MSc-IT

ADVANCED CONCEPTS IN OBJECT ORIENTED PROGRAMMING

GAUHATI UNIVERSITY

Institute of Distance and Open Learning

Semester- I

MSc-IT

Paper: INF 1016

(under CBCS)

Advanced Concepts in Object Oriented Programming

www.idolgu.in

M.Sc.-IT-19-I-1016

GAUHATI UNIVERSITY

Institute of Distance and Open Learning

First Semester

(under CBCS)

M.Sc.-IT

Paper: INF-1016

ADVANCED CONCEPTS IN OBJECT ORIENTED PROGRAMMING

Contents:

BLOCK I: OBJECT ORIENTED PROGRAMMING
Unit 1 : Introduction to Object Oriented Programming
Unit 2 : Introduction to C++
Unit 3 : Control Statements in C++
Unit 4 : Array and Strings in C++
Unit 5 : Pointers and Reference Variables in C++
Unit 6 : Concept of Function in C++
Unit 7 : Data abstraction
Unit 8 : Inheritance
Unit 9 : Polymorphism
Unit 10 : Exception handling
Unit 11 : File handling

BLOCK II: OBJECT ORIENTED DESIGN
Unit 1 : Introduction to OO Design
Unit 2 : Object Modeling Techniques (OMT) tools
Unit 3 : Phases of Object-Oriented Development

Contributors:

Dr. Swapnanil Gogoi (Block I : Units- 1,4,5 and 9)
Assistant Professor, GUIDOL

Dr. Khurshid Alam Borbora (Block I: Units- 2,6,7 and 10)
Assistant Professor, GUIDOL

Dr. Ridip Dev Choudhury (Block I : Unit- 3)
Associate Professor, HCB School of Science and Technology
Krishna Kanta Handiqui State Open University, Assam

Dr. Dipen Nath (Block I : Unit- 8)
Assistant Professor, Mangaldai College

Dr. Naba Jyoti Sarma (Block I: Unit- 11)
Assistant Professor, Nalbari Commerce College (Block II: Unit-3)

Mr. Bireswar Banik (Block II:Units- 1 and 2)
Assistant Professor, Nalbari Commerce College

Content Editor:

Dr. Pranab Das
Asst. Prof. (Senior), Assam Don Bosco University
Azara, Guwahati

Course Coordination:

Prof. Dandadhar Sarma Director, IDOL, Gauhati University
Prof. Anjana Kakoti Mahanta Prof., Dept. Computer Science, G.U.

Cover Page Designing:

Bhaskar Jyoti Goswami IDOL, Gauhati University

May, 2022

© Copyright by IDOL, Gauhati University. All rights reserved. No part of this work may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise.

Published on behalf of Institute of Distance and Open Learning, Gauhati University by the Director, and printed at Gauhati University Press, Guwahati-781014.

BLOCK I:

OBJECT ORIENTED PROGRAMMING

UNIT 1: INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

UNIT STRUCTURE

1.1 Introduction
1.2 Objectives
1.3 Procedural Programming
1.4 Structured Programming
1.5 Object Oriented Programming (OOP) Paradigm
 1.5.1 Differences between OOP and Procedural Programming
1.6 Properties of Object Oriented Programming
 1.6.1 Encapsulation
 1.6.2 Data Abstraction
 1.6.3 Inheritance
 1.6.4 Polymorphism
1.7 Concept of Class and Object
1.8 Dynamic Binding
1.9 Message Passing
1.10 Advantages and Disadvantages of Object Oriented Programming
1.11 Applications of Object Oriented Programming
1.12 Object Based Language
1.13 Summing Up
1.14 Possible Questions
1.15 References and Suggested Readings

1.1 INTRODUCTION

The concept of structured programming was introduced in the later part of 1960s. Procedural programming with structured programming approach became very popular in 1970s and 1980s. It was observed that the complexity of software development with procedural programming had increased all the time. It happened due to the requirement of more dynamic approach for software development. At

that time, a new programming paradigm was explored for the development of software systems with greater extensibility, reusability, reliability and maintainability. As a result, Object Oriented Programming paradigm was introduced to the computer software industry.

The basic concepts of procedural programming and structured programming are discussed in the first part of this unit. The main aim of this unit is to introduce the Object Oriented Programming (OOP) paradigm. The basic properties and features of OOP are discussed along with the advantages and applications of OOP.

1.2 OBJECTIVE

After reading this unit you are expected to be able to learn:

- What is Procedural Programming?
- What is Structured Programming?
- What is Object Oriented Programming?
- Different between Object Oriented Programming and Procedural Programming.
- Different properties of Object Oriented Programming.
- The concept of class and object.
- The concept of dynamic binding and message passing.
- About the advantages and applications of Object Oriented Programming.
- What is Object Based Language?

1.3 PROCEDURAL PROGRAMMING

Procedural programming is a type of Imperative programming. In Imperative programming, a program consists of clearly defined step by step commands to computer. Computer executes these commands or instructions to perform a specific computational task and obtain the desired outputs.

In Procedural programming, a program is organized as a set of sub-programs or procedures termed as functions. Each function is a collection of step by step instructions to perform some specific task. Functions have to use different data as per requirement of their part of job and these data are stored in different variables or memory locations. One of these functions is termed as main function where the program execution starts. Functions other than the main function may be called at any time of the program execution by the main function or other functions or itself to obtain the desired output.

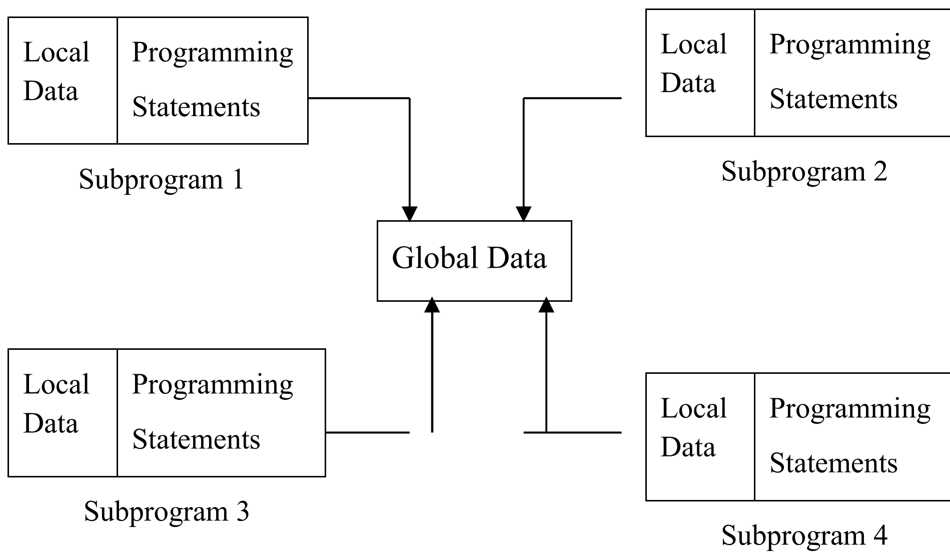


Figure 1.1: General Structure of Procedural Programming

Example of Procedural programming language: FORTRAN, COBOL, ALGOL etc.

Some other important features of Procedural programming are stated as follows:

1) Top down approach of program design is followed in Procedural programming.
2) More importance is given on the algorithms or functions than the data in Procedural programming.
3) Variables are broadly categorized into two groups namely local and global. A local variable is declared inside a

function and its scope is limited to that function. On the other hand, global variables are not declared inside any of the function available in the program. So, all available functions can access the declared global variables.

4) Data can be passed from function to function using parameter passing mechanism.
5) Maintenance of data security is difficult as data are stored in global variables.
6) Built-in or library or predefined functions may available for specific computational task.
7) Extensibility and reusability are difficult to achieve.
8) It is difficult to characterize real world objects in the program design.

1.4 STRUCTURED PROGAMMING

Structured programming was introduced to handle the complexity of large software projects. In this approach, a program is divided into independent sub-programs or modules. Each module consists of a set of interrelated functions. Three types of control structure are used in structured programming to control the execution of a program. These three control structures are mentioned as follows:

1) Sequence control structure: It allows execution of the instructions one by one in an orderly manner.
2) Selection control structure: It allows program to opt for a particular control flow from two or more available flows.
3) Iteration control structure: It allows execution of one or more instructions repeatedly two or more times as required by the program.

Example of Structured programming language: C, Ada, Pascal etc.

Some other important features of Structured programming are stated as follows:

1) Programs are easy to understand and maintain.
2) It is a programmer friendly approach and so program development becomes easier.

3) Error detection and correction in a program are easier.
4) Algorithm is more important than data in this programming approach.
5) Global data are shared by all modules of a program. As a result, data security and integrity may be compromised in this programming approach.
6) Each module may have local data as per requirement of its computational job.
7) Information can be passed from procedure to procedure.
8) Modification of a program or inclusion of new features to a program is difficult to carry out.
9) Reusability is also difficult to achieve in Structural programming.

STOP TO CONSIDER

Most of the Structured Programming languages can also be considered as Procedural Programming. For example: C language can also be considered as an example of Procedural programming.

1.5 OBJECT ORIENTED PROGRAMMING PARADIGM

The concept of Object Oriented Programming was introduced in 1960s and it became very popular in 1990s. The core intend of this concept is to increase extensibility, reusability, reliability and maintainability of software systems. In this way of programming, relationship between real world objects with the program design is established to solve a computational problem.

Object oriented programming (OOP) can be defined as a programming paradigm that is based on the concept of class and object. An object is an instance of some class and it contains data and functions. In OOP, programs are organized as a collection of cooperative objects.

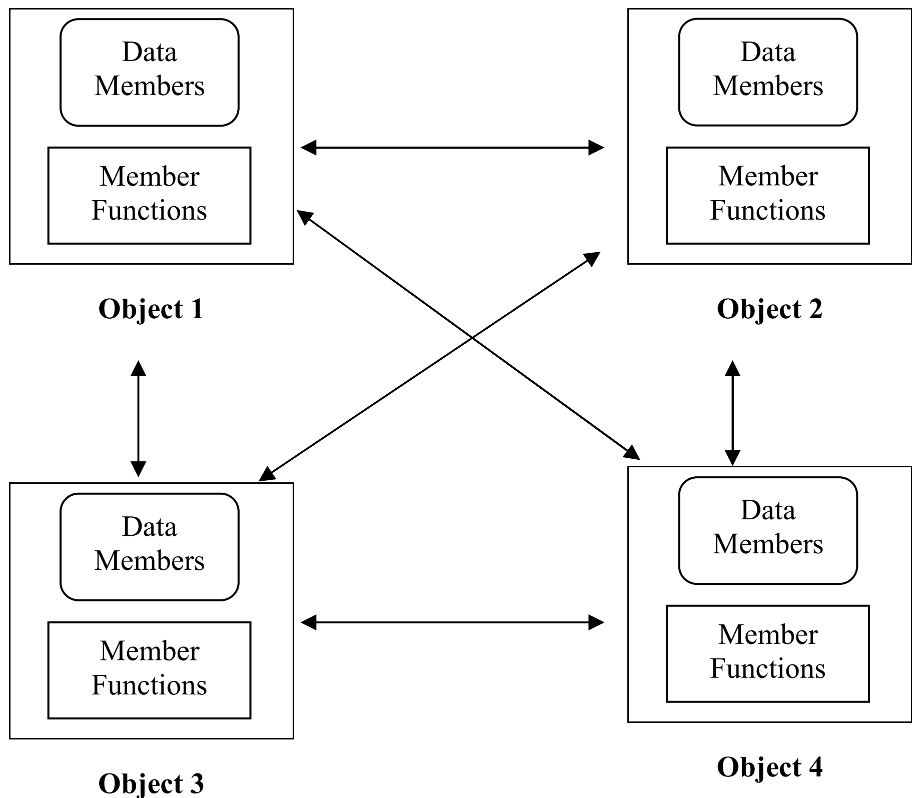


Figure 1.2: General Structure of Object Oriented Programming

Examples of OOP languages: C++, Java, C#, Python, R, Visual Basic.NET etc.

Some other important features of Object Oriented Programming are stated as follows:

1) Program development is easier using OOP.
2) In OOP, importance is given more on data than the function or algorithm.
3) In OOP, data can be hidden within a class to confirm limited or restricted data access.
4) Data security and data integrity can be maintained in OOP with the help of the mechanism termed as data hiding.
5) Modification of an existing program or inclusion of new features to a program can be easily performed using the concept of inheritance in OOP.
6) Partion work in a project based on objects.

7) Message passing strategies simplify the interface specification with external applications by allowing objects to communicate with one another.

1.5.1 Differences between OOP and Procedural Programming

Main differences between OOP and Procedural Programming are stated as follows:

1) In case of procedural programming, functions have been given more importance than data and a program is divided into functions. On the other hand, in case of object oriented programming, data are more important than functions and a program is divided into objects.
2) In procedural programming, access specifiers are not available and as a result data hiding is not possible. On the other hand, in object oriented programming, data hiding is possible with the help of access specifiers like private and protected. So, data are more secured in object oriented programming.
3) In case of procedural programming, modification of programs by adding new functions and data is a complex process. On the other hand, in case of object oriented programming, modification of programs can be easily performed by using inheritance. So inclusion of new functionality and data can be easily performed in object oriented programming.

1.6 PROPERTIES OF OBJECT ORIENTED PROGRAMMING

Object Oriented Programming (OOP) paradigm has four key properties that are Encapsulation, Abstraction, Inheritance and Polymorphism.

1.6.1 Encapsulation

Data security and data integrity are two most important goal of OOP. Now, the question is how it can be achieved in OOP? The answer of this question is Encapsulation. Encapsulation is one of the core

properties of OOP by which data are kept together with the methods or functions that operate on those data in a single unit. In this process, data access can be restricted from the outside world of the corresponding unit.

Encapsulation is implemented using the concept of 'class'. Creating a class in OOP provide the mechanism to combine data and related functions in a single unit. A class can hide its data from any outside unauthorized access.

1.6.2 Abstraction

In Object Oriented Programming, Abstraction is the second core property. It provides users a mechanism that represents only the essential elements to them and hides irrelevant background details from them. It helps user to implement complex software systems without considering the hidden complexity. There are two types of abstraction available in OOP that are Data abstraction and Control abstraction**.** Hiding the details about data is called as Data abstraction and hiding the implementation details is referred as Control abstraction.

In OOP, Abstraction can be provided to the users by the process of Encapsulation. A class is created to provide abstraction by hiding all the necessary elements or information from the outside world of that class. So, in OOP, classes use the concept of data abstraction and as a result they are also termed as Abstract Data Types (ADTs).

1.6.3 Inheritance

Inheritance is the third basic characteristic of OOP. In Biology, inheritance refers the process of receiving features or qualities by children genetically from their parents. Similarly, in OOP, inheritance refers to the process of obtaining the properties of one class by another class. So, we can state that a child class inherits some or all properties of its parent class. Parent class can also be termed as base class and child class can also be referred as derived class or sub class. Now consider a situation where it may happen that different types of objects are available from different classes which share some common properties. In such situation, the concept of inheritance can be used where one parent class can be created with the common properties and child classes will inherit these properties.

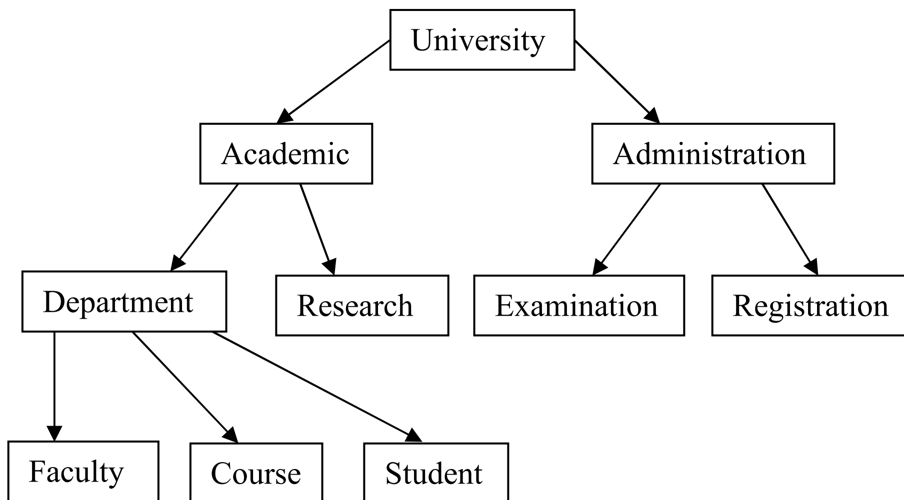


Figure 1.3: Example of Inheritance

Now consider figure 1.3 as an example of inheritance. In this example, Academic and Administration are the derived classes or sub classes of University class. So, both of these sub classes inherit some common characteristics of University class like different basic details of the University. Department and Research are the sub classes of Academic class. Again Faculty, Course and Student are the sub classes of Department class.

As inheritance allows the access of some or all properties of an existing class by a newly created derived class, we can say that reusability of code and information can be possible with the help of inheritance. On the other hand, new features can be added to a software system by creating new sub classes to existing classes. In this process, modifications of existing classes are not required and new derived classes possess new properties as well as the properties of their parent classes. So extensibility is easier in OOP with the help of inheritance.

1.6.4 Polymorphism

Polymorphism is is another important OOP concept. Polymorphism provides the mechanism to use one function name for the implementation of more than one computational job. For example, a function name 'Summation' can be used to estimate the summation of two integer numbers and it can also be used to estimate the summation of two real numbers using the concept of polymorphism. It means that

in a particular program, multiple functions with same name but different computational jobs are possible. An operator can also be used for different purposes using this feature. For example, operator '+' can be used to perform arithmetic addition of two numbers and it can also be used to concatenate two strings using the concept of polymorphism.

There are two types of Polymorphism available in OOP that are Compile time polymorphism and Runtime polymorphism. Compile time polymorphism can be divided into two types that are Function overloading and Operator overloading. On the other hand, Runtime polymorphism can be termed as Function overriding or Virtual function.

1.7 CONCEPT OF CLASS AND OBJECT

We have already learnt that encapsulation is implemented by the concept of 'class' in OOP. We can define 'class' as a model or an outline that combine functions or operations with related data in a single unit. So a class can be referred as a user defined data type which contains one or more functions and related data or properties. A class can also be considered as an abstract data type. Accessing information in a class can be restricted from external entities by using access specifiers like private and protected. This feature is called data hiding in OOP. Data security and integrity can be maintained using data hiding in OOP.

In general, classes can be categorized into two types that are abstract class and concrete class. If a class contains one or more function declarations but the implementation or code of these functions are not available, then the class is referred as abstract class. Derived classes of an abstract class contain the implementations of such functions. On the other hand, if a class contains the implementations of all its functions then it is termed as concrete class.

In OOP, object can be defined in many ways, in the simplest term, it is actually instantiation of a class. So, we can state that an object is nothing but a variable of a class. In this way, an object can be defined as a block of memory locations which store code of one or more functions and related information. Functions of an object perform their task by accessing the information available inside the object.

Object of a class is used to represent real-world entities in computer memory to develop a software system.

For example: Consider a class 'Student' with the following data members and functions.

Class name : Student

Data members :

1) Student_Name
2) Student_Roll_Number
3) Student_Address
4) Student_Contact
5) Student_Department
6) Student_Course
7) Student_Percentage

Functions :

1) Input_Student_Details( )
2) Display_Student_Details( )

Now, if we create instances or variables of class 'Student' like 'Student1', 'Student2', 'Student3' then these three are the objects of class 'Student'. Representation of object 'Student1' is shown as follows.

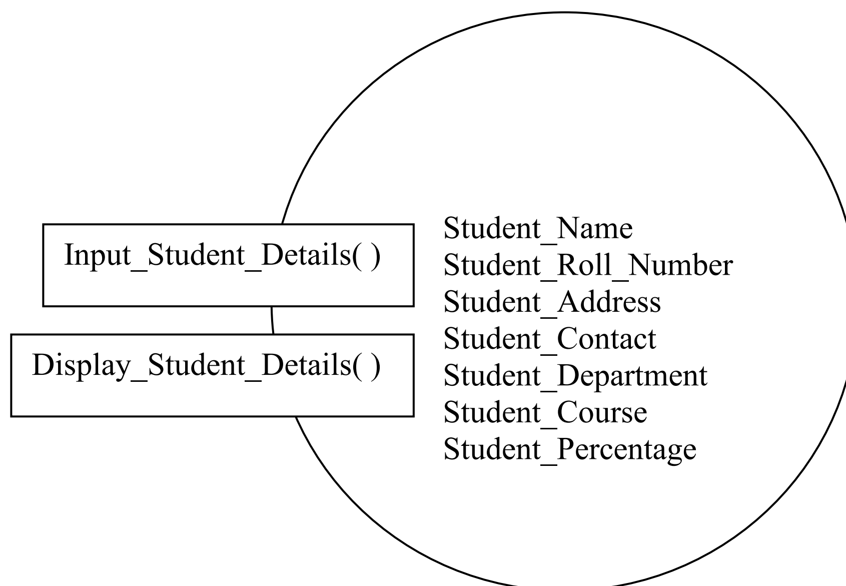


Figure 1.4: Representation of object 'Student1'

STOP TO CONSIDER

Object of an abstract class cannot be created.

1.8 DYNAMIC BINDING

Let us consider a situation where the function name is same in both derived class and base class. Both these functions contains same number and type of parameters. But the both functions have different implementations or we can say that both perform different jobs.During a function call , which function will be executed is depends upon the object that is being referenced. If derived class object is referenced, then function in the derived class will be executed otherwise the function in the base class will be executed. So, which among the two functions with the same name will be executed is identified only at runtime. When linking of a function call to its actual code is identified only at the run time then this process is called dynamic binding. In OOP dynamic binding can be implemented with the help of inheritance and function overriding.

1.9 MESSAGE PASSING

An object is a unit of one or more data and functions. In OOP, a program is a collection of different objects from different classes. If required, these objects may communicate each other by sending and receiving messages. This process is called Message Passing. Message Passing is implemented in OOP by calling a member function of an object and the parameter passed to that function is the message or information to be sent.

1.10 ADVANTAGES AND DISADVANTAGES OF OOP

Advantages of OOP are stated as follows:

1) The complexity of developing programs is reduced in OOP as it is similar to working with real-world objects. So, complex and large systems can be easily implemented by OOP.
2) OOP is more reliable and secured as data integrity and data security can be maintained efficiently.
3) Error detection and correction is easy in OOP as each object is an independent and isolated entity.
4) In OOP, modification can be made in any class without affecting other portions of the program.
5) Re-usability of code is possible in OOP.
6) In OOP, new features can be included by writing new classes and objects without modifying existing classes and objects.
7) In OOP, inheritance can be implemented that allows derivation of new classes from existing classes.

Disadvantages of OOP are stated as follows:

1) In OOP, program size may be larger than procedural programs and as a result object-oriented programs are slower.
2) OOP is not appropriate for all types of software development.

1.11 APPLICATIONS OF OOP

In the present era, Object Oriented Programming (OOP) becomes very popular in the software industry and among software

developers due to its different significant advantages. As a result, day by day its application areas have been increased.

Some of the important application areas of OOP are stated as follows:

- Object Oriented databases
- Development of Real time system
- Embedded systems
- Client-Server systems
- Decision support systems
- Object oriented Operating Systems
- Simulation and Modeling
- Multimedia applications
- Graphical User Interfaces
- Office automation systems
- Development of Expert Systems
- Artificial Intelligence
- Computer Aided Design and Computer Aided Manufacturing systems
- Software services related to Internet

1.12 OBJECT BASED LANGUAGE

Object Based languages are the programming languages which support encapsulation and object identity but does not support inheritance, polymorphism and message passing.

Examples of Object Based Languages: JavaScript, Visual Basic (VB), Ada etc.

STOP TO CONSIDER

Object Oriented Programming languages support all the properties of Object Based Programming languages. On the other hand, Object Based Programming languages do not support all the features of Object Oriented Programming languages.

CHECK YOUR PROGRESS

1. Multiple choices

(A) Which of the following is not true in case of Procedural programming?
(i) A program is organized as a collection of methods.
(ii) Reusability is difficult to achieve
(iii) Data is more important than function.
(iv) None of the above

(B) Data security is difficult to maintain in Procedural programming because ________.
(i) data are stored in local variable.
(ii) data are stored in global variable.
(iii) data are not important.
(iv) None of the above.

(C) Which of the following is not a key control structure available in Structured programming?
(i) Jump control structure
(ii) Sequence control structure
(iii) Selection control structure
(iv) Iteration control structure

(D) Which of the following is not a feature of Structured programming approach?
(i) Programs are easier to understand .
(ii) Error correction and detection are easier.
(iii) Extensibility can be easily achieved.
(iv) None of the above.

(E) Which of the following is a feature of Object Oriented programming approach?
(i) Reusability is easier to achieve.
(ii) Method is important than data
(iii) Maintenance of data integrity is difficult.
(iv) All of the above.

(F) Which of the following is a key property of OOP?
   (i) Encapsulation
   (ii) Inheritance
   (iii) Polymorphism
   (iv) All of the above

(G) In OOP, Encapsulation is implemented by the concept of _____.
   (i) object
   (ii) class
   (iii) function
   (iv) None of the above

(H) When a class acquires the properties of another class then the class is referred as _______classes.
   (i) base
   (ii) derived
   (iii) parent
   (iv) abstract

(I) Which of the following is an advantage of inheritance?
   (i) Enhanced data security
   (ii) Reusability
   (iii) Extensibility
   (iv) Both (ii) and (iii)

(J) A derived class can access some or all properties of its ______ class.
   (i) base
   (ii) abstract
   (iii) child
   (iv) None of the above

(K) _________ allows using one function name for multiple computational jobs.
   (i) Encapsulation
   (ii) Inheritance
   (iii) Polymorphism
   (iv) Abstraction

(L) Which of the following is not a type of polymorphism?

(i) Function overloading
(ii) Function overriding
(iii) Operator overloading
(iv) Operator overriding

(M) Which of the following is not compile time polymorphism?

(i) Function overriding
(ii) Function overloading
(iii) Operator overloading
(iv) None of the above

(N) Which mechanism provides linking of a function call to its actual code only at the run time?

(i) Message passing
(ii) Abstraction
(iii) Dynamic binding
(iv) None of the above

(O) Object Based languages do not support ______.

(i) Inheritance
(ii) Polymorphism
(iii) Message passing
(iv) All of the above

2. State whether true or false

(A) Object of an abstract class cannot be created.
(B) Concrete class does not contain the implementations of all its declared functions.
(C) In OOP, data hiding is possible with the help of inheritance.
(D) Object Based languages support encapsulation.
(E) Object can be defined as a variable of a class.

1.13 SUMMING UP

In this unit, the basic concepts of procedural programming and structured programming were discussed followed by introduction to Object Oriented Programming (OOP).

In Procedural programming, a program is organized as a set of sub-programs or procedures termed as functions. In this approach, more importance is given on the algorithms or functions than the data. In Procedural programming, variables are categorized into two types that are local and global. Extensibility, reusability and data security are difficult to achieve in Procedural programming.

In Structured programming approach, a program is divided into independent sub-programs or modules. Each module consists of a set of interrelated functions. Sequence control structure, Selection control structure and Iteration control structure are three types of control structure that are used in structured programming to control the execution of a program. Programs in Structured programming approach are easy to understand and maintain. Error detection and correction in a program are also easier. In this approach, algorithm is more important than data. Reusability and extensibility are difficult to achieve in Structural programming.

Object oriented programming (OOP) can be defined as a programming paradigm that is based on the concept of class and object. In OOP, importance is given more on data than on function or algorithm. In this programming paradigm, data can be hidden within a class to confirm limited or restricted data access and hence data security and data integrity can be maintained in this programming approach. Reusability and extensibility are easier to achieve in OOP.

Object Oriented Programming (OOP) paradigm has four key properties that are Encapsulation, Abstraction, Inheritance and Polymorphism. Encapsulation is the mechanism which combines related data and the methods or functions in a single unit. In OOP, Encapsulation is implemented using the concept of ‘class’. Abstraction provides users a mechanism that represents only the essential elements to them and hides irrelevant background details from them. Hiding the details about data is called as Data abstraction and hiding the implementation details is referred as Control abstraction. In OOP,

inheritance refers the process of obtaining the properties of one class by another class. Reusability and extensibility can be possible with the help of inheritance in OOP. Polymorphism provides the mechanism to use one function name for the implementation of more than one computational job. It also provides the mechanism to use an operator for different purposes. There are two types of Polymorphism available in OOP that are Compile time polymorphism and Runtime polymorphism. Function overloading and Operator overloading are Compile time polymorphism and Function overriding or Virtual function is the Runtime polymorphism.

A class can be defined as a model or an outline that combine functions or operations with related data in a single unit. An abstract class does not contain the implementations or code of one or more declared functions. If a class contains the implementations of all its functions, then it is termed as concrete class.

In OOP, an object can be defined as instantiation of a class.

Dynamic binding is the mechanism where linking of a function call to its actual code can be identified only at the run time.

Message Passing is the process by which objects can communicate each other by sending and receiving messages.

Some of the important application areas of OOP are Object Oriented databases, Development of Real time system, Embedded systems, Client-Server systems etc.

Object Based languages are the programming languages which support encapsulation and object identity but does not support inheritance, polymorphism and message passing.

ANSWER TO CHECK YOUR PROGRESS

1. (A) (iii), (B) (ii), (C) (i), (D) (iii), (E) (i), (F) (iv), (G) (ii), (H) (ii), (I) (iv), (J) (i), (K) (iii), (L) (iv), (M) (i), (N) (iii), (O) (iv)
2. (A) True, (B) False, (C) False, (D) True, (E) True

1.14 POSSIBLE QUESTIONS

1) Write down the important features of Procedural programming and Structured programming.
2) Define Object Oriented Programming (OOP). Give example of any two OOP languages.
3) Differentiate between OOP and Procedural programming.
4) Write down the advantages and disadvantages of Structured programming.
5) Write down the advantages and disadvantages of OOP.
6) Explain different key properties of OOP.
7) Write down the advantages of Inheritance in OOP.
8) Define dynamic binding.
9) What is message passing?
10) Give any three application areas of OOP.
11) Differentiate between Object Oriented Programming and Object Based Programming.

1.15 REFERENCES AND SUGGESTED READINGS

1) Venugopal, K. R., Rajkumar, Ravishankar, T. *Mastering C++*. Tata McGraw-Hill Education, 2001.
2) Balagurusamy, E. *Object Oriented Programming with C++*. Tata McGraw-Hill, 2006

UNIT 2 : INTRODUCTION TO C++

Unit Structure:

2.1 Introduction
2.2 Unit Objectives
2.3 History of C++
2.4 Features of C++
2.5 Structure of a C++ Program
2.6 Writing, Compiling and Executing a C++ Program
2.7 Errors in C++
2.8 C++ Character Set
2.9 C++ Tokens
 2.9.1 Keywords
 2.9.2 Identifiers
 2.9.3 Constants, Operators and Special Characters
2.10Data Types
 2.10.1Primary/Built-in Data type
 2.10.2 Derived Data Type
 2.10.3 User Defined Data Type
 2.10.4 typedef
2.11Variables& Storage Classes
2.12Output and Input in C++
2.13Operators
 2.13.1Assignment Operator
 2.13.2Arithmetic Operators
 2.13.3 Relational Operators
 2.13.4 Logical Operators
 2.13.5 Increment and Decrement Operators
2.13.6 Conditional Operator
2.13.7 Bitwise Operators
2.13.8 Special Operators
2.14 Operator Precedence and Associativity
2.15 Summing Up
2.16 Answers to Check Your Progress
2.17 Possible Questions
2.18 References and Suggested Readings

2.1 INTRODUCTION

C++ is an object oriented programming language. Because C++ is a superset of C, most C constructs are allowed in C++ and have the same meaning.Almost all C and C++ programmes are nearly identical, with a few minor exceptions.The C++ language is case-sensitive. This means that letters in uppercase and lowercase are regarded to be distinct. A variable called sum is not the same as Sum, which is not the same as SUM.C++'s object-oriented features enable programmers to write complex programs that are clear, scalable, and easy to maintain.

2.2 UNIT OBJECTIVES

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

- know history of C++ language and its features.
- understand the structure of a C++ program.
- know how to write, compile and execute a C++ program in Windows and Linux operating system.
- Identify the different types of errorsthat program compilation.
- Explain different types of tokens like keywords, identifiers, constants, strings and operators in C++
- know about data types and its categories in detail.
- learnthe basic concepts of variables, Input/Output and Functions in C++.
- explain different operators and their use.Also understand the concept of operator precedence and associativity.

2.3 HISTORY OF C++

The C++ programming language has a history going back to 1979, when Bjarne Stroustrup was doing work for his Ph.D. thesis. One of the languages, Stroustrup had the opportunity to work with was a language called Simula which is regarded as the first language to support the object-oriented programming paradigm. Stroustrup found that this paradigm was very useful for software development, however the Simula language was far too slow for practical use.

Shortly thereafter, he began work on "C with Classes", which as the name implies was meant to be a superset of the C language. His

language included classes, basic inheritance, inlining, default function arguments, and strong type checking in addition to all the features of the C language.

In 1983, the name of the language was changed from C with Classes to C++. The ++ operator in the C language is an operator for incrementing a variable, which gives some insight into how Stroustrup regarded the language.

In 1985, Stroustrup's reference to the language entitled *The C++ Programming Language* was published. That same year, C++ was implemented as a commercial product.In 1990, *The Annotated C++ Reference Manual* was released. The same year, Borland's Turbo C++ compiler was released as a commercial product. Turbo C++ added a plethora of additional libraries which had a considerable impact on C++'s development. Although Turbo C++'s last stable release was in 2006, the compiler is still widely used.

2.4 FEATURE OF C++

C++ Language is simple in terms of syntax and functionalities. Following are the important features of C++ language.

- Robust language with a very rich set of operators and library functions. C++ has provisions for creation of programmers' own library of functions.
- Due to its variety in data-types and also rich set of powerful in-built functions, the programs written in it are fast and efficient.
- C++ is an object-oriented language, unlike C which is a procedural language. This is one of the most important features of C++. It employs the use of objects while programming. These objects help you implement real-time problems based on data abstraction, data encapsulation, data hiding, and polymorphism.
- Although C++ is not platform-independent as compiled programs on one operating system won't run on another operating system But in another term, portability refers to using the same piece of code in varied environments.

- It is important to note that C++ is a high-level programming language, unlike C which is a mid-level programming language. It makes it easier for the user to work in C++ as a high-level language as we can closely associate it with the human-comprehensible language, that is, English.

2.5 STRUCTURE OF A C++ PROGRAM

The structure of C++ program with its programming elements is shown below:

Pre-processor directive (e.g., Header File inclusion)
Global Variable(s)
User-defined Function/ClassDeclarations
Main Function
User-defined Function/Class Definitions

A simple C++ program that prints 'Hello World' on the monitor(output screen) is shown below.

```
#include<iostream.h>
void main()
{
	cout<<"Hello World!";
}
```

Now, let's try to understand the above program as per the structure mentioned above.

First line is the **Pre-processor Directives**. We know that natural languages like Assamese, English etc. have their own dictionary/library. Thus, C++ also has its library and it consists of some pre-written files, known as **header files**. Here, in this program, the header file, namely '**iostream.h**' is included as it the basic header file that needs to be included for a simple C++ program.

STOP TO CONSIDER

The extension of a header file in C++ is '**.h**'. The header file, **iostream.h**, contains the basic input/output functions and other items those are important for a simple C++ program.

The program doesn't have any declaration for **Global Variable(s)**.

The program doesn't have any**User-defined Function/Class Declaration**.

Every C++ program must have a function, main () where the execution begins. All the statements of the programs are written inside the main() function enclosed within { }. The statement

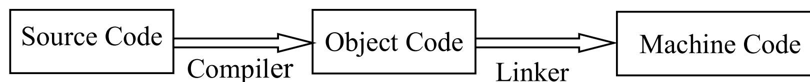
```
cout<<"Hello World!";
```

will print the message, **Hello World!**, on the computer screen/monitor. The necessity of the use of '**void**' will be understood in the following sections/units.

Each and every line in the above program is called a **Statement**. As in English language a sentence is marked end with full-stop(**.**), a **C++ statement** ends with a semicolon(**;**) except a few e.g., the above mentioned Pre-processor Directives.

2.6 WRITING, COMPILING AND EXECUTING A C++ PROGRAM

From writing a C++ program to its execution, various softwares are required. These are namely: a **Text Editor** for editing; a **Compiler** fortranslating C++ program to machine language instructions ; a **Debugger**to identify coding errors at various programming stages; a **Linker**to link object modules of a program into a single object file.



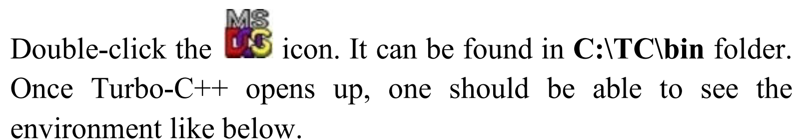
Thus, each of the softwares is to be installed in your machine separately for this purpose. There are software packages available bundled with all the above-mentioned system softwares with additional functionalities. Thus, these softwares enable a user/programmer to write, compile, debug and execute (run) a C++ program. This kind of packages is generally termed as **Integrated Development Environment** (IDE). Not only for C++,**IDEs**are also

available for other programming languages. **Turbo-C++ is an IDE for C language**.

For a C+ program to execute, the file (saved with the extension **.cpp** or **.CPP**) containing statements written in C++ has to be translated from a high-level language to a low-level language (executable code). This task is accomplished with the help of compiler and linker. A compiler takes in a source code and produces an object code which is further passed to a linker. A linker links the source program with the external entities such as the header files and other user-defined files, if any, to produce the final executable code.

***For Windows users*, the procedure for writing and executing a C++ program in Turbo-C++ is shown below.**

1. Open the **Turbo-C++ IDE**.

   Double-click the  icon. It can be found in **C:\TC\bin** folder. Once Turbo-C++ opens up, one should be able to see the environment like below.

2. Select '**New**' from the '**File**' menu.

   Write the program and save it in a location of your choice. The default location where a program is saved is '**C:\TC\bin**'. To save a program, press **F2** or select '**Save**' from the '**File**' menu. The extension of the file must be **.cpp** or **.CPP** for a C++ program.

3. To compile select '**Compile**' from the '**Compile**' menu. (or press **Alt+F9**)

   If there are some errors in the program, the error messages will be displayed in the 'Message' box. In Example 1 a semi-colon is missing at the end of the statement. Such compile-time errors need to be corrected for a successful compilation.

4. After a program is compiled, it can be executed. To do so, select 'run' from the 'run' menu.

   The output of the program will be displayed on the screen. For our example, '**Hello World**' will be displayed.

2.7 ERRORS IN C++

There are various types of errors that may occur during compilation and execution of a C++ program. These are:

- **Syntax Error**

  Such error occurs when a statement does not comply by the rules of the language. Those errors are detected by the compiler and need to be corrected before the code is executed. So, this kind of errors is also known as a compile-time error. Some common compile-time errors are statement missing semi-colon(s), mis-spelt keywords, undefined identifier etc. The example below has a statement missing semi-colon.

  Example-1:

```
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();
    int a,b     //semi-colon(;) missing
    a=2;
    b=3         //semi-colon(;) missing
    cout<<"Sum="<<a+b;
    getch();
}
```

  Compilation Output: Declaration syntax error, statement missing semi-colon

- **Semantic Error**

  These errors violate the meaning or the logic of the language and hence fail to produce the desired result. These errors are also detected by the compiler most of the time. Consider the following statements of C++ program.

```
int a=2, b=4, c;
a + b = c;
```

  Compilation Output: Lvalue required

- **Run-time Error**

  Even after successful compilation, there are times when a program fails to execute properly. Some situations which may generate run-time errors are:

- when a number is divided by zero,
- while trying to open a file that does not exist,
- accessing an array beyond its boundary etc.

Consider the following statements,

```
b = x-y;
c = a / b;
```

The symbols '–' and '/' means subtraction and division respectively. Now, the above statements are correct. But if it happens that for the first statement the value of **b**is **0**. Then in the second statement **a** will be divided by **0**, **a/0** which is an invalid operation and will lead to termination of the program execution.

2.8 C++ CHARACTER SET

<table>
<tr><th colspan="2">C++ Character Set</th></tr>
<tr><td>Letters:</td><td>Digits:</td></tr>
<tr><td>Lower Case: a…..z
Upper Case: A….Z</td><td>0,1,….,9</td></tr>
<tr><td colspan="2">Symbols:</td></tr>
<tr><td>, Comma</td><td>} Right brace</td></tr>
<tr><td>. Period</td><td>[Left bracket</td></tr>
<tr><td>; Semicolon</td><td>] Right bracket</td></tr>
<tr><td>: Colon</td><td><Opening angle bracket/</td></tr>
<tr><td>? Question mark</td><td>or less than sign</td></tr>
<tr><td>' Apostrophe</td><td>>Closing angle bracket/or</td></tr>
<tr><td>" Quotation mark</td><td>greater than sign</td></tr>
<tr><td>! Exclamation</td><td>/ Slash</td></tr>
<tr><td># Hash</td><td>\ Backslash</td></tr>
<tr><td>$ Dollar sign</td><td>| Vertical bar</td></tr>
<tr><td>^ Caret</td><td>= Equal sign</td></tr>
<tr><td>& Ampersand</td><td>- Minus sign</td></tr>
<tr><td>* Asterisk</td><td>_ Underscore</td></tr>
<tr><td>(Left parenthesis</td><td>+ Plus sign</td></tr>
<tr><td>) Right parenthesis</td><td>~ Tilde</td></tr>
<tr><td>{ Left brace</td><td></td></tr>
<tr><td colspan="2">White Spaces:</td></tr>
<tr><td>Blanks</td><td>New Line</td></tr>
<tr><td>Horizontal Tab</td><td>Carriage Return</td></tr>
<tr><td>Vertical Tab</td><td>Form Feed</td></tr>
</table>

Table-2.1: C++ Character Set

A character denotes an alphabet, digit or special symbol. Like natural languages, computer language will also have well defined character set, which is useful to build the programs.

The C++ Character Set includes alphabets: a to z (lower case), A to Z (upper case), digits: 0 to 9, special symbols: !, @, #, $, % and many more. It also includes white spaces such as blank, tab, new line, form feed etc. ***Table-2.1*** list the C++ Character Set.

The C++ language follows the ASCII (American Standard Code for Information Interchange) for representing characters where each character has a unique 7-bit binary value representation. The characters are coded from 0000000 to 1111111, forming a total of 128 characters.

There are few ASCII characters which are unprintable, i.e., they will not be displayed on the screen. These are used only to perform some specific functions aside from displaying text. Examples are backspace, newline, alarm.

2.9 C++ TOKENS

In a C++ program, Tokens are the building blocks. A Token is the smallest individual element or unit in a Program. Tokens are composed of the characters, symbols etc. Programs are coded using these tokens according to the rules of the language. In C++language, tokens can be classified under five categories and they are:

- Keywords
- Identifiers
- Constants
- Operators
- Special Symbols/Characters

Let us consider the following C++ program.

Program-1:

```
void main()
{
        int  x;
        cout<<"Enter a number:";
        cin>>x;
        x = x + 1.2 * x;
        cout<<"Incremented value:"<<x;
}
```

Tokens used in the above **Program-1** under different categories are presented in the **Table: 2.2**.

Type of Tokens	Tokens Used
Keywords	int, void
Identifier	x, main, printf
Constant	Enter a number:, 1.2, Incremented value:
Operators	Addressof (&), Addition (+), Multiplication (*)
Special Symbols	(,), {, }, ;, ", %, &, :

Table: 2.2

2.9.1 Keywords

Keywords are the reserved words that have well-defined purposes. A **Keyword** should be used carefully and not for any other purposes like naming a variable or function. Keywords, when used in programs, should not be modified or altered from their defined format.

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

Table: 2.3

They are specific to programming languages, that is, every programming language has their own set of keywords. **Table-2.3** gives the complete set of keywords for the C++ language.

2.9.2 Identifiers

Identifiers are the name given to the entities such as variables, constants, functions, files, structures, classes etc. Just as persons, cities or streets have names, the C++ entities such as variables, functions, files etc. are given unique names (identifiers) for their identification in a C++ program.

Rules for Naming Identifiers:

Identifiers are basically composed of alphanumeric characters i.e. alphabets or digits. The basic rules for naming an **Identifier** are:

- A valid identifier can have letters, digits and underscores.
- The first character should be an alphabet or an underscore.
- No special symbols except the underscore, is allowed in an Identifier.
- The Identifiers could be of any length but only the first 31 characters are significant.
- Keywords cannot be used as identifiers.

Following are some examples of identifiers-

Identifier	Valid?	Remark
Sum	valid	
char	invalid	keywords are not allowed
price#	invalid	special symbols not allowed
var 1	invalid	blank space not allowed
avg_num	valid	

Although any combination of letters, numbers and underscore is an identifier, it is advisable to create an identifier that reflects the meaning and purpose of the entity.

2.9.3 Constants, Operators and Special Characters

Constants can be defined as fixed values that do not alter during the execution of a program. Following are the different types of **Constants**:

- Integer Constants
- Real Constants
- Character Constants
- String Constants

- Special Character Constants
- Symbolic Constants

Operators are one of the important building blocks in C++ language. Operators are used to perform specific mathematical and logical computations/comparisons on operands. Few examples of operators are: +, -, /, * etc. In later units Operators will be discussed in detail.

Special Symbols are the symbols other than the operators. These are used in programs for various purposes as and when necessary. Refer to ***Table-2.2*** for special symbols used in **Program-1**.

Now, let's discuss different types of **Constants** one-by-one.

- **Integer Constants:**

Integer Constant is a whole number (without ***decimal point***). It can be defined as a ***sequence of digits*** (from **0 to 9**). An **Integer constant** can be ***preceded*** by **–** (for **-**negative value) or optional **+** (for positive value). Following are some examples of **Integer constants**,

Integer Constant	Valid?	Remark
1	valid	
300	valid	
-20	valid	
+15	valid	
20.3	invalid	Not a whole number (without decimal point)
1,200	invalid	Comma not permitted between digits

- **Real Constants:**

Real Constant is a number containing ***fractional part*** (with ***decimal point***). It is also called **Floating Point Constant**. Like **Integer Constant** it also can be ***preceded*** by **–** or optional **+**. For example, following are some valid Real Constants.

20.5 -12.40 +2.67 -.50

A **Real Constant** can also be expressed in exponential form. The form is:

mantissa *e* exponent

Mantissa can either be an integer or a real number. The **Exponent** is an integer with – sign or + sign(optional). ‘**e**’ separates the **mantissa** and **exponent**. Instead of ‘**e**’ we can write ‘**E**’ also. Following examples illustrates this representation.

In Real Form	In Exponential Form
1286.45	12.8645e2 or 12.8645E2
0.034	3.4e-2 or 3.4E-2
1,200	Comma not permitted between digits

- **Character Constants:**

A **Character Constant** is a ***single alphabet/digit/symbol/blank space*** enclosed in ***single quotes*** (**‘’**). For example, **‘a’**, **‘A’**, **‘5’**, **‘9’** are valid **Character Constants**. But do not confuse **‘5’** and **5** as first one is a **Character Constant** and other one is an **Integer Constant**.

- **String Constants:**

A **String Constant** consists of a sequence of characters (*alphabets/digits/symbols/blank spaces*) enclosed in ***double quotes*** (**“ ”**). Following are few examples of valid **String Constants**.

“A” “IDOL” “Hello! Welcome to IDOL”   “1+2+3+4”

- **Special Character Constants:**

There are some **Special Character Constants** which are basically used in functions that display data. These character constants combine ‘\’ with an alphabet/symbol. These character constants are also termed as **Backslash Character Constants**. Following **Table-2.4** lists the different Special Character Constants available in C++ with their meaning.

Special Character Constants	Meaning
‘**\n**’	adding new line
‘**\a**’	adding an alert(bell)
‘**\b**’	applying backspace
‘**\f**’	adding form feed
‘**\r**’	adding carriage return
‘**\t**’	applying horizontal tab
‘**\v**’	applying vertical tab
‘**\?**’	adding question mark in the output
‘\\’	adding back slash in the output

'\0'	**null** value
'\''	adding single quote in the output
'\"'	adding double quote in the output

Table-2.4

Program-2(in ***Turbo-C++***)**:** Using few of the above special character constants

```
#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        cout<<"Courses offered by GUIDOL\a";
        cout<<"\nMA in Assamese";
        cout<<"\nMA in English";
        cout<<"\tMA in Economics";
        cout<<"\nMA in History";
        cout<<"\\ MA in Political Science";
        cout<<"\tand many other courses\?\?\?";
        getch();
}
```

Now, before discussing the ***Program-2***, we need to know some basic statements in C++. In case of Turbo-C++ (in ***Windows***) the statements,

```
        clrscr();
        getch();
```

are necessary. But in case of the ***Unix/Linux***, the above statements are not necessary at places where they were put in the above program.

The function **clrscr()** is used to clear the screen. It is contained in the header file '**conio.h**'. It is basically used in programs using Turbo-C++ IDE(in Windows) as because while running the programs the output screenmay contain the output from earlier executed programs. So, here in the ***Program-2*** this function is used before the first output statement (i.e. **cout**).But in Unix/Linux, the C++ library does not contain the header file '**conio.h**' and so **clrscr()** function cannot be used.

Like **clrscr()** function,the **getch()** function also contained in '**conio.h**'. So, in Unix/Linux this function does not work. This function takes a character from the keyboard. In the ***Program-2*** this function is the mentioned as the last statement but logically it is not

required.Calling the **getch()** function as the last statement keeps the program waiting for a character input(i.e. a key to be pressed) to complete execution. This, in turn, lets the users view the previous outputs displayed on the screen.

The other statements except the last one will display the messages put within **" "** (double quotes) on to the screen. The **cout** is used to display data on to the screen. In later units, the concept of **cout** will be discussed in detail. The << is known as insertion operator and it is used along-with **cout**for displaying data on to the screen.

Now let's discuss the output of the above program.

Output:

```
Courses offered by GUIDOL
MA in Assamese
MA in English
        MA in Economics
MA in History\ MA in Political Science
        and many other courses???
```

Explanation:

- ✓ First **cout** displays the message (within **" "**).
- ✓ Second and third **cout** displays the messages (within **" "**) in new lines because of the special character constant **'\n'**.
- ✓ Fourth **cout** displays the message in new line but after a tab space because of the special characters **'\n'** and **'\t'** respectively.
- ✓ Fifth **cout** displays the message in new line because of the special character **'\n'**.
- ✓ Sixth **cout** displays the message in the same line just after the last message in the same line printed with the symbol '\' and a single space before the message because of the special character '\\' followed by a space. This message is not displayed in a new line for not using the **'\n'** special character.
- ✓ The last **cout** displays the message in new line but after a tab space and three **'?'** marks are at the end of the message because of **'\t'** and three **'\?'** special character constants.

- **<u>Symbolic Constants:</u>**

A **Symbolic Constant** can be defined as the combination of a ***name*** (except keywords) and a **constant value**.

The syntax of defining **Symbolic Constant** is

#define symbolic-constant-name constant-value

and it should be defined before "**main()**". In the process of compiling a program, first there is a pre-processing step in which, apart from other tasks, symbolic constants are processed, i.e. wherever in the program the **symbolic-constant-name** appears it is replaced by the **constant value**. Consider the following C program.

Program-3(in ***Turbo-C***++)**:** Demonstrates the use of Symbolic Constant.

```
#include<iostream.h>
#include<conio.h>
#define  PI  3.14
void main()
{
        int rad;
        float area;
        clrscr();
        cout<<"Enter the value of Radius:";
        cout<<rad;
        area = PI *rad*rad;
        cout<<"The area of the circle"<<area;
        getch();
}
```

In the above program (Program 3)**PI** is the **symbolic constant** with the value **3.14**. So, the statement,

```
        area = PI*rad*rad;
```

during pre-processing, becomes,

```
        area = 3.14*rad*rad;
```

i.e. **PI** is replaced by **3.14**(as defined).

In the last **cout** statement, you have noticed the use of **<<** operator twice. In a **cout**statement, we can display more than one data by the repeating the **<<** before each and every data. Suppose in the above program, we need to display the value of **PI** and **rad** in a single **cout**, then the statement will be:

```
        cout<<PI<<rad;
```

2.10 DATA TYPES

In a Program, we can store/assign data and use the stored data along with other functions. C++ offers a wide variety of "**Data Types**". C+ also allows creation of new **Data Types** and its customization as per need of a program. Data Types in C++ can be broadly classified in three classes (Fig-2.1), namely:

- Primary/Built-in Data Type,
- Derived Data Type and
- User-Defined Data Type.

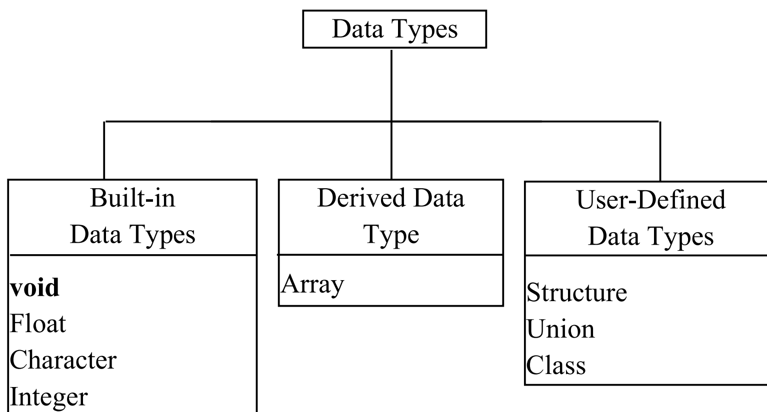


Fig-2.1: Data Types in C++

2.10.1 Primary/Built-in Data Type

The **Built-in Data Types** available in C++ are listed in ***Fig-2.2***.C++ supports five **fundamental typeschar**, **int**, **float**, **double** and **void**. The others are the extensions of the **fundamental types**.

Each of the **fundamental type** has its ***size*** (in **bytes**) and thus depending on the ***size*** it has ***value range***. The ***size*** in **bytes** (**1 byte = 8 bits**) and range of each of the **fundamental types** are listed in the following ***Table-2.5***.

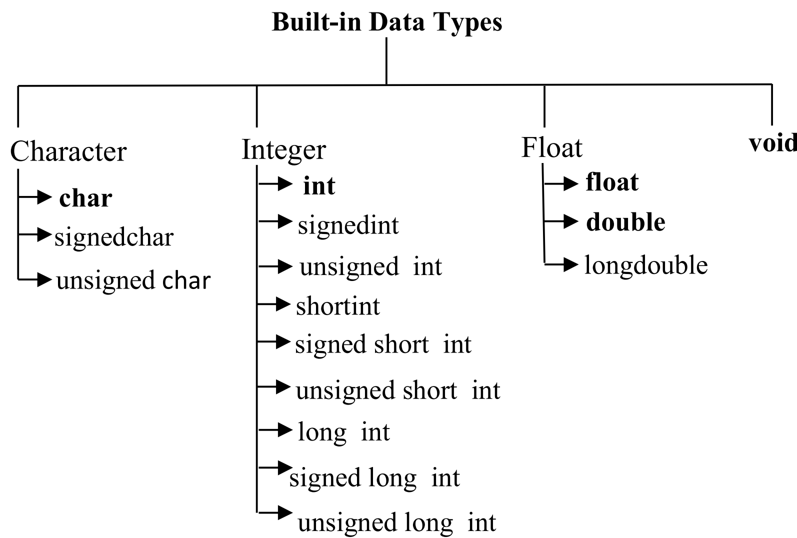


Fig-2.2: Built-in Data Types in C++

Data Type	Size (in bytes)	Value Range
char	1	-128 to 127
int	2	-32,768 to 32,767
float	4	3.4e-38 to 3.4e+38
double	8	1.7e-308 to 1.7e+308

Table-2.5

For a **character** data, the **fundamental type** is **char** and it is of ***size*1byte**. It is to be noted that every ***alphabet/number/symbol*** is associated with an **ASCII** value (a whole number). For example, **ASCII** value for '**A**' is **65**, '**a**' is **97**, '**=**' is **104**, '+' is **43** etc.

The fundamental type for an **Integer** data is **int**. The size of **int** type depends on the **word size** for a particular machine. A**word** is defined as the maximum number of bits that a CPU can process at a time. A **word size** can be as high as **64 bits(8 bytes)**. **For our discussion, let's consider that the word size is of 16 bits(2 bytes).**

The terms, **short** and **long** (from ***Fig-2.2***), are called **Modifiers**, i.e. they are used to ***modify/extend*** the **size** of the **fundamental types** and thus the ***value range*** also increases.

- In case of **short**, the ***size*** is same as the ***size*** of the associated **fundamental type**. For example, ***sizes*** of **int** data-type and **short int** data-type are the same, i.e. **2 bytes**.
- But the **long** modifier generally ***doubles*** (exception in few cases) the ***size*** of the associated **fundamental type**. For example, the ***size*** of the **long int** data-type is **4 bytes**, i.e. the ***double*** the ***size*** of **int/short int** type (**2 bytes**).

The terms, **signed** and **unsigned** (from ***Fig-2.2***), are called **Qualifiers**, which have no effect of the ***size*** of the type but have effect on the ***value range***.

- In case of **signed** data, the ***left-most bit*** is reserved for the sign (positive ornegative), and so this will allow data to be either negative or positive.
- In case of **unsigned** data, no bit is reserved for the **sign** and thus all the bits are used for the data.
- If **signed/unsigned**is not mentioned then ***by default*** that fundamental type will be treated as **signed** one.

And therefore, in **signed** data, the **values ranges** from a **–ve** to a **+ve** value. But in **unsigned** data, the ***values ranges*** from **0** to a **+ve** value. For example, in case of **signed char** data-type the ***size*** is **1 byte** and ***value range*** is **-128 to 127**. But in case of **unsigned char** data-type the ***size*** is also **1 byte** but the ***value range*** is **0 to 255** as for the ***maximum value*** (**255**) all the 8 bits will be **1**s [**11111111**].

STOP TO CONSIDER

The maximum value in 8 bits is **255**

$(1*2^7)+ (1*2^6)+(1*2^5)+ (1*2^4)+ (1*2^3)+ (1*2^2)+ (1*2^1)+ (1*2^0)$

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1

The following ***Table-2.6*** illustrates these more clearly.

Type	Size	Value Range

	(in bytes)	
char/signed char	1	-128 to 127
unsigned char	1	0 to 255
int/signed int/short int/signed short int	2	-32,768 to 32,767
unsigned int/unsigned short int	2	0 to 65535
long int/signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
float	4	3.4e-38 to 3.4e+38
double	8	1.7e-308 to 1.7e+308
long double	10	3.4e-4932 to 1.1e+4932

Table-2.6

2.10.2 Derived Data Type

Array is the **Derived Data Type** supported in C++. Basically an **array** is a list of continuous memory locations (in primary memory) of same type. **Array** not only supports ***fundamental types*** or ***its variations*** but also **User Defined Types**.

2.10.3 User Defined Data Type

The term, "**User Defined Data Type**", is self explanatory. This kind of data type is created by the **User** (the programmer) according to his/her need. These are three user defined data types and they are: **Class,Structure**, **Union** and **Enum**.

Class, Structure and **Union** will be discussed in later units.

Lets' discuss about **Enum**. We know that the set from where an integer, real or character value can be considered. For example, an integer(**int**) may be any value ranging from **-32,768 to 32,767** (from ***Table-2.6***). But there may be situations where we need to restrict the range/pool of values according to the need for a specific purpose. So, **Enum**(enumeration) is a **User Defined Data Type** that can take one value from the values those are predefined. **enum** keyword is used to define the enumerated data type. The syntax for defining this type is:

enum enum-name {value-1, value-2, …………, value-n};

where, **enum-name** is the name of type and **value-1**, **value-2**,…. is the list of values.

2.10.4 typedef

The **typedef** keyword is used to temporarily (in most of the cases) assign an ***alias*** i.e. ***alternative name*** to a **fundamental/derived/user defined** data type.

The syntax for using typedef is:

typedef existing-type-name alternative-name;

For example, in a program we have to work with a data-type **unsigned long int**. This data type name is a long one. Now, we can assign, say **ulint**, as a new name which is much shorter than name of above the type using the **typedef**. So, we can do this by using the following statement:

```
typedef   unsigned long int  ulint;
```

Now, in the program, when we have to declare a variable of type **unsigned long int** we can use the new name **ulint** instead. Suppose we want to declare a variable called **SUM** of the above type then we can type,

ulint SUM;

STOP TO CONSIDER

typedef is used with user defined data types, when names of the data types become slightly complicated and too long to use in programs.

2.11 VARIABLES& STORAGE CLASSES

2.11.1 Variables

A **variable** is a **named memory location** (in main memory) where one can store different values (of a particular ***type***) at different times. In **Program-1**, the **identifier** '**x**' is a **variable** which can store an ***integer value***. Now, you may think of how you we can say that variable '**x**' is an **integer variable**!!!

```
int   x;
```

The above statement in the ***Program-1*** ensures that '**x**' is an integer variable as **int** is mentioned before '**x**'.

Like registering (declaring) a name to a newly created company before it starts operating, a**variable** should also be **declared** (i.e. **named**) before its use.

The syntax for declaring a variable is:

data-type   variable-name;

Here, data-type refers to the type values the variable can store at a time. This kind of statement is known as **Variable Declaration Statement**.

The ***rules for naming*** a **variable** is the same the rules for naming an **identifier** as already discussed in ***Section 2.9.2***.

There are different **ways for declaration** of **variable(s)** in a C+ program and these are:

- ✓ A **variable** should be declared as in the syntax mentioned above.
- ✓ For declaring more than one variable of same type in ***one statement***, the syntax is:

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

where a, b, c and sum are integer variables. We may also declare the above four variables ***individually*** like,

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

- ✓ For variables of different types, different statements are required for each of the types. Suppose you want to declare a, b as integer variables and x, y as floating point variables. Then we have to declare them as:

```
int  a, b;
float  x, y;
```

2.11.2 Storage Classes

As mentioned earlier a variable is a storage areaof primary memory where we can store/assign a value. Apart from primary memory, the CPU registers are also a kind of memory locations for the variables declared in a C++ program. Here comes the concept of **Storage Class**.

Storage Class is basically related with declaration of variables. It basically specifies the part of storage space (memory/registers) to allocate memory for variables declared in C++ program. It also specifies the scope of a variable i.e. the lifetime of a variable during

execution. **Lifetime** means that whether the declared variable will exist during the execution of the program or will exist only within the block (generally related with Function) in which the variable is declared. These two kinds of lifetime are termed as **Global** and **Local**. The Storage class also determines the variable visibility level i.e. a variable may have global lifetime but only visible from within the block in which it is defined.

Four storage classes are provided in C++and they are listed in the ***Table-2.7*** with their meaning.

Storage Class	Meaning
auto	Local variable with Local lifetime, i.e. only to Function(block) in which it is declared. It is the ***default specifier***.
static	Local variable with Global lifetime.
extern	Global variable with Global lifetime, i.e., accessible from everywhere in a C++ program.
register	Local variable whose storage space is the CPU register.

Table-2.7

Thus, the syntaxfor a variable declaration that uses a storage class is:

Storage_Class_Specifier Data_Type Variable_Name;

Following are the few variable declaration statements that use the storage class specifiers.

```
auto   int  a;     ←Automatic Storage Class
static  int b;     ←Static Storage Class
extern  char c;    ←External Storage Class
register  int  d;  ←Register Storage Class
```

2.12 OUTPUT & INPUT IN C

In a C++ program to display output on to the screen and as well as to take input during the execution. There are functions which are used for output and input in C++.

Output in C++:

As discussed earlier, **cout** along with the << operator is used to display message, e.g., the statement

```
cout<<"Welcome to IDOL";
```

will display the message, Welcome to IDOL, on to the screen. The syntax of the **cout** is(in a simplified form):

cout<<Variable-1<<Variable-2<< …;

Input in C++:

In ***Program-2***, the statement

```
cin>>rad;
```

is an input statement. Here **cin** is used to take input during the execution of the program. The >> operator is known as extraction operator which is to be used along-with **cin**. Syntax of **cin** is,

cin>>Variable-1>>Variable-2>>…;

Consider the following C program.

Program-3: Program which takes a number as input and displays it.

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

void main()
{
        int  a;
        clrscr();
        cout<<"Enter an Integer Value: ";
        cin>>a;
        cout<<"The Value = "<<a;
        getch();
}
```

Output:

```
Enter an Integer Value: 50
The Value = 50
```

Explanation:

- ✓ When the first **cout** executes, it will display the message "Enter an Integer Value:"

- ✓ The execution of **cin** will display the **cursorblinking** at the end of the above message. The blinking **cursor** means the requirement of a value to be typed-in
- ✓ The last **cout** will now display the message with the value of **a**, i.e. **50**, as in the mentioned in the output.

Program-4: Program which takes two numbers as input and display them.

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

void main()
{
        int  a, b;
        clrscr();
        cout<<"Enter two numbers: ";
        cin>>a>>b;
        cout<<"The Values are "<<a<<"and"<<b;
        getch();
}
```

Output:

```
Enter two numbers: 5 100
The Value are 5 and 100
```

Or, the program can also be written as**:**

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

void main()
{
        int  a, b;
        clrscr();
        cout<<"Enter a number: ";
        cin>>a;
        cout<<"Enter another number: ";
        cin>>b;
        cout<<"The Values are "<<a<<"and "<<b;
        getch();
}
```

Output:

```
Enter a number: 5
```

Enter another number: 100
The Values are 5 and 100

2.13 OPERATORS

Operators in C++ are classified into the following categories:

1. Assignment Operator
2. Arithmetic Operators
3. Relational Operators
4. Logical Operators
5. Increment and Decrement Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators

Before discussing about the above categories of **operators**, let's first discuss about **Expressions** and the **Assignment Operator** '='.

2.13.1 Assignment Operator

Assignment operator is represented using the symbol '='. This operator is used to store value in a variable, i.e. Assignment operator is also used for

- assigning value of a variable to another variable,
- assigning the result of an expression to a variable (will be discussed in 2.5.2) and
- assigning the return value form a function call to a variable, which will be discussed in later units.

Few examples of assignment operator are mentioned the ***Table-2.8*** below:

Examples	Meaning
A=100	100 is assigned to the variable A (i.e., A now contains 100)
B=A	value of A is assigned to the variable B (i.e., B now also contains 100)
C=A+B	values of A and B are added and the total is assigned to the variable C
VAL=sum(10, 20)	the result of function 'sum()' is assigned to variable VAL (will be discussed in Unit: 4: Functions)

Table-2.8

2.13.2 Arithmetic Operators (and Expression)

In the following ***Table-2.9***, the arithmetic operators are listed.

Operators	Meaning
+	Addition
-	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulo (Remainder after division)

Table-2.9: Arithmetic Operators

The uses of arithmetic operators are illustrated in the ***Table-2.10***. Suppose, A and B two integer variables and they contain the values 50 and 10 respectively.

Operator	Example	Meaning	Result
+	A + B	Add A with B	60
-	A - B	Subtract B from A	40
*	A * B	Multiply A with B	500
/	A / B	Divide A by B	5
%	A % B	Remainder from A divide by B	0
- (unary)	- A	Multiply A with -1, i.e., will changes A's sign.	-50

Table: 2.10

Suppose we have to use a mathematical formula "**a+b+2ab**" in a C++ program. Now, consider the value of **a** is **2** and **b** is **3**. So, in a program, for the above tasks we can write statements:

```
int  a, b, res;
a=2;
b=3;
res = a+b+2*a*b;
```

Except the first one, all the other three statements are Assignment Statements. But in the last statement, the left-hand side of **=** is a variable and right-hand side is the mathematical operation. \The mathematical formula '**a+b+2*a*b**' is an expression.

2.13.3 Relational Operators

Relational Operators are used for comparison of two values (also stored in variables). ***Table-2.11*** presents the relational operators in C++ and their meanings.

Operators	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
= =	equal to
!=	not equal to

Table-2.11: Relational Operators

Relational operators are used in ***decision making/control statements*** such as ***if-else-elseif***, ***do-while***, ***while***, ***for*** etc. You will get acquainted with these kinds of statements in the following units.

Operator	Example	Meaning	Result
<	A < B	is A less than B	FALSE
>	A > B	is A greater than B	TRUE
<=	A <= B	is A less than or equal to B	FALSE
>=	A >= B	is A greater than or equal to B	TRUE
= =	A = = B	is A is equal to B	FALSE
!=	A != B	is A is not equal to B	TRUE

Table-2.12

The uses of relational operators are illustrated in the ***Table-2.12***. Suppose A and B are two integer variables having values 50 and 10 respectively. As you can see that the result of each of the expressions is either TRUE or FALSE, which means that if an expression satisfies the condition then it is TRUE otherwise FALSE.

2.13.4 Logical Operators

Logical operators operate only on Boolean values and yield a Boolean result of their own.

Operators	Meaning
&&	logical AND
||	logical OR
!	logical NOT

Table-2.13: Logical Operators

The ***Table-2.13*** presents the Logical Operators defined in C++. The uses of logical operators are illustrated in the ***Table-2.14*** where suppose A contains 28 and B contains 50.

<table>
<tr><th>Operator</th><th>Example</th><th>Meaning</th><th>Result</th></tr>
<tr><td>&&</td><td>A>30 && B>30</td><td>Is A and B greater than 30</td><td>FALSE</td></tr>
<tr><td>||</td><td>A>30 || B>30</td><td>is A greater than 30 OR B greater than 30/both A and B greater than 30</td><td>TRUE</td></tr>
</table>

Table-2.14

The use of '!' logical operator will be discussed in the following unit while working with different programs.

2.13.5 Increment and Decrement Operators

Like Arithmetic Operators, C++ offers **Increment** and **Decrement Operators**. These operators are illustrated in the following ***Table-2.15***.

Operators	Meaning
++	adds 1 to the value of the associated operand and update it
--	subtracts 1 from the value of the associated operand and update it

Table-2.15

The uses of these operators are illustrated in the ***Table-2.16*** where suppose both x and y contains 10.

Operator	Example	Meaning	Result
++	x++	adds 1 to the value of x and update x	value of **x** becomes **11**
	++x		
--	y--	subtracts 1 from the value of y and update y	value of **y** becomes **9**
	--y		

Table-2.16

Basically, the expression 'x++' is same as the expression 'x=x+1'. From the ***Table-2.16*** it is clear that if x contains the value 10, then the expression 'x++' will increment the value of x by 1 i.e., value of x will now be 11. Same as in case of the expression 'y--' but here the value of y will be decremented by 1.

2.13.5.1 Postfix Increment & Decrement Operations

Consider the following statement which contains a **Postfix Increment Expression**.

```
C = X++;
```

Here, the value of X will first be assigned to C and then X will be incremented. Assuming the value of X to be 10, the final output after the statement is executed will have C value as 10 and X value as 11.

Consider the following statement which contains a **Postfix Decrement Expression**.

```
C = X--;
```

Here, the value of X will first be assigned to C and then X will be decremented. Assuming the value of X to be 10, the final output after the statement is executed will have C value as 10 and X value as 9.

2.13.5.2 Prefix Increment & Decrement Operations

Consider the following statement which contains a **Prefix Increment Expression**.

```
C = ++X;
```

Here, the value of X will first be incremented and then it will be assigned to C. Assuming the value of X to be 10, the final output after the statement is executed will have C value as 11 and X value as 11.

Consider the following statement which contains a **Prefix Decrement Expression**.

```
C = --X;
```

Here, the value of X will first be decremented and then it will be assigned to C. Assuming the value of X to be 10, the final output after the statement is executed will have C value as 9 and X value as 9.

2.13.6 Conditional Operator

?: operator is known as **Conditional Operator** in C++. This operator is also called **Ternary Operator**.

The form of an expression which uses **?:** operator is mentioned below:

exp-1 ? exp2 : exp3

where, **exp-1**, **exp-2** and **exp-3** may be single variables, may be expressions or may be combinations of both. Let's understand the above form with the help of an example. Consider **a**, **b** and **res** are three integer variables. Now, suppose we want to find the maximum value between **a** and **b**. The following statement will do this task, which uses the **?:** operator.

```
res = (a>b) ? a : b;
```

Here, **(a>b)** is the **exp-1**, **a** is the **exp-2** and **b** is the **exp-3**. Now, you may be getting puzzled how this whole expression will be performed! Here is the answer given below:

- ✓ First, **(a>b)**, the **exp-1** is performed.
- ✓ Now, if **exp-1** satisfies, i.e. **a** is greater than **b**, then the value of **exp-2** is the result,
- ✓ But if **exp-1** does not satisfy, i.e. **a** is not greater than **b**, then the value of **exp-3** is the result.
- ✓ And the result of the expression will be assigned/stored to variable **res**.

And thus we will get the value of the maximum between **a** and **b** in the variable **res**.

Now, suppose variable **a** contains the value **10** and variable **b** contains the value **50**. Then, after the execution of the statement value of **b** will be stored in variable **res** as value of **b** is the maximum which is **50**.

Example-8: Program to demonstrate the use of conditional operator for finding maximum of two input numbers.

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

void main()
{
        int  a, b, max;
        clrscr();
        cout<<"Enter the value of a=";
```

```
        cin>>a;
        cout<<"Enter the value of b=";
        cin>>b;
        max=(a>b) ? a : b;
        cout<<"Maximum Value:"<<max;
        getch();
}
```

Output:

```
Enter the value of a=100
Enter the value of b=50
Maximum Value: 100
```

Explanation:

- ✓ Suppose, the two **cin**statements take input 100 for **a** and 50 for **b**.
- ✓ In the statement, **max=(a>b) ? a : b**, the condition (a>b) is evaluated and
    - o if condition satisfies, means **a** is the maximum, the value of **a** is assigned to **max**.
    - o if condition not satisfied, means **b** is the maximum, the value of **b** is assigned to **max**.
- ✓ The last**cout** will display the value of the variable **max**, which is **100** along with the set message.

2.13.7 Bitwise Operators

Following ***Table-2.17*** lists the **Bitwise Operators** in C++ language. These operators are used for bitwise manipulation of data.

Operators	Meaning
&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	shift bits left
>>	shift bits right

Table-2.17

These operators are my not be applied to float or double type of data.

2.13.8 Special Operators

Apart from the operators discussed above, there are other operators in C++ and they are:

sizeof Operator

Type-Cast Operator: **(type)**

Pointer Operators: **&** and *****

Member Selection Operators: **.** and →

The **sizeof** operator is used to get the number of bytes occupied by an operand/type. For example, **A** and **S** are integer variables. Consider the statement mentioned below:

S = sizeof(A);

Here the **sizeof** operator will return the value **2** as **A** is of **int** (integer) type and the size of **int** data-type is **2 bytes**. So, value in **S** is **2** after execution of the above statement.

Now, what is **Type-Cast Operator**, **(type)**? Basically this operator is used to convert the type of a data to another compatible type temporarily.

To understand this operator let's first consider the following statements.

int a=7;

float res;

res = a/2;

So, the variable **a** is assigned with **7**. After the execution of these statements, you may think that the value in **res** will be **3.5**. But the result will be **3**. Since, we are dividing **a** by **2** where **a** is an integer and therefore instead of getting **3.5** we will get **3** though the variable **res** is of type float.

But we are expecting that the last statement would assign **3.5** to the variable **res**. So, how to get **3.5** as a result of the above expression? The use of **Type-CastOperator** will give us our expected result, i.e. **3.5**. So, we have to replace the expression in the last statement with the expression '**(float) a / 2**' and so the last statement to be written as,

res = (float)a/2;

This means that the value of **a** (i.e. **7**) is temporarily converted to float type (i.e. **7.0**) without effecting **a** and then the converted value (i.e. **7.0**) is divided by **2**. Thus the result **3.5,** will be assigned to **res**.

The use of **Pointer Operator*** and **Member Selection Operators.** and → will be discussed in the following units.

The **Pointer Operator &** which is known as **Address-of Operator** is used to get the ***address of a location.***

Example-9: Program to demonstrate the use of the **sizeof()** and **(type)** type-cast-operator.Here in the program the three integer variables sub1, sub2 and sub3 are for storing the marks of three subjects.

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

void main()
{
        int  sub1, sub2, sub3;
        float average;
        int size;
        clrscr();
        cout<<"Enter marksforsubject-1, subject-2 and subject-3:";
        cin>>sub1>>sub2>>sub3;
        average=(float)(sub1+sub2+sub3)/3;
        cout<<"The average mark is: "<<average);
        size=sizeof(int);
        cout<<"\nThe size of int data-type is: "<<size<<" bytes";
        getch();
}
```

Output:

```
Enter marks for subject-1, subject-2 and subject-3:  50  65  45
The average mark is: 53.3333
The size of int data-type is: 2 bytes
```

Explanation:

- ✓ The **cin** statement take three inputs, suppose **50**, **65** and **45** for **sub1**, **sub2** and **sub3** respectively.
- ✓ In the statement next to **cin** the average of the three marks, taken as inputs(**50**, **65** and **45**), is calculated. The average may definitely in the form of a ***floatingpoint number***. But, all the three variables containing marks are of data type **int**. The expression "**(sub1+sub2+sub3)/3**" will give an integer but not

a ***floating point number***. So, to get a ***floating point number*** the result of the above expression is temporarily converted into float using the **(float)** operator. Now, the result of the expression "**(float)(sub1+sub2+sub3)**" becomes a **float** value and this is divided by **3** resulting a **float** value and it is assigned to the float variable **average**. Thus the variable average contains the value **53.3333**.

✓ In the statement,

```
size=sizeof(int);
```

the **sizeof()** operator will give the size of **int** data type in bytes and this value(i.e. **2**) is assigned to the variable **size**.

CHECK YOUR PROGRESS

1. IDE stands for ____________________.
2. ASCII stands for _____________________.
3. Division by zero (0) is a _________________ error
4. The size of the char data type is ______________.
5. Structure is an example of __________________ data type
6. **%** arithmetic operator is used to ________________.

State True or False:

7. sizeof() operator gives the size of a given type.
8. – (minus) operator can be used as binary and unary operators.
9. If **x=110**, then the output of the following statement is **111**.

   cout<<x++;

10. In postfix increment operation expression, the ++ operator is placed before the operand.

2.14 OPERATOR PRECEDENCE AND ASSOCIATIVITY

When an expression contains more than one operator then the concept of **Operator Precedence** applies. **Operator Precedence** can be defined as the rule for determination of which operation to be

performed first, which to be second and so on in case of an expression with more than one operator.

Consider the following statement, where a=2, b=5 and c=3.

```
x = a + b * c;
```

Now, you can think of how the expression part (right-hand side of =) will be evaluated. Here the evaluation may take place in two possible ways:

WAY-1:

- At first the values of **a** and **b** will be added and then
- the total value will be multiplied by the value in **c**.

i.e. **x = (2 + 5) * 3**
= 7 * 3
= 21

WAY-2:

- At first value of **b** is multiplied with the value of **c** and then
- the total value is added with the value in **a**.

i.e. **x = 2 + (5 * 3)**
= 2 + 15
= 17

We know that **WAY-2** is the actual way of evaluating this expression as according to mathematics, first multiplication(*) operation will take place and then the addition(+). Thus the value in **x** will be **17**. This is an example of application of **Operator Precedence**.

In case of Arithmetic Operators, there are two distinct levels of priority in C and they are:

High Priority Operators: ***** **/** **%** (same precedence)

Low Priority Operators: **+** **-** (same precedence)

STOP TO CONSIDER

While writing a mathematical expression that contains more than one operator with different precedence (or with same precedence), use brackets **()** to specify the evaluation more precisely. Consider the expression **a+b*c** and suppose you want the evaluation as **a+b** and then multiply with **c**, so to be precise write the expression as **(a+b)*c**.

Now, let’s try to understand what does **Associativity** mean?? **Associativity** also can be defined as the rule which needs to be applied for evaluation when an expression contains more than one operator with the same precedence. Associativity can be either **Left-to-Right** or **Right-to-Left**.

The Associativity of Arithmetic Operators with same precedence is **Left-to-Right**. We know that the operators, **+** and **–**, have the same precedence. Now, let’s see how the arithmetic expression in the following C statement will be evaluated.

```
X = 10 + 2 - 3
```

Here in the above example, the evaluation of the expression ‘**10 + 2 - 3**’ will start from **Left-to-Right**(because of **associativity**). So, the evaluation will be in the form mentioned below:

(10 + 2) – 3

i.e.

- first evaluation of ‘**10+2**’ will take place, then
- the value **3** will be subtracted from the result value of ‘**10+2**’.

So, after execution of the above statement, the variable X will contain the result of the evaluation which is **9**.

The following **Table-2.14** lists the **Precedence** and **Associativity** of the Operators present in C++.

Operator	Description	Associativity
( )	Parentheses	left-to-right
[ ]	Brackets (related to Array)	
.	Member Selection (using Object Name)	
->	Member Selection (using Pointer)	
++ --	Postfix Increment/Decrement	
++ --	Prefix Increment/Decrement	right-to-left
+ -	Unary Plus/Minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (convert value to temporary value of	
*	*type*)	
&	Dereference (related to Pointer)	

sizeof	Address of Operand For Size in Bytes	
* / %	Multiplication/Division/Modulus	left-to-right
+ -	Binary Addition/Subtraction	left-to-right
<< >>	Bitwise Left-Shift, Bitwise Right-Shift	left-to-right
< <= > >=	Relational is Less Than/Less Than or Equal To Relational is Greater Than/Greater Than or Equal To	left-to-right
== !=	Relational is Equal To/is Not Equal To	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise Exclusive OR	left-to-right
|	Bitwise OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary (Conditional)	right-to-left
=	Assignment	right-to-left
,	Comma (for separation of expressions)	left-to-right

Table-2.14: Operator Precedence and Associativity

2.15 SUMMING UP

In this Unit, the history of C++ language is briefly described. Different steps starting from writing a C++ program to its execution is described in this Unit. These steps are described here in a very well-organized manner using different screen-shots. The steps shown are for Windows operating system. The structure of C++ program is also described in detail.

This Unit describes the errors generally occur during compilation and execution of a C++ program with the help of examples. The errors described here are namely Syntax Errors, Semantic Errors and Run-time Errors.

Here, in this Unit the list of C++ character set is also given with different characters, numbers and symbols with their names.

A C++ program consists of tokens which can be categorically classified into namely: Keywords, Identifiers, Constants, Operators and Special Characters. All these different types of C++ tokens are very elaborately described in this Unit.

The concept of data type is described here in this Unit.There are three categories of data type and they are: Primary/Built-in (e.g., char, int), Derived (e.g., Arrary) and User Defined (e.g., Structure, Class). The size of each of the type is clearly discussed with different examples.

A variable as discussed in this Unit, is a storage space into which one can store data. It needs to be declared before its use. In declaration statement, the data type of the variable should be mentioned. As C++ language is case sensitive, the case in which a variable is declared should remainthe same during its use in a program.

Input and Output are basic requirements of a C++ program. The 'cout' function is used to display data/message on to the screen. The 'cin' function is used to take input from keyboard. The syntaxes of these two functions are described in this Unit.

There are different types of operators defined in C++. Few operators are necessary for arithmetic expressions while few are useful for conditional expressions and so on. The operators have precedence and associativity. This Unit gave a detailed description regarding the operators' precedence and associativity.

2.16 ANSWER TO CHECK YOUR PROGRESS

1. Integrated Development Environment
2. American Standard Code for Information Interchange
3. run time error
4. 1 byte
5. User defined
6. find remainder after division
7. True
8. True
9. False
10. False

2.17 POSSIBLE QUESTIONS

1. Who developed the C++ language?

2. What is a header file?

3. What do you understand by errors in C++?

4. What is keyword? Write down five keywords available in C++.

5. Write down the rules for naming identifiers in C++.

6. What do you understand by Data Type? Mention its different categories.

7. Mention the fundamental data types in C++ with their respective size.

8. What is typedef?

9. What is Variable? Write down the syntax for declaring a variable.

10. Mention the use of **cout** in C++.

2.18 REFERENCES AND SUGGESTED READINGS

1. Stroustrup, Bjarne.*The C++Programming Language*.
2. Kanetkar, Y. P.. *Let us C++*. BPB publications.

UNIT 3 : CONTROL STATEMENTS IN C++

Unit Structure

3.1 Introduction
3.2 Unit Objectives
3.3 Conditional Statement
    3.3.1 The if statement
    3.3.2 The if else statement
    3.3.3 Multiple if else statement
    3.3.4 Nested if else statement
    3.3.5The Switch statement
3.4 Loop Control Statement
    3.4.1 for loop
    3.4.2 while loop
    3.4.3 do while loop
3.5 Comparison of the loop statements
3.6 Nested loop
3.7 goto statement
3.8 break statement
3.9 continue statement
3.10 exit() function
3.11 Summing Up
3.12 Answers to Check Your Progress
3.13 Possible Questions
3.14 References and Suggested Readings

3.1 INTRODUCTION

In C++, all statements written in a program are executed from top to bottom one by one. In some cases, there may arise some situations where depending upon a logical condition, some actions have to be carried out. Control statements are used to execute/transfer the control from one part of the program to another depending on a condition. These statements are also called conditional statements.

Control statements are of the following two types:

- Conditional control
- Loop control

C++ has three major conditional control statements:

(a) **if**statement (b) **if-else** statement and (c) **switch** statement

On the other hand, in a program, there may arise some situations where a repetitive work has to be carried out until a specific condition is fulfilled. In that case, ***loop***control statements are used in a program. C++ has three loop control statements: (a) **for** (b) **while** and (c) **do while**
This unit introduces you the different conditional and loop control statements with some suitable examples.

3.2 UNIT OBJECTIVES

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

- use of different decision control statements
- work with different loop control statements in a program
- use goto, break and continue statement in a program

3.3 CONDITIONAL STATEMENT

Conditional statements are used to execute statement or group of statements based on some condition.

C++ supports following conditional statements.

(a) if statement (b) if else statement (c) if else if ladder
(b) nested if

3.3.1 The *if* statement

The *if* statement is a control statement that tests a particular condition. Whenever, the evaluated condition comes out to be true, then that action or the set of actions are carried out. Otherwise, the given sets of action(s) are ignored.

The *syntax* of ***if*** statement is:

```
if(condition) {
  //statement(s) will execute if the condition is true
```

```
}
```

Example 3.1 Write a program in C++ to display the message "You have entered a +ve number" if the user inputsa +ve number.

Solution:

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

void main()
{
  int number;
  clrscr();
  cout<<"Enter an integer:\t";
  cin>>number;
  if (number > 0)
          cout<<"You have entered a +ve number:";
  getch();
}
```

When the above code is compiled and executed, it produces the following result:

Output:
Enter a number: 10
You have entered a +ve number.

Example 3.2: Write a program in C++ to find the bigger of two numbers.

Solution:

```
#include <iostream.h>
#include <conio.h>
void main()
{
int a,b,big;
clrscr();
cout<<"Enter two numbers:\t";
cin>>a>>b;
big = a;
if (b>big)
          big = b;
cout<<"\n The bigger number is: "<<big;
getch();
```

```
}
```

Output:
Enter two numbers: 10 20
The bigger number is: 20

Example 3.3: Write a program in C++ to find the biggest of threenumbers.

Solution:

```
#include <iostream.h>
#include <conio.h>
void main()
{
int a,b,c,big;
clrscr();
cout<<"Enter three numbers:\t";
cin>>a>>b>>c;
big = a;
if(b>big)
        big = b;
if(c>big)
        big = c;
cout<<"\n The biggest number is: "<<big;
getch();
}
```

Output:
Enter three numbers: 10 25 9
The biggest numbers is: 25

> **STOP TO CONSIDER**
>
> If more than one statements has to be executed in an ***If*** statement, you should write those statements within { and }

3.3.2 The *if else* statement

if else statement is used to execute a statement block or a single statement depending on the value of a condition.

The syntax of ***if else*** statement is:

```
if(condition) {
  /* statement(s) will execute if the condition is true */
}
else {
/* statement(s) will execute if the condition is false */
}
```

If the condition evaluates to **true**, then the statement(s) inside the if block will be executed, otherwise, the statement(s) inside the else block will be executed.

Example 3.4 Write a program in C++ to find the bigger of two numbers.

Solution:

```
#include <iostream.h>
#include <conio.h>
void main()
{
int a,b;
clrscr();
cout<<"Enter two numbers:\t";
cin>>a>>b;
if(a>b)
        cout<<"The bigger number is: "<<a;
else
        cout<<"The bigger number is: "<<b;
getch();
}
```

Output:

Enter two numbers: 10 20
The bigger numbers is: 20

Explanation:

In this case, 10 will be assigned to the variable an and 20 will be assigned to the variable b.

Then the statement if(a>b) will be tested. Since it is false (as 10 <20), so the statement under the else part will be executed.

Example 3.5: Write a program in C++ to check whether a number entered by the user is even or odd.

Solution:

```
#include <iostream.h>
#include <conio.h>
void main()
{
int n;
clrscr();
cout<<"Enter a number:\t";
cin>>n;
if(n%2==0) /* Checking whether the remainder is 0 or not */
        cout<<"The entered number is even.";
else
        cout<<"The entered number is odd.";
getch();
}
```

Output 1:
Enter a number: 11
The entered numberis odd.

Output 2:

Enter a number: 14
The entered number is even.

Example 3.6: Write a program in C++ to check whether a character entered by the user is vowel or consonant.

Solution:

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

void main()
{
char c;
clrscr();
```

```
cout<<"Enter a character:\t";
cin>>c;
if(c=='a'||c=='A'||c=='e'||c=='E'||c=='i'||c=='I'||c=='o'||c=='O'||c=='u'||c=='U
')
        cout<<c<<" is a vowel.";
else
        cout<<c<<" is a consonant.";
getch();
}
```

Output 1:
Enter a character: e
e is a vowel.

Output 2:
Enter a character: x
x is a consonant.

Explanation: In this program, user is asked to enter a character which is stored in variable ***c***. Then, this character is checked, whether it is any one of these ten characters a, A, e, E, i, I, o, O and u, U using logical OR operator ||. If that character is any one of these ten characters, that alphabet is a vowel; if not,then that character is a consonant.

Example 3.7: Write a program to check whether a character is alphabet or not.

Solution:

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

void main()
{
char c;
clrscr();
cout<<"Enter a character: \n";
cin>>c;
if( (c>='a'&& c<='z') || (c>='A' && c<='Z'))
        cout<<c<<" is an alphabet.";
else
        cout<<c<<" is not an alphabet.";
getch();
}
```

Output 1:
Enter a character: g
g is an alphabet.

Output 2:
Enter a character: #
is not an alphabet.

Explanation: When a character is assigned to a variable, ASCII value of that character is stored instead of that character itself. For example: If 'g' is assigned to a variable, ASCII value of 'g' which is 103 is stored. If you see the ASCII table, the lowercase alphabets are from 97 to 122 and uppercase letters are from 65 to 90. If the ASCII value of number stored is between any of these two intervals then that character will be an alphabet. In this program, instead of number 97, 122, 65 and 90; we have used 'a', 'z', 'A' and 'Z' respectively which is basically the same thing.

3.3.3 Multiple *if else* statement

An ***if*** statement can be followed by number of ***else if else*** statements, which is very useful to test various conditions.

The *syntax* is as follows:

```
if(condition 1) {
  /* Statement(s) to be executedonly when condition 1 is true */
}
else if(condition 2) {
/* Statement(s) to be executed only when condition 2 is true */
}
else if(condition 3) {
/* Statement(s) to be executed only when condition 3 is true */
}
else {
  /* Statement(s) to be executedonly when none of the above conditionsare true */
}
```

Example 3.8: Write a program in C++ to check whether an character entered by the user is uppercase or lowercase.

Solution:

```
#include<stdio.h>
#include<conio.h>

void main ()
{
char a;
clrscr();
cout<<"Enter a character: \t";
cin>>a;
if (a > 64 && a <=91)
        cout<<"The character is an upper-case letter.";
else if (a > 96 && a <=123)
        cout<<"The character is alower-case letter.";
else
        cout<<"This is not an character.";
getch();
}
```

Output 1:
Enter a character: a
The character is a lower-case letter

Output 2:
Enter a character: B
The character is anupper-case letter

Output 3:
Enter a character: 10
This is not an character

Example 3.9: The marks obtained by a student in 5 different subjects are inputted through the keyboard. The student gets a division as per the following rules:

(i) Percentage above or equal to 85 - Distinction

(ii) Percentage above or equal to 75 - Star

(iii) Percentage above or equal to 60 - First division

(iv) Percentage between 50 and 59 - Second division

(v) Percentage between 30 and 49 - Third division

(vi) Percentage less than 30 - Fail

Write a program in C++ to calculate the division obtained by a student.

Solution:

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

void main()
{
int m1, m2, m3, m4, m5, per;
clrscr();
cout<<"Enter marks obtained by a student in five subjects:\t";
cin>>m1>>m2>>m3>>m4>>m5;
per = (m1+m2+m3+m4+m5)/5;
if(per>=85)
        cout<<"The result is: Distinction";
else if ((per>=75) && (per<85))
        cout<<"The result is: Star";
else if ((per>=60) && (per<75))
        cout<<"The result is: First division";
else if ((per>=50) && (per<60))
        cout<<" The result is: Second division";
else if ((per>=30) && (per<50))
        cout<<"The result is: Third division";
else
        cout<<"The result is: Fail";
getch();
}
```

Example 3.10: Write a program in C++ to display the day in a week depending upon the number entered by the user.

Solution:

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

void main( )
{
int day ;
clrscr();
cout<<"Enter a number between 1 and 7:\t";
cin>>day;
```

```
if(day==1)
        cout<<"The day is Monday.";
else if(day==2)
        cout<<"The day is Tuesday.";
else if(day==3)
        cout<<"The day is Wednesday.";
else if(day==4)
        cout<<"The day is Thursday.";
else if(day==5)
        cout<<"The day is Friday.";
else if( day==6)
        cout<<"The day is Saturday.";
else
        cout<<"The day is Sunday.";
getch();
}
```

3.3.4 Nested *if else* statement

Nested ***if else*** is when an ***if els***e statement appears within the body of another "if" or "else" statement.. There may be any number of ***if*** statements in the nested form.

The *syntax* of ***nestedif***statement is:

```
if ( condition 1)
        if ( condition 2)
        {
                <true block 1>
        }
        else
        {
                <false block 1>
        }
else
        if ( condition 3)
        {
                <true block 2>
        }
        else
        {
                <false block 2>
        }
```

Example 3.11: Write a program in C++ to find the biggest of any three numbers entered by the user using nested if else statement.

Solution:

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

void main()
{
int a, b, c, big ;
clrscr();
cout<<"Enter the first number:\t";
cin>>a;
cout<<"\nEnter the second number:\t";
cin>>b;
cout<<"\nEnter the third number:\t";
cin>>c;
if (a>b)
{
	if (a>c)
		big = a;
	else
		big = c;
}
else
{
	if (b > c)
		big = b;
	else
		big = c;
}
cout<<"The biggest number is: "<<big;
getch();
}
```

Example 3.12: Write a program in C++ to check whether a year is leap year or not.

Solution:

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

void main()
{
int year;
clrscr();
cout<<"Enter a year: \t";
```

```
cin>>year;
if(year%4==0)
     {
          if(year%100==0)  // Checking for a century year
         {
          if (year%400==0)
                    cout<<year<<" is a leap year.";
          else
                    cout<<year<<" is not a leap year.";
         }
else
cout<<year<<" is a leap year.");
}
else
cout<<year<<" is not a leap year.";
getch();
}
```

Output 1:

Enter year: 1900

1900 is not a leap year.

Output 2:

Enter year: 2012

2012 is a leap year.

Check your progress 1

1. Differentiate between ***if*** and ***if else*** statement.

----

2. What is nested if else statement?

----

3. What will be the output of the following C++ code?

```
void main()
{
        int a=20,b,c;
        if (a>35)
        {
          b = 10;
          c = 6;
        }
        else
        {
          b = 20;
          c = 5;
         }
cout<<"The value of b and c will be: <<b<<c;
}
```


3.3.5 The *switch* statement

Instead of using the *if else* statement, the *switch* statement can be used in its place. *switch* statement is used to execute a block of statements depending on the value of a variable or an expression.The syntax of the *switch* statement is as follows:

```
switch(<expression>) {
case<label 1>:
        statement(s);
        break;
case<label 2>:
        statement(s);
        break;
case<label 3>:
        statement(s);
        break;
   // you can have any number of case statements
default :
```

```
        statement(s);
        break;
}
```

Let us discuss about the above syntax of the switch statement.

- The control statement *switch* begins with the switch keyword followed by blocks containing different cases. Each case handles the statements corresponding to an option i.e. <label 1>, <label 2> etc. and ends with the *break* statement which transfers the control out of the *switch* structure to the original program.
- The compiler checks the values of the expression or variable. If this value matches with any one of the labels given in <case> value, then that statement block will be executed.
- The braces { } can be omitted when there is only one statement available in the statement block.
- Here the variable between the parentheses following the *switch* keyword is used to test the condition and is called the control variable.
- *break*is a statement which will transfer the control to the end of *switch* statement. (The details about the *break* statement will be discussed in the next section)
- The *default* block is optional i.e. this may be omitted while writing a program.

STOP TO CONSIDER

The control variable of the switch statement can only be of the type int, long or char. ***Switch***statement is compact and can be used to replace a nested ***if*** statement.

Example 3.13: Using switch statement, write a program in C++ to display the day of a week. When the user types 1, Monday should be displayed, for 2 Tuesday….etc.

Solution

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

void main()
{
	int choice;
	clrscr();
	cout<<"Enter your choice between 1 and 7:\t";
	cin>>choice;
	switch(choice)
	{
		case 1:
			cout<<"Monday";
			break;
		case 2:
			cout<<"Tuesday";
			break;
		case 3:
			cout<<"Wednesday";
			break;
		case 4:
			cout<<"Thursday";
			break;
		case 5:
			cout<<"Friday";
			break;
		case 6:
			cout<<"Saturday";
			break;
		case 7:
			cout<<"Sunday";
			break;
		default:
			cout<<"Invalid choice. Enter your choice
between 1 and 7";
			break;
	}
	getch();
}
```

Example 3.14: Write a program in C++ which will read two numbers from the user. Now perform the addition, subtraction, multiplication and division operations according to the user input.

Solution:

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

void main()
{
        int a,b,choice;
        clrscr();
        cout<<"Enter two numbers:\t";
        cin>>a>>b;
        cout<<"\nEnter your choice between 1 and 4:\t";
        cout<<"\n[1]. Addition\n";
        cout<<"[2]. Subtraction\n";
        cout<<"[3]. Multiplication\n";
        cout<<"[4]. Division\n";
        cin>>choice;

switch (choice)
{
case 1:
        cout<<"The addition of the two number is:"<<a+b;
        break;

case 2: cout<<"The Subtraction of the two number is:"<<a-b;
        break;
case 3:
        cout<<"The Multiplication of the two number is:"<<a*b;
        break;
case 4:
        cout<<"The Division of the two numbers is:"<<a/b;
        break;
default:
        cout<<"Invalid choice. Enter your choice between 1 and 4";
        break;
}
getch();
}
```

STOP TO CONSIDER

The ***default*** block is executed when none of the case labels matches with the value of the expression/variable

Check Your Progress 2

4. State whether the following statements are true or false:
   a) Every if statement can be converted into an equivalent switch statement.
   b) 'default' case is mandatory in a switch statement.
   c) An if statement may contain compound statements only in the else clause.
   d) A break statement must be used in a switch statement.
   e) The control variable of the switch could be of type int, long or char.

3.4 LOOP CONTROL STATEMENT

In programming, there may arise some situations where a repetitive work has to be carried out. For example, suppose we want to display the sentence "IDOL, Gauhati University" 100 times. What will we do? We can display the sentence by writing 100 *cout* statements. But, it will be a time consuming process. Similarly, suppose we want to display the numbers between 1 and 1000. So far we have learnt only one solution, displaying the numbers with 1000 *cout* statements. But, this is not a solution. The solution is, we have to use loop.

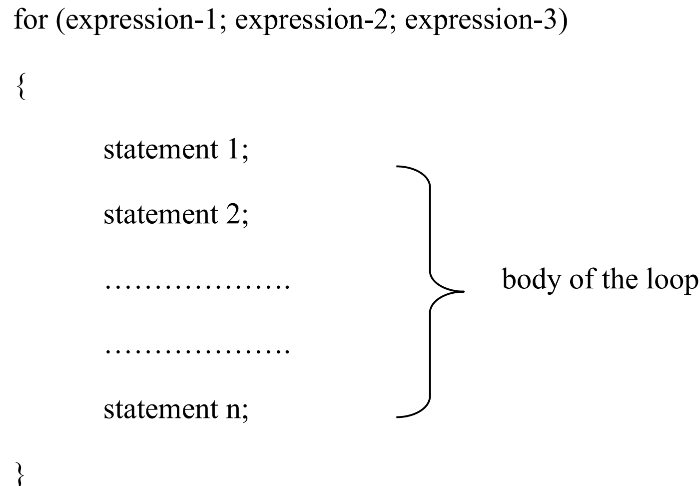
Loop control structures are used to execute and repeat a block of statements depending upon the loop value..In C++, there are three types of loop control statements.

- for loop
- while loop
- do while loop

In the following sections, we will discuss the use of these three loops with some examples.

3.4.1 The *for* loop

The *syntax* of **for***loop* is:



- **expression-1**is*initialization*statement.
  The*statement(s)* are *assignment statement(s)* used to set the *loop* control *variable/variables*. These *statement/statements* will *execute* before the first *iteration* of the *loop*.
- **expression-2**is *condition*.
  The *condition* determines the termination of the *loop*. Before every *iteration*, the *condition* is/are *checked* and if found *true* then the next *iteration* will take place.
- **expression-3**is *increment* or *decrement statement*.
  The *statement* denote how to change the states of the control *variable(s)* after each *iteration*.
- the section within "**{**" and "**}**" is called as the *body* of the *loop*.

Example 3.15: Write a program in C++ to display all the numbers between 1 and 100.

Solution:

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

void main()
{
int i;
```

```
clrscr();
cout<<"The nos. between 1 and 100 are: \n";
for(i=1; i<=100; i++)
cout<<i;
getch();
}
```

Explanation:

In the “for” *loop*,

- in**expression-1**, the *variable* “**i**” is initialized to1as the starting no. of the range.
- in**expression-2**, the *condition* is set as “i<=100” because the *loop* will continue as long as the value of “**i**” remains less than or equal to 100.
- in**expression-3**, the value of “i” is incremented by 1 after each *iteration*.
- in the *body* of the *loop*, the “cout<<” will display the current value of “i” with a tab space in between(\t means a space with tab).

Thus the numbers between1and 100 will be displayed.

Example 3.16: Write a program in C++ to display all the numbers between 1 and 100 those are divisible by 3.

Solution:

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

void main()
{
int i;
clrscr();
cout<<"The nos. between 1 and 100 those are divisible by 3 are:\t";
for(i=1; i<=100; i++)
{
 if(i%3==0)
 cout<<"\t"<<i;
}
getch();
```

```
}
```

Explanation:

In the “for” *loop*,

- in**expression-1**, the *variable* “**i**” is initialized to1as the starting no. of the range.
- in**expression-2**, the *condition* is set as “i<=100” because the *loop* will continue as long as the value of “i” remains less than or equal to 100.
- in**expression-3**, the value of “i” is incremented by1after each *iteration*.

In the *body* of the *loop*,

- the remainder of the division, “i” by 3is calculated and compared equality to 0 using “if” statement.
- and if the condition satisfies, i.e. the current value of “i” is divisible by 3, then “cout<<” will display the current value of “i” with a tab space in between. (\t means a space with tab)

Thus the numbers, those are divisible by 3, between 1 and 100 will be displayed.

Example 3.17: Write a program in C++ to find the sum of first *n* natural numbers where ***n*** is entered by user. (Note: 1,2,3... are called natural numbers.)

Solution:

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

void main()
{
int n, i, sum=0;
clrscr();
cout<<"Enter the value of n: \t";
cin>>n;
for(i=1; i<=n; i++)  //for loop terminates if i>n
{
sum=sum + i;
}
```

```
cout<<"The summation of the numbers between 1 and "<<n<<" is:
"<<sum;
getch();
}
```

Output:

Enter the value of n: 19
The summation of the numbers between 1 and 19 is: 190

Explanation: In this program, the user is asked to enter the value of *n*. Suppose, you have entered 19 then; i will be initialized to 1 at first. Then, the test expression in the for loop, i.e., (i<= n) becomes true. So, the code in the body of for loop will be executed which makes *sum* to 1. Then, the expression i++will be executed and again the test expression is checked, which becomes true. Again, the body of for loop will be executed which makes *sum* to 3 and this process will continue. When count will be 20, the test condition becomes false and the for loop will be terminated.

Example 3.18: Write a program in C++ to display all the even and odd numbers between 10 and 100. Also display the summation of all the even and odd numbers separately within that range.

Solution:

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

void main()
{
int i, sum=0;
clrscr();
cout<<"\nThe even numbers between 10 and 100 are: ";
for(i=10; i<=100; i++)
{
        if(i%2==0)
        {
                cout<<"\t"<<i;
                sum=sum + i;
        }
}
cout<<"\nThe summation of all the even numbers between 1 and
100 is:"<<sum;
```

```
/* sum is again initialised to 0 to calculate the summation of all odd
numbers between 1 and 100 */
sum = 0;
cout<<"\nThe odd numbers between 10 and 100 are:";
for(i=10; i<=100; i++)
{
	if(i%2!=0)
	{
		cout<<"\t"<<i;
		sum=sum + i;
	}
}
cout<<"\nThe summation of all the odd numbers between 1 and 100
is:"<<sum;
getch();
}
```

Example 3.19: Write a program in C++ to display all the even and odd numbers between a range of numbers where the starting number and the ending number of that range will be entered by the user. Also display the summation of all the even and odd numbers separately within that range.

Solution:

This program is same as the Example 3.18. Here the only modification is that the starting number and ending number of the *for loop* is not fixed. Those two numbers of the range will be entered by the user. Let us solve the problem as below:

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

void main()
{
int start_no, end_no, i, sum=0;
clrscr();
cout<<"Enter the starting no. and ending no. of the range: ";
cin>>start_no>>end_no;
cout<<"\nThe even numbers between "<<start_no<<" and
"<<end_no<<" are:";
for(i=start_no; i<=end_no; i++)
{
	if(i%2==0)
	{
```

```
                    cout<<"\t"<<i;
                    sum=sum + i;
          }
}
cout<<"\nThe summation of all the even numbers between
"<<start_no<<" and "<<end_no<<" is:"<<sum;
/* sum is again initialised to 0 to calculate the summation of all odd
numbers between starting number and ending number of the range
*/
sum = 0;
cout<<"\nThe odd numbers between "<<start_no<<" and
"<<end_no<<" are:";
for(i=start_no; i<=end_no; i++)
{
          if(i%2!=0)
          {
                    cout<<"\t"<<i;
                    sum=sum + i;
          }
}
cout<<"\nThe summation of all the odd numbers between
"<<start_no<<" and "<<end_no<<" is:"<<sum;
getch();
}
```

Example 3.20: Write a program in C++ to find the factorial of a number.

Solution:

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

void main()
{
int n, i;
long int fact;
clrscr();
cout<<"Enter a number:\t";
cin>>n;
if(n==0)
          cout<<"\nFactorial of 0 is 1\n";
else
{
          fact = 1;
          for(i=1; i<=n; i++)
```

```
        fact = fact * i;
        cout<<"The factorial of "<<n<<" is:"<<fact;
}
getch();
}
```

Output:

Enter a number: 3
The factorial of 3 is: 6

Explanation: For any positive number *n*, its factorial is calculated as factorial = 1*2*3*4...*n

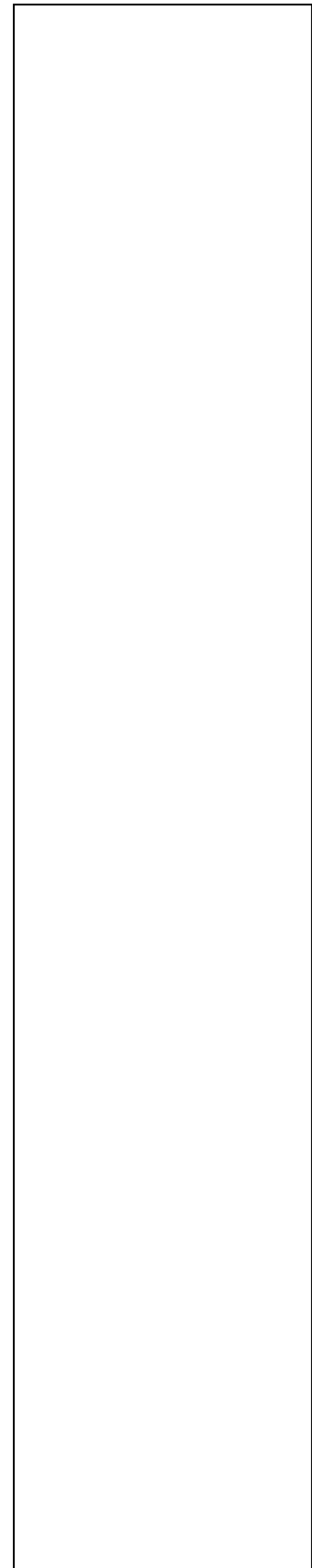
Example 3.21: Write a program in C++ to check whether a given number is prime or not.

Solution:

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

void main()
{
int n, i, flag=0;
clrscr();
cout<<"Enter a positive integer: ";
cin>>n;
for(i=2;i<=n/2;++i)
{
        if(n%i==0)
        {
                flag=1;
                break;
        }
}
if(flag==0)
        cout<<n<<" is a prime number.";
else
        cout<<n<<" is not a prime number.";
getch();
}
```

Output

Enter a positive integer: 5

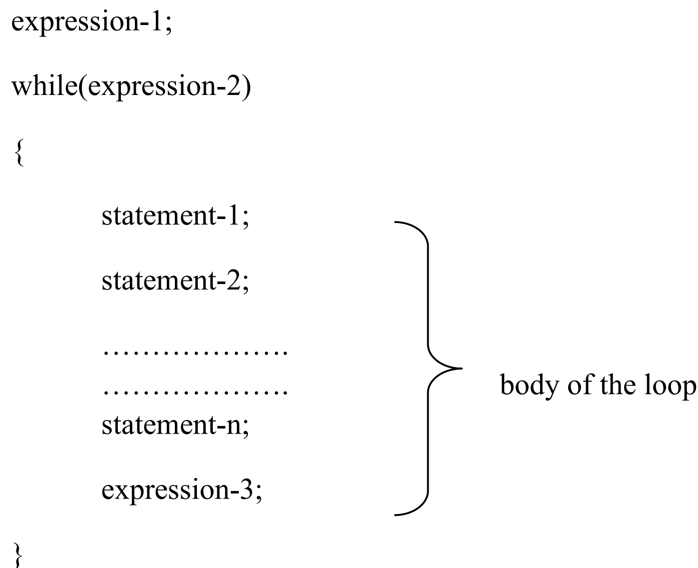


5 is a prime number.

Explanation: This program takes a positive integer from user and stores it in variable *n*. Then, for loop is executed which checks whether the number entered by user is perfectly divisible by *i* or not starting with initial value of *i* equals to 2 and increasing the value of *i*in each iteration. If the number entered by user is perfectly divisible by *i* then, *flag* is set to 1 and that number will not be a prime number but, if the number is not perfectly divisible by *i* until test condition i<=n/2 is true means, it is only divisible by 1 and that number itself and that number is a prime number.

3.4.2 The *while* loop

The *syntax* of **while***loop* is:



- **expression-1** contain *initializationstatement(s)* which is/are is mentioned before the starting of the **while** loop.
  The *statement(s)* are *assignment statement(s)* used to set the *loop* control *variable(s)*. These *statement(s)* will *execute* before the first *iteration* of the *loop*.

- **expression-2** contain *condition(s)*.
  The *condition(s)*, that determine the termination of the *loop*. Before every *iteration*, the *condition(s)* is/are *checked* and if found *true* then the next *iteration* will take place.

- **expression-3** contain *increment/decrement statement(s)*.
  The *statement(s)* denote how to change the states of the control *variable(s)* in each *iteration*.

- The section within "**{**" and "**}**" is called as the *body* of the *loop*.

Example 3.22: Write a program in C++ to display all the numbers between two input numbers using while loop.

Solution:

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

void main()
{
int start_no, end_no, i;
clrscr();
cout<<"Enter the starting no and ending no of the range:\t");
cin>>start_no>>end_no;
cout<<"The numbers between "<<start_no<<" and "<<end_no<<"
are:\t";
i=start_no;
while(i<=end_no)
{
	cout<<"\t",i);
	i++;
}
getch();
}
```

Example 3.23: Write a program in C++ to display the sum of all the digits of a number entered by the user using while loop.

Solution:

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

void main()
{
int num, rem, quo,sum=0;
clrscr();
cout<<"Enter a number:\t";
cin>>num;
```

```
while(num>0)
{
        rem = num%10;
        quo = num/10;
        sum = sum + rem;
        num = quo;
}
cout<<"The sum of digits of the number is: "<<sum;
getch();
}
```

Output:

Enter a number: 12345
The sum of digits of the number is: 15

Example 3.24: Write a program in C++ to displaythe factorial of a number using while loop.

Solution:

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

void main()
{
int num,fact;
clrscr();
cout<<"Enter a number:\t";
cin>>num;
fact=1;
while(num>0)
{
fact=fact*num;
--num;
}
cout<<"Factorial of the number is: "<<fact;
getch();
}
```

Output:

Enter a number.3
Factorial of the number is: 6

Example 3.25: Write a program in C++ to display the number of digits of an input number.

Solution:

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

void main()
{
long int n, count=0;
clrscr();
cout<<"Enter an integer: \t";
cin>>n;
while(n!=0)
{
n=n/10;
++count;
}
cout<<"Number of digits="<<count;
getch();
}
```

Output

```
Enter an integer: 34523
Number of digits=5
```

Explanation: This program takes an integer from user and stores that number in variable *n*. Suppose, user enters 34523. Then, while loopwill be executed because n!=0 will be true in first iteration. The codes inside the while loop will be executed. After the first iteration, value of *n* will be 3452 and *count* will be 1. Similarly, in second iteration *n* will be equal to 345 and *count* will be equal to 2. This process will go on and after fourth iteration, *n* will be equal to 3 and *count* will be equal to 4. Then, in the next iteration *n* will be equal to 0 and *count* will be equal to 5 and program will be terminated as n!=0 will become false.

Example 3.26: Write a program in C++ to display the characters from A to Z using while loop.

Solution:

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

void main()
{
char c;
clrscr();
c='A';
cout<<"The letters from A to Z are:\n";
while(c<='Z')
{
cout<<" "<<c;
++c;
}
getch();
}
```

Output

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

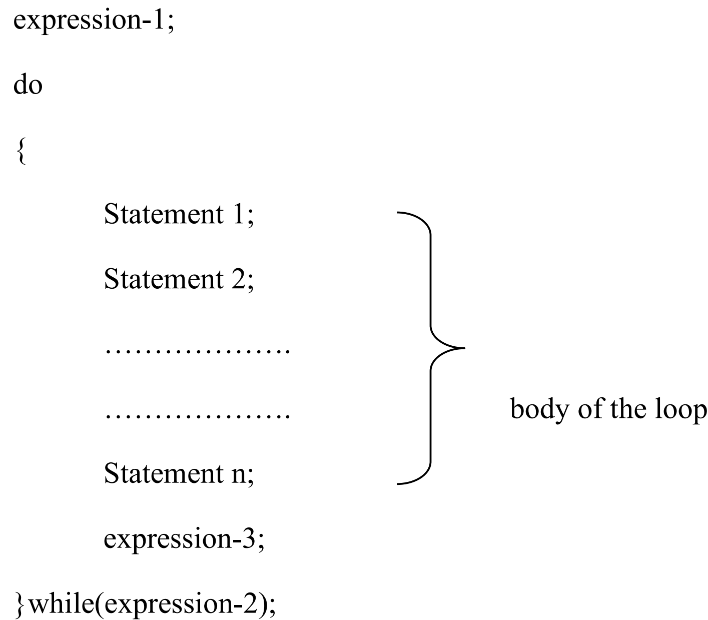
Example 3.27: Write a program in C++ to display the reverse of a number input by the user using while loop.

Solution:

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

void main()
{
long int num, rem,reverse=0;
clrscr();
cout<<"Enter a number:\t";
cin>>num;

while(num!=0)
{
 rem = num%10;
 reverse = reverse*10 + rem;
 num = num/10;
}
cout<<"\nThe reverse number is: "<<reverse;
getch();
}
```

3.4.3 The *do while* loop

The *syntax* of **do-while***loop* is:



- **expression-1** contain *initialization statement(s)* which is/are mentioned before the starting of the **while** loop.
  The *statement(s)* are *assignment statement(s)* used to set the *loop* control *variable(s)*. These *statement(s)* will *execute* before the first *iteration* of the *loop*.
- **expression-2** contain *condition(s)*.The *condition(s)*, that determine the termination of the *loop*. After every *iteration*, the *condition(s)* is/are checked and if found *true* then the next *iteration* will take place.
- **expression-3** contain *increment/decrement statement(s)*.
  The *statement(s)* denote how to change the states of the control *variable(s)* in each *iteration*.
- The section within "**{**" and "**}**" is called as the *body* of the *loop*.

Example 3.28: Write a program in C++ to check whether an input character is a vowel or not (using do while loop).

Solution:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        char c, ans;
        do
        {
        clrscr();
        cout<<"Enter the Character to check:\t";
        cin>>c;
        switch(c)
        {
                case 'a':
                case 'A':
                case 'e':
                case 'E':
                case 'i':
                case 'I':
                case 'o':
                case 'O':
                case 'u':
                case 'U': cout<<"The input character is a vowel";
                            break;
                default: cout<<"The input character is not a vowel";
        }
        cout<<"\nDo you want to continue(y/n):";
        cin>>ans;
        }while(ans=='y');
        getch();
}
```

Explanation*:*

Inside the **do-while** loop

- First the character to be checked is taken input in variable "c".
- In the "switch" statement, since the output will the same for all the vowels [i.e. 'a', 'A', 'e', 'E', 'i', 'I', 'o', 'O', 'u', 'U'], the cases with all the vowels(small letter and capital letter) are mentioned serially and in the last case, i.e., "case 'U', the display statement is mentioned for displaying "The input character is a vowel". And then because of the "break" statement, the "switch" statement will end.

The case "default" will satisfy if all the above cases do not satisfy and "The input character is not a vowel" will be displayed and the "switch" statement will end.

- After "switch" statement, the user is asked whether he/she wishes to continue by taking a character input to variable "ans".

- In the condition inside "while", the value of variable "ans" is compared with character 'y' for equality and if it satisfies then the "do-while" loop will continue otherwise the "do-while" loop will terminate and thus the program ends.

N.B. The programs that are shown using for loop and while loop can be easily converted using the do while loop.

3.5 COMPARISION OF THE LOOP STATEMENTS

	for loop	**while** loop	**do-while** loop
1.	A **for** loop is used to execute a block of statements depending on the condition which is evaluated at the beginning of the loop.	A **while** loop is used to execute a block of statements depending on the condition which is evaluated at the beginning of the loop.	A **do-while** loop is used to execute a block of statements depending on the condition which is evaluated at the end of the loop.
2.	The block of statements will not be executed when the **condition does not satisfy**, i.e., value of the condition is **false**.	The block of statements will not be executed when the **condition does not satisfy**, i.e., value of the condition is **false**.	The block of statements will not be executed when the **condition does not satisfy**, i.e., value of the condition is **false** but the block will execute at-least once irrespective of the value of the condition.

3.	Variable(s) is/are initialized at the beginning of the loop which is/are used to control the loop.	Variable(s) is/are initialized before the starting of the loop which is/are used to control the loop.	Variable(s) is/are initialized before the starting of the loop which is/are used to control the loop.
4.	The statements to change the state of the control variable(s) is/are mentioned within "**( )**" in expression-3.	The statements to change the state of the control variable(s) is/are mentioned just before the end of the body of the loop.	The statements to change the state of the control variable(s) is/are mentioned just before the end of the body of the loop.

3.6 NESTED LOOP

A loop may contain another loop within its body. This form of loop inside a loop is called *nested loop.*In a nested loop, the inner loop must terminate before the outer loop can be ended.

Example 3.29: Write a program in C++ to display the multiplication table of 1 and 2.

Solution:

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

void main()
{
	int i,j;
	clrscr();
	cout<<"The multiplication table of 1 and 2 are:\n";
	for(i=1;i<=2;i++)
	{
		for(j=1;j<=10;j++)
		cout<<i<<"x"<<j<<"="<<i*j<<"\n";
		cout<<"\n";
	}
```

```
    getch();
}
```

3.7 GOTO STATEMENT

The ***goto*** statement is used to transfer the control in a program from one point to another point unconditionally. This is also called unconditional branching. The syntax of the ***goto*** statement is:

```
goto label;
```

where***label*** is a valid identifier to indicate the destination where a control can be transferred.

3.8 break STATEMENT

The *break* statement causes an immediate exit from the innermost loop. When the keyword *break* is encountered inside any loop, control automatically passes to the statement after the loop. The break statement can also be used with *switch* statement that we studied earlier.

Example3.30:

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

void main()
{
int i, num;
clrscr();
for (i=1;i<=5;i++)
{
cout<<"\n\nEnter a number:\t";
cin>>num;
if(num<0)
{
```

```
cout<<"\nYou have entered a -ve number.\n";
break;
}
cout<<"\nThe value of i in the loop is:"<<i;
cout<<"\nThe number you have entered is:"<<num;
}
cout<<"\nGood bye";
getch();
}
```

Output:

```
Enter a number: 10
The value of i in the loop is:1
The number you have entered is: 10
Enter a number: 20
The value of i in the loop is: 2
The number you have entered is: 20
Enter a number: -5
You have entered a -ve number.
Good Bye
```

Explanation:

In this case, when we put the value as -5, the statement “if(num<0)” returns true, so the statement “You have entered a -ve number.” is displayed. Since we have used a ***break*** statement after that statement, the program control terminates the loop immediately. So, the statements i.e. **cout<<"\nThe value of i in the loop is:"<<i; and cout<<"\nThe number you have entered is:"<<num;**are not displayed and the statement **cout<<"\nGood bye";**is displayed.

3.9 continue STATEMENT

Sometimes we want to take the control to the beginning of the loop by passing the statements inside the loop which have not yet been executed. The *continue* statement forces the next iteration of the loop to take place, skipping any statement(s) following the *continue* statement in the body of the loop. The syntax of the *continue* statement is:

continue;

STOP TO CONSIDER

continue statement is not used with switch statement

Example 3.31:

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

void main()
{
int i, value;
clrscr();
for(i=1;i<=4;i++)
{
cout<<"\n\nEnter a number:\t";
cin>>value;
if(value<=0)
{
	cout<<"\nZero or -ve value found\n";
	continue;
}
cout<<"\nThe value of i in the loop is:"<<i;
cout<<"\nThe entered number is:"<<value;
}
getch();
}
```

Output:

Enter a number: 10
The value of i in the loop is: 1
The entered number is: 10
Enter a number: 20
The value of i in the loop is: 2
The entered number is: 20
Enter a number: -5
Zero or -ve value found
Enter a number: 30
The value of i in the loop is: 4

The entered number is: 30

Explanation:
In the above example, when we put the value as -5, the *if* condition returns true and the statement inside the *if* block " Zero or -ve value found " is displayed. But since there is a *continue* statement after that statement, the program control skips the next two **cout** statements and goes to the next iteration of the loop.

3.10 exit () function

The function exit() is used to terminate the program execution immediately. The syntax is:

exit(status);

where 'status' is the termination value returned by the program and is an integer. Normal termination usually returns 0.

Check Your Progress 3

5. State whether the following statements are ***true*** or ***false***

a) The while and for loops cannot be nested loops the way *if* statement can be nested.
b) Loop is a mechanism to execute a set of statements a number of times.
c) The break statement helps immediate exit from any part of the loop.
d) A while loop may always be converted to an equivalent for loop.
e) In a C program, use of goto statement is generally not recommended.
f) You can use one break statement in one loop.
g) The exit() function causes an exit from a function.
h) It is not possible to have a switch statement nested within while or for loops.
i) A continue statement causes an exit from a loop.

j) A do while loop is useful when the body of the loop will be executed at least once.
k) Multiple increment expressions in a for loop expression are separated by commas.
l) If a loop does not contain any statement in its loop body it is said to be an empty loop.
m) A loop can contain another loop in its body.
n) The while loop evaluates a test expression before allowing entry into the loop.
o) In nested loops, the inner loop must terminate before the outer loop terminates.
p) Statements inside a do while loop will be executed at least once.
q) The continue statement is used to skip some statements within a loop and start next iteration.
r) The break statement is used when it is required to exit from a loop other than by testing of termination condition.

6. Fill in the blanks:
   a) The initialisation, testing and increment can be done in the ___________ statement itself.
   b) Nesting can be done upto ________ level for while loop.
   c) Example of an infinite loop is ____________
   d) A ___________ is used to separate the three parts of the loop expression in a for loop.
   e) When the __________ statement is executed, the program skips the remaining statements in the loop and goes back to test the loop condition.
   f) An infinite for loop has _________ missing expression.
   g) A __________ loop can also be an empty loop, if it contains just a null statement in its body.
   h) The ___________ is executed at least once always before it evaluates the test expression.
   i) ____________ statement exits from some deeply nested structure at once.
   j) ___________ function forces exit from a program.

3.11 SUMMING UP

There are three ways for taking decisions in a program. First way is to use the **if else** statement, second way is to use the conditional operators and third way is to use the **switch** statement. The default scope of the **if** statement is only the next statement. So, to execute more than one statement they must be written in a pair of braces.

A **if** block need not always be associated with an **else** block. However, an **else** block is always associated with an **if s**tatement.

Loop structures are used to execute a statement/block of statements repeatedly a number of times. Three types of loops are used in C++: ***for***, ***while*** and ***do while***. In both ***for*** and ***while*** loop, the condition is checked before each iteration of the loop. But in case of ***do while*** loop, the condition is checked after each iteration of the loop.

The ***goto*** statement transfers control to a label. The ***break*** statement terminates the execution of the nearest enclosing ***do***, ***for***, ***while*** or ***switch*** statement in which it appears.

The continue statement passes control to the next iteration of the nearest enclosing ***do***, ***for*** or ***while*** statement in which it appears bypassing any remaining statements in the ***do***, ***for*** or ***while*** statement body.

3.12 ANSWERS TO CHECK YOUR PROGRESS

1.The *if* statement is a control statement that tests a particular condition. Whenever, the evaluated condition comes out to be true, then that action or the set of actions are carried out. Whereas, if else statement is used to execute a statement block or a single statement depending on the value of a condition.If the condition evaluates to true, then the statement(s) inside the if block will be executed, otherwise, the statement(s) inside the else block will be executed.

2.An if statement may have another ***if*** statement in the true condition block and false condition block. This compound statement is called ***nested if*** statement.

3.20 5

4.

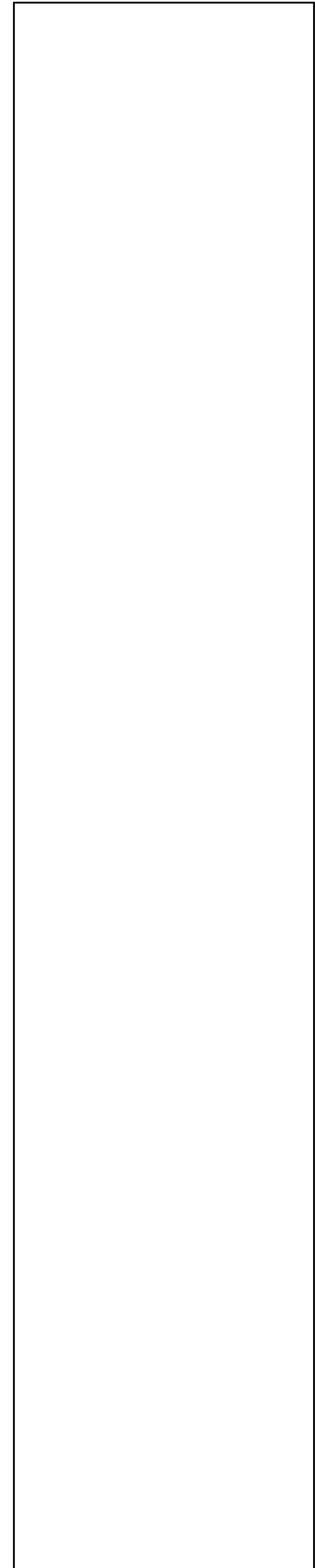
a) False
b) False
c) False
d) False
e) True

5.

a) False
b) True
c) True
d) True
e) True
f) False
g) False
h) False
i) False
j) True
k) True
l) True
m) True
n) True
o) True
p) True
q) True
r) True

6.

a) for
b) 3
c) while(1)
d) ;
e) continue
f) test
g) while or for
h) do while
i) goto
j) exit()



3.13 POSSIBLE QUESTIONS

Short answer type questions:

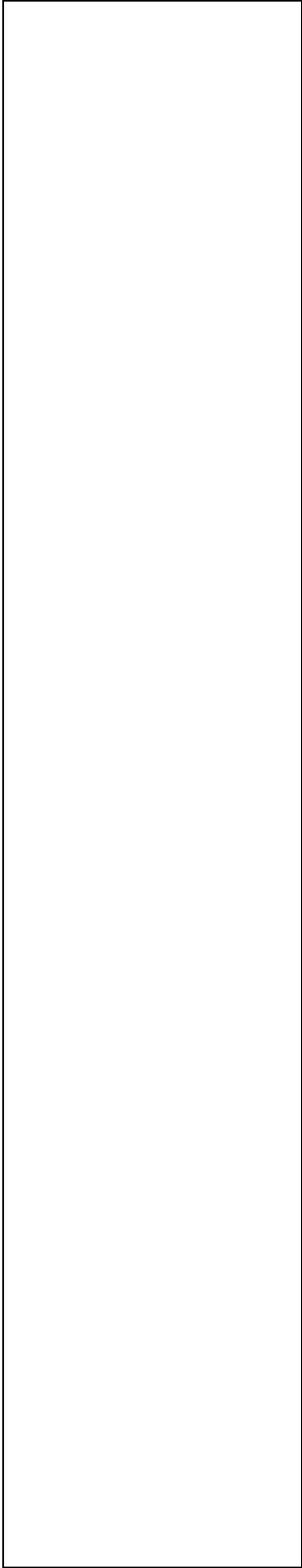
1. What is the effect of absence of break in switch case statement? What is the purpose of default?
2. What is the similarity and difference between break and continue statement?
3. What is the function of break statement in a loop?
4. Differentiate between while and do while loop.
5. Differentiate between break and exit ().

Long answer type questions:

1. Explain the various if structures with suitable examples.
2. Differentiate between if else and switch structures with an example.
3. What is nested if statement? Explain with an example.
4. Write a program in C++ to display the smallest of three numbers entered by the user.
5. Explain the loop control structures used in C++ with examples.
6. Write a program in C++ to convert a binary number into a decimal number.
7. Write a program in C++ to convert a decimal number into its equivalent binary number.
8. Write a program in C++ to check whether a number is palindrome or not.
9. Write a program in C++ to display the multiplication table of 5.
10. Write a program in C++ to display all the numbers divisible by 7 between 20 and 200.
11. Write a program in C++ to generate the first 100 positive integers divisible by 5.
12. Write a program in C++to display the prime numbers between 10 and 100.
13. Write a program in C++to display the factorial of all the numbers between 1 and 10.

3.14 REFERENCES AND SUGGESTED READINGS

1. Kanetkar, Yashavant P. *Let us C++*. BPB publications, 2020.
2. E Balagurusamy, *Object-Oriented Programming with C++*, Mc Graw Hill Publications, 2020



UNIT 4 : ARRAYS AND STRINGS IN C++

Unit Structure:

4.1 Introduction
4.2 Unit Objectives
4.3 Definition of Array
4.4 Types of Array and Declaration
  4.4.1 One Dimensional array
  4.4.2 Multi-Dimensional array
4.5 Operations on One Dimensional Array
  4.5.1 Initialization
  4.5.2 Read and Access Array Elements
  4.5.3 Searching and Sorting
4.6 Operations on Two Dimensional Array
  4.6.1 Initialization
  4.6.2 Read and Access Array Elements
4.7 Definition of String
4.8 Input and Display a String
4.9 Operations on Strings
4.10 Array of Strings
4.11 String Library Functions
4.12 Summing Up
4.13 Answers to Check Your Progress
4.14 Possible Questions
4.15 References and Suggested Readings

4.1 INTRODUCTION

We have already learnt how to declare a variable and assign a value to it. Sometimes in a C++ program, multiple inputs of similar type are required. In such cases, it may happen that the number of input data is very large so in such cases, it is not possible to declare multiple variables for all the input data. Here, we can use the concept of Array in C++ programming.

We have also learnt how to input a character. Sometimes, in a C++ program, we are required to input a word or a collection of some words or some sentences. In such cases, we can use character arrays to input strings.

4.2 UBIT OBJECTIVES

After reading this unit you are expected to be able to learn:

- What are Arrays?
- Different types of Arrays.
- How to declare and initialize an Array?
- Different operations on one dimensional array and two dimensional array.
- What is string?
- How to input and display a string?
- Different operations on strings.
- What is array of strings?
- Different library functions on strings.

4.3 DEFINITION OF ARRAY

An array is a collection of homogeneous pieces of data that are all identical in type and stored in consecutive memory locations. For example, in Fig.4.1, A is an integer array storing 10 integer numbers.

	0	1	2	3	4	5	6	7	8	9
A	23	45	32	67	8	12	123	89	90	22

Fig.4.1

Now let us consider the base address of A as 5012. Base address of an array is the memory address of the first element in the array. So, 5012 is the memory address of 23. Now according to the definition of array, the memory address of the second element of A is 5014 as the memory size of int type variable is 2 bytes. In this way, the memory address of the third and fourth element of A is 5016 and 5018 respectively as the array elements are stored in consecutive memory locations.

4.4 TYPES OF ARRAY AND DECLARATION

There are two types of array available in C++ programming that is explained as follows:

4.4.1 One Dimensional Array

The declaration syntax of one dimensional array is

Datatype arraynm [N];

Here datatype specifies type of the data to be stored in the array and arraynm is the name of the array. [N] specifies that arraynm is a one dimensional array and it can store maximum N number of elements.

For example: int arr[30] ;

Here, int specifies that the array will store int type of data and arr is the name of the array. The array arr can store maximum 30 number of int type data.

4.4.2 Multi-dimensional array

The declaration syntax of multi dimensional or N dimensional array is

Datatype arraynm [size1][size2][size3]……..[sizeN]

For example: example of declaring a 2 dimensional array is

float arrtwo [30][20] ;

Here [30][20] means arrtwo is a 2 dimensional array and it can store maximum 30×20 = 600 number of float type data. A two dimensional array also can be called as a matrix.

Again example of declaring a 3 dimensional array is

int arrthree[10][20][10];

Here arrthree is a three dimensional array which can store maximum 10X20X10 = 2000 number of int type data.

4.5 OPERATIONS ON ONE DIMENSIONAL ARRAY

There are different operations that can be performed on one dimensional arrays as explained in the following sections.

4.5.1 Initialization

A one dimensional array can be initialized by using the following statements.

```
int arr1[5] = {4 , 7 , 8 , 23 , 56} ;
int arr2[ ] = {2 , 7 , 1 , 9};
char arr3[ ] = {'A' ,  'H' ,  'J' ,  'B'};
```

By initializing a one dimensional array, we can store some initial values into the array at compile time. From the above statements, the array arr1 is initialized with the values 4 , 7 , 8 , 23 , 56 as first, second, third, fourth and fifth element of arr1 respectively. Now if an array is not initialized then it contains garbage values because by default the storage class of array is auto. So if the storage class of an array is declared to be static then all the array elements will be initialized to zero.

4.5.2 Read and Access Array Elements

We have to learn how to access individual element in a one dimensional array and how array elements can be read from standard input device. Accessing of array elements can be performed with the number in the brackets (for example: [4]) following the array name. This number is called as subscript. This number specifies the element's position in the array. In C++ programming, all the array elements are numbered, starting with subscript value 0. So, an array of size 20 has subscript values starting from 0 to 19. So if we want to access the 5th element of an array 'arr' then we can use 'arr[4]' . Now we can read and display the 5th element of an integer array 'arr' with the help of the following programming statements.

```
int  arr[30];
```

```
cin >> arr[4] ;    /*  an int type data is read from the standard
                       input   device into the 5th position of arr */
cout << arr[4] ;   /*  the 5th element of arr is displayed in the
                       standard output device*/
```

Now from fig. 4.1, A[0] will refer to the first element of A which is 23 and in this way A[9] will refer to the 10th element of A which is 22.

Here, we have an important point to note that is what happens when the subscript value used at the time of reading an array element is greater than or equal to the size of the array. In such cases, for C++ programming, data will be entered into the memory space outside the memory space allocated for the array.. So there should be a conditional statement in our C++ programs to check that the subscript value never exceed the array size at the time of reading array elements.

> **STOP TO CONSIDER**
>
> Direct access or random access of array elements is possible because the array elements are stored in consecutive memory locations.

Now, a C++ program to input and display *n* elements in an integer array is given below.

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

int main( )
{
    int arr[30];
    int i , n;
    clrscr( );

    cout << "\nHow many numbers you want to enter(maximum 30)= ";
    cin >> n;

    if( n>30 )
    {
        cout << "\nYour entered value exceeds the size of the array";
    }
```

```
    else
    {

        for( i = 0 ; i < n ; i++)
        {
            cout << "\nEnter the   " << i+1 << "th number = ";
            cin >> arr[i] ;
        }

        for( i = 0 ; i < n ; i++)
        {
            cout << "\nThe   "<<  i+1 << "th number in the array is=" << arr[i];
        }
    }
    getch( );
    return 0;
}
```

4.5.3 Searching and Sorting

Searching a particular element in an array is another important operation performed on arrays. This can be performed by comparison operation between the element to be searched and elements available in the array. Two fundamental searching algorithms are Linear search and Binary search.

Arranging array elements in ascending or descending order in an array is called the sorting operation. There are different algorithms available for sorting operation. For example: Bubblesort, Selection sort, Insertion sort etc.

A C++ program to search an element in an integer array using Linear search technique and display the subscript value where the element is present in the array is given as follows.

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

int main( )
{
    int arr[30];
    int i , n , sno , flag = 0;
    clrscr( );
```

```
    cout << "\nHow many numbers you want to enter(maximum 30)= ";
    cin >> n;

    if( n>30 )
    {
        cout << "\nYour entered quantity of numbers exceeds the size of the array";
    }
    else
    {

        for( i = 0 ; i < n ; i++)
        {
            cout << "\nEnter the   " << i+1 << "th number = ";
            cin >> arr[i] ;
        }
        cout << "\n Enter the number to be searched in the array = ";
        cin >> sno;
        for(i = 0 ; i < n ; i++)
        {
                if(sno = = arr[i])
                {
                        cout << "\n" <<sno << "  is present in the array";
                        cout << "\n Subscript value of the searched element is ="<<
i;
                        flag = 1;
                        break;
                }
        }

        if(flag = = 0)
        {
            cout << "\n" << sno <<"   is not present in the array";
        }
  }
  getch( );
  return 0;
}
```

Example 4.1: Write a C++ program to search a specific data in a one dimensional array using Binary search algorithm.

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

int main( )
{
        int arr[50] , i , n , start , mid , end , src_data;
        clrscr( );
        cout << "\n Enter the total number of data in the array:";
        cin >> n;
        if(n <= 50)
        {
                cout << "\n Enter data into the array:";
                for( i = 0 ; i < n ; i++)
                {
                        cout <<"\n Enter    ”<< i+1 << “th data:";
                        cin >> arr[i];
                }
                cout << "\n Enter the data to be searched:";
                cin >> src_data;

                cout << "\n The array data are:\n";
                for(i = 0 ; i < n ; i++)
                        cout << “\t” << arr[i];

                start = 0;
                end = n-1;
                while(start <= end)
                {
                    mid = (start + end)/2;
                    if(arr[mid] = = src_data)
                    {
                        cout << “\n” << src_data << “   is available in the array";
                        cout <<“\n Subscript value of the searched element is = ” <<
mid;
                        break;
                    }
                    else if(arr[mid] < src_data)
                        start = mid+1;
```

```
                else
                    end = mid-1;
            }

            if(start > end)
                    cout << "\n" << src_data << "   is not available in the
array;

    }
    else
    {
            cout << "\n The total number of data exceed the size of the array";
    }
    getch( );
    return 0;
}
```

Example 4.2: Write a C++ program to find out the minimum and the maximum number in an integer array.

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

int main( )
{
    int arr[30];      //arr is a one dimensional integer array with size
30
    int i , n , min , max;
    clrscr( );

    cout << "\nHow many numbers you want to enter(maximum 30)= ";
    cin >> n;
    if(n > 30)
    {
        cout <<"\nYour entered quantity of numbers exceeds the size of the array";
    }
    else
    {
        for( i = 0; i < n ; i++)
        {
            cout << "\nEnter the   "<< i+1 << "th number = ";
```

```
        cin >> arr[i];
    }
    min = arr[0];
    max = arr[0];
    for( i = 1 ; i < n ; i++)
    {

        if(arr[i] < min)
            min = arr[i];
        if(arr[i] > max)
            max = arr[i];
    }
    cout <<"\nThe minimum of the numbers present in the array is = "<<min;
    cout <<"\nThe maximum of the numbers present in the array is = "<<max;

  }
  getch( );
  return 0;
}
```

Example 4.3:

Write a C++ program to find out the summation of all the numbers present in an integer array.

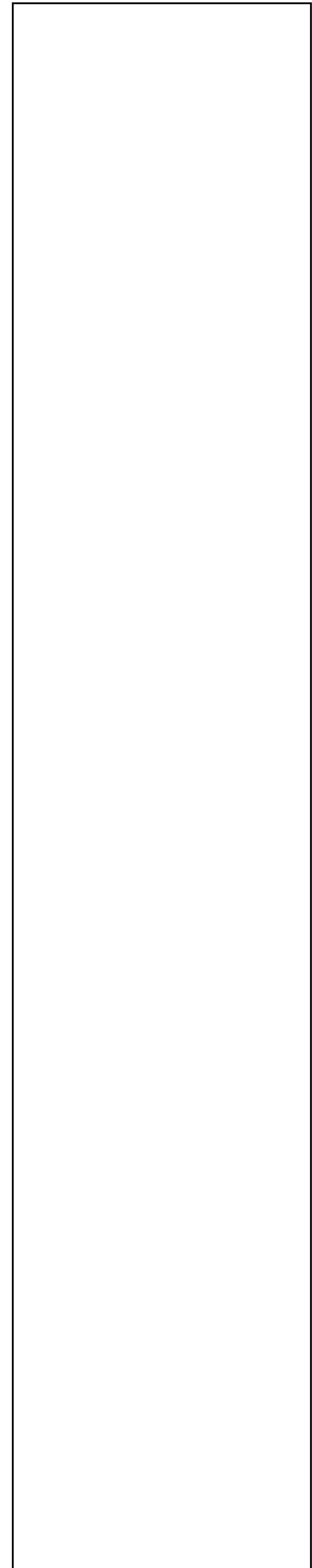
```
#include <iostream.h>
#include <conio.h>

int main( )
{

  int arr[30];
  int i , n , sumarr = 0;
  clrscr( );

  cout <<"\nHow many numbers you want to enter(maximum 30) = ";
  cin >> n;

  if(n > 30)
  {
```

```
        cout<<"\nYour entered quantity of numbers exceeds the size of the
array";
    }

    else
    {

        for(i = 0 ; i < n ; i++)
        {

            cout << "\nEnter the   "  << "th number = " << i+1;
            cin >> arr[i];
            sumarr = sumarr + arr[i];

        }

        cout << "\nThe summation of the numbers present in the array is="<< sumarr;

       }

       getch( );
       return 0;

   }
```

Example 4.4: Write a C++ program to sort some integer numbers stored in a one dimensional array using Selection sort algorithm

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

int main( )
{
        int arr[50] , i , j , n , index , min;
        clrscr( );
        cout <<"\n Enter the total number of data in the array:";
        cin >> n;
        if(n <= 50)
        {
                cout << "\n Enter data into the array:";
```

```
                    for(i = 0 ; i < n ; i++)
                    {
                            cout << "\n Enter  "<< i+1 << "th data:";
                            cin >> arr[i];
                    }
                    cout << "\n Before sorting the array data are:\n";
                    for(i = 0 ; i < n ; i++)
                            cout << "\t" << arr[i];

                    for(i = 0 ; i < n-1 ; i++)
                    {
                            index = i;
                            min = a[i];

                            for(j = i+1 ; j <= n-1 ; j++)
                            {

                                    if ( min > a[j] )
                                    {
                                            min = a[j];
                                            index = j;
                                    }
                            }

                            a[index] = a[i];
                            a[i] = min;
                    }
                    cout << "\n After sorting the array data are:\n";
                    for(i = 0 ; i < n ; i++)
                            cout << "\t" << arr[i];
    }
    else
    {
                    cout <<"\n The total number of data exceed the size
of the array";
     }
     getch( );
     return 0;
}
```



Example 4.5: Write a C++ program to sort some integer numbers stored in a one dimensional array using Bubble sort algorithm

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

int main( )
{
	int arr[50] , i , j , n , temp;
	clrscr( );
	cout <<"\n Enter the total number of data in the array:";
	cin >> n;
	if(n <= 50)
	{
		cout <<"\n Enter data into the array:";
		for(i = 0 ; i < n ; i++)
		{
			cout << "\n Enter  "<< i+1<< "th data:";
			cin >> arr[i];
		}
		cout << "\n Before sorting the array data are:\n";
		for(i = 0 ; i < n ; i++)
			cout << "\t" << arr[i];

		for (i = 0; i < n - 1;  i++)
		{

		      for(j = 0 ; j < n-i-1 ; j++)
		      {

			   if (arr[j] > arr[j + 1])
			   {

				temp = arr[j];
				arr[j] = arr[j + 1];
				arr[j + 1] = temp;


			    }

		       }
```

```
                }

                cout <<"\n After sorting the array data are:\n";
                for(i = 0 ; i < n ; i++)
                        cout << "\t" << arr[i];
    }
    else
    {
                cout <<"\n The total number of data exceed the size
of the array";
     }
     getch( );
     return 0;
}
```

Example 4.6: Write a C++ program to sort some integer numbers stored in a one dimensional array using Insertion sort algorithm

```
#include <iostream.h>
#include <conio.h>
int main()
{
        int arr[50] , i , j , n , key;
        clrscr( );
        cout <<"\n Enter the total number of data in the array:";
        cin >> n;
        if (n <= 50)
        {
                cout <<"\n Enter data into the array:";
                for(i = 0 ; i < n ; i++)
                {
                        cout << "\n Enter  "<<i+1<< "th data:";
                        cin >> arr[i];
                }
                cout <<"\n Before sorting the array data are:\n";
                for(i = 0 ; i < n ; i++)
                        cout << "\t" << arr[i];

                for(i = 1 ; i < n ; i++)
```

```
                {

                        key = a[i];
                        j = i-1;
                        while (j >= 0 && a[j] > key)
                        {
                                a[j+1] = a[j];
                                j--;
                        }
                        a[j+1] = key;
                }
                cout << "\n After sorting the array data are:\n";
                for(i = 0 ; i < n ; i++)
                        cout << "\t" << arr[i];
    }
    else
    {
                cout <<"\n The total number of data exceed the size
of the array";
    }
    getch( );
    return 0;
}
```

STOP TO CONSIDER

Let us declare an one dimensional array as int Arr[20]. Then Arr and &Arr[0] will provide the base address of the array.

4.6 OPERATIONS ON TWO DIMENSIONAL ARRAYS

4.6.1 Initialization

Two dimensional arrays can be initialized as follows:

```
int  arrtwo1[4][3] = {                              //  Fig. 4.2

                        {4, 8 , 9 },
```

```
{7, 9, 21},
{1, 8, 19},
{71, 6, 2}
};
int arrtwo2[4][3] = { 4 , 8 , 9 , 7, 9, 21 , 1 , 8 , 19 , 71 , 6 , 2};
int arrtwo3[ ][3] = { 4 , 8 , 9 , 7, 9 , 21 , 1 , 8 , 19 , 71 , 6 , 2};
```

Here three ways of initializing a two dimensional array are shown above. In case of initializing two dimensional arrays, it is necessary to mention the second dimension of the array; otherwise it will not work in C++ programming. So, in two dimensional array initializations as shown below will not work in C++ programming.

```
int arrtwo[ ][ ] = { 4 , 8 , 9 , 7, 9, 21 , 1 , 8 , 19 , 71 , 6 , 2};
int arrtwo[4][ ] = { 4 , 8 , 9 , 7, 9, 21 , 1 , 8 , 19 , 71 , 6 , 2};
```

	0	1	2
0	4	8	9
1	7	9	21
2	1	8	19
3	71	6	2

Fig. 4.2: Diagrammatic representation of the array arrtwo1 declared above

4.6.2 Read and Access Array Elements

We need two subscript values to access a specific cell of a two dimensional array. Here first subscript value will represent the row index and the second subscript value will represent the column

index of the specific cell of a two dimensional array. For example: consider the two dimensional array arrtwo1 declared above whose diagrammatic representation is given in fig. 4.2. Here arrtwo1[0][0] will refer to the first element in the array with row index 0 and column index 0 which is 4. Again arrtwo1[3][0] will refer to 71.

So arrtwo1[i][j] will give the array element with i^{th} row number and j^{th} column number of the array arrtwo1.

Now to display the array element with row number 3 and column number 2 of array arrtwo1 on the standard output device, the following programming statement in C++ can be used:

```
cout << arrtwo1[3][2];
```

So the output of the statement will be 2.

Again to read a new array element from the standard input device into the cell with row number 3 and column number 2 of array arrtwo1, the following programming statement can be used.

```
cin >> arrtwo1[3][2];
```

So result of this statement will be a new data from standard input device will replace the existing array element of arrtwo1 with row number 3 and column number 2 as arrtwo1 is initialized.

Example 4.7: Write a C++ program to find out the summation of all the numbers of a matrix with integer values.

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

int main( )
{
        int i , j , row_no , col_no, matrix[20][20] ,sum = 0;
        clrscr( );

        cout << "\n Please enter the number of rows =";
        cin >> row_no;
        cout << "\n Please enter the number of columns =";
        cin >> col_no;
```

```
        cout << "\nPlease enter the  matrix:";

        for (i = 0 ; i < row_no ; i++)
        {
                for (j = 0; j < col_no ; j++)
                {
                        cout <<"\nPlease enter the  ("<< i <<
",""<<j<< ")th data:";
                        cin >> matrix[i][j];
                }
        }

        for (i = 0 ; i < row_no ; i++)
        {
                for (j = 0; j < col_no ; j++)
                {
                        sum = sum+  matrix[i][j];
                }
        }

        cout << "\n The required summation is = "<< sum;
        getch( );
        return 0;
}
```

Example 4.8: Write a C++ program to find out the summation of all the diagonal elements of a symmetric matrix with integer values.

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

int main( )
{
        int i , j , row_no , col_no, matrix[20][20] ,sum = 0;
        clrscr( );

        cout << "\n Please enter the number of rows = ";
        cin >> row_no;
        cout <<"\n Please enter the number of columns = ";
        cin >> col_no;
```

```
        if(row_no != col_no)
        {
            cout <<"\n Wrong input. Symmetric matrix required here";
        }
        else
        {
            cout << "\nPlease enter the  matrix:";

            for (i = 0 ; i < row_no ; i++)
            {
                for (j = 0; j < col_no ; j++)
                {
                    cout <<"\nPlease enter the ("<<i<< ","<<j<< ")th data =";
                    cin >> matrix[i][j];
                }
            }

            for (i = 0 ; i < row_no ; i++)
            {
                for (j = 0; j < col_no ; j++)
                {
                        if(i = = j)
                        {
                                sum = sum +  matrix[i][i];
                        }
                }
            }

            cout << "\n The required summation is = "<< sum;

        }

        getch( );

        return 0;

}
```

Example 4.9: Write a C++ program to add two matrix containing integer data

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

int main( )
{
        int i , j , row_no , col_no ;
        int matrix1[20][20] , matrix2[20][20] ,
sum_matrix[20][20];
        clrscr( );

        cout << "\n Please enter the number of rows = ";
        cin >> row_no;
        cout << "\n Please enter the number of columns = ";
        cin >> col_no;

         cout << "\nPlease enter the first matrix:";

        for (i = 0 ; i < row_no ; i++)
        {
            for (j = 0; j < col_no ; j++)
            {
                cout << "\nPlease enter the ("<<i<< ","<<j<< ")th data of  matrix1=";
                cin >> matrix1[i][j];
             }
         }


        cout << "\nPlease enter the second matrix:";

         for (i = 0 ; i < row_no ; i++)
        {
             for (j = 0; j < col_no ; j++)
             {
                  cout << "\nPlease enter the ("<<i<< ","<<j<<
") the data of  matrix2=";

                  cin >> matrix2[i][j];
                }
```

```
 }

for (i = 0 ; i < row_no ; i++)
{
        for (j = 0 ; j < col_no ; j++)
        {
                sum_matrix[i][j] = matrix1[i][j] + matrix2[i][j];

        }
}

cout << "\n The first matrix is:\n";

for (i = 0 ; i < row_no ; i++)
{
        for (j = 0; j < col_no ; j++)
        {
                cout << "\t" << matrix1[i][j];
        }
        cout << "\n";
}

cout << "\n The second matrix is:\n";

for (i = 0 ; i < row_no ; i++)
{
        for (j = 0; j < col_no ; j++)
        {
                cout << "\t"<<matrix2[i][j];
        }
        cout << "\n";
 }

cout <<"\n The resultant matrix after summation is:\n";

for (i = 0 ; i < row_no ; i++)
{
        for (j = 0; j < col_no ; j++)
        {
                 cout <<"\t"<< sum_matrix[i][j];
         }
```

```
                cout <<"\n";
        }
        getch( );
        return 0;

}
```

Example 4.10: Write a C++ program to multiply two matrix containing integer data

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

int main()
{
    int i , j , k , r1 , r2 , c1 , c2 , sum = 0;
    int matrix1[20][20] , matrix2[20][20] , matrix_mult[20][20];

    cout <<"\nPlease enter the number of rows of the first matrix=";
    cin >> r1;
    cout <<"\nPlease enter the number of columns of the first matrix =";
    cin >>c1;

    cout <<"\nPlease enter the number of rows of the second matrix =";
    cin >> r2;
    cout << "\nPlease enter the number of columns of the second matrix =";
    cin >> c2;

    if (c1 != r2)
      cout <<"\n Matrix multiplication for these dimensions of matrices is not
possible";

    else
    {
        cout <<"\nPlease enter the first matrix:";

        for (i = 0 ; i < r1 ; i++)
        {
            for (j = 0; j < c1 ; j++)
            {
                cout << "\nPlease enter the ("<<i<< ","<<j<< ")th data of  matrix1=";
```

```
            cin >> matrix1[i][j];
        }
    }


    cout << "\nPlease enter the second matrix:";

    for (i = 0 ; i < r2 ; i++)
    {
         for (j = 0; j < c2 ; j++)
         {
            cout << "\nPlease enter the ("<<i<< ","<<j<< ")th data of  matrix2=";
            cin >> matrix2[i][j];
          }
    }

   for (i = 0; i < r1; i++)
   {
            for (j = 0; j < c2; j++)
            {
                     for (k = 0; k < r2; k++)
                     {
                        sum = sum + matrix1[i][k] * matrix2[k][j];
                     }

                     matrix_mult[i][j] = sum;
                     sum = 0;
            }
    }


   cout << "\n The first matrix is:\n";

    for (i = 0 ; i < r1 ; i++)
    {
            for (j = 0; j < c1 ; j++)
            {
                     cout << "\t"<<matrix1[i][j];
            }
            cout << "\n";
    }
```

```
            cout << "\n The second matrix is:\n";

             for (i = 0 ; i <r2 ; i++)
             {
                     for (j = 0; j < c2 ; j++)
                     {
                             cout <<"\t" << matrix2[i][j];
                     }
                     cout << "\n";
             }

             cout <<"\n The resultant matrix after multiplication is:\n";

             for (i = 0 ; i < r1 ; i++)
              {
                     for (j = 0; j < c2 ; j++)
                     {
                              cout <<"\t"<< matrix_mult[i][j];
                      }
                     cout << "\n";
             }
     }

     getch( );
     return 0;
}
```

STOP TO CONSIDER

Let us declare a two dimensional array as int Arr2[20][30]. Then Arr2, Arr2[0] and &Arr2[0][0] will provide the base address of the array.

CHECK YOUR PROGRESS

1. Multiple choices

(a) In C++ programming, the subscript value of an array is

starting from_____.

(i) 0
(ii) 1
(iii) Compiler dependent
(iv) None of the above

(b) int arr[20];
The meaning of the above C++ statement is_____.

(i) arr is an integer variable
(ii) arr is an integer array capable of storing 19 integer numbers
(iii) arr is an array capable of storing 20 data
(iv) arr is an integer array capable of storing 20 integer numbers

(c) int arr[5] = {5,2,0,1,4};
arr[3] = arr[1] + arr[4];
for(i = 0 ; i < 5 ; i++)
cout << arr[i];

The output of the above statements is

(i) 5 2 0 1 4
(ii) 5 2 0 4 4
(iii) 5 1 0 6 4
(iv) 5 2 0 6 4

(d) If arr is a character array and the memory address of arr[0] is 203 then memory address of arr[3] is _____.

(i) 204
(ii) 205
(iii) 206
(iv) None of the above

(e) If you don't initialize an array what will be the elements set to?

(i) 0
(ii) an undetermined value
(iii) a floating point number

(iv) the character constant '\0'

(f) What will happen if you try to put so many values into an array when you initialize it that the size of the array is exceeded?

(i) Nothing
(ii) possible system malfunction
(iii) Error message
(iv) Other data may be overwritten

(g) What will happen if you put too few elements in an array when you initialize it?

(i) Nothing
(ii) possible system malfunction
(iii) Error message
(iv) Unused spaces will be filled with 0's or garbage.

(h) In C++ programming, number of subscript values required in case of two-dimensional array is______.
(i) one
(ii) two
(iii) three
(iv) None of the above

(i) Which of the following is not true in case of array?
(i) Array is collection of homogeneous data.
(ii) In C++ programming, size of an array can be changed at runtime.
(iii) Array elements are stored in contiguous memory locations.
(iv) None of the above

(j) Which of the following statement is a correct way to declare a two-dimensional array which can store at most 100 real numbers?
(i) int A[100][100];
(ii) float A(100)(100);
(iii) float A[10][10];
(iv) float A[100][100];

(k) Which of the following statement is a correct way to declare a one-dimensional character array which can store at most 100 characters?

(i) int ch[100];
(ii) char ch(100);
(iii) char ch[99];
(iv) char ch[100];

(l) Which of the following statement is a correct way to initialize a one-dimensional character array?

(i) char ch[6]={'G','U','I','D','O','L'};
(ii) char ch(6)={'G','U','I','D','O','L'};
(iii) char ch[6]={G , U , I , D , O , L};
(iv) char ch[6]=['G','U','I','D','O','L'];

(m) Which of the following statement is a correct way to initialize a two-dimensional integer array?

(i) int Arr[ ][ ] = { 8 , 9 , 12 , 54 , 90 , 31};
(ii) int Arr[2][ ] = { 8 , 9 , 12 , 54 , 90 , 31};
(iii) int Arr[2][3] = { 8 , 9 , 12 , 54 , 90 , 31};
(iv) All of the above;

(n) Which of the following statement can be used to read an integer number and store it in the 5th position of a one-dimensional array?

(i) cin>>Arr[5];
(ii) cin>>Arr[4];
(iii) cin<<Arr[4];
(v) cin>>Arr(4);

(o) Which of the following statement can be used in C++ to display the data stored in a two-dimensional float type array where the subscript values of the particular cell are 4 and 5?

(i) cout >> A[4][5];
(ii) cout<<A[5][6]);
(iii) cout<<"%f"<<A[4][5];
(iv) cout<<A[4][5]);

2. **State whether true or false:**

(a) To declare an integer array we have to write int arr = size(20);

(b) Array can be used to store different types of data.

(c) In C++ programming ,the subscript value of an array is starting from 1.

(d) The subscript value of the last element of an array of size 10 is 9 in C++ programming.

(e) In C++ programming, an array cannot be initialized.

4.7 DEFINITION OF STRING

String is a collection of some characters stored in a one dimensional character array. A string is always terminated by '\0' which is called NULL character. The ASCII value of '\0' is zero. For example, in fig.4.3, A is a character array with array size 10 and it stores the string "Welcome".

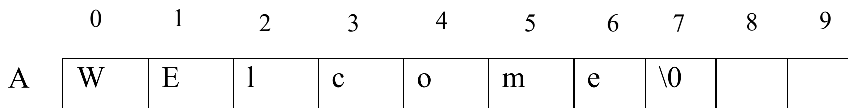


Fig.4.3

STOP TO CONSIDER

The maximum length of any string that will be stored in a character array, strn[N] is N-1 as string must be terminated by the NULL('\0') character.

4.8 INPUT AND DISPLAY A STRING

At first we require a character array to input a string. We can use standard input stream object, 'cin' and different library functions like gets ( ), getchar( ) to input a string. For example:

```
char str[40];
```

(a) Standard input stream object, ‘cin’ can be used as:

```
cin >>  str;
```

(b) gets( ) can be used as:

```
gets(str);
```

(c) getchar( ) can be used as:

```
int i = 0;
char ch;
while((ch = getchar()) != '\n')
{
        str[i] = ch;
        i++;
}
str[i] = '\0';
```

Here str is the character array where we have input a string using cin, gets( ) and getchar( ). But ‘cin’ is not capable to input a multi word string, so in case of multi word string we can use gets( ) or getchar( ). getchar( ) is a input function which can be used to input a character. So getchar( ) can be used to input all the characters of a string one by one with the help of a loop control and at the end , the NULL character(\0) is entered.

Now to display a string we can use standard output stream object, ‘cout’ and different library function like puts( ), putchar( ).

For example:

```
char str[40];
gets(str);
```

(a) Standard output stream object, ‘cout’ can be used as:

```
cout << str;
```

(b) puts( ) can be used as:

```
puts(str);
```

(c) putchar( ) can be used as:

```
int i = 0;
while(str[i] != '\0')
{
        putchar(str[i]);
        i++;
}
```

4.9 OPERATIONS ON STRINGS

There are different operations performed on strings that are discussed as follows.

(a) A string is initialized as:

```
char str[ ] = "Welcome";
Or
char str[ ] = {'W','e','l','c','o','m','e','\0'};
```

Here, in the first declaration, '\0' is not necessary. C++ compiler inserts the NULL character (\0) automatically.

(b) Length of a string can be estimated by just finding the subscript value of the NULL character (\0) in the character array where the string is stored. So searching operation is performed for the NULL character (\0) in the string and the subscript value of the NULL character (\0) is the required length of the string.

A C++ program to find out the length of a string is given below.

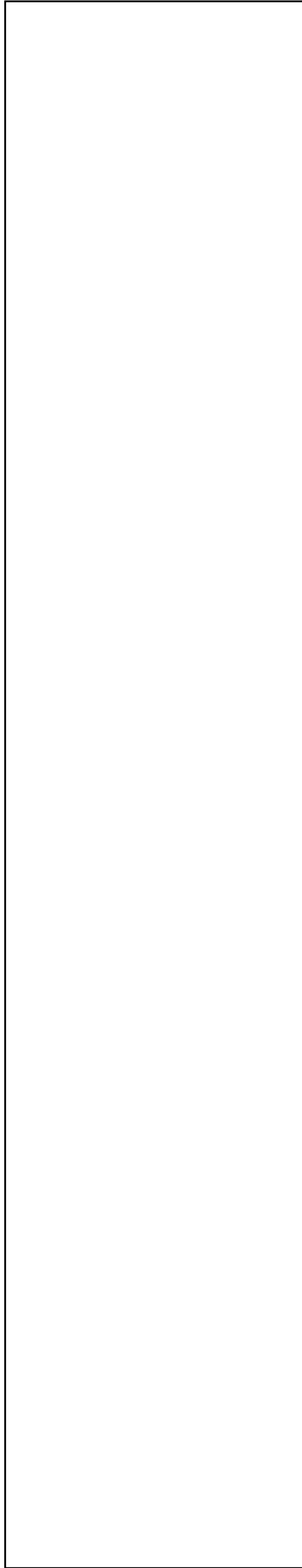
```
#include <iostream.h>
#include <conio.h>

int main()
{

        char ch, str[30];
        int slen = 0;
        clrscr();
```

```
    cout << "\nEnter a string =";
    gets(str);
    while(str[slen]! = '\0')
    {
        slen++;
    }
    cout << "\n The length of the string is ="<<slen;
    getch( );
    return 0;
}
```

(a) To concatenate one string at the end of another string, we have to find out the subscript value of the NULL character that is stored in the string where concatenation will be performed. Then we have to assign each character of the string that is to be concatenated to the other string in consecutive position starting from the estimated subscript value.

A C++ program to concatenate a string at the end of an another string is given below

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

int main()
{
    char str1[30], str2[30];
    int slen = 0, i = 0;
    clrscr( );
    cout << "\nEnter the first string =";
    gets(str1);
    cout << "\nEnter the second string =";
    gets(str2);
    while(str1[slen]! = '\0')
    {
        slen++;
    }
    while(str2[i]! = '\0')
    {
        str1[slen] = str2[i];
        slen++;
```



```
        i++;
    }
    str1[slen] = '\0';
    cout << "\n After concatenation the first string is=";
    puts(str1);
    getch( );
    return 0;
}
```

(c) To copy a string to an another string we have to just assign each character of the first string to the second string in consecutive subscript value positions starting from 0 to the subscript value where the NULL character(\0) is stored in the first string.

A C++ program to copy one string to another string is given below.

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

int main()
{
    char str1[30], str2[30];
    int i = 0;
    clrscr();
    cout <<"\nEnter the first string =";
    gets(str1);
    cout <<"\nEnter the second string =";
    gets(str2);
    while(str2[i]! = '\0')
    {
        str1[i] = str2[i];
        i++;
    }
    str1[i] = '\0';
    cout <<"\nAfter copy the first string is =";
    puts(str1);
    getch( );
    return 0;
}
```

Example 4.11:

Write a C++ program to search and find out the number of occurrences of a specific character in a string.

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

int main()
{

   char str[30] ;
   char ch ;
   int i = 0, chcount = 0 ;
   clrscr( );

   cout <<"\nEnter a string = ";
   gets(str) ;
   cout <<"\nEnter the character to be searched= " ;
   cin >> ch ;

   while(str[i] != '\0' )
   {
       if( str[i] == tolower(ch)  || str[i] == toupper(ch) )
           chcount++;
       i++;
   }

   if(chcount == 0 )
       cout <<"\n the character, ”<< ch<< “is not present in the
string";
   else
       cout<<"\nThe character is present in the string,”<<chcount<<
“no. of times";

   getch( );
   return 0;
}
```

Example 4.12:

Write a C++ program to count the number of vowels present in a string.

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

int main()
{
        char str[ 30 ];
        int i = 0,vcount = 0;
        clrscr();
        cout <<"\nEnter a string =";
        gets( str );
        while( str[i] != '\0' )
        {
                switch( str[i] )
                {
                        case 'a':
                        case 'A':
                        case 'e':
                        case 'E':
                        case 'i':
                        case 'I':
                        case 'o':
                        case 'O':
                        case 'u':
                        case 'U': vcount++;
                }
                i++;
        }

        if ( vcount == 0 )
          cout <<"\n No vowel present in the string";
        else
          cout <<"\nThe number of vowel present in the string is =" << vcount;

        getch( );
        return 0;
}
```

4.10 ARRAY ON STRINGS

Till now, we have learnt to read and display single strings by using one dimensional character arrays. But to read multiple strings, we require two dimensional character arrays where the first subscript value of the arrays indicates the total number of strings and the second subscript value indicate the maximum length of each strings. This is also called as array of strings. For example, let us consider a two dimensional character array Multi Strn[20][40]. Now Multi_Strn[20][40] can be used to read 20 strings where maximum length of each string can be40 .

A C++ program is shown below where N number of employees' names is read and displayed.

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

int main( )
{
        char Emp_names[50][100];
        int N, i;
        clrscr( );

        cout << "\n Enter the total number of names = ";
        cin >> N;

        if(N > 50)
        {
                cout <<"\n Maximum 50 names is possible";
        }
        else
        {
                cout<<"\n Enter names of ”<<N<<“number of
employees::”;
                for(i = 0 ; i < N ; i++)
                {
                        cout <<"\n Enter ”<<i+1<< “th name = ";
                        gets(Emp_names[i]);
                }
```

```
            cout <<"\n The list of employee names is:\n";
            for(i = 0 ; i < N ; i++)
            {
                    puts(Emp_names[i]);
                    cout<<"\n";
            }
    }
    getch( );
    return 0;
}
```

In the above program, Emp_names[50][100] is declared as two dimensional character array to store maximum 50 number of employee names. Here Emp_names[i] point to the ith employee name.

4.11 STRING LIBRARY FUNCTIONS

Some of useful library functions on strings and their functionalities are given in the following table (TABLE 4.1).

Table 4.1: Table For String Library Functions And Their Functionalities

String Library Function	Functionality
strlen(strn)	Returns the length of the string strn
strcpy(strn1,strn2)	Copies the string strn2 to the string strn1
strncpy(strn1,strn2,N)	Copies first N characters of the string strn2 to the string strn1
strcat(strn1,strn2)	Concatenate the string strn2 at the end of the string strn1
strcmp(strn1,strn2)	Compares the two strings strn1 and strn2. If it returns 0 then strn1 and strn2 are equal. If it returns a positive value then strn1 is greater than strn2. If it returns a negative value then strn2 is greater than the
strncmp(strn1,strn2,N)	Compares first n characters of two strings strn1 and strn2
strcmpi(strn1,strn2)	Compares two strings strn1 and strn2 without regard to case
strlwr(strn)	Converts the string strn to lowercase
strupr(strn)	Converts the string strn to uppercase
strdup(strn)	Returns a pointer to a string that is duplicate of the string strn
strchr(strn,chr)	Returns a pointer to the first occurrence of the character chr in the string strn. If chr is not available in strn then it returns NULL.
strrchr(strn,chr)	Returns a pointer the last occurrence of the character chr in the string strn. If chr is not available in strn then it returns NULL.
strstr(strn1,strn2)	Returns a pointer to the first occurrence of a string strn2 in the string strn1. If strn2 is not available in strn1 then it returns NULL.
strrev(strn)	Reverses the string strn
strset(strn,chr)	Sets all characters of the string strn to the character chr
strnset(strn,chr,N)	Sets first N characters of the string strn to the character chr

STOP TO CONSIDER

It is necessary to include the header file 'string.h' to use mentioned string library functions.

Now consider the following programming statements.

```
char strn1[ ] = "Gauhati University";
char strn2[ ] = "Welcome to IDOL";
int slen;
slen = strlen(strn1);
cout <<"\n Length of the string stored in strn1 is ="<< slen;
slen = strlen(strn2);
cout << "\n Length of the string stored in strn2 is ="<< slen;
strcpy(strn1,strn2);
cout << "\n String stored in strn1 is =";
puts(strn1);
cout <<"\n String stored in strn2 is =";
puts(strn2);
cout <<"\n"<< strcmp(strn1,strn2);
```

Now the output of the above programming statements is:

```
Length of the string stored in strn1 is = 18
Length of the string stored in strn2 is = 15
String stored in strn1 is = Welcome to IDOL
String stored in strn2 is = Welcome to IDOL
0
```

The first line of the output gives the length of the string "Gauhati University" stored in the character array strn1. The second line of the output gives the length of the string "Welcome to IDOL" stored in the character array strn2. To estimate these lengths, the string library function strlen( ) is used.

In the above programming statements, string library function strcpy( ) is used to copy the string stored in strn2 to the string in strn1. As a result, string "Welcome to IDOL" replaces the string "Gauhati University" in strn1. Due to this, the third line of the output displays the string stored in strn1 which is "Welcome to IDOL". The fourth

line of the output displays the string stored in strn2 which is also "Welcome to IDOL".

In the above programming statements, string library function strcmp( ) is also used to compare the strings stored in strn1 and strn2. Now at this moment, both strn1 and strn2 store the same string. So strcmp(strn1,strn2) returns 0 and as a result the fifth line of the output provide 0.

STOP TO CONSIDER

In case of multi word strings, the blank spaces between two words are also considered as characters in estimation of lengths of the strings. For example, the length of the string "Gauhati University" is 18.

Example 4.13: Write a C++ program to read N number of employee names and perform sorting operation using Bubble sort technique to arrange these names in alphabetical order. Use string library functions as required in the program.

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

int main()
{
        char Emp_names[50][100], temp[100];
        int N, i, j;
        clrscr();

        cout <<"\n Enter the total number of names = ";
        cin >> N;

        if(N > 50)
        {
                cout <<"\n Maximum 50 names is possible";
        }
        else
        {
                cout<<"\nEnter names of   "<<N<< "number of
employees::";
                for(i = 0 ; i < N ; i++)
```

```
        {
            cout << "\n Enter  ”<<i+1<< “th name = ";
            gets(Emp_names[i]);
        }
        cout <<"\n The list of employee names before sorting
is:\n";
        for(i = 0 ; i < N ; i++)
        {
            puts(Emp_names[i]);
            cout <<"\n";
        }

        for(i = 0 ; i < N-1 ; i++)
        {
            for(j = 0 ; j < N-1-i ; j++)
            {
                if(strcmp(Emp_names[j] ,Emp_names[j+1]) > 0)
                {
                    strcpy(temp , Emp_names[j]);
                    strcpy(Emp_names[j] , Emp_names[j+1]);
                    strcpy(Emp_names[j+1]          ,
temp);
                }
            }
        }
        cout << "\n The list of names after sorting is:\n";
        for(i = 0 ; i < N ; i++)
        {
            puts(Emp_names[i]);
            cout <<"\n";
        }
    }
    getch( );
    return 0;
}
```

CHECK YOUR PROGRESS

3. Multiple choices

(a) In C++ programming, a string is terminated by____.

(i) '\0'
(ii) '\n'
(iii) A blank
(iv) None of the above

(b) A string is stored in a

(i) Character array
(ii) Integer array
(iii) Both (i) and (ii)
(iv) None of the above

(c) A string is initialized as

(i) char st1[ ] = "IDOL";
(ii) char st1 = "IDOL";
(iii) char st1[ ] = {'I','D','O','L','\0'};
(iv) Both (i) and (iii)

(d) char name[20] = "Welcome to IDOL" ;
name[7] = '\0';
cout << name;

The output of the above statements is

(i) Welcome
(ii) Welcome t
(iii) IDOL
(iv) None of the above

(e) Which one of the following is appropriate for reading a multi word string ?

(i) cin
(ii) puts( )
(iii) gets( )
(iv) Both (ii) and (iii)

(f) If strcmp(s1,s2) returns -12 then it means

(i) s1 and s2 are equal strings
(ii) s1 is greater than s2
(iii) s2 is greater than s1
(iv) None of the above

(g) char str[20] = "Welcome";
for(int i = strlen(str) – 1 ; i>=0 ; i--)
cout << str[i];
The output of the above statements is
(i) Welcome
(ii) emocleW
(iii) Welcom
(iv) Error message from compiler

(h) strcpy(s1,s2);
The above statement means
(i) Copies the string s2 to the string s1
(ii) Copies the string s1 to the string s2
(iii) Copies the first n characters of the string s2 to the string s1
(iv) None of the above

(i) Which one of the following statement can be used to display multi-word string stored in the character array 'STR'?
(i) gets(STR);
(ii) cout<<STR;
(iii) puts(STR);
(iv) Both B and C

(j) The maximum length of any string that can be stored in a character array with array size 100 is_____.
(i) 90
(ii) 99
(iii) 100
(iv) 101

(k) Which of the following statement is a correct way to initialize a string?
(i) char string1[ ] = "GUIDOL";

(ii) char string1[ ] =
{‘G’,’U’,’I’,’D’,’O’,’L’,’\0’};
(iii) char string1[ ] = {G,U,I,D,O,L,\0};
(iv) Both A and B

(l) char Arr_string[50][300];
The maximum number of strings that can be stored in the array 'Arr_string' is ____.
(i) 50
(ii) 300
(iii) 299
(iv) None of the above

(m) char Arr_string[50][300];
The maximum length of each string that can be stored in the array 'Arr_string' is ____.
(i) 49
(ii) 50
(iii) 299
(iv) 300

(n) Which of the following header file has to be included in a C++ program to use string library functions?
(i) string.h
(ii) stdio.h
(iii) conio.h
(iv) stringio.h

(o) strncpy(str1 , str2 , N);
The above statement means
(i) Copies the string str2 to the string str1
(ii) Copies the string str1 to the string str2
(iii) Copies first N characters of the string ‘str2’ to the string ‘str1’
(iv) Copies first N characters of the string ‘str1’ to the string ‘str2’

(p) strcmpi(str1 , str2);
The above statement means
(i) Compares the two strings ‘str1’ and ‘str2’

(ii) Compares first i characters of two strings ‘str1’ and ‘str2’
(iii) Concatenate the string ‘str2’ at the end of the string ‘str1’
(iv) Compares two strings ‘str1’ and ‘str2’ without regard to case

(q) Which of the following statement reverses the string ‘str’?
(i) strdup(str);
(ii) strrev(str);
(iii) strrchr(str,chr);
(iv) strset(str,chr);

(r) Which of the following statement sets all characters of the string ‘str’ to the character ‘ch’?
(i) strrchr(str , ch);
(ii) strstr(str , ch);
(iii) strset(str , ch);
(iv) strset(str);

4. State whether true or false

(a) The length of a string is equal to the subscript value of the position where the NULL character is stored in the character array.
(b) Strings cannot be initialized.
(c) A string with multiple words cannot be entered by cin.
(d) strlwr( ) converts a string to its lower case.
(e) strcat(str1,str2) concatenates the string str1 at the end of the string str2.

4.12 SUMMING UP

The summery of this unit is given as follows:

- An array is a collection of similar type of data which are stored in consecutive memory locations.

- The declaration of an array has three parts, (a) type of the variable, (b)array name and (c) within brackets([ ]) the size of the array means how many elements can be stored in the array.
- Initialization of an one dimensional array can be implemented as follows int arr[5] ={ 12,23,34,45,56};
- Three ways of initializing a two dimensional array are given as follows.

  - int arrtwo[2][3]= {

    {2, 18 , 7 },

    {43, 91, 1}

    };

  - int arrtwo[2][3] = { 2 , 18 , 7, 43, 91 , 1 };
  - int arrtwo[ ][3] = { 2 , 18 , 7, 43 , 91 , 1 };

- Insertion and searching operation on an array can be performed with the name of the array and the subscript values.
- String is a collection of some characters stored in a character array.
- A string is always terminated by \0 which is called NULL character.
- The standard input stream object, 'cin' is used to input a string in C++. But 'cin' is not capable of entering multi word strings. 'gets( )' function can be used to input multi word strings.
- Multiple strings can be stored using two dimensional character array where the first subscript value of the array indicates the total number of strings and the second subscript value indicate the maximum length of each strings. This is also referred as array of strings.
- Some useful library functions on strings are strlen( ), strcpy( ), strcat( ), strlwr ( ), strupr( ), strcmp( ) etc.

We have to include the header file 'string.h' to use these functions.

4.13 ANSWER TO CHECK YOUR PROGRESS

1. (a) (i), (b) (iv), (c) (iv), (d) (iii), (e) (ii), (f) (iv), (g) (iv), (h) (ii), (i) (ii), (j) (iii), (k) (iv), (l) (i), (m) (iii), (n) (ii), (o) (iv)

2. (a) false, (b) false , (c) true , (d) false, (e) false

3. (a) (i), (b) (i), (c) (iv), (d) (i), (e) (iii), (f) (iii), (g) (ii), (h) (i), (i) (iv), (j) (ii), (k) (iv), (l) (i), (m) (iii), (n) (i), (o) (iii), (p) (iv), (q) (ii), (r) (iii)

4. (a) true (b) false (c) true (d) true (e) false

4.14 POSSIBLE QUESTIONS

1. Define array. Explain different types of array available in C++ programming. Give suitable examples.
2. Why concept of array is very important in programming?
3. Write a C++ program to construct a new array by merging two sorted integer array where the elements in the new array will also be sorted.
4. Write a C++ program to input a new element into an array at the position entered by the user.
5. Write a C++ program to calculate the summation of two integer arrays.
6. Write a C++ program to find out the number of even and odd numbers present in an integer array.
7. Write a C++ program to calculate the summation of all the even and odd numbers present in an integer array.

8. Write a C++ program to estimate the transpose of an input matrix.
9. Define string. Write down the differences between string and a character array.
10. Write a C++ program to check a string is palindrome or not.
11. Write a C++ program to reverse a string without using string library functions.
12. Write a C++ program to replace a particular character in a string by a character entered by the user.

4.15 REFERENCES AND SUGGESTED READINGS

1) Venugopal, K. R., Rajkumar, Ravishankar, T. *Mastering C++*. Tata McGraw-Hill Education, 2001.
2) Balagurusamy, E. *Object Oriented Programming with C++*. Tata McGraw-Hill, 2006

UNIT 5: POINTERS AND REFERENCE VARIABLES IN C++

Unit Structure:

5.1 Introduction
5.2 Unit Objectives
5.3 Definition of Pointer
5.4 Pointer to Array
 5.4.1 Pointer to One Dimensional Array
 5.4.2 Pointer to Two Dimensional arrays
 5.4.3 Pointer to Strings
 5.4.4 Array of Pointers
5.5 Pointer and Function
 5.5.1 Passing Memory Address to Function
 5.5.2 Passing Array to Function through Pointers
 5.5.3 Pointer to Function
5.6 Dynamic Memory Allocation
 5.6.1 Dynamic Memory Allocation Using 'new'
 5.6.2 Memory De-allocation Using 'delete'
5.7 Reference Variable
5.8 Summing Up
5.9 Answers to Check Your Progress
5.10 Possible Questions
5.11 References and Suggested Readings

5.1 INTRODUCTION

In earlier units, we have learnt to declare different types of variables to store different types of data. But in some situations, we are required to store and access the addresses of declared variables. So in such cases, we can declare pointer variables.

In C++ programming, a different type of variable is introduced that is called as reference variable.

5.2 OBJECTIVES

After reading this unit, you are expected to be able to learn:

- What is Pointer?
- About Pointer arithmetic
- Relationship of Pointer and Array
- Use of Pointer in Function
- Use of Pointer in Structure
- About dynamic memory allocation
- What is Reference variable?

5.3 DEFINITION OF POINTER

Pointer is a variable which can store the address of another variable of same type. Syntax of declaring a pointer variable is given as follows:

```
Data type  * variable_Name;
For example:  int  *ptr;
```

Here ptr is a pointer variable with data type int which means that ptr can store the memory address of any integer variable. Now ptr is a single pointer variable. We can also declare a double pointer variable which can store the memory address of any pointer variable of same data type i.e. pointer to pointer. For example:

```
int *ptr , **dptr;
```

Here dptr is a double pointer with data type int and it can store the memory address of any single pointer with data type int as shown below.

```
dptr  =  &ptr ;
```

Here dptr stores the address of ptr. The ‘&’ operator used in the above statements is ‘address of’ operator in C++ programming. The expression &ptr will give the memory address of ptr. Now let us consider the following programming statements:

```
int  p, *ptr, **ptr ;
p = 10;
```

```
ptr = &p;
dptr = &ptr;
```

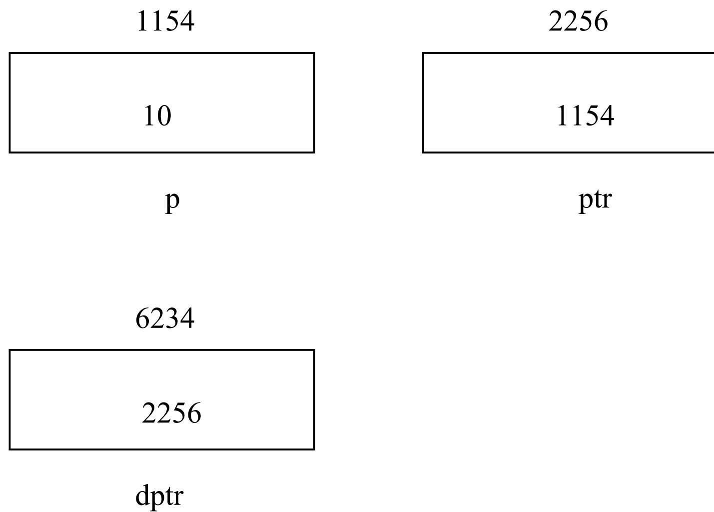


Fig. 5.1

In fig. 5.1 three diagrammatic representations of the three variables p, ptr and dptr are presented where the memory addresses of p, ptr and dptr are assumed to be respectively 1154, 2256 and 6234. Here p is an integer variable which contains an integer value 10. ptr is a single pointer variable with data type int which contains the address of the integer variable p. dptr is a double pointer variable with data type int which contains the address of the memory address of the single pointer variable ptr.

Now what will be the output of the following programming statements?

```
cout <<“\n The address of p =”<< &p;
cout <<“\n Value in p = ”<< p;
cout <<“\n The address of ptr = ”<< &ptr;
cout <<“\n Value in ptr = ”, ptr;
cout <<“\n The address of dptr = ”<< &dptr;
cout << “\n Value in p = ”<< dptr;
cout << “\n Value in p =”<< *ptr;
cout << “\n Value in ptr = ”<< *dptr;
```

Outputs of the above statements are:

The address of p = 1154
Value in p = 10
The address of ptr = 2256
Value in ptr = 1154
The address of dptr = 6234
Value in dptr = 2256
Value in p = 10
Value in ptr = 1154

Here the ‘ * ’ operator used in the above statements is ‘value of ’ operator in C++ programming. So using this operator we can access the value stored in some memory address.

STOP TO CONSIDER

In memory, each byte of memory locations is identified by the CPU with a unique code which is called as the physical address of the particular memory location. A pointer variable is used to store the physical address of the first byte of memory locations allocated for a variable of same type.

5.4 POINTER TO ARRAY

We have already learnt that the elements of an array are stored in contiguous memory locations as shown in fig. 5.2. In case of array, using the name of the array we will get the base address of the array. Now using pointer, we can use this base address to access the array elements.

	0	1	2	3	4	5	6
Arr	43	55	123	76	31	90	89
	6010	6012	6014	6016	6018	6020	6022

Fig. 5.2

5.4.1 Pointer to One Dimensional Array

In fig. 5.2, Arr is a one dimensional integer array of size 7 with base address 6010. Now consider the following programming statements.

```
int  *ptr1 , *ptr2 ;
int Arr[7] ;
ptr1 = Arr ;
ptr2 = &Arr[0] ;
cout <<"\n Base address of Arr = "<< Arr;
cout <<"\n ptr1 = "<< ptr1;
cout <<"\n ptr2 = "<< ptr2;
```

Output of the above statements:

Base address of Arr = 6010

ptr1= 6010

ptr2= 6010

Here ptr1 and ptr2 are two integer pointer both storing address of the same memory location that is the base address of array Arr[ ]. So using '&' operator we can get the address of the element with subscript value 0 in an array which is the base address of the array. So *Arr or *ptr1 or *ptr2 will refer to the element with subscript value 0 in array Arr that is 43.

Now we have the base address of the array and to access all the elements of the array, we can use pointer arithmetic.

There are four arithmetic operators that can be used on pointers that are ++, --, +, and -

Now after the operation ptr1++, ptr1 will point to the location 6012 because each time ptr is incremented, it will point to the next integer location which is 2 bytes next to the current location. So now *ptr1 will refer the integer value 55 which is the element with subscript value 1 in the array Arr. If ptr contains the base address of the array

Arr then ptr + i will point the element with subscript value i in the array Arr.

*(Arr + 0) and *Arr and Arr[0] refer the first element in the array Arr. So *(Arr + i) and Ar[i] refers to the $(i+1)^{th}$ element in the array Arr. Actually Arr[i] is internally converted to *(Arr + i) by the C++ compiler.

Pointers are also can be compared by using relational operators, such as = =, <, and >.

5.4.2 Pointer to Two Dimensional arrays

In case of two dimensional arrays, there are two subscripts values to refer an element. Consider the following programming statement:

```
int  A[4][4] ;
int  * ptr;
ptr = A;
```

Here A[ ][ ] is a two dimensional array which can store at most (4 × 4) = 16 integer elements. Let us consider the following diagrammatic representation of the two dimensional array A[ ][ ].

	0	1	2	3
0	20	25	8	12
1	24	32	67	54
2	43	65	71	59
3	89	76	21	41

Fig. 5.3

Let us the memory address of A[0][0] is 2014 which is the base address of A. In case of a two dimensional array like A, the 0th element of the array A is a one dimensional array. So *A or *(A+0) will give the memory address of first element of the first row of A that is the base address of the 0th one dimensional array in A. In this way, *(A+1) will give the base address of 1st one dimensional array. So *(A+i) will give the base address of ith one dimensional array.

We know that A[i][j] will refer the element from i^{th} row and j^{th} column. Using *(A+i) , we can refer the ith row, so using *(*(A+i)+j) we can refer the particular element from i^{th} row and j^{th}

column in A. Now with this concept we can have the following statements.

- A[0][0] and *(*(A)) will refer to the same element.
- A[i][j] and *(*(A+i)+j) will refer to the same element.
- A[i][j] and *(A[i] + j) will refer to the same element.
- A[i][j] and *((*A) + (i * col_no + j)) will refer to the same element, where col_no is the total number of columns in A.

Now from the above programming statements, ptr is a pointer variable and it stores the base address of A. So *ptr will give the element referred by A[0][0]. Now using *(ptr +2*4+3) , we can access the element referred by A[2][3] where 4 is the total number of columns in A as shown in fig. 5.3. In this way we can access A[i][j] by using *(ptr + i*col_no + j), where col_no is the total number of columns that is 4 in case of A.

5.4.3 Pointer to Strings

A character pointer can be used to assign the address of a string stored in some memory location. For example: consider the following programming statements:

```
char  *st1 = "Welcome to GUIDOL";
char  *st2;
char  str[ ] = "Welcome to Gauhati University";
st2 = str;
```

Here st1 is a character pointer which is used to assign the address of the string “Welcome to GUIDOL” stored in some memory location. Again str is a character array which is initialized with a string “Welcome to Gauhati University” and a character pointer st2 is used to assign the address of the string stored in str.

STOP TO CONSIDER

We know that a string is stored in a character array. Now address of a string means the physical address of the first character of the string that is stored in the character array with subscript value 0.

5.4.4 Array of Pointers

An array of pointers is an array which stores a collection of same type of pointers. For example: Let us consider the following statements:

```
int  *Aptr[6];
int A[ ] = {9 , 5 , 8 , 11 , 90 , 32};
Aptr[0] = &A[0];
Aptr[1] = &A[1];
Aptr[2] = &A[2];
Aptr[3] = &A[3];
Aptr[4] = &A[4];
Aptr[5] = &A[5];
```

Here Aptr[ ] is an array of integer pointers with size 6 that means it can store the memory addresses of 6 integer data. In the above statements, Aptr[ ] store the addresses of 6 integer data which are stored in the integer array A.

Now let us consider the following programming statement:

```
char  *strarr[ ] ={
                "Gauhati University",
                "GUIDOL",
                "ASSAM",
                "INDIA"
        };
```

Here strarr[ ] is a array of character pointers and it is used to store the base addresses of four strings. So strarr[0] will store the base address of the string “Gauhati University”. In this way, strarr[1], strarr[2] and strarr[3] will store the base addresses of the strings “GUIDOL” , “ASSAM” and “INDIA” respectively.

CHECK YOUR PROGRESS

1. Multiple choices

(a)
```
int a , *b;
a = 10;
b = &a;
cout << a+*b;
```
The output of the above statements is_____.

(i) 10
(ii) 20
(iii) 21
(iv) Error message

(b) If Arr is a two dimensional array then Arr[i] gives______.

(i) Data stored in Arr[i][0]
(ii) Address of Arr[i][0]
(iii) Address of Arr[0][i]
(iv) Garbage value

(c) Which of the following statement is equivalent to &Arr[0][0] where Arr is a two dimensional character array?

(i) Arr
(ii) Arr[0]
(iii) Arr[ ][0]
(iv) Both (i) and (ii)

(d) Which of the following statement is equivalent to *(*(Arr+i)+j) where Arr is a two dimensional array?

(i) Arr[i][j]
(ii) *(A[i] + j)
(iii) *((*A) + (i * C + j)), where C is the total number of columns in Arr.
(iv) All of the above

(e) Let us consider the following programming statements.

```
char  *S[ ] ={
                "Gauhati University",
                "IDOL",
                "Computer Science",
                "M.Sc.IT"
           };
puts(S[2]);
```

The output of the above statements is_____.

(i) Computer Science
(ii) IDOL
(iii) M.Sc.IT
(iv) None of the above

(f) Which of the following will refer the element that is referred by A[i][j]?

(i) *(*(A+i)+j)
(ii) *A+i+j
(iii) *(*A+i+j)
(iv) None of the above

2. Fill-in the blanks

(a) _____ operator is used to access the value stored in a variable pointed by a pointer.

(b) The base address of a one dimensional array Arr[50] can be accessed by ______.

(c) If ptr is a pointer variable which points to a character array where the base address of the array is 1133 then ptr++ will point to the memory location _______.

(d) An array of pointers can store_______.

5.5 POINTER AND FUNCTION

5.5.1 Passing Memory Address to Function

In unit 5, call by value or pass by value in function is already discussed where data is directly passed to functions. In this section we are going to discuss how pointers can be used to pass data to

functions. It is also referred as call by address or call by pointer or pass by address or pass by pointer.

In case of pass by address, the memory addresses of the actual parameters are passed to functions in the function calling statements and the formal parameters are the pointer variables that can store these addresses.

A C++ program to swap two integer numbers that are stored in two variables by defining a user defined function is shown below. In this program pass by address is used for argument passing to the user defined function.

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

void swap( int *, int *);
int main( )
{
        int num1 , num2;
        clrscr();
        cout <<"\n Enter the first number =";
        cin >> num1;
        cout <<"\n Enter the second number =";
        cin >> num2;
        cout << "\n Before swapping the input numbers are =";
        cout <<"\n First number = "<<num1;
        cout <<"\n Second number = "<<num2;
        swap(&num1 , &num2);
        cout << "\n After swapping the input numbers are =";
        cout << "\n First number = "<<num1;
        cout << "\n Second number = "<<num2;
        getch( );
        return 0;
}

void swap(int *n1 , int *n2)
{
        int temp;
        temp = *n1;
        *n1 = *n2;
```

```
        *n2 = temp;
}
```

In the above program, swap( ) is the user defined function whose functionality is to swap two numbers that are read in the function main( ). Here num1 and num2 are the actual parameters and n1 and n2 are the formal parameters. In main( ), swap( ) is called by passing the memory addresses of num1 and num2 using &("address of") operator. These addresses are stored in the formal parameters n1 and n2 respectively. Finally, in swap( ), swapping of the two numbers is performed by using *("value of") operator and a variable "temp".

5.5.2 Passing Array to Function through Pointers

In case of array, the base address of the array can be passed to a function. Now using pointer variable and pointer arithmetic we can access the array elements inside the function as shown below with the programming statements.

```
int main()
{
        int  Aone[40], Atwo[40][40] ;
        int  n,m ;
        clrscr();
        cout <<"\n Enter the number of elements in the array Aone =";
        cin >>n;
        input_one(Aone,n);
        cout <<"\n Display the elements in the array Aone\n";
        display_one(Aone,n);
        cout <<"\n Enter the number of rows in the array Atwo =";
        cin >>n;
        cout <<"\n Enter the number of columns in the array Atwo =";
        cin >>m;

        input_two(Atwo, n, m);
        cout <<"\n Display the elements in the array Atwo\n";
        display_two(Atwo,n,m);
        getch( );
```

```
        return 0;
}

void input_one(int *ptrone, int n)
{
        int   i;
        cout <<"\n Enter  "<<n<< "elements into the array";
        for(i = 0 ; i < n ; i++)
        {
                cout <<"\nEnter  "<<i+1<< "th element =";
                cin >> ptrone + i;
        }
}

void input_two(int *ptrtwo, int n, int  m)
{
          int   i , j ;
          for(i = 0 ; i < n ; i++)
          {
                for(j = 0 ; j < m ; j++)
                {
                    cout<<"\nEnter  ("<<i<<  ","<<j<<")  th  element
=";

                    cin >> ptrone + i * 40 + j;
                 }
          }
}

void input_display(int *ptrone , int n)
{
          int   i;
          for(i = 0 ; i < n ; i++)
          {
                cout <<"\t"<< *(ptrone+i);

          }
}

void display_two(int *ptrtwo, int n, int  m)
{
         int   i , j ;
```

```
        for(i = 0 ; i < n ; i++)
        {
                for(j = 0 ; j < m ; j++)
                {
                        cout <<"\t"<< *(ptrtwo + i * 40 + j);
                }

        }
}
```

5.5.3 Pointer to Function

A pointer to function or a function pointer is a pointer that stores the starting address of a function. It means that a function pointer points to the starting address of an executable code and it does not point to any data. The syntax to declare a pointer to function in C++ is presented below.

```
Return_DataType  (* Pointer_Name)( Argument list);
```

Here Return_DataType is the return type of a function which will be pointed by the function pointer, Pointer_Name and if the function has arguments then argument list will be provided as shown in the syntax.

The syntax to point a function by using a function pointer is stated below.

```
Pointer_Name = Function_Name;
```

Here Pointer_Name is a function pointer and Function_Name is a name of a function.

Let us consider the following C++ program to understand the use of function pointer.

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

int Is_Prime(int);  // Function Declaration
int main()
{
```

```
  int Num, flag;
  int (*Fptr)(int);  // Declaration of the function pointer, Fptr
  clrscr();
  cout<<"\n Enter a number=";
  cin>>Num;
  Fptr = Is_Prime;   //The starting address of Is_Prime() is assigned to
Fptr
  flag = Fptr(Num); // Calling of Is_Prime() using Fptr
  if(flag == 1)
  {
    cout<<"\n"<<Num<<" is a prime number";
  }
  else
  {
    cout<<"\n"<<Num<<" is not a prime number";
  }
  getch();
  return 0;
}

int Is_Prime(int N)   // Function definition
{
  int i;
  if(N==1)
   return 0;
  else
  {
    for(i = 2; i <= N/2 ; i++)
    {
          if( N%i == 0)
            return 0;
    }
    return 1;
  }
}
```

In the above program, Fptr is a pointer to function and it is used to hold the starting address of the function Is_Prime(). The output of the above program is stated below.

```
Enter a number= 7
7 is a prime number
```

A pointer to function can also be passed as parameter to other functions in C++. Let us consider the following C++ program to learn about it.

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

int Is_Prime(int);  // Function Declaration
void Check_Prime(int (*)(int)); //Function Declaration
int main()
{
  clrscr();
  Check_Prime(Is_Prime); // Function pointer as parameter
  getch();
  return 0;
}
void Check_Prime(int (*Fptr)(int)) //Function pointer Fptr as formal parameter
{
  int Num,i, flag;
  cout<<"\n Enter a Number=";
  cin>>Num;
  flag = Fptr(Num);
  if(flag == 1)
  {
    cout<<"\n"<<Num<<" is a prime number";
  }
  else
  {
    cout<<"\n"<<Num<<" is not a prime number";
  }
}
int Is_Prime(int N)   // Function definition
{
  int i;
  if(N==1)
   return 0;
  else
  {
```

```
    for(i = 2; i <= N/2 ; i++)
    {
         if( N%i == 0)
           return 0;
    }
    return 1;
  }
}
```

In the above program, a function pointer is passed as parameter to the function, Check_Prime(). Here the function pointer, Fptr points to the function, Is_Prime(). The output of the program is stated below.

```
Enter a Number= 12
12 is not a prime number
```

5.6 DYNAMIC MEMORY ALLOCATION

Memory allocation during program execution that is at the runtime of a program is called dynamic memory allocation. So when we need to allocate memory at runtime then dynamic memory allocation is performed.

5.6.1 Dynamic Memory Allocation Using 'new'

In C++, memory allocation at runtime that is dynamic memory allocation can be performed by using the operator "new".

The syntax to allocate memory by using "new" is:

```
Data_Type  *PTR;
PTR = new Data_Type;
```

Here Data_Type is any valid data type. "new" operator will allocate required memory to store data of type Data_Type and return the memory address. The pointer PTR will store this address.

For example, let us consider the following programming statements:

```
int   *PTR;
PTR = new int;
```

Here, PTR is an integer pointer that stores the memory address of the allocated memory locations by "new" operator. An integer data can be stored in the allocated memory space.

"new" operator can also be used to initialize the allocated memory. The syntax for this purpose is shown below.

```
PTR = new Data_Type(Value);
```

For example, let us consider the following programming statements:

```
char *PTR = new char( 'S');
cout<< *PTR;
```

Here, "new" is used to allocate memory to store a character data and it is initialized to 'S'. So, the output of the above statements is S.

Following syntax can be used to allocate memory for a one-dimensional array.

```
PTR = new Data-Type[Array_Size];
```

Here Array_Size is the number of array elements that can be stored in the allocated memory spaces.

For example, let us consider the following programming statements:

```
int  *PTR;
PTR  =  new int[10];
```

Similarly, following statements can be used to allocate memory for a two and a three dimensional array respectively.

```
int  *PTR1, *PTR2;
PTR1  =  new  int[4][12];      // Memory allocation for two
dimensional array
```

PTR2 = new int[7][8][9];   // Memory allocation for three dimensional array

STOP TO CONSIDER

If sufficient memory is not available for allocation then “new” returns a null pointer.

5.6.2 Memory Deallocation Using ‘delete’

In C++, memory deallocation can be performed by using the operator “delete”.
The syntax to deallocate memory by using “delete” is:

```
delete PTR;
```

Here, PTR is the pointer that stores the memory address of a data object created with “new” operator.

Following statement can be used to release memory of an array that is created by “new” operator.

```
delete [ ] PTR;
```

Here PTR is a pointer that points to an array created by “new” operator.

Program: Write a C++ program to show the use of ‘new’ and ‘delete’ operator.

```
#include <iostream.h>

using namespace std;

int main()
{

   int* ptr_int;          // declare an int pointer

   float* ptr_float;      // declare a float pointer


   ptr_int = new int;     // dynamically allocate memory
   ptr_float = new float; // dynamically allocate memory
```

```
    // assigning value to the memory
    *ptr_int = 17;
    *ptr_float = 99.99;

    cout<< "The value of ptr_int is: "<<*ptr_int<<"\n";
    cout<<"The value of ptr_float is: " <<*ptr_float;


    // deallocate the memory
    delete ptr_int;
    delete ptr_float;

   return 0;
}
```

5.7 REFERENCE VARIABLE

In C++, reference variable is a variable that offer an alternative name to an already declared variable. Use of reference variables in a C++ program is similar with the use of the value variables. Changes made to a reference variable are also reflected in the variable that is bound to the reference variable. So it can be realized that reference variable has the power of pointer variable. But when a reference variable is bound to a variable then at later stage, this binding cannot be changed.

The syntax to declare a reference variable is presented as follows.

```
Data_Type  & Reference-Var = Variable;
```

For example:

```
int   var;
int   &  ref_var =  var;
```

To understand reference variable in details, let us consider the following C++ program.

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

int main()
{
      int num = 10;
      int & ref  =  num;

      clrscr( );

     cout<<"\n Value in num = "<<num;
     cout<<"\n Value in num = "<<ref;

     ref  =  num + 20;

     cout<<"\n Value in num = "<<num;
     cout<<"\n Value in num = "<<ref;
     getch( );
     return 0;
}
```

In the above program, ‘num’ is an integer variable initialized with value 10. On the other hand ‘ref’ is the reference variable that is bound with the integer variable ‘num’. So the first two outputs of the above program are shown as follows.

```
Value in num = 10
Value in num = 10
```

In the above program the reference variable ‘ref’ is assigned by the value that is estimated by adding 20 with the value stored in the variable ‘num’. As a result, the value of the variable ‘num’ is also changed that is similar to the reference variable ‘ref’. So, the final two outputs of the program are shown as follows.

```
Value in num = 30
Value in num = 30
```

CHECK YOUR PROGRESS

3. Multiple choices

(a) Which of the following is a correct way to pass a string to a function where the string is stored in the character array str[40]?
(i) fun(str)
(ii) fun(&str[0])
(iii) fun(*str)
(iv) Both (i) and (ii)

(b) new is a ______ .
(i) operator
(ii) function
(iii) variable
(iv) object

(c) In C++, dynamic memory allocation can be performed by _____.
(i) malloc ( )
(ii) new
(iii) delete
(iv) None of the above

(d) In C++, dynamically allocated memory can be released by using _____.
(i) free ( )
(ii) new
(iii) delete
(iv) None of the above

(e) Which of the following will refer the element that is referred by A[i][j]?
(i) *(*(A+i)+j)
(ii) *A+i+j
(iii) *(*A+i+j)
(iv) None of the above

(f) Which of the following is a correct syntax to declare a reference variable that refers an integer variable 'num'?
(i) int ref_var = & num;
(ii) int num = & ref;
(iii) int & ref_var = num;
(iv) None of the above

(g) What is the output of the following C++ statements?

```
int  num;
int  & ref_var = num;
num = 20;
ref_var = ref_var + 10;
num = num -1;
cout <<  num;
```

(i) 19
(ii) 20
(iii) 30
(iv) 29

4. State whether true or false

(a) In C++, memory can be dynamically allocated by delete.
(b) Reference variable is an alternative name to an already declared variable.
(c) 'new' is a special function.
(d) A two dimensional array cannot be passed to a function by using a pointer.

5.8 SUMMING UP

The summery of this unit is given as follows:

- Pointer is a variable which can store the address of another variable of same type. Syntax of declaring a pointer variable is given as follows:
  Data type * variable_Name;

- Pointer to pointer is a pointer variable which stores the memory address of any pointer variable of same data type. For example: int **dptr.
- Using pointer, we can use the base address of an array to access the array elements. A[i] can be referred by *(A+i) and A[i][j] can be referred by using *(*(A+i)+j). An array of pointers is an array which stores a collection of same type of pointers. For example: int *Aptr[6];
- In case of array, the base address of the array can be passed to a function and in the function definition, using pointer variable and pointer arithmetic we can access the array elements.
- A pointer to function or a function pointer is a pointer that holds the starting address of a function. Pointers to function can also be passed as parameters to other functions.
- Dynamic memory allocation is the allocation of memory during program execution that is at the runtime of a program. In C++ programming language, dynamic memory allocation can be performed by the operator 'new'.
- 'delete' is the operator used to release a reserved memory space.
- In C++, reference variable is a variable that offer an alternative name to an already declared variable.
- The syntax to declare a reference variable is presented as follows.

```
Data_Type  & Reference-Var = Variable;
```

5.9 ANSWERS TO CHECK YOUR PROGRESS

1. (a). (ii) , (b). (ii) , (c). (iv) , (d). (iv) , (e). (i) , (f) (i)

2. (a). * , (b). Arr , (c). 1134 , (d). Collection of similar types of pointers

3. (a) (iv), (b) (i), (c) (ii), (d) (iii), (e) (i), (f) (iii), (g) (iv)

4. (a). False , (b). True , (c). True , (d). False

5.10 POSSIBLE QUESTIONS

1. Define pointer with suitable example.
2. What is pointer to pointer? Give example.
3. Explain call by address with suitable example.
4. Explain how a two dimensional array can be passed to a function using pointer. Give suitable example.
5. What do you mean by dynamic memory allocation?
6. Explain how pointer arithmetic can be used to access one dimensional and two dimensional arrays. Give examples.
7. Explain reference variable with a suitable example?

5.11 REFERENCES AND SUGGESTED READINGS

1) Venugopal, K. R., Rajkumar, Ravishankar, T. *Mastering C++*. Tata McGraw-Hill Education, 2001.
2) Balagurusamy, E. *Object Oriented Programming with C++*. Tata McGraw-Hill, 2006

UNIT 6: FUNCTIONS

Unit Structure:

6.1 Introduction

6.2 Unit Objectives

6.3 What is Function?

6.4 Structure of a Function

6.5 Declaration of a Function

6.6 Function Definition: Formal Parameters & return Statement

6.7 Function Call: Actual Parameter

6.8 Call By Value

6.9 Call By Address

6.10Types of User Defined Functions

6.11Passing Array To Function

6.12Passing String To Function

6.13Recursive Function

6.14 Summing Up

6.15 Answers to Check Your Progress

6.16 Possible Questions

6.17 References and Suggested Readings

6.1 INTRODUCTION

Functions are one of the important building blocks in C++. It is a block of code that perform a specific task and runs only when called in the main function. Functions adds the advantage of simplifying the code by breaking it into smaller units. Also the user is not required to write the same code again and again, once a function is written it can be called multiple times.

6.2 UNIT OBJECTIVES

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

- understand why function is necessary and its advantages,
- understand the components of a function – function prototype, function definition and function call statement, return-type and argument(s) of a function,
- integrate a function into a program,
- know the differentiate between function call by value and function call by address,
- understand the concept of recursive function and its use.

6.3WHAT IS FUNCTION?

A **function** can be defined as a group of statements that perform a task. A function may be **called**(used) from anywhere in a program for any number of times. There are two categories of Functions and they are:**library functions** and **user-defined functions**.

Library Functions are the functions those are implemented in the C++ library, available under different header files (with extension **.h**). We can use a function in our program whenever the tasks implemented in that function are to be performed. In the earlier units you have come across different library functions, e.g.,

clrscr(), **getch()** implemented in **conio.h**;

strlen(), **strcpy()** implemented in **string.h** etc.

A **User Defined Function**is a function which is implemented by a user (mainly a programmer). So, now onwards we will discuss about **User Defined Function**. **main()** is a special **user-defined function**which is mandatory to be implemented in every program as the execution of a program starts from it.

A program can have more than one user-defined function. Conventionally, functions are so designed that each one of themperforms some independent tasks and later integrated in a single program.

STOP TO CONSIDER

In a statement of a C++ program if a word contains '**( )**' at the end then that word with '**()**' is a function e.g., in the statement '**x=summation();**' then you remain sure that **summation()** is a function (may be user defined one or a library function).

The following are some advantages of using functions:

1. By defining functions, a programmer can divide the entire task of the program into simple subtasks.
2. In a program, a task containing multiple statements may have to be repeated a no. of times. In a function the taskcode can be written and wherever in the program, the task is required to be performed, the functionis called.Thus it reduces the size of the program instead of implementing the same set of code again and again in the program.
3. In C++, a function defined for a particular task can be shared or used by different programs.
4. The advantage of implementing the repetitive code as a function is that whenever there is a requirement of modification in the task-code, you just modify the code inside the function and the modification will be reflected in every use of the function.

6.4 STRUCTURE OF A C++FUNCTION

As already mentioned, a function is a group of statements, which perform a particular task; so, there are rules for its declaration, definition and use. From the previous units, it is clearly understoodthat how can a library (built-in) function be used in our program to perform a particular task for which it is designed.

A user defined function can occur in a program in the following ways.

- Function Declaration (or Function prototype)
- Function Definition:: Formal Arguments
- Function Call:: Actual Arguments

The sections, to follow, will explain the ways one by one. Consider the following program, ***Program-1***, where in the "**main()**" function, two integers are taken as input, then calls the user defined function

"**sum()**" passing the two input integers and gets the summation in return. Then the summation is displayed on to the screen.

Program-1:

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

int sum(int, int);        <-- Function Declaration or Prototype of function sum()

void main()               <-- Definition of main() function
{
        clrscr();
        int a, b, result;
        cout<<"Enter a number:";
        cin>>a;
        cout<<"Enter another number:";
        cin>>b;
        result=sum(a, b);     <-- Calling sum() function [a, b are Actual Arguments.

        cout<<"The Summation="<<result;
        getch();
}
int sum(int x, int y)     <-- Definition of function sum() [x, y are Formal Arguments.
{
        int s;
        s=x+y;
        return (s);
}
```

6.5 FUNCTION DECLARATION

Like variables, the declaration of a function is necessary before it is used. The function declaration is formally known as **Function Prototype**. As the name **prototype** means **model/blueprint**, the function prototype means the blueprint for the function which basically describes/informs the compiler about the return type, function name and data-type of the **parameters/arguments** passed

to it. Except the "**main()**" and the library functions, all other user defined functions should have a prototype.

The syntax for function declaration/prototype is:

return-type function-name (parameter-type-list);

You may get confused with the word **return-type,** well**,** it means data-type!!! Yes, only valid data-types (built-in/user defined) can be used as return-type for a function. Return-type basically describes the kind of the data/value, a function can return. If, a function should return an integer data/value then the return-type should be one among **int**/**short int**/**long int**. If a function does not return any value then the return-type should be **void**.

function-name is the name given to a function. The rules for naming a function are the same as that for a variable.

The **parameter-type-list** is the list of data-types for the data/values to be passed to the function as parameters, each separated by '**,**'. Some, along with the data-type a parameter name is given for each of the parameters in the list but this is optional.

The function prototype statement should be terminated by a semi-colon.From the ***Program-1***, the declaration of the function **sum()** is illustrated in ***Fig-6.1***.

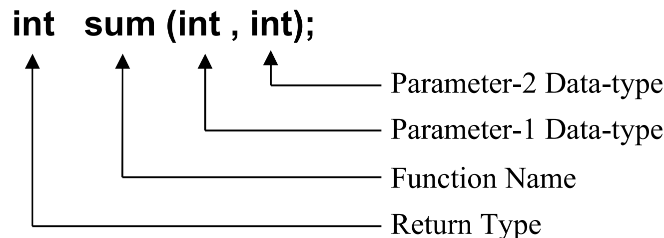


Fig-6.1

The above prototype tells the compiler that the function **sum()** takes two integers as arguments and returns an integer. The values/data which a function takes are called as parameters or arguments. A function in C++ can take as many arguments as it needs (or no arguments at all) but can return only one argument.

Example-1: Write a function prototype which takes a character as parameter and returns nothing.

```
void  fun_1 (char);
```

or, we can write as

```
void  fun_1 (char param);
```

Explanation:

- ✓ Since, the function returns nothing the return-type is mentioned as **void**.
- ✓ **fun_1** is the name of the function.
- ✓ '**char**' or '**char param**' is mentioned within **( )** brackets as it is said that the function takes a character as parameter.

Example-2: Write a function prototype which takes two floating point numbers as parameters and returns their summation.

```
float  summation (float, float);
```

or, we can write as

```
float  summation (float num1, float num2);
```

Explanation:

- ✓ Since it is said that the function returns the summation of two floating point numbers so, the return-type is mentioned as **float**.
- ✓ **summation** is the name of the function.
- ✓ '**float**','**float**' (or '**float num1', 'float num2**') are mentioned within **( )** brackets as it is said that the function takes two floating point numbers as parameters.

STOP TO CONSIDER

If the definition of a particular function (e.g. **sum**) is mentioned after the function, from where the 1st function is called upon (e.g. **main**), then the function declaration/prototype 1st function (**sum**) is mandatory. But, if the 1st function (**sum**) is defined before the 2nd function (**main**) then the declaration/prototype is optional.

6.6 FUNCTION DEFINITION: FORMAL PARAMETERS & THE return STATEMENT

The definition of a function tells exactly what the function is written for. A **Function Definition** comprises of the **function name**, **return-type**, ***number of parameters withtheir types*** and ***its body***. A **Function**is a block of statements that will be executed when the function is called. The syntax for function definition is:

return-type function-name(type param*1*, type param*2*,…..)

{

//Statements

}

In the above syntax,

- **return-type** is the **data type** of the value/data to be returned by the function.
- **function-name** is the name of the function.
- "**type param1, type param2, ……**" is the list of parameters to be passed to the function. Here, unlike in function prototype, name of the parameteralong with its type is mandatory for each of the parameters for the function.

STOP TO CONSIDER

You may notice in the ***syntax*** of the ***function definition*** that, at the end of the header statement there is no '**;**'. This so because, it is the start of the function definition.

Consider the definition of function **sum()** in ***Program-1***, whichis mentioned after the **main()**:

```
int sum(int x, int y)
{
        int s;
        s=x+y;          } Body of the function sum()
        return (s);
}
```

In theabove function **sum()**,

- Apart from return-type as int and function-name as sum, the function heading also contains **x** and **y** as two parameters of type **int**.
- **{** starts the body of the function.

- 1st statement, within the body of the function, is the declaration statement of the variable **s**.
- in the 2nd statement, within the body of the function, the value of **x** and **y** are added and stored into variable **s**. x and y will contain the values that will be passed when the function will be called.
- The last statement, within the body of the function, will return the value of the variable **s** to the function from where the function **sum()** will be called upon.

6.6.1 Formal Parameters/Arguments

Consider the definition of function **sum()** mentioned above. Here, **x** and **y** are the used as parameters/arguments for the data/values to be passed when the function **sum()**will be called and are called as **Formal Parameters** or **Formal Arguments**. Thus **Formal Parameters/Arguments** can be defined as the parameters/arguments that are mentioned in the definition of a function.

In the function **sum()**there is a variable **s** which is local to the function. But at the same time the parameters **x** and **y** are also can be treated as local variables of the function **sum()**.

Now, when the **sum()** is called with the parameter-values, the function **sum()** starts executing and the two parameter-values are stored in **x** and **y** respectively.

STOP TO CONSIDER
The names of the formal and the actual arguments may be the same or be different but the data-types must be the same.

6.6.2 The return Statement

In the definition of the **sum()** function of ***Program-1***, the **return** is used at the end of the function body along-with the variable **s** within **( )**. This means that the value of **s** is returned to the function from which the **sum()** is called upon, i.e. from within the **main()** function.

Basically, the **return** statement is used for two purposes:

- ✓ To return a value from **called function** to the **calling function**.For this, the ***value of the variable*** to be returned is mentioned at the end of **return** statement within **( )**.
- ✓ To end the execution of the called function and transferring the control back to the calling function. So, in this situation along-with the **return** statement no value/value of a variable is mentioned.

STOP TO CONSIDER

The main limitation in the use of the return statement is that you can use it to return only one value.

CHECK YOUR PROGRESS - I

1. What is function?
2. What do you understand by User Defined Function?
3. What is Function Prototype?
4. What is Formal Arguments or Formal Parameters?

State TRUE or FALSE:

5. A function always returns a value.
6. A function may or may not have parameters.
7. **return** statement is used to end a function.
8. Return type of a function can be **void**.

6.7 FUNCTION CALL: ACTUAL PARAMETER

Basically, the **Function Call** means the use of a function in a program where the function may be a library function or a user defined function.

Consider the **main()** function in ***Program-1***.

```
void main()
{
	clrscr();
	int a, b, result;
	cout<<"Enter a number:";
	cin>>a;
	cout<<"Enter another number:";
	cin>>b;
	result=sum(a, b);
	cout<<"The Summation="<<result;
	getch();
}
```

Here in the **main()** function,

- two ***integer inputs*** are stored into the variable **a** and **b** using two **cout**statements,
- in the statement,

```
result=sum(a, b);
```

the function **sum()** is used(or called) with the variable**a** and **b** mentioned within **( )**.

Actual Parameters/Arguments:

Consider the definition of function **main()** mentioned above. Asmentioned earlier, **a** and **b** are the two variables passed tothe function **sum()**when it is called. Here, **a** and **b** are known as **Actual Parameters** or **Actual Arguments**.

Now, let's discuss about how many ways a function can be called. There are two ways of calling a function (may be a library or user defined function) and they are:

- Call By Value
- Call By Address
- Call By Reference

In terms of parameter/argument passing the abovementioned ways of function calling are also known as:

- Pass By Value
- Pass By Address
- Pass By Reference

6.8 CALL BY VALUE

In this technique of calling a function, only the values are passed as parameters. In the above mentioned **main()** function while calling the function **sum()**, the variables **a** and **b** are passed as arguments(actual). Thus, the values stored in **a** and **b**are passed to the function **sum()** when it is called. As in this method of function call, the values are passed as parameters to the called function, hence this method of function call is known as **Call by Value** or **Pass by Value**.

Consider the ***Program-2***. Here in this program **sum()** is a user defined function, same as mentioned in **Program-1**. But the **main()** function(in ***Program-2***) is different from the **main()** in ***Program-1***.

Program-2: Demonstration of Call By Value.

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

void main()
{
        clrscr();
        int  result;
        result=sum(5, 2);
        cout<<"The Summation="<<result;
        getch();
}

int sum ( int x, int y )
{
        int  s;
        s=x+y;
        return (s);
}
```

Output:

```
The Summation=7
```

Explanation:

When the program runs, execution starts for the **main()** function.Statements in the **main()** start executing one-by-one. When, the following statement executes,

```
        result=sum(5, 2);
```

- ✓ the **sum()** function is called with actual arguments **5**, **2**.

- ✓ The execution control is now transferred to the function **sum()** with the values **5** and **2**, [passed from the **main()]** those eventually stored in the **x** and **y** respectively.
- ✓ Inside **sum()**, the values stored in **x** and **y**, i.e. **5** and **2** respectively, are added and then assigned to **s**.
- ✓ At the end of **sum()**, the **return** statement returns the value of **s**, i.e. **7**, and transfers the execution control back to the above statement in the **main()** function from where the function **sum()** was called.
- ✓ The returned value **7**[value of **s** in **sum()**] is assigned to the variable **result** in the **main()**.
- ✓ Now, the **cout** statement is executed and the output is produced onto the screen.

The above explanation is depicted in **Fig-6.2**.

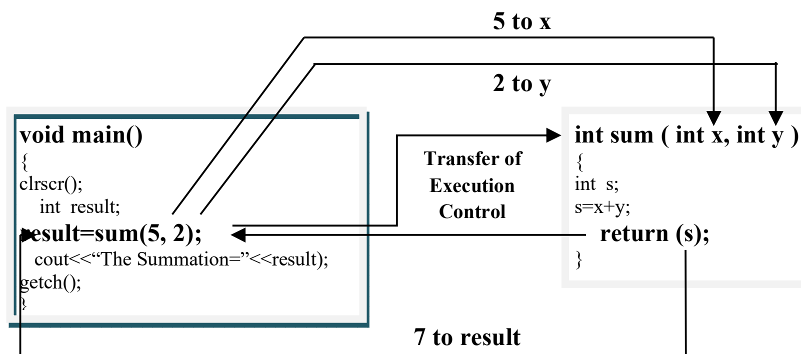


Fig-6.2: Illustration of function calling, parameter passing & returning a value (Program-2)

Now, let's try to explain the execution of ***Program-1***. Consider that **10** and **20** are the given input for the variable **a** and **b** respectively.

Output:

```
Enter a number:10
Enter another number:20
The Summation=7
```

Explanation:

In the **main()** function, when the following statement executes,

```
result=sum(a, b);
```

- ✓ the **sum()** function is called with actual arguments **10**, **20** as they were the given inputs for **a** and **b** using the two **cin**statements.
- ✓ The execution control is now transferred to the function **sum()**with the values of **a** and **b**, i.e.**10** and **20,** [passed from the **main()]**thoseeventuallystored in the **x** and **y** respectively.
- ✓ Inside **sum()**, the values stored in **x** and **y**, i.e. **10** and **20** respectively, are added and then assigned to **s**.
- ✓ At the end of **sum()**, the **return** statement returns the value of **s**, i.e. **30**, and transfers the execution control back to the above statement in the **main()** function from where the function **sum()** was called.
- ✓ The returned value **30** [value of **s** in **sum()**] is assigned to the variable **result** in the **main()**.
- ✓ Now, the **cout** statement is executed and the output is produced onto the screen.

The above explanation is depicted in **Fig-6.3**.

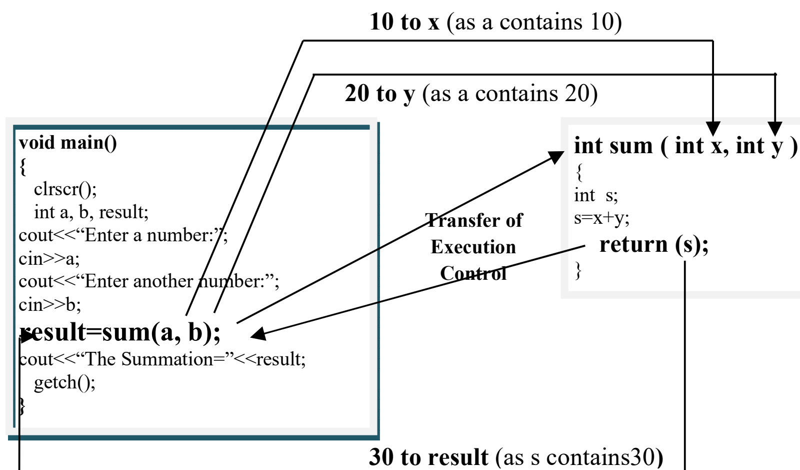


Fig-6.3: Illustration of function calling, parameter passing & returning a value (Program-1)

6.9 CALL BY ADDRESS

Before discussing the topic, let's first know about **Address**. The main memory is addressable which means that any object which resides in it has an **Address**. The variables can be accessed by their names as well their addresses.We already know that the **Address-**

of-Operator(**&**)is used to get the Address of the memory location used by a variable. Now the term, **Call By Address**, means that while calling a function we need to pass the address. And in the called function the data-item in that location can be accessed using the address passed. So, in the definition of the function the argument should be such that it can hold an address of a location. For this let's discuss about **Pointers**.

What is a Pointer?

A**Pointer Variable** can be defined as the variable which can store an address of a memory location. The declaration syntax of a pointer variable is the same as that of a variable with a '*' symbol before the variable name.For example,

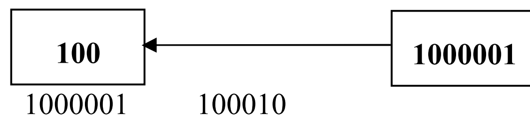
```
int  *a;
```

Consider the following statements, which clearly describe the use of a pointer variable.

```
int x =100;
int *p;
p=&x;
```

- ✓ The first statement declares an integer variable **x** as well as 100 is assigned to it.
- ✓ Second statement declares an integer pointer variable **p**.
- ✓ In the third statement, the address of the variable **x** is assigned to the pointer variable **p**.

Fig6-4 depicts the scenario after execution of the third statement while considering the address of **x** is **100001** and address of **p** is **100010**.



xp

Fig-6.4

Therefore, the variable **x** can be accessed using **x** itself and **p**. These are illustrated in the following statements.

```
cout<<"The value of x = "<<x;
cout<<"\nThe value of x = "<<*p;
```

Output:

```
The value of x = 100
The value of x = 100
```

Explanation:

- ✓ The first **cout** will display the value of **x** using itself.
- ✓ The second **cout** will display the value of **x** using **p**. So, here ***p** means the ***valueat the memory location*** whose **address** is stored in **p**. In other words, ***p** means the **value at the location pointed** by **p**.

Now, you may think of that, if ***p** and **x** mean the same location then what does **p** mean??? The p contains the address of x, i.e., 1000001. So, the following statement will display the content of **p**, i.e., the address of **x**, and which is **1000001**.

```
cout<<"The address of x (using p) ="<<p;
```

The **address** value will be displayed as a **hexadecimal number**.

The following statement will also display the address of x.

```
cout<<"The address of x (using x itself) ="<<&x;
```

Hope!!! that you have a got a clear idea about addresses and pointers. Now, let's come to the topic of discussion i.e.,**Call By Address** or **Pass By Address**.

To understand this, let's consider the ***Program-3*** which is a modification of ***Program-1***.

Program-3: Demonstrating the Call By Address/Pass By Address.

```
#include<iostream.h>
#include<conio.h>
int sum(int, int);

void main()
{
        clrscr();
        int a, b, result;
        cout<<"Enter a number:";
        cin>>a;
        cout<<"Enter another number:";
        cin>>b;
        result=sum(&a, &b);
        cout<<"The Summation="<<result;
        getch();
}
```

```
int sum(int *x, int *y)
{
        int s;
        s=*x+ *y;
        return (s);
}
```

The output of this program will be the same as the output of the ***Program-1*** if same inputs are considered for **a** and **b**.

Let's discuss the execution of the program in brief.In the **main()** function:

- ✓ Using **cin**, two inputs are taken and stored into the variables **a** and **b**.
- ✓ In the statement, marked as **bold**, the **sum()** function is called and **&a** (i.e. address of **a**) and **&b** (address of **b**) are passed as arguments.

Now, execution control is transferred to the **sum()** function. As the arguments from the **main()** function are addresses of **a**, **b** therefore the formal arguments in the definition of the function **sum()** are declared as **integer pointers**.

In the **sum()** function:

- ✓ The addresses of **a** and **b** passed from **main()** are stored into the pointer variables **x** and **y** respectively, i.e. **x** and **y** are pointing to variables **a** and **b** in of **main()** function.
- ✓ In the statement, **s=*x+*y;*x** and ***y** means the variables **a** and **b** of the **main()** function. So, the values of **a** and **b**, i.e. pointed by **x** and **y**, are added and stored into the variable **s**.
- ✓ The value of **s** is returned to **main()**.

Now, the execution control is transferred back to **main()** function. In the **main()** function:

- ✓ The returned value from **sum()** is then stored into the variable **result**.
- ✓ The value of the **result** variable is then displayed using **cout**.

6.10TYPES OF USER DEFINED FUCNTIONS

We have already discussed the basic concepts related to Function Prototype, Function Definition and Function Calling. Functions can be categorized in to different types depending on the return type and the arguments.

6.10.1 Function with No Argument and No Return Value:

Function with no argument means no argument list within **()** in a function definition and in function calling and function prototype.

Function with no return value means **void** as return type of the function.

Functionsof this type, defined by user are rare. But there are built-in/library functions of this type e.g, **clrscr()**, **getch()** etc. ***Program-4***is an example of a user defined function of this kind.

Program-4: Write a C++ program using the concept of functions to display the nos. from 1 to 10.

```
#include<iostream.h>
#include<conio.h>
void displaynums();

void main()
{
        clrscr();
        displaynums();
        getch();
}

void displaynums()
{
        int i;
        cout<<"The numbers from 1 to 10 are:n\";
        for (i=1; i<=100; i++)
        {
                cout<<i;
        }
}
```

Output:

```
The numbers from 1 to 10 are:
```

```
1 2 3 4 5 6 7 8 9 10
```

6.10.2 Function with Argument(s) but No Return Value

Function with arguments means there may be one or more than one argument in a function definition and in function calling and function prototype.

Function with no return value means **void** as return type of the function.This is illustrated in ***Program-5*** mentioned below.

Program-5: Write a C++ program usingfunctions which will take two numbers as arguments and displays the nos. between them.

```
#include<iostream.h>
#include<conio.h>
void displaynums(int start, int end);

void main()
{
	clrscr();
	displaynums(100, 500);
	getch();
}

void displaynums(int start, int end)
{
	int i;
	cout<<"The numbers from "<<start<<" to "<<end<<"are:\n";
	for (i=start; i<=end; i++)
	{
		cout<<i<<" ";
	}
}
```

Output:

```
The numbers from 1 to 10 are:
1 2 3 4 5 6 7 8 9 10 …………………….. 500
```

6.10.3 Function with Argument(s) and Return Value

Function with arguments means there may be one or more than one argument in a function definition and in function calling and function prototype.

Function with return value means data types other than **void** as return type of the function.This is illustrated in ***Program-6*** mentioned below.

Program-6: Write a C++ program using functions which will take two numbers as arguments and returns the summation of the nos.

```
#include<iostream.h>
#include<conio.h>
int summation(int start, int end);

void main()
{
	clrscr();
	int x, y, result;
	cout<<"Enter the starting no: ";
	cin>>x;
	cout<<"Enter the ending no: ";
	cin>>y;
	result=summation(x, y);
	cout<<"The summation of the numbers from "<<x<<" to "<<y<<" = "<<result;
	getch();
}

intsummation(int start, int end)
{
	int i, sum=0;
	for (i=start; i<=end; i++)
	{
		sum = sum + i;
	}
	return (sum);
}
```

Consider the input given to x and y in the main() function are 1 and 10 respectively.

Output:

```
Enter the starting no: 1
Enter the ending no: 10
```

```
The summation of the numbers from 1 to 10= 55
```

6.10.4 Function with No Argument but Return Value

In this type of functions, no arguments are declared in the definition of the functions but return type should be mentioned.

This is illustrated in ***Program-7*** mentioned below.

Program-7: Write a C++ program using functions which will return the summation of the nos. from 1 to 10.

```
#include<iostream.h>
#include<conio.h>
int summation();

void main()
{
        clrscr();
        int result;
        result=summation();
        cout<<"The summation of the numbers from 1 to 10= "<<result;
        getch();
}

int summation()
{
        int i, sum=0;
        for (i=1; i<=10; i++)
        {
                sum = sum + i;
        }
        return (sum);
}
```

Output:

```
The summation of the numbers from 1 to 10= 55
```

6.11PASSING ARRAY TO FUNCTION

In C++, , we can also pass array to a function as parameter. We know that the syntax of declaring aone-dimensionalarray is:

data-type array-name [size];

The same syntax is used while declaring an array as argument in a function definition but with a little modification:

```
void  process(int  a[])
{
        Statements…….
}

void  process(int  a[], int n)
{
        ……………….
        ……………….
}
```

where the argument **n** is for the size of the array-argument **a[]**.

The prototype of the function **process()** may be written as:

void process(int a[10]);

Or, **void process(int a[]);**

Or, **void process(int []);**

During the call to the function**sum()**, we have to pass two items as arguments: the **name** of the ***actual array*** as 1st argument and the ***no. of elements*** present in the array as the 2nd argument.

```
void main()
{
        ………………..
        process(arr, 20);
        …………..……
}
```

Program-8: Write a C++ program using functions to calculate the summation of the nos. in an integer array passed as parameter.

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

int arraySUM(int [], int);
```

```
void main()
{
	clrscr();
	int a[100], n, i, result;
	cout<<"How many nos. you want to enter: ";
	cin>>n;
	cout<<"Enter the nos:\n";
	for (i=0; i<n; i++)
	{
		cin>>a[i];
	}
	result= arraySUM(a, n);
	cout<<"The Summation:"<<result;
	getch();
}

int arraySUM(int arr[], int size)
{
	int i, sum=0;
	for (i=0; i<size; i++)
	{
		sum = sum + arr[i];
	}
	return (sum);
}
```

Output:

```
How many nos. you want to enter: 5
Enter the nos:
2
4
8
45
1
The Summation: 60
```

Program-9: Write a function in C++ to findmaximum of the nos. in the integer array passed as parameter.

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

void findmax(int [], int);

void main()
{
	clrscr();
```

```
        int a[100], n, i, result;
        cout<<"How many nos. you want to enter: ";
        cin>>n;
        cout<<"Enter the nos:\n";
        for (i=0; i<n; i++)
        {
                cin>>a[i];
        }
        findmax(a, n);
        getch();
}

int arraySUM(int arr[], int size)
{
        int i, max=arr[0];
        for (i=1; i<size; i++)
        {
                if(arr[i]>max)
                {
                        max=arr[i];
                }
        }
        cout<<"The Maximum No: "<<max;
}
```

Output:

```
How many nos. you want to enter: 5
Enter the nos:
2
4
8
45
1
The Maximum No: 45
```

6.12 PASSING STRING TO FUNCTION

Like **array**, a s**tring** can also be passed as parameter to a function. The syntax of defining a function which takes a **string** as parameter is:

```
return_type  function_name(char  string_array[])
{
        Statements.........
}
```

The ***Program-10*** will help you to understand the above facts.

Program-10: Write a C++ program using functionwhich returns the no. ofvowels in the string passed as parameter to it.

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

int vowelnos(char []);

void main()
{
	clrscr();
	char  strname[100];
	int no;
	cout<<"Enter a String: ";
	cin>>strname;
	no=vowelnos(strname);
	cout<<"The no. of Vowels="<<no;
	getch();
}

int vowelnos(char n[])
{
	int i, count;
	for (i=0, count=0; n[i]!='\0'; i++)
	{
		if(n[i]=='a' || n[i]=='e' || n[i]=='i' || n[i]=='o' ||
n[i]=='u' || n[i]=='A' || n[i]=='E' || n[i]=='I' || n[i]=='O' ||
n[i]=='U')
		{
			count++;
		}
	}
	return count;
}
```

Output:

```
Enter aString: WELCOME TO IDOL
The no. of Vowels= 6
```

6.13 RECURSIVE FUNCTION

Till now, all user defined functions discussed above are called inside the **main()**,. So, any function can be called from any other functions. Apart from this, C++ also enables a function to call itself. This

technique is called **Recursion**. **Recursive Function** is a function which calls itself.Consider the following ***Fig-6.5***.

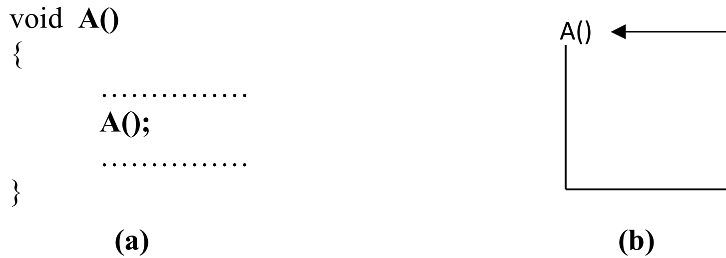


Fig-6.5

In the definition of **A()**, there is a statement which calls the function itself [***Fig-6.5(a)***]. The function **A()** is a **Resursive Function**. This is shown graphically in ***Fig-6.5(a)***.

There is another term known as **Indirect Recursion**. **Indirection Recursion** occurs when one function calls another function that then calls the first function. The following is an example of Indirect Recursion. Consider the following ***Fig-6.6***.

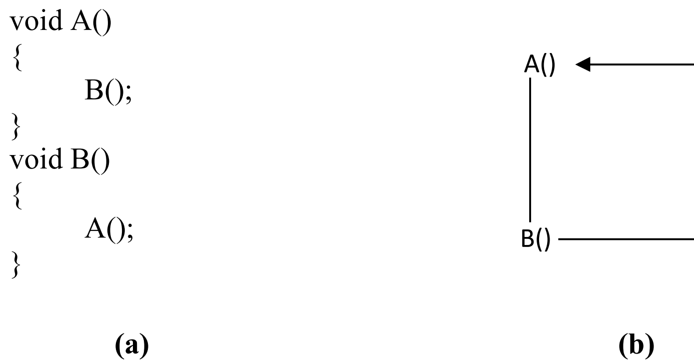


Fig-6.6

In the definition of **A()**, there is a statement which calls the function **B()** and in the definition of **B()** a statement calls the function **A()** [***Fig-6.6(a)***]. This is called **Indirect Recursion**. This is shown graphically in ***Fig-6.6(a)***.

Program-11: Write a C++ program which calculates the summation of the no. from 1 to 3 using recursive function.

```
#include<iostream.h>
#include<conio.h>
int calc(int n);
```

```
void main()
{
        cout<<"\nThe Summation = "<<calc(3));
}

int calc(int n)
{
   if(n==0)
        return 1;
   else
        return n+calc(n-1);
}
```

Output:

```
The Summation = 6
```

Explanation (Graphically):

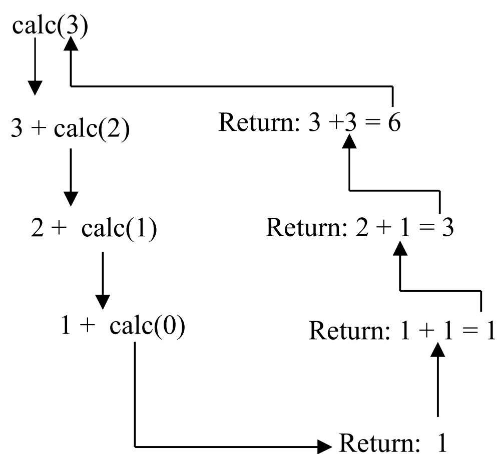


Fig-6.7

Program-12: Write a C++ program which calculates the factorial of a no. using recursive function.

```
#include<iostream.h>
#include<conio.h>
int  fact(int n);

void main()
{
        cout<<"\nThe Factorial of 3 = "<<fact(3));
```

```
}

int fact(int n)
{
   if(n==0)
         return 1;
   else
         return n*fact(n-1);
}
```

Output:

```
The Factorial of 3 = 6
```

Explanation (Graphically):

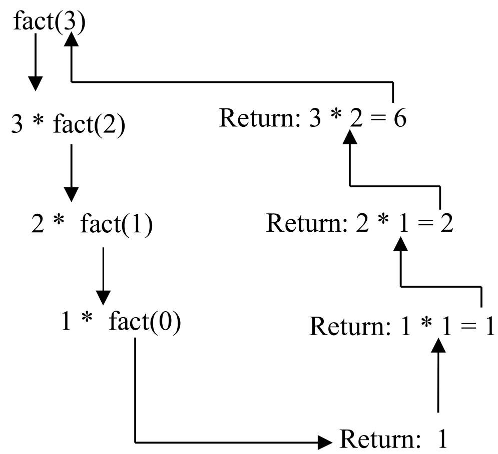


Fig-6.8

CHECK YOUR PROGRESS - II

9. What do you understand by Actual Parameters?

10. What are the two ways of calling a function?

11. What is Pointer? How is it related to Call by Address?

12. What do you understand by ***function with no argument and no return value***?

State True or False:

13. ***p** means the location whose address is stored in **p**.

14. In Call By Value the addresses of the storage locations are passed.

15. An array cannot be passed as parameters to a function.

Fill-in the Blanks:

16. _______________ function calls itself.

17. While passing an array to a function, the ________of the array should also be passed.

6.14 SUMMING UP

This Unitdiscusses the cocept of functions its types and its advantages.

As discussed in this Unit, C++ library consists of a no. of header files, e.g.,**iostream.h**, **conio.h**, **string.h** etc. These header files contain functions for different purposes.

The concept of User Defined Functions is also discussed in this Unit. These types of functions are very useful to cater out userneeds while writing C++ programs.

The two ways of calling a function, Call by Value and Call byAddress, are discussed with examples.

The unitgives the basic idea related with function definition, function call, and function prototype.

The types of user defined functions in relation with arguments and return type are discussed with the help of examples. Also functions with array as arguments are also discussed with examples.

6.15 ANSWERS TO CHECK YOUR PROGRESS

1. A function can be defined as a group of statements that perform a task. A function may be called (used) from anywhere in a program for any number of times.

2. User Defined Function is a function which is implemented by a user (mainly a programmer). main() is a special user-defined function from where the execution of a program starts.

3. Like variables, the declaration of a function is necessary before it is used. The function declaration is formally known as Function Prototype.

4. The Formal Parameters/Arguments can be defined as the parameters/arguments that are mentioned in the definition of a function.

5. False

6. True

7. False

8. True

9. Actual Parameters/Arguments can be defined as the items/values those are passed to a function when it is called.

10. There are three ways of calling a function and they are namely Call By Value/Pass By Value, Call By Address/Pass By Address and Call By Reference/Pass By Reference.

11. A Pointer or pointer Variable can be defined as the variable which can store an address of a memory location.

12. Function with no argument means no argument list within ( ) in a function definition and hence no argument in function calling and function prototype also. Function with no return value means void as return type of the function.

13. True

14. False
15. False
16. Recursive
17. name

6.16 POSSIBLE QUESTIONS

Short answer type questions:

1. What are the main advantages of using functions?
2. Write down the categories of functions?
3. What is the syntax of defining a function?
4. Why does the **return** statement used in a function?
5. What is the difference between a Function Definition and Function Prototype?
6. What is the use of the return statement?

Long answer type questions:

7. Write down the syntaxesfor Function Prototype, Function Definition and Function Call.
8. Differentiate between Call by Value and Call By Address.
9. How can you relate pointers with Call by Address? Discuss with the help of an example.
10. How can array be passed to a function? Discuss with the help of an example.
11. What is a Recursive Function? Discuss.
12. Write a C++ function which swaps the two arguments passed as parameters so that swapping reflects in the actual parameters.
13. Write a recursive function to evaluate the following series:

    S = 1 + 2 + …….. + n

6.17 REFERENCES AND SUGGESTED READINGS

1. Stroustrup, Bjarne.*The C++Programming Language*.
2. Balagurusamy, E.. *programming in C++*. Tata McGraw-Hill Education.
3. Kanetkar, Y. P.. *Let us C++*. BPB Publications.

UNIT 7: CLASSES AND OBJECTS

Unit Structure:

7.1 Introduction

7.2 Unit Objectives

7.3 Introduction to Objects and Classes

7.4 Defining Classes

7.5 Creating Objects

7.6 Access Specifiers in C++

7.7 Accessing Members of an Object

7.8 Constructors and Destructors

7.9 Friend Function

7.10 Summing Up

7.11 Answers to Check Your Progress

7.12 Possible Questions

.13 References and Suggested Readings

7.1 INTRODUCTION

Abstraction is one of the major concepts related to Programming Languages those support Object Oriented Programming Concepts. This concept handles complexity by hiding the unnecessary details to the user.

Suppose your job is to print documents in an Office. For this you just need to open the documents (to be printed) and give the print command. The printer will give printouts of the documents. You don’t require the knowledge about how a printer works internally. For your job, you just use it. Thus, the manufacturer of the printer hides the complexity of the printing process from its user. This can be termed as Abstraction.

Same is the case related to Object Oriented Programming Languages. As a part of Data Abstraction let’s discuss Object, Class, Member Function, Constructor, Destructor and Friend Function.

7.2 UNIT OBJECTIVES

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

- understand the concept of Abstraction
- know the various important issues related to Abstraction
- realize how it helps in adding securities to the sensible data
- learn how to implement Abstraction in C++

7.3 INTRODUCTION TO OBJECTS AND CLASSES

Any real-world entity, living or non-living, is termed as Object. For example, Ritz, Baleno, Swift Desire, i10, Santro, Duster, XUV300 etc. can be termed as object. As we know, today cars are categorized as Hatchback, Sedan, SUV etc. So, Ritz, i10, Santro are considered as Hatchback; Baleno, Swift Desire as considered as Sedan; Duster, XUV300 are considered as SUV. Each of these categorizations can be considered as a Class. Thus, we can say that

- Ritz, i10, Santro are the Objects of Class Hatchback,
- Baleno, Swift Dezire are the Objects of Class Sedan, and
- Duster, XUV300 are the Objects of Class SUV.

Now, you may have a preliminary idea about objects and classes.We will discuss the OOP concepts with the help of the C++ Programming.

7.4 DEFINING CLASSES

Class is basically the main building block in OOP. In terms of C++, class is a user-defined data type which has members to store data and perform certain activities. The members that stores data are termed as data members and those which perform activities are termed as member functions. The data members are called Attributes and member functions are called Methods.

In C++ by creating a class is basically the creation of a blueprint of data type.In C++ there are different ways of defining a class. The simplest way is given below:

```
class class-name
{
    access-specifier:
        data-type attribute-name-1;
        data-type attribute-name-2;
      return-type method-name-1()
        {
            ..............
        }
        return-type method-name-2()
        {
            ..............
        }
};
```

Let's now discuss the above way of defining a class point-wise.

- "class" is a keyword and is used to define a class. "class-name" is the name of the class to be defined.
- After the "access-specifier" may be any of "public", "private" and "protected". These three access-specifiers will be discussed in detail later in this unit.
- "attribute-name1", "attribute-name2" are the attribute names or the data members of the class. The "data-type", as we all know, is the type of the data that can be stored in the respective data-members.
- "method-name1()", "method-name2()" are the functions those are defined for performing certain task. The "return-type" is the data-type of the value to be returned from the respective functions. For each of the functions, the function statements are to be written within the braces "{ }".So, this is exactly how we write a function (as mentioned in earlier units).
- At last, we have to end the class definition by using "}" which is followed by ";".

STOP TO CONSIDER

More than one access-specifier can be used to declare and define data-members and member-functions within a single class.

For the time-being, let's consider the "public" as the access-specifier. Suppose we have to create a class named "showsymbol" which has,

- an attribute/data-member named "val" of type integer,
- a method named "input()" which will store an integer to "val" input by the user, and
- a method/member-function named "show()" which will display the symbol "*" no. of times that is stored in "val".

Now we will define the class "showsymbol".

```
class showsymbol
{
    public:
        intval;
       voidinput()
        {
        cin>>val;
        }
       voidshow()
        {
        int i;
           for(i=0; i<val; i++)
           {
              cout<<"*  ";
           }
        }
};
```

Thus, we have completed the definition class of the "showsymbol" following the above mentioned way. In this way, we are declaring and defining the attributes and methods of the class within the braces, { and }, of the class definition.

The other way of defining a class is to:

- ✓ declare the attributes inside the class definition, i.e. within the curly braces, { and }, used for the class.
- ✓ only declare the methods/functions withing the class, i.e. the prototype of the methods/functions.
- ✓ define the methods/function outside the class, i.e. not within the curly braces, { and }, used for the class. The syntax of defining a member function outside the class definition is:

```
return-type class-name::member-function-name()
{    …….    }
```

Now, we will see how to define the same "showsymbol" class in the other way just mentioned above.

```
class showsymbol
{
   public:

        intval;
      voidinput();
      voidshow();
};

voidshowsymbol::input()
{
cin>>val;
}
voidshowsymbol::show()
{
int i;
   for(i=0; i<val; i++)
   {
      cout<<"*  ";
   }
}
```

Here in the above code, you may notice the use of **::** symbol. This symbol is known, in C++, as Scope Resolution Operator. As the name suggests,this operator signifies that the following function belongs to the class (as member function) whose name precedes it. Thus, the functions (**input** and **show**) defined above are the member functions of the class **showsymbol**.

7.5 CREATING OBJECTS

An object of a class is just like variable of a primitive data-type. The difference is that an object contains different data-members of different data-types along-with different functions but a variable contains a single value of that particular data-type.But like variables, memory is allocated when we declare an object which depends on the memory required for that particular class type.

The syntax of declaration of an object of a class is:

class-name object-name;

The statement for creating an object, named obj1, for showsymbol class is:

showsymbol obj1;

Another way for creating the object is:

showsymbol obj1 = showsymbol();

7.6 ACCESS SPECIFIERS IN C++

The term **Access Specifier**indicates how the members of a class can be accessed from within and outside the class.Outside the class basically means accessing the members of the class from functionsthose are not the members of that respective class.There are three access-specifiers in C++ and they are **public**, **private** and **protected**.

Table-1 shows the accessibility of the access specifiers.

Table-1: Access Specifiers' Accessibility

	Access Specifier		
	public	**private**	**protected**
from same class	Yes	Yes	Yes
from derived class (inheritance)	Yes	No	Yes
from other classes	Yes	No	No

7.7 ACCESSING MEMBERS OF AN OBJECT

The **protected** access specifier is directly related to the concept of Inheritance, which will be discussed in the following unit. Therefore, here we will only discuss the other two access specifiers. i.e., **private** and **public**.

While accessing a member, data-member or member function, of a class we need to use '.' operator preceded by object name and followed by the member name. This is required when accessing from outside the class. But whenever we want to access member of a class from within itself(same class), we can directly use the member.

Let's now consider the above **showsymbol** class, with modifications, for ourdiscussion.

Program-1:

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

class showsymbol
{       private:
           int val;   //private data-member
        public:
        void input();   //public member-function
        void show();  //public member-function
};
void showsymbol::input()     //definition of input public member-
                             //function
{cin>>val;}
void showsymbol::show()    //definition of show public member-
                           //function
{
int i;
   for(i=0; i<val; i++)
   {cout<<"*  ";}
}
void main()
{
        clrscr();
        showsymbol obj1;     //creating object of showsymbol class
        cout<<"Enter the no. of occurrences= ";
        obj1.input();     //accessing input function from main
        cout<<"The output is:"<<endl;
        obj1.show();     //accessing show function from main
        getch();
}
```

Output:

```
Enter the no. of occurrences= 10
The output is:
**********
```

Explanation:

- ✓ The val is declared as private data-member of the class showsymbol.
- ✓ The input() and show() functions are declared as public member-function of the showsymbol class.
- ✓ The ***val*** data-member is used directly inside the above two member-functions as these functions are members of the same class showsymbol.
- ✓ In the main() function, an object is created, named **obj1**, of type showsymbol class. Thus, memory space is allocated for the object.
- ✓ In the main() function we are calling/using the member-functions of showsymbol class, input() and show(), for the object **obj1** using the '**.**' operator.
- ✓ Now, we run the program:
    - o Displays the message "Enter the no. of occurences= ".
    - o Now, the program is waiting for an integer input. This is due to the call of the **input()** public function for the object **obj1**. Which in turn actually executes the "cin>>val" statement,for the**val** private member of **obj1**.
    - o When user gives the input as **10**, this value is assigned to **val** member of **obj1**.
    - o Now, the message "The output is:" is displayed.
    - o The **show()** function for **obj1** is now called and which in turn shows ten(10) "*" symbols in the same line as the **val** contains the value **10**.

    (The uses of clrscr() and getch() library functions are already discussed in earlier units.)

7.8 CONSTRUCTORS AND DESTRUCTORS

7.8.1 Constructors

Constructor is a special kind of member function of a class which has the same name as the class and also has no return type. This is implicitly called when an object is created. When we declare an object of a class this member function is automatically called.

A object may require initialization of the data members, to do so constructors are used. It is to be noted that when we create a class

without defining a constructor, as in showsymbol class in Program-1, a **default constructor** is automatically created by the compiler.

A **Default Constructor** does not have any parameter.It is necessary when we want to initialize the data-members with default values.The constructors those are defined with parameters are known as **Parameterized Constructors**.

Let's try to understand all these with the example shown in Program-2 (considering the **showsymbol** class, with modifications, from Program-1).

Program-2:

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

class showsymbol
{       private:
          int val;   //private data-member
        public:
      showsymbol();   //default constructor
      void input();//public member-function
      void show();//public member-function
};

showsymbol::showsymbol()     //definition of default consturctor
{   val=10;}

showsymbol::showsymbol(int n)     //definition of parameterized
                                  //consturctor
{   val = n;}

void showsymbol::input()     //definition of input public member-
                             //function
{cin>>val;}

void showsymbol::show()    //definition of show public member-
                           //function
{
int i;
   for(i=0; i<val; i++)
   {cout<<"*  ";}
```

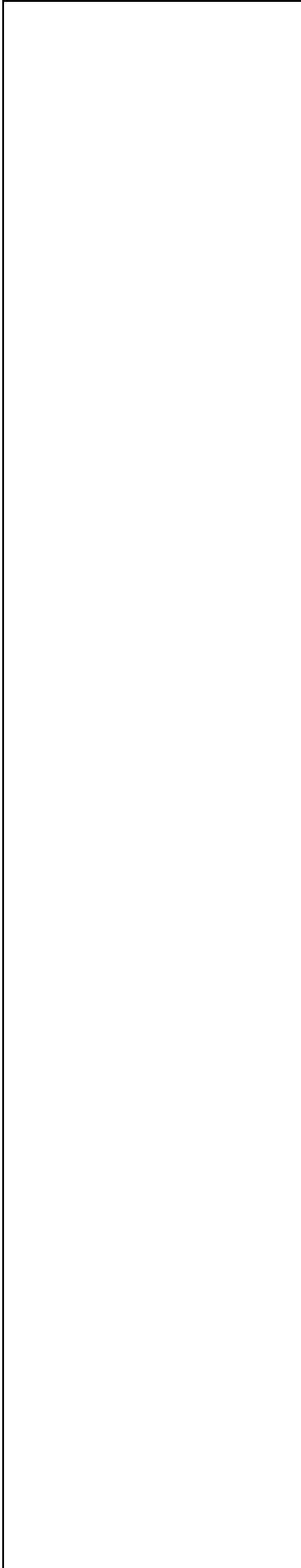
```
}
void main()
{
        clrscr();
        showsymbol obj1, obj2(20);          //creating object
        cout<<"The output for obj1:"<<endl;
        obj1.show();    //accessing show function of obj1
        cout<<"The output for obj2:"<<endl;
        obj2.show();      //accessing show function of obj2
        cout<<"Enter the no. of occurrences for obj1= ";
        obj1.input();       //accessing input function of obj1
        cout<<"Enter the no. of occurrences for obj2= ";
        obj2.input();       //accessing input function of obj2
        cout<<"The output for obj1:"<<endl;
        obj1.show();      //accessing show function of obj1
        cout<<"The output for obj2:"<<endl;
        obj2.show();      //accessing show function of obj2
        getch();
}
```

Output:

```
The output for obj1:
**********
The output for obj2:
********************
Enter the no. of occurrences for obj1= 5
Enter the no. of occurrences for obj2= 15
The output for obj1:
*****
The output for obj2:
***************
```

Explanation:

- ✓ A **Default Constructor** is defined where we initialize the value of **val** with **10**, i.e., whenever we declare an object the default value of **val** is **10**.
- ✓ A **Parameterized Constructor** is also defined where we initialize the value of **val** with the value in the parameter **n**, i.e., whenever we declare an object using this constructor the default value of **val** is the value in parameter **n**.



- ✓ When we declare the object **obj1**, the **val** member is set with the **10** as the constructor gets called is the **default constructor**.
- ✓ When we declare the object **obj2**, the **20** mentioned within **()** is the value for the parameter '**n**' of the parameterized constructor. And thus,**val** member is set with the **20** as here the **parameterized constructor** gets called.
- ✓ Thus for
    - **obj1**, **10 (ten)** '*' are displayed.
    - **obj2**, **20 (twenty)** '*' are displayed.
- ✓ Now, we take inputs for **obj1** and **obj2**, which are **5** and **15** respectively. This means the value of **val** for **obj1** is **5** and for **obj2** is **15**.
- ✓ Thus for
    - **obj1**, **5 (five)** '*' are displayed.
    - **obj2**, **15 (fifteen)** '*' are displayed.

STOP TO CONSIDER

A class can be defined with more than one parameterized constructor (with different signatures) but with only one default constructor.

As of now, constructors have some special characteristics and these are:

- ➢ They are used to initialize the mainly the data-members of objects.
- ➢ They should have the same name as the class.
- ➢ They should be declared as public member functions.
- ➢ They have no return types.
- ➢ They are automatically gets called when objects are created.

STOP TO CONSIDER

A constructor is automatically gets called when an object is created(declared). This object creation is generally done in functions those are not part of the class of which the object is declared. Hence, the constructors should be declared under public access.

Now, we will discuss about the **implicit** and **explicitconstructor call**. Considering the **showsymbol** class in Program-2,the objects' declaration

showsymbol obj1, obj(20);

is known as implicit constructor calling. We also know that, as discussed in **section 7.5**, the objects' declaration statement(s) can be written as:

showsymbol obj1 = showsymbol();

showsymbol obj2 = showsymbol(20);

Here, the constructors are called explicitly (explicit constructor call).

7.8.2 Destructor

It is used to destroy objects. Destructor isalso a special member function of a class which has the same name as the class preceded by '~' (tilde) symbol. A destructor does not take parameters and also has no return type.

The example of a destructor is:

~showsymbol() {

}

Like constructor, if we do not explicitly define a destructor the compiler will implicitly define a destructor. The definition of a destructor is of utmost need when we allocate memory during object construction. Consider the Program-3.

Program-3:

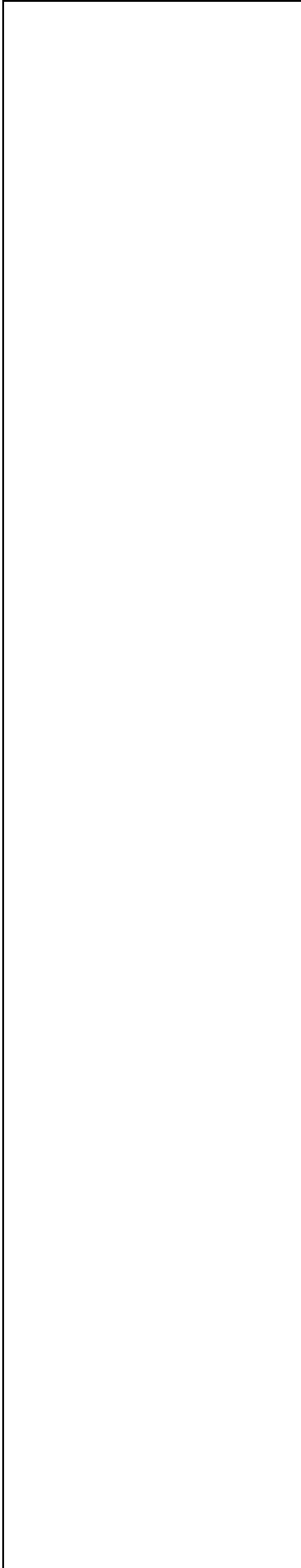
```
#include <iostream.h>
#include <conio.h>
#include <string.h>

class test {
   private:
   char *str;
   public:
   test (char*);    //Constructor
      showdata();     //Member function
   ~test();           //Destructor
};

test::test(char*data)     //Definition of constructor
{
str = new char[strlen(data)+1];
strcpy(str, data);
```

```
}

test::showdata()     //Definition of member function
{
  cout<<"The string inside is= "<<endl<<str;
}

test::~test()     //Definition of destructor
{
delete[] str;
}

void main () {
    clrscr();
  test obj1 = test("GUIDOL");      //Creation of object
obj1.showdata();              //Display the string data inside object
getch();
}
```

Explanation:

- ✓ Inside the **Constructor**,
    - First, we have dynamically allocated memory for the parameter to be passed using the length of the parameter.
    - Next, we assign the string value of the parameter to the allocated space using strcpy() function.
- ✓ In the **showdata()** member function, the string value inside the**str** data-member is displayed.
- ✓ Inside the **Destructor**, the allocated memory for the **str** data-member is deleted.
- ✓ Inside the **main()** function,
    - First, an object (**obj1**) is created and thus "GUIDOL" string is assigned to the **str** member of **obj1**.
    - Next, the string data inside str member of obj1 is displayed which is obviously "GUIDOL".
    - At last, the main() ends, i.e., the program ends.

Now, the question is when will be the destructor called to free-up the allocated memory? The answer is that when the object goes out-of-scope the destructor will be called implicitly and here in our case



the destructor is executed when end of the program reached, i.e., the end of **main()** function.

7.9 FRIEND FUNCTION

In C++, an outside function can be made friend to a class. The function that is made friend to a class is termed as **Friend Function** to that particular class. To make a function friend to a class we just have to declare the function as friend, using **friend keyword**, within the class definition. The syntax for declaring a function friend of a class is:

friend return-type function-name(arguments);

where return-type, function-name and arguments are the return type, name and argument of the friend function.A Friend Function can be declared under public or private access of a class.

Few points related to Friend Function are:

- ✓ A FriendFunction to a class is not within the scope of the classi.e., not a member-function of that class.
- ✓ As it is outside the scope-of the class therefore it cannot be invoked using objects of that class.
- ✓ It can only be called in the same way as a normal function.
- ✓ It can access the **private** and **protected** members of the class only by using an object of that class. Therefore, generally it has object of that class as argument.

The following program is an example for friend function.

Program-4:

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

class number {
    private:
      intnum;
    public:
      number (int);    //Constructor
int returndata();
        void showdata();
friend int sumnumbers(number, number);
```

```
};

number::number(int n)
{
num=n;
}

number::showdata()
{
cout<<num<<endl;
}

int  sumnumbers(number a, number b)
{
      int  res;
      res = a.num + b.num;
      return  res;
}

void main () {
     clrscr();
numberx(50),  y(100);
     int result;
cout<<"The value inside object x= ";
     x.showdata();
     cout<<"The value inside object y= ";
     y.showdata();
 int sumresult;
     result = sumnumbers(x, y);
     cout<<"The result of summation = "<<result;
getch();
}
```

STOP TO CONSIDER

A normal function can be made friend to any number of classes.

CHECK YOUR PROGRESS - III

1. What is constructor?
2. When is the default constructor invoked?
3. What is parameterized constructor?

State TRUE or FALSE:

4. A destructor has a return type.
5. In every class there must be a declaration for destructor.
6. Constructor cannot have arguments.
7. friend keyword is necessary for declaring a friend function.
8. A friend function can directly access the private members of the class to which it is declared as friend.
9. Division by zero (0) is a _______________ error.

7.10 SUMMING UP

This Unit contains discussions related to Classes and Objects.

- Classes are basically the blueprints of user defined data-types.
- A variable of a class type is termed as objects of that class.
- In C++, **class** keyword is used to define a class and the definition ends with a "**;**".
- The access specifiers in C++ are private, protected and public.
- The member functions of a class can be defined out the class definition using the scope resolution operator "**::**".
- Constructors are special member function used to initialize the data-members while creation of an object.
- A Constructor has the same name as the class name and it does not have return type.
- Constructors are called implicitly and explicitly.
- A Default Constructor does not have parameters.

- For classes without default constructor, the compiler automatically creates one.
- Destructor is also a special member function of a class and is automatically gets called when an objectof that class goes out-of-scope.
- Destructor also has the same name as the class name and is preceded by the "~" symbol. It does not have return type as well as parameters.
- Friend Function is a normal function which is made friend to a class using the **friend** keyword.
- Friend function of a class can access the private and protected members of that class using the object(s) generally passed as parameter(s) to it.

7.11 ANSWERS TO CHECK YOUR PROGRESS

1. A constructor is a special kind of public member function of a class which has the same name as the class name and has not return type.

2. A default constructor is invoked when and object declaration statement without any parameter is executed.

3. A parameterized constructor is a constructor which has arguments.

4.False

5. False

6. False

7. True

8. True

7.12 POSSIBLE QUESTIONS

1. How does an object of a class is created?
2. What is the syntax of defining a class?
3. Write down the role of constructor during object creation.

4. Write down the differences between default constructor and parameterized constructor.
5. What is Friend Function? How a function can be declared as a friend to a class.

7.13 REFERENCES AND SUGGESTED READINGS

1. Stroustrup, Bjarne.*The C++Programming Language*.
2. Balagurusamy, E.. *programming in C++*. Tata McGraw-Hill Education.
3. Kanetkar, Y. P.. *Let us C++*. BPB Publications.

UNIT-8: INHERITANCE

Unit Structure:

8.1 Introduction
8.2 Unit Objectives
8.3 Concepts of Inheritance
8.3.1 Advantages of Inheritance
8.3.2 Disadvantages of Inheritance
8.4 Casting up the Hierarchy
8.4.1 Difference between Upcasting and Downcasting
8.4.2 Details of Upcasting
8.4.3 Details of Downcasting
8.5 Types of Inheritance
8.5.1 Single Inheritance
8.5.2 Multiple Inheritance
8.5.3 Multilevel Inheritance
8.5.4 Hierarchical Inheritance
8.5.5 Hybrid Inheritance
8.6 Importance of Access Specifiers in Inheritance
8.7 Virtual Base Class
8.8 Summing Up
8.9 Answers to Check Your Progress
8.10 Possible Questions
8.11 References and Suggested Readings

8.1 INTRODUCTION

In the earlier chapters, the learners have been acquainted with various important aspects of object oriented programming (OOP) involving class, object, data members, member functions, data abstraction, data hiding, constructor, destructor, access specifiers, friend functions, polymorphism, etc. Apart from all these, inheritance plays a major role in implementing OOP in real sense, because this property of OOP is much closer to real life requirements of the current day software users. The code reusability and code optimization are two highly appreciated domains in the field of computer science. Both these two issues are well addressed by inheritance in true sense. If some codes are already developed, then there is no question of re-writing the similar codes again and

again, but the extension works need to be addressed in some other module and both modules could be tied up as and when needed. This is the basis of inheritance.Although, there are a lot of advantages in implementing inheritance, there are certain disadvantages too, which are all discussed in this chapter. The up-casting and down-casting in class hierarchy are two another key terms and they are addressed here. There are many types of inheritances ranging from single inheritance to hybrid inheritance. All these types of inheritances are discussed here with their coding syntax and appropriate example programs using C++ codes. The access specifiers available in OOP languages have a vital role in inheritance, because these are the programming tools that can restrict access to some private data from unauthorized access despite of the smooth maintaining of the class hierarchy. This aspect is covered in the chapter followed by the discussion on importance of virtual base class in inheritance.

8.2 UNIT OBJECTIVES

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

- Learn the importance of Inheritance in OOP languages.
- Know about the advantages and disadvantages of Inheritance.
- Acquire knowledge on Casting-up and Casting-down in Inheritance.
- Know about various types of Inheritance with definition and syntax.
- Know various types of Inheritance with example programs.
- Know the importance of various access specifiers in implementing Inheritance.
- Learn the importance of virtual base class in Inheritance.

8.3 CONCEPTS OF INHERITANCE

Inheritance is one of the most important properties of Object Oriented Programming (OOP) paradigm. The concept of extending classes is the basis of this mechanism. Inheritance is such a mechanism through which the properties from one class (base class) is inherited to another class (derived class). In another point of view,

we can say that the features and behaviors of a class are acquired by another class when we implement inheritance in a program. The class whose members are inherited is called the base class or super class and the class that inherits those members is called the derived class or sub class.

Definitions:

- **Base class**: The parent class whose properties are inherited by another class is called the base class. It is also termed as a **Super class.**
- **Derived class**: The class in which the properties are inherited from the parent class is called the Derived class. It is also termed as a **Sub class.**

It is to be understood that the classes are some abstract units. These classes do not take part in programming in a direct manner. But the instances of classes, which are popularly known as **objects,** are the main participatory units in the implementation of object oriented programming. We can come to know about an object by knowing its class also.

Let us suppose that we are not familiar with someone among the employees of an educational institute. But, if you come to know that he belongs to that institute, we would come to know that he has an employee-id, employee-name, salary and date-of-joining. In the advanced level of object oriented programming, these classes can be defined in terms of other connected classes. For example- teachers, library staffs, office staffs and security staffs are generally considered as some employees of an educational institution. In object oriented terminology, the teachers, library staffs, office staffs and security staffs are all *derived classes* or *sub classes* of the *employee* class. Similarly, the *employee* class is the *base class* or *super class* of teachers, library staff, office staff and security staff. The hierarchy of classes plays an important role in the implementation of inheritance.

8.3.1 Advantages of Inheritance

- The subclassesderive all the properties from the super class, thereby reusing the existing code.
- Programmers can reuse the codes in the superclass or base class as many times as the number of derived classes being formed.
- There will not be any wastage of memory space because the same properties are inherited and not duplicated.
- This also leads to faster progression and development time.
- Code enhances maintenance and memory utilization.
- Reduces code redundancy and enhances code reusability.
- Reduces source code size and improves code readability.

8.3.2 Disadvantages of Inheritance

- As the base class and the child class are tightly coupled in inheritance, hence any changes made in the codes of parent classes affect the child classes.
- In a class hierarchy, many data members remain unused and the memory allocated to them remains unutilized.
- The above mentioned unutilized memory affects the performance of the program if proper care is not taken in its implementation.

8.4 CASTING UP THE HIERARCHY

The OOP languages allow a derived class pointerto be treated as a base class pointer. This is called upcasting. Downcasting is an opposite process, which consists of converting base class pointer to derived class pointer.

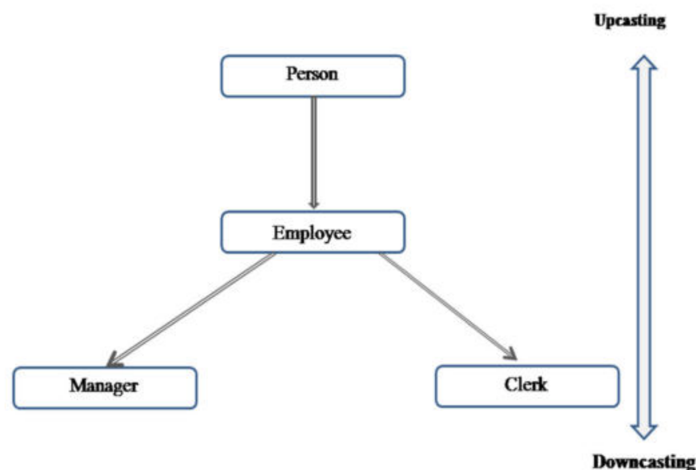


Fig-8.1: Upcasting and Downcasting in Inheritance

8.4.1 Difference between Upcasting and Downcasting

Definitions:

- **Upcasting:** Casting a derived class pointer to a base class pointer is known as upcasting. The figure below depicts the upcasting of derived class-1 pointer/reference to the base class pointer/reference (derived class 1 -> base class).

- **Downcasting:** Casting a base class pointer to a derived class pointer is known as downcasting. The figure below depicts the Downcasting of the base class pointer/reference to the derived class-2 pointer/reference (base ->derived class-2).

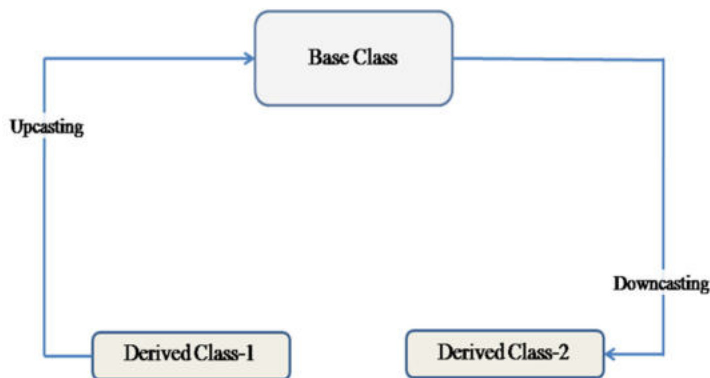


Fig-8.2: Casting in Inheritance

8.4.2 Details of Upcasting

The need of upcasting arises when we have to cast a subclass to a superclass.Upcasting is sometimes called widening. It happens automatically and no explicit work needs to be performed. Upcasting facilitates the access of parent class members, but it is not possible to access all the child class members on the contrary. Instead of all the members, we can access some specific members of the child class.

The following C++ code segment deals with three different shapes. The *Shape* class is defined first and the three classes *Circle*, *Triangle* and *Rectangle* are derived from it. A member

function is defined within each of the derived classes for communicating with the base class.

```
void play(Shape& s)
{
   s.draw();
   s.scaleup();
   s.scaledown();
   ....
}
```

This member function corresponds to any shape, so it is independent of the specific type of object (circle, triangle or rectangle)for the purpose of the actions i.e. drawing, scaling up or scaling down. Suppose, in some part of the program, we are using the play() function as mentioned in the following code.

```
Circle cir;
Triangle tri;
Rectangle rec;
play(cir);
play(tri);
play(rec);
```

Here, a circle object is being passed into a function that is asking for a shape as argument. Since a circle is a shape, it can be treated as the one to whom, the play() function needs to respond.Upcasting allows us to treat a derived type as though it were its base type. That is how we abstain ourselves from knowing what the exact type we are dealing with. This indicates that the play() function has no specific coding regarding a circle, or a triangle or a rectangle. If we proceed for writing that kind of code, which checks for all the possible types of shapes, it will become a messy code, and we need to change it every time we add a new kind of shape. In these codes, however, we are treating that each shape can be drawn, scaled up and scaled down.

Because implicit upcasting makes it possible for a base-class pointer (reference) to refer to a base-class object or a derived-class object, there is the need for dynamic binding. Virtual member functions are useful for implementing dynamic binding. If a member function

is virtual, then when we send a message to an object, the object will do the right thing, even when upcasting is involved. Note that the most important aspect of inheritance is not that it provides member functions for the new class, however, it is the relationship expressed between the new class and the base class.

```
class Parent
{
public:
void sleep() {}
};
class Child: public Parent
{
public:
void playVideoGame(){}
};
void main( )
{
  Parent parent;
  Child child;
  Parent *pParent = &child; //upcast-implicit type
  Child *pChild =(Child *)&parent;//downcast-explicit type
  pParent -> sleep();
  pChild ->playVideoGame();
}
```

A Child object is a Parent object, where, it inherits all the data members and member functions of the Parent object. So, anything that we can do to a Parent object, we can do to a Child object. Therefore, a function designed to handle a Parent pointercan perform the same acts on a Child object without any problem. The same idea is applicable if we pass a 'pointer to an object' as a function argument. Upcasting is transitiveif we derive a Child class from Parent, then Parent pointercan refer to a Parent or a Child object.

8.4.3 Details of Downcasting

The opposite process of Upcasting i.e. the conversion of base-class pointerto a derived-class pointer is called Downcasting.When we want to cast a super class to a sub class, we use Downcasting (or narrowing), and Downcasting is not directly possible. For example, in Java, it is done in an explicit manner.The Downcasting operator in C++ is basically extraordinarily slow compared to the performanceof other operators due to the fact that C++ allows multiple and virtual-inheritance.

Downcasting is not allowed without an explicit type cast. The reason for this restriction is that the '*is-a*' relationship is notsymmetric in most of the cases. A derived class could add new data members, and the class member functions that used these data members would not be able to apply onto the base class.As in the example, we derived Child class from a Parent class, adding a new member function, playVideoGame(). It would not make sense to apply the playVideoGame() function to a Parent object. However, if implicit downcasting were allowed, someone could accidentally assign the address of a Parent object to a pointer-to-Child i.e.

```
Child *pChild =  &parent; //can't convert from 'Parent *' to 'Child *'
                                            //error will arise here…
```

and use the pointer to invoke the playVideoGame() method as shown below.

```
pChild ->playVideoGame();
```

As the Parent need not contain a playVideoGame() function, the Downcasting in the above code segment could lead to some insecure operation. C++ provides a special explicit cast called *dynamic_cast* that performs this conversion.

Downcasting is the opposite of the basic object-oriented rule, which states objects of a derived class, can always be assigned to variables of a base class.The need for *dynamic_cast* generally arises when we need to perform derived class operations on a derived class object, but we avail only a pointer to a base class only.

8.5 TYPES OF INHERITANCE

There is a need of various forms of inheritances in order to meet the real life needs of the current day software developers as well as the users. Some of the most appreciated forms of such inheritances ranging from single inheritance to hybrid inheritance are described below with necessary diagrams and suitable example programs.

- Single Inheritance

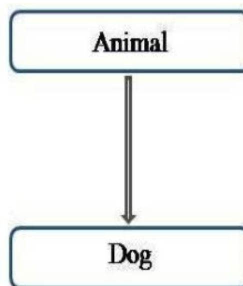


Fig-8.3: Single Inheritance

- Multiple Inheritance

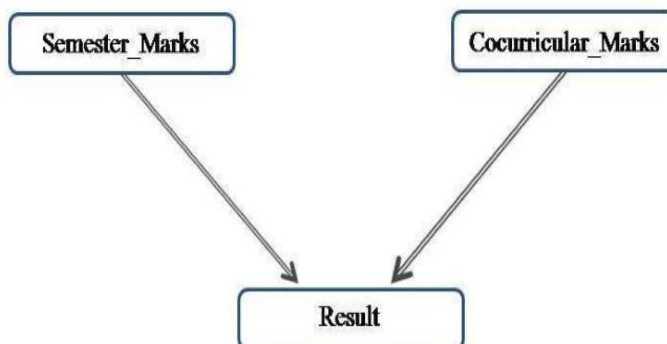


Fig-8.4: Multiple Inheritance

- Multi-level Inheritance

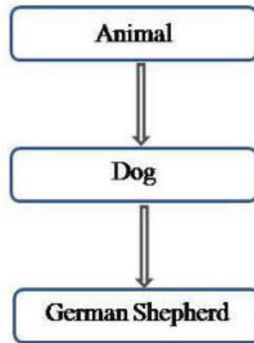


Fig-8.5: Multilevel Inheritance

- Hierarchical Inheritance

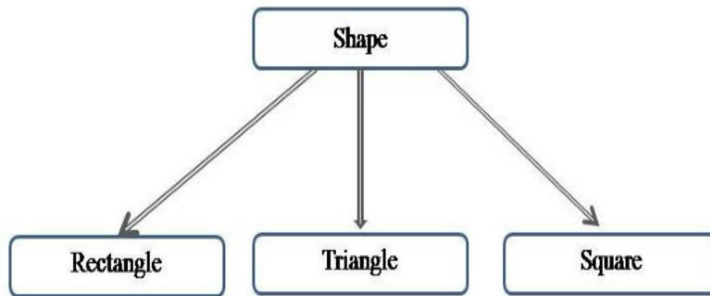


Fig-8.6: Hierarchical Inheritance

- Hybrid Inheritance

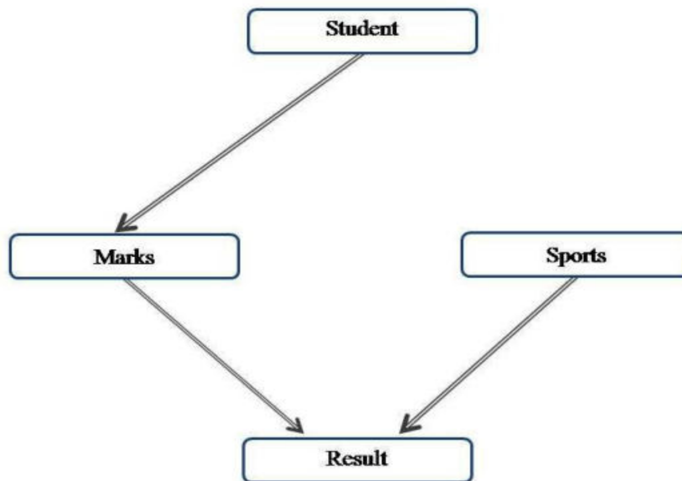


Fig-8.7: Hybrid Inheritance

8.5.1 Single inheritance

The simplest form of inheritance is the single inheritance. In this inheritance, a derived class is formed from a base class.In the example shown below, the class *Animal* is the base class or parent class and the class *Dog*is the derived class or child class. Here, the class *Dog*inherits the features and behaviours of the parent class *Animal*.Single inheritance is implemented in C++ as shown below.

Given below is a C++ based programming example of Single Inheritance.

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

class Animal
{
   private:
      string name="";
   public:
      int no_of_legs=4;
int no_of_tail=1;
};

class Dog : public Animal
{
   public:
      void act()
      {
          cout<<"\n The dog barks !!!";
```

```
    }
};

void main()
{
   Dog dog;
   cout<<"\n The dog has "<<dog.no_of_legs<<"
legs";
   cout<<"\n The dog has "<<dog.no_of_tail<<"
tail";
   dog.act();
}
```

Output:

```
The dog has 4 legs
The dog has 1 tail
The dog barks!!!
```

The class *Animal*is a base class and the class *Dog* is derived from *Animal* class. The class *Dog* inherits all the members of *Animal* class and can avail its own properties, which is clear from the output above.

8.5.2 Multiple inheritance

In this inheritance, a derived class is created from more than one base class. Although this form of inheritance is directly supported in C++, thisis not directly supported by Java and .NET Languages like C#, F# etc.In the given example, class *Result* inherits the properties and behaviours of the two base classes i.e.*Semester_Marks* and *Cocurricular_Marks* at same level. So, the class *Result* is the derived class here.Following is an example program to demonstrate Multiple Inheritance in C++.

```
#include <iostream.h>

class Semester_Marks
{
protected:
int rollNo,sub1, sub2;
public:
void getsmarks()
{
cout << "\n Enter the Roll No : ";
cin >> rollNo;
cout << "\n Enter the two best marks: ";
cin >>sub1>>sub2;
   }
};
```

```
class Cocurricular_Marks
{
protected:
int comarks;
public:
void getcomarks()
{
cout << "Enter the mark for cocurricular
activities: ";
cin >> comarks;
   }
};

class Result : public Semester_Marks, public
Cocurricular_Marks
{
  private:
   int total_marks, avg_marks;
  public:
   void display()
   {
      total_marks = sub1 + sub2 + comarks;
      avg_marks = total_marks / 3;
      cout << "\nRoll No: " << rollNo ;
      cout<< "\n Total Marks: " << total_marks;
      cout << "\n Average Marks: " << avg_marks;
   }
};


void main()
{
   Result res;
res.getsmarks();
res.getcomarks();
res.display();
}
```

Output:

```
Enter the Roll No: 25
Enter the two best marks: 40  50
Enter the mark for Cocurricular activities: 30
Roll No: 25
Total marks: 120
Average marks: 40
```

In the above example, we have three classes i.e. Semester_Marks, Cocurricular_Marks, and Result. The class Semester_Marks reads two of the semester best marksof the student. The class Cocurricular_Marks reads the markof the student in co-curricular activities.The Result class calculates the total_marks and avg_marks on receiving inputs from the two base classes.In this model, Result class is derived from Semester_Marks and Cocurricular_Marks as

we calculate Result from the semester marks as well as co-curricular activities marks.This exhibits the multiple inheritance neatly.

8.5.3 Multi-level inheritance

In this inheritance, a derived class is created from another derived class.In the given example, class *German_Shepherd* inherits the properties and behavior of class *Dog* and the class *Dog*inherits the properties and behavior of another class *Animal.* So, here *Animal* is the parent class of *Dog* and *Dog*is the parent class of *German_Shepherd*. So, here class *German_Shepherd* implicitly inherits the properties and behavior of class *Animal* along with class *Dog.*Thus it exhibits multi level inheritance. Following is an example C++ program to demonstrate Multilevel Inheritance.

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

class Animal
{
      private:
         string name="";
      public:
         int no_of_legs=4;
         int no_of_tail=1;
};


class Dog : public Animal
{
   public:
         void act()
         {
            cout<<"barksmuch !!!";
         }
};


class German_Shepherd :public Dog
{
   public:
        void behaviour()
      {
            cout<<"is very aggressive !!!";
   }
};

void main()
{
      German_Shepherdgs;
```

```
      cout<<"\n German Shepherd has
"<<gs.no_of_legs<<" legs";
      cout<<"\n German Shepherd has
"<<gs.no_of_tail<<" tail";
      cout<<"\n German Shepherd ";
      gs.act();
      cout<<"\n German Shepherd ";
      gs.behaviour();
}
```

Output:

```
German Shepherdhas 4 legs
German Shepherdhas 1 tail
German Shepherdbarksmuch !!!
German Shepherdis very aggressive !!!
```

This kind of inheritance can be treated as the extension of the Single inheritance. Here, a class *German_Shepherd*inherits from the class *Dog* that in turn inherits from the class *Animal*. We see here that the class *German_Shepherd*inherits the properties and methods of both the upper hierarchy classes above it, i.e. the *Dog*and the *Animal*class. The class *Dog*acts as a base class for *German_Shepherd* and acts as a derived class for the *Animal* class.

8.5.4 Hierarchical Inheritance

In this inheritance, more than one derived classes are generated from a single base class.In the given example, class *Shape (base class)* has three child or three derived classes i.e. Rectangle, Triangle and Square.Following is an example C++ program to demonstrate the Hierarchical Inheritance.

```
#include <iostream.h>
class Shape
{
      public:
            int x,y;
            void getdata(int n,int m)
            {
                  x= n;
                  y = m;
            }
};
class Rectangle : public Shape
{
      public:
            int area_rec()
            {
```

```
                int area = x*y;
                return area;
            }
};

class Triangle : public Shape
{
      public:
            int area_tri()
            {
                float area = 0.5*x*y;
                return area;
            }
};

class Square : public Shape
{
      public:
            int area_squ()
            {
                float area = x*x;
                return area;
            }
};

void main()
{
      Rectangle r;
      Triangle t;
      Square s;
      int length,breadth,base,height,side;

      cout << "\n Enter the length and breadth of
      the rectangle: "; cin>>length>>breadth;
      r.getdata(length,breadth);
      int rec_area = r.area_rec();
      cout << "\nArea of the rectangle = "
<<rec_area;

      cout << "\n Enter the base and height of
      the triangle: "; cin>>base>>height;
      t.getdata(base,height);
      float tri_area = t.area_tri();
      cout <<"\n Area of the triangle = "
<<tri_area;

      cout << "\n Enter the length of one side of
      the square: "; cin>>side;
      s.getdata(side,side);
      int squ_area = s.area_squ();
```

```
      cout <<"\n Area of the square = " <<
squ_area;
}
```

Output:

```
Enter the length and breadth of the rectangle: 10
5
Area of the rectangle = 50
Enter the base and height of the triangle: 10 5
Area of the triangle = 25
Enter the length of one side of the square: 5
Area of the square = 25
```

The above example is a classic example of Hierarchical Inheritance. We have a base class *Shape* and three derived classes i.e. *Rectangle*, *Triangle* and *Square*.While a method is used to read data in the *Shape* class, each of the three derived classes has its own method to calculate area. In the main function, data is read for each object and then calculation of area is done.

8.5.5 Hybrid inheritance

This is combination of more than one inheritance. Hence, it may be a combination of Multiple and Multilevel inheritance or Multilevel and Hierarchical inheritance or Hierarchical, Multilevel and Multiple inheritance.Since .NET Languages like C#, F# etc. do not support multiple inheritances, hence hybrid inheritance with a combination of multiple inheritancesis not supported by .NET Languages. Following is an example C++ program to demonstrate Hybrid Inheritance.

```
#include <iostream.h>
#include <string.h>
class Student
{
      private:
            int id;
            char name[];
      public:
            void getstudent()
      {
                  cout << "\n Enter student Id
and student name : ";
                  cin >> id >> name;
      }
};
```

```
class Marks: public Student
{
      protected:
           int phy, chem, math;
      public:
           void getmarks()
      {
                   cout << "\n Enter marks for
                   Physics, Chemistry &
                   Mathematics :";
                   cin >>phy>>chem >> math;
      }
};

class Sports
{
      protected:
            int spmarks;
      public:
            void getsports()
            {
                   cout << "\n Enter sports
mark:";
                   cin >> spmarks;
            }
};


class Result : public Marks, public Sports
{
      private:
            int total_marks;
            float avg_marks;
      public :
            void display()
            {
                   total_marks=phy+ chem + math +
spmarks;
                   avg_marks=total_marks/4.0;
                   cout << "\n Total marks =" <<
total_marks;
                   cout << "\n Average marks = "
<<avg_marks;
            }
};

void main()
{
           result res;
```

```
                res.getstudent();
                res.getmarks();
                res.getsports();
                res.display();
}
```

Output:

```
Enter student Id and student name : 25 Abhinav
Enter marks for Physics, Chemistry & Mathematics
:80 84 86
Enter sports mark:90
Total marks =340
Average marks =85
```

Here we have four classes i.e. *Student, Marks, Sports*and *Result*. The class*Marks*is derived from the *Student* class. On the other hand, the class *Result*is derived from *Marks* and *Sports*class.Finally, we calculate the result from the subject marks plus the sports mark.The output is generated by creating an object of class *Result* that has acquired the properties of all the other three classes indirectly.

8.6IMPORTANCE OF ACCESS SPECIFIERS IN INHERITANCE

There is an important role of access specifiers in inheritance. The action of access specifiers come into force when derived classes want to access the members of a class.When the derived classes inherit members, those members may change access specifiers in the derived class. This does not alter the access specifier type of the own members (non-inherited) of the derived classes.

If the inheritance is protected, its children are aware that they are inheriting from one base class. If the inheritance is private, no one other than its own object is aware of the inheritance. All the three types of inheritances in terms of access specifiers are discussed below.

i) Public Inheritance − When deriving from a public base class, public and protected members of the base class remains the same in the in the derived class.

Access specifier in base class	Access specifier when inherited publicly
Public	Public
Protected	Protected
Private	Inaccessible

Table-8.1: Status of access specifiers *in derived classes* when inherited *publicly*

ii) Protected Inheritance − When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

Access specifier in base class	Access specifier when inherited protectedly
Public	Protected
Protected	Protected
Private	Inaccessible

Table-8.2: Status of access specifiers *in derived classes* when inherited *protectedly*

iii) Private Inheritance − When deriving from a private base class, public and protected members of the base class become private members of the derived class.

Access specifier in base class	Access specifier when inherited privately
Public	Private
Protected	Private
Private	Inaccessible

Table-8.3: Status of access specifiers *in derived classes* when inherited *privately*

8.7 VIRTUAL BASE CLASS

As we can see in the figure below that the data members or the member functions of class A are inherited in a redundant way to class D via class B and class C. So, when any data member or member function of class A is accessed by an object of class D, it is very sure to have ambiguityas to which data member or member

function would have to be called. This confuses the compiler and it displays error.To resolve this ambiguity when class A is inherited in both class B and class C, it is declared as virtual base class by placing a keyword virtual as:

Syntax 1:

```
class B : public virtual A
{
};
```

Syntax 2:

```
class C : public virtual A
{
};
```

An example C++ program is shown below to have a clear understanding of the virtual base class concept.

```
#include <iostream.h>
class A
{
      public:
      int a;
      A() // constructor definition
      {
            a = 10;
      }
};

class B : public virtual A
{
};

class C : public virtual A
{
};

class D : public B, public C
{
};

int main()
{
      D object;               //      object
creation of class D
      cout << "a = " << object.a << endl;
      return 0;
}
```

Output:

```
a=10
```

The keyword *virtual* can be written before or after the public. Now only one copy of the data member or the member functionwill be copied to class B and class C and class A becomesthe virtual base class. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

CHECK YOUR PROGRESS

Multiple choice questions:

1. Which among the following best describes the Inheritance?
   i) Copying the code already written
   ii) Using the code already written once
   iii) Using already defined functions in programming language
   iv) Using the data and functions into derived segment

2. Which among the following is correct for a hierarchical inheritance?
   i) Two base classes can be used to be derived into one single class
   ii) Two or more classes can be derived into one class
   iii) One base class can be derived into other two derived classes or more
   iv) One base class can be derived into only two classes

3. Which type of inheritance leads to diamond problem?
   i) Single
   ii) Multiple
   iii) Multilevel
   iv) Hierarchical

4. What is upcasting?

i) Casting subtype to supertype
ii) Casting super type to subtype
iii) Casting subtype to super type and vice versa
iv) Casting anytype to any other type

5. Which among the following is safe?

i) Upcasting
ii) Downcasting
iii) Both upcasting and downcasting
iv) If upcasting is safe then downcasting is not, and vice versa

State whether *True* or *False:*

1. Multiple inheritance is supported by almost all high level languages.
2. Private data members are not accessible from derived classes.
3. Inheritance reduces code redundancy and enhances code reusability.
4. Casting a base class pointer to a derived class pointer is known as upcasting.
5. Private inheritance in terms of accessibility is better than protected inheritance.

Fill in the blanks:

1. The class from which other class inherits is ________ .
2. The inheritance type where a derived class is inherited from many base classes is _________ inheritance.
3. Inheritance is __________ in nature.
4. A/An ____________ class is one that cannot be instantiated.
5. __________ contains data members only.

Match the following:

1. Object	i)Inheritance
2.Virtual	ii)Access specifier
3. Inheritance	iii)Dynamic cast
4. Protected	iv)Ambiguity removal

5. Downcasting v)Scope resolution operator

vi)Reusability

vii) Class instantiation

8.8 SUMMING UP

This unit begins with the basic concept of inheritance with emphasis on its significance in object oriented programming. The advantages and disadvantages of inheritance are discussed. Casting up in the hierarchy is discussed and the terms Upcasting and Downcasting.There are various types of inheritance. Single inheritance is the basic of all and the other inheritances are multiple inheritance, multilevel inheritance, hierarchical inheritance and hybrid inheritance. All these five types are discussed in this chapter with their implementation.. The access specifiers, i.e. public, private and protected have a vital role in inheritance. The derived classes can be inherited from its base class in public mode, protected mode and private mode. All these modes have different levels of access to the data herein. Virtual base class is another very important facility in OOP to get rid of ambiguity issues. The final section of this unit is wind up with the discussion on the detailed cause of ambiguity.

8.9 ANSWERS TO CHECK YOUR PROGRESS

Multiple choice questions:

1.(iv) 2. (iii) 3. (ii) 4.(i) 5. (i)

State whether *True* or *False:*

1. False 2. True 3. True 4. False 5. False

Fill in the blanks:

1. Base Class 2. Multiple 3. Transitive 4. Abstract 5. Structure

Match the following:

1. (vii)   2. (iv)   3. (vi)   4. (ii)
5. (iii)

8.10 POSSIBLE QUESTIONS

Short answer type questions:

1) What is the difference between a class and an object?
2) What is inheritance? Mention the types of inheritances.
3) What is the difference between Multiple Inheritance and Hierarchical Inheritance?
4) Differentiate between:
   i) Public access specifierand Private access specifier
   ii) Upcasting and Downcasting
   iii) Single Inheritance and Hybrid Inheritance
5) What measure will you take to resolve the issue of ambiguity in inheritance?

Long answer type questions:

1) Explain the importance of inheritance in object oriented programming. Also mention the advantages and disadvantages of inheritance.
2) Differentiate between Upcasting and Downcasting in inheritance with necessary code segments.
3) Explain various types of inheritances along with necessary diagrams to ease the process of understanding the concept.
4) Explain the three types of inheritances in terms of access specifiers (public, protected and private) that a derived class can undergo.
5) Explain the importance of virtual base class with suitable program.

8.11 REFERENCES AND SUGGESTED READINGS

- E. Balagurusamy (2010). Object Oriented Programming With C++. Tata McGraw hill Education Pvt. Ltd. p. 213. ISBN 978-0-07-066907-9.
- Dr. K. R. Venugopal, Rajkumar Buyya (2013). Mastering C++. Tata McGraw hill Education Private Limited. p. 609. ISBN 9781259029943.
- Mitchell, John (2002). "Concepts in object-oriented languages"". Concepts in programming language. Cambridge, UK: Cambridge University Press. p. 287. ISBN 978-0-521-78098-8.
- Bjarne Stroustrup. “The Design and Evolution of C++”. Addison-Wesley Professional. Publication: April 8, 1994. Edition - 1

UNIT 9: POLYMORPHISM

Unit Structure:

9.1 Introduction
9.2 Unit Objectives
9.3 Definition of Polymorphism
9.4 Compile time polymorphism
9.5 Function Overloading
9.6 Operator Overloading
9.6.1 Unary Operator Overloading
9.6.2 Binary Operator Overloading
9.6.3 Operator Overloading Using Friend Function
9.6.4 Data Conversion Using Operator Overloading
9.6.5 Overloadable and Non-Overloadable Operators in C++
9.7 Run time Polymorphism
9.8 Virtual Function
9.8.1 Implementation
9.8.2 Pure Virtual Function
9.9 Summing Up
9.10 Answers to Check Your Progress
9.11 Possible Questions
9.12 References and Suggested Readings

9.1 INTRODUCTION

In unit 1, four important properties of Object Oriented Programming (OOP) have been introduced. Polymorphism is one of these OOP properties. In this unit, different types of polymorphism are discussed with their implementations.

9.2 UNIT OBJECTIVES

After reading this unit, you are expected to be able to learn:

- ❖ What is Polymorphism?
- ❖ Different types of Polymorphism in OOP.
- ❖ Definition and types of Compile time polymorphism.
- ❖ Definition and implementation of function overloading.

- ❖ Definition of operator overloading.
- ❖ Implementations of unary and binary operator overloading.
- ❖ What is runtime polymorphism?
- ❖ Definition of Virtual function and its implementation.
- ❖ What is dynamic binding?
- ❖ Definition and implementation of pure virtual function?
- ❖ Explain Abstract class.

9.3 DEFINITION OF POLYMORPHISM

The word "Polymorphism" is originated from two Greek word "Poly" and "Morphe". In Greek, the word "Poly" means "many" and the word "Morphe" means "form". In Object Oriented Programming (OOP), the word "Polymorphism" is used to represent one of its properties. It allows same function name for different functionalities and also allows using one operator for performing multiple operations. It means that a same function name can be used to represent multiple operations and depending upon the type of parameters or number of parameters or object, a particular operation will be executed. Similarly, Polymorphism allows an operator to perform differently functions depending upon the type and number of function parameters of the called operator.

In OOP, Polymorphism can be categorized into two major groups that are Compile time polymorphism and Run time polymorphism.

9.4 COMPILE TIME POLYMORPHISM

In computer programming, linking of function call to the function definition or body is referred as binding. If compiler can determine a binding at the compile time then it is termed as static binding. It means that compiler can be able to recognize all necessary information to call a function at compile time in case of static binding. The advantage of static binding is that it increases the efficiency of the program because before execution all necessary information is recognized. On the other hand, due to static binding, the flexibility of the program is decreased.

As we know Polymorphism allows the use of same function name for multiple functions where each function must be different from other functions in terms of the number of parameters or type of parameters or both. In this case, binding of function call to its function body is determined by the compiler at the compile time on the basis of the number of parameters or types of parameters or both. So this type of polymorphism is called as Compile time polymorphism. It means Compile time polymorphism is realized due to static binding. Function overloading and Operator overloading are the two types of Compile time polymorphism.

9.5 FUNCTION OVERLOADING

We have already learnt from the earlier section that Function overloading is a Compile time polymorphism. Function overloading allows the use of multiple functions with same name in a program where each function provides different functionality from other functions. In this case, the name of multiple functions is same but they must be different in terms of number of parameters or types of corresponding parameters or both. Readability of program can be improved due to function overloading.

Now we will try to learn function overloading with an example. Let us consider the following C++ program as an example of function overloading.

Program 9.1: C++ Program to Estimate Areas of Triangle, Square, Circle and Parallelogram.

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

int Area ( int , int );             // Area of Triangle
int Area ( int );                   //  Area of Square
float Area ( float , float );       //  Area of Circle
int Area ( int , int , char [ ] );  //  Area of Parallelogram

int main( )
{
        int Base , Vertical_Height , Length_Side ;
```

```
        float Radius , Pi = 3.14 ;
        cout << "\n Area of Triangle:" ;
        cout << "\n Enter base(in Meter) = " ;
        cin >> Base;
        cout << "\n Enter vertical height (in Meter) = " ;
        cin >> Vertical_Height;
        cout << "\n Area of the triangle = " ;
        cout << Area( Base , Vertical_Height) << " Square 
Meters\n" ;

        cout << "\n Area of Square:" ;
        cout << "\n Enter length of a side (in Meter) = " ;
        cin >> Length_Side;
        cout << "\n Area of the Square = " << Area( Length_Side) ;
        cout << " Square Meters\n"   ;

        cout << "\n Area of Circle:" ;
        cout << "\n Enter radius( in Meter )  = " ;
        cin >> Radius ;
        cout << "\n Area of the Circle =" << Area(Radius , Pi) ;
        cout << " Square Meters\n"   ;
        cout << "\n Area of Parallelogram:" ;
        cout << "\n Enter base ( in Meter) = " ;
        cin >> Base ;
        cout << "\n Enter vertical height( in Meter)  =" ;
        cin >> Vertical_Height;
        cout << "\n Area of the Parallelogram = " ;
        cout << Area( Base , Vertical_Height , "Parallelogram" ) ;
        cout << " Square Meters"   ;

        getch( );
        return(0);
}

int Area ( int B , int H )          // Function to estimate area of 
Triangle
{
        return( ( B * H )/2 );
}

int Area ( int LS )          // Function to estimate area of Square
```

```
{
        return( LS * LS );
}

float Area( float R , float PI )     // Function to estimate area of Circle
{
        return( PI * R * R );
}

int Area( int B , int H , char S[ ] )    /* Function to estimate area of
                                              Parallelogram   */

{
        if( strcmp ( "Parallelogram" , S) = = 0 )
                return( B*H );
       else
                return 0;
}
```

Output of the above program:

Area of Triangle:
Enter base(in Meter) = 14
Enter vertical height (in Meter) = 12
Area of the triangle = 84 Square Meters

Area of Square:
Enter length of a side (in Meter) = 7
Area of the Square = 49 Square Meters

Area of Circle:
Enter radius(in Meter) = 4.7
Area of the Circle = 69.3977 Square Meters

cout << "\n **Area of Parallelogram:**" ;
Enter base (in Meter) = 20
Enter vertical height(in Meter) = 12
Area of the Parallelogram = 240 Square Meters ;

Function overloading has been demonstrated with the help of the above C++ program (Program 9.1). The job of this program is to estimate the areas of Triangle, Square, Circle and Parallelogram. It has been observed that there are four functions with same name ‘Area’. But each function is different from others in terms of type of corresponding parameters or number of parameters or both. So, area of a Square can be estimated by the function, Area( ) which has only one parameter and the type of parameter is int. On the other hand, the function, Area( ) with two int parameters will provide the area of a Triangle. So these two functions demonstrate function overloading using different number of parameters. Again the function, Area( ) with two float type parameters will provide the area of a Circle. In this case, it demonstrates function overloading using different types of parameters. Finally, the function, Area( ) with two int parameters and one string parameter will estimate the area of a Parallelogram. Now which function will be linked at the time of function call is dependent upon the parameters passed to it. For example, if we pass only one int parameter then the function, Area( ) which is used to calculate the area of a Square will be linked and the area of a Square will be returned after estimation. We can understand this function overloading concept very easily by observing the output of the program mentioned above.

9.6 OPERATOR OVERLOADING

We have already learnt in Unit 2 about different types of operators available in C++. Each of these operators can be used to perform some specific operation on primitive or built-in data types in Structured Programming Languages like C. For example, ++ (Increment Operator) can be used to increase the value of its operand by one. The data type of the operand can be either int or float or char.

In Object Oriented Programming (OOP), additional meanings to an operator can be defined so that it can also be used with user defined classes or user defined data types. It means one operator can be used to operate on primitive data types as well as on user defined data types also. This feature of OOP is called Operator overloading. To achieve Operator overloading, special member functions or friend

functions has to be defined in a class. So, depending upon the operands, an overloaded operator will perform its task. The syntax rules of overloaded operators are same as theoriginal operator. One more important point is that using Operator overloading, the basic meaning of an operator cannot be changed.

The compiler can recognize at compile time about the defined operation for a particular operator. It means that Operator overloading is also a type of compile time polymorphism.

In C++, the keyword '*operator*' is used to overload operators. The general syntax of operator overloading in C++ is presented as follows.

```
Return_Type  operator  OperatorSymbol ( [Argument List])
{
        // Body of the function to add new operation to the operator
}
```

In the above syntax, Return_Type refers the type of the data that is returned from the operator overloading function. If the function returns an object then Return_Type refers the class name of the returned object. If the function does not return anything then Return_Type will be void. Again in the above syntax, OperatorSymbol represents the operator which is to be overloaded. Finally, [Argument List] refers the parameters passed to the function as per requirement.

STOP TO CONSIDER

All Object Oriented Programming languages don't support Operator overloading. For example, Java don't support Operator overloading

9.6.1 Unary Operator Overloading

If the number of operand to an operator is only one, then the operator is called as unary operator. For example: ++, --, ~ are unary operators. The syntax to overload unary operators in C++ is presented as follows.

```
Return_Type  operator  Unary_Operator_Symbol ( )
{
```

```
            // Body of the function to add new operation to the operator
}
```

From the above syntax, it can be observed that in C++ programming, no parameter is passed to the overloading function to overload a unary operator. In the above syntax, Unary_Operator_Symbol refers to the unary operator which is going to be overloaded.

Now, we will try to understand how to overload a unary operator by observing the following C++ program.

Program 9.2: C++ Program to Overload ++ (Increment) operator.

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

class Employee
{
        private:
          char Emp_Name[200],Emp_Id[10];
          long int Salary,Increment;
        public:
          void Input_Employee_Information( );
          void Display_Employee_Information( );
          void operator ++( );
} ;

void Employee :: Input_Employee_Information( )
{
        cout << "\n Enter Employee Id = ";
        cin >> Emp_Id;
        cout << "\n Enter Employee Name = ";
        gets( Emp_Name );
        cout << "\n Enter Salary = ";
        cin >> Salary;
        cout << "\n";
}
```

```
void Employee :: Display_Employee_Information( )
{
        cout << "\n Employee Id = " << Emp_Id;
        cout << "\n Employee Name = " << Emp_Name;
        cout << "\n Salary = " << Salary << "\n";
}

void Employee :: operator ++( )
{
        cout << "\n Enter Salary Increment = ";
        cin >> Increment;
        Salary = Salary + Increment;
}

int main( )
{
        Employee E1;
        clrscr( );
        cout << "\n Input Employee Information:";
        E1.Input_Employee_Information( );
        cout<<"\n Employee Information:\n";
        E1.Display_Employee_Information( );
        E1++;
        cout<<"\n    Employee    Information    After    Salary
Increment:\n";
        E1.Display_Employee_Information();
        getch( );
        return(0);
}
```

Output of the above program:

```
        Input Employee Information:
        Enter Employee Id = E0014
        Enter Employee Name = Mr. Ankur Duwarah
        Enter Salary = 87000

        Employee Information:
        Employee Id = E0014
        Employee Name = Mr. Ankur Duwarah
        Salary = 87000
```

Enter Salary Increment = 1000

Employee Information After Salary Increment:
Employee Id = E0014
Employee Name = Mr. Ankur Duwarah
Salary = 88000

In the above C++ program (Program 9.2), increment operator ++ is overloaded where the operand of ++ is the object of the class 'Employee'. From the output of the program, we can have observed that unary operator ++ operates on the object 'E1' of class 'Employee' and it increments the value of the variable 'salary' by 1000 that is entered by the user.

9.6.2 Binary Operator Overloading

If the number of operand to an operator is two then the operator is called as binary operator. For example: +, -, =,* are binary operators. The syntax to overload binary operators in C++ is presented as follows.

```
Return_Type operator Binary_Operator_Symbol (Argument )
{
        // Body of the function to add new operation to the operator
}
```

From the above syntax, it can be observed that in C++ programming, to overload a binary operator, we are required to pass one parameter to the overloading function. In the above syntax, Binary_Operator_Symbol refers to the binary operator which is going to be overloaded. In binary operator overloading, the overloading function which is the member function of the first operand is invoked and the second operand is explicitly passed to that function. So, the data members of the first object can be accessed directly inside the overloading function without using the dot operator. On the other hand, the data members of the second object can be accessed by using dot operator with the object which is the parameter of the overloading function. If the second operand is not an object then it can be accessed directly.

Now, we will try to understand how to overload a binary operator by observing the following C++ program.

Program 9.3: C++ Program to Overload + (Addition) operator

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

class Complex_Number
{
        private:
                float X, Y;
        public:
                void Input_Complex_Number( );
                void Display_Complex_Number( );
                Complex_Number operator +( Complex_Number );
};

void Complex_Number:: Input_Complex_Number( )
{
        cout << "\n Enter Real part = ";
        cin >> X;
        cout << "\n Enter Imaginary part = ";
        cin >> Y;
}

void Complex_Number :: Display_Complex_Number( )
{
        if( Y>0 )
        {
                if( Y= =1 )
                      cout << X << " + " << "i";
                else
                      cout << X << " + " << Y << "i";
        }
        else
        {
                if( Y= =-1 )
                      cout << X << " - i";
                else
                        cout << X << " - " << -1*Y << "i";
```

```
        }
}

Complex_Number Complex_Number :: operator
+(Complex_Number T)
{
        Complex_Number CN;
        CN.X = X + T.X;
        CN.Y = Y + T.Y;
        return( CN );
}

int main( )
{
        Complex_Number CN1,CN2,SUM_CN;
        clrscr( );
        cout << "\n Enter First Complex Number:";
        CN1.Input_Complex_Number( );
        cout << "\n Enter Second Complex Number:";
        CN2.Input_Complex_Number( );
        SUM_CN = CN1+CN2;
        cout << "\n First Complex Number is = ";
        CN1.Display_Complex_Number( );
        cout << "\n Second Complex Number is = ";
        CN2.Display_Complex_Number( );
        cout << "\n Addition of the two Complex Number = ";
        SUM_CN.Display_Complex_Number( );
        getch( );
        return(0);
}
```

Output of the above program:

```
Enter First Complex Number:
Enter Real part = 4
Enter Imaginary part = 5

Enter Second Complex Number:
Enter Real part = 8
Enter Imaginary part = 9
First Complex Number is = 4 + 5i
```

Second Complex Number is = 8 + 9i
Addition of the two Complex Number = 12 + 14i

In the above program (Program 9.3), the binary operator + (Addition) is overloaded to implement addition of two complex numbers. It is observed that CN1 is the first operand and CN2 is the second operand of + operator. CN1 and CN2 are the objects of the class 'Complex_Number'.

Now the statement 'SUM_CN = CN1+CN2;' will invoke the binary operator overloading function 'Complex_Number operator +(Complex_Number)' which is the member function of CN1 and the object CN2 is explicitly passed to this overloading function. Inside the overloading function, data members of CN1 are accessed directly and data members of CN2 is accessed by using the dot operator with the object T which is the formal parameter of the overloading function.

Program 9.4: C++ Program for String Concatenation by Overloading + (Addition) Operator

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

class String
{
        private:
                char string1[200];
        public:
                void Input_String( );
                void Display_String( );
                String operator +(String);

};

void String :: Input_String( )
{
        gets(string1);
}
```

```
void String :: Display_String( )
{
        cout<<"\n"<<string1;
        cout<<"\n";
}

String String::operator +( String St1 )
{
         String St2;
         strcpy(St2.string1,string1);
         strcat(St2.string1,St1.string1);
         return(St2);
}

int main( )
{
          String S1 , S2 , S3;
          clrscr( );
          cout << "\n Read the first string :: ";
          S1.Input_String( );
          cout << "\n Read the second string :: ";
          S2.Input_String( );
          S3 = S1 + S2;
          cout << "\n The first string :: ";
          S1.Display_String( );
          cout << "\n The second string :: ";
          S2.Display_String( );
          cout << "\n Output after string concatenation :: ";
          S3.Display_String( );
          getch( );
          return(0);
}
```

Output of the above program:

```
 Read the first string :: Gauhati
 Read the second string :: University
 The first string :: Gauhati
 The second string :: University
 Output after string concatenation :: GauhatiUniversity
```

Program 9.5: C++ Program to implement String Copy by Overloading = (Assignment) Operator

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

class String
{
        private:
                        char string1[200];
        public:
                        void Input_String(  );
                         void Display_String( );
                         void operator =(String);

};

void String :: Input_String( )
{
        gets(string1);
}

void String :: Display_String( )
{
        cout << string1;
        cout << "\n";
}

void String::operator =(String St1)
{
        strcpy( string1 , St1.string1 );
}


int main( )
{
        String S1 , S2;
```

```
        clrscr( );
        cout << "\n Read the first string::";
        S1.Input_String( );
        cout << "\n Read the second string::";
        S2.Input_String( );
        cout << "\n The input strings are::\n";
        cout << "\n The first string::";
        S1.Display_String( );
        cout << "\n The second string::";
        S2.Display_String( );

        S2 = S1;
        cout << "\n After copying first string to second string::";
        cout << "\n The first string::";
        S1.Display_String( );
        cout << "\n The second string::";
        S2.Display_String( );

        getch( );
        return(0);
}
```

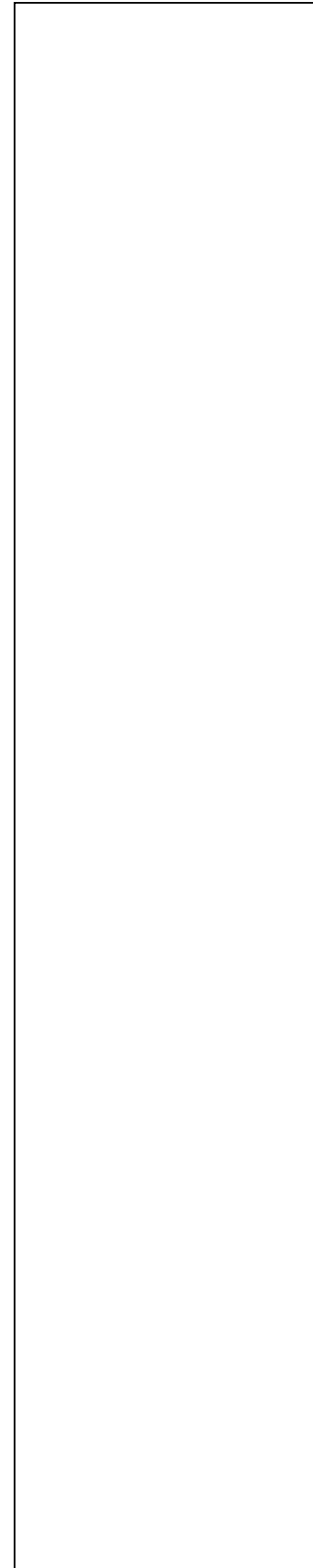
Output of the above program:

Read the first string:: Gauhati University
Read the second string:: IDOL
The input strings are::
The first string:: Gauhati University
The second string:: IDOL
After copying first string to second string ::
The first string :: Gauhati University
The second string :: Gauhati University

Program 9.6: C++ Program for String Comparison by Overloading = = (Equal to) Operator

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

```
class String
{
	private:
			char string1[200];
	public:
			void Input_String( );
			void Display_String();
			 int operator = =(String);
};

void String :: Input_String( )
{
	gets( string1 );
}

void String :: Display_String( )
{
	cout << "\n" << string1;
	cout << "\n";
}


int String :: operator = =( String St1 )
{
	strupr( string1 );
	strupr( St1.string1 );
	if( strcmp( St1.string1 , string1 ) = =0 )
			return(1);
	else
			return(0);

}

int main( )
{
	String S1 , S2;
	clrscr( );
	cout << "\n Read First String::";
	S1.Input_String( );
	cout << "\n Read Second String::";
	S2.Input_String( );
```



```
 if(S1= =S2)
{
            cout <<"\n Both are same strings";
            cout << "(Considering Both in Upper case
characters)";
 }
else
 {
            cout << "\n Both are different strings";
 }

getch( );
return(0);
}
```

Output of the above program:

Read First String :: Gauhati University
Read Second String :: Gauhati University
Both are same strings(Considering Both in Upper case characters)

9.6.3 Operator Overloading Using Friend Function

In C++, friend function can also be used to overload operators. When we use friend function to overload unary operators then one argument has to be passed to the operator overloading function. On the other hand, in case of binary operator overloading, we require to pass two arguments to the operator overloading function. The syntax of operator overloading using friend function is presented as follows.

```
friend Return_Type  operator  Symbol (Argument List )
{
        // Body of the function to add new operation to the operator
}
```

Let us try to understand the operator overloading using friend function by observing the following C++ program.

Program 9.7: C++ program to overload +(Addition) operator using friend function

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

class Complex_Number
{
        private:
                        float X,Y;
        public:
                        void Input_Complex_Number();
                         void Display_Complex_Number();
                         friend        Complex_Number        operator
+(Complex_Number,Complex_Number);
};

void Complex_Number::Input_Complex_Number( )
{
         cout << "\n Enter Real part = ";
         cin >> X;
         cout << "\n Enter Imaginary part = ";
         cin >> Y;
}

void Complex_Number :: Display_Complex_Number( )
{
         if( Y>0 )
        {
                        if( Y= =1)
                                cout << X << " + " << "i";
                        else
                                cout << X << " + " << Y << "i";
          }
          else
          {
                         if( Y= = -1)
                                cout << X << " - i";
                        else
                                cout << X << " - " << -1*Y << "i";
          }
}
```

```
Complex_Number operator +(Complex_Number T,
Complex_Number S)
{
	Complex_Number CN;
	CN.X = S.X + T.X;
	CN.Y = S.Y + T.Y;
	return( CN );
}

int main( )
{
	Complex_Number CN1 , CN2 , SUM_CN;
	clrscr( );
	cout << "\n Enter First Complex Number:";
	CN1.Input_Complex_Number( );
	cout << "\n Enter Second Complex Number:";
	CN2.Input_Complex_Number( );
	SUM_CN = CN1+CN2;
	cout << "\n First Complex Number is =";
	CN1.Display_Complex_Number( );
	cout << "\n Second Complex Number is =";
	CN2.Display_Complex_Number( );
	cout << "\n Addition of the two Complex Number =";
	SUM_CN.Display_Complex_Number( );
	getch( );
	return(0);
}
```

Output of the above program:

```
Enter First Complex Number:
Enter Real part = 3
Enter Imaginary part = 9

Enter Second Complex Number:
Enter Real part = 2
Enter Imaginary part = 4
First Complex Number is = 3 + 9i
Second Complex Number is= 2 + 4i
Addition of the two Complex Number = 5 +13i
```

In the above program (Program 9.7), operator +(Addition) is overloaded using the following function which is declared friend to the class 'Complex_Number':

```
Complex_Number operator +(Complex_Number T,
Complex_Number S)
{
        Complex_Number CN;
        CN.X = S.X + T.X;
        CN.Y = S.Y + T.Y;
        return( CN );
}
```

Two arguments T and S are passed to the operator overloading friend function because + is a binary operator. In this program, addition of two complex numbers is implemented. The statement 'SUM_CN = CN1+CN2;' will invoke the friend function mentioned above where CN1 and CN2 are passed as explicit arguments. CN1 and CN2 are the objects of class 'Complex_Number'. So the data members of CN1 are accessed using the dot operator with the object T which is the first formal parameter of the operator overloading friend function. On the other hand, the data members of CN2 are accessed using the dot operator with the object S which is the second formal parameter of the operator overloading friend function.

9.6.4 Data Conversion Using Operator Overloading

Operator overloading provides another feature in OOP that is data conversion. Data conversion refers the conversion of data from one form to its equivalent another form. For example: Conversion of Kilometre to Meter, Conversion of Hours to Minutes etc. In C++, data conversion can be implemented by overloading = (Assignment) operator.

Let us try to understand the data conversion by overloading = (Assignment) operator with the help of the following C++ program.

Program 9.8: C++ program to convert Hours to Minutes using Operator Overloading

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

class Hour
{
        private:
                        int No_Of_Hour ;
        public:
                        void Input( );
                        void Display( );
                        operator int( )
                        {
                                int No_Of_Min;
                                No_Of_Min = No_Of_Hour * 60;
                                return(No_Of_Min);
                        }
};

void Hour :: Input( )
{
        cout << "\n Enter Number of Hours =";
        cin >> No_Of_Hour;
}

void Hour :: Display( )
{
        cout << "\n Entered Number of Hours is = "<< No_Of_Hour ;
}

int main( )
{
        Hour H1;
        int M1;
        clrscr( );
        H1.Input( );
        H1.Display( );
        M1 = H1;
```

```
        cout<<"\n After conversion(Hours to Minutes), Number of minutes is =";
        cout << M1;
        getch( );
        return(0);
}
```

Output of the above program:

Enter Number of Hours = 3
Entered Number of Hours is = 3
After conversion (Hours to Minutes), Number of minutes is = 180

In the above program (Program 9.8), conversion of Hours to Minutes is implemented by overloading = (Assignment) operator. The statement ‘M1 = H1;’ invokes the conversion function ‘operator int( )’ which is the member function of the object H1. Then the conversion function will estimate the number of Minutes by multiplying the value stored in the variable ‘No_Of_Hour’ by 60. Here, the variable ‘No_Of_Hour’ is the data member of the object H1. The conversion function returns the estimated value and it is assigned to the variable M1.

9.6.5 Overloadable and Non-Overloadable Operators in C++

All available operators in an Object Oriented Programming (OOP) language are not overloadable. The overloadable and non-overloadable operators in C++ programming are presented as follows.

Overloadable Operators in C++ Programming:

+ , - , * , / , % , & , | , ~ , ^ , && , || , !

, = , == , > , <= , != , <= , >= , += , -= , *= ,

/= , %= , &= , |= , ^= , << , >> , <<= , >>= ,

++ , -- , [ ] , ( ) , -> , new , delete

Non-Overloadable Operators in C++ Programming:

. , :: , ?: , * , sizeof( )

STOP TO CONSIDER

There are some overloadable operators available in C++ programming which are not overloadable using friend function. These operators are = , ( ) , [ ] and -> .

CHECK YOUR PROGRESS

1. Multiple choices

(a) Which of the following is not true in case of Polymorphism?
(i) Multiple functions with same name.
(ii) Multiple operations with one operator.
(iii) Multiple variables with same name.
(iv) None of the above.

(b) Function overloading is a _______time Polymorphism.
(i) compile
(ii) run
(iii) pre-compile
(iv) None of the above

(c) Which of the following is true in function overloading?
(i) Multiple functions with same name but with different number of parameters.
(ii) Multiple functions with same name but with different types of corresponding parameters.
(iii) Multiple functions with same signature or prototype
(iv) Both (i) and (ii)

(d) Which of the following keyword is used in C++ to overload operators?
(i) overload
(ii) operator
(iii) operation

(iv) None of the above

(e) Which of the following operator cannot be overloaded in C++?

(i) **.** (dot operator)
(ii) **::** (Scope resolution operator)
(iii) **?:**
(iv) All of the above

(f) How many parameters have to be passed explicitly to overload a binary operator using special member function of a class?

(i) No parameter is required to pass
(ii) One parameter is required to pass
(iii) Two parameters are required to pass
(iv) None of the above.

2. Fill-in the blanks

(a) To overload a binary operator using friend function, we are required to pass ___ arguments to the operator overloading friend function.
(b) Data conversion can be implemented using ____overloading.
(c) ______ is an example of overloadable operator.
(d) The assignment operator (=) cannot be overloaded using _____function.

9.7 RUN-TIME POLYMORPHISM

At first, let us consider a situation where a derived class contains a member function whose name is same with a member function available in its base class but the functionalities of these functions are different. One more important point is that both of these functions have exactly same return type, same number of parameters and same data types of the corresponding parameters. Now the question is which member function (base class or derived class) should be executed if we try to declare a pointer to the base class or a reference variable to the base class and use it to invoke the function after pointing or referring a derived class object. The answer of this question is that the member function of the referred

derived class object should be executed. This process is termed as function overriding where base class member function is overridden by the derived class member function. It is another form of Polymorphism. In case of function overriding, compiler is able to recognize all necessary information to call the appropriate member function of base class or derived class at run time only. It means that when an overridden function is tried to be invoked then linking of the appropriate function can be recognized by the compiler at run time only. This type of binding of function call to its implementation is termed as Dynamic binding. So this type of polymorphism is termed as run time polymorphism. Function overriding is implemented in OOP using the concept of Virtual function.

STOP TO CONSIDER

If overridden functions are directly called using dot operator with a derived class objects then this operation will be compile time.

9.8 VIRTUAL FUNCTION

A virtual function in OOP is a member function of a base class and it can be overridden by member function available in the derived classes of the base class. A virtual function and its overridden functions are similar in terms of return type, function name, number of parameters and types of corresponding parameters. But they are different in terms of their functionalities. Virtual function allows function overriding or run time polymorphism in OOP. Most important point of this discussion is that a virtual function can be overridden only when it is accessed by a reference or a pointer to the base class where a derived class object is referred or pointed through this reference or pointer.

9.8.1 Implementation

The keyword '*virtual*' is used to declare a base class member function as virtual function. The syntax to declare and define virtual function in C++ is presented as follows.

```
class  Class_Name
{
```

```
        public:
                virtual  Return_Type  Function_Name ( Argument List)
                {
                        // Body of the virtual function

                }
};
```

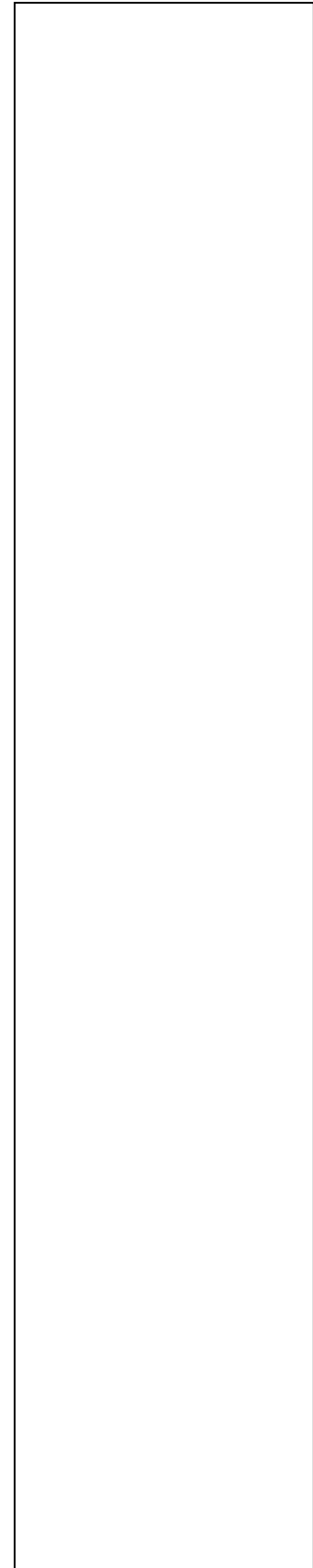
Let us try to understand the implementation of virtual function by observing the following C++ program.

Program 9.9: C++ program to implement Function Overriding using Virtual function

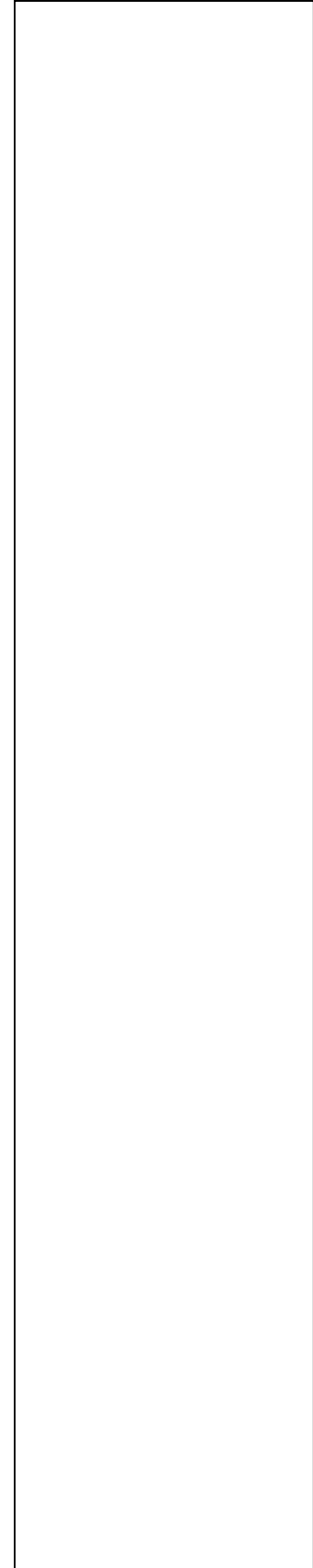
```
# include < iostream.h >
# include < conio.h >

class Shape_With_Sides
{
        private:
                int No_of_sides , Side[10] , i;
        public:
                virtual void Input_Data( );
                virtual void Display_Data( );
};

void Shape_With_Sides :: Input_Data ( )
{
        cout << "\n\n Enter Number of sides =";
        cin >> No_of_sides;
        for( i = 0 ; i < No_of_sides ; i++ )
        {
                cout << "\n Enter length of "<<i+1<< "th side =";
                cin >> Side[i];
        }
        cout << “\n”;

}
```

```
void Shape_With_Sides :: Display_Data( )
{
        cout << "\n Number of sides ="<<No_of_sides;
        if( No_of_sides < 3 )
                cout << "\n It is not a shape";
        else if( No_of_sides = = 3)
        {
                cout << "\n It is a Triangle shape";
                for( i = 0 ; i < No_of_sides ; i++ )
                {
                        cout << "\n Length of "<< i+1 << "th side =";
                        cout << Side[i];
                }
        }
        else if( No_of_sides = = 4)
        {
                cout << "\n It is a Quadrilateral shape";
                for( i = 0 ; i < No_of_sides ; i++ )
                {
                        cout << "\n Length of " << i+1 << "th side
=";
                        cout << Side[i];
                }


if((Side[0]==Side[1])&&(Side[1]==Side[2])&&(Side[2]==Side[3]))
                        cout << "\n It is a Square";
                else if
((Side[0]==Side[2])&&(Side[1]==Side[3])&&(Side[0]!=Side[1]))
                        cout << "\n It is a Rectangle";
                else
                        cout << "\n No data available";
        }
        else
                cout << "\n No data available right now";
}

class Triangle : public Shape_With_Sides
{
        private:
                int base, vertical_height;
```

```
        public:
                void Input_Data( );
                void Display_Data( );
};

void Triangle::Input_Data( )
{
        cout << "\n ****Triangle****";
        cout << "\n Enter base of the triangle =";
        cin >> base;
        cout << "\n Enter Vertical height of the triangle =";
        cin >> vertical_height;
        cout << "\n";

}

void Triangle::Display_Data( )
{
        cout << "\n Base of the triangle =";
        cout << base;
        cout << "\n Vertical height of the triangle =";
        cout << vertical_height;
        cout << "\n Area of the triangle =";
        cout <<(base*vertical_height)/2;
        cout << "\n";
}

class Square : public Shape_With_Sides
{
        private:
                int length;
        public:
                void Input_Data( );
                void Display_Data( );
};

void Square :: Input_Data( )
{
        cout << "\n****Square****";
        cout << "\n Enter the length of one side =";
        cin >> length;
```



```
        cout << “\n”;

}

void Square::Display_Data( )
{
        cout << "\n Length of one side in the square =";
        cout << length;
         cout << "\n Area of the square =";
         cout << (length*length);
         cout << "\n";
}

class Rectangle : public Shape_With_Sides
{
         private:
                  int width, height;
         public:
                 void Input_Data( );
                 void Display_Data( );
};

void Rectangle :: Input_Data( )
{
        cout << "\n****Rectangle****";
        cout << "\n Enter width of the rectangle =";
        cin >> width;
        cout << "\n Enter height of the rectangle =";
        cin >> height;
        cout << “\n”;

}

void Rectangle :: Display_Data( )
{
        cout << "\n Width of the Rectangle =";
        cout << width;
        cout << "\n Height of the Rectangle =";
        cout << height;
        cout << "\n Area of the Rectangle =";
        cout << width*height;
```



```
        cout << "\n";
}

int main( )
{
        Shape_With_Sides *Sh1;
        Triangle T1;
        Rectangle R1;
        Square S1;
        clrscr( );
        Sh1 = &T1;
        Sh1->Input_Data( );
        Sh1->Display_Data( );

        Sh1 = &S1;
        Sh1->Input_Data( );
        Sh1->Display_Data( );

        Sh1 = &R1;
        Sh1->Input_Data( );
        Sh1->Display_Data( );

        getch( );
        return(0);
}
```

Output of the above program:

```
****Triangle****
Enter base of the triangle = 3
Enter Vertical height of the triangle = 4

Base of the triangle = 3
Vertical height of the triangle = 4
Area of the triangle = 6

****Square****
Enter the length of one side = 7

Length of one side in the square = 7
Area of the square = 49
```

****Rectangle****
Enter width of the rectangle = 5
Enter height of the rectangle = 6

Width of the Rectangle = 5
Height of the Rectangle = 6
Area of the Rectangle = 30

In the above program (Program 9.9**),** we have **‘**Shape_With_Sides’ as the base class with ‘Triangle’, ‘Square’ and ‘Rectangle’ as its derived classes. Each of these four classes contains two functions with exactly same signature or prototype. These functions are ‘void Input_Data( )’ and ‘void Display_Data( )’. Now, in the base class, **‘**Shape_With_Sides’, these two functions are declared as virtual by using the keyword ‘*virtual*’.

In the main function, Sh1 is a pointer to the base class, ‘Shape_With_Sides’. T1 is the object of the class, ‘Triangle’. R1 is the object of the class, ‘Rectangle’. Finally, S1 is the object of the class, ‘Square’. Now the following points can be acknowledged by observing the output of the program.

a) When Sh1 is used to point T1 and it is used to invoke ‘void Input_Data( )’ and ‘void Display_Data( )’ then member functions of the class, ‘Triangle’ are called at run-time by overriding the corresponding virtual functions.

b) When Sh1 is used to point S1 and it is used to invoke ‘void Input_Data( )’ and ‘void Display_Data( )’ then member functions of the class ‘Square’ are called at run-time by overriding the corresponding virtual functions.

c) When Sh1 is used to point R1 and it is used to invoke ‘void Input_Data( )’ and ‘void Display_Data( )’ then member functions of the class ‘Rectangle’ are called at run-time by overriding the corresponding virtual functions.

If the functions, ‘void Input_Data( )’ and ‘void Display_Data( )’ are not declared as virtual in the base class, ‘Shape_With_Sides’ then the function overriding as shown in the above program is not going to be taken place. We can check this statement by changing the base class, ‘Shape_With_Sides’ and the main function as shown below.

```
class Shape_With_Sides
{
        private:
                int No_of_sides , Side[10] , i;
        public:
                void Input_Data( );
                void Display_Data( );
};

int main( )
{
        Shape_With_Sides *Sh1;
        Square S1;
        clrscr( );

        Sh1 = &S1;
        Sh1->Input_Data( );
        Sh1->Display_Data( );

        getch( );
        return(0);
}
```

Now, after execution of the program, the output will be:

```
Enter Number of sides = 3
Enter length of 1th side = 5
Enter length of 2th side = 8
Enter length of 3th side = 9

Number of sides = 3
It is a Triangle shape
Length of 1th side = 5
Length of 2th side = 8
Length of 3th side = 9
```

From the above output, it is observed that the virtual functions are not overridden by the member functions of the derived class ‘Square’.

9.8.2 Pure Virtual Function

When a virtual function declared in a base class doesn't have any definition or implementation then it is termed as Pure virtual function. The definition or implementation of a Pure virtual function may be available in the class that is derived from the base class where the Pure virtual function is declared. A Pure virtual function has to be defined in the derived class or it must be again declared as a Pure virtual function in that derived class. So, only derived class containing the definition of a Pure virtual function can invoke the Pure virtual function declared in its base class.

If a base class contain one or more Pure virtual functions then object of that class cannot be created. So a class containing Pure virtual functions is also termed as Abstract class. Abstract class is a type of class where at least one member function is available which don't have any definition or implementation. In OOP, Abstract classes are used as a model that can be utilized to derive new classes to provide new features or functionalities.

The syntax of writing Pure virtual function in C++ is presented as follows.

```
class  Class_Name
{
        public:
                virtual Return_Type   Function_Name(Argument List)
=  0;

};
```

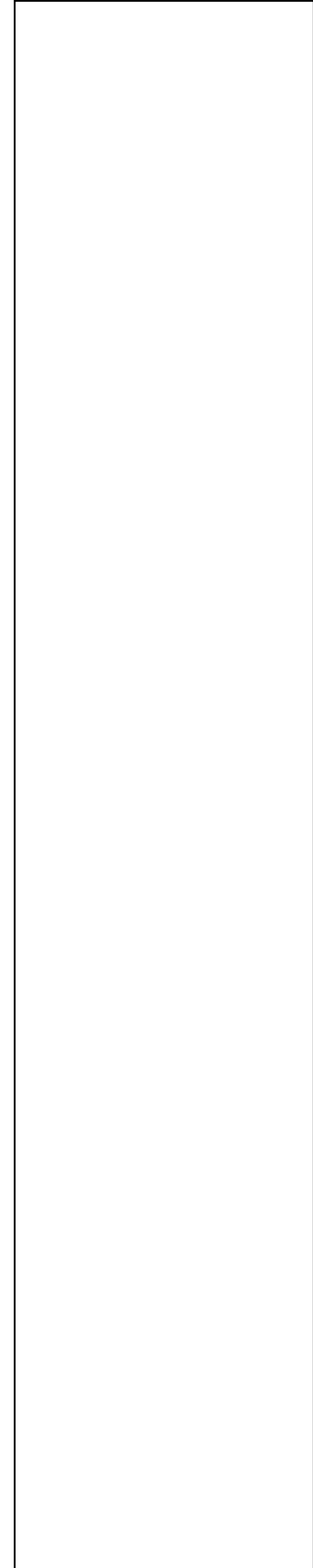
Let us try to understand the implementation of Pure virtual function by observing the following C++ program.

Program 9.10: C++ program to implement Pure virtual function

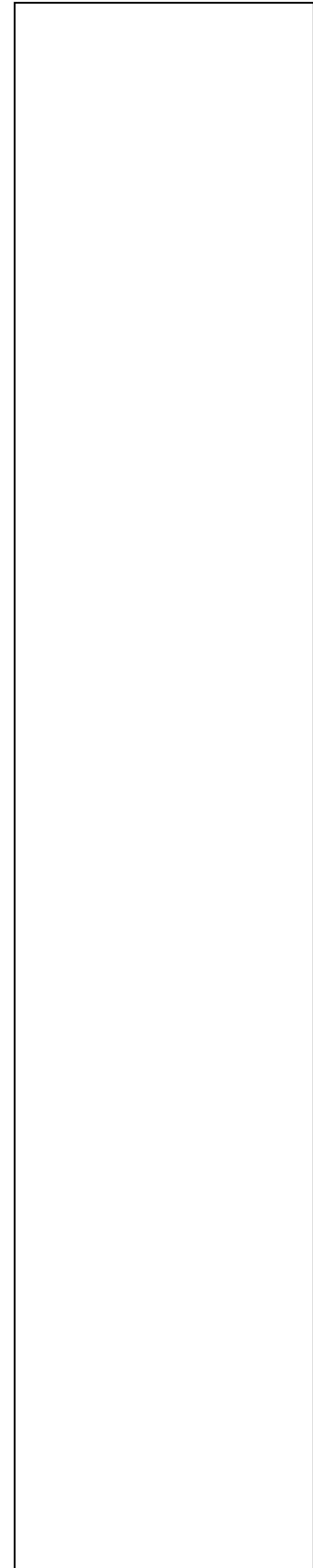
```
# include < iostream.h >
# include < conio.h >

class Shape
```

```
{

        public:
                        virtual void Input_Data( ) = 0;            //
Pure virtual function
                        virtual void Display_Data( ) = 0;          //
Pure virtual function
};


class Triangle : public Shape
{
        private:
                 int base, vertical_height;
        public:
                void Input_Data( );
                void Display_Data( );
};

void Triangle :: Input_Data( )
{
        cout << "\n ****Triangle****";
        cout << "\n Enter base of the triangle =";
        cin >> base;
        cout << "\n Enter Vertical height of the triangle =";
        cin >> vertical_height;
        cout << “\n”;

}

void Triangle :: Display_Data( )
{
        cout << "\n Base of the triangle =";
        cout << base;
        cout << "\n Vertical height of the triangle =";
        cout << vertical_height;
        cout << "\n Area of the triangle =";
        cout << (base*vertical_height)/2;
        cout << "\n";
}
```

```
class Square : public Shape
{
        private:
                int length;
        public:
                void Input_Data( );
                void Display_Data( );
};

void Square :: Input_Data( )
{
        cout << "\n****Square****";
        cout << "\n Enter the length of one side =";
        cin >> length;
        cout << "\n";

}

void Square :: Display_Data( )
{
        cout << "\n Length of one side in the square =";
        cout << length;
        cout << "\n Area of the square =";
        cout << (length*length);
        cout << "\n";
}

class Rectangle : public Shape
{
        private:
                int width, height;
        public:
                void Input_Data( );
                void Display_Data( );
};

void Rectangle :: Input_Data( )
{
        cout << "\n****Rectangle****";
        cout << "\n Enter width of the rectangle=";
        cin >> width;
```



```
        cout << "\n Enter height of the rectangle=";
        cin >> height;
        cout << "\n";

}

void Rectangle :: Display_Data( )
{
        cout << "\n Width of the Rectangle =";
        cout << width;
        cout << "\n Height of the Rectangle =";
        cout << height;
        cout << "\n Area of the Rectangle =";
        cout << width*height;
        cout << "\n";
}

int main( )
{
        Shape *Sh1;
        Triangle T1;
        Rectangle R1;
        Square S1;
        clrscr( );
        Sh1 = &T1;
        Sh1->Input_Data( );
        Sh1->Display_Data( );

        Sh1 = &S1;
        Sh1->Input_Data( );
        Sh1->Display_Data( );

        Sh1 = &R1;
        Sh1->Input_Data( );
        Sh1->Display_Data( );

        getch( );
        return(0);
}
```



Output of the above program:

****Triangle****
Enter base of the triangle = 4
Enter Vertical height of the triangle = 5

Base of the triangle = 4
Vertical height of the triangle = 5
Area of the triangle = 10

****Square****
Enter the length of one side = 5

Length of one side in the square = 5
Area of the square = 25

****Rectangle****
Enter width of the rectangle = 8
Enter height of the rectangle = 7

Width of the Rectangle = 8
Height of the Rectangle = 7
Area of the Rectangle = 56

In the above program (Program 9.10), two Pure virtual functions, 'void Input_Data( )' and 'void Display_Data( )' are declared in the base class, 'Shape'. The definitions of these two functions are available in the derived classes that are 'Triangle', 'Square' and 'Rectangle'.

CHECK YOUR PROGRESS

3. Multiple choices

(a) Which of the following is run-time Polymorphism?
(i) Function overloading
(ii) Operator overloading
(iii) Function overriding
(iv) None of the above

(b) Function overriding is achieved using _____.

(i) friend function
(ii) virtual function
(iii) inline function
(iv) None of the above

(c) If a virtual function does not contain the implementation then it is called as_____
(i) Pure virtual function
(ii) Incomplete virtual function
(iii) Complete virtual function
(iv) None of the above

(d) Which of the following keyword is used to declare a base class member function as virtual?
(i) abstract
(ii) virtual
(iii) operator
(iv) None of the above

(e) If a class contains one or more Pure virtual functions then it is also termed as _____.
(i) Virtual class
(ii) Concrete class
(iii) Abstract class
(iv) None of the above

9.9 SUMMING UP

- Polymorphism is one of the important properties of Object Oriented Programming (OOP). It allows using one function name for multiple functions with different functionalities. It also adds multiple meanings to an operator. In OOP, Polymorphism can be categorized into two major groups that are Compile time polymorphism and Run time polymorphism.
- Function overloading and Operator overloading are the two types of Compile time polymorphism.
- Function overloading allows the use of a single name to multiple functions with different functionalities. These functions must also be different with each other in terms of

number of parameters or types of corresponding parameters or both.

- Operator overloading allows addition of multiple meanings to an operator in OOP. The keyword 'operator' is used to declare operator overloading function in C++.
- Data conversion is also achieved by operator overloading in OOP.
- All available operators in an Object Oriented Programming (OOP) language are not overloadable.
- Function overriding is the run-time polymorphism. It allows execution of derived class member function instead of its base class member function at run time. The name and the prototype of these functions are exactly same.
- Function overriding is implemented in OOP using the concept of Virtual function. The keyword '*virtual*' is used to declare a base class member function as virtual function.
- If a virtual function doesn't have any definition or implementation then it is termed as Pure virtual function.
- Abstract class is a type of class where at least one member function is available which don't have any definition or implementation.

9.10 ANSWERS TO CHECK YOUR PROGRESS

1. (a). (iii), (b). (i), (c). (iv), (d). (ii) , (e). (iv) , (f). (ii)

2. (a). two, (b). operator, (c). &&, (d). friend

3. (a). (iii), (b). (ii), (c). (i), (d). (ii), (e). (iii)

9.11 POSSIBLE QUESTIONS

1) Define Polymorphism. Write down the categories of Polymorphism.
2) Explain Function overloading with a suitable example.
3) Why Operator overloading is called as compile time polymorphism?
4) Explain unary operator overloading with a suitable example.

5) How friend function can be used to overload a binary operator in C++? Give example.
6) Write down any three overloadable operators and three non-overloadable operators available in C++.
7) What is virtual function? Explain how virtual function is used to implement function overriding.
8) Explain Pure virtual function with an example.

9.12 REFERENCES AND SUGGESTED READINGS

1) Venugopal, K. R., Rajkumar, Ravishankar, T. *Mastering C++*. Tata McGraw-Hill Education, 2001.
2) Balagurusamy, E. *Object Oriented Programming with C++*. Tata McGraw-Hill, 2006

UNIT 10: EXCEPTION HANDLING

Unit Structure:

10.1 Introduction

10.2 Unit Objectives

10.3 Exception handling mechanism

    10.3.1 throwing mechanism

    10.3.2 catching mechanism

10.4 Throwing the same exception again

10.5 Exceptions in constructor and destructor

10.6 Error handling

10.7 Summing Up

10.8 Answers to check your progress

10.9 Possible questions

10.10 References and Suggested Readings

10.1 INTRODUCTION

Run-time abnormalities or unexpected conditions that a programme experiences during execution are known as exceptions.When the system handles these abnormalities orconditions improperly, it can lead to failures and system crashes. Exception failures are estimated to cause two thirds of system crashes and fifty percent of computer system security vulnerabilities. Exception handling is especially important in embedded and real-time computer systems because software in these systems cannot easily be fixed or replaced, and they must deal with the unpredictability of the real world. Robust exception handling in software can improve software fault tolerance and fault avoidance, but no structured techniques exist for implementing dependable exception handling. However, many exceptional conditions can be anticipated when the system is designed, and protection against these conditions can be incorporated into the system. Traditional software engineering techniques such as code walkthroughs and software testing can

illuminate more exceptional conditions to be caught, such as bad input for functions and memory and data errors. However, it is impossible to cover all exceptional cases. It is also difficult to design a dependable system that can tolerate truly unexpected conditions. In these cases, some form of graceful degradation is necessary to safely bring down the system without causing major hazards.

In C++, an exception is a situation that arises during the execution of a program code. In C++ program, an exception is a response to an exceptional situation that arises while a program is running, such as an attempt to divide by zero. Exceptions provide a way to transfer control from one part of a program to another part of the same program. Exception handling is built upon three keywords: try, throw and catch.All exceptions are derived from std:: exception class. It is a runtime error that can be handled. If we don't handle the exception, it prints the exception message and terminates the program.

Why Exception Handling?

Following are main advantages of exception handling over traditional error handling.

1) Separation of Error Handling code from Normal Code: In traditional error handling codes, there are *if else* conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling is separated from the normal flow.

2) Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

3) Grouping of Error Types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

10.2 UNIT OBJECTIVES

In this unit learners will learn how to handle exceptions in C++. Also learners will know the different ways to handle them. Precisely learners will learn:
Exception handling mechanism
Throwing mechanism
Catching mechanism
Throwing the same exception again:

10.3 EXCEPTION HANDLING MECHANISM

The purpose of the Exception Handling mechanism is to provide and detect a report of "exceptional circumstances" so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:Find the exception, Throw the exception, Catch the exception, Handle the exception

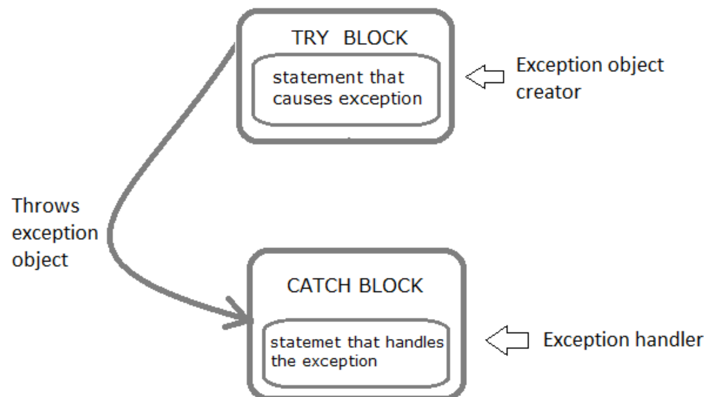


Fig: Exception Handling Mechanism

Try: The keyword *try* is used to preface a block of statements which may generate an exception. This block of statements is known as try block.

Throw: When an exception is detected, it is thrown using a *throw* statement in the try block.

Catch: The *catch* block catches an exception must immediately follow the try block that throws the exception.

Syntax of Exception Handling:

```
.......
.......
try
{
statement;     // generates an exception
  throw exception;  // throws an exception
  .........
}
catch(Exception-type e)
{
statement;  // processes the exception
}
........
........
```

10.3.1 Throwing Mechanisms

Throwing Mechanisms:

Exceptions can be thrown anywhere within a code block using *throw* statement. The operand of the *throw* statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs −

```
double division(int a, int b) {
```

```
if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}
```

10.3.2 Catching Mechanisms

The *catch* block following the *try* block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
   // protected code
} catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

Above code will catch an exception of ExceptionName type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −

```
try {
   // protected code
} catch(...) {
  // code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

```
#include <iostream>
```

```
using namespace std;

double division(int a, int b) {
if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x = 50;
   int y = 0;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
cerr<<msg<< endl;
   }

   return 0;
}
```

Define New Exceptions

You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way −

Example:

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

structMyException : public exception {
const char * what () const throw () {
      return "C++ Exception";
   }
};

int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
std::cout<< "MyException caught" <<std::endl;
std::cout<<e.what() <<std::endl;
   } catch(std::exception& e) {
      //Other errors
   }
}
```

CHECK YOUR PROGRESS

1. What is an exception in C++ program?
2. By default, what a program does when it detects an exception?
3. Why do we need to handle exceptions?
4. How Exception handling is implemented in the C++ program?
5. What is the correct syntax of the try-catch block?

10.4 THROWING THE SAME EXCEPTION AGAIN

Rethrowing an expression from within an exception handler can be done by calling *throw*, by itself, with no exception. This causes current exception to be passed on to an outer try/catch sequence. An exception can only be rethrown from within a catch block. When an exception is rethrown, it is propagated outward to the next catch block.

- Consider the following code:

```
#include <iostream>
using namespace std;
void MyHandler()
{
  try
  {
     throw “hello”;
  }
  catch (const char*)
  {
  cout <<”Caught exception inside MyHandler\n”;
  throw; //rethrow char* out of function
  }
}
int main()
{
  cout<< “Main start”;
  try
  {
MyHandler();
  }
catch(const char*)
  {
    cout <<”Caught exception inside Main\n”;
  }
     cout << “Main end”;
     return 0;
}
```

Output :
Main start

```
Caught exception inside MyHandler
Caught exception inside Main
Main end
```

Sometimes you want to do something with the exception you catch (like write to log or print a warning) and let it bubble up to the upper scope to be handled. To do so, you can rethrow any exception you catch:

```
try {
    ... // some code here
} catch (constSomeException& e) {
std::cout << "caught an exception";
    throw;
}
```

Using throw; without arguments will re-throw the currently caught exception.

To rethrow a managed std::exception_ptr, the C++ Standard Library has the rethrow_exception function that can be used by including the <exception> header in your program.

```
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

void handle_eptr(std::exception_ptreptr) // passing by value is ok
{
    try {
        if (eptr) {
std::rethrow_exception(eptr);
        }
    } catch(conststd::exception& e) {
std::cout << "Caught exception \"" <<e.what() << "\"\n";
    }
}

int main()
{
```

```
std::exception_ptreptr;
    try {
std::string().at(1); // this generates an std::out_of_range
    } catch(...) {
eptr = std::current_exception(); // capture
    }
handle_eptr(eptr);
} // destructor for std::out_of_range called here, when the eptr is destructed
```

CHECK YOUR PROGRESS

6. Which part of the try-catch block is always fully executed?
7. Which of the following is an exception in C++?
8. What is an error in C++?
9. What is the difference between error and exception?
10. What are the different types of exceptions?
11. Which keyword is used to throw an exception?

10.5 EXCEPTIONS IN CONSTRUCTOR AND DESTRUCTOR

When an exception is thrown from a constructor, the object is not considered instantiated, and therefore its destructor will not be called. But all destructors of successfully constructed base and member objects of the same master object will be called. Destructors of not yet constructed base or member objects of the same master object will not be executed. Example:

```
class A : public B, public C
{
public:
    D sD;
    E sE;
    A(void)
:B(), C(), sD(), sE()
    {
    }
```

};
Let's assume the constructor of base class C throws. Then the order of execution is:

B
C (throws)
~B
Let's assume the constructor of member object sE throws. Then the order of execution is:

B
C
sD
sE (throws)
~sD
~C
~B
Thus if some constructor is executed, one can rely on that all other constructors of the same master object executed before, were successful. This enables one, to use an already constructed member or base object as an argument for the constructor of one of the following member or base objects of the same master object.

What happens when we allocate this object with new?

Memory for the object is allocated
The object's constructor throws an exception
The object was not instantiated due to the exception
The memory occupied by the object is deleted
The exception is propagated, until it is caught
The main purpose of throwing an exception from a constructor is to inform the program/user that the creation and initialization of the object did not finish correctly. This is a very clean way of providing this important information, as constructors do not return a separate value containing some error code (as an initialization function might).

In contrast, it is strongly recommended not to throw exceptions inside a destructor. It is important to note when a destructor is called:

as part of a normal deallocation (exit from a scope, delete)
as part of a stack unwinding that handles a previously thrown exception.
In the former case, throwing an exception inside a destructor can simply cause memory leaks due to incorrectly deallocated object. In the latter, the code must be cleverer. If an exception was thrown as part of the stack unwinding caused by another exception, there is no way to choose which exception to handle first. This is interpreted as a failure of the exception handling mechanism and that causes the program to call the function terminate.

To address this problem, it is possible to test if the destructor was called as part of an exception handling process. To this, one should use the standard library function uncaught_exception, which returns true if an exception has been thrown, but hasn't been caught yet. All code executed in such a situation must not throw another exception.

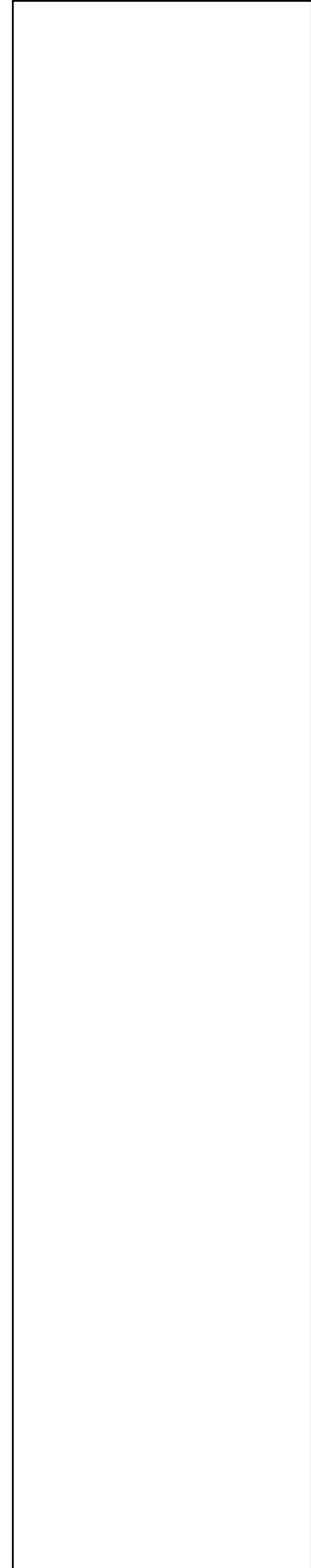
Situations where such careful coding is necessary are extremely rare. It is far safer and easier to debug if the code was written in such a way that destructors did not throw exceptions at all.

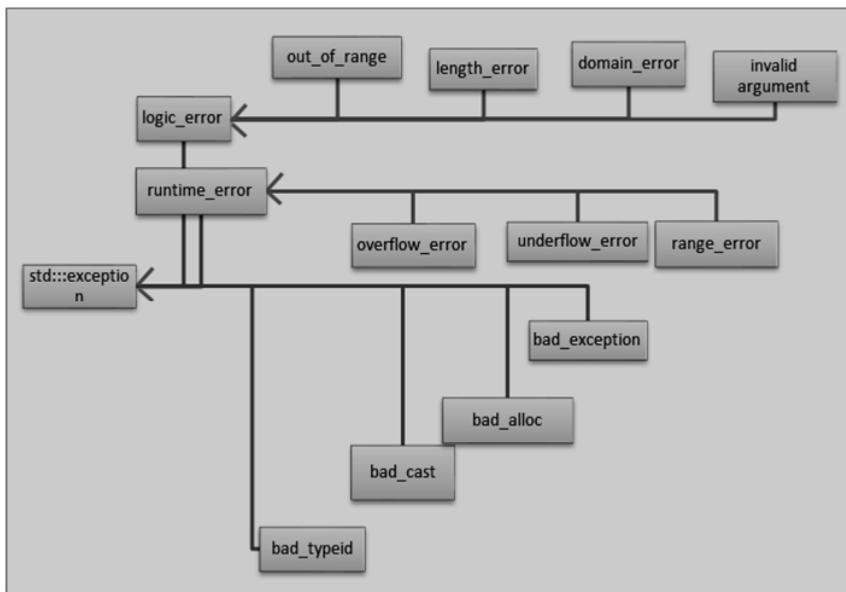
10.6 ERRORS HANDLING

Errors occurring at runtime create serious issues and most of the time, it hinders the normal execution of the program. That is why we have to handle these errors. This handling of errors in order to maintain the normal flow of the execution of the program is known as “Exception Handling”.

C++ Standard Exceptions

The following diagram shows the hierarchy of exceptions class std:: exception supported in C++





In the below, we will see the description of each of the above exceptions:

Exception	**Description**
std::exception	This is the parent class for all C++ standard exceptions.
std::bad_alloc	Thrown by 'new' operator when memory allocation goes wrong.
std::bad_cast	Occurs when 'dynamic_cast' goes wrong.
std::bad_typeid	This exception is thrown by typeid.
std::bad_exception	This exception is used to handle unexpected exceptions in the program.
std::runtime_error	Exceptions that occur at runtime and cannot be determined merely by reading the code.
std::overflow_error	This exception is thrown when mathematical overflow occurs.
std::underflow_error	This exception is thrown when mathematical underflow occurs.
std::range_error	When program stores value that is out of range, this exception is thrown.
std::logic_error	Those exceptions can be deduced by reading the code.

std::out_of_range	The exception is thrown when we access/insert elements that are not in range. For example, in the case of vector or arrays.
std::length_error	This exception is thrown in the case when the length of the variable is exceeded. For example, when a string of big length is created for type std::string.
std::domain_error	Thrown when the mathematically invalid domain is used.
std::invalid_argument	Exception usually thrown for invalid arguments.

How To Stop Infinite Loop In Exception Handling

Let us consider yet another case of an infinite loop.

If the code inside is bound to throw an exception, then we need to consider the actual place where we should put our try, catch block. That is whether the try catch block should be inside the loop or the loop should be inside the try-catch block.

There is no guaranteed answer regarding the placement of try catch block, however, it depends solely on the situation. One consideration is that the try catch block usually causes overhead for the program. So unless required, we should do away with an exception handling code in the infinite loop.

Let us consider the example shown below.

```
#include <iostream>

#include <stdexcept>

using namespace std;


int main (void)

 {

  int Sum=0, num;

  cout<<"Please enter number you wish to add(-99 to exit):"<<endl;
while( true ) { try{ cin>>num;
```

```
if(num == -99)
            throw -99;
         Sum+=num;
      }
     catch(...)
     {
         cout <<" Aborting the program..."<<endl;
         break;
      }
  }
cout << "Sum is:" << Sum << endl;
return 0;
 }
```

Here we have an infinite while loop. Inside the loop, we read an input number and add it to sum. In order to come out of the loop, we need to give the terminating condition inside the loop. We have given -99 as the terminating condition.

As seen, we have put this code in a try block and when the number entered is equal to -99, we throw an exception that is caught in the catch block.

Now in the above situation, if we put the entire while loop inside the try block, then there is bound to be overhead as while is an infinite loop. So the best place to put the try catch block is inside the loop.

The program gives the following output:

Output:

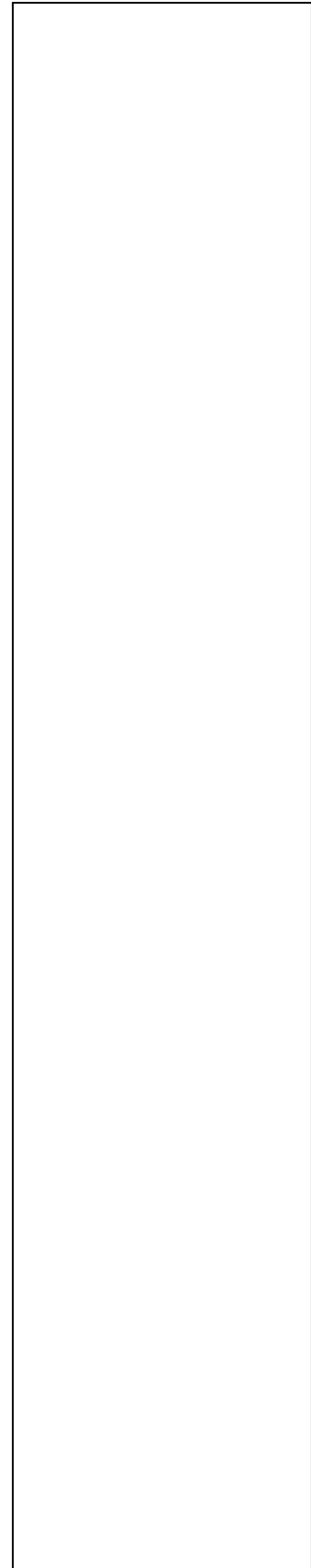
Please enter the number you wish to add(-99 to exit):

-99

Aborting the program…

Sum is:0

Note that the control is passed to catch block once -99 is entered as input.



Note that we have not given any specific exception object as an argument to catch. Instead, we have provided (…). This indicates that we are catching the generalized exception.

CHECK YOUR PROGRESS

12. What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;
void func(int a, int b)
{
    if(b == 0){
        throw "This value of b will make the product zero. "
       "So please provide positive values.\n";
    }
    else{
        cout<<"Product of "<<a<<" and  "<<b<<" is: 
"<<a*b<<endl;
    }
}
int main()
{
    try{
        func(5,0);
    }
    catch(const char* e){
        cout<<e;
    }
}
```

As the value of b = 0 is provided to the func() and the function is throwing an exception whenever the value of b = 0. Therefore the function throws the exception which will be printed on the screen.

13. What will be the output of the following C++ code?

```
#include <iostream>

#include <string>

#include <cstdlib>

using namespace std;

void func(int a, int b)

{

	if(b == 0){

		throw "This value of b will make the product zero. "

		"So please provide positive values.\n";

	}

	else{

		cout<<"Product of "<<a<<" and  "<<b<<" is: 
"<<a*b<<endl;

	}

}


int main()

{

	try{

		func(5,0);

	}

	catch(char* e){

		cout<<e;

	}

}
```

As the func() is throwing a const char* string but we the catch block is not catching any const char* exception i.e. exception thrown is

not handled therefore the program results into Aborted(core dumped).

14. What is Re-throwing an exception means in C++?

15. What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;
void func(int a, int b)
{
        if(b < 1){
                throw b;
        }
        else{
                cout<<"Product of "<<a<<" and  "<<b<<" is: 
"<<a*b<<endl;
        }
}
int main()
{
        try
      {
                try
           {
                func(5,-1);
                }
                catch(int b)
           {
                        if(b==0)
                                throw "value of b is zero\n";
```

```
                else
                        throw "value of b is less than zero\n";
            }
    }
    catch(const char* e)
  {
            cout<<e;
    }

}
```

Here the func() throws the value of b which is caught by the inner try-catch block, which again throws the message inorder to handle different cases of b which is caught by the outer try-catch block. Now as the value of b is negative the program outputs the message as shown.

16. Explain terminate() and unexpected() function.

17. Describe Exception handling concept with an example.

18. Illustrate Rethrowing exceptions with an example.

Multiple Choice Questions:

19. Generic catch handler is represented by _______________.

a. catch(..,)
b. catch(---)
c. catch(…)
d. catch(void x)

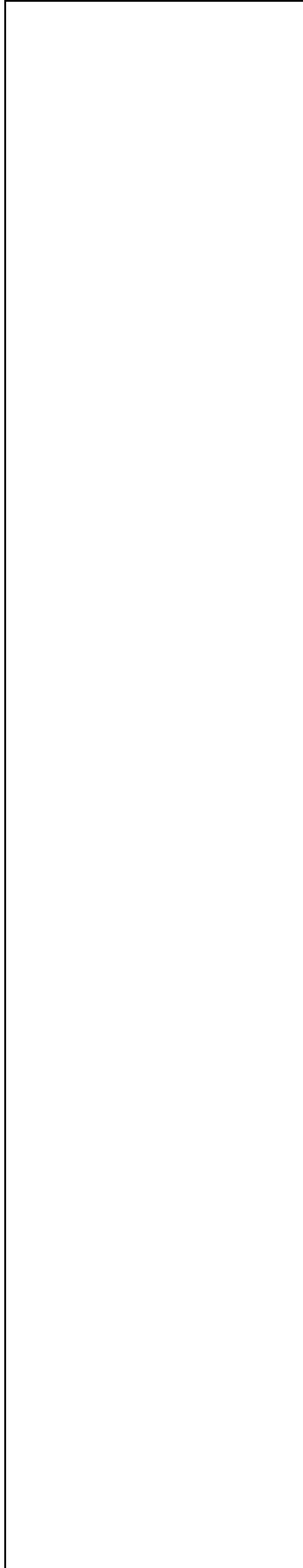
20. Throwing an unhandled exception causes standard library function ________________ to be invoked.

a. stop()

b. aborted()

c. terminate()

d. abandon()



21. Attempting to throw an exception that is not supported by a function call results in calling _____________ library function.

a. indeterminate()

b. unutilized()

c. unexpected()

d. unpredicted()

22. Return type of uncaught_exception() is________________.

a. int

b. bool

c. char *

d. double

23. How can we restrict a function to throw certain exceptions?

a. Defining multiple try and catch block inside a function

b. Defining generic function within try block

c. Defining function with throw clause

d. It is not possible in CPP to restrict a function

24. In nested try blocks, if both inner and outer catch handlers are not able to handle the exception, then______________.

a. Compiler executes only executable statements of main()

b. Compiler issues compile time errors about it

c. Program will be executed without any interrupt

d. Program will be terminated abnormally

25. If inner catch handler is not able to handle the exception then__________.

a. Compiler will look for outer try handler

b. Program terminates abnormally

c. Compiler will check for appropriate catch handler of outer try block

d. None of these

26. In nested try block, if inner catch handler gets executed, then_______________

a. Program execution stops immediately

b. Outer catch handler will also get executed

c. Compiler will jump to the outer catch handler and then executes remaining executable statements of main()

d. Compiler will execute remaining executable statements of outer try block and then the main()

27. Which of the following statements are true about Catch handler?

1. It must be placed immediately after try block T

2. It can have multiple parameters

3. There must be only one catch handler for every try block

4. There can be multiple catch handler for a try block T

5. Generic catch handler can be placed anywhere after try block.

a. Only 1, 4, 5

b. Only 1, 2, 3

c. Only 1, 4

d. Only 1, 2

State True or False:

28. We can prevent a function from throwing any exceptions.

29. An exception can be of only built-In type.

30. Irrespective of exception occurrence, catch handler will always get executed.

31. Functions called from within a try block may also throw exception.

32. Generic catch handler must be placed at the end of all the catch handlers.

33. In nested try blocks, there is no need to specify catch handler for inner try block. Outer catch handler is sufficient for the program.

10.7 SUMMING UP

Here we have learnt about the various aspects of exception handling in C++. In this unit precisely we have gone through the following: Exception handling mechanism: like throwing mechanism, catching mechanism, throwing the same exception again, exceptions in constructor and destructor, Error handling etc.

10.8 ANSWERS TO CHECK YOUR PROGRESS

1. What is an exception in C++ program?

An exception is defined as the problem in C++ program that arises during the execution of the program for example divide by zero error.

2. By default, what a program does when it detects an exception?

By default, whenever a program detects an exception the program crashes as it does not know how to handle it hence results in the termination of the program.

3. Why do we need to handle exceptions?

We need to handle exceptions in a program to avoid any unexpected behaviour during run-time because that behaviour may affect other parts of the program. Also, an exception is detected during run-time, therefore, a program may compile successfully even with some exceptions cases in your program.

4. How Exception handling is implemented in the C++ program?

C++ provides a try-catch block to handle exceptions in your program.

5. What is the correct syntax of the try-catch block?

Try-catch block has the following syntax:

```
try{
        // codes that needs to check for exceptions
}
catch(Exception E1){
    // codes for handling exception....
    // Exception E denotes the type of exception this block is
handling.
}
catch(Exception E2){
        // other exception that needs to be handled...
}
```

You can have any number of catch blocks catching different exceptions…..

6. Which part of the try-catch block is always fully executed?

Finally part of the try-catch block is always executed whether exceptions are caught or not.

7. Which of the following is an exception in C++?

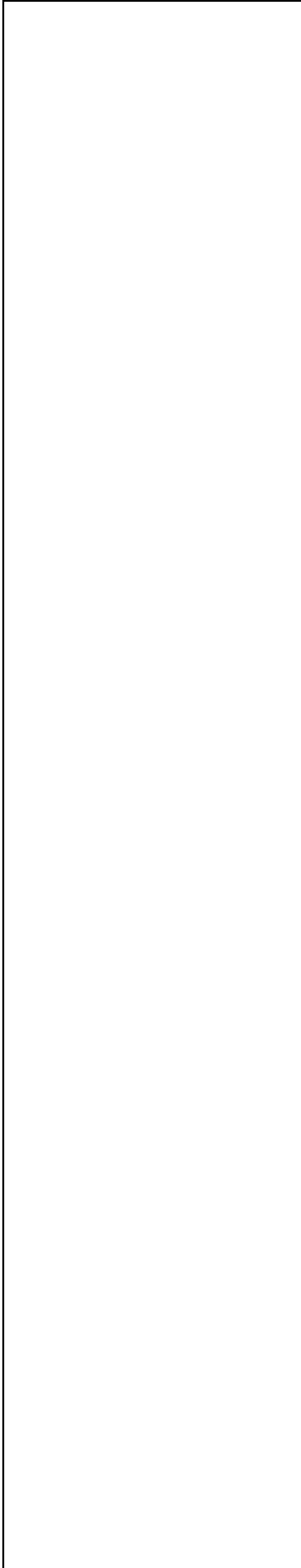
Exceptions are those which are encountered during run-time of the program. semicolon, variable not declared and the wrong expression are compile-time errors, therefore, they are not exceptions. Divide by zero is the problem that is encountered during run-time, therefore, it is an exception.

8. What is an error in C++?

An error occurs when rules and laws of a language is violated while writing programs in that language.

9. What is the difference between error and exception?

Exceptions can be handled during run-time whereas errors cannot be because exceptions occur due to some unexpected conditions during run-time whereas about errors compiler is sure and tells about them during compile-time.



10. What are the different types of exceptions?

There are two types of exceptions: Synchronous and asynchronous exceptions. Synchronous exceptions that are caused by the event which can be controlled by the program whereas Asynchronous exceptions are those which are beyond the control of the program.

11. Which keyword is used to throw an exception?

'throw' keyword is used to throw exceptions if something bad happens.

12. What will be the output of the following C++ code?

```
#include <iostream>

#include <string>

#include <cstdlib>

using namespace std;

void func(int a, int b)

{

        if(b == 0){

                throw "This value of b will make the product zero. "

               "So please provide positive values.\n";

        }

        else{

                cout<<"Product of "<<a<<" and  "<<b<<" is: 
"<<a*b<<endl;

        }

}

int main()

{

        try{

                func(5,0);

        }

        catch(const char* e){

                cout<<e;
```

```
        }
}
```

As the value of b = 0 is provided to the func() and the function is throwing an exception whenever the value of b = 0. Therefore the function throws the exception which will be printed on the screen.

Output:

This value of b will make the product zero. So please provide positive values.

13. What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;
void func(int a, int b)
{
        if(b == 0){
                throw "This value of b will make the product zero. "
                "So please provide positive values.\n";
        }
        else{
                cout<<"Product of "<<a<<" and  "<<b<<" is: 
"<<a*b<<endl;
        }
}

int main()
{
        try{
                func(5,0);
        }
        catch(char* e){
```

```
            cout<<e;
        }
}
```

As the func() is throwing a const char* string but we the catch block is not catching any const char* exception i.e. exception thrown is not handled therefore the program results into Aborted(core dumped).

Output:

terminate called after throwing an instance of 'char const*'

Aborted (core dumped)

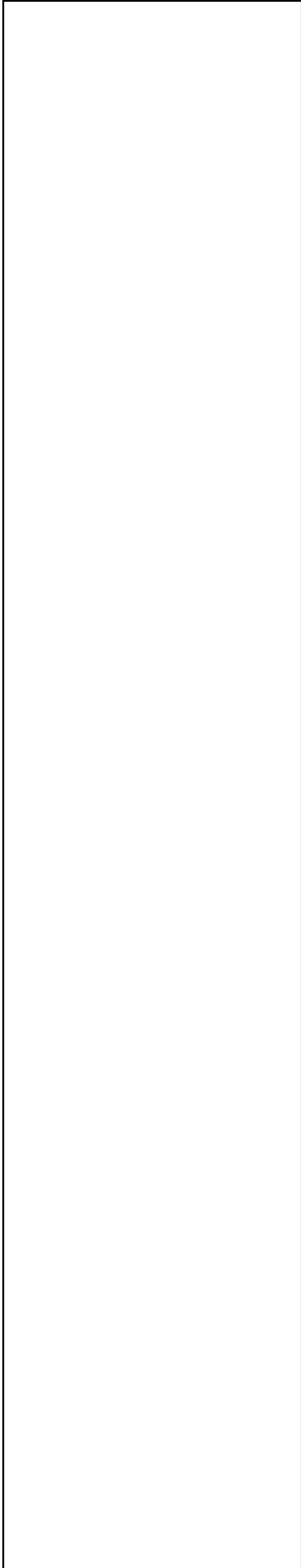
14. What is Re-throwing an exception means in C++?

Exception that is caught by a catch block but not handled by that catch block can be re-thrown by that catch block to further try-catch block.

15. What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;
void func(int a, int b)
{
        if(b < 1){
                throw b;
        }
        else{
                cout<<"Product of "<<a<<" and  "<<b<<" is:
"<<a*b<<endl;
        }
}
int main()
{
```

```
try
{
    try
    {
    func(5,-1);
    }
    catch(int b)
    {
        if(b==0)
            throw "value of b is zero\n";
        else
            throw "value of b is less than zero\n";
    }
}
catch(const char* e)
{
    cout<<e;
}

}
```



Here the func() throws the value of b which is caught by the inner try-catch block, which again throws the message inorder to handle different cases of b which is caught by the outer try-catch block. Now as the value of b is negative the program outputs the message as shown.

Output:

value of b is less than zero

16. Explain terminate() and unexpected() function.

terminate() is a library function which by default aborts the program

It is called whenever the exception handling mechanism cannot find a handler for a thrown exception..................

17. Describe Exception handling concept with an example.

Exceptions: Exceptions are certain disastrous error conditions that occur during the execution of a program. They could be errors that cause the programs to fail or certain conditions that lead to errors. If these run time errors are not handled by the program, OS handles them and program terminates.................

18. Illustrate Rethrowing exceptions with an example.

Rethrowing an expression from within an exception handler can be done by calling throw, by itself, with no exception. This causes current exception to be passed on to an outer try/catch sequence.

19. c. 20. c 21. c 22. b 23. c 24. d 25. c 26. d 27. c

28. True 29.False 30.False 31. True 32.True 33. False

10.9 POSSIBLE QUESIONS

1. Why use exceptions?

2. How do we use exceptions?

3. What shouldn't we use exceptions for?

4. What are some ways try / catch / throw can improve software quality?

5. How do exceptions simplify my function return type and parameter types?

6. What does it mean that exceptions separate the "good path" (or "happy path") from the "bad path"?

7. Can we throw an exception from a constructor? From a destructor?

8. How can we handle a constructor that fails?

9. How can we handle a destructor that fails?

10. How should we handle resources if my constructors may throw exceptions?

11. How do we change the string-length of an array of char to prevent memory leaks even if/when someone throws an exception?

12. What should we throw?

13. What should we catch?

14. What does throw; (without an exception object after the throw keyword) mean? Where would we use it?

15. How do we throw polymorphically?

16. When we throw this object, how many times will it be copied?

17. Why doesn't C++ provide a "finally" construct?

18. Why can't we resume after catching an exception?

10.10 REFERENCES AND SUGGESTED READINGS

1. C++ Primer 5th Edition: C++ Primer (5th Edition) 5th Edition written by Stanley B, introduces the C++ standard library from the outset. It helps you to write useful programs without the need to master every aspect of C++ language. The books cover many examples, and it also demonstrates how to make the best use of them.

2. C++ Pocket Reference 1st Edition: Accelerated C++: Practical Programming, written by Andrew Koenig. This introductory book that takes a practical approach to solve problems using C++. It covers a wide der scope of C++ programming than other introductory books.

3. C++ in One Hour a Day, Sams Teach Yourself 8th Edition: This book presents the language from a practical point of view. It helps you to learn how to use C++ to create faster, simpler, and more efficient C++ applications. You can understand how C++ features help you write efficient code using concepts like move constructors, lambda expressions, and assignment operators.

4. C++ All-in-One For Dummies 3rd Edition: C++ All-in-One For Dummies, 3rd Edition is an ideal handbook to C++ programmers. Author John Paul Mueller is a recognized authority in the computer industry which your ultimate guide to C++. This C++ book teaches you how you can work with objects and classes. It helps you to learn advanced coding skill using various C++ concepts.

5. C++: The Complete Reference Fourth Edition: The C++ Pocket Reference is a memory aid for C++ programmers written by Kyle Loudon. This pocket-sized reference book makes an ideal reference book to carry about, ensure that it will be handy when needed.

6. Starting Out with C++ from Control Structures to Objects plus: This C++ book was written by Tony Gaddis's accessible. It is step-by-step presentation helps the beginner as well as experienced developers. It helps them to understand important concepts of C++ programming language.

7. A Tour of C++ (C++ In-Depth Series) 1st Edition: Bjarne Stroustrup's book A tour of C++. It offers complete references for C++ core concepts and practical coding to give an experienced programmer to get what constitutes modern C++. This concise book contains a self-contained guide.

8. Effective Modern C++

9. C++ Standard Library, The: A Tutorial And Reference 2Nd Edition: The book offers comprehensive documentation of each library component, which includes an introduction to its purpose and design.

10. Accelerated C++: Practical Programming by Example 1st Edition: This is an advanced C++ learning book written by Scott Meyers. The book includes topics like The pros and cons of braced initialization, perfect forwarding, except specifications, and smart pointer makes functions. The relationships among std:: move, std:: forward. It also overs techniques which helps you to write correct, useful lambda expressions.

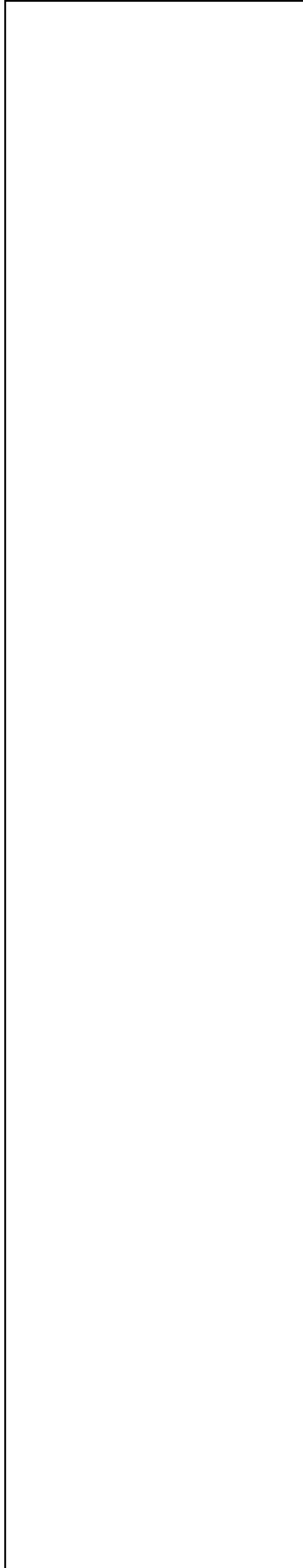
11. More Effective C++: This is an ideal C++ reference book written by Scott Meyers. It offers many C language reference and teaches C as the subset of C++ This book illustrates the C++ language with good examples throughout. It is highly recommended as a reference book.

12. Object-Oriented Programming In C++ 4th Edition: Robert Lafore wrote object-oriented Programming in C++. The book starts with the basic principles of the C++ programming language. It gradually introduces increasingly towards advanced topics.

13. C++ Programming: From Problem Analysis to Program Design 3rd Edition: C++ Programming: From Problem Analysis to Program Design, Third Edition is a book written by D.S. Malik. This programming book also teaches OOD methodology of sorting algorithms. It also teaches how to present additional material on abstract classes.

14. C++: A Detailed Approach to Practical Coding: A Detail approach to Practical Coding is a second book written by Nathan Clark. The author shares his 20 year's programming experience in this book. This book serves acts as a teaching guide and also a reference manual to accompany you through this wonderful world of programming.

15. C++17 STL Cookbook: This book helps you to understand the language's mechanics and library features and offers insight into how they work. The book takes an implementation-specific, problem-solution approach that helps you resolve such issues. It also covers core STL concepts, like containers, algorithms, lambda expressions, iterators.



UNIT 11: FILE HANDLING

Unit Structure:

11.1 Introduction
11.2 Unit Objectives
11.3 Basics of File Handling in C++
    11.3.1 File streams
11.4 Opening and Closing File
    11.4.1 Opening File
    11.4.2 Closing File
    11.4.3 end-of-file File detection
11.5 File Pointer
11.6 Sequential Access of File
11.7 Random Access File
11.8 Error Handling
11.9 Summing Up
11.10 Answers to Check Your Progress
11.11 Possible Questions
11.12 References and Suggested Readings

11.1 INTRODUCTION

In the previous units, we have discussed about the basic approaches of object-oriented programming and its importance. We have already seen a number of programming examples where we use cin and cout with operators >> and << for the input and output operations respectively. But all the examples taken in previous units works only in run time, there is no provision for storing data on computer and readthose data when we execute the program in the next time.

Through this unit, we aim to present what the file is, how files are accessed in C++. We will discuss programs for reading and writing in file. We have also described the file stream operations and use of buffer and pointers for manipulating files.

11.2 UNIT OBJECTIVES

At the end of the unit, you should be able to:

- understand the basic concept of file;
- open and close files for various I/O operations;

- Sequential access and random access of files;
- Importance of file in programming;
- manage buffer and pointers for I/O from files; and
- obtain a thorough understanding and practice of using file I/O functions in C++.

11.3 BASICS OF FILE HANDLING IN C++

A file is a container in a computer system that stores data, information, settings, or commands. There are several types of files available, such as directory files, data files, text files, binary file and graphic files and these several kinds of files contain different types of information. In the computer system, files are stored on hard drives, optical drives, discs, or other storage devices. In most of the operating systems, a file must be saved with a unique name within a given file directory. File handling in C++ is a mechanism to store the output of a program in a file and help perform various operations on it. Files help store these data permanently on a storage device.

The term “Data” is commonly referred to as known facts or information. Data are analysed to describe, diagnose, predict or prescribe its features. But to achieve it, we need to store the data.

Many real-world applications require reading and writing large number of data items. Usually, large volumes of data are need to be stored as files on disk. C++ provides mechanism for read and write data items from files. C++ file handling mechanism is quite similar to console input-output operations (cin and cout). It uses streams (called file streams) as an interface between programs and files. The Figure 11.1 illustrates the use of file streams for reading and writing from files.

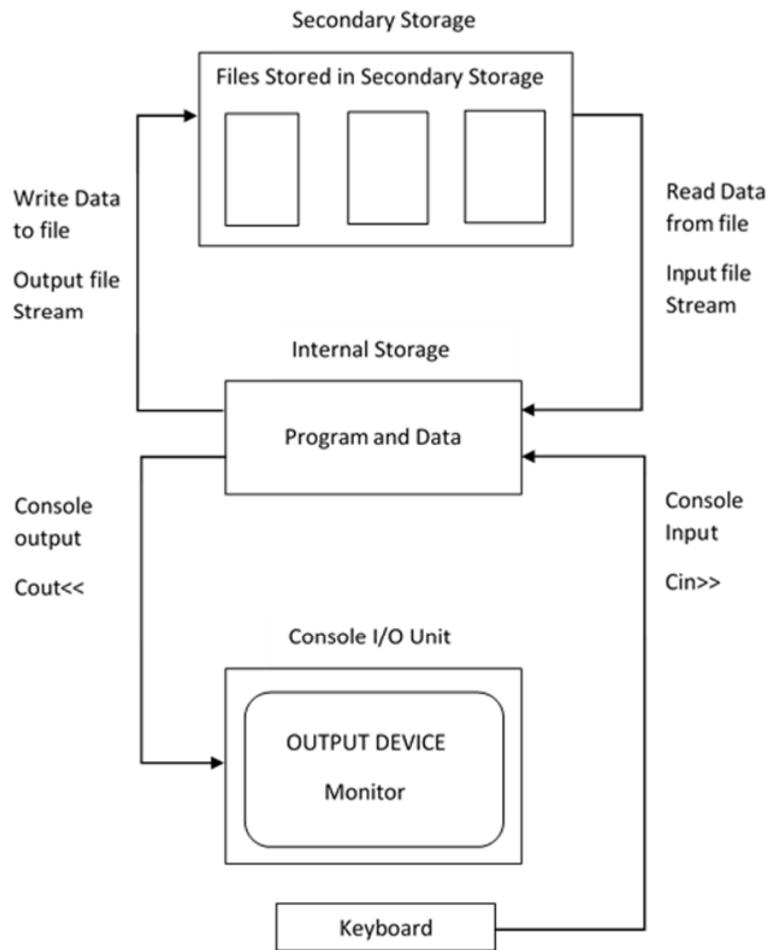


Figure 11.1 File Input and Output Steams

The Figure 11.1 shows that a program that reads data from file stored in secondary storage devices and writes data to the file. The input stream is connected to file used for read and output stream is connected to file used for write into the file. Same programs performing read and write operationsfrom/to file can also do reading and writing from standard input output devices using console I/O cin and cout at the same time.

11.3.1 FILE STREAMS

A stream is an abstraction that represents a device where input and output operations are performed. We can represent a stream as a destination or a source of characters of indefinite length. C++

contains a set of classes which provides us a library where methods for reading and writing from/to files defined. The classes that contain these methods are ifstream, ofstream and fstream. All these classes are derived from fstreambase class and also inherits features from iostream class.

- ofstream: This Stream class signifies the output file stream and is applied to create files for writing information to files. The ofstream class provides functions for output operations and include functions like put(), write() etc.
- ifstream: This Stream class signifies the input file stream and is applied for reading information from files. The ifstream class provides functions for input operations. This includes functions like get(), getline(), read(), seekg() etc.
- fstream: Both the classes ofstream and ifstream are declared in fstream and that is why this header file is included in all programs handling files. The fstream class provides support for simultaneous input and output operations. It also inherits all functions from istream and ostream classes through iostream.
- fstreambase: The fstreambase class provides operations common to the file streams ofstream, ifstream and fstream. It contains open() and close() functions. It also serves as base class for these three file stream classes.
- filebuf: The filebuf is used for setting the file buffers for read and write operations.

C++ provides us with the following operations in File Handling:

- Creating a file: open()
- Reading data: read()
- Writing new data: write()
- Closing a file: close()

11.4 OPENING AND CLOSING FILE

In most of the operating systems, a file must be saved with a unique name within a given file directory. However, certain characters cannot be used during creating a file as they are considered illegal. A filename consists of a file extension that is also called a suffix. The file extension contains few characters,

that follow the filename, and it helps to recognize the file format, type of file, and the attributes related to the file.

11.4.1 Opening File

A file must be opened before we can read from it or write to it. To open a file, we need the following

1. Valid file name
2. Structure or types of data stored in file
3. Opening method.

Before opening a file, we must check in the Secondary storage devices that whether the file exists in the secondary storage device or not. If the file already exists, we can open the file directly otherwise we need to create the file, then only we can open.

Before opening a file, we need to create the object of the respective stream classes and then we need to link the stream classes with the file via filename. As discussed earlier we can use either ofstream or fstream stream to open a file for writing and ifstream stream to open a file for reading purpose only. A file can be opened by one of the two following methods.

1. Using Constructor of stream classes
2. Using open function

Opening File using Constructor

We have already studied about constructors; constructors are used to initialize the class object when it is created. Here, we pass the filename as argument to constructors of steam class. Example-

```
ofstream fileout("student.txt");
```

Here fileout is the object of stream class ofstream, and student.txt is the filename. Here student.txt file is opened for writing, the object fileout will handle this file in the program. Similarly, in the following example data.txt file is open for reading

```
ifstream filein("data.txt");
```

Like “cin” and “cout” with the help of stream class object we can read and write to the file. For example

```
fileout<<"I am a student";
fileout<<strng1;
filein>>strng2;
```

we can also use the same file for read and write data as follows:

fstream fileinout(“course.txt”);

Opening File using open function

As stated earlier, the function open() can also be used to open a file. To open a file using open() function, we need the stream object and the stream object will same way handle the file as it handles it in the earlier example. This is done as follows:

stream-class stream_class_object

stream_class_object.open(“File_name”)

The above program can also be written using open function:

/* C++ program to Open and Close a File*/

/*This program demonstrates, how to open a file to store or retrieve information to/from it. And then how to close that file after storing or retrieving the information to/from it. */

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
	ofstream fout;
	ifstream fin;
	char fname[20], rec[80], ch;
	cout<<"Enter file name: ";
	cin>>fname;
	fout.open(fname);
	if(!fout)
	{
		cout<<"Error in opening the file "<<fname;
		exit(1);
	}
	cout<<"\nEnter a word to store in the file:\n";
	cin>>rec;
	fout<<rec;
	fout.close();			// File Close
	fin.open(fname);
	fin>>rec;
	cout<<"\nThe file contains:\n";
```

```
        cout<<rec;
        fin.close();
}
```

OUTPUT:

```
Enter file name: xyz.txt
Enter a word to store in the file:
student
The file contains:
student
```

File opening modes

C++ allows us to open file in different modes. In the open() function we can specify the modes of file to open. The syntax of open() is as below:

filestreamobject.open("filename", mode);

Mode specifies the modes of opening the file. Modes of file opening is defined in ios class.File Modes can be any of the following:

Mode	Description
ios::app	Append at the end of file
ios::ate	Move the File Pointer to end-of-file on opening.
ios::binary	To open a binary file.
os::in	To open file only for input.
ios::nocreate	Open the file if it already exists otherwise fails to open file.
ios::noreplace	Open only already existing file.
ios::out	Open file only for output.
ios::trunk	Delete the contents of the existing file and open as a blank file.

When we don't provide the modes of opening a file then default mode is used by open() function.

Default modes are:-

ifstream ios::in

ofstream ios::out

fstream ios::in | ios::out

Following program explain how open function works:

```
#include<iostream>
#include<fstream>
#include<string.h>
using namespace std;
int main()
{
	ofstream fout;
	ifstream fin;
	char fname[20];
	char string[80], ch;
	int len;
	cout<<"\nEnter a string to store in the file:\n";
	gets(string);
	cout<<"Enter file name: ";
	cin>>fname;
	len=strlen(string);
	fout.open(fname, ios::app);
	if(!fout)
	{
		cout<<"Error in opening the file "<<fname;
		exit(1);
	}
	for(int i=0;i<len;i++)
	fout.put(string[i]);
	fout.close();		// Close File open for output
	fin.open(fname, ios::in);
	fin.getline(string,80);
	cout<<"\nThe file contains:\n";
```

```
        puts(string);
        fin.close();
}
```

OUTPUT:

Enter a word to store in the file:
I am a student of IDOL.
Enter file name: xyz.txt
The file contains:
I am a student of IDOL.

11.4.2 Closing A File

A file is automatically closed when we go outside of the scope of stream object. We can also close a file using close()function call.

```
        fileout.close();
```

Closing a file will release the stream object. After closing one file we can open another file in the same object

Following programming example explain the operation on file

```
/* C++ program to Open and Close a File*/

#include<iostream>
#include<fstream>
using namespace std;
int main(){
        char fname[20];
        char rec[80], ch;
        cout<<"Enter file name: ";
        cin>>fname;
        ofstream fout(fname);
        if(!fout){
                cout<<"Error in opening the file "<<fname;
                exit(1);
        }
        cout<<"\nEnter a word to store in the file:\n";
        cin>>rec;
        fout<<rec;
        fout.close();
        ifstream fin(fname);
```

```
        fin>>rec;
        cout<<"\nThe file contains:\n";
        cout<<rec;
        fin.close();
}
```

OUTPUT:

```
Enter file name: abc.txt
Enter a word to store in the file:
student
The file contains:
student
```

11.4.3 End-of-File File Detection

End-of-file is necessary to detect the end of the file while reading a file. To detect the endof file, eof() function is used. The eof() function is defined in **ios** class. eof() function return nonzero (TRUE) value when there are no more data to be read from an input file and zero (FALSE) otherwise.

Rules for using end-of-file (eof()):

1. Always test for the end-of-file condition before reading data from file.
2. Use a while loop for getting data from the input file stream. A for loop is desirable only when we know the exact number of data items. In case of file, we do not know the exact numbers of data in the file.

Following programming example explains the use end-of-file.

```
#include <iostream>
#include <fstream>
#include <string.h>
using namespace std;
int main(){
        ifstream fin;   // object for reading from the file
        char fname[20];
        char rec[80], ch;
        cout<<"Enter file name: ";
        cin>>fname;
```

```
        fin.open(fname);        //open the file for reading
        cout<<"\nThe file contains:::\n";

        while(fin.eof() = = 0) //  End-of-file detection
        {
           fin.getline(rec,80); //read one line at a time from file
           cout<<rec<<"\n";
        }
        fin.close(); // close the file open for reading
}
```

OUTPUT:

```
Enter file name: data.txt

The file contains:::
Hi,
I am Rahul.
I am a student of IDOL.
I am from Guwahati.
```

CHECK YOUR PROGRESS - I

1. Which header file is required to use file I/O operations?

2. _______ used to create an output stream?

3. _______ used to create a stream that performs both input and output operations?

4. Which of the following is not used as a file opening mode?

a) ios::trunc b) ios::binary

c) ios::in d) ios::ate

True or False?

5. It is not possible to combine two or more file opening mode in open() method.

6. ios::in and ios::out are input and output file opening mode respectively.

7. Use of ios::trunc mode is to truncate an existing file to half

8. Default mode for opening file using ofstream class is ios::out

11.5 FILE POINTER

Each file in C++ is associated with file pointer. We can control the position of reading and writing operations in a file by manipulating the file pointers. That is, we can write anywhere or read from any location by manipulating the file pointer. Every file in C++ is associated with two pointers, one for input (get pointer) and other for output (put pointer).

File Pointer	Description
get	The get pointer allows us to read the content of a file when we open the file in read-only mode. It automatically points at the beginning of file, allowing us to read the file from the beginning.
put	It allows us to write the content to a file, when we open the file in *write-only* mode. It automatically points at the beginning of a file, starting us to write the content of a file from the start.

The get pointer is used to set a specified location in order to read from that desired location in the file. These pointers are automatically incremented every time after read and write operation. Every time we open a file for input, the input pointer is automatically set to the beginning of the file.

When we open the file in a read-only mode i.e. "ios::in", the get pointer is automatically set the beginning of the file, to allow us to read from the start. But C++ provides some functions which allow us to set the get pointer at the particular location in the file.

Following are the functions used to set the get pointer

Functions	Description
tellg()	Gives us the current location of the ***get*** pointer. When the file is opened in a *read-only* mode, tellg() returns zero i.e. the beginning of the file.
seekg(offset, refposition)	This function is used to set the location of the ***get*** pointer to a desired position/offset. The **position** variable is the new position in the file i.e. *an integer value representing the number of bytes from the beginning of the file*.

The syntax of seekg() function is

seekg(offset, reposition);

Here, the argument ***offset*** represents the number of bytes the file pointer to move from the location specified in refposition. The argument ***refposition*** can have any one value from following:

ios::beg To set the pointer at start of the file

ios::end To set the pointer at end of the file

ios::cur Current position of the pointer

The following program illustrates the use of seekg() and tellg() function:

```
//Modifying the data of file using pointer:

#include<iostream>
#include<fstream>
using namespace std;
int main()
{
	ifstream inf;
	inf.open("data.txt", ios::in);
	cout<<"The first location is : " <<inf.tellg() << "\n";
	char ch;
```

```
        cout<<"\n File is : \n";
        while(inf)
        {
        ch = inf.get();
        cout<<ch;
        }
        inf.clear();
        cout<<"\n\nReading the file skipping 5 character : \n";
        inf.seekg(5, ios::beg);
        while(inf)
        {
        ch = inf.get();
        cout<<ch;
        }
        return 0;
}
```

OUTPUT:

```
The first location is : 0
 File is :
I studying in Final Semester.
Reading the file from 5th character :
dying in Final Semester.
```

In the program

inf.seekg(5, ios::beg);

This code is used to set the get pointer at beginning of the file (ios::beg) and the pointer will skip 5 position from beginning (since offset is 5)

inf.clear();

Once we reach the EOF(End of file), the EOF flag is set and it does not allow us to access of file again for reading. Hence, we need to clear it. To clear EOF flag, clear() function is used.

We have discussed the get pointer, which is used to read the content of a file and some functions to manipulate the get pointer. Similar to get pointer ***put*** pointer is used to write the content to the file.

The features of put pointer are:

It allows us to write the content to a file, when we open the file in write or append mode.

It automatically points at the beginning of a file, when we open the file in write-only mode.
It points at the end of a file, when we open the file in append mode.
Similar to get function two functions are associated with put function. Following are two functions used along with put file pointer:

Functions	**Description**
tellp()	Gives us the current location of the ***put*** pointer. When the file is opened in a *write-only* mode, tellp() returns zero i.e. the beginning of the file.
seekp (offset, refposition);	This function is used to set the location of the ***put*** pointer to a desired position/offset. The *position* variable is the new position in the file i.e. *an integer value representing the number of bytes from the beginning of the file.*

The syntax of seekp() function is:

seekp(offset, reposition);

Here the argument ***offset*** represents the number of bytes the file pointer to move from the location specified in ***refposition***. The argument ***refposition*** can have any one value from following

ios::beg To set the pointer at start of the file
ios::end To set the pointer at end of the file
ios::cur Current position of the pointer

11.6 SEQUENTIAL ACCESS OF FILE

A Sequential file has to be Accessed in the same order the file was written. For example, a cassette tapes: we can play the music's in the tape same order as it was recorded. In the tape we can only fast-forward or rewind over songs but we cannot go to a specific song just typing any index. This type of files is called sequential files. In this type of file, it is difficult and sometimes impossible to write data in the middle or to delete data from middle.

Following are the few popular functions used in sequential access of file:

put() function: put() function, used to write a character to the file.The put function allows us to write one character at a time to a file. Syntax of put() function is as bellow-

```
void put(char ch);
```

When we use put function with file stream object we call the function with the help of stream object. For example

```
outf.put(ch);
```

get() function: get() functionis used to read a character from the file. The get function allows us to read one character at a time from a file. Syntax of get() function is as bellow-

```
char get();
```

When we use get function with file stream object we need to call the function with the help of stream object. For example

```
ch = inf.get();
```

getline() function: getline() is a standard library function that is used to read a string or a line from an input stream. It is defined under <string> header file. The getline() function extracts characters from the input stream and appends it to the string object until the delimiting character is encountered.

read() function: read() is a binary function, used to read (input) data from file. It is used to read objects stored in file. The syntax of read() is

```
read( (char *) & ob, sizeof(ob));
```

read() function with file stream object is used as follows

```
inf.read((char*)&V, sizeof(V));
```

where V is the variable. The function took two arguments address of the variable ((char*)&V) and size of the data (sizeof(V)) to store.

write() function: write() is also a binary function, used to write (output) data on to the file. It is used to write the objects to a file, which is stored in binary form.The syntax of write() is

```
write( (char *) & ob, sizeof(ob));
```

write() function with file stream object is used as follows

```
outf.write((char*)&V, sizeof(V));
```

where V is the variable. The function took two arguments address of the variable ((char*)&V) and size of the data (sizeof(V)) to store.

```
//Programming example of read() and write() function:
#include<iostream>
#include<fstream>
using namespace std;
class student {
private:
	char name[40];
	int rollno;
	int semester;
public:
	void putdata();
	void getdata();
};
void student :: putdata() {
	cout<<"Enter the name : ";
	cin.getline(name,40);
	cout<<"Enter the Roll No : ";
	cin>>rollno;
	cout<<"Enter the semester : ";
	cin>>semester;
}
void student :: getdata() {
	cout<<"The name is : " << name << "\n";
	cout<<"The roll no is : " << rollno << "\n";
	cout<<"The semester is : " << semester << "\n";
}
int main() {
	ofstream outf_b;
	outf_b.open("data.txt", ios::out);
	student ob1;
	cout<<"\nWriting data to file:: \n";
	ob1.putdata();
	outf_b.write( (char *) & ob1, sizeof(ob1));
	outf_b.close();
	ifstream inf_b;
	inf_b.open("data.txt", ios::in);
	student ob2;
```

```
        cout<<"\nReading the object from a file : \n";
        inf_b.read((char *) & ob2, sizeof(ob2));
        ob2.getdata();
        inf_b.close();
        return 0;
}
```

OUTPUT:

Writing data to file::
Enter the name : Rahul Barman
Enter the Roll No : 11
Enter the semester : 2
Reading the object from a file :
The name is : Rahul Barman
The roll no is : 11
The semester is : 2

11.7 RANDOM ACCESS FILE

A random-access file behaves like a large array of bytes. In array we have index and in file we have file pointer, by manipulating the file pointer we perform the read write operations. If the end-of-file is reached before the desired number of bytes has been read than EOF error occurred.

Unlike sequential access file where data in the files are read one by one, in random access file data can be read in any order. By controlling this file pointer, we can move forth and back within the file to perform reading and writing operations at any position we want in random fashion.

The random-access files are useful where data needs to be updated frequently. With the help of random-access file, we can perform the following operation-

1. Updating any data in the file
2. Displaying any part of file
3. Add new data to file
4. Delete existing data

With the help of file pointer and the file pointer manipulator we can perform these tasks.

The following program illustrate the random-access file

```
#include<iostream>
#include<fstream>
using namespace std;
class student {
private:
	char name[40];
	int rollno;
	int semester;
public:
	void putdata();
	void getdata();
};
void student :: putdata() {
	cout<<"The name is : " << name << "\n";
	cout<<"The roll no is : " << rollno << "\n";
	cout<<"The semester is : " << semester << "\n";
}
void student:: getdata() {
	cout<<"Enter the name : ";
	cin>>name;
	cout<<"Enter the Roll No : ";
	cin>>rollno;
	cout<<"Enter the semester : ";
	cin>>semester;
}
int main() {
	student stu;
	fstream iofile;
	iofile.open("student.dat",ios::ate|ios::in|ios::out|ios::binary);
	iofile.seekg(0,ios::beg);
	cout<<"\n Current content of the file:: \n";
	while(iofile.read((char*)&stu, sizeof(stu))) {
		stu.putdata();
	}
	iofile.clear();
	cout<<"\n Add data in end of file:: \n";
	stu.getdata();
	iofile.write((char*)&stu, sizeof(stu));
```

```
        iofile.seekg(0, ios::beg);
        cout<<"\n Content of the file after adding data:: \n";
        while(iofile.read((char*)&stu, sizeof(stu))) {
                stu.putdata();
        }
        Int obj;
        cout<<"\n Enter the object no to update ::";
        cin>>obj;
        int location=(obj-1)*sizeof(stu);
        if(iofile.eof())
        iofile.clear();
        iofile.seekp(location);
        cout<<"\nEnter the new values of the student\n";
        stu.getdata();
        iofile.write((char*)&stu, sizeof(stu));
        cout<<"\n Updated File:: \n";
        iofile.seekg(0);
        while(iofile.read((char*)&stu, sizeof(stu))) {
                stu.putdata();
        }
        iofile.close();
        return 0;
}
```

OUTPUT

Current content of the file::

The name is : Rahul

The roll no is : 7

The semester is : 3

The name is : Karen

The roll no is : 6

The semester is : 2

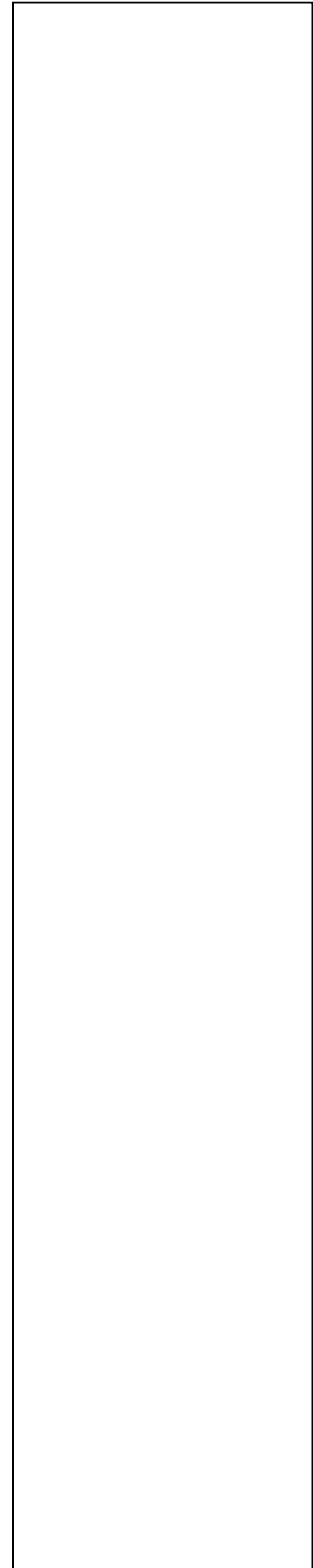
Add data in end of file::

Enter the name : Zuman

Enter the Roll No : 3

Enter the semester : 1

Content of the file after adding data::



The name is : Rahul

The roll no is : 7

The semester is : 3

The name is : Karen

The roll no is : 6

The semester is : 2

The name is : Zuman

The roll no is : 3

The semester is : 1

Enter the object no to update :: 2

Enter the new values of the student

Enter the name : Suman

Enter the Roll No : 5

Enter the semester : 1

Updated File::

The name is : Rahul

The roll no is : 7

The semester is : 3

The name is : Suman

The roll no is : 5

The semester is : 1

The name is : Zuman

The roll no is : 3

The semester is : 1

11.8 ERROR HANDLING

It's quite common that errors may occur during file operations. There may have different reasons for errors while working with files. The following are few common problems that leads to errors

- When trying to open a file for reading which is not exist.
- When trying to read from a file beyond end of file.

- When trying to read from a file that has opened in write mode.
- When trying to perform a write operation on a file that has opened in read mode.
- When trying to operate on a file that has not been open.

Following are the functions for handle errors in file-

int bad()	It returns a non-zero (true) value if an invalid operation is attempted or an unrecoverable error has occurred. Returns zero if it may be possible to recover from any other error reported and continue operations.
int fail()	It returns a non-zero (true) value when an input or output operation has failed.
int good()	It returns a non-zero (true) value when no error has occurred; otherwise returns zero (false).
int eof()	It returns a non-zero (true) value when end-of-file is encountered while reading; otherwise returns zero (false).

CHECK YOUR PROGRESS - II

1. Which of the following is not used to seek file pointer?

a) ios::set b) ios::end

c) ios::cur d) ios::beg

2. Which function is used in C++ to get the current position of file pointer in a file?

a) tellp() b) getpos()

c) getp() d) tellpos()

3. Which function is used to reposition the file pointer?

a) moveg() b) seekg()

c) changep() d) go_p()

4. Which of the following is used to move the file pointer to start of a file?

a) ios::beg b) ios::start

c) ios::cur d) ios::first

State True or False

5. eof() returns true if a file open for reading has reached the end.

6. fileObject.seekg(ios::end, n) is the currect syntax for set file pointer back to n position from end of file.

7. The function tellp() can be used to detect end of a file.

8. A C++ program can do reading and writing on the same file.

11.9 SUMMING UP

- In C++ we have **ifstream, ofstream** and **fstream** stream classes to deal with file handling. These classes are derived from **fstreambase** class and declared in header file **iostream**.
- A file can be open in two ways, either by using constructor or by using member function **open().**
- In open() function we can set the mode of the file.
- Each file is associated with two pointer get and put.
- With the help of seekg() and seekp() function we can manipulate the pointers.
- Frequently used functions in input output operation in files are get(), put(), read(), write().
- Random access file is useful when we need correction on recorded file.
- Error handling functions in C++ are used to handle different errors occurred during program execution.

11.10 ANSWERS TO CHECK YOUR PROGRESS

Check your Progress - I

1. <fstream>
2. ofstream

3. fstream
4. a) ios::trunc
5. False
6. TRUE
7. FALSE
8. TRUE

Check your Progress - II

1. a) ios::set
2. a) tellp()
3. b) seekg()
4. a) ios::beg
5. TRUE
6. FALSE
7. FALSE
8. FALSE

11.11 POSSIBLE QUESTIONS

Short answer type questions:

1. What are the steps involved in creating a file in C++?
2. Describe the various classes available for file operation?
3. Explain the advantages of random-access file over sequential access file.
4. How end of file is detected? Explain with example.

Long answer type questions:

1. Explain the different methods for opening a file?
2. Explain the open() function? Explain the different modes available for opening a file?
3. Explain the different functions related to random access file with example?

4. What is error handling in file? What are the functions available for file error handling in C++?
5. Write a program to maintain library book record using file.
6. Write a program to maintain inventory for a departmental store using File

11.12 REFERENCES AND SUGGESTED READINGS

1) E. Balaguruswamy, Object Oriented Programming with C++, Tata McGraw Hill, 2010.

2) P. Deitel and H. Deitel, C++: How to Program, PHI, 7th edition, 2010.

3) B. Strousstrup, Programming – Principles and Practices using C++, Addison Wesley, 2009

BLOCK II:

OBJECT ORIENTED DESIGN

UNIT 1: INTRODUCTION TO OOD

Unit Structure:

1.1 Introduction
1.2 Unit Objectives
1.3 Brief History
1.4 Object-Oriented Software Development Methodology
1.5 Object-Oriented Approaches
1.6 Object-Oriented Analysis
    1.6.1 Steps of performing object-oriented analysis
1.7 Object-Oriented Design
    1.7.1 Goals of Object-Oriented Analysis and Design
1.8 Object-Oriented Programming
    1.8.1 Objects
    1.8.2 Classes
1.9 Different Concepts of Object-Oriented Design
    1.9.1 Abstraction
    1.9.2 Encapsulation
    1.9.3 Inheritance
    1.9.4 Modularity
    1.9.5 Polymorphism
1.10 Generalization and Specialization
1.11 Link and Association
    1.11.1 Degree of Association
    1.11.2 Multiplicity
1.12 Summing Up
1.13 Answers to Check Your Progress
1.14 Possible Questions
1.15 References and Suggested Readings

1.1 INTRODUCTION

The object-oriented programming is widely used to solve different kinds of computing problems. The concept of object-oriented programming is to find solution to the problem based on real-life examples. Object-oriented design is used to prepare a plan to solve the software related problem diagrammatically. The whole design is based on objects and classes where objects interact with each other

to solve a problem. In this unit, we will learn the object-oriented approaches, the key concepts of object-oriented design and different components of it.

Object-oriented design is a process to design software. The fundamental steps of developingsoftware are: Requirement analysis, designing, coding, testing and maintenance of the software. There are many techniques and tools to develop software. The need for a tool may vary as per the requirement of the user and the change in the technology. Among all other design strategies, object-oriented development techniques are a popular one.

1.2 UNIT OBJECTIVES

The main objectives of this unit are:

- To familiarize with the object-oriented programming history
- To know the purpose and benefits of using object-oriented approaches
- To have a clear concept on different concepts related to object-oriented design
- To gather the knowledge on the importance of object-oriented analysis and design
- To know the use of link, association and basics object-orienteddesign

1.3 BRIEF HISTORY

In this section, we will show the evaluation of object-oriented programming with time:

- The term ‘object’ and ‘oriented’ are in discussions at MIT in the early 1960s.
- Alan Kay presented a detailed concept on “Object-Oriented Programming” in 1966-1967.
- In MIT, an ALGOL version named AED-0 was initiated which was capable of establishing a link between the data structures and procedures.
- A programming language name Simula was designed using object-oriented concept in the late 1960s. It was capable of run on the UNIVAC 1107 computer. Simula introduced the

fundamental concepts of OOP like class, object, inheritance and dynamic binding.

- The first version of Smalltalk programming language was built by Alan Kay, Dan Ingalls and Adele Goldberg inthe 1970s.
- Inmid-1980s, Brad Cox developed Objective-C which proved to be new beginning in OOP history. In parallel to this, Bjarne Stroustrup created the Object-Oriented C++.
- Grady Booch proposed the design concept in a programming language in a paper titledObject Oriented Design.
- In the 1990s, the popularity of using object-oriented programming began using the languages like C++, Visual FoxPro 3.0 etc.
- Eventually, object-oriented concepts were added to many languages like ADA, Fortran, Pascal, Basic, COBOL etc.
- In present days, languages like Python, Ruby, JAVA by Sun Microsystems, C#, Visual Basic.NET have completely emerged as object-oriented programming languages. And these languages are widely used in various fields.

1.4 OBJECT-ORIENTED SOFTWARE DEVELOPMENT METHODOLOGY

This methodology helps the programmer to work based onobject-oriented concepts to solve the problems. Object-oriented software development methodology includes the concept of requirement analysis, object-oriented design and implementation of the system. Unlike the traditional software development methods, it focuses on developing software using objects. These objects can be easily created, removed, modified and reused. Objects are also able to communicate between themselves.

1.5 OBJECT-ORIENTED APPROACHES

The object-oriented paradigm focuses on solving a problem using both theoretical and conceptual knowledge. Here, a system is assumed as a collection of various entities which are able to work and interact together to fulfil certain objectives. These entities may be physical (like animal, flower, person etc.) or maybe abstract

concepts (like files, function etc.). In object-oriented analysis, these entities are known as objects.

In an object-oriented approach, these objects consists of data and procedures. The main aim is to propose a design to improve the productivity and quality of the system. The whole object-oriented programming approach can be partitioned into three main phases. The first phase is Object-oriented Analysis (OOA) phase, where the requirements are analysed and the problem domain is identified in terms of objects. The Object-oriented design phase works to design a solution domain based on the objects identified in the previous phase. Object-oriented programming works on the implementation of the system using object-oriented concepts.

The benefits of object-oriented programming approaches are:

- Object-oriented system model works in more organized way than other traditional approaches.
- Object-oriented system can be designed and code easily.
- The maintenance cost is comparatively low in this approach.
- Object-oriented programming allows the reusability of design and code.
- This programming is more adaptive in nature. Modifications can be done easily as per requirement.
- This programming is reliable and more flexible for the programmers.

1.6 OBJECT-ORIENTED ANALYSIS (OOA)

Object-oriented Analysis is the starting phase of the object-oriented software development procedure. This phase gathers all the requirements to develop the system and identifies all the classes and relations between the classes. Objects are the instances of the class. All the requirements are organized as objects. OOA focuses to create real-world models using the object-oriented view of the real world.

STOP TO CONSIDER

Grady Booch coined the term Object-Oriented Analysis as "Object-Oriented Analysis is a method of analysis that examines requirements from the perspective of classes and objects found in the vocabulary of the problem domain".

1.6.1 Steps of Performing Object-Oriented Analysis

The working of OOA is shown in **Figure 1.1**. The key steps of performing **object-oriented analysis** are:

Defining the Problem: The first step in OOA is to analyse the problem for better understanding. The problem needs to be studied in detail and redefine the problem in engineering, so that a computer-based solution can be prepared stepwise from the problem definition using object-oriented concepts.

Requirement Analysis: The second step is to gather all the user requirements. From this, we can proceed to propose the solution domain in the design phase. A requirement specification document should be prepared based on all user requirements and the software requirement for solving the problem. The specification document should contain what the system actually does, what type of inputs are necessary, what are the outputs that system produces and how processes can be built to generate required outputs.

Identification of Objects: In this step, we need to collect all the objects and list the attributes contained in the object based on the requirement specification document built in the previous step. Objects can be related to real-life entity or can be of an abstract type.

Deciding the services of objects: After the identification of objects, the next step is to identify the services to be provided by each object. Each object is assigned some services to be performed. The strength of the object-oriented paradigm is that once the services are provided to the objects, it is guaranteed to be accomplished by the objects.

Establishing relationship among the objects: Objects need to interact among themselves to perform different services to produce

better results. This works on identifying the relationships among the objects so that objects can communicate during execution.

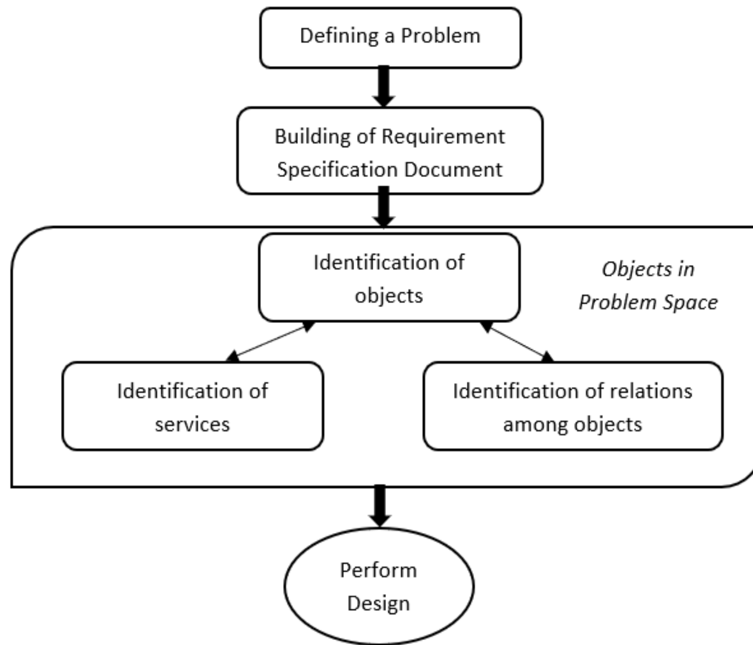


Figure 1.1: Working of object-oriented analysis

1.7 OBJECT-ORIENTED DESIGN (OOD)

The classes identified in the OOA phase are designed in the design phase. This phase also helps to build the user interfaces. As per the requirement, more numbers of classes and objects can be added in the design phase. Object-oriented design helps to build the solution domain by identifying the classes, relationship between them, identifying the constraints and designing the user interfaces. The outputs of object-oriented analysis are taken as input to the object-oriented design.

Object-oriented design follows the concept of **decomposition**. Here, decomposition means to divide the whole system into the hierarchy of various components. Each component has its own characteristics and functions. These smaller components are known as subsystems. OOD establishes communication between the subsystems using corresponding objects. The main advantage of decomposing the system in OOD is that all the subsystems are of lesser complexity, so they can be easily understandable and manageable. And the

subsystems can be easily modified or replaced depending on the situation without affecting the other subsystems. In OOD, to show the interaction between two classes and objects, associations and links are used.

Object-oriented design is composed of two main types of design: Object Design and System Design. We will study this detailed design in Unit 3. The models built using OOD are either static models or dynamic models. Based on that OOD can be categorized as of mainly three models:

i) Object Model
ii) State Model or Dynamic model
iii) Functional model

These three models can be represented with different diagrams. Object model can be represented with **class diagram** and **object diagram**. State models can be drawn **using state diagram**. **Data Flow Diagram (DFD)** is generally used to represent functional model. We will learn the functionalities of these three models and to design their respective diagrams using **Unified Modeling language (UML)** in the next unit.

In present days, many of the software designer use another model known as **interaction model**instead of functional model. The state diagrams of dynamic model focus on their respective class. But sometimes it becomes difficult to understand the entire system. Interaction model focuses on the relationship between the classes, how the objects interact with others to display efficient result. To present the interaction model, **Use Case diagrams, Sequence diagrams** and **Activity diagrams** are commonly used.

STOP TO CONSIDER

Grady Booch defined the term OOD as, "Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design".

1.7.1 Goals of Object-Oriented Analysis and Design (OOAD)

Both OOA and OOD are together known as **Object-Oriented Analysis and Design (OOAD)** which includes both the requirement analysis, identification of classes and design part of the object-oriented system. The different objectives or goals of object-oriented design and analysis are:

The main objective of performing object-oriented analysis and design is that it helps the programmer to build an efficient solution to complex problems based on the object-oriented concepts.

- Following the object-oriented analysis and design step, software development team management can have a better understanding of the problem domain.
- Graphical representations of the problem make it easier for the team management, stakeholders and the end-user to work together and check whether the software building process is on the right track or not.
- It increases the reusability of the project
- It is performed at the abstract level which helps to separate the actual programming part from the design.

1.8 OBJECT-ORIENTED PROGRAMMING (OOP)

Object-oriented programming is a pattern of writing programs where data and functions are put together in a skeleton-like structure called class, where data need to be processed and functions need to be performed. And the whole operation is executed using any instance of a class known as an object. There can be multiple classes in a program and ahierarchy of classes can be maintained using the concept of inheritance. Along with this, object-oriented programming follows the concepts of abstraction, encapsulation and polymorphism. This programming technique uses different access specifier to maintain the data hiding property.

Data hiding properties protect the data and member functions of a class from unauthorised access outside the class. The reason behind the popularity of object-oriented programming language is that it is intact with real-world examples and so, has the capability of solving complex problems in a better way.

STOP TO CONSIDER

Grady Booch defined the term OOP as, "Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of hierarchy of classes united via inheritance relationships".

1.8.1 OBJECTS

An object can be said as an identity that can be categorized by its behaviour. It is a component consisting of different properties and methods to make data useful. M. R. Blaha and J. R. Rumbaugh defined the term object in their book [2] as, "An object is a concept, abstraction, or thing with the identity that has meaning for an application". Simply, an object is called as an instance of a class. No memory is allocated during the defining of class. Memories are allocated to the objects of a class based on the size of data members (variables, constants etc.) defined inside the class.

Objects can be related with real life entity. For example, if we consider a class name as *flower*, then a single flower of any kind (say *lotus* or *rose*) can be considered as an individual object. Objects can also be of an abstract type which has conceptual existence. For example, if we consider a class name as *examination*, here different subjects like *subject1, subject2* etc. can be objects.

1.8.2 CLASSES

We have read an object is an instance of a class. So, a class is collection of similar kind of objects which possesses similar characteristics. A class is skeleton of program which consist of data members and the member functions. The term data member refers to the variable or constants declared within a class. The scope of these variables is within this class. The member functions are the procedures or functions defined in the class. The class does not consume any memory. These data members and member functions of the class can only be accessedthrough the objects of that class. So, the objects consist of all the characteristics defined in the class.

All the objects have the same number of attributes and properties. By grouping set of similar objects relating to a class, the concept of abstraction is introduced in the object-oriented programming. As in the previous example, all the properties of a flower are stored into the class *flower*. Any objects of class *flower* possess similar kind of properties.

Later in this unit and next unit, we will learn how to draw class diagram and object diagram in OOD.

CHECK YOUR PROGRESS - I

Fill in the blanks:

1. ____________________ is collection of similar kind of objects which possesses similar characteristics.
2. instance of a class known as an ________________

___________ properties protect the data and member functions of a class from unauthorised access outside the class

3. ____________________ design is a process to design software.

1.9 DIFFERENT CONCEPTS OF OBJECT-ORIENTED (OOD)

In this section, we will study important concepts related to object-oriented design and object-oriented programming.

1.9.1 Abstraction

Abstraction is a process of hiding the background detail from the user. It is one fundamental method to deal with complexities. We use different machines like any electronic device, electrical device, mechanical device etc. for their daily use. These devices work in a complex way as it looks from the outside. But to gather full knowledge on how exactly a device performs before using a device would be tougher for humans. People can use a device if they know how exactly the device operates or what will be the outcome based

on the given input. There is the concept of abstraction lying on. This concept allows hiding all the working procedures, functionalities that are not necessary for using a particular device. Let us take an example of an electric fan. We know that as we switch on the fan, it will start rotating and help us to cool within a certain period of time. When we switched off the fan, it stops. This information is enough for us to have the benefit of an electric fan. But behind this, many processes have to take place, conversion of energies has been going on. Abstraction is the concept to separating all the complex unnecessary information from the normal users.

In object-oriented programming also, the concept of abstraction is playing the same role. Abstraction focuses on to separate the important purpose from the unimportant aspects. For a particular thing, many different abstractions also can be possible based on the purpose of their use. The characteristics of a good model to identify the important information of a problem and removes the other.

STOP TO CONSIDER

Grady Booch defined the abstraction as "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer".

1.9.2 Encapsulation

Encapsulation is also based on the concept of information hiding.. Abstraction deals with the characteristics of the object but encapsulation focuses on the internal implementation for which characteristics of the objects are raised. Encapsulation hides the detailed information from the other objects. For this, the modification in any part of a code does not affect the whole program.

Encapsulation is a process of binding data with related methods and it prevents its accessibility from other objects. The programs must be encapsulated in the proper way. It is mainly used to hide the internal detail from the outside objects.

STOP TO CONSIDER

Grady Booch coined encapsulation as, "Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation".

1.9.3 Inheritance/ Hierarchy of Class

Inheritance is one of the key properties of object-oriented programming. Inheritance allows a class or object to inherit the characteristics of other classes or objects. The hierarchy of the classes states that one class can access the data and member functions of another class.

The class from which the properties can be inherited is known as base class or superclass or parent. The class which inherits the properties the parent class is known as derived class or subclass or child class. In C++, a child class can inherit the properties of multiple parent classes whereas a parent class can have multiple child classes. C++ supports different types of inheritance like single inheritance, multiple inheritance, multiple inheritance, hierarchical inheritance and hybrid inheritance. We have already studied the functionalities of different types of inheritance in this paper.

STOP TO CONSIDER

Grady Booch defined the term hierarchy as, "Hierarchy is a ranking or ordering of abstractions".

Here, we try to learn the organization of class hierarchies with an example. Let us take the example of different types of insurance policy. A person can obtain insurance from any brand or company. First the type of insurance is selected. Insurance can be of many types like health insurance, life insurance, motor vehicle insurance. All the insurance may have some common properties like brand name, number of years, policy number etc. But the features, procedures and benefits vary depending on type of insurance. So, in figure 1.2, the main or parent class is termed as *Insurance_Policy*

which may contain all the common properties. based on the type, it is divided into three child or subclasses, viz. *Health_Insurance*, *Life_Insurance*, *Motor_Vehicle_Insurance*. The second hierarchy level classes contain their features. Further, these child classes also can be decomposed into more subclasses. For example, the feature of health insurance may vary based on whether it is for self or for family. Like *Life_Insurance* class and *Motor_Vehicle_Insurance* class also can be divided. Thus, in OOP multiple classes can be organized in the level of hierarchy. Objects of lower-level classes can access the properties of their parent classes, whereas parent classes cannot access the properties of their child classes.

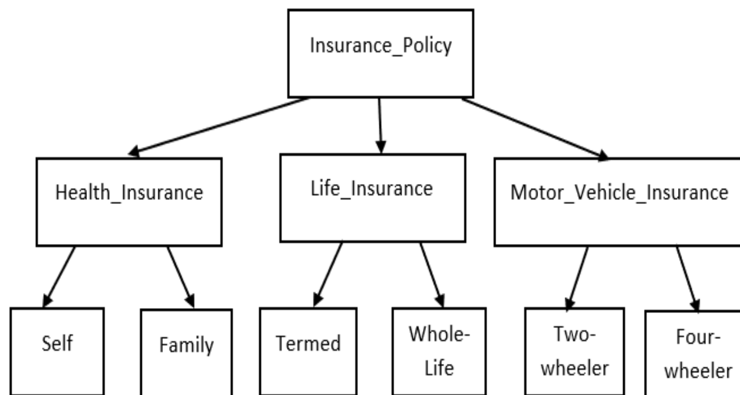


Figure 1.2: Hierarchy of classes types of insurance policy to understand inheritance

1.9.4 Modularity

Modularity is the concept of splitting a program into single components with aim to reduce complexity of the program. These different components are known as modules. It can be said as the degree to which it can be separated. It utilizes the concepts of abstraction and encapsulation in the form of modules.

STOP TO CONSIDER

Grady Booch defined the term modularity as, “Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules”.

1.9.5 Polymorphism

The term polymorphism describes the concept a object can obtain many forms depending on the situation. In object-oriented programming also, polymorphism works in similar way. It allows a common external interface to perform two or more operations in a different way based on what they are operating. Let us take an example, suppose a plus sign (+) is used in such a way that when it is placed in between two numbers in an operation, it shows the sum of those two numbers as result. Again, if it is placed between two strings and the operation is performed it gives the concatenated string by combining those two input strings. So, in that case, the plus sign is capable of performing the operation in different ways based on the input variable. We can perform this type of operation using an operator or function in OOP.

CHECK YOUR PROGRESS - II

State True or False:

5. Abstraction is a process of hiding the background details from the user.

6. In binary relationship, connection is established among the objects of same class.

7. In unary relationship the connection is established between objects of two classes.

8. A single object of one class may build relationship with more than one object of other class is known as one-to-many relationship.

1.10 GENERALIZATION AND SPECIALIZATION

The concept of generalization and specialization is related to hierarchy of classes. It is the relationship between a parent class and one or more child classes. The classes are organized by the similarities and differences between the classes, their properties. Generalization is a process to create a class of higher-level hierarchy known as parent class by combining all the common properties of

the child classes. The child classes have access to those common properties. With this, child classes may have some other properties.

Specialization is nothing but just the reverse of generalization. Specialization is a process of creating new specialized child classes from the parent class. These specialized classes hold the distinguish properties related to the parent class. It is used when a set of properties (attributes or methods) can be applied to some of the objects of a class, then a new child class is created which holds those special properties. Here, the higher-level class is split into lower-level classes.

Let us try to understand the concept of generalization and specialization witha single example shown in **Figure 1.3**. If we follow the concept of generalization, then two classes Polygon and Circle have some properties. their common properties like color of the figure, border color, border line, methods to rotate and display the figure are combined together to form a new class *Figure2d*, this class consists of common properties of two-dimensional figures of polygons and circles. So, lower-level hierarchy classes are combined to form a higher-level class.

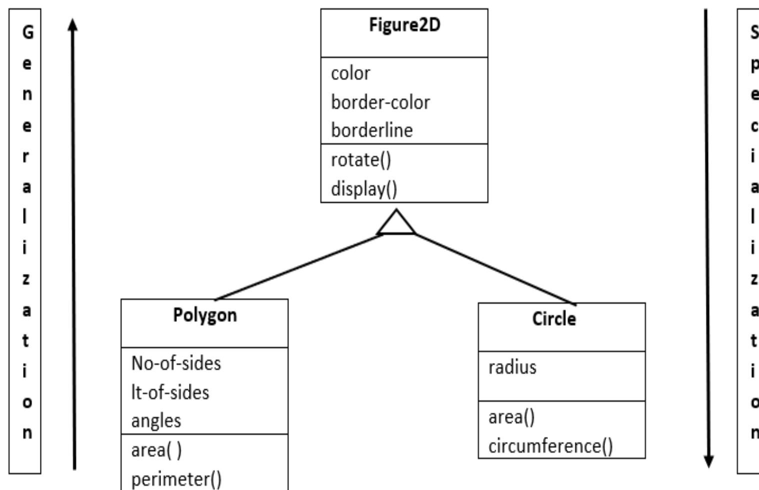


Figure 1.3: Example of generalization and specialization using 3 class hierarchy

If we follow the concept of specialization, the base class **Figure2D** is subdivided into two child classes viz**. Polygon and Circle**. Here, the derived classes can access the properties of the base class. And each derived class contain their distinguished specialized properties.

in the object-oriented design phase, we may need to perform generalization and specialization based on the context and demand of the system.

1.11 LINK AND ASSOCIATION

In the next chapter, we will study UML and how we can draw a class diagram, state diagram using UML etc. in OO Design. For this, we need to understand two important parts of object-oriented design to build the relationship among various objects and classes: links and associations.

A **link** is used to connect two or more objects. It is used to establish the relationship between the objects. The link between two objects can be physical or conceptual. Through a link, communication can be performed between two objects. A link is said to be as an instance of an association like we say, an object is an instance of class.

An **association** is used to build the relationship between classes. It is more descriptive in nature. That means, an association can be said a collection of group links sharing a common structure and common semantics. The links in an association are used to connect all the objects in a class. The syntax of link and association are shown here:

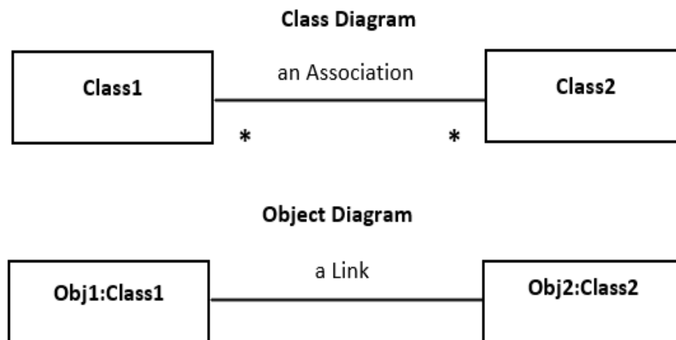


Figure 1.4: Association and Link using Class diagram and object diagram

Let us try to understand the use of links and associations with an example. Consider a model of a teacher, teaching subjects. Suppose, there is two class, Teacher and Subject. And there is an association between the classes as Teacher *teaches* Subject. There is a

possibility that a teacher can teach more than one subject, or one can be taught by one or more teachers. So, objects are connected via links. The class diagram and object diagram to show the link and association are presented in following **Figure 1.5**:

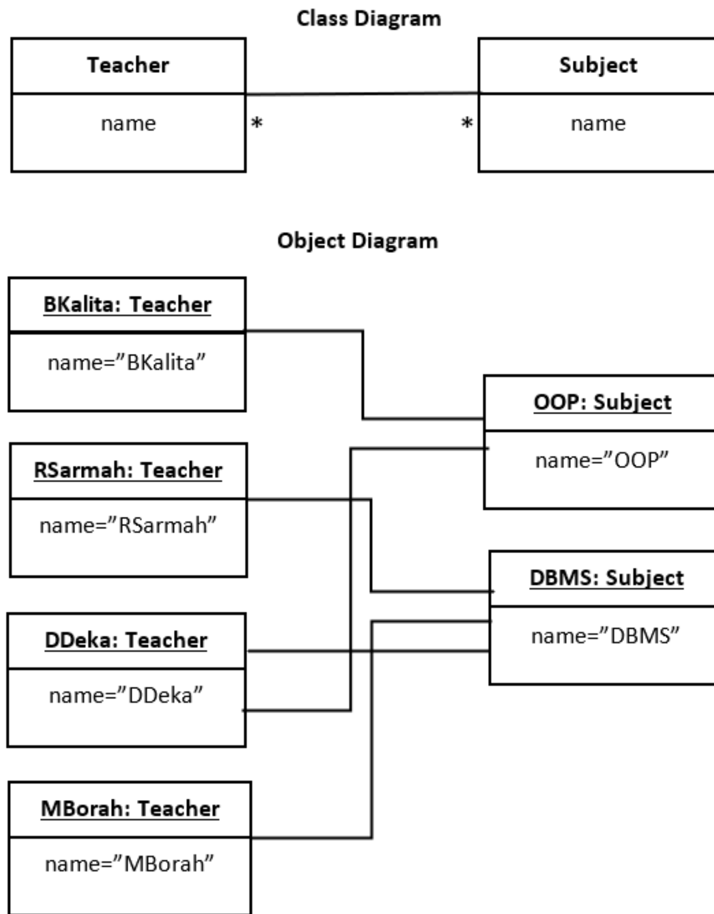


Figure 1.5: Class diagram and object diagram of Teacher and Subject class with many to many relationships to understand links and associations (many-to-many)

1.11.1 Degree of Association

The number of classes that are forming an association is known as degree of association. A degree of association can be of 1, 2, 3 or more.

Unary relationship: Here, the connection is established among the objects of same class.

Binary relationship: Here, the connection is established between objects of two classes. In this course, we will focus on only binary relationships.

N-ary relationship: Here, the connection is established among objects three or more than three classes.

1.11.2 Multiplicity

M. R. Blaha and J. R. Rumbaugh defined multiplicity in their book [2] as, "multiplicity specifies the number of instances of a class that may relate to a single instance of an associated class. Multiplicity constraints on the number of related objects". In a binary relationship, based on the concept of multiplicity, a relationship can be categorized in mainly three ways:

One-to-one association: Here, a single object of one class builds a relationship with a single object of another class.

One-to-many association: Here, a single object of one class may build relationship with more than one object of other class.

Many-to-many association: Here, multiple objects of the first class may associate with any object of the second class. Again, multiple of objects of the second class may build relationship with any object of the first class. As given example in **Figure 1.5**, a teacher may teach many subjects and a subject may be taught by many teachers. So, it is an example of many-to-many association. Here, the symbol '*' indicates there can be either zero or one or multiple associations can be held between the classes.

-

1.12 SUMMING UP

- Object-oriented approach focuses on finding a solution of a problem based on real-life examples.
- Object-oriented approach presents the problem domain in terms of smaller modules known as objects. Objects are consisting of data and procedures.
- Object-oriented software development methodology is composed of three main steps viz. requirement analysis, object-oriented design and implementation of the system.
- The main functions of object-oriented analysis are redefining the problem technically, collection of user and software requirements, identification of the classes and objects, assigning the services to the objects and identification of relations among the objects.
- Object-oriented design plans to build the solution domain based on the output of the object-oriented analysis phase.
- The functions of object-oriented design are identifying the classes based on previous steps, adding new classes on need basis, establishing relationships among them, identification of different constraints and designing the user interfaces.
- Decomposition is a process to divide the whole problem of high complexity into smaller subsystems of lower complexities.
- Object-oriented design can be performed using different models like class model, dynamic model and functional model. Inpresent days, designers are using interaction models instead of functional models.
- Object-oriented programming is to find the solution to the problem based on object-oriented design models.
- Data hiding is a property used in object-oriented programming to hide the internal data (data members, member functions) from unauthorised access.
- Objects are the basic entity of OOP which can be of either physical or conceptual existence.
- A class is a skeleton-like structure combing various data members and member functions. These can be accessed by the instances of the class i.e., objects.
- Abstraction is the process of hiding background detail from the users. Whereas encapsulation makes the system easier

for the users to handle by wrapping up the data and code into a single entity.

- Inheritance is a method through which one class can inherit the properties of another class. The class from which properties are inherited is known as base class and the class which inherits the properties is known as derived class.
- In OOD, relationships between multiple classes are established using associations where multiple objects are established using links. Links are the instances of association.

1.13 ANSWER TO CHECK YOUR PROGRESS

1. Class 2.object 3.Data hiding 4.Object-oriented

5. True 6.False 7. False 8. True

1.14 POSSIBLE QUESTIONS

1. Explain the brief history of evaluation of object-oriented programming.
2. Describe the following concepts of OOP with example:
   Abstraction, Encapsulation, Polymorphism
3. Define the term link and association. Explain their working in object-oriented design with an example.
4. What are the different types of models mainly used in object-oriented design?
5. Explain the working of object-oriented analysis with suitable diagram.
6. Explain the concepts of generalization and specialization in OOP.
7. What do you mean by degree of an association? What are the different types of associations between the classes in a relationship?
8. What do you mean by one-to-many and many-to-many association? Describe many-to-many association with an example with the help of class diagram and object diagram.

1.15 REFERENCES AND SUGGESTED READINGS

1. Booch, G.: Object-Oriented Analysis & Design with Applications. Pearson Education, 1994.
2. Blaha, M.R. and Rumbaugh, J.R.: Object-Oriented Modeling and Design with UML. Pearson, 2007.
3. Balagurusamy, E.: Object-Oriented Programming with C++. Tata McGraw-Hill, 2008 (fourth print).

UNIT 2 OBJECT MODELING TECHNIQUES (OMT) TOOLS

Unit Structure:

2.1 Introduction
2.2 Unit Objectives
2.3 Object-oriented modeling
2.4 UML
    2.4.1 Goals/ Characteristics of UML
2.5 Different Models of OO Design
2.6 Class Model
    2.6.1 Class Diagram
    2.6.2 Object Diagram
2.7 Dynamic Model
    2.7.1 State Diagram
2.8 Functional Model
    2.8.1 Data Flow Diagram (DFD)
2.9 Interaction Model
    2.9.1 Use Case diagram
2.10 Summing Up
2.11 Answers to Check Your Progress
2.12 Possible Questions
2.13 References and Suggested Readings

2.1 INTRODUCTION

In the previous unit, we have learned the purpose of object-oriented design, different object-oriented approaches and the important concepts related to object-oriented design. With this, we have got the basic idea of different diagrams based on different models of object-oriented design.

In this unit, we will learn how to draw various diagrams using different tools in object-oriented design. In object-oriented modeling, one of the most used methods to perform design is Unified Modeling Language. Different software is available to draw diagrams using UML tools. Object-oriented design is composed of different models. Designers used the models as per the requirement of the problem domain. Each model is represented by specific

diagram. Like class models are represented using a class diagram and object diagram. Here, we will learn these design procedures in detail with examples.

2.2 UNIT OBJECTIVES

The main objectives of this unit are:

- To understand the concept of object-oriented modeling
- To know the characteristics and use of UML
- To know various UML tools
- To have a clear concept of different models in object-oriented design
- To be able to draw the class diagram, object diagram, state diagram, DFD as per the problem.
- To have the knowledge of use case diagrams

2.3 OBJECT-ORIENTED MODELING

Object-oriented design (OOM) is a process of object-oriented design to design models for developing an application using different object-oriented concepts. A model can be said as an abstraction of a process for understanding its importance before implementing it. The models present how exactly the coding will be done. The execution of a program can be performed using any object-oriented programming language, based on the processes represented in a model.

The detailed design in object-oriented modeling consists of several phases. In the first phase, the models look more abstract since it focuses on the external detail of the system. Then, the model evolves at different phases, and more details are included like how the internal processes will be implemented, what functions are required and how the entire system will be built.

The main reason for designing a model using an object-oriented approach before starting the actual implementation or coding is that:

Easy to Understand: The diagrammatic representation of any system is easy to understand for the users. Understanding the complete coding procedure is not easy for normal users. Model diagrams help

user like clients, end-user, stake holders to give feedback to the developer team, how they want the exact system should work.

Abstraction: To have a clear idea of the system requirements and the input-output of the system.

2.4 UML

Unified Modeling Language (UML) is a widely used modeling language to design any software. It is used to identify, picturise, create and document the models of any software system. UML was initially created to use notational systems in software design. It is adopted as a standard software design tool by the Object Management Group (OMG). Further, UML was also approved as an ISO standard by International Organization for Standardization (ISO). With time, different versions of UML were released and it is widely used by software designers over the world.

UML is widely used to model software systems. The main artifact in designing a model using Unified Modeling Language is an object. Using UML tools, different models like class model, object model, state model can be designed. These models can be either static or dynamic. The object-oriented design is performed using UML. So, understanding the concepts of object-oriented design is a prerequisite to use UML. As per the requirements, the object-oriented design approach is transformed into any form of UML diagram.

2.4.1 Goals/ Characteristics of UML

The main characteristics of UML are:

- UML is a General-Purpose modelling language.
- It designs diagrams based on the output of the object-oriented analysis (OOA) approach.
- The workflow of a system to be developed is visualized by UML.
- Both static and dynamic models can be presented using UML tools.
- UML tools are easy to use, understand and helps to design models conveniently.

2.5 DIFFERENT MODELS OF OOD

In this section, we will learn the working of different models of object-oriented design, their respective diagrams in detail. With this, we will study how to draw these diagrams based on the requirement using Unified Modeling Language (UML) tools.

2.6 CLASS MODEL

A class model is a static model which represents the structure of the system in terms of classes. The class model contains the classes, their properties, the association among the classes. The properties of classes contain different attributes and operations. These are also referred with the term data member and member functions. Associations among the classes are performed using the concept of generalization and specialization.

During the study of class models, we need to have clear concepts on the various concepts on class like enumeration, multiplicity, scope and visibility of the classes.

2.6.1 Class Diagram

As the name suggests, in object-oriented design, the class diagrams are used to represent the classes identified in the object-oriented analysis phase.

Class diagram presents different graphic notations for modeling the classes, showing their relationships and describing the objects. It is a static diagram. The advantages of class diagrams are:

- It is easy to understand and concise.
- It worked well for practice and plan easily.
- It has mapped to popular OOP languages like C++, JAVA, etc.
- It helps programmers to implement the program easily based on the model design.
- It is useful for abstract modeling.

The syntax of defining a class on a class diagram is shown in Figure 2.1. It is represented using a rectangular box consisting of three blocks. The class name is written on the top of the box. In the second block, the properties of the class are written known as attributes.

In object-oriented programming, these attributes are mentioned as data members or variables. The data types and other properties of the data members can also be written in the class diagram. The methods known as member functions are defined in the last block of the rectangular box. Let us try to learn how to draw class diagrams for two different scenarios.

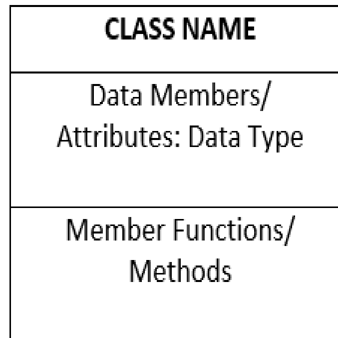


Figure 2.1: Syntax of representation of a class

Scenario 1: *In a company, employees are working in departments. A department can have multiple employees, but one employee can work in a single department. One department can control multiple projects. An employee is categorized as either 'Manager; or as 'Other Employee'. Different information as per the requirement can be stored in classes as attributes.*

So, the class diagram using the above scenario can be drawn as shown in Figure 2.2. The diagram consists of three main classes, where the class **Department** can be considered as the base class. The **Employee** class is associated with the **Department** class with '***works on***' relationship. **Department** class stores the name of the department as an attribute of string data type. The **Employee** class stores the employee ID and name of the employee as shown in the figure. Further, the concept of specialization is used in the **Employee** class to categorized it to **Manager** class or **Other_emp** class. **Department** class is associated with the

Project class as department undertakes the different projects. **Project** class stores the value of two data members, *project_name* and *duration*.

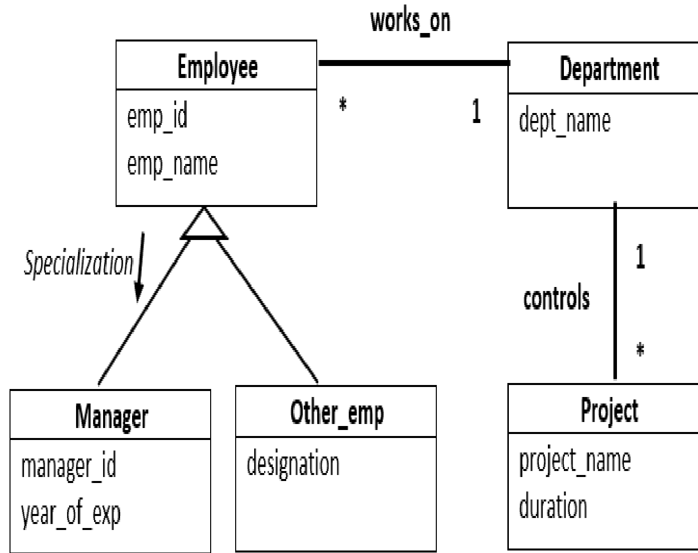


Figure 2.2: Class diagram of a company system as in scenario 1

Scenario 2:*Let us consider a basic system of online shopping. Customers can order multiple times, where order number is issued against each order. Customers can select payment option against each order. The payment system will generate bill number, mode of payment, amount and date of product. Payment class is further subdivided base on the payment options like internet banking, card payment and cash of delivery.*

Figure 2.3 shows the class diagram of basic online shopping system as described in scenario 2. After identification of classes, desirable attributes, their data types and methods of each class are also shown.

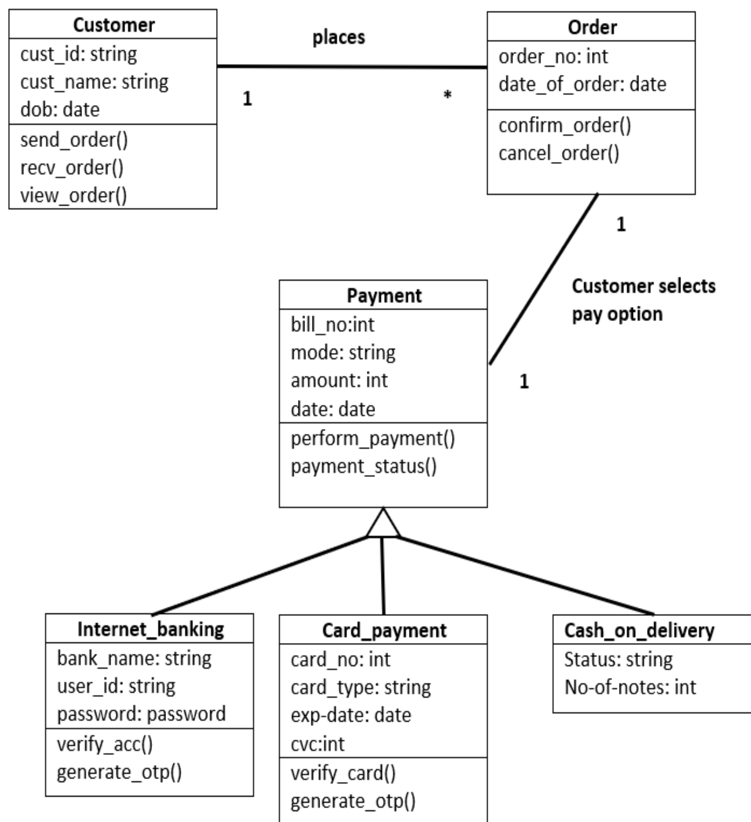


Figure 2.3: Class diagram of a basic online shopping system as in scenario 2

2.6.2 Object Diagram

As objects are the instances of a class, object diagrams are also instances of class diagram. It is derived from the class diagram. So, the object diagram is fully dependent on class diagrams.

Object diagrams are almost similar to the class diagrams. The key difference between the class diagram and the object diagram is that class diagram represents an abstract view of the systems showing relationships between the classes. Whereas, object diagram represents the instances of the classes at any particular point of time.

The syntax of declaring an object in an object diagram is shown in Figure 2.4. The objects in the object diagram are instantiated from respective classes. The object diagram shows the value assigned to

the attributes defined in the class. There can be more than one objects of a class.

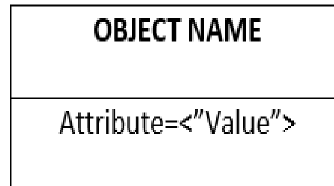


Figure 2.4: Syntax of an object in object diagram

The object diagram shows the links among different objects. The objects of the same classes can be linked in the diagram. The links among the objects of different classes are also presented in an object diagram. There can be more than one object diagram for a class diagram.

Figure 2.5 shows the object diagram of a company management system as described in scenario 1. This object diagram is instantiated from the class diagram presented in Figure 2.2.

A simple object diagram of the online shopping management system is shown in Figure 2.6. It is instantiated from Figure 2.3. Here, a single object is declared for each class; These objects are linked based on the associations of classes. The values assigned to the attributes are defined in the class diagram. Like this, we can draw any object diagrams from a class diagram.

CHECK YOUR PROGRESS - I

Fill in the blanks:

1. _____________ is widely used to model software systems.

2. A __________a static model which represents the structure of the system in terms of classes.

3. Class diagram is a ________________ diagram.

4. Object diagram is fully dependent on _____________

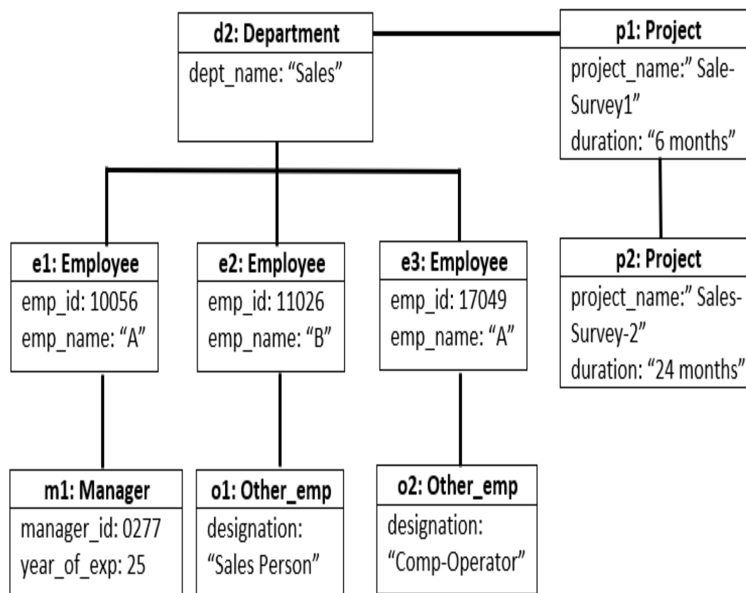


Figure 2.5: Object diagram of Company management system for class diagram shown in Figure 2.2

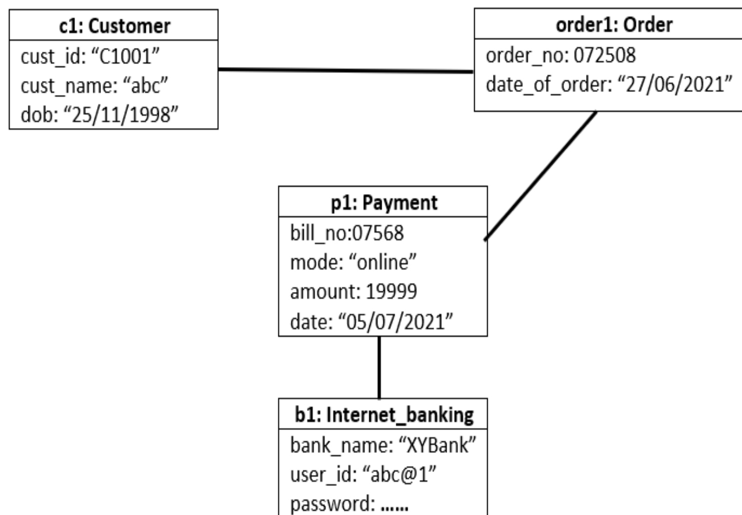


Figure 2.6: Object diagram of Online Shopping System for class diagram shown in Figure 2.3

2.7 DYNAMIC MODEL OR STATE MODEL

State models are known as dynamic model in OOD. State model represents the dynamic behaviour of a system. A state diagram is a collection of different states of the system which occur following a series of events.

A state can be said as an abstraction of the attribute values and links among the objects. Different values and links are grouped together to form the state model which shows the dynamic behaviour of the system. In UML, an event is an action occurring at a particular point of time. In a state diagram, the states are changed based on the events as per the need of the system. It changes the states during different phases of the system. Both the states and events in a dynamic model depend on the level of abstraction.

A transition is known as the change from one state to other. The change of state can be understood by the example of the physical changes of water. It is normally in liquid state. If we freeze it, for some time it changes to a solid state. If we start melting it, it returns to liquid state again. If the water is boiled for more times, it

transforms into gaseous state. Similar things can be happened with a software model, which changes its state as per the requirement.

2.7.1 State Diagram

Dynamic models or state models can be represented using the state diagram. The state diagram presents the transition or change in the state based on the events occurring in a system. The number of states in a state diagram isfinite. The main components of state diagrams are:

States: The states are represented using a rectangle symbol with rounded corner.

Initial state: the starting state of state diagram is known as initial state. It is represented using black circle.

Transition: The transition or change in state is shown by using an arrow. The arrow symbol is labelled by the event name.

Self transition: An arrow pointing to the same state is used for self-transition.

Final state: In a state diagram, the conclusive state is represented a filled circle with a circle notation.

The notations using UML of the state diagram are:

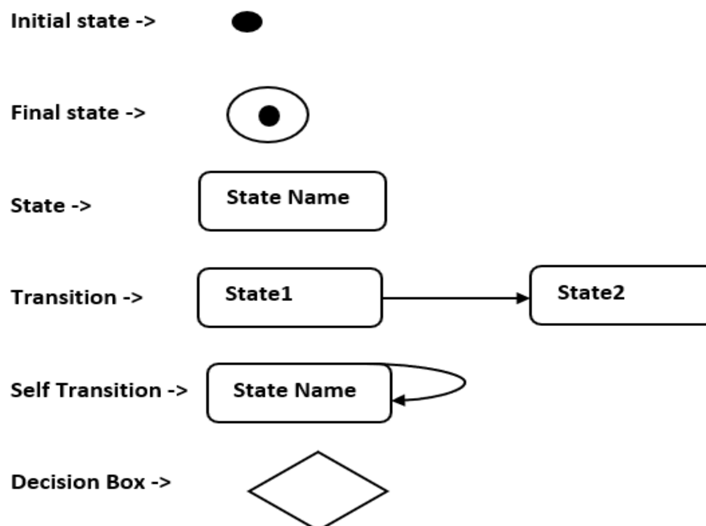


Figure 2.7 shows a partially completed state diagram of the household motor control system as derived in [2]. We try to understand the design of state diagram with this example. A motor primarily in *Off* state, when it is switched on, it changes its state to *Starting*. That is, it is preparing for run. Then, it to transforms to *Running* state and continue to run. From this, if we switched off the motor, it will go to *Off* state again. And if it is continued to run, it may go to *Too Hot* state and the overheating may harm the functions of the machine.

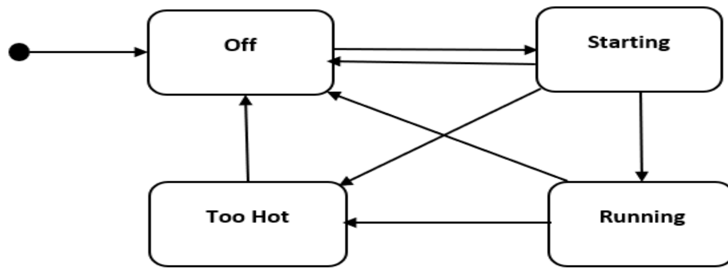
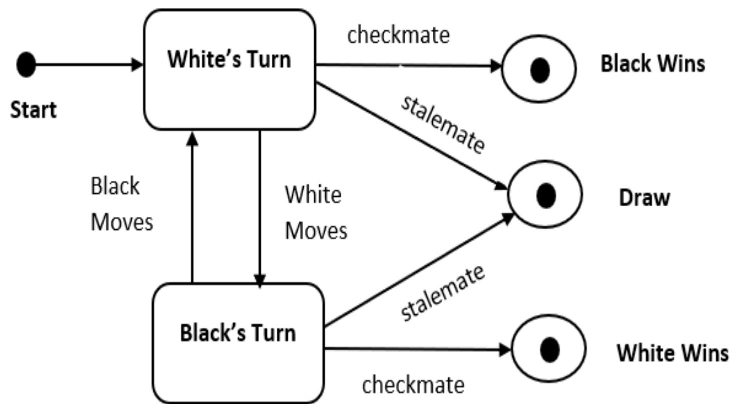


Figure 2.7: Partially completed State diagram of household motor control system



-

Figure 2.8: State diagram of a Chess Game

Let us consider another example as shown in Figure 2.8 which represents the state diagram for a chess game. This example is derived from [2]. This diagram consists of mainly two states namely *White's Turn* and *Black's Turn*. The states are changed

continuously in every move which starts from the *White's Turn* first. Depending on the checkmate and stalemate, the results of the match is decided, So, the final state of the game would be either *Black Wins* or *White Wins* or *Draw*. Thus, we can draw the state diagrams of a system.

2.8 FUNCTIONAL MODEL

In object-oriented analysis and design, the functional model specifies the overview of the entire system. It provides the overall summary of the functions that the system is going to perform. This model presents the system based on main three tasks: inputs of the system, processing and outputs of the system. To represent the functional model of a system, Data Flow diagram (DFD) is used. DFD helps to provide functions of the internal process of the system.

2.8.1 Data Flow Diagram (DFD)

Data Flow Diagram (DFD) represents the functions of the system in form of the inputs to be provided to the system, the processing parts of the system based on the inputs and the desired output of the system along with the detail of internal data stores in the system.DFD can be drawn in many phases or levels. These levels are namedas Level-0 (Context Diagram), Level-1, etc. The basic components of DFD are:

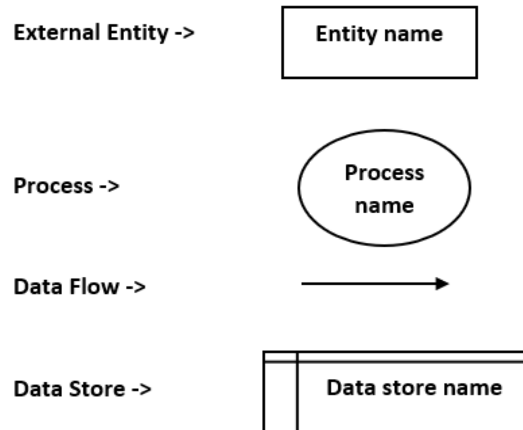
External Entity: These entities are named based on real-life objects. These entity takes the information and submit to the processes. It can ask any query to the process and gets information or response from the process.

Processes: In DFD, a process is the activity that executes the data flow of a system. DFD shows how the data flows through a process in a system. A process can have multiple sub-process under it.

Data Flows: Data flow depicts the movement of information among external entities, processes and the place where data is stored through arrows.

Data Store: Data store represents the place where data is stored. It does not perform any operations, simply stores the data. Processes can insert, update, delete or retrieve data to Data store.

The notations used in DFD are:



Let us consider an example of DFD of basic *Store Management System* as shown in Figure 2.9. This system consists of only two entities, *Customer* and *Seller*. Here, the seller or shopkeeper would enter product details which would be stored in the database.*Customer* has to register into the system. After successful registration, a customer gets Customer ID. A customer can order products, if the product is available, a customer will get order confirmation message. This Process is further subdivided into two sub processes namely *Registration* and *Selling Process* as shown in Figure 2.10.

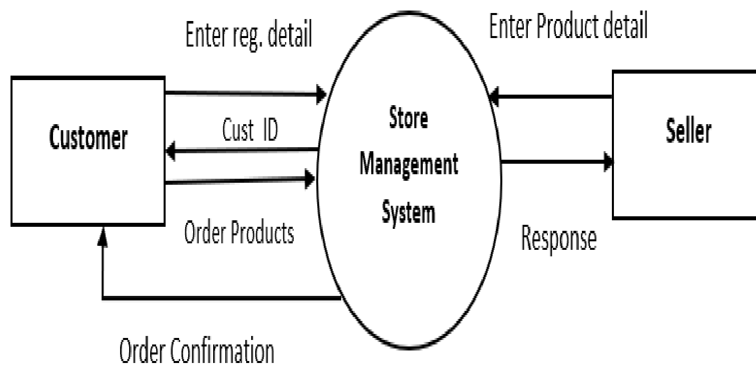


Figure 2.9: Level-0 DFD for basic store management system

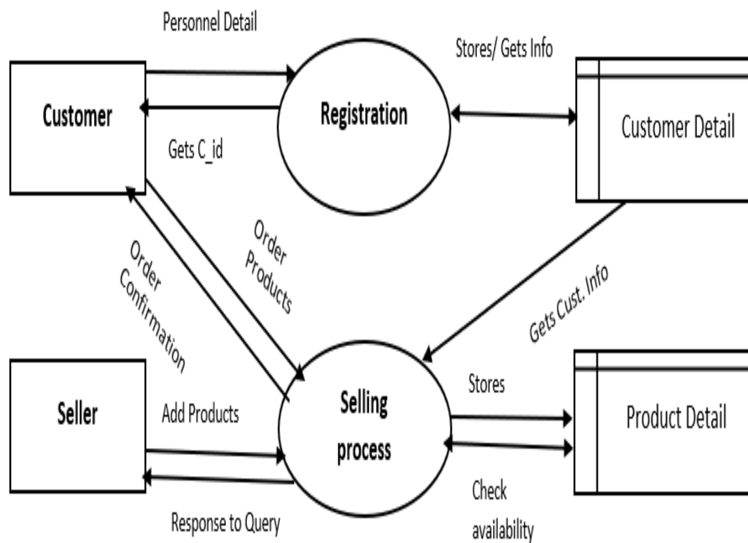


Figure 2.10: Level-1 DFD for basic store management system

CHECK YOUR PROGRESS - II

State True or False:

5. A transition is known as the change from one state to other.

6. The states and events in a dynamic model do not depend on the level of abstraction.

7. Dynamic models cannot be represented using the state diagram.

8. DFD is used to represent the functional model of a system.

2.9 INTERACTION MODEL

The Interaction model describes the interactions inside a system. The class model presents the objects present in the system; the state model presents the life-cycle of declared objects in terms of states whereas as interaction model presents how the interaction takes

place among these objects. Interaction models can be represented using different diagrams like use case diagram, sequence diagram etc. Here, we will study the designing of usecase diagram with an example.

2.9.1 Use Case Diagram

A use case is a representation of how a human being (known as actor) uses a process to reach his goal. Use case is a part of functionality that a system is able to provide by interacting with the actors. An actor is the person who act as a direct external user in a system. A use case is composed of one or more actors. Use case is a description of the role of each actor to accomplish particular goal using various processes. The use case of a system can be represented by a use case diagram.

Let us take an example of the operation of a vending machine as presented in [2]. The use case of the scenario is buying a beverage by a customer. The customer delivers the beverage after successful selection and payment made by customer. Figure 2.11 shows the use case diagram for this vending machine. It represents three actors namely Customer, Repair Technician and Store Clerk. Inside the rectangular box, four use cases are shown in which actors participate. The use cases are *Buy beverage, Performed Maintenance, Make repairs* and *Load Items*. Repair Technician participates in two use cases whereas other actors participate in one use case.

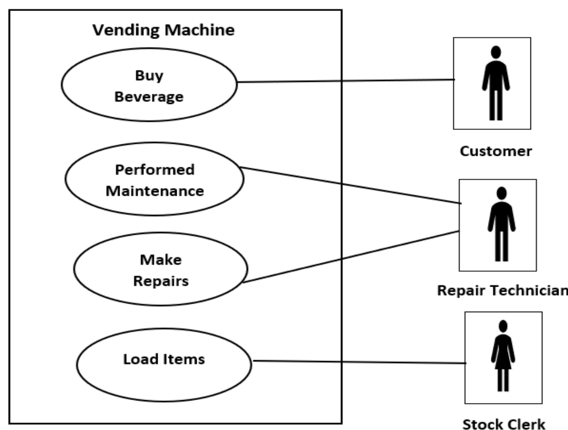


Figure 2.11: Usecase diagram for a vending machine

In this unit, we have learnt different object-oriented modeling techniques to design a system. Each model type is represented by a particular diagram. These diagrams help us easy to understand the workflow and behaviour of the system. Programmers perform the coding to execute the system by following these designed models. So, this is an important phase in developing a system using object-oriented concepts.

2.10 SUMMING UP

- Object-oriented modeling (OOM) is a process of designing a model to develop a system using different object-oriented concepts.
- A model is an abstraction of a process to understanding it before its implementation. The system is built following the model.
- Different models are there to design a system like class model, state or dynamic model, functional model etc.
- UML is mostly used general-purpose modeling language to identify, picturise, creation and documentation the models of any software system.
- The Class model is a static model to represent the structure of the whole system in terms of classes. The classes contain their different characteristics and methods.
- A Class model is represented using class and object diagrams.
- A Class diagram shows the classes, their relationships, the attributes and methods present in each class.
- An object diagram is an instantiation of the class diagram. A class can have more objects. Object diagram shows the links among different objects.
- Each object in an object diagram contains particular value of all attributes or data members of its class.
- The dynamic model or state model shows the dynamic behaviour of a system in terms of its change of state following a series of events.
- A state is nothing but an abstraction of attributes and links among the objects. Transition is change of a state to another in a system.

- A state model is represented by a state diagram. The main components of state diagram are initial state, final state, other states and transitions.
- Functional model specifies the overview of the whole system inputs, its processing and outputs to the system.
- Functional models are represented using Data Flow Diagram (DFD) in a system.

2.11 ANSWERS TO CHECK YOUR PROGRESS

1.UML	2. Class model	3.static	4. Class diagram
5. True	6. False	7.False	8.True

2.12 POSSIBLE QUESTIONS

1. Explain the concepts and importance of object-oriented modeling.
2. Describe the following models of OOD with example:
   Class model, Dynamic model, Functional model
3. Explain the important characteristics of UML.
4. Explain the working of class diagram and object diagram with a suitable example.
5. Why the dynamic models are need in object-oriented modeling? How it is different from a class model.
6. How can we draw a state diagram of a system? Explain with a suitable example.
7. What are the basic components of an DFD? Draw a data flow diagram of level-0 DFD for online shopping management system.
8. Prepare a class diagram of your family tree with name and age as attribute of each class. Draw an object diagram of this class diagram.
9. Draw a state diagram of complete telephone call procedure with its different states.

10. Explain the working of an interaction model.
11. Write the importance of suing a usecase diagram. Draw a usecase diagram of simple vending machine with explanation of its working.

2.13 REFERENCES AND SUGGESTED READINGS

1. Booch, G.: Object-Oriented Analysis & Design with Applications. Pearson Education, 1994.
2. Blaha, M.R. and Rumbaugh, J.R.: Object-Oriented Modeling and Design with UML. Pearson, 2007.
3. Balagurusamy, E.: Object-Oriented Programming with C++. Tata McGraw-Hill, 2008 (fourth print).

UNIT 3: PHASES OF OBJECT-ORIENTED DEVELOPMENT

Unit Structure:

3.1 Introduction
3.2 Unit Objectives
3.3 Object Oriented modelling
3.4 Object Oriented Methodologies
3.5 Object Oriented Process
 3.5.1 System Analysis
 3.5.2 System Design
 3.5.3 Object Design
 3.5.4 Implementation
3.6 System Analysis
 3.6.1 Object Model
 3.6.2 Dynamic Modelling
 3.6.3 Functional Modelling
3.7 System Design
3.8 Object Design
3.9 Summing Up
3.10 Answers to Check Your Progress
3.11 Possible Questions
3.12 References and Suggested Readings

3.1 INTRODUCTION

In software development, most of the methods used are based on functional or data-driven approach. Object-oriented approach is different from these approaches in many ways. In object-oriented methods, data and functions are integrated in to one group. It develops the software from some self-contained modules known as objects. These objects can be easily modified, replaced and reused. Object-oriented approach works based on the concepts of real-world systems for which it becomes a popular methodology for programmers. In object-oriented methodology, software can be treated as a collection of discrete objects in which data and the operations grouped together for desired outputs.

3.2 UNIT OBJECTIVES

The main objective of this chapter is to

- introduce the concept of object-oriented modelling.
- give an overview of object-oriented methodologies.
- introduces object-oriented methodology and
- discuss the phases of object-oriented design in detail.

3.3 OBJECT ORIENTED MODELLING

A model in object-oriented approach can be said as abstract view of the problem with an objective to understand its purpose before implementation. Since, working of a real system is very complex, so we need to simplify it. A model hides the exact working procedure and shows the important characteristics of the problem.

Through a model the problem is conceptualized and present distinctly. We can say that modelling helps us to deal with complexities of the system and makes the whole system easily understandable.

In object-oriented development, a model is an iterative process. In designing the model as the working of the system progresses, more detail is added to the model. Models are designed in different levels. In every level, a model can be subdivided into some smaller models for better understanding of their purposes.

3.4 OBJECT ORIENTED METHODOLOGIES

Object-oriented methodology is completely based on the concepts of classes and objects. These concepts are introduced based on the real-life examples. A class can be skeleton of a working procedure. An object is an instance of a class. We have discussed these in previous units.

An object can be created, modified, categorized, merged, described, removed or reused. In object-oriented development, software components can be modified and reused easily. It results

in high quality, higher productivity and lower maintenance cost of the software.

Object-oriented development uses different object-oriented techniques to analyse, design, implement and maintenance of the system. The developer determines what are the objects, their characteristics, responsibilities and relationships with other objects.

In object-oriented design phase, the whole architecture of the system is described. The objects of different class and their relationships are shown. In implementation phase, actual coding is performed by the programmer based on the design report. The program is connected with a database to store data so that the whole system becomes operational.

3.5 OBJECT ORIENTED PROCESS

As we studied earlier, object-oriented development is built with its basic element as object. For this, first the detail analysis is performed to collect all the requirements. Object-oriented development can be said as the traditional approach of system designing as it also follows a sequential process to design the system. The fundamental steps for designing a system using object-oriented process are:

- System Analysis
- System Design
- Object Design
- Implementation

3.5.1 System Analysis

Like any other design system, the first phase of object-oriented development is system analysis. Here, the interactions between the developer and the clients or the user take place. Developer collects the requirements and analyse them to have a clear idea of the system.

Based on the information gathered in the analysis phase, a document is prepared in which detail requirement of the system is specified. This documentation describes the different models and

their functioning. In this stage, the implementation details are not examined. We will discuss it in detail in section3.6.

3.5.2 System Design

The second step of the object-oriented process is system design. In this step, the entiresystem is designed based on requirement analysis and specification documents of the previous step. In this phase the system is divided into smaller parts known as sub-systems. The sub-systems interact amongthemselves to solve the problems. We will discuss it in detail in section3.7.

3.5.3 Object Design

In this phase, the details of the system analysis and system design are implemented. The Objects identified in the system design phase are designed in this phase. Here the implementation of these objects is decided in the form of data structures and therelationships between the objects are designed. For example, we can define a classstudent and then we can create several objects of this type.

In this phase, of the designer decides about the classes in the system based on the system requirements. The designer also decides, whether the classes need to be created freshly or inherited from existing classes. We will discuss it in detail in section 3.8.

3.5.4 Implementation

During this phase, the class objects and the interrelationships of these classes are translated intoactual code by using an object-oriented programming language. The required databases are created and the complete system is transformed into operational one.

3.6 SYSTEM ANALYSIS

In the system analysis or object-oriented analysis phase of software development, the system requirements are determined, the classes and the relationships among classes are identified.

There are three different analysis techniques used in conjunction with each other in object-oriented analysis. They are object modelling, dynamic modelling and functional modelling.

3.6.1 Object Model

The object model describes the structure of the objects. It describes the identity of the objects, relationships with the other objects, attributes, and the operations. The object model shows the primary view about how real-world problems are divided into objects and the objects interacts with each other. The object model provides the basic framework on which other models are positioned.

The object model is represented graphically by an object diagram. The object diagram contains the classes interconnected by lines. Each class represents a set of individual objects. The association lines establish relationships among classes. Each association line represents a set of links from the object of one class to the object of another class.

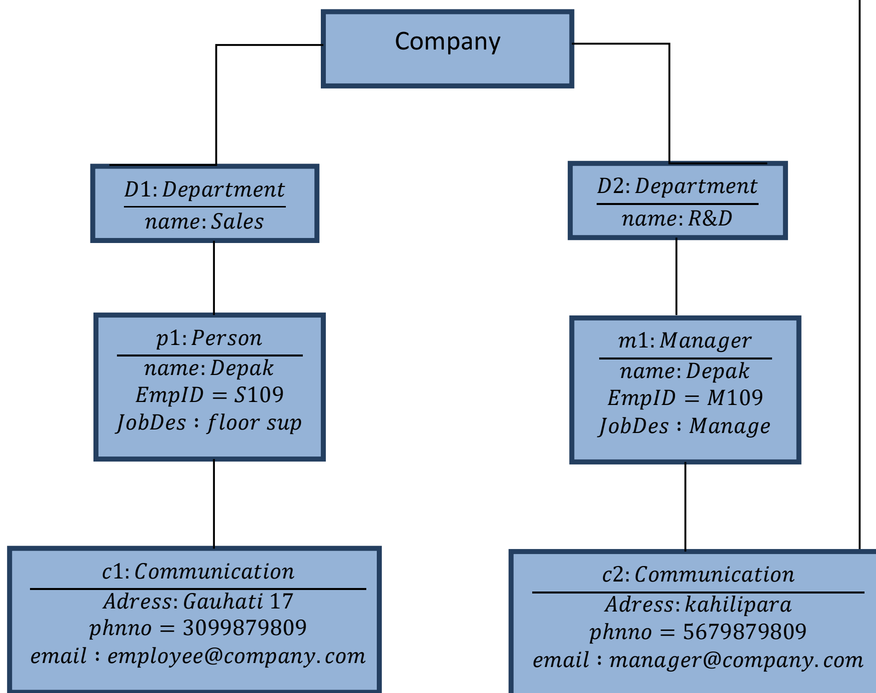


Fig3.1: Object Diagram

A class describes a collection of similar objects. It is a template where certain basic characteristics of a set of objects are defined. A class defines the basic attributes and the operations of the objects of that type. For creation of objects, instances of the class are to be created as per the requirement of the system.

Classes are built on the basis of abstraction, where a set of similar objects with common characteristics are listed. The characteristics concern to the system under observation are taken and the class definition is made.

3.6.2 Dynamic Modelling

Dynamic model describes aboutthe features that changes with the time. It is used to specify and implement control features of the system. It describes states, transitions, events and actions. The dynamic model includes event trace diagrams to describea scenario. An event is a task from one object to another, which occurs at a particular time. An event is a one-way transmission of information from one object to another. A scenario is a sequence of events thatoccurs during one particular execution. Each of the basic execution of the system is represented as a scenario. In the fig 3.2 A scenario of ATM cash withdrawal is explained.

The dynamic model is represented graphically by state diagrams. A state corresponds to the interval between two events received by an object and describes the "value" of the object for that time period. A state is an abstraction of an object's attribute values and links, where sets of values are grouped together into a state according to properties that affect the general behaviour of the object. Each state diagram shows the state and event sequences permitted in a system for one object class. The state diagram should follow the object-oriented development notation. The outcomes of dynamic modelling are event-trace diagram and state diagram.

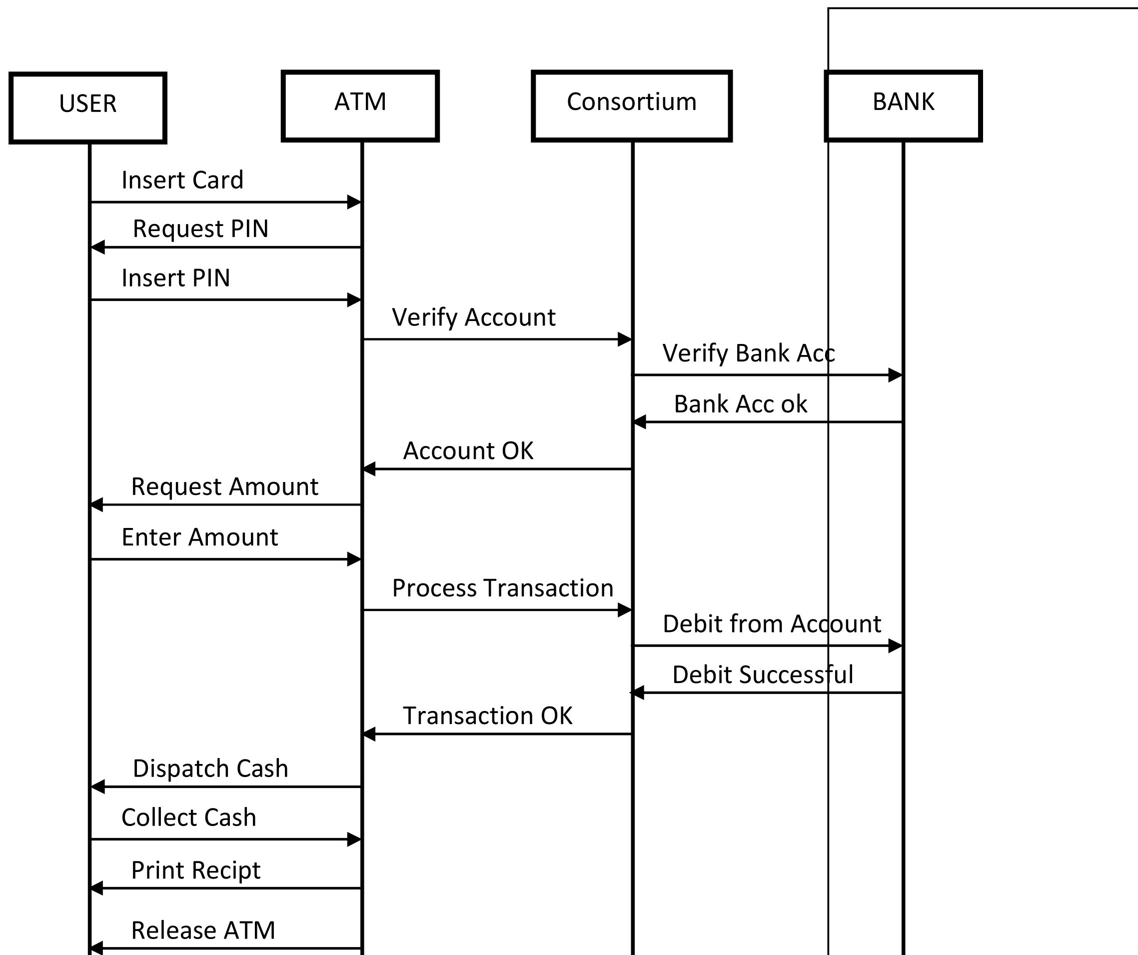


Fig3.2: Event Trace Diagram

3.6.3 Functional Modelling

The functional model describes computations and specifies those aspects of thesystem concerned with transformations of values. Functions, mappings, constraints, and functional dependencies are described in this model. The functional model describes what the system does, without any detail about how and when it is done.

The functional model is represented graphically with data flow diagrams, which show the flow of values from external inputs, through operations and internal data storage, to external outputs. Data flow diagrams show the dependencies between values and computation of results from the inputswith the helpof functions. Functions are invoked as actions in the dynamic model and are

shown as operations on objects in the object model. Data flow diagrams are discussed in detail in section 2.8.1.

3.7 SYSTEM DESIGN

Systems design is the process of defining the architecture, components, modules, interfaces, and data for a system. It is a process where system developers design the overall structure of the system. System design emphasises on how to solve the problem. The system design is divided into two parts – logical design and physical design.

The logical design of the system is an abstract representation of the inputs, outputs and data flows of the system. This is often conducted by modelling; modelling involves a diagrammatical representation of an actual system. System design includes Data flow diagram, Class Diagram, Entity-relationship diagrams etc.

The physical design relates to the actual input and output process of the system. Here how data is provided to the system, how it is authenticated, how it is processed, and how it is displayed are defined.

In object-oriented design methodology, system design is one of the phases of the software development life cycle. During this phase, developers decide the overall structure and style of the system. During system design, the following design decisions are to be handled.

a) Estimation of system performance
b) Make a reuse plan.
c) Organize the system into subsystems
d) Identify concurrency inherent in the problem
e) Allocate subsystems to hardware.
f) Estimating Hardware Resource Requirements
g) Manage data stores.
h) Handle global resources.
i) Choose a global control strategy
j) Handle boundary conditions
k) Set trade off priorities
l) Select an architectural style.

a) Estimating System Performance

A rough performance estimatesis calculated by the system. The purpose of this step is to check feasibility of the system. The execution of the system should be fast and free from common errors. In this phase, number of transactions to be processed by the system, response time needed, storage requirements etc are estimated.

b) Making a Reuse Plan

Reuse is one of the main advantages of object-orientedmethodology. There are two different types of reuses, using existing things or creating newreusable things. It is easier to reuse existing things than to design new things. Reusable things can be models, libraries, frameworks and patterns. The logic in a model can applied to multiple problems.

c) Organizing a system into Subsystems

A subsystem can be defined as a group of classes, associations, operations, events and constraints that are interrelated and can be used to solve a large problem. For example, file system is a subsystem of operating system. A system may be divided into smaller subsystems and each subsystem may further be divided in to smaller subsystems without affecting to the output result.

d) Identifying Concurrency

Concurrency is an important issue that needs to be addressed in design process as it may affect the design of classes and their interfaces. Concurrency is very important for improving the efficiency of the system. One important goal of the system design is to identify the objects that must work concurrently and synchronize the objects that have mutualactivity.

e) Allocation of Subsystems

We must allocate hardware to the each of the subsystems, either a general-purpose processor or specialized functional unit. The system designers must ensure the following:

- Estimate performanceand allocate the resources needed by system.

- Choose hardware and software to implement the subsystems
- Allocate software processors such that it satisfies performance and minimize inter process communication
- Determine the connectivity of the physical units related to the subsystems.
- Define the connection between nodes and communication protocols to be used.
- Consider the needs for redundant processing and provide infrastructures.
- Identify the interfacesapplied in deployment.

UML deployment diagram can be used to present the above steps. A deployment diagram shows how the systems will be physically distributed on the hardware.

f) Estimating Hardware Resource Requirements

To increase the performance of system, multiple processors or hardware may be used. The number of processors required depends on the computation requirement and the speed of the machine. The system designer estimates the requirement of processing power for asteady computing and high performance.

While implementing the system a decision needs to be made regarding which subsystems will be implemented on which set of hardware and software. In implementing the subsystems in hardware following are needs to be keeps in mind

- Cost: As the technology advances, the cost of hardware has come down still in system design the designer should keep an eye on the cost of implementation and hardware requirement.
- Performance: In this phase system designer need to ensure high performance system.

g) Management of Data Storage

System designer needs to decide on the data storage for implementation of the data structures, files and databases used in the system. In identifying the data storage complexity of the data, the size of the data, the access method, access time and portability needs to be kept in mind. Having considered these issues, the

designer must take a decision about whether data can be stored in flat files or in relational databases or in object databases.

h) Handling Global Resources

The system designer must identify the global resources and determine a mechanism for control. There are several kinds of global resources:

- Physical system: processors, tape drives and communication channels.
- Input, output: keyboard, mouse, display screen etc.
- Logical Ids: object IDs, filenames, and class names.
- Access to shared data: Databases

i) Choosing a Software Control Strategy

In designing the system, it is best to choose a single control style for the entire system. There are two kinds of control flows in a software system: External control and Internal control. External control concerns the flow of externally visible events among the objects in the system and the internal control refers to the flow of control within a process. It exists only in theimplementation and therefore is neither inherently concurrent nor sequential.

There are three kinds of external events: procedural-driven sequential, event-driven sequential and concurrent. In a procedure-driven system, the control lies within the program code. Procedures request external input and then wait for it, when input arrives, control resumes within the procedure that made the call.In event-driven, the developers attach application procedures to eventsand the dispatcher calls the procedures when the corresponding events occur. In concurrent control, the developers need to ensureconcurrent execution of the system.

j) Handling boundary Conditions

In designing of the system, the system designer must consider boundary conditions also needto address the issues like initialization, termination and failure.

- Initialization: It refers to initialization of constant data, parameters, global variables, tasks, guardian objects and classes as per their hierarchy.
- Termination: Termination is required for all the subsystems to release the reserved resources. In case of

concurrent system, a task must intimate other tasks about its termination.

- Failure: Itis the unplanned termination of the system, which can occur due to various reasons such as system fault,wrong user input, due to exhaustion of system resources, external breakdown or bugs in system. A good design must not affect remaining environment in case of any failure and there must be a mechanism for recording details of system activities and error logs.

k) Setting trade-off Priorities

The system designer must set priorities for each of the subsystems that must be used as a guide trade-off for the rest of the design. For example, system can be made faster by adding extra memory. Design trade-offs involve not only the hardware but also the process of developing the system. System designer must define the importance of the various criteria as a guide to design trade-offs. Design trade-offs affect entire character of the system. Priorities are generally specified as a statement of design philosophy.

l) Architectural Styles

Software architecture, according to ANSI/IEEE Standard 1471-2000, is defined as the "fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution."

The architecture of a system describes its gross structure. This structure illuminates the top-level design decisions, including things such as how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts. Additionally, an architectural description includes sufficient information to allow high-level analysis and critical appraisal.

3.8 OBJECT DESIGN

Object design is the third phase in the Object-Oriented Development. In object design,the designeradds more details and refinement to the system. In object design, the designer implements the objects discovered in analysis phase.

The operations identified in analysis phase are expressed as algorithms and internal operations. The classes, attributes and associations found in analysis phase are implemented with specific data structures. If required, new object, classes can be added in this phase. The following steps are performed in the object design phase:

- Combine all the three models
- Design algorithms for operations
- Optimize design
- Implementation of control
- Maximize inheritance
- Design associations
- Determine object representation
- Combine classes and associations into modules

a) Combine all The Three Models

The output of analysis phase are object model, dynamic model and functional model. In this phase the object designer converts actions and activities of the dynamic model and processes in the functional model into operations of the classes in the object model.

State diagrams constructed in the dynamic model provides the detail working of the object. Transition in the state diagram represents the change of states. In this phase the transitions are mapped into operations of the object. The action performed in a transition depends on both the event and the state of the object. So, the algorithm implemented on objects depends on the state of the object and the event.

An event generated by an object may represent an operation on another object. Events often occur in pairs; the first

event triggers an action and the second event returns the result or indicates the completion of the action.

Actions initiated by a transition in a state diagram may expand into a full-fledged data flow diagram (DFD) in the functional model. The collection of processes within the DFD represents the body of an operation. The object designer must convert the DFD into a linear sequence of steps in an algorithm.

b) Design Algorithms for Operations

Algorithms are designed for each operation in the DFD. The DFD depicts what are the operations and the algorithm explains how it is done. The algorithm designer must follow the following steps in designing algorithms for operations:

- Selection of an appropriate algorithm
- Selection of an appropriate data structure
- Adding new classes and operations as necessary
- Assigning appropriate responsibility to classes

In choosing an appropriate algorithm the following criteria need to be addressed:

Computational complexity: Determine the complexity of the algorithm and choose an algorithm, which takes lesser computation time.

Ease of implementation: We need to select a simpler algorithm for implementation.

Flexibility: The algorithm designed should be flexible and there should be provision for future extension. If an algorithm is highly optimized then it is very difficult to understand and modify.

Choose an appropriate data structure: Every algorithm uses some data structures, to make an algorithm efficient, an appropriate and cost-effective data structure should be chosen.

Add new classes and operations as necessary: To hold intermediate results, we may need to add new classes. A complex operation can be decomposed into many low-level operations.

Assign appropriate responsibility to classes: Most of the operations have an obvious target but there are some operations, which can be placed at many places. This problem gets more

complicated if the operation involves many objects. So, we need to take care of this issue.

c) Optimize The Design

The analysis model defines the logical information about the system, while the design model adds details about the information accesses. The system designer must make a balance between efficiency and clarity of the system.

The system designer can optimize the design by:

- Adding redundant associations to minimize access cost and maximize convenience
- Rearranging the computation for greater efficiency
- Saving the derived attributes to avoid re-computation of complicated expressions

Adding Redundant Associations for efficient access: During design, the structure of the object model is evaluated for an implementation. However, there may be situation where the associations found useful in analysis phase may not be useful in design phase. On the other hand, there may be some association which is defined for some other objects and may need to be redefined again for some other objects to simplify implementation.

Rearranging Execution Order for Efficiency: After adjustment of the structure of object model for optimization the next thing is the optimization of the algorithm. For optimization of algorithms, we need to find out the unnecessary codes and codes that does not lead to solutions and remove those codes.

Saving Derived Attributes to Avoid Re-computation: There may exist some data which can be derived from other data. This kind of data may be termed as redundant data. Computation time can be saved if we can eliminate the redundant calculations by using previously calculated data.

d) Implementation of Control

As part of the system design, the designer must implement state diagram designed in the dynamic model. There are three basic approaches to implement it:

- To use location within the program to hold state.
- Direct implementation of state machine mechanism
- Using concurrent tasks

The state diagram can be converted to code as follows:

1. Identify the main path in the diagram that leads to execution of events. Identify the names of states in the path.
2. Identify alternate paths, which branch off the main path. This becomes the conditional statements in the program.
3. To identify the loops, find out backward paths that branch off the main path. Multiple backward paths become nested loops in the programs.
4. The states and conditions correspond to exceptional conditions need to be handled through exception handling or by error subroutines.

Direct implementation of state machine mechanism: The direct approach to implement control is to have a state machine engine class that keeps track of execution of states and actions. Each object instance maintains its own independent variables. The basic flow of control can be traced by creating stubs of the action routines. A stub contains the minimal piece of information regarding functions or subroutines.

e) Maximize Inheritance

The specialization and generalization relationships are both reciprocal and hierarchical. Specialization is just the other side of the generalization. The definition of classes and operations can be adjusted to make the inheritance as large as possible. It can be done in the following ways.

- Rearranging and adjustment of classes and operations to increase inheritance.
- Abstracting common behaviour out of the group of classes.
- Use of delegation to share behaviour when inheritance is semantically invalid.

Rearranging and adjustment of classes and operations to increase inheritance

Inheritance leads to reusability of already existing functionality. Inheriting more functions increases the level of reusability. However, larger inheritance requires, functions definitions to be

redefined to serve multiple classes. The following adjustments can be used to increase the level of inheritance.

- Some operations may have fewer arguments than others. The missing arguments can be supplied in the function definition and they may be ignored where they are not necessary.
- Similar attributes in different classes may have different names. These attributes may be moved to a common ancestor class.
- Some operations may have fewer arguments because they are special cases of more general arguments. These special operations can be implemented by calling the general operations with appropriate parameters
- An operation may be defined on several different classes in a group but may not be required in other classes. In that case the operation can be defined as a common ancestor class and can be declared as no-operation on the classes where they are not required.

f) Design Associations

During the object design phase, all the associations in the object model are implemented. To make a decision for implementation, all the associations need to be analysed as they are used.

Analysing Association Traversal: Associations may be traversed either unidirectional or bi-directional. The unidirectional associations traversed in forward direction only and they are easy to implement. Bi-directional associations allow reverse and forward traversal, so bi-directional associations are always preferred over unidirectional associations so that we can add new behaviour or expand or modify the application rapidly.

One-way Associations: A unidirectional association can be implemented as a pointer.

Two-way Associations: There are three approaches to implement bi-directional associations as discussed below:

- Implement as an attribute in one direction only and perform a search when a backward traversal is required.
- Implement as attributes in both the directions. This results in fast access but if either attribute is updated then the other attribute must also be updated to keep the link consistent.

- Implement as a distinct association object, independent of either class. An association object is a set of pairs of associated objects stored in a single variable size object. For efficiency, an association object can be implemented using two dictionary objects, one for the forward direction and one for the backward direction.

Link Attributes: If an association has link attributes, then its implementation depends on the multiplicity. If the association is one-to-one, the link attributes can be stored as attribute of either object. If the association is many-to-one, the link attributes can be stored as attributes of the many object. If the association is many to many, the link attributes cannot be associated with either of the objects.

g) Determine Object Representation

The object designer has to choose when to use primitive types in representing the objects or when to combine the groups of objects. A class can be defined in terms of other classes but ultimately all data members have to be defined in terms of built-in data types supported by a programming language.

h) Packaging of Classes and Associations into Modules

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. Modules serve as the physical containers in which we declare the classes and objects. A module can be edited, compiled or imported separately. Different object-oriented programming languages support the packing in different ways. For example, Java supports in the form of package, C++ in the form of header files etc.

Following purposes can be solved by modularity.

- A module typically groups a set of class definitions and objects to implement some service or abstraction.
- A module is frequently a unit of division of responsibility within a programming team. A module provides an independent naming environment that is separate from other modules within the program.
- A Modules support team engineering by providing isolated name spaces.

Packaging involves the following three issues:

- Information Hiding

- Coherence of Entities
- Constructing Physical Modules

Information Hiding: During analysis phase we are not concerned with information hiding. So, visibilities of class members are not specified during analysis phase. It is done during object design phase. In a class, data members and internal operations should be hidden, so, they should be specified as private. External operations form the interface so they should be specified as public.

Coherence of Entities: Module, class, method etc. are entities. An entity is said to coherent, if it is organized on a consistent plan and all its parts fit together toward a common goal. Policy needs to be designed to make the modules more coherent.

Constructing Physical Modules: Modules of analysis phase have changed as more classes and associations have been added during object design phase. The object designer has to create modules with well-defined and minimal interfaces. The classes in a module should have similar kind of things in the system. There should be cohesiveness or unity of the purpose in a module. So that the classes, which are strongly associated can be put into a single module.

CHECK YOUR PROGRESS - I

1. Choose the incorrect statement in terms of Objects.
   a) Objects are abstractions of real-world
   b) Objects can't manage themselves
   c) Objects encapsulate state and representation information
   d) All of the mentioned
2. Which of the following points related to Object-oriented development (OOD) is true?
   a) OOA is concerned with developing an object model of the application domain
   b) OOD is concerned with developing an object-oriented system model to implement requirements
   c) All of the mentioned
   d) None of the mentioned

3. Which of the following is a disadvantage of OOD ?
   a) Easier maintenance
   b) Objects may be understood as stand-alone entities
   c) Objects are potentially reusable components
   d) None of the mentioned

4.Inherited object classes are self-contained.
   a) True
   b) False

3.9 SUMMING UP

- A model is a simplified representation of reality. It provides a means for conceptualization and communication of ideas in a precise and unambiguous form.
- Object Oriented Methodology is a new system development approach encouraging and facilitating reuse of software components.
- The basic steps of system designing using object-oriented methodology includes analysis, system design, object design and implementation.
- The object model describes the static, structural and data aspects of a system.
- The dynamic model describes the temporal, behavioural and control aspects of a system.
- The functional model describes the transformational and functional aspects of a system.
- Every system has all the three models. Each model describes one aspect of the system but at the same time contains references to the other models.
- An object is a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand.
- An object has the following four main characteristics - unique identification, set of attributes, set of states, and set of operations (behaviour).
- Class is a template where certain basic characteristics of a set of objects are defined. A class defines the basic attributes and the operations of the objects of that type.

- A link is a physical or conceptual connection between object instances.
- Systems design is the process or art of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. The system design process is generally divided into two sub phases – logical design and physical design.
- The logical design of a system pertains to an abstract representation of the data flows, inputs and outputs of the system.
- The physical design relates to the actual input and output processes of the system. This is laid down in terms of how data is input into a system, how it is verified/authenticated, how it is processed, and how it is displayed as output.
- Object oriented design is a process of refinement or adding details. The object-designer works to implement the objects discovered during analysis phase

3.10 ANSWERS TO CHECK YOUR PROGRESS

1. b) Objects can't manage themselves
2. c) All the mentioned
3. d) None of the mentioned
4. b) False

3.11 POSSIBLE QUESTIONS

Short answer type questions:

1. What is model? Why do we model?
2. What is object? Discuss the main characteristics of the object with examples from the real world.
3. What is class? Discuss the relationships between class and object.
4. Define association.

Long answer type questions:

1. What is object-oriented methodology? What are the advantages of object-oriented methodology?

2. What is object-oriented process? Discuss the steps of object-oriented process.
3. What are the three models involved in object-oriented Analysis? Define each one of them.
4. What do you mean by object design? What are the steps followed during the object design?
5. How can you combine object model, dynamic model and functional model to obtain operations on classes?
6. What are the steps an object designer has to follow during algorithm design?
7. What is dynamic model? How is it represented?
8. What is system design? What are two types of system design? Explain.

3.12 REFERENCES AND SUGGESTED READINGS

1. Object-Oriented Modeling and Design with UML, M. Blaha, J. Rumbaugh, Pearson Education
2. Object-Oriented Analysis & Design with the Unified Process, Satzinger, Jackson, Burd, Thomson
3. Object Oriented Analysis & Design, Grady Booch, Addison Wesley
4. Timothy C. Lethbridge, Robert Laganiere, Object Oriented Software Engineering, TMH