after executing the statement
f(m);

Such functions calls are known as calls by reference whose implementation is illustrated in Figure 6.2. Since the variable x and m are aliases, when the function increments x, m is also incremented. The value of m becomes 20 after the function is executed. In traditional C, we accomplish this operation using pointers and dereferencing techniques.



Figure 6.2: Call by Reference Mechanism

The call by reference mechanism is useful in object-oriented programming because it permits the manipulation of objects by reference and eliminates the copying of object parameters back and forth. Note that the references can be created not only for built-in data types but also for user-defined data types such as structures and classes. References work wonderful well with these user-defined data types.

Student Activity 3

1. What do you mean by dynamic initialization of variables?
2. What are reference variables?
3. When does a reference variable initialized?
4. Describe call by reference mechanism.

5.11 OPERATIONS AND EXPRESSIONS

We have already seen that individual constant, variables, array elements function references can be joined together by various operators to form expressions. We have also mentioned that C includes a large number of operators which fall into several different categories. In this section include a large number of operators which fall into various different categories. In this section we examine some of these categories in details. Specifically, we will see how arithmetic operators, unary operators, relational and logical operators, assignment operators and the conditional operator are used to form expressions.

The data items that operators act upon are called operands. Some operators require two operands, while others act upon only one operand. Most operators allow the individual operands to be expressions. A few operators permit only single variables as operands.

Arithmetic Operators

There are five arithmetic operators in C. They are

Operator	Function
+	addition
-	Subtraction
*	multiplication
/	division
%	remainder after integer division

The % operator is sometimes referred to as the modulus operator.

There is no exponentiation operator in C. However, there is a library function (pow) to carry out exponentiation.

The operands acted upon by arithmetic operators must represent numeric values. Thus, the operands can be integer quantities, floating-point quantities or characters (remember that character constants represent integer values, as determined by the computer's character set.) The remainder operator (%) requires that both operands be integers and the second operand be nonzero. Similarly, the division operator (/) requires that the second operand be nonzero.

Division of one integer quantity by another is referred to as integer division. This operation always results in a truncated quotient (i.e., the decimal portion of the quotient will be dropped). On the other hand, if a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-pointing quotient.

Example

Suppose that a and b are integer variables whose values are 8 and 4, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a + b	12
a - b	4
a * b	32
a / b	2
a % b	0

Notice the truncated quotient resulting from the division operation, since both operands represent integer quantities. Also, notice the integer remainder resulting from the use of the modulus operator in the last expression.

Now suppose that a1 and a2 are floating-point variables whose values are 14.5 and 2.0, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a1 + a2	16.5
a1 - a2	12.5
a1 * a2	29.0
a1 / a2	7.25

Finally, suppose that x1 and x2 are character-type variables that represent the character M and U, respectively. Several arithmetic expressions that make use of these variables are shown below, together with their resulting values (based upon the ASCII character set).

x1 - 77

x1+x2=162

x1+x2+5=215

Note that M is encoded as (decimal) 77, U is encoded as 85, and 5 is encoded as 53 in the

ASCII character set, as shown in...

Example

If one or both operands represent negative values, then the addition, subtraction, multiplication and division operations will result in values whose signs are determined by the usual rules of algebra. Integer division will result in truncation toward zero; i.e., the resultant will always be smaller in magnitude than the true quotient.

The interpretation of the remainder operation is unclear when one of the operands is negative. Most versions of C assign the sign of the first operand to the remainder. Thus, the condition

$$A = ((a/b) * b) + a \% b$$

will always be satisfied, regardless of the signs of the values represented by a and b.

Example

Suppose that x and y are integer variables whose values are 12 and -2, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
x+y	10
x-y	12
x*y	-24
x/y	-6
x%y	0

If x had been assigned a value of -12 and b had been assigned 2, then the value of x/y would still be -6 but the value of a % y would be 0. Similarly, if x and y had both been assigned negative values (-12 and -2, respectively), then the value of x/y would be 3 and the value of x%b would be -2.

Example

$$X = ((x/y)*y) + (x\%y)$$

Will be satisfied in each of the above cases. Most versions of C will determine the sign of the remainder in this manner, though this feature is unspecified in the formal definition of the language.

Examples

Here is an illustration of the results that are obtained with floating-point operands having different signs. Let y1 and y2 be floating-point variables whose assigned values are 0.70 and 3.50. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
y1+y2	2.72
y1-y2	-4.28
y1&y2	-2.63
y/y2	-0.2728

Operands that differ in type may undergo type conversion before the expression takes on its final value. In general, the final result will be expressed in the highest precision possible,

consistent with the data type of the operands. The following rules apply when neither operand is unsigned.

1. If both operands are floating-point types whose precision's differ (e.g., a float and a double), the lower-precision operand will be converted to the precision of the other operand, and the result will be expressed in this higher precision. Thus, an operation between a float and double will result in a double; a float and a long double will result in a long double; and a double and a long double will result in a long double. (Note: In some versions of C, all operands of type float are automatically converted to double.)
2. If one operand is a floating-point type (e.g., float, double or long double) and the other is a char or an int (including short int or long int), the char or int will be converted to the floating-point type and the result will be expressed as such. Hence, an operation between an int and a double will result in a double.
3. If neither operand is a floating-point type but one is long int, the other will be converted to long int and the result will be long int. Thus, an operation between a long int and an int will result in a long int.
4. If neither operand is a floating-point type or a long int, then both operands will be converted to int (if necessary) and the result will be int. Thus, an operation between a short int and an int will result in an int.

Operators are used to compute and compare values, and test multiple conditions. They can be classified as:

1. Arithmetic operators
2. Assignment operators
3. Unary operators
4. Comparison operators
5. Shift operators
6. Bit-wise operators
7. Logical operators
8. Conditional operators

Assignment Operators

Operator	Description	Example	Explanation
=	Assign the value of the right operand to the left	a = b	Assigns the value of b to a
+=	Adds the operands and assigns the result to the left operand	a +=b	Adds the of b to a The expression could also be written as a = a+b
-=	Subtracts the right operand from the left operand and stores the result in the left operand	a -=b	Subtracts b from a Equivalent to a = a-b
=	Multiplies the left operand by the right operand and stores the result in the left operand	a=b	Multiplies the values a and b and stores the result in a Equivalent to a = a*b
/=	Divides the left operand by the right operand and stores the result in the left operand	a/=b	Divides a by b and stores the result in a Equivalent to a = a/b
%=	Divides the left operand by the right operand and stores the remainder in the left operand	a%=b	Divides a by b and stores the remainder in a. Equivalent to a = x%y

Any of the operators used as shown below:

A <operator>=y

can also be represented as

`a=a <operator> b`

that is, b is evaluated before the operation takes place.

You can also assign values to more than one variable at the same time. The assignment will take place from the right to the left. For example,

`A = b = 0;`

In the example given above, first b will be initialized and then a will be initialized.

5.12 UNARY OPERATORS

Operator	Description	Example	Explanation
++	Increases the value of the operand by one	a++	Equivalent a = a+1
—	Decreases the value of the operand by one	a—	Equivalent to a = a-1

5.13 PREFIX AND POSTFIX NOTATIONS

The increment operator, ++, can be used in two ways:

As a prefix, in which the operator precedes the variable.

`++ivar;`

As a postfix, in which the operator follows the variable.

`ivar++;`

The following code segment differentiates the two notations:

`var1 = 20;`

`var2 = ++var1;`

The equivalent of this code is:

`var1 = 20;`

`var1 = var1 + 1; // Could also have been written as var1 += 1;`

`var2 = var1;`

In this case, both var1 and var2 are set to 21, but the value of var2 is 20.

The decrement operator can also be used in both the prefix and postfix forms.

If the operator is to the left of the expression, the value of the expression is modified before the assignment. Conversely, when the operator is to the right of the expression, the C statement, `var2 = var1++`; the original of var1 is assigned to var2. In the statement, `var2 = ++var1++`; the original value of var1 is assigned to var2. In the statement, `var2 = ++var1`; the incremented value of var1 is assigned to var2.

The operators, '++' and '--', are best used in simple expressions like the ones shown above.

5.14 COMPARISON OPERATORS

Comparison operators evaluate to true or false.

Operator	Description	Example	Explanation
=	Evaluates whether the operands are equal.	A==b	Returns true if the values are equal and false otherwise
!=	Evaluates whether the operands are not equal	a!=y	Returns true if the values are not equal and false otherwise
>	Evaluates whether the left operand is greater than the right operand	a>b	Returns true if a is greater than b and false
<	Evaluates whether the left operand is less than the right operand	a<b	Returns true if a is greater than or equal to b and false otherwise
>=	Evaluates whether the left operand is greater than or equal to the right operand	a>=b	Returns true if a is greater than or equal to b and false otherwise
<=	Evaluates whether the left operand is less than or equal to the right operand	A<=b	Returns true if a is less than or equal to b and false

5.15 SHIFT OPERATORS

Data is stored internally in binary format (in the form of bits). A bit can have a value of one or zero. Eight bits form a byte. The following table displays the binary representation of digits 0 to 8.

Decimal	Binary Equivalent
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000

Shift operators work on individual bits in a byte. Using the shift operator involves moving the bit pattern left or right. You can use them only on integer data type and not on the char, bool, float, or double data types.

Operator	Description	Example	Explanation
>>	Shifts bits to the right, filling sign bit at the left	a=10 >> 3	The result of this is 10 divided by 2 ³ . An explanation follows.
<<	Shifts bits to the left, filling zeros at the right	a=10 << 3	The result of this is 10 multiplied by 2 ³ . An explanation follows.

5.16 SHIFTING POSITIVE NUMBERS

If the int data type occupies four bytes in the memory, the rightmost eight bits of the number 10 are represented in binary as

0 0 0 0 1 0 1 0

When you do a right shift by 3(10 >> 3), the result is

0 0 0 0 0 0 0 1

10/2³, which is equivalent to 1.

When you do a left shift by 3 (10 << 3), the result is

0 1 0 1 0 0 0 0

10*2³, which is equivalent to 80

5.17 SHIFTING NEGATIVE NUMBERS

For negative numbers, the unused bits are initialized to 1. Therefore, -10 is represented as:

1 1 1 1 0 1 1 0

5.18 BIT-WISE OPERATORS

Operator	Description	Example	Explanation
& (AND)	Evaluates to a binary value after a bit-wise AND on the operands	a & b	AND results in a 1 if both the bits are 1, any other combination results in a 0
! (OR)	Evaluates to binary value after a bit-wise OR on the two operands	a ! b	OR results in a 0 when both the bits are 0, any other combination results in a 1.
^ (XOR)	Evaluates to a binary value after a bit-wise XOR on the two operands	a ^ b	XOR results in a 0 if both the bits are of the same value and 1 if the bits have different values.
~ (inversion)	Converts all 1 bits to 0s and all 0 bits to 1s		Example given below.

In the example shown in the table, a and b are integers and can be replaced with expressions that give a true or false (bool) result. For example, when both the expressions evaluate to true, the result of using the & operator is true. Otherwise, the result is false.

The ~ Operator

If you use the ~ operator, all the 12 in the bite are converted to 0s and vice versa. For example, 10011001 would become 01100110.

Logical Operators

Use logical operators to combine the results of Boolean expressions.

Operator	Description	Example	Explanation
&& false.	Evaluates to true, if both the conditions evaluate to true, false otherwise	a>6&& y<20	The result is true if condition 1 (a>6) and condition 2 (y<20) are both true. If one of them is false, the result is
	Evaluate to true, if at least one of the conditions evaluates to true and false if none of the conditions evaluate to true.	a>6 y < 20	The result is true if either condition1 (a>6) and condition2 (y<20) or both evaluate to true. If both the conditions are false, the result is false.

5.19 SHORT CIRCUIT LOGICAL OPERATORS

These operators (&&, ||) appear to be similar to the bit-wise & and | operators, except that they are limited to Boolean expressions only. However, the difference lies in the way these operators work. In the bit-wise operators, both the expressions are evaluated. This is not always necessary since:

false & a would always result in false

true | a would always result in true

Short circuit operators do not evaluate the second expression if the result can be obtained by evaluating the first expression alone.

For example

a < 6 && y > 20

The second condition (b>20) is skipped if the first condition is false, since the entire expression will anyway, be false. Similarly with the || operator, if the first condition evaluates to true, the second condition is skipped as the result, will anyway, be true. These operators, && and ||, are therefore, called short circuit operators.

If you want both the conditions to be evaluated irrespective of the result of the first condition, then you need to use bit-wise operators.

5.20 CONDITIONAL OPERATORS

Operator	Description	Example	Explanation
(condition) val1, val2	Evaluates to val1 if the condition returns true and val2 if the condition returns false	a = (b>c) ? b:c	A is assigned the value in b, if b is greater than c, else a is assigned the value of c.

This example finds the maximum of two given numbers.

If (num1 > num2)

```
{
imax = num1;
}
else
{
imax = num2;
}
```

In the above program code, we determine whether num1 is greater than num2. The variable, imax is assigned the value, num1, if the expression, (num1 > num2), evaluates to true, and the value, num2, if the expression evaluates to false. The above program code can be modified using the conditional operator as:

Imax = (num1 > num2) ? num1 : num2;

The ?: Operator is called the ternary operator since it has three operands.

This example calculates the grade of a Ram based on his marks.

Int marks = 0;


```
Cout << "Please enter marks of the Ram;
```

```
Cin >> marks;
```

Char grade = (marks < 80), the variable, grade is assigned the value, 'A'. If Ram's score is less than or equal to 80 grade is assigned the value 'B'.

The following code will display either "PASSED" or "FAILED", depending on the value in the variable, score.

```
Cout << {score > 50? "PASSED" or FAILED, depending on the value in the variable, score.
```

```
Cout << (score > 50? "PASSED" : "FAILED") << endl;
```

5.21 ORDER OF PRECEDENCE OF OPERATORS

The table shows the order of precedence of operators. Those with the same level of precedence are listed in the same row. The order can be changed by using parentheses at appropriate places.

Type	Operators
------	-----------

High Precedence	[] ()
-----------------	-------

Unary	+ - ~ ! ++ --
-------	---------------

Multiplicative	* / %
----------------	-------

Additive	+ -
----------	-----

Shift	<< >> >>>
-------	-----------

Relational	< <= >= >
------------	-----------

Equality	== !=
----------	-------

Bit-wise	& ^ !
----------	-------

Logical	&&
---------	----

Conditional	?:
-------------	----

Assignment	= += -= /= %=
------------	---------------

The unary operators, assignment operators and the conditional operator group from the right to the left. All other operators group from the left to the right.

Student Activity 4

1. What are operators? List various types of operators available in C++.
2. Describe prefix and postfix notations.
3. List comparison operators available in C++.
4. What are shift operators
5. List bit-wise operators available in C++.
6. What is the function of '~' operator?
7. Give the order of precedence of operators.

5.22 THE CONST KEYWORD

The keyword, const (for constant), precedes the data type of a variable.

```
Const <data type> <name of variable> = <value>;
```

Example

```
Const float g = 9.8;
```

```
Cout << "The acceleration due to gravity is" << g << "m/s2";
```

```
G=g+4; // ERROR!! Const variables cannot be modified.
```

The const keyword specifies that the values of the variable will not change throughout the program. This prevents the programmer from changing the value of variable, like the one in the example given above. If the keyword, const, has been used while defining the variable, the compiler will report an error if the programmer tries to modify it.

5.23 OPERATOR AND FUNCTION OVERLOADING

Overloading basically means assigning additional meanings to an entity. If an operator, for instance, has two or more meanings, we would say that the operator has been overloaded. Similarly, if a function has two or more definitions and therefore, two or more ways of functioning, we would say that the function has been overloaded.

It is an important technique that has enhanced the power of extensibility of C++. C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as *operator overloading*.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can almost create a new language of our own by the creative use of the function and operator overloading techniques. All the C++ operators except the following can be overloaded (given additional meaning).

- access operators of class member (.,*)
- scope resolution operator (::)
- size operator (sizeof)
- conditional operator (?:)

The excluded operators are very few when compared to the large number of operators, which qualify for the operator overloading definition.

Only semantics of an operator can be extended and not its syntax, the grammatical rules that govern its use such as the number of operands, precedence and associativity. For example, the multiplication operator will enjoy higher precedence than the addition operator.

When an operator is overloaded, its original meaning is not lost but an additional meaning is attached to it. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called operator function, which describes the task. The general form of an operator function is:

```
<Returntype> classname:: operator op (<arg-list>)
{
    Function body // task defined
}
```

Where optional returntype is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator, operator op is the function name.

Operator functions must be either member function will have only one argument for unary operators and two for binary operators, while a member function has no argument for unary operators and only one for binary operators. This is because the object used to invoke the member functions is passed implicitly and therefore is available for the member function. This is not the case with friend functions. Arguments may be passed either by value or by reference.

Operator functions are declared in the class using prototypes as follows:

```
vector operator +(vector);           //vector addition
vector operator -( );               //unary minus
friend vector operator +(vector, vector); //vector addition
friend vector operator -(vector);    //unary minus
vector operator -(vector & a);       //subtraction
int operator ==(vector);             //comparison
friend int operator ==(vector, vector) //comparison
```

vector is a data type of class and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics).

The process of overloading may be summarized in the following steps:

1. First, create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator op() in the public part of the class. It may be either a member function or a friend function.
3. Define the operator functions to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

op x or x op

for unary operators and

x op y

for binary operators

op x (or x op) would be interpreted as

operator op (x)

for friend functions. Similarly, the expression x op y would be interpreted as either

x.operator op(y)

in case of member functions, or

operator op (x,y)

in case of friend functions, when both the forms are declared, standard argument matching is applied to resolve any ambiguity.

Overloading unary operators

Let us consider the unary minus operator. A minus operator, when used as unary taxes just

one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object should change the sign of each of its data items.

Example

Overloading unary minus operator.

```
#include <iostream.h>

class d3space
{
    int x;
    int y;
    int z;
public:
    void getdata (int a, int b, int c);
    void display (void);
    void operator -( ) //overload unary minus
};

void d3space :: getdata (int a, int b, int c)
{
    x=a;
    y=b;
    z=c;
}

void d3space :: display (void)
{
    cout << x << " ";
    cout << y << " ";
    cout << z << "\n";
}

void d3space :: operator- ( )           // Defining operator - ( )
{
    x=-x;
    y=-y;
    z=-z;
}

main ()
{
    d3space s;
    s.getdata (10,-20, 30);
    cout << "s : ";
    s.display ( ) ;
    -s;                                //activates operator -( )
    cout << "s : ";
    s.display ( ) ;
}
```

The output of the program:

S: 10-20 30

S: -10 20-30

Note that the function operator - () takes no argument. Then, what does this operator functions do? It changes the sign of data members of the objects S. Since this function is a member function of the same class, it can directly access the members of the object, which activated it. A statement like S2=-S1; will not work because, the functions operator - () does not return any value. It can work if the function is modified to return an object.

Lets us see how a binary operator may be overloaded.

```
#include<iostream.h>

class complex
{
    float x;                //real part
    float y;                //imaginary part
public:
    complex ( ) [ ]          //constructor1
    complex (float real, float imag) //constructor2
    { x=real; y=imag; }
    complex operator + (complex); //operator overloading
    void display (void);
};

complex complex :: operator + (complex c)
{
    complex temp;           //temporary
    temp.x = x + c.x;        //float addition
    temp.y = y + c.y;        //float additon
    return (temp);
}

void complex :: display (void)
{
    cout << x << "+j" << y <, "\n";
}

main ()
{
    complex c1, c2, c3;      //invokes constructor 1
    c1=complex (2.5, 3.5);   //invokes constructor 2
    c2 = complex (1.6, 2.7); //invokes constructor 2
    c3=c1+c2;

    cout <<"c1="; c1.display ();
    cout << "c2 ="; c2.display ();
    cout << "c3 ="; c3.display ();
}
```

The output of the program is:

```
c1=2.5+j3.5
```

```
c2=1.6+j2.7
```

```
c3=4.1+j6.2
```

Let us have a close look at the function operator + () and see how the operator overloading is implemented.

Complex complex : : operator + (complex c)

```
{
    complex temp;
    temp.x=x+c.x;
    temp.y=y+c.y;
    return(temp);
}
```

Note the following features of this function:

1. It receives only one complex type argument explicitly.
2. it returns a complex type value.
3. It is a member function of complex.

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from?

Let us look at the statement that invokes this function:

```
C3=C1+C2;    //invokes operator + ( ) function
```

A member function can be invoked only by an object of the same class. Here, the object C1 takes the responsibility of invoking the functions and C2 plays the role of an argument that is passed to function. The above innovation statement is equivalent to

```
C3=C1.operator+(c2);    //usual function call syntax
```

Therefore, in the operator + () function, the data members of C1 are accessed directly and the data members of C2 (that is passed as an argument) are accessed using the dot operator. Thus, both the objects are available for the function. For example, in the statement

```
temp.x = x + c.x;
```

c.x refers to the object C2 and x refers to the object C1. temp.x is the real part of temp that has been created specially to hold the results of addition of c1 and c2. The function returns the complex temp to be assigned to C3.

As a rule, in overloading of binary operators, the left-hand operand is used to invoke the operator functions and the right-hand operand is passed as an argument.

We can avoid the creation of the temp object by replacing the entire function body by the following statement:

```
return complex ((x+c.x), (y+c.y)); //invokes construct 2
```

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object. Such an object is

called a temporary object and goes out of scope as soon as the contents are assigned to another object. Using temporary objects can make the code shorter, more efficient and better to read.

Friend function may be used in the place of member functions for overloading a binary operator. The only difference being that a friend function requires two arguments to be explicitly passed to it while a member function requires only one.

The complex number program discussed in the previous section can be modified using a friend operator functions as follows:

1. Replace the member function declaration by friend function declaration.

```
friend complex operator +(complex, complex);
```

2. Redefine the operator functions as follows:

```
complex operator +(complex a, complex b)
{
    return complex ((a.x+b.x),(a.y+b.y));
}
```

In this case, the statement

```
C3=c1+c2;
```

Is equivalent to

```
C3=operator +(c1, c2);
```

In most cases, we will get the same results by the use of either a friend functions or a member functions. Why is the an alternative is made available?. There are certain situations where we would like to use a friend function rather than a member function. For instance, consider a situation where we need to use two different types of operands for a binary operator, say, one an object and another a built-in type data as shown below:

```
A=B+2;(or A=B*2;)
```

Where A and B are object of the same class. This will work for a member function but the statement.

```
A=2+B;(or A=2*B)
```

Will not work. This is because the left-hand operand which is responsible for invoking the member function should be an object of the same class. However, a friend function allows both the approaches. How?

It may be recalled that an object need not be used to invoke a friend function but can be passed as an argument. Thus, we can use a friend function with a built-in type data as the left-hand operand and an object as the right-hand operand.

Example:

```
#include <iostream.h>
```

```
const size=3;
```

```
class vector
```

```
{
    int v[size];
```

```

public:
vector ( ); //constructs null vector
vector (int * x); //constructs vector from array
friend vector operator * (int a, vector b); //friend 1
friend vector operator * (vector b, int a); //friend 2
friend istream & operator >> (istream &, vector &);
friend ostream & operator << (ostream &, vector &);
};

vector :: vector ( )
{
for (int i=0; i<size; i++)
v[i]=0;
}

vector : vector (int * x)
{
for (int i=0; i < size; i++)
v[i]=x[i];
}

vector operator * (int a, vector b)
{
vector c;
for (int i=0; i<size; i++)
c.v[i]=a*b.v[i];
return c;
}

vector operator *(vector b, int a)
{
vector c;
for (int i=0; i < size; i++)
c.v[i]=b.v[i] * a;
return c;
}

istream & operator >> (istream & din, vector & b)
{
for (int i=0; i<size; i++)

```



```
        din >> b.v[I];  
        return (din);  
    }  
  
    ostream & operator << (ostram & dout, vecto & b)  
{  
    dout << "(" << b.v[0];  
    for(int I=1; I<size; I++)  
        dout<< "," << b.v [I];  
    dout << ")";  
    return (dout);  
}  
  
    int x[size]=[2,4,6];  
main ()  
{  
    vector m;                //invokes constructor 1  
    vector n=x ;             //invokes constructor 2  
    cout << "Enter elements of vector m" << "\n";  
    cin >> m;  
    cout << "\n";  
    cout << "m=" << m << "\n"; // invokes operator << ( )  
    vector p, q;  
    p=2*m;                   //invokes friend 1  
    q=n*2;                   //invokes friend 2  
    cout << "\n";  
    cout << "p=" << p << "\n";    //invokes operator << ( )  
    cout << "q =" << q << "\n";  
}
```

The output of this program:

Enter elements of vector m

5 10 15

m=(5, 10, 15)

p=(10, 20, 30)

q=(4,8,12)

The program overloads the operator * tow times, thus overloading the operator function operator *() itself. In both the cases, the functions are explicitly passed two argument and they are invoked like any other overloaded function, based on the types of its arguments. This enables us to use both the forms of scalar multiplications such as

```
P=2*m;           //equivalent to p=operator*(2,m);
```

```
Q=n*2;           //equivalent to q=operator*( n,2);
```

Vector() creates a vector m and initializes all its elements to 0. The second constructor Vector (int *x); creates a vector and copies the elements pointed to by the pointer argument x into it. Therefore, the statements

```
Int x[3]={2, 4, 6};
```

```
Vector n=x;
```

create n as a vector with components 2, 4, and 6.

Note that we have used vector variables like m and n in input and output statements just like simple variables. This has been made possible by overloading the operators >>and<< using the functions:

```
Friend istream & operator >> (istream &, vector&);
```

```
Friend ostream & operator << (ostream &, vector &);
```

Istream and ostream are classes defined in the iostream.h file which has been included in the program.

5.24 MANIPULATION OF STRINGS USING OPERATORS

ANSI C implement strings using character arrays, pointers and string functions. There are no operators for manipulation the strings. One of the main drawbacks of string manipulations in C is that whenever a string is to be copied, the programmer must first determine its length and allocate the required amount of memory.

Although these limitations exist in C++ as well, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers. For example, we shall be able to use statements like

```
String3 =string1+string2;
```

```
If(string 1>=string 2) string=string 1;
```

Strings can be defined as class objects, which can be then manipulated like the built-in types. Since the strings vary greatly in size, we use new to allocate memory for each string and a pointer variable to point to the string array. This, we must create string objects that can hold these two pieces of information, namely, length and location which are necessary for string manipulations. A typical string class will look as follows:

```
Class string
```

```
{
```

```
char *p;           //pointer to string
```

```
int len;           // length of string
```

```
public:
```

```
.....           //member functions
```

```
.....           //to initialize and
```

```
.....           //manipulate strings
```

```
};
```

We shall consider an example program to illustrate the application of overloaded operators to strings. The program overloads two operators, + and <= just to show how they are implemented. This can be extended to cover other operator as well.

```
#include <string.h>
```

```
#include <iostream.h>
```

```
class string
```

```
{
```

```
    char *p;
```

```
    int len;
```

```
    public:
```

```
    string ( ) (len=0; p=0; )           //create null string
```

```
    string (const char * s);           //create string from array s
```

```
    string (const string &s);          //copy constructor
```

```
    string ( ) (delete p;)              //destructor
```

```
    friend string operator + (const string & s, const string & t);
```

```
    friend int operator <=(const string & s, const string & t);
```

```
    friend void show (const string s);
```

```
};
```

```
string : : string (const char * s)
```

```
{
```

```
    len=strlen (s);
```

```
    p=new char [len +];
```

```
    strcpy (p,s);
```

```
}
```

```
string : : string (const string & s)
```

```
{
```

```
    len =s.len;
```

```
    p=new char [len +1];
```

```
    strcpy (p, s.p);
```

```
}
```

```
string operator +(const string & s, const string & t)
```

```
{
```

```
    string temp;
```

```
    temp.len=s.len+t.len;
```

```

temp.p=new char [temp.len+1];

strcpy (temp.p, s.p);
strcat(temp.p, t.p);
return (temp);
}

int operator <=(const string & s, const string & t)
{
    int m=strlen(s.p);
    int n=strlen (t.p);
    if (m<=n) return (1);
    else return (0);
}

void show (const string s)
{
    cout << s.p;
}

main ()
{
    string s1="New";
    string s2="York";
    strings3="Delhi";
    string t1, t2, t3, t4;
    t1=s1;
    t2=s2;
    t3=s1+s2;
    t4=s1+s3;
    cout << "\nt1="; show (t1);
    cout << "\n2="; show (t2);
    cout << "\n";
    cout << "\nt3="; show (t3);
    cout<< "\nnt4="; show (t4);
    cout << "\n\n";
    if (t3 <=t4)
    {
        show (t3);

```

```
        cout << " smaller than ";  
        show (t4);  
        cout << "\n";  
    }  
  
    else  
{  
        show (t4);  
        cout (t4);  
        cout << "smaller than ";  
        cout << "\n";  
    }  
}
```

The following is the output of program:

T1=New

T2=York

T3=New York

T4=New Delhi

New York smaller than New Delhi

5.25 RULES FOR OVERLOADING OPERATORS

Although it looks simple to redefine the operator, there are certain restrictions and limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is we cannot redefine the plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators,. That cannot be overridden.
5. There are some operators that cannot be overloaded.
6. We cannot use friend functions to overload certain operators. However, member functions can be used to overload them.
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values. But, those overloaded by means of a friend function take one reference argument (the object of the relevant class).
8. Binary operators overloaded through a member function take on explicit argument and those, which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left-hand operand must be and object of the relevant class.
10. Binary arithmetic operators such as +, -, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Operators that cannot be overloaded	
sizeof	sizeof operator
.	Membership operator
.*	Pointer-to-member operator
::	Scope resolution operator
?:	conditional operator

Where a friend cannot be used	
=	Assignment operator
()	Function call operator
[]	Subscripting operator
->	sClass member access operator

5.26 FUNCTION OVERLOADING

As stated earlier, overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in OOP, i.e. the same function performing varying jobs.

Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded `addit()` function handles different types of data as shown below:

//Declarations

```
int addit (int a, int b);           //prototype 1
int addit (int a, int b, int c);    //prototype 2
double addit (double p, double y);  //prototype 3
double addit (int p, double q);     //prototype 4
double addit (double p, int q);     //prototype 5
```

//Function calls

```
cout << addit (5, 10);             //uses prototype 1
cout << addit (15, 10.0)           //uses prototype 4
cout << addit (12.5, 7.5)         //uses prototype 3
cout << addit (5, .10, 15);        //uses prototype 2
```

```
cout << addit (0.75, 5);           //uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate functions for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as, char to int and float to double to find a match.
3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

```
long square(long n)
double square(double x)
```

A function call such as

```
Square(10);
```

will cause an error because int argument can be converted to either long or double, thereby creating an ambiguous situation as to which version of square() should be used.

4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Program listed below illustrates function overloading of function named volume().

```
#include <iostream.h>

int volume (int);

long volume (long, int, int);

double volume (double, int);

main ()
{
    cout << volume(10) << "\n";
    cout << volume(2.5, 8) << "\n";
    cout << volume(100L, 75, 15);
}

int volume(int s)           //cube
{
    return (s*s*s);
}

double volume (double r, int h) //cylinder
{
    return (3.14519 * r * r * h);
}
```

```
long volume (long l, int b, int h) //rectangular box
```

Implementing OOPs Concepts

```
{  
    return (l*b*h);  
}
```

The output of program would be:

1000

157.2595

112500

Overloading of the functions should be done with caution. We should not overload unrelated functions and should reserve functions overloading for functions that perform closely related tasks. Sometimes, the default arguments may be used instead of overloading. This may reduce the number of functions to be defined.

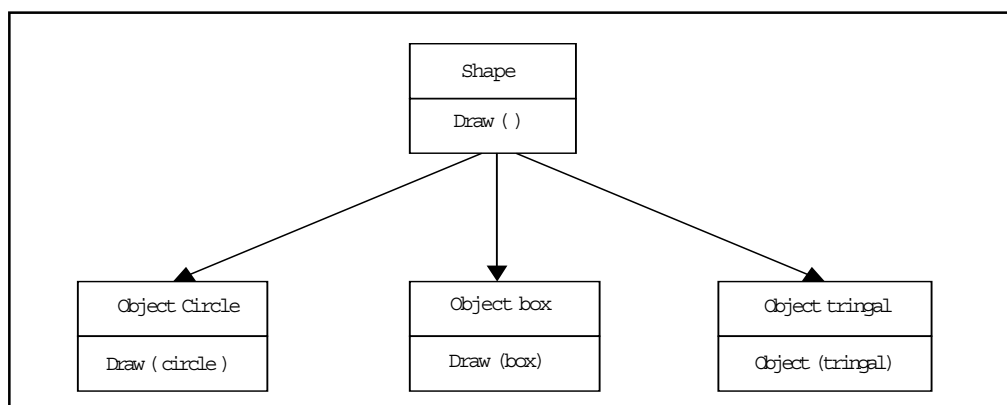
Student Activity 5

1. What is the function of 'const' keyword?
2. Define function overloading.
3. Define operator overloading.
4. Give example to explain the overloading of unary operators.
5. What are friend functions?
6. Give rules for operator overloading.

5.27 POLYMORPHISM

Polymorphism is an important OOP concept. Pleomorphism means the ability to take more than one form. For example, an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data used in the operation. For example, consider the operation of addition. For tow numbers, the operation will generate a sum. If the operands are strings, then the operation will produce a third string by contention. The diagram given below, illustrates that a single function name can be used to handle different number and types of arguments. This is something similar to a particular word having several different meanings depending on the context.

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



Polymorphism can be implemented using operator and function overloading, where the same operator and function works differently on different arguments producing different results. These polymorphisms are brought into effect at compile time itself, hence is known as *early binding*, *static binding*, *static linking* or *compile time polymorphism*.

However, ambiguity creeps in when the base class and the derived class both have a function with same name. For instance, let us consider the following code snippet.

```
Class aa
{
    Int x;

    Public:

        Void display() {.....} //display in base class
};

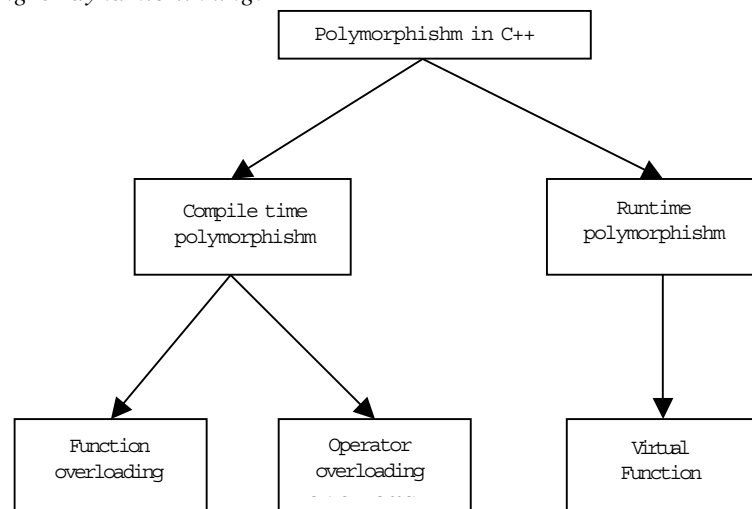
Class bb : public aa
{
    Int y;

    Public:

        Void display() {.....} //display in derived class
};
```

Since, both the functions aa.display() and bb.display() are same but at in different classes, there is no overloading, and hence early binding does not apply. The appropriate function is chosen at the run time – *run time polymorphism*.

C++ supports *run-time polymorphism* by a mechanism called *virtual function*. It exhibits *late binding* or *dynamic linking*.



As stated earlier, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. Therefore, an essential feature of polymorphism is the ability to refer to objects without any regard to their classes. It implies that a single pointer variable may refer to object of different classes.

However, a base pointer, even if it is made to contain the address of the derived class, always executes the function in the base class. The compiler ignores the content of the pointer and chooses the member function that matches the type of the pointer. Thus, the polymorphism stated above cannot be implemented by this mechanism.

C++ implements the runtime object polymorphism using a function type known as *virtual function*. When a function with the same name is used both in the base class and the derived class, the function in the base class is declared virtual by attaching the keyword *virtual* in the base class preceding its normal declaration. Then C++ determines which function to use at run time based on the type of object pointed to by the base pointer rather than the type of the pointer. Thus, by making the base pointer to point to different objects, one can execute different definitions of the virtual function as given in the program below.

```
#include <iostream.h>

class base
{
    public:
        void display()
        {
            cout<<"\n print base";
        }
        virtual void show()           //virtual function
        {
            cout<<"\n show base";
        }
};

class derived : public base
{
    public:
        void display()
        {
            cout<<"\n display derived";
        }

        void show()
        {
            cout<<"\n show derived";
        }
};

main()
{
    base bb;
    derived dd;

    base *baseptr;

    cout <<"\nbseptr points to the base \n";
```

```
baseptr = &bb;

baseptr -> display();    //calls base function display()

baseptr -> show();       //calls base function show()

cout <<"\n\nbaseptr points to the derived \n";

baseptr = &dd;

baseptr -> display();    //calls derived function display()

baseptr -> show();       //calls derived function show()

}
```

The output of this program would be:

```
Baseptr points to base
Display base
Show base

Baseptr points to derived
Display derived
Show derived
```

Here, we see that the same object pointer points to two different objects of different classes and yet selects the right function to execute. This is implementation of function polymorphism. Remember, however, that runtime polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. It is also interesting to note that since, all the C++ classes are derived from the Object class, a pointer to the Object class can point to any object of any class in C++.

Student Activity 6

1. Define polymorphism.
2. What is a virtual functions?
3. Define a stream.
4. Give a program to display use of different stream manipulations.

5.28 STREAMS

A stream is a source of sequence of bytes. A stream abstracts for input/output devices. It can be tied up with any I/O device and I/O can be performed in a uniform way. The C++ *iostream* library is an object-oriented implementation of this abstraction. It has a source (producer) of flow of bytes and a sink (consumer) of the bytes. The required classes for the stream I/O are defined in different library header files.

To use the I/O streams in a C++ program, one must include *iostream.h* header file in the program. This file defines the required classes and provides the buffering. Instead of functions, the library provides operators to carry out the I/O. Two of the Stream Operators are:

```
<<    : Stream insertion for output.

>>    : Stream extraction for input.
```

The following streams are created and opened automatically:

```
cin     : Standard console input (keyboard).

cout    : Standard console output (screen).
```

cprn : Standard printer (LPT1).
 cerr : Standard error output (screen).
 clog : Standard log (screen).
 caux : Standard auxiliary (screen).

Example:

The following program reads an integer and prints the input on the console.

```
#include <iostream>      // Header for stream I/O.

int main(void)
{
    int p;                // variable to hold the input integer

    cout << "Enter an integer: ";

    cin >> p;

    cout << "\n You have entered " << p;
}
```

Streams can also be tied up with data files. The required stream classes for file I/O are defined in fstream.h and/or strstream.h.

fstream : File I/O class.

ifstream	Input file class.
istrstream	Input string class.
ofstream	Output file class.
ostrstream	Output string class.
strstream	String I/O class.

There are some special functions that can alter the state the stream. These functions are called manipulators. Stream manipulators are defined in iomanip.h.

dec	Sets base 10 integers.
endl	Sends a new line character.
ends	Sends a null (end of string) character.
flush	Flushes an output stream.
fixed	Sets fixed real number notation.
hex	Sets base 16 integers.
oct	Sets base 8 integers.
ws	Discard white space on input.
setbase(int)	Sets integer conversion base (0, 8, 10 or 16 where 0 sets base 10).
setfill(int)	Sets fill character.
setprecision(int)	Sets precision.
setw(int)	Sets field width.
resetiosflags(long)	Clears format state as specified by argument.
setiosflags(long)	Sets format state as specified by argument.

The stream classes have a variety of member functions to give them their required functionalities. Thus, there is a function to open the stream, one for reading/writing, one for closing the stream and the like. The stream class member functions are listed below:

void .close()	Closes the I/O object.
int .eof()	Returns a nonzero value (true) if the end of the stream has been reached.
char .fill(char fill_ch void)	Sets or returns the fill character.
int .fail()	Returns a nonzero value (true) if the last I/O operation on the stream failed.
istream& .get(int ch)	Gets a character as an int so EOF (-1) is a possible value.
istream& .getline(char* ch_string, int maxsize, char delimit)	Get a line into the ch_string buffer with maximum length of maxsize and ending with delimiter delimit.
istream& .ignore(int length[, int delimit])	Reads and discards the number of characters specified by length from the stream or until the character specified by delimit (default EOF) is found.
iostream& .open(char* filename, int mode)	Opens the filename file in the specified mode.
int .peek();	Returns the next character in the stream without removing it from the stream.
int .precision(int prec void)	Sets or returns the floating point precision.
ostream& .put(char ch)	Puts the specified character into the stream.
istream& .putback(char ch)	Puts the specified character back into the stream.
istream& .read(char* buf, int size)	Sends size raw bytes from the buf buffer to the stream.
long .setf(long flags [, long mask])	Sets (and returns) the specified ios flag(s).
long .unsetf(long flags)	Clears the specified ios flag(s).
int .width(int width void)	Sets or returns the current output field width.
ostream& .write(const char* buf, int size)	Sends size raw bytes from buf to the stream.

A file may be opened for a number of file operations. The corresponding stream must be set with the intended operation. The different file stream modes are indicated by File Access Flags as listed below:

ios::app	Open in append mode.
ios::ate	Open and seek to end of file.
ios::in	Open in input mode.
ios::nocreate	Fail if file doesn't already exist.
ios::noreplace	Fail if file already exists.
ios::out	Open in output mode.
ios::trunc	Open and truncate to zero length.
ios::binary	Open as a binary stream.

Examples:

1. Basic Program File Structure and Sample Function Call with stream I/O

```
#include <iostream>    // Header for stream I/O.

#include <iomanip>       // Header for I/O manipulators.

// Function declaration (prototype) with default values.
```

```

float sum(float required_term, float optional_term = 0.0); //main function
int main(void)
{
    int p;           // Output numeric precision.
    float a;         // Units and description of a.
    float b;         // Units and description of b.
    cout.setf(ios::showpoint);
    cout << "Enter the output precision (an integer): ";
    cin >> p;
    if (!cin)
    {
        cout << "Input error.\n";
        return 1;
    }
    cout << setprecision(p);
    cout << "Enter two real numbers to be summed: ";
    cin >> a >> b;
    if (!cin)
    {
        cout << "Input error\n";
        return 2;
    }
    cout << "Entered values: a = " << a << " and b = " << b << endl;
    cout << "sum(a, b) = " << sum(a, b) << endl;
    cout << "sum(a) = " << sum(a) << endl;
    return 0;
}

// Define the sum function.
float sum(float x, float y)
{
    return (x + y);
}

```

2. This program displays use of different stream manipulators.

```
#include <iostream.h>
```

```
#include <iomanip.h>

int main(void)
{
    int I = 100;

    cout << setfile('.');

    cout << setiosflags(ios::left);

    cout << setw(20) << "Decimal";

    cout << resetiosflags(ios::left);

    cout << setw(6) << dec << I << endl;

    cout << setiosflags(ios::left);

    cout << setw(20) << "Hexadecimal";

    cout << resetiosflags(ios::left);

    cout << setw(6) << hex << I << endl;

    cout << setiosflags(ios::left);

    cout << setw(6) << oct << I << endl;

}
```

The output of the program:

```
Decimal.....100
Hexadecimal.....64
Octal.....144
```

3. This program exchanges the values of two variable.

```
#include <iostream>    // Header for stream I/O.
#include <fstream>     // Header for file I/O.
#include <iomanip>      // Header for I/O manipulators.

// Function declaration use references and without default values.
void swap(float& first_indentifier, float& second_indentifier);

// Main program
int main(void)
{
    int p = 3;        // Ouput numeric precision.
    float a = 2.5f;   // Units and description of a (f indicates float value).
    float b = 7.5f;   // Units and description of b.
    ofstream fout;    // Declare an output file object.
    fout.open("swap.out", ios::out); // Open the output file.
    if (!fout)        // See if the file was opened successfully.
```

```

{
    cout << "Can't open output file!\n";
    return 1;
}

fout.setf(ios::showpoint);          // Show decimal points.
fout << setprecision(p); // Set the precision.
fout << "Before swapping...\n";
fout << "a = " << a << " and b = " << b << endl;
swap(a, b);
fout << "After swapping...\n";
fout << "a = " << a << " and b = " << b << endl;
fout.close();                      // Close the out file.
return 0;
}

// Define the swap function. Use references so argument changes are returned.
void swap(float& x, float& y)
{
    float hold;
    hold = x;
    x = y;
    y = hold;
    return;
}

```

4. This program demonstrates the reading of data files containing comments and displaying them on the screen. The lines in the input file starting with '!' will be treated as comment line. The input file name is comments.txt having the following sample data:

```

!
! These comment lines should be ignored.
!

```

This is the first non-comment line.

This is the second non-comment line.

Almost done.

This is the last non-comment line.

// Included files.

#include <iostream> // Header for console I/O


```
#include <fstream>      // Header for file I/O.

#include <string>        // Header for STL strings.

// Define skip_comments function.

int skip_comments(ifstream& file, char mark)

{
    /*

    This function skips the all the lines at the start of the specified file that begin with the
    specified character. The file must already be open when this function is called. Error
    checking not yet included.

    */

    const int MAX_CHARS = 100;    // This is the maximum characters per line.

    while (file.peek() == mark)

    {
        file.ignore(MAX_CHARS, '\n');
    }

    return 0;
}

//main function

int main()

{
    // Open input file.

    ifstream fin;                // Declare an input file object.

    fin.open("comments.txt", ios::in); // Open the input file in project
                                     //folder.

    if (!fin)                    // See if the file was opened successfully.

    {

        cout << "Can't open input file. \n";

        return 1;

    }

    cout << "About to skip comments.\n";

    skip_comments(fin, '!');

    string sometext;

    getline(fin, sometext);

    while (!fin.fail())

    {

        cout << sometext << '\n';

        getline(fin, sometext);
```

```

}

    if (fin.fail() && !fin.eof())
    {

        cout << "Error while reading file. \n";

        fin.close();

        return 2;

    }

return 0;

}

```

5.29 SUMMARY

C++ language is made up of letters, words, sentences and constructs just like English language. A collection of characters, much like a word in English language are called tokens. A token can be a keyword, or identifier, constant, string or an operator. Keywords are the reserve words that cannot be used as names of variable or other user-defined program elements. Identifiers refer to the names of variables, functions, arrays, classes etc. created by the programmer, Basic types, Derived types and user defined are the three types of data in C++. There are five basic data types: Char, int, float, double and void.

An array represents named list of finite number of similar data elements. A function is a named part of a program that can be invoked from the other parts of the program. A pointer is a variable that holds a memory address of another variable. A reference is an alternative name for an object. A constant is a data item whose data value can never change during the program run.

Operators are the symbols that represent specific operations. Arithmetic operators are unary +, Unary-, +, -, *, / and % . '%' uses pure integer division thereby requires integer operands. Comparison operators, also called relational operators, compare the relationships among values. The order of evaluation among logical operators is NOT, AND, OR, i.e.!, &&, ||.

Along with malloc(), calloc () and free() functions, C++ also defines two unary operators, new and delete, that perform the task of allocating and freeing the memory.

5.30 KEYWORDS

Tokens: A collections of characters, much live a word in English language. The smallest individual unit in a program.

Keywords: Explicitly reserved identifiers that cannot be used as names for the program variables or other user defined program elements.

Identifies: The names of variables, functions, arrays, classes, etc. created by the programmer.

Enumerated data Type: An enumerated data type is user defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code.

Derived data type: C++ allows programmers to derive complex data types using simple basic data types. These data types are referred to as derived data types.

Operands: The data items that operators acted upon all called operands.

Expression: A combination of variables, constants and operators written according to some rules.

Character constant: One or more characters enclosed in single quotes.

String constant: Sequence of characters enclosed in double quotes.

Scope resolution operator: This operator enables a program to access a global variable when a local variable of the same name is in scope.

5.31 REVIEW QUESTIONS

1. What is meant by token? Name the tokens available in C++?
2. What are keywords? Can keywords be used as identifiers?
3. What is an integer constant? Write integer forming rule of C++.
4. How many types of integer constants are allowed in C++? How are they written?
5. What is a character constant in C++?
6. What is meant by a floating constant in C++? How many ways can a floating constant be represented into?
7. Which character is automatically added to strings in C++?
8. How are floating constants represented in C++? Give examples to support your answer.
9. How are string-literals represented and implemented in C++?
10. Which are operators? What is their functions? Give examples of some unary and binary operators?
11. Fill in the blanks:
 - (a) _____refer to the names of variables, functions, arrays, classes, etc.
 - (b) ANSIC recognizes only the first_____in a name.
 - (c) The _____automatically enumerates a list of words by assigning them values.
 - (d) The application of _____in C++ is similar to that in C.
 - (e) Pointers are extensively used in C++ for memory management and achieving_____.

Answers to Review Questions

11. (a) Identifiers (b) 32 Characters
(c) Enum keywords (d) Arrays
(e) Polymorphism

5.32 FURTHER READINGS

Herbert Schildt, *The complete Reference-C++*, Tata Mc Graw Hill

Robert Lafore, *Object Oriented Programming in Turbo C++*, Galgotia Publications.

E. Balagurusamy, *Object Oriented Programming through C++*, Tata McGraw Hill.

UNIT

6

CONDITIONS AND CONTROL STATEMENTS

LEARNING OBJECTIVES

After studying this unit, you should be able to:

- Describe various conditional expressions
- Describe loop statements
- Describe breaking control statements

UNIT STRUCTURE

- 6.1 Introduction
- 6.2 The if....else Construct
- 6.3 Branching: The if-else Statement
- 6.4 LOOPING: THE While STATEMENT
- 6.5 More Looping: The do-While Statement
- 6.6 Still More Looping: The for Statement
- 6.7 Identifier Syntax
- 6.8 Expressions
- 6.9 Statements
- 6.10 Summary
- 6.11 Keywords
- 6.12 Review Questions
- 6.13 Further Readings

6.1 INTRODUCTION

C++ program may require that a logical test be carried out at some particular point within the program. One of several possible actions will then be carried out, depending on the outcome of the logical test. This is known as branching. There is also a special kind of branching, called selection, in which one group of statements is selected from several available groups. In addition, the program may require that a group of instructions be executed repeatedly, until some logical condition has been satisfied. This is known as looping. Sometimes the required number of repetitions is known in advance; and sometimes the computation continues indefinitely until the logical condition becomes true.

All of these operations can be carried out using the various control statements included in C.

Example

Several logical expressions are given below.

Count <= 100

sqrt x + y + z > 0.00

```
Answer == 0  
Balance >= cutoff  
Ch 1 < 'A'  
Letter 1 = 'a'
```

The first four expressions involve numerical operands. Their meaning should be readily apparent.

In the fifth expression, `ch1` is assumed to be a `char` type variable. This expression will be true if the character represented by `ch 1` comes before `T` in the character set, i.e., if the numerical value used to encode the character is less than the numerical value used to encode the letter `T`.

The last expression makes use of the `char`-type variable `letter`. This expression will be true if the character represented by `letter` is something other than `x`.

In addition to the relational and equality operators, C++ contains two logical connectives (also called logical operators), `&&` (AND) and `||` (OR), and the unary negative operator `!`. The logical connectives are used to combine logical expressions, thus forming more complex expressions. The negation operator is used to reverse the meaning of a logical expression (e.g., from true to false).

Conditional Expressions

The conditional expressions allow to choose the set-of-instructions for execution depending upon an expression's truth value. C++ provides two types of conditional statements : `if` and `switch`.

6.2 THE IF...ELSE CONSTRUCT

The `if` conditional construct is followed by a logical expression in which data is compared and a decision is made based on the result of the comparison.

Syntax

```
if (boolean_expr)  
{  
    statements;  
}  
else  
{  
    statements;  
}
```

6.3 BRANCHING: THE IF-ELSE STATEMENT

The `if-else` statement is used to carry out a logical test and then take one of two possible actions, depending on the outcome of the test (i.e., whether the outcome is true or false).

The `else` portion of the `if-else` statement is optional. Thus, in its simplest general form, the statement can be written as

```
if (expression) statement.
```

The expression must be placed in parentheses, as shown. In this form, the statement will be executed only if the expression has a nonzero value (i.e., if expression is true). If the expression has a value of zero (i.e., if expression is false), then the statement will be ignored.

The statement can be either simple or compound. In practice, it is often a compound statement which may include other control statements.

Example

Consider the following program:

```
#include <iostream.h>

int main()
{
    char chr;
    cout << "Please enter a character:" ;
    cin >> chr;
    if (chr == 'X')
        cout << endl << "The character is X";
    else
        cout << endl << "The character is not X";
    return 0;
}
```

Note that the operator, ==, used for comparing two data items, is different from the assignment operator, =,

In an if...else block of statements, the conditions is evaluated first. If the condition is true (value is non-zero), the statements in the immediate block are executed, if the condition is false (value is zero) the statements in the else block are executed.

In the above example, if the input character is 'X', the message displayed is, 'the character is X' otherwise, the message, 'The character is not X' is displayed . Also, note that the condition has to be specified within parenthesis.

If, however, the operator, =, is used instead of the operator, ==, the statement is executed as an assignment. For example, if the statement, if (chr == 'X'), was written as, if (chr = 'X'), then, chr would have been assigned the value, 'X' and the condition would be evaluated as true. Thus, the message, the character is 'X' would have been displayed, regardless of the input.

This program could also have been written as given below:

```
#include <iostream.h>

int main()
{
    char chr;
    cout << "Please enter a character: ";
    cin >> chr;
    if (chr != 'X')
        cout << endl << "The character is no X";
    else
        cout << endl << "The character is X";
    return 0;
}
```

The operator, !=, is an equivalent of not equal to, and is written without any space between != and =.

The general form of an if statement which includes the else clause is

If (expression) statement 1 else statement 2

If the expression has a nonzero value (i.e., if expression is true), then statement 1 will be executed. Otherwise (i.e., if expression is false), statement 2 will be executed.

The following, for example, program uses a nested if...else construct to check if the input character is uppercase or lowercase.

```
#include <iostream.h>
int main()
{
    char inp;
    cout << "Please enter a character: ";
    cin >> inp;
    if (inp >= 'A'
        if (inp <= 'Z')
            cout << endl << "Uppercase";
        else if (inp >= 'a')
        {
            if (inp <= 'z')
                cout << endl << "Lowercase";
            else
                cout << endl << "Input character > z";
        }
        else
        {
            cout << endl << "input character > z but less than a";
        }
    else
    {
        cout << endl << "Input character less than A";
    }
    return 0;
}
```

The following program shows how the clarity of steps can be increased by using braces ({and}) for enclosing parts of the code that follow an if or an else statement.

```
#include <iostream.h>
int main()
{
    char inp;
    cout << "Enter a character: ";
    cin >> inp;
```

```

if (inp >= 'A')
{
    if (inp <= 'z')
    {
        cout << endl << "Uppercase";
    }
    else if (inp >= 'a')
    {
        if (inp <= 'z')
        {
            cout << endl << "Lowercase";
        }
    }
}
return 0;
}

if ( circle) {
    scanf ( ` &f` , & radius);
    area = 3. 14159 * radius * radius;
    printf ( * Area of circle = &f* , area);
}
else {
    scanf ( `&f   &f` , & length , & width) ;
    area = length * width;
    printf ( * Area of rectangle = &f` , area);
}

```

The above example shows how an area can be calculated for either of two different geometric figures. If circle is assigned a nonzero value, however, then the length and width of a rectangle are read into the computer, the area is calculated and then displayed. In each case, the type of geometric figure is included in the label that accompanies the value of the area.

Example

Here is an example of three nested if - else statements.

```

If ( ( time >= 0.) && ( time < 12.) ) printf ( * Good Morning*);
else if ( ( time >= 12.) && ( time < 18.) ) printf ( * Good Afternoon *);
else if ( ( time >= 18.) && ( time < 24.) ) printf ( * Good Evening *);
else printf ( * Time is out of range *);

```

This example causes a different message to be displayed at various times of the day. Specifically, the message Good Morning will be displayed if time has a value between 0 and 12 ; Good Afternoon will be displayed if time has a value between 12 and 18; and Good Evening will be displayed if time has a value between 18 and 24. An error message (Time is out of range) will be displayed if the value of time is less than zero, or greater than or equal to 24.

The following program uses a cascading if . . . else construct to determine if the input character is a vowel, else it prints an appropriate message.

```
#include <iostream.h>

int main ()
{
    char in_chr;

    cout << " Enter a character in lowercase " <, endl;

    cin >> in_chr;

    if ( in_chr == `e`)
        cout << endl << " Vowel  a" << endl;

    else if ( in_chr == `e`)
        cout <, endl << "Vowel  e" <, endl;

    else if ( in_chr == `i )
        cout << endl << " Vowel  i " << endl);

    else if ( in_chr == `o ` )
        cout <, endl << " Vowel  o " << endl;

    else if ( in_chr == `u`)
        cout <, endl << " Vowel u" << endl;

    else
        cout << endl << " The character is not a Vowel" << endl;

    return o;
}
```

The switch statement

C++ provides a multiple-branch selection statement known as switch. This selection statement successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed.

The syntax is:

Switch (expression)

```
{
    Case Constant 1: Statement sequence 1; break;
    Case Constant 2: Statement sequence 2; break;
    Case constant-1: Statement sequence n-1; break;
    [default: Statement sequence n];
}
```

The expression is evaluated and its values are matched against the values of the constants specified in the case statements. When a match is found, the statement sequence associated with that case is executed unit the break statement or the end of switch statement is reached. The default statement gets executed when no match is found. The default statement is optional and, if it is missing, no action takes place if all matches fail.

A case statement cannot exit by itself, outside of a switch, the break statement, used under switch, is one of C++ jump statements. When a break statement is encountered in a switch statements, program execution jumps to the line of code following the switch statement i.e., outside the body of switch statement.

Eg: Program input number of week's day (1-7) and translate it to its equivalent name. (e.g. 1 to Sunday, 2 to Monday ———)

```
#include <iostream.h>

int main ( )
{ int. dow;

  Cout << "Enter number of weeks day (1-7):";

  Cin >> dow;

  Switch (dow)
  {

    Case 1: cout<<"\n" << "Sunday"; break
    Case 2: cout<<"\n" << "Monday"; break;
    Case 3: cout<<"\n" << "Tuesday";
    Case 4: cout<<"\n" << "Wednesday";
    Case 5: cout<<"\n" << "Thursday";
    Case 6: cout<<"\n" << "Friday";
    Case 7: cout<<"\n" << "saturday";

    Default: cout<<"\n" << "wrong number of day";

  }

  return 0;

}
```

Student Activity 1

1. What is the significance of a test condition in an if statement?
2. Write the syntax of an if-else statement.
3. What will be the output of the following code fragment:

```
Int year;
Cin >> year;
If (year % 100 ==0)
If (year % 400 ==0)
Cout << "LEAP";
Else cout << "Leap";
Cout << "not century year";

If the input given is (i) 2000 (ii) 1900 (iii) 1971?
```

4. What is the significance of a break statement in switch statement?

6.4 LOOPING: THE WHILE STATEMENT

The while statement is used to carry out operations, in which a group of statements is executed repeatedly, till a condition is satisfied.

The general form of the while statement is

```
while (expression) statement
```

The statement will be executed repeatedly, as long as the expression is true (i.e., as long as expression has a nonzero value). This statement can be simple or compound, though it is usually a compound statement. It must include some feature that eventually alters the value of the expression, thus providing a stopping condition for the loop.

Consecutive Integer Quantities

In this example, we try to display the consecutive digits 0,1,2, 9.

```
#include <iostream.h>

main ( )    /* display the integers 0 through 9 */
{
    int digit =0;
    while ( digit <= 9) {
        cout<<digit<<endl;
        ++ digit;
    }
}
```

Initially, a digit is assigned a value of 0. The while loop then displays the current value digit, increase its value by 1 and then repeats the cycle, until the value of digit exceeds 9. The net effect is that the body of the loop will be repeated 10 times, resulting in 10 consecutive lines of output. Each line will contain a successive integer value, beginning and ending with 9. Thus, when the program is executed, the following output will be generated.

```
0
1
2
3
4
5
6
7
8
9
```

This program can be written more conspicuity as

```
#include <iostream.h>

main ( )    /* display the integers 0 through 9 */
{
    int digit = 0;
    while (digit <= 9) {
        cout<<digit<<endl;
        digit += 1;
    }
}
```

when executed, this program will generate the same output as the first program.

Example

The following code generates the Fibonacci series between 1 and 100 . In this series, each number is the sum of its two preceding numbers. The series starts with 1.

```
#include <iostream.h>

int main ( )
{
    int num1 = 1, num2 = 1;
    cout << num1 << endl;
    while ( num2 < 100)
    {
        cout << num2 << endl;
        num2 += num1;
        num1 = num2 - num1;
    }
    return 0;
}
```

Output

1
1
2
3
4
5
8
13
21
34
55
89

As you can see, each number in the series is the sum of the two preceding numbers.

The Break Statement

The break statement causes the program flow to exit the body of the while loop. The following program code illustrates the use of the break statement.

```
#include <iostream.h>

int main()
{
    int num1 = 1, num2 = 1;
    cout << num1 << endl;
    while (num2 < 150)
    {
        cout << num2 << endl;
```

```
        num2 += num1;  
        num1 = num2 - num1;  
        if (num2 == 89)  
            break  
    }  
    return 0;  
}
```

The output of the previous program is:

```
1  
1  
2  
3  
5  
8  
13  
21  
34  
55
```

The control exits the loop when the condition, `num2 == 89`, becomes true.

6.5 MORE LOOPING: THE DO-WHILE STATEMENT

Sometimes, however, it is desirable to have a loop with the test for continuation at the end of each pass. This can be accomplished by means of the do-while statement.

The general form of the do-while statement is

```
do statement while (expression);
```

The statement will be executed repeatedly, as long as the value of expression is true (i.e., is nonzero). Notice that the statement will always be executed at least once, since the test for repetition does not occur until the end of the first pass through the loop. The statement can be either simple or compound, though most applications will require it to be a compound statement. It must include some feature that eventually alters the value of expression so the looping action can terminate.

Here is another example to do the same thing, using the to display the consecution digits (0, 1, 1, 2, 9) using the do while loop.

```
#include <studio.h>  
  
main() /* display the integers 0 through 9 */  
{  
    int digit = 0;  
    do {  
        printf ("%d\n", digit++);  
    }while (digit <= 9);  
}
```

This program below accepts characters from keyboard until the character, 'I', is entered and displays whether the total number of consonant characters entered is more than, less than, or equal to the total number of vowels entered.

```
#include <iostream.h>

int main ( )
{
    int constant, vowel ;
    char inp ;
    consonant = vowel 0 ;
    do
    {
        inp = ` ;
        cout << endl << " Enter a character ( ! to quit ) ";
        cin >> inp;
        switch ( inp)
        {
            case `A`:
            case `a`:
            case `E` :
            case `e` :
            case `I`      :
            case `i`      :
            case `O` :
            case `o` :
            case `U` :
            case `u` :      vowel = vowel + 1;
                           break ;
            case `!` :      break;
            default      :      consonant = consonant + 1;
        }
    } while ( inp != `!`)
        if ( constant > vowel )
            cout << endl << " Consonant count greater than vowel count" ;
        else if ( consonant < vowel )
            cout << endl < " Vowel count greater than consonant count";
        else
            cout << endl << " Vowel ! and consonant counts are equal ";
    return 0;
}
```

The do-while loop displays the current value of digit, increases its value by 1, and then tests to see if the current value of digit exceeds 9. If so, the loop terminates, otherwise, the loop continues, using the new value of digit. Note that the test is carried out at the end of each pass through the loop. The net effect is that the loop will be repeated 10 times, resulting in 10 successive lines of output.

Calculation for the average of numbers

```
/* calculate the average of n number */
#include <stdio.h>

main ()
{
    int n, count = 1;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf ( "How many numbers ?");
    scanf ( "%d" , %n);
    /* read in the numbers 8/
    do {
        printf ( 8 x = 8);
        scanf ( "%f" , %x);
        sum += x;
        ++ count ;
    } while ( count <= n);
    /* calculate the average and display the answer*/
    average = sum /n
    printf ( " \n The average is %f \n ",average);
}
```

6.6 STILL MORE LOOPING: THE FOR STATEMENT

The for statement includes an expression that specifies an initial value for an index, another expression that determines whether or not the loop is continued, and a third expression that allows the index to be modified at the end of each pass.

The general form of the for statement is

For (expression 1; expression 2; expression 3) statement

where expression 1 is used to initialize some parameter (called an index) that controls the looping action, expression 2 represents a condition that must be true for the loop to continue execution, and expression 3 is used to alter the value of the parameter initially assigned by expression 1. Typically , expression 1 is an assignment expression, expression 2 is a logical expression and expression 3 is a unary expression or an assignment expression.

When the for statement is executed, expression 2 is evaluated and tested at the beginning of each pass through the loop, and expression 3 is evaluated at the end of each pass. Thus, the for statement is equivalent to

```
expression 1;
while (expression 2) {
    statement
    expression 3;
}
```

The looping action will continue as long as the value of expression 2 is not zero, that is as long as the logical condition represented by expression 2 is true.

Example**Consecutive Integer Quantities**

The program will display the consecutive integers 0 : 2... 9 using for loop

```
#include <stdio.h>

main () /* display the numbers 0 through 9 */
{
    int digit;
    for ( digit = 0, digit <= 9; ++digit)
        printf ( "%d\n", digit);
}
```

The first line of the for statement contains three expressions, enclosed in parentheses. The first expression assigns an initial value 0 to the integer variable digit; the second expression continues the looping action as long as the current value of digit does not exceed 9 at the beginning of each pass, and the third expression increases the value of digit by 1 at the end of each pass through the loop.

Student Activity 2

1. What is the difference between a while and do-while loop?
2. Write a while loop that displays numbers 2, 4, 6, 8, ..., 18, 20.
3. Write an equivalent while loop for the following for loop:

```
For (int I=2, sum = 0; I <=20, I=I+2)
    Sum += I;
```
4. The break statement causes an exit from -----.
5. A continue statement within a loop causes control to go to -----.
6. What are jump statements?

6.7 IDENTIFIER SYNTAX

Identifiers are names that are given to various program elements, such as variables, functions and arrays. Identifiers consist of letters and digits, in any order, except that first character must be a letter. Both upper and lowercase letters are permitted, though common usage favors the use of lowercase letters for most types of identifiers. Upper and lowercase letters are not interchangeable (i.e., an uppercase letter is not equivalent to the corresponding lowercase letter.) The underscore character (_) can also be included, and is considered to be a letter. An underscore is often used in the middle of an identifier. An identifier may also begin with an underscore, though this is rarely done in practice.

Example 1

The following names are valid identifiers.

a a12 sum _ 1

The following names are not valid identifiers for the reasons stated.

"Y" illegal characters (")
 order - no illegal characters (-)

An illegal character can be arbitrarily long. Some implementations of C recognize only the first eight characters, though most implementations recognize more (typically, 31) characters. Additional characters are carried along for the programmer's convenience.

As a rule, an identifier should contain enough characters so that its meaning is easy to understand. On the other hand, an excessive number of characters should be avoided.

There are certain reserved words, called keywords, that have standard, predefined meanings in C. These keywords can be used only for their intended purpose; they cannot be used as programmer-defined identifiers.

The standard keywords are:

auto	extern	sizeof
break	floatn	static
case	for	sturct
char	goto	switch
const	if	typedef
continue	int	union
default	long	unsigned
do	register	void
double	return	volatile
else	short	while
enum	signed	

Some compilers may also include some or all of the following keywords:

ada	far	near
asm	fortran	pascal
entry	huge	

Some C compilers may recognize other keywords. Consult a reference manual to obtain a complete list of keywords for your particular compiler.

Note that the keywords are all lowercase. Since uppercase and lowercase characters are not equivalent, it is possible to utilize an uppercase keyword as an identifier. Normally, however, this is not done, as it is considered a poor programming practice.

6.8 EXPRESSIONS

An expression represents a single data item, such as a number or a character. The expression may consist of a single entity, such as a constant, a variable, an array element or a reference to a function. It may also consist of some combination of such entities interconnected by one or more operators.

Expressions can also represent logical conditions that are either true or false. However, in C the conditions true and false are represented by the integer values 1 and 0, respectively. Hence logical-type expressions really represent numeral quantities.

Example

Several simple expressions are shown below.

```
x + y
a = b
z = x + y
a <= b
```

```
a == b
++ 1
```

The first expression involves use of the addition operator (+). This expression represents the sum of the values assigned to the variables x and y.

The second expression involves the assignment operator (=). In this case, the expression causes the value represented by b to be assigned to a.

In the third line, the value of the expression (x + y) is assigned to the variable z. Note that this combines the features of the first two expressions (addition and assignment).

The fourth expression will have the value 1 (true) if the value of a is less than or equal to the value of b. Otherwise, the expression will have the value 0 (false). In this expression, <= is a relational operator that compares the values of the variables a and b.

The fifth expression is a test for equality (compare with the second expression, which is an assignment expression). Thus, the expression will have the value 1 if a is equal to the value of b. Otherwise, the expression will have the value 0 (false).

The last expression causes the value of the variable i to be increased by 1 (i.e., incremented). Thus, the expression is equivalent to

```
i = i + 1
```

The operator ++, which indicates incrementing, is called a unary operator because it has only one operand.

The C language includes many different kinds of operators and expressions.

6.9 STATEMENTS

A statement causes the computer to carry out some action. There are three different classes of statements in C. They are expression statements, compound statements and control statements.

An expression statement consists of an expression followed by a semicolon. The execution of an expression statement causes the expression to be evaluated.

Example

Several expression statements are shown below.

```
x = 3;
z = x + y;
++i;
printf("Area = %f", area);
;
```

The first two expression statements are assignment-type statements. Each causes the value of the expression on the right of the equal sign to be assigned to the variable on the left. The third expression statement is an incrementing-type statement, which causes the value of i to increase by 1.

The fourth expression statement causes the printf function to be evaluated. This is a standard C library function that writes information out of the computer. In this case, the message Area = will be displayed, followed by the current value of the variable area. Thus, if area represents the value 100, the statement will generate the message.

```
Area = 100
```

The last expression statement does nothing, since it consists of only a semicolon. It is simply a mechanism for providing an empty expression statement in places where this type of statement is required. Consequently, it is called a null statement.

A compound statement consists of several individual statements enclosed within a pair of braces { }. The individual statements may themselves be expression statements, compound statements or control statements. Thus, the compound statement provides a capability for embedding statements with other statements. Unlike an expression statement, a compound statement does not end with a semicolon.

Example

A typical compound statement is shown below.

```
{  
    pi = 3.141593;  
    circumference = 2 * pi * radius;  
    area = pi * radius * radius;  
}
```

Student Activity 3

1. What is an identifier?
2. What are expressions?
3. What are statements in C++?

6.10 SUMMARY

Statements are the instructions given to the computer to perform any kind of action. The simplest statement is null statement (;). C++ provides two types of selection statements: if and switch.

The if-else statement tests an expression and depending upon its truth value one of the two sets-of-actions is executed. The if-else statement can have another if statement.

C++ provides one more selection statement known as switch that tests a value against a set of integer constants (that includes character also). A switch statement can be nested also.

C++ provides three loops: for, while and do-while. The for loop gathers all its loop control elements on the top of the loop. A for loop can have multiple initializations and update expressions separated by commas. The loop control elements in a for loop are optional. The while loop evaluates a test-expression before allowing entry into the loop. The do-while is executed at least once always as it evaluates the test expression at the end of the loop.

A goto statement transfers the program control anywhere in the program. A break statement can appear in any of the loops and a switch statement. Whichever statement it appears in, it causes the termination of the statement there and then and the control passes over to the statement following the loop containing break.

The continue statement abandons the current iteration of the loop by skipping over the rest of the statements in the loop-body. It immediately transfers control to the evaluation of the test expression of the loop for the next iteration of the loop.

6.11 KEYWORDS

Infinite loop: A loop that never ends

Nested loop: A loop that contains another loop inside body.

Statement: Instruction given to the computer to perform any kind of action.

Loop: A group of instructions the computer executes repeatedly until some terminating condition is satisfied.

Identifiers: Names that are given to various program elements, such as variables, functions and arrays.

Expressions: An expression represents a single data item, such as a number or a character. The expression may consist of a single entity, such as a constant a variable, an array elements or a reference to a functions.

Statements: A statements causes the computer to carry out some action.

6.12 REVIEW QUESTIONS

1. What is iteration construct?
2. What is the purpose of break statement?
3. What is conditional statement?
4. Can if statement be nested?
5. What is the purpose of for loop?
6. What is nested if statement?
7. Can a switch statement be nested?
8. What is abnormal exit?
9. Which is the best loop in C++?
10. Find the syntax error (s), if any, in he following programs:

```
#include<iostream.h>
{
int x, y;
cin>>x;
for (y = 0; y< 10, Y++)
if x == y
cout << Y+X;
else
cout>>Y;
```

11. Given the following for loop:

```
Coust int sz= 25;
For (int I = 0, sum = 0; I < sz; I++)
Sum += I;
Cout << sum;
```

Write the equivalent whiel loop for the above code.

12. Write a C++ program to print fibonacci series i.e. 0, 1, 1, 2, 3, 5, 8----
13. Fill in the blanks:

- (a) The ____statements is used to carry out a logical test and then take one of two possible actions, depending on the outcome of the test.
- (b) In an If-else block of statements, the ____is evaluated first.
- (c) The operator____is an equivalent of not equal to.

- (d) The _____statement is used to carry out operations, in which a group of statements is executed repeatedly, till a condition is satisfied.
- (e) The first line of the for statement contains_____expressions, enclosed in _____.
- (f) _____are names that are given to various program elements.

Answers

13. (a) If- else (b) condition
(c) != (d) while
(e) three, parentheses (f) Identifiers

6.13 FURTHER READINGS

Herbert Schildt, *The complete Reference-C++*, Tata Mc Graw Hill

Robert Lafore, *Object Oriented Programming in Turbo C++*, Galgotia Publications.

E. Balaghrusamy, *Object Oriented Programming through C++*, Tata McGraw Hill.

UNIT

7

DESIGN ISSUES

LEARNING OBJECTIVES

After studying this unit, you should be able to:

- Define SAD
- Describe procedural approach to solve a problem.
- Describe object-oriented approach
- Describe the environment of a system.
- Understand the relationship between SDLC and modelling language.

UNIT STRUCTURE

- 7.1 Introduction
- 7.2 SAD (Software Analysis & Design)
- 7.3 Object-Oriented Analysis and Design Issues
- 7.4 Object-Oriented Approach
- 7.5 The Big Picture
- 7.6 System Users
- 7.7 Context of a system
- 7.8 The Environment of a system
- 7.9 Relationship between SDLC and Modelling Language
- 7.10 Depicting Elaboration
- 7.11 Depicting Construction
- 7.12 Diagrammatic Conventions
- 7.13 Depicting Implementation
- 7.14 Depicting Transition
- 7.15 Summary
- 7.16 Keywords
- 7.17 Review Questions
- 7.18 Further Readings

7.1 INTRODUCTION

The system objectives outlined during the feasibility study serves as the basis from which the work of system design is initiated. Much of the activities involved at this stage is of technical nature requiring a certain degree of experience in designing system, sound knowledge of computer related technology and through understanding of computers available in the market and the various facilities provided by the vendors. Nevertheless, a system cannot be designed in isolation without the active involvement of the user. The user has a vital role to play at this stage too. As we know that data collected during feasibility study will be utilized systematically during the system design. It should, however, be kept in

mind that detailed study of the existing system is not necessarily over with the completion of the feasibility study. Depending on the plan of feasibility study, the level of detailed study will vary and the system design stage will also vary in the amount of investigation that still needs to be done. Sometimes, but rarely, the investigation may form a separate stage between feasibility study and computer system design. Designing a new system is a creative process which calls for logical as well as lateral thinking. The logical approach involves systematic moves towards the end-product keeping in mind the capabilities of the personal and the equipment. This is to ensure that no efforts are being made to fit previous solutions into new situations.

7.2 SAD (SOFTWARE ANALYSIS & DESIGN)

SDLC Phase – Software Development Life Cycle as described by Grady Booch, James Rumbaugh and Ivan Jacobson, consists of four phases that any software intensive system goes through. These are explained below –

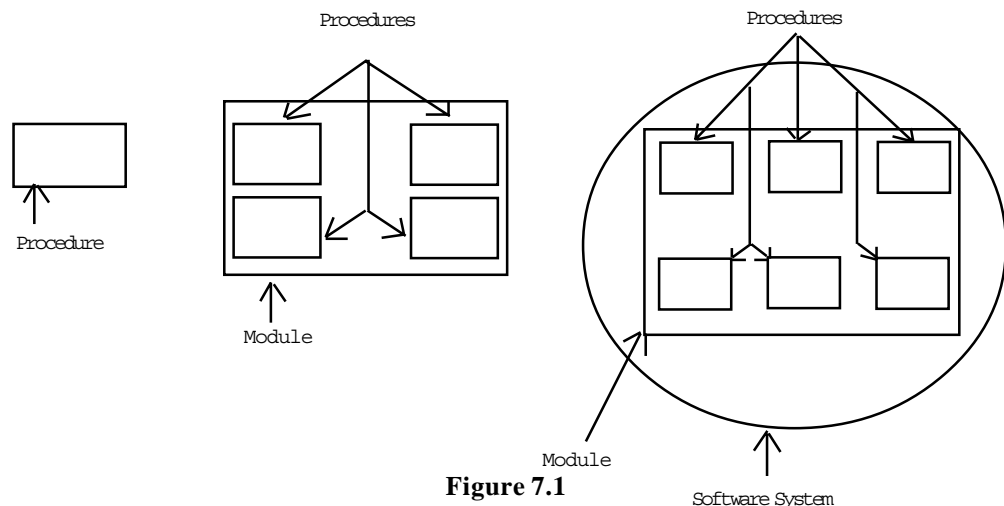
1. **Inception:** This is the first phase. The idea of a new system envisaged at this stage.
2. **Elaboration:** This is the phase where system vision and architecture are defined.
3. **Construction:** At this stage, the software is coded to form an executable base which is ready to be deployed. The software is constantly tested to meet the requirement.
4. **Transition:** This is the last phase in which the software is handed over to the end user and is tested for defects. Defects identified are then documented and corrected.

Iteration is the activity that is common to all phases of the SDLC. Each phase consists of one or more iteration cycles.

7.3 OBJECT-ORIENTED ANALYSIS AND DESIGN ISSUES

Procedural Approach

The best way to solve a problem is to first break down into smaller parts and then solve each of the smaller part. The solutions are then aggregated to solve overall problem. Each small part or problem is considered to be a task. To perform a task, a block code procedure, is written. Procedures can be grouped together to form a module. Any one of the procedure can be called from another one. All modules integrated together form the software system.



In the procedural approach, portions of the code are so interdependent that the code in one

application cannot be reused in another.

People have often asked why software cannot be constructed the way an aircraft or a high-rise building is constructed – by putting together several small constituent components to build the whole. In attempt to solve the problems inherent to the procedural approach, programmers turned to the object-oriented approach.

7.4 OBJECT-ORIENTED APPROACH

According to a Grady Booch, a leading exponent of the object-oriented approach, an object has the following characteristics –

1. It has a state.
2. It may display a behavior.
3. It has a unique identity.

The state to an object is indicated by set of attributes and their values. The behavior refers to the change of these attributes over a period of time.

Most of the modern languages and OS use and support object-orientation. For example popular languages like C ++, Visual C ++ and Java.

There are some standard Modelling Language used for modelling software systems of varying complexities. Systems can range from enterprise information systems to distributed Web-based systems. There are set of notations, and rules using the same. The focus is on creating simple, well-documented and easy to understand software models.

7.5 THE BIG PICTURE

Modelling Language enables system engineers to create a standard blueprint of any system. The system thus represented is easy to understand and visualize. It provides a number of graphical tools. The tools can be used to visualize a system from different view points. Diagrams are used to present multiple views of a system. These multiple views of the system together represent the model of the system.

7.6 SYSTEM USERS

They can be human users or other systems that interact directly with the system. The role of the system user is defined, based on the functions performed.

7.7 CONTEXT OF A SYSTEM

A system addresses a problem and provides all possible solutions to the problem with a framework. This framework is referred to as context of a system. The context of the problem within the system must to defined before a solution is worked out.

7.8 THE ENVIRONMENT OF A SYSTEM

It specifies the functionality of a system from a user's point of view. The environment consists of elements that lie inside a system and are responsible for the working of that system, as well as the elements that lie outside of that system and what they expect the system to provide.

Student Activity 1

1. Describe the four phases of SDLC.
2. Describe the procedural approach to solve a problem.
3. What are the characteristic of an object?
4. Who are system users?
5. What do you mean by the environment of a system.

7.9 RELATIONSHIP BETWEEN SDLC AND MODELLING LANGUAGE

The notations provided by the Modelling Language are used to depict the various views of the system. Different views map to different phases of the SDLC. For example, in the elaboration phase, a set of diagrams may be used to depict the proposed design. In the construction phase, a different type of diagram may be used to depict software components.

7.10 DEPICTING ELABORATION

It represents the goals and objectives of various users and their expectation from the system. The view represents only that part of the system with which the user interacts.

In it the cases consist in the elements that lie inside the system and are responsible for the working, i.e., the functionality, and the behavior of the system. Therefore, the system performs to generate results requested by the users of the system. They represent all interactions that can take place between a user and the system.

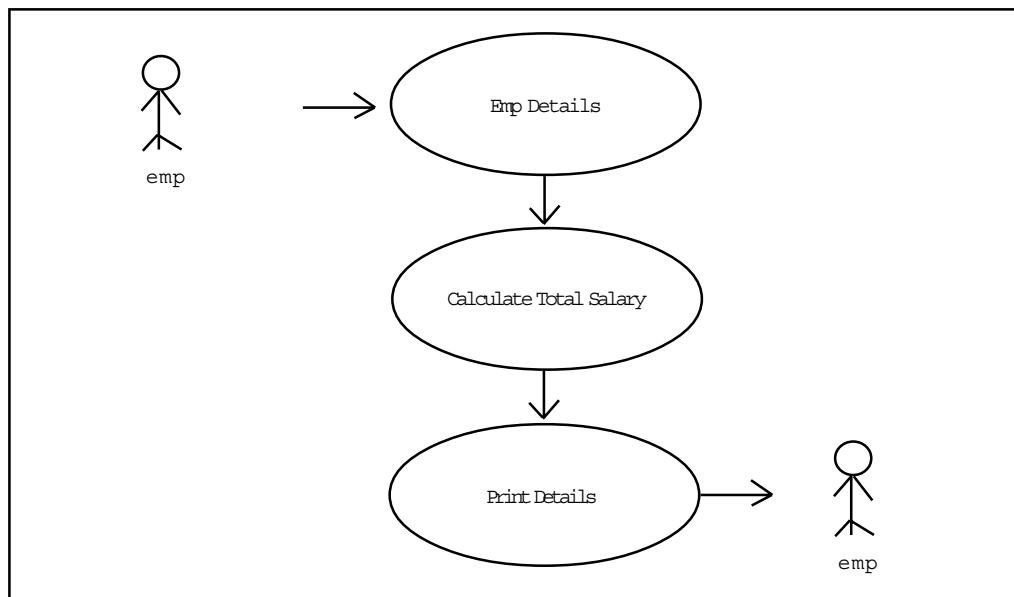


Figure 7.2: Elaboration diagram

7.11 DEPICTING CONSTRUCTION

The construction represents the static aspect of a system. It comprises class and object diagrams. Class diagram depicts various classes and their associations. Object diagrams depict various objects (instances) and their links with each other. Links are pathways of communication between objects.

A class diagram models the following:

- a. The information and associated behavior that is independent of its surroundings.
- b. The communication between the surroundings and its inner workings.

- c. Control behavior specific to one or more cases.

7.12 DIAGRAMMATIC CONVENTIONS

Representation

1

Meaning

represents only one

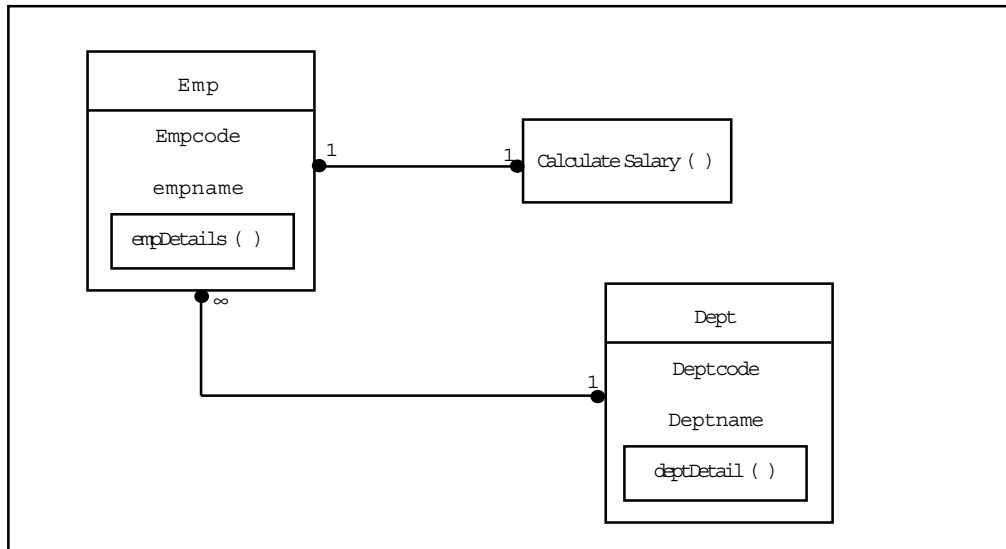
¥

represents more than one

1.....4

represents the range.

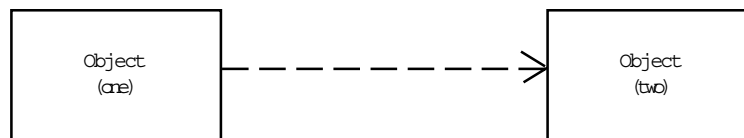
Construction Diagram



7.13 DEPICTING IMPLEMENTATION

It depicts various aspects of software system implementation. For example – source code, runtime implementation, and configuration management of software releases. Typically, a class contains a behavior that can be used across cases, projects and operating systems, the class can be mapped to a component, which are used to represent the implementation view of a system. They represent various components of a system and their relationships, for example – source code, object code and execution code. Basically implementation diagrams are used to model static implementation view of a system. These diagrams are used to depict the physical aspects of an object-oriented system. It actually helps in visualizing, specifying and documenting the component base of a system.

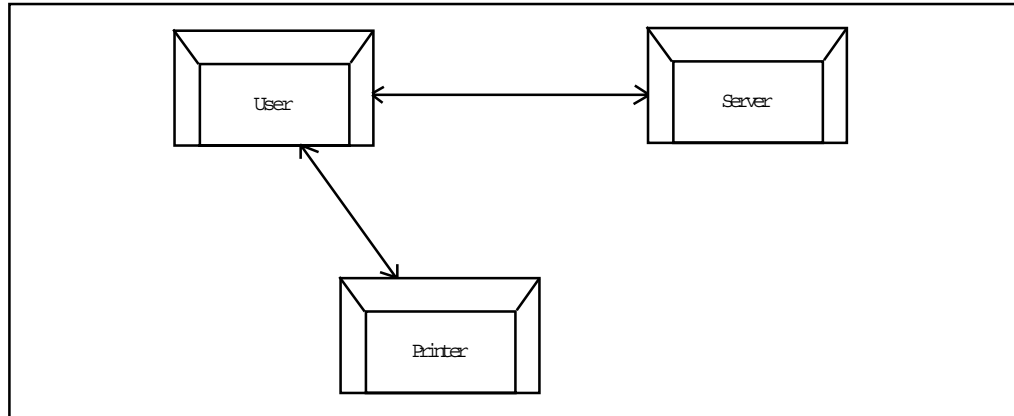
Implementation Diagram



7.14 DEPICTING TRANSITION

The transition depicts the physical distribution of the various components used in the system. This is also known as deployment. It depicts various nodes that form a part of the physical hardware requirement for the deployment of the system. Computers, printers, etc. are the typical examples of the nodes. The deployment or transition represents the distribution, delivery and installation of the component parts that constitute the physical aspect of the system.

Transition/Deployment Diagram



Student Activity 2

1. Give the elaboration diagram to describe the relationship between SDLC and modelling language.
2. Give the construction diagram to describe the relationship between SDLC and modelling language.
3. What does transition depict?

7.15 SUMMARY

Software development life cycle consists of four phases that any software intensive system goes through including Inception, Elaboration, Construction and Transition. In the procedural approach, portions of the code are so interdependent that the code in one application cannot be repeated in another.

An object has a state, may display a behaviour and has a unique identity. The state to an object is indicated by set of attributes and their values.

Modeling language enables system engineers to create a standard blueprint of any system. A system address a problem and provides all possible solutions to the problem with a framework.

The notations provided by the modelling Language are used to depict the various views of the system. The elaboration view represents only that part of the system with which the user interacts. The construction represents the static aspect of a system. Implimentation depicts various aspects of software system Implementation. The transition depicts the physical distribution of the various components used in the system. This is also known as deployment.

7.16 KEYWORDS

System Environment: It consists of elements that lie inside a system and are responsible for the working of that system, as well as the elements that lie outside of that system.

System Users: Human users or other systems that interact directly with the system.

Construction: Represents the static aspects of a system.

Transition: Depicts the physical distribution of the various components used in the system.

7.17 REVIEW QUESTIONS

1. Explain the four phases that any software intensive system goes through.
2. Briefly explain:
 - a. Procedural Approach;
 - b. Object-Oriented Approach;
 - c. Construction (with a diagram); and
 - d. Depicting Transition (with a diagram).
3. Fill in the Blanks
 - (a) SDLC phase consists of four phase i.e. — — — and —
 - (b) _____ enables System engineers to create a standard blueprint of any system.
 - (c) The _____ represents the static aspect of a system.
 - (d) _____ diagrams are used to model static implementation view of a system.
 - (e) The transition depicts the physical distribution of various components used in the system is known as_____.

Answers to Review Questions

3. (a) Inception, Elaboration, construction, Transition.
- (b) Modelling language.
- (c) Construction
- (d) Implementation
- (e) Deployment

7.18 FURTHER READINGS

Herbert Schildt, *The complete Reference-C++*, Tata Mc Graw Hill

Robert Lafore, *Object Oriented Programming in Turbo C++*, Galgotia Publications.

E. Balagurusamy, *Object Oriented Programming through C++*, Tata McGraw Hill.

UNIT

8

ARRAYS, LISTS, STACKS AND QUEUES

LEARNING OBJECTIVES

After studying this unit, you should be able to:

- Define arrays-notation, declaration and Initialization
- Describe sorting in array
- Describe getline () function
- Describe one dimensional, two dimensional and more than two dimensional arrays
- Define pointers, linked linked lists, -traversal and searching
- Describe Stacks and Queues

UNIT STRUCTURE

- 8.1 Introduction
- 8.2 One Dimension Arrays
- 8.3 Int Array Sorting
- 8.4 Strings - One-Dimensional char Arrays
- 8.5 Two or more Dimensional Arrays
- 8.6 Introduction to Pointers
- 8.7 Lists
- 8.8 Introduction to Linked Lists
- 8.9 Advantages of Linked Lists Over Arrays
- 8.10 Types of Linked Lists
- 8.11 Creating a Linked List Class
- 8.12 Stacks
- 8.13 Queues
- 8.14 Trees
- 8.15 Binary Trees
- 8.16 Implementing Binary Tree
- 8.17 Summary
- 8.18 Keywords
- 8.19 Review Questions
- 8.20 Further Readings

8.1 INTRODUCTION

Arrays are a data structure, which hold multiple variables of same data type. An array is a sequence of objects of which have the same type. The objects are called elements of array and are numbered consequently 0,1,2,..... These numbers are called index values or

subscripts of the array. The term 'subscript' is used because as a mathematical sequence, an array would be written with subscripts; n_0, n_1, n_2, \dots . These numbers locate the elements position within the array, thereby giving direct access into the array. In this chapter we will discuss to link data to specific variable names. A pointer can point to an unnamed data value. With pointers you gain a "different view" of your data.

The data structure is a process of how the data items are organized in the computer memory for the processing and for the subsequent usages. In other words, the data structure is a technique or method of study, how data items are interrelated, mathematically or logically.

This unit introduces the basic concepts of data structure. Data structure is an important concept for the programmer. Different types of data values are related to each other. To enable programmer to make use of the relationship these data values must be in an organized form. The organized collection of data is called data structure. So a data structure is:

Data structure = organized data + Allowed operations so, the data structure is the extension of data type. A data type is:

Data type = Data Values + operations.

A variable can store only one value at a time. In a situation where thousands of variables have to be stored it would not be feasible for you to define variables, each with a different name. The solution is to create one array variable, of the char data type, containing values. Each value in the array is known as an element of the array.

An array is a collection of elements of a single data type stored in adjacent memory locations. For example, you can have an array containing the salary of 30 employees in a company. Since the salaries are all positive integers, they can be stored in an array of the char data type. Each score is stored in separate array element and all the elements are stored consecutively in the computer memory.

Before an array variable can be initialized, its data type and dimension size (number of elements in the array) must be defined. The definition and the initialization of an array variable can take place within the same statement.

Each element of the array can be accessed by its subscript number. The subscript number specifies the position of an element within the array (also called the index of the element). The first element of an array has an index of 0, and the last element has an index one less than the dimension size of the array.

All the elements of an array must be of the same data type. Thus, there cannot be an array with five elements, of which three are char elements and two are float elements.

An array can have more than one dimension also of 30 students in four subjects char data type. The first dimension will contain 30 elements and the size of each dimension. The total number of elements in the array would be 120.

8.2 ONE DIMENSION ARRAYS

Syntax for defining an array variable with one dimension

```
<data_type> <variable_name>[<dimension_size>];
```

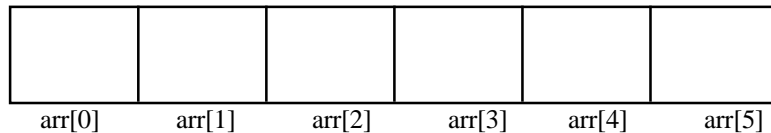
Examples

```
Char name [40];  
Float income [1000];
```

This example, shows one dimensional array:

```
char arr [6];
```

The size of the first dimension of the array is 6. When the array definition is executed, memory is allocated for the variable, arr. If a single char variable occupies 1 byte, then arr will occupy 6 bytes. These will be positioned one after another in the memory. Here is a schematic representation: 6

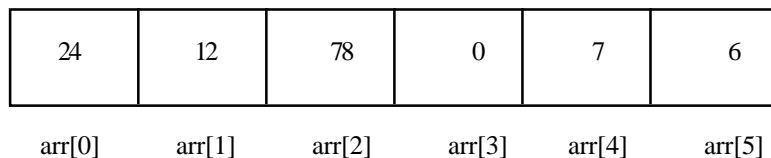


In the above diagram, arr is the name of array variable. The five elements of the array are arr [0], arr [1], and so on and 0,1,2,3,4,5 are the subscripts (or indices) of the elements. Note that the index starts with 0.

In order to initialize this array, you need to initialize the individual elements:

```
Arr [0] = 24;  
Arr [1] = 12;  
Arr [2] = 78;  
Arr [3] = 0;  
Arr [4] = 7;  
Arr [5] = 6;
```

The elements can be initialized in any order.



```
cin >> arr [2];    // the user enters a value to be stored in the  
                  // element with an index of 2
```

They can be initialized as

```
arr [5] = arr [0] + arr [3];
```

The array could also have been initialized at the time of definition as shown below:

```
char arr [5] = {24, 12, 78, 0, 7};
```

In case if you are initializing an array at the time of its definition, then it is not necessary to specify the dimension of the array

```
char arr[] = {24, 12, 78, 0, 7, 6};
```

Note: The dimension size of an array must be specified at the time of its definition unless if the initialization is in the same statement;

An array must be given a constant dimension size, which has to be at least 1 which means that a variable cannot be used to specify the dimension size of an array. For example,

```
#include <iostream>
#define CAPACITY 50
int main ()
{
    int var = 21
    const int x = 30;
    char arr1 [CAPACITY - 5];    // Valid definition - CAPACITY is a macro
    char arr2 [x];               // Valid definition - x is a const variable
    char arr2 [var];             // May not compile - dimension size should
                                // not be a non-const variable

    return 0;
}
```

An array cannot be initialized with another array. The only way to copy an array into another is to copy its individual elements:

```
int xy[2] = { 10,20}; ab [2]; // value
xy = ab                        // Error as array cannot be initialized
                                // with another array.

xy[2] = abc[1];               // valid
xy[0] = abc[2];               // valid
```

They cannot be an array of references

```
Int a,b,c; int x,y,z,
Int & rif = {x,y,z}; // error //
```

This cannot be an array of references

```
Int x, y, z
Int & ref = {x, y, z}; Error //
```

8.3 INT ARRAY SORTING

After initializing an integer array it should be sorted in ascending or descending order. There are various methods of sorting the array, e.g., Bubble method as "Bubble sort". To understand Bubble sort consider the following example,

```
Int array [3] = { 50, 3, 15}
```

The objective is to put values in ascending order. The sorting process has multiple steps, as stated below:

50	3	15
arr[0]	arr[1]	arr[2]

The basic aim is to put the value in ascending order which involves the following steps.

- Compare element, arr [0], with the rest of the elements. If arr [0] is larger than any element, swap the values:

```
int temp;
```



```
if (arr[0] > arr[1])
{
    temp = arr[0];
    arr[0] = arr[1]
    arr[1] = temp;
}

if (arr[0] > arr[2])
{
    temp = arr[0]
    arr[0] = arr[2]
    arr[2] = temp;
}
```

This result of this step would be:

3	50	15
arr[0]	arr[1]	arr[2]

1 Compare element, arr [1], with all the elements with an index greater than 1 in the array. If arr [1] is larger than any element, swap the values:

```
if (arr[1] > arr[2])
{
    temp = arr[1]
    arr[1] = arr[2]
    arr[2] = temp;
}
```

The result of this step would be (the array is now sorted):

3	15	50
arr[0]	arr[1]	arr[2]

arr [0] arr [1] arr [2]

- Compare the next element in the array with all the remaining elements. Continue this process until you reach the end of the array. In the above case, the dimension size of the array was small, hence it required only two passes to sort it. Larger arrays will require more number of passes.

Here is a complete program that takes in ten salaries and displays them in the descending order:

```
#include <iostream.h>

int main()
{
    float salaries [10], temp;

    // Enter all the 10 salaries
```

```

for (int ctr=0; ctr < size of(salaries)/sizeof(float); ctr++)
{
    cout << endl << "Enter salary of employee number EOO"
        << ctr + 1 << " : " ;
    cin >> salaries [ctr];
}
// sort the array
for (int k = 0; k < size of (salaries) / size of (float); k++)
{
    if (salaries [k] < salaries [j])
    {
        temp = salaries [k];
        salaries [k] = salaries [j];
        salaries [j] = temp;
    }
}
// Display the contents of the sorted array
cout << "The sorted salaries are " << endl;
for (int m = 0; m < size of (salaries)/size of (float); m++)
{
    cout << salaries [m] << end:
}
return 0;
}

```

8.4 STRINGS - ONE-DIMENSIONAL CHAR ARRAYS

A string constant is a one-dimensional array of characters terminated by a NUL ('\0'). For e.g.

```
Char s t r [ ] = "ROBERI";
```

Diagrammatically it can be represented as under

R	O	B	E	R	T	O
Str[0]	Str[1]	Str[2]	Str[3]	Str[4]	Str[5]	Str[6]

The C++ compiler automatically inserts the NUL character ('\0') at the end of a string, enclosed within double quotes (" "). The NUL character has the ASCII code, 0, and is not

the same as the digit 'O', which has the ASCII code, 48. The NUL character cannot be displayed on the screen. An array will be called a string if it terminates with a NUL character. The statement `cout << str;` will display "ROBERT". The object `cout` reads and displays all the characters until it reads the NUL character.

For example, if we define a character array as

```
Char arr[6]      =      {R,O,B,E,R,T};
```

And then give the following statement:

```
Cout    << arr;
```

In this case, C++ does not automatically append the NUL character in the array. The output could be:

```
ROBERT-----
```

The output contains junk characters after 'T'. This is because the `cout` object does not consider the length of the array when displaying the characters; it stops when it reads a NUL character. If the array does not contain a NUL character, the `cout` object keeps reading the memory locations. This may lead to abnormal lamination of program overcome by display of characters in the array.

```
For (int j = 0; j < size of (arr) / size of (char); j++)  
{  
    cout << arr [j];
```

would be the outcome

```
ROBERT
```

If you insert a NUL character in the array, `cout` will read the string only until and including the NUL character. For example,

```
#include <iostream.h>  
int main ( )  
{  
    char str [10] = { 'R' 'T' 'V' 'E' 'R' ' '\0' } ;  
    count << str ; // outputs RIVER  
    str [ 3 ] = '\0' ;  
    cout << endl << str; // outputs RIV  
    cout << endl << str; //outputs  
    return 0;  
}
```

Reversing a String

The following example, reverses a string and copies it into another:

In the code just given, the statements.

```
Pos = 0;  
While (source [pos] != '\0')  
{  
    pos = pos + 1 ;  
}
```

position the variable, pos, at the location of the NUL character in the array named sources. This statement could also have been written as:

```
pos = -1;
while (source [ ++pos ] ) ;
```

the statement, for (- - pos; pos >= 0 ; dest [j++] = source [pos - -]; , first position the variable, pos, on the character just before the NUL character. Then ,it copies individual characters into the array named dest. The value of the variable, pos, is decreased to traverse the array, dest, from the end to the start. It could also have been written as follows:

```
int j = 0;
pos = pos - 1;
while (pos >= 0)
{
    dest [j] = source [pos];
    j = j + 1;
    pos = pos - 1;
}
```

The statement, dest [j] = 0; is required because the array must terminate with a NUL character.

The Getline () Function of the C in Object

The cin object does not allow you to enter blank spaces from the keyboard. To solve this problem, the getline () function can be used. For example,

```
#include <iostream.h>
int main()
{
    char str1 [10] = str2 [10], str3 [10];
    cout << endl << Enter str1 (max 6 characters): " ;
    cin.getline (str1,7): cout << endl << Enter str2(max 9 characters):" ;
    cin.getline (str2, 10):
    cout << endl << Enter str3(max4 characters): " ;
    cin.getline (str3, 5):
    cout << endl << "The string str1 is : " << str1 << endl;
    cout << endl << "The string str2 is : " << str1 << endl;
    cout << endl << "The string str3 is : " << str1 << endl; return 0;}
}
```

The statement, cin.getline (str,7); , stores six character inside the array, str, including blank space. If the user types more than six characters, the first six characters are stored in the array, and the seventh character is the NUL character.

Syntax

```
cin.getline (<name_of_the_char_array>,<number_of_input_ charaters_to_store +1>);
```

The strlen() Function

This function returns the number of characters in a string, excluding the NUL character. To use this function, you have to include the file, cstring, in your program.

Syntax

```
strlen (<name_of_string>);
```

The following program shows how:

```
#include <iostream.h>
#include <cstring.h>

{
    char str1[] = "Corvus splendens";
    char str2[] = { 'C', 'R', '\0', 'O', 'W', '\0' };
    cout << endl << "Number of characters in str1: " << strlen (str1);
        // Output is 16
    cout << endl << "Number of characters in str2: " << strlen (str1);
        // Output is 2
    return 0;
}
```

The strcpy() Function

This function copies the contents of a source string into a destination string. If the size of the destination string is smaller than the source, the destination string is truncated. To use this function, would be included in the program the file, `cstring`,

Syntax

```
strcpy (<name_of_destination_string>,<name_of_source_string>);
```

The following program demonstrates the `strcpy ()` function:

```
#include <iostream.h>
#include <cstring.h>

int main()
{
    char source1[] = "Panthera Leo";
    char dest1[20];
    strcpy (dest1, source1);
    cout << endl << "source1 contains:"
        << source1; //Output is Panthera Leo
    cout << endl << "dest1 contains:"
        << dest1;  //Output is Panthera Leo
    return 0;
}
```

Note: The size of the destination must be greater or equal to the source array.

The strcat() Function

This function appends the contents of a source string into a destination string. The size of the destination string should be greater than the size of the source string. To use this truncated you have to included the file, `cstring`, in your program.

Syntax

Strcpy (<name_of_destination_string>,<name_of_source_string>);

The following program demonstrates the strcpy () function:

```
#include <iostream.h>
#include <cstring.h>
int main()
{
    char source1[] = "Panthera Leo";
    char dest1[40]; = "LION";
    strcat (dest1, source1);
    cout << endl << "source1 contains:"
        << source1; //Output is Panthera Leo
    cout << endl << "dest2 contains:"
        << dest1; //Output is Panthera Leo
    return 0;
}
```

Note: the size of the destination array must be by through to take the combined length of both the string.

The strcmp() function

This function compares two strings (supplied as its parameters) and returns an integer value. If the first string argument is less than the second, strcmp () returns a value less than zero. If the first argument is equal to the second, the function returns a value equal to zero. If the first argument is greater than the second, the function returns a value greater than zero.

Syntax

Strcpy (<string 1>,<string2>);

The comparison is performed by comparing the characters in the corresponding positions in the two strings. When the two characters differ, the function returns a value specifying that the first string is greater than, less than, or equal to the second depending on which of the characters has a greater ASCII value.

The following table illustrates the working of strcmp ()

Return Value	Meaning	Example
Less than zero	The ASCII value of the character of the first string is less than that of the corresponding character of the second string	inum= strcmp ("ABC", "abc"); will return -32 (the ASCII difference of 'A' and 'a')
Zero	The strings are identical	inum= strcmp (ABC", "ABC"); will return -zero.
More than zero	The ASCII value of the character of the first string is greater than that of the corresponding character of the second string	inum= strcmp ("veronica" , "Betty"); will return -32 (the ASCII difference of 'V' and 'B')

Example, showing the strcmp () function:

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

```
{
    char str1 [] = "ABCD";
    char str2 [] = "ABCD";
    cout << endl << strcmp (str1, str2);    // OUTPUT is 0 since both
                                              // strings are identical

    char str3 [] = "abcd";
    char str4 [] = "ABCD";
    cout << endl << strcmp (str3, str4);    // OUTPUT is 32 since ASCII code
                                              // of 'a' is 97
                                              // and that of 'A' is 65, hence
                                              // 97 - 65 = 32

    char str5 [] = "ABCD";
    char str6 [] = "abcd";
    cout << endl << strcmp (str5, str6);    // OUTPUT is -32
                                              // since 65 - 97 = -32

    char str7 [] = "ABCD";
    char str8 [] = "ABCD";
    cout << endl << strcmp (str7, str8);    // OUTPUT is 68 since
                                              // ASCII code of 'D' is 68 and the
                                              // fourth character in str8 is
                                              // NUL, hence 68 - 0 = 68

    return 0;
}
```

8.5 TWO OR MORE DIMENSIONAL ARRAYS

Syntax for two or more dimensional arrays:

```
<data_type> <variable_name>[<dimension1_size>][<dimension2_size>;
```

Examples

```
Char names_of_twenty_employees[30] [100];
```

```
Int employee_codes_of_five_employees_and_ their_ages[5] [2];
```

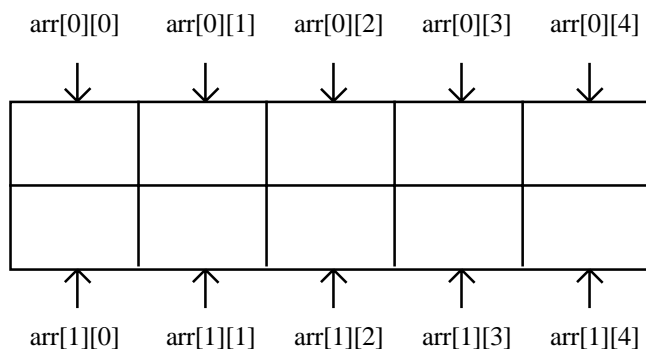
Two-Dimensional Character Arrays

Consider the following arrays shows two-dimensional character array

```
Char arr[2] [5];
```

the size of the first dimension of the array is 2 and the size of the second dimensions of the array is 5.

Total number of elements in the array is $2 * 5 = 10$. When the array is defined, memory is allocated for the variable, arr. If a single char variable occupies 1 byte, then arr will occupy 10 bytes. These bytes will be positioned one after another in the memory as shown in the design.



The array could be thought of as being composed of two one-dimensional arrays. The array can be initialized at the time of definition, thus:

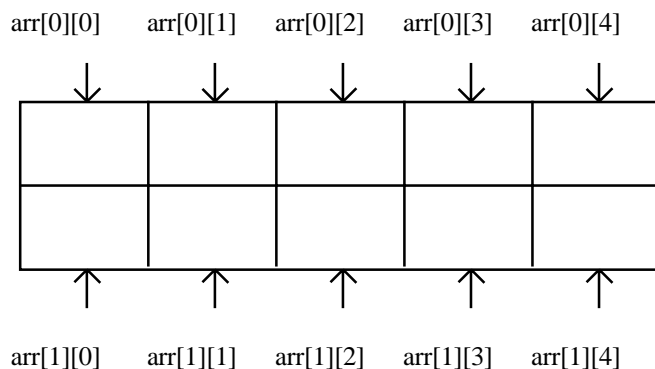
```
Char arr[] [5] = {
    "COW"
    "BULLS"
};
```

or the individual elements could be initialized thus:

```
char arr[2] [5] = {
    'C', 'O', 'W', '\0',,
    'B', 'U', 'L', 'L', '\0'
```

Notice how the element `arr[0] [4]` has been left uninitialized. While initializing the array at the time of its definition, it is not compulsory to specify the size of dimension `[0]`, but the sizes of the higher dimensions must be specified. The inner braces ('{ }') are not compulsory but are included for improved code readability.

The initialized array can now be visualized thus:



Now, the following statements will give the respective outputs:

```
Cout << arr[0]; // Output COW
Cout << arr[1]; // Output BULL
```

The element, `arr[0] [4]`, contains an undefined character.

A single element can be displayed thus,

```
Cout << arr; //0xbfffd3c or a similar hexadecimal number
```

The output is the address of the element, `arr[0] [0]`. You will read the explanation for this output in Section 4.

As mentioned earlier, the elements of a two-dimensional array are positioned one after another in memory. The `cout` object reads all the characters from the start of a row. It stops

reading as soon as it reads a NUL character. If the object cannot find a NUL character, it continues to read the next row.

The following program demonstrates this:

```
#include <iostream.h>

int main()
{
    char names[] [5] = {
        'T', 'o', 'm', 'm', 'y',
        'B', 'i', 'r', 'd', '\0'
    }

    cout << "The string in names[0] is: << names[0] << endl;
    cout << "The string in name[1] is: << names[1] << endl;
    cout << "The character in names[1] [2] is: " << names[1] [2] << endl;
    return 0;
}
```

The output is:

The string in names[0] is: TomyBird

The string in names[1] is: Bird

The character in names [1] [2] is: r

The following program takes 10 names, each having a maximum size of 30 characters, from the user and sorts them:

```
#include <iostream.h>
#include <cctype.h>

int main()
{
    char students[10][31], temp[31];
    int ctr = 0, row = 0, col = 0;
    char replay = 'Y';
    cout << "Please enter a maximum of 30 characters per name";
    do
    {
        cout << endl << "student number " << ctr + 1 << ":";
        cin >> students[ctr];
        ctr++;
        cout << endl << "Do you wish to enter another name? (y/n):";
        cin >> rely;
    } while (toupper(replay) == 'Y' && ctr < 10);
    // Sort the array
    for (int row = 0; row < ctr; row++)
    {
        if (strcmp (students[row], students[j]) > 0)
```

```

        {
            strcpy(temp,students[row]);
            strcpy(students[row],students[j]);
            strcpy(students[j],temp);
        }
    }
}

// Display all the names
for(row = 0; row < ctr; ++row)
{
    cout << students[row] << endl;
}

return 0;
}

```

Two-Dimensional Arrays of Data Types other than Character

Here is an example of initialization of a two-dimensional array of the double data type:

```

#include <iostream.h>
#include <iomanip.h>
#include <cmath.h>
#include <cstdlib.h>
int main()
{
    double randoms[5][10], tot = 0.00;
    stand(time(NULL));
    for(int r=0; r < 5; ++r)
    {
        for(int c = 0; c < 10; c++)
        {
            tot += randoms[r][c] = rand();
            cout << setw(20) << randoms[r][c];
        }
    }
    cout << endl << "The average is" << tot / sizeof(randoms);
    return 0;
}

```

The function, `random()`, generates a random number. This generator must be initialized with `srand()`. The function, `time()`, returns the time since January 1, 1970, measured in seconds.

Arrays with More than Two Dimensions

This is how you will initialize an integer array having three dimensions:

```

#include <iostream.h>
int main()

```

```
{
    int coordinates[10][10][10], x,y,z;
    for (x = 0; x < 10; ++x)
    {
        for(y = 0; y < 10; ++y)
        {
            for(z = 0; z < 10; ++z)
            {
                coordinates[x][y][z] = 0;
            }
        }
    }
    return 0;
}
```

Suppose you want to store the names and addresses of three students. You can do so in a 3-dimensional array, as shown in the example below:

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
    char details[3][2][50] = {
        { "Hector McQuinn",
          "New Orient Street, Denver"
        },
        { "Megan Bernard",
          "ABC Road, Trainsville"
        },
        { "Simon Doyle:",
          "Nile Villa, Shipworks"
        }
    }
    for(int page = 0; page < 3; ++page)
    {
        cout << endl << setw(40) << setfill('*')
              << "PAGE NUMBER" << page+1;
        for (int line = 0; line < 2; ++line)
        {
            cout << endl << "LINE" << line + 1
                  << ":" << details [page] [line];
        }
    }
    return 0;
}
```

The output of the previous program is:

*****PAGE NUMBER 1

LINE 1:Hector McQuinn

LINE 3:New Orient Street, Denver

*****PAGE NUMBER 2

LINE 1:Megan Bernard

LINE 2: ABC Road, Trainsville

*****PAGE NUMBER 3

LINE 1:Simon Doyle

LINE 2: Nile Villa, Shipworks

You can think of a three-dimensional character array as a collection of pages - each page having multiple lines and each line having multiple characters. Hence, you can imagine the character array, `char details[3][2][50]`, as having three pages, two lines per page and 50 characters per line.

Similarly, think of a four-dimensional character array as a collection of books. For example, the array, `char book[5][16][55][100]`, can be considered as five books, 16 pages per book, 55 lines per page and 100 characters per line. This array will contain 440000 elements ($5 \times 16 \times 55 \times 100$). This will take up $440000/1024 = 429.7$ K of memory, assuming one byte for a variable of the `char` data type.

C++ allows you to define any number of dimensions in an array. But, you have to be careful not to exceed the memory limit of your computer!

Student Activity 1

1. What are arrays?
2. Describe various types of arrays.
3. Write a program to sort an int array.
4. Define strings.
5. Write a program to reverse a string.
6. Describe the following functions:

(a) <code>getline()</code>	(b) <code>Strlen()</code>
(c) <code>Strcpy()</code>	(d) <code>Strcat()</code>
(e) <code>Strcmp()</code>	

8.6 INTRODUCTION TO POINTERS

Consider the following snippet:

```
int som_int;
```

The above statement implies that space in memory is reserved for the variable `some int`. Assume that this variable has the value 645 and is stored at memory location (or address) 1009.

Also assume that there is another variable, `ptr_to_int`, which holds the address 1009, i.e.,

```
ptr_to_int = 1009;
```

What is ptr_to_int? It is called a pointer. A pointer is a variable which contains the address of another variable.

Declaration of Pointers

Ptr_to_int is a pointer variable which contains the address of some other variable (of type int in this case). Obviously, a pointer variable is not the same type as an integer variable. Thus how are they declared?

Observe the declaration below:

```
int ptr_to_int;    /*different from the declaration */  
  
                /* int some_int;  */
```

Note that the above declaration cannot be interpreted as: ptr_to_int is an integer variable. The * makes the difference and the implication is: ptr_to_int is a pointer to an integer variable.

Initialisation of pointers

Consider the piece below:

```
int x, *ptr;  
  
ptr = &x;        /* ptr pointing to x */
```

It is good practice to initialise a pointer as soon as it is declared. Since the pointer is just an address, potentially it can address any portion of the memory. Random pointers, which have not been initialised, are a major cause of C programs crashing unceremoniously.

Remember that C assumes, that one knows what one is doing and if you want to drive off the edge of a cliff, it won't stop you. It will cheerfully go about its task.

Coming back to the declaration above, &x returns the address of the variable x and since ptr also contains an address, the assignment:

```
ptr = &x;    /*both sides of the assignment are addresses */ is valid. Now, ptr is pointing to  
x
```

Manipulation of pointers

Now, that the declaration and initialisation of pointer variables is clear, how does one manipulate pointer variables?

Consider the scenario below:

```
int x, y, *ptr  
  
x = 645;    /* set x to 645*/  
  
ptr = &x;    /* make ptr point to x */  
  
y = ptr;    / set y to whatever ptr is pointing to */
```

The second assignment ensures that ptr is pointing to x.

Consider the third assignment

```
y = *ptr;
```

The *ptr is used to obtain whatever ptr is pointing to. Since ptr is pointing to x, *ptr returns the value of x. Thus, y will be set to x.

Student Activity 2

1. Define Pointer.
2. How will you declare and initialize pointers?
3. How will you manipulate pointers.

8.7 LISTS

A data structure is a logical method of representing data in memory using the simple and complex data types provided by the language.

8.8 INTRODUCTION TO LINKED LISTS

A linked list is a chain of structures in which each structure consists of data as well as pointers which store the address (link) of the next logical structure in the list.

8.9 ADVANTAGES OF LINKED LISTS OVER ARRAYS

1. It is not necessary to know the number of elements and allocate memory for them beforehand. Memory can be allocated as and when necessary.
2. Insertions and deletions can be handled efficiently without having to restructure the list.

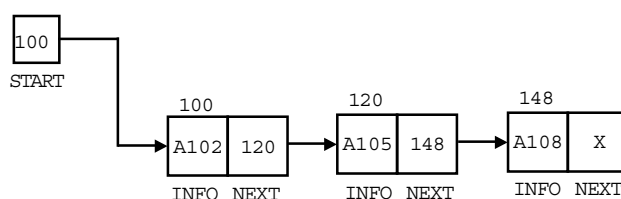


Figure 8.1: Shows a Linked List of only the Item Numbers in the Invoice Shown Earlier

Note the following points about the above linked list.

1. It consists of 3 nodes (an element in a linked list is called a node).
 - a. Each node consists of two parts:
 - i. the first part contains the information (we will refer to it as INFO).
 - ii. the second part is a pointer which contains the address of the next node (which we will refer to as NEXT). Thus, each node points to the next node.
2. The last node is the only node in the list which does not point to any node (indicated by X in its pointer NEXT). The contents of the pointer NEXT of this node are set to NULL (value zero). NULL acts as a flag to indicate end-of-list.
3. START is a pointer which stores the address of the first node in the list, thus keeping track of the beginning of the list.
4. The numbers shown above each node and within NEXT are memory addresses.

An important point to keep in mind henceforth, is that the memory addresses at which the nodes are shown to be stored are arbitrary addresses used for the purpose of illustration and explanation of concepts only. In reality these addresses might be very different.

8.10 TYPES OF LINKED LISTS

There are different types of linked lists. Two of them are mentioned below:

- Singly linked lists: Each node contains data and a single link which attaches it to the next node in the list.

- Doubly linked lists: Each node contains data and two links, one to the previous node and one to the next node.

This session deals with singly linked lists and the methods of manipulating them. Doubly linked lists will be discussed in the next session.

Student Activity 3

1. What are Lists?
2. What are the advantages of linked lists over arrays.
3. Describe various types of linked lists.

8.11 CREATING A LINKED LIST CLASS

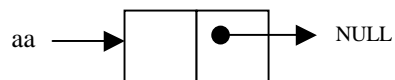
Let us create a linked list class named – list - of integers. The list will have a node type private member and a pointer *first* to it. Pointer *first* will be used to manipulate the list.

```
class List
{
private:
    struct node
    {
        int data;
        node *next;
    };
    node *first;
public:
    List(int);
};
List::List(int val)
{
    first=new node;
    first->data=val;
    first->next=NULL;
}
```

When an object of this class is constructed providing an integer type value, the value is inserted into the List. Let us create an object of this class with an integer value of 7.

```
List aa = List(7);
```

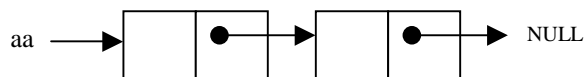
The list as depicted below:



Insertion

This List class is of not much use because it simply makes a node that's all. Let us provide a method to the class so that we may insert some values into the list. Inserting a value (let us say 11) at the end of the list is depicted below:

1. Create a new node and insert the value (11) into its data field.
2. Set the next pointer of first to temp.



```

public:
    void insert(int a)
    {
        node *temp;
        temp = new node;
        temp->data=i;
        temp->next=NULL;
        node *t;
        t=first;
        while(t->next != NULL)
            t=t->next;
        t->next= temp;
    }

```

Now we can insert an integer value in the list once it is constructed as shown below:

```

List aa = List(7);
aa.insert(11);           //11 will be inserted into the list aa

```

Calling insert method with 11 as argument will insert 11 into the list aa.

Traversing

The List class still does not have a method to show what is stored in it. Let us add a public method to the class that allows us to traverse the list.

```

public:
void List::traverse()
{
    node *temp;
    temp=first;
    while(temp != NULL)
    {
        cout<<"\n"<<temp->data;
        temp=temp->next;
    }
}

```

Now the class is equipped with a method that can be called to display what all integers are stored in this – starting from the first element to the last.

```

aa.traverse();

```

Updation

Let us write a method that takes two integer values – say a and b. The method inserts a after b in the list. If b is not found then a will be inserted in the end of the list.

```

public:
    void update(int a, int b)
    {
        node *temp, *last;

```



```
        temp = new node;
        temp->data=a;
        temp->next=NULL;
        node *t;
        t=first;
        while(t->next != NULL || t->data != b)
            t=t->next;

    if t->next==NULL
    t->next= temp;
        else
        {
            last = t->next;
            t->next=temp;
            temp->next=last;
        }
    }
```

Deletion

Let us introduce a method to delete a data item if available in the list.

```
    public:
        void remove(int a)
        {
            node *temp, *last;
            temp=first;
            last=first;
            while(temp->next != NULL || temp->data != a)
            {
                last=temp;
                temp=temp->next;
            }

            if temp->data == a
                last->next = t->next;
        }
```

Let us now write a program that uses this class for list processing.

```
#include <iostream.h>
#include <conio.h>
void main()
{
    char ch='1';
    int val, i;
    while (ch !='0')
    {
```

```
clrscr();
cout<<"*****MENU*****\n";
cout<<"Create list          1 \n";
cout<<"Insert into list      2 \n";
cout<<"Traverse list         3 \n";
cout<<"Update list           4 \n";
cout<<"Delete an element      5 \n";
cout<<"Exit                 0 \n";
switch(ch)
{
    case '1':  {
                cout<<"\n Enter the value:";

                cin>>val;
                List aa = new List(val);
                break;
            }

    case '2':  {
                cout<<"\n Enter the value:";

                cin>>val;
                aa.insert(val);
                break;
            }

    case '3':  {
                aa.traverse();
                break;
            }

    case '4':  {
                cout<<"\n Enter the value to insert:";

                cin>>val;

                cout<<"\n Enter after value:";

                cin>>i;
                aa.update(val,i);
                break;
            }

    case '5':  {
                cout<<"\n Enter the value to delete:";

                cin>>val;
                aa.remove(val);
                break;
            }
}
```

Student Activity 4

1. What is the significance of null pointer in a link list?
2. Name the two parts of the linked list.
3. Name at least two operations that can be performed on linked list.
4. What is the difference between array and linked list?

8.12 STACKS

Introduction

Stack is an ordered list in which there are only one end, for both insertions and deletions. Elements are inserted and deleted from the same end called “TOP” of the stack. Stack is called Last In First Out (LIFO) list. Since the first element in the stack will be the last element out of the stack.

In particular, the elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

Basic operations associated with stacks are:

- **Push** : Insertion of any element is called “Push” operation and Deletion from the stack is called the “Pop” operation.
- **Pop**: The most and least accessible elements in a stack are known as the “Top” and “Bottom” of the stack respectively.

A common example of a stack phenomenon, which permits the selection of only its end element, is a pile of trays in a cafeteria. Plates can be added to this pile only on the top and removed only from the top.

Suppose following digits are pushed, in order, onto an empty stack: 1, 2, 3, 4, 5, 6, 7, 8, 9.

Figure 8.1 shows the way of picturing such a stack. When these elements will be popped from stack the order will be: 9, 8, 7, 6, 5, 4, 3, 2, 1.

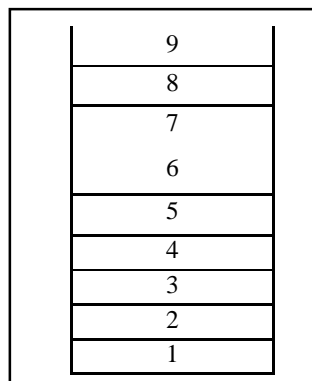


Figure 8.1

Stacks may be represented in the computer in various ways, usually by means of one-way list or a linear Array, unless otherwise stated or implied. One of the implementation using an array of 100 integers is listed in the following code:

```
#define MAX 100

main()
{
    int TOP= -1;
    int stk[MAX];
}
```

The location of the top element of the stack is stored in an integer variable TOP. The condition : $TOP = -1$

indicates that the stack is empty. When we represent any stack through an array, we have to predefine the size of stack and we can not enter more elements than that predefined size say MAX. Hence, the condition

$TOP = MAX - 1$

indicates the full stack.

Push

Push is a stack operation in which a value is put on the top of the stack. Whenever any element is pushed on to the stack the value of TOP is increased by 1. This can be implemented as:

$TOP = TOP + 1$

The following method implements push operation:

```
push (int x, int stk[ ], int MAX, int TOP)
{
    If (TOP >= MAX-1)
    {
        cout << "stack is Full";
        return;
    }
    else
    {
        top = top+1;
        stk [top] = x;
    }
}
```

Pop

Extracting the topmost element of a stack and storing the same in a suitable variable is known as *pop* operation. Whenever any element is popped from the stack the value of TOP is decreased by 1, this can be implemented as:

$TOP = TOP - 1$

The following method implements pop operation:

```
in pop(int x, int stk[ ], int top)
{
    If (TOP < 0)
    {
        cout << "stack is Empty";
        return;
    }
    else
    {
```

```

        x = stk[top];
        top = top-1;
        return (x);
    }
}

```

Lets us look one of the applications of stack.

Evaluation of Expressions

One of the biggest technical hurdles faced when conceiving the idea of higher level Programming Languages, is to generate machine language instructions, which would properly evaluate any arithmetic expression. A complex assignment statement such as:

$$z \leftarrow x / y ** a + b * c - x * a$$

might have several meanings; even if it were uniquely defined, by the full use of parenthesis.

An expression is made up of operands, operators and delimiters. The above expression has five operands x, y, a, b and c. The first problem with understanding the meaning of an expression is to decide in what order the operations are carried out. This means that every language must uniquely define such an order. To fix the order of evaluation we assign to each operator a priority. A set of sample priorities are as follows:

Operator	Priority	Associativity
()	8	Left to Right
^ or **, unary -, unary+, ![not]	7	Right to Left
*, /, %	6	Left to Right
+, -	5	Left to Right
<, <=, >, >=	4	Left to Right
=, !=	3	Left to Right
&&	2	Left to Right
	1	Left to Right

But by using parenthesis we can override these rules and such expressions are always evaluated with the inner most parenthesized expression first.

The above notation of any expression is called *Infix Notation* (In which operators come in between the operands). The notation is a traditional notation, which needs operator's priorities and associativities. But how can a compiler accept such an expression and produce correct Polish Notation or Prefix form (In which operators come before operands) Polish Notation has several advantages over Infix Notation such as: there is no need for considering priorities while evaluating them, there is no need of introducing parenthesis for maintaining order of execution of operators.

Similarly, Reverse Polish Notation or Postfix Form also has same advantages over Infix Notation, In this notation operators come after the operands.

Infix	Prefix	Postfix
x + y	+ x y	x y +
x + y * z	+ x + y z	x y z * +
x + y - z	- + x y z	x y z * + -

Stacks are frequently used for converting INFIX form into equivalent PREFIX and POSTFIX forms.

Arrays, Lists, Stacks and
Queues

Take an infix expression as input from the user and store it in a string (Array of characters). This can be done by the following C++ code:

```
#include<stdio.h>
#include<stdlib.h>
#define MAXCOLS 80
#define TRUE 1
#define FALSE 0

struct stack
{
    int top;
    char items[MAXCOLS];
};

..

..

void postfix(char *, char *);
int isoperand(char);
void popand test(stack *, char *, int *);
int pced(char, char);
void push(stack *, char);
char pop(stack *);

..

..

main()
{
    char inix[MAXCOLS];
    char postr[MAXCOLS];
    int pos = 0;
    while ((inix[pos++] = getchar()) != '\n');
    inix[--pos] = '\0';
    cout << "the original infix expression is" << inix;
    postfix(inix, postr);
    cout << "\n" << postr;
}

postfix(char inix[], char postr[])
{
    int position, und;
    int outpos = 0;
    char topsymb = '+';
    char symb;
```

```
struct stack opstk;
opstk.top = -1;
for(position=0; (symb=infix[position]) != '\0'; position++)
    if(isoperand(symb))
        postr[outpos++]=symb;
    else
        {
            popandtest(&opstk, &topsymb, &und);
            while(!und && prcd(topsymb, symb))
            {
                postr[outpos++]=topsymb;
                popandtest(&opstk, &topsymb, &und);
            }
            if(!und)
                push(&opstk, topsymb);
            if(und || (symb != '('))
                push(&opstk, symb);
            else
                topsymb=pop(&opstk);
        }
    while(!empty(&optstk))
        postr[outpos++] = pop(&opstr);
    postr[outpos]='\0';
return;
}
```

Student Activity 5

1. What do you understand by push and pop?
2. Write applications of stack.
3. Why stack is called LIFO data structure?
4. If the elements are pushed in the stack in the order 10, 20, 30, 40, 50 and 60. List the order in which they are popped out from the stack.
5. Evaluate the following postfix expression using a stack and show the contents of stack after execution of each operation.
 - (i) 100, 40, 8, + 20, 10, -, + *
 - (ii) 5, 6, 9, +, 80, 5, *, -, /
 - (iii) True, false, true, false, Not, or True, OR, OR, and
6. Change the following infix expression to postfix:
 - (i) $(A+B) \times c + D/E - F$
 - (ii) $(A+B) - c * (D_E)$

8.13 QUEUES

Queues arise quite naturally in the computer solution of many problems. Perhaps the most common occurrence of a queue in Computer Applications is for the scheduling of jobs.

Queue is a Linear list which has two ends, one for insertion of elements and other for deletion of elements. The first end is called 'Rear' and the later is called 'Front'. Elements are inserted from Rear End and Deleted from Front End. Queues are called First In First Out (FIFO) List, since the first element in a queue will be the first element out of the queue. In other words, the order in which the elements enter a queue is the order in which they leave.

In Batch Processing the jobs are "Queued up" as they are read in and executed, one after another in the order they were received. This ignores the possible existence of priorities.

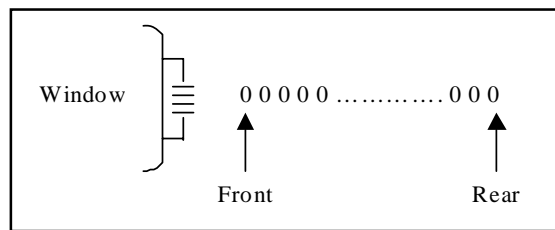


Figure 8.2 A Possible Queue

Queues may be represented in the computer in various ways, usually by means of one way lists or linear Arrays, unless otherwise stated or implied. Each Queue will be maintained by a linear array queue [] and two integer variables front and REAR containing the location of the front element of the queue and the location of the Rear element of the queue.

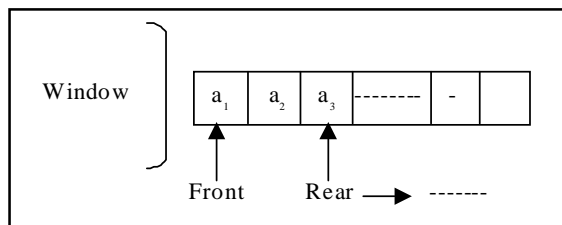


Figure 8.3: Queue Represented By Array

When we represent any Queue through an Array, we have to predefine the size of Queue and we cannot enter more elements than that predefined size, say max.

Initially $\text{Rear} = -1$, latest inserted element in queue.

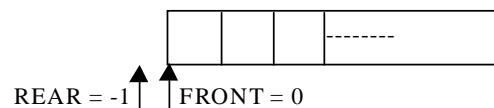
Initially $\text{FRONT} = 0$, because Front points to the first inserted element which is not yet deleted. Initially queue is empty, hence $\text{FRONT} = 0$.

Similarly : Condition for empty queue is

$$\text{FRONT} = 0$$

$$\text{REAR} = -1$$

When $\text{FRONT} = \text{REAR}$ there is exactly one element in Queue.



(a) Empty Queue

(b) Queue with Exactly One Element

Figure 8.4

Whenever an element is added to the Queue, the value of REAR is increased by 1; this can be implemented as:

```
REAR = REAR + 1;
```

Or

```
REAR ++;
```

provided that REAR is less than MAX-1, which is the condition for full Queue.

Lets us see some C++ code for a queue and its operations.

```
# define MAX 100  
main ()  
{  
    int FRONT = -1, REAR = 0;  
    int Queue[MAX];  
}
```

Insertion

Mthod for Insertion in Queue :

```
insert q (int x)  
{  
    If (FRONT == -1)  
    {  
        QUEUE [Rear] = x;  
        Front ++;  
    }  
    If (REAR = MAX - 1)  
    {  
        printf("Queue is full");  
        return;  
    }  
    else  
    {  
        REAR ++;  
        Queue[REAR] = x;  
    }  
}
```

Deletion

Whenever an element is deleted from the Queue, the value of FRONT is increased by 1; this can be implemented by the following assignment.

$$\text{FRONT} = \text{FRONT} + 1$$

Method for Deletion from Queue:

```
int Deleteq()
{
    int x;
    If (FRONT = REAR+1)
    {
        printf ("Queue is Empty");
        return;
    }
    else
    {
        x = Queue[FRONT]; FRONT++; return(x);
    }
}
```

But this representation of queue has a major drawback. Suppose a situation when, queue becomes full i.e. $\text{REAR} = \text{MAX}$, but we delete some elements from the Queue and FRONT is now pointing to $A[i]$ such that $0 \leq i < \text{MAX}$ as shown in Figure 8.5. We cannot insert any element in Queue though there is space in Queue, and if we utilize that space by inserting elements on the positions $A[0]$, $A[1]$ — $A[i-1]$, $A[i]$ we are violating the FIFO property of Queue. One way to do this is to simply move the entire queue to the beginning of the array, changing FRONT and REAR accordingly, and then inserting ITEM as above. This procedure may be very expensive.

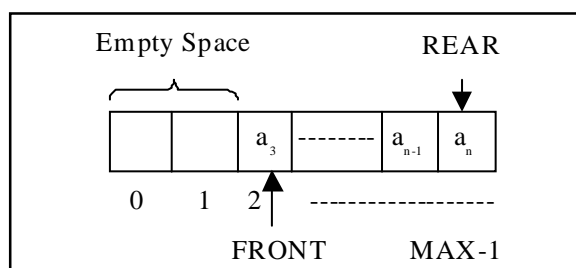


Figure 8.5

8.14 TREES

Introduction

Tree data structure is mainly used to represent data containing a hierarchical relationship between elements. Familiar examples of such structure are: family trees, the hierarchy of positions in an organization, an algebraic expression involving operations for which certain rules of precedence are prescribed etc.

For example suppose we wish to use a data structure to represent a person and all of his or her descendants. Assume that the person's name is Mahesh and he has 3 children, Raja, Mohan and Vinod. Also suppose that Mohan has 3 children, Ajay, Meera and Navin and

Raja has one child Ramesh. We can represent Mahesh and his descendants quite naturally with the tree structure shown in Figure 8.6.

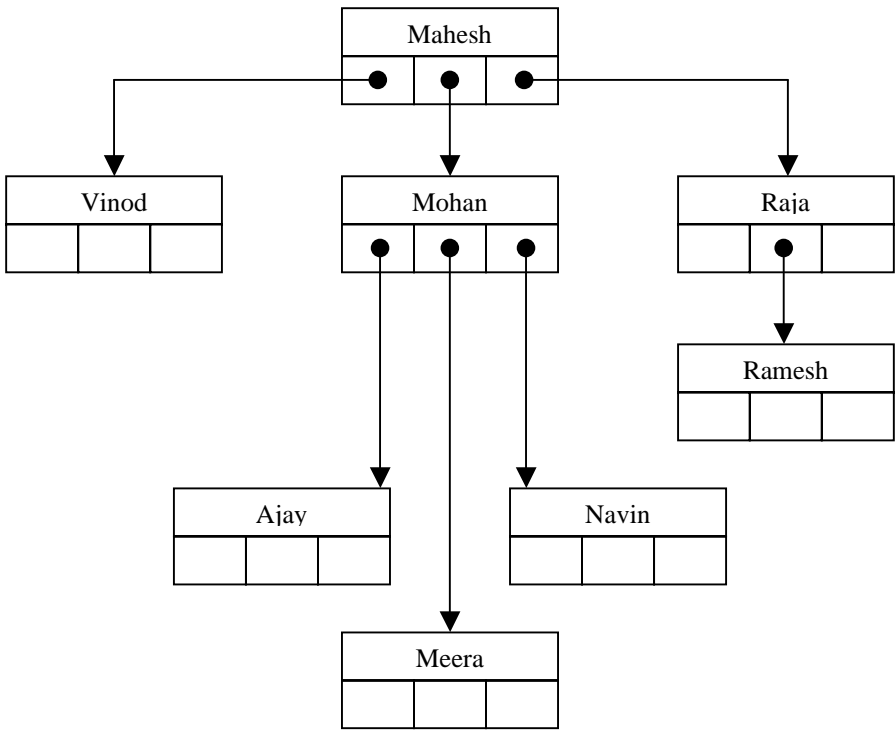


Figure 8.6

A tree is a finite set of one or more nodes such that there is a specially designated node called the *Root* and remaining nodes are partitioned into $n \geq 0$ disjoint sets S_1, \dots, S_n where each of these sets is a tree. S_1, \dots, S_n are called the subtrees of the root.

The condition that S_1, \dots, S_n be disjoint sets prohibits subtrees from ever connecting together. A node stands for the item of information plus the branches to other items. The number of subtrees of a node is called its *degree*. Nodes that have degree zero is called *leaf* or *Terminal nodes*. Children of the same parent are called ‘Siblings’ .

The ‘level’ of a node is defined by initially letting the root be at level 1. If a node is at level l , then its children are at level $l+1$. The height or depth of a Tree is defined as the maximum level of any node in the Tree.

A ‘forest’ is a set of $n \geq 0$ disjoint trees.

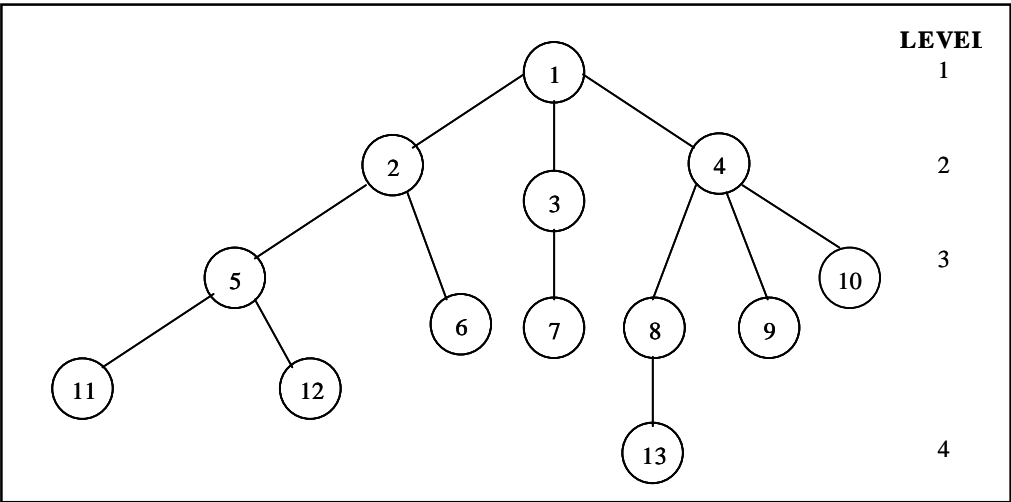


Figure 8.7: An Example Tree

A Binary Tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the Root of the tree. The other two subsets are themselves Binary Trees, called the left and right subtrees of the original tree. A left or right subtree can be empty. A node of a Binary Tree can have at most two Branches.

A Complete Binary Tree is a Strictly Binary Tree of depth 'd' whose all leaves are at level d.



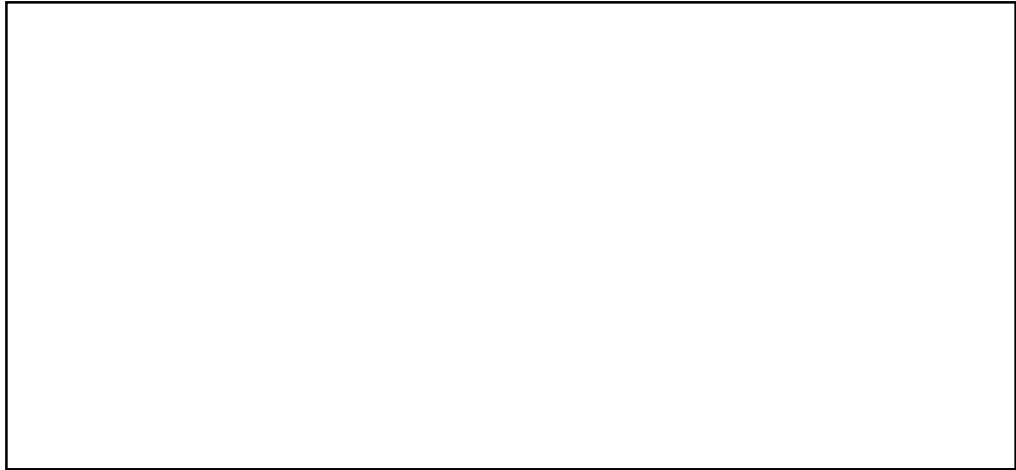


Figure 8.10: The Complete Binary Tree

Traversal

The traversal of a Binary Tree is an act of visiting each node in the tree exactly once. A full traversal produces a linear order for the information in a tree. When traversing a binary tree we want to treat each node and its subtrees in the same fashion. If we let L,D,R stand for moving left, printing the data, and moving right when at a node then there are six possible combinations of traversal: LDR, LRD, DLR,DRL,RDL andRDL. If convention is adopted then we traverse left before right then only three traversals remain i.e. LDR,LRD,DLR. To these, the names INORDER, POSTORDER and PREORDER are assigned because there is a natural correspondence between these traversals and producing the INFIX, POSTFIX and PREFIX forms of an expression.

Preorder

To traverse a non empty tree in preorder we perform the following three operations:

- i) Visit the root
- ii) Traverse the left Subtree in Preorder
- iii) Traverse the right Subtree in Preorder.

Inorder

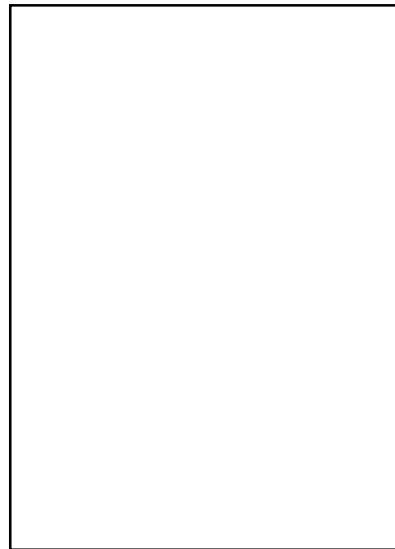
- i) Traverse the left Subtree in Inorder
- ii) Visit the root
- iii) Traverse the right Subtree in Inorder.

Postorder

- i) Traverse the Left Subtree in Postorder.
- ii) Traverse the right Subtree in Postorder.
- iii) Visit the root.

Consider the Tree of Figure 8.11.

- In Inorder Traversal the order will be
 $A/B**C*D+E$
which is Infix form of the expression.
- In Postorder Traversal the order will be:
 $ABC**/D*E+$
which is postfix form of the Traversal

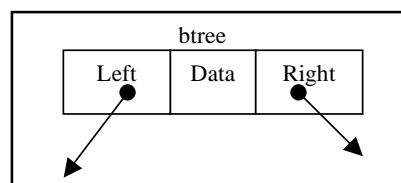
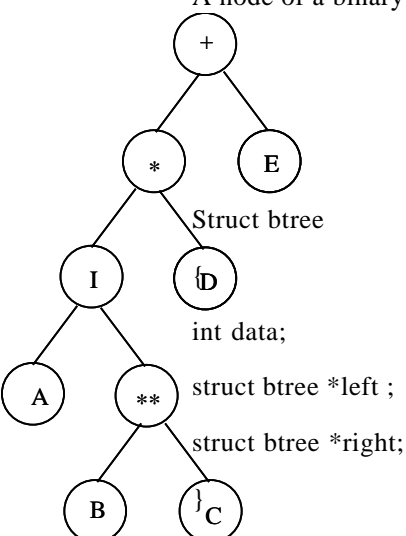
**Figure 8.11: Binary Tree**

- In preorder Traversal the order will be :
+*/A**BCDE
which is the prefix form of the expression.

The above mentioned Traversal Methods can be implemented either recursively or non recursively.

8.16 IMPLEMENTING BINARY TREE

A node of a binary tree *btree* can be represented as:

**Figure 8.12: Node of a Binary Tree**

In the above structure two pointers of struct btree type are defined. One points to the left child of this node and other points to the right child of the node.

Addition

Algorithm to implement a Binary Tree proceed in two phases. The first phase builds a binary tree, and the second traverses the tree. In the first phase as we read, the number for insertion into a Binary Tree, is compared with the contents of a node in the tree, a left branch is taken if the number is smaller than the contents of the node and a right branch is taken if it is greater or equal to the contents of the node. Code to implement this is given below:

```
insert (btree *s, int num)
```

```

{                                     /*s is a pointer points to the existing tree*/
    struct btree *temp, *p;
    temp = new btree;
    temp -> data = num;
temp -> left = temp -> right = NULL;
    if(s==NULL)                      /*if current tree is input*/
        s=temp;
    else
    {
        p=s;
        while(p -> left != NULL && p -> right != NULL)
        {
            if (p -> data <= num) //if num is greater
                p=p -> right ; //than p ? data go Right
            p=p -> left; // else go left
            else
        }
        if (p -> data <= num)
            p -> right =temp;
        else
            p -> left =temp;
    }
}

```

The functioning of the above code can be explained through an example. Suppose we have to make a tree with the following data :

5, 3, 7, 8, 2, 4, 6, 1.

When 5 came the tree was empty hence tree is formed having only one node pointed by temp and s. Both the child pointers of this node are NULL.

Then next number is 3. In the while Loop this number is compared with 5 because 5 is greater than 3, control is moved to the left child of 5 which is NULL. After exiting from while loop 3, is attached to the left of 5 Next number is 7 which is greater than 5 hence, it is attached to the right of 5. Proceeding in the same manner the Binary tree of Figure 8.13 is produced.



Figure 8.13

Such a binary tree has the property that all elements in the left subtree of a node n are less than the contents of n , and all elements in the right subtree of n are greater than or equal to the contents of n . Binary Tree with this property is called a Binary Search Tree.

In the second phase we can Traverse a Tree in either of the three fashions (ie. Inorder, Preorder, Postorder). A Recursive Method can be used to traverse a Binary Tree. The method starts by checking whether the tree is empty or not. If the tree is not empty i.e. our pointer s of struct `btree` type doesn't point to `NULL` then we will proceed. Depending upon the desired fashion of traversing, we will call recursively the traversing function by passing either left child pointer or right child pointer of the node as an argument. The function to implement this is given below:

```
/* Inorder Traversing of a Binary Search Tree */
inorder (btree *s)
{
    if (s!=NULL)
    {
        inorder(s->left);
        cout << s->data;
    }    inorder(s->right);
    return;
}

/*Preorder Traversing of a Binary Search Tree*/
preorder (btree*s)
{
    if (s != NULL)
    {
        cout << s ->data;
        preorder(s ->left);
        preorder(s ->right);
    }
    return;
}

/* Post order Traversing of Binary Search Tree */
postorder (btree *s)
{
    if (s!=NULL)
    {
        postorder (s ->right);
        postorder (s ->left);
        cout << s ->data;
    }
    return;
}
```


Deletion

One of the important operations associated with any Data Structure is deletion of the specified data item. Assuming that we will pass the specified data item, we wish to delete from the Binary Tree to the delete function.

There are four possible cases:

- No node contains the specified data
- The node containing the data has no children
- The node containing the data has exactly one child
- The node containing the data has two children.

In case (a) we will only need to print the message that the data item is not present in the tree.

Figure 8.14 represents the case (b). We have to delete the node containing data 4. The node is leaf node and it has no children, it can be deleted by making its parent pointing to NULL. Which of the two links (Left or Right) of its parent node is set to NULL depends upon whether the node being deleted is a left child or a right child of its parent.

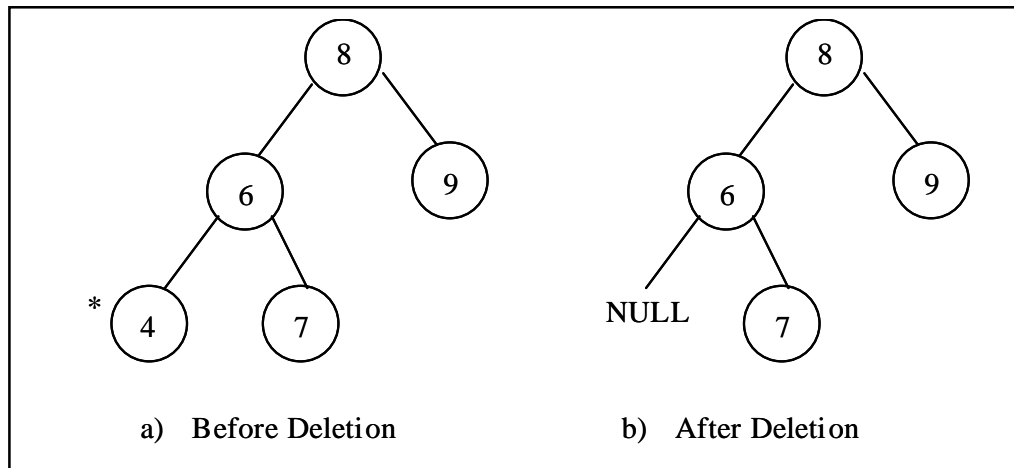


Figure 8.14

Fig 13.16 represents case (c) in which node to be deleted has one child; we have to adjust the pointer of the parent of the node to be deleted such that after deletion it points to the child of the node being deleted. In Fig 13.15(a) node 1 with data 1 is to be deleted. After deletion left pointers of node 6 is made to point to child of node 1 i.e. 4 and now structure would be as shown in Figure 8.15(b).

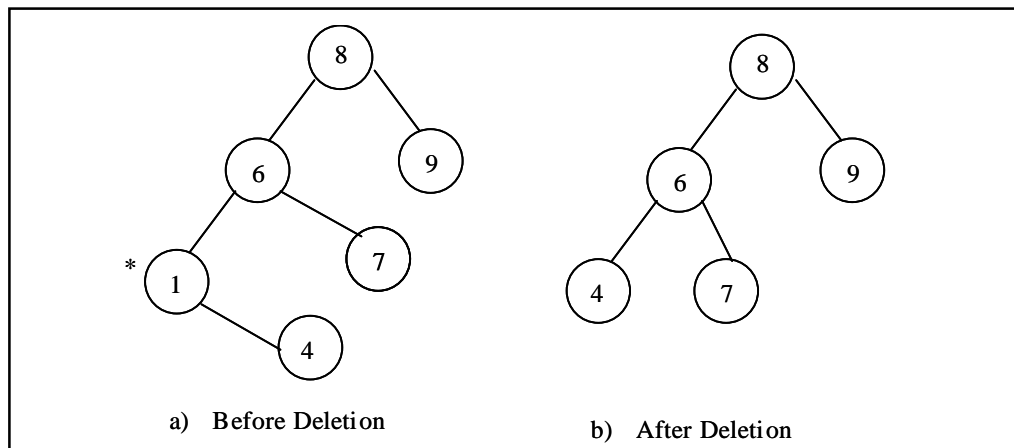


Figure 8.15

In case (d) the value is replaced by the smallest value in the right subtree or the largest key value in the left subtree; Subsequently empty node is deleted recursively, Figure 8.16 represents this case. If the node with data value 6 is to be deleted then first its value is replaced by smallest value in its right subtree. Adjustment is done for the parent of this node (with smallest value) according to case (b) or (c), then the node is deleted.

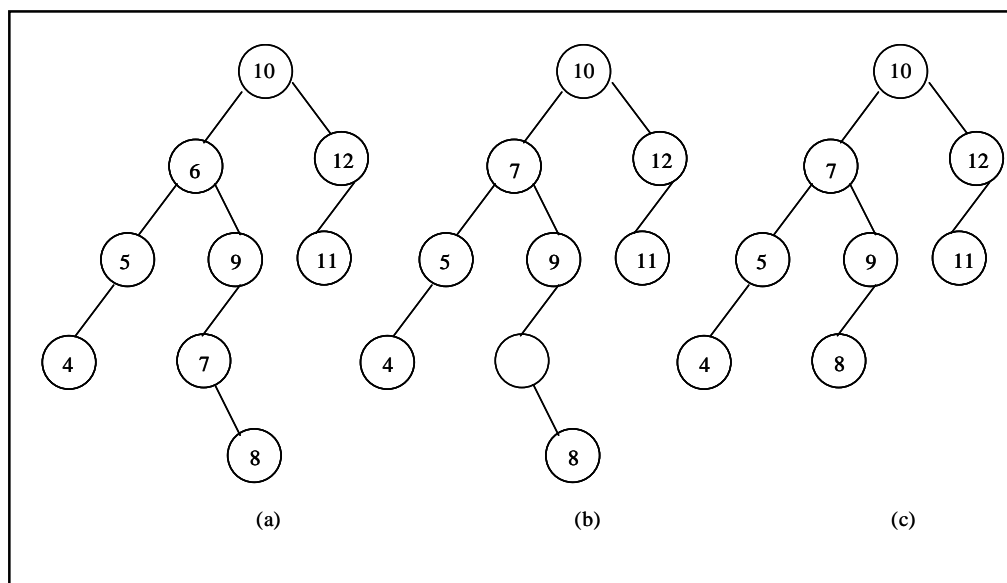


Figure 8.16

Function for implementing the deletion procedure is given below:

```

delete(btree *root, int num)
{
    int found;
    btree *par, *x, *xsucc;
    if(root== NULL)
    {
        cout << "\n Empty Tree";
        return;
    }
    par = x = NULL;
    found = search (root, num, par & x)
    /*Search function to find the node to be deleted, implement it the way you want*/
    if (found == 0)
    {
        cout << "\n Data to be deleted, not found";
        return;
    }
    /*If the node to be deleted has two children */
    if (x->left!=NULL && x->right!=NULL)
    {
        par = x;
        xsucc = x-> right;
    }
}
  
```

```
        while(xsucc->left != NULL)
        {
            par = xsucc;
            xsucc = xsucc->left;
        }
        x->data = xsucc->data;
        x = x->xsucc;
    }
    /* If the node to be deleted has no child */
    if (x->left==NULL && x->right==NULL)
    {
        if(par->right==x)
            par->right=NULL;
        else
            par->left=NULL;
        free(x);
        return;
    }
    /* If the node to be deleted has only right child */
    if (x->left==NULL && x->right!=NULL)
    {
        if(par->right=x)
            par->right=NULL;
        else
            par->left=NULL;
        else
            par->left=NULL;
        free(x);
        return;
    }
    /* If the node to be deleted has only right child */
    if(x->left==NULL && x->right !=NULL)
    {
        if (par->left=x)
            par->left=x->right;
        else
            par->right=x->right;
        free(x);
        return;
    }
    /* If the node to be deleted has only left child */
    if (x->left!=NULL && x->right==NULL)
```

```

{
    if (par->left == x)
        par->left = x->left;
    else
        par->right = x->left;
    free (x);
    return;
}

```

Student Activity 6

1. Distinguish between stacks and queues.
2. Why queue is called FIFO structure?
3. How will you find numbers of elements in circular queue?
4. What is the drawback of linear queue?
5. Write an algorithm to implement a binary tree.
6. What is binary search tree?

8.17 SUMMARY

Arrays are the data structure, which hold multiple variables of the same data type. Arrays can be one-dimensional or multi-dimensional. A two-dimensional array can be considered as an array of one-dimensional arrays.

Global arrays can be initialized when they are called. Two dimensional arrays may be initialized by a list of initial values enclosed in braces following their declaration.

Once an array is declared. Individual elements of the array are referred using subscript or index number. This number specifies the element's position in the array.

Array can be used as arguments to the functions. Array arguments are represented by the data type and sizes of the array in a functions declaration. When the function declaration. When the function is called. To properly access the array the function needs that all the array dimensions must be specified.

A string is defined in C++ as an array of characters. Each string is terminated by the NULL character. String are stored in fixed or variable length structure or linked structure. String related functions are grouped into string.h header file.

The data structure is a process of how the data items are organized in the computer memory for the processing and for the subsequent usage.

A linked list is a chain of structures in which each structure consists of data as well as pointers structure. The list will have a node type private member and a pointer first to it. Pointer first will be used to manipulate the list a value can be inserted at the end of the list. We can traverse the list i.e., we can find that what is stored in the list. The list can be updated and data item also can be deleted.

Stack is an ordered list in which there are only one end, for both insertions and deletions. Stack is called last in first out (LIFO) list. Basic operations include 'push' and 'pop' push operation is used to put a value on the top of the stack. Pop operation used on the top of the stack. Pop operation is used to extract the top most element of a stack and storing the same in a suitable variable.

Queue is a linear list which has two ends, one for insertion of elements and other for deletion of elements. The first end is called 'rear' and the later is called 'front'. Queues are

called first In first out (FIFO) list.

Tree data structure is mainly used to represent data containing hierarchical relationship between elements. A tree is a finite set of one or more nodes. There is a specially designated node called the Root and remaining nodes are partitioned into subtrees. Nodes that have degree zero is called leaf and children of the same parent are called siblings. A forest is a set of $n - 1$ disjoint trees.

A Binary tree is either empty or partitioned into three disjoint subsets. The first subset is a single element, called two root of the tree. The other two subsets are themselves Binary trees. A node of binary tree can have at most two branches.

8.18 KEYWORDS

Array: A named list of a finite number of similar data elements.

Data item: Single unit of values of certain type.

Data type: Named group of data with similar characteristics and behaviour.

Two-dimensional Array: An array in which each element is itself a one-dimensional array.

String: An array of characters

Data structure: The organized collection of data is called data structure.

Traversal: Traversal is a method to process each element individually and separately.

Search: Search is the important activity of data structure in which a particular element is searched in order to find its location whether the given element is present or not.

Linked List: The linked consists of a series of nodes, which are node contains the element and a pointer to a record containing its successor. We called this the LINK pointer. The last node LINK pointer contains zero, which is typically known as NULL pointer.

Stack: A stack is a list with the restriction that a new node can be added to a stack and removed from a stack only at the top.

LIFO: A stack is called (Last In First out data structure.

Push: When the element is added into the stack, the operation is called push.

Pop: when the element is deleted from the stack, the operation is called Pop.

Infix Expression: When the operator is placed in between the operands, the expression is called infix expression, e.g. $A + B$.

Postfix Expression: When the operator is placed after the operands, the expression is called postfix expression. E.g. $AB +$

Prefix Expression: When the operator is placed before the operands, the expression is called prefix expression, e.g. $+AB$.

Queue: A queue is similar to a checkout line on the bus stand. The first person online is served first and the person can enter the line only at the end.

FIFO: The element are deleted from the queue from the end called front.

Rear: The element are inserted in the queue from the end called rear.

Tree: data structure is mainly used to represent data containing hierarchical relationship between elements.

Binary Tree: Binary is either empty or partitioned into there disjoint subsets i.e. root, binary trees.

8.19 REVIEW QUESTIONS

1. How will you calculate the length of a one-dimensional array?
2. What is the address of the first elements of an array called?
3. For an int array B [20] with base address 100, what will be the address of B [12]?
4. What is meant by base address of an array?
5. Given two 2-D arrays, what is the condition to calculate the sum of the given arrays?
6. An array x [10] [20] is stored in memory with each element requiring 4 bytes of storage. If the base address of the array is 2000, calculate the location of x [3] [5] when the array x is stored using column major orders.
7. Write a user-defined function in c++ to display those elements of a two-dimensional array T {4} [4] which are divisible by 100. Assume the content of the array is already present and the function prototype is as follows:
Void show hundred (int T [4] [4];
8. What are stacks?
9. With an example, explain the representation of stacks.
10. What is a node?
11. What is the difference between leaf and children?
12. What is forest?
13. What is a binary tree?
14. What are the various forms of binary tree?
15. Why is the traversal of binary tree required?
16. What are the various traversal orders of a tree?
17. How to implement a binary tree? Give a full working example.
18. Write a program to implement stack as a circular linked list.

8.20 FURTHER READINGS

Herbert Schildt, *The complete Reference-C++*, Tata Mc Graw Hill

Robert Lafore, *Object oriented Programming in Turbo C++*, Galgotia Publications.

E. Balagurusamy, *Object Oriented Programming through C++*, Tata McGraw Hill.

UNIT

9

TEMPLATES AND ERROR HANDLING

LEARNING OBJECTIVES

After studying this unit, you should be able to:

- Describe function of templates.
- Describe classes of templates.
- Describe templates specialization.
- Error handling and error isolation.
- Describe watch values, break point and stepping.
- Describe function of templates.
- Describe templates specialization.
- Error handling and error isolation.
- Describe watch values, break point and stepping.

UNIT STRUCTURE

- 9.1 Introduction
- 9.2 Function of Templates
- 9.3 Classes of Template
- 9.4 The typename Keyword
- 9.5 Template Specialization
- 9.6 Point of Instantiation
- 9.7 Name Resolution
- 9.8 Error Handling
- 9.9 Error Isolation
- 9.10 Watch Values
- 9.11 Breakpoints
- 9.12 Stepping
- 9.13 Concurrent Object-oriented Systems
- 9.14 Summary
- 9.15 Keyword
- 9.16 Review Question
- 9.17 Further Readings

9.1 INTRODUCTION

The template is one of C++'s most sophisticated and high-powered features. Although not part of the original specification for C++, it was added several years ago and is supported by

all modern C++ compilers. Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class with several different types of data without having to explicitly recode specific versions for each data type.

9.2 FUNCTION OF TEMPLATES

In C++ when a function is overloaded, many copies of it have to be created, one for each data type it acts on. In the example of the `max()` function, which returns the greater of the two values passed to it this function would have to be coded for every data type being used. Thus, you will end up coding the same function for each of the types, like `int`, `float`, `char`, and `double`. A few versions of `max()` are

```
Int max ( int x, int y)
{
    return x > y ? x : y
}

char max ( char x, char y )
{
    return x > y ? x : y
}

double max (double x , double y)
{
    return x > y ? x : y
}

float max ( float x, float y)
{
    return x > y ? x : y
}
```

Here you can see, the body of each version of the function is identical. The same code has to be repeated to carry out the same function on different data types. This is a waste of time and effort, which can be avoided using the template utility provided by C++.

"A template function may be defined as an unbounded functions " all the possible parameters to the function are not known in advance and a copy of the function has to be created as and when necessary. Template functions are using the keyword, `template`. Templates are blueprints of a function that can be applied to different data types. The definition of the template begins with the `template` keyword. This is followed by a comma-separated list of parameter types enclosed within the less than (<) and greater than (>) signs.

Syntax of Template

Template < class type 1, type 2 ... >

Void function - name (type 2 parameter 1, type 1 parameter 2) { ... }

Example

```
Template < class X >
X min ( X a , X b )
{
    return (a < b) ? a : b ;
}
```


}

This list of parameter types is called the formal parameter list of the template, and it cannot be empty. Each formal parameter consists of the keyword, type name, followed by an identifier. The identifier can be built-in or user-defined data type, or the identifier type. When the function is invoked with actual parameters, the identifier type is substituted with the actual type of the parameter. This allows the use of any data type. The template declaration immediately precedes the definition of the function for which the template is being defined. The template declaration, followed by the function definition, constitutes the template definition. The template of the max () function is coded below:

```
#include <iostream.h>

template < class type >

type max ( type x , type y)

{

    return x > y ? x : y ;

}

int main ( )

{

    cout << " max ( 'A', 'a') : " << max ( 'A', 'a') << endl ;

    cout << " max ( 30, 40 ) : " << max ( 30 , 40) << endl;

    cout << " max ( 45 . 67F, 12 . 32 F) : " << max (45. 67F, 12 . 32 F) <,

    endl;

    return 0 ;

}
```

Output

```
max ( `A` , `a`)      : a
max ( 30 , 40 )       : 40
max ( 45.67F, 12 . 32F ) : 45 . 67
```

In the example, the list of parameters is composed of only one parameter. The next line specifies that the function takes two arguments and returns a value, all of the defined in the formal parameter list. See what happens if the following command is issued, keeping in mind the template definition for max () :

```
max ( a , b ) ;
```

in which a and b are integer type variables. When the user invokes max () , using two int values, the identifier, 'type' ,is substituted with int, wherever it is present. Now max () works just like the function int max (int, int) defined earlier, to compare two int values. Similarly, if max () is invoked using two double values, 'type' is replaced with 'double'. This process of substitution, depending on the parameter type passed to the function, is called template instantiation. The template specifies how individual functions will be constructed, given a set of actual types. The template facility allows the creation of a blueprint for a function like max (). The template or blueprint can then be instantiated for all data types, eliminating duplication of the source code. The identifier, 'type', can be used within the function body of a function that, otherwise, remains unchanged.

For example, the template for a function called square 9, which calculates the square of any number (int, float, or double type) passed to it can be given as :

```
#include <iostream.h>

template < class type >
type square ( type a)
{
    type b;
    b= a*a ;
    return b.;
}

int main ( )
{
    cout << " square (25 )           : " << square ( 25) << endl;
    cout << " square 40             : " << square 40 << endl;
    return 0 ;
}
```

Output

```
Square ( 25 . 45F) : 1125
Square 90          : 1600
```

Here is another example of the use of template function. Consider the following classes:

```
class IRON
{
private :
    float density ;
public :
    IRON ( ) {density = 8.9 }
    Float Density ( ) { return density ;}
};
```

9.3 CLASSES OF TEMPLATE

Template classes may be defined as the layout and operations for an unbounded set of related classes. Built in data types can be given as template arguments in the list for template classes.

Syntax

```
Template < class type 1 , ... >
Class class - name
{
    public
    type 1 var,
    ...
};
```

Suppose you want two different classes for the same data type this can be achieved through the following code:

```
#include <iostream.h>

template < class T, int z >
class W
{
    public :
    T a ;
    W ( T q )
    {
        a = z + q ;
        cout << " a = " << a << endl ;
    }
};

int main ( )
{
    W < int,10 > one ( 100) ;           // Displays a = 110
    W < int , 10 > + two ptr = & one ;   // No object is created
    W < int, 10 + 3 > three = 200;       // Displays a = 230
    W < float, 40 > four = 100.45 ;     // Displays a = 140.45
    Return 0;
}
```

Rules for Using Templates

Listed below are the rules for using templates:

- The name of a parameter can appear only once within a formal parameter list. For example:

```
Template <class type>
Type max ( type a, type b)
{
    return a > b ? a : b ;
}
```

- The keyword, class must be specified before the identifier.
- Each of the formal parameters should form a part of the signature of the function. For example:

```
Template < class T >
Void func ( T a )
{
    .....
}
```

- Templates cannot be used for all overloaded functions. They can be used only for those functions whose function body is the same and argument types are different.
- A built-in data type is not to be given as a template argument in the list for a template function but the function may be called with it . For example:

```

Template < class ZZ , int > // wrong
Void fn ( ZZ , int ) ;

Template < class Z >      // right
Void fn ( ZZ , int ) ;

```

- Template arguments can take default values. The value of these arguments become constant for that particular instantiation of the template. For example:

```

Template < class X , int var = 69 >
Class Aclass
{
    public :
    void func ( )
    {
        X array [ var ] ;
    }
};

```

9.4 THE TYPENAME KEYWORD

The keywords `typename` and `class` can be freely interchanged. For example

```

Template < class T >
Int maxindex ( T arr [ ] , int size )
{
    ...
}

```

9.5 TEMPLATE SPECIALIZATION

While template functions and classes are usable for any data type, that would hold true only, as long as the body of the functions or the class is identical throughout. So, if you have,

```

Template < typename T .
Void swap ( T + lhs , T + rhs )
{
    T tmp ( + lhs ) ;
    + lhs = + rhs ;
    + rhs = tmp ;
}

```

the previous template will be instantiated correctly for any of the pointer types passed to it - except for the data types `char +`, `unsigned char +`, and `signed char +`. Since, the body of the function.

Differs for these types, you would use template specialize the template function.

Specialization would consist of instantiating the template definition for a specific data type. So, when the compiler needs to instantiate the template, it finds a predefined version already existing and uses it. For example, just after the previous template declaration, you could

create a template specialization for char + like this:

```
Template e >
Void swap < char > ( char + lhs , char + rhs )
{
char + tmp = new char [ strlen (lhs) + 1 ] ;
strcpy ( tmp , lhs) .;
strcpy ( tmp , lhs);
strcpy ( lhs , rhs) ;
strcpy ( rhs , tmp ) ;
}
```

9.6 POINT OF INSTANTIATION

A template function gets instantiated under the following circumstances:

- Implicitly instantiated because it is referenced from a function call that depends on a template argument.
- Implicitly instantiated because it is referenced within a default argument in a declaration.
- The point of instantiation of a function template specialization immediately follows the declaration or definition that refers to the specialization.

9.7 NAME RESOLUTION

Functions are found using the usual lookup rules :

- Look for a function with an exact match to the function being called.
- If a template declaration is found, see if there is a matching template instance use if found.
- If the template instance does not exist and if the template definition is visible, instantiate the template.

And use it.

If the template instance does not exist, and the definition is not visible, generate an error.

Student Activity 1

1. What are templates?
2. Give the syntax to declare a template.
3. Define the template class.
4. Give the rules for using templates.
5. What do you mean by template specialization.
6. When can a template function be instantiated?
7. What happen if the template instance does not exist?

9.8 ERROR HANDLING

We now know the syntactic errors and execution errors usually produce error messages when compiling or executing a program. Syntactic errors are relatively easy to find and correct, even if the resulting error messages are unclear. Execution errors, on the other hand, can be much more troublesome. When an execution error occurs, we must first determine its location (where it occurs) within the program. Once the location of the execution error has

been identified, the source of the error (why it occurs) must be determined. Location of the execution error occurred often assists, however, in recognizing and correcting the error.

Closely related to execution errors are logical errors. Here the program executes correctly, carrying out the programmer's wishes, but the programmer has supplied the computer with instructions that are logically incorrect. Logical errors can be very difficult to detect, since the output resulting from a logically incorrect program may appear to be error-free. Moreover, logical errors are often hard to locate even when they are known to exist (as, for example).

Methods are available for finding the location of execution errors and logical errors within a program. Such methods are generally referred to as debugging techniques. Some of the more commonly used debugging techniques are described below.

9.9 ERROR ISOLATION

Error isolation is useful for locating an error resulting in a diagnostic message. If the general location of the error is not known, it can frequently be found by temporarily deleting a portion of the program and then rerunning the program to see if the error disappears. The temporary deletion is accomplished by surrounding the instructions with comment markets (`/*` and `*/`), causing the enclosed instructions to become comments. If the error message then disappears, the deleted portion of the program contains the source of the error.

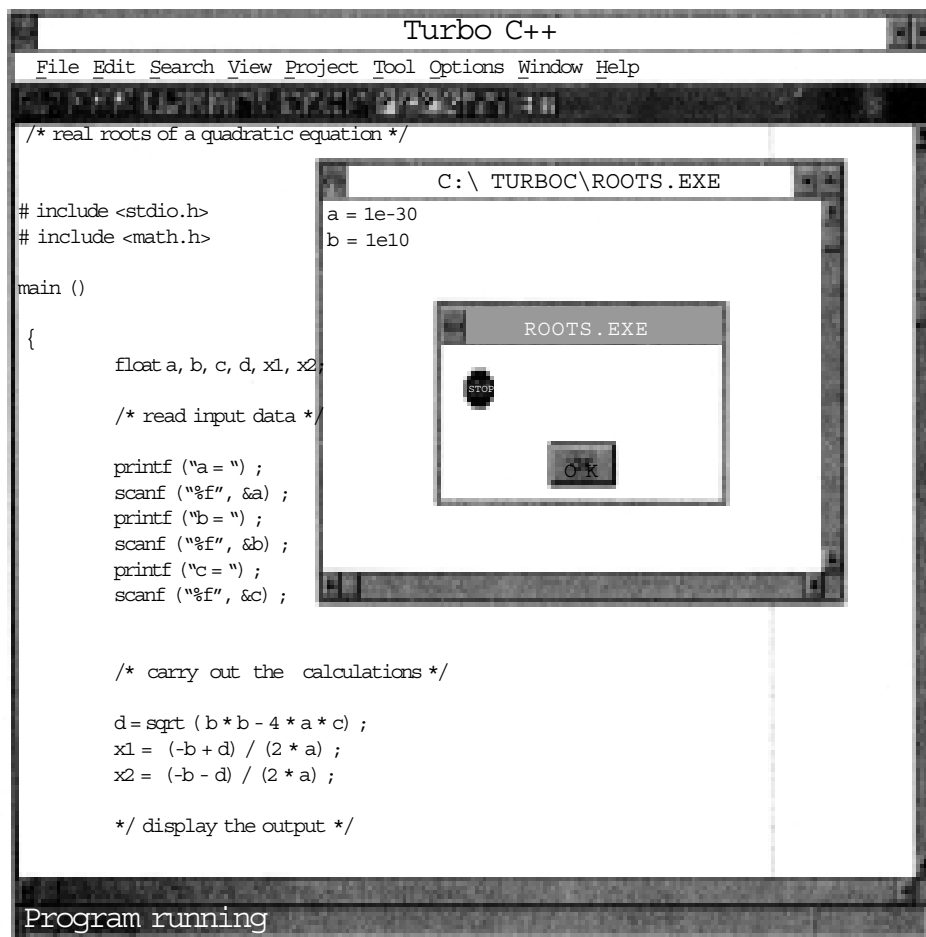


Figure 9.1

A closely related technique is that of inserting several unique `printf` statements, such as

```
Printf("debugging - line 1/n");
```

```
Printf("Debugging - line 2/n");
```

etc. at various places within the program. When the program is executed, the debug messages will indicate the approximate location of the error. Thus, the source of the error will lie somewhere between the last printf statement whose message did appear, and the first printf statement whose message did not appear.

9.10 WATCH VALUES

A watch value is the value of a variable or an expression which is displayed continuously as the program executes. Thus, you can see the changes in a watch value as they occur, in response to the program logic. By monitoring a few carefully selected watch values, you can often determine where the program begins to generate incorrect or unexpected values.

9.11 BREAKPOINTS

A breakpoint is a temporary stopping point within a program. Each breakpoint is associated with a particular instruction within the program. When the program is executed, the program execution will temporarily stop at the breakpoint, before the instruction is executed. The execution may then be resumed, until the next breakpoint is encountered. Breakpoints are often used in conjunction with watch values, by observing the current watch value at each breakpoint as the program executes.

9.12 STEPPING

Stepping refers to the execution of one instruction at a time, typically by pressing a function key to execute each instruction. In Turbo C++, for example, stepping can be carried out by pressing either function keys F7 or F8. (F8 steps over subordinate functions, whereas F7 steps through the functions.) By stepping through an entire program, you can determine which instructions produce erroneous results or generate error messages.

9.13 CONCURRENT OBJECT-ORIENTED SYSTEMS

Concurrent objects are objects that exist and operate in the same environment simultaneously.

Our most important goals of object-oriented software development is to construct a model of the real world. We do this because we want to build a conceptual model, or a description, of the real world. A closer look at the real world reveals that concurrent activities appear everywhere. Concurrent activities are those activities that take place simultaneously.

The entities in the real world are often nested, and so are the objects in software systems. Each of these nested objects may encapsulate nested activities again, and multiple interactions may occur between these objects in parallel. For example, consider a bank with a number of clerks serving customers. Provided it is open, customers may enter the bank and wait in line until they can interact with one of the clerks at the counter. The clerks may in turn interact with other employees or departments of the bank in order to fulfill the requests of the customers. All these activities are happening at the same time and therefore are concurrent.

According to the object-oriented paradigm, each object is an autonomous agent, capable of handling received requests. *Active objects* are objects that are capable of controlling and scheduling the received requests before they are served by the object.

This observation leads to the following conclusions: to properly model real-world situations, concurrent activities must be modeled. Each object must be capable of dealing with multiple, concurrent requests. Requests may be serialized or trigger at the same time.

As stated before, our primary motivation for adopting concurrency lies with the modeling issues that were just discussed. It should be remarked, however, that an object-oriented model lends itself much more for a realisation on a distributed or parallel architecture,

because objects are concurrent *by nature*. It is thus relatively easy to identify tasks that can be executed in parallel.

There are many alternative approaches for introducing concurrency and synchronisation in an object-oriented language. The synchronization can be handled in two different ways:

code level: In this scheme, messages are always accepted for execution. Concurrency is controlled through conventional mechanisms such as semaphores and monitors, which appear as statements that are embedded within the implementations of method bodies.

This category is referred to as *passive* objects and is exemplified by Smalltalk-80. The internal state of objects is only protected against inconsistencies when the methods that affect the state contain synchronisation statements.

object level: In this case we speak about *active* objects: upon reception, messages can be delayed until apt for execution. Thus, synchronisation occurs at the object boundary, thereby protecting the internal consistency of objects. Various types of synchronisation mechanisms can be applied at this level. Examples of languages that synchronise at the object level are POOL, Procol etc.

For concurrent object-oriented systems a systematic development framework is provided by Rumbaugh method explained below:

Rumbaugh method (also known as Object Modeling Technique) of developing concurrent object oriented systems is applicable in all the phases of development. It enumerates the steps that must be taken to accomplish the outcome(s) of that phase. It roughly consists of the following steps:

1. Identify objects, classes, hierarchy and relationships.
2. Identify the synchronization constraints.
3. Specify the object and system behavior, i.e. dynamic behaviour.
4. Specify the functional model using DFD or otherwise.
5. Iterate through the steps to refine the model to the acceptable levels.

The performance and applicability of an object-oriented system can be greatly enhanced if several instances of objects are allowed to co-exist and co-execute in a single environment. Such systems facilitate execution of multiple objects in a single interactive environment.

As a matter of fact concurrency can exist at different levels of abstractions. Concurrency of programs (multiprogramming), for example, has been implemented as multiprocessing or multitasking. Concurrency within an object can be achieved by multithreading.

No matter which level concurrency is in question, a concurrent system must have the ability to control three aspects of execution:

- Concurrent execution of methods
- Inter-method synchronization and communication
- Security

Traditionally, concurrency within objects has been implemented on the tunes of multi-tasking systems. As a process provides abstraction in the later case, threads abstract the later case.

Thread

A thread is a sequence of instructions to be executed within a program. Each thread has its own set of resources such as instruction pointer, set of registers and stack memory. The virtual address space is common to all threads within a process. This enables all the threads to access data on the heap.

Normal processes consist of a single thread of execution that starts in a single method (*main()* in UNIX). Each line of the code is executed exactly one line at a time. Earlier the normal way to achieve concurrency in a program was to use the *fork()* and *exec()* system calls (or equivalent system calls in other operating systems) to create several processes. Each of these processes used to be executed as a single thread of execution.

Since there are a lot of similarities between a process and a thread, a thread is often referred to as a lightweight process.

Threads can be created as instances of the Thread class. It has attributes and methods that create and control a thread, set its priority, and get its status. The namespace of the Thread class is System.Threading assembled in mscorlib.dll. Given below is its syntax.

```
[ComVisibleAttribute(true)]  
[ClassInterfaceAttribute(ClassInterfaceType::None)]  
  
public ref class Thread sealed : public CriticalFinalizerObject, _Thread
```

A single process can create as many threads as required to execute different portions of the code. The program code to be executed by a thread is specified using ThreadStart delegate or the ParameterizedThreadStart delegate. The ParameterizedThreadStart delegate allows one to pass data to the thread procedure.

The following program demonstrates simple threading functionality offered by C++. The code must be compiled using /clr option.

```
using namespace System;  
  
using namespace System::Threading;  
  
public ref class ExampleThread  
{  
public:  
    // The ThreadProc method is called when the thread starts. It loops five times, writing to  
    // the console and yielding the rest of its time slice each time, and then ends.  
  
    static void ThreadProc()  
    {  
        for ( int i = 0; i < 5; i++ )  
        {  
            Console::Write( "Thread Procedure: " );  
            Console::WriteLine( i );  
            Thread::Sleep(0);  
        }  
    }  
};  
  
int main()  
{  
    Console::WriteLine( "From Main Thread: Start a second thread." );
```

```
// Create the thread, passing a ThreadStart delegate that represents the
// ThreadExample::ThreadProc method. For a delegate representing a
// static method, no object is required.
Thread^oThread=gcnew Thread(gcnew ThreadStart(&ExampleThread::ThreadProc ));
// Start ThreadProc. Note that on a uniprocessor, the new thread does not get any
// processor time until the main thread is preempted or yields. Uncomment the
// Thread.Sleep that follows Start() to see the difference.

    Thread->Start();

        //Thread::Sleep(0);

        for ( int i = 0; i < 4; i++ )
        {

            Console::WriteLine( "From Main Thread: Work Work Work." );

            Thread::Sleep( 0 );

        }

Console::WriteLine( "From Main Thread: Call Join(), to wait until ThreadProc ends." );

    oThread->Join();

    Console::WriteLine( "From Main thread: Press Enter to end program." );

    Console::ReadLine();

    return 0;

}
```

Here is the result of a run of this simple example.

From Main Thread: Start a second thread.

From Main Thread: Work Work Work.

Thread Procedure: 0

From Main Thread: Work Work Work.

Thread Procedure: 1

From Main Thread: Work Work Work.

Thread Procedure: 2

From Main Thread: Work Work Work.

Thread Procedure: 3

From Main Thread: Call Join(), to wait until ThreadProc ends.

Thread Procedure: 4

From Main Thread: Press Enter to end program.

Multithreading

A single process can be further broken into several threads so as to enhance the overall performance of the process. This is called multithreading. Hence, multithreading is a technique for achieving concurrency within a process.

Multithreading vs. Multiprocessing

Multithreading allows an application with multiple threads running within a process. On the other hand, multiprocessing refers to an application organized across multiple processes.

Context switching and synchronization costs are comparatively lower in threads. Since the address space is shared among threads no extra work is required to access them. However, because of this the failure of one thread in a process can ring down all the other threads of that process. In contrast, since processes are insulated from each other by the operating system, an error in one process cannot bring down another process. Processes may run on behalf of different users and therefore may have different permissions unlike threads.

Limitations of Threads

Despite having so much promises threads do carry a lot of pitfalls along the way. Programmers should take proper caution while programming with threads. Some of the caveats of thread programming is discussed below.

Race Conditions

A race condition is said to exist where the behavior of code depends on the interleaving of multiple threads. Single-threaded codes runs as a single sequence of statements from which we can assume that data does not change between statements. However, we cannot make the same assumption for a single program statement which may compile into more than one lower level statements, i.e., we cannot guarantee the outcome of a statement such as *total++*; if *total* is shared between multiple threads. This is perhaps the most fundamental problem with multi-threaded programming. Consider the following code that depicts a logical race condition.

```
int SharedVar = 20;
void* ThreadOne(void*)
{
    while(SharedVar > 0)
    {
        CallSomeMethod();
        —SharedVar;
    }
}
```

When this code is executed as a single thread, *CallSomeMethod* will execute 20 times. However, imagine that we start a number of threads, all executing *ThreadOne()*. However, with more than one thread existing simultaneously the method *CallSomeMethod()* will most likely be executed too many times. Exactly how many times depends on the number of threads spawned, computer architecture, operating system scheduling and luck. The problem arises because we do not test and update *SharedVar* as an atomic operation, so there is a period where the value of *SharedVar* is incorrect. During this time other threads can pass the test when they really shouldn't have.

The value of *SharedVar* on exit tells us how many extra times *CallSomeMethod()* is called. The value of *SharedVar* on exit will be 0 if there is a single thread running *CallSomeMethod()*.

Race conditions can be avoided using an object called mutex.

A mutex is synchronization primitive provided by the operating system that can be used to ensure that a section of code executes by one thread at a time. A mutex has two states - *locked* and *unlocked*. In locked state any further attempt to lock it will suspend the thread. The waiting threads can acquire lock on mutex and resume execution only when the mutex becomes unlocked. The thread that locks the mutex can only unlock it. Here is the solution for the above code using mutex.

```
int SharedVar = 20;
mutex MutexVar;
void ThreadTwo()
{
    while(SharedVar > 0)
    {
        bool flag = false;
        {
            mutex::lock(MutexVar);
            if(SharedVar > 0)
            {
                --SharedVar;
                flag = true;
            }
        }
        if(flag) CallSomeMethod();
    }
}
```

As is evident, the shared variable is checked and updated as an atomic operation so that the race condition does not occur.

Deadlock

The concurrency may lead to a situation where one or more threads wait for resources that can never become available. Such a condition is called a deadlock. Consider the following code that illustrates a deadlock.

```
mutex X;
mutex Y;
void ThreadThree()
{
    int ct = 0;
    while(true)
    {
        mutex::lock(X);
        mutex::lock(Y);
        cout << "ThreadThree in action" << ++ct << "\n";
    }
}
```

```
void ThreadFour()
{
    int ct = 0;
    while(true)
    {
        mutex::lock(Y);
        mutex::lock(X);
        cout << "ThreadFour in action" << ++ct << "\n";
    }
}
```

When this code is run a deadlock is possible. The two threads may enter a situation where ThreadThree is waiting for ThreadFour to release the locks while ThreadFour waits for ThreadThree to release the locks.

To resolve this simple deadlock all we need to do is to ensure that we lock resources in a consistent order. Thus, changing ThreadFour to lock X before Y ensures there will be no deadlock.

9.14 SUMMARY

In C++ when a function is overloaded, many copies of it have to be created, one for each data type it act on. A template function may be defined as an unbounded functions. The definition of the templates begins with the template keyword followed by a comma-separated list of parameter types enclosed within the less than (<) and greater than (>) signs.

Templates classes may be defined as the layout and operations for an unbounded set of related classes. While template functions and classes are usable for any data type, that would hold true only, as long as the body of functions or the class is identical throughout.

Specialization would consist of instantiating the templates definition for a specific data type if the templates instance does not exist, and the definition is not visible, generate an error.

Syntactic errors are relatively easy to find and correct, even if the resulting error messages are unclear. Execution errors, on the other hand, can be much more trouble some. Logical errors can be very difficult to detect, since that output resulting from a logically incorrect program may appear to be error-free. Errors isolation is useful for locating an error resulting in a diagnostic message.

9.15 KEYWORD

Template: A template function may be defined as an unbounded functions. All the possible parameters to the function are not known in advance and a copy of the function has to be created as and when necessary.

Template Classes: Template classes may be defined as the layout and operations for an unbounded set of related classes.

Syntactic errors: Syntactic errors are relatively easy to find and correct, even if the resulting error messages are unclear.

Logical errors: Logical error can be very difficult to detect, since the output resulting from a logically incorrect program may appear to be error free.

Watch value: Watch value is the value of a variable or an expression which is displayed continuously as the program executes.

Break point: A breakpoint is a temporary stopping point within a program.

Stepping: Stepping refers to the execution of one instruction at a time, typically by pressing a function key to execute each instruction.

9.16 REVIEW QUESTION

1. What are Templates?
2. Define template instantiation.
3. List down the rules for using templates.
4. How can you get a template function instantiated?
6. Explain the use of
 - a. WATCHVALUES
 - b . BREAKPOINTS
 - c. STEPPING.
7. Fill in the Blanks:
 - (a) A template function may be defined as an_____
 - (b) Template functions and classes are usable for any_____
 - (c) _____would consist of instantiating the template definition for specific data type.
 - (d) _____errors are relatively easy to find and correct, even if the resulting error messages are unclear.
 - (e) Error isolation is useful for locating an error resulting in a _____message.
 - (f) The goal of object-oriented software development is to construct a model of _____.
 - (g) _____activities are those activities that take place simultaneously.
 - (h) _____objects are objects that are capable of controlling and scheduling the received before they are served by the object.
 - (i) There are many alternative approaches for introducing concurrency and _____is an object-oriented language.
 - (j) For concurrent object-oriented systems a systematic development framework is provided by_____method.

Answers to Review Questions

- | | |
|---------------|-------------------------|
| (a) Real word | (b) Concurrent |
| (c) Active | (d) Synchronization |
| (e) Rum Baugh | (f) unbounded functions |
| (g) data type | (h) specialization |
| (i) syntactic | (j) diagnostic |

9.17 FURTHER READINGS

Herbert Schildt, *The complete Reference-C++*, Tata Mc Graw Hill.

Robert Lafore, *Object Oriented Programming in Turbo C++*, Galgotia Publications.

E. Balagurusamy, *Object Oriented Programming through C++*, Tata McGraw Hill.