THIRD EDITION

With ANSI C++ Standards

Programming with C++

D Ravichandran

The McGraw-Hill Companies

Programming with C++

Third Edition

About the Author

D Ravichandran is currently based in Hyderabad and is a corporate trainer in software engineering, data structures and algorithms, and programming languages. He was earlier a senior faculty in the Department of Computing, Middle East College of Information Technology, Muscat, Sultanate of Oman. He was also a faculty member of Department of Computer Science and Engineering, Pondicherry Engineering College, Pondicherry, for more than 15 years. He is an expert in several computer programming languages and has more than two decades of professional programming experience. A prolific writer, he has already published many books in the field of computer science and information technology. His affiliations include a life membership of the Indian Society for Technical Education and a membership of the Computer Society of India.

The McGraw-Hill Companies

Programming with C++

Third Edition

D Ravichandran

Corporate Trainer in Software Engineering
Data Structures and Algorithms and Programming Languages
Hyderabad

Tata McGraw Hill Education Private Limited
NEW DELHI

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

Published by the Tata McGraw Hill Education Private Limited,
7 West Patel Nagar, New Delhi 110 008.

Programming with C++, 3/e

ISBN-13 digits: 978-0-07-068189-7
ISBN-10 digits: 0-07-068189-9

Vice President and Managing Director—McGraw-Hill Education: Asia Pacific Region: *Ajay Shukla*

Head—Higher Education Publishing and Marketing: *Vibha Mahajan*
Manager: Sponsoring—SEM & Tech Ed: *Shalini Jha*
Asst Sponsoring Editor: *Surabhi Shukla*
Development Editor: *Surbhi Suman*
Executive—Editorial Services: *Sohini Mukherjee*
Jr Production Manager: *Anjali Razdan*
Dy Marketing Manager—SEM & Tech Ed: *Biju Ganesan*

General Manager—Production: *Rajender P Ghansela*
Asst General Manager—Production: *B L Dogra*

Typeset at Bukprint India, B-180A, Guru Nanak Pura, Laxmi Nagar-110 092 and printed at
Avon Printers, Plot No. 16, Main Loni Road, Jawahar Nagar, Industrial Area, Shahdara, Delhi 110 094

Cover Printer: SDR Printers

RQXCRRQZDLCZL

Dedicated to my son

Suseekaran

for his love and support

Contents

Preface to the Third Edition

The book not only discusses the issues concerning the mystery of ANSI C++ but also makes a conscious effort to relate those insights to contemporary programming. This timeless and enlightening information is presented in a clear and concise manner. The new edition offers a fresh perspective of what ANSI C++ means and where ANSI C++ fits into the scheme of software life cycles. Thus, readers can gain requisite expertise by acquiring ANSI C++ programming skills and design ideas.

Aim of the Book

A welcome introduction to the world of programming, this book discloses facts and techniques on ANSI/ISO C++ and provides a knowledge base for advanced, standard-compliant, and efficient use of C++. It not only covers the syntax and semantics of ANSI C++ but also reveals the secrets of object-oriented programming through various topics, namely, classes, objects, inheritance, polymorphism and dynamic binding, and generic programming through STL. It offers stimulating insights into the loftiest thoughts and realisations of what ANSI C++ is and its relationship to modern software life cycles. A must read for all those who want to increase their understanding and awareness of object-oriented programming concepts, the book also serves the purpose of a handy reference for C++ programming professionals.

Users

The target audience for this book is two-fold—(i) computer novices who do not have any prior programming knowledge, and (ii) experienced C++ developers who seek a guide for enhancing their design and programming proficiency. Specifically, it can be used by undergraduate students of CSE, IT, ECE, EEE, Electronics and Instrumentation Engineering, BCA/MCA, and BSc/MSc (Computer Science/IT). Moreover, it would be an ideal reference for students of diploma and DOEACC courses in computer science and computer training institutes.

New to the Edition

- Broader and in-depth coverage of Object Oriented Programming concepts in 2 new chapters—*Chapter 1: Introduction to Object Oriented Programming and Chapter 2: Building ANSI C++ Programs*
- Detailed coverage of Standard Template Libraries (STL) in three new chapters—*Chapters 17: STL - Containers, Chapter 18: STL - Iterators* and *Chapter 19: STL - Algorithms and Function Objects*
- Enhanced coverage for topics such as data types, arithmetic operators, IOStreams, functions and program structures, special member functions, and exception handling
- Inclusion of new section on Namespaces in *Chapter 15: Templates, Namespace and Exception Handling*

Salient Features

The revised edition has been thoroughly updated with ANSI/ISO C++ syntax. This text offers one of the best reviews of ANSI C++ since it gives access to the most important concepts in object-oriented programming found anywhere. It introduces the syntax and features of C++ programming languages in a simple manner. The concepts are very well exemplified with program codes containing the inputs and outputs of the sample programs. It first explains the basic concepts (like functions, arrays, pointers and structures) and then progresses with the discussion on OOP concepts (like classes, objects, inheritance, polymorphism and templates) which will be helpful for the beginners in better understanding of the implementation and applications of the C++ language.

The book is impregnated with the following salient features:

- Offers a concise introduction to C++ and Object-Oriented Programming (OOP).
- Emphasises the use of software tools and covers the software engineering topics in detail.
- Provides pictorial representation in the form of syntax diagrams, flowcharts and Object Modeling Technique (OMT) class notation diagrams given.
- Elucidates the language features through executable codes which are tested on various compilers such as Linux GNU C++ and .Net Microsoft Visual C++.
- Facilitates the readers with simple and easy-to-understand format of the program execution (i.e., sample input and output).
- Explains how to avoid and correct typical errors.
- Describes concept review problems to test programming proficiency of readers on various ANSI C++ topics in a special section. Interactive exercises using the computer make learning fun.
- Refreshed and enhanced pedagogy includes **Programming Examples (359), Review Questions (439), Concept Review Problems (380) and Programming Exercises (197).** Answers to the Concept Review Problems are included in the Appendix.

The pedagogical features and their benefits are explained below:

Highlights	Description and Benefit	Examples
Introduction	Each chapter begins with an Introduction which helps the reader get a brief summary of the background and contents of the chapter.	Refer pages 1, 13, 32, etc.
Sections and Sub-sections	Object-oriented programming using C++ is one of the most widely discussed, debated, and examined elements of modern software life cycles, and also one of the most mysterious and misunderstood subjects. Therefore, each chapter has been neatly divided into sections and sub-sections so that the subject matter is studied in a logical progression of ideas and concepts.	Refer pages 35, 63, 79, etc.
Programming code with sample input and output	A set of programming codes, totaling to 314 problems is present in relevant chapters. All sample programs are well graded and tested using the different versions of the ANSI C++ compiler. These self-learning codes with sample inputs and outputs enable students to strive towards better comprehension of the concepts and also, master the programming skills.	Refer pages 87, 130, 222, etc.

Highlights	Description and Benefit	Examples
Flowcharts and Diagrams	Flowcharts and syntax diagrams presented at appropriate locations demystify the complexity of the difficult topics like pointers, strings, streams, inheritance polymorphism, file handling, templates, etc. Object Modeling Technique (OMT) diagrams easily illustrate the advance topics, functional relationships and definition sketches for mathematical models.	Refer pages 114, 133, 141, etc.
Review Questions	These are very useful for the faculty in setting class work, assignments, quizzes and examinations and help students in revising the learnt concepts.	Refer pages 13, 31, 71, etc.
Programming Exercises	This section takes an unbiased look at some of the more interesting and relevant ideas relating to programming. The practice questions help students get a clearer picture of the software design and coding.	Refer pages 111, 176, 247, etc.
Concept Review Problems	This section concentrates on a wide range of concepts such as syntax and semantic analysis of the code, spotting and identifying the logical errors, technical and complexity analysis. This enhances your knowledge and understanding of software engineering and also, improves your programming skills.	Refer pages 72, 105, 160, etc.
Answers to Concept Review Problems	Answers provided for all the Concept Review Problems at the end of the book as Appendix A help check your understanding of the learnt concepts.	Refer pages 820, etc.
References and Bibliography	A comprehensive list of references given at the end of the book further enhances the subject knowledge.	Refer pages 836, etc.

Organisation

This book consists of nineteen chapters which are as follows:

Chapter 1 presents the concepts and features of **Object-Oriented Programming** (OOP) and highlights some of the key terms of the OOP paradigm which are extensively used in this book.

Chapter 2 gives an overview of the latest addenda to ANSI/ISO C++ compiler and also suggests **how to build an ANSI C++ program** under various platforms, namely, GNU C++ for Linux and .Net VC++ for Windows.

Chapter 3 introduces the fundamentals of C++ programming language and summarises the most significant **data types**, **operators** and **expressions** used in ANSI C++.

Chapter 4 focuses on developing simple C++ programs with emphasis on the **Input and Output Streams** <iostream> and highlights the features of manipulator functions and Input and Output (I/O) stream flags.

Chapter 5 describes the principles and guidelines in the design and evolution of C++ through **control statements** which has become the standard for any programming language.

Chapter 6 deals with user-defined **functions** and **program structures** and stresses on how to define and use the different types of arguments (namely, actual, formal, local and global variables); how to use the recursive functions, nested functions and preprocessors.

Chapter 7 explains the importance of **array** data types in C++. It describes how to define, declare and use single dimensional, multidimensional and character arrays. Array notation, array initialisation and types of data storage such as static, automatic, and free store are also dealt with numerous examples.

Chapter 8 delves on the syntax and semantics of **pointer** data type which is one of the strengths of the C++ language. In addition, it demonstrates the use of strings and advanced memory management techniques using complex pointer data types and also guides the user how to avoid common pointer related errors.

Chapter 9 deals with functional characteristics of **structure** and **union** data types. It also describes how to declare, define and use the array of structure, structure within structure, pointer to structure, union tags and bit fields.

Chapter 10 elucidates the salient features of object-oriented programming and explains how **classes and objects** can be defined, declared and used in C++. Special attention is given for defining the various types of class declarations.

Chapter 11 covers the syntax and semantics of the **special member functions** such as constructors, destructors, inline member functions, static class members and friend functions as well as their role in class design. It also demonstrates several techniques and guidelines for an effective usage of these special member functions.

Chapter 12 discusses one of the most important features of the OOP, namely, **inheritance**. Single and multiple inheritance, types of derivation, public inheritance, private inheritance, protected inheritance, container classes and member access control are explained with suitable number of examples.

Chapter 13 exemplifies the concepts of **function and operator overloading**, and explores the benefits as well as the potential problems of operator overloading. It discusses the restrictions that apply to operator overloading and also explains how to avoid the common errors while using operator and function overloading.

Chapter 14 narrates the central attraction of the OOP—**polymorphism** with pointers and **virtual functions**. Early binding, virtual functions, late binding, pure virtual functions, abstract base classes, constructors under inheritance, destructors under inheritance, virtual destructors and virtual base classes are presented, with well-graded examples.

Chapter 15 presents the various aspects of designing and implementing **templates**, including class templates, function templates, and template issues that are of special concern. This chapter describes the standard **exception handling** using the keywords—try, catch and throw. It also elucidates the rationale behind the addition of **namespaces** to the language and the problems that namespaces solve. Furthermore, how to declare, define and use the namespace alias, nested namespace, unnamed namespace and namespace std, are covered in this chapter.

Chapter 16 gives the **data file operations** in C++ and focuses on how to read and write a class of objects from the files of secondary storage devices. The ANSI-ISO C++ streams and file processing commands are dealt with suitable illustrations.

Chapters 17–19 provide coverage on introduction to the **Standard Template Library (STL)** and generic programming in general. It discusses the principles of generic programming, focusing on STL as an exemplary framework of generic programming. These chapters also demonstrate the use of STL components such as containers, algorithms, iterators, allocators, adapters, binders, and function objects.

Online Learning Center

The accompanying web supplement *http://www.mhhe.com/ravichandran/cp3e* provides an additional resource for students and instructors.

Acknowledgements

I am grateful to Dr T Sundararajan, Professor, Department of Civil Engineering, Pondicherry Engineering College, for his timely support, encouragement, valuable comments, suggestions and many innovative ideas in carrying out this project. I am indebted to my teachers, mentors and professors who taught me the art of computer programming during my studentship at Indian Institute of Technology, Kharagpur, especially, to Prof. Swapna Banerjee, Prof. N B Chakaraborthy and Prof. J C Biswas.

I extend my appreciation to Mr Christian Wolff, Heidelberg, Germany, for his continuous motivation, love and advice in my life. I would like to express my gratitude towards Mr Arun, Mr Walid, Mr Shariq Ali and Dr. Gulam Ahmed, Middle East College of Information Technology, Muscat, Sultanate of Oman, for their technical comments and suggestions. I am thankful to my students [in India and abroad] who have helped me a lot in bringing out this edition and would like to specially acknowledge the efforts of Mr Al Walid Al Busaidi, Muscat; Mr Ashwin Kumar Chummun, UK; Mr Gowathaman, France; Mr Sampath Reddy, US; Mr Sudheer Reddy, US; Mr Tushar Ranjan Sahoo and Dr Ram Niranjan Sahoo JIPMER, Pondicherry.

My earnest thanks are also due to the editorial and publishing professionals at Tata McGraw-Hill for their keen interest and support in bringing out this book in record time. There have been several professors who have participated in the review process of this book. I would like to sincerely acknowledge them for their valuable suggestions and encouragement.

Akshay Girdhar
Guru Nanak Dev Engineering College, Ludhiana, Punjab

Amit Jain
Bharat Institute of Technology, Meerut, Uttar Pradesh

Harish Kumar
Panjab University, Chandigarh, Punjab

Prashant Sharma
Anand Engineering College, Agra, Uttar Pradesh

Dinesh Kumar Tyagi
Birla Institute of Technology and Science, Pilani, Rajasthan

Md Tanwiruddin Haider
National Institute of Technology, Patna, Bihar

Mahua Banerjee
Xavier Institute of Social Service, Ranchi, Jharkhand

N K Kamila
C V Raman College of Engineering, Bhubaneswar, Orissa

Pranam Paul
Dr B C Roy Engineering College, Kolkata, West Bengal

Sajal Mukhopadhya
National Institute of Technology, Durgapur, West Bengal

Kanhaiya Lal
Birla Institute of Technology, Patna, Bihar

Poornachandra Sarang
University of Mumbai, Mumbai, Maharashtra

Manisha J Somavanshi
Indira Institute of Management, Pune, Maharashtra

T V Gopal
Anna University, Chennai, Tamil Nadu

N Shanthi
K S Rangasamy college of Technology, Tiruchengode, Tamil Nadu

Annappa
National Institute of Technology, Surathkal, Karnataka

CH V K N S N Moorthy
R K Institute of Science and Technology, Hyderabad, Andhra Pradesh

M M Naidu
S V University College of Engineering, Tirupati, Andhra Pradesh

Finally, I thank my parents, son and wife for the love, encouragement and comfort they have extended to me throughout my career.

D Ravichandran

Feedback

The readers of the book are encouraged to send their comments, queries and suggestions at the following email id—*tmh.csefeedback@gmail.com*, mentioning the title and author name in the subject line. Also, please report to us any piracy of the book spotted by you.

Chapter 1

Introduction to Object Oriented Programming

This chapter focuses on the definitions, basic concepts and salient features of Object Oriented Programming (OOP). The pros and cons of Structured Procedural Programming (SPP) with Object Oriented Programming (OOP) are also summarised. Major applications of OOP are also highlighted in this chapter. It also describes how C++ can be used to improve productivity and software quality by offering features such as classes, objects, data hiding, encapsulation, inheritance, polymorphism and templates.

1.1 INTRODUCTION

A major challenge for software engineering today is to improve the software programming process as modern software life cycle has been changing very dramatically since the late nineties wherein the code re-usability, reliability and maintainability are the key features. The very aim of using an object oriented programming language is to handle a complex software design in a very easy, simple and efficient manner. Redesigning and maintaining the source code costs much more than the reusability of the source code. The turnover time and software cost are drastically brought down. The main aim of designing the C++ language is to support both a procedure oriented style and an object oriented programming paradigm. In that sense, C++ is a hybrid language which supporzts both the procedural as well as object oriented programming styles.

Softwares designed using object oriented technology can meet up the challenges of large real world systems by enhancing the ability to produce reliable and maintainable code. Through object oriented programming and design, such software can naturally evolve to meet changing needs. To effectively accomplish this, one must learn new ways of thinking about programming and problem solving.

Therefore, Object Technology (OT) is drawing attention and consideration in many areas of computing, such as

- programming
- data bases

- system analysis and design
- computer architecture
- operating systems
- expert systems, and
- internet client/server programming

1.2 WHAT IS OBJECT ORIENTED PROGRAMMING (OOP)?

Object oriented programming, or OOP, is a software development philosophy based on the following central ideas:

- encapsulation
- inheritance
- information hiding
- data abstraction and
- polymorphism

Object Oriented Programming has revolutionised the very art and practice of writing computer applications. Object is the basic unit of object oriented programming. Designing an object-oriented model involves defining a set of classes. A class is a template from which objects are created. The template, or blueprint, provided by a class specifies a set of data and methods that all objects created according to its specifications will contain.

Hence, the object oriented programming approach has the advantage of producing more reliable softwares for complex and large-scale systems.

1.3 STRUCTURED PROCEDURAL PROGRAMMING (SPP)

In the late seventies, Structured Procedural Programming (SPP) was widely used for designing and developing softwares. Structured programming is a programming paradigm that to a large extent relies on the idea of dividing a program into functions and modules (Fig. 1.1).

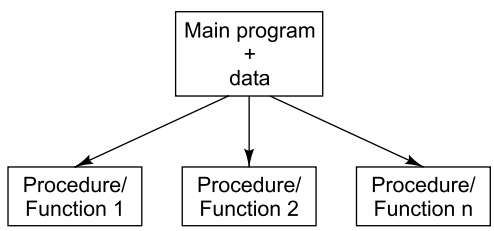


Fig. 1.1 *Procedural Programming Approach*

As programs became larger for real life applications, they were broken down into smaller units, such as functions, procedures, and subroutines. Functions can be grouped together into modules according to their functionality, objectives and tasks. In other words, SPP emphasises mostly functional decomposition and procedural abstraction for designing and developing software systems.

However, SPP was found to be unsuitable for handling complex software systems due to lack of code reusability, extensibility and maintainability. One of the main drawbacks of SPP is that data and functions have to be stored separately and the data has to be globally accessed, as the systems are modularised on the basis of functions. Information hiding and data encapsulation are not supported in SPP and therefore, every function can access every piece of data. Functions have unrestricted access to global data. Changing the global data in a module causes program side effects and that code becomes unreliable and error prone in a complex system.

Some of the examples for procedural languages are ‘C’, Pascal, and Fortran.

It is well known that 'C' is widely accepted as a well structured programming language for a variety of applications. It has many advantages over other high level programming languages. But it has flaws and limitations that has made it unsuitable for complex programming projects.

1.4 OBJECT ORIENTED PROGRAMMING (OOP)

Object Oriented Programming (OOP) alleviates some of the problems mentioned above. The OOP approach has the advantage of producing better structured and more reliable softwares for complex systems, greater reusability, more extensibility, and easy maintainability.

In object oriented programming, systems are modularised on the basis of data structures (objects). Object's state (data types) and behavior (operations) are encapsulated. Message passing ensures that an object's internal state can be accessed only if permitted, as encapsulation prevents unauthorised access (Fig. 1.2).

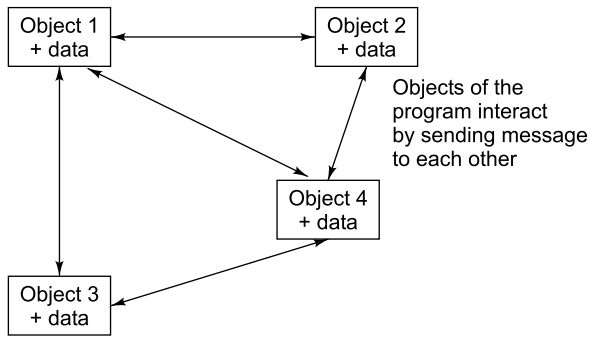


Fig. 1.2 *Object Oriented Programming (OOP) Approach*

Real world is represented more closely by objects mimicking external entities. Objects of the program interact by sending messages to each other. Each object is responsible to initialise and destroy itself correctly. Consequently, there is no longer the need to explicitly call a creation or termination procedure.

1.5 CHARACTERISTICS OF OOPs

Following are the major characteristics for considering any programming languages to be object oriented:

- objects
- classes
- data abstraction
- data encapsulation
- information hiding
- message passing
- inheritance
- dynamic binding
- polymorphism, and
- overloading

1.5.1 Objects

In Object Oriented Programming (OOP) paradigm, objects are the fundamental building blocks for designing a software. In other words, an object is a collection of data members and the associated member functions are known as methods. Objects are identified by its unique name (Fig. 1.3(b)). An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data.

An object has three characteristics:

- name
- state
- behaviour

(a) Name It is a unique identity for representing an object of a class.

(b) State It is a representation of attributes by internal data structures.

(c) Behaviour It is a set of allowed operations, functions and methods.

An object communicates with another by passing messages using some protocols.

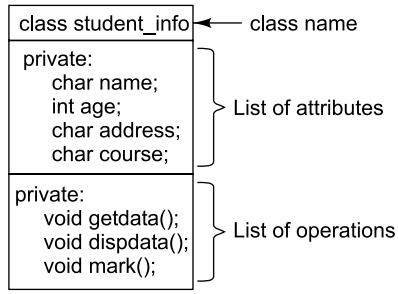


Fig. 1.3(a) *UML object Diagram for the Class Student_info*

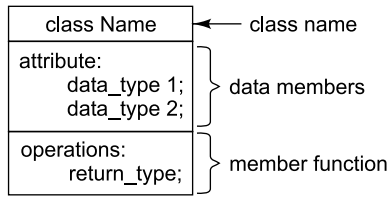


Fig. 1.3(b) *UML Class and Object Diagram*

1.5.2 Class

A class is a template for constructing objects (Fig. 1.3(a) & (b)). Objects are instances of a class, i.e., specific occurrences of a class. Once a class is defined, any number of objects of that class are easily created. The code or class implementation (or class body) contains the definition of each method (member functions).

1.5.3 Data Abstraction

Data abstraction is an encapsulation of an object's state and behaviour. Data abstraction increases the power of programming language by creating user defined data types (Fig. 1.4). Data abstraction also represents the needed information in the program without presenting the details.

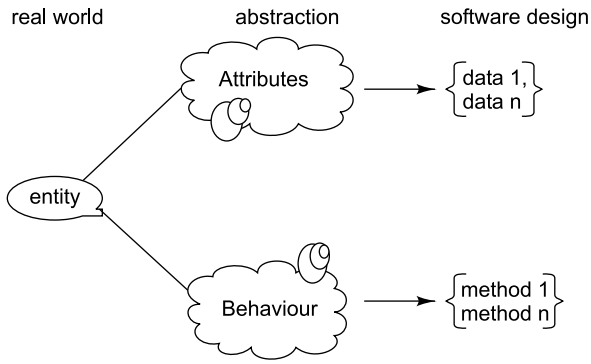


Fig. 1.4 *Data Abstraction*

1.5.4 Data Encapsulation

Data encapsulation combines data and functions into a single unit called class. When using data encapsulation, data is not accessed directly; it is only accessible through the methods (functions) present inside the class. Data encapsulation enables data hiding, which is an important concept possible of OOP (Fig. 1.5).

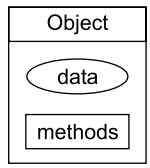


Fig. 1.5 *Data Encapsulation*

1.5.5 Information Hiding

Information hiding means that the implementation details of an object's state and behaviour are hidden from users and other objects to protect the state and behaviour from unauthorised access (Fig. 1.6).

Fig. 1.6 *Information Hiding*

1.5.6 Message Passing

In OOPs, processing is accomplished by sending messages to objects (Fig. 1.7). How messages are executed or implemented is defined in the class methods. Methods give the implementation details for the messages and represent a class behaviour. A message passing is equivalent to a procedure call or a function call of a procedural programming.

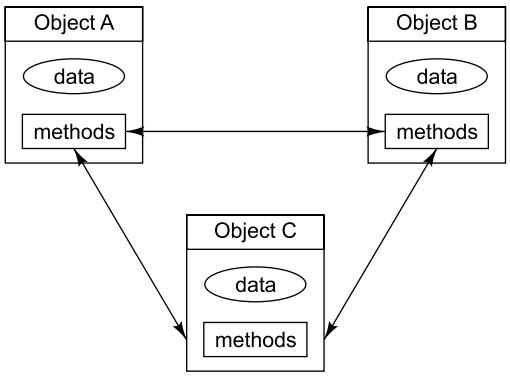


Fig. 1.7 *Message Passing within Objects*

1.5.7 Inheritance

Inheritance is the process of forming a new class from an existing class or base class. The base class is also known as *parent class* or *super class*. The new class that is formed is called *derived class*. Derived class is also known as a 'child class' or 'sub-class'.

Sub-classes (derived classes) inherit some or all of the properties of their superclasses (base classes). Inheritance organises classes into a hierarchy, allowing implementation and structure to be shared. Thus, reuse becomes automatic as the code from a superclass can be reused by a subclass. New classes inherit both the state and behaviour from existing classes.

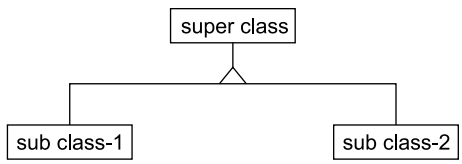


Fig. 1.8 *Single Inheritance*

In single inheritance, each subclass has only one immediate superclass (Fig. 1.8).

In multiple inheritance, each subclass has more than one superclass (Fig. 1.9).

Inheritance helps to reduce the overall code size of the program, which is an important advantage of object-oriented programming.

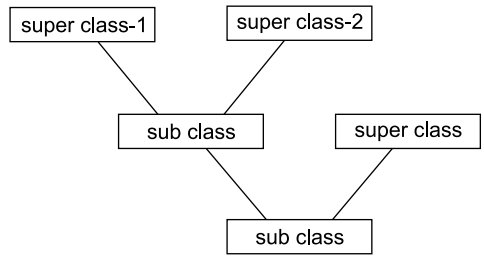


Fig. 1.9 *Multiple Inheritance*

1.5.8 Dynamic Binding

In dynamic binding, message passing to the objects can be done during run time. Late binding or run time binding have the same meaning as dynamic binding.

In a procedural programming, procedure call or function invocation can be done only during compilation and it is called as static binding or early binding of a compiler. In a pure object oriented programming, message passing to the objects can be done only during run time.

1.5.9 Polymorphism

Polymorphism uses dynamic binding and virtual methods by which different descendant objects can respond in their own unique ways to the same method. Polymorphism enables programmers to manipulate subclass objects using superclass references.

1.5.10 Overloading

Overloading allows an object to have different meanings depending on its context. There are two types of overloading, namely, operator overloading and function overloading. When an exiting operator begins to operate on a new data type, it is called *operator overloading*. When a message passing to the objects is done with a different data type, or class, then it is called as *function overloading*. Overloading is one type of polymorphism.

1.6 ADVANTAGES OF OOPs

The major advantages, benefits and merits of OOPs are given below:

1. Since OOP provides a better syntax structure, modelling real world problems is easy and flexible.
2. Complex software systems can be modularised on the basis of objects and classes.
3. Creation and maintenance of an OOP code is easy and hence reduces software development time.
4. Since OOP technique supports reusable software components libraries, software reengineering can be synthesised, implemented and realised easily. For example, reusable software components libraries are called as *standard templates* in C++ and *packages* in Java.
5. The abstract data type concept decouples the object specification and object implementation.
6. Data encapsulation and information hiding increases software reliability and modifiability.
7. Polymorphism and dynamic binding increases flexibility of code by allowing the creation of generic software components.
8. Inheritance allows software code to be extensible and reusable. New attributes and new operations can be added through the creation of new child object classes without modifying the original code.

1.7 DISADVANTAGES OF OOPs

The main drawbacks of using OOPs in modern software development life cycles (SDLC) are:

1. OOP software development, debugging and testing tools are not standardised.
2. The functional data and process decomposition tools such as Entity-Relationship (ER) diagrams and Data Flow Diagrams (DFD) modules need to be adapted to allow for OOP decomposition of classes and objects.
3. Even though OOP reduces the software development time, OOP has a steep learning curve. Software engineers, system analysts and programmers need to learn to model real world problems into a set of interacting objects and class hierarchies.
4. OOP decomposition of a large hierarchy of classes is complex and difficult to manipulate especially for computer novices.
5. In real life software applications, hierarchy of classes must be decomposed properly using OOP based tools as inheritance and polymorphism can hide from a bad design.
6. Polymorphism and dynamic binding also require long processing time, due to overhead of function calls during the run time.

1.8 COMPARISON OF STRUCTURED PROCEDURAL PROGRAMMING (SPP) AND OBJECT ORIENTED PROGRAMMING (OOP)

In order to understand the basic concepts in C++, a programmer must have a command of the basic terminology in object-oriented programming. Table 1.1 summarises the major characteristics and salient features of SPP and OOP:

Table 1.1 *Major characteristics and salient features of SPP and OOP*

Characteristics	*SPP*	*OOP*
1. Program modularisation	On the basis of functions	On the basis of data structures called objects and classes
2. Design approach	Top-down	Bottom-up
3. Data	Move openly around the system from function to function.	Data is mostly hidden or permits restricted access due to public, private and protected rights.
	By default, all data are public and hence provide global access.	By default, all data are private and hence provide local access only.
4. Problem emphasis	Represented by logical entities and control flow	Represented more closely with interacting objects and classes.
5. Usage of abstraction	Procedural abstraction	Class and object abstraction
6. Function call (method invocation)	Programmers are responsible for calling the active procedures to pass parameters and arguments	Active objects communicate with each other by passing messages to activate their operations
7. Unit/module structure	Statement or expression	Object treated as a software component

(*Contd*)

(*Contd Table 1.1*)

Characteristics	*SPP*	*OOP*
8. Code reusage	It does not support code reusability	Due to class hierarchy a part of the state and behaviour can be re-engineered to a subclass
9. Problem decomposition	Functional approach	Object oriented approach
10. Function and procedure	Mostly shares global data	State (data types) and behaviour (methods) are tied together in a data structure called an object
11. Code Binding	By default, all function calls are static or during compilation time.	It supports both early (static) binding and late (dynamic) binding.
	There is no provision for late binding.	Method invocation can be done even during run time.
12. Polymorphism	It does not support. No concept of virtual functions.	Polymorphism and dynamic binding are the major features of OOP
13. Languages	Procedure oriented languages, such as C, Pascal, Modula-2, FORTRAN	Object Oriented Languages, such as C++, Java, C#, Smalltalk, Simula, CLOS

1.9 STEPS IN DEVELOPING OOP PROGRAMS

There is a fundamental shift in the way programs are designed, developed, tested, and maintained using OOP methodology. Programmers should realise that in the OOP approach, object oriented decomposition is mostly emphasised in place of functional decomposition. Secondly, grouping of object classes on the basis of hierarchical dependency of classes is maintained instead of grouping functions together.

Following are the steps to develop a new software system using the object oriented approach:

1. State the problem, that is, the user requirements are analysed and given in a simple descriptive language.
2. Identify the object classes and their attributes (data members) and the data operations (member functions) associated with each object class as a guideline.
3. The system specification is broken into a number of modules, with each module consisting of one or more object classes.
4. Declare and define each object class by encapsulating its attributes and operations.
5. Identify the message passing between interacting object classes by identifying requests answered and services required by each object class.
6. Identify inheritance relations and class hierarchies on the basis of dependencies between object classes.
7. Create a logical OOP model for the proposed system that shows the interaction between objects.
8. Develop algorithms for member functions of each object class to process its data members.
9. Plan an implementation strategy to code OOP program modules using an appropriate OOP language.
10. Prepare, test, deliver, and maintain plans for the new software.

1.10 STRUCTURE OF OBJECT ORIENTED PROGRAMS

In C++, classes are first declared and are normally put in a separate header file. Then the member functions for each class are defined. Finally, the user code is written to create instances of classes (objects) and to perform the required tasks. A class's attributes and behaviours are implemented using data members (instance variables) and member functions, respectively.

Information hiding is implemented by declaring members (data or functions) with one of the three categories of accessibility: private, public, and protected. Any function (member or nonmember) can access a public member. Only member functions of the class can access a private member. When members are protected, they can be accessed by member functions of the base class and its derived classes but not by nonmember functions. A derived class inherits all members from the base class.

The member functions can be divided into two categories:

1. Constructor, which creates and initializes an object, and destructor, which destroys the object. They carry the same name as the class and are called automatically.
2. Implementor functions, which perform the required operations of the object.

PROGRAM 1.1

A program to illustrate how to construct a single linked list using object oriented programming technique.

```
// Implementation of Single Linked List
// List demonstration
#include <iostream>
#include <cstdlib>
#include <cstdio>
using namespace std;
struct node_info {
    int data;
    struct node_info *ptr;
};
class single_list {
    private:
        struct node_info *head;
        struct node_info *list;
    public:
        void list_initialize();
        void create_list();
        void traverse_list();
        void menu();
};

void single_list :: list_initialize()
{
    head = list = NULL;
}

void single_list :: menu()
{
    cout << "Implementation of Single Linked list \n";
    cout << " a -> adding an element to the list \n";
    cout << " t -> traversing the list \n";
    cout << " m -> help menu \n";
    cout << " q -> quit from the main program \n";
}
```

```
void single_list :: create_list()
{
     struct node_info *temp;
     temp = (struct node_info *) malloc(sizeof(struct node_info));
     cout <<" Enter data to the current Node\n";
     cin >> temp->data;
     if (head == NULL) {
         head = temp;
         temp->ptr = NULL;
         list = temp;
     }
     else {
         list->ptr = temp;
       temp->ptr = NULL;
         list = temp;
     }
}

void single_list :: traverse_list()
{
     struct node_info *temp;
     if ( head == NULL ){
         cout << "The list is empty \n";
         return;
     }
     else {
         temp = head;
         while ( temp != NULL ) {
             cout <<" Content of the Node =";
             cout << temp->data << "\n";
             temp = temp->ptr;
         }
     } //end of else statement
}

int main()
{
     single_list obj;
     char ch;
     obj.list_initialize();
     obj.menu();
     cout << "enter your choice \n";
     while (( ch = getchar()) != 'q') {
          switch (ch) {
               case 'a':
                     obj.create_list();
                     break;
               case 't':
                     obj.traverse_list();
                     break;
               case 'm':
                     obj.menu();
                     break;
               case 'q':
                     exit(1);
          } // end of switch statement
      } // end of while statement
     return 0;
}
```

Output of the above program

```
Implementation of Single Linked list
 a -> adding an element to the list
 t -> traversing the list
 m -> help menu
```

```
q -> quit from the main program
Enter your choice
a
Enter data to the current Node
10
a
Enter data to the current Node
20
a
Enter data to the current Node
30
a
Enter data to the current Node
40
t
Content of the Node =10
Content of the Node =20
Content of the Node =30
Content of the Node =40
```

1.11 OBJECT ORIENTED LANGUAGES

It is well known that Object Oriented Programming provides major advantages in the creation and maintenance of software. These include shorter development time and a high degree of code sharing and flexibility. These advantages make object oriented programming an important technology for building complex software systems now and in the future. A number of languages are claimed to be object oriented. The following are certain well-known Object Oriented Programming languages:

- smalltalk
- Common Lisp Object System (CLOS)
- Object Pascal
- Object C
- C++
- Java
- C#

1.12 IMPORTANCE OF C++

C++ is a general-purpose, platform-neutral compiled programming language that supports various programming paradigms, including procedural programming, object-based programming, object-oriented programming, generic programming and functional programming.

In recent times, the object oriented programming (OOP) paradigm has become popular in modern software life cycles. Now, C++ has the status of a structured programming language with Object Oriented Programming (OOP) methodology, in which the software reusability, testability, maintainability, portability and reliability are the key features and requisities of modern software development.

Despite a serious competition with Java, Visual Basic, and Microsoft .NET family of languages (e.g. C#), C++ is the only truly object oriented language adopted by the IT industry. The most significant impact of C++ is its ability to support many different programming paradigms. C++ easily adapts to procedural abstraction, modular abstraction, data abstraction and most importantly object oriented programming.

The important features added in C++ are a class construct with private, public and protected sections. The derived classes provide both single and multiple inheritance mechanism. The additional facilities included are as follows:

- polymorphism
- dynamic binding
- virtual functions
- run time type checking
- overloading functions and operators
- exception handling
- standard template libraries

C++ has become quite popular due to the following reasons:

- It supports all features of both structured programming and object oriented programming.
- It gives the easiest way to handle data hiding and encapsulation with the help of powerful keywords such as class, private, public and protected.
- Inheritance, one of the most powerful design concept is supported with single inheritance and multiple inheritance of base class and derived classes.
- Polymorphism through virtual functions, virtual base classes and virtual destructors give the late binding of the compiler.
- It provides overloading of operators and functions.
- C++ focuses on function and class templates for handling parameterized data types.
- Exception handling is done by the extra keywords, namely, try, catch and throw.
- C++ provides special member functions such as friends, static methods, constructors and destructors for the class objects in order to create and destroy the objects easily and effectively.
- Standard Template Library (STL) supports not only containers, iterators and algorithms to perform operations such as searching and sorting but also templates for generic algorithms.

REVIEW QUESTIONS

1. What is meant by object-oriented paradigm?
2. Explain the importance of OOP technology.
3. Define the following terms with respect to OOP:
   (a) objects (b) classes (c) data abstraction
   (d) data encapsulation (e) information hiding (f) message passing
   (g) inheritance (h) dynamic binding (i) polymorphism
   (j) overloading
4. What are the demerits of using a structured procedural programming?
5. Explain how an OOP technique improves the software system.
6. Summarise the advantages and disadvantages of OOP.
7. Compare the Structured Procedural Programming (SPP) with that of an Object Oriented Programming (OOP).
8. Explain the steps involved in developing OOP.
9. Summarise the major Object Oriented Languages used in the field of software engineering.
10. Elucidate the importance of C++.

Chapter 2

Building ANSI C++ Programs

This chapter presents the history of C++ language and also highlights some of the key terms that are used extensively in ANSI/ISO C++. This chapter also shows the ways and means of how to write, compile, debug and execute a C++ program under different environments, namely, GNU C++ under Linux/UNIX OS and Visual C++ under .NET Windows platform.

2.1 INTRODUCTION

This section is for those people who want to learn programming in C++ and do not necessarily have any previous knowledge of other programming languages. Of course, any knowledge of other programming languages or any general computer skill can be useful to better understand this tutorial, although it is not essential.

2.2 HISTORY OF C++

The evolution of C++ has been a continuous and progressive process, rather than a series of brusque revolutions. C++ today is very different from what it was in 1983, when it was first named "C++". Many features have been added to the language since then; older features have been modified, and a few features have been deprecated or removed entirely from the language. Some of the extensions have radically changed the programming styles and concepts.

The list of extensions include: const member functions, exception handling, templates, new cast operators, namespaces, the Standard Template Library, bool type, and many more. These have made C++ the powerful and robust multipurpose programming language that it is today.

2.2.1 Origins of C++

Simula67 was the first object oriented programming language which was introduced in late 1960s. It was mainly used for writing event-driven simulations. Classes, inheritance and virtual member functions were integral features of the Simula67.

Smalltalk first appeared way back in 1972 which offered not only a pure object-oriented programming environment but also added many features from Simula67, Ada and Modula-2. In fact, Smalltalk and Simula67 were not only the precursors to object oriented programming but also made considerable contributions to development of OOP.

2.2.2 C with Classes

The style of C programming was so innovative and revolutionary that it became a standard in the software industry for designing Operating Systems (OS) under UNIX. C is not only a general purpose programming language but also widely accepted for a variety of applications. It has many advantages over other high level programming languages. But it has flaws and limitations that has made it unsuitable for complex programming projects.

In 1979, Bjarne Stroustrup of AT&T started to experiment with extensions to C to make it a better tool for implementing large-scale projects. By adding classes to C, the resultant language 'C with classes' could offer better support for encapsulation and information hiding. The core of the C language was retained in "C with classes" and that added most of the OOP features from Smalltalk and Simula67.

2.2.3 Enter C++

Several modifications and extensions were made to 'C with classes' in 1983 in order to meet the requirement for the software industry and consequently the language 'C with classes' was renamed as "C++".

Between 1985 and 1989, C++ underwent a major reform. Protected members, protected inheritance, templates, and multiple inheritance were added to the language. A number of compilers and extensions to the language were introduced by many software vendors. It was clear that C++ needed to become standardised.

2.2.4 ANSI Committee Established

In 1989, American National Standards Institution (ANSI) constituted a committee for standardising C++. The official name of the committee was X3J16, and later it was changed to J16.

The ANSI C committee used 'The C Programming Language' by Kernighan and Ritchie as a starting point. Likewise, the ANSI C++ committee used the 'Annotated C++ Reference Manual' by Ellis and Stroustrup as its base document for building and standardising.

2.2.5 The Importance of Standardising C++

The following points highlight the importance of standardising C++:

1. Language Stability It is no doubt that C++ is probably the most widely used programming language both in commercial and in academic establishments today. Learning it from scratch is a demanding and time consuming process. It is guaranteed that, henceforth, learning C++ is a one-time investment rather than an iterative process.

2. Code Stability Due to different versions of C++ compilers from various software vendors, code stability and uniformity becomes an important problem faced by the developers. Therefore, ANSI standard specifies a set of deprecated features that might become obsolete in the future. Other than that, fully ANSI-compliant code is guaranteed to work in the future.

3. Manpower Portability Since the language commonality is enforced, C++ programmers can switch more easily to different environments, projects, compilers, and to different software companies.

4. Easier Portability The standard defines a common denominator for all platforms and compiler vendors, enabling easier porting of software across various operating systems and hardware architectures.

2.3 THE LATEST ADDENDA TO ANSI/ISO C++

This section summarises a panorama of the latest addenda to the ANSI/ISO C++ Standard. This section also explains some of the key terms that are used in the standard:

1. New Typecast Operators The new cast operators make the programmer's intention clearer and self-documenting. The following keywords are used to handle the typecast operation in ANSI C++:

- static_cast
- dynamic_cast
- const_cast
- reinterpret_cast

2. Run Time Type Identification The Run Time Type Identification (RTTI) of an object can be accomplished by the following keywords:

- typeid
- type_info

3. Built-in Bool Type The built-in bool data type was added to the ANSI C++ standard. The use of explicit keywords such as 'true', 'false', and 'bool' is self-documenting and is more evident than the use of int values. Hence, readability and portability are the major advantages of using a standardised Boolean data type.

4. Namespaces Namespaces were the latest feature to be added to the language. Namespaces are used to prevent name conflicts and to facilitate configuration management and version control in large-scale projects. Most of the components of the Standard Library are grouped under namespace std.

5. Exception Handling Exception handling is used to report and handle runtime errors and that has been refined and improved in ANSI C++. The following keywords are used to handle the error handling mechanism:

- try
- catch
- throw

6. Constructing Safer Classes and Objects ANSI/ISO C++ supports the safe form of constructing classes and objects from unintentional modification of data member or its member functions. The following keywords are used for declaring objects as a mutable object member, const data or const member function, etc.

- const
- explicit
- mutable

7. Templates The old pattern of using macros was replaced with the templates. A template is a mold or a blueprint from which related functions or classes are instantiated. The new template features give advantage of writing compact codes for functions and classes.

8. The Standard Template Library (STL) The Standard Template Library, or STL, comprises a substantial part of ANSI/ISO C++ addition. STL is a collection of generic containers, iterators, function objects, allocators

and algorithms. Some of the examples for generic containers are vector, list, and stack. Generic algorithms are used for sorting, finding, merging, and transforming these containers.

9. New Form of Standard I/O Streams The standard stream and string classes have been templatized to support both narrow and wide characters. The keyword wchar_t is used to handle wide character streams.

10. New Form of Using Header File The use of header file has been modified in ANSI C++ standard. The new standardised class libraries such as complex, string, exception, etc., are also added.

11. Memory Management The Standard now defines deploying the auto_ptr for the safe release of dynamically created objects. Overloading of the operators new and delete and advanced memory management techniques are the additional features in the ANSI C++.

12. Constructors and Destructors Fundamental data types can be initialised by a special constructor. In addition, the standard also defines a pseudo destructor for each of these types.

2.4 POSSIBLE FUTURE ADDITIONS TO C++

It is speculated that the following new features will be added to C++ in future:

- Automatic garbage collection
- Object persistence
- Support for concurrency and multithreading
- Extensible member functions
- Dynamically linked libraries
- Rule-based programming

However, automatic garbage collection, concurrency, and object persistence are already implemented in many other object oriented programming languages. In future, they may be added to C++ as well.

2.5 C++ VERSUS C

C++ is not only derived from the C language, but also a superset of C that means almost every correct statement in C is also correct in C++. The most important elements added to C are concerned with classes, objects and object oriented programming.

The following features of the C++ language or library are not supported in C. A major portion of C++ and its library fall into this category. A partial list of these features includes:

- anonymous unions
- classes
- constructors and destructors
- exceptions and try/catch blocks
- external function linkages
- function overloading
- member functions
- namespaces
- new and delete operators
- operator overloading
- reference types
- standard template library (STL)

- template classes
- template functions

2.6 VERSIONS OF C++

Many software vendors have released ANSI C++ specific compilers for different platforms. Following is a partial list of well-known C++ compilers that are used globally. Some of the compilers can be downloaded for free from the internet and the others have to be acquired by making the specified price.

- Apple C++
- Borland C++
- GNU C++ for Linux
- IBM C++ for IBM power, System Z
- HP C++ for Unix and HP C++ for OpenVMS
- Intel C++ for Windows and Linux
- SGI C++
- Sun C++
- Microsoft Visual C++ under .Net
- Turbo C++

Compatibility Notes The ANSI C++ standard which is accepted as an international standard is of recent origin. It was released in November 1997, even though C++ language exists from 1980. Therefore, there are many compilers which do not support all the new capabilities included in ANSI C++, specially those released prior to the publication of the standard.

2.7 SOURCE PROGRAM NAMES

C++ is a hybrid language, supporting both the structured programming language "C" and Object Oriented Programming"C++". Therefore, a C++ compiler can accommodate both C and C++ languages. So, it is essential to have a method to determine the type of code that is processed. To perform a C compilation, the source program files have an extension of .C. On the other hand, files have an extension of .CPP for specific C++ compilation.

Unlike several other programming languages, C++ does not impose a specific programming paradigm on its users. This liberty has two major advantages: it enables reuse of C code with minimal or no modifications at all, and it enables designers to choose the paradigm that suits their needs best.

2.8 COMPILING AND DEBUGGING C++ PROGRAMS

This section introduces key mechanics of C++ such as warnings, errors, portable code generation and performance optimisation. Compiling is the process of translating a source code into an executable code. Debugging is the art of making that code error free and make it run. These are the mechanical aspects of programming. Programmers need to know how to run compilers and debuggers which are the essential tools for a good programmer.

2.8.1 The Compilation Process (Fig. 2.1)

Compilation involves the following four separate processes which always take place in the following order:

(a) **Preprocessor**
The preprocessor expands directives such as #include in a program. This output is piped directly to the compiler.

(b) **Compiler**
The compiler translates the preprocessed C and C++ statements into the assembly language and stores it in an intermediate file.

(c) **The assembler**
The assembler translates the assembly language statements into object code and stores it in an intermediate file ending in ".obj".

(d) **The Linker**
The linker combines the program's object code files with any required libraries to create the finished executable code. This output is stored as a.out in case of Linux/UNIX C++ compiler and ". EXE" for the Visual C++ compiler or Borland Turbo C++ compiler.

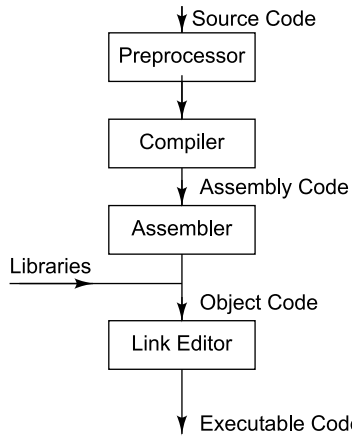


Fig. 2.1 The Compilation Process

2.8.2 Types of Program Errors

The following section deals with the different types of errors and warning messages during the compilation and execution of a program. Errors are caused by syntactical mistakes in source code such as typographical errors, missing semicolons and other kinds of faulty constructions. In general, programming errors can be classified into two types, namely, compile time errors and runtime errors.

(a) Compile Time Errors The compile time errors are caused due to the improper use of C++ syntax and semantics of the language. All syntax errors and some of the semantic errors (the static semantic errors) are detected by the compiler during the compilation stage. The C++ compiler generates a message indicating the type of error and the position in the C++ source file where the error has occurred. It is to be noted that the actual error could have occurred before the position signalled by the compiler.

(b) Run Time Errors Run time errors are another class of errors which are not identified during compilation time. These errors are caused by dynamic semantic errors and logical errors in a program that cannot be detected by the compiler during the debugging stage. The program is compiled and executed but does not generate the required result.

It is the responsibility of the programmer to find and fix the run time errors if any, in a program to get the desired output. There is a difference between a warning and an error. A warning is a message the compiler prints when it discovers a potential problem in the source code. An error is a mistake in the syntax that prevents the compiler from finishing its job.

(c) Warnings Warnings are frequently caused by missing declarations, values of inappropriate types and various kinds of improper constructions. Despite the warning, the program's source code is syntactically correct, so these types of problems do not prevent the compiler from creating a finished code file. However, that code might not run correctly. Warnings are sometimes called compile-time errors.

2.9 STAGES OF PROGRAM DEVELOPMENT (FIG. 2.2)

A little knowledge is necessary before one can write and compile programs on any computer system. Every programmer goes through the following three-step cycle during a program development:

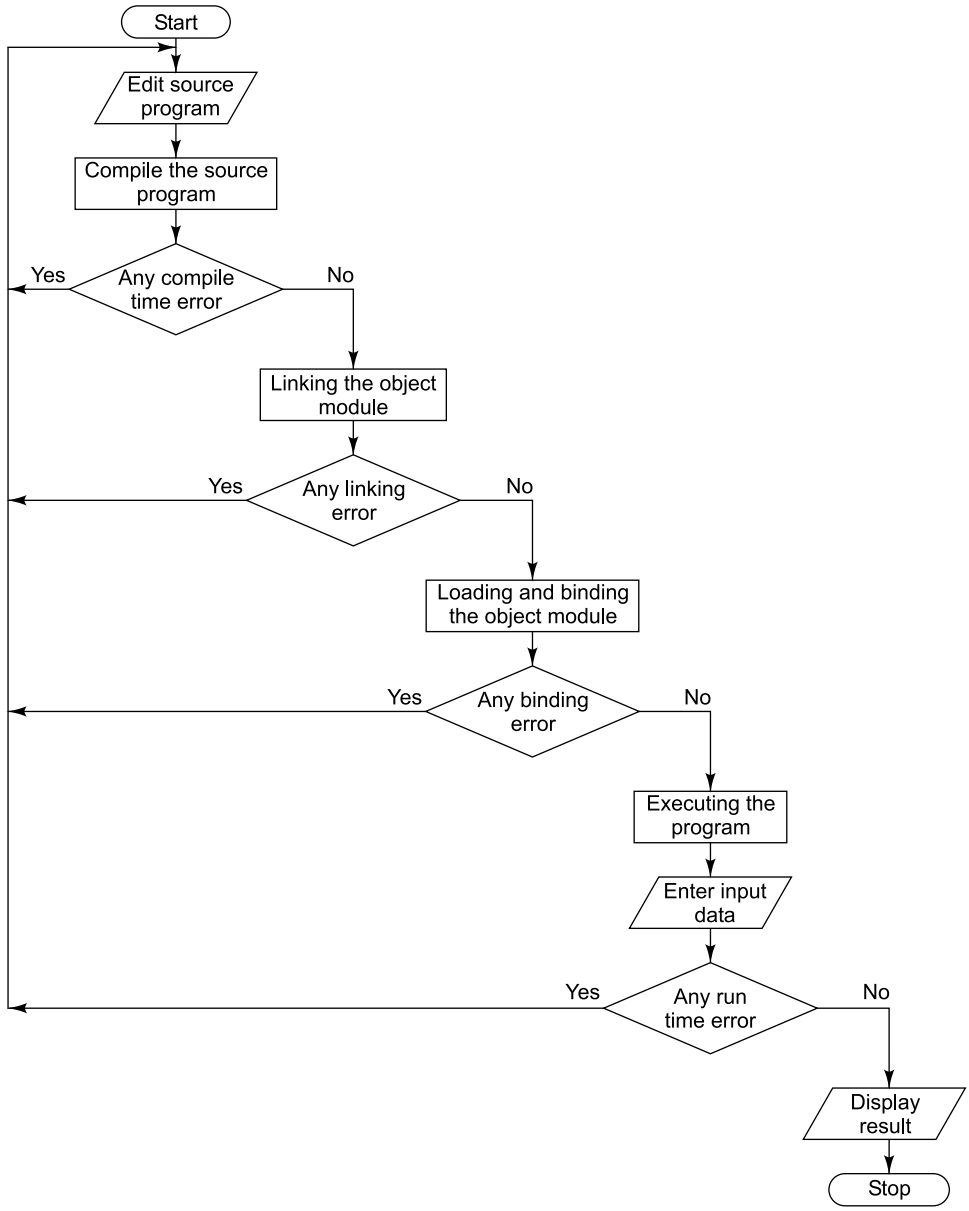


Fig. 2.2 *Stages of Program Development*

1. Writing the program into a file
2. Compiling the program
3. Running the program

During program development, the programmer may repeat this cycle many times, refining, testing and debugging a program until a satisfactory result is achieved.

2.10 COMPILING GNU GCC/G++ IN LINUX

Gcc stands for gnu compiler collection. Gnu is a type of licence for free, open source software. Majority of gnu softwares are for unix-based systems. A compiler is a system software used to convert source code into a file that the computer can execute. So gcc refers to a collection of Unix-based, free softwares which convert source code into machine code.

(1) *Editing the Source Program*

Writing, editing and entering a program into a computer system and storing it as a source file is called the preparation of the program text. Linux supports vi (Visual Editor) for editing a C++ program. The type of file assumed when specifying a C++ source files depends on its extension, which must be one of the following lists:

.C
.cc
.cxx
.cpp and
.c++

For example,

```
$ vi  rst.cpp
```

(2) *Compiling the Source Program*

The GNU gcc/g++ compiler can perform preprocessing, compilation, assembly and linking of a project from a single call to gcc/g++. Its format is:

```
$ gcc  rst.c   (for C program)
```

or

```
$ g++  rst.cpp   (for C++ specific compilation)
```

At this point if there are errors in the source program, the compiler will show them on the screen. One should correct these errors by opening the program in the text editor, namely, vi editor.

After the errors have been corrected, one may use the same gcc or g++ command to compile it again. If there are no errors in the source code, the compiler will return the prompt without any error messages. This means that the source code has been compiled into a separate executable file which by default is named as a.out.

(3) *Running the Compiled Program*

To run the code by typing in a.out at the prompt as follows:

```
$ ./a.out
```

The most common way of calling g++ to compile a single source file in C++ is:

```
$ g++ source le -o exec le
```

where source file is the C++ source file to compile and execfile is the name of the output file, generally the executable file, which must always be preceded by the `-o` option.

For example,

```
$ g++  rst.cpp -o  rst
```

Instead of `a.out`, g++ creates an executable named “first” and one can execute the file in the following way:

```
$./ rst
```

To know more about the GNU g++ compiler options, one can use the Linux g++ manual. The command is as follows:

```
$ man g++
```

2.11 COMPILING C/C++ PROGRAM IN UNIX

Just as C++ is a superset of C, the C++ compilers are very similar to C compilers in that their options are usually a superset of C compiler options. The basic compiling information about C is also applicable to C++, with the following exceptions:

CC (upper case), g++ and gcc are all C++ compiler commands on the UNIX systems that provide C++. Source filename extension conventions are compiler-dependent. Extensions include:

```
.C (upper case)
.c (lower case)
.cxx
.cpp
.cc
.c++
```

(a) Writing the Program The easiest way to enter a source program is using a text editor like vi, emacs or xedit. To edit a file called `rst.c` using `vi` as

```
$ vi  rst.cpp
```

(b) Compiling the Program The C++ compiler is invoked with CC (upper-case), g++ or gcc. There are additional compiler options specific to C++.

For C program compilation, one of the following compiler commands is used:

```
$ cc  rst.c
```

or

```
$ gcc  rst.c
```

For C++ specific compilation, one of the following compiler commands is invoked:

```
$ CC  rst.cpp
```

or

```
$ g++  rst.cpp
```

(c) Running the Program To run a program under UNIX, the following command is used:

```
$./a.out
```

In Unix systems, any file can be labelled as an executable, and it is common to either use the .o extension, or to have no extension at all.

```
$ g++ helloworld.cpp -o helloworld
```

Now, instead of `a.out`, `g++` creates an executable named "helloworld"

```
$./helloworld
```

To know more about the `cc/CC/gcc/g++` compiler options, one can use the UNIX manual. The command is as follows:

```
$ man CC
```

2.12 BUILDING C++ UNDER MICROSOFT .NET PLATFORM

This section explains how to edit, compile and build a C++ program under Visual Studio .NET Framework which is one of the most widely used platforms for learning and developing ANSI C++ programs.

(1) *Visual Studio .NET Framework (Fig. 2.3)* Visual Studio supports the Microsoft .NET Framework, which provides the Common Language Runtime (CLR) and unified programming classes. Visual Studio .NET is the tool for rapidly building high performance desktop applications and Web and ASP applications. It supports the following programming languages:

- Visual C++
- Visual Basic
- Visual C#
- Visual J#

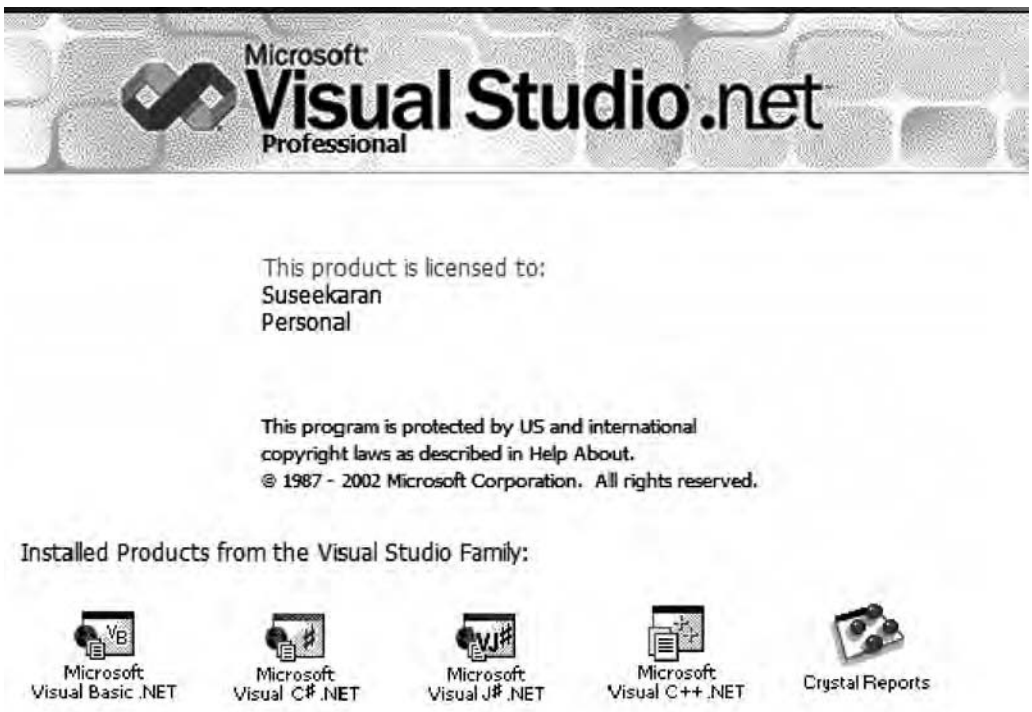


Fig. 2.3 *Visual Studio .NET Framework*

The .NET Framework is a multi-language environment for building, deploying, and running XML Web services and applications. It consists of three main parts:

- Common Language Runtime
- Unified programming classes
- ASP.NET

The framework provides developers with a unified, object-oriented, hierarchical, and extensible set of class libraries (APIs).

(2) *Microsoft Visual C++ .NET (Fig. 2.4)* It is well known that Microsoft Visual C++ .NET 2003 provides the dynamic development environment for creating Microsoft Windows-based and Microsoft NET-based applications, dynamic Web applications, and XML Web services using the C++ development language. Visual C++ .NET includes the industry-standard Active Template Library (ATL) and Microsoft Foundation Class (MFC) libraries, advanced language extensions, and powerful integrated development environment (IDE) features that enable developers to edit and debug source code efficiently.

It provides developers with a proven, object-oriented language for building powerful and performance-conscious applications. With advanced template features, low-level platform access, and an optimizing compiler, Visual C++ .NET delivers superior functionality for generating robust applications and components. The product enables developers to build a wide variety of solutions, including Web applications, smart-client Microsoft Windows-based applications, and solutions for thin-client and smart-client mobile devices. C++ is the world's most popular system-level language, and Visual C++ .NET 2003 gives developers a world-class tool with which to build software.

The following steps are used to create, edit and build a C++ program under Microsoft Visual .NET studio. To select the Visual Studio .NET from the Start Menu, Click Start Menu and select All Programs -> Microsoft Visual Studio .NET 2003 –> Press Microsoft Visual Studio .NET 2003.

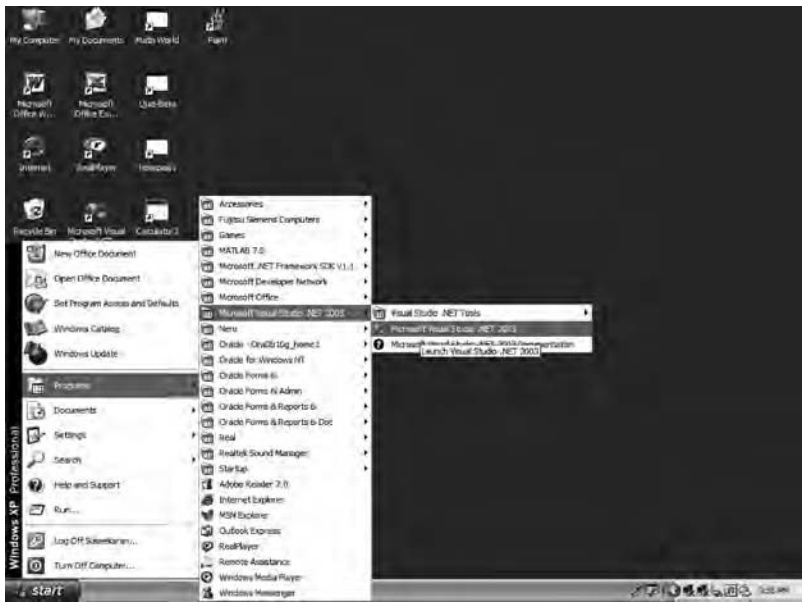


Fig. 2.4 *Microsoft Visual C++ .NET*

(3) *Start Page (Fig. 2.5)* The Start Page has been re-designed for this release. One can still set the user preferences for IDE behaviour and access new or existing projects, but with a user interface designed to be easier to navigate. Both the My Profile and Project sections now have their own tabs. The Online Resources tab now contains useful Microsoft related online developer resources.

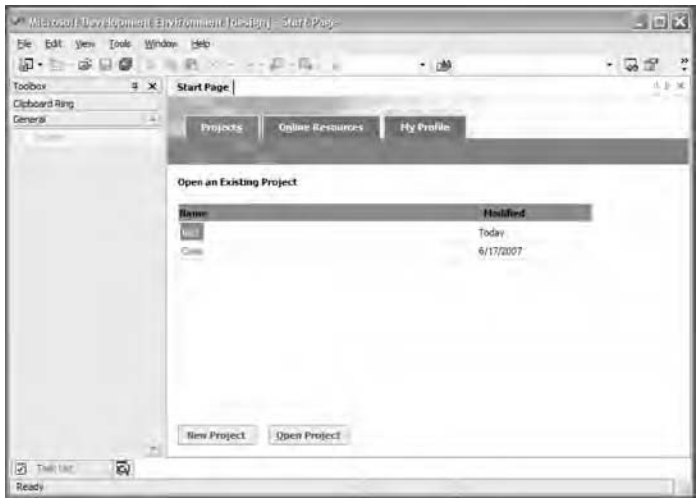


Fig. 2.5 *Microsoft Development Environment Design — Start Page*

(4) *Integrated Development Environment (IDE)* In the Menu bar, select File → New → Project

`Visual C++ Projects`

An application wizard provides a user interface that is used to create a project, modelled after a project template, and generate source files and directories for applications.

The wizard provides program structure, basic menus, toolbars, icons, and appropriate #include statements. Visual C++ application wizards work in conjunction with application frameworks and libraries to create starter programs for the user.

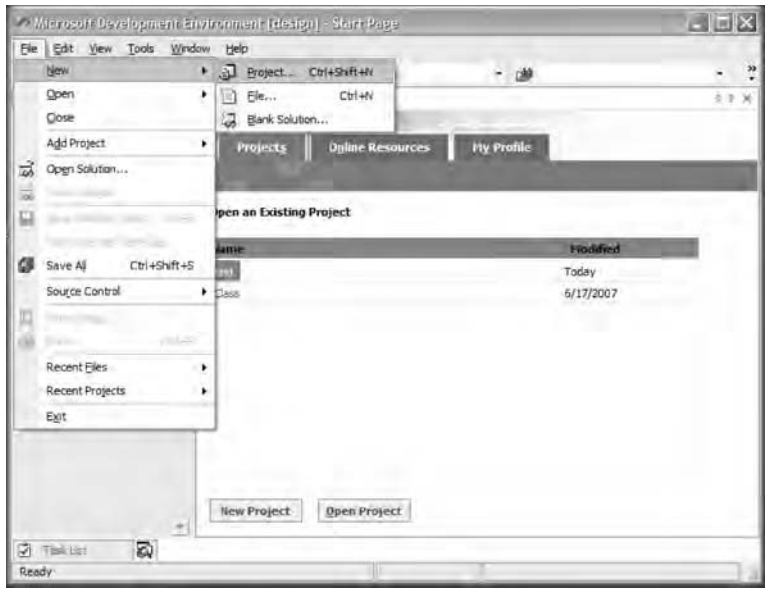


Fig. 2.6 *Creating a Project with a Visual C++ Application Wizard*

(5) *Creating and Managing Projects (Fig. 2.6)* Every type of Visual C++ project has an application wizard that helps the user generate new projects quickly and easily, modelled from the project template.

(a) Project types Visual Studio contains a project template or application wizard for the following project types. Each wizard helps to create projects:

- ASP.NET Web Service Template
- Class Library Template
- Console Application Template
- Empty Project Template
- Windows Control Library Template
- Windows Forms Application Template
- Windows Service Template

To open an application wizard, the New Project dialog box has to be used to specify the project properties like the name, or the directory and solution where your project will reside.

To open a Visual C++ application wizard

1. On the File menu, click New, and then click Project. The New Project dialog box appears.
2. In the Project Types pane, select the Visual C++ Projects folder. An icon for every type of C++ project appears in the Templates pane.
3. In the Templates pane, select an icon to choose a project type. A message appears under both panes indicating the type of project the user is going to create.
4. Specify your project properties, or skip this step to use Visual Studio default project properties.
5. Click OK, and the wizard for your project type opens.

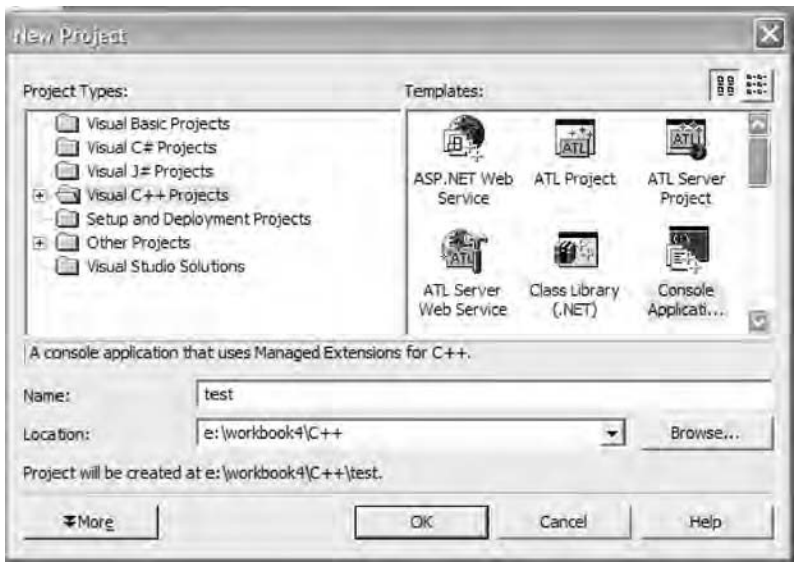


Fig. 2.7 *Creating a Project with a Console Application Template*

(b) Console application template (Fig. 2.7) The Console Application project template adds the necessary items needed to create a console application. Console applications are typically designed without a graphical user interface and are compiled into a stand-alone executable file. A console application is run from the command line with input and output information being exchanged between the command prompt and the running application.

As information can be written to and read from the console window, this makes the console application a great way to learn new programming techniques without having to be concerned with the user interface.

The template automatically adds the essential project references and files to use as a starting point for your application. Header files "Stdafx.h" — Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj

(6) *Adding and Removing Solution Items (Fig. 2.8)*

(a) Solution Explorer To efficiently manage the items that are required for development such as references, data connections, folders, and files, Visual Studio .NET provides two containers: solutions and projects. An interface for viewing and managing these containers and their associated items, Solution Explorer, is provided as part of the integrated development environment (IDE).

Solution Explorer provides the user with an organised view of his projects and their files as well as ready access to the commands that pertain to them. A toolbar associated with this window offers commonly used commands for the item you highlight in the list. To access Solution Explorer, select Solution Explorer on the View menu.

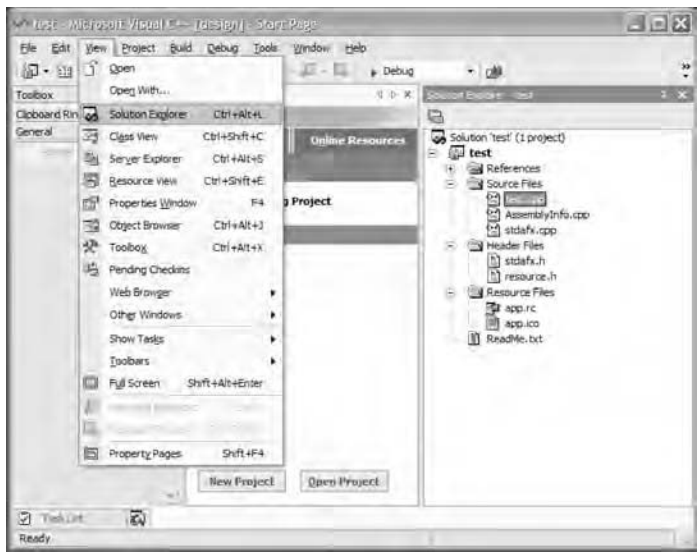


Fig. 2.8 *Selecting the Solution Explorer on the View Menu*

(b) Removing Solution Items (Fig 2.9) One can remove an item from the Solution Items folder. Removing is not the same as permanently deleting an item. Removing takes away the item's association with the solution. The file that represents the former item still remains on disk.

To remove a solution item

1. In Solution Explorer, select the item the user wants to remove.
2. On the Edit menu, choose Remove.
3. One can re-add the item as long as the file for the item still exists.

(c) Adding Solution Items To add a new solution item

1. In Solution Explorer, select the Solution node.
2. On the Project menu, choose Add New Item.
3. From Add New Solution Item, choose a template.
4. Choose Open to add the item to the Solution Items folder.

To add an existing item to a solution

1. In Solution Explorer, select the solution.

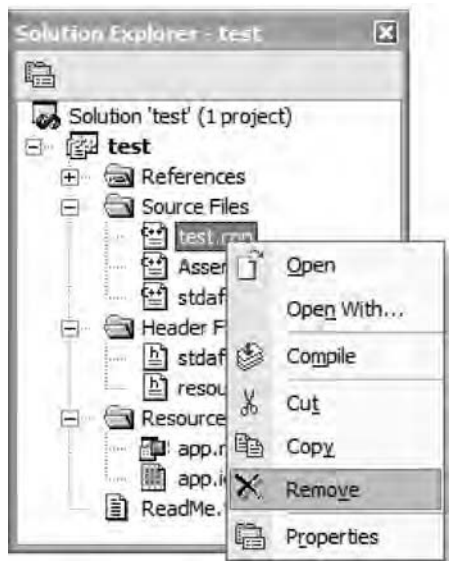


Fig. 2.9 *Removing Solution Items*

2. On the Project menu, choose Add Existing Item.
3. From Add New Solution Item, choose the item the user wants to add.
4. Choose Open to add the item to the Solution Items folder.

(7) *Adding New Project Items (Fig. 2.10(a) & (b))* Adding a project item is one way to extend the functionality of an application. Examples of project items include HTML pages, Class files, Web Services, ASP pages, Dataset files, and Style sheets. The types of files that the user can add to a project are determined by the project template used to create it.

To add a new project item

1. In Solution Explorer, select a target project.
2. On the Project menu, select Add New Item.
3. Select a Category in the left pane.
4. Select an item Template in the right pane.
5. Select Open.

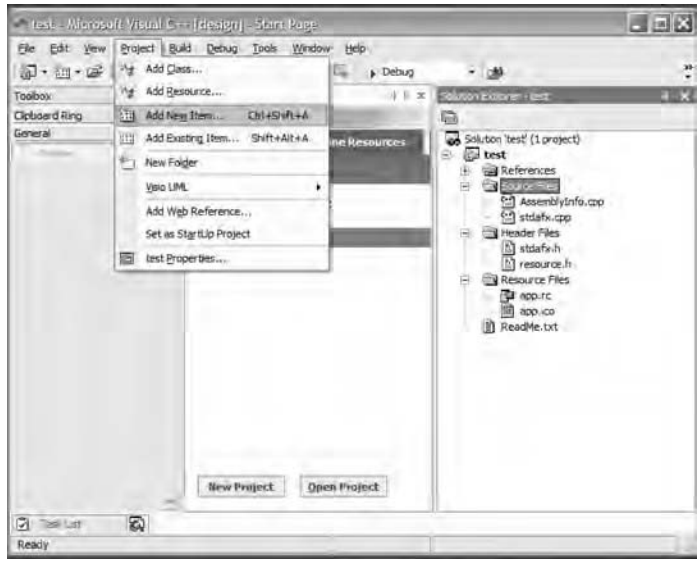


Fig. 2.10(a) *Adding New Project Item*

Fig. 2.10(b) *Adding New Project Item*

(8) *Editing and Saving C++ Programs (Fig. 2.11a)*

(a) Editing Start entering a C++ program on the editor pane.

For example,

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main()
{
        cout << "this is a test program by Ravich \n";
        return 0;
}
```

(b) Saving (Fig. 2.11b) Save a file, On the File menu, click Save First.cpp

Save a copy of a file

1. On the File menu, click Save As.
2. In the File name box, enter a new name for the file.
3. Click Save.

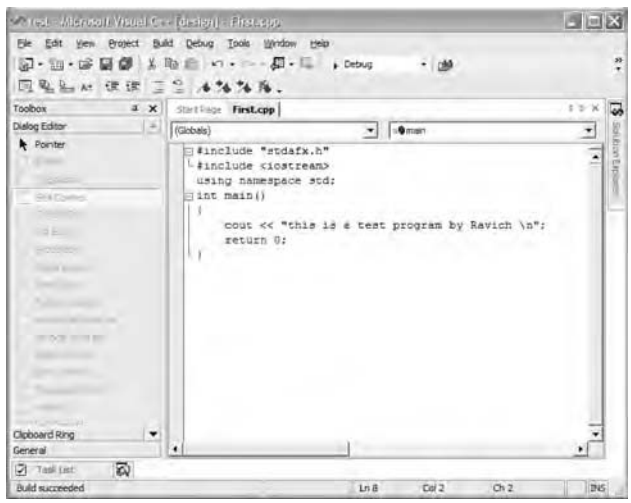


Fig. 2.11(a) *Editing a C++ Program on the Editor Pane*

(9) *Preparing and Managing Builds (Fig. 2.12)* Visual Studio .NET offers a variety of ways to help the user to organise files which are required for a solution or a project.

To build or rebuild a single project

1. In Solution Explorer, select or open the desired project.
2. On the Build menu, choose Build [Project Name] or Rebuild [Project Name].

Note: "Cleaning" a solution or project deletes any intermediate and output files, leaving only the project and component files, from which new instances of the intermediate and output files can then be built.

To build or rebuild an entire solution, on the Menu bar, select Build –> Build Solution.

On the Build menu, choose Build Solution or Rebuild Solution. Choose Build or Build Solution to compile only those project files and components that have changed since the last build. Choose Rebuild Solution to "clean" the solution first, and then build all project files and components.

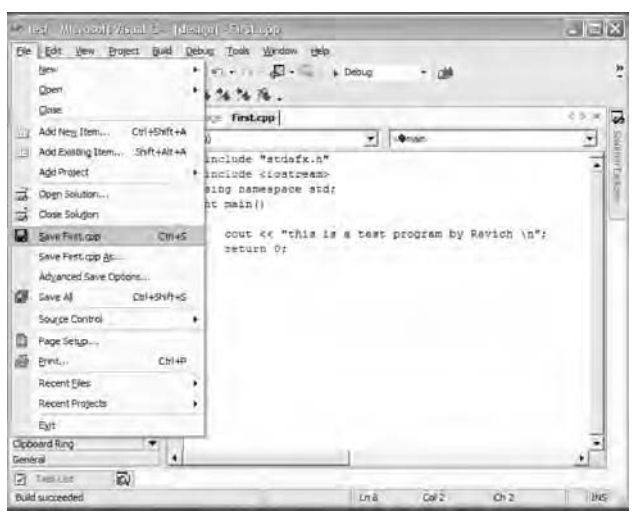


Fig. 2.11(b) *Saving a C++ Program*

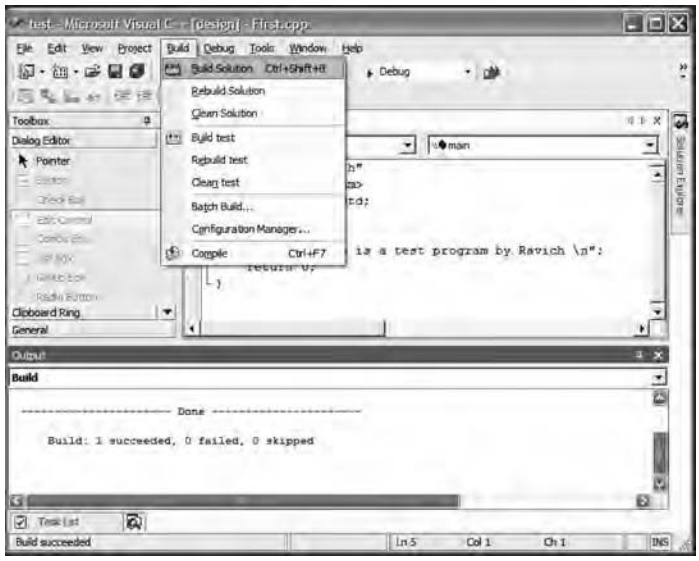


Fig. 2.12 *Build the C++ Program*

(10) *Executing and Debugging Visual C++ Console Applications (Fig. 2.13)* To run the program, on the Menu bar, select Debug and select the option "Start Without Debugging". Results will be displayed on the new Output window (Fig. 2.14).

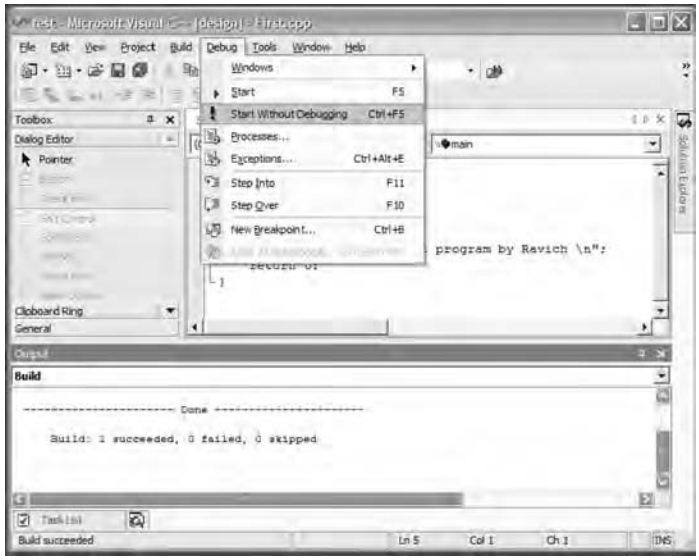


Fig. 2.13 *Run the C++ Program*

A solution and its individual projects are typically built and tested in a Debug build. Developers will compile a Debug build repeatedly, at each step in their development process. Debugging is a two-step process. First, compile-time errors are corrected. These errors can include incorrect syntax, misspelled

keywords, and type mismatches. Next, the debugger is used to detect and correct such problems as logic errors and semantic errors that are detected at run-time.

Visual Studio provides additional considerations for debugging Visual C++ console applications. These considerations include, specifying command-line arguments, starting the application from the command prompt rather than from Visual Studio, directing output to the Output window, and troubleshooting Console window behaviour.

To debug a Visual C++ console application

1. Open the project in Visual Studio
2. Choose Start without Debugging (Ctrl + F5)

Visual Studio automatically creates required settings for the Debug and Release configurations.

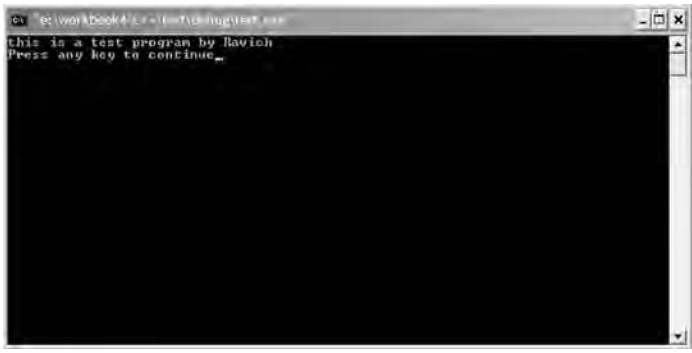


Fig. 2.14 *Output Window with Results*

(11) *Quit from the Visual Studio.NET Environment* To exit from the Visual studio .NET environment, select Exit option from the File menu (Fig. 2.15). Visual studio closes all files and the main control will be transferred to Windows OS.

1. On the File menu, click Exit.

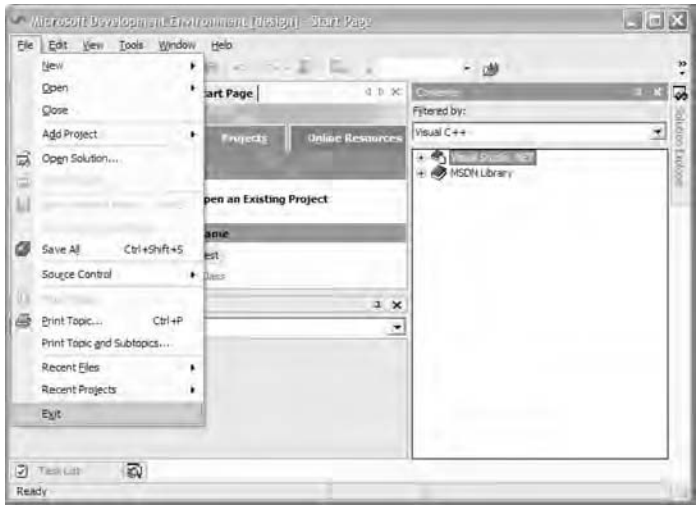


Fig. 2.15 *Exit from the Visual Studio .NET Environment*

REVIEW QUESTIONS

1. Summarise the history of ANSI C++ language.
2. Explain the importance of standardising C++ compiler.
3. What are the latest addenda to ANSI/ISO C++ language?
4. Explain the pros and cons of C++ versus C.
5. List the different versions of C++ compilers available in the market today.
6. What is the difference between compiling and debugging of C++ programs?
7. Explain the following terms with respect to C++ language:
   (a) Preprocessor (b) Compiler
   (c) Assembler (d) Linker
8. Elucidate the various stages of program development in C++.
9. Explain the difference between compile time and run time errors.
10. Summarise the commands used for compiling GNU gcc/g++ in Linux OS.
11. List the steps involved in compiling C/C++ programs in UNIX OS.
12. Elucidate the various steps involved in building Visual C++ programs under Visual studio .NET environment.

Data Types, Operators and Expressions

Chapter 3

This chapter deals with data types, literals, constants, variables and some statements used in C++ language. The entire C++ data types and operators are discussed in a rudimentary manner with various illustrations. Elementary definitions and concepts of C++ program are stressed in this chapter and the subsequent chapters reserved for detailing all the features of C++ language.

3.1 IDENTIFIERS AND KEYWORDS

The program elements are the basic functional blocks of a C++ program. They consist of numbers, identifiers, expressions and statements. C++ identifiers are the various program entities such as variables, constants, data types, functions structs, unions and classes.

Identifiers are used in a program not only to declare basic program elements like constants or variables but also to define the name of a function. Using the combination of the following C++ character sets, one can generate a program element such as user defined identifiers and statements.

The character sets used in ANSI/ISO C++ are:

```
Lower case : a..z
Upper case : A.. Z
Digits : 0..9
Special characters : +  -  *  /  =  ++ -- .  ,  :
                     ;  '  <   <=  >  >=  == !=
                     . (   ) {  }   [  ]  ^ ! | &
                       and blank space
```

In C++, the lower case and upper case letters (alphabets) are distinct. Hence, there is restriction on the use of upper case or lower case letters while writing identifiers and in fact, mixing of lower case and upper case letters is also allowed.

For example, the following identifiers will be treated separately because the lower and upper case letters are used differently.

```
paybill
PAYbill
PAYBILL
```

In general, identifiers can be classified into two types: user defined identifier and built-in identifier or keywords. Built-in identifiers or keywords are meant for intended purpose in the compiler. User defined identifier are the name of the program elements.

The following rules are used to write a user defined identifier in C++:

(1) An identifier consists of a sequence of letters and digits. However, the first character of an identifier must be an alphabetic character, either uppercase or lowercase, or an underscore (_) character.
(2) Identifiers are case sensitive, `fileName` is different from `FileName`.
(3) User defined identifiers cannot be exactly the same spelling as that of keywords or reserved words. The keywords are the standard or predefined meaning for intended purpose for developing program modules.
(4) There should not be spaces between the characters.
(5) The ANSI/ISO C++ does not impose any limit on the number of characters in an identifier.
(6) Use of two sequential underscore characters (_ _) at the beginning of an identifier, or a single leading underscore (_) followed by a capital letter, is reserved for C++ implementations in all scopes. User should avoid making such a type of identifiers in a program because of possible conflicts with current or future reserved identifiers.

Some of the Valid Identifiers

```
hello_world
pay12
_abc
DEFINED
pay_bill_my_address
```

Some of the Invalid Identifiers

`5thcross` – the first character must be an alphabet
`while` – reserved word and it cannot be used as a user defined variable
`pay bill` – blank spaces are not allowed

3.1.1 Reserved Words (Keywords)

The following are the list of keywords that are commonly used in C++ and enforced by the ANSI C++ committee. They have a standard and predefined meaning in C++ and are used only for their intended purpose and these words must not be used for user defined variables in the C++ program. Total keywords used in ANSI/ISO C++ are 73. The reserved words can be classified into six groups, namely, (i) declaration words (ii) statement words (iii) storage allocation identifiers (iv) C++ specific Keywords (v) New keywords added in ANSI C++ (vi) Alternative representations of operators in ANSI C++.

(i) Variable Declaration Words

```
char        int         sizeof
double      long        typedef
enum        short       unsigned
float       signed
```

(ii) Statements Words

```
break       else        struct
case        for         switch
continue    goto        union
default     if          while
do          return
```

(iii) Storage Allocation Identifier

```
auto          static
const         void
extern        register
volatile
```

(iv) C++ Specific Keywords

In order to add features to C, a number of new keywords were created for C++. In general, C++ provides more language features and fewer restrictions than ANSI C. The C++ language consists of both ANSI C keywords along with the following new additional or reserved words. The new keywords are:

```
asm           friend          protected       throw
catch         inline          public          class
new           template        virtual         operator
this          delete          private         try
```

(v) New Keywords Added in ANSI C++

```
typeid        namespace       wchar_t
typename      static_cast     const_cast
bool          using           true
dynamic_cast  explicit        reinterpret_cast
mutable       false
```

(vi) Alternative Representations of Operators in ANSI C++

```
and           compl           or_eq
and_eq        not             xor
bitand        not_eq          xor_eq
bitor         or
```

3.1.2 Standard Identifiers

Standard identifiers are a set of built-in constants, variable names and functions. They are not user defined identifiers but are called as standard identifiers as they are built in the C++ compiler for predefined purpose in the user program. Standard identifiers are very useful for developing programs in an easy manner and are mainly used for mathematical library, file routines, etc.

There are differences between the standard identifiers and the keywords. Keywords are used to construct a C++ program whereas standard identifiers assist the user to develop an efficient program in an easy manner. Without standard identifiers, one can develop a C++ program but without keywords one cannot do anything. Keywords are language dependent and hence, these must be same for all versions of the C++ compilers, whereas standard identifiers may vary from one compiler version to another and need not be the same.

Some of the examples for the standard identifiers are:

```
cin           min           sin
cout          max           cos
NULL          open          pow
EOF           close         log
```

3.2 DATA TYPES

The data type of a variable is important because it determines the operations that are allowed and the range of values that can be stored. C++ defines several types of data and each type has unique characteristics. Because data types differ, all variables must be declared prior to their use and a variable declaration always include a type specifier. The compiler requires this information in order to generate correct code. In C++, there is no concept of a "type less" variable. The built-in data types are integers, characters, floating point values and Boolean values. The core of the C++ type system are the seven basic data types shown here:

Type	Meaning
char	character
wchar_t	wide character
int	integer
float	floating point
double	double floating point
bool	Boolean
void	valueless

C++ allows certain of the basic types to have modifiers preceding them. A modifier alters the meaning of the base type so that it more precisely fits the need of various situtations. The data type modifiers are listed here:

- signed
- unsigned
- long
- short

The modifiers `signed`, `unsigned`, `long` and `short` can be applied to `int`. The modifiers `signed` and `unsigned` can be applied to the `char` type. The type `double` can be modified by `long`.

3.3 C++ SIMPLE DATA TYPES

3.3.1 Bool

The bool type is designed to hold only two types of values: `true` or `false`.

3.3.2 Char

The `char` data type is used to represent and store characters. Internally, every character is represented by a small integer. What characters are available and how they are represented internally depends on the machine on which the program runs. The most common character sets are ASCII (American Standard Code for Information Interchange) code. ASCII is the character set used on most personal micro and minicomputers as well as several large and mainframes.

There are 128 ASCII characters. That means ASCII characters are 7-bits (values between 0 to 127) but are usually put into an 8-bit byte. The char data type can be classified into the following four types:

```
char  ---|-- plain char
         |
         |-- wchar_t
         |
         |-- signed char
         |
         |-- unsigned char
```

(i) Plain Char Any character belonging to the ASCII character set is considered as a character data type whose maximum size is 8 bits long. The keyword char is used to represent the character data type in C++. Character constants and literals are always represented within single quotes.

(ii) Wchar_t wchar_t is meant for wide characters for storing unicode.

(iii) Signed Char A plain char is always signed. Therefore, signed char and plain char are referred to the same data type whose minimum range is from –128 to 127. The signed char types are simply more explicit synonyms used in a program.

(iv) Unsigned Char The most significant bit of a number is referred as a sign bit when a number is represented in the binary form. In the case of unsigned char, the sign bit is used to store for the character

representation rather than for the sign. Therefore, minimum range of unsigned char is from 0 to 255.

The plain char, signed char and unsigned char are three distinct types. A char, a signed char and an unsigned char occupy the same amount of memory space. The classification of the char data type is dependent on the version of the C++ compiler.

3.3.3 Int

int data type can be classified into the following three types:

```
int ---|-- plain int
       |
       |-- signed int
       |
       |-- unsigned int
```

(i) Plain Int The plain `int` is a standard int data type whose minimum range is from –32, 768 to 32, 767.

(ii) Signed Int A plain int is always signed. Therefore, signed int and plain int are referred to the same data type whose minimum range is from –32, 768 to 32, 767. The signed int types are simply more explicit synonyms used in a program.

(iii) Unsigned Int The most significant bit of a number is referred as a sign bit when a number is represented in the binary form. In the case of unsigned int, the sign bit is used to store for the number representation rather than for the sign. The unsigned int is used to represent and assign only the positive numerals in a program. Therefore, minimum range of unsigned int is from 0 to 65, 535.

3.3.4 Short

Short data type can be classified into the following four types:

```
short ---|-- short
         |
         |-- short int
         |
         |-- signed short
         |
         |-- unsigned short
```

(i) Short The short data type is used to store an integer data whose minimum size is larger than or equal to char and shorter than or equal to type int.

(ii) Short Int The short int data type is the same as that of int data type, whose minimum range is from –32, 768 to +32, 767.

(iii) Signed Short The signed short data type is the same as that of short data type, whose minimum size is larger than or equal to char and shorter than or equal to type int.

(iv) Unsigned Short The unsigned short data type is the same as that of short int without sign bit. In the case of unsigned short, the sign bit is used to store for the number representation rather than for the sign. The unsigned short is used to represent and assign only the positive numerals in a program. Therefore, minimum range of unsigned short is from 0 to 65, 535.

3.3.5 Long

The long data type can be classified into the following three types:

```
long  ---|-- long
         |
         |-- signed long
         |
         |-- unsigned long
```

(i) Long A long int data type can be referred to as plain long.

(ii) Signed Long The signed long data type is the same as long integer whose minimum range is from –2, 147, 483, 648 to +2, 147, 483, 647. The signed long int data types are simply more explicit synonyms used in a program.

(iii) Unsigned Long The unsigned long data type is used to represent and assign only the positive numerals in a program. Therefore, minimum range of unsigned long int is from 0 to 4, 294, 967, 295.

The number of bits used to represent short int, int and long int is implementation dependent so long as the minimum ranges maintained. The keyword short is a synonym for short int, unsigned for unsigned int and signed for signed int.

The typical integer data types and its sizes in C++ are given in the following Table 3.1.

Table 3.1

Data types	*Size in bytes*
char	1
short	2
int	2 or 4
long	4 or 8

3.3.6 Float

The numbers which are stored in the form of floating point representation with binary mantissa and exponent are known as *floating point numbers* or *real numbers*. They can be declared as 'float' in C++ whose maximum size is a rational number approximately between –0.17e38 and 0.17e38. The smallest value other than 0 that can be represented is 0.29e-38 in C++. The real data type can be classified into the following three types:

```
real or        ---|-- float
floating point    |
                  |-- double
                  |
                  |-- long double
```

(i) Float A float provides at least 6 significant digits and usually requires 32 bits of storage. The minimum range of float data type is from –3.4e38 to +3.4e38, with six digits of precision.

(ii) Double The keyword double is used to represent double precision floating point numbers in C++. The size of 'double' is a rational number in the same range as float and is stored in the form of floating point representation with binary mantissa and exponent.

A double provides at least 10 significant digits usually requires 64 bits of storage. The minimum range of double data type is from –1.7e308 to +1.7e308, with ten digits of precision.

(iii) Long Double

A long double potentially provides even more significant digits and larger range of values. The minimum range of long double data type is from –1.7e4932 to +1.7e4932, with ten digits of precision. However, many implementations treat doubles and long doubles as synonyms.

We use floats when we need to save storage or want to avoid the overhead of double precision operations. We use doubles when we need more significant digits and we are less concerned with storage. We use long doubles when our implementation provides even more significant digits or a wider range of values for them.

The typical floating point types and its sizes in C++ are given in the following Table 3.2.

Table 3.2

Data types	*Size in bytes*
float	4
double	8
long double	12 or 16

The following Table 3.3 shows the minimum range of each type as specified by the ANSI/ISO C++ standard:

Table 3.3 *Minimal range of each type as specified by the ANSI/ISO C++ standard*

Type	*Minimal range*
bool	holds true (non-zero) or false (zero) value
char	–128 to 127
wchar_t	wide characters such as unicode
signed char	–128 to 127
unsigned char	0 to 255
int	–32, 768 to 32, 767
signed int	same as int
unsigned int	0 to 65, 535
short int	–32, 768 to 32, 767
signed short int	same as short int
unsigned short int	0 to 65, 535
long int	–2, 147, 483, 648 to 2, 147, 483, 647
signed long int	same as long int
unsigned long int	0 to 4, 294, 967, 295
float	–3.4e38 to +3.4e38
double	–1.7e308 to +1.7e308
long double	–1.7e4932 to +1.7e4932
void	void type

3.4 LITERALS

The lexical class of constants in ISO C is called literals in ANSI/ISO C++ and can be classified into five types, namely, integers, floating point numbers, characters, strings and booleans.

```
Types of literals ---|-- integer literal
                     |
                     |-- floating point literal
                     |
                     |-- character literal
```

```
                    |
                    |-- string literal
                    |
                    |-- boolean literal
```

3.4.1 Integer Literals

An integer literal is a sequence of digits that has no period or exponent part. Integer literals may be specified in decimal, octal or hexadecimal notation.

```
Types of integer literals ---|-- decimal integer literal
                             |
                             |-- octal integer literal
                             |
                             |-- hexadecimal integer literal
```

(a) Decimal Integer Literal A decimal integer literal (base ten) begins with a digit other than 0 and consists of a sequence of decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). In other words, a decimal integer literal consists of a nonempty sequence of digits, the first of which is not 0. Some examples of decimal integer literal are given below:

12389	–234567	999908
765908	886666	–123

(b) Octal Integer Literal

An octal integer literal (base eight) begins with the digit 0 and consists of a sequence of octal digits (0, 1, 2, 3, 4, 5, 6, 7). Some of the examples for the octal integer literal are given below:

01117	01234	076565
07654	07777	023456

There is a question as to whether "0" is decimal or octal, but it does not matter in practice.

(c) Hexadecimal Integer Literal

An hexadecimal integer literal (base sixteen) begins with 0x or 0X and consists of a sequence of hexadecimal digits, which include the decimal digits and the letters 'a' through 'f' (or 'A' through 'F') with decimal values 10 through 15. Some of the examples for the hexadecimal integer literal are given below:

0x12345	0x3455	0x121ff
0X6ab	0Xabc123	0XFFF

(d) Long Integer Literals Any integer literal may be immediately followed by the one of the letters 'l' or 'L' to indicate a literal of type long.

decimal long literal = digit + long marker
octal long literal = 0 (octal) + long marker
hexadecimal long literal = hex-marker (hexadigit) + long marker

Some of the examples for the long integer literals

871056L
056523L
0x5464abcL
0X234affL

3.4.2 Floating Point Literals

Floating point literal may be written with a decimal point, a signed exponent or both. A floating point literal is always interpreted to be in decimal radix. Real numbers can be classified into two forms (i) standard forms and (ii) exponential or scientific form.

```
real numbers ---|-- standard form
                |
                |-- exponential (scientific) form
```

(i) Standard Form Without normalising, if a real number is represented then it is called a standard form. Representaion of the real numbers so far discussed are called the *standard form*. Such a representation have a few applications in the computer usage as very large and very small numbers cannot be easily accommodated or represented.

(ii) Exponential (Scientific) Form Very large and very small numbers can be represented very easily. In C++ using scientific notation as these numbers are stored in the computer memory in the normalised form. The scientific notation is also called as exponential notation or floating point expression. In C++, to represent real numbers using floating point notation, the letter e (alternately E) is used to realise the exponent part or 10 raised to the power.

The number before the letter e (E) must be an integer or real constant, with or without a sign or decimal point. If a decimal point is included, it must be preceded and followed by a digit. The number after the letter e (E) should be an integer value (preferably signed). This number is called the exponent. In practice, the letter E is used to indicate that the value following it is the exponent.

For example,

	0.00016435	is	1.6435×10^{-4}	(conventional)
			1.6435E-4	(C++ method)
and	26472000.0	is	2.6472×10^{7}	
			2.6472E7	

Following are some examples of valid real numbers in exponential form:

1.6356E5
2.1898E+5
0.123E4
1.456E–6
1.34322E–07
–1.23E–6
+1.345E–26
2E10
2E–10

The following real number representations are invalid:

.E5 - a digit is needed to the left of the decimal point
2.678E+1.4 - exponenent part cannot have a decimal part
1.345E +/–5 - two symbols + and – cannot be represented together
1.1E - a digit is needed to the right of the exponenent part
2,345.4E-6 - comma is not permitted

Types of floating point literals in C++ Floating point types come in three sizes:

```
Types of foating point numbers ---|-- float (single precision)
                                  |
                                  |-- double (double precision)
                                  |
                                  |-- long double (extended precision)
```

The exact size of single precision, double precision and extended precision is implementation dependent.

(a) Double literal By default, a floating point literal is of type 'double'. The following are the valid floating point literals of type double.

1.23	–1.e10	3e1
.23	1.23e–10	0.
0.23	1e–3	.0
1.0	0.22E–6	2e+9

(b) Float literal The suffix f or F is used along with a floating point literal in order to represent type float. For example,

3.1456F	–11.453f
2.01f	23.45E3f

(c) Long double literal The suffix l or L is used to represent a floating point literal of type long double. For example,

3.1452L	–11.45e2L
2.01L	2.234E7L

3.4.3 Character Literals

A character literal is one or more characters enclosed in single quotes, as in 'x', optionally preceded by the letter L, as in L 'x'.

```
Types of character literals ---|-- Narrow (ordinary) character literal
                               |
                               |-- Wide character literal
                               |
                               |-- Escape sequence characters
                               |
                               |-- Trigraph sequence characters
```

(a) Narrow (Ordinary) Character Literal A narrow (ordinary) character literal is one or more characters enclosed in single quotes. In other words, a character literal that does not begin with L is an ordinary character literal, also referred to as a narrow-character literal.

An ordinary character literal that contains a single c-char has type char, with value equal to the numerical value of the encoding of the c-char in the execution character set. An ordinary character literal that contains more than one c-char is a multicharacter literal. A multicharacter literal has type int and implementation defined value.

For example,

'a'
'?'
'1'

(b) Wide Character Literal A character literal that begins with the letter L, such as L'x', is a wide-character literal. A wide-character literal has type wchar_t. The value of a wide-character literal containing a single c-char has value equal to the numerical value of the encoding of the c-char in the execution wide-character set. The value of a wide-character literal containing multiple c-chars is implementation defined.

For example,

L'a'
L'?'
L'1'

(c) Escape Sequence Characters The backslash (\) is used to denote non-graphic characters and other special characters for specific operation. These characters are called *escape sequence characters*. Table 3.4 summarises the escape sequence characters that are used in C++:

Table 3.4

Escape sequence	*Character*	*Meaning*
\a	BEL	alert a bell character
\n	NL or LF	newline or line feed
\t	HT	horizontal tab
\b	BS	backspace
\r	CR	carriage return
\f	FF	form feed
\v	VT	vertical tab
\\	\	back slash
\’	’	single quote
\”	”	double quote
\0	NULL	null character
\?	?	question mark
\000	000	octal value
\xhhh	hhh	hexadecimal value

Note that the null character ‘\0’ is to be distinguised from 0 (zero, not the alphabet ‘o’)

(d) Trigraph Sequence Characters Before any other processing takes place, each occurrence of one of the following sequences of three characters (“trigraph sequences” is replaced by the single character indicated in Table 3.5.

Table 3.5

trigraph	*replacement*
??=	#
??/	\
??’	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

For example,

```
??=define arraycheck(a,b) a??(b??) ??!??! b??(a??)
```

becomes

```
#define arraycheck(a,b) a[b] || b[a]
```

Trigraph replacement is done left to right, so that when two sequences which could represent trigraphs overlap, only the first sequence is replaced. Characters that result from trigraph replacement are never part of a subsequent trigraph.

For example,

The sequence "???=" becomes "?=", not "?#".

The sequence "??????????" becomes "???", not "?".

3.4.4 String Literals

A string literal is a sequence of characters surrounded by double quotes, optionally beginning with the letter L, as in "..." or L"...".

```
Types of string literals ---|-- Narrow (ordinary) string literal
                            |
                            |-- Wide string literal
```

(a) Narrow (Ordinary) String Literal A narrow or ordinary string literal is a sequence of characters surrounded by double quotes and that does not begin with L is an ordinary string literal, also referred to as a narrow string literal. An ordinary string literal has type "array of n const char" and static storage duration, where n is the size of the string and is initialised with the given characters.

For example,

"abc"

"this is a test program by Ravich"

"1234"

(b) Wide String Literal A string literal that begins with L, such as L"computer", is a wide string literal. A wide string literal has type "array of n const wchar_t" and has static storage duration, where n is the size of the string and is initialized with the given characters.

The following are for wide string literal representation:

L"abc"

L"this is a test program"

L"1234"

3.4.5 Boolean Literals

The Boolean literals are the keywords `false` and `true`. Such literals have type bool. They are not lvalues. The value of the boolean literal is given below:

- true
- false

3.5 VARIABLES

A variable is the symbolic address of a location in memory where data can be stored. Variables are object of the program elements that may change its contents of the value during program execution.

In some programming languages like FORTRAN one must be cautious in selecting the identifiers for integers and reals as there is a lot of restriction in the use of variable names. Sometimes, the above restriction will make the source code look clumsy. In C++, no such rules are used to identify a variable separately as integer and real in a program, except, the standard rules applicable for forming a user defined identifier.

Every identifier in a C++ program has a type associated with it. This type determines what operations can be applied to the name and how such operations are interpreted. Variables in C++ can be classified into four types based on its data type: (i) integer variables, (ii) real variables, (iii) character variables, and (iv) boolean variables

```
Types of variables  --|--  Integer variables
                      |
                      |--  Real or floating point variables
                      |
                      |--  Character variables
                      |
                      |--  Boolean variables
```

(a) Integer Variables Identifiers which are used to hold integer data items are called *integer variables*. All integer variables that are used within a program should be declared in the variable declaration part of the program. The standard identifier 'int' is used to declare the integer group of variables.

The general syntax for declaring an int variable is:

```
int id1,id2;
```

where `id1`, `id2` are the list of integer variables that are to be used in a program.

For example, following are some valid integer variable declarations:

```
int a,b;
int x = 10,y = 20,z = 30;
```

In the first statement, the user defined variables 'a' and 'b' are declared as integer variables in which 'a' and 'b' are the symbolic representation of the memory address for manipulating only with an integer data type. If the variables are defined as integer group and any attempt is made to assign any other data type such real or boolean or character, then computer will display an error message. The second statement of the variable declaration of the above example contains three variables as integer type namely 'x', 'y' and 'z'.

(b) Real Variables Identifiers which are used to manipulate floating point numbers (real numbers) are called *real variables*. One of the standard identifiers, namely, float, double or long double is used to declare the real group of variables.

The general syntax of the real variable declaration is:

```
float id1,id2,idn;
```

where `id1,id2` and `idn` are the list of real variables that are to be used in a program.

```
double fd1,fd2;
```

where `fd1` and `fd2` are the list of real variables of type double.

For example, following are some valid real variable declarations:

```
float x,y;
float abc = 1.2f;
double a = 1.1,b = 2.2e-4,c = 1.1e-3;
```

Integer data can be assigned to floating point variables whereas, the other way is not permitted. Any other data type such as boolean or character is not permitted to be assigned to a real type variable.

(c) Character Variables Variables which are used to store and manipulate only a single alphanumeric character is known as *character variable*. The standard identifier 'char' stands for character and is used to declare the character group of variables.

The general syntax of the character variable declaration is:

```
char id1,id2,idn;
```

where `id1,id2` and `idn` are the list of character variables that are to be used in a program.

For example, following are some valid character variable declaration:

```
char ch1,ch2;
char ch = 'a', ch3 = '?';
```

where `ch1,ch2`, `ch` and `ch3` are character variables that are used to handle only a single alphanumeric character. No other data type is permitted to be assigned to a character variable.

(d) Boolean Variables Variables which are used to handle only the boolean value of either 'true' or 'false' type are called boolean variables. The boolean data can be assigned from one variable to another or sometimes it can be initialized but it cannot be given as input from the keyboard. The standard identifier 'bool' is used to declare boolean variables in C++.

The general syntax of the boolean variable declaration is:

```
bool id1,id2,idn;
```

where `id1`, `id2` and `idn` are the list of boolean variables that are to be used in a program.

For example, following are some valid boolean variable declarations:

```
bool flag1 = true,flag2 = false;
bool flag = 1;
```

where `flag1` and `flag2` are boolean variables that are used to handle only a boolean value such as 'true' or 'false'.

Some invalid variable declarations Following are some examples of invalid variable declarations. The reasons for such an invalid declaration are also given.

```
(1) float  rate of interest;   // error
```

Note that there is no space permitted in the user defined variable if it is to be treated as a single variable.

```
(2) integer a,b; // error
```

Note that the data type 'int' is wrongly placed as integer.

```
(3) char 'a', 'b' // error
```

where variables 'a' and 'b' are not the user defined identifiers.

```
(4) bool cflag;//error
    char cflag;
```

Note that same variable name cannot be given to two different identifiers.

```
(5) int x1,y1,x1; // error
```

where the user defined identifier `x1` has repeated twice which is also not permitted.

3.6 THE CONST DATATYPE

A constant is similar to a variable except that it holds one value for its entire existence. The compiler will issue an error if one tries to change a constant. In C++, the `const` modifier is used to declare a constant variable.

The main purpose of using the constant in a progam is

- to prevent inadvertent errors caused by the users
- to give names to unclear literal values
- to facilitate changes to the code

The general syntax of const variable is given below:

const data_type identifier = initial_value;

For example,

```
const int i = 30;
```

where `i` is a user defined const int variable and initialized with 30.

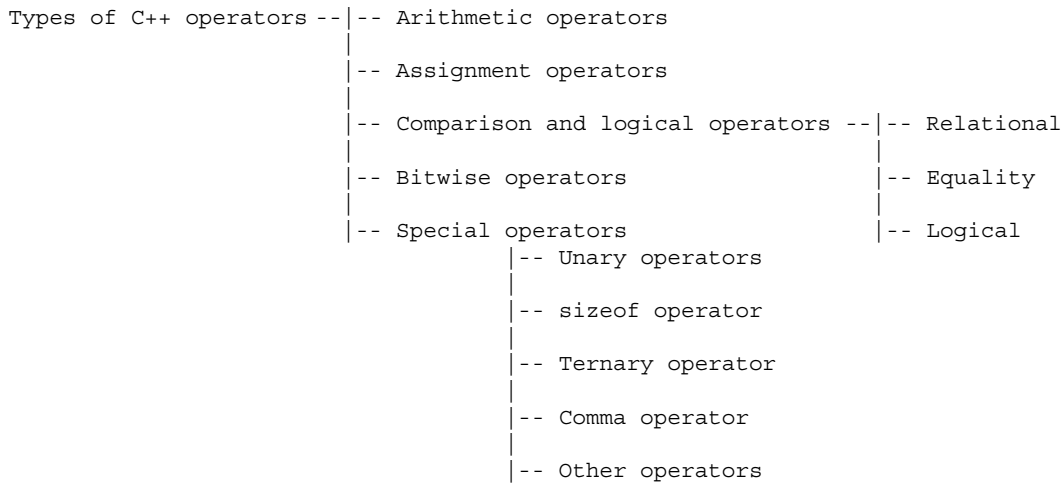
```
const float pi = 3.142;
const char password = '?';
```

The value of const data type is unalterable or unchangeable in a program. The const variables are initialized at the time of declaration and they cannot be modified or altered within a program. The const variables are called as unalterable or unchangeable variables in a program.

3.7 C++ OPERATORS

Operators are a set of symbols or notations which are used to perform a predefined operation within objects. For example, plus (+) symbol is used to add the contents of the two operands. In general, an operator is placed between the operands. Based on operator usage in the expression, the C++ operators can be classified into various groups such as arithmetic, logical, assignment, bitwise logical operators, etc.

In the following section, the different types of C++ operators and their usages are explained in detail.

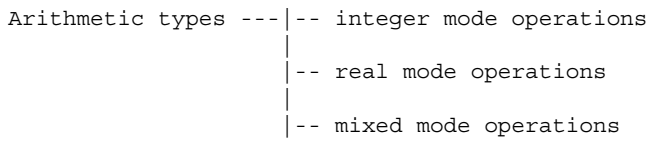


3.8 ARITHMETIC OPERATORS

In any computer programming language, arithmetic operations are the most basic and common operations which is to be performed within two objects. Operators that are used to perform the arithmetic operations are called arithmetic operators.

Arithmetic operators are known as binary operators as they require two variables to be evaluated. An arithmetic operator with single operand is meaningless and hence the computer cannot evaluate with single operand. For example, if one wants to multiply any two numbers, one has to enter or feed the multiplicand and the multiplier. That is why, it is considered as a binary operator. In other words, an operator which require two operands to be evaluated is called a binary operator.

Based on the data types, an arithmetic expression can be classified into three types, namely, integer mode, real mode and mixed mode operations.



3.8.1 Integer Mode Arithmetic

Integer mode operators are the set of arithmetic operators used only for integer type operands, constants, or expressions. The integer type expression always returns only an integer quantity.

Following are the integer operators that are used for integer data types or operands in Table 3.6.

Table 3.6

operator	*meaning*
+	addition
-	subtraction
*	multiplication
/	division
%	modulus

(a) Modulus Operator % The modulus operator % gives the remainder after performing integer divisions of two operands. For example, let us take two integer variables i and j. Then, i % j gives the remainder after dividing i by j.

Some examples for % operator are:

```
  9/2     =  1
- 9/2     = -1
  9/(-2) =  1
- 9/(-2) = -1
  4/2     =  0
  4/(-2) =  0
  2/3     =  2
```

(b) Division Operator / The division operator / is used to get a quotient of two integer quantities after performing integer division. For example, let us take two integer variables 'i' and 'j'. Then, `i/j` gives the quotient after dividing 'i' by 'j'.

Some examples for / operator are:

```
- 9/2     = -4
  9/2     =  4
- 9/(-2) =  4
  9/(-2) = -4
  1/(-2) =  0
- 1/(-2) =  0
  0/2     =  0
```

The following rules are applied whenever % operator is used in an arithmetic expression:

- second operand of the % operator cannot be zero
- % operator gives the result either as an integer or zero.

Strictly speaking, % operator is used only for the integer data type and cannot be used for real mode or mixed mode data type. For example, the following simple arithmetic expression shows how C++ compiler evaluates the operators of the data types within the operands.

a = 1 , b = 2 and c = 3;

```
(1) a * b / c
    1* 2 / 3
    (1 * 2) / 3
       2 / 3
         0
```

The above expression evaluates and gives the value of the statement as 0.

Consider the following example,

```
(2) a * b % c + 1
    1 * 2 % 3 +1
    (( 1 * 2) % 3 ) +1
     ( 2 % 3) +1
      2 + 1
        3
```

The second expression evaluates and gives the value as 3.

Following are some of invalid usage of the % and / operators in an expression:

```
15 % 0       - second operand of % must be nonzero
23.3 % 1.1   - both operands of % must be integer data types
17 / 0       - second operand of / must be nonzero
```

3.8.2 Real Mode Arithmetic

Real mode arithmetic expressions are formed using real mode quantities such as constants, variables, functions with real mode arithmetic operators. Following are the real mode arithmetic operators that are used in C++ compiler in Table 3.7.

Table 3.7

operator	*meaning*
+	addition
-	subtraction
*	multiplication
/	division

For example, following variables are declared in real mode whose values are assigned as ;

x = 1.1, b = 2.2 and c = 3.3

```
(1)
    a + b * c
    1.1 + ( 2.2 * 3.3 )
    1.1 + 7.26
       8.36
```

The above expression evaluates and gives the value as 8.36

Consider the following second expression

```
(2)
    a + b - c * a
    1.1 + 2.2 - 3.3 * 1.1
    1.1 + 2.2 - (3.3 * 1.1)
    1.1 + 2.2 - 3.63
    3.3 - 3.63
     -0.33
```

The above expression evaluates and displays the value as –0.33

Note that while using division / operator, the second operand must be nonzero.

3.8.3 Mixed Mode Arithmetic

When an integer and real quantities such as constants, variables, expressions are mixed to form an expression, it is called as *mixed mode arithmetic operations*. While constructing mixed mode arithmetic expressions, one has to note that the integer specific operators such as / and % should be declared only as integer data type. These operators yield the result also an integer mode.

For example, integer division / and real mode division / give two different results as:

Integer Mode

```
int a = 6,b = 8;
int result;
result = a/b;
that is, 6 / 8   = 0 (integer mode division)
```

Real Mode

```
float a = 6, b = 8;
float result;
result = a/b;
that is, 6.0 / 8.0 = 0.75    (real mode division)
```

Mixed Mode

```
int a = 6;
float b = 8f;
result = a/b;
that is, 6/8 = 0.75   (mixed mode division)
```

In the mixed mode operation, if an expression involving both integer and real values are to be evaluated, C++ compiler first converts the integer operands to real mode and then the expression will be evaluated. The result of such an expression is always a real value but the C++ compiler drops the digits after the decimal point. In order to improve the precision, the cast operation is done. Casting is a process of converting one data type to another without dropping the precision or number of digits. The cast operator will be discussed subsequently in this chapter. In general, the casting operation is done in the following form:

```
int a,b;
a = 6;
b = 8;
double result  = (double)a / (double)b;
```

3.8.4 Arithmetic Operator Precedence

The operator precedence is a set of rules, in which a C++ operator gets evaluated in an hierarchical order. The list of priority levels in which C++ arithmetic operators are carried out is given in the following Table 3.8.

Table 3.8

priority level	*operators*
highest	parentheses
high	% / *
low	+ -

When an arithmetic expression containing two or more operators are to be evaluated, the operations are performed one at a time, in sequence. The associativity of the arithmetic operators are from left to right.

For example, consider the following arithmetic expression and see how it gets evaluated:

```
x  = 1, y  = 4 and  z  = 2
x + y % z * 5 - y
```

The above expression will be evaluated in the following manner as per the precedence rules designed in the C++ compiler.

```
x + (( y % z) * 5) - y
1 + ((4 % 2) * 5) - 4
1 + (( 4 % 2) * 5) - 4
1 + (0 * 5) - 4
1 +  0 - 4
1 - 4
  - 3
```

The expression gives the result as –3

3.8.5 Use of Parentheses

For example, consider the following arithmetic expression and the order of evaluation:

```
x  = 38, y  = 100 and z  = 64
x + y / z
```

If no parenthesis is used, then the C++ compiler apply the precedence rules based on the default setting of the compiler and hence, the above expression will be evaluated as:

```
x + ( y / z )
38 + (100 / 64)
 38 + 1
  39
```

The / operator has higher priority than the addition operator + and so, the above expression gives the result as 39.

In case, one intends to evaluate the addition operator first, then the parentheses are required so that the default order of precedence of arithmetic operators can be changed. The purpose of using parentheses is to change the order of precedence from the default setting by the compiler, to the user choices. Of course, inside the parentheses, the same order of precedence will be applied.

For example, consider the following expression:

```
(x +  y) / z
(38 + 100) / 64
 138 / 64
  2
```

The parentheses have the highest priority of all arithmetic operators and so the above expression gives the value as 2.

3.8.6 Subexpression

An expression that is enclosed in parentheses within another expression is called a *subexpression*. For example, suppose the following variables are defined, whose values are assigned as:

```
a  = 125, b  = 40, c = 6, d  = 5 and e  = 72
(a + b % c) * d - e
```

In the above expression, `(a + b % c)` is a subexpression and within the subexpression, the same order of the precedence is used to evaluate an expression. The above expression is evaluated as follows:

```
(a + b % c) * d - e
(125 + 40 % 6) * 5 - 72
 (125 + 4) * 5 - 72
  129 * 5 - 72
  645 - 72
   573
```

In case, a subexpression itself is enclosed within many parentheses, the innermost parentheses will be evaluated first. For example, consider the following expression:

```
a  = 5, b  = 4, c  = 6, d  = 3, e  = 9
(a*b-c) % e + (( b+c) *e / d)
 (5*4 -6) % 9 + ((4+6) * 9 / 3)
  (20 -6) % 9 + ((10 * 9 / 3)
    14 % 9 + (90 / 3)
     5 + 30
       35
```

The above expression gives the value as 35.

3.9 ASSIGNMENT OPERATORS

A statement which contains an assignment operator is called the assignment statement and it is used to store the value of an expression in the computer memory for further reference. In other words, an assignment operator is used to assign back to a variable, a modified value of the present one. The assignment statement is one of the most common statements in any computer programming language.

The general syntax of the assignment statement is given in Table 3.9(a).

```
variable  = expression;
```

where expression is any valid arithmetic or logical expression.

Table 3.9(a)

Operator	*Meaning*
=	Assign Right Hand side (RHS) value to the Left Hand Side (LHS)

The symbol = is used as an assignment operator, and it is evaluated at the last.

The assignment operator does two things. First, the expression on the right side of the assignment statement is evaluated. Then the result is stored in the variable on the left side of the assignment operator.

The type of expression on the RHS must agree with the type of the target variable on the LHS of the assignment operator. For example, one can only assign an expression of type Boolean to a variable of Boolean type, or an expression of string type to a variable of string type. There are two exceptions. For example, an integer may be assigned to a real variable (but not vice versa) and a char value may be assigned to a string variable (but not vice versa). Note that the expression on the right may be any valid C++ expression.

Following are valid C++ assignment statements:

```
count = 10;
a = b+y;
nextchar = nextchar+1;
index = 2*count+1;
```

Some of the invalid assignment expressions are:

```
x := a+b;          - the colon symbol (:) is not allowed
index+1 = index;   - the assignment operator is placed wrongly
a+b+d = c;         - LHS should be a single variable but not an expression
```

The corrected code of the above invalid statements are given below:

```
x = a+b;
index = index+1;
c = a+b;
```

3.10 ARITHMETIC ASSIGNMENT OPERATORS

An assignment operator is used to assign back to a variable, a modified value of the present holding. The following Table 3.9(b) given the summary of arithmetic assignment operator used in C++:

Table 3.9(b)

Operator	*Meaning*
=	Assign right hand side (RHS) value to the left hand side (LHS).
+=	Value of LHS variable will be added to the value of RHS and assign it back to the variable in LHS.
–=	Value of RHS variable will be subtracted from the value of LHS and assign it back to the variable in LHS.
*=	Value of LHS variable will be multiplied by the value of RHS and assign it back to the variable in LHS.
/=	Value of LHS variable will be divided by the value of RHS and assign it back to the variable in LHS.
%=	The remainder will be stored back to the LHS after the integer division is carried out between the LHS variable and the RHS variable.

(*Contd*)

Operator	Meaning
>>=	Right shift and assign to the LHS.
<<=	Left shift and assign to the LHS.
&=	Bitwise AND operation and assign to the LHS.
|=	Bitwise OR operation and assign to the LHS.
^=	Bitwise complement and assign to the LHS.

The symbol = is used as an assignment operator and it is evaluated at the last. Remember that equal to = is an operator and not an equation maker and hence, it can appear anywhere in place of another operator. The following are valid C++ statements.

```
a = b = c+4;
c = 3*(d = 12.0/x);
```

For example,

`x += y` is equal to `x = x+y`
`x -= y` is equal to `x = x-y;`

3.11 COMPARISON AND LOGICAL OPERATORS

Operators that are used to compare and relate two quantities of numbers, strings, or characters in C++ are called comparison operators. In general, the comparison and logical operators are used in a program to make a decision or a selection based on some condition.

Comparison and logical operators can be classified into three types: (i) relational operators, (ii) equality operators, and (iii) logical operators. This section shows how to define and use a comparison and logical operators in C++. The following Table 3.10 is a list of operators that are used in C++ for decision making purpose.

Table 3.10

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
= =	Equal to
!=	Not equal to
&&	Logical AND
||	Logical OR
!	Not

Note that all of the above operators return '1' for true and '0' for false.

3.11.1 Relational Operators

Operators that are used to relate and compare any two items are called *relational operators*. In other words, relational operators compare values to see if they are equal or if one of them is greater than the other and so on. An expression that uses a relational operator is known as a *relational expression*.

Relational operators in C++ produce only a one or a zero result. These are often specified as "true" or "false" respectively, since these are the only two values possible. The following operators are used to perform the relational operations of the two variables or expressions.

Operators that are used to perform relational operations of two variables or arithmetic expressions are given below in the Table 3.11.

Table 3.11

Operator	*Meaning*
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

The general syntax of the relational operators is:

```
expression_1 relational_operator expression_2
```

The `expression_1` will be compared with `expression_2` and depending upon the relation like greater than, greater than or equal to and so on, the result will be either "0" or "1".

For example, following is a list of simple relational expression and their corresponding results:

Expression	*Logical Result*	*C++ Return Value*
3 > 4	false	0
6 <= 2	false	0
10 > –32	true	1
(23*7) >= (–67+89)	true	1

For example, consider the following variables which are defined as integer data type whose values are assigned as:

```
a  = 4, b = 6, c  = 8,
(a + b * c ) != ( a * b + c)
```

Then the expression evaluates in the following manner:

```
(4 + 6 * 8) != (4 * 6 + 8)
  (4 + 48) != (24 + 8)
    52 != 32
     false, which returns 0.
```

The above expression evaluates as 'false'. The relational operators have the lowest priority than the arithmetic operators. If more than one operator occurs in the same precedence of a given expression, then the evaluation will be done from left to right.

3.11.2 Equality Operators

Operators that are used to equate two quantities are called *equality operators*. When two items are compared, the result will be either equal or not equal. The two quantities can be any arithmetic expression, character or string data types. The result of the equality expression is always one of the values either "0" or "1"

These operators are normally represented by using the symbol = = (equal to) and ! = (not equal to). Note that the single equal to sign = is the assignment operator in C++. Proper care must be taken while using these operators. The following Table 3.12 gives the summary of equality operators used in C++:

Table 3.12

Operator	*Meaning*
= =	Equal to
!=	Not equal to

Like the relational operators, the equality operators also produce the result, either '0' or '1', depending on the condition used in a program.

The general syntax of the equality operators is:

```
expression_1  equality_operator expression_2
```

The `expression_1` will be equated with `expression_2` and depending on the relation like equal to and not equal to, the logical result will be either 'true' or 'false'.

For example, following is a simple expression which uses the equality of two items:

Expression	*Logical Result*	*C++ Return Value*
3 = = 4	false	0
6 != 2	true	1
10 != –32	true	1
(23*7) = = (–67+89)	false	0
'a' = = 'A'	false	0
'a' != 'b'	true	1

For example, consider the following variables which are defined as integer data type whose values are assigned as:

```
a  = 4, b = 6, c  = 8,
(a + b * c ) != ( a * b + c)
```

The above expression evaluates in the following manner:

```
(4 + 6 * 8) != ( 4 * 6 + 8)
  (4 + 48) != ( 24 + 8)
    52 != 32
     true,  which returns  1.
```

The above expression evaluates as true. The relational and the equality operators have the lowest priority than the arithmetic operators. If more than one operator occurs in the same precedence of a given expression, then the evaluation will be done from left to right.

3.11.3 Logical Operators (Boolean Operators)

Operators that are used to make a decision or a selection of a program based on the logical values of either 'true' or 'false' are called as logical operators. The logical operators are also called *Boolean operators*. Any expression either integer arithmetic or character or boolean data which uses logical operators is called *logical expression*. Of course, the result of the logical expression will be one of the boolean values i.e., either 'true' or 'false'. Recall that in C++, 'true' is equivalent to any nonzero value, and 'false' is equivalent to zero.

Boolean variables are used in programs for the following reasons:

- to avoid multiple evaluations of boolean expressions
- to improve program clarity
- to serve as indicators, called flags, of program status.

Logical operators AND and OR are lower in precedence than the relational operators < and > which are lower than the arithmetic operators + and –. The operator AND is higher than the operator OR. Some other languages place the logical operators higher than the relational operators, which require parentheses.

The list of the logical (Boolean) operators that are used in C++ is given in Table 3.13.

Table 3.13

Operatior	*Meaning*
&&	Logical AND
||	Logical OR
!	Logical negation

(a) Logical AND (&&) The logical AND operator (&&) gives a value of TRUE, if and only if, both its operands have the value TRUE. Otherwise, the value is FALSE. In other words, a compound expression is true, when two conditions (expressions) are true.

The general syntax of the logical AND (&&) operator usage is as given below:

```
expression_1  &&  expression_2
```

where `expression_1` and `expression_2` must be one of these expressions: integer arithmetic expressions, character data or Boolean data types. When a character data type is used for logical operations, character is converted to integer and are thus allowed in the expression.

The following table 3.14 summarises the various possible conditions and their results of the logical AND (&&):

Table 3.14

Situation	*Results*
true && true	true
true && false	false
false && true	false
false && false	false

Consider the following variables which are declared as integer data with the values assigned as:

For example,

```
a = 4, b = 5, and c = 6
    (a < b) && (b < c)
```

Then, the logical expression evaluates in the following manner:

```
(4 < 5) && ( 5 < 6)
  true  &&  true
      true
```

The result of the above expression is true which returns 1.

Short circuit evaluation of logical AND A logical operator expression consists of two expressions separated by one of the logical operators (&&) and (||). For each of the logical operators described in this section, the second operand is not evaluated at all if the value of the first operand provides sufficient information to determine the result of the logical operator expression.

In a short circuit evaluatioon, the logical AND (&&) will not evaluate the second condition, if the first condition is false, due to the result will be false.

For example, consider the following expression

```
int a = 10, b = 20, c = 30;
(a > b) && (b < c)
(10 > 20)
 false
```

The expression (b < c) will not evaluate when the first expression is false. In case, the first expression is true, the logical AND (&&) operator forces the second expression to be evaluated.

```
(a < b) && (b > c)
(10 < 20)
true &&  (20 >30)
true && false
false,
```

which returns 0

(b) Logical OR (||) The logical OR operation gives a value of TRUE if either or both of the operands has a value TRUE; otherwise the value is FALSE. In other words, a compound expression is false, when two conditions (expressions) are false.

The general syntax of the logical OR operator (||) usage is given below:

```
expression_1 ||  expression_2
```

where `expression_1` and `expression_2` must be one of these expressions: integer arithmetic expressions, character data or Boolean data types. When a character data type is used for logical operations, character is converted to integer and are thus allowed in the expression.

The following Table 3.15 summarises the various possible conditions and their results of the logical OR (||):

Table 3.15

Situation	*Results*
true || true	true
true || false	true
false || true	true
false || false	false

Consider the following variables which are declared as integer data with the values assigned as:

```
a = 4, b = 5, and c = 6
    (a < b) || (b > c)
```

Then, the logical expression evaluates in the following manner:

```
(4 < 5) || ( 5 > 6)
 true  || false
      true
```

The result of the above expression is true which returns 1.

Short circuit evaluation of logical OR In a short circuit evaluation, the logical OR (||) will not evaluate the second condition if the first condition is true, due to the result will be true.

For example, consider the following expression

```
int a = 10, b = 20, c = 30;
(a < b) || (b > c)
(10 < 20)
true
```

The expression `(b < c)` will not evaluate when the first expression is true. In case, the first expression is false, the logical OR (||) operator forces the second expression to be evaluated.

```
(a > b) || (b < c)
(10 > 20)
false ||  (20 < 30)
false || true
true, which returns 1.
```

(c) Logical Negation Operator (!)

The logical negation or NOT operator (!) is used to change the value of a logical expression false to true or from true to false. The results of logical negation operator (!) are in Table 3.16

Table 3.16

Situation	*Results*
!(true)	false
!(false)	true

The general form of the expression is:

```
!(expression)
```

Depending on the value of the expression, i.e. whether it is true or false the result will be complement of the value of the expression.

Consider the following variables which are declared as integer data with the values are assigned as:

```
a = 4, b = 5, and c = 6
```

(1) !(a < b) Then, the logical expression evaluates in the following manner:

```
!(4 < 5)
!(true)
 false
```

The result of the above expression is false which returns 0.

(2) For example, consider the following complex logical expression:

```
!(a < b) || (c > b)
!(4 < 5) || ( 6 > 5)
 !(true) ||  true
  false || true
      true, which returns 1.
```

This expression evaluates to true as the logical negation operator has the highest priority among the relational and logical operators and that is why the above expression gets the value as 'true'.

(3) Consider the following logical expression:

```
! ((a < b) || (b > c))
! ((4 < 5) || (5 > 6))
   ! (true  || false)
    ! (true)
     false, which returns 0.
```

The above expression evaluates to false because the parentheses have the first priority and so the inside parentheses will be evaluated first. That is why, the above expression gives the value as 'false'. To resolve ambiguities in the order of application of Boolean operators, the following Table 3.17 precedence rules are applied:

Table 3.17

Operator	*Priority*
!	highest (evaluate first)
&&	intermediate
||	Lowest (evaluate last)

Parentheses can be used to change the order of evaluation.

The negation operator has a higher precedence level than AND and OR. Parentheses are usually necessary to arrive at desired order of evaluation. Since the logical AND has a higher precedence than the logical OR, some care should be taken wherever required.

For example,

```
expression1 || expression2 && expression3 || expression4
```

The computer will be evaluated as

```
expression1 || ( expression2 && expression3 ) || expression4
```

Suppose, one intends to evaluate the above expression in the following manner:

```
(expression1 || expression2 ) && (expression3  || expression4 )
```

Then, use of parentheses is absolutely essential.

For example, consider the following expression;

```
a = 4, b = 5, and c = 6
```

(1)

```
(a < b) || (b > c) && (a > b) || (a > c)
(4 < 5) || ( 5 > 6) && ( 4 > 5) ||( 5 > 6)
 true  ||  false  &&  false || false
  true  ||  (false && false) || false
       true || false || false
            true || false
              true,  which returns 1.
```

The logical AND (&&) has higher precedence over the logical OR (||) and that is why, the above expression evaluates as 'true'.

(2) Now consider the second expression:

```
((a < b) || (b > c)) && ((a > b) || (a > c))
((4 < 5) || (5 > 6)) && ((4 > 5) || (5 > 6))
    (true || false) && (false || false)
           true && false
            false,  which returns 0.
```

To avoid unpredictable or undesirable results, it is better to use parentheses.

The precedence rules that are used in C++ operators are given in Table 3.18:

Table 3.18

Precedence	*Operators*
highest	! unary + unary – * / % && + – ||
Lowest	< <= != == >= >
The last	=

Note that operations within parentheses are performed before operations not enclosed in parentheses. In the case of nested parentheses, operations within the inner parentheses are evaluated before those in the outer parentheses. Otherwise, operations having the same precedence are evaluated in the left to right sequence of their appearance.

3.12 BITWISE OPERATORS

This section deals with the bitwise operations which are supported by the C++ compilers. There are some situations wherein bitwise operations are to be performed, which is possible in C++.

The following operators are used for the bitwise logical decision making. Normally, most of the high level programming languages do not support the bitwise operations. The bitwise operators are one of the salient features of C++. The following operations can be performed using bitwise operators:

- bitwise AND
- bitwise OR
- bitwise exclusive OR
- bitwise left shift
- bitwise right shift
- bitwise complement

The bitwise operators that are supported in the C++ are given in Table 3.19

Table 3.19

Operator	*Meaning*
~	Bitwise complement
&	Bitwise AND
|	Bitwise inclusive OR
^	Bitwise exclusive OR (XOR)
>>	Bitwise right shift
<<	Bitwise left shift

(a) Bitwise Complement Operator (~) The complement operator (~) switches all the bits in a binary pattern, that is, all the zeroes become ones and all the ones become zeroes. The complement of a pattern is often useful in signalling and controlling other devices where several different signals may be complement to each other. The following example shows how the bitwise complement works without using a sign bit:

Variable	*Value*	*Binary pattern*
x	23	00010111 (8 bits)
~x	232	11101000
y	ff	11111111
~y	00	00000000

The general syntax of the bitwise NOT (~) operator is:

```
var2  = ~var1;
```

where `var2` and `var1` are declared as one of the simple data types, namely, int, short int, etc.

For example, the following C++ program segment shows how to use the bitwise NOT operator (~):

```
int a = 5,b;
b = ~a;
cout << "a = " << a;
cout << "b = " << b;
```

(b) Bitwise Logical Operators This section shows us how to use the bitwise logical operations. It has already been discussed in the previous section that C++ supports various types of logical (boolean) operators that are used for making a decision or a selection of the part of a program. The following bitwise logical operators are used for logical decision making in C++:

- bitwise AND (&)
- bitwise OR (|)
- bitwise exclusive OR (^)

(i) Bitwise AND operator (&) The bitwise AND operation will be carried out between the two bit patterns of the two operands. For example,

Variable	*Value*	*Binary pattern*
x	5	0101
y	2	0010
x & y	0	0000
a	6	0110
b	3	0011
a & b	2	0010

To generate a 1 bit in the result, bitwise AND needs a one in both numbers. Masking bits can be done using bitwise AND. The most useful part of the bitwise operations is a bitwise AND. Normally it is called a mask operator and one may select the particular pattern as either a one or a zero and it selects certain specific bits and ignores the others.

The general syntax of the bitwise AND (&) operator is:

```
var3  = var2 & var1;
```

where `var2` and `var1` are declared as one of the simple data types, namely, int, short int etc.

For example, the following C++ program segment shows how to use the bitwise AND operator (&):

```
int a = 5,b = 2,c;
c = a & b;
cout << "a = " << a;
cout << "b = " << b;
cout << "c = " << c;
```

(ii) Bitwise OR operator (|) The bitwise OR operations are similar to the bitwise AND and the result is 1 if any one of the bit value is 1. The symbol (|) represents the bitwise OR.

For example,

Variable	*Value*	*Binary pattern*
x	5	0101
y	2	0010
x | y	7	0111
a	6	0110
b	1	0001
a | b	7	0111

The general syntax of the bitwise OR operator (|) is:

```
var3  = var2 | var1;
```

where `var2` and `var1` are declared as one of the simple data types, namely, int, short int, etc.

For example, the following C++ program segment shows how to use the bitwise OR operator (|):

```
int a = 5,b = 2,c;
c = a | b;
cout << "a = " << a;
cout << "b = " << b;
cout << "c = " << c;
```

(iii) Bitwise Exclusive OR (XOR) Operator (^) The bitwise exclusive OR will be carried out by the notation (^). To generate a 1 bit in the result, a bitwise exclusive OR needs a 1 in either number but not both.

Variable	*Value*	*Binary pattern*
x	5	0101
y	2	0010
x ^ y	7	0111
a	6	0110
b	3	0011
a ^ b	5	0101

The general syntax of the bitwise XOR operator (^) is:

```
var3  = var2 ^ var1;
```

where var2 and var1 are declared as one of the simple data types, namely, int, short int. etc.

For example, the following C++ program segment shows how to use the bitwise XOR operator (^):

```
int a = 5,b = 2,c;
c = a ^ b;
cout << "a = " << a;
cout << "b = " << b;
cout << "c = " << c;
```

(c) Shift Operations Shift operations take binary patterns and shift the bits to the left or right, keeping the same number of bits, by dropping shifted bits off the end and filling in with zeroes from the other end. C++ provides two types of shift operations such as left shift and right shift.

(i) Shift bitwise left operator (<<) The << operator is used for left shifting.

Variable	Value	Binary pattern
x	33	00100001 (8 bits)
x << 1	the bit pattern of the x value is left shifted once.	
x	66	0 01000010

x << 3 the bit pattern of the x value is left shifted by thrice.

```
        0 0 1 0 0 0 0 1
0       0 1 0 0 0 0 1 0
0       1 0 0 0 0 1 0 0
1       0 0 0 0 1 0 0 0
```

The resultant bit pattern will be

```
0 0 0 0 1 0 0 0
```

The general syntax of the bitwise shift left operator (<<) is:

```
var2  = var1 << n;
```

where var2 and var1 are user defined identifiers whose data types are integers. The counter value n is used to shift the bit pattern of the var1, leftwise n times and the resultant value of the var1 is assigned to the var2.

For example, the following C++ program segment shows how to use the bitwise shift left operator (<<):

```
int a = 33,b = 3,c;
c = a << b;
cout << "a = " << a;
cout << "b = " << b;
cout << "c = " << c;
```

(ii) Shift bitwise right operator (>>) The right shift >> operator is used for right shifting.

Variable	*Value*	*Binary pattern*
y	41	00101001
y >> 1	the bit pattern of the x value is right shifted once.	
y	20	00010100 1 skipped

y >> 3 the bit pattern of the y value is right shifted thrice

```
0 0 1 0 1 0 0 1
0 0 0 1 0 1 0 0   1
0 0 0 0 1 0 1 0   0
0 0 0 0 0 1 0 1   0
```

the resultant bit pattern will be

```
0 0 0 0 0 1 0 1
```

The general syntax of the bitwise shift right operator (>>) is:

```
var2  = var1 >> n;
```

where `var2` and `var1` are user defined identifiers whose data types are integers. The counter value n is used to shift the bit pattern of the var1, right n times and the resultant value of the var1 is assigned to the var2.

For example, the following C++ program segment shows how to use the bitwise shift right operator (>>):

```
int a = 5,b = 2,c;
c = a >> b;
cout << "a = " << a;
cout << "b = " << b;
cout << "c = " << c;
```

3.13 BITWISE ASSIGNMENT OPERATORS

All the bitwise operators may appear with the assignment operator just like the arithmetic operators discussed earlier.

Assignment operators

- Arithmetic assignment operators
- Bitwise assignment operators

The bitwise assignment operators are summarised in the following table.

Bitwise Assignment Operators

The summary of the bitwise assignment operators is given in table 3.20:

Table 3.20

Operator	*Meaning*
>>=	Right shift and assign to the LHS
<<=	Left shift and assign to the LHS
&=	Bitwise AND operation and assign to the LHS
|=	Bitwise OR operation and assign to the LHS
^=	Bitwise exclusive OR and assign to the LHS

For example,

(1) `n >>= 1` means `n = n >> 1`
n will be shifted right by 1 and then assigned back to variable `n`.

(2) `a <<= b` means `a = a << b`
the content of the a will be shifted left by the content of `b` and then assigned back to `a`.

(3) `x &= 4` means `x = x&4`
The bitwise AND operation will be carried out between the bit patterns of the two operands, namely `x` and `4` and then assigned back to `x`.

(4) `i ^= k` means `i= i^k`
The bitwise exclusive OR operation will be carried out between the bit patterns of the two operands, namely `i` and `k` and then assigned back to `i`.

(5) `j |= 5` means `j = j|5`
The bitwise OR operation will be carried out between the bit patterns of the two operands, namely *j* and 5 and then assigned back to `j`.

3.14 SPECIAL OPERATORS

There are some special operators used in the C++ language to perform a particular type of operation. These special operators are mostly used for pointer and memory manipulation. Some of the examples for the special operators are unary operators, ternary operator, incrementer, decrementer and sizeof operators. These operators are very unique and special in C++.

3.14.1 Unary Operators

The unary operators require only a single expression to produce a line. Unary operators usually precede their single operands. Sometimes, some unary operators may be followed by the operands such as incrementer and decrementer. The most common unary operation is unary minus, where a minus sign precedes a numerical literal, a variable or an expression.

The unary operators that are used in c++ are given in the Table 3.21.

Table 3.21

Operator	*Meaning*
*	Contents of the storage field to which a Pointer is pointing (refer chapter 8)
&	Address of a variable (refer chapter 8)
–	Negative value (minus sign)
!	Negation (0, if value # 0 and 1,if value = 0)
~	Bitwise complement
++	Incrementer
– –	Decrementer
type	Forced type of conversion
sizeof	Size of the subsequent data type or type in byte

(a) Pointer Operator (*) The pointer operator is used to get the content of the address operator pointing to a particular memory element or cell.

(b) Address Operator (&) The address operator & is used to get the address of the other variable in an indirect manner.

The pointer operator (*) and the address (&) are explained in Chapter 8 on "Pointers and Strings."

(c) Incrementer and Decrementer Two special operators are used in C++, namely, incrementer and decrementer. These operators are used to control the loops in an effective and compact manner.

(i) Incrementer The ++ (double plus) symbol or notation is used for incrementing by 1. For example,

++i; is equal to i = i+1;
i++; is equal to i = i+1;

There are two types of incrementers: prefix incrementer (++i), and postfix incrementer (i++). For the time being, let us take it that they can be used according to a programmer's liking but there are some special conditions under which these incrementers are used very particularly.

In prefix incrementer, first it is incremented and then the operations are performed. On the other hand, in the postfix incrementer, first the operations are performed and then it is incremented. However, the result of the incremented value will be the same in both the cases.

For example,

```
        int i = 7;
(1)  x = ++i;
(2)  x = i++;
```

After execution, the value of i in both cases will be set to 8. But in the first case, the value of x is set to 8 due to prefix incrementer and the second case, the value of x is set to 7 only due to postfix incrementer. In the case of postfix, first the value is assigned and then incrementation is performed.

(ii) Decrementer The decrementer is also similar to the incrementer. The – – (double minus) symbol or notation is used for decrementing by 1. For example, consider the following expression,

`--i` is equal to `i = i-1;`
`i--` is equal to `i = i-1;`

In this also, there are two types of decrementers, prefix decrementer (– –i) and postfix decrementer (i– –).

In prefix decrementer, first it is decremented and then the operations are performed. On the other hand, in the postfix decrementer, first the operations are performed and then it is decremented. However, the result of the decremented value will be the same in both the cases.

For example,

```
        int i = 7;
(1)  x = --i;
(2)  x = i--;
```

After execution, the value of i in both cases will be set to 6. But in the first case, the value of x is set to 6 due to prefix decrementer and the second case, the value of x is set to 7 due to postfix decrementer.

Summary of the incrementers and decrementers is given in Table 3.22 The difference between the prefix and postfix is subtle but can be very important.

Table 3.22

Operator	*Symbol*	*Form*	*Meaning*
prefix increment	++	++i	increment i, then get value of i
prefix decrement	– –	– –i	decrement i, then get value of i
postfix increment	++	i++	get value of i, then increment i
postfix decrement	– –	i– –	get value of i, then decrement i

Like the unary minus operator, the increment and decrement operators are unary. The operand must be a scalar lvalue, it is illegal to increment or decrement a constant or a structure.

```
++5;  // error
```

(d) The Sizeof Operator In general, the `sizeof` operator is used to find the size of aggregate data objects such as arrays, structures, classes and objects. The sizeof operator is used to give the direction to the C++ compiler to reserve the memory size or block to the particular data type which is defined in the structure type of data in the linked list. The sizeof operator can be used in one of the following forms:

```
sizeof(t);
or
sizeof t;
```

where *t* is any data type or expression.

The sizeof operator accepts two types of operands: an expression or a data type. However, the expression may not have type function, or void or be a bitfield. Moreover, the expression itself is not evaluated, the compiler determines only what type the result would be.

If the operand is an expression, sizeof returns the number of bytes that the result occupies the memory;

```
sizeof(13 + 5)          - returns the size of an int (4 if ints are four bytes long)
sizeof(113.0 + 5)       - returns the size of a double (8 if doubles are eight bytes long)
```

For expressions, the parentheses are optional, so the following is legal:

```
sizeof x;
```

By convention, however, the parentheses are usually included. The operand can also be a data type, in which case the result is the length in bytes of objects of that type:

```
sizeof(char)   - 1 byte
sizeof(short)  - 2 byte
sizeof(float)  - 4 byte
```

The parentheses are required if the operand is a data type. Note that the results of most sizeof expressions are implementation dependent. The only result that is guaranteed is the size of a char, which is always 1. One can also use the sizeof operator to obtain information about the sizes of the objects in the C++ environment. The following program prints the size of the basic data types:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "char = " << sizeof(char) << endl;
    cout << "short = " << sizeof(short) << endl;
    cout << "int = " << sizeof (int) << endl;
    cout << "long = " << sizeof (long) << endl;
    cout << "float = " << sizeof(float) << endl;
    cout << "double = " << sizeof (double) << endl;
    return 0;
}
```

Output of the above program

```
char = 1
short = 2
int = 4
long = 4
float = 4
double = 8
```

(e) Cast Operator The cast operator is to convert the set of declared data type to some other required type. C++ provides a specific and a special way for converting one data type to the other using a cast operator.

3.14.2 Ternary Operator (?:)

C++ includes a very special operator called the ternary or conditional operator. It is called ternary because it uses three expressions. The ternary operator acts like a shorthand version of the if-else construction. The general format of the ternary operator is:

```
expression1 ? expression2 : expression3
```

which results in either `expression2` or `expression3` being evaluated. If `expression1` is true, then `expression2` is evaluated; otherwise, `expression3` is evaluated. The conditional operator is used in place of a single if-else to make an assignment.

For example,

```
if ( (value % 2) == 0)
    even = true;
else
    even = false;
```

This can be written as

```
even = ( (value  % 2) == 0) ? true: false;
```

Another example would be to find the larger of two numbers,

```
if (first > second)
    min = first ;
else
    min = second;
```

This can be written as

```
min = (first > second) ? first: second;
```

3.14.3 Comma Operator

C++ uses the comma in two ways. The first use of comma is as a separator in the variable declaration.

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

Another use is as an operator in an expression for loop. It is explained in detail in the Chapter 5 on "Control statements".

3.14.4 Other Operators

The following are the special operators used in C++ for representing arrays, structures, classes, unions and pointers.

(i) Parentheses for grouping expressions
(ii) Membership operators

(a) Parentheses for Grouping Expressions () The precedence and associativity of each operator determines whether it takes effect before or after the next operator in an expression. Often, it is more convenient to control this order of evaluation. When parentheses are put around an element of an expression, that element evaluates before anything outside the parentheses.

For example, (a+b) *c would cause the addition to be performed the multiplication, whereas a+b*c would cause the multiplication to be performed.

(b) Membership Operator There are several kinds of variables used in C++, containing a set of values rather than just one, namely arrays structures and unions. To represent the variables, membership operators are used, which are represented as `[] . ->`

3.15 TYPE CONVERSION

This section presents how to convert and promote one data type to another using the cast operator. The type conversion is to convert the set of declared type to some other required type. It is easy to convert values from one type to another type in a C++ program.

In certain situations, when some variables are declared as integer but sometimes it may be required to get the result as floating point numbers.

For example, assume that the following type has been declared

```
int x;
float y = 11.99912;
```

The question now arises whether, for example,

```
x = y;
```

where x is an integer type and y is a floating point type, is a valid assignment.

The answer to this question is in affirmative, but we should be aware that truncation will take place. This happens most often between float and double, and between int and char, since converting between float and int or double and int involves cutting off all the decimal places.

Conversion can be carried out in two ways:

- Converting by assignment
- Using cast operator

(a) Converting by Assignment It is a usual way of converting a value from one data type to another by using the assignment operator (equal to sign). This means that we can convert a value from one type to another just by assigning a float variable's value to a double value variable, a char variable to an int variable or an int variable to a float variable.

For example,

```
int x,y,z;
float a,b,c;
double dvalue;
x = 10;
a = 3101.2567
dvalue = 3546879.908
b = x;
y = a;
z = dvalue;
```

The first one is an integer value assigned back to a floating point number. In the second example, the float variable is assigned back to the integer variable, and in the third, the double value is assigned to the float variable.

In C++, converting by assignment operator is not recommended to the programmer, as it will truncate the fractional or real parts and one may not get the desired results. To avoid this, there is a special way of converting one data type to the other, by using the cast operator.

(b) Cast Operator Converting by assignment operator is carried out automatically but one may not get the desired result. The cast operator is a technique to forcefully convert one data type to the other. The operator used to force this conversion is known as the 'cast operator' and the process is known as 'casting'. The cast operator takes on the format.

```
(cast-type) expression;
```

or

```
cast-type (expression);
```

As an example, to force a floating point number to an integer, we could use the following.

```
result = (int)(19.2/4);
```

or

```
result = int (19.2/4);
```

The cast operation (int) casts not only the 19.2 but the entire expression. Thus, results receive the value 4, rather than the entire quotient 4.8. The cast operator takes precedence over most other operations.

To demonstrate the use of the cast operator, let us consider the following example:

```
char ch;
int x,y,z;
float abc;
```

(1) `x = (int)ch;`

where ch is a character variable and force to convert as the integer data type

(2) `abc = float(y) / float(z);`

where the expression (y/z) is an integer data type and force to convert as a floating point number.

One of the uses of casts, is to promote an integer to a floating point number to ensure that the result of a division operation is not truncated, as illustrated in the following example.

```
3/2
```
which yields 1

because fractional part is truncated.

using cast operation

```
result  = float (3) / float(2);
```

result is 1.5 because the 3 is converted to a float.

Note that the cast operator has very high precedence, so the preceding expression is parsed as if it had been written

```
((float) 3) /2
```

Another use of the cast operator is to convert function arguments. Most of the runtime mathematical library functions expect its arguments to be of type double. If the variables are integers, one needs to cast them to double before pass them as arguments.

The ANSI C++ standard supports a new syntax for declaring the type of arguments that makes this sort of cast unnecessary. The most frequent and important uses of casts involve pointers and data initialization.

3.16 ANSI C++ TYPE CASTING

In C++, one can certainly use C-style casts, but ANSI/ISO C++ provides several casting operators that old C++ doest not support. These operators and their purposes are listed in the following Table 3.23:

Table 3.23 *Casting operators available in ANSI/ISO C++*

Operator	*Description*
dynamic_cast	Returns a valid object pointer only if the object used as its operand is of an expected type.
static_cast	can be used to explicitly perform any implicit type conversion, much like the ANSI C cast.
const_cast	can be used to remove any const, volatile or unsigned attribute from a class.
reinterpret_cast	Allows any pointer type to be converted into any other pointer type; also allows any integral type to be converted into any pointer type and vice versa.

(a) Dynamic_Cast The dynamic_cast operator is used to support and manipulate runtime identification of class objects addressed by pointer or reference. The general syntax of the dynamic_cast operator is,

```
dynamic_cast <type>(expression)
```

(b) Static_Cast The static_cast operator is used to make explicit casts, without runtime type checking and turns off warning messages. The general syntax of the static_cast operator is,

```
static_cast <type>(expression)
```

One can use static_cast to make explicit the kinds of casts that the compiler could actually perform implicitly (although it might issue a warning). For example, the following program illustrate show a static_cast operator is used for casting from a float value to an int value.

```
#include <iostream>
using namespace std;
int main()
{
    int sum;
    int abc = 10;
    float pi = 3.99;
    int sum = static_cast <int>(pi) + abc;
    cout << " sum = " << sum << endl;
 }
```

Output of the above program

```
sum = 13
```

(c) Const_Cast The const_cast operator is used to remove the constness in a program. The general syntax of the const_cast operator is,

```
const_cast <type>(expression)
```

(d) Reinterpret_Cast The reinterpret_cast operator is used to support and realize low level reinterpretation of the bit pattern of the expression. The general syntax of the reinterpret_cast operator is,

```
reinterpret_cast <type> (expression)
```

3.17 SUMMARY OF ANSI C++ OPERATORS

The C++ language includes all C operators and adds several new operators. Operators specify an evaluation to be performed on one of the following:

- One operand (unary operator)
- Two operands (binary operator)
- Three operands (ternary operator)

Operators follow a strict precedence, which defines the evaluation order of expressions containing these operators. Operators associate with either the expression on their left or the expression on their right; this is called "associativity." The following Table 3.24 shows the precedence and associativity of C++ operators (from highest to lowest precedence). Operators in the same segment of the table have equal precedence and are evaluated left to right in an expression unless explicitly forced by parentheses.

Table 3.24

Operator	*Name or Meaning*	*Associativity*
::	Scope resolution	None
.	Member selection (object)	Left to right
–>	Member selection (pointer)	Left to right
[]	Array subscript	Left to right
()	Function call	Left to right
()	member initialization	Left to right
++	Postfix increment	Left to right
– –	Postfix decrement	Left to right
typeid()	type name	Left to right
const_cast	Type cast (conversion)	Left to right
dynamic_cast	Type cast (conversion)	Left to right
reinterpret_cast	Type cast (conversion)	Left to right
static_cast	Type cast (conversion)	Left to right
sizeof	Size of object or type	Right to left
++	Prefix increment	Right to left
– –	Prefix decrement	Right to left
~	One's complement	Right to left
!	Logical not	Right to left
–	Unary minus	Right to left

(Contd)

Operator	*Name or Meaning*	*Associativity*
+	Unary plus	Right to left
&	Address-of	Right to left
*	Indirection	Right to left
new	Create object	Right to left
delete	Destroy object	Right to left
()	Cast	Right to left
.*	Pointer-to-member (objects)	Left to right
–>*	Pointer-to-member (pointers)	Left to right
*	Multiplication	Left to right
/	Division	Left to right
%	Modulus	Left to right
+	Addition	Left to right
–	Subtraction	Left to right
<<	Left shift	Left to right
>>	Right shift	Left to right
<	Less than	Left to right
>	Greater than	Left to right
<=	Less than or equal to	Left to right
>=	Greater than or equal to	Left to right
= =	Equality	Left to right
!=	Inequality	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise inclusive OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
e1?e2:e3	Conditional	Right to left
=	Assignment	Right to left
*=	Multiplication assignment	Right to left
/=	Division assignment	Right to left
%=	Modulus assignment	Right to left
+=	Addition assignment	Right to left
– =	Subtraction assignment	Right to left
<<=	Left-shift assignment	Right to left
>>=	Right-shift assignment	Right to left
&=	Bitwise AND assignment	Right to left
|=	Bitwise inclusive OR assignment	Right to left
^=	Bitwise exclusive OR assignment	Right to left
throw expr	throw expression	Right to left
,	Comma	Left to right

3.18 ANSI C++ ALTERNATE PUNCTUATION TOKENS

The ANSI/ISO C++ provides the following keywords as synonyms for punctuation tokens. These keywords are also recognized by the C++ preprocessor. The ANSI C++ alternate punctuation takens are given in Table 3.25.

Table 3.25

Keywords	*Operator Meaning*
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

REVIEW QUESTIONS

1. What is an identifier? Explain how a user defined identifier is different from a standard identifier.
2. What is meant by a keyword or reserved word? List all the keywords that are used in the C++ language.
3. What are the rules to be followed for forming a user defined identifier in C++?
4. Summarise the various data types that are supported in C++.
5. How do the following data types different from one another?
   (a) short (b) signed
   (c) unsigned (d) hexadecimal
   (e) octal (f) long
6. What is an integer data type? Explain the different types of integers represented in C++.
7. What is a floating point number? List a few applications of using floating point numbers in a real life problems.
8. Elucidate how a floating point number is realised and represented in C++.
9. Explain the difference between numeral and non-numeral representation of data in a programming concept.
10. Explain the difference between simple data type and aggregated data type. Give suitable examples for your explanation.
11. What is meant by standard data type? In what way a standard data type is different from a user defined data type?
12. What is a constant data type and what are the merits and demerits of defining constant data type in a program?

13. What is a character data? Explain how a character data is different from an integer item.
14. Explain how a Boolean data is represented in C++? List a few applications of using Boolean data in a program.
15. What is a character literal? In what way a character literal is different from the boolean literal?
16. What is a string literal? What are the rules to be followed to form a string literal?
17. What is a variable? Explain the differences between variables and constants.
18. What are the rules to be followed to define and use a variable in C++?
19. Explain the following with suitable examples:
(a) integer variable (b) real variable
(c) character variable (d) bool variable
20. What is an operator? List the various types of operators that are used in the C++ language.
21. Can you define a statement without an operator? Explain.
22. List the operators that are used only for the integer arithmetic operations in C++.
23. List the operators that are used for the real arithmetic operations in C++.
24. What is the importance of using precedence rules in C++ operators?
25. Summarise the rules associated with assignment operator.
26. What is meant by the comparison and logical operators? How are they different from arithmetic and assignment operators?
27. List all the operators that are used for the comparison and logical decison making in C++.
28. What is meant by equality operator? In what way is an equality operator different from an assignment operator?
29. Explain the importance of arithmetic assignment operators.
30. Distinguish between binary minus and unary minus operators.
31. What is a modulus operator and how does it work in C++?
32. What is meant by bitwise operator? List a few applications of using a bitwise operator in a program.
33. What are the differences between logical && and the bitwise & operator ?
34. List the various bitwise operators used in C++.
35. Explain the following bitwise operators with a suitable example.
(a) Bitwise AND (b) Bitwise OR
(c) Bitwise exclusive OR (d) Bitwise left shift
(e) Bitwise right shift (f) Bitwise complement
36. What is a unary operator? List out the different types of unary operators used in C++.
37. What is meant by incrementer and decrementer in C++?
38. Explain the uses of the following special operators in C++:
(a) pointer operator (b) address operator
(c) `sizeof` operator (d) ternary operator
(e) comma operator
39. What is meant by membership operator in C++?
40. What is the use of type conversion in C++?
41. What is meant by cast operator? How is it different from other operators and what is the associativity?
42. Summarise all the operators that are used in C++ along with their associativity.

CONCEPT REVIEW QUESTIONS

1. Some of the following user defined identifiers are written wrongly. Find the errors (if any):
(a) pay bill gross (b) a+b (c) 'a' (d) xy12

(e) 21a (f) signed (g) private (h) "and"
(i) _a123 (j) Rs 10 (k) $value123 (l) pay_bill

2. Find out the valid and invalid representation of the following integer literals:
(a) 7,000 (b) +0.0 (c) Rs 10 (d) 1e2
(e) 7.1e–2 (f) –2 (g) sum (h) 7756
(i) 10u (j) 1232L (k) 67565UL (l) –765LU
(m) 656 (n) 43211 (o) 65467FL (p) 543el

3. Find out the valid and invalid representation of the following octal and hexadecimal literals:
(a) ox1000U (b) +0 (c) 0abcUL (d) 0XabcdefUL
(e) 7.1eXl (f) –02L (g) 0xfffabcdfL (h) 0886
(i) x89 (j) LUx0123 (k) 0x123U (l) –0765
(m) –0X8765 (n) 01234560 (o) L "ff345" (p) L'oxa'

4. Find out the valid and invalid representation of the following floating point literals:
(a) Rs 45.55 (b) 1.1e–2 (c) 0.1e–03 (d) 1,12.2
(e) 1e–2 (f) 0 (g) –0.0001 (h) 1.1e–1.1
(i) 1.1f (j) –2.2e–3L (k) 0.0F (l) 1223E1
(m) 3.3333L (n) 9.999999FL (o) .1E–3F (p) 0.F

5. Determine the valid character and string literals among the following:
(a) L'a' (b) 'a+b' (c) L "a" (d) x
(e) L "Hello C++" (f) 'Computer' (g) abc123 (h) LU123
(i) "this is a test" (j) L "\n" (k) '\n' (l) L '\0'

6. Write the following algebraic expression in C++:
(a) $\frac{a}{b} + 10$ (b) $\frac{5}{9}$ (f–32) (c) $\frac{a}{b} * \frac{c}{d}$ (d) a $\frac{a + v}{b - 1}$

7. Find the syntax error/(s), if any, in each of the following. Assume that all variables are declared as real data types.
(a) result = pay bill +5.0; (b) x / y *c = sum;
(c) y = 1.0/–2.5 *y; (d) pay = Rs 100 +basic* DA;
(e) sum = s (s–a) (s–b) (s–c); (f) final = a % 3;
(g) value = a / b + 1.0 *a*b;

8. Write the algebraic expression corresponding to each of the following C++ statements:
(a) g = x * (x * x + y * y)/ (x *x – y* y +2.0);
(b) sum = x*x/a/b;
(c) y = x + x/z + x–2.0;
(d) root = (b*b –4.0 *a *c)/ (2.0*a);
(e) interest = p * n *r /100.0;

9. Find the final value of each of the following arithmetic expression. Assume that all variables are declared as real data types and initialised with the following data:

```
double a = 4, b = 3, c = 2, d = 1;
```

(a) v1 = b–c *d / a–d;
(b) v2 = (a * b +c+d) *a;
(c) v3 = a*a + c *d / a–b;
(d) v4 = ((a+b) *(c*a +(a*a))– b *b) / ((c+d) * (c–d));
(e) v5 = (a –2.0 * (a–b))/ (d–c) + a–b*a;
(f) v6 = b*b –4.0*a*c/2*a ;
(g) v7 = (b *b – 4.0 *a*c)/(2.0*a);

10. Determine the final value of each of the following arithmetic expression. Assume that all variables are declared as integer data types and initialized with the following data:

```
int a  = 1, b  = 2, c  = 3, d  = 4;
```

(a) sum1 = b–c*d / a–d;
(b) sum2 = (a % b +c / d)*a;
(c) sum3 = a % a + c*d / a–b;
(d) sum4 = ((a+b)% (c / a +(a*a))– b / b)+((c+d)*(c–d));
(e) sum5 = (a / (–2) % (a–b)) – (d–c) + a–b*a;
(f) sum6 = b*b –4 / a*c % 2*a ;
(g) sum7 = (b *b – 4*a *c)+ (a % d);

11. Determine the final value of each of the following Boolean expression. Assume that the variables a, b and c are declared as integer data types and the variables ch1 and ch2 as character types. The variables are initialised with the following data:

```
int a = 4, b = 3, c = -2;
ch1 = 'a', ch2 = 'b';
```

(a) exp1 = (a < b) && (b >c);
(b) exp2 = (a > b) || (ch1 != ch2) && (a < c);
(c) exp3 = (a == b) || (ch1 == ch2) && (a <= c);
(d) exp4 = !(a > b) && (ch1 <= ch2);
(e) exp5 = !(ch1 >= ch2) || (a >= c) && (b >= c);
(f) exp6 = (a == b) && (a > b) && !(ch1 <= ch2);
(g) exp7 = (a+b * c) > (a*c);
(h) exp8 = ((a % 5) + (b / c) + (a + b)) <= (a*b);
(i) exp9 = (!(ch1 <= ch2)) || (!(a <= c)) && (!(a>=b));
(j) exp10 = (a % b+b % c+a % c) == (a / b+b / c+a / c);
(k) exp11 = !((a*b) + (a*c)) <= ((b*a) + a % c);

12. Determine the final value of each of the following Boolean expression. Assume that all variables are declared as Boolean data types and initialized with the following data:

```
bool x  = true;
bool y  = false;
bool z  = false;
```

(a) f1 = x || y && z ;
(b) f2 = x && y || !z;
(c) f3 = !x && !y && !z;
(d) f4 = !(x && !x || !y);
(e) f5 = !(!z && !y && !x);
(f) f6 = x && y && z || (x && !(y || x));
(g) f7 = !x || !y && !z || y && z;
(h) f8 = y && !z || !(!y && z);

13. Determine the final value of each of the following arithmetic assignment expression. Assume that all variables are declared as integer data types and initialized with the following data:

```
int a = 1, b  = 2, c  = 3, d  = 4;
```

(a) a += b*c–d;
(b) b –= (++c / ++d)*a;
(c) c *= a % b + a++ + ++b;
(d) d /= a+b–c;
(e) a %= b*a;
(f) b += b*b –4 / a*c % 2*a;

Input and Output Streams

Chapter 4

This chapter presents the preliminary concepts of the structure of the C++ program. The emphasis is on the various types of declaration and arithmetic operations of the C++ language using the basic input and output statements and the definition of Input and Output (I/O) streams using the header files such as <iostream> and <iomanip> and formatting of input and output streams with manipulators. Numerous illustrative examples are given to explain the above concepts.

4.1 COMMENTS

C++ supports two types of comments: one retaining the C style comments and other is a C++ style that introduces the comment to end of line delimiter. C programmers will be already familiar with comments delimited by /* and */ while C++ gives the comments to end of line delimiter //.

For example, C programming style of comments are given below:

```
/*
#include <stdio.h>
void main()
{
     printf (" Hello World !\n");
} */
```

The C++ programming style of comments are as follows:

```
//   #include <iostream>
//   using namespace std;
//   int main()
//   {
//        cout <<" Hello, C++ world\n";
//        return 0;
//   }
```

In general, /*... */ comments style is used for large block of statements and the // style is used for one line comments.

For example, for a block of statements,

```
/* this is
    a test
       program
       by ravic */
```

For a single line, the comments are as mentioned below :

```
// i++; C++ style
```

4.2 DECLARATION OF VARIABLES

ANSI C and C++ share a common base syntax. The rules that apply to variable declaration and creation are the same. One of the most obvious differences between the two languages is in the declaration of variables. In C, it is essential to declare all the variables within a scope, before executable statements are to be defined. On the other hand, C++ allows a user to mix data declaration within functions and executable code.

For example, the following program segment illustrates how variables are declared after the executable statements are defined.

```
int main()
{
    int x;
    printf (" this is a test program \n");
    printf (" by Ravich \n");
    float x,y; // allows in C++
}
```

C++ allows declaration of variables to be placed very close to their point of actual usage. The main advantages of using this manner of declaration are:

- easier to follow and understand the variable declaration, if it is required to go through the program for further enhancements.
- from software engineering point of view, the maintainability, and modifiability of the code is much cheaper.
- C++ has made less prone to errors.
- testability of the code is less complex.
- C++ also permits to declare the index variables within the for loop statement itself.

For example,

Case 1

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

Case 2

```
for (int i=0, j = 10; i<= n-1; j- ,i++) {
   -------
   -------
}
```

4.3 THE MAIN () FUNCTION

C does not define a specific format for the main () function. The definition of the main function look like this:

```
main()
{
    /*
     main program code
     */
}
```

Otherwise, in the following way the main() function can be defined with command line arguments:

```
main (int argc, char *argv[])
{
    /*
     main program code
     */
}
```

But, the ANSI/ISO standard C++ explicitly defines main () as matching one of the two following two prototypes:

```
int main()
{
    // main program code here
    return 0;
}
```

Otherwise, the main () function is defined with the command line arguments:

```
int main ( int argc, char *argv[])
{
    // main program code here
    return 0;
}
```

where `argc` is the number of arguments passed to the program and `argv [0]` are the addresses of the passed arguments. `argv[0]` is equivalent to `argv[argc-1]`.

C++ compilers will also give an error or warning message if it does not return a value from main. Whenever a main function returns a null value, it is essential to declare a main function with the prototype void.

Restrictions of using main () function in C++ Several restrictions apply to the main function that do not apply to any other C++ functions. The main function:

- Cannot be overloaded (see Overloading).
- Cannot be declared as inline.
- Cannot be declared as static.
- Cannot have its address taken.
- Cannot be called.

4.4 SIMPLE C++ PROGRAMS

The skeleton of a typical ANSI/ISO C++ program structure is given below:

```
#include <iostream>
using namespace std;
int main()
```

```
{
    return 0;
}
```

PROGRAM 4.1

The following is a standard C++ program structure to display the message "Hello C++ world, Many Greetings to you" on the video screen.

```
#include <iostream>
using namespace std;
int main()
{
    cout << " Hello C++ world, Many Greetings to you ";
    return 0;
}
```

The function `main()` must be placed before the begin statement. It invokes other functions to perform its job. The ({) symbol or notation is used as the begin statement. The declaration of variables and the type of operations are placed after the begin statement. The end statement is denoted by the symbol (}). In C++, the semicolon (;) serves the purpose of a statement terminator rather than a separator.

Statements are terminated by a semicolon and are grouped within braces {...}. Most statements contain expression, sequences of operators, function calls, variables and constants that specify computation. Variable and function names are of arbitrary lengths and consist of upper and lower case letters, digits and underscore and they may not start with a numeral. All C++ keywords are written in lowercase letters.

The symbol back-slash (\) followed by a lowercase `n` is used for line feed or a new line. In C++, it is considered as a single character

```
\n   Newline or line feed
```

PROGRAM 4.2

A program without using a new line character in the cout stream statement is given below:

```
#include <iostream>
using namespace std;
int main()
{
     cout << " Hello, C++ world! ";
     cout << " Many greetings to you ";
}    return 0;
```

Output of the above program

```
Hello, C++ world!  Many greetings to you
```

PROGRAM 4.3

A modified program is shown below for displaying the message in two lines using a new line character `(\n)` *in the* `cout` *statement.*

```
#include <iostream>
using namespace std;
```

```
int main()
{
   cout << " Hello, C++ world\n";  //newline character is inserted
   cout << " Many greetings to you ";
   return 0;
 }
```

Output of the above program

```
Hello, C++ world
Many greetings to you
```

4.5 PROGRAM TERMINATION

In C++, there are several ways to exit a program:

- Call the exit () function.
- Call the abort () function.
- Execute a return statement from main.

(a) The Exit () Function The `exit()` function, declared in the standard include file STDLIB.H, terminates a C++ program. The value supplied as an argument to exit is returned to the operating system as the program's return code or exit code. By convention, a return code of zero means that the program completed successfully.

Issuing a return statement from the main function is equivalent to calling the exit function with the return value as its argument.

```
#include <iostream>
using namespace std;
int main()
{
      cout <<"Hello, C++ world!\n";
      exit(0);
}
```

Output of the above program

```
Hello, C++ world!
```

Note that one can use the constants EXIT_FAILURE and EXIT_SUCCESS, defined in STDLIB.H, to indicate success or failure of the program.

```
#include <iostream>
using namespace std;
int main()
{
      cout <<"Hello, C++ world!\n";
      EXIT_SUCCESS;
}
```

Output of the above program

```
Hello, C++ world!
```

(b) The Abort () Function The `abort()` function, also declared in the standard include file STDLIB. H, terminates a C++ program. The difference between `exit` and `abort` is that `exit` allows the C++ run-time termination processing to take place (global object destructors will be called), whereas `abort` terminates the program immediately.

```
#include <iostream>
using namespace std;
```

```
int main()
{
      cout <<"Hello, C++ world!\n";
      abort ();
}
```

Output of the above program

```
Hello, C++ world!
```

(c) The Return Statement Issuing a return statement from main is functionally equivalent to calling the exit function.

Consider the following example:

```
#include <stdlib.h>
int main()
{
    exit(3);
    return 3;
}
```

The `exit` and `return` statements in the preceding example are functionally identical. However, C++ requires that functions that have return types other than `void` return a value. The return statement allows one to return a value from main.

4.6 FEATURES OF IOSTREAM

It is well known that C++ is a tool for Object Oriented Programming (OOP). In order to handle the various object oriented features of a program, it is essential to have robust input and output facilities. Most of the object oriented programs handle three items: objects, message passing and optional parameters with the message from the outside world. C++ provides a new way to perform the input and output operations called iostream method.

The standard header file input and output stream `(iostream)` contains a set of small and specific general purpose functions for handling input and output data. The I/O stream is a sequence of following characters written for the screen display or read from the keyboard. The standard input and output operations in C++ are normally performed by using the I/O stream as `cin` for input and `cout` for output.

It is well known that the C programming language comes with an extensive library of functions for handling input and output but it cannot add new format specifier to `printf()/scanf()` functions. C++ supports input and output of numbers, characters and strings more conveniently and efficiently than C. The ANSI/ISO standard C++ compiler supports both C-style of I/O header <cstdio> and C++ form of I/O header <iostream>.

4.6.1 Differences in Standard C++ Iostream

The iostream library is one of the main differences between the Standard C++ Library and previous versions of run-time libraries. Details of the iostream implementation have changed, and it may be necessary to rewrite parts of C++ code that use iostream if one wants to link with the Standard C++ Library.

A list of non-standard versions or any old iostream headers is given below:

```
iostream.h
istream.h
ostream.h
iomanip.h
```

```
ios.h
fstream.h
streamb.h and
strstrea.h
```

The new Standard C++ iostream headers are given below and all without the .h extension.

```
<iostream>
<istream>
<ostream>
<iomanip>
<ios>
<iosfwd>
<fstream>
<sstream>
<streambuf> and
<strstream>
```

One can include the standard headers in any order, a standard header more than once, or two or more standard headers that define the same macro or the same type. Including a standard header within a declaration is not permitted. It is advisable not to define macros that have the same names as that of macros within the standard header.

4.6.2 Types of IOstreams

In fact, C++ does not have built-in input and output functions for handling input and output of data from the outside world, but it supports different kinds of stream related header files for providing these facilities.

The Input and Output (I/O) streams fall into two groups, namely, byte oriented and wide character oriented.

For Conventional Byte Oriented Operations cin, cout, cerr, and clog are byte oriented, performing conventional byte-at-a-time transfers.

```
cin          standard input
cout         standard output
cerr         standard error with limited buffering
clog         similar to cerr but with full buffering
```

For Handling Wide Characters wcin, wcout, wcerr, and wclog are wide oriented, translating to and from the wide characters that the program manipulates internally.

```
wcin         standard input for handling wide character
wcout        standard output for handling wide character
wcerr        standard error with limited buffering for wide character
wclog        similar to cerr but with full buffering for wide character
```

Operations on a stream manipulation are different for byte oriented and wide character oriented. One cannot perform operations of a different orientation on the same stream. Therefore, a program cannot operate interchangeably on both cin and wcin.

4.6.3 ios

The stream library is a hierarchy of classes and summarizes in Table 4.1. The streambuf class is the basis of all streams. It defines the basic characteristics of buffers that hold characters for input and output. The ios class is derived from streambuf. It defines the basic formatting and error control capabilities used in streambuf. The ios is a virtual base class for the classes istream (input stream) and ostream (output stream). The iostream (input/output stream) class is derived from both istream and ostream. The ‘ios’ correspond

to input and output streams (hence io). The definitions based on 'istream' correspond to input streams and those based on 'ostream' to output streams. C++ allows six types of stream classes, namely,

- `istream`
- `ostream`
- `iostream`
- `wistream`
- `wostream`
- `wiostream`

Table 4.1

Stream classes	*Meaning*
`istream`	The istream consists of input functions to read a stream of characters from the keyboard.
`ostream`	The ostream consists of output functions to write a character onto the screen.
`iostream`	The iostream supports both input/output stream of functions to read a stream of characters from the keyboard and to display a stream of objects onto the video screen.
`wistream`	The wistream is a type basic_istream specialised on wchar_t.
`wostream`	The wostream is a type basic_ostream that is specialised on wchar_t.
`wiostream`	The wiostream is a type basic_iostream specialised on wchar_t.

4.6.4 Output Streams

An output stream object is a destination for bytes. The three most important output stream classes are `ostream`, `ofstream`, and `ostrstream`.

(i) ostream The ostream class, which can be configured for buffered or unbuffered operation, is best suited to sequential text-mode output. All functionality of the base class, ios, is included in ostream.

(ii) ofstream The ofstream class supports disk file output. If one needs an output-only disk, construct an object of class ofstream. One can specify whether ofstream objects accept binary or text-mode data before or after opening the file. Many formatting options and member functions apply to ofstream objects, and all functionality of the base classes ios and ostream is included.

(iii) ostrstream The ostrstream class supports output to in-memory strings. To create a string in memory using I/O stream formatting, construct an object of class ostrstream.

4.6.5 Input Streams

An input stream object is a source of bytes. The three most important input stream classes are istream, `ifstream`, and `istrstream`.

(i) istream The `istream` class is best used for sequential text-mode input. One can configure objects of class istream for buffered or unbuffered operation. All functionality of the base class, ios, is included in istream.

(ii) ifstream The `ifstream` class supports disk file input. If one needs an input-only disk file, construct an object of class ifstream. One can specify binary or text-mode data. If one declares a filename in the constructor, the file is automatically opened when the object is constructed.

(iii) istrstream The `istrstream` class supports input from in-memory strings. To extract data from a character array that has a null terminator, allocate and initialise the string, then construct an object of class istrstream.

4.6.6 <iostream> Members

The following are the member functions of the <iostream> class presents in Table 4.2.

```
cin          wcin
cout         wcout
cerr         wcerr
clog         wclog
```

Table 4.2

Iostream Member	*Meaning*
cerr	The object controls unbuffered insertions to the standard error output as a byte stream.
cin	The object controls extractions from the standard input as a byte stream.
clog	The object controls buffered insertions to the standard error output as a byte stream.
cout	The object controls insertions to the standard output as a byte stream.
wcerr	The object controls unbuffered insertions to the standard error output as a wide stream.
wcin	The object controls extractions from the standard input as a wide stream. Once the object is constructed, the call wcin.tie returns &wcout.
wclog	The object controls buffered insertions to the standard error output as a wide stream.
wcout	The object controls insertions to the standard output as a wide stream.

4.7 KEYBOARD AND SCREEN I/O

(a) cout The `cout` is used to display an object onto the standard device, normally the video screen. The insertion operator (the double less than sign <<) is used along with the cout stream.

The general syntax of the cout stream is,

```
cout << variable 1 << variable 2 <<....<< variable n ;
```

Following are a few examples to illustrate the cout member function

(1)

```
int x = 123;
float y = -45.67f;

cout << x << y ;
```

The output will be `123-45.67`

The insertion operator << will not add any space between any two data items and it is up to the programmer to define and introduce the space between them.

(2)

```
int x = 123;
float y = -45.67f;
cout << x << '\t' << y ;
```

The output will be `123 -45.67`

(3) Use of different data types freely on one line is permitted.

```
int x = 123;
float y = -45.67f;
```

```
cout << " x = " << x << '\t' << " y = " << y ;
```

The output will be `x = 123 y = -45.67`

(4) The `cout` stream is used to display different data types.

```
//cout is used to display different data types
#include <iostream>
using namespace std;
int main()
{
      int num = 3;
      cout << " number = " << num << "\n";
      char ch = 'a';
      cout << " character = " << ch << "\n";
      float fa = -34.45f;
      cout << " real number = " << fa << "\n";
  }
```

Output of the above program

```
number = 3
character = a
real number = -34.45
```

(b) cin The `cin` is used to read a number, a character or a string of characters from a standard input device, normally the keyboard. The extraction operator (the double greater than sign >>) is used along with the cin operator.

The general syntax of the `cin` is

```
cin >> variable 1 >> variable 2 >> ...variable n;
```

For example, the following are certain valid `cin` member function usage in C++.

```
int a,b;
float f1,f2;
char c1,c2;
```

(1)

```
cin >> a >> b >> f1 >> f2 >> c1 >> c2 ;
```

(2)

```
cin >> a >> b;
cin >> f1 >> f2;
cin >> c1 >> c2;
```

(3)

```
cout << " enter two integers \n";
cin >> a >> b;
cout << " enter two floating point numbers \n";
cin >> f1 >> f2;
cout << " enter any two characters \n";
cin >> c1 >> c2;
```

PROGRAM 4.4

A program to read any two numbers through the keyboard and to perform simple arithmetic operations (i.e., addition, subtraction, multiplication and division) and display the results using `cin` and `cout` functions.

```
//using cout and cin stream
#include <iostream>
using namespace std;
```

```
int main()
{
    int a,b,sum,diff,prod;
    float div;
    cout << " Enter any two numbers" << endl;
    cin >> a >> b;
    sum = a+b;
    diff = a-b;
    prod = a*b;
    div = (float)a/(float)b;
    cout << " a = " << a << " b = " << b << " sum = " << sum << endl;
    cout << " a = " << a << " b = " << b << " diff = " << diff << endl;
    cout << " a = " << a << " b = " << b << " prod = " << prod << endl;
    cout << " a = " << a << " b = " << b << " div = " << div << endl;
    return 0;
}
```

Output of the above program Enter any two numbers

```
10  20
a = 10 b = 20 sum = 30
a = 10 b = 20 diff = -10
a = 10 b = 20 prod = 200
a = 10 b = 20 div = 0.5
```

(c) `cerr` and `clog` In addition to the function cout, C++ provides other functions of the class ostream called `cerr` and `clog`. They are used to redirect error messages to other devices. The output of `cerr` is unbuffered, while the output of clog is buffered. The use of the above functions are illustrated below.

```
//using cerr and clog
#include <iostream>
using namespace std;
int main()
{
    cout << " Hello, C++ world ! using cout \n";
    cerr << " many greetings to you ! using cerr \n";
    clog << " hello, computer ! using clog \n";
}   return 0;
```

Output of the above program

```
Hello, C++ world ! using cout
many greetings to you ! using cerr
hello, computer ! using clog
```

PROGRAM 4.5

A program to demonstrate how to use the wcout method for displaying the wide characters.

```
#include <iostream>
using namespace std;
int main()
{
    wchar_t ch = L'C';
    wcout << ch << endl;
    return 0;
}
```

Output of the above program

```
C
```

Summary of C++ I/O streams

The summary of C++ I/O streams in given in Table 4.3

Table 4.3

Stream	*Meaning*
cin	keyboard input (stdin)
cout	screen output (stdout)
cerr	standard error device output (stderr)
clog	buffered output of standard error (stderr)
wcin	keyboard input (stdin) for handling wide characters
wcout	screen output (stdout) for handling wide characters
wcerr	standard error device output (stderr)
wclog	buffered output of standard error (stderr)

4.8 MANIPULATOR FUNCTIONS

Manipulator functions are special stream functions that change certain characteristics of the input and output. They change the format flags and values for a stream. The main advantage of using manipulator functions is that they facilitate the formatting of input and output streams.

The following are the list of standard manipulators used in a C++ program. To carry out the operations of these manipulator functions in a user program, the header file input and output manipulator `<iomanip>` must be included.

4.8.1 Predefined Manipulators

The Table 4.4 presents the summary of predefined manipulation normally lined in the stream classes.

Table 4.4

Manipulators	*Meaning*
`endl`	generates a carriage return or line feed character.
`ends`	attach a null terminating character (‘\0’) at the end of a string.
`flush`	used to cause the stream associated with the output to be completely emptied.
`hex,dec,oct`	Set base for integers.
`resetiosflags`	Clears the specified flags.
`setbase`	Set base for integers.
`setfill`	Sets the character that will be used to fill spaces in a right-justified display.
`setiosflags`	Sets the specified flags.
`setprecision`	Sets the precision for floating-point values.
`setw`	Specifies the width of the display field.
`ws`	ignores the leading white space that preceds the first field.

(a) endl The `endl` is an output manipulator to generate a carriage return or line feed character. The endl may be used several times in a C++ statement. For example,

(1)
```
cout << " a " << endl << "b" << endl;
```
(2)
```
cout << " a = " << a << endl;
cout << " b = " << b << endl;
```

PROGRAM 4.6

A program to display a message on two lines using the endl manipulator and the corresponding output is given below.

```
//using endl manipulator
#include <iostream>
using namespace std;
int main()
{
    cout << " Hello, C++ world!";
    cout << endl;
    cout << " Many greetings to you ";
}   return 0;
```

Output of the above program

```
Hello, C++ world!
Many greetings to you
```

The `endl` is the same as the non-graphic character to generate line feed (\n).

PROGRAM 4.7

A program to illustrate the usage of the line feed character and the endl manipulator and the corresponding output are given below.

```
#include <iostream>
using namespace std;
int main()
{
    int a;
    a = 20;
    cout << " a = " << a << endl;
    cout << " a = " << a << "\n";
    cout << " a = " << a << '\n';
    return 0;
}
```

Output of the above program

```
a = 20
a = 20
a = 20
```

(b) ends The `ends` is a manipulator used to attach a null terminating character ('\0') at the end of a string. The ends manipulator takes no argument whenever it is invoked. This causes a null character to the output.

PROGRAM 4.8

A program to show how a null character is inserted using ends manipulator while displaying a string onto the screen.

```
//using ends manipulator
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int number = 231;
    cout << '\"' << " number = " << number << ends;
    cout << '\"' << endl;
    return 0;
}
```

Output of the above program

```
" number = 231 "
```

(c) flush The `flush` member function is used to cause the stream associated with the output to be completely emptied. This argument function takes no input parameters whenever it is invoked. For output on the screen, this is not necessary as all output is flushed automatically. However, in the case of a disk file being copied to another, it has to flush the output buffer prior to rewinding the output file for continued use. The function flush () does not have anything to do with flushing the input buffer.

PROGRAM 4.9

A program to show how to use the flush () member function for displaying a string onto the screen.

```
//using flush member function
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    cout << " Hello\n";
    cout << " C++, World \n";
    cout.flush();
    return 0;
}
```

Output of the above program

```
Hello
C++, World
```

(d) Setbase () The `setbase()` manipulator is used to convert the base of one numeric value into another base. Following are the common base converters in C++.

`dec` - decimal base (base = 10)
`hex` - hexadecimal base (base = 16)
`oct` - octal base (base = 8)

In addition to the base conversion facilities such as to bases `dec`, `hex` and `oct`, the `setbase()` manipulator is also used to define the base of the numeral value of a variable. The prototype of `setbase ()` manipulator is defined in the `iomanip` header file and it should be included in user program. The `hex`,

dec, oct manipulators change the base of inserted or extracted integral values. The original default for stream input and output is dec.

PROGRAM 4.10

A program to show the base of a numeric value of a variable using hex,oct *and* dec *manipulator functions.*

```
//using dec,hex,oct manipulator
#include <iostream>
using namespace std;
int main()
{
    int  value;
    cout << " Enter number" << endl;
    cin >> value;
    cout << " Decimal base = " << dec << value << endl;
    cout << " Hexadecimal base =  " << hex << value << endl;
    cout << " Octal base =  " << oct << value << endl;
    return 0;
}
```

Output of the above program

```
Enter number
15
Decimal base = 15
Hexadecimal base = f
Octal base = 17
```

PROGRAM 4.11

A program to show the base of a numeric value of a variable using setbase manipulator function.

```
//using setbase manipulator
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int  value;
    cout << " Enter number" << endl;
    cin >> value;
    cout << " Decimal base = " << setbase(10) << value << endl;
    cout << " Hexadecimal base =  " << setbase(16) << value << endl;
    cout << " Octal base =  " << setbase(8) << value << endl;
    return 0;
}
```

Output of the above program

```
Enter number
15
Decimal base = 15
Hexadecimal base = f
Octal base = 17
```

(e) setw () The setw() stands for the set width. The setw() manipulator is used to specify the minimum number of character positions on the output field a variable will consume.

The general format of the setw manipulator function is

```
setw( int w)
```

which changes the field width to w, but only for the next insertion. The default field width is 0.

For example,

```
cout << setw(1) << a << endl;
cout << setw(10) << a << endl;
```

Between the data variables in C++ space will not be inserted automatically by the compiler. It is up to a programmer to introduce proper spaces among data while displaying onto the screen.

PROGRAM 4.12

A program to display the content of a variable without inserting any space

```
#include <iostream>
using namespace std;
int main()
{
    int  a,b;
    a = 200;
    b = 300;
    cout << a << b << endl;
    return 0;
}
```

Output of the above program

```
200300
```

PROGRAM 4.13

A program to insert a tab character between two variables while displaying the content onto the screen.

```
//using tab character
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int  a,b;
    a = 200;
    b = 300;
    cout << a << '\t' << b << endl;
    return 0;
}
```

Output of the above program

```
200     300
```

PROGRAM 4.14

A program to display the data variables using setw manipulator functions.

```
//using setw manipulator
#include <iostream>
#include <iomanip>
using namespace std;
```

```
int main()
{
    int  a,b;
    a = 200;
    b = 300;
    cout << setw(5) << a << setw(5) << b << endl;
    cout << setw(6) << a << setw(6) << b << endl;
    cout << setw(7) << a << setw(7) << b << endl;
    cout << setw(8) << a << setw(8) << b << endl;
    return 0;
}
```

Output of the above program

```
 200 300
  200   300
   200    300
    200     300
```

(f) setfill() The setfill() manipulator function is used to specify a different character to fill the unused field width of the value.

The general syntax of the setfill() manipulator is

```
setfill( char f)
```

which changes the fill character to `f`. The default fill character is a space. For example,

```
setfill('.'); // fill a dot (.) character
setfill('*') //fill an asterisk (*) character
```

PROGRAM 4.15

A program to illustrate how a character is filled in the unused field width of the value of the data variable.

```
//using setfill manipulator
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int  a,b;
    a = 200;
    b = 300;
    cout << setfill('*');
    cout << setw(5) << a << setw(5) << b << endl;
    cout << setw(6) << a << setw(6) << b << endl;
    cout << setw(7) << a << setw(7) << b << endl;
    cout << setw(8) << a << setw(8) << b << endl;
    return 0;
}
```

Output of the above program

```
**200**300
***200***300
****200****300
*****200*****300
```

(g) setprecision() The setprecision() is used to control the number of digits of an output stream display of a floating point value. The setprecision() manipulator prototype is defined in the header file `<iomanip>`.

The general syntax of the setprecision manipulator is setprecision (int p)

which sets the precision for floating point insertions to `p`. The default precision is 6.

PROGRAM 4.16

A program to use the setprecision manipulator function while displaying a floating point value onto the screen.

```
//using setprecision manipulator
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    float  a,b,c;
    a = 5;
    b = 3;
    c = a/b;
    cout << setprecision(1) << c << endl;
    cout << setprecision(2) << c << endl;
    cout << setprecision(3) << c << endl;
    cout << setprecision(4) << c << endl;
    cout << setprecision(5) << c << endl;
    cout << setprecision(6) << c << endl;
    return 0;
}
```

Output of the above program

```
2
1.7
1.67
1.667
1.6667
1.66667
```

(h) ws The manipulator function `ws` stands for white space. It is used to ignore the leading white space that precedes the first field.

PROGRAM 4.17

A program to illustrate how to use the ws manipulator function for reading a set of strings from the keyboard.

```
//using ws manipulator
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    char name[100];
    cout << " enter a line of text \n";
    cin >> ws;
    cin >> name;
    cout << " typed text  = " << name << endl;
    return 0;
}
```

Output of the above program

```
enter a line of text
this is a test program by Ravich
typed text  = this
```

(i) `Setios ags` and `Resetios ags` The setiosflags manipulator function is used to control different input and output settings. The I/O stream maintains a collection of flag bits.

The setiosflags manipulator performs the same function as the setf function. The flags represented by the set bits in `f` are set. The general syntax of the setiosflags is:

```
setiosflags (long f)
```

The resetiosflags manipulator performs the same function as that of the resetf function. The flags represented by the set bits in `f` are reset. The general syntax of the resetiosflags is as follows:

```
resetiosflags (long f)
```

PROGRAM 4.18

A program to demonstrate how `setios ags` is set while displaying a base of a numeral.

```
//using basefield bit format flag
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout <<"enter a number\n";
    cin >> num;
    cout << " default display decimal = " << num << "\n";
    cout.setf(ios::oct,ios::basefield);
    cout << " octal = " << num << "\n";
    cout.setf(ios::hex,ios::basefield);
    cout << " hexadecimal = " << num << "\n";
    cout.setf(ios::dec,ios::basefield);
    cout << " decimal = " << num << "\n";
    return 0;
}
```

Output of the above program

```
enter a number
15
default display decimal = 15
octal = 17
hexadecimal = f
decimal = 15
```

4.9 INPUT AND OUTPUT (I/O) STREAM FLAGS

To implement many of the above manipulators, I/O streams have a flag field that specifies the current settings. The flag names and their meanings are given in the following Table 4.5 below:

Table 4.5

Stream Flags	*Meaning*
`boolalpha`	displays the bool value as true or false in the stream.
`dec`	decimal base (show integers in decimal format)
`fixed`	displays a floating-point number in fixed-decimal notation.
`hex`	hexadecimal base (show integers in hexadecimal format)
`internal`	pad after sign or base indicator

(*Contd*)

Stream Flags	*Meaning*
`left`	left justification of output
`noboolalpha`	displays the bool value as 1 or 0 in the stream.
`noshowbase`	Turns off indicating the notational base in which a number is displayed.
`noshowpoint`	Displays only the whole-number part of floating-point numbers whose fractional part is zero.
`noshowpos`	Causes positive numbers to not be explicitly signed.
`noskipws`	Cause spaces to be read by the input stream.
`nounitbuf`	Causes output to be buffered and processed when the buffer is full.
`nouppercase` `scientific`	Specifies that hexadecimal digits and the exponent in notation appears in lowercase.
`oct`	octal base (shows integers in octal format)
`right`	right justification of output
`scientific`	Causes floating point numbers to be displayed using scientific notation (use `E` for floating notation).
`showbase`	show base for octal and hexadecimal numbers
`showpoint`	show the decimal point for all floating point numbers
`showpos`	show '+' to positive integers
`skipws`	skip white space during input
`unitbuf`	flush all stream after insertions
`uppercase` `scientific`	Specifies that hexadecimal digits and the exponent in notation appear in uppercase.

(a) Turning the Bit Format Flag On In order to change the state of the cout object, the bits that represent its state must be changed. The setf() function is invoked for setting the bit format flags of the I/O stream. The general format of the setf() function is

```
cout.setf(flags to be set);
```

For example,

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

The bitwise OR (|) operator is used in the argument list of the setf () function in order to change the bit format flag more than one.

For example,

```
cout.setf(ios::showbase | ios::showpoint | ios:: uppercase);
```

(b) Turning the Bit Format Flag Off The usetf () function is used to change the bits directly off. This function takes exactly one argument to turn off the bit pattern.

The general syntax of the unsetf () function is,

```
cout.unsetf(flags to be turned off);
```

For example,

```
cout.unsetf(ios::uppercase);
```

The bitwise OR (|) operator is used in the argument list in order to turn off more than one bit format flag of the I/O stream.

(c) Boolalpha and Noboolalpha Format Flag

(1) Boolalpha The boolalpha format flag is used to specify that variables of type bool appear as true or false in the stream. By default, variables of type bool are displayed as 1 or 0.

The general syntax of the boolalpha is,

```
cout << boolalpha;
```

(2) Noboolalpha The noboolalpha format flag is used to specify that variables of type bool appear as 1 or 0 in the stream. In other words, the noboolalpha reverses the effect of boolalpha format flag setting.

The general syntax of the noboolalpha is,

```
cout << noboolalpha;
```

PROGRAM 4.19

A program to display the contents of the bool variable using `boolalpha` and `noboolalpha` format flag.

```
//using boolalpha and noboolalpha
#include <iostream>
using namespace std;
int main()
{
    bool flag = true;
    cout << boolalpha;
    cout << "\n setting boolalpha \n";
    cout << "\n Flag value = " << flag;
    cout << "\n setting noboolalpha \n";
    cout << noboolalpha;
    cout << "\n Flag value = " << flag;
    return 0;
}
```

Output of the above program

```
setting boolalpha
Flag value = true
setting noboolalpha
Flag value = 1
```

(d) Basefield Bit Format Flag The basefield format flag is used to display integers in the proper base.

`ios::dec` – show integers in decimal format

`ios::oct` – show integers in octal format

`ios::hex` – show integers in hexadecimal format

Only one of the above can be set at any time. These format flags control the base in which numbers are displayed. By default, `dec` is set. The syntax of the basefield bit format flag setting is

```
cout.setf(ios::dec,ios::basefield);
cout.setf(ios::oct,ios::basefield);
cout.setf(ios::hex,ios::basefield);
```

PROGRAM 4.20

A program to display the given integer numbers using different bases, namely, decimal, octal and hexadecimal format.

```
//using dec,oct,hex format flag
#include <iostream>
using namespace std;
int main()
{
    int num;
```

```
    cout <<"enter a number\n";
    cin >> num;
    cout << " decimal = " << num << "\n";
    cout.setf(ios::oct,ios::basefield);
    cout << " octal = " << num << "\n";
    cout.setf(ios::hex,ios::basefield);
    cout << " hexadecimal = " << num << "\n";
    cout.setf(ios::dec,ios::basefield);
    cout << " decimal = " << num << "\n";
    return 0;
}
```

Output of the above program

```
enter a number
10
decimal = 10
octal = 12
hexadecimal = a
decimal = 10
```

(e) Showbase and Noshowbase Format Flag

(1) Showbase The showbase format flag is used to display the base for octal and hexadecimal numbers. If showbase is set, this flag prefaces integral insertions with the base indicators used with C++ constants. If `hex` is set, for instance, an `0X` will be inserted in front of any integral insertion.

The syntax of the showbase flag is,

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

(2) Noshowbase The noshowbase format flag is used to turn off indicating the notational base in which a number is displayed. By default, noshowbase format flag is set. In other words, the noshowbase manipulator effectively calls the unsetf() function to invalidate the base setting.

The syntax of the noshowbase flag is,

```
cout << noshowbase;
```

PROGRAM 4.21

A program to display the base of the given integer numbers using showbase and noshowbase format flag.

```
//using showbase and noshowbase format flag
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout <<"enter a number\n";
    cin >> num;
    cout.setf(ios::showbase);
    cout << " decimal = " << num << "\n";
    cout.setf(ios::oct,ios::basefield);
    cout << " octal = " << num << "\n";
    cout.setf(ios::hex,ios::basefield);
    cout << " hexadecimal = " << num << "\n";
    cout.setf(ios::dec,ios::basefield);
    cout << " decimal = " << num << "\n";
    cout << noshowbase;
    cout << "calling noshowbase and default is decimal\n";
    cout << " decimal = " << num << "\n";
    return 0;
}
```

Output of the above program

```
enter a number
10
decimal = 10
octal = 012
hexadecimal = 0xa
decimal = 10
calling noshowbase and default is decimal
decimal = 10
```

(f) *Showpos and noshowpos Format Flag*

(1) Showpos The showpos format flag is used to display the sign of an integer. If this flag is set, a "+" sign will be inserted before any integral insertion. It remains unset by default. Note that on positive decimal output, the '+' is assumed, but by default it will not appear. If the number is negative, the '-' sign will always appear.

The syntax of the showpos flag is,

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

(2) Noshowpos The noshowpos format flag causes positive numbers to not be explicitly signed. By default, noshowpos format flag is set. In other words, the noshowpos manipulator effectively calls the unsetf() function to invalidate the setting of a positive sign.

The syntax of the showpos flag is,

```
cout << noshowpos
```

PROGRAM 4.22

A program to show the sign of the given number using the showpos and noshowpos format flag.

```
//using showpos and noshowpos flag
#include <iostream>
using namespace std;
int main()
{
    int num = 71;
    cout << " decimal = " << num << "\n";
    cout.setf(ios::showbase);
    cout.setf(ios::showpos);
    cout.setf(ios::oct,ios::basefield);
    cout << " octal = " << num << "\n";
    cout.setf(ios::hex,ios::basefield);
    cout << " hexadecimal = " << num << "\n";
    cout.setf(ios::dec,ios::basefield);
    cout << " decimal = " << num << "\n";
    cout << "calling noshowpos format flag \n";
    cout << noshowpos;
    cout << " decimal = " << num << "\n";
    return 0;
}
```

Output of the above program

```
decimal = 71
octal = 0107
hexadecimal = 0x47
decimal = +71
calling noshowpos format flag
decimal = 71
```

(g) Uppercase and Nouppercase Format Flag

(1) uppercase The uppercase format flag is used to display output in uppercase. The following notations appear in uppercase

- a hexadecimal number `(A, B, C, D, E and F)`
- the base of a hexadecimal number `(0X)`
- a floating point number in the scientific notation `(4.3E3)`

The syntax of the uppercase format flag is,

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

(2) nouppercase The nouppercase manipulator specifies that hexadecimal digits and the exponent in scientific notation appear in lowercase. In other words, the nouppercase manipulator effectively calls the unsetf () function to revert back to lowercase.

By default, the following notations always appear in lowercase

- a hexadecimal number `(a,b,c,d,e,f)`
- the base of a hexadecimal number `(0x)`
- a floating point number in the scientific notation `(4.3e3)`

The syntax of the nouppercase format flag is,

```
cout << nouppercase;
```

PROGRAM 4.23

A program to display the base of the given hexadecimal number in uppercase using the flag setting.

```
//using uppercase and nouppercase flag
#include <iostream>
using namespace std;
int main()
{
    int num = 9999;
    cout.setf(ios::showbase);
    cout.setf(ios::hex,ios::basefield);
    cout << " hexadecimal = " << num << "\n";
    cout.setf(ios::uppercase | ios::showbase);
    cout.setf(ios::hex,ios::basefield);
    cout << " hexadecimal = " << num << "\n";
    cout << nouppercase;
    cout <<"calling nouppercase and default is lowercase\n";
    cout.setf(ios::hex,ios::basefield);
    cout << " hexadecimal = " << num << "\n";
    return 0;
}
```

Output of the above program

```
hexadecimal = 0x270f
hexadecimal = 0X27F
calling nouppercase and default is lowercase
hexadecimal = 0x270f
```

(h) Formatting Floating Point Numbers The following sections explain how floating values are formatted using the different flag settings in C++.

PROGRAM 4.24

A program to show the floating point numbers without any special formatting.

```
//using field justification
#include <iostream>
using namespace std;
int main()
{
    float a,b,c,d;
    a = 1.23456789;
    b = 34.56;
    c = 1.34E2;
    d = -123.5677;
    cout << "a = " << a << "\n";
    cout << "b = " << b << "\n";
    cout << "c = " << c << "\n";
    cout << "d = " << d << "\n";
    return 0;
}
```

Output of the above program

```
a = 1.23457
b = 34.56
c = 134
d = -123.568
```

(i) Showpoint and noshowpoint Format Flag

(1) Showpoint The showpoint bit format flag is used to show the decimal point for all floating point values. By default, the number of decimal position is six. The showpoint and precision is used to display zeros after the decimal point.

The syntax of the showpoint flag is,

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

(2) Noshowpoint The noshowpoint manipulator flag is used to display only the whole-number part of floating-point numbers whose fractional part is zero. By default, noshowpoint format flag is set.

The noshowpoint manipulator effectively calls the unsetf() flag to undo the showing of all decimal values of a floating point number.

The syntax of the noshowpoint flag is,

```
cout << noshowpoint;
```

PROGRAM 4.25

A program to display a floating point value with all decimal places using showpoint and noshowpoint format flag.

```
//using showpoint and noshowpoint format flag
#include <iostream>
using namespace std;
int main()
{
  float a,b,c,d;
  a = 1.23456789;
  b = 34.000;
```

```
   c = 1.3400;
   d = -123.56770001;
   cout.setf(ios::showpoint);
   cout << "a = " << a << "\n";
   cout << "b = " << b << "\n";
   cout << "c = " << c << "\n";
   cout << "d = " << d << "\n";
   cout << "calling noshowpoint \n";
   cout << noshowpoint;
   cout << "a = " << a << "\n";
   cout << "b = " << b << "\n";
   cout << "c = " << c << "\n";
   cout << "d = " << d << "\n";
   return 0;
}
```

Output of the above program

```
a = 1.23457
b = 34.0000
c = 1.34000
d = -123.568
calling noshowpoint
a = 1.23457
b = 34
c = 1.34
d = -123.568
```

(j) Precision The precision member function is used to display the floating point value as defined by the user. The general syntax of the precision is,

```
cout.precision (int n)
```

where `n` is the number of decimal places of the floating value to be displayed. For example,

```
cout.precision(5);
```

which displays a floating point number of five decimals.

PROGRAM 4.26

A program to display a floating point value in the formatted form using the showpoint and precision member function.

```
//using showpoint and precision member function
#include <iostream>
using namespace std;
int main()
{
    float a,b,c,d;
    a = 1.23456789;
    b = 34.56;
    c = 1.34E2;
    d = -123.5677;
    cout.setf(ios::showpoint);
    cout.precision(5);
    cout << "a = " << a << "\n";
    cout << "b = " << b << "\n";
    cout << "c = " << c << "\n";
    cout << "d = " << d << "\n";
    return 0;
}
```

Output of the above program

```
a = 1.2346
```

```
b = 34.560
c = 134.00
d = -123.57
```

(k) Floatfield Bit Format Flag Sometimes a floating point value may have to be displayed in scientific notation rather than in fixed format.

(1) Scientific When scientific is set, floating point values are inserted using scientific notation. There is only one digit before the decimal point followed by the specified number of precision digits which in turn is followed by an uppercase or a lowercase e depending on the setting of uppercase and the exponent value.

The general syntax of the scientific notation is,

```
cout.setf(ios::scientific, ios::adjustfield);
```

(2) Fixed When fixed is set, the value is inserted using decimal notation with the specified number of precision digits following the decimal point. If neither scientific nor fixed is set (the default), scientific notation will be used when the exponent is less than 4 or greater than precision. Otherwise, fixed notation is used.

The general syntax of the fixed notation is,

```
cout.setf(ios::fixed, ios::adjustfield);
```

PROGRAM 4.27

A program to display a floating point value in both scientific and fixed notation using the floatfield format flag.

```
//using fixed and scientific flag
#include <iostream>
using namespace std;
int main()
{
   float a,b,c,d;
   a = 1.23456789;
   b = 34.56;
   c = 1.34E2;
   d = -123.5677;
   cout.setf(ios::showpoint);
   cout.precision(4);
   cout.setf(ios::fixed, ios::floatfield);
   cout << " display in conventional notation \n";
   cout << "a = " << a << "\n";
   cout << "b = " << b << "\n";
   cout << "c = " << c << "\n";
   cout << "d = " << d << "\n";
   cout.setf(ios::scientific, ios::floatfield);
   cout << " display in scientific notation \n";
   cout << "a = " << a << "\n";
   cout << "b = " << b << "\n";
   cout << "c = " << c << "\n";
   cout << "d = " << d << "\n";
   return 0;
}
```

Output of the above program

```
display in conventional notation
a = 1.2346
b = 34.5600
c = 134.0000
d = -123.5677
```

```
display in scientific notation
a = 1.2346e+00
b = 3.4560e+01
c = 1.3400e+02
d = -1.2357e+02
```

(l) Adjustfield Bit Format Flag The adjust field consists of three field settings

`ios::left`	left justification
`ios::right`	right justification
`ios::internal`	pad after sign or base indicator

(1) Left Only one of these may be set at any time. If left is set, the inserted data will be flush left in a field of characters widthwide. The extra space, if any, will be filled by the fill character (specified by the fill function).

The general syntax of the left justification format flag is,

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

(2) Right If right is set, the inserted data will be flush right. The general syntax of the right justification format flag is,

```
cout.setf(ios::right,ios::adjustfield);
```

(3) Internal If `internal` is set, the sign of a numeric value will be flush left while the numeric value flush right, and the area between them will contain the pad character. The general syntax of the internal justification format flag is,

```
cout.setf(ios::internal,ios::adjustfield);
```

PROGRAM 4.28

A program to demonstrate how a field justification of an integer value is carried out using the adjust field format flag.

```
//using field justification
#include <iostream>
using namespace std;
int main()
{
    int num = 71;

    cout.fill('*');
    cout.setf(ios::showpos);
    cout.setf(ios::left,ios::adjustfield);
    cout.width(6);
    cout << num << "\n";
    cout.setf(ios::right,ios::adjustfield);
    cout.width(6);
    cout << num << "\n";
    cout.setf(ios::internal,ios::adjustfield);
    cout.width(6);
    cout << num << "\n";
    return 0;
}
```

Output of the above program

```
+71***
***+71
+***71
```

(m) Fill and Width If the total number of characters needed to display a field is less than the current field width, the extra output spaces will be filled with the current fill character. In C++, it is permitted to use any character to serve as the fill character. But by default it is blank.

(1) Fill The fill() member function is used to specify a new fill character. Once it is specified, it remains as the fill character unless it is changed.

The general syntax of the fill() member function is,

```
cout.fill(char ch);
```

where `ch` is a character to be filled. For example,

```
cout.fill('*');
cout.fill('0');
```

(2) Width The width() member function is used to specify the size of the data variable.

The general syntax of the width() member function is,

```
cout.width(int n);
```

where `n` is a total field width of a variable. For example,

```
cout.width(10);
cout.width(3);
```

PROGRAM 4.29

A program to demonstrate how a fill and width member function is used to display a numeral in C++.

```
//using fill and width
#include <iostream>
using namespace std;
int main()
{
  cout.width(10);
  cout.fill('x');
  int num = 6;
  cout << num << " \n";
  cout.width(15);
  cout.fill('b');
  num = 12345;
  cout << num << endl;
  return 0;
}
```

Output of the above program

```
xxxxxxxxx6
bbbbbbbbbb12345
```

(n) Stdio This flag flushes the stdout and stderr devices defined in stdio.h. This is unset by default.

The general syntax of the stdio flag is,

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

```
//using stdio
#include <iostream>
using namespace std;
int main()
{
    cout.setf(ios::stdio);
    cout << " this is a test program to flush out \n";
    cout << " the input stream\n";
}   return 0;
```

(o) Skipws and Noskipws The non-printable character such as space bar, tab character, form feed or newline and backspace are called as whitespaces.

(1) Skipws The skipws format flag cause spaces to not be read by the input stream. In other words, if skipws format flag is set, leading white space is ignored on extraction. By default skipws is set. The manipulator effectively calls setf(ios:: skipws).

PROGRAM 4.30

A program to demonstrate how to use the skipws format flag for reading a set of strings with white spaces.

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
    string s1, s2, s3;
    cout << "Enter three names: ";
    cin >> skipws >> s1 >> s2 >> s3;
    cout << "." << s1  << "." << endl;
    cout << "." << s2 << "." << endl;
    cout << "." << s3 << "." << endl;
}   return 0;
```

Output of the above program

```
Enter three names:
Bangalore Mumbai Hyderabad
.Bangalore.
.Mumbai.
.Hyderabad.
```

(2) Noskipws The noskipws format flag cause spaces to be read by the input stream. The manipulator effectively calls unsetf (ios:: skipws),

PROGRAM 4.31

A program to demonstrate how to use the noskipws format flag for reading a set of strings with white spaces.

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
    string s1, s2, s3;
    cout << "Enter three names: ";
    cin >> noskipws >> s1 >> s2 >> s3;
    cout << "." << s1  << "." << endl;
    cout << "." << s2 << "." << endl;
    cout << "." << s3 << "." << endl;
}   return 0;
```

Output of the above program

```
Enter three names:
Bangalore Mumbai Hyderabad
.Bangalore.
..
..
```

The noskipws reads whitespace as an input and therefore the above output is displayed.

(p) Unitbuf and Nounitbuf

(1) unitbuf The unitbuf format flag causes output to be processed when the buffer is not empty. In other words, when unitbuf is set, the stream is flushed after every insertion. This flag is unset by default.

The general syntax of the unitbuf is,

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

(2) nounitbuf The nounitbuf format flag causes output to be buffered and processed on when the buffer is full. The manipulator effectively calls unsetf (ios:: unitbuf),

```
//using unitbuf
#include <iostream>
using namespace std;
int main()
{
    cout.setf(ios::unitbuf);
    cout << " this is a test program to flush out \n";
    cout << " the input stream\n";
    return 0;
}
```

Output of the above program

this is a test program to flush out the input stream

The following table summarises the constants used with setf() manipulators.

Arguments for setflags

First argument (flag name)	*Second argument*
`ios:: skipws`	

`ios::left`
`ios::right`
`ios::internal`
	`ios::adjustfield`

`ios::dec`
`ios::oct`
`ios::hex`
	`ios::basefield`

`ios::boolalpha`
`ios::showbase`
`ios::showpos`
`ios::uppercase`
`ios::showpoint`	

`ios::scientific` `ios::fixed`	`ios::floatfield`
`ios::unitbuf` `ios::stdio`	

REVIEW QUESTIONS

1. How are the variables or the user defined identifiers declared in C++?
2. What is an expression? How is a n expression different from the variables?
3. What are the different types of I/O streams used in C++?

4. Explain the salient features of the following header files.
   (i) input and output stream class `(iostream>`
   (ii) input and out manipulator `(iomanip>`
   (iii) input and output `(ios)`
5. What is a manipulator? List the merits and demerits of it.
6. Explain the syntactic rules governing `cin` and `cout`.
7. Explain the following:
   (i) `istream`
   (ii) `ostream`
   (iii) `iostream`
8. What are the various standard manipulators used in C++?
9. How is the comment statement represented in C++? What are the uses of defining a comment statement in a program?
10. Explain the syntactic rules governing the following manipulators.
    (i) `setbase` (ii) `setfill`
    (iii) `setw` (iv) `setprecision`
11. How is the basefield of a numeral defined in C++?
12. What is an I/O stream flag? What stream flags are more suitable than standard manipulator?
13. List the various I/O stream flags and their meanings.
14. Explain the syntactic rules governing the following I/O stream flags.
    (i) `showbase` (ii) `showpoint`
    (iii) `showpos` (iv) `scientific`
15. What is an adjust field format flag?
16. Explain the various kinds for formatting of a floating point value used in C++?

CONCEPT REVIEW PROBLEMS

1. Determine the output of each of the following program when it is executed.
   (a)

```
#include <iostream>
using namespace std;
int main()
{
    int x(10);
    cout << "value of x = " << x << endl;
    return 0;
}
```

   (b)

```
#include <iostream>
using namespace std;
int main()
{
    int x(10),y(20),sum(0);
    sum = x+y;
    cout << " x = " << x <<"  y = "<< y;
    cout << "  sum = " << sum << endl;
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10,y = 20,sum = 0;
    sum = x+y;
    cout << " x = " << x <<"  y = "<< y;
    cout << "  sum = " << sum << endl;
    return 0;
}
```

(d)

```
#include <iostream>
int main()
{
    using namespace std;
    bool b = cout.bad();
    cout << b << endl;
    b = cout.good( );
    cout << b << endl;
    return 0;
}
```

(e)

```
#include <iostream>
int main(void)
{
    using namespace std;
    double i = 1.23e100;
    cout << i << endl;
    cout << uppercase << i << endl;
    int j = 10;
    cout << hex << nouppercase << j << endl;
    cout << hex << uppercase << j << endl;
}   return 0;
```

(f)

```
#include <iostream>
int main()
{
    using namespace std;
    int j = 100;
    cout << showbase << j << endl;
    cout << hex << j << showbase << endl;
    cout << oct << j << showbase << endl;
    cout << dec << j << noshowbase << endl;
    cout << hex << j << noshowbase << endl;
    cout << oct << j << noshowbase << endl;
}   return 0;
```

(g)

```
#include <iostream>
int main()
{
    using namespace std;
    bool b = true;
    cout << b << endl;
```

```
    boolalpha( cout );
    cout << b << endl;
    noboolalpha( cout );
    cout << b << endl;
    cout << boolalpha << b << endl;
}   return 0;
```

(h)

```
#include <iostream>
int main()
{
    using namespace std;
    cout << !cout << endl;
}   return 0;
```

(i)

```
#include <iostream>
int main()
{
    using namespace std;
    cout << (bool)&cout << endl;
}   return 0;
```

2. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    bool a = true;
    bool b = true;
    int c = a-b;
    cout << c << endl;
}   return 0;
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    bool a = true;
    bool b = true;
    int c = a+b;
    cout << c << endl;
}   return 0;
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    bool a = true;
    bool b = false;
    int c = a*b;
    cout << c << endl;
}   return 0;
```

(d)

```
#include <iostream>
using namespace std;
```

```
int main()
{
    bool a = 20;
    bool b = 210;
    int c = a+b;
    cout << c << endl;
}   return 0;
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    bool a = 0;
    bool b = -1;
    int c = a+b;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = a+b = "<< c << endl;
}   return 0;
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    bool a = -200;
    bool b = -100;
    int c = a+b;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = a+b = "<< c << endl;
}   return 0;
```

(g)

```
#include <iostream>
using namespace std;
int main()
{
    bool a = -200;
    bool b = -200;
    bool c = a+b;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = a+b = "<< c << endl;
}   return 0;
```

(h)

```
#include <iostream>
using namespace std;
int main()
{
    bool a = 0;
    bool b = 20;
    bool c = a+b;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = a+b = "<< c << endl;
```

```
    }   return 0;
```

(i)

```
#include <iostream>
using namespace std;
int main()
{
    bool a = 10;
    bool b = false;
    int x = a-b;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "x = a-b = " << x << endl;
    return 0;
}
```

(j)

```
#include <iostream>
using namespace std;
int main()
{
    wchar_t ch1 = L'A';
    wchar_t ch2 = L'B';
    wchar_t ch3 = L'C';
    wcout << ch1;
    wcout << ch2;
    wcout << ch3;
    return 0;
}
```

(k)

```
#include <iostream>
using namespace std;
int main()
{
    wchar_t ch1 = L'A';
    wchar_t ch2 = L'B';
    wchar_t ch3 = L'C';
    wcout << ch1 << endl;
    wcout << ch2 << endl;
    wcout << ch3 << endl;
    return 0;
}
```

(l)

```
#include <iostream>
using namespace std;
int main()
{
    wchar_t s1[] = L"Hello";
    wchar_t s2[] = L"C++";
    wchar_t s3[] = L"program";
    wcout << s1 << endl;
    wcout << s2 << endl;
    wcout << s3 << endl;
    return 0;
}
```

PROGRAMMING EXERCISES

1. Write a program in C++ to perform the following:
   (i) Area of a circle
   (ii) Circumference of a circle
   (iii) Area of a triangle
   (iv) Area of a rectangle
2. Write a program in C++ to find the simple interest for a given principal, rate of interest, and number of years.
3. Write a program in C++ to find the compound interest for a given principal, rate of interest and number of years.
4. Write a program in C++ to solve a quadratic equation.
5. Develop a program in C++ to read a set of five real numbers from the keyboard and to find out their sum and average.
6. Write a program in C++ to read the name, age, sex, height and weight of a student and to display with proper heading for each variable.
7. Write a program in C++ to accept a single character from the keyboard. Using `printf`, display the character or keystroke and its decimal, hexadecimal and octal values.
   Display in the following format

```
Character       Decimal     Hexadecimal     Octal
```

Chapter 5

Control Statements

This chapter mainly gives how to implement a control statement for decision making purpose in a program. The various types of conditional expressions that are used for the multiway selection or decision making namely, if, if-else and switch-case statements are explained with numerous illustrative examples in this chapter. The different kinds of loop statements such as for loop, while loop, and do-while loop are discussed in this chapter. This chapter also presents how to use the goto statement, the break statement and the continue statement that are used for breaking the control within the loop statements.

5.1 CONDITIONAL EXPRESSIONS

Conditional statements are a set of C++ statements that are used for checking the truth of one or more conditions and perform different set of calculations, depending on the conditions. In other words, conditional expressions are mainly used for decision making or selecting a particular portion of a program for execution. In the subsequent sections, the structures of the control statements and the different usages are explained. The following statements are used to perform the task of conditional operations in C++:

- The `if` statement
- The `if-else` statement
- The `switch-case` statement

5.1.1 The `if` Statement

The simplest of the conditional statement is the `if` statement. A condition or logical expression is tested for selecting a particular portion of a program for execution. If the condition is true, a particular segment of the program is executed. The segment may consist of either single statement(s) or a compound statement(s) enclosed in a pair of `{` and `}` braces as delimiters. If the condition is false, the particular segment is passed over or ignored.

The general syntax of the `if` statement is:

```
if (expression)
    statement;
```

where `expression` is a simple or a compound condition for testing the if statement. The expression can be any valid statement such as arithmetic, logical or boolean statement which evaluates to either 'true' or 'false'.

The general syntax of the `if` for a block of statements is:

```
if (expression)
{
    statement_1;
    statement_2;
    -------
    -------
    statement_n;
}
```

The expression is evaluated and if it is 'true', then the statement following the 'if' is executed. In case, the given expression is 'false', then the statement is skipped and execution continues with the next statement.

Consider the following program segment which shows how to use a simple `if` statement in C++:

```
#include <iostream>
using namespace std;
int main()
{
    int a,b;
    a = 20;
    b = 10;
    if (a > b)
       cout << " Largest value = " << a << endl;
    return 0;
}
```

If the given condition evaluates to true, then the computer will print the message 'Largest value = 20' and if not, it will simply skip the statement `(cout statement)`.

Another example is given below which highlights the execution of how to represent a block of statements using the `if` condition. The compound or block of statements are always considered as a single statement in C++. It is necessary to use the braces begin `({)` and end `(})` as delimiter.

```
#include <iostream>
using namespace std;
int main()
{
    int a,b;
    a = 5;
    b = 2;
    if (a > b)
    {
       cout << " one " << endl;
       cout << " two " << endl;
       cout << " three " << endl;
    }
    return 0;
} // end of the main
```

Output of the above program

```
one
two
three
```

If the condition is 'false', the entire block of if-statements are skipped or ignored by the C++ compiler.

5.1.2 The `if-else` Statement

The general form of the syntax and a flow chart showing the effects of an if-else are given in Fig. 5.1. The if-else statement is one of the most widely used conditional expressions in C++.

The general syntax of the if–else conditional expression is given below:

```
if (expression)
    statement_1;
else
    statement_2;
```

If the condition is 'true', then the statement_1 or the block of statements between the expression will be executed. If it is 'false', statement_2 or the block of statements between else and end will be performed. This differs from the 'if' statement where there is only one option that is either execution or skipping of a statement or a block of statements.

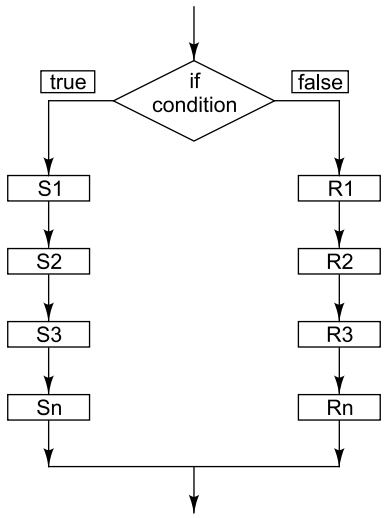


Fig. 5.1(a) *Flow chart for if–else statement*

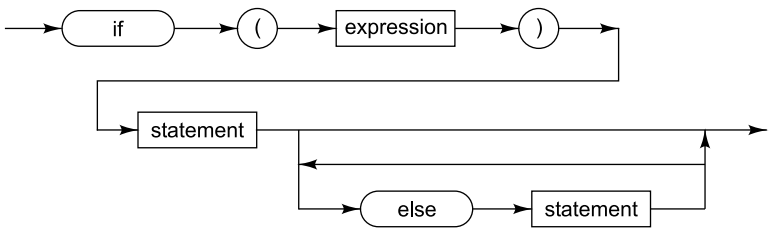


Fig. 5.1(b) *Syntax diagram of* `if-else` *statement*

The 'if-else' statement begins with the keyword 'if' followed by a boolean expression. This is followed by a C++ statement or a group of statements. Finally, the reserved word 'else' is written, again followed by a C++ statement or a group of statements. Note that there is a semicolon used in each statement that gives the meaning of termination of the statement in C++.

The 'if-else' statement is executed as follows. The boolean expression is first evaluated and if it is true, then statement_1 is executed and statement_2 is ignored. If it is false, then the statement_1 is skipped

and statement_2 is executed. In this case, either of the two statements are executed depending upon the evaluated value of the expression.

For example, the following 'if-else' structure is used to check whether a given number is odd or even:

```
#include <iostream>
using namespace std;
int main()
{
    int number;
    cout << "enter a number " << endl;
    cin >> number;
    if ( (number % 2) == 0 )
          cout << "number is even " << endl;
    else
          cout << " number is odd " << endl;
    return 0;
}
```

Output of the above program

```
enter a number
10
number is even
```

When the expression (number % 2 == 0) is executed and found true, the message 'number is even' will be displayed. If the expression is false, then the else part of the statement that is the message 'number is odd' will be displayed.

To represent a block of statements, the use of the keywords begin and end are essential. For example, if-else conditional expression is realised with a block of statements as given below:

```
if (expression)
{
    statement_1;
    statement_2;
    -------
    -------
}
else
{
    statement_1;
    statement_2;
    -------
    -------
}
```

The syntactic ambiguity The else part is optional in the if-else structure. Omitting the else part leads to confusion especially in a nested if-else sequence.

For example, consider the following if-else construction:

```
if (expression_1)
  if (expression_2 )
       statement_1;
else
       statement_2;
```

In the above if-else construction, the else part gives a syntactic ambiguity. That is, it is not clear whether the else part belongs to the inner if statement or to the outer if statement. The syntactic ambiguity of the above if-else construction can be resolved by interpreting the construction as equivalent to:

```
if (expression_1)
{
   if (expression_2)
        statement_1;
   else
      statement_2;
}
```

Note that the else statement without begin–end statements leads to confusion, like, whether it belongs to the inner if or outer if statement. Normally, begin({) and end(}) statements are used to clearly indicate proper association.

Special features of the if–else statement

Note that there is a semicolon after each of the statement but not after the if (expression). In other words, if we have

```
if ( a >b);
  temp = a;
```

then the C++ compiler will interpret it as

```
if ( a > b)
     ;
temp = a;
```

which is meaningless if the actual intention is to write

```
if ( a > b)
    temp = a;
```

The following are some of the sample if-else constructions that are most widely used:

(1)

```
if ( expression )
{
    -------
    -------
}
else
{
    -------
    -------
}
```

(2)

```
if (expression)
{
   if (expression)
   {
       -------
       -------
   }
   else
   {
       -------
       -------
   }
}
else
{
       -------
       -------
}
```

(3)

```
if (expression)
{
    if (expression)
    {
        -------
        -------
    }
    else
    {
        -------
        -------
    }
}
else
{
    if (expression)
    {
        -------
        -------
    }
    else
    {
        -------
        -------
    }
}
```

The generalised if structure permits us to make a multiway decision, the syntax of which is as:

```
if (expression)
{
    statement 1
    statement 2
    -------
    -------
}
else if (expression)
{
    statement 1
    statement 2
    -------
    -------
}
else if (expression)
{

    statement 1
    statement 2
    -------
    -------
}
else
{
    statement 1
    statement 2
    -------
```

```
        -------
    }
```

PROGRAM 5.1

A program to read any two numbers from the keyboard and display the largest value of them.

```
// example 5.1
// if-else structure
#include <iostream>
using namespace std;
int main()
{
    float a,b;
    cout <<" Enter any two numbers " << endl;
    cin >> a >> b;
    if ( a > b)
       cout <<" Largest value is = " << a << endl;
    else
       cout <<" Largest value is = " << b << endl;
    return 0;
}
```

Output of the above program

Enter any two number

```
10   20
Largest value is = 20
```

PROGRAM 5.2

A program to find the largest value of any three numbers.

```
// example 5.2
#include <iostream>
using namespace std;
int main()
{

    float a,b,c;
    cout <<" Enter any three numbers " << endl;
    cin >> a >> b >> c;
    if ( a > b)
    {
        if (a > c)
            cout <<" Largest value is " << a;
        else
            cout <<" Largest value is = " << c;
    }
    else
    {
        if ( b > c)
            cout <<" Largest value is = " << b;
        else
            cout <<" Largest value is = " << c;
    }
    return 0;
} // end of the main program
```

Output of the above program

Enter any three numbers

```
10  20  30
Largest value is = 30
```

PROGRAM 5.3

A program to find the largest value among any four numbers.

```
//  example 5.3
#include <iostream>
using namespace std;
int main()
{
    float a,b,c,d;
    cout <<" Enter any four numbers " << endl;
    cin >> a >> b >> c >> d;
    if ( a > b)
    {
       if ( a > c)
       {
            if ( a > d )
               cout <<" largest = " << a;
            else
               cout <<" largest = " << d;
       }
       else
       {
            if ( c > d )
               cout <<" largest = " << c;
            else
               cout <<" largest = " << d;
       }
    } // end of outer most if part
    else
    {
       if ( b > c )
       {
            if ( b > d)
               cout <<" largest = " << b;
            else
               cout <<" largest = " << d;
       }
       else
       {
            if ( c > d )
               cout <<" largest = " << c;
            else
               cout <<" largest = " << d;
       }
    } // end of outer most else part
    return 0;
} // end of the main program
```

Output of the above program

Enter any four numbers

```
10   20   30   40
largest = 40
```

PROGRAM 5.4

A program to display the name of the day in a week, depending upon the number which is entered by the keyboard using the if–else structure.

```
// example 5.4
#include <iostream>
using namespace std;
int main()
{
    int day;
    cout <<" Enter a number between 1 and 7 " << endl;
    cin >> day;
    if ( day == 1)
       cout <<" Monday " << endl;
    else if ( day == 2)
       cout <<" Tuesday " << endl;
    else if ( day == 3)
       cout <<" Wednesday " << endl;
    else if ( day == 4)
       cout <<" Thursday " << endl;
    else if ( day == 5)
       cout <<" Friday " << endl;
    else if ( day == 6)
       cout <<" Saturday " << endl;
    else if ( day == 7)
       cout <<" Sunday " << endl;
    else
       cout <<" Enter a correct number " << endl;
    return 0;
} // end of main program
```

Output of the above program

Enter a number between 1 and 7

```
7
Sunday
```

Nested `if-else` structure

Whenever an if statement is constructed within another if statement, it is called as a nested if statement. C++ permits one to realise nested if statements to any level of nesting with or without else clauses.

The general syntax of the nested if-without else part, is given below :

```
if (boolean expression_1)
    if (boolean expression_2)
          if (boolean expression_3)
              -------
              -------
              if (boolean expression_n)
                  statement;
```

The statement inside the nested `if` conditions will be executed, if and only if, the value of every boolean expression evaluates to true. Furthermore, the conditional expression is evaluated in the order of boolean expression_1, boolean expression_2 and so on. If one of the boolean expression gets evaluated as false, then further execution of the boolean expression is stopped automatically.

```
#include <iostream>
using namespace std;
int main()
{
```

```
    int math,physics,chemistry;
    math = 60;
    physics = 45;
    chemistry = 70;
    if ( math >= 50)
        if ( physics >= 45)
            if ( chemistry >= 45)
                cout << " pass  " << endl;
            else
                cout << " fail " << endl;
    return 0;
}
```

Output of the above program

```
pass
```

The nested if condition can be realised in a single boolean expression using boolean operators in a compound boolean expression. For example, the following program segment illustrates how to implement a single boolean expression with compound boolean conditions:

```
#include <iostream>
using namespace std;
int main()
{
    int math,physics,chemistry;
    math = 60;
    physics = 45;
    chemistry = 70;
    if ( (math >= 50) && (physics >= 45) && (chemistry >= 45))
        cout << " pass  " << endl;
    else
        cout << " fail " << endl;
    return 0;
}
```

PROGRAM 5.5

A program to solve a quadratic equation and to find out the different types of roots for given coefficients.

```
// example 5.5
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    float a,b,c,det;
    float x1,x2,x;
    cout <<"enter the coefficients for a,b,c " << endl;
    cin >> a >> b >> c;
    if (a == 0 )
    {
        x = -c/b;
        cout <<"roots are linear = " << x;
    }
    else
    {
        det = b*b-4*a*c;
        if (det == 0)
        {
```

```
            x = -b/(2*a);
            cout <<"roots are identical(x1 = x2 ) " << endl;
            cout << x;
        }
        else if ( det < 0)
            cout <<"roots are complex" << endl;
        else
        {
            x1 =(-b+sqrt(det))/(2*a);
            x2 =(-b-sqrt(det))/(2*a);
            cout <<"roots are real " << endl;
            cout <<" x1 = " << x1 << endl;
            cout <<" x2 = " << x2 << endl;
        }
    } // end of outer else part
    return 0;
} // end of the main program
```

Output of the above program

```
enter the coefficients for a,b,c
0   4    2
roots are linear = -0.5
enter the coefficients for a,b,c
1   4   2
roots are real
x1 = -0.585786
x2 = -3.41421
enter the coefficients for a,b,c
2   1   4
roots are complex
enter the coefficients for a,b,c
2  4  2
roots are identical(x1 = x2 )
-1
```

PROGRAM 5.6

A program to compare two given characters such as equal to, less than or greater than using if–else statements.

```
// example 5.6
#include <iostream>
using namespace std;
int main()
{
    char ch1,ch2;
    cout << "enter a first character" << endl;
    ch1 = cin.get();
    cin.ignore(); // skip a newline character
    cout << "enter a second character" << endl;
    ch2 = cin.get();
    cout << "ch1 = " << ch1 << endl;
    cout << "ch2 = " << ch2 << endl;
    if (ch1 > ch2 )
        cout << "ch1 > ch2 " << endl;
    else if (ch1 < ch2 )
        cout << "ch1 < ch2 " << endl;
    else
        cout << "ch1 = ch2" << endl;
    return 0;
}
```

Output of the above program

```
enter a first character
a
enter a second character
a
ch1 = a
ch2 = a
ch1 = ch2

enter a first character
a
enter a second character
b
ch1 = a
ch2 = b
ch1 < ch2

enter a first character
b
enter a second character
a
ch1 = b
ch2 = a
ch1 > ch2
```

5.1.3 The switch-case Statement

The switch-case statement is a special multiway decision maker that tests whether an expression matches one of the number of the constant values, and perform a part of a statement or a block of statements within the case structure. The general form of the syntax and flow chart diagram of the switch-case statement is given in Fig. 5.2. A switch case structure is one such statement in C++ wherein the delimiter begin '{' and end '}' should be used.

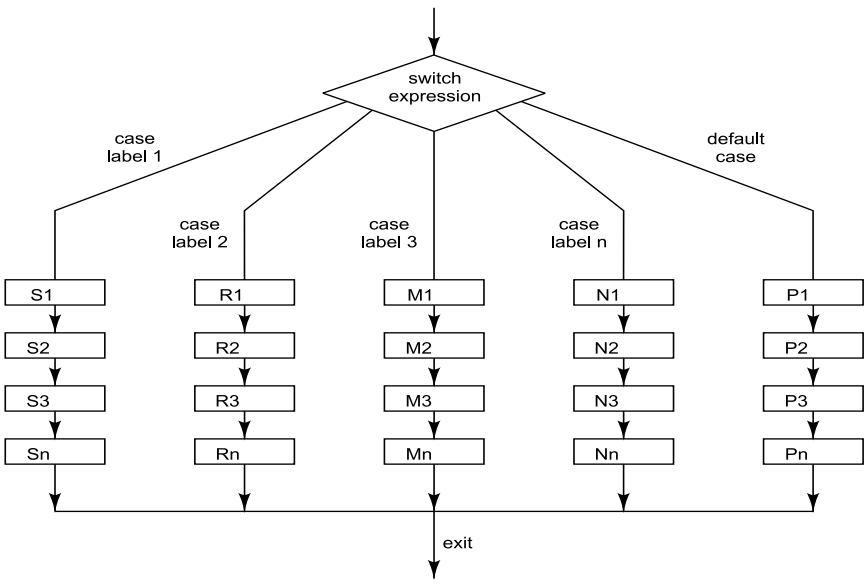


Fig. 5.2(a) *Flow chart for Switch-case Statement*

The general syntax of the case statement is:

```
switch (expression) {
   case constant_1
        statement
   case constant_2
        statement
   -------
   -------
   -------
   -------
   case constant_n
        statement
   default
        statement
} // end of switch-case statement
```

The case statement consists of an expression (the selection) and a list of statements, each being associated with one or more constant values of the type of the selector. Sometimes case label is called as a *case constant*. Note that case labels are entirely different from the program labels, which are used along with a `goto` statement.

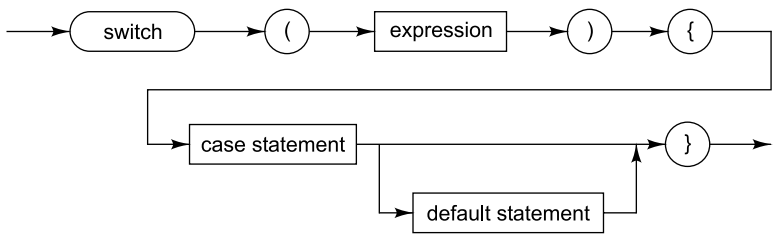


Fig. 5.2(b) *Syntax Diagram of Switch-case Statement*

The expression whose value is being compared, can be any valid expression including the value of the variable, an arithmetic expression, logical comparison, bitwise expression or the return value from a function call, but not a floating point expression. The constants in each of the case statements must obviously be of the same type. The expression value is checked against each of the specified cases and when a match occurs, the statement following that is executed. Again, to maintain generality the statement can be either a simple or a compound statement.

The value that follows the keyword `case` can only be constants and cannot be expressions. They may be integers or characters, but not floating point numbers or character strings. If it is necessary to check the value of an expression against another expression, then the `if-else` structure should be used. Case statements are generally used for multiway decision making only but not for repetitive purpose.

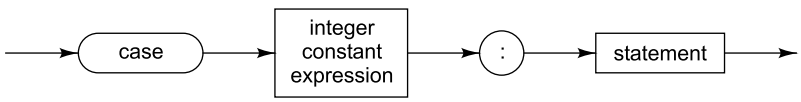


Fig. 5.2(c) *Syntax Diagram of Case Statement*

The last case of this statement which is called the *default case* is optional and should be used according to the program's specific requirement. It is analogous to the last `else` of the `if-else-if` structure.

Although the default case is optional its position is fixed; it must follow all the other cases. Typically, it can be used for error trapping or impossible cases. If the default case is not included in a switch statement and the expression is not matched by any other cases, nothing happens and the statement following the switch constant is executed.

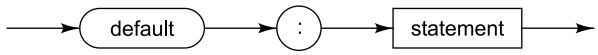


Fig. 5.2(d) *Syntax Diagram of Default Case Statement*

Execution of the switch constant in C++ follows a different logic. No statements are executed until a case has been matched or the default case has been encountered. However, it continues to execute all statements once a case has been matched, irrespective of the fact whether those statements belong to the case that has been matched or not.

As an example, consider a program segment written in C++:

```
input_ch = cin.get();
switch (input_ch) {
    case 'R':
          cout << "Red " << endl;
    case 'W':
       cout << "White " << endl;
    case 'B':
          cout << "Blue " << endl;
}
```

Due to the fall through effect, when `input_ch` is `'R'` the `Red, White` and `Blue` will be printed, when the choice is `'W'`, then `White` and `Blue` will be printed and when the choice is `'B'`, only `Blue` will be printed.

C++ offers a method of overcoming this side effect with the help of the break statement. The break statement causes an immediate exit from the switch construct. As the default case is placed at the end of the case constant it is not necessary to include a break statement there. However, in general it is advisable to use the break statement explicitly whenever exclusion of a case statement is required. Thus, the general format of the mutually exclusive switch constant is:

```
switch (expression) {
    case constant_1 :
              statement_1
              -------
              -------
              break;
    case constant_2 :
              statement_1
              -------
              -------
              break;
              -------
              -------
    case constant_n :
              -------
              -------
              statement_1
              break;
} // end of switch-case structure
```

Although there is no clear cut rule for the use of the `if-else` and `switch` constants, it is easy to use the former for isolating ranges in the expression being evaluated, while the latter is used for decision based on discrete values along with one default case for everything else.

Thus, if in a program for intercepting vowels out of a stream of characters it is required to print them, it would be more meaningful to make use of the switch constant than the `if-else` constant. The following example will help in understanding this, along with the use of the `break` statement.

```
#include <iostream>
using namespace std;
int main()
{
    char input_ch;
    cout << "enter a character" << endl;
    input_ch = cin.get();
    switch (input_ch) {
        case 'A' :
        case 'a' :
              cout << "A " << endl;
              break;
        case 'E' :
        case 'e' :
              cout << "E " << endl;
              break;
        case 'I' :
        case 'i' :
              cout << "I " << endl;
              break;
        case 'O' :
        case 'o' :
              cout << "O " << endl;;
              break;
        case 'U' :
        case 'u' :
              cout << "U " << endl;
              break;
        default :
              cout << "Not a vowel " << endl;
              break;
    }    // end of the switch-case structure
    return 0;
}
```

The above case structure can easily be realised with multiple `if-else` structure which is given below:

```
if ((inputchar == 'a') || (inputchar == 'A'))
     cout << "A " << endl;
else if ((inputchar == 'e') || (inputchar == 'E'))
     cout << "E " << endl;
else if ((inputchar == 'i') || (inputchar == 'I'))
     cout << "I " << endl;
else if ((inputchar == 'o') || (inputchar == 'O'))
     cout << "O " << endl;
else if ((inputchar == 'u' ) || (inputchar == 'U'))
     cout << "U " << endl;
else
     cout << "Not a vowel " << endl;
```

Invalid case expressions For example, following are some invalid constructions of the case structure in C++:

(1) The case label must not be a floating point number. Case labels should be either integer constants or character constants.

```
switch (value) {       // invalid
    case '10.11' :
         -------
         -------
         break;
    case '10.1111' :
         -------
         -------
         break;
    default :
         -------
         -------
 } // end of the switch-case structure
```

(2) The case label must not be a string expression or a string of characters.

```
switch (value) {       // invalid
    case "good" :
         -------
         -------
         break;
    case "bad" :
         -------
         -------
         break;
    default :
         -------
         -------
 } // end of the switch-case structure
```

(3) The case label cannot be an expression using arithmetic, logical or combination of all.

```
 switch (expression) {       // invalid
     case 'a+b' :
          -------
          -------
          break;
     case 'a-b':
          -------
          -------
          break;
     default :
          -------
          -------
 } // end of the switch-case structure
```

(4) The case label or case constant must not be repeated in the same case construction.

```
switch (expression) {       // invalid
    case 1,2 :
         -------
         -------
         break;
    case 3,4,5:
         -------
         -------
```

```
            break;
        case 2,6,7 :
            -------
            -------
            break;
        default :
            -------
            -------
    } // end of the switch-case structure
```

Note that case label 2 is repeated twice in the above example and hence, it is invalid.

(5) The case label or case constant cannot have a range of numbers.

```
switch (expression) {       // invalid
    case 1..2 :
        -------
        -------
        break;
    case 3..5:
        -------
        -------
        break;
    case 6..9 :
        -------
        -------
        break;
    default :
        -------
        -------
} // end of the switch-case structure
```

The above case construction is invalid, as the case constants have been defined using a range of numbers.

PROGRAM 5.7

A program to display the name of the day in a week, depending upon the number entered through the keyboard using the switch-case statement.

```
// example 5.7
#include <iostream>
using namespace std;
int main()
{
    int day;
    cout << "Enter a number between 1 and 7 " << endl;
    cin >> day;
    switch (day) {
        case 1 :
            cout << " Monday " << endl;
            break;
        case 2 :
            cout << " Tuesday " << endl;
            break;
        case 3 :
            cout << " Wednesday " << endl;
            break;
        case 4 :
            cout << " Thursday " << endl;
            break;
```

```
        case 5 :
             cout << " Friday " << endl;
             break;
        case 6 :
             cout << " Saturday " << endl;
             break;
        case 7 :
             cout << " Sunday " << endl;
             break;
        default :
             cout << " enter a correct number " << endl;
             break;
    } // end of the switch-case statement
    return 0;
}  // end of main program
```

Output of the above program

Enter a number between 1 and 7
1
Monday

PROGRAM 5.8

A program to perform simple arithmetic operations such as addition, subtraction, multiplication, and division using the switch-case statement.

```
//  example 5.8
#include <iostream>
using namespace std;
int main()
{
    float x,y,z;
    char ch;
    cout << "enter any two numbers" << endl;
    cin >> x >> y;
    cin.ignore(); //skip a line feed character
    cout << "enter any one of the following codes" << endl;
    cout << "a - addition" << endl;
    cout << "s - subtraction" << endl;
    cout << "m - multiplication" << endl;
    cout << "d - division" << endl;
    ch =  cin.get();
    switch(ch) {
        case 'a':
             z = x+y;
             cout << " x = " << x << " y = " << y;
             cout << " x+y = " << z;
             break;
        case 'b':
             z = x-y;
             cout << " x = " << x << " y = " << y;
             cout << " x-y = " << z;
             break;
        case 'c':
             z = x*y;
             cout << " x = " << x << " y = " << y;
             cout << " x*y = " << z;
             break;
        case 'd':
             z = x/y;
             cout << " x = " << x << " y = " << y;
             cout << " x/y = " << z;
```

```
            break;
    } // end of the switch-case statement
    return 0;
} // end of main
```

Output of the above program

```
enter any two numbers
10   20
enter any one of the following codes
a - addition
s - subtraction
m - multiplication
d - division
a
x = 10 y = 20 x+y = 30
```

PROGRAM 5.9

A program to demonstrate how to use the switch–case statement for more than one case label is to be executed.

```
// example  5.9
#include <iostream>
using namespace std;
int main()
{
    int number;
    cout << "enter a number below 10" << endl;
    cin >> number;
    switch (number) {
        case 1:
        case 2:
            cout << "number = 1 or 2" << endl;
            break;
        case 3:
        case 4:
        case 5:
            cout << "number = 3 or 4 or 5" << endl;
            break;
        case 6:
        case 7:
        case 8:
        case 9:
            cout << "number = 6 or 7 or 8  or 9" << endl;
            break;
        case 0:
            cout << "number = 0" << endl;
            break;
        default:
            cout << " enter a correct number " << endl;
            break;
    } // end of the switch-case statement
    return 0;
} // end of the main program
```

Output of the above program

```
enter a number below 10
1
number = 1 or 2
```

PROGRAM 5.10

A program to find the number of vowels and consonants in a given sentence using the case statement.

```
// example 5.10
#include <iostream>
using namespace std;
int main()
{
    char name[100];
    int i,max, nvow ,ncons;
    char ch;
    cout << "enter a sentence and terminate with '@'" << endl;
    i = 0;
    while ( ( ch = cin.get()) != '@' ) {
         name[i++] = ch;
    }
    name[i++] = '\0';
    max = i-1;
    cout << "entered sentence is" << endl;
    for ( i = 0; i <= max; ++i) {
         cout << name[i];

    }
    nvow = 0;
    ncons = 0;
    i = 0;
    while ( name[i] != '\0') {
        switch (name[i]) {
             case 'a':
             case 'e':
             case 'i':
             case 'o':
             case 'u':
                  nvow = nvow+1;
                  break;
             case 'b':
             case 'c':
             case 'd':
             case 'f':
             case 'g':
             case 'h':
             case 'j':
             case 'k':
             case 'l':
             case 'm':
             case 'n':
             case 'p':
             case 'q':
             case 'r':
             case 's':
             case 't':
             case 'v':
             case 'w':
             case 'x':
             case 'y':
             case 'z':
                  ncons = ncons+1;
                  break;
         }
         i = i+1;
    } // end of while loop
    cout << "No of vowels = " << nvow << endl;
    cout << "No of consonants = " << ncons << endl;
    return 0;
}
```

Output of the above program

```
enter a sentence and terminate with '@'
this is a test
@
entered sentence is
this is a test
No of vowels = 4
No of consonants = 7

enter a sentence and terminate with '@'
aeiou
@
entered sentence is
aeiou
No of vowels = 5
No of consonants = 0
```

5.2 LOOP STATEMENTS

This section presents how to construct a loop statement in which a set of statements or a sequence of instructions are repeatedly executed until the predefined condition is met, to exit from the loop. Loop statements are also called as iterative statements. More technically, repetition is called as iteration.

The main application of using the loop statements in a program are:

- to read, write and process a large scale data easily, for example array processing
- to perform iterative computing
- business application, and so on

Now it is clear that loop statements are essential to construct systematic block styled programming. In C++, there are various ways one can implement the control structures using the different types of loop operations. Sometimes, other high level languages may not support all the features of the C++ control structures.

The loop or repetition can be classified into two types: (i) definite loop and (ii) indefinite loop. The definite loop is a loop construction in which the termination condition is met by changing the initial value within the loop or changing its boolean condition in order to meet the termination condition. An indefinite loop is a kind of loop construction that never meets its termination condition and can be terminated only by means of an external operation to the program like resetting or rebooting the computer.

The following loop structures are supported by C++:

(1) for loop
(2) while loop
(3) do-while loop

The `for` loop is a definite control structure where the number of passes through the loop is determined prior to execution. On the other hand, the control structure such as `while` and `do-while` are called as indefinite control structures, as the number of iteration or repetition of the loop body may vary depending on the value of an arbitrary boolean expression.

5.2.1 The for loop

The `for` statement is one of the most commonly used statements for constructing loop operations in C++. The for loop is a definite control structure where the number of passes through the loop is determined prior to execution. The loop variable of the for loop is automatically changed from an initial value to a final value if the incrementer or decrementer is used.

In other words, the for loop consists of three expressions. This first expression is used to initialise the index value, the second to check whether or not the loop is to be continued again and the third to change the index value for further iteration.

The general syntax of the for loop is,

```
for (control_variable = initial_value; final_value;
                              incrementer/decrementer)
    Statement
```

where the control_variable is a variable that controls the loop operation in which the C++ compiler determines how many times that loop should be repeated. The initial_value is an expression that initializes the control variable and the final_value is an expression that determines the maximum or minimum value depending on individual cases like incrementing the loop or decrementing the loop. The general syntax of the for loop and a flow chart showing the effects of a for loop are given in Fig. 5.3.

The control variable must be of ordinal type, such as integer, character or enumerated data. The initial value and the final value must be of data type compatible with control variable. The control variable of the for statement cannot be altered within the for loop. The control variable is incremented or decremented by 1. To declare a control variable of the for loop with real data type is valid in C++ but it is not mostly used.

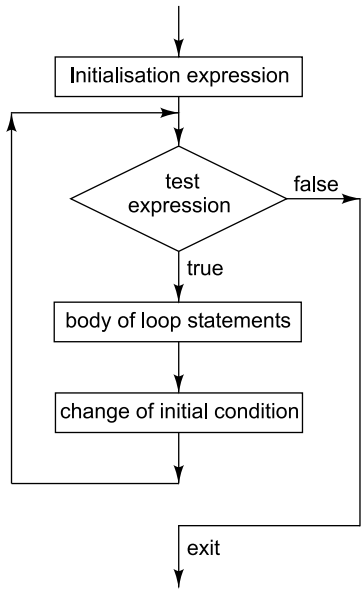


Fig. 5.3(a) *Flow Chart for the* `for` *Loop*

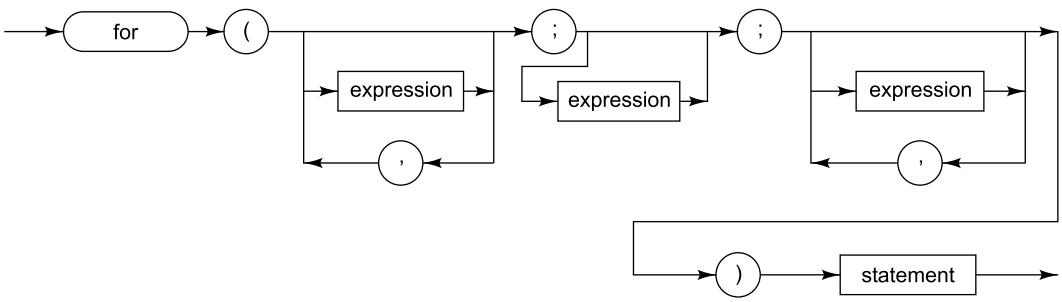


Fig. 5.3(b) *Syntax Diagram of* `for` *Statement*

C++ permits two types of for loop construction: one for incrementing from a low value to a high value and another decrementing by one in each iteration from an upper to lower value which is the final value of the loop.

(a) Incrementing Type The general syntax of the for loop for incrementing operation of a single statement is:

```
for (variable = expression_1;expression_2;expression_3)
     statement;
```

For a block of statements, the keywords begin({) and end(}) are enclosed.

```
for (variable = expression_1; expression_2; expression_3)
{
     statement_1;
     statement_2
     -------
     -------
```

```
        statement_n;
    }
```

where variable is called as the control variable of the for loop and that it can be any identifier of integer, char or enumerated data type but not real data type. Expression_1 is called as the initial value and gives the initial value of the control variable. Expression_2 is called as the final value and it gives the final value of the control variable. The expression_3 is used to change the initial value of the expression_1.

(b) Decrementing Type The general syntax of the for loop for decrementing operation of a single statement is:

```
for (variable = expression_1; expression_2; expression_3)
    statement;
```

For a block of statements, the keywords begin({) and end(}) are enclosed.

```
for (variable = expression_1; expression_2; expression_3)
{
    statement 1;
    statement 2
    -------
    -------
}
```

where variable is called as the control variable of the for loop and that it can be of any identifier of integer, char or enumerated data type. Expression_1 is called as the initial value and it gives the initial value of the control variable normally the maximum size. Expression_2 is called as the final value and it gives the final value of the control variable, that is, the minimum number. The expression_3 is used to change the initial value of the expression_1.

For example, the following C++ segment illustrates how to construct the for loop.

(1)

```
initial = 1;
final = 10;
for (counter = initial; counter <= final; ++counter)
    cout << "within the for loop ";
```

The above program segment displays the message 'within the for loop' 10 times. The initial value 1 is assigned to the loop variable counter and incremented by one. The counter value is compared with the final value that is defined in the for loop statements. Depending on the result of the comparison, the for loop determines whether, the loop is to be repeated again or not. When the content of the control variable is matched with that of the final value, the loop automatically terminates and exits out of the loop.

(2) When a for loop is used to construct a block of statements, it is essential to use the keywords begin '{' and end '}' as a delimiter. For example, the following for loop illustrates how to construct a compound or a block of statements:

```
initial = 1;
final = 10;
for (counter = initial; counter <= final; ++counter) {
    cout <<" within the for loop " << endl;
    cout <<" multiple statements of execution" << endl;
}
```

Following are some valid C++ programming segments using the for loop:

(3) To compute the sum of the first 10 natural numbers.

```
#include <iostream>
using namespace std;
int main()
{
```

```
    int i,sum = 0;
    for (i = 1; i <= 10; ++i) {
        sum = sum+i;
        cout << " i = " << i;
        cout << " sum = " << sum << endl;
    }
    return 0;
}
```

Output of the above program

i = 1 sum = 1
i = 2 sum = 3
i = 3 sum = 6
i = 4 sum = 10
i = 5 sum = 15
i = 6 sum = 21
i = 7 sum = 28
i = 8 sum = 36
i = 9 sum = 45
i = 10 sum = 55

(4) To write the 26 lower case letters of the English alphabet.

```
#include <iostream>
using namespace std;
int main()
{
    char ch;
    for (ch = 'a'; ch <= 'z'; ++ch)
        cout << ch;
    return 0;
}
```

Output of the above program

abcdefghijklmnopqrstuvwxyz

(5) To write the numbers from 10 to 1 decrementing them by 1

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int j;
    for (j = 10; j >= 1; - j)
        cout << setw(5) << j;
    return 0;
}
```

Output of the above program

```
10    9    8    7    6    5    4    3    2    1
```

PROGRAM 5.11

A program to find the sum and average of a given set of numbers using for loop.

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int i,n;
    float sum,av,x;
    cout << "How many numbers " << endl;
    cin >> n;
    sum = 0;
    for ( i = 0; i <= n-1; ++i) {
        cout << "enter a number " << endl;
        cin >> x;
        sum += x;
    }
    av = sum/x;
    cout << "Sum = " << sum << " and Average = "<< av;
    cout << endl;
    return 0;
}
```

Output of the above program

```
How many numbers
4
enter a number
10
enter a number
20
enter a number
30
enter a number
40
Sum = 100 and Average = 2.5
```

Nested for loops Nested for loop is a kind of loop construction in which a for loop is declared within another for loop. Whenever a nested for loop is declared in a program, the index variable of the for loop must be sufficiently different in order to distinguish the control variables of the for loops. The order of execution of the for loop is that the innermost for loop will be executed first.

The general syntax of the nested for loop is:

```
for (outer_control = expression_1; expression_2; expression_3) {
    for (inner_control = expression_1; expression_2; expression_3) {
        statements;
        statements;
        -------
        -------
    } // end of inner loop
}  // end of outer loop
```

PROGRAM 5.12

A program to demonstrate how a nested for loop structure can be realised in C++.

```
// nested for loop
#include <iostream>
using namespace std;
int main()
{
    int i,j,counter = 0;
    for ( i = 1; i <= 3; ++i ) {
```

```
        cout << " \n i = " << i;
        cout << " Inside j loop " << endl;
        for ( j = 1; j <= 3; ++j ) {
            cout << "  j = " << j << endl;
            counter++;
        }
    }
    cout << "counter value = " << counter;
    return 0;
}
```

Output of the above program

```
i = 1 Inside j loop
j = 1
j = 2
j = 3
i = 2 Inside j loop
j = 1
j = 2
j = 3
i = 3 Inside j loop
j = 1
j = 2
j = 3
counter value = 9
```

For example, consider the following program segment, where in the same index variable is used both in the outer and in the inner for loops.

```
#include <iostream>
using namespace std;
int main()
{
    int j;
    for ( j = 1; j <= 3; ++j) {
        for (j = 1; j <= 2; ++j)
            cout << " j = " << j << endl;
    }
    return 0;
}
```

Output of the above program

```
j = 1
j = 2
```

Note that the C++ compiler will not display any significant error message and that it cannot differentiate the control variables of the outer for loop from that of the inner for loop. In the above program, the control variable `'j'` will be executed only twice but not 6 times.

In the following program, there are two for loops which are embedded within another for loop. The inner for loops are constructed such that both the control variables are not altered by each other. In other words, the inner for loops are disjointed loops and hence, the same control variables are permitted for both loops.

PROGRAM 5.13

A program to demonstrate how to declare the same index variable when the inner loops are disjointed.

```
#include <iostream>
#include <cstdio>
```

```
using namespace std;
int main()
{
    int i,j;
    for ( i = 1; i <= 3; ++i) {
        for ( j = 1; j <= 3; ++j)
            putchar('*');
        for ( j = 1; j <= 3; ++j)
            putchar('-');
        cout << "" << endl;
    }
    return 0;
}
```

Output of the above program

```
***---
***---
***---
```

PROGRAM 5.14

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    int row,column;
    for ( row = 1; row <= 10; ++row) {
        for (column = 1; column <= row; ++column)
            putchar('+');
        for (column = 1; column <= row; ++column)
            putchar('*');
        cout << "" << endl;
    }
    return 0;
}
```

Output of the above program

```
+*
++**
+++***
++++****
+++++*****
++++++******
+++++++*******
++++++++********
+++++++++*********
++++++++++**********
```

Some unusual constructions of for loop

Case 1 Sometimes, one may write the for loop in C++ in the following way, which is even though valid, is less desirable than the conventional form.

```
#include <iostream>
using namespace std;
int main()
{
   int i;
   i = 0;
   for ( ; i < 4; )
```

```
        cout << i++ << endl;
    return 0;
}
```

Output of the above program

```
0
1
2
3
```

Case 2 Sometimes, the initial condition may not be required to be declared for some cases, like reading a string from a keyboard or a file. For that case;

```
for (; (ch = cin.get()) != '\n';)
```

PROGRAM 5.15

A program to read a line from the keyboard using for loop.

```
// reading character input
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cout <<"enter a string " << endl;
    for (; (ch = cin.get()) != '\n';)
        cout.put(ch);
    return 0;
}
```

Output of the above program

```
enter a string
this is a test

this is a test
```

The for loop is used to read a character from the keyboard as long as the test condition is being a new line or a carriage return. The for loop will be repeated until an Enter key is pressed. Suppose, the first character itself is an Enter key, then the for loop will not be repeat.

Case 3 Sometimes, one may enter the for loop as:

```
for (;;)
```

Here, the for loop is valid but it will execute the loop indefinitely because there is no condition to be checked to terminate this loop.

```
// reading character input
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cout << "enter a string " << endl;
    for (;;) {
       ch = cin.get();
       cout.put(ch);
    }
    return 0;
}
```

Output of the above program

```
enter a string
this is a test

this is a test
```

Cntrl+C is pressed to break the for loop. Here, the for loop will be repeated for ever until a user presses a termination key to stop from the program execution.

Case 4 Another type of for loop is:

```
for ( i = 0; i <= 9; i++);
```

This loop will be repeated till the i value becomes 9. There is no statement used in this loop and the semicolon is used for terminating the loop. It will just repeat the loop operation as long as the given condition becomes true. One may require this type of operation for introducing time delay in a program.

Case 5 The comma operator can be used in conjunction with the for statement which permits two different expressions to appear in a situation where only one expression would ordinarily be used. For example,

```
for ( expression 1a, expression 1b; expression 2 ; expression 3){
    statements
    statements
}
```

where expression 1a and expression 1b are the two expressions separated by the comma operator and normally only expression 1 would appear.

The two expressions would typically initialize two separate indices that would be used simultaneously within the for loop. Another way of using the comma operator in a for statement is:

```
for (expression1; expression2 ; expression 3a,expression 3b) {
    statement
    statement
    -------
    -------
}
```

Here, expression 3a and expression 3b, separated by the comma operator, appear in place of a single expression, expression 3. In this application two separate expressions would typically be used to alter (e.g. incrementer or decrementer) the two different indices used simultaneously within the loop. For example, one index may count forward while the other backward.

PROGRAM 5.16

A program to display the number between 0 to 9 as well as 9 to 0 simultaneously one by one using for statement.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int i,j;
    for ( i = 0,j = 9; i <= 9; i++,j--)
        cout << "i = " << i << setw(8) << "j = " << j <<'\n';
    return 0;
}
```

Output of the above program

```
i = 0     j = 9
i = 1     j = 8
i = 2     j = 7
i = 3     j = 6
i = 4     j = 5
i = 5     j = 4
i = 6     j = 3
i = 7     j = 2
i = 8     j = 1
i = 9     j = 0
```

5.2.2 The while loop

The second type of loop, the while loop, is used when we are not certain about the execution of the loop. After checking whether the initial condition is true or false and finding it to be true only, then the while loop will enter into the loop operations. The general form of the while loop is:

For a single statement,

```
while (boolean condition)
       statement;
```

For a block or compound statements the braces { and } must be included as a delimiter of the loop.

```
while (boolean condition) {
       statement 1;
       statement 2;
       -------
       -------
}
```

The boolean condition can be any valid C++ expression and the boolean expression includes the value of a variable. The boolean condition can be a singular a compound condition and the condition is actually the test condition. In the for loop, increment or decrement is done automatically, whereas, the while loop does not explicitly contain initialisation or incrementation parts of the loop, which are normally provided by the programmers.

The general form of the while syntax diagram and a flow chart showing the effects of the while loop are given in Fig. 5.4. Prior to each cycle of the loop, the boolean expression that appears after the keyword while is evaluated and tested. A new cycle is started, if and only if, the value of the boolean expression is true. Otherwise, the next statement after the While statement is executed. The statement following the keyword while represents the body of the loop. It is imperative that this statement eventually causes the value of the boolean expression to be false, when it is evaluated prior to beginning of a new cycle, so that the loop will be terminated after a finite number of cycles.

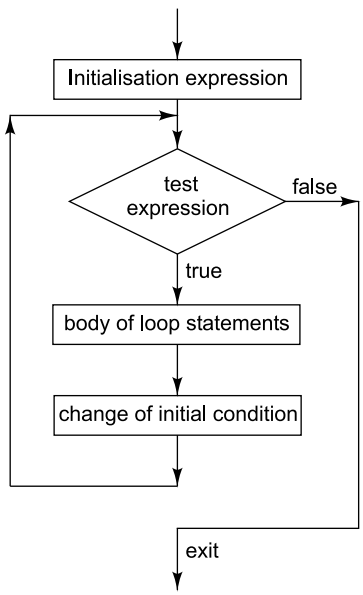


Fig. 5.4(a) *Flow Chart for While Loop*

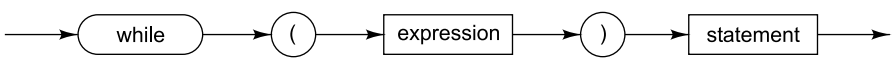


Fig. 5.4(b) *Syntax Diagram of while Statement*

The following program segment shows the use of the initial and the test conditions to be used in a program.

```
initial_condition
while (test_condition) {
    statement 1;
    statement 2;
    change of the initial_condition;
}
```

Comments on the above while loop

(1) The loop may never be executed as test is at the start of the loop as in a for statement.
(2) The begin '{' and end '}' braces are used when the number of statement in the loop is more than one.
(3) Part of the loop will alter some aspect of the Boolean expression, otherwise the loop executes forever or not at all.

PROGRAM 5.17

A program to demonstrate how to construct a while loop in C++.

```
#include <iostream>
using namespace std;
int main()
{
    bool flag = true;   //initial condition
    while (flag) {      // test condition
       cout << "inside the while loop \n";
       flag = false;  // change of initial condition
    }
    return 0;
}
```

Output of the above program

```
inside the while loop
```

A following program explains a while loop where the change of initial condition is skipped inside the while loop. Hence the following program runs for ever until a user interrupts through external commands like, pressing Cntrl + C for termination from the program execution.

```
#include <iostream>
using namespace std;
int main()
{
    bool flag = true;
    while (true) {
       cout << "inside the while loop \n";
    }
    return 0;
}
```

PROGRAM 5.18

A program to find the sum of the first n natural numbers using while loop.

```
sum = 1+2+3 + ..... n
// 1+2+3+...+n
#include <iostream>
using namespace std;
int main()
{
    int digit,n,sum;
    cout << "How many numbers " << endl;
    cin >> n;
    digit = 1;
    sum = 0;
    while (digit <= n) {
       sum += digit;
       digit++;
    }
    cout << " 1+2+3 + ... " << n << " = ";
    cout << sum << endl;
    return 0;
}
```

Output of the above program

```
How many numbers
10
1+2+3 + ... 10 = 55
```

In the above program, the initial condition is assigned to 0 and the test condition is used to check whether the digit value has become the value of n. If not, it will repeat the loop, the final statement of the program digit++ will be incremented by one. During each execution of the while loop, the value of the initial index is changed.

PROGRAM 5.19

A program to display the first n odd numbers using while loop.

```
// to display first n odd numbers
#include <iostream>
using namespace std;
int main()
{
    int n,counter;
    cout << " maximum number ? " << endl;
    cin >> n;
    counter = 1;
    while (counter <= n) {
       cout << counter << '\t';
       counter = counter+2;
    }
    return 0;
}
```

Output of the above program

```
maximum number?
11
1  3  5  7  9  11
```

In each iteration, the counter value is incremented by 2 and the final condition is checked and when it is met, the loop control automatically exits from the loop.

PROGRAM 5.20

A program to find the sum and average of the given numbers using the while loop.

```
// to find sum and average of a given numbers
#include <iostream>
using namespace std;
int main()
{
    int i,n;
    float sum = 0,av,x;
    cout << "How many numbers " << endl;
    cin >> n;
    i = 1;
    while (i <= n) {
       cout << "enter a number " << endl;
       cin >> x;
       sum += x;
       i++;
    }
    av = sum/n;
    cout << "Sum = " << sum << " and Average = " << av;
    return 0;
}
```

Output of the above program

```
How many numbers
4
enter a number
1
enter a number
2
enter a number
3
enter a number
4
Sum = 10 and Average = 2.5
```

Nested while loop Nested while loop is a kind of loop construction such that a while loop is embedded within another while loop. C++ permits to realise any level of loops to be embedded with in any other control blocks. In other words, one can construct absolutely any level of loops within any other loops provided each innermost loop must terminate with boolean conditions. Otherwise, the inner loop that is not defined with proper termination condition, may execute indefinitely.

The general syntax of the nested while loop is:

```
while (outer_loop_condition) {
   while (inner_loop) {
       while (innermost_loop) {
           -------
           -------
       } // end of innermost loop
       -------
       -------
    } // end of inner loop
```

```
        -------
        -------
    } // end of outer loop
```

For example, the following program segment illustrates how to declare a nested while loop:

PROGRAM 5.21

```
// nested while loop
#include <iostream>
using namespace std;
int main()
{
    int i,j;
    i = 1;
    while ( i <= 3) {
        cout << " \n i = " << i << endl;
        j = 1;
        while ( j <= 3) {
            cout << " j = " << j;
            j++;
        }
        i++;
        cout << "" << endl;
    }
    return 0;
}
```

Output of the above program

```
i = 1
j = 1 j = 2 j = 3

i = 2
j = 1 j = 2 j = 3

i = 3
j = 1 j = 2 j = 3
```

The value of j of the inner while loop will be executed 3 times for each iteration of the outer while loop. The value of i will be displayed 3 times. Totally, the above nested structure repeats its control operation 9 times.

Role of incrementer and decrementer in C++ One must be always cautious while using the incrementer or decrementer as i++ and ++i. The following program segment to find the sum of the first 10 numbers shows the distinct use of the two types of incrementers.

case (a)

```
sum = 0;
i = 1;
while (i <= 10)
   sum += i++;
```

case (b)

```
sum = 0;
i = 1;
while ( i <= 10 )
   sum += ++i;
```

In both cases, the value of i will be the same for each iteration but the value of sum is different due to the post-increment and pre-increment operators. In the first case, the value of `i` will be assigned to the sum before incrementation due to the post-increment operator `(i++)`.

```
sum += i++;
```

Hence, the value of sum and i for the first case, are as sum = 55, i = 11

In the second case, the value of i will be incremented and then assigned to the sum due to the pre-increment operator (++i).

```
sum += ++i;
```

Hence, the value of sum and i for the second case, are as sum = 65, i = 11

Consider another program segment to compute the sum of the first 3 numbers, using post-incrementor and pre-incrementor as given below:

```
#include <iostream>
using namespace std;
int main()
{
    int i,j,sum1,sum2;
    sum1 = sum2 = 0;
    i = j = 0 ;
    while ( i <= 2) {
       sum1 += i++;
       cout << " i++ = " << i << endl;
       sum2 += ++j;
       cout << " ++j = " << j << endl;
       cout << " sum1 = " << sum1 << endl;
       cout << " sum2 = " << sum2 << endl;
    }
}

sum2 = 14
```

At the end, sum1 will be calculated for 4, and the sum2 will be 6.

Use of while loop to input data Normally, the while loop is used for reading data and processing it until the end of data is found. The above use is illustrated through a program segment which reads a set of characters from the keyboard until it finds a carriage return key or a new line and displays it on the video screen.

```
//  reading a line of characters using while loop
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cout << " enter a line of text" << endl;
    ch = cin.get();
    while ( ch != '\n') {
       cout.put(ch);
       ch = cin.get();
    }
    return 0;
}
```

Output of the above program

```
enter a line of text
this is a test

this is a test
```

Another compact C++ code is,

```
// reading a line of characters using while loop
// version 2.cpp
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cout << " enter a line of text " << endl;
    while ((ch = cin.get()) != '\n') {
       cout.put(ch);
    }
    return 0;
}
```

Sometimes, one may use more than one test condition using different logical expressions as a single statement. For example, one expression can be used to check both the new line and the tab character.

```
ch = cin.get();
while ( ch != '\n' && ch != '\t') {
    cout.put(ch);
    ch = cin.get();
}
```

5.2.3 The do–while loop

The do-while loop is another repetitive loop used in C++ program. Whenever one is sure of the test condition, then the do-while loop can be used as it enters into the loop at least once and then checks whether the given condition is true or false. As long as the test condition is true, the loop operations or statements will be repeated again and again. As in the case of a while loop, here also three expressions are used to construct this loop. The expression_1 is used to initialise the index value which normally appears out of the loop. Expression_2 is used to change the index value and the expression_3 is used to check whether or not the loop is to be repeated again.

The general syntax of the do-while loop is:

```
do {
    statement_1
    statement_2
    -------
    -------
}
while (boolean condition);
```

The boolean condition in the above type of loop is always written as the last statement. That is why this loop is executed at least once. The general form of the do-while syntax diagram and a flow chart showing the effects of the do-while loop are given in Fig. 5.5.

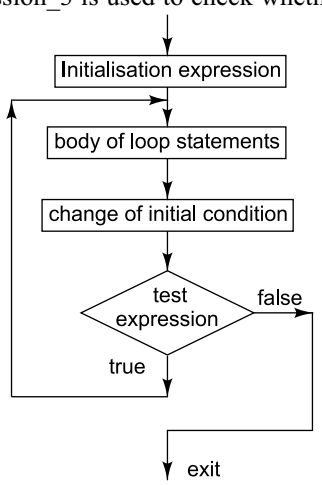


Fig. 5.5(a) *Flow Chart for* `do-while` *Loop*

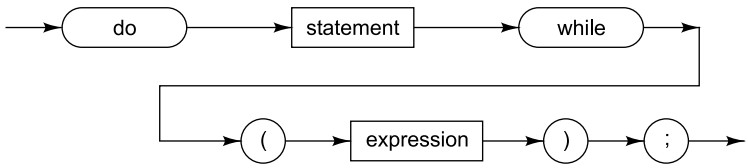


Fig. 5.5(b) *Syntax Diagram of* `do-while` *Statement*

Note that the do-while control structure does not need a begin '{' and end '}' braces to bracket more than one statement, as the do–while structure itself acts as a compound statement. The above syntax structure can be modified as:

```
expression_1
do {
   statement 1
   statement 2
   -------
   -------
   expression_2
}
while (expression_3);
```

where expression_1 is an initialisation statement of the index value and the expression_2 is a statement in which the index value changes and the expression_3 is a boolean statement which is used for the test condition of the loop i.e., for deciding whether the loop should be repeated or not and based on the test condition only, the loop enters or exits. The test condition may be of a single boolean condition or a compound condition.

Comments on the do–while loop

(1) Note that the body of the do-while loop can consist of more than one statement and hence, it may not be required to include begin '{' and end '}' statements as delimiters.
(2) The body of the loop will be executed at least once as the test for termination of the loop appears at the end of the loop. The for loop and the while loop, however, may never be executed as the test condition is at the beginning.
(3) There will be at least one statement in the body of the loop that alters some part of the boolean expression. If this is not present, then the loop will never terminate.

PROGRAM 5.22

A program to find the sum and average of the given numbers using a `do-while` *loop.*

```
// to display sum and average using do-while
#include <iostream>
using namespace std;
int main()
{
    int i = 1,n;
    float sum = 0,av,x;
    cout << "How many numbers \n";
    cin >> n;
    do {
       cout << "enter a number \n";
       cin >> x;
       sum += x;
```

```
        i++;
    }
    while (i <= n);
    av = sum/n;
    cout << "Sum = " << sum << " and Average = " << av;
    return 0;
}
```

Output of the above program

```
How many numbers
4
enter a number
10
enter a number
20
enter a number
30
enter a number
40
Sum = 100 and Average = 25
```

PROGRAM 5.23

A program to generate a fibonacci series of `'n'` *numbers, where* `n` *is defined by the programmer. (series should be: 0 1 1 2 3 5 8 13 21 34 and so on.)*

```
// to generate a fibonacci series of 'n' numbers
// using do-while statement
#include <iostream>
using namespace std;
int main()
{
    int i,n;
    int fib0,fib1,fib;
    cout << "How many numbers ? \n";
    cin >> n;
    fib0 = 0;
    fib1 = 1;
    cout << fib0 << '\t' << fib1 << '\t';
    i = 3;
    do {
       fib = fib0+fib1;
       cout << fib << '\t';
       fib0 = fib1;
       fib1 = fib;
       i++;
    }
    while ( i <= n);
    return 0;
}
```

Output of the above program

```
How many numbers ?
6
0  1  1  2  3  5
```

PROGRAM 5.24

A program to find the factorial of a given number using `do-while` *loop.*

```
// factorial of a given number
#include <iostream>
using namespace std;
int main()
{
    int i,n;
    long int fact;
    cout << "enter a number \n";
    cin >> n;
    fact = 1;
    if ( n == 0)
        cout << "factorial = " << fact;
    else {
        i = 1;
        do {
           fact = fact*i;
           i++;
        }
        while ( i <= n);
        cout << "factorial = " << fact;
    }
    return 0;
}
```

Output of the above program

```
enter a number
5
factorial = 120
```

Nested do–while loop Nested `do-while` loop is a kind of loop construction wherein a `do-while` loop is embedded within another `do-while` loop. C++ permits to realise any level of loops to be embedded within any other control blocks. In other words, one can construct absolutely any level of loops within any other loops, provided, each innermost loop must terminate with boolean conditions, otherwise, the inner loop that is not defined with proper termination condition, may execute indefinitely.

PROGRAM 5.25

A program to demonstrate the construction of nested do–while structure.

```
// nested do-while loop
#include <iostream>
using namespace std;
int main()
{
    int i,j;
    cout << "demonstration of nested do-while loop " << endl;
    i = 1;
    do {
       cout << "i = " << i << "\n";
       j = 1;
       do {
          cout << "j = " << j << '\t';
          j++;
       }
```

```
        while ( j <= 5);
        cout << "" << endl;
        i++;
    }
    while ( i <= 3);
    return 0;
}
```

Output of the above program

```
demonstration of nested do-while loop
i = 1
j = 1  j = 2 j = 3  j = 4  j = 5

i = 2
j = 1  j = 2 j = 3  j = 4  j = 5

i = 3
j = 1  j = 2 j = 3  j = 4  j = 5
```

The 'value of j' of the inner do-while loop will be executed 5 times for each iteration of the outer do-while loop. The 'value of i' will be displayed 3 times. Totally, the above nested structure repeats its control operation 15 times.

5.3 NESTED CONTROL STRUCTURES

So far, the declaration and implementation of nested for loop, nested while loop, and nested `do-while` loop have been discussed. In practical situations, one control statement will be embedded within another control block. This section presents few examples for constructing a nested control strcuture in C++. The termination condition for the inner and outer loops must be distinct and defined properly, otherwise, the loop will be executed indefinitely. The outer loops should not be overlapped with the inner loops. When loops are nested, the innermost loop, is executed first.

For example, the following program segment illustrates how to define a nested control structure:

Case 1

```
for ( expression ) {
    while (condition) {
       for (expression) {
            -------
            -------
       }  // end of for loop
    } // end of while loop
} //end of i loop
```

Case 2 The following program segment contains both a do–while and while block within a for statement:

```
for (expression) {
   do {
      -------
      -------
   }
   while (condition);

   while ( condition) {
      -------
      -------
   }
} // end of for loop
```

PROGRAM 5.26

A program to generate the following series of numbers using nested loop structure:

```
1
1 2
1 2 3
.....
......
1 2 3 4 5 6 7 8 9

// Generation of series of numbers
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int i,j;
    cout << "Generation of series of numbers \n ";
    i = 1;
    do {
       for ( j = 1; j <= i; ++j)
            cout << setw(3) << j;
       cout << endl;
       i++;
    }
    while ( i <= 9);
    return 0;
}
```

Output of the above program

```
Generation of series of numbers
  1
  1  2
  1  2  3
  1  2  3  4
  1  2  3  4  5
  1  2  3  4  5  6
  1  2  3  4  5  6  7
  1  2  3  4  5  6  7  8
  1  2  3  4  5  6  7  8  9
```

PROGRAM 5.27

A program to generate the following series of numbers using nested loop structure:

```
9 8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
......
.....
1

// Generation of series of numbers
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int i,j;
```

```
    cout << "Generation of series of numbers " << endl;
    i = 9;
    do {
       for ( j = i; j >= 1; --j)
            cout << setw(3) << j;
       cout << endl;
       i--;
    }
    while ( i >= 1);
    return 0;
}
```

Output of the above program

```
Generation of series of numbers
  9  8  7  6  5  4  3  2  1
  8  7  6  5  4  3  2  1
  7  6  5  4  3  2  1
  6  5  4  3  2  1
  5  4  3  2  1
  4  3  2  1
  3  2  1
  2  1
  1
```

5.4 BREAKING CONTROL STATEMENTS

Sometimes, it may be required to break the control within a control block. In such situations C++ permits the use of a special technique in which control is transferred from one part of a program to another part. Following statements are used for breaking the control in a C++ program.

(1) break statement
(2) continue statement
(3) goto statement

5.4.1 Label Declaration

Any statement may be labelled. The label must be any valid identifier of C++. The numerals are not permitted to use as the label statement in the C++ language. Label should not be defined in the declaration part of a program. The label statement defined in any part of a program is different from the case label, if it is defined within a program structure.

For example, consider the following program segment which illustrates how a label is defined and used in a segment of a C++ program.

```
#include <iostream>
int main()
{
    -------
    -------
    error:
         -------
         -------
         goto error;
    again:
         -------
         -------
         goto again;
}
```

5.4.2 The goto Statement

The goto statement is used to alter the program execution sequence by transferring the control to some other part of the program. The syntax diagram of the goto statement is given in Fig. 5.6.

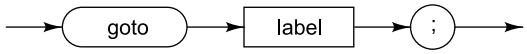


Fig. 5.6 *Syntax Diagram of* `goto` *Statement*

The general syntax of the goto statement is:

```
goto label;
```

where label is any valid C++ identifier such that control could be transferred to the destination.

There are two ways of using goto statements in a program, namely, as a conditional goto and as an unconditional goto.

(a) Unconditional goto The unconditional goto statement is used just to transfer the control from one part of the program to the other part, without checking any condition. Normally, a good programmer will not prefer to use the unconditional goto statement in his program as it may lead to a very complicated problem like a never ending process. The use of unconditional goto statement is illustrated in the following segment:

```
#include <iostream>
using namespace std;
int main()
{
    again:
        cout << "Welcome to the computer world \n";
    goto again;
        cout << "output data \n";
    return 0;
}
```

In the above program, cout <<"output data \n", will not be executed for ever, as the control always gets transferred from goto label, to the first part of the executable statement. Hence, the statements that are defined after the goto statement will not be executed at all. The 'cout <<"output data \n";' is known as the unreachable code. This type of goto statement usage is called as unconditional goto. In such cases, control cannot be terminated until the computer is reset through an external medium.

(b) Conditional goto Conditional goto is used to transfer the control of the execution from one part of the program to other part under certain situations.

Comments on goto statements

(1) Any statement can be labelled.
(2) It is forbidden to jump into a block created by a repetitive statement or a conditional statement, i.e. a do-while loop, for loop, or while loop, switch-case and if blocks.
(3) If a label is declared, it must prefix a statement.

The goto statement label is a label that appears in both the execution and in the declaration sections. The following sample program explains the usage of conditional goto statements.

PROGRAM 5.28

A program to find the sum of only positive integers using the goto statement.

```
// using goto statement
#include <iostream>
```

```
using namespace std;
int main()
{
    int x,i,n,sum;
    cout << "How many numbers ?\n";
    cin >> n;
    i  = 1;
    sum  = 0;
    while (i <= n) {
       cout << "enter a number\n";
       cin >> x;
       if (x < 0) goto error;
       sum  += x;
       ++i;
    }
    error:
    cout <<"sum of values (only positive) = " << sum;
    return 0;
}
```

Output of the above program

```
How many numbers?
4
enter a number
1
enter a number
2
enter a number
-4
sum of values (only positive) = 3
```

The above program calculates the sum of only the positive integers. If a negative number is entered, control will break from the while loop and displays the sum of the positive numbers that have already been entered. In case, the first number itself is a negative number, the computer will display sum as 0, because there is no provision in the above program to iterate for finding the sum of positive numbers.

Note that there is similarity between case labels and goto labels. However, they are independent and distinct. For example, consider the following program segment which shows how a case label is different from the goto label, even though, both labels contain the same form:

PROGRAM 5.29

```
#include <iostream>
using namespace std;
int main()
{
    char a,b;
    b = 'a';
    a:
    switch (b) {
        case 'a':
              cout << "hello \n";
              break;
        case 'b':
              cout << "two \n";
              goto a;
    }
    return 0;
}
```

Output of the above program

```
hello
```

Note that the `case` label and the `goto` label (equal to 'a' in the above example) are two different entities. Though the same form is used for `case` and `goto` labels, it does not mean that both are same. The compiler will generate separate address for `case` and `goto` labels individually and hence, no error message will be displayed.

Invalid construction of the goto statements Following are some program structures that are wrongly constructed:

Case 1 Note that a transfer of control using `goto` statments to the middle of a loop statement is illegal.

For example,

```
//  invalid
while (condition) {
    -------
    -------
    error:
    cout <<"a message \n";
}
goto error;
```

Case 2 It is invalid usage of the `goto` statement when control is transferred from outside to inside the control block such as `for` loop, `do-while` and `while-do`.

```
// illegal jump of goto statement within the control block
   goto again;
   for ( i = 1;  i <= n; ++i) {
      -------
      -------
      again:
      cout <<"error \n";
   }
```

Case 3 Note that it is an illegal if `goto` statements jump inside the `if-else` statements.

```
-------
-------
goto error;
if (condition ) {
    -------
    -------
}
else if (condition)
{
    -------
    -------
    error:        // illegal
}
else
{
    -------
    -------
  repeat:       // illegal
    -------
    -------
}
goto repeat;
```

5.4.3 The break statement

The `break` statement is used to terminate the control from the loop statements of the `switch-case` structure. The break statement is normally used in the switch-case loop and in each case condition, the break statement must be used. If not, the control will be transferred to the subsequent case condition also. The syntax diagram of the break statement is given in Fig. 5.7.

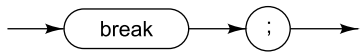


Fig. 5.7 *Syntax Diagram of* `break` *Statement*

The general format of the break statement is:

```
break;
```

where the `break` is a keyword in the C++ program and the semicolon must be inserted after the break statement. Two uses of the break statements are illustrated below:

(1) Break statement used with switch-case structure

```
switch (day)  {
   case 1:
       cout <<"Monday \n";
       break;
   case 2:
       cout <<"Tuesday \n";
       break;
   default:
       cout <<"All days \n";
} //  end of switch-case structure
```

If we have written the switch-case structure like this,

```
switch (day)  {
   case 1:
       cout <<"Monday \n";
   case 2:
       cout <<"Tuesday \n";
       break;
   default :
       cout <<"All days \n";
} //  end of switch-case structure
```

then the computer will print the message like this when the value of the `day = 1`

```
Monday
Tuesday
All days
```

Since there is no break statement in the `case 1`, the computer will transfer the control to other cases also. To avoid this sort of undesired results, normally the break statement will be used in the each case section.

(2) Break statement used in a `while` loop A break statement is used in other loops also and it is explained with the following illustration:

PROGRAM 5.30

```
// using break statement inside the while loop
#include <iostream>
using namespace std;
int main()
{
    int value,i;
    i = 0;
```

```
    while ( i <= 10) {
       cout <<" enter a number " << endl;
       cin >> value;
       if ( value <= 0) {
          cout << "zero or negative value found \n";
          break;
       }
       i++;
    }
    return 0;
}
```

Output of the above program

```
enter a number
1
enter a number
0
zero or negative value found
```

The above program segment will process only the positive integers. Whenever a zero or negative value is encountered, the computer will display the message "Zero or negative value found" as an error and exit from the while loop.

5.4.4 The Continue Statement

The continue statement is used to repeat the same operations once again even if it checks the error. The syntax diagram of the continue statement is given in Fig. 5.8.

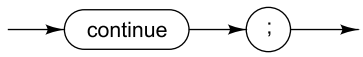


Fig. 5.8 *Syntax Diagram of* `continue` *Statement*

The general syntax of the continue statement is,

```
continue;
```

where `continue` is a keyword followed by the semicolon. The `continue` statement is used for the inverse operation of the `break` statement.

The following program segment illustrates the use of the `continue` statement operation.

PROGRAM 5.31

```
// using continue statement
#include <iostream>
using namespace std;
int main()
{
    int x,i,n,sum;
    cout << "How many numbers ? " << endl;
    cin >> n;
    i = 1;
    sum = 0;
    while (i <= n) {
       cout << "enter a number " << endl;
       cin >> x;
       if (x < 0) continue;
       sum  += x;
       ++i;
    }
    cout <<"sum of values (only positive) = " << sum << endl;
    return 0;
}
```

Output of the above program

```
How many numbers?
3
enter a number
1
enter a number
1
enter a number
-10
enter a number
-20
enter a number
-1
enter a number
10
sum of values (only positive) = 12
```

This program segment will process only the positive integers. Whenever a zero or negative value is encountered, the computer will display the message "Zero or negative value found" as an error and it continues the same loop as long as the given condition is satisfied.

REVIEW QUESTIONS

1. What is a conditional expression? List a few applications of using a conditional expression in real life problems.
2. What is a nested `if` statement?
3. When two `if` statements are nested, what is the rule that determines which `else` clause matches which `if`?
4. What is the difference between these two operators = and ==?
5. What are the different ways of constructing a multiway `if-else` structure?
6. Elucidate few examples of the nested `if-else` structure with suitable examples.
7. What is a case statement and how is it different from the multiway `if-else` structure?
8. What is a boolean expression and how it can be used in a program for decision making?
9. Draw a syntax diagram of the `switch-case` structure that is used in C++.
10. What is looping in C++? What are the advantages of using loops in C++?
11. What are the different types of loop statements that are used in C++?
12. What is `for` loop? Under what circumstances the `for` loop is used to construct a looping in C++?
13. What are nested `for` loops?
14. What is a `while` loop and how does it differ from the `for` loop?
15. What is the appropriate place or circumstance under which the `while` loop is better than the `for` loop?
16. Draw a flow chart and the syntax diagram of the `while-do` loop.
17. What is `do-while` loop?
18. How the `do-while` loop varies from the `while-do` loop? Explain.
19. Draw a flow chart and the syntax diagram of the `do-while` loop.
20. Explain the following loop control statements as used in C++.
    (a) `label` statement (b) `goto` statement
    (c) `break` statement (d) `continue` statement

21. What is the use of a label statement in C++?
22. List the merits and demerits of `while-do` loop and `do-while` loop
23. Why is `goto` not necessary for a structured programming language like C++?
24. Summarise the syntactic rules of the following loop statements:
(a) `for` loop
(b) `while-do` loop
(c) `do-while` loop
25. Explain the following with suitable examples:
(a) conditional expression (b) arithmetic expression
(c) logical expression (d) boolean expression

CONCEPT REVIEW PROBLEMS

1. Determine the output of each of the following programs when it is executed.
(a)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b = 20, c = 30;
    if (( a > b) && ( ++b < ++c))
        cout <<" a = " << a <<" b = " << b <<"\n";
    else
        cout <<" b = " << b <<" c = " << c <<"\n";
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b = 20, c = 30;
    if (( a < b) && ( b++ < c++))
        cout <<" a = " << a <<" b = " << b <<"\n";
    else
        cout <<" b = " << b <<" c = " << c <<"\n";
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b = 20, c = 30;
    if (( a < b) || ( ++b < c++))
        cout <<" b = " << b <<" c = " << c <<"\n";
    else
        cout <<" a = " << a << "b = " << b <<"\n";
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b = 20, c = 30;
    if (( a > b) || ( b++ < c++))
        cout <<" b = " << b <<" c = " << c <<"\n";
    else
        cout <<" a = " << a <<" b = " << b <<"\n";
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b = 20, c = 30;
    if (( a > b) && ( b++ < c++))
        cout <<" value of a = " << a <<"\n";
    else
    {
       if (( a < b) || ( a++ < c++ ))
           cout <<"a+b+c = " << a+b+c <<"\n";
       else
           cout <<"a*b*c = " << a*b*c <<"\n";
    }
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b = 20, c = 30;
    if (( a > b) && ( b++ < c++))
        cout <<" value of a = " << a <<"\n";
    else
    {
       if (( a < b) && ( a++ < c++ ))
           cout <<"a+b+c = " << a+b+c <<"\n";
       else
           cout <<"a*b*c = " << a*b*c <<"\n";
    }
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b = 20, c = 30;
    if (( a < b) || ( b++ > c++ ))
    {
```

```
            if (( a < c) && ( b++ > c++))
                cout <<" value of a+b = " << a+b <<"\n";
            else
                cout <<" value of a*b = " << a*b <<"\n";
        }
        else
        {
            if (( a < b) && (a++ < c++ ))
                cout <<"a+b+c = " << a+b+c <<"\n";
            else
                cout <<"a*b*c = " << a*b*c <<"\n";
        }
        return 0;
    }
```

(h)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b = 20, c = 30,d = 40;
    if (( a++ < d) || ( b++ < c++ ))
    {
       if (( a++ < c) || ( b++ < d++))
       {
          if (( a++ <= d) && ( b++ <= c++))
             cout <<" value of a+b = " << a+b <<"\n";
          else
             cout <<" value of a*b = " << a*b <<"\n";
       }
    }
    else
    {
       if (( a < b) && (a++ < c++ ))
             cout <<"a+b+c = " << a+b+c <<"\n";
       else
             cout <<"a*b*c = " << a*b*c <<"\n";
    }
    return 0;
}
```

2. Determine the output of each of the following programs when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    int x,i = 2;
    x = 75;
    switch ( x % i) {
        case 1:
             cout <<"case 1 \n";
             break;
        case 2:
             cout <<"case 2 \n";
             break;
```

```
            default:
                cout <<"default case \n";
                break;
        }
        return 0;
    }
```

(b)

```
    #include <iostream>
    using namespace std;
    int main()
    {
        int x,i = 2;
        x = 75;
        switch ( x % i) {
            case 1:
                cout <<"case 1 \n";
                break;
            case 2:
                cout <<"case 2 \n";
                break;
            default:
                switch (i) {
                    case 1:
                        cout <<"inner case 1 \n";
                        break;
                    case 2:
                        cout <<"inner case 2 \n";
                        break;
                    default:
                        cout <<"default inner \n";
                        break;
                }
            break;
        }
        return 0;
    }
```

(c)

```
    #include <iostream>
    using namespace std;
    int main()
    {
        int x,i = 3;
        x = 75;
        switch ( x % i) {
            case 1:
                cout <<"case 1 \n";
                break;
            case 2:
                cout <<"case 2 \n";
                break;
            default:
                switch (i % 4) {
                    case 1:
                        cout <<"inner case 1 \n";
                        break;
```

```
                    case 2:
                         cout <<"inner case 2 \n";
                         break;
                    case 3:
                         cout <<"inner case 3 \n";
                         break;
                    default:
                         cout <<"default inner \n";
                         break;
               }
               break;
          }
          return 0;
     }
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
     int x = 0,i = 4;
       switch (i) {
          case 1:
               x += 1;
          case 2:
               x += 2;
          case 3:
               x += 3;
          case 4:
               x += 4;
               cout <<" x = " << x << "\n";
          default:
               x += 5;
               cout <<" x = " << x << "\n";
               break;
     }
     return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
     int x = 0,i = 4;
     switch (i) {
          default:
               x += 5;
               cout <<" x = " << x << "\n";
          case 4:
               x += 4;
               cout <<" x = " << x << "\n";
          case 1:
               x += 1;
          case 2:
               x += 2;
```

```
            case 3:
                 x += 3;
                cout <<" x = " << x << "\n";
         }
         return 0;
    }
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    int i,x = 1,y = 4;
    y = x % y;
    cout <<" y = " << y << "\n";
    switch (y) {
        case 1:
             for ( i = 1; i <= 5; ++i)
                 x += x;
             break;
        case 2:
             for ( i = 5; i <= 0; --i)
                 x += i;
             break;
        case 3:
             i = 1;
             while ( i <= 5) {
                 x += 2;
                 i++;
             }
             break;
        case 4:
             i = 1;
             do {
                x += 4;
                i++;
             }
             while ( i <= 5);
             break;
        default:
             x += 5;
    }
    cout <<" x = " << x << "\n";
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 75;
    a = a / 10;
    cout <<"a = " << a << "\n";
    switch (a) {
```

```
            case 9:
                 cout <<"grade - A \n";
                 break;
            case 8:
                 cout <<"grade - B \n";
                 break;
            case 7:
                 cout <<"grade - C \n";
                 break;
            case 6:
                 cout <<"grade - D \n";
                 break;
            default:
                 cout <<" grade - fail \n";
                 break;
        }
        return 0;
    }
```

3. Convert the following `switch-case` structure into an equivalent `if-else` structure.

```
#include <iostream>
using namespace std;
int main()
{
    int a = 75;
    a = a % 10;
    cout <<"a = " << a;
    switch (a) {
        case 9:
             cout <<"grade - A \n";
             break;
        case 8:
             cout <<"grade - B \n";
             break;
        case 7:
             cout <<"grade - C \n";
             break;
        case 6:
             cout <<"grade - D \n";
             break;
        default:
             cout <<" grade - fail \n";
             break;
    }
    return 0;
}
```

4. Convert the following `if-else` structure into an equivalent `switch-case` structure.

```
#include <iostream>
using namespace std;
int main()
{
    char grade;
    grade = 'E';
    if (grade == 'A')
       cout <<" Excellent \n";
    else if (( grade == 'B') || (grade == 'C'))
```

```
        cout <<" Good \n";
    else if ((grade == 'D') || (grade == 'E'))
        cout <<" Poor \n";
    else
        cout <<" Grade - Fail \n";
    return 0;
}
```

5. Determine the output of each of the following programs when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    int p,r = 1,s = 2, t = 3;
    p = (10 % 2) / 3;
    switch (p) {
        case 0:
            r++;
            cout << " r = " << r <<"\n";
            break;
        case 1:
            s++;
            cout << " s = " << s <<"\n";
            break;
        case 2:
        case 3:
        case 4:
            t++;
            cout << " t= " << t <<"\n";
            break;
    }
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    int p,r = 1,s = 2, t = 3;
    p = (10 % 2) / 3;
    if ( p++ == 0)
    {
        r++;
        cout << " r = " << r <<"\n";
    }
    else if ( p++ == 1)
    {
        s++;
        cout << " s = " << s <<"\n";
    }
    else if ( ( p++ == 2) || ( p++ == 3) || ( p++ == 4))
    {
        t++;
        cout << " t= " << t <<"\n";
    }
```

```
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    int p,r = 1,s = 2, t = 3;
    p = (10 % 2) / 3;
    if ( ++p == 0)
    {
        r++;
        cout << " r = " << r <<"\n";
    }
    else if ( p++ == 1)
    {
        s++;
        cout << " s = " << s <<"\n";
    }
    else if ( ( p++ == 2) || ( p++ == 3) || ( p++ == 4))
    {
        t++;
        cout << " t= " << t <<"\n";
    }
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    int p,r = 1,s = 2, t = 3;
    p = (10 % 2) / 3;
    if ( (++p)++ == 0)
    {
        r++;
        cout << " r = " << r <<"\n";
    }
    else if ( p++ == 1)
    {
        s++;
        cout << " s = " << s <<"\n";
    }
    else if ( ( p++ == 2) || ( p++ == 3) || ( p++ == 4))
    {
        t++;
        cout << " t= " << t <<"\n";
    }
    cout <<" p = " << p <<"\n";
    return 0;
}
```

6. Determine the output of each of the following programs when it is executed.

(a)

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int j = 0;
    for (;j;)
       cout <<" j = " << j++ <<"\n";
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    int j = 1;
    for (;j;)
       cout <<" j = " << j++ <<"\n";
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    int j = -6;
    for (;j;)
       cout << j++ << '\t';
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    int j = -6;
    for (;;);
       cout << j++ << '\t';
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    int j = -6;
    for (;;)
       cout << j++ <<"\n";
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    int j = -6;
    for (;;j++)
```

```
        cout << j <<"\n";
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
int main()
{
    char ch;
    for (ch = 'A'; ch <= 90; ++ch)
        cout.put(ch);
    return 0;
}
```

(h)

```
#include <iostream>
using namespace std;
int main()
{
    int j;
    for ( j = 1; j <= 10; ++j)
    {
        cout <<"j = " << j << '\t';
        j++;
    }
    return 0;
}
```

(i)

```
#include <iostream>
using namespace std;
int main()
{
    for (int j = 0; j <= 10; j = j+3)
    {
        cout <<"j = " << j << '\t';
    }
    return 0;
}
```

(j)

```
#include <iostream>
using namespace std;
int main()
{

    for ( float j = 0; j <= 1; j += 0.2)
        cout <<"j = " << j << '\t';
    return 0;
}
```

7. Determine the output of each of the following programs when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    bool flag = true;
    for (int i = 0; flag; ++i)
```

```
        cout << "i = " << i <<"\n";
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    for (int i = 0; false; ++i)
        cout << "i = " << i <<"\n";
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    int i = 0;
    for (;;);
        cout << "i = " << i <<"\n";
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    int i = 0;
    for (true; ++i; false)
        cout << "i = " << i <<"\n";
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    for (float a = -1.1; true; a = a+0.5)
        cout << "a = " << a <<"\n";
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    for (char ch = 'A'; ch <= (int)ch; ++ch )
        cout << "ch = " << ch <<"\n";
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
int main()
{
```

```
    for (int i = 0;;)
        cout << "i = " << ++i <<"\n";
    return 0;
}
```

(h)

```
#include <iostream>
using namespace std;
int main()
{
    for (int i = 0;;)
        cout << "i = " << i <<"\n";
    return 0;
}
```

8. Determine the output of each of the following programs when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    bool flag = true;
    int i = 0;
    while (flag)
       cout << "i = " << i++ <<"\n";
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    bool flag = true;
    int i = 0;
    while (flag)
       cout << "i = " << ++i <<"\n";
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    int i = 0;
    while ( !!!(true) )
       cout << "i = " << ++i <<"\n";
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    int i = 0;
    while ()
       cout << "i = " << ++i <<"\n";
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    int j = 0;
    while ( j ) {
       cout << j << '\t';
       j++;
    }
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    int j = 1;
    while ( j ) {
       cout << j << '\t';
       j++;
    }
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
int main()
{
    int j = -6;
    while ( j )
       cout << j++ << '\t';
    return 0;
}
```

(h)

```
#include <iostream>
using namespace std;
int main()
{
    int i = 0,n = 10,counter = 0;
    while ( i < n){
       cout <<"Komputer \n";
       i += 4;
       cout <<"Komputer \n";
       i -= 2;
       counter++;
       cout << " i = " << i << endl;
   }
   cout <<"No of iterations = " << counter << endl;
   return 0;
}
```

(i)

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int i = 0,n = 10;
    int j = 0,inner = 0,outer = 0;
    while ( i < n){
       cout <<"Komputer \n";
       j = i % 3;
       while ( j < 4) {
          cout <<"Hello world \n";
          j += 4;
          inner++;
       }
       i += 4;
       outer++;
    }
    cout <<"No of iterations of the inner ";
    cout <<"loop = " << inner << endl;
    cout <<"No of iterations of the outer ";
    cout <<"loop = " << outer << endl;
    return 0;
}
```

9. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    bool flag = 1;
    do
       cout << " flag = " << flag << endl;
    while (flag);
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    bool flag = 1;
    do {
       cout << " flag = " << flag << endl;
       flag = !(flag);
    }
    while (flag);
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    bool flag = true;
    do {
       cout << " flag = " << flag << endl;
       !(flag);
```

```
    }
    while (flag);
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    int i = 0, n = 5;
    do {
       cout <<" Hello";
       i += 2;
       cout << " C++ world" << endl;
       i -= 1;
    }
    while (i < n);
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    int i = 0, n = 5;
    do {
       cout <<" Hello";
       i += 2;
       cout << " C++ world" << endl;
       i -= 1;
       if ( i == 3) break;
    }
    while (i < n);
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    int i = 0, n = 5;
    do {
        cout <<" Hello";
        i += 2;
        cout << " C++ world" << endl;
        i -= 1;
        if ( i == 5) {
             n = 6;
             continue;
        }
    }
    while (i < n);
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
int main()
```

```
{
    int i = 0, n = 5;
    do {
       cout <<" Hello C++ world" << endl;
       ++i;
       if ( i == 2) goto again;
    }
    while (i < n);
    again:
       cout << " i = " << i << endl;
    return 0;
}
```

PROGRAMMING EXERCISES

1. Write a program in C++ to find the square of the numbers from 1 to 100 using
(i) `for` loop
(ii) `while-do` loop
(iii) `do-while` loop
The output should be as follows:

Number Square
. .
. .

2. Modify the above program so that it prints only the squares of numbers (not the numbers) from 1 to 125, each on its own line. It should print 1 4 9 16 25, and so on.

3. Write a program in C++ that prints the numbers and its cube from 1 to 10 using the following control statements:
(i) `if-else` (ii) `for` loop
(iii) `while-do` loop (iv) `do-while` loop

4. Modify the above program so that it prints the number, square and the cubes of only odd numbers from 0 to 100. The output should be like

Number Square Cube
. . .
. . .

5. Modify program 4 so that it prints the number, square, and cube of only even numbers from 0 to 100 with the same format of output.

6. Write a program in C++ that will print all the numbers less than 2000 that are evenly divisible by 10 using
(i) `if-else` and `goto` statement (ii) `for` loop
(iii) `while-do` loop (iv) `do-while` loop
The output should be like 10 20 30 .....

7. Write a program in C++ to find the sum of the following series using
(i) `for` loop (ii) `while-do` loop
(iii) `do-while` loop (iv) `goto` statement
(a) `sum = 1- 2 + 3 - ... + n`
(b) `sum = 1 + 3 + 5 + ... + n`
(c) `sum = 1 - 2 + 4 - ... + n`

(d) $sum = 1 + \frac{2}{2!} - \frac{3}{3!} + \cdots \frac{n}{n!}$

(e) $sum = x + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots \frac{x^n}{n!}$

(f) $sum = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \frac{x^n}{n!}$

(g) $sum = 1^2 + 2^2 + 3^2 + 4^2 + \ldots + n^2$

(h) $sum = 1^3 + 2^3 + 3^3 + 4^3 + \ldots + n^3$

(i) $sum = 1 + 2^2 + 4^2 + \ldots + n^2$

(j) $sum = 1 + 3^2 + 5^2 + \ldots + n^2$

8. Write a program in C++ to generate the following series of numbers:

(i)

```
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

(ii)

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1
7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
```

9. Write a program in C++ that determines whether a given number is a prime number or not and prints it, using

(i) `for` loop

(ii) `while-do` loop

(iii) `do-while` loop

(Hint: Prime number is a number which is divisible only by 1 and by itself. 3 is a prime number since it is divisible by 1 and 3, whereas 6 is not a prime number because, it is divisible by 1, 2 and 3.)

10. Write a program in C++ to read a number `n` from the standard input device, i.e. keyboard, and again read a digit and check whether the digit is present in the number `n`. If it is so, count how many times it is repeated in the number `n`.

For example, let `n` = 12576

digit to be checked 5.

The digit is present once.

11. Write a program in C++ to read a number `n`, and digit `d`, and check whether `d` is present in the number `n`. If it is so, find out the position of `d` in the number `n`. For example,

Let `n` = 75689 and `d` = 5

The digit `d`, i.e., 5 is present at the position 4 from left to right.

12. Write a program in C++ to read a number `n` and find out the sum of the integers from 1 to 2, then from 1 to 3, then 1 to 4, and so on and to display the sum of the integers from 1 to `n`. For example,

from 1 to 2 = 1

1 to 3 = 3

1 to 4 = 6

1 to 5 = 10

1 to 6 = 18

13. Write a program in C++ to read a positive number n and to generate the following series using
(i) for loop
(ii) while-do loop
(iii) do-while loop
(a) number = 1 2 3 4 … n
(b) number = 0 2 4 6 … n
(c) number = 1 3 5 7 … n
(d) number = 1 2^2 3^2 4^2 … n^2
(e) number = 1 2^3 3^3 4^3 … n^3

14. Write a program in C++ to read a positive integer number n and to generate the numbers in the following form.
For example,
Enter a number: 5
Output 5 4 3 2 1 0 1 2 3 4 5
Enter a number: 7
Output 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7

15. Write a program in C++ to generate the following pyramid of numbers.

```
            0
          1 0 1
        2 1 0 1 2
      3 2 1 0 1 2 3
    4 3 2 1 0 1 2 3 4
  5 4 3 2 1 0 1 2 3 4 5
6 5 4 3 2 1 0 1 2 3 4 5 6
```

16. Write a program in C++ to read a hex number and to find out the 15's and 16's complement of the given hex number using comma operator in the for loop.

17. Write a program in C++ to read an octal number and to find out the 7's and 8's complement of the given octal number using comma operator in the for loop.

18. Write a program in C++ to read a set of real numbers from a standard input device and check whether any 0 has been entered. If it is so, using break statement, just stop the execution and display a message "zero entered".

19. Write a program in C++ to read a positive integer number n and to generate the following numbers up to n, using goto statement.
number = 1 2 3 4 … n
number = 0 2 4 6 … n
number = 1 3 5 7 … n
number = 1 2^2 3^2 4^2 … n^2

20. Write a program in C++ to read a positive integer number n from a standard input device and to display the number and digit. For example, n = 5678
Output
5
6
7
8

Functions and Program Structures

Chapter 6

This chapter deals with the method of declaring an user-defined function and its use in the C++ program. The scope of variables such as local, global, formal and actual arguments are explained with numerous examples. This chapter also covers how to realize recursive function that constitutes one of the salient features of C++. The standard functions which are supported for the mathematical operations, file operations, string operations are also discussed in this chapter. Advanced topics such as passing a function or function to another subprogram are explained in Chapter 8 on "Pointers".

6.1 INTRODUCTION

A complex problem can be decomposed into small or easily manageable parts. Each small module is called a subprogram. A subprogram is a program unit which performs a particular task. Subprograms are very useful to read, write, debug, modify and easy to use in the main program. It cannot execute itself and it is called either in the main program or by other subprogram. The main advantages of using a subprogram are:

- simple to write correctly a small subprogram
- easy to read, write and debug a subprogram
- easier to maintain or modify a subprogram
- it tends to be self documenting and highly readable.
- it may be called any number of times in any place with different parameters.

6.1.1 Types of Functions

In C++, a function can be classified into two types: *built-in* functions and *user-defined* functions.

```
functions —|— built-in functions
           |
           |— user-defined functions
```

Some examples for built-in functions are: mathematical libraries, namely, `cos()`, `sin()`, etc. This chapter elucidates both built–in and user-defined functions with suitable examples.

Most of the high level languages such as BASIC, FORTRAN, or Pascal allow both subroutines or procedures and functions. In these languages, the formal arguments may be passed and returned only by the subroutines or procedures. The functions may take some formal arguments and can only return one value and then only through the function name itself.

In C++, there is no difference between a function and a procedure. The functions may or may not take some formal arguments for calling a portion of the program. The function may or may not transfer back values to a called function block. It may behave like a traditional procedure or as a function or as both. Secondly, most high level languages differentiate between a main program and subprograms. In C++, all modules are called *functions* and they all have the same structure and the data type declaration and uses.

The following section explains how to use and realise the various user-defined functions in C++ for both as a procedure and as a function. In C++, there is no separate form or method to write a procedure and function, it gives the power of flexibility to the user to construct the user-defined functions to behave like traditional subprograms such as procedures and functions. There is only one form of writing a subprogram in C++ and that is one of the importance strength of the C++ language.

6.2 DEFINING A FUNCTION

A function is a subprogram or a subroutine, similar in form to a program. It must be declared in the declaration part of a program and is executed, when its name is called. A function definition has a name, a parentheses pair containing zero or more parameters and a body. For each parameter, there should be a corresponding declaration that occurs before the body. Any parameter not declared is taken to be an `int` by default. It is good programming practice to declare all parameters. The syntax diagram of function definition is given in Fig 6.1.

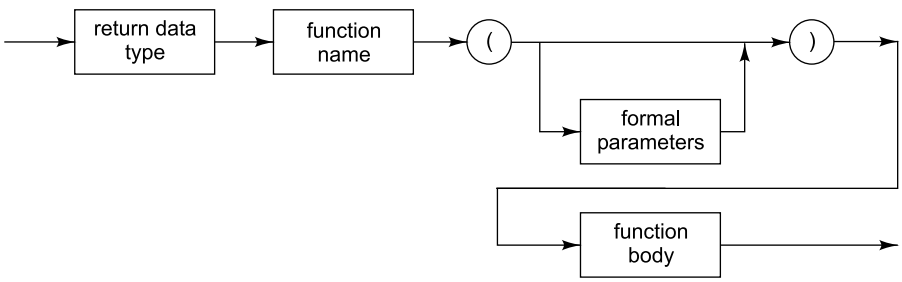


Fig. 6.1 *Syntax Diagram of Function Definition*

A function is defined the same way the main function is defined. All function definitions follow the same pattern; it's basically the function prototype with the function's body added to it. The function definition always consists of the following:

(1) function's return value
(2) function's name
(3) function's parameter list
(4) actual function body, enclosed in curly braces

The general format of the function definition is given below:

```
return_value functionname (datatype argument1,datatype argument2...)
{
     body of function
     -------
     -------
     return something
}
```

(a) Declaring the Type of a Function Function refers to the type of value it would return to the calling portion of the program. Any of the basic data types such as `int, float, char,` etc. may appear in the function declaration. In many compilers, when a function is not supposed to return any value, it may be declared as type `void,` which informs the compiler not to save any temporary space for a value to be sent back to the calling program.

For example,

```
void functionname (...)
int  functionname (...)
float functionname (...)
char functionname (...)
```

(b) Function Name The function name is an user-defined identifier and the syntax rules to form a function name is same as the rules of the variables. Normally, a function is named such that it is reminiscent of the function operation, as it is easy for the user to keep track of functions, whenever, a transfer of similar function is used in the main program.

For example,

```
counter ();
square ();
display_value();
output ();
```

(c) Formal Arguments Any variable declared in the body of a function is said to be local to that function. Other variables which are not declared either as arguments or in the function body, are considered "global" to the function and must be defined externally. The storage classes or scope of variables are discussed subsequently in this chapter.

For example,

(1)

```
void square (int a, int b) // a, b are the formal arguments
{
   -------
   -------
}
```

Note that the variables `a` and `b` are called formal arguments. Sometimes, a function may be invoked without passing any parameters.

(2)

```
int counter ( float x1, float x2, int y1, int y2)
{ // x1, x2, y1, and y2 are the formal arguments
  -------
  -------
  return (int_value);
}
```

(3)

```
float output (void)    // function without formal argument
{
   statement;
   -------
   -------
   return(float_value);
}
```

(d) Function Body After declaring the function, function name and formal arguments, a statement or a block of statements is placed between the begin and the end statements. In C++, each function is declared

almost like a main program and it consists of label, constant, type, variable declarations with begin and end statements. The syntax diagram of function body is given in Fig. 6.2.

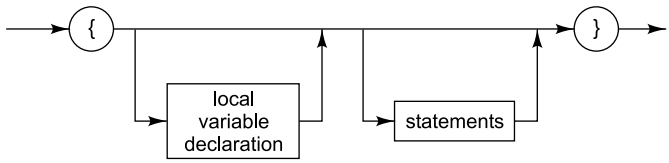


Fig. 6.2 *Syntax Diagram of Function Body*

For example,

```
void all_add ( int a,int b,int c, float x, float y)
{
    int i,j,n;   // local variables, if any
    statement_1;
    -------
    -------
    -------
    // body of the function
}
```

(e) Calling Functions In the C++ language, the act of transferring control to a function is known as calling the function. The following steps take place when a function is called:

(1) The compiler makes a note of the location from which the function was called and makes a copy of the parameter list, if any.
(2) Any storage required for the function to execute is temporarily created.
(3) The called function starts executing, using copies of the data that was supplied in the parameter list.
(4) After the function has finished executing, control is returned to the calling function, and memory used by the function is released.

6.3 THE RETURN STATEMENT

The keyword return is used to terminate function and return a value to its caller. The return statement may also be used to exit a function without returning a value. The return statement may or may not include an expression. The syntax diagram of return statement is given in Fig. 6.3.

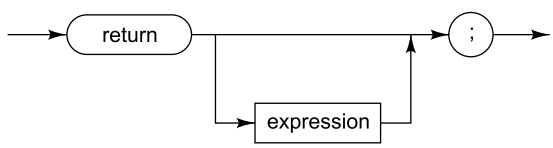


Fig. 6.3 *Syntax Diagram of Return Statement*

The general syntax of the return statement is,

```
return;
return (expression );
```

The return is a full-fledged C++ statement that can appear anywhere within a function body. A function can also have more than one return although it is good programming style for a function to have a single entrance and a single exit.

Following are some valid return statements:

```
return;
return (54);
return (x+y);
return (++i);
return ++j;   // correct, but not good style
```

The return statement terminates the execution of the function and pass on the control back to the calling environment. For example, a few valid function declarations are,

(1)

```
int sum (int x,int y)
{
    return(x+y);    // return int value
}
```

(2) A function can declare more than one return statement

```
float maximum(float a, float b)
{
    if ( a > b)
        return(a);
    else
        return(b);
}
```

6.4 FUNCTION PROTOTYPES

Even though, C++ is an object-oriented programming, still functions play a vital role in any C++ program. After all, functions do the actual work in a program.

In order to construct an efficient class for a complex software, it is essential to have sophisticated member functions. In that view, C++ supports very good features for the function usage. C++ retains the base elements of function definition, declaration and invocation from C.

The type of function in which the function returns a value and the formal arguments, if any, must be defined in the function declaration part. The following three things are to be considered, whenever a function usage is involved in C++:

- function declaration
- function definition
- function calling or invocation

In this section, how a function is declared, defined, and invocated from a portion of a program to another part, using a function, is explained.

(a) Function Declaration A function declaration is a process in which a function identifier or name, return type, the number and type of each parameter in the list are defined. The syntax diagram of function declaration is given in Fig. 6.4.

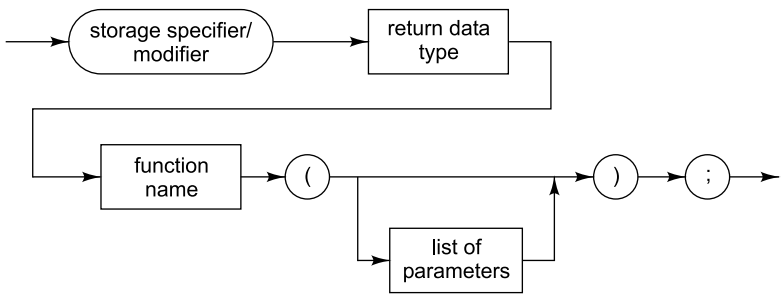


Fig. 6.4 *Syntax Diagram of Function Declaration*

The general syntax of the function declaration in C++ are given below:

```
return_type  function_name ( arg name,arg name... arg_n name);
```

Some of the valid examples for the function declaration in C++ is given below:

```
void display (void );
int calculate ( int, double, char);
void message (int &, float &);
```

Some users use an alternative form for the parameter list declaration. They include dummy variable name chosen to indicate the meaning of the parameter or arguments. For example,

```
int sum (int x, int y, float number);
```

(b) Function Definition The definition of a C++ function also follows the same style as that of a C function. The function declaration should be repeated as a header of the function definition part and the formal arguments should be declared because they are needed in the body of the function.

For example, the following program example shows how a function definition is written in C++:

```
void  funct ( int x, int y, float abc)
{
    -------
    -------
    // body of the function
    return ;
}
```

A return statement is necessary to send a value back to the calling portion of the program through the function name.

For example,

(1)

```
void square (int a, int b)
// a, b are the formal arguments
{
    -------
    -------
}
```

(2)

```
int counter ( float x1, float x2, int y1, int y2)
{     // x1, x2, y1, and y2 are the formal arguments
    -------
    -------
    return (int value);
}
```

(3)

```
float output (void) // function without formal argument
{
      statement;
      -------
      -------
      return(float value);
}
```

(c) Function Calling Calling a function in C++ is the same as that of function invocation C. A function identifier or name will stand alone as a single statement or it may be part of an expression of a statement. The syntax diagram of function call is given in Fig. 6.5.

For example,

```
funct (a,b); // stand alone
```

or

```
sum = sum (x,y); // part of an expression
```

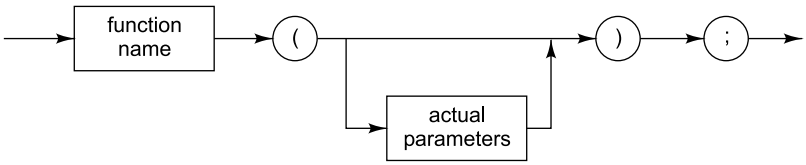


Fig. 6.5 *Syntax Diagram of Function Call*

Whenever a return value is not used in the calling portion of a program or a function must perform some activity independently of its return value, it is better to have a function as a stand alone in a line. For example,

```
swap (a,b); // no return statement is required
display (void); //displays a message
```

For example, the following program segment shows how to declare, define and invoke a function in C++.

```
void main()
{
     void funct (formal arguments, if any );
     // function declaration
     -------
     -------
     funct (actual arguments); // function invocation
     -------
     -------
}
void funct ( formal arguments ) // function definition
{
     -------
     -------
}
```

6.5 TYPES OF USER DEFINED FUNCTIONS

User-defined functions can be classified into three types based on the data communication between the called and the calling portions of the program as detailed below:

- a function is called without taking any formal arguments from the calling portion of a program. The called function also does not return any value to the calling portion (Type 1).
- a function is called by passing some formal arguments from the calling portion of a program but the called function does not return any value to the calling portion (Type 2).
- a function is called by passing some formal arguments to the function from a calling portion of a program and returns some value to the calling environment (Type 3).

6.5.1 Type 1 (No Data Communication between Functions)

It is the simplest way of writing a user-defined function in C++. There is no data communication between the calling portion of a program and called function block. The function is invoked by a calling environment by not feeding any formal arguments and also the function does not return any value to the calling portion. Only the transfer of control takes place between the main program and a function block.

The general syntax of the Type-1 category of the function usage:

```
#include <iostream>
using namespace std;
```

```
int main()
{
    void function_name (void); /* function declaration */
    -------
    -------
}
void function_name (void)
{
    -------
    -------
    /* no return data type */
}
```

For example, the following program segment illustrates how to declare and invoke a user-defined function of Type 1 category:

(1) A program contains a single function of type 1 category, namely, display ().

```
#include <iostream>
using namespace std;
int main()
{
    int x,y;
    void display (void); /* function declaration */
    -------
    -------
    display(); /* function calling */
}
void display (void) /* function definition */
{
   statement;
}
```

(2) A program contains two functions namely `inputdata` and `outputdata` of Type 1 category.

```
#include <iostream>
using namespace std;
int main()
{
    int x,y;
    void inputdata (void); /* function declaration */
    void outputdata (void);
    -------
    -------
    inputdata(); /* function calling */
    outputdata();
    -------
    -------
}
void inputdata(void) /* function definition */
{
   statement;
   -------
   -------
   /* void function does not return anything */
}
void outputdata(void) /* function definition */
{
```

```
        statement;
        -------
        -------
        /* void function does not return anything */
    }
```

PROGRAM 6.1

A program to demonstrate how to declare and invoke a function in the main program based on the Type 1 category.

```
// Example 6.1
#include <iostream>
using namespace std;
int main()
{
    void draw_box(void);
    cout <<"calling a function \n";
    draw_box();
    cout <<"Again calling the same function \n";
    draw_box();
    return 0;
}
void draw_box()
{
    cout <<"*****\n";
    cout <<"*   *\n";
    cout <<"*   *\n";
    cout <<"*****\n";
}
```

Output of the above program

```
Calling a function
*****
*   *
*   *
*****
Again calling the same function
*****
*   *
*   *
*****
```

PROGRAM 6.2

A program to display the following numbers and invoke a function in the main program based on the Type 1 category.

```
5    4    3    2    1
4    3    2    1
3    2    1
2    1
1
#include <iostream>
#include <iomanip>
using namespace std;
```

```
int main()
{
    void display(void);
    display();
    return 0;
}
void display()
{
    for (int i = 5; i >= 1;   i)
    {
        for (int j = i; j >= 1;   j)
            cout << j << setw(5);
        cout << "\n";
    }
}
```

Output of the above program

```
5    4    3    2    1
4    3    2    1
3    2    1
2    1
1
```

6.5.2 Type 2 (One Way Data Communication)

The second type of writing an user-defined function is passing the formal arguments to a function but the called function does not return any value to the caller. It is one way data communication between a calling portion of the program and the function block. In other words, Type 2 function declaration and invocation is known as the call by value or value parameter.

The general syntax of the Type 2 category of function declaration and usage is given below:

```
#include <iostream>
using namespace std;
int main()
{
   void function_name (list of parameters); /* function declaration */
}
void function_name (list of parameters) /* function definition */
{
     -------
     -------

     /* no return data type */
}
```

For example, the following program segment illustrates how to declare and invoke a user-defined function of Type 2 category:

```
#include <iostream>
using namespace std;
int main()
{
    int x,y;
    void display (int x,int y); /* function declaration */
    -------
    -------
    display(x,y); /* function calling */
}
void display (int a,int b) /* function definition */
```

```
{
    -------
    -------
    statement;
    /* function does not return anything as it has been defined
       as a void category */
}
```

PROGRAM 6.3

A program to find the area and the circumference of a circle for a given value of radius using a function type 2 category.

```
// Example 6.3
#include <iostream>
const float pi =  3.14159;
using namespace std;
int main()
{
    void area_circle (float a);
    void circumference (float a);
    float radius;
    cout <<"enter radius \n";
    cin >> radius;
    area_circle(radius);
    circumference(radius);
    return 0;
}
void area_circle(float radius)
{
    double area;
    area = pi * radius * radius;
    cout <<" Area of the circle = " << area;
    cout << "\n";
}

void circumference (float radius)
{
    double circum;
    circum = 2 * pi * radius;
    cout <<"Circumference of the circle = " << circum;
    cout << "\n";
}
```

Output of the above program

```
enter radius
10
Area of the circle = 314.159
Circumference of the circle = 62.8318
```

The list of parameters that can be defined in the function heading can be anything, such as `int`, `float` and `char`. Even the list of parameters can also be aggregated data types, namely, `arrays`, `structs` and `unions`.

The list of parameters that are defined in the function heading need not follow any order. All the parameters need not necessarily be written in the same line. However, each variable and their data type must be separated by a comma. Some of valid declarations of a function with a list of parameters are given below:

```
void sample ( int i, float abc, char ch);
```

An alternative form for declaring the above function sample is:

```
void sample ( int i,
              float abc,
              char ch);
```

Various forms of declaring formal parameters in the function heading are given below. In the following usages, the list of parameters have the same effect.

(1) `void calculate (int x, int y, int z, float abc,char ch);`

(2) `void calculate (int ,int,int,float,char);`

The following form of declaring formal parameters in the function heading gives error message:

```
void calculate (int x,y,z); /* error */
```

Each parameter of the variable in a function heading should be defined its data type individually.

```
void calculate (int x,int y,int z);
```

There is no restriction to define the list of parameters in the function header. C++ permits the user to define the list of parameters in any order but the only condition is that the number of parameters and the sequence of the data types must be same in the actual and formal argument lists. In other words, the data type and its parameters should be matched in both the function declaration heading and the function definition header.

PROGRAM 6.4

A program to find the sum of the given two numbers using a function type 2 category.

```
#include <iostream>
using namespace std;
int main()
{
    void sum(int a, int b);
    int x = 10, y = 20;
    sum(x,y);
    return 0;
}

void sum (int item1, int item2)
{
    int temp;
    temp = item1+item2;
    cout <<"sum = " << temp;
    cout << "\n";
}
```

Output of the above program

```
sum = 30
```

6.5.3 Type 3 (Two Way Data Communication)

The third type of writing a user-defined function is passing some formal arguments to a function from the calling portion of the program and the computed values, if any, is transferred back to the caller. Data is communicated both from the calling portion and a function block.

The general syntax of the Type 3 category of function declaration and usage is given below:

```
#include <iostream>
using namespace std;
int main()
{
   return_data_type function_name (list of parameters); /* function declaration */
```

```
    -------
    -------
}


// function definition
return_data_type  function_name (list of parameters)
{
    -------
    -------
    return ( return_data_type);
}
```

For example, the following program segment illustrates how to declare and invoke a user-defined function of Type 3 category:

```
#include <iostream>
using namespace std;
int main()
{
    int x,y,total;
    int sum (int x,int y); /* function declaration */
    -------
    -------
    total = sum(x,y); /* function calling */
}

int sum (int a,int b) /* function definition */
{
   statement;
   -------
   -------
   return (int_data_type);  /* function should return something
                            as it has been defined as Type 3 category */
}
```

PROGRAM 6.5

A program performs a simple arithmetic operations, such as addition, subtraction, multiplication and division on two numbers using a function of Type-3 category.

```
// Example 6.5
// demonstration of Type-3 function
#include <iostream>
using namespace std;
int main()
{
    int x,y,value;
    char ch;
    void menu();

    float newvalue;
    int add (int x,int y);
    int sub (int x,int y);
    int mult (int x,int y);
    float divd (int x,int y);
    cout <<"Enter any two integers \n";
    cin >> x >> y;
```

```
    menu();
    while (( ch = cin.get()) != 'q') {
       switch (ch) {
           case 'a':
                value = add(x,y);
                cout <<"x = " << x << "\t y = " << y;
                cout <<"\t sum = " << value;
                cout << "\n";
                break;
           case 's':
                value = sub (x,y);
                cout <<"x = " << x << "\t y =" << y;
                cout <<"\t diff = " << value;
                cout << "\n";
                break;
           case 'm':
                value = mult (x,y);
                cout <<"x = " << x <<"\t y = " << y;
                cout <<"\t product = " << value;
                cout << "\n";
                break;
           case 'd':
                newvalue = divd (x,y);
                cout <<"x = " << x << "\t y = " << y;
                cout <<"\t Quotient = " << newvalue;
                cout << "\n";
                break;
         } // end of case statement
      } // end of while loop
      return 0;
}
void menu()
{
     cout <<" a -> addition \n";
     cout <<" s -> subtraction \n";
     cout <<" m -> multiplication \n";
     cout <<" d -> division \n";
     cout <<" q -> quit \n";
}

int add (int x,int y)
{
    return (x+y);
}

int sub (int x,int y)
{
    return (x-y);
}

int mult (int x,int y)
{
    return (x*y);
}

float divd (int x,int y)
{
    return (float (x) / float (y));
}
```

Output of the above program

```
Enter any two integers
1   2
a -> addition
s -> subtraction
m -> multiplication
```

```
d -> division
q -> quit
a
x = 1  y = 2  sum = 3
s
x = 1  y = 2  diff = -1
m
x = 1  y = 2  product = 2
d
x = 1  y = 2  Quotient = 0.5
q
```

PROGRAM 6.6

A program to find the factorial of a given number function of Type 3 category.

```
// Example 6.6
#include <iostream>
using namespace std;
int main()
{
    int n;
    long int factorial (int n);
    long int fact;
    cout <<" enter a number \n";
    cin >> n;
    fact = factorial(n);
    cout <<"Factorial = " << fact << "\n";
    return 0;
}
long int factorial (int max)
{
    int i;
    long int value = 1;
    if ( max <= 0)
       return (value);
    else
    {
       for ( i = 1; i <= max; ++i)
           value *= i;
       return (value);
    }
}
```

Output of the above program

```
enter a number
5
Factorial = 120
```

PROGRAM 6.7

A program to generate a Fibonacci series of numbers using a function technique.

```
Fibonacci series is:

0       1       1       2       3       5       8       13      21
34      55      89      144     233     377
```

```
// Example 6.7
// fibonacci series
#include <iostream>
using namespace std;
int main()
{
     void fibonacci ( int max);
     int n;
     cout <<"How many terms \n";
     cin >> n;
     again : if ( n <= 0) {
          cout <<"enter only a positive number\n";
          goto again;
     }
     fibonacci(n);
     return 0;
}

void fibonacci(int max)
{
     int i,fib0,fib1,fib;
     fib0 = 0;
     fib1 = 1;
     cout <<"Fibonacci series \n";
     cout << fib0 << '\t';
     cout << fib1 << '\t';
     i = 3;
     while ( i <= max)
     {
        fib = fib0+fib1;
        cout << fib << '\t';
        fib0 = fib1;
        fib1 = fib;
        i++;
     }
     cout << "\n";
}
```

Output of the above program

```
How many terms
6
Fibonacci series
0  1  1  2  3  5
```

PROGRAM 6.8

A program to find the sum of the given series of non-negative integers using a function. sum = 1+2+3+4....n

```
// sum = 1+2+3+4 ...n
#include <iostream>
using namespace std;
int main()
{
     int find_sum(int max);
     int n,value;
     cout <<" enter a maximum number \n";
     cin >> n;
     value = find_sum(n);

     cout <<"1+2+3+4...  " << n <<" = " << value;
     cout << "\n";
     return 0;
}
```

```
int find_sum(int max)
{
   int i,value;
   value = 0;
   if ( max <= 0)
       return (value);
   else
   {
       for ( i = 1; i <= max; ++i)
           value += i;
       return (value);
   }
}
```

Output of the above program

```
enter a maximum number
10
1+2+3+4...  10 = 55
```

PROGRAM 6.9

A program to generate power series using a function technique.

```
// Example 6.9
// generation of power series
#include <iostream>
using namespace std;
int main()
{
    void power_series(float base,int n);
    float x;
    int n;
    again: cout <<"enter values for the base and n \n";
    cin >> x >> n;
    if ( n <= 0)
    {
        cout <<"enter only a positive number for n \n";
        goto again;
    }
    cout <<"Power series \n";
    power_series(x,n);
    return 0;
}

void power_series(float base,int n)
{
    float power = 1;
    int i = 1;
    cout <<"counter     x      x^counter  \n";
    cout <<"-----------------------------------\n";
    while ( i <= n)
    {
        power = power*base;
        cout << i <<'\t'<< base <<'\t'<< power;
        i++;
        cout <<"\n";
     }
}
```

Output of the above program

```
enter values for the base and n
2  10
Power series
```

counter	x	x^counter
1	2	2
2	2	4
3	2	8
4	2	16
5	2	32
6	2	64
7	2	128
8	2	256
9	2	512
10	2	1024

PROGRAM 6.10

A program to find the sum of the following series using a function.

$$\text{sum} = 1 - \frac{1}{2!} + \frac{1}{3!} - \frac{1}{4!} + \cdots \frac{1}{n!}$$

```
// Example 6.10
// sum = 1-(1/2!)+ (1/3!)-(1/4!)... (1/n!)
#include <iostream>
using namespace std;
int main()
{
    int n;
    void calculate(int j);
    cout <<"Enter value for n \n";
    again:
    cin >> n;
    if ( n <= 0) {
       cout <<"enter only a positive number \n";
       goto again;
    }
    calculate(n);
    return 0;
}
void calculate (int n)
{
    float sum = 0.0,newtemp;
    int i,sign,temp;
    int factorial (int i);
    sign = 1;
    i = 1;
    while ( i <= n) {
       temp = factorial(i);
       newtemp = (float)1/temp;
       sum = sum+sign*newtemp;
       cout <<"i = " << i <<'\t' << "fact = "<< temp;
       cout <<"\t sign = " << sign << '\t' <<" sum = " << sum;
       cout << "\n";
       sign = (-1)*sign;
       i++;
    }
    cout <<" sum = 1-(1/2!)+(1/3!)-... = " << sum;
    cout <<"\n";
}
int factorial (int i)
{
    int j,value = 1;
    for ( j = 1; j <= i; j++)
```

```
        value *=j;
    return (value);
}
```

Output of the above program

```
Enter value for n
5

i = 1      fact = 1       sign = 1       sum = 1
i = 2      fact = 2       sign = -1      sum = 0.5
i = 3      fact = 6       sign = 1       sum = 0.666667
i = 4      fact = 24      sign = -1      sum = 0.625
i = 5      fact = 120     sign = 1       sum = 0.633333
sum = 1-(1/2!)+(1/3!)-... = 0.633333
```

PROGRAM 6.11

A program to find the sum of the following series using a function.

$$\text{sum} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \frac{x^n}{n!}$$

```
// Example 6.11
// sum = x - (x^3/3!) + (x^5/5!) + ...
#include <iostream>
using namespace std;
int main()
{
    int n;
    float x;
    void calculate(float base, int max);
    cout <<"Enter the value for x and n \n";
    again:
    cin >> x >> n;
    if ( n <= 0) {
        cout <<"enter only a positive number \n";
        goto again;
    }
    calculate(x,n);
    return 0;
}
void calculate (float x, int n)
{
    float sum = 0.0,temp,pow;
    int i,sign,fact;
    int factorial (int i);
    float calpower (float x,int i);
    sign = 1;
    i = 1;
    while ( i <= n) {
       pow = calpower(x,i);
       fact = factorial(i);
       temp = (float)pow/fact;
       sum = sum+sign*temp;
       cout <<"i = " << i << '\t' <<" fact = " << fact;
       cout << "\t pow = " << pow << '\t';
       cout <<" sign = " << sign <<'\t'<<"sum = " << sum;
       cout << "\n";
       sign = (-1)*sign;
       i = i+2;
    }
    cout <<" sum = x-(x^3/3!)+(x^5/5!)-...= " << sum;
```

```
    cout << "\n";
}

float calpower(float base,int max)
{
    int j = 1;
    float power;
    power = 1;
    while ( j <= max) {
        power = power*base;
        j++;
    }
    return (power);
}

int factorial (int i)
{
    int j,value = 1;
    for ( j = 1; j <= i; j++)
        value *=j;
    return (value);
}
```

Output of the above program

```
Enter the value for x and n
2 5

i = 1   fact = 1      pow = 2     sign = 1     sum = 2
i = 3   fact = 6      pow = 8     sign = -1    sum = 0.666667
i = 5   fact = 120    pow = 32    sign = 1     sum = 0.933333

sum = x-(x^3/3!)+(x^5/5!)-...= 0.933333
```

6.6 ACTUAL AND FORMAL ARGUMENTS

Arguments are user-defined identifiers or variables, constants and other program elements. Sometimes, these arguments are also called as parameters. They are defined in the subprogram definition part of a program or at the time of invoking a subprogram such as a function. Generally, arguments can be classified into two types: actual and formal data types.

```
Arguments  |  actual arguments
           |
           |  formal arguments
```

The comparison and contrast between the actual and formal arguments are discussed below in detailed manner.

6.6.1 Actual Arguments

Actual argument is a variable or expression contained in a function call that replaces the formal parameter (which is the part of the function declaration). The general syntax of declaring formal and actual arguments in a function is given below:

```
#include <iostream>
using namespace std;
int main()
{
    function declaration_part (list of formal arguments);
    local variable declaration_part;
```

```
    -------
    -------
    function_invocation(list of actual arguments);
}

function_definition_part (list of formal arguments)
{
    -------
    -------
}
```

The formal arguments and actual arguments can have the same name or it can be different. But the order of defining the arguments and their data types must be the same both in the actual and in the formal argument list. In other words, the data type in which it is defined in the formal arguments must match with the actual arguments.

For example, consider the following program segment in which actual and formal arguments are declared:

```
#include <iostream>
using namespace std;
int main()
{
    int funct (int a, int b); /* a and b formal parameters */
    int x,y,temp;
    -------
    -------
    funct(x,y);  /* x and y are the actual parameters */
}

int funct (int a, int b)  /* a and b formal parameters */
{
    local variable declaration, if any
    -------
    -------
    return (int_data_type);
}
```

Note that the variables x and y are actual arguments which are defined in the main program. The contents of the actual parameters are copied onto the formal arguments in a subprogram whenever they are invoked through a function call. In the case of variable parameters in a subprogram, the address of the actual and formal arguments are the same. In other words, formal arguments are dummy variables that are created by the C++ compiler at the time of using a particular subprogram. When the control of the program moves to the other part of the program, dummy variables which are created already will be purged automatically.

6.6.2 Formal Arguments

Formal arguments are user-defined identifiers that are declared in a function header and they are used throughout the function. Whenever the function is invoked, formal parameter is replaced by actual parameter.

The use of formal arguments are illustrated in the following program segment:

```
#include <iostream>
using namespace std;
int main()
{
      void sum(int a, int b); // a and b are formal arguments
      int x = 10, y = 20;
      sum(x,y); // actual arguments
```

```
}

void sum (int item1, int item2) // item1 and item2 are formal arguments
{
     int temp;
     temp = item1+item2;
     cout << "sum =" << temp;
}
```

The variable declared in the function declaration part is called the formal or dummy arguments. The order of the parameters must be the same both in the main and in the subprogram.

For example, the following program is an invalid way of declaring actual and formal arguments:

```
#include <iostream>
using namespace std;
int main()
{
    void display( float x1, float y2, char ch1, char ch2, int t1, int t2); //* function declaration */
    int x,y;
    char s1,s2;
    float a,b;
    -------
    -------
    display (x,y,s1,s2,a,b);  /* data mismatch error */
}

void display (float x, float y, char s1, char s2, int t1,int t2)
{
    -------
    -------
}
```

The arguments that are declared in the function invocation do not match with the parameter that are defined in the actual function implementation. For example, in the above program segment, `x` and `y` are the variables that are defined as integer data in the function invocation but in the function definition part, these are declared as real data type. Hence, C++ compiler treats these variables as separate and finds data mismatch between formal and actual parameters list.

6.7 LOCAL VS GLOBAL VARIABLES

In general, variables which are declared in a program can be classified into two types: local and global.

```
variables --|-- local variables
            |
            |-- global variables
```

This section explains how these variables can be defined and used in a program. The comparison and contrast between the local and global variables are also discussed with suitable examples.

6.7.1 Local Variables

Identifiers that are declared such as label, const, type, variables in a function block are said to belong to the particular subprogram or block and these identifiers are known as local parameters or variables.

For example, consider the following program segment wherein a local variable is declared and used:

(1)

```
#include <iostream>
using namespace std;
int main()
{
      int function_f11( int x, int y);
      int x,y,z;
      -------
      -------
      function_f11 (x,y);
}
int function_f11 ( int x, int y)
{
      int temp,i,j;   // local variables
      -------
      -------
}
```

(2) In another example, the local variable is declared within the different block of a function with the same name. In each block, within the curly braces {and}, the local variables are different entities. The life and scope of the local variables are pertained only to the particular block. When the control is terminated from the particular block, the local variables, if any, will be purged automatically by the compiler. The life and scope of the variables are only limited to the particular block or a function module.

PROGRAM 6.12

A program to illustrate how to declare the local variable.

```
#include <iostream>
using namespace std;
int main()
{
    void display();
    cout <<" demonstrating local variables" << endl;
    display();
    return 0;
}

void display(void)
{
    int i = 10;
    {
        int i = 20;
        {
            int i = 30;
            cout << "innermost i = "  << i;
            cout << endl;
        }
        cout << " inner i = " << i;
        cout << endl;
    }
    cout << "outer i = " << i;
    cout << endl;
}
```

Output of the above program

```
demonstrating local variables
inner most i = 30
```

```
inner i = 20
outer i = 10
```

6.7.2 Global Variables

Global variables are declared in the declaration part of the main program block. The scope of the variables are same both in the main and in the subprogram.

For example, the following program segment illustrates the usage of the global variable declaration.

```
#include <iostream>
using namespace std;
int x = 10; /* global variable */
int main()
{
    void funct1(void);
    void funct2(void);
    int y;

    cout <<" value of x in the main program = x;
    -------
    -------
    funct1();
    funct2();
}

void funct1(void)
{
    -------
    -------
    cout <<" value of x inside funct1 = " << x;
}

void funct2(void)
{
    -------
    -------
    cout <<" value of x inside funct2 = " << x;
}
```

Since the variable x is declared as global, it can be accessed both in the main and in the subprograms. The value of the variable x will be same in both the functions `funct1 ()` and `funct2 ()`.

6.8 DEFAULT ARGUMENTS

One of the most useful facilities available in C++ is the facility to define default argument values for functions. In the function prototype declaration, the default values are given. Whenever a call is made to a function without specifying an argument, the program will automatically assign values to the parameters from the default function prototype declaration. Default arguments facilitate easy development and maintenance of programs.

For example, the following program segment illustrates the default argument declaration:

```
#include <iostream>
using namespace std;
void sum ( int x = 10, int y = 20); // function prototype declaration
                                    // with default argument list
```

```
void main()
{
     int a,b;
     sum (); // function calling, with default parameters
     -------
     -------
}

void sum(int a1,int a2) // function definition
{
     int temp;
     temp = a1+a2; // a1 = 10 and a2 = 20 by default arguments
}
```

PROGRAM 6.13

A program to find the sum of the given numbers using default argument declaration.

```
//default argument declaration
#include <iostream>
using namespace std;
void sum ( int a,int b,int c = 6, int d = 10);
void main()
{
     int a,b,c,d;
     cout << " enter any two numbers \n";
     cin >> a >>b;
     sum (a,b); // sum of default values
}

void sum (int a1,int a2,int a3,int a4)
{
     int temp;
     temp = a1+a2+a3+a4;
     cout << " a = " << a1 << endl;
     cout << " b = " << a2 << endl;
     cout << " c = " << a3 << endl;
     cout << " d = " << a4 << endl;
     cout << " sum = " << temp;
}
```

Output of the above program

```
enter any two numbers
11  21

a = 11
b = 21
c = 6
d = 10
sum = 48
```

The above program can be slightly modified, invoking the function `sum()` with user-defined input data as parameters. By this, the default values are not assigned to the function `sum()`.

```
//default argument declaration
#include <iostream>
using namespace std;
void main()
{
```

```
    void sum ( int a = 2,int b = 4,int c = 6, int d = 10);
    int a,b,c,d;
    cout << " enter four numbers \n";
    cin >> a >> b >> c >> d;
    sum (a,b,c,d);
}

void sum (int a1,int a2,int a3,int a4)
{
    int temp;
    temp = a1+a2+a3+a4;
    cout << " a = " << a1 << endl;
    cout << " b = " << a2 << endl;
    cout << " c = " << a3 << endl;
    cout << " d = " << a4 << endl;
    cout << " sum = " << temp;
}
```

Output of the above program

```
enter four numbers
1 2 3 4

a = 1
b = 2
c = 3
d = 4
sum = 10
```

The function call without arguments is valid in C++. The default arguments are given only in the function prototypes and should not be repeated in the function definition. The following program calculates the sum of the default values when the function is called without arguments.

```
//default argument declaration
#include <iostream>
using namespace std;
void sum ( int a = 2,int b = 4,int c = 6, int d = 10);
void main()
{
     int a,b,c,d;
     sum (); // sum of default values
}

void sum (int a1,int a2,int a3,int a4)
{
     int temp;
     temp = a1+a2+a3+a4;
     cout << " a = " << a1 << endl;
     cout << " b = " << a2 << endl;
     cout << " c = " << a3 << endl;
     cout << " d = " << a4 << endl;
     cout << " sum = " << temp;
}
```

Output of the above program

```
a = 2
b = 4
c = 6
```

```
d = 10
sum = 22
```

A few special cases of the function prototypes with default arguments and invoking a function are illustrated below.

Case 1

```
//default argument declaration
#include <iostream>
using namespace std;
void main()
{
     void sum ( int a = 2,int b,int c = 6, int d = 10);
     -------
     -------
     sum (b); // invalid
}
```

Error: The C++ compiler displays the error message as the default value missing following the parameter "a".

Case 2

```
//default argument declaration
#include <iostream>
using namespace std;
void main()
{
     void sum ( int a = 2,int b = 3,int c, int d);
     -------
     -------
     sum (c,d); // invalid
}
```

A function may have more than one default parameter. The default parameters must be grouped consecutively and are available only at the end of a function declaration. Following is a valid way of a function declaration and calling a function with default arguments.

Case 3

```
//default argument declaration
#include <iostream>
using namespace std;
void main()
{
     void sum ( int a ,int b ,int c = 5 , int d = 8);
     -------
     -------
     sum (a,b); // valid
}
```

6.9 STRUCTURE OF THE C++ PROGRAM

C++ is a block structure language. The main idea of using a block-structured language is the easiness with which a program can be written and to realise a complex program in a modular form. In other words, a modular program is a systematic development of a language style in which there is a main part containing various functions and function calls.

In order to exploit the full potential of the modular approach, it is necessary to invoke one or more modules within another or by itself. In this context, it will be convenient to introduce the term block to refer to the whole of any program and a module.

For example, the following program segment illustrates the complete structure of the C++ program:

```
#include <iostream>
using namespace std;
int main()
{
    variable declaration part
    function declaration part
    -------
    -------
    function calling
}

function1_definition_part()
{
    local variable declaration part;
    -------
    -------
}

function2_definition_part()
{
    local variable declaration part;
    -------
    -------
}

function_n_definition_part()
{
    local variable declaration part;
    -------
    -------
}
```

Note that the structure of a C++ program consists of a program header followed by a block. A block is defined as a sequence of declarations, followed by a set of statements enclosed within begin and end. The following program segment illustrates the various program structures and styles of writing a program in C++:

(1) A simple program structure with main function:

```
#include <iostream>
using namespace std;
int main()
{
    declaration part, if any
    -------
    -------
    return 0;
}
```

(2) A program consists of a set of functions. For example, functions `funct1()` and `funct2()` are declared as a part of the program.

```
#include <iostream>
using namespace std;
int main()
{
    variable declaration part
    funct1 declaration part
    funct2 declaration part
    -------
    -------
    funct1_calling
    funct2_calling

}

funct1_definition_part()
{
    local variable declaration part;
    -------
    -------
}

funct2_definition_part()
{
    local variable declaration part;
    -------
    -------
}
```

(3) A more complex program structure is given below, which consists of two functions, namely, `funct1()` and `funct2()`. Within the function `funct1()`, two more functions `funct11()` and `funct12()` have been declared. Inside the function `funct2()`, another function `funct21()` is defined. The various invocation of these functions are also marked in the corresponding part of the program.

```
#include <iostream>
using namespace std;
int main()
{
    variable declaration part
    funct1 declaration part
    funct2 declaration part
    -------
    -------
    funct1_calling
    funct2_calling

}

funct1_definition_part()
{
    local variable declaration part;
    funct11 declaration part
    funct12 declaration part
    -------
    -------
    funct11_calling;
```

```
    funct12_calling;
}

funct11_definition_part()
{
    local variable declaration part;
    -------
    -------
}

funct12_definition_part()
{
    local variable declaration part;
    -------
    -------
}

funct2_definition_part()
{
    local variable declaration part;
    funct21 declaration part
    -------
    -------
    funct21_calling;
}

funct21_definition_part()
{
    local variable declaration part;
    -------
    -------
}
```

6.10 ORDER OF THE FUNCTION DECLARATION

The order of a function declaration is one of the important steps in writing a complex and multiple usages of the functions. In C++, the hierarchy is not very strictly maintained for function usage in a program. Users can have a choice to design or to use their own style or pattern of declaring functions. In some programming languages such as Pascal, Modula-2 or Ada, the function which is first referenced must be defined first and so on. In other words, a called subprogram must be defined above the calling portion of a program.

A function calls as "calling up" other functions and put called functions above the ones doing the calling. But in C++, it is so flexible that user can have its own style of pattern to write his functions. The only thing the ANSI C++ expects from the user that function declaration, function definition and function invocation should be maintained.

For example, the following program segment shows how to define a called function which must be defined above the function which is calling other functions:

Case 1

```
#include <iostream>
using namespace std;
int main()
{
```

```
      function first(list of parameters); /* function declaration
      function second(list of parameters);
      function third( list of parameters) ;
      -------
      -------
      first(); /* function invocations */
      second();
      third();
}

function first(list of parameters) /* function definition */
{
      -------
      -------
} /* call up function */

function second (list of parameters) /* function definition */
{
      -------
      -------
} /* call up function */

function third (list of parameters) /* function definition */
{
      -------
      -------
} /* call up function */
```

In the above program segment, three functions are declared, in which third function calls the first and second functions and so the same order is maintained for declaring the functions. Function first is defined initially which is followed by the second function. The third function is defined after the first and second functions but before the main program block.

Case 2 The function definitions can be carried out even before the main() function declaration part.

```
#include <iostream>
using namespace std;

function first(list of parameters) /* function definition */
{
      -------
      -------
} /* call up function */

function second (list of parameters) /* function definition */
{
      -------
      -------
} /* call up function */

function third (list of parameters) /* function definition */
{
      -------
      -------
} /* call up function */
```

```
void main()
{

     function first(list of parameters); /* function declaration */
     function second(list of parameters);
     function third(list of parameters) ;
     -------
     -------
     first(); /* function invocations */
     second();
     third();
}
```

Case 3 The order of defining the functions in C++ is so flexible that user can choose his own choice of writing in any order. The function definition need not be followed the same hierarchy of the functions declaration in the main() program part.

```
#include <iostream>
using namespace std;
int main()
{

     function first(list of parameters); /* function declaration */
     function second(list of parameters);
     function third(list of parameters) ;
     -------
     -------
     first(); /* function invocations */
     second();
     third();
}

function first(list of parameters) /* function definition */
{
     -------
     -------
} /* call up function */

function third (list of parameters) /* function definition */
{
     -------
     -------
} /* call up function */

function second (list of parameters) /* function definition */
{
     -------
     -------
} /* call up function */
```

In the above program segment, three functions are declared, in which third function is defined between the function definitions of first and second.

6.11 MUTUALLY INVOCATED FUNCTIONS

It may be recalled that a function cannot be invoked unless the function declaration appears before the statement that invokes it. However, sometimes it may be necessary to call a function that has not been declared before the invoking statement.

For example, one may have functions `one` and `two`, each of which calls the other:

```
/* function definition part */
function one ( list of paramters);
{
    declaration of function two();
    statementes, if any;
    two (actual paramters )
}

/* function definition part */
function two( list of parameters )
{
    declaration of function one();
    statementes, if any;
    one (actual paramters )
}
```

As written above, function `two` is correct, since it references function `one` which has been defined. But function `one` is not correct as it references function `two` which has not been defined. Reversing the order of the direction does not solve the above problem, as in that case, function `two` will inherit the error. When there is a mutual recursion call taking place, then one of the functions must be defined earlier, before it is referenced in the call up function.

```
/* function definition part */
function one ( list of paramters);
{
   statementes;
   two (actual paramters )
}

/* function definition part */
function two( list of parameters )
{
   statementes;
   one (actual paramters )
}
```

PROGRAM 6.14

A program to demonstrate how to call up the functions mutually each other in C++.

```
// demonstration of function with mutual invocations
#include <iostream>
using namespace std;
int main()
{
    int x,i,sum;
    int one (int y, int sum);
```

```
    x = 10;
    sum = 0;
    one (x,sum);
    return 0;
}
int one (int x, int sum)
{
    int two (int y, int sum);
    sum = x+sum;
    cout <<" inside the function one (sum = " << sum << ")\n";
    if (sum != 100 )
       two(x,sum);
}

int two (int y, int sum)
{
    int one (int x, int sum);
    sum = y+sum;
    cout <<" inside the function two (sum = " << sum << ")\n";
    if (sum != 100)
        one(y,sum);
}
```

Output of the above program

```
inside the function one (sum = 10)
inside the function two (sum = 20)
inside the function one (sum = 30)
inside the function two (sum = 40)
inside the function one (sum = 50)
inside the function two (sum = 60)
inside the function one (sum = 70)
inside the function two (sum = 80)
inside the function one (sum = 90)
inside the function two (sum = 100)
```

6.12 NESTED FUNCTIONS

When a function is written within another function, it is called a *nested function*. The number of nesting of functions (within another function) is virtually infinitive in C++. However, the number of nesting of the functions is restricted depending on the particular version compiler being used a system. The scope and the arguments that are declared and used in nested functions are complicated for debugging, testing and maintaining the source code.

For example, the following program segment illustrates how to declare a function within another function (nested functions) in C++:

Case 1

```
#include <iostream>
using namespace std;
int main()
{
    function f1();/* declaration */
    variable declaration; /* main program  */
    -------
    -------
    f1();
}

function f1() /* definition part */
```

```
{
    function f11(); /* declaration */
    /* variables local to f1 */
    -------
    -------
    f11(); /* calling the function f11() */
}

function f11() /* definition part */
{
    /* variables local to f11 */
    -------
    -------
}
```

Within the function `f1()`, another function `f11()` is declared. The parameters that are declared in the function `f11()` can be accessed by both functions `f11()` and `f1()`. The scope of the function `f11()` is limited only to the function `f1()`.

Case 2 Another example for the nested function is given below:

```
#include <iostream>
using namespace std;
int main()
{
    function f1();/* declaration */
    variable declaration;
    /* main program  */
    -------
    -------
    f1();
}

function f1() /* definition part */
{
    function f11(); /* declaration */
    function f12();
    /* variables local to f1 */
    -------
    -------
    f11(); /* calling the function f11() */
    f12();
}

function f11() /* definition part */
{
    /* variables local to f11 */
    -------
    -------
}

function f12() /* definition part */
{
    /* variables local to f12 */
    -------
    -------
}
```

In the above program segment, function `f1()` that is embedded of two other functions `f11()` and `f12()`. The scope of the variables of the functions `f11()` and function `f12()` are limited only to the function `f1()`. However, the function `f11()` is independent of the function `f12()` and so the variables that are declared within the function `f11()` cannot be accessed by the function `f12()` or vice versa.

6.13 SCOPE RULES

The scope of the variable is defined as a set of functions within which the variable is defined. The variable may be declared in any number of places either in the main program or in the subprogram. Scope rules are limited to the particular block or a subprogram. The data type is limited to a particular block even if it is used with the same variable name.

For example, the following program segment illustrates how to declare the same variable name in different functions of the same program.

```
#include <iostream>
using namespace std;
int main()
{
    function_f1();
    function_f2();
    int x;
    -------
    -------
}

function_f1()
{
    float y;
    y = -30;  /* variable y belongs to function f1() */
    -------
    -------
}

function_f2()
{
    char y;
    y = 'c'; /* variable y belongs to function f2() */
}
```

The variable `y` is declared in the functions `f1()` and `f2()` with different data types which is permitted in C++, as the variables that are declared in the function `f1()` are local to that particular function block and these variables are not seen in other functions. Hence, C++ compiler treats these variables as separate arguments with its own data types.

Each function declaration has a structure similar to a program, i.e., each consists of a heading and a block. Hence, function declarations may be nested within other functions. Labels, constants, type, variables, and function declarations are local to the function in which they are declared. That is, their identifiers have significance only within the program text that constitutes the block. This region of program text is called the scope of these identifiers.

As blocks may be nested, so also scopes. Objects that are declared in the main program, i.e. not local to some function or function, are called global and have significance throughout the entire program. The scope of an identifier declaration is the block in which the declaration occur, and all blocks are nested within that

block. If an identifier is declared in a block and if another block is nested within that block, then the scope of the first declaration specifically excludes the second block and any block it contains.

Consider the following program segment which illustrates how scope rules are applicable to the various parameters in the different function blocks:

```
#include <iostream>
using namespace std;
int main()
{
    function_f1();
    function_f2();
    -------
    -------
}

function_f1()
{
    function_f11(); /* declaration */
    -------
    -------
}
function_f11()
{
    function_f11_inner();/* declaration */
    local variables, if any
    -------
    -------
    function_f11_inner();
}

function_f11_inner()
{
    local variable, if any
    -------
    -------
}

function_f2()
{
    function_f21(); /* declaration */
    function_f22();
    -------
    -------
}

function_21()
{
    local variables, if any
    -------
    -------
}
```

```
function_22()
{
    local variables, if any
    -------
    -------
}
```

The following table summarises the scopes of the individual function blocks and their accessibility in other blocks or modules of the above program segment is given below:

Block	Modules which access the blocks
main()	main()
function_f1()	function_f1(), main()
function_f11()	function_f11(), function_f1(), main()
function_f11_inner()	function_f11_inner(), function_f11() function_f1(), main()
function_f2()	function_f2(), main()
function_f21()	function_f21(), function_f2(), main()
function_f22()	function_22(), function_f2, main()

6.14 SIDE EFFECTS

Software testing and maintenance are the important and time consuming tasks. When a program is defined in which the subprogram alters the value of the global variable and if it is not echoed in the main program part or block, then debugging, testing and maintaining such codes are really time consuming and it is also difficult to trace the errors in them.

Alteration of a global variable by a subprogram is called a side effect. The accidental alteration of global value during the invocation of a subprogram can be an extremely difficult error to detect and correct.

PROGRAM 6.15

A program that demonstrates how a side effect occurs when a global variable is altered in a function.

```
// demonstration of side effects
#include <iostream>
using namespace std;
int global_g = 0;
int main()
{
    int sum (int x);
    cout <<" sum1 = " << sum(0);
    cout <<" \n sum2 = " << sum(0);
    cout <<" \n sum3 = " << sum(0);
    cout <<" \n sum4 = " << sum(0);
    return 0;
}

int sum (int x)
{
    x = x+global_g;
    global_g = global_g +1; // global values is altered (side effects)

    return(x);
}
```

Output of the above program

```
sum1 = 0
sum2 = 1
sum3 = 2
sum4 = 3
```

A function should perform some task using the input parameters and return a single value. A function can modify the contents of a global variable and perform input/output statement. The operation in which the content of global variable is altered in a subprogram is called as a side effect. In most cases, side effects should be avoided. Side effects can cause program bugs that may be difficult to identify and correct. All information passed to or from a function should be done through the parameter list and not through global variables. This will make the function a self-contained independent module, which can be tested and debugged on its own.

PROGRAM 6.16

A program to demonstrate how to prevent the side effects of global variables being declared as const data type.

```
// prevention of side effects
#include <iostream>
using namespace std;
const int global_g = 0;
int main()
{
    int sum (int x);
    cout <<" sum1 = " << sum(0);
    cout <<" \n sum2 = " << sum(0);
    cout <<" \n sum3 = " << sum(0);
    cout <<" \n sum4 = " << sum(0);
    return 0;
}

int sum (int x)
{
    x = x+global_g;
    global_g = global_g +1; // error, global values cannot be
                            // altered, there is no side effects
    return(x);
}
```

Compilation error `const` value is for `read only`. It cannot be changed. In this way, one can avoid side effects even if there is a global variable.

6.15 STORAGE CLASS SPECIFIERS

The storage class specifier refers to how widely it is known among a set of functions in a program. In other words, how the memory reference is carried out for a variable. Every identifier in C++ has not only a type such as integer, char, double, and so on but also a storage class that provides information about its visibility, lifetime and location. For example, if the variable belongs to an automatic type, that means whenever the variable is used in a main program or a function, the computer will automatically reserve a memory space for it. Normally, in C++, a variable can be declared as any one of the following groups:

(1) Automatic variable
(2) Register variable

(3) Static variable
(4) External variable

The keywords that are used for declaring the storage class specifiers in C++ are as follows:

- `auto`
- `register`
- `static`
- `extern`

The syntax diagram of a variable storage specifier is given in Fig. 6.6

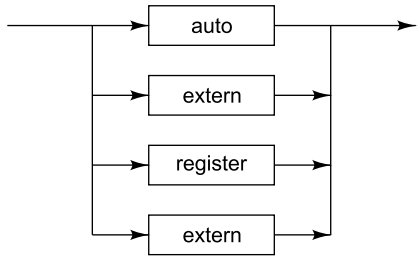


Fig. 6.6 *Variable storage specifier*

Three variable storage modifiers are used in ANSI/ISO C++:

- `const`
- `volatile`
- `mutable`

Any of the storage class modifiers may appear before or after the variable name in a declaration but by convention they come before the variable name.

The syntax diagram of a variable storage modifier is given in Fig. 6.7.

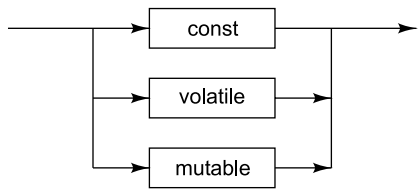


Fig. 6.7 *Variable storage modifier*

6.15.1 Automatic Variable

The auto and register storage-class specifiers can be used only to declare names used in blocks or to declare formal arguments to functions. The term "`auto`" comes from the fact that storage for these objects is automatically allocated at run time (normally on the program's stack).

The auto storage-class specifier declares an automatic variable, a variable with a local lifetime. It is the default storage-class specifier for block-scoped variable declarations. An auto variable is visible only in the block in which it is declared. Since variables with auto storage class are not initialised automatically, one should either explicitly initialise them when it is declared, or assign initial values to them in statements within the block. The values of uninitialised auto variables are undefined. (A local variable of auto or register storage class is initialised each time it comes in scope if an initialiser is given.)

Automatic variables can be declared not only at the beginning of a function but also at the beginning of a compound statement (also called a block). Local variables are given the storage class auto by default. One can use the keyword auto to make the storage class explicit but no one does. In other words, the keyword `auto` is somewhat superfluous and is rarely used.

For example, the following program segment illustrates how the keyword `auto` is used in the C++ program.

```
#include <iostream>
using namespace std;
int main()
{
    auto int a,b,c;
    -------
    -------
}
```

is exactly equivalent to

```
#include <iostream>
using namespace std;
int main()
{
    int a,b,c;
    -------
    -------
}
```

The general syntax of an automatic variable is,

```
storage_class data_type  variable_1,variable_2...variable_n;
```

Here the storage class is an automatic, so it can be written as

```
auto int x,y,z;
auto float a1,a2;
auto char name1;
```

But the above declaration can also be written as

```
int x,y,z;
float a1,a2;
char name1;
```

However, both the declarations are same. The keyword `auto` is used only if one desires to declare a variable explicitly as an automatic variable. However, a variable which is declared as an automatic cannot be accessed outside of the function. The scope of the automatic variable is within a block and the duration is also temporal. The keyword `auto` can be used as function arguments.

PROGRAM 6.17

In the following program the identifier `'i'` is used for three distinct variables.

```
// using auto storage identifier
#include <iostream>
using namespace std;
int main()
{
    void display ();
    int i = 10; // inside the main
    display();
    cout << "value of i (inside main) = " << i;
    cout << '\n';
    return 0;
}

void display()
{
   int i = 20; // i in the local variable
   {
       int i = 3;  // i within the compound statement
       cout << "value of i in the compound statement = " << i;
       cout << '\n';
       {
           int i = -5; // innermost declaration
           cout << "value of i (innermost ) = " << i;
           cout << '\n';
       }
    }
    cout << "value of i (local to the function) = " << i;
    cout << '\n';
}
```

Output of the above program

value of i in the compound statement = 3
value of i (innermost) = – 5
value of i (local to the function) = 20
value of i (inside main) = 10

The automatic variables have two distinct advantages. First, memory space is used economically since it is used only as long as it is needed. Secondly, their local scope prevent from affecting other functions due to inadvertent usage. Hence, variables in other functions need not necessarily be given different names.

6.15.2 Register Variable

The `register` keyword specifies that the variable is to be stored in a machine register, if possible. The keyword 'register' is used for automatic variables that are accessed very frequently in a program. The keyword `register` may be used only for variables declared within a function.

Automatic variables are stored in the memory. As accessing a memory location takes time (much more time than accessing one of the machine's registers), one can make the computer to keep only a limited number of variables in their registers for fast processing. Whenever some variables are to be read or repeatedly used, they can be assigned as register variables.

The general syntax of a register variable is,

```
register datatype variable1,variable2......variable n;
```

The keyword register is used to declare that the storage class of the variable is a register type.

For a limited number of variables it is possible to make the computer to keep them permanently in fast registers. Then the keyword register is added in their declaration. For example,

```
function (register int n)
{
     register char temp;
     -------
     -------

}
```

If possible, machine registers sometimes called *accumulators*, can be assigned to the variable n and temp, which would increase the speed. If there are not enough register variables then the request will simply be ignored. However, a variable which is declared as a register cannot be accessed outside of the function. The scope of the register variable is within a block and the duration is also temporal. The keyword `register` can be used as function arguments.

PROGRAM 6.18

A program to display the number and its square from 0 to 10 using register variables.

```
// using register variable declaration
#include <iostream>
using namespace std;
int main()
{
    // function declaration
    int funct (register int x, register int y);
    register int x,y,z;
    x = 0;
```

```
    y = 0;
    cout << "x         square(x)\n";
    cout << "_____________________ \n";
    do {
         z = funct (x,y);
         cout << x << '\t' << z << '\n';
         ++x;++y;
    }
    while (x <= 10);
    return 0;
}

int funct (register int x, register int y)
{
     register int temp;
     temp = x*y;
     return (temp);
}
```

Output of the above program

```
x       square(x)
0           0
1           1
2           4
3           9
4           16
5           25
6           36
7           49
8           64
9           81
10          100
```

There are some restrictions on register variables. As a machine register is usually a single word, many compilers allow only those variables that fit into a word to be placed in registers. This means that int, char, or pointer variables can only be placed in registers. In addition, most compilers have only a few registers available to user programs, usually two or three.

Suppose, some of the variables have been declared as register variables and if these variables are not the correct data type such as `char` or `int` and if there are not enough registers available, then the C++ compiler will automatically ignore the register data type and it keeps them in the memory, treating them as automatic variables.

6.15.3 Static Variable

The static variables are defined within a function and they have the same scope rules of the automatic variables but in the case of static variables, the contents of the variables will be retained throughout the program.

The keyword `static` may be used to declare variables both within and outside a function except for function arguments. When the static variables are declared within a function, the static causes variables to have fixed duration instead of the default automatic duration. For static variables those are declared outside a function, the static keyword gives the variable file scope instead of program scope. The keyword `static` should not be used as function arguments.

The general syntax of the static variable is,

```
static datatype variable 1,variable2,....variable n;
```

where `static` is a keyword used to define the storage class as a static variable.

The following declarations are valid

```
static int x,y;
static int x = 100;
static char a1,ch;
```

PROGRAM 6.19

A program to display 1 to 10 with addition of 100 using the automatic variable.

```
// using automatic variable
#include <iostream>
using namespace std;
int main()
{
    int funct(int x); // function declaration
    int i,value;
    for ( i = 1; i <= 10; i++) {
        value = funct (i);
        cout << i << '\t' << value << '\n';
    }
    return 0;
}

int funct (int x)
{
    int sum = 100;  // automatic variable
    sum += x;
    return (sum);
}
```

Output of the above program

```
1          101
2          102
3          103
4          104
5          105
6          106
7          107
8          108
9          109
10         110
```

The above program can be modified using sum as a `static` variable:

PROGRAM 6.20

A program to display `1` to `10` with addition of `100` using the static variable.

```
// using static storage modifier
#include <iostream>
using namespace std;
int main()
{
    int funct( int x); // function declaration
    int i,value;
    for ( i = 1; i <= 10; i++) {
        value = funct (i);
```

```
        cout << i << '\t' << value << '\n';
    }
    return 0;
}

int funct ( int x)
{
    static int sum = 100; // static variable
    sum += x;
    return (sum);
}
```

Output of the above program

```
1       101
2       103
3       106
4       110
5       115
6       121
7       128
8       136
9       145
10      155
```

Since `sum` has a permanent memory space it retains the same value in the period of time between leaving function and again entering it later. In contrast to automatic variable, static variables are initialised only once. Hence in the above program, `sum` has the value 100 only.

6.15.4 External Variable

Variables which are declared outside the main are called external variables and these variables will have the same data type throughout the program, both in main and within the functions. The keyword `extern` specifier may be used for declaring both within and outside a function except for function arguments.

Normally, variables are declared only at the beginning of blocks. But one can also declare variables outside any function, anywhere in the file. Functions can access these variables, which resemble the global variables of Pascal and the common variables of FORTRAN, simply by referring to them by their name. Global variables have the same lifetime and initlialisation rules as the static variables. This implicit initialisation is convenient and is taken advantage of by countless programs, but explicitly initialising global variables makes programs more readable.

The general syntax of the external variable is,

```
extern datatype variable 1,varibale2,....variable n;
```

where `extern` is a keyword used to define the storage class as external variables.

The following declarations are valid

```
extern int x,y;
extern float p,q,r;
extern char a1,ch;
```

PROGRAM 6.21

A program to define a variable as an external data type and to display the contents of the variable.

```
// using extern storage modifier
#include <iostream>
using namespace std;
```

```
extern int x = 10; // extern variable should be initialised
int main()
{
    void funct (void);
    cout << "content of x (in main) = " << x << '\n';
    funct();
    cout << "content of x (in main) = " << x << '\n';
    return 0;
}

void funct (void)
{
    cout << "content of x (inside a function) = " << ++x;
    cout << '\n';
}
```

Output of the above program

```
content of x (in main) = 10
content of x (inside a function) = 11
content of x (in main) = 11
```

In the above program, x has been declared as the global variable, in other words, as any external variable. One has to declare the external variables only once but these external variables can be used in both main and functions without declaring the data type again.

For example, the following program gives error message while compiling the program due to non-initialisation of extern variables. By default, the variables that are declared as extern, should be initialised by the user as it cannot be initialised by the compiler automatically.

Since the `extern int x` is not initialised with data, the compiler gives error message during the compilation time.

```
// using extern storage modifier
#include <iostream>
using namespace std;
extern int x; // extern variable should be initialised
int main()
{
      void funct (void);
      cout << "content of x (in main) = " << x;
      funct();
      return 0;
}

void funct (void)
{
      cout << "content of x (inside a function) = " << x;
}
```

6.15.5 The `const` Modifier

The `const` modifier is used along with a variable declaration exclusively for read only purpose. User cannot change the value of the variable once the variables are declared as `const` type.

PROGRAM 6.22

A program to define a variable as a const data type and display the contents of the variable.

```
// using const storage modifier
#include <iostream>
using namespace std;
const int x = 10;
int main()
{
    void funct (void);
    cout << "content of x (in main) = " << x << '\n';
    funct();
    return 0;
}

void funct (void)
{
     cout << "content of x (inside a function) = " << x;
     cout << '\n';
}
```

Output of the above program

```
content of x (in main) = 10
content of x (inside a function) = 10
```

For example, the following program gives error message while compiling the program due to change of the `const` value. By default, the `const` variable is meant for read only purpose. Hence, it gives error message.

```
// error
#include <iostream>
using namespace std;
const int x = 20;
int main()
{
   x++;    //altering the cost variable
   cout << "x = " << x;
   return 0;
}
```

PROGRAM 6.23

A program to define a variable as a const data type and display the contents of the variable without initialisation.

```
//const storage modifier, error due to uninitialisation
#include <iostream>
using namespace std;
const int x;
int main()
{
     void funct (void);
     cout << "content of x (in main) = " << x <<'\n';
     funct();
     return 0;
}
void funct (void)
{
     cout << "content of x (inside a function) = " << x;
     cout << '\n';
}
```

Compile time error The variables that are declared as `const`, should be initialised. Since `const int x` is uninitialised, compiler gives the error message.

6.15.6 The Volatile Modifier

The `volatile` modifier is used along with a variable for giving directions to the compiler to turn off certain optimisations. This is especially useful for device registers and other data segments while operating on the floating point operations and the system level calls.

PROGRAM 6.24

A program to define a variable as a volatile data type and display the contents of the variable.

```
// using volatile storage modifier
#include <iostream>
using namespace std;
volatile int x = 10;
int main()
{
    void funct (void);
    cout << "content of x (in main) = " << x << '\n';
    funct();
    cout << "content of x (after function call) = " << x;
    cout << '\n';
    return 0;
}
void funct (void)
{
    x++;
    cout << "content of x (inside a function) = " << x;
    cout << '\n';
}
```

Output of the above program

```
content of x (in main) = 10
content of x (inside a function) = 11
content of x (after function call) = 11
```

6.16 RECURSIVE FUNCTIONS

This section explains how to declare, define and invoke a recursive subprogram. A subprogram which calls itself directly or indirectly again and again, is known as a *recursive subprogram*.

It is well known that a function is a subprogram and when a function is invoked or called by itself again and again is called as a *recursive function*. When a function is invoked or called by itself in a part of program, it is known as *recursive function call*.

C++ permits the declaration and invocation of a function recursively. The way in which a recursive function is declared, defined and called are all similar to a normal function. Whenever a function is called either as a normal or as a recursive call, the value of the function must be assigned to the left hand side of the variable.

The main advantages of using recursive subprograms are:

(1) Recursive functions are very useful while constructing data structures, like, linked lists, double linked lists and binary trees.

(2) Recursive call is faster.

(3) There is minimum transfer of control.
(4) Recursive functions normally take less memory space than normal or conventional functions.
(5) The code size becomes small, if it is designed using recursive methods.

But developing, testing and maintaining recursive codes are always very difficult than usual techniques.

There is much difference between normal and recursive functions. Normal function will be called by the part of a program, whenever, the function name is used. On the other hand recursive function will be called by itself, directly or indirectly, as long as the given condition is satisfied.

For example, the following program segment shows how to declare a recursive function in C++.

```
#include <iostream>
void main()
{
     void funct1(); /* function declaration */
     -------
     -------
     funct1(); /*  function calling  */
}
void funct1() /*  function definition */
{
     -------
     -------
     funct1(); /* function calls recursively */
}
```

The following program shows how to declare the `main()` function for a recursive call in C++.

```
#include <iostream>
using namespace std;
int main()
{
    cout << "this is a test program \n";
    main();
    return 0;
}
```

Output of the above program

The message 'this is a test program' will be displayed for ever until the user interrupts the execution by pressing the Cntrl+C keys from the command window.

PROGRAM 6.25

A program to find the sum of given non-negative integer numbers using a recursive function.

```
   sum = 1+2+3+4 ...n
// sum = 1+2+3+4 ....n using recursive function
#include <iostream>
using namespace std;
int main()
{
    int sum (int);
    int n,temp;
    cout << "Enter any integer number \n";
    cin >> n;
    temp = sum(n);
    cout << "1+2+3...  " << n;
    cout <<" and its sum = " << temp << '\n';
    return 0;
```

```
}

int sum (int n)  // recursive function
{
    int sum (int); // local function declaration
    int value = 0;
    if ( n == 0)
       return (value);
    else
       value = n+sum(n-1);
    return (value);
}
```

Output of the above program

```
Enter any integer number
5
1+2+3...  5 and its sum = 15
```

The following illustrations will be helpful to understand the recursive function call.

For value 1

```
= 1 + sum (1-1)
= 1 + 0
= 1
```

For value 2

```
= 2 + sum (2-1)
= 2 + 1+sum (1-1)
= 3
```

For value 3

```
= 3 + sum (3-1)
= 3 + 2 + sum (2-1)
= 3 + 2 + 1+ sum (1-1)
= 6
```

PROGRAM 6.26

A program to find the factorial of the given number using recursive function.

The factorial of n (written n!) is the product of all integers between 1 to n. (assume n is non-negative).

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1) & n > 0 \end{cases}$$

```
// factorial of a given number using recursive function
#include <iostream>
using namespace std;
int main ()
{
    long int fact (long int);
    long int x,n;
    cout << "Enter any integer number \n";
    cin >> n;
    x = fact (n);
    cout << "value = " << n << " and its factorial = " << x;
    cout << '\n';
    return 0;
}

long int fact (long int n) // recursive function
{
```

```
    long int fact( long int); // local function declaration
    int value = 1;
    if ( n == 1 )
       return (value);
    else
    {
       value = n * fact (n-1);
       return (value);
   }
}
```

Output of the above program

```
Enter any integer number
5
value = 5 and its factorial = 120
```

The following steps illustrate how a factorial of a given number is calculated using recursive call.

For value 1

```
= 1*fact (1-1)
= 1
```

For value 2

```
= 2 *fact (2-1)
= 2*1
= 2
```

For value 3

```
= 3*fact (3-1)
= 3*2*fact (2-1)
= 3*2*1
= 6
```

6.17 PREPROCESSORS

Preprocessor is a program that modifies the C++ source program according to directives supplied in the program. The original source program is usually stored in a file. The preprocessor does not modify this program file, but creates a new file that contains the processed version of the program. This new file is then submitted to the compiler. The preprocessor makes the program easy to understand and port it from one platform to another. A preprocessor carries out the following actions on the source file before it is presented to the compiler. These actions consist of

- replacement of defined identifiers by pieces of the text,
- conditional selection of parts of the source file,
- inclusion of other files, and
- renumbering of source files and the renaming of the source files itself.

The general rules for defining a preprocessor are,

(a) All preprocessor directives begin with the sharp sign (#).
(b) They must start in the first column and on most C++ compiler there can be no space between the number sign and the directive.
(c) The preprocessor directive is terminated not by a semicolon.
(d) Only one preprocessor directive can occur in a line.
(e) The preprocessor directives may appear at any place in any source file: outside/inside functions or inside compound statements.

The C++ preprocessor is a simple macroprocessor that conceptually processes the source text of a C++ program before the compiler parses the source program. The preprocessor is controlled by special

preprocessor command lines, which are lines of the source file beginning with the character '#'. Note that character '#' has no other use in the C++ language.

The Common C++ preprocessor directives and their uses are:

Directive	Uses
`#include`	insert text from another file
`#define`	define preprocessor macro
`#undef`	remove macro definitions
`#if`	conditionally include some text based on the value of the constant expression
`#ifdef`	conditionally include some text based on predefined macro name
`#ifndef`	conditionally include some text with the sense of the test opposite to that of #ifdef
`#else`	alternatively include some text, if the previous # if, #ifdef, or #ifndef tests failed
`#elif`	combination of #if and #else
`#endif`	terminate conditional text
`#line`	give a line number for compiler messages
`#error`	terminate processing early

6.17.1 Simple Macro Definitions

A macro is simply a substitution string that is placed in a program.

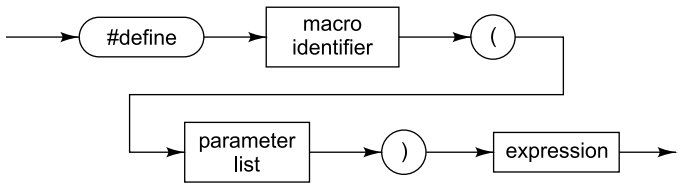


Fig. 6.8 *Syntax Diagram of Macro Declaration*

For example, consider the following program segment,

```
#define MAX 100
void main()
{
   char name[MAX];
   for ( i=0; i<= MAX-1; ++i)
}
```

which is internally replaced with the following program

```
void main()
{
    char name[100];
    for (i=0; i<= 100-1; ++i)
}
```

and subsequently compiled. Each occurrence of the identifier MAX as a token is replaced with the string 100 that follows the identifier in the #define line. The simple form of macro is particularly useful for introducing named constants into a program. This makes it easier to change the number later.

For example, the following are valid #define statements.

```
#define TRUE  1
#define FALSE 0
```

```
#define EOF -1
#define SIZE 5
#define TRACK_SIZE  6
#define MAX_BLOCK  100
```

The syntax of the #define command does not require an equal sign or any other special delimiter token after the name being defined.

Some invalid #define statements are illustrated below.

(1)

```
#define SIZE  =3
```

The #define statement is used to merely substitute the string constants into the source program. For example,

```
#define SIZE  =3
#include <iostream>
void main()
{
     int value [SIZE];
     for (i=0; i<= SIZE-1; ++i)
     -------
     -------
}
```

The macro substitutes every occurrence of the SIZE with the =3, modifying the above source program as,

```
#include <iostream>
void main()
{
     int value [=3];    /*  syntax error  */
     for (i=0; i<= =3-1; ++i)      /*  syntax error  */
     -------
     -------
}
```

(2) The #define statement does not take the semicolon.

```
#define  MAX  100;  /*  error  */
```

For example,

```
#define MAX  100;
#include <iostream>
void main()
{
     char name [MAX];
     for (i=0; i<= MAX-1; ++i)
     -------
     -------
}
```

The macro substitutes every occurrences of the MAX with the 100; modifying the above program segment, as

```
#include <iostream>
void main()
{
     char name [100;];   /*  syntax error  */
     for (i=0; i<= 100;-1; ++i)  /* syntax error  */
     -------
```

```
        -------
    }
```

6.17.2 Macro with Parameters

A more complex form of a macro definition declares the names of formal parameters within parentheses, separated by commas.

```
#define name (variable 1,variable2,...variable n) subtitution_string
```

For example,

```
#define PRODUCT (x,y)    ((x) *(y))
#define max (x,y)     ((x) > (y) ? (x) : (y))
#define min(x,y)     ((x) < (y) ? (x) : (y))
```

Macros operate purely by textual substitution of tokens. The C++ compiler parses the source program only after the completion of the macro expansion processes are completed. Hence, care must be taken to get the desired results. For example,

```
#define PRODUCT (x)  x*x
```

If we call this macro definition in the C++ program,

```
#define PRODUCT (x)  x*x
void main()
{
    PRODUCT (10);
    -------
    -------
}
```

`PRODUCT (10)` expands to `10 *10`. However, the expression

`PRODUCT (a+1)` expands to

```
a+1*a+1
```

When the above expression is executed, it is interpreted as `a+ (1*a) +1` because the multiplication has higher precedence over the addition. This will not produce the same result as `(a+1) *(a+1)` unless `a` happens to be zero. It would be somewhat safer to put parentheses around occurrences of the formal parameters in the definition of `PRODUCT` as,

```
#define PRODUCT (x)  (x) * (x)
```

Even this definition does not provide complete protection against precedence problems.

For example,

```
#define PRODUCT (x)  (x)* (x)
#include <iostream>
void main()
{
    (short) PRODUCT ( a+1)
    -------
    -------
}
```

The preprocessor substitutes the macro as

```
(short) (a+1) * (a+1)
```

then, it would be parsed as

```
((short) (a+1)) * (a+1)
```

as a cast has higher precedence over multiplication. Hence proper parentheses should be used within the macro definition statement to get the desired results. The correct way of defining the `PRODUCT` macro is

```
#define PRODUCT (x)  (( x) * (y))
```

6.17.3 Other Preprocessing Techniques

As the preprocessor merely substitutes a string of text for another without any checking, a wide variety of substitutions are possible and hence, it is possible to make any typed source program look like another language. To illustrate the above, if someone has a strong liking for Assembly level programming and its syntax, then Assembler symbols can be used in C++ by just including as many #define statements so as to convert them to valid C++ symbols before the compilation.

PROGRAM 6.27

A program for microprocessor simulator using the macro definition.

```
#include <stdio.h>
#define SWAP(x,y,type) {type  temp;temp=(x);(x)=(y);(y)=temp;}

#define MOV(d,s)       (d)=(s);
#define ADD(d,s)       (d)=(d)+(s);
#define MUL(d,s)       (d) = (d)*(s);      /* s - source */
#define DIV(d,s)       (d) = (d)/(s);     /* d - destination */
#define XCHG(d,s)      SWAP(d,s,SWAPTYPE);
#define OUT(value)     printf("%d",(value));
#define OUTB(value)    printf("%c",(value));
#define OUTS(label)    printf("%s",(label));
#define LF             printf ("\n");
#define ASCII(label,string)  char *label ={ string };
#define WORD(label)    int label;
#define BYTE(label)    char label;

#define START          main() {
#define END            }

#define SWAPTYPE    int

ASCII(msg,"Microprocessor simulator ")

START
   WORD(result)
   WORD(r1)
   WORD(r2)
   MOV ( result ,0)
   MOV ( r1 ,20)
   MOV ( r2,20)
   MOV ( result ,r1)
   ADD ( result ,r2)
   LF
   OUTS( " Sum of = ")
   OUT ( result)
   LF
   MOV ( result ,r1)
   MUL ( result ,r2)
   OUTS ( "Multiplication of =");
   OUT (result )
   LF
   MOV ( result ,r1)
   DIV ( result ,r2)
   OUTS ( "Division of =")
   OUT  ( result)
END
```

Output of the above program

```
Sum of = 40
```

```
Multiplication of = 400
Division of = 1
```

6.17.4 Conditional Compilation

The preprocessor conditional compilation commands allow lines of the source text to be passed through or eliminated by the preprocessor on the basis of a computed condition.

The following are the preprocessor conditional commands.

```
#if
#else
#endif
#elif
```

The above commands are used in the following way:

```
#if  constant expression
    group of lines 1
#else
    group of lines 2
#endif
```

A group of lines may contain any number of lines of the text or any kind, even other preprocessor command lines, or no lines at all. The #else command may be omitted.

The `#elif` command The `#elif` command is fairly a recent addition to C++. It is supported by very few compilers only. The `#elif` command is like a combination of `#if` and `#else`. It is used between `#if` and `#endif` in the same way as `#else` but has a constant expression to evaluate in the same way as `#if`.

It is used in the following format:

```
#if constant expression
   group of lines 1
#elif constant expression
   group of lines 2
#elif  constant expression
   group of lines 3
-------
-------
#else
     group of lines n
#endif
```

The same commands can be written using `#if, #else` and `#endif`.

For example,

```
#if expression
    -------
    -------
#elif expression
    -------
    -------
#elif expression
    -------
    -------
#else
    -------
    -------
#endif
```

In case a particular C++ compiler is not supporting the `#elif` conditional command, then the above commands may be modified as:

```
#if expression
    -------
    -------
#else
#if expression
    -------
    -------
#else
#if  expression
    -------
    -------
#else
    -------
    -------
#endif
#endif
#endif
```

6.18 HEADER FILES

A header file contains the definition, global variable declarations, and initialisation by all the file in a program. Header files are not compiled separately. The header file can be included in the program using the macro defintion #include command. A header file can be declared as a first line in any C++ program. For example, the standard input/output stream <iostream> header file contains the macro definitions and functions needed by the program input/output statements.

The header file can be declared in one of the following ways:

```
#include <myprogram.h>
```

or

```
#include "myprogram.h"
```

6.19 STANDARD FUNCTIONS

Standard libraries are used to perform some predefined operations on characters, strings, etc. The standard libraries are invoked using different names such as library functions, built-in functions or predefined functions. As the term library function indicates, there are a great many of them, actually they are not part of the language. Many facilities that are used in C++ programs need not be part of the C++ language. Most of the C++ compilers support the following standard library facilities.

- operations on characters <cctype>
- operations on strings <cstring>
- mathematical operations <cmath>
- storage allocation procedures
- input / output operations <cstdio>

REVIEW QUESTIONS

1. What is a function? List out the advantages and disadvantages of using functions in C++.
2. How a function is declared in C++?

3. What is meant by call by reference and call by value?
4. What is the purpose of the return statement?
5. What is meant by the function arguments, function call and return values?
6. List out the rules normally governing the use of the return statement.
7. In C++, can a function be called from more than one place within a program?
8. How is a function declaration different from the function definition?
9. What is meant by the scope of variables? Summarise the variable types of storage class in C++?
10. What is an automatic variable and what is the use of it?
11. Explain how data can be initialised in the automatic variable.
12. What is meant by the register variable and what is the scope of it?
13. How is a register variable different from an automatic variable?
14. What is a static variable and what is its scope?
15. How are data elements initialised in the case of static type variable?
16. How is a static variable different from an automatic variable?
17. What is the use of the external data type in C++?
18. What is a recursive function? List out the merits and demerits of the function.
19. How is a recursive function different from an ordinary function?
20. What is the storage class used in a recursive function?
21. What is a preprocessor in C++?
22. What is a macro and how is it different from a preprocessor?
23. What is a header file in C++? What is the purpose of using these files in C++?
24. What is meant by command line arguments?
25. What is a standard function and how is it useful for a developing program?
26. List out the C++ preprocessor directives and their uses.
27. What is meant by macro operators?
28. Distinguish between a `#include` and `#define`.
29. What is meant by conditional compilation?
30. List out the various operators of `<cmath>` library function.
31. Summarise the purpose of `<cstring>` function.
32. What are the different functions involved in `<cctype>`?

CONCEPT REVIEW PROBLEMS

1. Determine the output of each of the following programs when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    void display();
    int x = 1;
    cout << x;
    display();
    cout <<'\t'<< x;
    return 0;
}
```

```
void display()
{
     int x = 5;
     cout <<'\t' << x;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
     void display1(void);
     void display2(void);
     int x = 1;
     cout << x;
     display1();
     cout << '\t' << x;
     display2();
     cout << '\t' << x;
     return 0;
}
void display1()
{
     int x = 5;
     cout <<'\t' << x;
}
void display2()
{
     int x = 10;
     cout <<'\t' << x;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    void display1(void);
    int x = 1;
    cout << x;
    display1();
    cout <<'\t'<< x;
    return 0;
}
void display1()
{
     void display2(void);
     int x = 5;
     cout <<'\t' << x;
     display2();
     cout <<'\t' << x;
}
void display2()
{
     int x = 10;
     cout <<'\t' << x;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    int display1(int x);
    int x = 1;
    cout << x;
    x = display1(x);
    cout <<'\t' << x;
    return 0;
}
int display1(int x)
{
    int  y = 5;
    x += y;
    cout <<'\t' << x;
    return (x);
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    int display1(int x);
    int x = 1;
    cout << x;
    x = display1(x);
    cout <<'\t' << x;
    return 0;
}
int display1(int x)
{
    void display2(int x);
    int  y = 5;
    x += y;
    cout <<'\t' << x;
    display2(x);
    return (x);
}
void display2(int x)
{
     int  y = 10;
     x += y;
     cout <<'\t' << x;
}
```

(f)

```
#include <iostream>
using namespace std;
int x = 10;
int main()
{
    void display1();
    cout << x;
```

```
        display1();
        cout <<'\t' << x;
        return 0;
}
void display1()
{
        int  y = 1;
        x += y;
        cout <<'\t' << x;
}
```

(g)

```
#include <iostream>
using namespace std;
static int x = 10;
int main()
{
    void display1();
    cout << x;
    display1();
    cout <<'\t' << x;
    return 0;
}
void display1()
{
    int  y = 1;
    x += y;
    cout <<'\t' << x;
}
```

(h)

```
#include <iostream>
using namespace std;
extern int x = 10;
int main()
{
    int display1();
    cout << x;
    x = display1();
    cout <<'\t'<< x;
    return 0;
}
int display1()
{
    static int  x = 1;
    x++ ;
    cout <<'\t'<< x;
    return (x);
}
```

(i)

```
#include <iostream>
using namespace std;
extern int x = 10;
int main()
{
    void display1();
    cout << x;
```

```
        display1();
        cout <<'\t'<< x;
        return 0;
}
void display1()
{
        static int  x = 1;
        x++;
        cout <<'\t'<< x;
}
```

2. What will be the output of each of the following programs when it is executed?

(a)

```
#include <iostream>
using namespace std;
int main()
{
    int sum (int i,int j);
    int i = 10,j = 20,total;
    total = sum (i,j);
    cout << " total = " << total;
    return 0;
}
int sum (int a, int b)
{
    cout << "i = " << a << '\t';
    cout << "j = " << b << '\t';
    return (a+b,a-b,a*b);
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    int sum (int i,int j);
    int i = 10,j = 20,total;
    total = sum (i,j);
    cout << " total = " << total;
    return 0;
}
int sum (int a, int b)
{
    cout << "i = " << a << '\t';
    cout << "j = " << b << '\t';
    return;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    int display();
    int total;
    total = display();
    cout <<"total = " << total;
    return 0;
```

```
}
int display()
{
    int i = 10, j = 20, k = 30, m = 40;
    return (i,j,k,m);
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    int display (int i);
    int sum,i = 10;
    sum = display (i);
    cout <<"sum = " << sum;
    return 0;
}

int display (int a)
{
    return (a++);
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    int display_main (int i);
    int sum,i = 10;
    sum = display_main (i);
    cout <<"sum = "<< sum;
    return 0;
}

int display_main (int i)
{
    int display_one (int i);
    int a;
    a = display_one(i);
    return (a++);
}

int display_one (int i)
{
    int display_two (int i);
    int a;
    a = display_two(i);
    return (a++);
}

int display_two (int i)
{
    return (i++);
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    int display_main (int i);
    int sum,i = 10;
    sum = display_main (i);
    cout <<"sum = "<< sum;
    return 0;
}
int display_main (int i)
{
    int display_one (int i);
    int a;
    a = display_one(i);
    return (++a);
}

int display_one (int i)
{
    int display_two (int i);
    int a;
    a = display_two(i);
    return (++a);
}

int display_two (int i)
{
    return (++i);
}
```

(g)

```
// default function call
#include <iostream>
using namespace std;
int main()
{
    int display();
    int sum;
    sum = display();
    cout <<"sum = "<< sum;
    return 0;
}
int display()
{
}
```

(h)

```
// default function call
#include <iostream>
using namespace std;
int main()
{
    int display1();
    int display2();
    int display3();
```

```
    int sum;
    sum = display1()+ display2() + display3();
    cout <<"sum = " << sum;
    return 0;
}
int display1()
{
}
int display2()
{
}
int display3()
{
}
```

3. Determine the output of each of the following programs when it is executed:

(a)

```
#include <iostream>
using namespace std;
int main()
{
    void display();
    display();
    cout << "In main ...\n";
    return 0;
}
void display()
{
     void display_inner();
     display_inner();
     cout << "Within a function 1 \n";
}
void display_inner()
{
     void display_innermost();
     display_innermost();
     cout <<"now inside the function 2 \n";
}
void display_innermost()
{
     cout << "Innermost ...\n";
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    int display(int a);
    int a = 10;
    int b = display(a);
    cout << " b = " << b;
    return 0;
}
int display(int i)
{
    int display_inner(int a);
```

```
    int j = display_inner(i);
    cout << "j = " << j;
    return (++j);
}
int display_inner(int i)
{
    return (++i);
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    int display(int a);
    int a = 10;
    display(a);
    cout << " a = " << a;
    return 0;
}
int display(int i)
{
    int display_inner(int a);
    display_inner(i);
    cout << "i = " << i;
    return (++i);
}
int display_inner(int i)
{
    return (++i);
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    int display(int a);
    int a = 10;
    a = display(a);
    cout << " a = " << a;
    return 0;
}
int display(int i)
{
    int display_inner(int a);
    i = display_inner(i);
    cout << " i = " << i;
    return (++i);
}
int display_inner(int i)
{
    return (++i);
}
```

(e)

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int display(int a);
    int a = 10;
    a = display(a);
    cout << " a = " << a;
    return 0;
}
int display(int i)
{
    int display_inner(int a);
    i = display_inner(i);
    cout << " i = " << i;
    return (i++);
}
int display_inner(int i)
{
    return (i++);
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    int display(int i);
    int sum,a;
    sum = display(a);
    cout << " sum = " << sum;
    return 0;
}
int display(int i)
{
    return (++i);
}
```

(g)

```
#include <iostream>
using namespace std;
static int a;
int main()
{
    int display(int i);
    int sum;
    sum = display(a);
    cout << " sum = " << sum;
    return 0;
}
int display(int i)
{
    return (++i);
}
```

(h)

```
#include <iostream>
using namespace std;
static int a;
```

```
int main()
{
    int display(int i);
    int sum;
    sum = display(a) + display(a) + display (a);
    cout << " sum = " << sum;
    return 0;
}
int display(int i)
{
    return (i++);
}
```

(i)

```
#include <iostream>
using namespace std;
static int a;
int main()
{
    int display(int i);
    int sum;
    sum = display(a) + display(a) + display (a);
    cout << " sum = " << sum;
    return 0;
}
int display(int i)
{
    return (++i);
}
```

(j)

```
#include <iostream>
using namespace std;
int main()
{
    const volatile int display (const volatile int a);
    const volatile int a = 10;
    cout << " value of a (in main) = " << a << '\n';
    display (a);
    cout << " value of a (after function call) = " << a;
    cout << '\n';
    return 0;
}

const volatile int display (const volatile int a)
{
      cout << " value of a (inside function) = " << a;
      cout << '\n';
}
```

(k)

```
#include <iostream>
using namespace std;
int main()
{
    const static int a = 10;
    cout << "a = " << a << '\n';
```

```
    return 0;
}
```

PROGRAMMING EXERCISES

1. Write a function in C++ to find the sum of the following series:

(a) sum = 1 + 2 + 3 + ··· + n

(b) sum = 1 + 3 + 5 + ··· + n

(c) sum = 1 + 2 + 4 + ··· + n

(d) $sum = 1 + \frac{2}{2!} - \frac{3}{3!} + \cdots \frac{x}{n!}$

(e) $sum = x + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots \frac{x^n}{n!}$

(f) $sum = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \frac{x^n}{n!}$

(g) $sum = 1^2 + 2^2 + 3^2 + 4^2 + \cdots + n^2$

(h) $sum = 1^3 + 2^3 + 3^3 + 4^3 + \cdots + n^3$

(i) $sum = 1 + 2^2 + 4^2 + \cdots + n^2$

(j) $sum = 1 + 3^2 + 5^2 + \cdots + n^2$

2. Write a function in C++ to generate a Fibonacci series of 'n' numbers, where n is defined by a programmer.
(The series should be: 1 1 2 3 5 8 13 21 32 and so on.)

3. Write a function in C++ to generate the following series of numbers:

(a) number = 1 2 3 4 ··· n

(b) number = 0 2 4 6 ··· n

(c) number = 1 3 5 7 ··· n

(d) number = 1 2^2 3^2 4^2 ··· n^2

(e) number = 1 2^3 3^3 4^3 ··· n^3

4. Write a function in C++ to generate the following pyramid of numbers:

```
                0
             1  0  1
          2  1  0  1  2
       3  2  1  0  1  2  3
    4  3  2  1  0  1  2  3  4
 5  4  3  2  1  0  1  2  3  4  5
6  5  4  3  2  1  0  1  2  3  4  5  6
```

5. Develop a program in C++ to find the largest of any three numbers using a macro definition.

6. Write a macro in C++ to find the odd and even numbers from a given set of numbers.

7. Write a macro in C++ to find the cube of any three numbers.

8. Write a macro to find the power of given numbers.

9. Write a macro in C++ to swap two data items.

10. Write a macro in C++ to find the factorial of a given number.

11. Write a macro in C++ to find the sum of the following series:

(i) sum = 1-3+5-7···n

(ii) sum = 2*2 +4*4 +6*6+ ··· n*n

(iii) $sum = 1 + \frac{1}{2!} + \frac{1}{3!} + \cdots \frac{1}{n!}$

Chapter

Arrays

7

This chapter explains how to declare and realise single dimensional and multidimensional array data types and how to develop a modular program using subprogram concepts with the help of call by value and call by reference methods of the array data type. This chapter also elucidates how to realise the non-numeric data which is one of the most attractive features in C++. The various operations of the non-numeric data, such as string length, string compare, string copy, string concatenate, etc. are also discussed.

7.1 INTRODUCTION

One of the most attractive features of C++ is that it supports various user-defined data types to cater the requirements of a variety of applications in business, scientific and engineering disciplines. It has been explained in previous chapters that C++ data types, in general, can be classified into two types: simple and structured. A simple variable contains the built-in or standard data types, such as integer, floating point number, and character. One of the common features of a simple data type is that each variable represents only a single data item. For example, an integer variable which is declared can hold only a single integer quantity. In other words, a single integer variable cannot have the memory space to accommodate more than one data element.

When there is a necessity to process more than a two data elements, it is advisable to use common variable names. To define and realise common variables in C++, structured data types are used. The main characteristics of a structured data type is to define a variable which holds more than one data item. For example, to process the grades of a student in a class, it is not a good practice to declare an individual variable for each subject for all students. It is quite impossible and also the variables are redundant in nature. To avoid the unnecessary repetition of same variables, array data type is used. Array is a structured data type and it has many advantages over the conventional data types.

7.2 ARRAY NOTATION

An array is a collection of identical data objects which are stored in consecutive memory locations under a common heading or a variable name. In other words, an array is a group or a table of values referred to by the same variable name. The individual values in an array are called elements. Array elements are also variables.

Arrays are sets of values of the same type, which have a single name followed by an index. In C++, square brackets appear around the index right after the name, with the first element referred to by the number. Whenever an array name with an index appears in an expression, the C++ compiler assumes that element to be of an array type.

7.3 ARRAY DECLARATION

Arrays, records, sets and files are structured data types which are used to store and process more than a single entity. It is a group or table of values in which all items are of homogeneous data type. For example, if an array is declared as an integer type, then the particular array cannot read, write or store any other data type other than an integer. If attempts are made to process with non-integer data types, a significant error message will be displayed.

Before one attempts to use an array variable in a program, it must be declared well in advance with sufficient information to the compiler at the time of compiling. The compiler expects information from the user, such as name of the array, type of elements stored in that array and the maximum number of elements likely to be stored in it.

Array declaration is a process of declaring the name and type of an array and setting the number of elements in the array. In C++, there is no separate statement called DIMENSION in which the array size is defined as in the case of some programming languages like FORTRAN. Sometimes, dimensioning of an array is also called as array declaration.

An array is a static allocation of memory space that is required for the variables of a program. When the array size is large declared requiring large memory space and at the time of execution of a program, if only a small portion of the declared memory is used, then the rest of the heap memory space is getting wasted. In order to use the heap memory effectively and efficiently, it is required to handle array declaration with due care. It is not a good programming practice to grab the enitre system memory for one's requirement alone.

Before any linear or multidimensional array is used in a program one must provide to the compiler or to the interpreter the following information:

(1) Type of array (i.e. integer, floating point number, char type, etc.)
(2) Name of the array
(3) Number of subscripts in the array (i.e. whether the array is one or multi-dimensional etc.)
(4) Total number of memory locations required to be allocated or more specifically the maximum value of each subscript)

In general, one-dimensional array may be expressed as

```
storage_class data_type array_name [expression]
```

where the storage_class refers to the scope of the array variable such as external, static, or an automatic; and `data_type` is used to declare the nature of the data elements stored in the array, like character type, integer and floating point. `Array_name` is the name of the array, and an expression is used to declare the size of the memory locations required for further processing by the program.

The storage class is optional. Default values are automatic for arrays defined within a function or a block and extended for arrays defined outside the function.The syntax diagram of array declaration is given in Fig. 7.1.

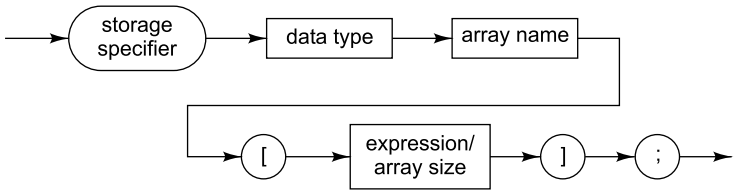


Fig. 7.1 *Syntax Diagram of Array Declaration*

Some valid one-dimensional array declarations are:

```
int marks [300];
char line [90];
static char page [8];
 oat coordinate [400];
```

In the above illustrations, the first one has been declared as a mark of 300 integer numbers; the second is a line of 90 characters; the third is a static array which consists of 8 characters and the fourth is a set of 400 floating point numbers stored in the array of coordinate.

Some invalid array declaration are:

```
int value [0];
static int values [0.002];
 oat numbers [-90];
char s[$];
```

For an expression, positive integer numbers should be placed for the memory allocations whenever an array in C++ is declared.

7.4 ARRAY INITIALISATION

Automatic arrays cannot be initialised, unlike automatic variables in the older versions of the C++ compiler. However, external and static arrays can be initialised if it is desired. The latest version of the ANSI C++ compiler supports all forms of array initialisation. The initial values must appear in the same order in which they will be assigned to the individual array elements, enclosed in braces and separated by commas.

The general format of the array initialisation is,

```
storage_class data_type array_name [expression] =
                        {element1, element2,.. element n};
```

where the `storage_class` is used to declare the scope of the arrays like static, automatic or external; `data_type` is the nature of data elements such as integer, floating, or character, etc.; the `array_name` is used to declare the name of the array; and the elements are placed one after the other within the braces and finally ends with the semicolon. The syntax diagram of array intialisation is given in Fig. 7.2.

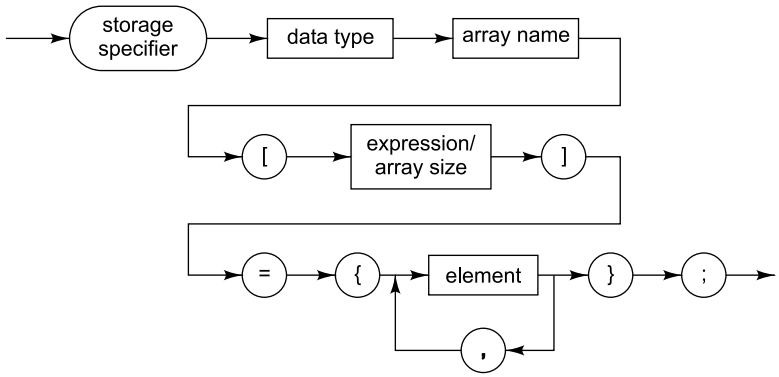


Fig. 7.2 *Syntax Diagram of Array Initialisation*

For example,

```
int values [7] = {10,11,12,13,14,15,16};
 oat coordinate[5] = {0,0.45,-0.50,-4.0,5.0};
char sex [2] = {'M','F'};
char name [5] = {'R','a','v','i' ,'c'};
```

The results of each of the above array element are:

```
values [0] = 10
values [1] = 11
values [2] = 12
values [3] = 13
values [4] = 14
values [5] = 15
values [6] = 16

coordinate [0] = 0
coordinate [1] = 0.45
coordinate [2] = -0.50
coordinate [3] = -4.0
coordinate [4] = 5.0

sex[0] = 'M'
sex[1] = 'F'

name [0] = 'R'
name [1] = 'a'
name [2] = 'v'
name [3] = 'i'
name [4] = 'c'
```

Note that in C++, the first element is always placed in the 0th place; it means that the array index starts from 0 to n–1, where n is the maximum size of the array declared by the programmer.

Some unusual way of initialising the array elements are discussed below. For example, if the array elements are not assigned explicitly, initial values will be automatically set to zero. Consider the following array declaration

```
int number [5] = {1,2,3};
```

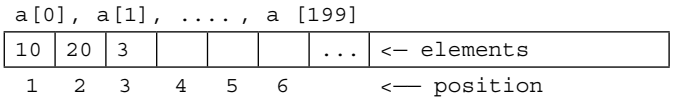
For the above, the elements will be assigned to the array in the following way:

```
number [0] = 1
number [1] = 2
numebr [2] = 3
number [3] = 0
number [4] = 0
```

7.5 PROCESSING WITH ARRAYS

In this section manipulating array elements individually and separately as a simple variables are discussed. Reading and writing array elements is one of the very important steps when array structures are used in a program. C++ compiler cannot read and write a whole array in a single command.

Consider the symbolic representation of an array as shown below, wherein the maximum size of the array is 200 number.



If one would like to avail more than the declared size, then, the compiler will treat only the first `n` elements as significant and the rest will be omitted, where `n` is the maximum size of the array. We can have access to these variables in any order we like and can use them in the same way as simple variables.

For example, we can write

```
a[15] = 3;
a[83] = 2;
cout << a[83]*a[15] + 1;
```

which will print `7`. Instead of contents of `83`,

Any integer expression can be used as a subscript in the array. For example,

```
int i,j;
for (i=0; i<=100; ++i) {
   a[i] = 0;
   -------
   -------
}
```

Some invalid array index

```
#include <iostream>
using namespace std;
int main()
{
    oat x;
    oat a[100];

   for ( x = 0; x <= 10.11; ++0.1) {  // error
      a[x] = 10;
      -------
      -------
   }
}
```

Note that the array index must be an integer data type.

In the array declaration the number of elements (here, 100) has to be specified as a constant. However this constant need not necessarily be written as a sequence of digits. It is generally considered good

programming practice to use named constants for this purpose. In C++, named constants are defined by a #de ne preprocessor control line. For example,

```
const int MAX_SIZE = 100;
int main()
{   int i;
    int a[MAX_SIZE);
    for (i=0; i<= MAX_SIZE-1; ++i)
    a[i] = 1;
    -------
    -------
}
```

In C++, simple operations involving entire arrays are not permitted. So every character must be treated as a separate variable for processes like assignment operations, comparison operations, and so on. For example, though a and b are two arrays having the same storage class, data type, and maximum size, assignment and comparison operations should be carried out only on element by element basis.

```
int a[4] = { 4,5,6,7};
int b[4] = { 1,2,3,4 };
```

The following operations are invalid:

(1)

```
if (a == b)
    cout <<" Array elements are different \n";
```

(2)

```
while (a > b )  // a and b are array types
{
    cout << "array processing \n";
    -------
    -------
}
```

7.5.1 Writing an Array Data Type

There is no single statement or command available to display an entire array of elements. For example, the use of the cout() stream function in the following program segment is invalid:

```
const int MAX = 20;
int main()
{
    int a[MAX],b[MAX],c[MAX];
    cout << a;  // error
    cout << b;  // error
    cout << c;  // error
    -------
    -------
    return 0;
}
```

Note that in the above program segment, a single cout() function is used to display the contents of an entire array of matrix, whose size is 20 elements. C++ does not support reading or writing an array by a single statement and hence a significant error message will be displayed.

To write the contents of an array, the subscripts must be enclosed along with the variables of the array type in the cout() function. For example, one can use the cout() function in the following way to display the contents of an array.

```
const int MAX = 20;
int main()
{
    int a[MAX];
    cout << a[0];
    cout << a[1];
    cout << a[2];
    cout << a[3];
    -------
    -------
    cout << a[19];
}
```

In the above program, the `cout()` function is repeated 20 times to display the elements of the array. But one can avoid using unnecessary repetition of the `cout()` function by using the loop statement for displaying the contents of an array.

For example, in the above program segment, one can use the loop statement to display the elements of the array.

```
const int MAX = 20;
int main()
{
    int a[MAX];
    int i;
    -------
    -------
    for ( i = 0; i <= MAX-1; ++i)
        cout << a[i];
}
```

Formatting array elements C++ compiler is so flexible that a user can easily format array data elements. It is up to the user to introduce the space between the data of the array elements as the compiler does not give any space between the elements while displaying the array data on the screen.

PROGRAM 7.1

A program to initialise a set of numbers in an array and to display them onto a standard output device.

```
#include <iostream>
using namespace std;
int main()
{
    int a[10] = { 0,1,2,3,4,5,6,7,8,9 };
    int i;
    cout <<"Contents of the array " << endl;
    for ( i = 0; i <= 9; ++i)
        cout << a[i];
    return 0;
}
```

Output of the above program

```
Contents of the array
0123456789
```

Note that space will not be inserted between the data items automatically by the compiler. It is up to a programmer to introduce proper spaces or tab set between data items.

The following programs show how the output of array elements will be displayed, if the format command is used along with `cout()` stream functions.

PROGRAM 7.2

A program to initialise a set of numbers in an array and to display them onto a standard output device along with a newline character (\n) between the elements.

```
#include <iostream>
using namespace std;
int main()
{
      int a[10] = { 0,1,2,3,4,5,6,7,8,9 };
      int i;
      cout <<"Contents of the array " << endl;
      for ( i = 0; i <= 9; ++i)
          cout << a[i] << '\n';
      return 0;
}
```

Output of the above program

```
Contents of the array
0
1
2
3
4
5
6
7
8
9
```

PROGRAM 7.3

A program to initialise a set of numbers in an array and to display them onto a standard output device along with a tab space character (\t) between the elements.

```
#include <iostream>
using namespace std;
int main()
{
      int a[10] = { 0,1,2,3,4,5,6,7,8,9 };
      int i;
      cout <<"Contents of the array " << endl;
      for ( i = 0; i <= 9; ++i)
            cout << a[i] << '\t';
      return 0;
}
```

Output of the above program

```
Contents of the array
0 1 2 3 4 5 6 7 8 9
```

PROGRAM 7.4

A program to demonstrate how to use the `iomanip` functions for formatting the elements of an array:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int a[10] = {0,1,2,3,4,5,6,7,8,9};
    int i;
    cout <<"Contents of the array " << endl;
    for (i = 0; i <= 9; ++i)
           cout << setw(5) << a[i];
    return 0;
}
```

Output of the above program

```
Contents of the array
0    1    2    3    4    5    6    7    8    9
```

7.5.2 Reading an Array Data Type

As there is no single statement or command available to read an entire array of elements, the use of the `cin()` function in the following program segment is invalid:

```
const int MAX = 20;
int main()
{
    int a[MAX],b[MAX],c[MAX];
    cin >> a;  // error
    cin >> b;  // error
    cin >> c;  // error
    ----------
    ----------
}
```

In the above program segment, as a single statement is used to read the contents of an entire array of matrix having 20 elements, a significant error message will be displayed.

To read the contents of an array, the subscripts must be enclosed along with the variables of the array type in the `cin()` function. For example, one can use the `cin()` function in the following way to get the contents of an array from the keyboard.

```
const int MAX = 20;
int main()
{
    int a[MAX],b[MAX],c[MAX];
    cin >> a[0];
    cin >> a[1];
    cin >> a[2];
    cin >> a[3];
    ----------
    ----------
    cin >> a[19];
}
```

In the above program, the `cin()` function is repeated 20 times to get the elements of the array. But one can avoid unnecessary repetition of `cin ()` function by using a loop statement for reading data items from

the keyboard. For example, in the above program segment, one can use the following loop statement to get the elements of the array.

```
const int MAX = 20;
int main()
{
    int a[MAX],b[MAX],c[MAX];
    int i;
    for ( i = 0; i <= MAX-1; ++i)
        cin >> a[i];
        -------
        -------
}
```

PROGRAM 7.5

A program to read 'n' numbers from the keyboard (where 'n' is defined by the programmer) to store it in an one-dimensional array and to display the content of that array onto the video screen.

```
// example 7.5
#include <iostream>
using namespace std;
int main()
{
    int a[100];
    int i,n;
    cout <<" How many numbers are in the array ? \n";
    cin >> n;
    cout <<"Enter the elements \n";
    for (i = 0; i <= n-1; ++i)
        cin >> a[i];
    cout <<" Contents of the array \n";
    for (i = 0; i <= n-1; ++i)
        cout << a[i] << '\t';
    return 0;
}
```

Output of the above program

```
How many numbers are in the array?
5
Enter the elements
10   11   12   13   14
Contents of the array
10   11   12   13   14
```

PROGRAM 7.6

A program to read a set of numbers from the keyboard and to find out the largest number in the given array (the numbers are stored in a random order).

```
// example 7.6
#include <iostream>
using namespace std;

int main()
{
    int a[100];
```

```
    int i,n,larg;
    cout <<" How many numbers are in the array ? \n";
    cin >> n;
    cout <<"Enter the elements \n";
    for (i = 0; i <= n-1; ++i)
        cin >> a[i];
    cout <<"Contents of the array \n";
    for (i = 0; i <= n-1; ++i)
        cout << a[i] << '\t';
    larg = a[0];
    for ( i = 0; i <= n-1; ++i) {
        if ( larg < a[i])
            larg = a[i];
    }
    cout <<" \n Largest value in the array = " << larg;
    return 0;
}
```

Output of the above program

```
How many numbers are in the array?
6
Enter the elements
11   22  -33    44   -65   3
Contents of the array
11   22  -33    44   -65   3
Largest value in the array = 44
```

PROGRAM 7.7

A program to read a set of numbers from the standard input device and to sort them in ascending order.

```
// example 7.7
#include <iostream>
using namespace std;
int main()
{
    int a[100];
    int i,j,n,temp;
    cout <<"How many numbers are in the array ? \n";
    cin >> n;
    cout <<"Enter the elements \n";
    for (i = 0; i <= n-1; ++i)
        cin >> a[i];
    cout <<"Contents of the array (unsorted form) \n";
    for (i = 0; i <= n-1; ++i)
        cout << a[i] << '\t';
    //sorting block
    for (i = 0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j)
            if (a[i] < a[j]) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
       }
    }
    cout <<"Contents of the array (sorted form)\n";
    for (i = 0; i <= n-1; ++i)
        cout << a[i] << '\t';
    return 0;
}
```

Output of the above program

```
How many numbers are in the array?
7
Enter the elements
44   -24   33    2     80    76    -7
Contents of the array (unsorted form)
44   -24   33    2     80    76    -7
Contents of the array (sorted form)
-24  -7    2     33    44    76    80
```

7.6 ARRAYS AND FUNCTIONS

The entire array can be passed on to a function in C++. An array name can be used as an argument for the function declaration. No subscripts or square brackets are required to invoke a function using arrays. The following program illustrates how to invoke a function using an array declaration.

```
#include <iostream>
using namespace std;
const int MAX = 100;
int main()
{
    int sumarray(int a[], int n);
    int a[MAX];
    int sum;
    -------
    -------
    sum = sumarray(a,n);
 }

int sumarray(int x[] ,int max)
{
    /* local variable declaration, if any
     -------
     -------
     body of the function  */

     return(value);
}
```

PROGRAM 7.8

A program to read a set of numbers from the keyboard and to find out the sum of all elements of the given array using a function.

```
// example 7.8
#include <iostream>
using namespace std;
const int MAX = 100;
int main()
{
    void display (int a[],int n); // function declaration
    int sumarray(int a[], int n);
    int a[MAX];
    int i,n,sum;
```

```
    cout <<"How many numbers are in the array ? \n";
    cin >> n;
    cout <<"Enter the elements \n";
    for (i = 0; i <= n-1; ++i)
        cin >> a[i];
    cout <<"contents of the array \n";
    display (a,n);
    sum = sumarray(a,n);
    cout <<"\n sum of the elements of the array = " << sum;
    cout <<"\n";
    return 0;
}
void display (int a[],int n)
{
    int i;
    for (i = 0; i <= n-1; ++i)
        cout << a[i] << '\t';
}

int sumarray(int x[] ,int max)
{
    int i,temp = 0;
    for (i = 0; i <= max-1; ++i)
        temp = temp +x[i];
    return(temp);
}
```

Output of the above program

```
How many numbers are in the array?
5
Enter the elements
1   2   3   4   5
Contents of the array
1   2   3   4   5
sum of the elements of the array = 15
```

PROGRAM 7.9

A program to read a set of numbers from the keyboard and to sort out the given array of elements in ascending order using a function.

```
// example 7.9
#include <iostream>
using namespace std;
const int MAX = 100;
int a[MAX];
int main()
{
    void getdata (int n); // function declaration
    void display(int a[], int n);
    void sort(int a[],int n);
    int n;
    cout <<"How many numbers are in the array ?\n";
    cin >> n;
    getdata(n);
    cout <<"Unsorted array" << endl;
    display(a,n);
    sort(a,n);
    cout <<"\n Sorted array " << endl;
    display(a,n);
    return 0;
```

```
} // end of the main program

void getdata(int n)
{
    int i;
    cout <<"Enter the elements \n";
    for (i = 0; i <= n-1; ++i)
        cin >> a[i];
}

void display (int a[],int n)
{
    int i;
    for (i = 0; i <= n-1; ++i)
        cout << a[i] << '\t';
}

int sort (int a[], int n)
{
    int temp,i,j;
    for (i = 0; i <= n-1; ++i) {
        for (j = 0; j <= n-2; ++j)
            if (a[i] < a[j]) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
    }
}
```

Output of the above program

```
How many numbers are in the array?
6
Enter the elements
11    22     -34     5     -6     47
Unsorted array
11    22     -34     5     -6     47
Sorted array
-34   -6     5       11    22     47
```

PROGRAM 7.10

A program to read a number `n`, and print it out digit by digit, as a series of words. For example, the number 756, should be printed as "`Seven Five Six`".

```
// example 7.10
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX = 20;
int a[MAX];
int main()
{
    void   nd(long int number);
    long int n;
    start:
        cout <<"Enter a number" << endl;
        cin >> n;
        if ( n < 0){
           cout <<" enter only positive numbers" << endl;
           goto start;
```

```
        }
         nd(n);
    return 0;
}
void  nd(long int n)
{
    void display (int b[], int max);
      int b[MAX];
      int i,j,max;
      i = 0;
    while (n > 0)
    {
        a[i] = n % 10;
        n  = n / 10;
        ++i;
    }
    cout <<" Number of digits = " << i;
    cout << "\n";
    max = i;
    --i;
    for ( j = 0; j <= max-1; ++j){
        b[j] = a[i];
        --i;
    }
    display(b,max);
    cout <<"\n";
}

void display (int b[], int max)
{
    int j;
    cout <<"number and its words" << endl;
    for ( j = 0; j <= max-1; ++j)
        cout << b[j];
    cout << '\t';
    j = 0;
    while (j != max) {
        switch (b[j]) {
            case 1:
                cout <<" one " << setw(6);
                break;
            case 2:
                cout <<" two " << setw(6);
                break;
            case 3:
                cout <<" three " << setw(6);
                break;
            case 4:
                cout <<" four " << setw(6);
                break;
            case 5:
                cout <<"  ve " << setw(6);
                break;
            case 6:
                cout <<" six " << setw(6);
                break;
            case 7:
                cout <<" seven " << setw(6);
                break;
            case 8:
                cout <<" eight " << setw(6);
                break;
            case 9:
                cout <<" nine " << setw(6);
                break;
            case 0:
```

```
                cout <<" zero " << setw(6);
                break;
            } // end of switch-case statement
           j = j+1;
    } // end of while statement
}
```

Output of the above program

```
Enter a number
45678
Number of digits = 5

number and its words
45678 four  ve six seven eight
```

PROGRAM 7.11

A program to read a set of numbers and to store it as a one-dimensional array; again read a number 'd' and check whether the number 'd' is present in the array. If it is so, print how many times the number 'd' is repeated in the array.

```
// example 7.11
#include <iostream>
using namespace std;
const int MAX = 20;
int a[MAX];
bool  ag;
int main()
{
    void   nd(long int number, int digit);
    long int n;
    int digit;
    st_one:
       cout <<"Enter a number" << endl;
       cin >> n;
       if (n < 0){
           cout <<" enter only positive numbers" << endl;
           goto st_one;
        }
    st_two:
       cout <<"Enter a digit to be checked ?" << endl;
       cin >> digit;
       if ( (digit < 0) || (digit > 9){
           cout <<" enter only positive and single digit number\n";
           goto st_two;
        }
         nd(n, digit);
        return 0;
}

void  nd(long int n, int digit)
{
       void display (int b[], int max, int digit);
       int b[MAX];
       int i,j,max;
       i = 0;
       while (n > 0)
       {
           a[i] = n % 10;
           n  = n / 10;
           ++i;
     }
```

```
      max = i;
      --i;
      for ( j = 0; j <= max-1; ++j){
          b[j] = a[i];
          --i;
    }
    display(b,max,digit);
}

void display (int b[], int max, int digit)
{
      int j,counter, ag;
      counter = 0;
       ag = false;
      for ( j = 0; j <= max-1; ++j){
            if ( b[j] == digit) {
               counter = counter + 1;
                ag = true;
            }
      } // end of for loop
      if (  ag == true) {
         cout <<" given digit = " << digit <<"  is present ";
         cout <<"in the number ";
         for ( j = 0; j <= max-1; ++j)
             cout << b[j];
         cout <<"\n and repeats = " << counter <<" times \n";
      }
      else
      {
         cout <<"entered digit = " << digit <<" is not present ";
         cout <<"in the number ";
         for ( j = 0; j <= max-1; ++j)
             cout << b[j];
         cout << "\n";
      }
}
```

Output of the above program

```
Enter a number
23452
Enter a digit to be checked?
2
given digit = 2  is present in the number 23452
and repeats = 2 times
```

PROGRAM 7.12

A program to read a set of numbers and store it as a one-dimensional array; again read a number `n` *and check whether it is present in the array. If it is so, print the position of* `n` *in the array and also check whether it is repeated in the array.*

```
// example 7.12
#include <iostream>
using namespace std;
const int MAX = 20;
int a[MAX];
bool  ag;
int main()
{
    void   nd(long int number, int digit);
    long int n;
```

```
    int digit;
    st_one:
        cout <<"Enter a number \n";
        cin >> n;
        if (n < 0){
            cout <<" enter only positive numbers\n";
            goto st_one;
        }
    st_two:
        cout <<"Enter a digit to be checked ?\n";
        cin >> digit;
        if ( (digit < 0) || (digit > 9)){
            cout <<" enter only positive and single digit number\n";
            goto st_two;
    }
     nd(n, digit);
    return 0;
}

void  nd(long int n, int digit)
{
    void display (int b[], int max, int digit);
    int b[MAX];
    int i,j,max;
    i = 0;
    while (n > 0)
    {
        a[i] = n % 10;
        n  = n / 10;
        ++i;
    }
    max = i;
    --i;
    for ( j = 0; j <= max-1; ++j){
        b[j] = a[i];
        --i;
    }
    display(b,max,digit);
}

void display (int b[], int max, int digit)
{
    int j,k,counter, ag;
    counter = 0;
     ag = false;
    k = 0;
    for ( j = 0; j <= max-1; ++j){
        if ( b[j] == digit) {
            counter = counter + 1;
             ag = true;
            a[k] = 1+j;
            k++;
        }
    } // end of for loop
    if (  ag == true) {
        cout <<" given digit = "<< digit << " is present \n";
        cout <<" in the number  ";
        for ( j = 0; j <= max-1; ++j)
        cout << b[j];
        cout <<"\n and its position is ";
        for ( j = 0; j <= k-1; ++j)
             cout << a[j];
        cout <<" from left to right \n";
    }
    else
    {
        cout <<" entered digit  = " << digit <<" is not present \n";
```

```
        cout <<" in the number ";
        for ( j = 0; j <= max-1; ++j)
            cout << b[j];
        cout << "\n";
    }
}
```

Output of the above program

```
Enter a number
34562
Enter a digit to be checked?
2
given digit = 2 is present
in the number  34562
and its position is 5 from left to right

Enter a number
12345
Enter a digit to be checked?
9
entered digit  = 9 is not present
in the number 12345
```

7.7 MULTIDIMENSIONAL ARRAYS

Multidimensional arrays are defined in the same manner as one-dimensional arrays, except that a separate pair of square brackets are required for each subscript. Thus, a two-dimensional array will require two pairs of square brackets; a three-dimensional array will require three pairs of square brackets, and so on.

The genearal format of the multidimensional array is,

```
storage_class data_type arrayname [expression1][expression2]...
                    [expression n];
```

where `storage_class` refers to the scope of the array variable such as external, static, automatic or register; `data_type` refers to the nature of the data elements in the array such as character type, integer type or floating point, etc., and `arrayname` is the name of the multidimensional array. `expression1, expression2 . . . expression n` refers to the maximum size of the each array locations.

For example,

```
 oat coordinate x[10][10];
int value [50][10][5];
char line [10][80];
static double records [100][100][10];
```

In the above illustration, the first line defines the coordinate as a floating point array having 10 rows, 10 columns with a total of 100 elements. The second one is a three-dimensional integer array whose maximum size is 2500 elements.

Multidimensional array initialisation Similar to one-dimensional array, multidimensional arrays can also be initialised, if one intends to assign some values to these elements. It should be noted that only external or static arrays can be initialised. For example, consider the following two-dimensional array declaration:

(1)

```
int x[2][2] = { 1,2,3,4};
```

where x is a two-dimensional array of integer numbers whose maximum size is 4 and the assignments would be

```
x[0][0] = 1;
```

```
x[0][1] = 2;
x[1][0] = 3;
x[1][1] = 4;
```

(2)

```
static  oat sum [3][4] = {0.1,0.2,0.3,0.4,0.6,0.7,0.8,0.9,1,1,1,1};
```

where sum is a static two-dimensional array of floating point numbers and the maximum size of the array is 3 rows and 4 columns having a total of 12 elements.

The assignments would be

```
sum [0][0] = 0.1  sum [0][1] = 0.2  sum [0][2] = 0.3  sum[0][3] = 0.4
sum [1][0] = 0.6  sum [1][1] = 0.7  sum [1][2] = 0.8  sum[1][3] = 0.9
sum [2][0] = 1    sum [2][1] = 1    sum [2][2] = 1    sum[2][3] = 1
```

The natural order in which the initial values are assigned can be altered by forcing groups of initial values enclosed within braces, i.e. {...}.

For example, in the following a two-dimensional array can be declared.

```
int A[3][3] = {
              { 1,2,3},
              { 4,5,6},
              { 7,8,9}
              };
```

In the above array declaration, the three values in the first inner pair of braces are assigned to the array element in the first row; the values in the second pair of braces are assigned to the array element in the second row and so on.

The elements in the above array A will be assigned as

```
A[0][0] = 1  A[0][1] = 2     A[0][2] = 3
A[1][0] = 4  A[1][1] = 5     A[1][2] = 6
A[2][0] = 7  A[2][1] = 8     A[2][2] = 9
```

Now consider the following two-dimensional array definition,

```
int matrixa[3][3] = {
                    {1,2},
                    {4,5},
                    {7,8}
                    };
```

This definition assigns values only to the first two elements in each row, and hence, the array elements will have the following initial values

```
matrixa[0][0] = 1  matrixa[0][1] = 2     matrixa[0][2] = 0
matrixa[1][0] = 4  matrixa[1][1] = 5     matrixa[1][2] = 0
matrixa[2][0] = 7  matrixa[2][1] = 8     matrixa[2][2] = 0
```

Note that the last element in each row is assigned a value zero.

If we declare the two-dimensional array as

```
int matrixa[3][3] = {1,2,3,4,5,6,7};
```

then following assignment will be carried out :

```
matrixa[0][0] = 1  matrixa[0][1] = 2     matrixa[0][2] = 3
matrixa[1][0] = 4  matrixa[1][1] = 5     matrixa[1][2] = 6
matrixa[2][0] = 7  matrixa[2][1] = 0     matrixa[2][2] = 0
```

Two of the array elements will be again assigned zero, though the order of the assignment will be different. If one declares the array definition as

```
int a[3][3] =     {
                  {1,2,3,4},
```

```
                        {5,6,7,8},
                        {9,10,11,12}
                        };
```

then, the maximum array allocation for the above two dimensional declaration is 3×3 = 9 elements. But the initial definition of the array elements have exceeded the declared size and hence, normally, the compiler will produce the error message. To avoid the above problem, the array may be declared and written as,

```
      int a[3][4] =     {
                        {1,2,3,4},
                        {5,6,7,8},
                        {9,10,11,12}
                        };
```

PROGRAM 7.13

A program to read the elements of a given matrix of order n × n and to display the contents of the matrix on the screen.

```
// example 7.13
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX = 10;
int main()
{
     oat a[MAX][MAX];
    int i,j,n;
    cout <<" order of the matrix " << endl;
    cin >> n;
    cout <<" enter the elements" << endl;
    for (i = 0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j)
            cin >> a[i][j];
    }
    cout <<" output of the matrix" << endl;
    for (i = 0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j){
            cout << setw(4) << a[i][j];
        }
        cout <<"" << endl;
    }
    return 0;
}
```

Output of the above program

```
order of the matrix
3
enter the elements
1   2   3
4   5   6
7   8   9
output of the matrix
   1   2   3
   4   5   6
   7   8   9
```

PROGRAM 7.14

A program to initialise a set of numbers in a two-dimensional array and to display the content of the array on the screen.

```
// example 7.14
#include <iostream>
#include <iomanip>
using namespace std;
const int N = 3;
const int M = 4;
int main()
{
    int i,j;
    double a[N][M] = {
                     {1,2,3,4},
                     {5,6,7,8},
                     {9,10,11,12}
                     };
    cout <<"Contents of the array " << endl;
    for (i = 0; i <= N-1; ++i) {
        for (j = 0; j <= M-1; ++j) {
            cout << setprecision(2);
            cout << setw(4) << a[i][j];
        }
        cout <<"" << endl;
    }
    return 0;
}
```

Output of the above program

```
Contents of the array
   1    2    3    4
   5    6    7    8
   9   10   11   12
```

PROGRAM 7.15

A program to initialise only a few elements of a two-dimensional array and to display the content of the array on the screen.

```
// example 7.15
#include <iostream>
#include <iomanip>
using namespace std;
const int N = 3;
const int M = 4;
int main()
{
    int i,j;
     oat a[N][M] = {
                   {1,2,3},
                   {5,6,7},
                   {9,10,11}
                   };
    cout <<"Contents of the array " << endl;
    for (i = 0; i <= N-1; ++i) {
        for (j = 0; j <= M-1; ++j)
```

```
        cout << setw(4) << a[i][j];
        cout <<"" << endl;
    }
    return 0;
}
```

Output of the above program

```
Contents of the array
   1   2   3   0
   5   6   7   0
   9  10  11   0
```

PROGRAM 7.16

A program to read the elements of the given two matrices of order $n \times n$ and to perform the matrix addition.

```
// example 7.16
// matrix addition
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX = 100;
int main()
{
    // function declaration
    void output ( oat a[MAX][MAX],int n);
    void add ( oat a[MAX][MAX], oat b[MAX][MAX],int n);
     oat a[MAX][MAX],b[MAX][MAX];
    int i,j,n;
    cout <<"Order of matrix " << endl;
    cin >> n;
    cout <<"Enter the elements of A Matrix " << endl;
    for (i =  0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j)
            cin >> a[i][j];
    }
    cout <<"Enter the elements of B Matrix " << endl;
    for (i =  0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j)
            cin >> b[i][j];
    }
    cout <<"Output A[i][j] " << endl;
    output (a,n);
    cout <<"" << endl;
    cout <<"Output B[i][j] " << endl;
    output (b,n);
    add (a,b,n);
    return 0;
}

void add ( oat a[MAX][MAX],  oat b[MAX][MAX] , int n)
{
    void output ( oat c[MAX][MAX],int n); // function declaration
     oat c[MAX][MAX];
    int i,j,k;
    for (i = 0; i <= n-1; ++i) {
       for (j = 0; j <= n-1; ++j)
           c[i][j] = a[i][j]+b[i][j];
    }
    cout <<"" << endl;
    cout <<"Output of C[i][j] matrix" << endl;
    output(c,n);
}
```

```
void output ( oat x[MAX][MAX],int n)
{
     int i,j;
     for (i = 0; i <= n-1; ++i) {
     for (j = 0; j <= n-1; ++j)
          cout << setw(4) << x[i][j];
          cout <<"" << endl;
     }
}
```

Output of the above program

```
Order of matrix
4

Enter the elements of A Matrix
1  1  1  1
1  1  1  1
1  1  1  1
1  1  1  1

Enter the elements of B Matrix
2  2  2  2
2  2  2  2
2  2  2  2
2  2  2  2

Output A[i][j]
   1   1   1   1
   1   1   1   1
   1   1   1   1
   1   1   1   1

Output B[i][j]
   2   2   2   2
   2   2   2   2
   2   2   2   2
   2   2   2   2

Output of C[i][j] matrix
   3   3   3   3
   3   3   3   3
   3   3   3   3
   3   3   3   3
```

PROGRAM 7.17

A program to read the elements of the given two matrices of order n × n and to perform the matrix subtraction.

```
// example 7.17
// matrix subtraction
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX = 100;
```

```
int main()
{
    // function declaration
    void output ( oat a[MAX][MAX],int n);
    void sub ( oat a[MAX][MAX], oat b[MAX][MAX],int n);
     oat a[MAX][MAX],b[MAX][MAX];
    int i,j,n;
    cout <<"Order of matrix " << endl;
    cin >> n;
    cout <<"Enter the elements of A Matrix " << endl;
    for (i =  0; i <= n-1; ++i) {
       for (j = 0; j <= n-1; ++j)
           cin >> a[i][j];
    }
    cout <<"Enter the elements of B Matrix " << endl;
    for (i =  0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j)
            cin >> b[i][j];
    }
    cout <<"Output A[i][j] " << endl;
    output (a,n);
    cout <<"" << endl;
    cout <<"Output B[i][j] " << endl;
    output (b,n);
    sub (a,b,n);
    return 0;
}

void sub ( oat a[MAX][MAX],  oat b[MAX][MAX] , int n)
{
    void output ( oat c[MAX][MAX],int n); // function declaration
     oat c[MAX][MAX];
    int i,j,k;
    for (i = 0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j) {
            c[i][j] = a[i][j]-b[i][j];
        }
    }
    cout <<"" << endl;
    cout <<"Output of C[i][j] matrix" << endl;
    output(c,n);
}

void output ( oat x[MAX][MAX],int n)
{
    int i,j;
    for (i = 0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j)
          cout << setw(4) << x[i][j];
        cout <<"" << endl;
    }
}
```

Output of the above program

```
Order of matrix
3
Enter the elements of A Matrix
1  1  1
1  1  1
1  1  1
Enter the elements of B Matrix
2  2  2
2  2  2
2  2  2
```

```
Output A [i][j]
   1   1   1
   1   1   1
   1   1   1

Output B [i][j]
   2   2   2
   2   2   2
   2   2   2
Output of C [i][j] matrix
  -1  -1  -1
  -1  -1  -1
  -1  -1  -1
```

PROGRAM 7.18

A program to read the elements of the given two matrices of order n × n and to perform the matrix multiplication.

```
// example 7.18
// matrix multiplication
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX = 100;
int main()
{
    // function declaration
    void output ( oat a[MAX][MAX],int n);
    void mul ( oat a[MAX][MAX], oat b[MAX][MAX],int n);
     oat a[MAX][MAX],b[MAX][MAX];
    int i,j,n;
    cout <<"Order of matrix " << endl;
    cin >> n;
    cout <<"Enter the elements of A Matrix " << endl;
    for (i =  0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j)
            cin >> a[i][j];
    }
    cout <<"Enter the elements of B Matrix " << endl;
    for (i =  0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j)
            cin >> b[i][j];
    }
    cout <<"Output A[i][j] " << endl;
    output (a,n);
    cout <<"" << endl;
    cout <<"Output B[i][j] " << endl;
    output (b,n);
    mul (a,b,n);
    return 0;
}

void mul ( oat a[MAX][MAX],  oat b[MAX][MAX], int n)
{
    void output ( oat c[MAX][MAX],int n); // function declaration
     oat c[MAX][MAX];
    int i,j,k;
    for (i = 0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j) {
```

```
            c[i][j] = 0.0;
            for (k = 0; k <= n-1; ++k)
                c[i][j] = c[i][j]+a[i][k]*b[k][j];
        }
    }
    cout <<"" << endl;
    cout <<"Output of C[i][j] matrix" << endl;
    output(c,n);
}

void output ( oat x[MAX][MAX],int n)
{
    int i,j;
    for (i = 0; i <= n-1; ++i) {
        for (j = 0; j <= n-1; ++j)
            cout << setw(4) << x[i][j];
        cout <<"" << endl;
    }
}
```

Output of the above program

```
Order of matrix
3
Enter the elements of A Matrix
1  1  1
1  1  1
1  1  1

Enter the elements of B Matrix
1  1  1
1  1  1
1  1  1

Output A [i][j]
   1   1   1
   1   1   1
   1   1   1

Output B [i][j]
   1   1   1
   1   1   1
   1   1   1

Output of C [i][j] matrix
   3   3   3
   3   3   3
   3   3   3
```

PROGRAM 7.19

A program to find the sum of the elements of a given three dimensional array in which data are read from the keyboard.

```
// example 7.19
// three dimensional array
#include <iostream>
```

```
#include <iomanip>
using namespace std;
const int MAX = 10;
int main()
{
     oat a[MAX][MAX][MAX];
    int i,j,k,n;
     oat total;
    cout <<"Order of the three dimensional matrix ?";
    cin >> n;
    cout <<"Enter the elements " << endl;
    for (i = 0; i <= n-1; ++i) {
        for ( j = 0; j <= n-1; ++j){
            for ( k = 0; k <= n-1; ++k)
                cin >> a[i][j][k];
        }
    }
    //  nding the sum of the elements
    total = 0;
    for (i = 0; i <= n-1; ++i) {
        for ( j = 0; j <= n-1; ++j){
            for ( k = 0; k <= n-1; ++k)
                total = total+a[i][j][k];
        }
    }
    // displaying the contents of the array
    cout <<" contents of the array " << endl;
    for (i = 0; i <= n-1; ++i) {
        for ( j = 0; j <= n-1; ++j){
           for ( k = 0; k <= n-1; ++k){
               cout <<" [" << i << j << k << "] = " ;
               cout << setw(4) << a[i][j][k];
           }
           cout <<"" << endl;
       }
       cout <<"" << endl;
    }
    cout <<" " << endl;
    cout <<" sum of the elements = " << total;
    return 0;
}// end of the main program
```

Output of the above program

```
Order of the three dimensional matrix ?
2
Enter the elements
1 1
2 2
3 3
4 4

 contents of the array
 [000] = 1 [001] = 1
 [010] = 2 [011] = 2

 [100] = 3 [101] = 3
 [110] = 4 [111] = 4

 sum of the elements = 20
```

7.8 CHARACTER ARRAY

The procedure for declaring character array is almost the same as for other data types such as integer or floating point. One can declare the character array by means of alphanumeric characters. The general format of the character array is,

```
storage_class  character_data_type array_name[expression];
```

where the storage class is optional and it may be either one of the scope of the variable such as automatic, external, static, or register; the array name can be any valid C++ identifier and the expression a positive integer constant.

For example,

```
char page [40];
char sentence[300];
static char line[50];
```

The basic structure of the character array is

R	A	V	I	C	\0

Each element of the array is placed in a definite memory space and each element can be accessed separately. The array element should end with the null character as a reference for the termination of a character array.

Initialising the character array

Like an integer or a floating point array, the character array can also be initialised,

For example,

```
char colour[3] = "RED";
```

The elements would be assigned to each of the character array positions in the following way:

```
colour [0] = 'R';
colour [1] = 'E';
colour [2] = 'D';
```

The character array always terminate with the null character, that is, a back-slash followed by a letter zero (not letter `'l289'` or `'o'`), and the computer will treat them as a single character. The null character will be added automatically by the C++ compiler provided there is enough space to accommodate the character.

For example,

```
char name [5] = "ravic"  /* wrong */
```

The following assignment would be for the each cell,

```
name [0] = 'r';
name [1] = 'a';
name [2] = 'v';
name [3] = 'i';
name [4] = 'c';
```

The above declaration is wrong because there is no space to keep the null character in the array as a termination character and it can be corrected by redefining the above array as

```
char name [6] = "ravic"  /* right */
```

The following assignment would be for the each cell,

```
name [0] = 'r';
name [1] = 'a';
name [2] = 'v';
```

```
name [3] = 'i';
name [4] = 'c';
name [5] = '\0';
```

Following is a valid array declaration:

```
char line[] = "this is a test program"
```

The square brackets can be empty, as the array size would have been specified as part of the array definition.

PROGRAM 7.20

A program to initialise a set of characters in a one-dimensional character array and to display the content of the given array.

```
// example  7.20
#include <iostream>
using namespace std;
int main ()
{
    int  i;
    static char name[5] = { 'r','a','v','i','c'};
    cout <<"Contents of the array" << endl;
    for (i = 0; i <= 4; ++i)
        cout <<"name[" << i <<"] = " << name[i] << '\n';
   return 0;
}
```

Output of the above program

```
Contents of the array
name[0] = r
name[1] = a
name[2] = v
name[3] = i
name[4] = c
```

PROGRAM 7.21

A program to initialise a string of characters in a one-dimensional array and to display the content of the array.

```
// example  7.21
#include <iostream>
using namespace std;
int main()
{
    int  i;
    static char name[] = "this is a test program";
    cout <<"Contents of the array" << endl;
    for (i = 0; name[i] != '\0'; ++i)
        cout <<"name[" << i <<"] = " << name[i] << '\n';
    return 0;
}
```

Output of the above program

```
Contents of the array
name[0] = t
name[1] = h
```

```
name[2] = i
name[3] = s
name[4] =
name[5] = i
name[6] = s
name[7] =
name[8] = a
name[9] =
name[10] = t
name[11] = e
name[12] = s
name[13] = t
name[14] =
name[15] = p
name[16] = r
name[17] = o
name[18] = g
name[19] = r
name[20] = a
name[21] = m
```

PROGRAM 7.22

A program to read a set of lines from the keyboard and to store it in a one-dimensional array and to display the content of the array on the screen.

```
// example  7.22
// reading a set of lines from keyboard
// and displaying onto the video screen

#include <iostream>
using namespace std;
const int MAX = 1000;
int main()
{
    char line[MAX];
    char ch;
    int i;
    cout <<"enter a set of lines and terminate with @\n";
    i = 0;
    while (( ch = cin.get()) != '@') {
        line[i++] = ch;
   }
   line[i++] = '\0';
   cout <<" output from the array\n";
   for ( i = 0; line[i] != '\0'; ++i){
       cout.put(line[i]);
   }
   return 0;
}
```

Output of the above program

```
enter a set of lines and terminate with @
this is a test program
by Ravich
@
output from the array
this is a test program
by Ravich
```

PROGRAM 7.23

A program to read a set of lines from the keyboard; store it in a one-dimensional array; find the number of characters of a given text and also display the contents of the array on the screen.

```
// example  7.23
//  nding a number of characters of a given text
#include <iostream>
using namespace std;
const int MAX = 1000;
int main()
{
    int number (char line[] );
    char line[MAX];
    char ch;
    int i,n_ch;;
    cout <<"enter a set of lines and terminate with @\n";
    i = 0;
    while (( ch = cin.get()) != '@') {
        line[i++] = ch;
    }
    line[i++] = '\0';
    n_ch = number (line);
    cout <<" output from the array\n";
    for ( i = 0; line[i] != '\0'; ++i){
        cout.put(line[i]);
    }
    cout <<" Number of characters = " << n_ch;
    return 0;
}

// function to  nd the number of characters
int number (char s[])

{
    int i = 0;
    while (s[i] != '\0')
        ++i;
    return(i);
}
```

Output of the above program

```
enter a set of lines and terminate with @
this is
a test
program
@
output from the array
this is
a test
program
Number of characters = 24
```

PROGRAM 7.24

A program to read a set of lines from the keyboard; store it in a one-dimensional array; find the number of characters and lines in a given text and also display the contents of the array on the screen.

```
// example  7.24
//  nding a number of characters and lines of a given text
#include <iostream>
using namespace std;
const int MAX = 1000;
int main()
{
    int number_ch (char line[]);
    int number_line(char line[]);
    char line[MAX];
    char ch;
    int i,n_ch,n_ln;
    cout <<"enter a set of lines and terminate with @\n";
    i = 0;
    while ((ch = cin.get()) != '@') {
        line[i++] = ch;
    }
    line[i++] = '\0';
    n_ch = number_ch (line);
    n_ln = number_line(line);
    cout <<" output from the array\n";
    for ( i = 0; line[i] != '\0'; ++i){
        cout.put(line[i]);
    }
    cout <<" Number of characters = " << n_ch;
    cout <<"\n Number of lines = " << n_ln;
    cout << '\n';
    return 0;
}

// function to  nd the number of characters
int number_ch (char s[])
{
    int i = 0;
    while (s[i] != '\0')
        i++;
    return(i-1);
}

// function to  nd the number of lines
int number_line (char s[])
{
    int i = 0, n_ln = 0;
    while (s[i] != '\0')
    {
        if ( s[i] == '\n')
            n_ln++;
        ++i;
    }
    return(n_ln);
}
```

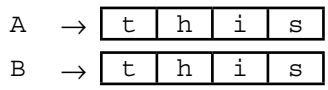
Output of the above program

```
enter a set of lines and terminate with @
this
is
test
@

output from the array
this
is
test
Number of characters = 12
Number of lines = 3
```

PROGRAM 7.25

A program to read a set of lines from the keyboard; store it in a one-dimensional array A; copy the contents of A to an array B and display the contents of arrays A and B separately.

For program,




```
// example  7.25
// string copy
#include <iostream>
using namespace std;
const int MAX = 1000;
int main()
{
    void stringcopy (char dest[], char source[] );
    char line[MAX],page[MAX];
    char ch;
    int i;
    cout <<"enter a set of lines and terminate with @\n";
    i = 0;
    while (( ch = cin.get()) != '@') {
        line[i++] = ch;
    }
    line[i++] = '\0';
    stringcopy (page,line);
    cout <<" output from the array (line)\n";
    for ( i = 0; line[i] != '\0'; ++i){
        cout.put(line[i]);
    }
    cout <<" output from the array (page)\n";
    for ( i = 0; page[i] != '\0'; ++i){
        cout.put(page[i]);
    }
    return 0;
}

// function to perform the string copy
void stringcopy (char dest[], char source [])
{
    int i = 0;
    while (source[i] != '\0') {
        dest[i] = source[i];
        i++;
    }
    dest[i++] = '\0';
}
```

Output of the above program

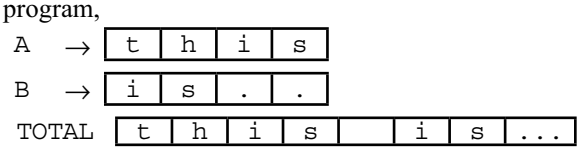
```
enter a set of lines and terminate with @
this is
a test
program by
Ravich
@

output from the array (line)
this is
```

```
a test
program by
Ravich

output from the array (page)
this is
a test
program by
Ravich
```

PROGRAM 7.26

A program to read a set of lines from the keyboard; store it in the array `A`; again read a set of lines from the keyboard and store it in the array `B`; copy the contents of array `A` and array `B` into an array TOTAL and display the contents of arrays `A` and `B` and `TOTAL` separately.

For program,




```
// example  7.26
// string concatenate
#include <iostream>
using namespace std;
const int MAX = 1000;
int main()
{
    void stringconcat (char dest[], char sour1[],char sour2[]);
    char source1[MAX],source2[MAX],total[MAX];
    char ch;
    int i;
    cout <<"enter a set of lines and terminate with @ \n";
    i = 0;
    while ((ch = cin.get()) != '@') {
           source1[i++] = ch;
    }
    source1[i++] = '\0';
    cin.get(); // delete an extra line feed character
    cout <<" Another input data " << endl;
    cout <<"enter a set of lines and terminate with @ \n";
    i = 0;
    while ((ch = cin.get()) != '@') {
           source2[i++] = ch;
    }
    source2[i++] = '\0';
    stringconcat (total,source1,source2);
    cout <<" output from the array (source1)\n";
    for ( i = 0; source1[i] != '\0'; ++i){
        cout.put(source1[i]);
    }
    cout <<" output from the array (source2)\n";
    for ( i = 0; source2[i] != '\0'; ++i){
        cout.put(source2[i]);
    }
    cout <<" output from the array (total)\n";
    for ( i = 0; total[i] != '\0'; ++i){
        cout.put(total[i]);
    }
```

```
    return 0;
}

// function to perform the string concatenate
void stringconcat (char dest[], char sour1 [], char sour2[])
{
    int i = 0, j;
    while (sour1[i] != '\0') {
        dest[i] = sour1[i];
        i++;
    }
    j = 0;
    while (sour2[j] != '\0') {
        dest[i] = sour2[j];
        i++;
        j++;
    }
    dest[i++] = '\0';
}
```

Output of the above program

```
enter a set of lines and terminate with @
this is a
test
@

Another input data
enter a set of lines and terminate with @
program by
Ravich
@

output from the array (source1)
this is a
test

output from the array (source2)
program by
Ravich

output from the array (total)
this is a
test
program by
Ravich
```

PROGRAM 7.27

A program to read a set of lines from the keyboard; store it in a one-dimensional array `A`*; remove white spaces such as horizontal tab, vertical tab, back space, line feed and new line from the contents of the array and display the contents of array.*

```
// example  7.27
// removing white space
const int SIZE = 1000;
#include <iostream>
using namespace std;
int main()
```

```
{
    char source[SIZE];
    char ch;
    int i,max;
    cout <<"enter a set of lines and terminate with @ \n";
    i = 0;
    while ((ch = cin.get()) != '@') {
        source[i++] = ch;
    }
    source[i++] = '\0';
    max = i;
    cout <<" output from the array\n";
    for (i = 0; i <= max-1; ++i)
        cout.put(source[i]);
    cout <<" output after removing white space \n";
    // A single space between ' ' is for a space character
    for (i = 0; i <= max-1; ++i) {
        if ((source[i] == ' ') || (source[i] == '\t') ||
           (source[i] == '\n') || ( source[i] == '\\') ||
           (source [i] == '\r') || (source [i] == '\f'))
           cout <<"";
        else
           cout.put (source[i]);
    }
    return 0;
}
```

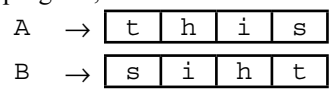
Output of the above program

```
enter a set of lines and terminate with @
this is a test
program by
Sampath K Reddy
@

output from the array
this is a test
program by
Sampath K Reddy

output after removing white space
thisisatestprogrambySampathKReddy
```

PROGRAM 7.28

A program to read a set of lines from the keyboard; store it in a one-dimensional array A; perform the string reversal of the given text. The last character of the array is displayed as the first character and so on.

For program,




```
// example  7.28
#include <iostream>
using namespace std;
const int SIZE = 1000;
int main()
{
    void reverse(char s[SIZE],char d[SIZE]);
    char source[SIZE],dest[SIZE];
```

```
    char ch;
    int i,max;
    cout <<"enter a set of lines and terminate with @ \n";
    i = 0;
    while ((ch = cin.get()) != '@') {
       source[i++] = ch;
    }
    source[i++] = '\0';
    max = i;
    cout <<" output from the array\n";
    for (i = 0; i <= max-1; ++i)
        cout.put(source[i]);
    reverse(source,dest);
    cout <<" After string reversal \n";
    for (i = 0; i <= max-1; ++i)
        cout.put(dest[i]);
    return 0;
}

void reverse(char s[SIZE], char d[SIZE])
{
   int stringlen(char a[SIZE]);
   int i,j;
   j = stringlen(s);
   i = 0;
   while ( j >= 0) {
      d[i] = s[j];
      i++;
      j--;
    }
}
int stringlen(char a[SIZE])
{
   int i=0;
   while ( a[i] != '\0')
     i++;
   return (i);
}
```

Output of the above program

```
enter a set of lines and terminate with @
this is a
test program
@

output from the array
this is a
test program

After string reversal
margorp tset
a si siht
```

REVIEW QUESTIONS

1. What is an array? Explain how an array variable is different from an ordinary variable.
2. What is an array indexing?
3. Explain how the individual elements are accessed and processed in an array.
4. State the rules used to declare a one-dimensional array.

5. What is meant by array initialisation?
6. Explain the salient features of an array and their uses.
7. How are arrays usually processed in C++?
8. Can an array name be used as an argument to a function? Explain.
9. Summarise the rules for passing arrays to a function.
10. What is a multidimensional array and how is it different from a one-dimensional array?
11. Summarise the syntatic rules to be governed for declaring a multidimensional array.
12. How are data elements intilialised in a multidimensional array? What are the scope rules for the multidimensional array?
13. Explain how the individual elements of a multidimensional array are accessed and processed.
14. What is a character array? How is it different from other data type arrays?
15. Distinguish between a character array and a string.
16. Can a list of strings be stored within a two-dimensional array?
17. Explain the differences between an array of characters and an array of integers.
18. Explain the pros and cons of a multidimensional array over a single dimensional in C++.
19. Summarises the various formats for declaring two-dimensional arrays in C++.

CONCEPT REVIEW PROBLEMS

1. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    const int a[6] = {1,2,3,4};
    cout <<"contents of the array\n";
    for (int i = 0; i <= 5; ++i)
        cout << a[i] << '\t';
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    int a[5] = {1,2,3,4,5};
    cout <<"contents of the array\n";
    for (int i = 0; i <= 4; ++i) {
        a[i] = i*i;
        cout << a[i] << '\t';
    }
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
```

```
int main()
{
    const int a[5] = {1,2,3,4,5};
    cout <<"contents of the array\n";
    for (int i = 0; i <= 4; ++i) {
        a[i] = i*i;
        cout << a[i] << '\t';
    }
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    static int a[5] = {1,2,3,4,5};
    void display (int a[],int n);
    int n = 5;
    cout <<"contents of the array in main\n";
    for (int i = 0; i <= n-1; ++i) {
        a[i] = i*i;
        cout << a[i] << '\t';
    }
    display(a,n);
    return 0;
}
void display(int a[],int n)
{
    cout <<"\ncontents of the array in function\n";
    for (int i = 0; i <= n-1; ++i) {
        a[i] = i*i;
        cout << a[i] << '\t';
    }
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    static int a[5] = {1,2,3,4,5};
    void display (int a[],int n);
    int n = 5;
    cout <<"contents of the array in main\n";
    for (int i = 0; i <= n-1; ++i) {
        a[i] = a[i] % 2;
        cout << a[i] << '\t';
    }
    display(a,n);
    return 0;
}
void display(int a[],int n)
{
    cout <<"\ncontents of the array in function\n";
```

```
    for (int i = 0; i <= n-1; ++i) {
        a[i] = i % 2;
        cout << a[i] << '\t';
    }
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    int a[] = {1,2,3,4,5};
    cout <<"Contents of the array\n";
    for (int i = 0; i <= 4; ++i)
        cout <<"a[" << i <<"] = " << a[i] << '\n';
    return 0;
}
```

2. What will be the output of each of the following program when it is executed?

(a)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int a[3][3] = {
                  {1},
                  {0,1},
                  {0,0,1}
                  };
    cout <<"Contents of the array \n";
    for (int i = 0; i <= 2; ++i) {
        for (int j = 0; j <= 2; ++j)
            cout << a[i][j] << setw(5);
        cout << "\n";
    }
    return 0;
}
```

(b)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int sum(int a[3][3], int n);
    void display(int a[3][3],int n);
    int n = 3,total;
    int a[3][3] = {
                  {1},
                  {0,1},
                  {0,0,1}
                  };
    display(a,n);
    total = sum(a,n);
    cout << "Sum of all elements in the array = "<<total <<"\n";
```

```
}
void display (int a[3][3],int n)
{
     cout <<"Contents of the array \n";
     for (int i = 0; i <= n-1; ++i) {
        for (int j = 0; j <= n-1; ++j)
          cout << a[i][j] << setw(5);
        cout << "\n";
     }
}
int sum (int a[3][3],int n)
{
     int temp = 0;
     for (int i = 0; i <= n-1; ++i) {
        for (int j = 0; j <= n-1; ++j)
            temp = temp+a[i][j];
     }
     return (temp);
}
```

(c)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int sum(int a[3][3], int n);
    void display(int a[3][3],int n);
    int n = 3,total;
    int a[3][3] = {
                  {1,2,13},
                  {4,15,6},
                  {17,8,9}
                  };
    display(a,n);
    total = sum(a,n);
    cout << "Sum of all diagonal elements in the array = "<<total;
    cout <<"\n";
}
void display (int a[3][3],int n)
{
    cout <<"Contents of the array \n";
    for (int i = 0; i <= n-1; ++i) {
        for (int j = 0; j <= n-1; ++j)
            cout << setw(6) << a[i][j];
        cout << "\n";
    }
}
int sum (int a[3][3],int n)
{
    int temp = 0, max = n-1;
    for (int i = 0; i <= n-1; ++i) {
        for (int j = 0; j <= n-1; ++j)
          if ( j == max) {
             temp = temp+a[i][j];
             max--;
          }
```

```
    }
    return (temp);
}
```

(d)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int sum(int a[3][3], int n);
    void display(int a[3][3],int n);
    int n = 3,total;
    int a[3][3] = {
                  {1,2,13},
                  {4,15,6},
                  {17,8,9}
                  };
    display(a,n);
    total = sum(a,n);
    cout << "Sum of all diagonal elements in the array = "<<total;
    cout <<"\n";
}

void display (int a[3][3],int n)
{
    cout <<"Contents of the array \n";
    for (int i = 0; i <= n-1; ++i) {
        for (int j = 0; j <= n-1; ++j)
            cout << setw(6) << a[i][j];
        cout << "\n";
    }
}

int sum (int a[3][3],int n)
{
    int temp = 0;
    for (int i = 0; i <= n-1; ++i) {
        for (int j = 0; j <= n-1; ++j)
            if ( i == j)
               temp = temp+a[i][j];
    }
    return (temp);
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    int a[5] = {1,2,3,4,5};
    int sum (int a[],int n);
    int n = 5,total = 0;
    cout <<"contents of the array in main\n";
      for (int i = 0; i <= n-1; ++i) {
          a[i] = a[i] % 5;
```

```
            cout << a[i] << '\t';
        }
        total = sum(a,n);
        cout << "\n sum of all elements = " << total << "\n";
        return 0;
}

int sum (int a[],int n)
{
    int temp = 0;
    for (int i = 0; i <= n-1; ++i)
       temp = temp+a[i];
    return (temp);
}
```

PROGRAMMING EXERCISES

1. Write a program in C++ to read an integer number and find out the sum of all the digits till it reduces to a single digit using an array. For example,

   (i) `n = 1256`
   `sum = 1+2+5+6 = 14`
   `sum = 1+4 = 5`

   (ii) `n = 7896`
   `sum = 7+8+9+6 = 30`
   `sum = 3+0 = 3`

2. Write a program in C++ to read a set of numbers up to n (where n is defined by the programmer) and print the contents of the array in reverse order.
   For example, for `n = 4`, let the set be
   `26 56 51 123`
   which should be printed as
   `123 51 56 26`

3. Write a program in C++ to read `n` numbers (where `n` is defined by the programmer) and find the average of the non-negative integer numbers. Find also the deviation of the numbers.

4. Write a program in C++ to read a set of numbers and store it in a one-dimensional array; to find the largest and the smallest number. Find also the difference between the two numbers. Using the difference, find the deviation of the numbers of the array.

5. Write a program in C++ to read a set of numbers to store it in a one-dimensional array A; copy the elements into another array B in the reverse direction; find the sum of the individual elements of the array A and B. Store the results in another array C and display all the three arrays.

6. Write a program in C++ to read a four digit positive integer number `n` and generate all the possible permutation of numbers using the above digits. For example, `n = 7812` then the permutations are
   `7821`
   `8721`
   `8712`
   `2871`
   `2817`
   (**Hint:** Read a number `n` and separate it digit by digit and store it in an array and then generate a permutation.)

7. Write a program in C++ to read a two-dimensional square matrix A and display its transpose. Transpose of matrix A is obtained by interchanging all the elements of rows and columns of original matrix A.
8. Write a program in C++ to read a two-dimensional array and find the sum of the elements in each row and column separately and display the sum of the elements in the rows and columns.
9. Write a program in C++ to generate a magic square A, where the sum of the elements in each row and column are the same.
10. Write a program in C++ to read a set of lines and find out the number of characters, words, and lines in a given text.
11. Write a program in C++ to read a line and find out the number of vowels (a, e, i, o, u) and consonants present in the given line.
12. Write a program in C++ to read a set of lines from the stdin and print out the longest line.
13. Write a program in C++ to read a line, encode the line and display the original and encoded form. The encode should be

```
a b c d ... z
    z y x w ... a
```

14. Write a program to read an encoded form of a line given in the above problem (No 16) and display the decoded form on the stdout.
15. Write a program to read any four characters and print out all the possible combinations. For example, for ABCD

```
ACBD
ADBC
ADCB
.
.
.
```

16. Write a program to read a student's name and his average mark. If a student gets less than 40 then declare that he has failed or else passed. Prepare a computer list to give the list of names in alphabetical order separately for passed and failed students.
17. Write a program to read the names of books, authors and accession numbers of all books in a library. Check whether the given accession number is present in the array, if it is not, print the name of the book and the author's name.
18. Write a program to read a set of lines from stdin and store them in an array A. Again read a string S from the stdin and check whether the given string S is in the array A. If it is so, print that line and also how many times the string is repeated in the array A.
19. Write a program to read a set of lines from stdin and store them in an array `A`. Again read a string `S` from the `stdin` and check whether the given string `S` is in the array `A`. If it is not, remove string `S` from the array `A` and print the updated array on the `stdout`. For example,

```
A = concatenate
S = cat
```

The updated A is `conenate`

20. Write a program to read a set of lines from stdin and store them in an array `A`. Again read strings `S1` and `S2` from the stdin and check whether the given string `S1` is in the array `A`. If it is not, replace the string `S1` with string `S2` and print the updated array. For example,

```
A  = concatenate
S1 = cat
S2 = 123
```

The updated `A` is `con123enate`

Pointers and Strings

Chapter 8

This chapter deals with one of the important topics namely, pointer. This chapter also elucidates how a pointer variable is declared, used and manipulated in C++ programming. The major operations and applications of the pointer declarations, pointer arithmetic, pointer operators, pointer to functions, pointer to arrays and pointer to pointer are covered in a very simple manner and illustrated with numerous examples.

8.1 INTRODUCTION

It is well known that pointer data type is somewhat difficult to understand by the novice programmers but it is one of the strengths of the C++ language. The pointer is a powerful technique to access the data by indirect reference as it holds the address of that variable where it has been stored in the memory.

8.1.1 Pointer Declaration

A pointer is a variable which holds the memory address of another variable. Sometimes, only with the pointer a complex data type can be declared and accessed easily. The pointer has the following advantages:

- It allows to pass variables, arrays, functions, strings and structures as function arguments.
- A pointer allows to return structured variables from functions.
- It provides functions which can modify their calling arguments.
- It supports dynamic allocation and deallocation of memory segments.
- With the help of a pointer, variables can be swapped without physically moving them.
- It allows to establish links between data elements or objects for some complex data structures such as linked lists, stacks, queues, binary trees, tries and graphs.
- A pointer improves the efficiency of certain routines.

A pointer contains a memory address. Most commonly, this address is the location of another variable where it has been stored in memory. If one variable contains the address of another variable, then the first

variable is said to point to the second. In C++, pointers are distinct such as integer pointer, floating point number pointer, character pointer, etc. A pointer variable consists of two parts, namely, (i) the pointer operator, and (ii) the address operator.

8.1.2 Pointer Operator

A pointer operator can be represented by a combination of * (asterisk) with a variable. For example, if a variable of integer data type and also declared * (asterisk) with another variable, it means the variable is of type "pointer to integer". In other words, it will be used in the program indirectly to access the value of one or more integer variables.

The general format of the pointer declaration is given below:

```
data_type *pointer_variable;
```

where `data_type` is a type of pointer variable such as integer, character and floating point number, structs, arrays, etc. and the `pointer_variable` is any valid C++ identifier. Note that the asterisk must be preceded by the pointer variable.

For example,

```
int *ptr;
```

where `ptr` is a pointer variable which holds the address of an integer data type. The internal representation of `ptr` is shown in the following Fig. 8.1.

Fig. 8.1 *Memory Address of a variable x*

All pointer variables must be declared before it is used in C++ programs like other variables. When a pointer variable is declared, the variable name must be preceded by an asterisk (*). This identifies the variable as a pointer.

Following are valid pointer declarations.

```
 oat *fpointer;
double *dpoint;
char *mpoint1;
```

The base type of the pointer indicates the type of variable the pointer is pointing to. Technically, any type of pointer can point to anywhere in the memory. All pointer arithmetic is done relative to its base type. Hence, it is important to declare pointers correctly. For example, a character data item may require to store only a byte (8 bits); an integer may require to store two bytes (16 bits); a floating point number may require four bytes (32 bits), a double precision number may require eight bytes (64 bits). The number of bytes required to store a particular data items varies from machine to machine.

8.1.3 Address Operator

An address operator can be represented by a combination of & (ampersand) with a pointer variable. For example, if a pointer variable is an integer type and also declared & with the pointer variable, then it means that the variable is of type "address of". In other words, it will be used in the program to indirectly access the value of one or more integer variables.

The & is a unary operator that returns the memory address of its operand. A unary operator requires only one operand.

For example,

```
m = &ptr;
```

Note that the pointer operator & is an operator that returns the address of the variable following it. Therefore, the preceding assignment statement could be verbalised as "`m` receives the address of `ptr`".

The other operator * is the complement of &. It is a unary operator that returns the value of the variable located at the address that follows.

The operation of * translates to the phrase "at address". Note that the symbol * represents both the multiplication sign and the "at address", and the symbol & represents both bitwise AND and the "address of" sign. When these are used as pointer operators, they have no relationship to the arithmetic operators that happen to look like the same. Both the pointer operators, & and *, have a higher precedence over all other arithmetic operators except the unary minus, with which they have equal precedence.

The pointer and address operators & and * are the members of the same precedence group as the other unary operators such as -, ++, --, !, sizeof and cast operator. Note that the group unary operators have higher precedence over the group of arithmetic operators, and the associatives of the unary operators are from left to right.

8.1.4 Pointer Expressions

(a) Pointer Assignment A pointer is a variable data type and hence the general rule to assign its value to the pointer is same as that of any other variable data type. For example,

```
int x,y;
int *ptr1,*ptr2;
```

(1)

```
ptr1 = &x;
```

The memory address of variable x is assigned to the pointer variable ptr1.

(2)

```
y = *ptr1;
```

The contents of the pointer variable ptr1 is assigned to the variable y, not the memory address.

(3)

```
ptr1 = &x;
ptr2 = ptr1;     // address of ptr1 is assigned to ptr2
```

The address of the ptr1 is assgined to the pointer variable ptr2. The contents of both ptr1 and ptr2 will be the same as these two pointer variables hold the same address. The following Fig. 8.2 shows how an address of ptr1 is assigned to ptr2.

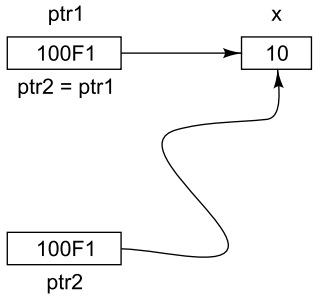


Fig. 8.2 *Address of ptr1 is assigned to ptr2*

Some invalid pointer declaration

(1)

```
int x;
int x_pointer;
x_pointer = &x;
```

Error: pointer declaration must have the prefix of * (unary operator).

(2)

```
 oat y;
 oat *y_pointer;
y_pointer = y;
```

Error: While assigning variable to the pointer variable the address operator (&) must be used along with the variable y.

(3)

```
int x;
char *c_pointer;
c_pointer = &x;
```

Error: Mixed data type is not permitted.

(b) Finding the Address of an Object As we described earlier, every variable has a unique address that identifies its storage location in memory. For some applications, it is useful to access the variable through its address rather than through its name. To obtain the address of a variable, programmer has to use the ampersand (&) operator.

Suppose, for instance, that j is the long int whose address is 248600, the statement,

```
ptr = &j;
```

stores the address value 248600 in the variable ptr. When reading an expression, the ampersand operator is translated as "address of". One would read this statement as: " Assign the address of j to the ptr". The following program points the value of the variable called j and the address of j:

```
#include <iostream>
using namespace std;
int main()
{
    int j  = 100;
    cout <<" value of j = " << j << endl;
    cout <<" address of j = " << &j << endl;
    return 0;
}
```

Output of the above program

```
value of j = 100
address of j = 0x1ab4fff2
```

The address represents the actual location of j in memory. The particular address listed above is arbitrary. Note that one cannot use the ampersand operator on the left hand side of an assignment expression. For instance, the following is illegal since one cannot change the address of an object:

```
&ptr = 1000;  // error
```

(c) Initialising Pointers One can initialise a pointer variable just like any other type of variable in a program. However, the initialisation value must be an address. The following pointer declaration is valid

```
int j;
int *ptr = &j;
```

However, one cannot reference a variable before it is declared, so the following declarations would be illegal:

```
int *ptr = &j;
int j;  // error
```

(d) The NULL Pointer The C++ language supports the notion of a null pointer. A null pointer is a method or approach in which a pointer variable is guaranteed not to point to a valid object. In other words, a null pointer is any pointer assigned the integral value zero.

For example,

```
char *ptr;
ptr = 0;  // make ptr a null pointer
```

Another form of using the null pointer is given below:

```
ptr = NULL;
```

NULL is a built-in constant for assigning the integral value zero.

Null pointers are particularly useful in control flow statements since the zero valued pointer evaluates to false, whereas all other pointer values evaluates to true. For example, the following while loop continues iterating while ptr is null pointer.

```
char *ptr;
-------
-------
while (ptr) {
  -------
  -------
}  // iterate until ptr is a null pointer
```

This use of null pointers is particularly prevalent in applications that are arrays of pointers.

PROGRAM 8.1

A program to assign an address of an integer variable to the pointer variable and display the content of and address of the pointer.

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    int  *ptr;
    x = 10;
    ptr = &x;
    cout << " x = " <<  x <<" and ptr  = " << ptr;
    cout << endl;
    cout << " x = " << x << " and *ptr = " << *ptr;
    cout << endl;
    return 0;
}
```

Output of the above program

```
x = 10 and ptr  = 0xbfffe7d4
x = 10 and *ptr = 10
```

In the above program, x is an integer variable and ptr is declared as the pointer to an integer. Initially, value 10 is assigned to the integer variable x. The address of the variable x is assigned to the ptr.

```
    ptr = &x        //address of x is assigned to the variable ptr
   *ptr = &x        //content of x is assigned to the *ptr
```

PROGRAM 8.2

A program to assign a character variable to the pointer and display the content of the pointer.

```
#include <iostream>
using namespace std;
int main()
```

```
{
    char x,y;
    char *pointer;
    x = 'c';  // assign character
    pointer = &x;
    y = *pointer;
    cout << " value of x = " << x;
    cout << endl;
    cout << " pointer value = " << y;
    cout << endl;
    return 0;
}
```

Output of the above program

```
value of x = c
pointer value = c
```

PROGRAM 8.3

A program to display the address and the content of a pointer variable.

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    int *ptr;
    x = 10;
    ptr = &x;
    cout << "value of x = "<< x << '\n';
    cout << "contents of ptr = " << *ptr << '\n';
    cout << "Address of ptr = " << ptr << '\n';
    return 0;
}
```

Output of the above program

```
value of x = 10
contents of ptr = 10
Address of ptr = 0xbffff654
```

PROGRAM 8.4

A program to assign the pointer variable to another pointer and display the contents of both the pointer variables.

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    int *ptr1,*ptr2;
    x = 10;
    ptr1 = &x;
    ptr2 = ptr1;
    cout << "value of x = " << x << '\n';
    cout << "Contents of ptr1 = " << *ptr1 << '\n';
    cout << "Contents of ptr2 = " << *ptr2 << '\n';
    return 0;
}
```

Output of the above program

```
value of x = 10
Contents of ptr1 = 10
Contents of ptr2 = 10
```

8.2 POINTER ARITHMETIC

As a pointer holds the memory address of a variable, some arithmetic operations can be performed with pointers. C++ supports four arithmetic operators that can be used with pointers, such as

```
Addition            +
Subtraction         -
Incrementation      ++
Decrementation      --
```

Pointers are variables. They are not integers, but they can be displayed as unsigned integers. The conversion specifier for a pointer is added and subtracted. For example,

`ptr++` causes the pointer to be incremented, but not by 1.
`ptr--` causes the pointer to be decremented, but not by 1 .

The following program segment illustrates the pointer arithmetic.
The integer value would occupy bytes 2000 and 2001

```
int value , *ptr;
value = 120;
ptr = &value;
ptr++;
cout << ptr;
```

The above C++ program segment displays 2002.
Figure 8.3(a) shows how a pointer arithmetic is carried out for an `int` data type.

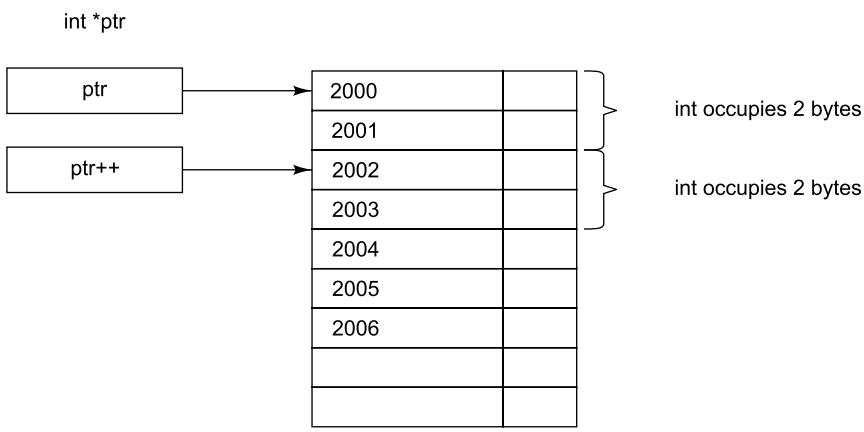


Fig. 8.3(a) *Pointer Arithmetic*

The pointer `ptr` is originally assigned the value 2000. The incrementation, `ptr++`, increments the number of bytes of the storage class for the particular machine. If the system used four bytes to store an integer, then `ptr++` would have resulted in `ptr` being equal to 2004.

The general rule for pointer arithmetic is that pointer performs the operation in bytes of the appropriate storage class. Figure 8.3(b) illustrates how a pointer arithmetic is performed for a floating point data type.

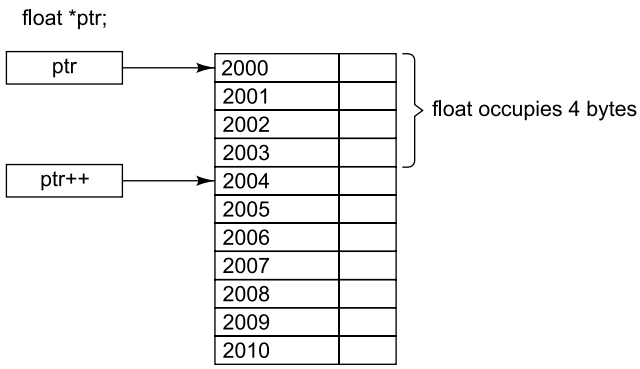


Fig. 8.3(b) *Pointer Arithmetic*

PROGRAM 8.5

A program to display the memory address of a variable using pointer before incrementation and after incrementation.

```
// pointer arithmetic
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int value ;
    int *ptr;
    value = 120;
    ptr = &value;
    cout << "Memory address before incrementation = " << ptr;
    cout << endl;
    ptr++;
    cout << "Memory address after incrementation = " <<  ptr;
    cout << endl;
    return 0;
}
```

Output of the above program

```
Memory address before incrementation = 0xbfffe7d4
Memory address after incrementation = 0xbfffe7d8
```

PROGRAM 8.6

A program to display the memory address of a variable using pointer before decrementation and after decrementation.

```
// pointer arithmetic
#include <iostream>
using namespace std;
int main()
{
```

```
     oat value ;
     oat *ptr;
    value = 120.00;
    ptr = &value;
    cout << "Memory address  = " << ptr <<endl;
    ptr--;
    cout << "Memory address after decrementer = " << ptr << endl;
    return 0;
}
```

Output of the above program

```
Memory address  = 0xbffff8d4
Memory address after decrementer = 0xbffff8d0
```

Each time a pointer is incremennted, it points to the memory location of the next element with its base type. Each time it is decremented, it points to the location of the previous element. In the case of pointers to characters, this often produces what appears to be "normal" arithmetic. However, all other pointers increase or decrease the length of the data type they point to.

For example, assume 1 byte characters and 2 byte integers. When a character pointer is incremented, its value increases by 1. However, when an integer pointer is incremented, its value increases by 2. The reason for this is that each time a pointer is incremented or decremented, it is incremented or decremented relative to its length of its base type so that it always points to the next element.

All pointer arithmetic are done relative to the base type of the pointer so that the pointer is always pointing to the appropriate element of the base type. The pointers are not limited to only incrementing or decrementing. A pointer variable may be added or subtracted to or from integers. For example,

```
ptr = ptr +9 ;
```

makes `ptr` to point to the ninth element of `ptr` type beyond the element it currently points to.

PROGRAM 8.7

A program to display the memory address of a variable using a pointer; add an integer quantity with the pointer and to display the contents of the pointer.

```
// pointer arithmetic
#include <iostream>
using namespace std;
int main()
{
    int x;
    int *ptr1,*ptr2;
    x = 10;
    ptr1 = &x;
    ptr2 = ptr1+6;
    cout << "\n value of x = " << x;
    cout << "\n Contents of ptr1 = " << *ptr1;
    cout << "\n Address of ptr1 = " << ptr1;
    cout << "\n Address of ptr2 = (ptr1+6) = " << ptr2;
    cout << "\n Contents of ptr2 = " << *ptr2;
    return 0;
}
```

Output of the above program

```
value of x = 10
Contents of ptr1 = 10
Address of ptr1 = 0xbffff1d4
Address of ptr2 = (ptr1+6) = 0xbffff1ec
Contents of ptr2 = 1073829932 (garbage value)
```

The memory address of the pointer variable `ptr1` is `0xbffff1d4` and an integer value 6 is added with a pointer `ptr2`. For one integer, the pointer variable takes 4 bytes to store, then, the resultant address value is `0xbffff1ec`. The content of the pointer is a garbage data.

No other arithmetic operations are allowed other than addition and subtraction with pointers on integers. To be specific, pointers are not permitted to perform the following arithmetic operations.

(i) To multiply or divide,
(ii) To operate the bitwise shift and mask operations, and
(iii) To add or subtract type float or type double to pointers.

PROGRAM 8.8

A program to display the address and content of a pointer variable; subtract with an integer quantity and to display the address of and the contents the pointer variable.

```
// pointer arithmetic
#include <iostream>
using namespace std;
int main()
{
    int x;
    int *ptr1,*ptr2;
    x = 10;
    ptr1 = &x;
    ptr2 = ptr1-2;
    cout << "value of x =" << x <<endl;
    cout << "Contents of ptr1 = "  << *ptr1 <<endl;
    cout << "Address of ptr1 =" << ptr1<<endl;
    cout << "Address of ptr2 = (ptr1-2) =" << ptr2 << endl;
    cout << "Contents of ptr2 =" << *ptr2 <<endl;
    return 0;
}
```

Output of the above program

```
value of x =10
Contents of ptr1 = 10
Address of ptr1 =0xbfffed54
Address of ptr2 = (ptr1-2) =0xbfffed4c
Contents of ptr2 =-1073746612
```

PROGRAM 8.9

A program to display the contents of the pointer variables using arithmetic operation.

```
// pointer arithmetic
#include <iostream>
using namespace std;
int main()
{
    int x,y;
    int *ptr;
    x = 10;
    ptr = &x;
    cout << "\n value of x =" << x << " and pointer =" << *ptr;
    y = *ptr +1;
    cout << "\n value of y =" << y << " and pointer =" << *ptr;
    return 0;
}
```

Output of the above program

```
value of x =10 and pointer =10
value of y =11 and pointer =10
```

PROGRAM 8.10

A program to display the contents of a pointer variable before and after incrementation.

```
//pointer arithmetic
#include <iostream>
using namespace std;
int main()
{
    int x,y;
    int *ptr;
    x = 10;
    ptr = &x;
    cout << "\n value of x =" << x <<" and pointer =" << *ptr;
    y  = ++ *ptr;
    cout << "\n value of y =" << y <<" and pointer =" <<  *ptr;
    cout << '\n';
    return 0;
}
```

Output of the above program

```
value of x = 10 and pointer = 10
value of y = 11 and pointer = 11
```

PROGRAM 8.11

A program to display the contents and the address of a pointer variable using different types of incrementation.

```
//pointer arithmetic
#include <iostream>
using namespace std;
int main()
{
    int x,y;
    int *ptr, *ptr2;
    x = 10;
    ptr = &x;
    cout << "content of pointer ="<<*ptr<<endl;

    *ptr = *ptr +1;   // (*ptr++)
    y = *ptr;
    cout << "value of y =" << y << '\t';
    cout << "and pointer *ptr = *ptr +1 =" << *ptr<<endl;

    *ptr += 1;   // (*ptr++)
    y = *ptr;
    cout << "value of y = " << y << '\t';
    cout << "and pointer *ptr += 1 =" << *ptr<<endl;

    (*ptr)++;  // parentheses are necessary
    y = *ptr;
    cout << "value of y = " << y << '\t';
    cout << " and pointer (*ptr)++ =" << *ptr<<endl;

    ++ *ptr;
```

```
    y = *ptr;
    cout << "value of y =" << y << '\t';
    cout << "and pointer (++ *ptr) =" << *ptr<<endl;

    ++ *ptr;
    ptr2 = ptr;
    cout << "pointer1 = " << *ptr << '\t';
    cout << "and pointer2 =" << *ptr2 <<endl;
    return 0;
}
```

Output of the above program

```
content of pointer = 10
value of y = 11  and pointer *ptr = *ptr + 1 = 11
value of y = 12  and pointer *ptr + = 1 = 12
value of y = 13  and pointer (*ptr)++ = 13
value of y = 14  and pointer (++ *ptr) = 14
pointer1 = 15    and pointer2 = 15
```

PROGRAM 8.12

A program to display the content of a pointer variable using an ordinary and pointer arithmetic.

```
// pointer arithmetic
#include <iostream>
using namespace std;
int main()
{
    int x,y;
    int *xpointer;
    int temp;
    temp = 3;
    x = 5* (temp+5);
    xpointer = &temp;
    y = 5* (*xpointer+5);
    cout << "x = " << x <<endl;
    cout << "y = " << y <<endl;
    return 0;
}
```

Output of the above program

```
x = 40
y = 40
```

The above program consists of two expressions — one, an ordinary arithmetic expression and the other a pointer expression.

```
x = 5 * (temp +5)
where temp = 3
x = 5 *(3 + 5)
  = 5 * (8)
  = 40
```

Using the pointer arithmetic,

```
xpointer = &temp;
```

where `temp` = 3;

```
y = 5 * (*xpointer + 5)
  = 5 *  (3 + 5)
  = 5 * 8 = 40
```

8.2.1 Summary of Pointer Arithmetic

Pointer arithmetic	*description*
`ptr++`	`ptr = ptr+sizeof(data_type)` use original value of `ptr` and then `ptr` is incremented after statement execution
`++ptr`	`ptr = ptr+sizeof( data_type)` original `ptr` is incremented before execution of statement
`ptr--`	`ptr = ptr-sizeof(data_type)` use original value of `ptr` and then `ptr` is decremented after statement execution
`--ptr`	`ptr = ptr-sizeof( data_type)` original `ptr` is decremented before the execution of statement
`*ptr++`	`*(ptr++)` retrieve the content of the location pointed to by pointer and then increment `ptr`
`* ++ptr`	`*(++ptr)` increment pointer and then retrieve the content of the new location pointed to by `ptr`
`(*ptr)++`	increment content of the location pointed to by `ptr`. For pointer type content, use pointer arithmetic else use standard arithmetic
`++ *ptr`	`++ (*ptr)` increment the content of the location pointed to by `ptr` depending on the type of the content
`-- *ptr`	`-- (*ptr)` decrement content of the location pointed to by `ptr` depending on the type of the content
`*ptr--`	`*(ptr--)` retrieve the content of the location pointed to by `ptr` and then decrement `ptr`
`* --ptr`	`*(--ptr)` decrement `ptr`, then retrieve the content of the new location pointed to by `ptr`
`(*ptr)--`	retrieve content `*ptr` of the location pointed to by `ptr`, then decrement the content of that location; `ptr` is not changed

8.3 POINTERS AND FUNCTIONS

Pointers are very much used in a function declaration. Sometimes only with a pointer a complex function can be easily represented and accessed. The use of pointers in a function definition may be classified into two groups; they are call by value and call by reference.

8.3.1 Call by Value

C++ passes parameters into a function using call by value in which copies of the actual parameters are made in temporary variables and these are passed. Even if they are represented with the same variable names, internally they are treated as different items.

The call by value type of writing a user-defined function is passing some formal arguments to a function but the function does not return back any value to the caller. It is one way data communication between a calling portion of the program and the function block.

Whenever a portion of the program invokes a function with formal arguments, control will be transferred from the main to the calling function and the value of the actual argument is copied to the function. Within the function, the actual value copied from the calling portion of the program may be altered or changed. When the control is transferred back from the function to the calling portion of the program, the altered values are not transferred back. This type of passing formal arguments to a function is technically known as *call by value*.

The main characteristics of the call by value are:

(1) The formal parameter assumes only the value of the actual parameter. The address of the actual parameter is not known in the subprograms and hence, whatever changes made in the formal parameter in a subprogram cannot be reflected back to the calling portion of a program.
(2) In other words, formal and actual parameters are two distinct variables and the C++ compiler reserves the memory space which required for these parameters separately, of course, by default, automatically.
(3) Value parameters do not allow the transfer of values from the subprograms to the calling portion of a program.
(4) The actual parameter may be any expression, constants, variables, functions or a combination of them.
(5) Call by value also allows declaration of parameters of the formal arguments as arrays, structs, unions and classes.
(6) C++ compiler treats the formal parameters as call by value, by default.
(7) Value parameters do not require any extra keywords or any specific commands in the C++ language, for their definition.

For example, the following program segment illustrates the use of the call by value.

```
#include <iostream>
using namespace std;
int main ()
{
    void funct (int x , int y); // function declaration
    int x = 10,y = 20;
    funct (x,y);  //  call by value
    cout << " x = " << x << " ,y = " << y << endl;
    return 0;
}
void funct (int a, int b)
{
   a = a*a; //  new values will not be re ected into the main
   b = b*b;
}
```

Output of the above program

```
x = 10 ,y = 20
```

PROGRAM 8.13

A program to exchange the contents of two variables using a call by value.

```
// call by value
#include <iostream>
using namespace std;
int main()
{
    int x,y;
    void swap (int, int);
    x = 100;
    y = 20;
    cout << " values before swap () " << endl;
    cout << " x =  " << x << " and  y = "<< y <<endl;
    swap (x,y); // call by value
    cout << " values after swap () " << endl;
    cout << " x =  " << x << " and  y = "<< y <<endl;
    return 0;
}
```

```
void swap( int x, int y)  // values will not be swapped
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Output of the above program

```
values before swap ()
x =  100 and  y = 20
values after swap ()
x =  100 and  y = 20
```

Since the above function is declared call by value, the desired result is not obtained. As a single variable is transferred to the function, it protects the value of this variable from alteration within the function. On the other hand, it prevents the altered value being transferred back from the function to the calling portion of the program.

PROGRAM 8.14

A program to display the contents of a variable using call by value in both main and a function.

```
// call by value
#include <iostream>
using namespace std;
int main()
{
    int x = 100;
    void display(int x);
    cout << x;
    display(x); // value is passed, not address
    cout<< '\t' << x;
    return 0;
}
void display(int x)
{
    cout<< '\t' << ++x;
}
```

Output of the above program

```
100  101  100
```

In the above program, parameters are passed to a function using call by value in which copies of the actual parameters are made in temporary variables and these are passed. Even if they are represented with the same variable names, internally they are treated as different items. When the control is transferred back from the function to the calling portion of the program, the altered values are not transferred back.

8.3.2 Call by Reference

The call by reference is a type of method invocation in which the address or the location of the parameters are passed to the calling portion of a function. In other words, when a function is called by a portion of the program, the address of the actual arguments are copied onto the formal arguments, even though, they are referenced in different variable names.

The values that are altered or changed within the function are reflected into the calling portion of a program in the altered form itself, as the formal arguments and the real arguments are referencing the same memory location or address. Only the user-defined variable names are different in the function block and in

the calling portion of a program but technically the scope and the life of the variables are same throughout the program. If the contents of the variable parameters are altered in one part of the program, these will be reflected back, wherever they are referred in the program. This type of declaring a function in C++ is *technically known as call by reference*, or *call by address* or *call by location*.

The main characteristics of call by reference are:

(1) The formal parameter gets the address of the actual parameter and hence, the formal and actual variables are same. Though, sometimes user-defined identifiers are referred in different names in the formal arguments and in the list of the actual arguments, they are internally considered the same items.
(2) In order to differentiate, the method of declaring call by reference and call by value, it is essential to use the indirection operator `(*)` as a prefix to the variables at the time of declaring the variables in the function definition part.
(3) Whenever the contents of the formal parameter are changed in a subprogram, the new values are reflected back to the actual parameter also as both the actual and formal variables hold the same address. So, the content of the address will be same for both actual and formal parameters.
(4) In the call by reference, more than one value of the parameters can be returned from a subprogram to the calling portion of a program.
(5) Since the parameters are treated by default as call by value, it is a must to define the parameters prefixed with the indirection operator `(*)` in order to give directions to the compiler that such parameters should be treated as variable parameters.
(6) In the case of call by reference, the actual parameter list must have only the variables. It cannot have constants or function parameters.

For example, the following program segment illustrates the use of call by reference.

```
#include <iostream>
using namespace std;
int main ()
{
    void funct (int *x , int *y); // function declaration
    int x = 10,y = 20;
    funct (&x,&y);  //  call by reference
    cout << " x = " << x << " ,y = " << y << endl;
    return 0;
}
void funct (int *a, int *b)
{
   *a = *a * *a; //  new values will be re ected throughout the program
   *b = *b * *b;
}
```

Output of the above program

```
 x = 100 ,y = 400
```

PROGRAM 8.15

A program to exchange the contents of two variables using call by reference (version 1).

```
// call by reference
//swap function version 1.cpp
#include <iostream>
using namespace std;
int main()
```

```
{
    int x,y;
    void swap (int *x, int *y);
    x = 100;
    y = 20;
    cout << " values before swap () " << endl;
    cout << " x =  " << x << " and  y = "<< y <<endl;
    swap (&x,&y); // call by reference
    cout << " values after swap () " << endl;
    cout << " x =  " << x << " and  y = "<< y <<endl;
    return 0;
}
void swap(int *x, int *y)  // values will be swapped
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Output of the above program

```
values before swap ()
x =  100 and  y = 20
values after swap ()
x =  20 and  y = 100
```

PROGRAM 8.16

A program to exchange the contents of two variables using call by reference (version 2).

```
//call by reference
//swap function version 2.cpp
#include <iostream>
using namespace std;
int main()
{
    int x,y;
    void swap (int &x, int &y);
    x = 100;
    y = 20;
    cout << " values before swap () " << endl;
    cout << " x =  " << x << " and  y = "<< y <<endl;
    swap (x,y);
    cout << " values after swap () " << endl;
    cout << " x =  " << x << " and  y = "<< y <<endl;
    return 0;
}

void swap(int &x, int &y)  // values will be swapped
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Output of the above program

```
values before swap ()
x =  100 and  y = 20
values after swap ()
x =  20 and  y = 100
```

PROGRAM 8.17

A program to display the contents of a variable using call by reference in both main and a function.

```
// call by reference
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    void display(int *x);
    cout << x;
    display(&x); // address is passed, not the content
    cout<< '\t' << x;
    return 0;
}
void display(int *x)
{
    cout<< '\t' << ++*x;
}
```

Output of the above program

```
10  11  11
```

The call by reference is a type of function definition, declaration and invocation in which the address of the parameter is passed to the function. The function can use the address to access and alter the memory allocated to the parameter. When the control is transferred back from the function to the calling portion of the program, the altered values are reflected throughout the program.

PROGRAM 8.18

A program to display the contents of a variable and its memory address using call by reference in both main and a function.

```
// call by reference
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    void display(int *ptr);
    cout << "Address of x = " << &x;
    cout << endl;
    cout << "Contents of x = " << x << '\n';
    display(&x); // address is passed, not the content
    return 0;
}
void display(int *ptr)
{
    cout <<"Address of the x (inside function) = " << ptr;
    cout << endl;
    cout << "Contents of x =" << *ptr;
}
```

Output of the above program

```
Address of x = 0xbfffef54
Contents of x = 10
Address of the x (inside function) = 0xbfffef54
Contents of x = 10
```

The formal parameter gets the address of the actual parameter and hence, the formal and actual variables are same. Though, sometimes user-defined identifiers are referred in different names in the formal arguments and in the list of the actual arguments, they are internally considered the same items.

8.4 POINTERS TO FUNCTIONS

In the previous section, we have seen how a pointer data type is so important for the program development. This section presents how a function can be referenced with pointer data type and how these concepts can be implemented and realised in C++.

Pointers to functions are a powerful tool because they provide an elegant way to call different functions based on the input data. C++ allows functions to be referenced by a pointer. A pointer to a function must be declared to be a pointer variable to the data type returned by the functions, like void, int, float, and so on. In addition, the argument type of the function must also be specified when the pointer is declared. The argument declaration is a list of formal arguments, separated by commas and enclosed in parentheses.

The general syntax of a pointer to a function is,

```
return_type (* variables) ( list of parameters);
```

For example, following are valid examples.

(1)

```
void (*ptr)( oat, oat,int);
```

In the above declaration, a pointer to a function returns `void` and takes the formal arguments of two `oat` and one `int`.

(2)

```
oat (*ptr) (char,double,int, oat);
```

In the second declaration of a pointer to a function, it returns a floating point value and takes the formal arguments of `char`, `double`, `int` and `oat`.

Note that a pointer variable name and de-referencing operator `(*)` are enclosed in parentheses and precedes the list of argument data types. Suppose, the following a pointer to a function declaration is written without parentheses.

```
void *ptr ( oat, oat,int)
```

then C++ compiler interprets it as in the following way which is not the intention of the above declaration.

```
void * (ptr ( oat, oat,int));
```

After the declaration of a pointer to a function, the address of the function must be assigned to a pointer.

```
ptr = &average; // where average is a function name
```

The starting address of the function average is loaded into a pointer `ptr` which is pointing to a function.

The following program segment shows how a pointer to a function is declared, assigned and invoked for finding the average of three numbers,

```
//pointers to functions
void main ()
{
     oat average ( oat, oat, oat); //function declaration
     oat a,b,c,avg;
     oat (*ptrf)( oat, oat, oat);// pointer to function declaration
    ptrf = &average;
    -------
    -------
    avg = (*ptrf)(a,b,c); // function calling using pointer
```

```
}
 oat average ( oat x,  oat y,  oat z)
{
   -------
   -------
}
```

PROGRAM 8.19

A program to find the sum of three numbers using a pointer to function method.

```
//pointers to functions
#include <iostream>
using namespace std;
int main ()
{
     oat average ( oat, oat, oat); //function declaration
     oat a,b,c,avg;
     oat (*ptrf)( oat, oat, oat);// pointer to function declaration
    ptrf = &average;
    cout << " enter three numbers \n";
    cin >> a >> b >> c;
    avg = (*ptrf)(a,b,c); // function calling using pointer
    cout << " a = " << a << endl;
    cout << " b = " << b << endl;
    cout << " c = " << c << endl;
    cout << " Average = " << avg << endl;
    return 0;
}
 oat average ( oat x,  oat y,  oat z)
{
     oat temp;
    temp = (x+y+z)/3.0;
    return(temp);
}
```

Output of the above program

```
enter three numbers
1  2  3

 a = 1
 b = 2
 c = 3
 Average = 2
```

PROGRAM 8.20

A program to demonstrate how a pointer to a function is declared to perform simple arithmetic operations such as addition, subtraction, multiplication and division of two numbers.

```
//pointers to functions 2.cpp
#include <iostream>
using namespace std;
int main ()
{
     oat add ( oat, oat); //function declaration
     oat sub ( oat, oat);
     oat mul ( oat, oat);
     oat div ( oat, oat);
```

```
     oat (*ptradd)( oat, oat);// pointer to function declaration
     oat (*ptrsub)( oat, oat);
     oat (*ptrmul)( oat, oat);
     oat (*ptrdiv)( oat, oat);
    void menu(void);
     oat a,b,value;
    char ch;
    ptradd = &add;
    ptrsub = &sub;
    ptrmul = &mul;
    ptrdiv = &div;
    cout << " demonstration of pointer to functions \n";
    cout << " enter any two numbers \n";
    cin >>a >> b ;
    cout << " a = " << a << endl;
    cout << " b = " << b << endl;
    menu();
    while (( ch = cin.get()) != 'q') {
       switch (ch) {
          case 'a' :
              value = (*ptradd)(a,b);
              cout << " Addition of two numbers = " << value ;
              cout << endl;
              break;
          case 's' :
              value = (*ptrsub)(a,b);
              cout << " Subtraction of two numbers = "<< value;
              cout << endl;
              break;
          case 'm' :
              value = (*ptrmul)(a,b);
              cout << " Multiplication of two numbers = "<< value;
              cout << endl;
              break;
          case 'd' :
              value = (*ptrdiv)(a,b);
              cout << " Division of two numbers = " << value;
              cout << endl;
              break;
        } // end of switch-case statement
    }    // end of while loop
    return 0;
}

void menu()
{
    cout << " a -> addition " << endl;
    cout << " s -> subtraction " << endl;
    cout << " m -> multiplication " << endl;
    cout << " d -> division " << endl;
    cout << " q -> quit " << endl;
    cout << " option,please ? \n";
}

 oat add ( oat x,  oat y)
{
    return(x+y);
}

 oat sub ( oat x,  oat y)
{
    return(x-y);
}

 oat mul ( oat x,  oat y)
{
```

```
    return(x*y);
}
 oat div ( oat x,  oat y)
{
    return(x/y);
}
```

Output of the above program

```
demonstration of pointer to functions
enter any two numbers
10 20

a = 10
b = 20
a -> addition
s -> subtraction
m -> multiplication
d -> division
q -> quit
option, please?
a
Addition of two numbers = 30
s
Subtraction of two numbers = -10
m
Multiplication of two numbers = 200
d
Division of two numbers = 0.5
q
```

8.5 PASSING A FUNCTION TO ANOTHER FUNCTION

In this section, how a function can be passed as a formal argument to another function is described using a pointer technique. C++ allows a pointer to pass one function to another as an argument. Passing a function to another function as a function parameter is one of the major attractions and strengths of the C++ language. The general syntax of passing a function to another function is

```
return_type function_name ( pointer_to_function ( other arguments));
```

For example, the following declaration of passing a function to another function is valid.

```
 oat calculation (  oat (*) (  oat, oat), oat, oat);
```

where the function calculation returns a type `oat` and takes the formal argument of a pointer to another function and two other `oat` types. As a pointer to function declaration itself is a pointer data, it returns a type `oat` and takes a formal argument of two floating point values.

PROGRAM 8.21

A program to demonstrate how a function can be passed to another function as a formal argument. This program peforms addition and subtraction of two floating point numbers by another function which takes the formal arguments of the functions add (), sub () and returns the result.

```
//passing a function to another function
#include <iostream>
```

```
using namespace std;
int main ()
{
     oat add ( oat, oat); //function declaration
     oat sub ( oat, oat);
     oat action (  oat (*) (  oat, oat), oat, oat);
     oat (*ptrf)( oat, oat);// pointer to function declaration
     oat a,b,value;
    char ch;
    cout << " passing a function to another function\n";
    cout << " enter any two numbers \n";
    cin >> a >> b ;
    cout << " a -> addition " << endl;
    cout << " s -> subtraction " << endl;
    cout << " option,please ? \n";
    cin >> ch;
    if ( ch == 'a')
      ptrf = &add;
    else
      ptrf = &sub;
    cout << " a = " << a << endl;
    cout << " b = " << b << endl;
    value = action (ptrf,a,b);
    cout << " Answer = " << value << endl;
    return 0;
}

 oat add ( oat x,  oat y)
{
    oat ans;
   ans = x+y;
   return(ans);
}

 oat sub ( oat x,  oat y)
{
    oat ans;
   ans = x-y;
   return(ans);
}

 oat action ( oat (*ptrf)( oat, oat),  oat x,  oat y)
{
    oat answer;
   answer = (*ptrf)(x,y);
   return(answer);
}
```

Output of the above program

```
passing a function to another function
enter any two numbers
1   2
a -> addition
s -> subtraction
option, please?
a

a = 1
b = 2
Answer = 3
```

8.6 POINTERS AND ARRAYS

In C++, there is a close correspondence between array data type and pointers. An array name in C++ is very much like a pointer but there is a difference between them. The pointer is a variable that can appear on the left side of an assignment operator. The array name is a constant and cannot appear on the left side of an assignment operator. In all other respects, both the pointer and the array version are the same.

8.6.1 Pointer and One Dimensional Array

In C++, pointers and one-dimensional arrays have a close relationship. Consider the following valid declaration,

```
int value[20];
int *ptr;
```

where the array variable `value` is an array type, and the address of the first element can be declared as

`value[0]` - which holds the address of the zeroth element of the array value.

The pointer variable `ptr` is also an address so the declaration `value [0]` and `ptr` is same as both hold addresses. The following is a valid assi]gnment:

```
ptr = &value[0];
```

The address of the zeroth element is assigned to a pointer variable `ptr`. Figure 8.4 shows the relationship between the pointer and one-dimensional array.

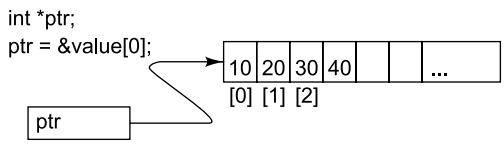


Fig. 8.4 *Pointer and One-dimensional Array*

If the pointer is incremented to the next data element, then the address of the incremented value of the pointer will be same as the value of the next element.

```
ptr++ == value[1];
```

For example, the following equalities are valid.

```
ptr+6 == &value[6]
*ptr == &value[0]
*ptr == value[]
*(ptr+6) == value[6]
ptr++   == &value[1]
```

For example,

```
int sarray[200];
int *sptr;
sptr = sarray;
```

The value sarray is converted to a pointer to the first element of the array. It is exactly as if it has been written

```
sptr = &sarray[0];
```

Secondly, array subscripting is defined in terms of pointer arithmetic. That is, the expression

```
a[i]
```

is defined to be the same as

```
*( (a) + (i) )
```

which is to say the same as

```
* ( &(a)[0] + (i))
```

PROGRAM 8.22

A program to display the content of an array using a pointer arithmetic (version 1).

```
// pointers and arrays
#include <iostream>
using namespace std;
int main()
{
    static int a[4] = { 11,12,1,14 };
    int i,n,temp;
    n = 4;
    cout << " Contents of the array "<<endl;
    for (i=0; i<= n-1; ++i) {
        temp = *((a) + (i));
        cout << " value = " <<  temp << endl;
    }
    return 0;
}
```

Output of the above program

```
Contents of the array
value = 11
value = 12
value = 13
value = 14
```

PROGRAM 8.23

A program to display the content of an array using a pointer arithmetic (version 2).

```
// version 2
#include <iostream>
using namespace std;
int main()
{
    static int a[4] = { 11,12,13,14 };
    int i,n,temp;
    n = 4;
    cout << " Contents of the array "<<endl;
    for (i=0; i<= n-1; ++i) {
       temp = *(&(a)[0] + (i));
       cout << " value = " << temp <<endl;
    }
    return 0;
}
```

Output of the above program

```
Contents of the array
value = 11
value = 12
value = 13
value = 14
```

8.6.2 Pointer and Multidimensional Array

A pointer to an array contains the address of the first element. In a one-dimensional array, the first element is &[0]. In a two-dimensional array, it is

```
&value[0][0]
```

For example,

```
int value[][];
int *ptr;
ptr = value;
```

The address of the zeroth row and the zeroth column of the two-dimensional array value is assigned to the pointer variable value. Figure 8.5 shows the relationship between the pointer and multi-dimensional array.

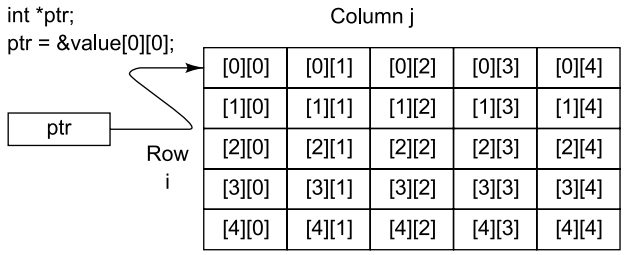


Fig. 8.5 *Pointer and Multi-dimensional Array*

Suppose, if `ptr++` is written, the pointer variable will be incremented to the next data element in the two-dimensional array that is equal to `value[0][1]` because a two-dimensional array is stored by rows.

So, the following equality is true.

```
ptr+1 == &value[0][1]
```

For example,

```
 oat value[20][30];
 oat *ptr1;
ptr1 = &value[0][0];   // pointer initialisation
ptr1+4  == &value[0][4];
ptr1+30 == &value[1][0];
```

The 30th element of the `value` is the first element in the second row because the counting starts at 0, so the 30th element is `value [1][0]`.

For example, if `s` is a 2 by 3 array as defined above, then the expression

`s[1][2]` is expressed as

`*(*(s+1) +2)` which is evaluated in the following order

```
s
s+1
*(s+1)
*(s+1)+2
*(*(s+1)+2)
```

PROGRAM 8.24

A program to display the contents of a two-dimensional array using a pointer arithmetic.

```
// pointers and multidimensional arrays
#include <iostream>
using namespace std;
int main()
{
    static int a[2][3] = {
                          {11,12,13},
                          {14,15,16}
                         };
    int *ptr;
    int i,j,n,m,temp;
    n = 2;
    m = 3;
    cout << " Contents of the array "<<endl;
    for (i = 0; i <= n-1; ++i) {
        for (j = 0; j<=m-1; ++j) {
            temp = *( *(a+i) + j );
            cout << temp << '\t';
        }
        cout << endl;
```

```
    }
    return 0;
}
```

Output of the above program

```
Contents of the array
11  12  13
14  15  16
```

8.7 ARRAYS OF POINTERS

Arrays of pointers are frequently used to access arrays of strings. The pointers may be arrayed like any other data type. The declaration for an integer pointer array of size 10 is

```
int *ptr[10];
```

makes

```
ptr[0],ptr[1],ptr[2]...ptr[9]
```

an array of pointers.

Figure 8.6 shows the representation of an array of pointers.

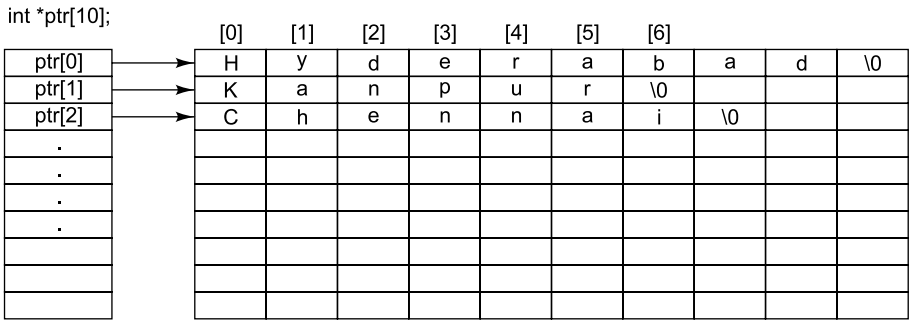


Fig. 8.6 *Array of Pointers*

For example, one can use

```
int a[10],b[10],c[20];
int *ptr[40];
```

where `ptr` is an array of pointers that can be used to point to the first elements of the arrays `a`, `b` and `c`.

The following are valid assignment statements in C++:

```
ptr[0] = &a[0];
ptr[9] = &b[0];
ptr[19] = &c[0];
```

The following declaration is valid in C++ for the two-dimensional arrays.

```
char *text [row][col];
```

for array of characters to pointers.

For example,

```
char *name[10][15];
name[1][] = "Hyderabad";
name[2][] = "Bangalore";
```

and so on.

PROGRAM 8.25

A program to display the contents of pointers using an array of pointers.

```
// arrays of pointers
#include <iostream>
using namespace std;
int main()
{
    char *ptr[];
    ptr[0] = "Bangalore";
    ptr[1] = "Hyderabad" ;
    ptr[2] = "Mumbai";
    cout << " Contents of pointer 1 = "    << ptr[0]<<endl;
    cout << " Contents of pointer 2 = "    << ptr[1]<<endl;
    cout << " Contents of pointer 3 = "    << ptr[2]<<endl;
    return 0;
}
```

Output of the above program

```
Contents of pointer 1 = Bangalore
Contents of pointer 2 = Hyderabad
Contents of pointer 3 = Mumbai
```

8.8 POINTERS AND STRINGS

This section deals with one of the important applications of pointers with character arrays. This section also covers how to read and write a string; how to perform string length, string copy and string concatenate etc., using a pointer technique. As a matter of fact that many string operations in C++ are usually implemented, realised and performed using pointers to character arrays and pointer arithmetic.

(a) Introduction A string is an array of characters terminated by a null character. A null character is a character with a numeric value of zero. It is represented in C++ by the escape sequence '\0'. A string constant, sometimes called a string literal, is any series of characters enclosed in double quotes. Figure 8.7 shows the storage pattern of pointer and character array.

It has a data type of array of `char` and each character is the string takes up one byte. In addition, the compile automatically appends a null character to designate the end of the string.

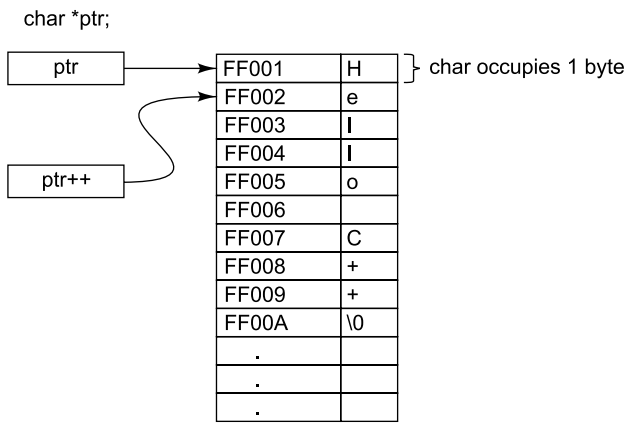


Fig. 8.7 *Pointer and Character Array*

(b) Declaring and Initialising Strings To store a string in memory, one needs to declare an array of type `char`. One may initialise an array of chars with a string literals. One must allocate enough characters to hold the string if one wants to specify an array size for declaring string data type.

For example,

```
static char str[] = "this text";
```

The array is one element longer than the number of characters in the string to accommodate the trailing null character. str [], therefore, is ten characters in length.

If the following example, for instance, the first four elements are initialised with the characters 'y','e','s' and '\0'. The remaining elements receive the default initial value of zero.

```
static char str[10] = "yes";
```

The following statement, however, is illegal due to insufficient memory space to accommodate all elements.

```
static char str[3] = "four"; // error
```

(c) String Assignments C++ treats string literals like other arrays and that it interprets a string constant as a pointer to the first character of the string. This means that one can assign a string constant to a pointer that point to a char. The following program illustrates how to assign and process the string literals with pointers.

```
#include <iostream>
using namespace std;
int main()
{
     char *ptr = "this";
     cout << "string \n";
     while (*ptr != '\0') {
       cout << *ptr;
       ptr++;
     }
     return 0;
}
```

Output of the above program

```
string
this
```

(d) String vs Chars It is important to recognise the difference between string constants and character constants. In the following two declarations, one byte is allocated for char and two bytes are allocated for the string "a", (an extra byte for the terminating null character), plus additional memory is allocated for the pointer ptr.

```
char ch = 'a';  // one byte is allocated for 'a'
char *ps = "a";
```

In the above declaration, two bytes are allocated for "a", plus an implementation defined number of bytes are allocated for the pointer ps.

Figure 8.8 shows the internal storage representation of a character and string data type.

Fig. 8.8 *Difference between a Character and String*

It is illegal to assign a character constant through a dereferenced pointer:

```
*ptr = 'a';  //  invalid
```

But it is correct to assign a string to a dereferenced char pointer:

```
*ptr = "a";   // valid
```

Since a string is intialised as a pointer to a char and a dereferenced pointer has the type of the object that it points to assign a pointer value to a char variable. The following program illustrates how to define, declare and realise the character assignment and string assignment with pointer.

```
#include <iostream>
```

```
using namespace std;
int main()
{
    char ptr1 = 'a';
    char *ptr2 = "a";
    cout << " character = " << ptr1 << endl;
    cout << " string = " << *ptr2 << endl;
    return 0;
}
```

Output of the above program

```
character = a
string = a
```

It is illegal to assign a string to a pointer (without dereferencing it).

```
char ptr2 = "a";  //invalid
```

It is also correct to assign a character constant to a pointer.

```
char *ptr1 = 'a';  //invalid
```

The following program displays error message during compile time for converting `char` to ponter *.

```
#include <iostream>
using namespace std;
int main()
{
    char *ptr1 = 'a';   //error,   ptr1 = 'a'; is right
    char ptr2 = "a";   //error, *ptr2 = "a";  is right
    cout << " character = " << ptr1 << endl;
    cout << " string = " << ptr2 << endl;
    return 0;
}
```

(e) Pointers and String Operations Many string operations in C++ are usually performed by using pointers to the array and then using pointer arithmetic. As strings tend to be accessed strictly in sequential order, pointers use the obvious choice. Strings are one-dimensional arrays of type char. In C++, a string is terminated by null character or '\0'. String constants are written in double quotes.

For example,

`"this"` is a character string of 5 and the last element being the null character `'\0'`. Therefore, a string constant such as `"a"` is not the same as the character constant `'a'`. Many string operations such as string length, string compare, string copy, string concatenate, etc. are performed using pointer method.

PROGRAM 8.26

A program to read a string from the stdin and to display it in the video screen using a pointer technique.

```
#include <iostream>
using namespace std;
int main()
{
    void display(char *str);
    char str[80],ch;
    int i = 0;
    cout << "enter a line of text \n";
    while ((ch = cin.get()) != '\n')
       str[i++] = ch;
```

```
        str[i++] = '\0';
        cout<<"Typed string is :\n";
        display(str);
        cout<< endl;
        return 0;
   }
   void display(char *ptr)
   {
      while (*ptr != '\0') {
          cout.put(*ptr);
          ptr++;
      }
   }
```

Output of the above program

```
enter a line of text
this is a test program by Ravich
Typed string is :
this is a test program by Ravich
```

PROGRAM 8.27

A program to find the number of characters in a given string using a pointer method.

```
// string length version 1
#include <iostream>
using namespace std;
int main()
{
     void display(char *str);
     int stringlength(char *str);
     char str[80],ch;
     int i = 0,nch;
     cout << "enter a line of text \n";
     while ((ch = cin.get()) != '\n')
         str[i++] = ch;
     str[i++] = '\0';
     cout<<"Typed string is :\n";
     display(str);
     nch = stringlength(str);
     cout<<"\n";
     cout<<"Number of Characters :" << nch << endl;
     return 0;
}
void display(char *ptr)
{
     while (*ptr != '\0') {
        cout.put(*ptr);
        ptr++;
      }
}
int stringlength (char *ptr)
{
     int i = 0;
     while (*ptr != '\0'){
         i++;
         ptr++;
     }
     return(i);
}
```

Output of the above program

```
enter a line of text
this is a test
Typed string is:
this is a test
Number of Characters: 14
```

The above string length function can be modified in the following form:

```
// string length version 2
int stringlength (char *ptr)
{
    char *source = ptr;
    while (*ptr != '\0')
       ptr++;
    return(ptr-source);
}
```

PROGRAM 8.28

A program to copy the contents of one string to another string using a pointer method.

```
// string copy version 1
#include <iostream>
using namespace std;
int main()
{
    void display(char *str);
    char *stringcopy(char *dest, char *source);
    char source[80],dest[80],ch;
    int i = 0;
    cout << "enter a line of text \n";
    while ((ch = cin.get()) != '\n')
        source[i++] = ch;
    source[i++] = '\0';
    cout<<"Contents of the source_ptr:\n";
    display(source);
    stringcopy(dest,source);
    cout<<"\nContents of the dest_ptr :\n";
    display(dest);
    cout << "\n";
    return 0;
}

void display(char *ptr)
{
    while (*ptr != '\0') {
        cout.put(*ptr);
        ptr++;
    }
}

char *stringcopy (char *dest_ptr,char *source_ptr)
{
    while (*source_ptr != '\0'){
      *dest_ptr = *source_ptr;
       dest_ptr++;
       source_ptr++;
    }
    *dest_ptr++ = '\0';
    return (dest_ptr);
}
```

Output of the above program

```
enter a line of text
this is a test string
Contents of the source_ptr:
this is a test string
Contents of the dest_ptr:
this is a test string
```

The above string copy function can be modified in the following form:

```
// string copy version 2
char *stringcopy (char *dest_ptr,char *source_ptr)
{
    while (*source_ptr != '\0'){
      *dest_ptr++ = *source_ptr++;
    }
    *dest_ptr++ = '\0';
    return (dest_ptr);
}
```

The string copy function may be written in the following compact form also.

```
// string copy version 3
char *stringcopy (char *dest_ptr,char *source_ptr)
{
    while ((*dest_ptr++ = *source_ptr++) != '\0');
    *dest_ptr++ = '\0';
    return (dest_ptr);
}
```

PROGRAM 8.29

A program to concatenate the given two strings into a one string using a pointer method.

```
// string concatenate version 1
#include <iostream>
using namespace std;
int main()
{
    void display(char *str);
    char *strconcatenate (char *dest, char *source1,
            char *source2);
    char source1[80],source2[80],dest[280],ch;
    int i,j;
    i = 0;
    cout << "enter a line of text \n";
    while ((ch = cin.get()) != '\n')
        source1[i++] = ch;
    source1[i++] = '\0';
    i = 0;
    cout << "enter another line of text \n";
    while ((ch = cin.get()) != '\n')
        source2[i++] = ch;
    source2[i++] = '\0';
    cout<<"\n Contents of the source_ptr1:\n";
    display(source1);
    cout<<"\n Contents of the source_ptr2:\n";
    display(source2);
    strconcatenate(dest,source1,source2);
    cout<<"\n Contents of the dest_ptr :\n";
    display(dest);
```

```
    cout << "\n";
    return 0;
}

void display(char *ptr)
{
    while (*ptr != '\0') {
        cout.put(*ptr);
        ptr++;
    }
}

char *strconcatenate (char *dest_ptr, char *source1,char *source2)
{
    while (*source1 != '\0') {
       *dest_ptr = *source1;
       dest_ptr++;
       source1++;
    }
    *dest_ptr++ = ' ';
    while (*source2 != '\0') {
       *dest_ptr = *source2;
       dest_ptr++;
       source2++;
    }
    *dest_ptr++ = '\0';
   return (dest_ptr);
}
```

Output of the above program

```
enter a line of text
this is a test

enter another line of text
program by Ravich

Contents of the source_ptr1:
this is a test

Contents of the source_ptr2:
program by Ravich

Contents of the dest_ptr:
this is a test program by Ravich
```

The above string concatenate function can be modified in the following form:

```
// string concatenate version 2
char *strconcatenate (char *dest_ptr, char *source1,char *source2)
{
     while (*source1 != '\0') {
              *dest_ptr++ = *source1++;
     }
     *dest_ptr++ = ' ';
     while (*source2 != '\0') {
              *dest_ptr++ = *source2++;
     }
     *dest_ptr++ = '\0';
     return (dest_ptr);
}
```

The string concatenate function may be written in the following compact form also.

```
// string concatenate version 3
char *strconcatenate (char *dest_ptr, char *source1,char *source2)
{
     while ((*dest_ptr++ = *source1++) != '\0');
         *dest_ptr--;
         *dest_ptr++ = ' ';
     while ((*dest_ptr++ = *source2++) != '\0');
         *dest_ptr++ = '\0';
     return (dest_ptr);
}
```

In addition to the string functions used in the previous examples, there are many others in the standard library:

	String functions in the standard library
`strcpy()`	copies a string to an array
`strncpy()`	copies of a portion of a string to an array
`strcat()`	appends one string to another
`strncat()`	copies a portion of one string to another
`strcmp()`	compares two strings
`strncmp()`	compares two strings up to a specified number of characters
`strchr()`	finds the first occurrence of a specified character in a string
`strcoll()`	compares two strings based on an implementation defined collating sequence.
`strcspn()`	computes the length of a string that does not contain specified characters
`strerror()`	maps an error number with a textual error message
`strlen()`	computes the length of a string
`strpbrk()`	finds the first occurrence of any specified characters in a string
`strrchr()`	finds the last occurrence of any specified characters in a string
`strspn()`	computes the length of a string that contains only specified character.
`strstr()`	finds the first occurrence of one string embedded in another
`strtok()`	breaks a string into a sequence of tokens
`strxfrm()`	transforms a string so that it is as suitable as an argument to strcmp().

8.9 POINTERS TO POINTERS

An array of pointers is the same as pointers to pointers. As an array of pointers is easy to index because the indices themselves convey the meaning of a class of pointers. However, pointers to pointers can be confusing. The pointer to a pointer is a form of multiple of indirections or a class of pointers.

In the case of a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the variable that contain the values desired. To declare a pointer to a pointer, precede the variable name with two successive asterisks. Multiple indirections can be carried on to whatever extent desired, but there are a few cases where more pointer to a pointer is needed or written. A pointer to a pointer is a construct used frequently in sophisticated programs.

Excess indirection is difficult to follow and process as it leads to conceptual errors.

```
pointer ---------→ variable

pointer ---------→ pointer ---------→ variable
```

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional * in front of the variable name.

For example,

```
int **ptr;
```

declares `ptr` to be a pointer to a pointer to an `int`. To dereference the pointer and access the `int`, one has to use two asterisks. For example,

```
j = **ptr;  // assigns an integer to j
```

Consider the following series of declarations:

```
int value = 5;
int *ptr = &value;
int **ptr_to_ptr = &ptr;
```

These declarations result in the storage pattern is shown in Fig. 8.9.

Both `ptr` and `ptr_to_ptr` are pointers but the `ptr` contains the address of an `int`, whereas `ptr_to_ptr` contains the address of a pointer to an `int`.

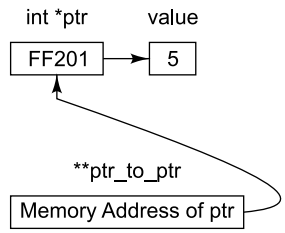


Fig. 8.9 *Storage pattern of pointer to pointer*

PROGRAM 8.30

A program to declare the pointer to pointer variable and display the contents of these pointers.

```
// pointer to pointer 1.cpp
#include <iostream>
using namespace std;
int main()
{

    int value;
    int *ptr1;
    int **ptr2;
    value  = 120;
    cout << " value   " << value << endl;
    ptr1 = &value;
    ptr2 = &ptr1;
    cout << "pointer 1 = " << *ptr1 <<endl;
    cout << "pointer 2 = " << **ptr2 <<endl;
    return 0;
}
```

Output of the above program

```
value   120
pointer 1 = 120
pointer 2 = 120
```

PROGRAM 8.31

A program to declare the pointer to a pointer to a pointer variable and display the contents of these pointers.

```
// pointer to pointer to pointer
#include <iostream>
using namespace std;
int main()
{
    int *ptr1;
    int **ptr2;
    int ***ptr3;
    int data;
    data = 100;
```

```
    ptr1 = &data;
    ptr2 = &ptr1;
    ptr3 = &ptr2;
    cout << "\n contents of ptr1 = " << *ptr1;
    cout << "\n contents of ptr2 = " << **ptr2;
    cout << "\n contents of ptr3 = " << ***ptr3;
    return 0;
}
```

Output of the above program

```
contents of ptr1 = 100
contents of ptr2 = 100
contents of ptr3 = 100
```

8.10 DECIPHERING COMPLEX DECLARATIONS

This section explains how to decode and figure out the complex pointer declarations. It is well known that C++ allows us to declare arbitrarily complex data types, including types such as arrays of pointers to functions and functions that return pointers to functions. These type declarations appear in numerous places, including variable declarations, function headers and type casts.

8.10.1 Simple Data Type Declarations

It is easy to figure out and decipher the meaning if we declare a variable with one of the basic types.

```
int    i;     // an int
double  d;    // a double
```

We can declare several more complicated type by combining the identifier with a single `*`, `()` and `[]`. Prefacing the identifier with * declares a pointer to the base type.

```
int  *ptr;   // pointer to an int
char *cptr;  // pointer to a char
```

Following the identifier with [] declares an array of the base type.

```
double dtab[10];  // array of 10 doubles
char ctab[10];    // array of 10 characters
```

We need to include a subscript only if we are defining storage and we are not following the declaration with an initialisation expression.

Following the identifier with a `()`, possibly with an enclosed list of types, declares a function.

```
int  ifunct(void);  // function returning an int
double  dfunct(void); // function returning a double
```

Here, we are assuming we want to declare functions that take no parameters. Otherwise, we would replace the `void` with a list of type declarations for these parameters.

8.10.2 Complex Pointer Declarations

We declare even more complex types, such as pointers to functions and array of pointers, by combining the pieces used to form the previous declarators. However, one has to take care of the operator precedence.

The best strategy for deciphering a declaration is to start with the variable name by itself and then each part of the declaration, starting with the operators that are closest to the variable name. In the absence of parentheses to affect binding, one would add all of the function and array operators on the right side of the variable name first (since they have higher precedence) and then add the pointer operators on the left side.

(1) Array of Pointers Consider the following declaration,

(a) `char *ptr[];`

would be deciphered through the following steps:

```
ptr[]          –  is an array
*ptr[]         –  is an array of pointers
char *ptr[]    –  is an array of pointers to chars.
```

(b) `int *ptr[10];`

would be figure out in the following form:

```
ptr[10]        –  is an array of 10 elements
*ptr[10]       –  is an array of 10 elements of pointers
                  or (is an array of 10 pointers)
int *ptr[10]   –  is an array of 10 elements of pointers
                  to int or (is an array of 10 pointers to int)
```

The simplest combinations are an array of pointers:

```
double *iptr[10];   // array of 10 pointers to double
char *dptr[10];     // array of 10 pointers to char
```

(2) Function Returning a Pointer Consider the following declaration,

(a) `int *ptrfunct();`

would be deciphered through the following steps:

```
ptrfunct()          –  is a function
*ptrfunct()         –  is a function returning a pointer
int *ptrfunct()     –  is a function returning a pointer to int
```

(b) `char *abc(void);`

would be figure out in the following form:

```
abc(void)           –  is a function taking void data type
*abc(void)          –  is a function taking void data type and returning a pointer
char *abc(void)     –  is a function taking void data type and returning a pointer to char.
```

(3) Pointer to an Array Consider the following pointer to an array declaration,

(a) `char (*ptr)[];`

When we combine declarators, `*` has lower precedence than either `()` or `[]`. Parentheses must be used to override the normal precedence.

```
(*ptr)          –  is a pointer to
(*ptr)[]        –  is a pointer to an array
char (*ptr)[]   –  is a pointer to an array of characters
```

(b) `int (*ptr)[10];`

would be figure out in the following form:

```
(*ptr)          –  is a pointer to
(*ptr)[10]      –  is a pointer to an array of 10 elements
int (*ptr)[10]  –  is a pointer to an array of 10 elements of integer data types or (pointer to an array
                   of 10 int)
```

(4) Pointer to a Function Consider the following a pointer to a function declaration,

(a) `int (*ptr)();`

would be deciphered through the following steps:

```
(*ptr)          –  is a pointer to
(*ptr)()        –  is a pointer to a function
int (*ptr)()    –  is a pointer to a function returning int
```

(b) `void (*x) (void);`

would be figure out in the following form:

```
(*x)              – is a pointer to
(*x)(void)        – is a pointer to a function taking void data type
void (*x)(void)   – is a pointer to a function taking void data type and returning void data
```

(5) Array of Pointers to Functions Consider the following an array of pointers to a function declaration. Parentheses can be used to change the precedence order.

(a) `int (*ptr[]) ();`

would be decomposed as follows

```
ptr[]             – is an array
*ptr[]            – is an array of pointers
(*ptr[])()        – is an array of pointers to functions
int (*ptr[])()    – is an array of pointers to functions returning int.
```

If this declaration had been written without the parentheses as

```
int *ptr[]();
```

it would have been translated as an array of functions returning pointers to int, which is illegal declaration since array of functions are invalid.

(b) `void (*table[]) (void);`

would be deciphered through the following steps:

```
table[]                  – is an array
*table[]                 – is an array of pointers
(*table[]) (void)        – is an array of pointers to a function taking void data type
void (*table[]) (void)   – is an array of pointers to a function taking void data type and
                           returning void data.
```

REVIEW QUESTIONS

1. What is a pointer? What are the uses of pointers in C++?
2. How is a pointer variable different from an ordinary variable?
3. What is meant by the address of a memory cell?
4. What is meant by address operator?
5. What are the scope rules of a pointer variable?
6. What is the use of an indirection operator?
7. How is a pointer variable declared in C++?
8. What is the relationship between a pointer and an array?
9. How can an indirection operator be used to access a multidimensional array?
10. Explain how a portion of an array can be passed onto a function.
11. How can a one-dimensional array of pointers be used to represent a collection of strings?
12. Under what conditions two pointer variables can be added?
13. Under what conditions two pointer variables can be subtracted?
14. Under what conditions two pointer variables can be compared?
15. What is the relationship between a pointer and a function name?
16. What is meant by pointer to pointer? What is its advantage?
17. What is meant by the call by value and call by reference?
18. What are the advantages of declaring a pointer variable in a function declaration?
19. Under what conditions the call by reference is preferred than the call by value?

20. What is meant by pointers to pointers?
21. What is an array of pointers?
22. What is the difference between the array of pointers and pointer to the array?
23. Explain how a pointer to function can be declared in C++.
24. What is meant by passing a function to another function as an argument?

CONCEPT REVIEW PROBLEMS

1. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10;
    int *ptr1;
    ptr1 = &a;
    cout << " ++(*ptr1) = " << ++(*ptr1) <<"\n";
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10;
    int *ptr1;
    ptr1 = &a;
    cout << " (*ptr1)++ = " << (*ptr1)++ <<"\n";
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10;
    int *ptr1;
    ptr1 = &a;
    cout << " *ptr1++ = " << *ptr1++ <<"\n";
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10;
    int *ptr1;
    ptr1 = &a;
```

```
    cout << " ++*ptr1 = " << ++*ptr1 <<"\n";
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10;
    int *ptr1;
    ptr1 = &(++a);
    cout << " *ptr1 = " << *ptr1 <<"\n";
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10;
    int *ptr1;
    ptr1 = &(a++);
    cout << " *ptr1 = " << *ptr1 <<"\n";
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
int main()
{
    bool  ag = false;
    bool *ptr1;
    ptr1 = & ag;
    cout << " *ptr1 = " << *ptr1 <<"\n";
    return 0;
}
```

(h)

```
#include <iostream>
using namespace std;
int main()
{
    bool  ag = true;
    bool *ptr1;
    ptr1 = & ag;
    cout << " *ptr1 = " << *ptr1 <<"\n";
    return 0;
}
```

(i)

```
#include <iostream>
using namespace std;
int main()
{
    void display (int *abc);
```

```
    int a = 10, *ptr;
    ptr = &a;
    display(ptr);
    return 0;
}
void display(int *abc)
{
    cout <<"*ptr = " << *abc;
    cout << "\n";
}
```

2. What will be the output of each of the following program when it is executed?

(a)

```
#include <iostream>
using namespace std;
int main()
{
    void display (int *abc);
    int a = 10, *ptr;
    ptr = &a;
    cout << *ptr << '\t';
    display(ptr);
    cout << *ptr << endl;
    return 0;
}
void display(int *abc)
{
    cout << (*abc)++ << '\t';
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    void display (int *abc);
    int a = 10, *ptr;
    ptr = &a;
    cout << *ptr << '\t';
    display(ptr);
    cout << *ptr << endl;
    return 0;
}
void display(int *abc)
{
    cout << ++(*abc) << '\t';
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    void display (int *);
    int a = 10, *ptr;
    ptr = &a;
    cout << *ptr << '\t';
```

```
    display(ptr);
    cout << *ptr << '\t';
    return 0;
}
void display(int *abc)
{
    void display2(int *);
    cout << ++(*abc) << '\t';
    display2(abc);
}

void display2(int *a)
{
    cout << ++(*a) << '\t';
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    void display (int *);
    int a = 10, *ptr;
    ptr = &a;
    cout << *ptr << '\t';
    display(ptr);
    cout << *ptr << '\t';
    return 0;
}
void display(int *abc)
{
    void display2(int *);
    cout << ++(*abc) << '\t';
    display2(abc);
    cout << *abc << '\t';
}

void display2(int *a)
{
    cout << ++(*a) << '\t';
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    void display (int *);
    int a = 10, *ptr;
    ptr = &a;
    cout << *ptr << '\t';
    display(ptr);
    cout << *ptr << '\t';
    return 0;
}
void display(int *abc)
{
```

```
    void display2(int *);
    cout << ++(*abc) << '\t';
    display2(abc);
    cout << *abc << '\t';
}
void display2(int *a)
{
    cout << (*a)++ << '\t';
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    void display (int *);
    int a = 10, *ptr;
    ptr = &a;
    cout << *ptr << '\t';
    display(ptr);
    cout << *ptr << '\t';
    return 0;
}
void display(int *abc)
{
    void display2(int *);
    cout << ++(*abc) << '\t';
    display2(abc);
    cout << *abc << '\t';
}

void display2(int *a)
{
    cout << —(*a) << '\t';
}
```

(g)

```
#include <iostream>
using namespace std;
int main()
{
    void display(int *ptr);
    const int x = 10;
    cout << " x = " << x << endl;
    display(&x);
    cout << " x = " << x << endl;
    return 0;
}
void display(int *ptr)
{
    cout << " x = " << *ptr << endl;
}
```

3. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
```

```
using namespace std;
int main()
{
    static char *ptr[4];
    ptr[0] = "Hyderabad";
    ptr[1] = "Mumbai";
    ptr[2] = "Chennai";
    ptr[3] = "New Delhi";
    cout << ptr[3] << endl;
    cout << ptr[2] << endl;
    cout << ptr[1] << endl;
    cout << ptr[0] << endl;
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    static char *ptr[4];
    ptr[0] = "Hyderabad";
    ptr[1] = "Mumbai";
    ptr[2] = "Chennai";
    ptr[3] = "New Delhi";
    for (int i = 0; i <= 3; ++i)
      cout << ptr[i] << endl;
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    int *ptr1;
    int **ptr2;
    int data;
    data = 100;
    ptr1 = &(++data);
    ptr2 = &ptr1;
    cout << "contents of ptr1 = " << *ptr1;
    cout << "\n contents of ptr2 = " << **ptr2;
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    int *ptr1;
    int **ptr2;
    int ***ptr3;
    int data;
    data = 10;
    ptr1 = &data;
```

```
    ptr2 = &ptr1;
    ptr3 = &ptr2;
    cout << "\n contents of ptr1 = " << *ptr1;
    cout << "\n contents of ptr2 = " << **ptr2;
    cout << "\n contents of ptr3 = " << ***ptr3;
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 2, b = 3, c = 5;
    int *ptr;
    cout << a << '\t' << b << '\t' << c;
    cout << endl;
    ptr = &a;
    *ptr = 10;
    ptr = &b;
    *ptr = 20;
    ptr = &c;
    *ptr = 30;
    cout << a << '\t' << b << '\t' << c;
    cout << endl;
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    int abc = 10;
    int *ptr = &abc;
    cout << "*ptr = " << *ptr << endl;
    ++abc;
    cout << "*ptr = " << *ptr << endl;
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
int main()
{
    int abc = 10;
    int *ptr1, *ptr2;
    ptr1 = &abc;
    cout << "*ptr1 = " << *ptr1 << endl;
    ptr2 = ptr1;
    ++abc;
    cout << "*ptr1 = " << *ptr1 << endl;
    cout << "*ptr2 = " << *ptr2 << endl;
    return 0;
}
```

PROGRAMMING EXERCISES

1. Write a program in C++ to read a set of characters using a pointer and to print in the reverse order.
   Input : ravic
   Output : civar
2. Write a program in C++ to find a given string in a line of text using a pointer.
3. Write a program in C++ to compare the two given strings using a pointer.
4. Write a program in C++ to check whether a given string is a palindrome or not using the pointer method.
   (Note that a palindrome is a string that reads the same both forward and backward.
   For example, madam, radar, malayalam, otto, 12321, etc.)
5. Write a program in C++ to find the number of words in a set of lines using a pointer.
6. Write a program in C++ to sort out a set of names in the alphabetical order using pointer technique.
7. Write a program in C++ to find the number of vowels in each word of a given text using a pointer.
8. Write a program to read a set of lines from stdin; store them in an array A; again read a string S from the stdin and check whether the given string S is in the array A. If it is, then print that line and also how many times it repeats in the array A using pointer method.
9. Write a program to read a set of lines from stdin; store them in an array A; again read a string S from the stdin and check whether the given string S is in the array A. If is, then remove the string S from the array A and print the updated array on the stdout using pointer. For example,
   A = concatenate
   S = cat
   The updated A is concotenate
10. Write a program to read a set of lines from stdin and store them in an array A; again read strings S1 and S2 from the stdin and check whether the given string S1 is in the array A. If it is, then replace the string S1 with string S2 and print the updated array using pointer. For example,
    A = concatenate
    S1 = cat
    S2 = 123
    The updated A is con123enate

Structures, Unions and Bit Fields

Chapter 9

This chapter presents how to declare, define and use a structure data type which is one of the salient features of the C++ language. This chapter also explains how to realise the various topics of the structures such as arrays of structures, arrays within a structure, structure within a structure with suitable illustrations. This chapter also deals with the unions, bit fields and enumerated data types.

9.1 INTRODUCTION

It is well known that C++ supports a wide variety of data types like simple, standard and structured types. This is one of the strengths of the C++ language. In Chapter 7, arrays of one-dimensional and multidimensional of the structured data types have been introduced. In this chapter we discuss about one more data type known as a structure which is a group of variables placed under a common name. A structure has many fields and each field may be of different data type like integer, floating point, character or string or even a structure. Normally, a structure is a heterogeneous data type whereas an array is a homogeneous data type.

In C++, collection of heterogeneous (different) types of data can be grouped to form a structure. When this is done, the entire collection can be referred by a structure name. In addition, the individual components which are called fields or members can be accessed and processed separately.

Array is a structured data type in which components are the same and identical in nature and each component can be accessed and processed individually only with the index value. On other hand, components of the structure are accessed and processed by its member name or field name. The elements or components of the arrays are homogeneous whereas structures are heterogeneous. The keyword `struct` is used to define a structure data type.

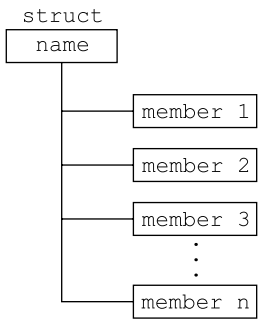
A structure data type also permits to declare one or more structures as a member of another structure. When a structure is declared as a member or field of another structure, it is called as a nested structure.

When an array is declared as a member of field of a structure, it is called an array within a structure. C++ also permits declaration of a structure of its array nature. Whenever a structure data type is defined as an array, it is called as array of structures. The various usages of the structure data type are elucidated in this chapter.

9.2 DECLARATION OF A STRUCTURE

It is well known that a structure consists of one or more fields of the same or different data types. Like any other variable, a structure variable must be defined before one attempts to use it in a program. Declaration of structure is one of the important steps for processing a structure data type in a program. Each field of a structure must have its own data type which can be one of either simple or structured data type.

The structure can be represented graphically as follows:



There are two important distinctions between arrays and structures. First, all the elements of an array must be of the same type. In a structure, the components or fields may have different data types. Secondly, a component of an array is referred by its position in an array, whereas each component of a structure has a unique name. Structures and arrays are similar in that both must be defined with a finite number of components.

The symbolic representation of the structure is:

```
struct user_de ned_name {
    member 1;
    member 2;
    -------
    -------
    member n;
};
```

A structure definition is specified by the keyword `struct`. This is followed by a user-defined name surrounded by braces, which describes the members of the structure. A member of a structure is a single unit.

The general format for a declaration of a structure is

```
storage_class struct  user_de ned_name {
      data_type    member 1;
      data_type    member 2;
      -------
      -------
      data_type    member n;
};
```

The storage class is optional, whereas the keyword `struct` and the braces are essential. The user-defined name is usually used, but there are situations in which it is not required. The data type and members

are any valid C++ data objects such as `short integer`, `oat` and `char`. The syntax diagram of struct declaration is given in Fig. 9.1

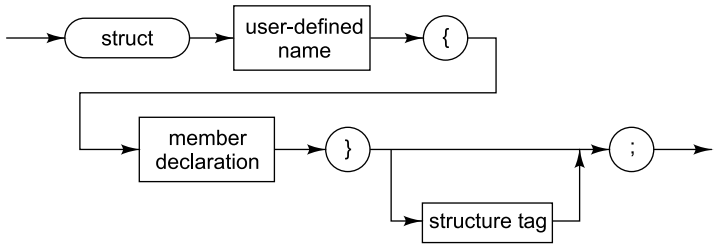


Fig. 9.1 *Syntax Diagram of Struct Declaration*

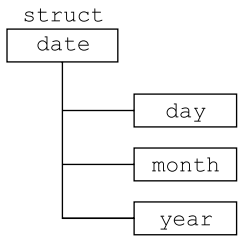
For example, the date of a day can be arranged in the following form:

```
int day;
int month;
int year;
```

The date is a structure consisting of three members and all the members are of the same data type.

```
struct date {
    int day;
    int month;
    int year;
};
```

The struct date can be represented symbolically as:



(2) Similarly, a student's particulars can be placed like this:

```
Roll no, age, sex, height and weight
```

The structure declaration for the above can be as follows:

```
struct student {
    int rollno;
    int age;
    int sex;
    int height;
    int weight;
};
```

Each member of a structure, is specified by a variable name with a period and the member name. The period is a structure member operator which we can be referred simply as the period operator. For example, the date is a structure consisting of three members which can be referred in program as:

```
struct date {
    int day;
```

```
    int month;
    int year;
};
int main()
{
    struct date today;
    today.day;
    today.month;
    today.year;
    -------
    -------
    return 0;
}
```

Assigning values to members of the structure can be done as follows:

```
today.day = 10;
today.month = 2;
today.year = 1993;
```

When a structure member is named using the period operator, the variable name and member name are still separate identifiers. For example, the variable '`today.year`' exceeds more than eight characters but in the structure, each one is a separate identifier so truncation will not take place for both as they are taken as a single variable.

9.3 PROCESSING WITH STRUCTURES

This section presents how to read, write and assign data to the structure variables. Each member of a structure is specified by the following variable name with a period and the member name. The period is a structure member operator which we shall hereafter refer to simply as the period operator.

To access a component in an array, subscript or an index is used. To access the field of a structure, structure variable name followed by the period and the field identifier of the structure is used. The general syntax of accessing the field name is:

```
structure_variable_name. eld_name  = variable;
```

For example, consider the following structure declaration:

```
#include <iostream>
using namespace std;
int main()
{
    struct student_info {
        int rollno;
        int age;
        int sex;
        int height;
        int weight;
    };
    student_info student; // creating a variable
    -------
    -------
    return 0;
}
```

The following manner one can access the individual field of a structure with period operator:

```
student.rollno
student.age
student.sex
student.height
student.weight
```

Field designators with simple types are treated just like ordinary variables. The following manner one can assign the values or data to the fields of a structure.

```
student.rollno  = 100;
student.age  = 13;
student.sex  = 'M';
student.height  = 167.77
student.weight  = 45.67
```

When a structure member is named using the period operator, the variable name and member name are still separate identifiers. For example, the variable '`student.rollno`' can exceed more than eight characters but in the structure, each one is a separate identifier and hence truncation will not take place treating both as a single variable.

(a) Reading and Writing of a Structure This section presents how to read and write a structure variable from standard input and output devices. For example, date is a structure consisting of three members which can be used to refer in a program as:

```
#include <iostream>
using namespace std;
int main()
{
    struct date {
        int day;
        int month;
        int year;
    };
    struct date today;
    -------
    -------
    return 0;
}
```

C++ compiler will not read or write an entire structure using a single command like this:

```
cin >> today;  // error
cout << today; // error
```

It will read or write the members of a structure separately as: To read a value for the fields of the structure date, one can use the cin function to get the input data from the keyboard in the following format:

```
// reading a structure
cin >> today.day;
cin >> today.month;
cin >> today.year;
```

Similarly, one can use the cout function to display the contents of a structure in the following form:

```
// displaying onto the video screen
cout << today.day;
cout << today.month;
cout << today.year;
```

PROGRAM 9.1

A program to demonstrate how to initialise the members of the structure and display the contents of the structure onto the video screen.

```
// example 9.1
#include <iostream>
using namespace std;
int main()
{
   struct date {
       int day;
       int month;
       int year;
    };
    struct date today;
    today.day = 10;
    today.month = 5;
    today.year = 2007;
    cout << "Today's date is :";
    cout <<today.day << "/" << today.month << "/" << today.year;
    return 0;
}
```

Output of the above program

```
Today's date is: 10/5/2007
```

(b) The Structure Tag It is possible to define a structure variable name in the structure type declaration itself. The general format of the structure tag declaration is:

```
storage_class struct  user_de ned_name {
      data_type   member 1;
      data_type   member 2;
      -------
      -------
      data_type   member n;
} variable 1, variable 2...variable n;
```

For example, the following structure declaration of `a,b` could be written as

```
/* structure tag declaration  */
   struct student {
       int rollno;
       int age;
       int sex;
       int height;
       int weight;
   } a,b;
```

PROGRAM 9.2

A program to assign some values to the member of a structure and to display the structure on the video screen using the structure tag.

```
#include <iostream>
using namespace std;
int main()
```

```
{
   struct sample {
       int x;
        oat y;
   } obj;
   obj.x = 20;
   obj.y = -23.44;
   cout <<"Contents of the structure \n";
   cout <<" x = " << obj.x << '\n';
   cout <<" y = " << obj.y << '\n';
   return 0;
}
```

Output of the above program

```
Contents of the structure
x = 20
y = -23.44
```

(c) Other Declaration A field of a structure is a unique name for the particular structure. The same field or member name can be given to the other structures also with different data types. C++ compiler will treat each structure member as a separate variable and reserves memory space according to the corresponding data types. For example, the following declaration is valid in C++, even though the same field name is used to represent the different data types in the two different structures.

For example, consider the following valid declaration:

```
struct  rst {
    int a;
     oat b;
    char c;
};
struct second {
    char a;
    int b;
     oat c;
};
```

In the above two structures, namely, the first and second have three members and all the three are of different data types but have the same names. It is advisable to use different names for the different structures so that no confusion arises while using them in a program. In this section, same member names have been used to explain the different usages and declaration in C++, whereas in a practical situation, the names of members should be distinct.

PROGRAM 9.3

A program to declare the same field name for the different data types in two structures; assign some values into the corresponding fields and to display the contents of the structure.

```
// example 9.3
// other structure declaration
#include <iostream>
using namespace std;
int main ()
{
    struct  rst {
        int a;
         oat b;
```

```
        char c;
    };
    struct second {
        char a;
        int b;
         oat c;
    };
    struct  rst one;
    struct second two;
    one.a = 23;
    one.b = 11.89;
    one.c = 'f';
    two.a = 'm';
    two.b = 5;
    two.c = -45.78;
    cout <<" contents of the  rst structure \n";
    cout <<" a = " << one.a <<" ,b = " << one.b << " ,c = " << one.c;
    cout <<"\n contents of the second structure \n";
    cout <<" a = " << two.a <<" ,b = " << two.b << " ,c = " << two.c;
    cout << endl;
    return 0;
}
```

Output of the above program

```
contents of the  rst structure
a = 23 ,b = 11.89 ,c = f
contents of the second structure
a = m ,b = 5 ,c = -45.78
```

In the above two structures, namely, `rst` and `second` are defined with three members and each member is unique of its data type. However, C++ compiler permits to declare the same field name for different structures. In a practical situation, the names of structure members should be distinct.

(d) Copying a Structure to Another If two structures are of the same data type, the value of one structure can be assigned as the value of another structure. For example, consider the following structure declaration:

```
struct student_info {
    int rollno;
    char sex;
    int age;
     oat height;
     oat weight;
};
struct student_info student1,student2;
```

Since the structure variables `student1` and `student2` are the data type of `student_info`, the following assignment statement is valid:

```
student2  = student1; /* valid */
```

For example, the following values are assigned to the individual members of the structure student1:

```
student1.rollno  = 101;
student1.sex  = 'M';
student1.age  = 23;
student1.height  = 123.45;
student1.weight  = 56;
```

then the student2 = student1 will be assigned and the contents of each of the individual fields of student1 will be copied to the student2.

```
student2.rollno  = 101;
student2.sex  = 'M';
student2.age  = 23;
student2.height  = 123.45;
student2.weight  = 56;
```

PROGRAM 9.4

A program to assign some values to the member of a structure and copy the contents of the one structure into another and display the contents of the structure on the video screen.

```
// copying a structure to another
#include <iostream>
using namespace std;
int main()
{
     struct student_info {
         int rollno;
          oat height;
          oat weight;
     };
     struct student_info studold,studnew;
     studold.rollno = 1001;
     studold.height = 174.67;
     studold.weight = 45;
     cout <<"Contents of the old structure \n";
     cout <<" Roll no = " << studold.rollno << '\n';
     cout <<" Height  = " << studold.height << '\n';
     cout <<" Weight  = " << studold.weight << '\n';
     studnew = studold; // copying the structure to another
     cout <<"\n Contents of the new structure \n";
     cout <<" Roll no = " << studnew.rollno << '\n';
     cout <<" Height  = " << studnew.height << '\n';
     cout <<" Weight  = " << studnew.weight << '\n';
     cout << endl;
     return 0;
}
```

Output of the above program

```
Contents of the old structure
Roll no = 1001
Height  = 174.67
Weight  = 45

Contents of the new structure
Roll no = 1001
Height  = 174.67
Weight  = 45
```

(e) Comparing Two Structures The members of a structure can appear in the condition part of `if` statement and `while` statement. All relational operators that are supported by C++ compiler are permitted to be used along with the members of structure. It is invalid to use one structure with another structure in a single statement. For example, consider the following structure type declaration:

```
struct date_info {
    int day;
    int month;
    int year;
};
struct date_info  date1,date2;
```

The following usage of relational operators are invalid:

(1)
```
if (date1 < date2)           // error
    cout << error \n";
```
(2)
```
while ( date1 != date2) {    // error
     -------
     -------
}
```
(3)
```
if ( date1.day <= date2)   // error
    cout << error \n";
```

To decide whether two structures are equal, it is necessary to compare the individual members of the structure. For example, to test equality of date1 and date2, then the suitable statement is:

```
if ( (date1.day == date2.day) &&
   (date1.month == date2.month) &&
   (date1.year == date2.year))
     cout << "date1 is equal to date2 \n";
else
     cout << "both structures are different \n";
```

The following conditional expressions using relational operators are valid:

(1)
```
if (date1.day <= date2.day)
     cout << "valid \n";
```
(2)
```
while (date1.month != date2.month) {
     -------
     -------
}
```
(3)
```
if ( date1.year >= date2.year )
    cout << "valid \n";
```

PROGRAM 9.5

A program to assign some values to the member of a structure and compare the contents of one structure into another and display the result of structure comparison on the video screen.

```
#include <iostream>
using namespace std;
int main()
{
    struct date {
       int day;
       int month;
       int year;
    };
    struct date day1,day2;
    day1.day = 10;
    day1.month = 5;
    day1.year = 2003;
    day2.day = 10;
    day2.month = 5;
    day2.year = 2003;
```

```
    cout <<" day1 : ";
    cout << day1.day << "/" << day1.month;
    cout << "/" << day1.year << '\n';
    cout <<" day2 : ";
    cout << day2.day << "/" << day2.month;
    cout << "/" << day2.year << '\n';
    if ((day2.day== day1.day) && (day2.month == day1.month) &&
         (day2.year == day1.year))
       cout <<"two structs consist of the same data\n";
    else
       cout <<"two structs are different \n";
    return 0;
}
```

Output of the above program

```
 day1 : 10/5/2003
 day2 : 10/5/2003
two structs consist of the same data
```

9.4 INITIALISATION OF STRUCTURE

A structure can be initialised in the same way as any other data type in C++. In keeping with the array analogy, a structure must be either static or external. For example,

```
students' particulars
    rollno
    age
    sex
    height
    weight
static struct student_info student = { 95001,24,'M',167.9,56.7 };
```

The C++ compiler assigns the following values to each of the fields:

```
Roll no  = 95001
Age      = 24
Sex      = M
Height   = 167.9
Weight   = 56.7
```

PROGRAM 9.6

A program to initialise the members of a structure and display the contents of the structure on the screen.

```
// structure initialization
#include <iostream>
using namespace std;
int main ()
{
    struct student_info {
        long int rollno;
        int age;
        char sex;
         oat height;
         oat weight;
    };
    struct student_info  student = { 20071,24,'M',167.9,56.7 };
    cout <<" Contents of structure \n";
```

```
    cout <<" Roll no  = " << student.rollno << '\n';
    cout <<" Age      = " << student.age << '\n';
    cout <<" Sex      = " << student.sex << '\n';
    cout <<" Height   = " << student.height << '\n';
    cout <<" Weight   = " << student.weight << '\n';
    return 0;
}
```

Output of the above program

```
Contents of structure
Roll no  = 20071
Age      = 24
Sex      = M
Height   = 167.9
Weight   = 56.7
```

The initialiser of a structure must be enclosed within a pair of braces. The constants to be assigned to the members of the structure must be in the same order in which the members are specified. There must be one to one correspondence between the members and the initialising values. If some of the structure members are not initialised, then the C++ compiler will automatically initialise them to zero. For example,

```
students' particulars
    rollno
    age
    sex
    height
    weight
  static struct  student_info student = {95001,24,'M'};
```

The C++ compiler assigns the following values to each of the fields:

```
Roll no  = 95001
Age      = 24
Sex      = M
Height   = 0
Weight   = 0
```

PROGRAM 9.7

A program to initialise some members of a structure and display the contents of the structure.

```
/*  some members of the structure are initialized
    and the rest of the members are
    initialized to zero by default */
#include <iostream>
using namespace std;
int main ()
{
    struct student_info {
        long int rollno;
        int age;
        char sex;
         oat height;
         oat weight;
    };
    struct student_info  student = { 20071,24,'M' };
    cout <<" Contents of structure \n";
    cout <<" Roll no  = " << student.rollno << '\n';
    cout <<" Age      = " << student.age << '\n';
```

```
    cout <<" Sex      = " << student.sex << '\n';
    cout <<" Height   = " << student.height << '\n';
    cout <<" Weight   = " << student.weight << '\n';
    return 0;
}
```

Output of the above program

```
Contents of structure
Roll no  = 20071
Age      = 24
Sex      = M
Height   = 0
Weight   = 0
```

In the above example, it has been declared that static student of structure has five fields such as rollno, age, sex, height and weight. The initial values are assigned only for the first three members of the structure so the C++ compiler will automatically assign zero to the rest of the members which are not initialised.

9.5 FUNCTIONS AND STRUCTURES

As discussed in Chapter 6 "Functions and Program Structures", a function is a very powerful technique to decompose a complex problem into separate manageable parts or modules. Each part is called a function and is very much used to convert a complicated program into a very simple one. As functions can be compiled separately, they can be tested individually and finally invoked into a main program as a whole.

A structure can be passed to a function as a single variable. The scope of a structure declaration should be an external storage class whenever a function in the main program is using a structure data types. The field or member data should be same throughout the program either in the main or in a function.

Designing and developing a subprogram with structure data type is one of the most important features of C++ programming as functions are a very powerful tool for designing a complicated problem in the easiest manner.

It is well known that there are two types of parameter passing techniques used in C++, namely, call by value and call by reference. C++ also permits to declare a function with structure data type either through call by value or through call by reference.

Types of Functions using Structured Data Type It is well known that C++ supports two types of calling a function from any part of a program: call by value and call by reference. When a call by reference is made, the indirection operator (*) must be prefixed along with the parameter list in order to indicate to the compiler that a variable parameter is being passed. By default, C++ takes a structure data type as call by value only.

The general syntax of the function using structure data type is:

```
return_type fname (struct struct_name parameters );
```

where `return_type` of a function may be one of the following namely, void, simple data type or structure data. The `fname` is a user-defined function identifier followed by a list of parameters of a structure data type.

(a) Call by Value with Structure Data Type Whenever a call by value is made, the values or contents of a structure will be copied into a function as dummy arguments and if any changes are made in the subprogram, that will not be reflected back to the calling portion of the program. The scope and visibility of the value parameters are limited only to the particular function block.

For example, the following program segment illustrates how call by value of a function with a structure data type can be realised.

```
#include <iostream>
using namespace std;
struct sample_info {
    int x;
     oat y;
};
struct sample_info  rst;
int main ()
{
    void display (struct sample_info out); // function declaration
    -------
    -------
    display (out); // function calling
    -------
    -------
}
void display (struct sample_info out) // function de nition
{
    -------
    -------
    out.x = 10;
    out.y = -20.20;
    -------
    -------
}
```

PROGRAM 9.8

A program to demonstrate how a call by value of the function with structure data type in its arguments is realised.

```
// passing a struct to a function
// using a call by value technique
#include <iostream>
using namespace std;
struct date {
    int day;
    int month;
    int year;
};
int main()
{
   struct date today;
   void display_newdate (struct date newdate);
   today.day = 10;
   today.month = 5;
   today.year = 2003;
   cout <<"Before a function call \n";
   cout <<"Today's date is :";
   cout << today.day << "/" << today.month << "/" << today.year;
   cout << endl;
   display_newdate(today);
   cout <<"After a function call \n";
   cout <<"Today's date is :";
   cout << today.day << "/" << today.month << "/" << today.year;
   cout << endl;
   return 0;
```

```
}
void display_newdate(struct date today)
{
   today.day++;
   today.month++;
   today.year++;
}
```

Output of the above program

```
Before a function call
Today's date is: 10/5/2003
After a function call
Today's date is: 10/5/2003
```

(b) Call by Reference with Structure Data Type Whenever a call by reference is made, the address of the structure will be copied into a function subprogram and if changes are made in the subprogram, that will also be reflected back to the calling portion of a program because the address of the structure is same in both the calling and called up portions of the program. The scope and visibility of the variable parameters are the same throughout the program.

A program segment to illustrate how call by reference of a procedure subprogram with a structure data type can be implemented in C++.

```
#include <iostream>
using namespace std;
struct sample_info {
    int x;
     oat y;
};
struct sample_info * rst;
int main ()
{
    void display (struct sample_info *out); // function declaration
    ------
    ------
    display (out); // function calling
    ------
    ------
}
void display (struct sample_info *ptr) // function de nition
{
    ------
    ------
    ptr->x = 10;
    ptr->y = -20.20;
    ------
    ------
}
```

PROGRAM 9.9

A program to demonstrate how a call by reference of the function with structure data type in its arguments is realised.

```
// passing a struct to a function
// using a call by reference technique
```

```
// method 1.c using (->) operator
#include <iostream>
using namespace std;
struct date {
    int day;
    int month;
    int year;
};
int main()
{
    struct date *today;
    void display_newdate (struct date *newdate);
    today->day = 10;
    today->month = 5;
    today->year = 2003;
    cout <<"Before a function call \n";
    cout <<"Today's date is :";
    cout << today->day << "/" << today->month << "/" << today->year;
    cout << endl;
    display_newdate(today);
    cout <<"After a function call \n";
    cout <<"Today's date is :";
    cout << today->day << "/" << today->month << "/" << today->year;
    cout << endl;
    return 0;
}
void display_newdate(struct date *today)
{
    today->day++;
    today->month++;
    today->year++;
}
```

Output of the above program

```
Before a function call
Today's date is: 10/5/2003
After a function call
Today's date is: 11/6/2004
```

PROGRAM 9.10

A program to perform the following arithmetic operations of a complex number using a structure data type.

```
    (1) Addition of two complex numbers
    (2) Subtraction of two complex numbers
    (3) Multiplication of two complex numbers
    (4) Division of two complex numbers
// complex number operations using function
struct complex {
     oat real;
     oat imag;
};
struct complex add ( struct complex a, struct complex b);
struct complex sub ( struct complex a, struct complex b);
struct complex mul ( struct complex a, struct complex b);
struct complex div ( struct complex a, struct complex b);
#include <iostream>
using namespace std;
int main ()
{
    struct complex a,b,c;
```

```
    int ch;
    void menu();
    cout <<"enter a  rst complex number \n";
    cin >> a.real >> a.imag;
    cout <<" enter a second complex number \n";
    cin >> b.real >> b.imag;
    menu();
    while (( ch = getchar()) != 'q') {
       switch (ch) {
           case 'a' :
                c = add(a,b);
                cout <<" Addition of two complex numbers \n";
                cout << c.real << "+i" << c.imag << '\n';
                break;
           case 's' :
                c = sub (a,b);
                cout <<" Subtraction of two complex numbers \n";
                cout << c.real << "+i" << c.imag << '\n';
                break;
           case 'm' :
                c = mul (a,b);
                cout <<" Multiplication of two complex numbers \n";
                cout << c.real << "+i" << c.imag << '\n';
                break;
           case 'd' :
                c = div (a,b);
                cout <<" Division of two complex numbers \n";
                cout << c.real << "+i" << c.imag << '\n';
                break;
       } // end of switch
    }
    return 0;
}

void menu()
{
    cout <<" complex number operations \n";
    cout <<" menu () \n";
    cout <<" a -> addition \n";
    cout <<" s -> subtraction \n";
    cout <<" m -> multiplication \n";
    cout <<" d -> division \n";
    cout <<" q -> quit \n";
    cout <<" option, please ? \n";
}

struct complex add ( struct complex a, struct complex b)
{
    struct complex c;
    c.real = a.real+b.real;
    c.imag = a.imag+b.imag;
    return(c);
}

struct complex sub ( struct complex a, struct complex b)
{
    struct complex c;
    c.real = a.real-b.real;
    c.imag = a.imag-b.imag;
    return(c);
}
```

```
struct complex mul ( struct complex a, struct complex b)
{
    struct complex c;
    c.real = (a.real*b.real)-(a.imag*b.imag);
    c.imag = (a.real*b.imag)+(a.imag*b.real);
    return(c);
}

struct complex div ( struct complex a, struct complex b)
{
    struct complex c;
     oat temp;
    temp = (b.real*b.real)+(b.imag*b.imag);
    c.real = ((a.real*b.real)+(a.imag*b.imag))/temp;
    c.imag = ((b.real*a.imag)-(a.real*b.imag))/temp;
    return(c);
}
```

Output of the above program

```
enter a  rst complex number
1  1
enter a second complex number
2  2
complex number operations
menu ()
a -> addition
s -> subtraction
m -> multiplication
d -> division
q -> quit
option, please ?

a
Addition of two complex numbers
3+i3

s
Subtraction of two complex numbers
-1+i-1

m
Multiplication of two complex numbers
m
0+i4

d
Division of two complex numbers
0.5+i0
q
```

9.6 ARRAYS OF STRUCTURES

It is well known that an array is a group of identical data which is stored in consecutive memory locations in a common heading or common variable name. A similar type of structures that are placed in a common heading or a common variable name is called an array of structures. For an example, we would like to process the student's particulars for the entire school. That means there are more than one or two students.

So we need this type of structure facilities.

For example, the following program segment illustrates how to declare an array of structure data type in C++.

```
struct  student_info {
    int rollno;
    int age;
    char sex;
     oat height;
     oat weight;
};
student_info student[300];
```

The `student [300]` is a structure variable. It may accommodate the structure of a student up to 300. Each record may be accessed and processed separately like individual elements of an array.

Reading and writing of an array of a structure is similar to the conventional array methods. The `cin` function can be used to get an input data from the keyboard. The `cout` method are used to display the contents of the structures onto the video screen.

Initialisation of Arrays of Structures A structure can be initialised in the same way as that of array data in C++. In keeping with the array analogy, a structure must be either static or external. The structure can be initialised in the following way:

```
struct  student_info {
   long int rollno;
   int age;
   char sex;
    oat height;
    oat weight;
};
student_info student[3] = {
                          {95001,24,'M', 167.9,56.7},
                          {95002,25,'F', 156.6,45},
                          {95003,27,'M', 189.9,78}
                          };
```

The C++ compiler will assign the values to the individual elements of a the particular structure in the following way:

For the first record,

```
student[0].rollno = 95001;
student[0].age = 24;
student[0].sex = 'M';
student[0].height = 167.9;
student[0].weight = 56.7;
```

For the second record,

```
student[0].rollno = 95002;
student[0].age = 25;
student[0].sex = 'F';
student[0].height = 156.6;
student[0].weight = 45;
```

Like this, the C++ compiler will assign the individual elements to each record. Suppose, if any of the structure is not initialised then the compiler will automatically assign zero to the elements of the particular record.

For example,

```
struct  student_info {
     long int rollno;
     int age;
     char sex;
      oat height;
      oat weight;
 };
 student_info student[5] = {
                           { 95001,24,'M' },
                           { 95002,25,'F'}
                           };
```

For the above, the C++ compiler will assign the values to the individual elements of the particular structure in the following way:

```
student[0].rollno = 95001;
student[0].age = 24;
student[0].sex = 'M';
student[0].height = 0;
student[0].weight = 0;
```

For the second record,

```
student[0].rollno = 95002;
student[0].age = 25;
student[0].sex = 'F';
student[0].height = 0;
student[0].weight = 0;
```

For the third and the rest of the structure, it will assign zeros as they are not initialised explicitly in the above declaration. Even the C++ declaration allows to initialise the values to the members of a structure in the following format:

```
struct  student_info {
    long int rollno;
    int age;
    char sex;
     oat height;
     oat weight;
 };
 student_info student[3] = {
                           95001,24,'M', 167.9,56.7,
                           95002,25,'F', 156.6,45,
                           95003,27,'M', 189.9,78,
                           };
```

In this case, each member of a structure must be initialised and cannot be skipped to assign zero automatically by the C++ compiler. Some of the structures can be initialised zero automatically if they are not initialised explicitly in the declaration part.

PROGRAM 9.11

A program to demonstrate how data items are intialised in the array of structures and display the contents of the variables onto the screen.

```
// arrays of structure initialisation
#include <iostream>
using namespace std;
const int MAX = 3;
int main ()
{
    struct student_info {
        long int rollno;
        int age;
        char sex;
         oat height;
         oat weight;
    };
    struct student_info student[MAX] = {
                                        {20071,24,'M', 167.9,56.7},
                                        {20072,25,'F', 156.6,45},
                                        {20073,27,'M', 189.9,78}
                                        };
    int i;
    for (i = 0; i <= MAX-1; ++i) {
        cout <<"\n contents of structure = " << i+1 << '\n';
        cout <<" Roll no = " << student[i].rollno << '\n';
        cout <<" Age     = " << student[i].age << '\n';
        cout <<" Sex     = " << student[i].sex << '\n';
        cout <<" Height  = " << student[i].height << '\n';
        cout <<" Weight  = " << student[i].weight << '\n';
    }
    return 0;
}
```

Output of the above program

```
Contents of structure = 1
Roll no = 20071
Age     = 24
Sex     = M
Height  = 167.9
Weight  = 56.7

Contents of structure = 2
Roll no = 20072
Age     = 25
Sex     = F
Height  = 156.6
Weight  = 45

Contents of structure = 3
Roll no = 20073
Age     = 27
Sex     = M
Height  = 189.9
Weight  = 78
```

PROGRAM 9.12

A program to initialise a few members of an array of structures and display the contents of all the structures.

```
// arrays of structure - initialization
#include <iostream>
using namespace std;
```

```
const int MAX = 4;
int main ()
{
    struct student_info {
        long int rollno;
        int age;
        char sex;
         oat height;
         oat weight;
    };
    struct student_info student[MAX] = {
                                        {2001,24,'M', 167.9,56.7},
                                        {2002,25,'F', 156},
                                        {2003,27,}
                                        };
    int i;
    for (i = 0; i <= 3; ++i) {
        cout <<" \ncontents of structure = "<< i+1 << '\n';
        cout <<" Roll no = " << student[i].rollno << '\n';
        cout <<" Age     = " << student[i].age << '\n';
        cout <<" Sex     = " << student[i].sex << '\n';
        cout <<" Height  = " << student[i].height << '\n';
        cout <<" Weight  = " << student[i].weight << '\n';
    }
    return 0;
}
```

Output of the above program

```
Contents of structure = 1
Roll no = 2001
Age     = 24
Sex     = M
Height  = 167.9
Weight  = 56.7

Contents of structure = 2
Roll no = 2002
Age     = 25
Sex     = F
Height  = 156
Weight  = 0

Contents of structure = 3
Roll no = 2003
Age     = 27
Sex     =
Height  = 0
Weight  = 0
```

9.7 ARRAYS WITHIN A STRUCTURE

It has already been discussed that C++ permits declaration of both simple and structured data types as a field of a structure. The components of a structure can also be an array data type that is one of the structured data type groups in C++. Whenever a structure is defined with member of an array data type, the fields or members can be accessed and processed using conventional array techniques.

The general syntax of the array within a structure is:

```
storage_class struct  user_de ned_name {
     data_type    member_1 [SIZE];
     data_type    member_2 [SIZE];
     -------
     -------
     data_type    member_n [SIZE];
};
```

Where the `base_type` of an array can be any valid C++ data objects such as `int`, `oat`, `char`, etc. Once the structure has been declared, it may be used. The user defined variables `v1`, `v2` and `vn` are the structure types whose field data types are an array.

For example, the following program segment illustrates how to declare a structure whose member is an array type:

(1)

```
struct student {
    char name[20];
    int  subj[7];
};
```

(2)

```
struct employee {
    char name[20];
    char sex;
    char address[20];
    char place[10];
    char pincode;
};
```

The C++ compiler allows to initialise the members of a structure even if the array data type is a member to it. In the following way one can initialise the members,

```
struct  student_info {
    char name[20];
    long int rollno;
    int age;
    char sex;
     oat height;
     oat weight;
 };
student_info student[MAX] = {
                            {"Sampath Reddy", 2001,24,'M', 167.9,56.7},
                            {"Makesh", 2003,25,'F', 156.6,45},
                            {"kumar ",3003,27,'M' }
                            };
```

The C++ compiler assigns the values to its members of a structure as

```
student[0].name[] = "Sampath Reddy";
student[0].rollno = 2001;
student[0].age = 24;
student[0].sex = 'M';
student[0].height = 167.9;
student[0].weight = 56.7;
```

and so on. In case if some of the members are not initialised explicitly, the compiler will treat them as zero.

PROGRAM 9.13

A program to demonstrate how data items are intialised in the array of structures and display the contents of the variables onto the screen. This program illustrates how to define, declare and realise the array within the array of structures.

```
// initialization of array of structure
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
   struct student_info {
       char *st_name;
       long int rollno;
       char sex;
        oat height;
        oat weight;
    };
    struct student_info stud[MAX] = {
                                    {"Sampath Reddy", 200701,'M',175,65},
                                    {"Kuppusamy", 200702,'M', 155,60},
                                    {"Jacob Daniel", 200703,'M', 181,80},
                                    };
    int i;
    cout <<"Contents of the student_info :\n";
    for ( i = 0; i <= MAX-1; ++ i) {
       cout <<"\n Record No = " << i+1 << '\n';
       cout <<"Name    : " << stud[i].st_name << '\n';
       cout <<"Roll no : " << stud[i].rollno << '\n';
       cout <<"sex     : " << stud[i].sex << '\n';
       cout <<"Height  : " << stud[i].height << '\n';
       cout <<"Weight  : " << stud[i].weight << '\n';
    }
    return 0;
}
```

Output of the above program

```
Contents of the student_info:
Record No = 1
Name    :    Sampath Reddy
Roll no :    200701
sex     : M
Height  : 175
Weight  : 65

Record No = 2
Name    : Kuppusamy
Roll no : 200702
sex     : M
Height  : 155
Weight  : 60

Record No = 3
Name    : Jacob Daniel
Roll no : 200703
sex     : M
Height  : 181
Weight  : 80
```

PROGRAM 9.14

A program to read a set of names from the keyboard and to sort the names in an alphabetical order. This program demonstrates how to implement and realise the array within the array of structures.

```
// sorting of names based on alphabetical order
// using bubble sort algorithm
#include <iostream>
#include <cstring>
using namespace std;
const int MAX = 100;
struct student_info {
    char name[20];
};
int main()
{
    int i,j,n;
    char ch;
    struct student_info stud[MAX];
    void output_data(struct student_info sample[],int n);
    void sort_data(struct student_info sample[],int n);
    cout <<" How many names ?\n";
    cin >> n;
    getchar(); // to skip any extra line feed
    cout <<"enter name ...\n";
    for ( i = 0; i <= n-1; ++i) {
        cout <<"Name : [" << i+1 << "] ";
        j = 0;
        while (( ch = getchar()) != '\n')
            stud[i].name[j++] = ch;
        stud[i].name[j++] = '\n';
        stud[i].name[j++] = '\0';
    }
    cout <<" \n \n";
    cout <<"Unsorted order ... \n";
    cout <<"---------\n";
    output_data(stud,n);
    sort_data(stud,n);
    cout <<"\n \n";
    cout <<"Sorted order ... \n";
    cout <<"---------\n";
    output_data(stud,n);
    return 0;
}

void output_data(struct student_info sample[],int n)
{
   int i,j;
   char ch;
   for ( i = 0; i <= n-1; ++i) {
        j = 0;
        while ((ch =sample [i].name[j++]) != '\n')
           putchar(ch);
        putchar('\n');
   }
}

/* bubble sort algorithm */
void sort_data(struct student_info a[],int n)
{
   int i,j;
   char temp[20];
   for ( i = 0; i <= n-1; ++i ) {
```

```
       for ( j = 0; j <= n-2; ++j) {
           if (strcmp (a[j].name,a[j+1].name) >= 0)
           {
               strcpy (temp,a[j].name);
               strcpy (a[j].name,a[j+1].name);
               strcpy (a[j+1].name,temp);
           }
       } /* end of j loop */
    } /* end of i loop  */
}
```

Output of the above program

```
How many names ?
7
enter name ...
Name: [1] Velusamy.L
Name: [2] Kandasamy.K
Name: [3] Antony Paul
Name: [4] Marry Josphine
Name: [5] Ahmed Gulam
Name: [6] Anbu Mani
Name: [7] Chrisite Peter
```

Unsorted order ...

```
Velusamy.L
Kandasamy.K
Antony Paul
Marry Josphine
Ahmed Gulam
Anbu Mani
Chrisite Peter
```

Sorted order ...

```
Ahmed Gulam
Anbu Mani
Antony Paul
Chrisite Peter
Kandasamy.K
Marry Josphine
Velusamy.L
```

PROGRAM 9.15

A program to read students' information such as name, roll number, age, sex, height and weight from the keyboard and to sort student's structures in an alphabetical order in which name is key for sorting. The sorted and unsorted structures are displayed onto the video screen.

```
// sorting of array of structures (name is the key)
#include <iostream>
#include <cstring>
using namespace std;
const int MAX = 100;
struct student_info {
   char name[20];
   long int rollno;
   char sex[5];
    oat height;
```

```
    oat weight;
};
int main()
{
    int i,j,n;
    char ch;
    struct student_info stud[MAX];
    void output_data(struct student_info sample[],int n);
    void sort_data(struct student_info sample[],int n);
    cout <<" How many records ?\n";
    cin >> n;
    getchar(); // to skip any extra line feed
    cout <<"enter data ...\n";
    for ( i = 0; i <= n-1; ++i) {
        cout <<"\n Record = " << i+1 << '\n';
        cout <<"Name : ";
        j = 0;
        while (( ch = getchar()) != '\n')
              stud[i].name[j++] = ch;
        stud[i].name[j++] = '\n';
        stud[i].name[j++] = '\0';
        cout <<"Roll no : ";
        cin >> stud[i].rollno;
        cout <<"sex     : ";
        cin >> stud[i].sex;
        cout <<"Height  : ";
        cin >> stud[i].height;
        cout <<"Weight  : ";
        cin >> stud[i].weight;
        getchar(); // skip white space,if any
    }
    cout <<" \n \n";
    cout <<"Unsorted order ... \n";
    cout <<"---------\n";
    output_data(stud,n);
    sort_data(stud,n);
    cout <<"\n \n";
    cout <<"Sorted order ... \n";
    cout <<"---------\n";
    output_data(stud,n);
    return 0;
}

void output_data(struct student_info sample[],int n)
{
    int i,j;
    char ch;
    cout <<"Name   Roll_No   Sex    Height   Weight \n";
    cout <<"-------------- \n";
    for ( i = 0; i <= n-1; ++i) {
        j = 0;
        while ((ch =sample[i].name[j++]) != '\n')
              putchar(ch);
        cout <<'\t' << sample[i].rollno << '\t';
        cout << sample[i].sex << '\t';
        cout << sample[i].height << '\t';
        cout << sample[i].weight << '\t';

        cout <<"\n";
        cout <<"------------------ \n";
    }
}

/* bubble sort algorithm */
void sort_data(struct student_info a[],int n)
{
    struct student_info temp;
```

```
    for ( int i = 0; i <= n-1; ++i ) {
       for ( int j = 0; j <= n-2; ++j) {
           if (strcmp (a[j].name,a[j+1].name) >= 0)
           {
               temp = a[j];
               a[j] = a[j+1];
               a[j+1] = temp;
           }
       } /* end of j loop */
    } /* end of i loop  */
}
```

Output of the above program

```
How many records?
7
enter data ...

Record = 1
Name     : Sampath Reddy
Roll no  : 20071
sex: M
Height   : 178
Weight   : 67

Record = 2
Name: Sudheer Reddy
Roll no  : 20078
sex      : M
Height   : 167
Weight   : 90

Record = 3
Name: Velusamy.L
Roll no  : 20072
sex: M   : M
Height   : 156
Weight   : 67

Record = 4
Name     : Mary Peter
Roll no  : 20074
sex      : F
Height   : 145
Weight   : 50

Record = 5
Name     : Ahmed Saif
Roll no  : 20079
sex      : M
Height   : 190
Weight   : 90

Record = 6
Name     : Antony Paul
Roll no  : 20070
sex      : M
Height   : 145
```

```
Weight   : 56

Record = 7
Name     : Kuppusamy.K
Roll no  : 200711
sex      : M
Height   : 186
Weight   : 78

Unsorted order

-------------------------------------------------------
Name                    Roll_No   Sex   Height   Weight
-------------------------------------------------------
Sampath Reddy            20071     M      178      67
Sudheer Reddy            20078     M      167      90
Velusamy.L               20072     M      156      67
Mary Peter               20074     F      145      50
Ahmed Saif               20079     M      190      90
Antony Paul              20070     M      145      56
Kuppusamy.K             200711     M      186      78

Sorted order

-------------------------------------------------------
Name                    Roll_No   Sex   Height   Weight
-------------------------------------------------------
Ahmed Saif               20079     M      190      90
Antony Paul              20070     M      145      56
Kuppusamy.K             200711     M      186      78
Mary Peter               20074     F      145      50
Sampath Reddy            20071     M      178      67
Sudheer Reddy            20078     M      167      90
Velusamy.L               20072     M      156      67
-------------------------------------------------------
```

9.8 STRUCTURES WITHIN A STRUCTURE (NESTED STRUCTURE)

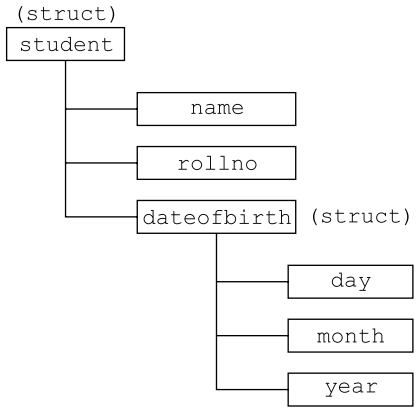
So far, whatever discussions have been made pertaining to the declaration of the field of a structure are restricted only to simple and array data types. In C++, it is permitted to declare a structure as a member of another structure. When a structure is declared as the member of another structure, it is called as nested structure or structure within a structure. The main advantages of using nested structure is to process and realise complex structures in an easy manner. Storing and retrieving of nested structures are much simpler than the conventional structures if there are many fields to be processed.

The general format for a declaration of a nested structure is as follows:

```
storage_class struct outer_structname {
    data_type   member 1;
    data_type   member 2;
    -------
    -------
    data_type   member n;
};
storage_class struct user_de ned_name {
    outer_structname   member 1;
    data_type   member 2;
};
```

The storage class is optional, whereas the keyword `struct` and the braces are essential. The `user-de ned name` is usually used, but there are situations in which it is not required. The data type and members are any valid C++ data objects such as `short int`, `oat`, `char` and `struct`. The level of nesting of structures depends on the compiler being used in the machine.

The nested structure can be represented graphically as:



To process the individual elements in a nested structure, first represent the structure variable name and the first structure, and then the field name of the first structure.

```
(struct variable name).(struct  rst name).  eld name = variable;
```

For example, the following program segment illustrates how to declare a nested structure in C++. The structure student_info contains three members, namely, name, rollno and dob where dob is a struct data type. The dob is declared as struct which consists of three members, namely, day, month and year.

```
struct date {
    int day;
    int month;
    int year;
};
struct student_info {
    char name[20];
    long int rollno;
    struct date dob;
};
struct student_info a;
int main ()
```

```
{
    // nested structure assignments
    -------
    -------
    a.name[] = " sample ";
    a.rollno = 95001;
    a.dob.day = 21;
    a.dob.month = 12;
    a.dob.year = 1990;
    -------
    -------
    return 0;
}
```

PROGRAM 9.16

A program to demonstrate how to define, declare and realise a nested structure in C++.

```
// nesting struct (struct within struct)
#include <iostream>
using namespace std;
int main()
{
   struct date {
       int day;
       int month;
       int year;
   };
   struct student_info {
       long int rollno;
       char sex;
        oat height;
        oat weight;
       struct date dateofbirth;
   };
   struct student_info student;
   student.rollno = 2003101;
   student.sex = 'M';
   student.height = 175;
   student.weight = 65;
   student.dateofbirth.day = 10;
   student.dateofbirth.month = 5;
   student.dateofbirth.year = 1964;
   cout <<"Contents of the student_info :\n";
   cout <<"\n Roll no : " << student.rollno;
   cout <<"\n sex     : " << student.sex;
   cout <<"\n Height  : " << student.height;
   cout <<"\n Weight  : " << student.weight;
   cout <<"\n Date of birth :";
   cout << student.dateofbirth.day << "/";
   cout << student.dateofbirth.month << "/";
   cout << student.dateofbirth.year << '\n';
   return 0;
}
```

Output of the above program

```
Contents of the student_info:
Roll no         : 2003101
sex             : M
Height          : 175
```

```
Weight          : 65
Date of birth   : 10/5/1964
```

PROGRAM 9.17

A program to demonstrate how to define, declare and realise a deep nested structure in C++.

```
//  nesting struct (struct within struct)
#include <iostream>
using namespace std;
int main()
{
   struct date {
       int day;
       int month;
       int year;
   };
   struct physical_info {
       char *st_name;
       long int rollno;
       char sex;
        oat height;
        oat weight;
   };
   struct course_details {
       char *cu_name;
       int  no_semester;
       struct date datejoin;
   };
   struct student_info {
       struct physical_info basic;
       struct course_details course;
       struct date datebirth;
   };
   struct student_info stud;
   stud.basic.st_name = "Sampath Reddy";
   stud.basic.rollno = 2003101;
   stud.basic.sex = 'M';
   stud.basic.height = 175;
   stud.basic.weight = 65;
   stud.course.cu_name = "B.Tech";
   stud.course.no_semester = 8;
   stud.course.datejoin.day = 10;
   stud.course.datejoin.month = 5;
   stud.course.datejoin.year = 2003;
   stud.datebirth.day = 23;
   stud.datebirth.month = 12;
   stud.datebirth.year = 1978;
   cout <<"Contents of the student_info :\n";
   cout <<"\n Name    : " << stud.basic.st_name;
   cout <<"\n Roll no : " << stud.basic.rollno;
   cout <<"\n sex     : " << stud.basic.sex;
   cout <<"\n Height  : " << stud.basic.height;
   cout <<"\n Weight  : " << stud.basic.weight;
   cout <<"\n Course Detail's :";
   cout <<"\n Course Name  :" << stud.course.cu_name;
   cout <<"\n No of Semester  :" << stud.course.no_semester;
   cout <<"\n Date of joining the course :";
   cout << stud.course.datejoin.day << "/";
   cout << stud.course.datejoin.month << "/";
   cout << stud.course.datejoin.year ;
   cout <<"\n Date of birth :";
   cout << stud.datebirth.day << "/";
   cout << stud.datebirth.month << "/";
```

```
    cout << stud.datebirth.year << '\n';
    return 0;
}
```

Output of the above program

```
Contents of the student_info :

Name      : Sampath Reddy
Roll no   : 2003101
sex       : M
Height    : 175
Weight    : 65
Course Detail's:
Course Name:  B.Tech
No of Semester:  8
Date of joining the course: 10/5/2003
Date of birth: 23/12/1978
```

PROGRAM 9.18

A program to read a students' information such as name, roll number, sex and date of joining like day, month and year of the institute from the keyboard and to sort the student's structures in an alphabetical order in which name is key for sorting. The sorted and unsorted structures are displayed onto the video screen.

```
// sorting of array of nested structures (name is key)
#include <iostream>
#include <cstring>
using namespace std;
const int MAX = 100;
struct date {
    int day;
    int month;
    int year;
};
struct physical_info {
    char name[20];
    long int rollno;
    char sex[5];
     oat height;
     oat weight;
};
struct student_info {
    struct physical_info  basic;
    struct date  dateofbirth;
};
int main()
{
    int i,j,n;
    char ch;
    struct student_info stud[MAX];
    void output_data(struct student_info sample[],int n);
    void sort_data(struct student_info sample[],int n);
    cout <<" How many records ?\n";
    cin >> n;
    getchar(); // to skip any extra line feed
    cout <<"enter data ...\n";
    for ( i = 0; i <= n-1; ++i) {
       cout <<"\n Record = " << i+1 << '\n';
       cout <<"Name : ";
       j = 0;
```

```
        while ((ch = getchar()) != '\n')
        stud[i].basic.name[j++] = ch;
        stud[i].basic.name[j++] = '\n';
        stud[i].basic.name[j++] = '\0';
        cout <<"Roll no : ";
        cin >> stud[i].basic.rollno;
        cout <<"sex     : ";
        cin >> stud[i].basic.sex;
        cout <<"Height  : ";
        cin >> stud[i].basic.height;
        cout <<"Weight  : ";
        cin >> stud[i].basic.weight;
        cout <<"Enter date of Birth (dd-mm-yy) \n";
        cin >> stud[i].dateofbirth.day;
        cin >> stud[i].dateofbirth.month;
        cin >> stud[i].dateofbirth.year;
        getchar(); // skip white space,if any
    }
    cout <<" \n \n";
    cout <<"Unsorted order ... \n";
    cout <<"---------\n";
    output_data(stud,n);
    sort_data(stud,n);
    cout <<"\n \n";
    cout <<"Sorted order ... \n";
    cout <<"----------\n";
    output_data(stud,n);
    return 0;
}

void output_data(struct student_info sample[],int n)
{
    int i,j;
    char ch;
    cout <<"Name   Roll  Sex  Height  Weight DateofBirth\n";
    cout <<"-------------------- \n";
    for ( i = 0; i <= n-1; ++i) {
        j = 0;
        while ((ch =sample[i].basic.name[j++]) != '\n')
            putchar(ch);
        cout << '\t'<< sample[i].basic.rollno;
        cout << '\t'<< sample[i].basic.sex;
        cout << '\t'<< sample[i].basic.height;
        cout << '\t'<< sample[i].basic.weight;
        cout << '\t' << sample[i].dateofbirth.day;
        cout << "/" << sample[i].dateofbirth.month;
        cout << "/" << sample[i].dateofbirth.year;
        cout <<"\n";
        cout <<"-------------------- \n";
    }
}

/* bubble sort algorithm */
void sort_data(struct student_info a[],int n)
{
    int i,j;
    struct student_info temp;
    for (i = 0; i <= n-1; ++i ) {
      for (j = 0; j <= n-2; ++j) {
           if (strcmp (a[j].basic.name,a[j+1].basic.name) >=0)
           {

                  temp = a[j];
                  a[j] = a[j+1];
                  a[j+1] = temp;
            }
```

```
        } /* end of j loop */
    } /* end of i loop  */
}
```

Output of the above program

```
How many records?
6
enter data ...

Record = 1
Name     : Velusamy
Roll no  : 27001
sex      : M
Height   : 167
Weight   : 89
Enter date of Birth (dd-mm-yy) 12 12 1980

Record = 2
Name     : Antony
Roll no  : 27002
sex      : M
Height   : 156
Weight   : 78
Enter date of Birth (dd-mm-yy) 21 11 1980

Record = 3
Name     : Mary
Roll no  : 27004
sex      : F
Height   : 145
Weight   : 45
Enter date of Birth (dd-mm-yy) 10 10 1980

Record = 4
Name     : Anbu
Roll no  : 27006
sex      : M
Height   : 164
Weight   : 67
Enter date of Birth (dd-mm-yy) 12 12 1979

Record = 5
Name     : Arul
Roll no  : 27007
sex      : M
Height   : 178
Weight   : 67
Enter date of Birth (dd-mm-yy) 13 11 1981
Record = 6
Name     : Sinha
Roll no  : 27009
sex      : M
Height   : 156
Weight   : 67
Enter date of Birth (dd-mm-yy) 25 10 1981
```

```
Unsorted order

-------------------------------------------------------------------------
Name                        Roll     Sex   Height   Weight   Date of Birth
-------------------------------------------------------------------------
Velusamy                   27001     M      167       89      12/12/1980
Antony                     27002     M      156       78      21/11/1980
Mary                       27004     F      145       45      10/10/1980
Anbu                       27006     M      164       67      12/12/1979
Arul                       27007     M      178       67      13/11/1981
Sinha                      27009     M      156       67      25/10/1981
-------------------------------------------------------------------------

Sorted order

-------------------------------------------------------------------------
Name                        Roll     Sex   Height   Weight   Date of Birth
-------------------------------------------------------------------------
Anbu                       27006     M      164       67      12/12/1979
Antony                     27002     M      156       78      21/11/1980
Arul                       27007     M      178       67      13/11/1981
Mary                       27004     F      145       45      10/10/1980
Sinha                      27009     M      156       67      25/10/1981
Velusamy                   27001     M      167       89      12/12/1980
-------------------------------------------------------------------------
```

9.9 POINTERS AND STRUCTURES

So far, it has been shown that a member of a structure could be an ordinary data type such as int, float, char or even a structure also. In this section, how a pointer variable can be declared as a member to a structure is discussed. In Chapter 6 on pointers, it has been stated that a pointer is a variable which holds the memory address of a variable of basic data types such as int, float or sometimes an array. A pointer can be used to hold the address of a structure variable too. The pointer variable is very much used to construct complex data bases using the data structures such as linked lists, double linked lists and binary trees.

The following declaration is valid

```
struct sample {
    int x;
     oat y;
    char s;
};
struct sample *ptr;
```

where `ptr` is a pointer variable holding the address of the structure sample and is having three members such as `int x`, `oat y` and `char s`.

The pointer to structure variable can be accessed and processed in one of the following ways:

```
(*structure name). eld name = variable;
```

The parentheses are essential because the structure member period (.) has a higher precedence over the indirection operator (*). The pointer to structure can also be expressed using dash (–) followed by the greater than sign (>).

```
structure name ->  eld name = variable;
```

The following assignment is a valid pointer structure:

Type 1 The pointer to structure variable can be accessed and processed in the following ways:

```
(*structure name).  eld name = variable;
```

The parentheses are essential because the structure member period (.) has a higher precedence over the indirection operator (*).

```
#include <iostream>
using namespace std;
int main()
{
      struct sample {
          int x;
           oat y;
          char s;
      };
      struct sample *ptr;
      (*ptr).x = 10;
      (*ptr).y = -23.45;
      (*ptr).s = 'd';
      -------
      -------
      return 0;
}
```

PROGRAM 9.19

A program to assign some values to the member of a structure using an indirection operator.

```
//pointers and structures
//method 1
#include <iostream>
using namespace std;
int main()
{
    struct sample{
        int x;
        int y;
    };
    sample  *ptr;
    sample one;
    ptr = &one;
    (*ptr).x = 10;
    (*ptr).y =  20;
    cout << "contents of x = "<< (*ptr).x << endl;
    cout << "contents of y = "<< (*ptr).y << endl;
    return 0;
}
```

Output of the above program

```
contents of x = 10
contents of y = 20
```

Type 2 The pointer to structure can also be expressed using dash (–) followed by the greater than sign (>).

```
structure name ->  eld name = variable;
```

The pointer to structure variable can be accessed and processed in the following ways:

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
        int x;
         oat y;
        char s;
    };
    struct sample *ptr;
    ptr->x = 10;
    ptr->y = -23.45;
    ptr->s = 'd';
    -------
    -------
    return 0;
}
```

PROGRAM 9.20

A program to assign some values to the member of a structure using a pointer structure operator.

```
//pointers and structures
// method 2

#include <iostream>
using namespace std;
int main()
{
   struct sample{
       int x;
       int y;
   };
   sample one;
   sample *ptr;
   ptr = &one;
   ptr->x = 10;
   ptr->y =  20;
   cout << "contents of x = "<< ptr->x << endl;
   cout << "contents of y = "<< ptr->y << endl;
   return 0;
}
```

Output of the above program

```
contents of x = 10
contents of y = 20
```

PROGRAM 9.21

A program to read a set of values from the keyboard using a pointer to structure operator and display the contents of the structure on the screen.

```
//pointers and structures
#include <iostream>
using namespace std;
int main()
{
    struct sample{
        int x;
        int y;
    };
    sample *ptr;
    cout <<  "enter value for x and y \n";
    cin>> ptr->x >> ptr->y;
    cout << "contents of x = "<< ptr->x << endl;
    cout << "contents of y = "<< ptr->y << endl;
    return 0;
}
```

Output of the above program

```
enter value for x and y
10   20
contents of x = 10
contents of y = 20
```

The indirection operator is also used to assign the pointer variable to a structure. For example,

```
#include <iostream>
using namespace std;
int main()
{
    struct sample{
        int *ptr1;
         oat *ptr2;
    };
    sample * rst,obj;
     rst = &obj;
    int value1;
     oat value2;
    -------
    -------
    (* rst).ptr1 =&value1;
    (* rst).ptr2 =&value2;
    -------
    -------
    return 0;
}
```

PROGRAM 9.22

A program to declare a pointer variable as a member of a structure and display the contents of the structure.

```
//pointers and structures
#include <iostream>
using namespace std;

int main()
{
    struct sample{
        int *ptr1;
         oat *ptr2;
    };
    sample * rst,obj;
    int value1;
     oat value2;
    value1 = 10;
    value2 = -20.20;
     rst = &obj;
     rst->ptr1 = &value1;
     rst->ptr2 = &value2;
    cout << "contents of the  rst member = "<< * rst->ptr1 << endl;
    cout << "contents of the second member = "<< * rst->ptr2 << endl;
    return 0;
}
```

Output of the above program

```
contents of the  rst member = 10
contents of the second member = -20.2
```

9.10 UNIONS

It is well known that a structure is a heterogeneous data type which allows to pack together different types of data values as a single unit. Union is also similar to a structure data type with a difference in the way the data is stored and retrieved.

The union stores values of different types in a single location. A union will contain one of the many different types of values (as long as only one is stored at a time). The declaration and the usage of union is same as structures. Union holds only one value for one data type. If a new assignment is made, the previous value is automatically erased.

The symbolic representation of a union declaration is:

```
union  user_de ned_name {
       member  1;
       member  2;
       -------
       -------
       member n;
};
```

The keyword `union` is used to declare the union data type. This is followed by a `user_de ned_name` surrounded by braces which describes the member of the union.

The general format of the union is,

```
storage_class  union user_de ned_name {
      data_type  member 1;
      data_type  member 2;
      -------
      -------
      data_type  member 1;
};
```

The storage class is optional. The keyword union and the braces are essential. The data type and members can be any valid C++ data objects such as `short int`, `oat` and `char`.

The syntax diagram of union declaration is given in Fig. 9.2.

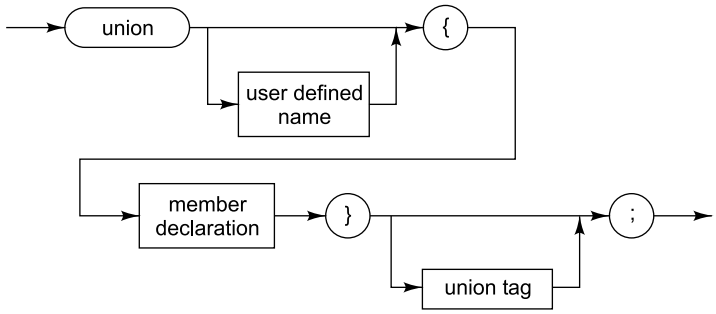


Fig. 9.2 *Syntax Diagram of Union Declaration*

A union may be a member of a structure and a structure may be a member of a union. Moreover, structures and unions can be mixed freely with arrays.

9.10.1 The Union Tag

C++, permits the definition of a union data type without declaring a variable. This is called as union tag. The general format of the union tag declaration is:

```
union  user_de ned_name {
       data_type    member 1;
       data_type    member 2;
       -------
       -------
       data_type    member n;
} variable 1 , variable 2 ... variable n;
```

For example,

```
union  sample {
   int  rst;
    oat second;
   char third;
} one,two;
```

where `one` and `two` are the union variables similar to data size of the `sample`.

9.10.2 Processing with Union

A period operator is used in between the union variable name and the field name. Once a union type is defined, variables for the union data types can be declared.

```
union  value {
    int ch;
    double dd;
};
union value x;
```

Similar to structures, the dot operator is used to access a union's individual fields. To assign to the integer field of x one can use.

```
x.ch;
x.dd;
```

For example,

```
x.ch = 12;
x.dd = -123.4456;
```

9.10.3 Initialization of Unions

Static and external structures can be initialised when they are defined, and it may seem reasonable to allow the same for unions. However, a union has only one active member at any given time and it is up to the programmer to keep track of the active member, as this information is not inherently stored with the union itself.

Although pointers to unions may be used just like pointers to structures, unions themselves may not be passed as function arguments used in assignment statements or returned by a function. A variable may be a pointer to a union first as a pointer can point to a structure.

```
union value {
   int one;
    oat two;
   char three;
};
union value *ptr;
```

The member can be referred by using the pointer operator

```
item1 = ptr->one;
item2 = ptr->two;
item3 = ptr->three;
```

A union can be a member of a structure and it can appear as any member of the structure. Whenever, a union is declared as a member of a structure, it should not be the first member, but the last one.

For example,

```
#include <iostream>
using namespace std;
int main ()
{
    struct value {
        int slno;
        char sex;
        union item {
            int one;
             oat two;
            char three;
        };
        int i;
    }; // end of struct declaration
    struct value *u;
    -------
    -------
    u->item.one = 10;
    -------
    -------
}
```

PROGRAM 9.23

A program to initialise the members of a union and display the contents of the union.

```
// union 1.cpp
#include <iostream>
using namespace std;
int main ()
{
    union value {
        int i;
         oat f;
    };
    union value x;
    x.i = 10;
    x.f = -1456.45;
    cout << "  rst member = " << x.i << endl;
    cout << " second member = "<< x.f << endl;
    return 0;
}
```

Output of the above program

```
 rst member = -994701722
second member = -1456.45
```

In the above program, the union consists of two members such as an int and a float. Only the float values are stored and displayed correctly, and the integer values are displayed wrongly. The union only holds a value for one data type which requires a larger storage among their members.

PROGRAM 9.24

A program to declare a member of a union as a structure data type and to display the contents of the union.

```
//union 2.cpp
#include <iostream>
using namespace std;
int main ()
{
    struct date {
        int day;
        int month;
        int year;
    };
    union value {
       int i;
        oat f;
       struct date bdate;
    };
    union value x;
    x.i = 10;
    x.f = -1456.45;
    x.bdate.day = 12;
    x.bdate.month = 7;
    x.bdate.year = 1995;
    cout << "  rst member = "<< x.i << endl;
    cout << " second member = " << x.f << endl;
    cout << " structure : " << endl;
    cout << x.bdate.day << "/" << x.bdate.month << "/" ;
    cout << x.bdate.year << endl;
    return 0;
}
```

Output of the above program

```
 rst member = 12
second member = 1.68156e-44
```

```
structure :
12/7/1995
```

In the above program, the union consists of three members such as an int, a float and a struct. Only the values of struct members are stored and displayed correctly. The values of int and float are displayed with certain garbage values. It is noted that the union only holds a value for one data type which requires a larger storage among their members.

PROGRAM 9.25

A program to declare a union as a pointer data type and display the contents of the union using pointer operator.

```
// union 3.cpp
#include <iostream>
using namespace std;
int main ()
{
    union value {
        int i;
         oat f;
     };
     union value *ptr;
     ptr->i = 10;
     ptr->f = -1456.45;
     cout << "  rst member = " << ptr->i << endl;
     cout << " second member = "<< ptr->f << endl;
     return 0;
}
```

Output of the above program

```
 rst member = -994701722
second member = -1456.45
```

9.11 BIT FIELDS

A bit field is a special type of structure member in the sense several bit fields can be packed into an int. While bit fields are variables, they are defined in terms of bits rather than characters or integers. Bit fields are useful for maintaining single or multiple bit flags in an int without having to use logical AND and logical OR operations to set and clear them. They can also assist in combining and dissecting bytes and words that are sent to and received from external devices.

The formal declaration of a bit field is same as the declaration of a structure, but there is a difference in accessing and using a bit field in a structure. The number of bits required by a variable must be specified and followed by a colon while declaring a bit field. The bit fields can be signed or unsigned integers, from 1 to 16 bits in length. The number of bits will depend on the machine being used.

The bit field is very useful with data items where only a few bits are required to indicate a true or false condition. Secondly, the bit field is used to save the memory space. The number of bits required by each variable is declared in a structure. So, C++ will accommodate all these bits into a packed binary form.

The general format of the bit field declaration is:

```
struct user_de ned_name {
    data_type       member 1;
    data_type       member 2;
    -------
    -------
```

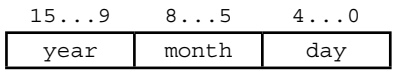
```
    data_type       member n;
};
```

where the individual elements have the same meaning as in structure declaration. Each member declaration must now indicate a specification indicating the size of the corresponding bit field. To do so, the member name must be followed by a colon and an unsigned integer indicating the size of field. The interpretation of these bit fields may vary from one C++ compiler to another. For example, some C++ compilers may order the bit field from right to left, whereas other C++ compilers will order from left to right.

For example,

```
struct date {
     unsigned int day : 5;  // day is 5 bits
     unsigned int month :4; // month is 4 bits
     unsigned int year :7 ; // year is 7 bits
};
```

Declare a structure with these fields:
day, month and year as shown below



The entire structure bits is a single 16 bit word, day takes up 5 bits, month takes up 4 bits and year takes up 7 bits. The way of accessing a bit field in a structure is similar to accessing another structure field. The period operator is used to access a bit field of a structure.

For example,

```
struct date {
     unsigned int day : 5;
     unsigned int month :4;
     unsigned int year :7 ;
};
struct date birthday; // holds a date
birthday.day = 16;
birthday.month = 3;
birthday.year = 1994;
```

There is one restriction. One cannot take the address of a bit field; which means that one cannot use `scanf` or `cin` to read values into a bit field. Instead, one has to read into a temporary variable and then assign its value to the bit field. Even the bit fields may be accessed in a structure using a pointer operator or indirection operator.

For example,

Case 1 Accessing a bit field using a pointer operator.

```
struct date {
     unsigned int day : 5;
     unsigned int month :4;
     unsigned int year :7 ;
};
struct date *bday; // holds a date
bday->day = 16;
bday->month = 3;
bday->year = 1994;
```

Case 2 Accessing a bit field using an indirection operator.

```
struct date {
```

```
        unsigned int day : 5;
        unsigned int month :4;
        unsigned int year :7 ;
    };
    struct date *bday; // holds a date
    (*bday).day = 16;
    (*bday).month = 3;
    (*bday).year = 1994;
```

PROGRAM 9.26

A program to declare the member of a structure using a bit field data type and display the contents of the structure.

```
//bit  eld 1.cpp
#include <iostream>
using namespace std;
int main ()
{
    struct value {
        unsigned day :5;
        unsigned month :4;
        unsigned year :7;
    };
    struct value obj;
    obj.day = 12;
    obj.month = 7;
    obj.year = 95;
    cout << " Date : " << obj.day << "/" << obj.month;
    cout << "/" << obj.year << endl;
    cout << " obj requires " << sizeof(obj) << " bytes" << endl;
    return 0;
}
```

Output of the above program

```
Date: 12/7/95
obj requires 4 bytes
```

PROGRAM 9.27

A program to declare the member of a structure as a bit field data type using a const definition and display the contents of the structure.

```
//bit  eld 2.cpp - using const width de nitions
#include <iostream>
using namespace std;
const int  BF1 = 5;
const int  BF2 = 4;
const int  BF3 = 7;
int main ()
{
    struct value {
        unsigned day :BF1;
        unsigned month :BF2;
        unsigned year :BF3;
    }a;
    a.day = 12;
    a.month = 7;
```

```
    a.year = 95;
    cout << " Bit  eld using the const " << endl;
    cout << " Date : " << a.day << "/" << a.month << "/";
    cout <<  a.year << endl;
    return 0;
}
```

Output of the above program

```
Bit  eld using the const
Date: 12/7/95
```

PROGRAM 9.28

A program to initialise the member of a structure as a bit field data type using a const definition and display the contents of the structure.

```
//bit  eld 3.cpp - bit  eld initialization
#include <iostream>
using namespace std;
const int  BF1 = 5;
const int  BF2 = 4;
const int  BF3 = 7;
int main ()
{
    struct value {
        int i;
        unsigned day :BF1;
        unsigned month :BF2;
        unsigned year :BF3;
         oat f;
    };
    struct value a = { 10,23,7,95,-123.4 };
    cout << " Bit  eld initialization " << endl;
    cout << " integer value =" << a.i << endl;
    cout << " Date :";
    cout << a.day << "/" <<a.month <<"/" <<  a.year << endl;
    cout << "  oating point value = "<< a.f;
    return 0;
}
```

Output of the above program

```
Bit  eld initialization
integer value = 10
Date: 23/7/95
 oating point value = -123.4
```

9.12 TYPEDEF

The `typedef` is used to define new data items that are equivalent to the existing data types. Once a user-defined data is declared, then new variables, arrays, structures, and so on can be declared in terms of this new data types.

The general format of the user-defined data types

```
typedef datatype newtype;
```

where `typedef` is a keyword for declaring the new data items and data type is an existing data type being converted to the new name.

For example,

```
typedef  int  integer;
typedef   oat real;

integer i,j;
real a,b;
```

where the declaration is same

```
int i,j;
 oat a,b;
```

The `typedef` is used in a program to make it readable and portable.

PROGRAM 9.29

A program to define the variables using typedef and to display the contents of the variable.

```
//using typedef
#include <iostream>
using namespace std;
int main ()
{
    typedef int integer;
    typedef  oat real;
    typedef char character;
    integer i,j;
    character ch;
    real a,b;
    i = 10;
    j = 30;
    ch = 'm';
    a = -23.45;
    b = 34.89;
    cout << " using typedef " << endl;
    cout << " i = " << i << '\t';
    cout << " j = " << j << endl;
    cout << " ch = " << ch << endl;
    cout << " a = " << a << '\t';
    cout << " b = " << b << endl;
    return 0;
}
```

Output of the above program

```
using typedef
i = 10       j = 30
ch = m
a = -23.45   b = 34.89
```

Even in the array declaration, the user-defined data type can be used. For example,

```
typedef  char  value[30];
value name;
```

is equivalent to the following declaration

```
char  name[30];
```

The following `typedef` in an array declaration is valid.

(1)

```
typedef  char  value[20];
value  name;
```

(2)

```
typedef  char value;
value name[20];
```

The `typedef` is used for declaring the structure data items also.

PROGRAM 9.30

A program to declare the member of a structure using typedef and to display the contents of the structure.

```
//using typedef in a structure
#include <iostream>
using namespace std;
int main ()
{
    struct  rst {
        int a;
         oat b;
        char c;
    };
    typedef struct  rst  number ;
    number one;
    one.a = 23;
    one.b = -13.45;
    one.c = 'n';
    cout << " contents of the structure " << endl;
    cout << one.a << '\t' << one.b << '\t' << one.c <<endl;
    return 0;
}
```

Output of the above program

```
contents of the structure
23    -13.45   n
```

The following are valid `typedef` declarations in the structure data type.

(1)

```
typedef  struct   rst {
      int a;
       oat b;
      char c;
} number;
number one;
```

(2)

```
typedef  struct {
      int day;
       oat month;
      char year;
} date;
typedef  struct {
      char name[20];
      int rollno;
      date dob;
} student;
student a[200];
```

(3)

```
typedef  struct {
```

```
        int day;
         oat month;
        char year;
} date;
typedef  struct {
        char name[20];
        int rollno;
        date dob;
} student[200];
student a;
```

(4)

```
typedef  struct {
        int day;
         oat month;
        char year;
} date;
typedef  struct {
        char name[20];
        int rollno;
        date dob;
} a[200];
```

9.13 ENUMERATIONS

Enumeration data types are available in other high level programming languages such as Pascal and Ada. It is also supported by all C++ compilers. An enumeration data type is a set of values represented by identifiers called enumeration constants. The enumeration constants are specified when the type is defined.

The general format of the enumeration data type is,

```
enum  user_de ned_name {
   member 1;
   member 2;
   ------
   ------
   member n;
};
```

where `enum` is a keyword for defining the enumeration data type and the braces are essential. The members of the enumeration data type such as `member 1`, `member 2` and `member n` are the individual identifiers. Once the enumeration data type is defined, it can be declared in the following ways:

```
storage_class  enum  user_de ned_name variable1,variable2 ..variablen
```

where the storage class is optional. For example, following are valid enumeration data type declarations:

(1)

```
enum sample {
        mon,tue,wed,thu,fri,sat,sun };
enum sample day1,day2,day3;
```

(2)

```
enum drinks {
        cola,maza,limca,rasna };
enum drinks ravi,raju,rani;
```

(3)

```
enum games {
```

```
        tennis,chess,shuttle,swimming,walking };
    enum games student,staff;
```

The enumeration data type declaration can be written in a single declaration as:

```
    enum sample {
          mon,tue,wed,thu,fri,sat,sun } day1,day2,day3;
```

which is exactly equivalent to

(1)

```
    enum sample {
          mon,tue,wed,thu,fri,sat,sun } day1;
    enum sample day2,day3;
```

(2)

```
    enum sample {
          mon,tue,wed,thu,fri,sat,sun };
    enum sample day1;
    enum sample day2;
    enum sample day3;
```

The enumeration constants can be assigned to the variable like

```
    day1 = mon;
```

`day2 = tue` and so on.

Enumeration constants are automatically assigned to integers starting from 0, 1, 2 etc. up to the last number in the enumeration.

PROGRAM 9.31

A program to declare the enumeration data type and to display the integer values on the screen.

```
// enumeration 1.cpp
#include <iostream>
using namespace std;
int main ()
{
    enum sample {
       mon,tue,wed,thu,fri,sat,sun }
       day1,day2,day3,day4,day5,day6,day7;
    day1 = mon;
    day2 = tue;
    day3 = wed;
    day4 = thu;
    day5 = fri;
    day6 = sat;
    day7 = sun;
    cout << " Monday = " << day1 << endl;
    cout << " Tuesaday = " << day2 << endl;
    cout << " Wednesday = " << day3 << endl;
    cout << " Thursday = " << day4 << endl;
    cout << " Friday = " << day5 << endl;
    cout << " Saturday = " << day6 << endl;
    cout << " Sunday = " << day7 << endl;
    return 0;
}
```

Output of the above program

```
Monday = 0
Tuesday = 1
```

```
Wednesday = 2
Thursday = 3
Friday = 4
Saturday = 5
Sunday = 6
```

These integers are normally chosen automatically but they can also be specified by the programmer. For example,

```
enum sample {
        mon,tue,wed = 10,thu,fri,sat = 15,sun }
        day1,day2,day3,day4,day5,day6,day7;
```

The C++ compiler assigns the enumeration constants as

```
Monday = 0
Tuesday = 1
Wednesday = 10
Thursday = 11
Friday = 12
Saturday = 15
Sunday = 16
```

Even negative integers are permitted to be defined as enumeration constants. For example, if,

```
enum sample {
       mon,tue,wed = 10,thu,fri = -1,sat,sun }
      day1,day2,day3,day4,day5,day6,day7;
```

then the C++ compiler assigns the following enumeration constants:

```
Monday = 0
Tuesday = 1
Wednesday = 10
Thursday = 11
Friday = -1
Saturday = 0
Sunday = 1
```

REVIEW QUESTIONS

1. What is a structure and what are its uses?
2. Distinguish a structure data type with other data type variables.
3. How is a structure different from an array?
4. Summarise the rules governing the declaration of a structure.
5. Describe how a structure can be initialised and what are the scope rules for that.
6. What is meant by a member or field of a structure?
7. What is the difference between declaration of a structure and initialisation of a structure?
8. What is meant by an array of fields in a structure and how is it different from an array?
9. How are the data elements of a structure accessed and processed?
10. What is meant by an array of structure?
11. How does the formal argument of a structure passed in a function call?
12. Can the return statement be used within a calling function of a structure?
13. What is meant by a structure within a structure?
14. Summarize a few real life applications of a structure data type.

15. What is a bit field and what is its use?
16. Describe how a bit field can be used within a structure declaration.
17. What is meant by a union? Differentiate between a structure data type and a union.
18. What is the advantage of using a union in C++?
19. Explain how a bit field can be used within a union data type.
20. What is a user-defined data type? List its merits and demerits.
21. Explain the salient features of the typedef.
22. How many data items can be stored in a union at any given time?
23. Is the structure tag required? Give an example of a structure with no tag.
24. List the merits and demerits of the enumeration data types.
25. What is the use of declaring an anonymous union in C++?
26. Summarise a few real-life applications of a structure data type.
27. Explain how the structures of two different types of fields can be compared and assigned.
28. Explain the various methods of declaring structures in C++.

CONCEPT REVIEW PROBLEMS

1. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
         int x;
         int y;
    };
    struct sample *ptr,obj;
    ptr = &obj;
    (*ptr).x = 100;
    (*ptr).y = -200;
    cout <<"Contents of x = " << ++(*ptr). x << '\n';
    cout <<"Contents of y = " << ++(*ptr). y << '\n';
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
        int *x;
        int *y;
    };
    struct sample *ptr,obj;
    int ix = 10,iy = -20;
    ptr = &obj;
    ptr->x = &ix;
    ptr->y = &iy;
    cout <<"Contents of x = " << *(*ptr).x << '\n';
```

```
    cout <<"Contents of y = " << *(*ptr).y << '\n';
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
        int *x;
        int *y;
    };
    struct sample *ptr,obj;
    int ix = 10,iy = -20;
    ptr = &obj;
    ptr->x = &ix;
    ptr->y = &iy;
    cout <<"Contents of x = " << ++(*(*ptr).x) << '\n';
    cout <<"Contents of y = " << ++(*(*ptr).y) << '\n';
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
        int *x;
        int *y;
    };
    struct sample *ptr,obj;
    int ix = 10,iy = -20;
    ptr = &obj;
    (*ptr).x = &ix;
    (*ptr).y = &iy;
    cout <<"Contents of x = " << ++(*ptr->x) << '\n';
    cout <<"Contents of y = " << ++(*ptr->y) << '\n';
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
        int x;
        int y;
    };
    struct sample *ptr,obj;
    ptr = &obj;
    ptr->x = 10;
    ptr->y = -20;
    cout <<"Contents of x = " << ++(*ptr).x << '\n';
    cout <<"Contents of y = " << ++(*ptr).y << '\n';
```

```
    return 0;
}
```

2. What will be the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
        int x;
        int y;
    };
    struct sample *ptr,obj;
    ptr = &obj;
    ptr->x = 10;
    ptr->y = -20;
    (*ptr).x++;
    (*ptr).y++;
    cout <<"Contents of x = " << (*ptr).x << '\n';
    cout <<"Contents of y = " << (*ptr).y << '\n';
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
        int x;
        int y;
    };
    struct sample *ptr,obj;
    ptr = &obj;
    ptr->x = 10;
    ptr->y = -20;
    ++ptr->x;
    ++ptr->y;
    cout <<"Contents of x = " << ptr->x << '\n';
    cout <<"Contents of y = " << ptr->y << '\n';
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
        int x;
        int y;
    };
    struct sample *ptr,obj;
    ptr = &obj;
    ptr->x = 10;
    ptr->y = -20;
```

```
    (ptr++)->x;
    (ptr++)->y;
    cout <<"Contents of x = " << ptr->x << '\n';
    cout <<"Contents of y = " << ptr->y << '\n';
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
        int *ptr1;
        int *ptr2;
    };
    struct sample *abc,obj;
    int value1 = 10,value2 = -20;
    abc = &obj;
    abc->ptr1 = &value1;
    abc->ptr2 = &value2;
    cout <<"Contents of x = " << *abc->ptr1 << '\n';
    cout <<"Contents of y = " << *abc->ptr2 << '\n';
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
        int *ptr1;
        int *ptr2;
    };
    struct sample *abc,obj;
    int value1 = 10,value2 = -20;
    abc = &obj;
    abc->ptr1 = &value1;
    abc->ptr2 = &value2;
    abc->ptr1++;
    abc->ptr2++;
    cout <<"Contents of x = " << *abc->ptr1 << '\n';
    cout <<"Contents of y = " << *abc->ptr2 << '\n';
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
int main()
{
    struct sample {
        int *ptr1;
        int *ptr2;
    };
    struct sample *obj,ptr;
```

```
    int value1 = 10,value2 = -20;
    obj = &ptr;
    obj->ptr1 = &value1;
    obj->ptr2 = &value2;
    ++(obj->ptr1);
    ++(obj->ptr2);
    cout <<"Contents of x = " << *obj->ptr1 << '\n';
    cout <<"Contents of y = " << *obj->ptr2 << '\n';
    return 0;
}
```

PROGRAMMING EXERCISES

1. (a) Develop a program in C++ to create a database for the on-line Banking system. Your database should consist of the following information:
customer name
customer code
Type of Account (Savings, Current, Fixed)
Amount deposited
Contact Address
E-mail ID

(b) Your program should do the following things:
- build a master table
- list a table
- insert a new entry
- delete an old entry
- edit entry
- search for a structure to be printed
- sort entries

2. Develop a program in C++ to create a data base with the following items using a structure data type:
name of the patient
sex
age
ward number
bed number
nature of the illness
date of admission

Your program should have the facilites as in 1 (b).

3. Develop a program in C++ to create a pay roll system of an organisation assuming that the following information can be read from the keyboard
employee name
employee code
designation
account number
date of joining
basic pay

DA, HRA and CCA
deductions like PPF, GPF, CPF, LIC, NSS, NSC, etc.

Your program should have the facilities as in 1(b).

4. Develop a program in C++ to prepare the mark sheet of a university examination assuming that the following items are read from the keyboard:
name of the student
roll number
subject code
subject name
internal marks
external marks

Your program should have the facilities as in 1(b).

5. Develop a program in C++ to create a library information system
accession number
name of the author
title of the book
year of publication
publisher's name
cost of the book

Your program should have the facilities as in 1(b).

6. Develop a program in C++ to create a data base of the personnel information system containing
name
date of birth
blood group
height
weight
insurance policy number
contact address
telephone number
driving licence number, etc.
Your program should have the facilities as in 1(b).

7. Develop a program in C++ to create a database for the Employee Information System (EIS) of an organisation and your data base should consist of the following information:
employee name
employee code
designation
years of experience
age

Your program should have the facilities as in 1(b).

Classes and Objects

Chapter 10

In the previous chapter, it has been explained how different data items could be grouped into a single entity as a structure or a union. Classes and objects are the main ideas of the object-oriented programming tools. How these items are realised in C++ are discussed in this chapter. The array of class objects, pointers to classes, and classes within a class are explained in this chapter with numerous illustrative examples. This chapter mainly covers how a class of objects can be defined, declared and used in a program. The various topics of classes, namely, class constructors, destructors, copy constructor, conversion constructor, this argument, inline functions and dynamic memory allocation operators such as new and delete are explained in the subsequent chapters.

10.1 INTRODUCTION

So far, how to develop C++ programs without the use of classes and objects have been explained without exploiting the full potential of C++ by using objects and classes. A class is a user-defined data type which holds both the data and functions. The internal data of a class is called member data (or data member) and the functions are called member functions. The member functions mostly manipulate the internal data of a class. The member data of a class should not normally be addressed outside a member function. The variables of a class are called objects or instances of a class.

The word 'class' is a fundamental and powerful keyword in C++. It is significantly useful as it is used to combine the data and operations of a structure into a single entity. The class construct differs from the conventional C language struct construct. The class construct provides support for data hiding, abstraction, encapsulation, single inheritance, multiple inheritance, polymorphism and public interface functions (methods) for passing message between objects. This section stresses only the elementary concepts of the object-oriented programming (OOP) terminology and the subsequent sections deals with implementation of these topics in C++.

(a) Data Abstraction In OOP, the data abstraction is defined as a collection of data and methods (functions).

(b) Data Hiding In C++, the class construct allows to declare data and methods, as a public, private and protected group. The implementation details of a class can be hidden. This is done by the data hiding principle.

(c) Data Encapsulation The internal data (the member data) of a class are first separated from the outside world (the defined class). They are then put along with the member functions in a capsule. In other words, encapsulation groups all the pieces of an object into one neat package. It avoids undesired side effects of the member data when it is defined out of the class and also protects the intentional misuse of important data. Classes efficiently manage the complexity of large programs through encapsulation.

(d) Inheritance C++ allows a programmer to build hierarchy of classes. The derivation of classes is used for building hierarchy. The basic features of classes (parent classes or basic classes) can be passed onto the derived classes (child classes). In practice, the inheritance principle reduces the amount of writing; as the derived classes do not have to be written again.

(e) Polymorphism In OOP, polymorphism is defined as how to carry out differnt processing steps by a function having the same messages. Polymorphism treats objects of related classes in a generic manner.

The following equivalent terminology is used between the function oriented programming and OOP:

Function oriented programming	*Object oriented programming (OOP)*
User-defined types	Classes
Variables	Objects
Structure members	Instance variables
Functions	Methods
Function call	Message passing

10.2 STRUCTURES AND CLASSES

It has already been stated in the previous chapter that a structure contains one or more data items (called members) which are grouped together as a single unit. On the other hand, a class is similar to a structure data type but it consists of not only data elements but also functions which are operated on the data elements. Secondly, in a structure, all elements are public by default, while in a class it is private. The data and functions can be defined in a class as one of the sections such as private, public and protected. Function defined within a class have a special relationship to the member data and member functions (methods). The Object Modelling Technique (OMT) of a class notation is given in Fig. 10.1.

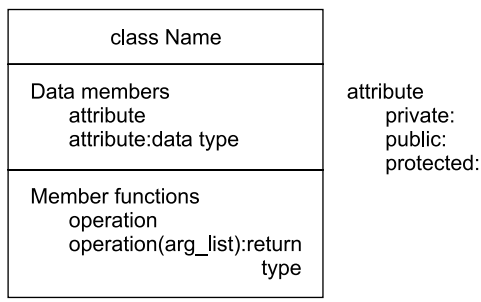


Fig. 10.1 *Object Modelling Technique (OMT) of a Class Notation*

The general syntax of the class construct is:

```
class user_de ned_name {
    private :
        data_type members
        implementation operations
        list of friend functions
        list of friend functions
    public :
        data_type members
        implementation operations
    protected :
        data_type operations
        implementation operations
};
class user_de ned_name variable1,variable2..variable n;
```

The keyword typedef is not required since a class name is a type of name. The keywords private, protected and public are used to specify the three levels of access protection for hiding data and function members internal to the class.

(a) Private In private section, a member data can only be accessed by the member function and friends of this class. The member functions and friends of this class can always read or write private data members. The private data member is not accessible to the outside world (out of the class).

(b) Protected The members which are declared in the protected section, can only be accessed by the member functions and friends of this class. Also, these functions can be accessed by the member functions and friends derived from this class. It is not accessible to the outside world.

(c) Public The members which are declared in the public section, can be accessed by any function in the outside world (out of the class). The public implementation operations are also called as member functions or methods, or interfaces to out of the class. Any function can send messages to an object of this class through these interface functions.

The public data members can always read and write outside this class. A member function can be inline, which means the member function can be defined within the body of the class constant. The keyword inline is used for short functions and efficient storage like the register keyword.

A class may be defined using one of the following keywords:

```
class
struct
union
```

A structure is a class declared with the class key struct; its members and base classes are public by default. A union is a class declared with the class key union; its members are public by default and it holds only one member at a time.

For example, the following declaration illustrates the default member access specifier:

(1) A class is declared with the keyword 'class',

```
class sample {
     int a;
      oat x;
     char ch;
 };  // A class by default has all its members private
```

(2) A class is declared with the keyword 'struct',

```
struct sample {
     int a;
```

```
        oat x;
       char ch;
};  // A struct by default has all its members public
```

(3) A class is declared with the keyword 'union',

```
union sample {
    int a;
     oat x;
    char ch;
};
```

A union by default has all its members public and it holds only one member at a time.

Class is a key concept of C++. A class is a user-defined type and it is the unit of data hiding and encapsulation. Polymorphism is supported through classes with virtual functions. The class provides a unit of modularity.

10.3 DECLARATION OF A CLASS

A class is a user-defined data type which consists of two sections, a private and a protected section that holds data and a public section that holds the interface operations.

A class definition is a process of naming a class and data variables, and methods or interface operations of the class. In other words, the definition of a class consists of the following steps:

(i) Definition of a class
(ii) The internal representation of data structures and storage
(iii) The internal implementation of the interface
(iv) The external operations for accessing and manipulating the instance of the class.

A class declaration specifies the representation of objects of the class and the set of operations that can be applied to such objects.

The general syntax of the class construct is:

```
class user_de ned_name {
    private :
        data_type members
        implementation operations
        list of friend functions
        list of friend functions
    public:
        list of friend functions
        data_type members
        implementation operations
    protected :
        list of friend functions
        data_type operations
        implementation operations
};
class user_de ned_name variable1,variable2..variable n;
```

The keyword typedef is not required since a class name is a type of name. The keywords private, protected and public are used to specify the three levels of access protection for hiding data and function members internal to the class.

The syntax diagram of a class declaration is given in Fig. 10.2(a).

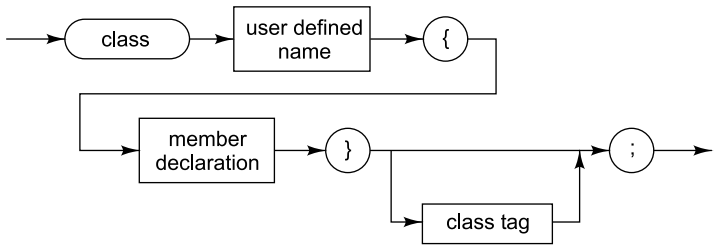


Fig. 10.2(a) *Syntax Diagram of Class Declaration*

A class declaration introduces the class name into the scope where it is declared and hides any class, object, function or other declaration of that name in an enclosing scope.

```
class item {
   // member lists
};
```

Class members can be one of the following member lists:

- data
- functions
- classes
- enumerations
- bitfields
- friends
- data type names

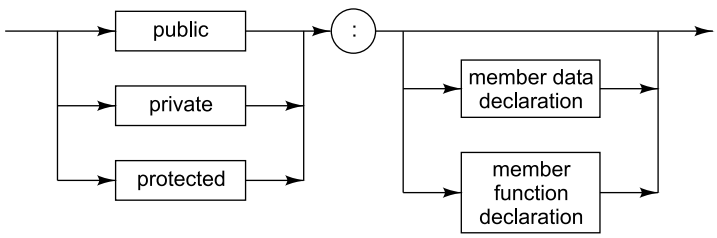


Fig. 10.2(b) *Syntax Diagram of Class Member Declaration*

For example,

(1) A class date is defined as day, month and year of a member data variable without any method or any member function.

```
class date {
    private :
       int day;
       int month;
       int year;
};
class date today;  // today is object of class date
```

(2) The student's particulars such as rollno, age, sex, height and weight can be grouped as:

```
class student
```

```
{
  private :
      long int rollno;
      int age;
      char sex;
       oat height;
       oat weight;
  public :
       oat weight;
      void getinfo ();  //member function
      void disinfo (); // member function
}; // end of class de nition
```

The class definition is permitted to be declared within the class declaration itself.

```
class date {
  public :
      int day;
      int month;
      int year;
} today;  // now the object is created of class date
```

The keyword class is essential for defining a class data type in C++. The important difference between structures and classes lies in the validity of area of the members. As we have already seen, in a class, by default, all members are private while in a structure or in a union it is public.

Fig. 10.2 (c) *Syntax Diagram of Member Data Declaration*

The following declaration is identical while accessing the member of the class.

(1)

```
class sample {
    int x;
    int y;
 };   // by default members are private
```

(2)

```
class sample {
   private :
       int x;
       int y;
};
```

Note that the keyword `'private'` is used for declaring the data items of a class explicitly as a private group.

Some special features of declaring a class data type Note that classes may be unnamed. An unnamed class cannot have constructors and destructors, and cannot be passed as an argument or returned as a value. Class objects may be assigned, passed as arguments to functions or returned by functions.

In C++, it is possible to declare an empty class having a nonzero size.

```
class  sample { };   // class with no members
```

Note, that a member may not be declared twice in the member list. The member list defines the full set of members of the class and no member can be added elsewhere.

The following class declarations are invalid.

(1)

```
class sample {
   private :
      int one;
      int two;
      int one;
};
// error, the data member 'one' has been redeclared
```

(2)

```
class item {
   private :
       oat x;
      char ch;
};

item:: y;
// error, no member can be added elsewhere, other than class declaration
```

(3)

```
class sample {
   private :
         int x,y;
   public :
         int one;
         void setdata();
         void getdata();
         void display();
         void setdata(); // error, redeclaration
};
```

The same rule applies for function declaration also.

(4) Note that a single name can denote several function members provided their types are sufficiently different. The same name cannot denote both a member function and a member data.

```
class xy {
   private :
         int funct;
   public :
         int funct();  // error, same name is used for both
};                     // data member and function
```

(5)

```
class abc {
   public :
         int funct();
         int (*funct) ();
};
// error, pointer to a function and the function name are same
```

Note that a member declaration cannot contain an initialiser. A member of a class can be initialised by a special member function called a constructor.

A data member may not be `auto`, `extern` or `register`. The following are illegal declarations of the data types:

```
class sample {
    private :
       auto int x,y;
    extern a,b;
       register one,two;
};
// error, illegal data type declaration
```

10.4 MEMBER FUNCTIONS

A function declared as a member (without the friend specifier) of a class is called as a member function. Member functions are mostly given the attributes of public because they have to be called outside the class either in a program or in a function.

The member functions of a class are designed to operate upon three data types. It can typically be classified under three types:

- manager functions
- accessor functions
- implementor functions

(a) Manager Functions Manager functions are used to perform initialisation and clean up of the instance of the class objects. Some of the examples for the manager functions are constructor and destructor functions. More of these functions are discussed in the next chapter.

(b) Accessor Functions The accessor member functions are the constructor functions that return information about an object's current state. An example for the accessor function is a const member function. These accessor member functions are explained in the next chapter.

(c) Implementor Functions These are the functions that make modifications to the data members. These functions are also called as mutators. In this section, the focus is on how to define and use the implementor functions which are one of the key concepts in defining the data hiding and data encapsulation.

As a class contains not only a data member but also a function which are called methods, it must be defined before it is be used.

For example, the following program segments illustrate how a data member and member function are defined in C++:

```
class sample {
   private :
       int x;
       int y;
   public :
       int sum() {       //member function
           return(x+y);
       }
       int diff() {     //member function
           return(x-y);
       }
 }; // end of class de nition
```

The member functions `sum ()` and `diff ()` are defined quite normally within the class declaration. The member function can be more complex as they can have local variable, parameters etc.

Defining a member function of a class outside its scope In C++, it is permitted to declare the member functions either inside the class declaration or outside the class declaration. A member function of a class is defined using the :: (double colon symbol) scoping operator.

The syntax diagram of member function declaration is given in Fig. 10.2(d).

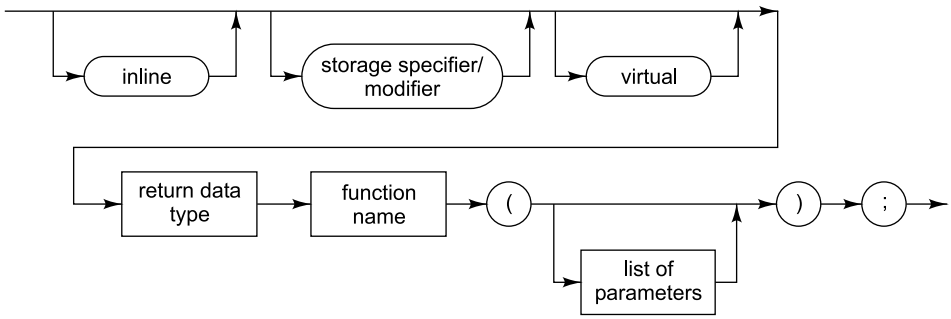


Fig. 10.2 (d) *Syntax Diagram of Member Function Declaration*

The general syntax of the member function of a class outside its scope is

```
return_type class_name :: member_functions(argument 1,2...n)
```

Note, that the type of member function arguments must exactly match with the types declared in the class definition of the class_name.

The important point to note is the use of the scope resolution operator (::) along with the class name in the header of the function definition. Only the scope operator identifies the function as a member of a particular class. Without this scope operator, the function definition would create an ordinary function, subject to the usual function rules of access and scope.

For example, the following program segment shows how a member function is declared outside the class declaration. The member functions are defined separately as part of the program.

```
class sample {
   private :
      int x;
      int y;
   public :
      int sum();   // member function declaration
      int diff(); // member function declaration
}; // end of class de nition
int sample :: sum()  // member function de nition
{
      return(x+y);
}

int sample ::diff()  // member function de nition
{
      return(x-y);
}
```

The use of the scope operator double colon (::) is important for defining the member functions outside the class declaration. For example, the following program segment illustrates the importance of the scoping of member functions in a class.

```
class  rst {
   private :
      int x;
      int y;
   public :
      int sum();
      int diff();
}; // end of class de nition
class second {
   private :
      int x;
      int y;
   public :
      int sum() ;
      int diff() ;
}; // end of class de nition
 rst one;
second two;
int  sum()  // error, scope of the member function is not de ned
{
     return(x+y);
}
```

In the above program segment, both classes are defined with the same member function names while accessing these member functions, which is an error. The scope of the member function sum() is not defined. When accessing the member function sum(), control will be transferred to both classes one and two. So the scope resolution operator (::) is absolutely necessary for defining the member functions outside the class declaration.

```
int one :: sum()  // correct
{
    return(x+y);
}

int two ::sum()  // correct
{
    return(x+y);
}
```

10.5 DEFINING THE OBJECT OF A CLASS

The terms objects and classes have been used loosely throughout the preceding section. In general, a class is a user-defined data type, while an object is an instance of a class template. A class provides a template, which defines the member functions and variables that are required for objects of the class type. A class must be defined prior to the class declaration.

The general syntax for defining the object of a class is:

```
class user_de ned_name {
    private :
              // methods
    public :
              // methods
```

```
    protected :
              // methods
};
user_de ned_name  object 1,object 2 ... object n;
```

where object 1, object 2 and object n are the identical class of the user_defined_name.

A class definition is very similar to a C structure definition. The class definition defines the member variables and functions.

For example, the following program segments show how to declare and to create a class of objects.

(1) Student's information such as roll no, age, sex, height and weight are grouped as

```
class student_info
 {
  private :
     long int rollno;
     int age;
     char sex;
      oat height;
      oat weight;
  public :
     void getinfo ();
     void disinfo ();
     void process();
     void personal();
}; // end of class de nition
student_info obj;
// obj is the object of the class student_info
```

The OMT of a class student_info is given in Fig. 10.3.

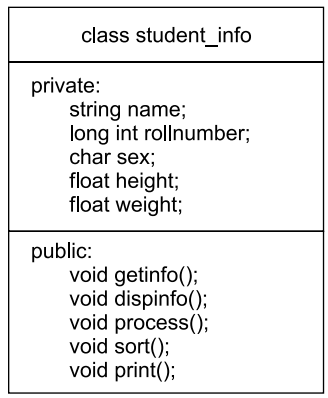


Fig. 10.3 *OMT of a Class Student_info*

(2) The hospital information system such as patient name, sex, age, etc. can be grouped as

```
class hospital_info {
   private :
      char patient_name[20];
      char sex;
      int age;
      char  fathers_name[20];
      char address[20];
      char illness[30];
      int wardno;
      int bedno;
   public :
      void getinfo();
      void display_info();
      void payment();
      void operation();
}; end of class declartion
hospital_info obj1,obj2;
// obj1 and obj2 are the two objects
// of the class hospital_info
```

The OMT of a class hospital_info is given in Fig. 10.4.

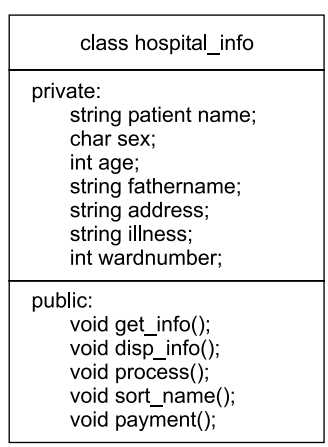


Fig. 10.4 *OMT of a Class Hospital_info*

(3) The employee particulars such as employee_name, employee_code, designation, address, monthly salary and age can be grouped as

```
class employ_info {
   private :
```

```
        char employee_name[20];
        int employee_code;
        char designation [20];
        char address[30];
         oat monthly_salary;
        int age;
    public :
        void salary_payment();
        void saving();
        void tax_payment();
        void get_info();
        void display_info();
};  // end of class de nition
employ_info x,y; // x and y are the two objects of the class employ_info
```

The OMT of a class employee_info is given in Fig. 10.5.

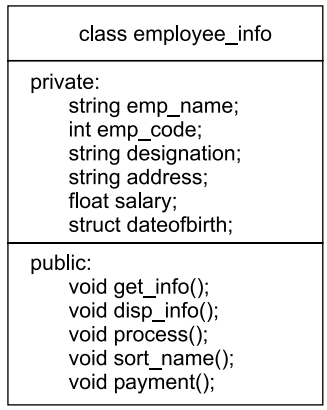


Fig. 10.5 *OMT of a Class Employee_info*

10.6 ACCESSING A MEMBER OF CLASS

There are two ways one can access a member of a class similar to accessing member of a struct or union construct. A data or function member of a class construct is accessed using the . (period) operator.

The general syntax for accessing a member of class is

```
class_object.data_member
class_object.function_member
```

For example,

```
class sample {
    private :
        int x;
        int y;
    public :
        int sum();
        int diff();
}; // end of class de nition
```

```
void main (void)
{
     sample one;
     one.sum(); // accessing the member function sum()
     one.diff(); // accessing the member function diff()
     -------
     -------
}
```

PROGRAM 10.1

A program to assign values to the data members of a class such as day, month, year and display the contents of the class on the screen.

```
// class 1.cpp
#include <iostream>
using namespace std;
int main()
{
    class date {
        public :
            int day,month,year;
    };
    class date today;
    today.day = 10;
    today.month = 5;
    today.year = 2007;
    cout << " Today's date is = " << today.day << "/";
    cout << today.month << "/" << today.year << endl;
    return 0;
}
```

Output of the above program

```
Today's date is = 10/5/2007
```

While the keyword public is not used to define the members of a class, the C++ compiler assumes, by default, that all its members are private. The data members are not accessible outside the class. For example, the following program demonstrates the accessibility of the members.

```
// class 2.cpp
#include <iostream>
using namespace std;
int main()
{
    class date {    // by default, members are private
       int day,month,year;
    };
    class date today;
    today.day = 10;
    today.month = 5;
    today.year = 2007;
    cout << " Today's date is = " << today.day << "/";
    cout << today.month << "/" << today.year << endl;
}
```

The following error message will be displayed during the compilation time.

date.day is a private
date.month is a private
date.year is a private

PROGRAM 10.2

A program to demonstrate how to define both data member and member function of a class within the scope of class definition.

The OMT of a class date is given in Fig. 10.6.

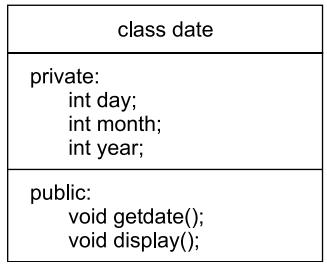


Fig. 10.6 *OMT of a Class Date*

```
//class with data and member function
#include <iostream>
using namespace std;
class date {
    private :
        int day,month,year;
    public :
       void getdata( int d,int m,int y)
       {
          day = d;
          month = m;
          year = y;
       }
       void display (void)
       {
          cout << " Today's date is = " << day << "/";
          cout << month << "/" << year << endl;
       }
}; // end of class de nition

int main()
{
    date today;
    int d1,m1,y1;
    d1 = 10;
    m1 = 5;
    y1 = 2007;
    today.getdata(d1,m1,y1);
    today.display();
    return 0;
}
```

Output of the above program

```
Today's date is = 10/5/2007
```

PROGRAM 10.3

A program to read the data variables of a class by the member function and display the contents of the class objects on the screen.

```
//class with data and member function
#include <iostream>
using namespace std;

class date {
    private :
       int day,month,year;
    public :
       void getdata()
       {
           cout << " enter the date ( dd-mm-year) " << endl;
           cin >> day >> month >> year;
       }
       void display ()
       {
           cout << " Today's date is = " << day << "/";
           cout << month << "/" << year << endl;
       }
}; // end of class de nition

int main()
{

    date today;
    today.getdata();
    today.display();
    return 0;
}
```

Output of the above program

```
enter the date ( dd-mm-year)
12 5 2007
Today's date is = 12/5/2007
```

PROGRAM 10.4

A program to illustrate the use of the simple arithmetic operations such as addition, subtraction, multiplication and division using a member function. These methods are defined within the scope of a class definition.

```
// member functions are de ned within the class de nition
#include <iostream>
using namespace std;
class sample {
    private :
       int x,y;
    public :
       void getinfo(){
          cout << " enter any two numbers ? " << endl;
          cin >> x >> y ;
       }
       void display(){
```

```
            cout << " x = " << x << endl;
            cout << " y = " << y << endl;
            cout << " sum = " << sum() << endl;
            cout << " dif = " << diff() << endl;
            cout << " mul = " << mult() << endl;
            cout << " div = " << div() << endl;
         }
         int sum(){
            return(x+y);
         }
         int diff(){
            return(x-y);
         }
         int mult(){
            return(x*y);
         }
          oat div(){
            return( ( oat)x/( oat)y);
         }
}; // end of class de nition
int main()
{
     sample obj1;
     obj1.getinfo();
     obj1.display();
     obj1.sum();
     obj1.diff();
     obj1.mult();
     obj1.div();
     return 0;
}
```

Output of the above program

```
enter any two numbers ?
1   2
x = 1
y = 2
sum = 3
dif = -1
mul = 2
div = 0.5
```

PROGRAM 10.5

A program to illustrate the use of the simple arithmetic operations such as addition, subtraction, multiplication and division using a member function. These are defined out of the scope of a class definition.

```
//methods are de ned out of the class de nition
#include <iostream>
using namespace std;
class sample {
   private :
       int x;
       int y;
   public :
       void getinfo();
       void display();
       int sum();
       int diff();
       int mult();
        oat div();
```

```
}; // end of class de nition
void sample :: getinfo()
{
    cout << " enter any two number ? " << endl;
    cin >> x >> y ;
}

void sample :: display()
{
    cout << " x = " << x << endl;
    cout << " y = " << y << endl;
    cout << " sum = " << sum() << endl;
    cout << " dif = " << diff() << endl;
    cout << " mul = " << mult() << endl;
    cout << " div = " << div() << endl;
}

int sample :: sum()
{
    return(x+y);
}
int sample :: diff()
{
    return(x-y);
}

int sample :: mult()
{
    return(x*y);
}

 oat sample :: div()
{
    return(( oat)x/ ( oat)y);
}

int main()
{
    sample obj1;
    obj1.getinfo();
    obj1.display();
    obj1.sum();
    obj1.diff();
    obj1.mult();
    obj1.div();
    return 0;
}
```

Output of the above program

```
enter any two number?
1 3
x = 1
y = 3
sum = 4
dif = -2
mul = 3
div = 0.333333
```

PROGRAM 10.6

A program to find the area of a circle whose radius is given as input using an OOP technique.

```
//  nding the area of a circle
#include <iostream>
#include <cmath>
using namespace std;
const  oat pi = 3.14159;
class circle {
    private :
         oat radius,area;
    public :
       void get_radius();
       void  nd_area();
       void display_area();
}; // end of class de nition

void circle ::get_radius()
{
     cout << "enter radius of a circle \n";
     cin >> radius;
}
void circle ::  nd_area()
{
       area = pi * radius * radius;
}

void circle :: display_area()
{
    cout << endl;
    cout << " radius  = " << radius << '\n';
    cout << " Area of a circle = " << area << '\n';
}

int main()
{
    circle obj;
    obj.get_radius();
    obj. nd_area();
    obj.display_area();
    return 0;
}
```

Output of the above program

```
enter radius of a circle
10
radius  = 10
Area of a circle = 314.159
```

PROGRAM 10.7

A program to find the sum of the following series using an OOP technique.

sum = 1 + 3 + 5 + 7 + ... n

```
//summing of series
// sum = 1 + 3 + 5 ... n
#include <iostream>
using namespace std;
class abc {
    private:
        int n;
    public:
        int sum(int n);
};
```

```
int abc :: sum (int n)
{
    int temp = 0;
    for (int i = 1; i <= n; i += 2)
        temp += i;
    return (temp);
}

int main()
{
    int max;
    abc obj;
    cout << "enter a value for n ";
    cin >> max;
    int total = obj.sum(max);
    cout <<"Summing of series \n";
    cout <<" 1 + 3 + 5 + ... " << max << " = " << total;
    cout << endl;
    return 0;
}
```

Output of the above program

```
enter a value for n
9
Summing of series
1 + 3 + 5 + ... 9 = 25
```

PROGRAM 10.8

A program to find the factorial of a given number using an OOP technique.

```
// nding factorial of a number
#include <iostream>
using namespace std;
class abc {
    private:
        int n;
    public:
        long int fact (int n);
};

long int abc :: fact(int n)
{
    long int temp = 1;
    for (int i = 1; i <= n; ++i)
        temp *= i;
    return (temp);
}

int main()
{
    abc obj;
    int max;
    cout << " enter a number \n";
    cin >> max;
    long int total = obj.fact(max);
    cout << "Factorial " << max << "! = " << total;
    cout << endl;
    return 0;
}
```

Output of the above program

```
enter a number
5
Factorial 5! = 120
```

PROGRAM 10.9

A program to solve a quadratic equation using an OOP technique.

```
//solution of quadratic equation using OOP
#include <iostream>
#include <cmath>
using namespace std;
class equation {
    private :
          oat a,b,c;
    public :
         void getinfo( oat a, oat b, oat c);
         void display();
         void equal( oat a,  oat b);
         void imag();
         void real( oat a,  oat b,  oat det);
}; // end of class de nition

void equation ::getinfo( oat aa,  oat bb,  oat cc)
{
    a = aa;
    b = bb;
    c = cc;
}
void equation :: display()
{
    cout << endl;
    cout << " a = " << a << '\t';
    cout << " b = " << b << '\t';
    cout << " c = " << c << endl;
}

void equation :: equal( oat a,  oat b)
{
     oat x;
    x = -b/(2*a);
    cout << " roots are equal = " << x <<endl;
}

void equation :: imag()
{
    cout << " roots are imaginary \n";
}

void equation :: real( oat a,  oat b,  oat det)
{
     oat x1,x2,temp;
    temp = sqrt(det);
    x1 = (-b+temp)/(2*a);
    x2 = (-b-temp)/(2*a);
    cout << " roots are real \n";
    cout << " x1 = " << x1 << endl;
    cout << " x2 = " << x2 << endl;
}

int main()
{
```

```
    class equation equ;
     oat a,b,c;
    cout << " enter three numbers \n";
    cin >> a >> b >> c;
    equ.getinfo(a,b,c);
    equ.display();
    if ( a == 0) {
        oat temp;
       temp = -c/b;
       cout << " linear roots = " << temp << endl;
    }
    else {
        oat det;
       det = b*b-4*a*c;
       if (det == 0)
          equ.equal(a,b);
       else if ( det < 0)
          equ.imag();
       else
          equ.real(a,b,det);
    }
    return 0;
} // end of main program
```

Output of the above program

```
enter three numbers
0  1  2
a = 0  b = 1  c = 2
linear roots = -2

enter three numbers
2  4  2
a = 2  b = 4  c = 2
roots are equal = -1
```

PROGRAM 10.10

A program to perform simple complex number arithmetic operations using an OOP technique.

```
// complex number operations using OOP
#include <iostream>
#include <cstdio>
using namespace std;
class complex {
   private :
        oat areal;
        oat aimag;
        oat breal;
        oat bimag;
   public :
       void getinfo( oat a,  oat,  oat c,  oat d);
       void display ();
       void menu();
       void add ( oat areal, oat aimag,  oat breal , oat bimag);
       void sub ( oat areal, oat aimag,  oat breal , oat bimag);
       void mul ( oat areal, oat aimag,  oat breal , oat bimag);
       void div ( oat areal, oat aimag,  oat breal , oat bimag);
};

void complex::getinfo(  oat x, oat y,  oat z , oat w)
{
```

```
    areal = x;
    aimag = y;
    breal = z;
    bimag = w;
}
void complex :: display()
{
    cout << "  rst complex number \n";
    cout << areal;
    if (aimag < 0)
       cout << "-i" << (-1)*aimag << endl;
    else
       cout << "+i"<< aimag << endl;
    cout << " second complex number \n";
    cout << breal;
    if (bimag < 0)
       cout << "-i" << (-1)*bimag << endl;
    else
       cout << "+i"<< bimag << endl;
}

void complex :: menu(void)
{
    cout << " complex number operations \n";
    cout << " menu () \n";
    cout << " a -> addition \n";
    cout << " s -> subtraction \n";
    cout << " m -> multiplication \n";
    cout << " d -> multiplication \n";
    cout << " q -> quit \n";
    cout << " option, please ? \n";
}

void complex :: add( oat areal,  oat aimag,  oat breal,  oat bimag)
{
     oat creal,cimag;
    creal = areal+breal;
    cimag = aimag+bimag;
    cout << " Addition of two complex numbers \n";
    cout << creal;
    if (cimag < 0)
       cout << "-i" << (-1)*cimag << endl;
    else
       cout << "+i"<< cimag << endl;
}

void complex :: sub( oat areal,  oat aimag,  oat breal,  oat bimag)
{
     oat creal,cimag;
    creal = areal-breal;
    cimag = aimag-bimag;
    cout << " Subtraction of two complex numbers \n";
    cout << creal;
    if (cimag < 0)
       cout << "-i" << (-1)*cimag << endl;
    else
       cout << "+i"<< cimag << endl;
}

void complex :: mul( oat areal,  oat aimag,  oat breal,  oat bimag)
{
     oat creal,cimag;
    creal = (areal*breal)-(aimag*bimag);
    cimag = (areal*bimag)+(aimag*breal);
    cout << " Multiplication of two complex numbers \n";
    cout << creal;
```

```
    if (cimag < 0)
        cout << "-i" << (-1)*cimag << endl;
    else
        cout << "+i"<< cimag << endl;
}

void complex :: div( oat areal,  oat aimag,  oat breal,  oat bimag)
{
     oat creal,cimag;
     oat temp;
    temp = (breal*breal)+(bimag*bimag);
    creal = ((areal*breal)+(aimag*bimag))/temp;
    cimag = ((breal*aimag)-(areal*bimag))/temp;
    cout << " Division of two complex numbers \n";
    cout << creal;
    if (cimag < 0)
        cout << "-i" << (-1)*cimag << endl;
    else
        cout << "+i"<< cimag << endl;
}

int main()
{
    complex comp;
     oat x,y,z,w;
    char ch;
    cout << "enter a  rst complex number \n";
    cin >> x >> y;
    cout << " enter a second complex number \n";
    cin >> z >> w;
    comp.getinfo(x,y,z,w);
    comp.display();
    comp.menu();
    while (( ch = getchar()) != 'q') {
        switch (ch) {
            case 'a' :
               comp.add(x,y,z,w);
               break;
            case 's' :
               comp.sub (x,y,z,w);
               break;
            case 'm' :
               comp.mul (x,y,z,w);
               break;
            case 'd' :
               comp.div (x,y,z,w);
               break;
        } // end of switch
    }
    return 0;
} // end of main program
```

Output of the above program

```
enter a  rst complex number
1 1
enter a second complex number
2 2
 rst complex number
1+i1
second complex number
2+i2
complex number operations
menu ()
```

```
a -> addition
s -> subtraction
m -> multiplication
d -> multiplication
q -> quit
option, please?

a
Addition of two complex numbers
3+i3

s
Subtraction of two complex numbers
-1-i1

m
Multiplication of two complex numbers
0+i4

d
Division of two complex numbers
0.5+i0a
q
```

PROGRAM 10.11

A program to read a set of characters from the keyboard and store it in the character array and display its contents onto the video screen using an OOP technique.

```
#include <iostream>
using namespace std;
class abc {
   public:
      char a[200];
      void getdata();
      void display();
};
void abc :: getdata()
{
   char ch;
   int i = 0;
   cout <<"enter a line of text and terminate with @\n";
   while ((ch = cin.get()) != '@') {
      a[i++] = ch;
   }
   a[i++] = '\0';
}
void abc :: display()
{
   cout << "contents of a character array \n";
   for (int i = 0; a[i] != '\0'; ++i)
      cout.put(a[i]);
}
int main()
{
   abc obj;
   obj.getdata();
   obj.display();
   return 0;
}
```

Output of the above program

```
enter a line of text and terminate with @
this is
a test
program
by Ravich
@

contents of a character array
this is
a test
program
by Ravich
```

PROGRAM 10.12

A program to read a set of characters from the keyboard and store it in the character array and find out the number of characters that are stored in the array; display its contents onto the video screen using an OOP technique.

```
//counting number of characters
#include <iostream>
using namespace std;
char a[200];
class abc {
    public:
       void getdata();
       void display();
       int count(char a[]);
};
void abc :: getdata()
{
    char ch;
    int i = 0;
    cout <<"enter a line of text and terminate with @\n";
    while ((ch = cin.get()) != '@') {
       a[i++] = ch;
    }
    a[i++] = '\0';
}
void abc :: display()
{
    cout << "contents of a character array \n";
    for (int i = 0; a[i] != '\0'; ++i)
       cout.put(a[i]);
}
int abc :: count(char a[])
{
    int i = 0;
    while (a[i] != '\0')
       ++i;
    return (i-1);
}
int main()
{
    abc obj;
    int total_ch;
    obj.getdata();
    obj.display();
    total_ch = obj.count(a);
```

```
    cout <<"\n Number of characters = " << total_ch;
    return 0;
}
```

Output of the above program

```
enter a line of text and terminate with @
this is
a test
@
contents of a character array
this is
a test
Number of characters = 15
```

10.7 ARRAY OF CLASS OBJECTS

An array is a user-defined data type whose members are homogeneous and stored in contiguous memory locations. For practical applications such as designing a large size of data base, arrays are very essential. The declaration of an array of class objects is very similar to the declaration of the array of structures in C++.

The general syntax of the array of class objects is:

```
class user_de ned_name {
    private :
       // methods
    public :
       // methods
    protected :
       // methods
};

class user_de ned_name object[MAX];
```

where, MAX is a user defined size of the array of class objects.

The various types of declarations of an array of class objects are illustrated below:

(1)

```
const int MAX = 200;
class employee {
    private :
       -------
       -------
    public :
       -------
       -------
};

class employee obj[MAX];
```

The class employee has been declared as an object of size 200

(2)

```
class library {
    private :
       -------
       -------
    public :
```

```
            -------
            -------
    } object [100];
```

C++ permits to declare the array of class objects on the class declaration itself.

(3)

A class can be declared without defining a user-defined name in the class tag.

```
    class {
        private :
            -------
            -------
        public :
            -------
            -------
     } library [100];
```

where `library` is an object of the class without tag name whose size is 100.

PROGRAM 10.13

A program to read students' particulars such as roll number, age, sex, height and weight from the keyboard and display the contents of the class on the screen. The class 'student_info' is defined as an array of class objects. This program shows how to create an array of class objects and how to access these data member and member functions in C++.

```
// array of class objects
#include <iostream>
using namespace std;
const int MAX = 100;
class student_info
{
  private :
     long int rollno;
     int age;
     char sex;
      oat height, weight;
  public :
     void getinfo();
     void disinfo();
}; // end of class de nition

void student_info :: getinfo()
{
     cout << " Roll no :";
     cin  >> rollno;
     cout <<" Age  :";
     cin >> age;
     cout << " Sex : ";
     cin >> sex;
     cout << " Height : ";
     cin >> height;
     cout << " Weight : ";
     cin >> weight;
}

void student_info :: disinfo ()
{
     cout << endl;
     cout << " Roll no  = " << rollno << endl;
```

```
     cout << " Age      = " << age << endl;
     cout << " Sex      = " << sex << endl;
     cout << " Height   = " << height << endl;
     cout << " Weight   = " << weight << endl;
}

int main()
{
    student_info  object[MAX]; // array of objects
    int i,n;
    cout << " How many students ? \n" << endl;
    cin >> n;
    cout << " enter the following imformation \n" << endl;
    for (i = 0; i <= n-1; ++i) {

        int j = i;
        cout << endl;
        cout << " record = " << j+1 << endl;
        object[i].getinfo();
    }
    cout << " contents of class \n";
    for (i = 0; i <= n-1; ++i) {
        object[i].disinfo();
    }
}
```

Output of the above program

```
How many students?
2

enter the following imformation
record = 1
Roll no  :  20071
Age      :  21
Sex      :  M
Height   :  170
Weight   :  56

record = 2
Roll no  :  20072
Age      :  20
Sex      :  F
Height   :  160
Weight   :  50

contents of class
Roll no  = 20071
Age      = 21
Sex      = M
Height   = 170
Weight   = 56

Roll no  = 20072
Age      = 20
Sex      = F
Height   = 160
Weight   = 50
```

10.8 POINTERS AND CLASSES

In Chapter 6 on pointers it has already been stated that a pointer is a variable which holds the memory address of another variable of any basic data types such as `int`, `oat` or sometimes an array. In the previous chapter, it has also been illustrated how a pointer can be used to hold the address of a structure variable too. The pointer variable is very much used to construct complex data bases using the data structures such as linked lists, double linked lists and binary trees.

So far, it has been shown that a data member and a member function of a class could be an ordinary data type such as int, float, char and even a class also. In this section, how a pointer variable can be declared as a member to a class is discussed.

The following declaration of creating an object is valid in C++.

```
class sample {
   private :
      int x;
       oat y;
      char s;
   public :
      void getdata();
      void display();
};
sample *ptr;
```

where `ptr` is a pointer variable that holds the address of the class object sample and consists of the three data members such as int `x`, float `y` and char `s`, and also holds member functions such as getdata() and display().

The pointer to an object of class variable will be accessed and processed in one of the following ways,

```
(*object name).member name = variable;
```

The parentheses are essential since the member of class period (.) has a higher precedence over the indirection operator (*). Or, the pointer to the member of a class can be expressed using dash (–) followed by the greater than sign (>).

```
object name -> member name = variable;
```

Following are valid declarations of using pointer to the member of a class.

Case 1 A member of class object can be accessed by the indirection operator which has been shown in the following program segment:

```
class student  {
    private :
       -------
       -------
    public :
       -------
       -------
 };  // end of class de nition

void main(void)
{
    student *ptr;
      -------
      -------
```

```
        (*ptr).data_member;
        (*ptr).member_function();
}
```

Case 2 A member of class object can be accessed by the pointer of a class operator which has been shown in the following program segment:

```
class student  {
   private :
      -------
      -------
   public :
      -------
      -------
};  // end of class de nition

int main()
{
    student *ptr;
    -------
    -------
    ptr->data_member;
    ptr->member_function();
    return 0;
}
```

PROGRAM 10.14

A program to assign some values to the member of class objects using a pointer structure operator (–>).

```
// pointers and classes 1.cpp
#include <iostream>
using namespace std;
class student_info  {
    private :
       long int rollno;
       int age;
       char sex;
        oat height;
        oat weight;
    public :
       void getinfo ();
       void disinfo ();
};  // end of class de nition

void student_info :: getinfo()
{
    cout << " Roll no :";
    cin  >> rollno;
    cout <<" Age  :";
    cin >> age;
    cout << " Sex : ";
    cin >> sex;
    cout << " Height : ";
    cin >> height;
    cout << " Weight : ";
    cin >> weight;
}

void student_info :: disinfo()
```

```
{
    cout << endl;
    cout << " Roll no  = " << rollno << endl;
    cout << " Age      = " << age << endl;
    cout << " Sex      = " << sex << endl;
    cout << " Height   = " << height << endl;
    cout << " Weight   = " << weight << endl;
}

int main()
{
    student_info *ptr; // ptr is an object of class student
    cout << " enter the following information " << endl;
    ptr->getinfo();
    cout << " \n contents of class " << endl;
    ptr->disinfo();
    return 0;
}
```

Output of the above program

```
enter the following information
Roll no  : 200710
Age      : 23
Sex      : M
Height   : 176
Weight:  67

contents of class
Roll no   =   200710
Age       =   23
Sex       =   M
Height    =   176
Weight    =   67
```

PROGRAM 10.15

A program to assign some values to the member of class objects using an indirection operator.

```
// pointers and classes 2.cpp
#include <iostream>
using namespace std;
class student_info  {
    private :
       long int rollno;
       int age;
       char sex;
        oat height;
        oat weight;
    public :
       void getinfo();
       void disinfo();
};  // end of class de nition

void student_info :: getinfo()
{
    cout << " Roll no :";
    cin  >> rollno;
    cout <<" Age  :";
    cin >> age;
    cout << " Sex : ";
```

```
    cin >> sex;
    cout << " Height : ";
    cin >> height;
    cout << " Weight : ";
    cin >> weight;
}

void student_info :: disinfo()
{
    cout << endl;
    cout << " Roll no  = " << rollno << endl;
    cout << " Age      = " << age << endl;
    cout << " Sex      = " << sex << endl;
    cout << " Height   = " << height << endl;
    cout << " Weight   = " << weight << endl;
}

int main()
{
    student_info *ptr; // ptr is an object of class student
    cout << " enter the following information " << endl;
    (*ptr).getinfo();
    cout << " \n contents of class " << endl;
    (*ptr).disinfo();
    return 0;
}
```

Output of the above program

```
enter the following information
Roll no   :  200711
Age       :  22
Sex       :  F
Height    :  156
Weight    :  71

contents of class
Roll no   = 200711
Age       = 22
Sex       = F
Height    = 156
Weight    = 71
```

PROGRAM 10.16

A program to find the distance between the given two points using the pointer to class objects technique.

```
// classes and pointers
#include <iostream>
#include <cmath>
using namespace std;
class  point {
    private :
       int x,y;
    public :
       point (int xnew,int ynew);
       inline int getx() {
           return(x);
       }
       inline int gety() {
           return(y);
```

```
        }
         oat  nddist(point a, point b);
}; // end of class de nition

point :: point (int xnew, int ynew)
{
    x = xnew;
    y = ynew;
}

 oat point ::  nddist (point a, point b)
{
     oat temp;
    temp = ((b.y-a.y)*(b.y-a.y)) + ((b.x-a.x)*(b.x-a.x));
    return (sqrt(temp));
}
int main()
{
    point aobj(4,3),bobj(0,-1);
    point *aptr = &aobj;
    point *bptr = &bobj;
    aptr->getx();
    aptr->gety();
    bptr->getx();
    bptr->gety();
     oat value;
    value = aptr-> nddist(aobj,bobj);
    cout << " distance between two points =" << value << endl;
    return 0;
}
```

Output of the above program

```
distance between two points = 5.65685
```

10.9 UNIONS AND CLASSES

In the previous chapter, a union has been defined as a user-defined data type whose size is sufficient to contain one of its members. At most, one of the members can be stored in a union at any time. A union is also used for declaring classes in C++. The members of a union are public by default.

A union allows to store its members only one at a time. A union may have member functions including constructors and destructors, but not virtual functions. A union may not have base class. An object of a class with a constructor or a destructor or a user-defined assignment operator cannot be a member of a union. A union can have no static data member. The virtual function, constructor, destructor and static members are discussed in subsequent chapters.

The general syntax of a union declaration is:

```
union user_de ned_name {
   private :
      //methods
   public :
      // methods
   protected :
      // methods
};
user_de ned_name object;
```

It is possible in C++ to declare a union without a user-defined name or a union tag that is called as an anonymous union.

The syntax of the anonymous union declaration is:

```
union {
  // methods
  // methods
};
```

The names of the members of an anonymous union must be distinct from other names. A global anonymous union must be declared static. An anonymous union may not have protected or private members. An anonymous union may not have a member function also.

For example, the following anonymous union declarations are invalid.

Case 1 An anonymous union may not have private member. Hence, the following declaration of union gives compiler error message.

```
union {
   private :
      int x;
       oat y;
   public :
      void getvalue();
      void display();
};
```

Case 2 An anonymous union may not have a member function. Hence, the following declaration is invalid and gives error message while compiling.

```
union {
   public :
      int x;
       oat y;
      void getdata();
      void display();
};
```

The following declaration is valid. A union with a name or union tag may have member functions.

```
union sample {
   public :
      int a;
      char name;
      void display();
      void sum();
};
```

Note that a union with name or union tag may have private members also. For example, the following declaration is valid:

```
union sample {
   private :
      int x;
       oat y;
   public :
      void get();
      void display();
};
```

PROGRAM 10.17

A program to demonstrate how to define a union as a class object data type. This program reads the values of the data members from the keyboard and displays the contents of the union on the screen.

```
// unions and classes
#include <iostream>
using namespace std;
union sample  {
    private :
       int x;
        oat y;
    public :
       void getinfo ();
       void disinfo ();
};  // end of class de nition

void sample :: getinfo ()
{
     cout << " value of x ( in integer ) :";
     cin  >> x;
     cout <<" value of y  ( in  oat)  :";
     cin >> y;
}

void sample :: disinfo ()
{
     cout << endl;
     cout << " x  = " << x << endl;
     cout << " y  = " << y << endl;
}

int main()
{
    sample obj;
    cout << " enter the following information " << endl;
    obj.getinfo();
    cout << " \n content of union " << endl;
    obj.disinfo();
    return 0;
}
```

Output of the above program

```
enter the following information
value of x ( in integer ) : 34
value of y  ( in  oat)  : -12.45

content of union
x  = 13107
y  = -12.45
```

10.10 CLASSES WITHIN CLASSES (NESTED CLASS)

C++ permits declaration of a class within another class. A class declared as a member of another class is called as a nested class or a class within another class. The name of a nested class is local to the enclosing class. The nested class is in the scope of its enclosing class.

The general syntax of the nested class declaration is shown below.

```
class outer_class_name {
      private :
          // data
      protected :
          //data
          // methods
      public :
          // methods
      class inner_class_name {
           private :
               // data of inner class
           public :
               // methods of inner class
      }; // end of inner class declaration
}; // end of outer class declaration

outer_class_name object1;
outer_class_name :: inner_class_name object2;
```

Note that simply declaring a class nested in another does not mean that the enclosing class contains an object of the enclosed class. Nesting expresses scoping, not containment of such objects.

For example, a nested class declaration is shown in the following program segments.

Example 1

```
class student_info {
   private :
      char name[20];
      long int rollno;
      char sex;
   public :
      student_info(char *na,long int rn,char sx);
      void display();
      class date {
          private :
             int day;
             int month;
             int year;
          public :
             date (int dy, int mh, int yr);
             void show_date();
      };     // end of date class declaration
};// end of student_info class declaration
```

Example 2

```
class student_info {
   private :
      char name[20];
      long int rollno;
      char sex;
   public :
      student_info(char *na,long int rn,char sx);
      void display();
      class date {
```

```
        private :
           int day;
           int month;
           int year;
        public :
           date (int dy, int mh, int yr);
           void show_date();
           class age_class {
               private :
                  int age;
               public :
                  age_class (int age_value);
                  void show_age();
        }; // end of age_class;
    };  // end of date class declaration
};// end of student_info class declaration
```

Member functions of a nested class have no special access to members of an enclosing class; they object the usual access rules. Member functions of an enclosing class have no special access to member of a nested class; they obey the usual access rules.

The following program segment illustrates how the member functions of the nested classes are accessed.

```
class outer {
   int a;
   void outer_funt(int b);
   class inner {
       int x;
       void inner_funt(int y);
    };
};
outer obj1;  // creating an object for the outer class
outer::inner::obj2;//creating an object for the inner class
obj1.outer_funt(); // accessing an outer class member function
obj2.inner.funct(); // accessing of inner class member function
```

When a class is declared as a member of another class, it contains only the scoping of the outer class. The object of an outer class does not contain the object of the inner class.

The member function of the outer class can be defined as.

```
void outer::outer_funt(int b);
{
    // methods
}
The member function of the inner class is de ned as.
void outer :: inner:: inner_funt( int y);
{
  // methods
}
```

The following declaration is an invalid way of calling an inner member function of the nested class.

```
outer obj1;  // creating an object for the outer class
outer:: inner::obj2;//creating an object for the inner class
obj1::obj2::inner_funct(); // error
```

Nesting expresses only scoping for the inner class, not the containment of such object.

PROGRAM 10.18

A program to define a nested class 'student_info' which contains data members such as name, roll number and sex, and also consists of one more class 'date', whose data members are day, month and year. The values of the student_info are to be read from the keyboard and the contents of the class have to be displayed on the screen. This program shows how to create a nested class and their objects, and also how to access these data members and member functions in C++.

```
// classes within classes (nested classes) demonstration
#include <iostream>
#include <string>
using namespace std;
class student_info {
   private :
      char name[20];
      long int rollno;
      char sex;
   public :
      student_info(char *na, long int rn, char sx);
      void display();
      class date {
          private :
          int day,month,year;
      public :
          date (int dy,int mh, int yr);
          void show_date();
      };  // end of date class declaration
};// end of student_info class declaration

student_info :: student_info(char *na,long int rn,char sx)
{
    strcpy (name,na);
    rollno = rn;
    sex = sx;
}

student_info::date :: date(int dy, int mh, int yr)
{
    day = dy;
    month = mh;
    year = yr;
}

void student_info:: display()
{
    cout << " student's_name Rollno sex  date_of_birth(dd-mm-yr) \n";
    cout << " ------------------------------------------------- " << endl;
    cout << name << "     " ;
    cout << rollno << "     ";
    cout << sex << "     ";
}

void student_info::date::show_date()
{
    cout << day << '/' << month << '/' << year << endl;
    cout << " -------------------------------" << endl;
}

int main()
{
    student_info obj1("Sampath Reddy",200710,'M');
```

```
    student_info::date obj2(13,7,94);
    obj1.display();
    obj2.show_date();
    return 0;
}
```

Output of the above program

```
student's_name Rollno  sex  date_of_birth(dd-mm-yr)
-----------------------------------------------
Sampath Reddy 200710   M     13/7/94
-----------------------------------------------
```

PROGRAM 10.19

A program to define a nested class 'student_info' which contains data members such as name, roll number and sex, and also consists of one more class 'date', whose data members are day, month and year. Again, the class is defined with one more class 'age_class' whose data member is age. The values of the student_info are read from the keyboard and contents of the class have to be displayed onto the screen. This program shows how to create a nested class and their objects and also how to access these data members and member functions in C++.

```
// classes within classes (nested classes) demonstration
// program 2.cpp
#include <iostream>
#include <string>
using namespace std;
class student_info {
    private :
        char name[20];
        long int rollno;
        char sex;
    public :
        student_info(char *na,long int rn,char sx);
        void display();
    class date {
        private :
            int day,month,year;
        public :
            date (int dy, int mh, int yr);
            void show_date();
        class age_class {
            private :
                int age;
            public :
                age_class (int age_value);

                void show_age();
        }; // end of age_class;
    };  // end of date class declaration
};// end of student_info class declaration

student_info :: student_info(char *na,long int rn,char sx)
{
       strcpy (name,na);
       rollno = rn;
       sex = sx;
}

student_info::date :: date(int dy, int mh,int yr)
{
```

```
        day = dy;
        month = mh;
        year = yr;
}
student_info::date :: age_class::age_class (int age_value)
{
        age = age_value;
}

void student_info:: display()
{
    cout << " student's name   Roll_no  sex  date of birth   age \n";
    cout << " ------------------------------------------------------" << endl;
    cout << name << "     ";
    cout << rollno << "     ";
    cout << sex << "     ";
}

void student_info::date::show_date()
{
    cout << day << '/' << month << '/' << year << '\t';
}

void student_info::date::age_class::show_age()
{
    cout << age << endl;
    cout << " --------------------------------- " << endl;
}

int main()
{
    student_info obj1("Suhail Ahmed",20071,'M');
    student_info::date obj2(31,9,1990);
    student_info::date::age_class obj3(17);
    obj1.display();
    obj2.show_date();
    obj3.show_age();
    return 0;
}
```

Output of the above program

```
Student's name   Roll_no sex date of birth   age
-----------------------------------------------
Suhail Ahmed     20071    M  31/9/1990       17
-----------------------------------------------
```

PROGRAM 10.20

A program to define an array of nested class 'student_info' which contains data members such as name, roll number and sex, and also consists of one more class 'date', whose data members are day, month and year. Again, the class is defined with one more class 'age_class' whose data member is age. The values of the student_info are read from the keyboard and contents of the class have to be displayed on the screen. This program shows how to create an array of nested class and their objects, and also how to access these data members and member functions in C++.

```
// array of nested classes  objects
#include <iostream>
#include <string>
```

```
using namespace std;
const int MAX = 100;
class student_info {
    private :
       char name[20];
       long int rollno;
       char sex;
    public :
       void getbase();
       void display();
    class date {
       private :
          int day,month,year;
       public :
          void getdate();
          void show_date();
       class age_class {
          private :
             int age;
          public :
             void getage ();
             void show_age();
       }; // end of age_class;
    };  // end of date class declaration
};// end of student_info class declaration

void student_info :: getbase()
{
    cout << " enter a name : ";
    cin >> name;
    cout << " roll no :";
    cin >> rollno;
    cout << " sex :";
    cin >> sex;
}
void student_info::date :: getdate()
{
    cout << " enter a date of birth ( dd-mm-yr) ";
    cin >> day >> month >> year;
}
void student_info::date ::age_class:: getage ()
{
    cout << " enter an age :";
    cin >>age;
}
void student_info:: display()
{
    cout << name << "     " <<'\t';
    cout << rollno << "     ";
    cout << sex << "     ";
}
void student_info::date::show_date()
{
    cout << day << '/' << month << '/' << year << '\t';
}
void student_info::date::age_class::show_age()
{
    cout << age << endl;
}
int main()
{
    student_info obj1[MAX];
    student_info::date obj2[MAX];
    student_info::date::age_class obj3[MAX];
    int n,i;
    cout << " how many students ?\n";
```

```
    cin >> n;
    cout << " enter the following inoformation \n";
    for (i=0; i<= n-1; ++i) {
        int j = i+1;
        cout << " \n object : " << j << endl;
        obj1[i].getbase();
        obj2[i].getdate();
        obj3[i].getage();
    }
    cout << " Contents of the array of nested classes \n";
    cout << " ------------------------------------------";
    cout  << endl;
    cout << " student's name   Roll_no  sex  date of birth age";
    cout << endl;
    cout << " ------------------------------------------------";
    cout << endl;
    for (i=0; i<= n-1; ++i) {
        obj1[i].display();
        obj2[i].show_date();
        obj3[i].show_age();
    }
    cout << "------------------------------------------------";
    cout << endl;
    return 0;
}
```

Output of the above program

```
how many students?
2
enter the following information

object : 1
enter a name: Suhail
roll no: 20071
sex: M
enter a date of birth ( dd-mm-yr) 12  3  1990
enter an age: 17

object: 2
enter a name:  Sudheer
roll no: 20072
sex: M
enter a date of birth ( dd-mm-yr) 21  5  1991
enter an age: 16

Contents of the array of nested classes
---------------------------------------------------------------
student's name      roll_no     sex     date of birth     age
---------------------------------------------------------------
Suhail               20071       M        12/3/1990        17
Sudheer              20072       M        21/5/1991        16
---------------------------------------------------------------
```

10.11 SUMMARY OF STRUCTURES, CLASSES AND UNIONS

The access control and constraints of structures, classes, and unions are summarised in Table 10.1.

Table 10.1

Structures	*Classes*	*Unions*
class-key is struct	class-key is class	class-key is union
Default access is public	Default access is private	Default access is public
No usage constraints	No usage constraints	Use only one member at a time

REVIEW QUESTIONS

1. What is an object and how objects can be defined in C++?
2. List the pros and cons of object-oriented programming over the structured programming.
3. What are the salient features of the object-oriented design in software engineering?
4. What is data hiding and data encapsulation?
5. What is an inheritance? What are the few applications of inheritance?
6. Define polymorphism.
7. What is a class? How is a class different from an object?
8. What are the syntactic rules governing the definition of a class data type?
9. Explain the following with respect to the object-oriented paradigm.
   (i) private
   (ii) public
   (iii) protected
10. In what way a member function is different from a conventional user-defined function?
11. What is a scope resolution operator? How is it useful for defining the data member and member function of a class?
12. Explain the difference between a data member of a class and the conventional variables in C++.
13. How is the member function of a class accessed in C++?
14. What is an array of class objects? How are the array of class objects defined in C++?
15. In what way is a union data type useful for constructing a class object in C++?
16. Explain the syntactic rules of defining the union data type using a class object.
17. What is a nested class? How is a nested class is defined and declared in C++?
18. List the merits and demerits of declaring a nested class in C++.
19. How is a pointer variable used to declare a member function of a class?
20. Explain the pointer techniques to access the member functions of a class objects.

CONCEPT REVIEW PROBLEMS

1. Determine the output of each of the following program when it is executed.
   (a)

```
#include <iostream>
#include <iomanip>
using namespace std;
struct abc {
   void display();
```

```
};

void abc :: display()
{
   for (int i = 1; i <= 5; ++i) {
      for (int j = 1; j <= i; ++j)
         cout << setw(3) << j;
      cout << endl;
   }
}

int main()
{
    abc obj;
    obj.display();
    return 0;
}
```

(b)

```
#include <iostream>
#include <iomanip>
using namespace std;
struct abc {
   void display();
};

void abc :: display()
{
   for (int i = 1; i <= 5; ++i) {
      for (int j = i; j >= 1; --j)
         cout << setw(3) << j;
      cout << endl;
   }
}

int main()
{
    abc obj;
    obj.display();
    return 0;
}
```

(c)

```
#include <iostream>
#include <iomanip>
using namespace std;
struct abc {
   void display();
};

void abc :: display()
{
   for (int i = 5; i >= 1; --i) {
      for (int j = i; j >= 1; --j)
         cout << setw(3) << j;
      cout << endl;
   }
```

```
}

int main()
{
    abc obj;
    obj.display();
    return 0;
}
```

(d)

```
#include <iostream>
#include <iomanip>
using namespace std;
struct abc {
   void display();
};

void abc :: display()
{
   for (int i = 5; i >= 1; --i) {
      for (int j = 1; j <= i; ++j)
         cout << setw(3) << j;
      cout << endl;
   }
}

int main()
{
    abc obj;
    obj.display();
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
const int n = 10;
class abc {
   public:
      int sum(int n);
};

int abc :: sum (int n)
{
   int temp = 0;
   for (int i = 0; i <= n; i += 2)
        temp += i;
   return (temp);
}

int main()
{
   abc obj;
   int total = obj.sum(n);
   cout << "total = " << total << endl;
   return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
const int n = 10;
class abc {
   public:
      int sum(int a[], int n);
};

int abc :: sum (int a[], int n)
{
   int temp = 0;
   for (int i = 0; i <= n-1; ++i) {
   if ( a[i] % 2 == 0)
      temp += i;
   }
   return (temp);
}

int main()
{
   abc obj;
   int a[] = {1,2,3,4,5,6,7,8,9,10};
   int total = obj.sum(a,n);
   cout << "total = " << total << endl;
   return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
class abc{
   private:
      int a;
   public:
      int a;
      void display();
};
void abc::display()
{
      cout << " value of a = " << a <<"\n";
}
int main()
{
    abc obj;
    obj.display();
    return 0;
}
```

2. What will be the output of each of the following programs when it is executed?

(a)

```
#include <iostream>
using namespace std;
struct abc {
    public:
       void dispabc();
```

```
        struct xyz {
           public:
              void dispxyz();
        };
};
void abc :: dispabc()
{
    cout << "Hello ";
}
void abc :: xyz :: dispxyz()
{
    cout << "C++ world \n";
}
int main()
{
    abc obj1;
    abc::xyz obj2;
    obj1.dispabc();
    obj2.dispxyz();
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
struct abc {
   public:
      void dispabc();
      struct xyz {
         public:
           void dispxyz();
           class pqr {
              public:
                 void dispqr();
           };
      };
};
void abc :: dispabc()
{
    cout << "Hello \n";
}
void abc :: xyz :: dispxyz()
{
    cout << "C++ \n";
}
void abc :: xyz :: pqr :: dispqr()
{
    cout << "world \n";
}
int main()
{
    abc obj1;
    abc::xyz obj2;
    abc::xyz::pqr obj3;
    obj1.dispabc();
    obj2.dispxyz();
```

```
    obj3.dispqr();
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
struct abc {
   private:
      void dispabc();
      struct xyz {
          void dispxyz();
      };
};
void abc :: dispabc()
{
    cout << "Hello \n";
}
void abc :: xyz :: dispxyz()
{
    cout << "C++ \n";
}
int main()
{
    abc obj1;
    abc::xyz obj2;
    obj1.dispabc();
    obj2.dispxyz();
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
struct abc {
   void dispabc();
   private:
      struct xyz {
          void dispxyz();
      };
};
void abc :: dispabc()
{
    cout << "Hello \n";
}
void abc :: xyz :: dispxyz()
{
    cout << "C++ \n";
}
int main()
{
    abc obj1;
    abc::xyz obj2;
    obj1.dispabc();
    obj2.dispxyz();
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
struct abc {
    void dispabc();
};
void abc :: dispabc()
{
    cout << "Hello \n";
}
int main()
{
    abc obj1;
    obj1.dispabc();
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
struct abc {
    private:
       void dispabc();
};
void abc :: dispabc()
{
    cout << "Hello \n";
}
int main()
{
    abc obj1;
    obj1.dispabc();
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
class abc{
   public:
      int a;
      void setdata(const int a);
      void display();
};
void abc::setdata(const int aa)
{
    a = aa;
}
void abc::display()
{
    cout << " value of a = " << ++a <<"\n";
}
int main()
{
    abc obj;
    obj.setdata(10);
```

```
    obj.display();
    return 0;
}
```

(h)

```
#include <iostream>
using namespace std;
static class abc{
   public:
      int a;
      void display();
};
void abc::display()
{
    cout << " value of a = " << a <<"\n";
}
int main()
{
    abc obj;
    obj.display();
    return 0;
}
```

3. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
static int i;
class abc
{
    public:
       void display();
};
void abc :: display()
{
    cout << "i = " << i;
    cout << "\n";
}
int main()
{
    abc obj;
    obj.display();
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
volatile int i;
class abc
{
    public:
       void display();
};
void abc :: display()
{
```

```
    cout << "i = " << i;
    cout << "\n";
}
int main()
{
    abc obj;
    obj.display();
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
const int i = 10;
class abc
{
    public:
       void display();
};
void abc :: display()
{
    cout << "i = " << ++i;
    cout << "\n";
}
int main()
{
    abc obj;
    obj.display();
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
const int i;
class abc
{
    public:
       void display();
};
void abc :: display()
{
    cout << "i = " << i;
    cout << "\n";
}
int main()
{
    abc obj;
    obj.display();
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
class abc
{
```

```
        private:
            int i;
        public:
            void display();
};
void abc :: display()
{
    cout << "i = " << i;
    cout << "\n";
}
int main()
{
    abc obj;
    obj.display();
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
class abc {
  public:
     void display()
     {
       cout << "class tag \n";
     }
} obj;
int main()
{
    obj.display();
    return 0;
}
```

PROGRAMMING EXERCISES

1. Write an object-oriented program in C++ that prints the factorial of a given number.
2. Write an object-oriented program in C++ that prints whether a given number is prime or not.
3. Write an object-oriented program in C++ to read any five real numbers and print the average.
4. Write an object-oriented program in C++ to generate a Fibonacci series of 'n' numbers, where n is defined by a programmer.
   (The series should be: 1 1 2 3 5 8 13 21 32 and so on.)
5. Write an object-oriented program in C++ to find the sum of the following series:
   (a) sum = 1 + 2 + 3 + ... + n
   (b) sum = 1 + 3 + 5 + ... + n
   (c) sum = 1 + 2 + 4 + ... + n
   (d) sum = $1-\frac{1}{1!}+\frac{2}{2!}-\frac{3}{3!}+\cdots$
   (e) sum = $x+\frac{x^2}{2!}+\frac{x^4}{4!}\cdots\frac{x^n}{n!}$

(f) sum = $x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots \frac{x^n}{n!}$

(g) sum = $1^2 + 2^2 + 3^2 + 3^2 + 4^2 + \cdots + n^2$

(h) sum = $1^3 + 2^3 + 3^3 + 4^3 + \cdots + n^3$

(i) sum = $1 + 2^2 + 4^2 + \cdots + n^2$

(j) sum = $1 + 3^2 + 5^2 + \cdots + n^2$

6. Write an object-oriented program in C++ to generate the following figures:

(i)

```
$
$ $
$ $ $
$ $ $ $
$ $ $ $ $
$ $ $ $ $ $
$ $ $ $ $ $ $
$ $ $ $ $ $ $ $
$ $ $ $ $ $ $ $ $
```

(ii)

```
9
9 8
9 8 7
9 8 7 6
9 8 7 6 5
9 8 7 6 5 4
9 8 7 6 5 4 3
9 8 7 6 5 4 3 1
9 8 7 6 5 4 3 2 1
```

(iii)

```
* * * * * * * * *
* * * * * * * *
* * * * * * *
* * * * * *
* * * * *
* * * *
* * *
* *
*
```

(iv)

```
! $ ! $ ! $ ! $ !
$ ! $ ! $ ! $ !
! $ ! $ ! $ !
$ ! $ ! $ !
! $ ! $ !
$ ! $ !
! $ !
$ !
!
```

7. Write an object-oriented program in C++ to generate the following pyramid of numbers.

```
0
1 0 1
2 1 0 1 2
3 2 1 0 1 2 3
4 3 2 1 0 1 2 3 4
5 4 3 2 1 0 1 2 3 4 5
6 5 4 3 2 1 0 1 2 3 4 5 6
```

8. Write an object-oriented program in C++ to read an integer number and find out the sum of all the digits until it comes to a single digit. For example,

(i)
```
n = 1256
sum = 1+2+5+6 = 14
sum = 1+4 = 5
```

(ii)
```
n = 7896
sum = 7+8+9+6 = 30
sum = 3+0 = 3
```

9. Write an object-oriented program in C++ to read a number n, and print it out digit by digit as a series of words. For example, the number 756 should be printed as "Seven Five Six".

10. Write an object-oriented program in C++ to read a set of numbers up to n, where n is defined by the programmer and print the contents of the array in the reverse order.

For example, for n = 4, the set
`26 56 51 123 would be as`
`123 51 56 26`

11. Write an object program in C++ to read n numbers, where n is defined by the programmer and find the average of the non negative integer numbers and the deviation of the numbers.

12. Write an object oriented program in C++ to read a set of numbers and store it as an one dimensional array; again read a number `'d'` and check whether the number `'d'` is present in the array. If it is so, print out how many times the number `d` is repeated in the array.

13. Write an object oriented program in C++ to read a set of numbers and store it as a one-dimensional array; again read a number n, and check whether it is present in the array. If it is so, print out the position of n in the array and also check whether it repeats in the array.

14. Write an object-oriented program in C++ to read a set of numbers and store it as a one-dimensional array and find out the largest and the smallest number. Find out the difference between the two numbers. Using the difference, find the deviation of the numbers of the array.

15. Write an object-oriented program in C++ to read a set of numbers to store it as a one-dimensional array; copy the elements in another array B in the reverse direction; find the sum of the individual elements of array A and array B; store the results in an another array C and display all the three arrays.

16. Write an object-oriented program in C++ to read a four digit positive integer number n and generate all the possible permutations of those numbers digit by digit.
For example, for `n` = 7812 the permutations are
7821
8721
8712
2871
2817
(**Hint:** Read a number n and separate a number digit by digit and store it in an array, and then generate a permutation).

17. Write an object-oriented program in C++ to read a two-dimensional square matrix A, and display its transpose.

18. Write an object oriented-program in C++ to read a two-dimensional array and find the sum of the elements row wise and column wise separately, and display the sums of the rows and columns.

19. Write an object-oriented program in C++ to generate a magic square A, where the sum of the elements in the row wise and column wise are the same.

20. Write an object-oriented program in C++ to read a set of lines and find out the number of characters, words, and lines in a given text.

21. Write an object-oriented program in C++ to read a line and find out the number of vowels (a, e, i, o, u) and consonants present in the given line.

22. Write an object-oriented program in C++ to read a set of lines from the `stdin` and print out the longest line.

23. Write an object-oriented program to read student's name and his average marks. If a student gets less than 40 then declare that he has failed or else he passed. Prepare a computer list to give the list of names in alphabetical order separately for passed and failed students.

24. Write an object-oriented program to perform trigonometric operations on the complex numbers.

25. Write an object-oriented program to read a set of lines from stdin and store them in an array A; again read a string S from the stdin and check whether the given string S is in the array A. If it is so, print that line and also how many times it repeats in the array A.

26. Write an object-oriented program to read a set of lines from `stdin` and store them in an array A;

again read a string `S` from the `stdin` and check whether the given string `S` in the array `A`. If it is, remove string `S` from the array `A` and print the updated array on the `stdout`. For example,

```
A = concatenate
S = cat
```

The updated `a` is `conenate`.

27. Write an object-oriented program to read a set of lines from `stdin` and store them in an array `A`; again read strings `S1` and `S2` from the `stdin` and check whether the given string `S1` is in the array `A`. If it does, replace the string `S1` with string `S2` and print the updated array. For example,

```
A  = concatenate
S1 = cat
S2 = 123
```

The updated `A` is `con123enate`

28.(a) Develop an object-oriented program in C++ to read the following information from the keyboard:

employee name
employee code
designation
years of experience
age

(b) Construct an object-oriented data base to carry out the following methods:

(i) Build a master table
(ii) List a table
(iii) Insert a new entry
(iv) Delete old entry
(v) Edit an entry
(vi) Search for a record that to be printed
(vii) Sort entries

29. Develop an object-oriented program in C++ to create a data base of the following items.

name of the patient
sex
age
ward number
bed number
nature of the illness
date of admission

Your program should have the facilites as listed in 28(b).

30. Develop an object-oriented program in C++ to create a pay roll system of an organisation. The following information may be read from the keyboard.

employee name
employee code
designation
account number
date of joining
basic pay
DA, HRA and CCA
deductions like PPF, GPF, CPF, LIC, NSS, NSC, etc.

Your program should have the facilities as listed in 28(b).

31. Develop an object-oriented program in C++ to prepare the mark sheet of a University examination.

The following items may be read from the keyboard:
- name of the student
- roll number
- subject code
- subject name
- internal marks
- external marks

Your program should have the facilities as listed in 28(b).

32. Develop an object-oriented program in C++ to create a library information system containing the following for all books in the library:
- accession number
- name of the author
- title of the book
- year of publication
- publisher's name
- cost of the book

Your program should have the facilities as listed in 28(b).

33. Develop an object-oriented program in C++ to create a data base of the personnel information system having the following information:
- name
- date of birth
- blood group
- height
- weight
- insurance policy number
- contact address
- telephone number
- driving licence number, etc.

Your program should have the facilities as enumerated in 28(b).

Special Member Functions

Chapter 11

This chapter mainly deals with the special member functions which are used for initialising and destroying the objects of a class. The various topics of the special member functions such as constructors, default constructors, copy constructors and destructors are explained. This chapter also presents how a friend of a class can be declared, defined and accessed in a program; how an object can be created and destroyed dynamically using the memory allocation operators new and delete.

11.1 INTRODUCTION

The special member functions are a set of functions that can be declared only as class members and invoked automatically by the C++ compiler. These functions affect the way objects of a given class are created, destroyed, copied, and converted into objects of other types. Another important property of many of these functions is that they can be called implicitly (by the compiler). The following is a list of the special member functions that are defined, declared and used in C++:

(1) Constructors
(2) Destructors
(3) Conversion functions
(4) Operator new function
(5) Operator delete function
(6) Assignment operator (operator=) function

(1) Constructors The constructors are the special member functions that are used in class objects to enable automatic initialisation of objects.

(2) Destructors The destructors are used to perform clean up after objects are explicitly or implicitly destroyed.

(3) Conversion Functions Objects of a given class type can be converted to objects of another type. This is done by constructing an object of the target class type from the source class type and copying the result to

the target object. This process is called conversion by constructor. Objects can also be converted by user-supplied conversion functions.

(4) Operator New Function C++ supports dynamic allocation and deallocation of objects using the new and delete operators. These operators allocate memory for objects from a pool called the free store. The new operator is the special member function which is used for dynamically allocates storage for class objects.

(5) Operator Delete Function The operator delete function is used to release storage allocated using the new operator. The delete operator calls the operator delete function, which frees memory back to the available pool.

(6) Assignment Operator (Operator=) Function The assignment operator (operator=) function is used when an assignment takes place between class objects.

11.2 CONSTRUCTORS

A constructor is a special member function for automatic initialisation of an object. Whenever an object is created, the special member function, i.e., is the constructor will be executed automatically. A constructor function is different from all other nonstatic member functions in a class because it is used to initialise the variables of whatever instance being created. Note that a constructor function can be overloaded to accommodate many different forms of initialisation.

Syntax rules for writing constructor functions The following rules are used for writing a constructor function:

- A constructor name must be the same as that of its class name.
- It is declared with no return type (not even void).
- It cannot be declared `const` or `volatile` but a constructor can be invoked for a `const` and `volatile` objects.
- It may not be `static`.
- It may not be `virtual`.
- It should have public or protected access within the class and only in rare circumstances it should be declared private.

The general syntax of the constructor function in C++ is,

```
class class_name {
    private :
       -------
       -------
    protected :
       -------
       -------
    public :
       class_name();  // constructor
       -------
       -------
};
class_name :: class_name()
{
    -------
    -------
}
```

The following examples illustrate the constructor declaration in a class definition.

(1)

```
class employee {
    private :
        char name[20];
        int ecode;
        char address[20];
    public :
        employee();   // constructor
        void getdata();
        void display ();
};
employee() :: employee();   // constructor
{
    -------
    -------
}
```

(2)

```
class account {
   private :
         oat balance;
         oat rate;
   public :
        account() //constructor
        {
           -------
           -------
        }
        void create_acct();
};
```

When the constructor function is invoked Constructors and destructors can be explicitly called. A constructor is automatically invoked when an object begins to live. Under the following circumstances, a constructor function is invoked automatically by the C++ compiler.

- The constructor is called before main () starts for execution
- Whenever an object is created in any of the following ways
  - a global variable
  - a local variable
  - or as a static variable.
- An auto variable of class is defined within a block and the location of its definition is reached.
- A temporary instance of class needs to be created.
- During use of the dynamic memory allocation operator new.

The following is an invalid declaration of a constructor function:

(1)

```
class account {
     private :
         oat balance;
         oat rate;
     public :
        account()    //constructor
        {
          -------
          -------
```

```
            }
            void create_acct();
    };
    void account :: account() {  // error
       -------
       -------
    }
```

Note that a constructor member function should not be defined with return type or a void type.

PROGRAM 11.1

A program to demonstrate how to use a special member function, namely, constructor in C++.

```
#include <iostream>
using namespace std;
class abc
{
   public:
      abc() {
         cout << "for class constructor\n";
      }
};
int main()
{
    abc obj;
    return 0;
}
```

Output of the above program

```
for class constructor
```

PROGRAM 11.2

A program to generate a series of Fibonacci numbers using the constructor where the constructor member function has been defined in the scope of class definition itself.

```
//generation of the  bonacci series using
// constructor
#include <iostream>
using namespace std;
class  bonacci {
    private :
       unsigned long int f0,f1, b;
    public :
        bonacci ()   // constructor
       {
           f0 = 0;
           f1 = 1;
           cout << "Fibonacci series of  rst 10 numbers\n";
           cout << f0 << '\t' << f1 <<'\t';
            b = f0+f1;
       }
       void increment ()
       {
           f0 = f1;
           f1 =  b;
            b = f0+f1;
       }
```

```
        void display()
        {
           cout <<  b << '\t';
        }
}; // end of class construction
int main()
{
     bonacci  number;
    for (int i = 3; i <= 10; ++i) {
       number.display();
       number.increment();
    }
    return 0;
}
```

Output of the above program

```
Fibonacci series of  rst 10 numbers
0    1    1    2    3    5    8    13   21   34
```

PROGRAM 11.3

A program to display a series of Fibonacci numbers using the constructor where the constructor member function has been defined out of the class definition using the scope resolution operator.

```
//generation of the  bonacci series using
// constructor using scope resolution operator
#include <iostream>
using namespace std;
class  bonacci {
    private :
       unsigned long int f0,f1, b;
    public :
        bonacci ();   // constructor
       void increment ();
       void display() ;
}; // end of class construction
 bonacci ::  bonacci()  // constructor
{
    cout << "Fibonacci series of  rst 10 terms\n";
    f0 = 0;
    f1 = 1;
     b = f0+f1;
    cout << f0 << '\t' << f1 << '\t';
}

void  bonacci :: increment ()
{
    f0 = f1;
    f1 =  b;
     b = f0+f1;
}
void  bonacci ::display()
{
    cout <<  b << '\t';
}
int main()
{
     bonacci  number;
    for (int i = 3; i <= 10; ++i) {
       number.display();
       number.increment();
    }
    return 0;
}
```

Output of the above program

```
Fibonacci series of  rst 10 terms
0   1   1   2   3   5   8   13  21  34
```

PROGRAM 11.4

A program to simulate a simple banking system in which the initial balance and the rate of interest are read from the keyboard and these values are initialised using the constructor member function. The program consists of the following methods:

- *To initialise the balance amount and the rate of interest using the constructor member function*
- *To make a deposit*
- *To withdraw an amount from the balance*
- *To find the compound interest based on the rate of interest*
- *To know the balance amount*
- *To display the menu options*

```
//demonstration of constructor
//simulation of simple banking system
#include <iostream>
using namespace std;
class account {
    private :
         oat balance ;
         oat rate;
    public :
        account();  // constructor
        void deposit ();
        void withdraw ();
        void compound();
        void getbalance();
        void menu();
}; //end of class de nition

account :: account() // constructor
{
    cout << " enter the initial balance \n";
    cin >> balance;
    cout << " interest rate in decimal\n";
    cin >> rate;
}

//deposit
void account :: deposit ()
{
     oat amount;
    cout << " enter the amount : ";
    cin >> amount;
    balance = balance+amount;
}

//attempt to withdraw
void account :: withdraw ()
{
     oat amount;
    cout << " how much to withdraw ? \n";
    cin >> amount;
    if (amount <= balance) {
       balance = balance-amount;
       cout << " amount drawn = " << amount << endl;
```

```
        cout << " current balance = " << balance << endl;

     }
    else
        cout << 0;
}

void account :: compound ()
{
      oat interest;
     interest = balance*rate;
     balance = balance+interest;
     cout << "interest amount = " << interest <<endl;
     cout << " toal amount = " << balance << endl;
}

void account :: getbalance()
{
     cout << " Current balance = " ;
     cout << balance << endl;
}

void account :: menu()
{

     cout << " d -> deposit\n";
     cout << " w -> withdraw \n";
     cout << " c -> compound interest\n";
     cout << " g -> get balance \n";
     cout << " q -> quit\n";
     cout << " m -> menu\n";
     cout << " option, please ? \n";
}

int main()
{
    class account acct;
    char ch;
    acct.menu();
    while ( (ch = cin.get()) != 'q')  {
       switch (ch) {
         case 'd' :
               acct.deposit();
               break;
         case 'w' :
               acct.withdraw();
               break;
         case 'c' :
               acct.compound();
               break;
         case 'g' :
               acct.getbalance();
               break;
         case 'm':
               acct.menu();
               break;
       } // end of switch statement
    }
    return 0;
}
```

Output of the above program

```
enter the initial balance
1000
interest rate in decimal
0.2
```

```
d -> deposit
w -> withdraw
c -> compound interest
g -> get balance
q -> quit
m -> menu
option, please?

g
Current balance = 1000
d
enter the amount : 1000
g
Current balance = 2000
w
how much to withdraw?
500
amount drawn = 500
current balance = 1500
c
interest amount = 300
toal amount = 1800
g
Current balance = 1800
q
```

11.2.1 Copy Constructors

Copy constructors are always used when the compiler has to create a temporary object of a class object. The copy constructors are used in the following situations:

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

The copy constructor can accept a single argument of reference to same class type. The purpose of the copy constructor is copy objects of the class type. The general format of the copy constructor is,

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

The symbolic representation of the above format is;

```
X :: X ( X &ptr)
```

where `X` is user-defined class name and `ptr` is pointer to a class object `X`.

The copy constructor may be used in the following format also using a const keyword.

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

The symbolic representation of the above format is,

```
X :: X ( const X &ptr)
```

where `X` is user-defined class name and `ptr` is pointer to a class object `X`.

Normally, the copy constructors take an object of their own class as arguments and produce such an object. The copy constructor usually do not return a function value as constructors cannot return any function values.

The following program segment demonstrates how to define a copy constructor for finding a series of Fibonacci numbers.

```
 bonacci ::  bonacci()  // constructor
{
   f0 = 0;
   f1 = 1;
    b = f0+f1;
}

 bonacci ::  bonacci(  bonacci &ptr)  //copy constructor
{
   f0 = ptr.f0;
   f1 = ptr.f1;
    b = ptr. b;
}
```

PROGRAM 11.5

A program to generate a series of Fibonacci numbers using a copy constructor where the copy constructor is defined within the class declaration itself.

```
//generation of the  bonacci series using
// copy constructor
#include <iostream>
#include <iomanip>
using namespace std;
class  bonacci {
    public :
       unsigned long int f0,f1, b;
        bonacci ()
       {
         f0 = 0;
         f1 = 1;
          b = f0+f1;
    }
     bonacci ( bonacci &ptr) {
         f0 = ptr.f0;
         f1 = ptr.f1;
          b = ptr. b;
    }
    void increment ()
    {
         f0 = f1;
         f1 =  b;
          b = f0+f1;
    }
    void display()
    {
        cout << setw(4) <<  b;
    }
}; // end of class construction
int main()
{
    int n;
     bonacci obj;
    cout << " How many numbers are to be displayed \n";
    cin >> n;
    cout << obj.f0 << setw(4) << obj.f1;
    for (int i = 2; i <= n-1; ++i) {
         obj.display();
         obj.increment();
    }
    cout << endl;
    return 0;
}
```

Output of the above program

```
How many numbers are to be displayed
8
0    1    1    2    3    5    8    13
```

PROGRAM 11.6

A program to generate a series of Fibonacci numbers using a copy constructor where the copy constructor is defined out of the class declaration using scope resolution operator.

```
//generation of the  bonacci series using
// copy constructor using scope resolution operator
#include <iostream>
#include <iomanip>
using namespace std;
struct  bonacci {
   public :
      unsigned long int f0,f1, b;
       bonacci ();   // constructor
       bonacci( bonacci &ptr);// copy constructor
      void increment ();
      void display() ;
}; // end of class construction

 bonacci ::  bonacci()  // constructor
{
   f0 = 0;
   f1 = 1;
    b = f0+f1;
}

 bonacci ::  bonacci(  bonacci &ptr)  //copy constructor
{
   f0 = ptr.f0;
   f1 = ptr.f1;
    b = ptr. b;
}

void  bonacci :: increment ()
{
   f0 = f1;
   f1 =  b;
    b = f0+f1;
}
void  bonacci :: display()
{
   cout << setw(4) <<  b;
}

int main()
{
   int n;
    bonacci obj;
   cout << "How many numbers are to be displayed ?\n";
   cin >> n;
   cout << obj.f0 << setw(4) << obj.f1;
   for (int i = 2; i <= n-1; ++i) {
      obj.display();
      obj.increment();
   }
   cout << endl;
   return 0;
}
```

Output of the above program

```
How many numbers are to be displayed?
6
0   1   1   2   3   5
```

11.2.2 Default Constructors

The default constructor is a special member function which is invoked by the C++ compiler without any argument for initialising the objects of a class. In other words, a default constructor function initialises the data members with no arguments. It can be explicitly written in a program. In case, a default destructor is not defined in a program, the C++ compiler automatically generates it in a program. The purpose of the default constructor is to construct a default object of the class type.

The general syntax of the default constructor function is,

```
class class_name {
    private :
       -------
       -------
    protected :
       -------
       -------
    public :
       class_name();  // default constructor
       -------
       -------
};
class_name :: class_name () {}  // without any arguments
```

The typical form of the default constructor is:

```
class_name :: class_name (int = 0) {} // without any arguments
```

Case 1 The default constructor can be declared in the following form:

student() {} // default constructor

```
// demonstration of default constructor
#include <iostream>
class student {
   private :
       char name[20];
       long int rollno;
       char sex;
        oat height;
        oat weight;
   public:
       student() {}  // default constructor
       void display();
 }; // end of class declaration
```

Case 2 The default constructor can be declared in the following form also:

student(int = 0) {} // constructor

```
// demonstration of default constructor
#include <iostream>
class student {
   private :
       char name[20];
```

```
        long int rollno;
        char sex;
         oat height;
         oat weight;
    public:
        student(int = 0) {} // constructor
        void display();
};  // end of class declaration
```

Case 3 If the default constructor is not declared explicitly, then it will be created automatically by the compiler.

```
#include <iostream>
class student {
   private :
        char name[20];
        long int rollno;
        char sex;
         oat height;
         oat weight;
   public:
        void display();
};  // end of class declaration
```

PROGRAM 11.7

A program to demonstrate the default initialisation of the constructor member function of a class object of the students' information system such as name, roll number, sex, height and weight.

```
// demonstration of default constructor
#include <iostream>
using namespace std;
class student {
   private :
       char name[20];
       long int rollno;
       char sex;
        oat height;
        oat weight;
   public :
       student();  // constructor
       void display();
};

student :: student()
{
   name[0] = '\0';
   rollno = 0;
   sex = '\0';
   height = 0;
   weight = 0;
}

void student :: display()
{
   cout << " name   = " << name <<endl ;
   cout << " rollno = " << rollno <<endl;
   cout << " sex    = " << sex << endl;
   cout << " height = " << height << endl;
```

```
    cout << " weight = " << weight << endl;
}

int main()
{
    student a;
    cout << " demonstration of default constructor \n";
    a.display();
    return 0;
}
```

Output of the above program

```
demonstration of default constructor
name     =
roll no  =  0
sex      =
height   =  0
weight   =  0
```

PROGRAM 11.8

A program to demonstrate the default initialisation of the constructor member function of a class object where the default constructor is defined within the scope of the class definition itself.

```
// demonstration of default constructor
#include <iostream>
#include <string>
using namespace std;
class student {
   private :
        char name[20];
        long int rollno;
        char sex;
         oat height;
         oat weight;
   public :
        student(char na[] ="\0",long int rn = 0,char sx = '\0',
                    oat ht = 0, oat wt=0);// constructor
        void display();
};

student:: student(char na[],long int rn,char sx,  oat ht,  oat wt)
{
    strcpy(name, na);
    rollno = rn;
    sex = sx;
    height = ht;
    weight = wt;
}

void student :: display()
{
    cout << " name   = " << name <<endl ;
    cout << " rollno = " << rollno <<endl;
    cout << " sex    = " << sex << endl;
    cout << " height = " << height << endl;
    cout << " weight = " << weight << endl;
}

int main()
{
```

```
    student a;
    cout << " demonstration of default constructor \n";
    a.display();
    return 0;
}
```

Output of the above program

```
demonstration of default constructor
name   =
rollno = 0
sex    =
height = 0
weight = 0
```

PROGRAM 11.9

A program to demonstrate the default initialisation of the constructor member function of a class object where the default constructor is created by the compiler automatically when the default constructor is not defined by the user.

```
#include <iostream>
using namespace std;
class student {
   private :
       char name[20];
       long int rollno;
       char sex;
        oat height;
        oat weight;
   public :
       void display();
};

void student :: display()
{
   cout << " name   = " << name <<endl ;
   cout << " rollno = " << rollno <<endl;
   cout << " sex    = " << sex << endl;
   cout << " height = " << height << endl;
   cout << " weight = " << weight << endl;
}

int main()
{
   student a;
   a.display();
   return 0;
}
```

Output of the above program

```
name   = □□□□□□□
rollno = 1108544020
sex    = X
height = 3.98791e-34
weight = 36.7598
```

Automatic variables are initialised with a garbage value if it is not initialised by the user explicitly.

11.2.3 Overloading Constructors

The overloading constructor is a concept in OOPs in which the same constructor name is called with different arguments. Depending upon the type of argument, the constructor will be invoked automatically by the compiler to intialise the objects.

PROGRAM 11.10

A program to demonstrate how to define, declare and use the overloading of constructors to initialise the objects for different data types.

```
//overloading of constructor
#include <iostream>
using namespace std;
class abc {
   public:
      abc();
      abc(int);
      abc( oat);
      abc(int,  oat);
};
abc :: abc()
{
   cout <<"calling default constructor \n";
}

abc :: abc (int a)
{
   cout << "\n calling constructor with int \n";
   cout << " a = " << a << endl;
}

abc :: abc ( oat fa)
{
    cout <<"\n calling constructor with  oating point number \n";
    cout <<" fa = " << fa << endl;
}

abc :: abc(int a,  oat fa)
{
   cout << " \n calling constructor with int and  oat \n";
   cout << " a = " << a << endl;
   cout << " fa = " << fa << endl;
}

int main()
{
   abc obj;
   abc(10);
   abc(1.1f);
   abc(20,-2.2);
   return 0;
}
```

Output of the above program

```
calling default constructor

calling constructor with int
a = 10

calling constructor with  oating point number
```

```
fa = 1.1

calling constructor with int and  oat
a = 20
fa = -2.2
```

11.2.4 Constructors in Nested Classes

It is well known that a constructor is a special member function which is used to initialise the class objects whenever an object is created. The constructor member function can be used to initialise the class objects of nested classes. The nested class is a technique in which a class is declared as a member of another class. In other words, a class within a class is called as nested class. The scope rules to access the nested class constructor is the same as that of the member functions of the nested classes. The scope resolution operator (::) is used to identify the outer and inner class objects and the constructor member functions.

PROGRAM 11.11

A program to demonstrate how to declare, define and call a constructor member function in a nested class.

```
#include <iostream>
using namespace std;
class abc {
   public:
      abc();
      class x {
        public:
           x();
      };
};
abc:: abc()
{
   cout << "abc - class constructor \n";
}
abc::x :: x()
{
   cout << "x - class constructor \n";
}

int main()
{
    abc obj;
    abc::x obj2;
    return 0;
}
```

Output of the above program

```
abc - class constructor
x - class constructor
```

PROGRAM 11.12

A program to demonstrate how to declare, define and call a constructor member function in a nested class. The firing order of the nested class constructor is illustrated in the following program.

```
#include <iostream>
using namespace std;
class abc {
```

```
    public:
       abc();
       class x {
           public:
             x();
             class y {
                public:
                   y();
                   class z {
                      public:
                         z();
                   };
             };
       };
};
abc:: abc()
{
    cout << "abc - class constructor \n";
}
abc::x :: x()
{
    cout << "x - class constructor \n";
}

abc::x :: y :: y()
{
    cout << "y - class constructor \n";
}

abc::x ::y :: z :: z()
{
    cout << "z - class constructor \n";
}

int main()
{
    abc obj1;
    abc::x  obj2;
    abc::x::y obj3;
    abc::x::y::z  obj4;
    return 0;
}
```

Output of the above program

```
abc - class constructor
x - class constructor
y - class constructor
z - class constructor
```

11.3 DESTRUCTORS

A destructor is a function that automatically executes when an object is destroyed. A destructor function gets executed whenever an instance of the class to which it belongs goes out of existence. The primary usage of the destructor function is to release space on the heap. A destructor function may be invoked explicitly.

Syntax Rules for Writing a Destructor Function The rules for writing a destructor function are:

- A destructor function name is the same as that of the class it belongs except that the first character of the name must be a tilde (~).
- It is declared with no return types (not even void) since it cannot ever return a value.
- It cannot be declared static, const or volatile.

- It takes no arguments and therefore cannot be overloaded.
- It should have public access in the class declaration.

The general syntax of the destructor function in C++ is,

```
class class_name {
    private :
       // data variables
       // methods
    protected :
       // data
    public :
       class_name();  // constructor
       ~class_name(); // destructor
       // other methods
};
```

The following examples illustrate the destructor function declaration in a class definition:

(1)

```
class employee {
    private :
        char name[20];
        int ecode;
        char address[20];
    public :
        employee();   // constructor
        ~employee();  // destructor
        void getdata();
        void display ();
};
```

(2)

```
class account {
    private :
         oat balance;
         oat rate;
    public :
        account();      //constructor
        ~account();     // destructor
        void create_acct();
};
```

When the Destructor Function is Invoked Under the following circumstances, a destructor function is invoked:

- After the end of main () for all static, local to main () and global instances of class.
- At the end of each block containing the auto variable of class.
- At end of each function containing an argument of class.
- To destroy any unnamed temporaries of class after their use.
- When an instance of class allocated on the heap is destroyed via delete.

PROGRAM 11.13

A program to simulate a simple banking system in which the initial balance and the rate of interest are read from the keyboard and these values are initialised using the constructor member function. The destructor member function is defined in this program to destroy the class objects created using the constructor member function. The program consists of the following methods:

- *To initialise the balance and the rate of interest using the constructor member function.*
- *To make a deposit.*
- *To withdraw an amount from the balance.*
- *To find the compound interest based on the rate of interest.*
- *To know the balance amount.*
- *To display the menu options.*
- *To destroy the object of class, the destructor member function is defined.*

```
//program for class demonstration
// constructor and destructor
#include <iostream>
using namespace std;
class account {
    private :
         oat balance;
         oat rate;
    public :
        account();  // constructor
        ~account();  // destructor
        void deposit ();
        void withdraw ();
        void compound();
        void getbalance();
        void menu();
}; //end of class de nition

account :: account() // constructor
{
    cout << " enter the initial balance \n";
    cin >> balance;
    cout << " interest rate \n";
    cin >> rate;
}
account :: ~account() // constructor
{
    cout << " data base has been deleted \n";
}

//deposit
void account :: deposit ()
{
     oat amount;
    cout << " enter the amount  ";
    cin >> amount;
    balance = balance+amount;
}

//attempt to withdraw
void account :: withdraw ()
{
     oat amount;
    cout << " how much to withdraw ? \n";
    cin >> amount;
    if (amount <= balance) {
       balance = balance-amount;
       cout << " amount drawn = " <<amount << endl;
       cout << " current balance = " << balance << endl;
    }
    else
        cout << 0;
}

void account :: compound ()
{
```

```
     oat interest;
    interest = balance*rate;
    balance = balance+interest;
    cout << "interest amount = " << interest <<endl;
    cout << " toal amount = " << balance << endl;
}

void account :: getbalance()
{
    cout << " Current balance = " ;
    cout << balance << endl;
}

void account :: menu()
{
    cout << " d -> deposit\n";
    cout << " w -> withdraw\n";
    cout << " c -> compound interest\n";
    cout << " g -> get balance\n";
    cout << " m -> menu \n";
    cout << " q -> quit \n";
    cout << " option, please ? \n";
}

int main()
{
    account acct;
    char ch;
    acct.menu();
    while ( (ch = cin.get()) != 'q')  {
        switch (ch) {
           case 'd' :
                 acct.deposit();
                 break;
           case 'w' :
                 acct.withdraw();
                 break;
           case 'c' :
                 acct.compound();
                 break;
           case 'g' :
                 acct.getbalance();
                 break;
           case 'm':
                 acct.menu();
                 break;
        } // end of switch statement
    }
    cout << endl;
    return 0;
}
```

Output of the above program

```
enter the initial balance
1000
interest rate
0.2
d -> deposit
w -> withdraw
c -> compound interest
g -> get balance
m -> menu
q -> quit
option, please?
```

```
d
enter the amount
500
g
Current balance = 1500

w
how much to withdraw?
1000
amount drawn = 1000
Current balance = 500
c
interest amount = 100
toal amount = 600
q
data base has been deleted
```

11.3.1 Destructors in Nested Classes

It is well known that a destructor is a special member function which is used to destroy the class objects automatically whenever an object is released. The destructor member function can also be used to implement the nested class objects. The firing order of destructor under nested class is that the innermost class object will be released first and so on. The scope rules to access the nested class destructor is the same as that of the member functions of the nested classes. The scope resolution operator (::) is used to identify the outer and inner class objects and the constructor/destructor member functions.

PROGRAM 11.14

A program to demonstrate how to declare, define and call a destructor member function in a nested class.

```
#include <iostream>
using namespace std;
class abc {
  public:
    ~abc();
    class x {
      public:
         ~x();
         class y{
             public:
                ~y();
                class z {
                    public:
                       ~z();
                };
         };
    };
};

abc:: ~abc()
{
   cout << "abc - class destructor \n";
}

abc::x :: ~x()
{
```

```
    cout << "x - class destructor \n";
}

abc::x ::y :: ~y()
{
    cout << "y - class destructor \n";
}

abc::x ::y :: z :: ~z()
{
    cout << "z - class destructor \n";
}

int main()
{
    abc obj1;
    abc::x  obj2;
    abc::x::y obj3;
    abc::x::y::z  obj4;
    return 0;
}
```

Output of the above program

```
z - class destructor
y - class destructor
x - class destructor
abc - class destructor
```

PROGRAM 11.15

A program to demonstrate how to declare, define and call a constructor and a destructor member function in a nested class. This program illustrates the firing order of constructor and destructor under the nested class.

```
#include <iostream>
using namespace std;
class abc {
  public:
     abc();
    ~abc();
     class x {
        public:
           x();
          ~x();
           class y{
              public:
                 y();
                ~y();
                 class z {
                    public:
                       z();
                      ~z();
                 };
           };
     };
};

abc:: abc()
{
   cout << "abc - class constructor \n";
}
```

```
abc:: ~abc()
{
    cout << "abc - class destructor \n";
}

abc::x :: x()
{
    cout << "x - class constructor \n";
}

abc::x :: ~x()
{
    cout << "x - class destructor \n";
}
abc::x ::y :: y()
{
    cout << "y - class constructor \n";
}

abc::x ::y :: ~y()
{
    cout << "y - class destructor \n";
}

abc::x ::y :: z :: z()
{
    cout << "z - class constructor \n";
}

abc::x ::y :: z :: ~z()
{
    cout << "z - class destructor \n";
}

int main()
{
    abc obj1;
    abc::x  obj2;
    abc::x::y obj3;
    abc::x::y::z  obj4;
    return 0;
}
```

Output of the above program

```
abc - class constructor
x - class constructor
y - class constructor
z - class constructor
z - class destructor
y - class destructor
x - class destructor
abc - class destructor
```

11.4 INLINE MEMBER FUNCTIONS

The keyword inline is used as a function specifier only in function declarations. The inline specifier is a hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation.

The advantages of using inline member functions are:

- the size of the object code is considerably reduced,
- it increases the execution speed, and
- the inline member functions are compact function calls.

The general format of the inline member function declaration is:

```
class user_de ned_name {
    private :
        -------
        -------
    public :
        inline return_type function_name(parameters);
        inline return_type function_name(parameters);
        -------
        -------
};
```

The syntax diagram of inline method declaration is given in Fig. 11.1.

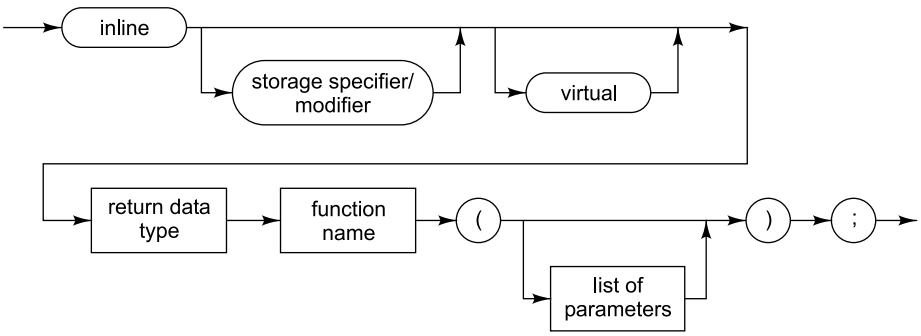


Fig. 11.1 *Syntax Diagram of Inline Method Declaration*

Whenever a function is declared with the inline specifier, the C++ compiler merely replaces it with the code itself so the overhead of the transfer of control between the calling portion of a program and a function is reduced. The inline specifier can be used either as a member of a class or a global function.

To define inline member specifier is well suited whenever a function is small, straight forward and are not called from too many different places.

PROGRAM 11.16

A program to illustrate an inline member function to read a data member of a class from the keyboard and to display it on the screen.

```
//inline demonstration
#include <iostream>
using namespace std;
class sample {
    private :
       int x;
    public :
       void getdata();
       void display();
};
```

```
inline void sample:: getdata()
{
    cout << " enter a number ? \n";
    cin >> x;
}

inline void sample :: display()
{
    cout << " entered number is = " << x ;
    cout << endl;
}

int main()
{
    sample obj;
    obj.getdata();
    obj.display();
    return 0;
}
```

Output of the above program

```
enter a number?
10
entered number is = 10
```

PROGRAM 11.17

A program to perform the simple arithmetic operations such as addition, subtraction, multiplication and division using a member function and these functions are defined as an inline substitution.

```
//demonstration of inline member function
#include <iostream>
using namespace std;
class sample {
    private :
        int x;
        int y;
    public :
        inline void getinfo();
        inline void display();
        inline int sum();
        inline int diff();
        inline int mult();
        inline  oat div();
}; // end of class declaration

inline void sample :: getinfo()
{
    cout << " enter any two numbers ? " << endl;
    cin >> x >> y ;
}

inline void sample :: display()
{
    cout << " x = " << x << endl;
    cout << " y = " << y << endl;
    cout << " sum = " << sum() << endl;
    cout << " dif = " << diff() << endl;
    cout << " mul = " << mult() << endl;
    cout << " div = " << div() << endl;
}
```

```
inline int sample :: sum()
{
    return(x+y);
}

inline int sample :: diff()
{
    return(x-y);
}

inline int sample :: mult()
{
    return(x*y);
}

inline  oat sample :: div()
{
    return(( oat)x / ( oat)y);
}

int main()
{
    sample obj1;
    obj1.getinfo();
    obj1.display();
    obj1.sum();
    obj1.diff();
    obj1.mult();
    obj1.div();
    return 0;
}
```

Output of the above program

```
enter any two numbers?
3   4
x = 3
y = 4
sum = 7
dif = -1
mul = 12
div = 0.75
```

PROGRAM 11.18

A program to solve a quadratic equation using an object-oriented programming technique in which the member functions are defined as an inline substitution.

```
//solution of quadratic equation using object oriented programming
// methods are de ned as inline substitution
#include <iostream>
#include <cmath>
using namespace std;
class equation {
    private :
         oat a;
         oat b;
         oat c;
    public :
        inline void getinfo( oat a, oat b, oat c);
        inline void display();
```

```
        inline void equal( oat a,  oat b);
        inline void imag();
        inline void real( oat a, oat b, oat det);
}; // end of class declaration

inline void equation ::getinfo( oat aa, oat bb, oat cc)
{
    a = aa;
    b = bb;
    c = cc;

}

inline void equation :: display()
{
    cout << endl;
    cout << " a = " << a << '\t';
    cout << " b = " << b << '\t';
    cout << " c = " << c << endl;
}

inline void equation :: equal( oat a , oat b)
{
     oat x;
    x = -b/(2*a);
    cout << " roots are equal = " << x << endl;
}

inline void equation :: imag()
{
    cout << " roots are imaginary \n";
}

inline void equation :: real( oat a , oat b, oat det)
{
     oat x1,x2,temp;
    temp = sqrt(det);
    x1 = (-b+temp)/(2*a);
    x2 = (-b-temp)/(2*a);
    cout << " roots are real \n";
    cout << " x1 = " << x1 << endl;
    cout << " x2 = " << x2 << endl;
}

int main()
{
    class equation equ;
     oat aa,bb,cc;
    cout << " enter three numbers \n";
    cin >> aa >> bb >> cc;
    equ.getinfo(aa,bb,cc);
    equ.display();
    if ( aa == 0) {
         oat temp;
        temp = -cc/bb;
        cout << " linear roots = " << temp << endl;
    }
    else {
         oat det;
        det = bb*bb-4*aa*cc;
        if (det == 0)
            equ.equal(aa,bb);
            else if (det < 0)
            equ.imag();
        else
            equ.real(aa,bb,det);
    }
```

```
    return 0;
} // end of main program
```

Output of the above program

```
enter three numbers
2   5   2
a = 2  b = 5  c = 2
roots are real
x1 = -0.5
x2 = -2

enter three numbers
0   2   1
a = 0  b = 2  c = 1
linear roots = -0.5
```

11.5 STATIC CLASS MEMBERS

The static data type is one of variables that has been discussed in Chapter 6 on "Functions and Program Structures." The main characteristic of the static variables is that the static variables are automatically initialised to zero unless it has been initialised by some other value explicitly.

In C++, static member of a class can be categorised into two types, static data member and static member function. Whenever a data or function member is declared as a static type, it belongs to a class, not to the instances or objects of the class. Both the data member and member function can have the keyword `static`.

11.5.1 Static Data Member

Static data members are data objects that are common to all the objects of a class. They exist only once in all objects of this class. They are already created before the finite object of the respective class. The static members are used in information that is commonly accessible.

This property of static data members, may lead a person to think that static members are basically global variables. This is not true. Static members can be any one of the groups: public, private and protected, but not global data. The normal access rules for class members are also valid for static member data. If static member of a class is public, it can used as a normal variable.

The main advantage of using a static member is to declare the global data which should be updated while the program lives in the memory.

A static data member of a class has the following properties;

(1) The access rule of the data member of a class is same for the static data member also. For example, if a static member is declared as a private category of a class, then non-member functions cannot access these members. If a static member is declared as public, then any member of the class can access.

(2) Whenever a static data member is declared and it has only a single copy, it will be shared by all the instance of class. That is, the static member becomes global instances of the class.

(3) The static data member should be created and initialised before the main () function control block begins.

The general syntax of the static data member declaration is:

```
class user_de ned_name {
    private :
       static data_type variables;
       static data_type variables;
```

```
        -------
        -------
    public :
        -------
        -------
};
```

For example, the following program segment shows how to declare the static data type.

```
class sample {
   private :
       static int sum; // static data declaration
   public :
       void display();
};

int sample:: sum = 0; // static data de nition
int main()
{
   -------
   -------
}
```

The keyword static should not be repeated again in the static data member definition part.

PROGRAM 11.19

A program to demonstrate how an automatic initialisation of a static member is carried out and the contents of the variable displayed.

```
//automatic initialization of a static member
#include <iostream>
using namespace std;
class sample {
   private:
       static int counter;
   public:
       inline void display();
};
int sample :: counter;
inline void sample:: display()
{
   cout << " content of the static data member = " << counter;
   cout << endl;
}

int main()
{
    sample obj;
    obj.display();
    return 0;
}
```

Output of the above program

```
content of the static data member = 0
```

PROGRAM 11.20

A program to define a static data member which has the initial value of 100 and to find out the sum of the following series:

```
sum = 1+2+3...10
```

The summing of series is to be repeated five times.

This program is to demonstrate how the compiler reserves the special storage allocation as a global space so that the content of a variable lives from the start of the program to the end.

```
//static data member
#include <iostream>

using namespace std;
class sample {
    private :
        static int counter;
    public :
        void display();
};

int sample::counter = 100;

void sample:: display()
{
    int i;
    for (i = 0; i <= 10; ++i) {
        counter = counter+i;
    }
    cout << " sum of the counter value = " << counter;
    cout << endl;
}

int main()
{
    sample obj;
    int i;
    for ( i = 0; i < 5; ++i) {
        cout << " count = " << i+1 << endl;
        obj.display();
        cout << endl;
    }
    return 0;
}
```

Output of the above program

```
count = 1
sum of the counter value = 155

count = 2
sum of the counter value = 210

count = 3
sum of the counter value = 265

count = 4
sum of the counter value = 320

count = 5
sum of the counter value = 375
```

PROGRAM 11.21

A program to display how many instantiation of a class object has been created using static data member declaration. This program is to demonstrate how the static data member is used to keep track of the number of instantiation of a class created. Whenever a new object of the class is made, the static counter updates automatically.

```
//static data member
#include <iostream>
using namespace std;

class sample {
     private :
          static int count;   // static data member declaration
     public :
          sample();
          void display();
};
 //static data de nition
int sample :: count = 0;
sample :: sample ()
{
     ++count;
}

void sample :: display()
{
     cout << " counter value = " << count << endl;
}
int main()
{
     sample obj1,obj2,obj3,obj4;
     obj4.display();
     return 0;
}
```

Output of the above program

```
counter value = 4
```

11.5.2 Static Member Functions

The keyword static is used to precede the member function to make a member function static. The static function is a member function of class and the static member function can manipulate only on static data member of the class. The static member function acts as global for members of its class without affecting the rest of the program.

The purpose of static member is to reduce the need for global variables by providing alternatives that are local to a class. A static member function is not part of objects of a class. Static members of a global class have external linkage. A static member function does not have a `this` pointer so it can access nonstatic members of its class only by using `.` or `->`.

The static member function cannot be a virtual function. A static or nonstatic member function cannot have the same name and the same arguments type. And further, it cannot be declared with the keyword `const`. The static member function is instance dependent, it can be called directly by using the class name and the scope resolution operator. If it is declared and defined in a class, the keyword static should be used only on the declaration part.

PROGRAM 11.22

A program to check how many instances of the class object are created using the static member function.

```
//both static data member and static function member
#include <iostream>
using namespace std;
class sample {
    private :
        static int count;   // static data member declaration
    public :
        sample();
        static void display(); // static member function
};
//static data de nition
int sample :: count = 0;
sample :: sample ()
{
    ++count;
}

void sample :: display()
{
    cout << " counter value = " << count << endl;
}
int main()
{
    cout << " before instantiation of the object " << endl;
    sample::display();
    sample obj1,obj2,obj3,obj4;
    cout << " after instantiation of the object " << endl;
    sample::display();
    return 0;
}
```

Output of the above program

```
before instantiation of the object
counter value = 0
after instantiation of the object
counter value = 4
```

PROGRAM 11.23

A program to check how many instances of the class object are created using the static member function, where static member function is defined with inline code substitution.

```
//using inline member function
#include <iostream>
using namespace std;
class sample {
    private :
        static int count;   // static data member declaration
    public :
        sample();
        static void display(); // static member function
};
//static data de nition
int sample :: count = 0;
inline sample :: sample ()
```

```
{
   ++count;
}

inline void sample :: display()
{
    cout << " counter value = " << count << endl;
}
int main()
{
   cout << " before instantiation of the object\n";
   sample::display();
   sample obj1,obj2,obj3,obj4;
   cout << " after instantiation of the object\n";
   sample::display();
   return 0;
}
```

Output of the above program

```
before instantiation of the object
counter value = 0
after instantiation of the object
counter value = 4
```

PROGRAM 11.24

A program to demonstrate how a static data is accessed by a static member function.

```
//accessing static member function
#include <iostream>
using namespace std;
class alpha {
    private:
        static int x;
    public :
        alpha();
        static void display() // static member function
        {
            cout << " content of x ";
            cout << " after incrementd by one = " << x << endl;
        }
};
class beta {
    private:
        int y;
    public :
        void getdata()
        {
            cout << " enter a value for y \n";
            cin >> y;
        }
        void display() //member function
        {
            cout << " content of y = " << this->y << endl;
        }
};

int alpha:: x = 10;
alpha :: alpha()
{
    ++x;
}
```

```
int main()
{
    alpha objx;
    beta objy;
    objy.getdata();
    alpha::display();
    objy.display();
    return 0;
}
```

Output of the above program

```
enter a value for y
10
content of x  after incremented by one = 11
content of y = 10
```

11.6 FRIEND FUNCTIONS

The main concepts of the object-oriented programming paradigm are data hiding and data encapsulation. Whenever data variables are declared in a private category of a class, these members are restricted from accessing by non-member functions. The private data values can be neither read nor written by non-member functions. If any attempt is made directly to access these members, the compiler will display an error message as "inaccessible data type". The best way to access a private data member by a non-member function is to change a private data member to a public group. When the private or protected data member is changed to a public category, it violates the whole concept of data hiding and data encapsulation.

To solve this problem, a friend function can be declared to have access to these data members. Friend is a special mechanism for letting non-member functions access private data. A friend function may be either declared or defined within the scope of a class definition. The keyword `friend` inform the compiler that it is not a member function of the class. If the friend function is declared within the class, it must be defined outside the class, but should not be repeated the keyword friend. The general syntax of the friend function is,

```
friend return_type function_name (parameters);
```

where `friend` is a keyword used as a function modifier.

A friend declaration is valid only within or outside the class definition.

(1) The following is a valid program segment shows how a friend function is defined within the scope of a class definition

```
class alpha {
    private:
       int x;
    public :
       void getdata();
       friend void display (alpha abc)
       {
           cout << " value of x = " << abc.x;
           cout << endl;
       }
}; // end of class de nition
```

(2) The following program segment shows how a friend function is defined out of the class definition

```
class alpha {
    private:
```

```
        int x;
    public :
        void getdata();
        friend void display (alpha abc);
}; // end of class de nition

void display(alpha abc)//non-member function without scope::operator
{
   cout << " value of x = " << abc.x;
   cout << endl;
}
```

The following is an invalid declaration of the friend function. The keyword friend should not be repeated in both the function declaration and definition.

```
class alpha {
      private:
         int x;
      public :
         void getdata();
         friend void display (alpha abc);
};
friend void display(alpha abc)//the keyword friend is repeated
{
    cout << " value of x = " << abc.x;
    cout << endl;
}
```

The friend declaration is unaffected by its location in the class. The C++ compiler permits the declaration of a friend function either in a public or a private section, which does not affect its access right. The following declarations of a friend function are valid:

(1) The friend function disp () is declared in the public group

```
class  rst {
    private :
       int x;
    public :
       void getdata();
       friend void disp();
};
```

(2) The friend function disp () is declared in the private group

```
class second {
    private :
       int x;
       friend void disp();
    public :
       void getdata();
};
```

(a) Accessing Private Data by Non-Member Function through Friend The private data members are available only to the particular class and not to any other part of the program. A non-member function cannot access these private data. It is now understood that C++ language is not just an enhanced version of C or the one which introduces only classes and objects.

In case, the keyword struct is used for declaring a class object, all its members are public by default. There is no data hiding between data members and the outside world. The friend function is a special type of function which is used to access the private data of any class. In other words, they are defined as non-member functions with the ability to manipulate data members directly or to call function members that are

not part of the public interface. The friend class has the right to access as many members of its class.

Each time a friend function accesses the private data, naturally the level of privacy of the data encapsulation gets reduced. Only if it is necessary to access the private data by non-member functions, then a class may have a friend function, otherwise it is not necessary.

PROGRAM 11.25

A program to access the private data of a class by non-member function through friend, where the friend function is declared in the location of public category.

```
//declaring friend function
#include <iostream>
using namespace std;
class sample {
    private :
        int x;
    public :
        void getdata();
        friend void display(class sample);
};

void sample :: getdata()
{
    cout << " enter a value for x \n";
    cin >> x;
}

void display(class sample abc)
{
    cout << " Entered number is :" << abc.x;
    cout << endl;
}

int main()
{
    sample obj;
    obj.getdata();
    cout <<" accessing the private data by non-member function \n";
    display(obj);
    return 0;
}
```

Output of the above program

```
enter a value for x
100
accessing the private data by non-member function
Entered number is: 100
```

PROGRAM 11.26

A program to access the private data of a class by non-member function through friend, where the friend function is declared in the location of private category.

```
//declaring friend function
#include <iostream>
using namespace std;
class sample {
    private :
```

```
        int x;
        friend void display(class sample);
    public :
        void getdata();

};

void sample :: getdata()
{
    cout << " enter a value for x \n";
    cin >> x;
}

void display(class sample abc)
{
    cout << " Entered number is :" << abc.x;
    cout << endl;
}

int main()
{
    sample obj;
    obj.getdata();
    cout <<" accessing the private data by non-member function \n";
    display(obj);
    return 0;
}
```

Output of the above program

```
enter a value for x
20
accessing the private data by non-member function
Entered number is: 20
```

PROGRAM 11.27

A program to access the private data of a class by non-member function through friend, where the friend function is defined within the scope of a class definition itself.

```
// friend function is de ned within the scope of a class de nition
#include <iostream>
using namespace std;
class sample {
    private :
        int x;
    public :
        inline void getdata();
        friend void display(sample abc)
        {
           cout << " Entered number is :" << abc.x;
           cout << endl;
        }
};

inline void sample :: getdata()
{
    cout << " enter a value for x \n";
    cin >> x;
}

int main()
{
```

```
    sample obj;
    obj.getdata();
    cout <<" accessing the private data by non-member function \n";
    display(obj);
    return 0;
}
```

Output of the above program

```
enter a value for x
300
accessing the private data by non-member function
Entered number is: 300
```

(b) Friend Function with Inline Substitution Note that friend function may also have inline member functions. If the friend function is defined within the scope of the class definition, then the inline code substitution is automatically made. If it is defined outside the class definition, then it is required to precede the return type with the keyword inline in order to make an inline code substitution.

PROGRAM 11.28

A program to access the private data of a class by non-member function through a friend specifier, where the friend function is defined with inline code substitution.

```
//declaring friend function with inline code
#include <iostream>
using namespace std;
class sample {
    private :
        int x;
    public :
        inline void getdata();
        friend void display(class sample);
};

inline void sample :: getdata()
{
    cout << " enter a value for x \n";
    cin >> x;
}

inline void display(class sample abc)
{
    cout << " Entered number is :" << abc.x;
    cout << endl;
}

int main()
{
    sample obj;
    obj.getdata();
    cout <<" accessing the private data by non-member function \n";
    display(obj);
    return 0;
}
```

Output of the above program

```
enter a value for x
10
accessing the private data by non-member function
```

```
Entered number is: 10
```

The storage class modifier inline is used in both places, friend function declaration and the function definition.

PROGRAM 11.29

A program to access the private data of a class by non-member function through a friend specifier, where the friend function is defined with inline code substitution. The keyword inline is used in both the function declaration and the function definition.

```
#include <iostream>
using namespace std;
class sample {
    private:
       int x;
    public :
       inline void getdata();
       inline friend void display(class sample);
};

inline void sample :: getdata()
{
    cout << " enter a value for x \n";
    cin >> x;
}

inline void display(class sample abc)
{
    cout << " Entered number is :" << abc.x;
    cout << endl;
}
int main()
{
    sample obj;
    obj.getdata();
    cout <<" accessing the private data by non-member function \n";
    display(obj);
    return 0;
}
```

Output of the above program

```
enter a value for x
100
accessing the private data by non-member function
Entered number is: 100
```

(c) Granting Friendship to Another Class A class can have friendship with another class. For example, let there be two classes, first and second. If the class first grants its friendship with the other class second, then the private data members of the class first are permitted to be accessed by the public members of the class second. But on the other hand, the public member functions of the class first cannot access the private members of the class second.

PROGRAM 11.30

A program to demonstrate how a class first has granted its friendship to the class second to access the private data of the class first through the public member function of the class second.

```
//granting friendship with another class
#include <iostream>
using namespace std;
class  rst {
    friend class second;
    private :
       int x;
    public :
       void getdata();
};
class second {
    public :
       void disp(class  rst temp);
};
inline void  rst:: getdata()
{
    cout << " enter a number ? \n";
    cin >> x;
}

inline void second ::disp(class  rst temp)
{
    cout << " entered number is = " << temp.x ;
    cout << endl;
}
int main()
{
     rst objx;
    second objy;
    objx.getdata();
    objy.disp(objx);
    return 0;
}
```

Output of the above program

```
enter a number?
10
entered number is = 10
```

A following program will not be compiled due to an error

```
//granting friendship with another class
#include <iostream>
using namespace std;
class  rst {
   friend class second;
   public :
      void display(second objb)
      {
         cout << " object of second = " << objb.y << endl;
      }
};
class second {
   private:
      int y;
   public :
      void getdata()
      {
         cout << " enter a value for y = \n";
         cin >> y;
      }
```

```
};
int main()
{
     rst objx;
    second objy;
    objy.getdata();
    objx.display(objy);
    return 0;
}
```

Compile time error

```
forward declaration of class second
```

Though the class first has granted its friendship to the class second, it cannot access the private data of the class second through its public member function display () of the class first.

(d) Two Classes Having the Same Friend A non-member function may have friendship with one or more classes. When a function has declared to have friendship with more than one class, the friend classes should have forward declaration. It implies that it needs to access the private members of both classes.

The general syntax of declaring the same friend function with more than one class is,

```
class second; //forward declaration
class  rst {
    private :
       -------
       -------
    public :
       friend return_type fname(class  rst,class second...);
};
class  rst {
    private :
       -------
       -------
    public :
       friend return_type fname(class  rst,class second...);
};
```

PROGRAM 11.31

A program to calculate the sum of private data of the class first with a private data of another class second through the common friend function.

```
//friend function is same for both classes
#include <iostream>
using namespace std;
class second; //forward declaration
class  rst {
    private :
       int x;
    public :
       inline void getdata();
       inline void display();
       friend int sum( rst,second);
};

class second {
    private :
```

```
        int y;
    public :
        inline void getdata();
        inline void display();
        friend int sum( rst,second);
};

inline void  rst :: getdata()
{
    cout << " enter a value for x \n";
    cin >> x;
}
inline void second :: getdata()
{
    cout << " enter a value for y \n";
    cin >> y;
}

inline void  rst :: display()
{
    cout << " entered number is (x) = ";
    cout << x << endl;
}

inline void second :: display()
{
    cout << " entered number is (y) = ";
    cout << y << endl;
}

int sum ( rst one, second two)
{
    int temp;
    temp = one.x+two.y;
    return(temp);
}

int main()
{
     rst a;
    second b;
    a.getdata();
    b.getdata();
    a.display();
    b.display();
    int te = sum(a,b);
    cout << " sum of the two private data variables (x+y)";
    cout << " = " << te << endl;
    return 0;
}
```

Output of the above program

```
enter a value for x
10
enter a value for y
20
entered number is (x) = 10
entered number is (y) = 20
sum of the two private data variables (x+y) = 30
```

In the above example, the function sum () needs to have the right to access for fetching the private data members of the both classes in order to add the contents together. Therefore, the function takes formal arguments of the both classes.

```
friend int sum( rst,second);
```

11.7 DYNAMIC MEMORY ALLOCATIONS

Two operators, namely, `new` and `delete` are used in dynamic memory allocations which are described in detail in this section.

(a) New The new operator is used to create a heap memory space for an object of a class. In C, there are special functions used to create a memory space dynamically, viz. `malloc()`, `calloc()` and `alloc()`. C++ provides a new way in which dynamic memory is allocated. In reality, the new keyword calls upon the function operator `new()` to obtain storage.

Basically, an allocation expression must carry out the following three things:

(i) Find storage for the object to be created,
(ii) Initialise that object, and
(iii) Return a suitable pointer type to the object.

The new operator returns a pointer to the object created. Functions cannot be allocated this way using new operator but pointers to functions can be used for allocating memory space.

The general syntax of the new operator is,

```
data_type pointer  = new data_type;
```

where `data_type` can be a short integer, float, char, array or even class objects.

For example,

```
new int;   // an expression to allocate a single integer
new  oat;  // an expression to allocate a  oating value
```

If the call to the new operator is successful, it returns a pointer to the space that is allocated. Otherwise it returns the address zero if the space could not be found or if some kind of error is detected.

(b) Delete The delete operator is used to destroy the variable space which has been created by using the new operator dynamically. It is analogous to the function `free()` in C. In reality, the delete keyword calls upon the function operator `delete()` to release storage which was created using the new operator.

The general syntax of the delete operator is:

```
delete  pointer ;
```

As the new operator returns a pointer to the object being created, the delete operator must define a only pointer name but not with data type. The delete operator takes no arguments for deleting a single instance of a memory variable created by a new operator.

For example,

```
char *ptr_ch = new char; // memory for a character is allocated
int *ptr_i   = new int; // memory for an integer is allocated
delete ptr_ch;    // delete memory space
delete ptr_i;    // delete
```

Note that the delete operator is used for only releasing the heap memory space which was allocated by the new operator. If attempts are made to release memory space using delete operator that was not allocated by the new operator, then it gives unpredictable results. The following usage of the delete operator is invalid, as the delete operator should not be used twice to destroy the same pointer.

```
char *ptr_ch = new char; // allocating space for a character
delete ptr_ch;
delete ptr_ch; // error
```

PROGRAM 11.32

A program to create a dynamic memory allocation for the standard data types: integer, floating point, character and double. The pointer variables are initialised with some data and the contents of the pointers are displayed on the screen.

```
//using new and delete operators
#include <iostream>
using namespace std;

int main()
{
    int *ptr_i = new int (25);
     oat *ptr_f = new  oat(-10.1234F);
    char *ptr_c = new char('a');
    double *ptr_d = new double (1234.5667L);
    cout << "contents of the pointers " << endl;
    cout << "integer = " << *ptr_i << endl;
    cout << " oating point value  = " << *ptr_f << endl;
    cout << "char = " << *ptr_c << endl;
    cout << "double = " << *ptr_d << endl;
    delete ptr_i;
    delete ptr_f;
    delete ptr_c;
    delete ptr_d;
    return 0;
}
```

Output of the above program

```
contents of the pointers
integer = 25
 oating point value  = -10.1234
char = a
double = 1234.57
```

PROGRAM 11.33

A program to read two integers from the keyboard and perform simple arithmetic operations using the pointer technique. The memory space for the variables are allocated by the new operator.

```
//using new and delete operators
#include <iostream>
using namespace std;
int main()
{
    int *ptr_a = new int;
    int *ptr_b = new int;
    int *ptr_sum = new int;
    int *ptr_sub = new int;
    int *ptr_mult = new int;
     oat *ptr_div = new  oat;
    cout << " enter any two integers \n";
    cin >> *ptr_a >> *ptr_b;
    *ptr_sum = *ptr_a + *ptr_b;
    *ptr_sub = *ptr_a - *ptr_b;
    *ptr_mult = *ptr_a * *ptr_b;
    *ptr_div = ( oat)*ptr_a / ( oat)*ptr_b;
    cout << " Addition of (*ptr_a + *ptr_b) = " << *ptr_sum;
    cout << endl;
    cout << " Subtraction of (*ptr_a - *ptr_b) = " << *ptr_sub;
```

```
    cout << endl;
    cout << " Multiplication of (*ptr_a * *ptr_b) = " << *ptr_mult;
    cout << endl;
    cout << " Division of (*ptr_a / *ptr_b) = " << *ptr_div;
    cout << endl;
    delete ptr_a;
    delete ptr_b;
    delete ptr_sum;
    delete ptr_sub;
    delete ptr_mult;
    delete ptr_div;
    return 0;
}
```

Output of the above program

```
enter any two integers
1 2
Addition of (*ptr_a + *ptr_b) = 3
Subtraction of (*ptr_a - *ptr_b) = -1
Multiplication of (*ptr_a * *ptr_b) = 2
Division of (*ptr_a / *ptr_b) = 0.5
```

(c) Array Data Type When an object is an array data type, a pointer to its initial element is returned.

```
new int;
new int[20];
```

Both the expressions return a pointer to the first element of the array as

```
int *;
```

The general syntax of the new operator for the array data type is,

```
data_type pointer  = new data_type[size];
```

where `data_type` can be a short integer, float, char, array or even class objects, and size is the maximum number of elements that are to be accommodated.

For example,

(1) An expression to allocate a memory space for 20 integers using new operator.

```
int *ptr_a = new int[20];
```

(2) An expression to create memory space for 100 characters using new operator.

```
char *ptr_ch = new char[100];
```

(d) Use of New Operator to Allocate Memory for a Two-Dimensional Array A two-dimensional array can be declared using the new operator as,

```
new int [10][20];
```

which returns

```
int (*)[20];
```

The general syntax of the new operator for the two-dimensional array data type is,

```
data_type (pointer)[size]  = new data_type[size][size];
```

where `data_type` can be a short integer, float, char, array or even class objects and size is the maximum number of elements that are to be accommodated.

For example,

(1) An expression to allocate memory space for of 5×5 integers using new operator.

```
int (*ptr_a)[5] = new int [5][5];
```

(2) An expression to create memory space for 10×10 characters using new operator.

```
int (*ptr_c)[10] = new int [10][10];
```

The following section shows how the delete operator is used to destroy the objects created by the new operator for the array data type. The expression for the delete operator is same for both the one-dimensional and multidimensional arrays.

The general syntax of the delete operator for an array data type is,

```
delete [] ptr_a; //delete an array of pointer ptr_a;
```

For example, the following program segment shows how to use the delete operator in a one-dimensional array.

```
int main()
{
    char *ptr_ch = new char[100];
     oat *ptr_f = new  oat [20];
    -------
    -------
    delete [] ptr_ch;
    delete [] ptr_f;
}
```

For example, the following program segment shows how to use the delete operator in a two-dimensional array.

```
int main()
{
    int (*ptr_c)[10] = new int [10][10];
     oat (*ptr_f)[20] = new  oat [20][20];
    -------
    -------
    delete [] ptr_c;
    delete [] ptr_f;
}
```

PROGRAM 11.34

A program to allocate contiguous memory space for an array of integers using the new operator and the object of the array is destroyed by the delete operator. This program reads a set of numbers from the keyboard and displays it on the screen.

```
//using new and delete operators for array data type
#include <iostream>
using namespace std;
int main()
{
    int *ptr_a = new int[20];
    int *ptr_n = new int;
    cout << " how many numbers are there ? \n";
    cin >> *ptr_n;
    for ( int i = 0; i<= *ptr_n -1; ++i) {
        cout << " element a[" << i <<"] = ";
        cin >> ptr_a[i];
    }
    cout << " contents of the array \n";
    for (int i = 0; i<= *ptr_n -1; ++i) {
        cout << ptr_a[i] ;
        cout << '\t';
    }
    delete ptr_n;
    delete [] ptr_a;
```

```
    return 0;
}
```

Output of the above program

```
how many numbers are there?
5
element a[0] = 11
element a[1] = 22
element a[2] = 33
element a[3] = 44
element a[4] = 55

contents of the array
11  22  33  44  55
```

PROGRAM 11.35

This program demonstrates how memory is allocated for a multidimensional array of data elements using the new operator and destroying it using the delete operator. A program to read a two-dimensional matrices A and B; perform the matrix addition of these matrices and display the added elements on the screen.

```
//using new and delete operators for two dimensional
//array data type
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int (*ptr_a)[5] = new int [5][5];
    int (*ptr_b)[5] = new int [5][5];
    int (*ptr_c)[5] = new int [5][5];
    int *ptr_n = new int;
    cout << " order of the matrix ? \n";
    cin >> *ptr_n;
    cout << " enter the elements of A matrix \n";
    for ( int i = 0; i<= *ptr_n -1; ++i) {
        for ( int j = 0; j<= *ptr_n -1; ++j)
            cin >> ptr_a[i][j];
    }
    cout << " enter the elements of B matrix \n";
    for (int i = 0; i<= *ptr_n -1; ++i) {
        for (int j = 0; j<= *ptr_n -1; ++j)
            cin >> ptr_b[i][j];
    }
    // matrix addition
    for (int i = 0; i<= *ptr_n -1; ++i) {
        for (int j = 0; j<= *ptr_n -1; ++j) {
            ptr_c[i][j] = ptr_a[i][j] + ptr_b[i][j];
        }
    }
    cout <<  " Contents of the C matrix \n";
    for (int i = 0; i<= *ptr_n -1; ++i) {
      for (int j = 0; j<= *ptr_n -1; ++j) {
        cout << setw(3) << ptr_c[i][j];
      }
      cout << endl;
    }
    delete ptr_n;
    delete [] ptr_a;
    delete [] ptr_b;
```

```
    delete [] ptr_c;
    return 0;
}
```

Output of the above program

```
order of the matrix ?
3
enter the elements of A matrix
1  1  1  1
1  1  1  1
1  1  1  1
1  1  1  1
enter the elements of B matrix
2  2  2  2
2  2  2  2
2  2  2  2
2  2  2  2
Contents of the C matrix
3  3  3  3
3  3  3  3
3  3  3  3
3  3  3  3
```

PROGRAM 11.36

A program to create memory space for a class object using the new operator and to destroy it using the delete operator.

```
//using new and delete operators
#include <iostream>
using namespace std;
class sample {
    private :
       int x;
        oat y;
    public :
       void getdata();
       void display();
};
void sample :: getdata()
{
    cout << " enter an integer value \n";
    cin >> x;
    cout << " enter a  oating point number \n";
    cin >> y;
}

void sample :: display()
{
    cout << " entered numbers are \n";
    cout << " x = " << x << '\t' << " y = " << y << endl;
}

int main()
{
    sample *ptr;
    ptr = new sample;
    ptr->getdata();
```

```
    ptr->display();
    delete ptr;
    return 0;
}
```

Output of the above program

```
enter an integer value
10
enter a  oating point number
2.34
entered numbers are
x = 10     y = 2.34
```

11.8 THIS POINTER

It is well known that a pointer is a variable which holds the memory address of another variable. Using the pointer technique, one can access the data of another variables indirectly. The `this` pointer is a variable which is used to access the address of the class itself. Sometimes, the this pointer may have return data items to the caller. In other words, the this pointer is a pointer accessible only within the non-static member functions of a class, struct, or union type. It points to the object for which the member function is called. Static member functions do not have a this pointer. The general syntax of the this pointer is:

```
this
this->class_member
(*this).class_member
```

For example, the following assignment statements are equivalent:

```
void Date::set Month( int mn ) {
    month = mn;    // These three statements are equivalent
    this->month = mn;
    (*this).month = mn;
}
```

PROGRAM 11.37

A program to demonstrate how to use the this pointer for accessing the members of a class object.

```
#include <iostream>
using namespace std;
class date_info {
    public:
       int day,month,year;
    void setdate(int d,int m, int y);
       void display();
};

void date_info :: setdate(int d, int m, int y)
{
    //three assignment statements are equivalent
    day = d;
    this->month = m;
    (*this).year = y;
}

void date_info :: display()
{
```

```
    cout << "Today's date is : " << this->day << "/";
    cout << (*this).month << "/" << this->year << '\n';
}
int main()
{
    date_info obj;
    obj.setdate(10,10,2007);
    obj.display();
    return 0;
}
```

Output of the above program

```
Today's date is: 10/10/2007
```

An object's this pointer is not part of the object itself; it is not reflected in the result of a `sizeof` statement on the object. Instead, when a non-static member function is called for an object, the address of the object is passed by the compiler as a hidden argument to the function.

PROGRAM 11.38

A program to display the object's address of a class using this pointer.

```
//for accessing member data with this pointer
#include <iostream>
using namespace std;
class sample {
    private :
        int value;
    public :
        inline void display();
};

inline void sample ::display()
{
    this->value = 20;
    cout << "Contents of the value = " << this->value ;
    cout << endl;
}

int main()
{
    sample obj1;
    obj1.display();
    obj2.display();
    obj3.display();
    return 0;
}
```

Output of the above program

```
Object's address = 0x24e0fff2
Object's address = 0x24e0fff0
Object's address = 0x24e0ffee
```

The above program creates three objects, obj1, obj2, obj3 and displays each object's address using this pointer. The display () member function is used to give the address of the object. The this pointer can be treated like any other pointer to an object. The following is a valid declaration of the class object in C++.

```
inline void sample ::display()
{
      this->value = 20;
      cout << "Contents of the value = " << this->value ;
```

```
        cout << endl;
}
```

The above program segment is same as the following:

```
inline void sample ::display()
{
        value = 20;
        cout << "Contents of the value = " << value ;
        cout << endl;
}
```

PROGRAM 11.39

A program to demonstrate how the this pointer is used to access the member data of a class.

```
//for accessing member data with this pointer
#include <iostream>
using namespace std;
class sample {
    private :
       int value;
    public :
       inline void display();
};

inline void sample ::display()
{
    this->value = 20;
    cout << "Contents of the value = " << this->value ;
    cout << endl;
}

int main()
{
    sample obj1;
    obj1.display();
    return 0;
}
```

Output of the above program

```
Contents of the value = 20
```

The object's address is available from within the member function as the this pointer. Most uses of this are implicit. The expression *this is commonly used to return the current object from a member function:

```
return *this;
```

The `this` pointer is also used to guard against self-reference:

Type of this Pointer

(a) Const Whenever member data is declared as `const`, it is meant for read only purpose and it cannot be modified. The const member cannot invoke member functions that are not const.

(b) Volatile The volatile member data is loaded from memory each time it is accessed and it disables certain optimisations.

It is an error to pass a const object to a member function that is not const. Similarly, it is an error to pass a volatile object to a member function that is not volatile. Member functions declared as const cannot change member data—in such functions, the `this` pointer is a pointer to a const object.

Note that constructors and destructors cannot be declared as `const` or `volatile`. They can, however, be invoked on const or volatile objects.

11.9 MUTABLE

The keyword `mutable` is used to access the `const` data member of a class. We have already seen that the `this` keyword can only be applied to non-static and non-const data members of a class. If a data member is declared mutable, then it is legal to assign a value to this data member from a const member function.

```
mutable member-variable-declaration;
```

For example, the following code will compile without error because m_accessCount has been declared to be mutable, and therefore can be modified by GetFlag even though GetFlag is a const member function.

PROGRAM 11.40

A program to demonstrate how to use the mutable modifier for accessing the const member data of a class.

```
// mutable.cpp
#include <iostream>
using namespace std;
class abc
{
    public:
       void setdata(int a);
       void display();
       void GetFlag() const
       {
         m_accessCount++;
       }
    private:
       mutable int m_accessCount;
};

void abc :: setdata(int a)
{
     m_accessCount = a;
}

void abc::display()
{
     cout << "AccessCount = " << m_accessCount << "\n";
}

int main()
{
    abc obj;
    obj.setdata(100);
    obj.display();
    obj.GetFlag();
    return 0;
}
```

Output of the above program

```
AccessCount = 100
```

REVIEW QUESTIONS

1. Describe how the data member of a class can be initialised in C++.
2. What is a constructor? What are the uses of declaring a constructor member function in a program?
3. What are the rule governing the declaration of a constructor?
4. When does a constructor member function is invoked in a class?
5. In what way a constructor is different from an automatic initialisation?
6. List the merits and demerits of copy constructor.
7. What are the rules to be followed for declaring a copy constructor member function in C++
8. What is a default constructor?
9. Under what circumstances a default constructor is well suited for automatic initialisation of objects?
10. Explain the pros and cons of default constructors.
11. What is an `inline` substitution?
12. Explain the difference between the inline code over the macro.
13. Explain how an inline member function is defined in C++.
14. What is a static class member?
15. What are the merits and demerits of static data members over the global data variables?
16. Explain how a static member is defined and declared in C++.
17. What is a `friend` function?
18. Explain the pros and cons of declaring a `friend` class in a program.
19. How is a heap memory allocated in C++?
20. What are dynamic allocations operators?
21. What are the differences between the static and dynamic allocation of memory?
22. What is meant by `this` operator?
23. What are the places `this` pointer used?
24. Explain how data member of a class can be destroyed in C++.
25. What is a destructor and what are the uses of declaring a destructor member function in a program?
26. What are the rules governing the declaration of a destructor member function?
27. When does a destructor member function is invoked in a class?
28. In what way a destructor is different from a `delete` operator?
29. Explain the importance of `mutable` modifier.

CONCEPT REVIEW PROBLEMS

1. Determine the output of each of the following program when it is executed.
   (a)

```
#include <iostream>
using namespace std;
class abc
{
    abc() {
       cout << "Calling constructor\n";
    }
};
int main()
```

```
{
    abc obj;
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
class abc
{
    protected:
         abc() {
            cout << "Calling constructor\n";
         }
};
int main()
{
    abc obj;
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
class abc
{
    public:
       void abc() {
            cout << "Calling constructor\n";
       }
};
int main()
{
    abc obj;
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
class abc
{
    public:
        abc() {
           cout << "Calling constructor\n";
           return 0;
        }
};
int main()
{
    abc obj;
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
class abc
```

```
{
    public:
       abc() {
          cout << "Calling constructor\n";
          return;
       }
};
int main()
{
    abc obj;
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
class abc
{
    public:
       static abc() {
            cout << "Calling constructor\n";
       }
};
int main()
{
    abc obj;
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
class abc
{
    public:
       extern abc() {
          cout << "Calling constructor\n";
       }
};
int main()
{
    abc obj;
    return 0;
}
```

2. What will be the output of each of the following program when it is executed?

(a)

```
#include <iostream>
using namespace std;
union abc
{
   abc() {
       cout << "Calling constructor\n";
   }
};
int main()
{
```

```
    abc obj;
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
struct abc
{
    abc() {
       cout << "Calling constructor\n";
    }
};
int main()
{
    abc obj;
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
struct abc
{
    ~abc() {
        cout << "Calling destructor\n";
    }
};
int main()
{
    abc obj;
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
struct abc
{
    ~abc() {
        cout << "Calling destructor\n";
        return;
    }
};
int main()
{
    abc obj;
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
struct abc
{
    virtual ~abc() {
        cout << "Calling destructor\n";
        return;
```

```
    }
};
int main()
{
    abc obj;
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
struct abc
{
    virtual abc() {
        cout << "Calling constructor \n";
    }
};
int main()
{
    abc obj;
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
struct abc {
    abc();
};

abc :: abc()
{
    struct sample {
        sample() {
           cout << " deep constructor \n";
        }
    };
}
int main()
{
    abc obj;
    return 0;
}
```

(h)

```
#include <iostream>
using namespace std;
struct abc {
    abc();
};

abc :: abc()
{
    struct sample {
        sample();
    };
}
abc::sample :: sample()
```

```
{
    cout << " deep constructor \n";
}
int main()
{
    abc obj;
    return 0;
}
```

3. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
class abc {
    public:
       abc();
       class x {
          public:
             x();
            class y{
               public:
                  y();
            };
       };
};
abc:: abc()
{
    cout << "abc - class constructor \n";
}
abc::x :: x()
{
    cout << "x - class constructor \n";
}
abc::x:: y:: y()
{
    cout << " y - class constructor \n";
}
int main()
{
    abc obj;
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
class abc {
    public:
       abc();
       class x {
           public:
              x();
    };
};
abc:: abc()
{
```

```
    cout << "abc - class constructor \n";
}
abc::x :: x()
{
    cout << "x - class constructor \n";
}

int main()
{
    abc::x obj2;
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
union abc {
    ~abc();
     union x {
          ~x();
           union y {
                ~y();
                 union z {
                      ~z();
                 };
           };
     };
};

abc:: ~abc()
{
    cout << "abc - class destructor \n";
}

abc::x :: ~x()
{
    cout << "x - class destructor \n";
}

abc::x ::y :: ~y()
{
    cout << "y - class destructor \n";
}

abc::x ::y :: z :: ~z()
{
    cout << "z - class destructor \n";
}

int main()
{
    abc obj1;
    abc::x  obj2;
    abc::x::y obj3;
    abc::x::y::z  obj4;
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
union abc {
    ~abc();
     struct x {
         ~x();
          union y {
               ~y();
                struct z {
                     ~z();
                };
          };
    };
};

abc:: ~abc()
{
    cout << "abc - class destructor \n";
}

abc::x :: ~x()
{
    cout << "x - class destructor \n";
}

abc::x ::y :: ~y()
{
    cout << "y - class destructor \n";
}

abc::x ::y :: z :: ~z()
{
    cout << "z - class destructor \n";
}

int main()
{
    abc obj1;
    abc::x  obj2;
    abc::x::y obj3;
    abc::x::y::z  obj4;
    return 0;
}
```

PROGRAMMING EXERCISES

1. Write an object-oriented program in C++ that prints the factorial of a given number using a constructor and a destructor member function.
2. Write an object-oriented program in C++ that prints the factorial of a given number using a copy constructor and a destructor member function.

3. Write an object-oriented program in C++ that determines whether a given number is a prime or not and prints using default constructor and destructor member functions.
4. Write an object-oriented program in C++ to read any five real numbers and print the average using a static member class.
5. Write an object-oriented program in C++ to find the sum of the following series using
   - constructor member function
   - copy constructor
   - default constructor member function
   - destructor member function
   - inline member function

   (a) sum = $1 + 2 + 3 + \cdots + n$

   (b) sum = $1 + 3 + 5 + \cdots + n$

   (c) sum = $1 + 2 + 4 + \cdots + n$

   (d) sum = $1 - \frac{1}{1!} + \frac{2}{2!} - \frac{3}{3!} \cdots \frac{n}{n!}$

   (e) sum = $x + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots \frac{x^n}{n!}$

   (f) sum = $x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots \frac{x^n}{n!}$

   (g) sum = $1^2 + 2^2 + 3^2 + 4^2 + \cdots + n^2$

   (h) sum = $1^3 + 2^3 + 3^3 + 4^3 + \cdots + n^3$

   (i) sum = $1 + 2^2 + 4^2 + \cdots + n^2$

   (j) sum = $1 + 3^2 + 5^2 + \cdots + n^2$
6. Write an object-oriented program in C++ to generate the following series of numbers using constructor, destructor and inline member functions.

(i)

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1
7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
```

(ii)

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
```

(iii)

```
9 8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
7 6 5 4 3 2 1
6 5 4 3 2 1
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

(iv)

```
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

7. Write an object-oriented program in C++ to read an integer number and find the sum of all the digits until it reduces to a single digit using a constructor, destructor, default constructor and inline member functions.
For example,
(i) n = 1256
sum = 1+2+5+6 = 14
sum = 1+4 = 5
(ii) n = 7896
sum = 7+8+9+6 = 30
sum = 3+0 = 3

8. Write an object-oriented program in C++ to read a number n and print it digit by digit in words using inline member function. For example, consider the number 756, which should be printed as "Seven Five Six".

9. Write an object-oriented program in C++ to read a set of numbers up to `n` (where `n` is defined by the programmer) and print the contents of the array in the reverse order using dynamic memory allocation operators new and delete.
For example, for n = 4, let the set be
26 56 51 123 should be printed as
123 51 56 26

10. Write an object-oriented program in C++ to read n numbers (where `n` is defined by the programmer) and find the average of the non-negative integer numbers. Also find, the deviation of the numbers using new and delete operators.

11. Write an object-oriented program in C++ to read a set of numbers and store it in a one-dimensional array; again read a number '`d`' and check whether the number `d` is present in the array or not. If it is so, print out how many times the number `d` is repeated in the array using new and delete operators.

12. Write an object-oriented program in C++ to read a set of numbers and store it in a one-dimensional array; again read a number n and check whether it is present in the array or not. If it is so, print out the position of `n` in the array and also check whether it repeats in the array using new and delete operators.

13. Write an object-oriented program in C++ to read a set of numbers and store it in a one-dimensional array; find the largest and the smallest number and the difference of the two numbers. Using the difference, find the deviation of the numbers of the array, through new and delete operators.

14. Write an object-oriented program in C++ to read a two-dimensional square matrix A and display its transpose using new and delete operators.

15. Write an object-oriented program in C++ to read a two-dimensional array; find the sum of the elements row-wise and column-wise separately, and display the sums using new and delete operators.

16. Write an object-oriented program in C++ to generate a magic square `A` (where the sum of the elements along with the row-wise and column-wise is the same) using new and delete operators.

17. Write an object-oriented program in C++ to read a set of lines and find out the number of characters, words, and lines in a given text using static member class.

18. Write an object-oriented program in C++ to read a line and find out the number of vowels (a, e, i, o, u) and consonants present in the given line using static member functions.

19. Write an object-oriented program to perform trigonometric operations on complex numbers using a friend function.

20. Write an object-oriented program with constructor, default constructor, copy constructor and

destructor to read a set of lines from stdin and store them in an array A; again read a string S from the stdin and check whether the given string S is in the array A. If it is, print that line and also how many times it has been repeated in the array A.

21. Write an object-oriented program with constructor, default constructor, copy constructor and destructor to read a set of lines from stdin and store them in an array A; again read a string S from the stdin and check whether the given string S is in the array A. If it is, remove the string S from the array A and print the updated array on the stdout. For example,

```
A = concatenate
S = cat
The updated a is conenate
```

22. Write an object-oriented program with constructor, default constructor, copy constructor and destructor to read a set of lines from stdin and store them in an array A; again read two strings S1 and S2 from the stdin and check whether the given string S1 is in the array A. If it is, replace the string S1 with the string S2 and print the updated array.

For example,

```
A  = concatenate
S1 = cat
S2 = 123
```

The updated A is con123enate

23. Develop an object-oriented program in C++ to read the following information from the keyboard:

employee name
employee code
designation
years of experience
age

Construct the data base with suitable member functions for initialising and for destroying the data, viz. constructor, default constructor, copy constructor, destructor, static member functions, friend class, this pointer, inline code and dynamic memory allocation operators—new and delete.

24. Develop an object-oriented program in C++ to create a data base of the following items:

name of the patient
sex
age
ward number
bed number
nature of the illness
date of admission

Construct the data base with suitable member functions for initialising and destroying the data, viz. constructor, default constructor, copy constructor, destructor, static member functions, friend class, this pointer, inline code and dynamic memory allocation operators—new and delete.

25. Develop an object-oriented program in C++ to create a pay roll system of an organisation assuming that the following information can be read from the keyboard:

employee name
employee code
designation
account number
date of joining
basic pay

DA, HRA and CCA
deductions like PPF, GPF, CPF, LIC, NSS, NSC, etc.

Construct the data base with suitable member functions for initialising and destroying the data, viz. constructor, default constructor, copy constructor, destructor, static member functions, friend class, this pointer, inline code and dynamic memory allocation operators—`new` and `delete`.

26. Develop an object-oriented program in C++ to prepare the mark sheet of a university examination assuming that the following items can be read from the keyboard:

name of the student
roll number
subject code
subject name
internal marks
external marks

Construct the data base with suitable member functions for initialising and destroying the data, viz. constructor, default constructor, copy constructor, destructor, static member functions, friend class, this pointer, inline code and dynamic memory allocation operators—`new` and `delete`.

27. Develop an object-oriented program in C++ to create a library information system containing the following for all books in the library:

accession number
name of the author
title of the book
year of publication
publisher's name
cost of the book

Construct the data base with suitable member functions for initialising and destroying the data, viz. constructor, default constructor, copy constructor, destructor, static member functions, friend class, this pointer, inline code and dynamic memory allocation operators—`new` and `delete`.

28. Develop an object-oriented program in C++ to create a data base of the personnel information system containing the following information:

name
date of birth
blood group
height
weight
insurance policy number
contact address
telephone number
driving licence number, etc.

Construct the data base with suitable member functions for initialising and destroying the data, viz. constructor, default constructor, copy constructor, destructor, static member functions, friend class, this pointer, inline code and dynamic memory allocation operators—`new` and `delete`.

Single and Multiple Inheritance

Chapter 12

Inheritance, one of the most powerful features of object-oriented programming (OOP) is introduced and explained. The focus is on how a class can be defined and declared as a base class and derived class for implementing the mechanism of single inheritance and multiple inheritance. The syntax and semantic rules of the class hierarchies and levels of inheritance are discussed and illustrated with numerous examples.

12.1 INTRODUCTION

It is well known that the C++ is a tool for object-oriented programming that supports most of the OOP features such as data hiding, data encapsulation, inheritance, polymorphism, virtual functions, etc. Data hiding and encapsulation are important features of the object-oriented programming and how these concepts are implemented using classes are explained in Chapter 10 on "Classes and objects". The class alone is not enough to support and design OOP. In order to maintain and reuse the class objects easily, it is required to relate disperate classes into another. This chapter presents the salient features of the inheritance that probably is the most powerful feature of the object-oriented programming. Inheritance is the process of creating new classes from an existing class. The existing class is known as the base class and the newly created class is called as a derived class.

The general representation of single and multiple inheritance is given in Fig. 12.1(a) and (b).

In some OOP languages such as Simula, the base and derived classes are called as super and subclasses respectively. The derived class inherits all capabilities of the base class. It can also add some more features to this class. The base class is unchanged by its process.

The main advantages of the inheritance are:

- reusability of the code,
- to increase the reliability of the code, and
- to add some enhancements to the base class.

Once the base class is written and debugged, it need not be changed again when there are circumstances to add or modify the member of the class.

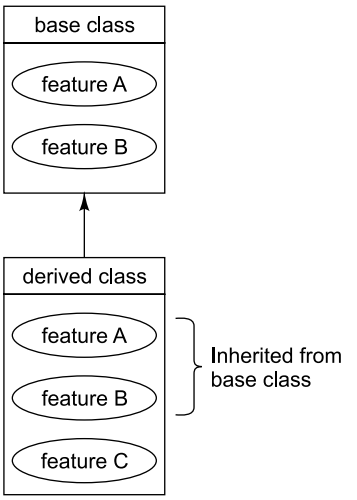


Fig. 12.1(a) *Single Inheritance*

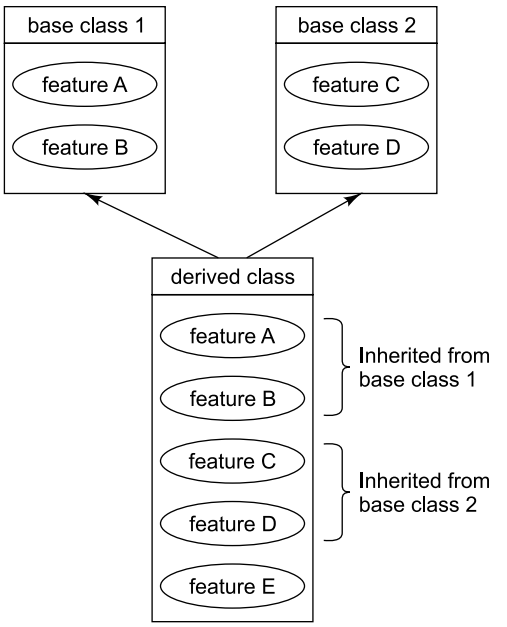


Fig. 12.1(b) *Multiple inheritance*

12.2 SINGLE INHERITANCE

Single inheritance is the process of creating new classes from an existing base class. The existing class is known as the direct base class and the newly created class is called as a singly derived class.

Single inheritance is the ability of a derived class to inherit the member functions and variables of the existing base class.

Defining the Derived Class The declaration of a singly derived class is as that same of an ordinary class. A derived class consists of the following components:

(i) the keyword class
(ii) the name of the derived class
(iii) a single colon
(iv) the type of derivation (private, public or protected)
(v) the name of the base or parent class
(vi) the remainder of the class definition

The general syntax of the derived class declaration is as follows.

```
class  derived_class_name : private/public/protected  base_class_name
{
      private :
            // data members
      public :
            // data members
            // methods
      protected :
            // data members
 };
```

For example, the following program segment illustrates the declaration of the single inherited class. The base class consists of two parts — data member and member function. The data member consists of name, rollno and sex and are defined as a private group. The member functions getdata () and putdata () are declared in the base class. The members of a base class can be referred as if they were members of the derived class.

```
class  basic_info {
   private :
      char name[20];
      long int rollno;
      char sex;
   public :
      void getdata();
      void display();
}; // end of class de nition
class physical_ t :public basic_info
{
   private :
       oat height;
       oat weight;
   public:
      void getd();
      void disp();
}; //end of class de nition
```

The derived class inherits the properties of its base classes, including its data member and member functions. The physical_fit is a derived class which has two components — private and public. In addition to its new data members such as height and weight, it may inherit the data members of the base class. The derived class contains not only the methods of its own but also of its base classes.

PROGRAM 12.1

A program to read the data members of a basic class such as name, roll number and sex from the keyboard and display the contents of the class on the screen. This program does not use any inheritance concepts.

```
//program 1.cpp
#include <iostream>
#include <iomanip>
using namespace std;
class basic_info {
   private :
      char name[20];
      long int rollno;
      char sex;
   public :
      void getdata();
      void display();
 }; // end of class de nition

void basic_info :: getdata()
{
     cout << " enter a name ? \n";
     cin >> name;
     cout << " roll no ? \n";
     cin >> rollno;
     cout << " sex ? \n";
     cin >> sex;
}
void basic_info :: display()
{
     cout << name << "      ";
     cout << rollno << "      ";
     cout << sex << "   ";
}

int main()
{
    basic_info   a;
    cout << "enter the following information \n";
    a.getdata();
    cout << " ———————————— \n";
    cout << "  Name    Rollno    Sex  \n";
    cout << " ———————————— \n";
    a.display();
    cout << endl;
    cout << " ———————————— \n";
    return 0;
}
```

Output of the above program

```
enter the following information
enter a name?
Ravich
roll no?
```

```
20071
sex?
M
-----------------------
Name      Rollno     Sex
-----------------------
Ravich    20071      M
-----------------------
```

PROGRAM 12.2

A program to read the derived class data members such as name, roll number, sex, height and weight from the keyboard and display the contents of the class on the screen. This program demonstrates a single inheritance concept which consists of a base class and a derived class.

The OMT of a class notation for single inheritance is given in Fig. 12.2.

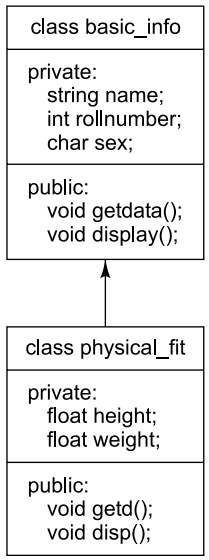


Fig. 12.2 *OMT of a Class Notation for Single Inheritance*

```
//single inheritance
#include <iostream>
#include <iomanip>
using namespace std;
class  basic_info {
   private :
       char name[20];
       long int rollno;
       char sex;
   public :
       void getdata();
       void display();
 }; // end of class de nition

class  physical_ t :public basic_info
```

```
{
   private :
        oat height;
        oat weight;
   public:
       void getdata();
       void display();
 }; //end of class de nition

void basic_info :: getdata()
{
   cout << " enter a name ? \n";
   cin >> name;
   cout << " roll no ? \n";
   cin >> rollno;
   cout << " sex ? \n";
   cin >> sex;
}
void basic_info :: display()
{
   cout << name << "      ";
   cout << rollno << "      ";
   cout << sex << "   ";
}

void physical_ t :: getdata()
{
   basic_info::getdata();
   cout << " height ? \n";
   cin >> height;

   cout << " weight ?\n";
   cin >> weight;
}

void physical_ t :: display()
{
   basic_info::display();
   cout << height << "    ";
   cout << weight << "    ";
}

int main()
{
    physical_ t   a;
    cout << "enter the following information \n";
    a.getdata();
    cout << " ———————————————————— \n";
    cout << "  Name    Rollno      Sex  Height Weight \n";
    cout << " ———————————————————— \n";
    a.display();
    cout << endl;
    cout << " ———————————————————— \n";
    return 0;
}
```

Output of the above program

```
enter the following information
enter a name?
Kandasamy
roll no?
27003
sex?
M
height?
```

```
179
weight?
78
-----------------------------------------
Name          Rollno    Sex    Height   Weight
-----------------------------------------
Kandasamy     27003      M      179       78
-----------------------------------------
```

12.3 TYPES OF BASE CLASSES

Any class can serve as a base class. A derived class may be defined as a base of another class. A base class can be classified into two types, direct base and indirect base. In this section, how these base classes are realised in C++, are explained in detail.

12.3.1 Direct Base Classes

A base class is called a direct base if it is mentioned in the base list.

For example, following are valid derived class declarations:

(1)

```
class baseA {
   -------
   -------
};

class derivedB : public baseA {
   -------
   -------
};
```

where class baseA is a direct base.

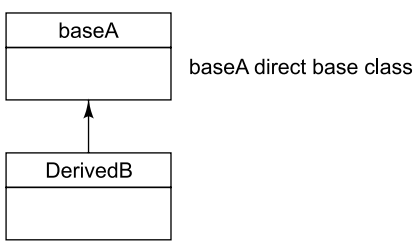


Fig. 12.3(a) *Direct Base Class*

(2)

```
class baseA {
   -------
   -------
};

class baseB  {
   -------
   -------
```

```
};

class derivedC : public baseA, public baseB {
   -------
   -------
};
```

where both classes baseA and baseB are the direct base.

(3) A class may be derived from any number of base classes. For example,

```
class baseA {
   -------
   -------
};

class baseB {
   -------
   -------
};
class baseC {
   -------
   -------
};

class baseD {
   -------
   -------
};

class derivedE : public baseA, public baseB,
      public baseC, public baseD
{
   -------
   -------
};
```

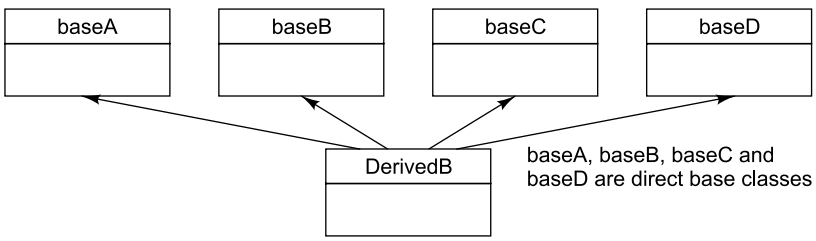


Fig. 12.3(b) *Direct Base Class*

The following are **invalid** declarations:

Note that a class which has been named but not yet declared cannot be used as a base class.

(i)

```
class baseA ;
```

```
class derivedB : public baseA {
   -------
   -------
};
```

The base class baseA is undeclared and an error message will be displayed by the compiler.

(ii) A class may not be specified as a direct base class of a derived class more than once.

```
class baseA {
   -------
   -------
};
class derivedB :public baseA,public baseA {
   -------
   -------
};
```

The base class baseA has been declared twice as the direct base of the derived class derivedB. It is an invalid way of constructing a derived class in C++.

12.3.2 Indirect Base Classes

A derived class can itself serve as a base class subject to access control. When a derived class is declared as a base of another class, the newly derived class inherits the properties of its base classes including its data members and member functions. A class is called as an indirect base if it is not a direct base, but is a base class of one of the classes mentioned in the base list.

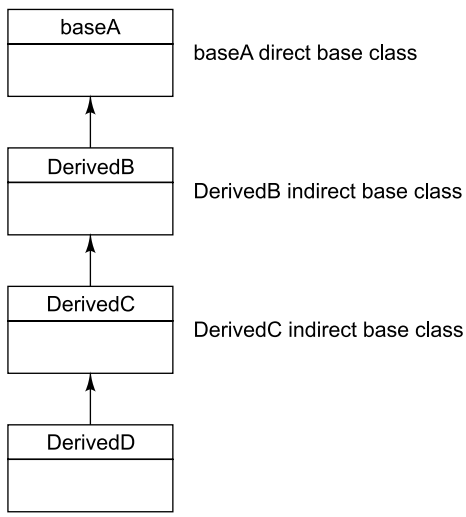


Fig. 12.4(a) *Indirect Base Class*

For example, following are valid declarations:

(1)

```
class baseA {
   -------
   -------
};
class derivedB : public baseA {
```

```
    -------
    -------
};

class derivedC : public derivedB {
    -------
    -------
};
```

Note that the class derivedB is a base of the class derivedC that is called as an indirect base.

(2) A class may be specified as an indirect base more than once.

For example, the following declaration is also valid:

```
class baseA {
    -------
    -------
};

class baseB : public baseA {
    -------
    -------
};

class baseC : public baseA {
    -------
    -------
};

class derivedD : public baseB,
      public baseC {
    -------
    -------
};
```

The class baseA has inherited both the derived classes baseB and baseC, in the sense that inheritance means building new abstractions from old ones, where one class inherits data and member functions from another.

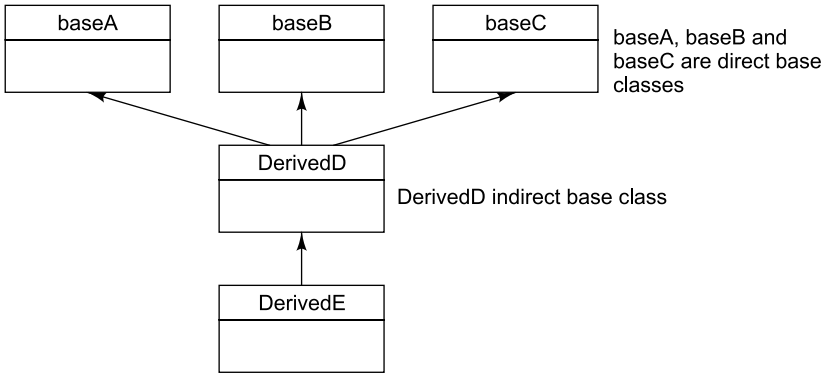


Fig. 12.4(b) *Indirect Base Class*

PROGRAM 12.3

A program to get the information of a derived class data members via name, roll number, sex, height and weight from the keyboard; again to read data for another derived class such as name, roll number, sex, course, semester and rank and display the contents of the newly created class on the screen.

This program demonstrates a single inheritance concept which consists of a single base class and two derived classes. The data members and member functions of the base class are independently accessed by these two derived classes. The OMT of a class notation for single inheritance is given in Fig. 12.5.

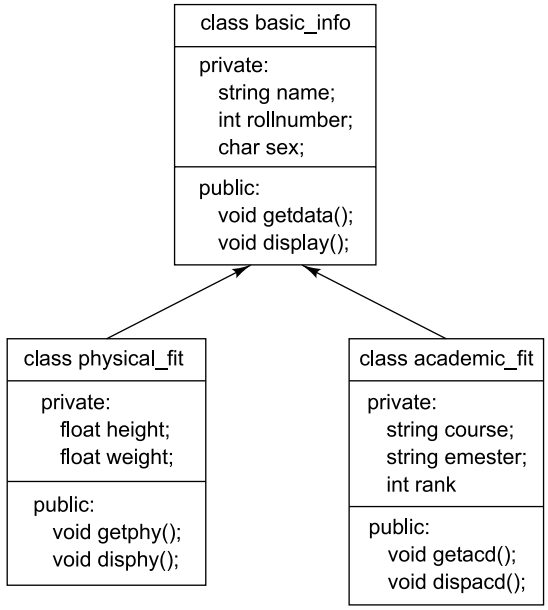


Fig. 12.5 *OMT of a Class Notation for Single Inheritance*

```
// two derived classes access the same base class
//single inheritance
#include <iostream>
#include <iomanip>
using namespace std;
class  basic_info {
   private :
      char name[20];
      long int rollno;
      char sex;
   public :
      void getbaseinfo();
      void dispbaseinfo();
}; // end of class de nition

class  physical_ t :public basic_info
{
   private :
       oat height;
       oat weight;
```

```
   public:
       void getphy();
       void disphy();
}; //end of class de nition

class academic_ t : public basic_info
{
   private :
      char course[20];
      char semester[10];
      int rank;
   public :
       void getacd();
       void dispacd();
}; // end of class de nition

void basic_info :: getbaseinfo()
{
   cout << " enter a name ? \n";
   cin >> name;
   cout << " roll no ? \n";
   cin >> rollno;
   cout << " sex ? \n";
   cin >> sex;
}
void basic_info :: dispbaseinfo()
{
   cout << name << "      ";
   cout << rollno << "      ";
   cout << sex << "   ";
}

void physical_ t :: getphy()
{
   basic_info::getbaseinfo();
   cout << " height ? \n";
   cin >> height;
   cout << " weight ?\n";
   cin >> weight;
}

void physical_ t :: disphy()
{
   basic_info::dispbaseinfo();
   cout << height << "    ";
   cout << weight << "    ";
}

void academic_ t :: getacd()
{

   basic_info::getbaseinfo();
   cout << " course name ( BTech/MCA/DCA etc) ?\n";
   cin >> course;
   cout << " semester ( rst/second etc)? \n";
   cin >> semester;
   cout << " rank of the student \n";
   cin >> rank;
}

void academic_ t :: dispacd()
{
   basic_info :: dispbaseinfo();
   cout << course << "      " ;
   cout << semester << "     ";
   cout << rank << "     ";
```

```
}
int main()
{

   physical_ t   p;
   academic_ t   a;
   cout << "enter the following information for physical  tness\n";
   p.getphy();
   cout << "enter the following information for academic  tness\n";
   a.getacd();
   cout << " Physical Fitness of the student \n";
   cout << " ------------------------ \n";
   cout << "  Name    Rollno   Sex  Height Weight \n";
   cout << " ------------------------ \n";
   p.disphy();
   cout << endl;
   cout << " ------------------------ \n";
   cout << endl;
   cout << " Academic performance of the student \n";
   cout << " ------------------------------ \n";
   cout << "  Name    Rollno   Sex   Course   Semester  Rank \n";
   cout << " ------------------------------ \n";
   a.dispacd();
   cout << endl;
   cout << " ------------------------------ \n";
   return 0;
}
```

Output of the above program

```
enter the following information for physical  tness
enter a name?
Velusamy.K
roll no?
27001
sex?
M
height?
175
weight?
90
enter the following information for academic  tness
enter a name?
Velusamy.K
roll no?
27001
sex?
M
course name ( BTech/MCA/DCA etc)?
B.Tech
semester ( rst/second etc)?
I
rank of the student
2
Physical Fitness of the student
-------------------------------------------------
  Name          Rollno     Sex     Height    Weight
-------------------------------------------------
  Velusamy.K    27001       M       175       90
-------------------------------------------------
```

```
Academic performance of the student
-------------------------------------------------------------
  Name          Rollno     Sex     Course     Semester     Rank
-------------------------------------------------------------
  Velusamy.K    27001       M      B.Tech        I          2
-------------------------------------------------------------
```

12.4 TYPES OF DERIVATION

Inheritance is a process of creating a new class from an existing class. While deriving the new classes, the access control specifier gives the total control over the data members and methods of the base classes. A derived class can be defined with one of the access specifiers, viz. private, public and protected.

12.4.1 Public Inheritance

The most important type of access specifier is public. In a public derivation,

- each public member in the base class is public in the derived class.
- each protected member in the base class is protected in the derived class.
- each private member in the base class remains private in the base class.

The general syntax of the public derivation is:

```
class base_class_name {
   -------
   -------
}
class derived_class_name : public base_class_name
{
   -------
   -------
}
```

For example, the following program segment illustrates how to access each member of the base class by the derived class members:

```
class baseA {
   private :
        int x;
   protected:
        int y;
   public :
        int z;
};
class derivedD : public baseA { // public base class
   private :
        int w;
};
```

The class derivedD is derived from the base class baseA and the access specifier is public. The data members of the derived class derivedD is

```
int x;
int y;
int z;
int w;
```

The following table shows the access specifier of the data member of the base class in the derived class:

Member	*Access status in derivedD*	*How obtained*
x	not accessible	from class baseA
y	protected	from class baseA
z	public	from class baseA
w	private	added by class derivedD

12.4.2 Private Inheritance

In a private derivation,

- each public member in the base class is private in the derived class.
- each protected member in the base class is private in the derived class.
- each private member in the base class remains private in the base class and hence it is visible only in the base class.

The general syntax of the private derivation is,

```
class base_class_name {
   -------
   -------
}
class derived_class_name : private base_class_name
{
   -------
   -------
}
```

For example, the following program segment illustrates how to access each member of the base class by the derived class members.

```
class baseA {
   private :
        int x;
   protected:
        int y;
   public :
        int z;
};
class derivedD : private baseA { // private base class
   private :
        int w;
};
```

The class derivedD is derived from the base class baseA and the access specifier is private. The data members of the derived class derivedD is

```
int x;
int y;
int z;
int w;
```

The following table shows the access specifier of the data member of the base class in the derived class:

Member	*Access status in derivedD*	*How obtained*
x	not accessible	from class baseA
y	private	from class baseA
z	private	from class baseA
w	private	added by class derivedD

12.4.3 Protected Inheritance

In a protected inheritance,

- each public member in the base class is protected in the derived class.
- each protected member in the base class is protected in the derived class.
- each private member in the base class remains private in the base class and hence it is visible only in the base class.

The general syntax of the protected derivation is,

```
class base_class_name {
    -------
    -------
}
class derived_class_name : protected base_class_name
{
    -------
    -------
}
```

For example, the following program segment illustrates how to access each member of the base class by the derived class members.

```
class baseA {
    private :
       int x;
    protected:
       int y;
    public :
       int z;
};
class derivedD : protected baseA {  //protected base class
    private :
       int w;
};
```

The class derivedD is derived from the base class baseA and the access specifier is protected. The data members of the derived class derived is

```
int x;
int y;
int z;
int w;
```

The following table shows the access specifier of the data member of the base class in the derived class:

Member	*Access status in derivedD*	*How obtained*
x	not accessible	from class baseA
y	protected	from class baseA
z	protected	from class baseA
w	private	added by class derivedD

12.5 AMBIGUITY IN SINGLE INHERITANCE

Whenever a data member and member function are defined with the same name in both the base and the derived classes, these names must be without ambiguity. The scope resolution operator (::) may be used to refer to any base member explicitly. This allows access to a name that has been redefined in the derived class.

For example, the following program segment illustrates how ambiguity occurs when the `getdata ()` member function is accessed from the `main ()` program.

```
class baseA {
   public :
      int i;
      getdata();
};
class baseB {
   public :
      int i;
      getdata();
};
class derivedC : public baseA, public baseB {
   public :
      int i;
      getdata();
}
int main()
{
   derivedC obj;
   obj.getdata();
   return 0;
}
```

The members are ambiguous without scope operators. When the member function `getdata ()` is accessed by the class object, naturally, the compiler cannot distinguish between the member function of the class `baseA` and the class `baseB`. Therefore it is essential to declare the scope operator explicitly to call a base class member as illustrated below:

```
obj.baseA::getdata();
obj.baseB::getdata();
```

PROGRAM 12.4

A program to demonstrate how ambiguity is avoided in single inheritance using scope resolution operator.

```
// ambiguty in the single inheritance
#include <iostream>
using namespace std;
class baseA {
   private :
      int i;
   public :
      void getdata(int x);
      void display();
};
class baseB {
   private :
      int j;
   public :
      void getdata(int y);
      void display();
};
class derivedC : public baseA,public baseB
{
};

void baseA :: getdata(int x)
{
   i = x;

}
void baseA :: display()
{
   cout << " value of i = "  << i << endl;
}

void baseB :: getdata(int y)
{
     j = y;
}

void baseB :: display()
{
   cout << " value of j = "  << j << endl;
}

int main()
{
   derivedC objc;
   int x,y;
   cout << " enter a value for i ?\n";
   cin >> x;
   objc.baseA::getdata(x); // member is ambiguous without scope
   cout << " enter a value for j ?\n";
   cin >> y;
   objc.baseB::getdata(y);
   objc.baseA::display();
   objc.baseB::display();
   return 0;
}
```

Output of the above program

```
enter a value for i?
10
enter a value for j?
20
value of i = 10
value of j = 20
```

12.6 ARRAY OF CLASS OBJECTS AND SINGLE INHERITANCE

This section emphasises on how an array of class objects can be inherited from a base class. Once a derived class has been defined, the way of accessing a class member of the array of class objects are same as the ordinary class types. The general syntax of the array of class objects in a singly derived class is:

```
class baseA {
   private :
        -------
        -------
   public :
        -------
        -------
};
class derivedB : public baseA  {
   private :
        -------
        -------
   public :
        -------
        -------
};
int main()
{
   derivedB obj[100];//array of class objects of the derived class
   -------
   -------
   return 0;
}
```

PROGRAM 12.5

A program consists of a base class and a derived class. The base class data members are name, roll number and sex. The derived class data members are height and weight. The derived class has been declared as an array of class objects. The member functions are used to get information on the derived class from the keyboard and display the contents of the array of class objects on the screen.

This program illustrates how to define an array of class objects using a single inheritance.

```
//single inheritance using array of objects
#include <iostream>
#include <iomanip>
using namespace std;
const int  MAX = 100;
class  basic_info {
   private :
      char name[20];
      long int rollno;
      char sex;
   public :
      void getdata();
      void display();
}; // end of class de nition
class  physical_ t :public basic_info
{
```

```
   private :
       oat height;
       oat weight;
   public:
      void getdata();
      void display();
}; //end of class de nition

void basic_info :: getdata()
{
   cout << " enter a name ? \n";
   cin >> name;
   cout << " roll no ? \n";
   cin >> rollno;
   cout << " sex ? \n";
   cin >> sex;
}
void basic_info :: display()
{
   cout << name << "     ";
   cout << rollno << "     ";
   cout << sex << "  ";
}

void physical_ t :: getdata()
{
   basic_info::getdata();
   cout << " height ? \n";
   cin >> height;
   cout << " weight ?\n";
   cin >> weight;
}

void physical_ t :: display()
{
   basic_info::display();
   cout << height << "   ";
   cout << weight << "   ";
}

int main()
{
   physical_ t a[MAX];
   int i,n;
   cout << " How many students ? \n";
   cin >> n;
   cout << "enter the following information \n";
   for (i = 0; i <= n-1; ++i) {
        cout << "record : " << i+1 << endl;
        a[i].getdata();
   }
   cout << endl;
   cout << " ------------------------------------ \n";
   cout << "  Name    Rollno   Sex  Height Weight \n";
   cout << " ------------------------------------ \n";
   for (i = 0; i <= n-1; ++i) {
        a[i].display();
        cout << endl;
   }
   cout << endl;
   cout << " ------------------------------------ \n";
   return 0;
}
```

Output of the above program

```
How many students?
3
enter the following information
record: 1
   enter a name?
   Antony
   roll no?
   20071
   sex?
   M
   height?
   167
   weight?
   65

record: 2
   enter a name?
   Sulaiman
   roll no?
   20073
   sex?
   M
   height?
   179
   weight?
   90

record: 3
   enter a name?
   Sandeep
   roll no?
   20076
   sex?
   M
   height?
   187
   weight?
   78
-----------------------------------------------------------------
 Name         Rollno     Sex     Height     Weight
-----------------------------------------------------------------
 Antony      20071        M        167         65
 Sulaiman    20073        M        179         90
 Sandeep     20076        M        187         78
-----------------------------------------------------------------
```

12.7 MULTIPLE INHERITANCE

In the previous section, we have seen how a derived class could inherit more enhancements and additional features from the base class. In the original implementation of C++, a derived class could inherit from only one base class. Even with this restriction, the object-oriented paradigm is a flexible and powerful

programming tool. The latest version of the C++ compiler implements the multiple inheritance. In this section, how a class can be derived from more than one base class is explained.

Multiple inheritance is the process of creating a new class from more than one base classes. The syntax for multiple inheritance is similar to that of single inheritance. For example, the following program segment shows how a multiple inheritance is defined:

```
class baseA {
   -------
   -------
};
class baseB {
   -------
   -------
};

class C : public baseA,public baseB
{
   -------
   -------
};
```

The class C is derived from both classes baseA and baseB.

Multiple inheritance is a derived class declared to inherit properties of two or more parent classes (base classes). Multiple inheritance can combine the behaviour of multiple base classes in a single derived class. Multiple inheritance has many advantages over the single inheritance such as rich semantics and the ability to directly express complex structures. In C++, derived classes must be declared during the compilation of all possible combinations of derivations and the program can choose the appropriate class at run time and create object for the application.

The abstract class representation of single and multiple inheritance is given in Fig. 12.6(a) and (b).

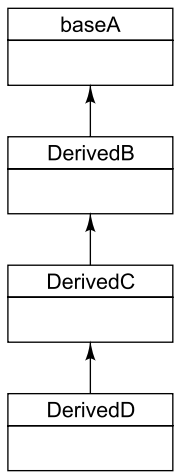


Fig. 12.6(a) *Abstract Class Representation of Single Inheritance*

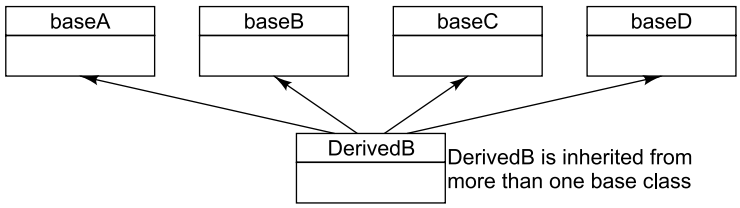


Fig. 12.6(b) *Abstract Class Representation of Multiple Inheritance*

In a single inheritance, a derived class has a single base class. In multiple inheritance, a derived class has multiple base classes. In a single inheritance hierarchy, a derived class typically represents a specialisation of its base class. In a multiple inheritance hierarchy, a derived class typically represents a combination of its base classes.

The rules of inheritance and access do not change from a single to a multiple inheritance hierarchy. A derived class inherits data members and methods from all its base classes, regardless of whether the inheritance links are private, protected or public.

(1) Multiple inheritance with all public derivation.

```
class baseA {    // base class 1
     -------
     -------
};
class baseB {    // base class 2
     -------
     -------
};
class baseC {    // base class 3
     -------
     -------
};
class derivedD : public baseA,public baseB,public baseC
{
     -------
     -------
};
```

(2) Multiple inheritance with all private derivation.

```
class baseA {    // base class 1
     -------
     -------
};
class baseB {    // base class 2
     -------
     -------
};
class baseC {    // base class 3
     -------
     -------
};
class derivedD : private baseA,private baseB,private baseC
{
     -------
     -------
};
```

(3) Multiple inheritance with all mixed derivation.

```
class baseA {    // base class 1
     -------
     -------
};
class baseB {    // base class 2
     -------
     -------
};
class baseC {    // base class 3
     -------
     -------
};
class derivedD : private baseA,public baseB,private baseC
{
     -------
     -------
};
```

PROGRAM 12.6

A program to illustrate how a multiple inheritance can be declared and defined in a program. This program consists of two base classes and one derived class. The base class basic_info contains the data members: name, roll number and sex. Another base class academic_fit contains the data members: course, semester and rank. The derived class financial_assit contains the data member amount besides the data members of the base classes. The derived class has been declared as public inheritance. The member functions are used to get information of the derived class from the keyboard and display the contents of class objects on the screen.

The OMT of a class notation for multiple inheritance is given in Fig. 12.7.

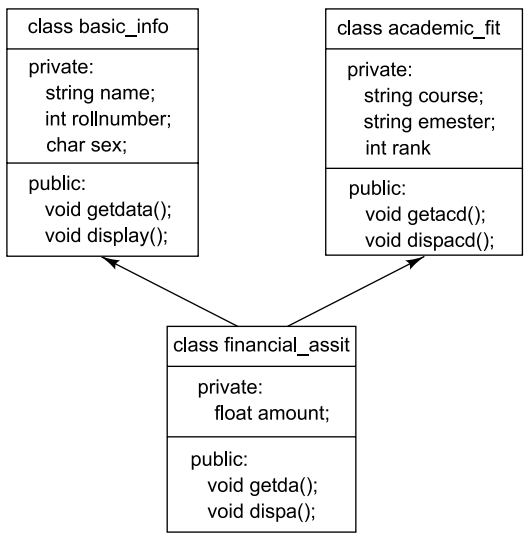


Fig. 12.7 *OMT of a Class Notation for Multiple Inheritance*

```
//multiple inheritance
#include <iostream>
using namespace std;
class  basic_info {
   private :
      char name[20];
      long int rollno;
      char sex;
   public :
      void getdata();
      void display();
 }; // end of class de nition

class academic_ t {
   private :
      char course[20];
      char semester[10];
      int rank;

   public :
      void getdata();
      void display();
}; // end of class de nition

class  nancial_assit : private basic_info, private academic_ t
{
   private :
       oat amount ;
   public :
      void getdata();
      void display();
}; // end of class de nition

void basic_info :: getdata()
{
   cout << " enter a name ? \n";
   cin >> name;
   cout << " roll no ? \n";
   cin >> rollno;
   cout << " sex ? \n";
   cin >> sex;
}
void basic_info :: display()
{
   cout << name << "      ";
   cout << rollno << "      ";
   cout << sex << "   ";
}

void academic_ t :: getdata()
{

   cout << " course name ( BTech/MCA/DCA etc) ?\n";
   cin >> course;
   cout << " semester ( rst/second etc)? \n";
   cin >> semester;
   cout << " rank of the student \n";
   cin >> rank;
}

void academic_ t :: display()
{
   cout << course << "      " ;
   cout << semester << "     ";
   cout << rank << "     ";
}
```

```
void  nancial_assit :: getdata()
{
   basic_info:: getdata();
   academic_ t::getdata();
   cout << " amount in rupees ?\n";
   cin >> amount;
}

void  nancial_assit :: display()
{
   basic_info:: display();
   academic_ t::display();
   cout << amount << "    ";
}

int main()
{
    nancial_assit f;

   cout << "enter the following information for  nancial assistance\n";
   f.getdata();
   cout << endl;
   cout << " Academic Performance for Financial Assistance \n";
   cout << " ———————————————————————— \n";
   cout << "  Name    Rollno    Sex    Course   Semester  Rank    Amount \n";
   cout << " ———————————————————————— \n";
   f.display();
   cout << endl;
   cout << " ———————————————————————— \n";
   return 0;
}
```

Output of the above program

```
enter the following information for  nancial assistance
 enter a name?
 AjitKumar
 roll no?
 20071
 sex?
 M
 course name (BTech/MCA/DCA etc)?
 MCA
 semester ( rst/second etc)?
 First
 rank of the student
 3
 amount in rupees?
 10000

 Academic Performance for Financial Assistance
-------------------------------------------------------------------
 Name          Rollno     Sex     Course    Semester    Rank    Amount
-------------------------------------------------------------------
 AjitKumar     20071       M       MCA       First       3      10000
-------------------------------------------------------------------
```

12.7.1 Array of Class Objects and Multiple Inheritance

This section deals mainly with, how an array of class objects can be inherited from the multiple base classes. Once a derived class has been defined, the way of accessing class members of the array of class objects are same as the ordinary class types.

The general syntax of the array of class objects in a multiple inheritance is:

```
class baseA {
   -------
   -------
};
class baseB {
   -------
   -------
};
class derivedC : public baseA, public baseB  {
   -------
   -------
};
int main()
{
   derivedB obj[100];//array of class objects of the derived class

   -------
   -------
   return 0;
}
```

PROGRAM 12.7

A program consists of two base classes and one derived class. The base class basic_info contains the data members: name, roll number and sex. Another base class academic_fit contains the data members: course, semester and rank. The derived class financial_assit contains the data member amount besides the data members of the base classes. The derived class has been declared as an array of class objects. The member functions are used to get information of the derived class from the keyboard and display the contents of the array of class objects on the screen.

```
//multiple inheritance using array of objects
#include <iostream>
using namespace std;
const int MAX = 100;
class  basic_info {
   private :
      char name[20];
      long int rollno;
      char sex;
   public :
      void getdata();
      void display();
}; // end of class de nition

class academic_ t {
   private :
      char course[20];
      char semester[10];
      int rank;
   public :
      void getdata();
      void display();
}; // end of class de nition

class  nancial_assit : private basic_info, private academic_ t
{
```

```
   private :
        oat amount ;
   public :
      void getdata();
      void display();
}; // end of class de nition

void basic_info :: getdata()
{
   cout << " enter a name ? \n";
   cin >> name;
   cout << " roll no ? \n";
   cin >> rollno;
   cout << " sex ? \n";
   cin >> sex;
}
void basic_info :: display()
{
   cout << name << "      ";
   cout << rollno << "      ";
   cout << sex << "   ";
}

void academic_ t :: getdata()
{
   cout << " course name ( BTech/MCA/DCA etc) ?\n";
   cin >> course;
   cout << " semester ( rst/second etc)? \n";
   cin >> semester;
   cout << " rank of the student \n";
   cin >> rank;
}

void academic_ t :: display()
{
   cout << course << "      " ;
   cout << semester << "     ";
   cout << rank << "     ";
}

void  nancial_assit :: getdata()
{
   basic_info:: getdata();
   academic_ t::getdata();
   cout << " amount in rupees ?\n";
   cin >> amount;
}

void  nancial_assit :: display()
{
   basic_info:: display();
   academic_ t::display();
   cout << amount << "    ";
}

int main()
{
    nancial_assit   f[MAX];
   int n;
   cout << " How many students ?\n";
   cin >> n;
   cout << "enter the following information for  nancial assistance\n";
   for (int i = 0; i <= n-1; ++i) {
       cout << " Record No : " << i+1 << endl;
       f[i].getdata();
       cout << endl;
```

```
    }
    cout << endl;
    cout << " Academic Performance for Financial Assistance \n";
    cout << " ------------------------------- \n";
    cout << "  Name    Rollno   Sex   Course   Semester  Rank   Amount \n";
    cout << " ------------------------------- \n";
    for (int i = 0; i <= n-1; ++i) {
        f[i].display();
        cout << endl;
    }
    cout << " ------------------------------- \n";
    return 0;
}
```

Output of the above program

```
How many students?
3
enter the following information for  nancial assistance
Record No: 1
enter a name?
Fateema
roll no?
20071
sex?
F
course name ( BTech/MCA/DCA etc)?
DCA
semester ( rst/second etc)?
Second
rank of the student
1
amount in rupees?
20000

Record No: 2
enter a name?
ArulRaj
roll no?
20072
sex?
M
course name ( BTech/MCA/DCA etc)?
DCA
semester ( rst/second etc)?
First
rank of the student
3
amount in rupees?
15000

Record No: 3
enter a name?
Sampath
roll no?
20075
sex?
M
course name ( BTech/MCA/DCA etc)?
```

```
DCA
semester ( rst/second etc)?
First
rank of the student
2
amount in rupees?
22000
 Academic Performance for Financial Assistance
-------------------------------------------------------
  Name    Rollno   Sex   Course   Semester Rank   Amount
-------------------------------------------------------
 Fateema  20071     F      DCA      Second   1     20000
 ArulRaj  20072     M      DCA      First    3     15000
 Sampath  20075     M      DCA      First    2     22000
-------------------------------------------------------
```

12.7.2 Ambiguity in the Multiple Inheritance

To avoid ambiguity between the derived class and one of the base classes or between the base class themselves, it is better to use the scope resolution operator :: along with the data members and methods.

For example, the following program segment illustrates how ambiguity occurs in both base classes and the derived class.

```
//ambiguity in multiple inheritance
#include <iostream>
using namespace std;
class baseA {
    public:
       int a;
};
class baseB {
    public:
       int a;
};
class baseC {
    public :
       int a;
};

class derivedD : public baseA,public baseB,public baseC
{
    public :
       int a;
};

int main()
{
    derivedD objd;
    objd.a = 10; //local to the derived class
    return 0;
}
```

Suppose one intends to access the data members of the base classes, then conflict occurs between the base classes themselves and the compiler cannot distinguish between the calls. It is up to the programmer to avoid such conflicts and ambiguities. Therefore, it is better to use the scope operator to avoid such ambiguities.

```
int main()
{
    derivedD objd;
    objd.a = 10;
    objd.baseA::a = 20; //accessing the base class A member
    objd.baseB::a = 30; //accessing the base class B member
    objd.baseC::a = 40; // accessing the base class C member
    return 0;
}
```

PROGRAM 12.8

A program to demonstrate how ambiguity is avoided in multiple inheritance using scope resolution operator.

```
#include <iostream>
using namespace std;
class baseA {
    public:
       int a;
};
class baseB {
    public:
       int a;
};
class baseC {
    public :
       int a;
};

class derivedD : public baseA,public baseB,public baseC
{
    public :
       int a;
};

int main()
{
    derivedD objd;
    objd.a = 10;
    objd.baseA::a = 20;
    objd.baseB::a = 30;
    objd.baseC::a = 40;
    cout << " value of a in the derived class = " << objd.a << endl;
    cout << " value of a in baseA = " << objd.baseA::a << endl;
    cout << " value of a in baseB = " << objd.baseB::a << endl;
    cout << " value of a in baseC = " << objd.baseC::a << endl;
    cout << endl;
    return 0;
}
```

Output of the above program

```
value of a in the derived class = 10
value of a in baseA = 20
value of a in baseB = 30
value of a in baseC = 40
```

12.8 CONTAINER CLASSES

In Chapter 10, how a class could be declared as a member of another class has been explained. When a class is declared and defined as a member of another class, it is known as a nested class. In this section, how a container class can be declared and defined in a program is explained. C++ allows to declare an object of a class as a member of another class. When an object of a class is declared as a member of another class, it is called as a container class. Some of the examples for container classes are arrays, linked lists, stacks and queues.

The general syntax for the declaration of container class is,

```
class user_de ned_name_1 {
   -------
   -------
};
class user_de ned_name_2 {
   -------
   -------
};
class user_de ned_name_n {
   -------
   -------
};
class derived_class :public/protected/private
{
   user_de ned_name_1 obj1; //object of the class one
   user_de ned_name_2 obj2; //object of the class two
   -------
   -------
   user_de ned_name_n objn; //object of the class n
};
```

For example, the following program segment illustrates how to declare the container class.

```
class  basic_info {
    private :
       char name[20];
    public :
       void getdata();
}; // end of class de nition

class academic_ t {
    private :
       int rank;
    public :
       void getdata();
}; // end of class de nition

class  nancial_assit
{
    private :
       basic_info bdata;   //object of class basic_info
       academic_ t acd;    // object of class academic_ t
        oat amount ;
    public :
```

```
        void getdata();
        void display();
};  // end of class de nition

int main()
{
    nancial_assit  objf;
   -------
   -------
   return 0;
}
```

PROGRAM 12.9

A program to demonstrate how a container class is declared and defined in a program. The program consists of two base classes and one derived class. The base class basic_info contains the data members: name, roll number and sex. Another base class academic_fit contains the data members: course, semester and rank. The derived class financial_assit contains the data member amount, besides the data members of the base classes. The objects of these two base classes are defined as members of the derived class. The member functions are used to get information on the derived class from the keyboard and display the contents of the class objects on the screen.

```
//demonstration of container class
#include <iostream>
using namespace std;
class  basic_info {
   private :
      char name[20];
      long int rollno;
      char sex;
   public :
      void getdata();
      void display();
}; // end of class de nition

class academic_ t {
   private :
      char course[20];
      char semester[10];
      int rank;
   public :
      void getdata();
      void display();
}; // end of class de nition

class  nancial_assit
{
   private :
      basic_info bdata;   //object of class basic_info
      academic_ t acd;   // object of class academic_ t
       oat amount ;
   public :
      void getdata();
      void display();
}; // end of class de nition

void basic_info :: getdata()
{
   cout << " enter a name ? \n";
```

```
    cin >> name;
    cout << " roll no ? \n";
    cin >> rollno;
    cout << " sex ? \n";
    cin >> sex;
}
void basic_info :: display()
{
    cout << name << "      ";
    cout << rollno << "      ";
    cout << sex << "  ";
}

void academic_ t :: getdata()
{
    cout << " course name ( BTech/MCA/DCA etc) ?\n";
    cin >> course;
    cout << " semester ( rst/second etc)? \n";
    cin >> semester;
    cout << " rank of the student \n";
    cin >> rank;
}

void academic_ t :: display()
{
    cout << course << "      " ;
    cout << semester << "    ";
    cout << rank << "    ";
}

void  nancial_assit :: getdata()
{
    bdata.getdata();
    acd.getdata();
    cout << " amount in rupees ?\n";
    cin >> amount;
}

void  nancial_assit :: display()
{
    bdata.display();
    acd.display();
    cout << amount << "   ";
}

int main()
{
     nancial_assit f;
    cout << "enter the following information \n";
    f.getdata();
    cout << endl;
    cout << " Academic Performance for Financial Assistance \n";
    cout << " ------------------------------------------------------ \n";
    cout << "  Name    Rollno   Sex   Course   Semester  Rank   Amount \n";
    cout << " ------------------------------------------------------ \n";
    f.display();
    cout << endl;
    cout << " ------------------------------------------------------ \n";
    return 0;
}
```

Output of the above program

```
enter the following information
enter a name?
Suseekaran
```

```
roll no?
20071
sex?
M
course name ( BTech/MCA/DCA etc)?
B.Tech(CSE)
semester ( rst/second etc)?
First
rank of the student
1
amount in rupees?
30000
 Academic Performance for Financial Assistance
-----------------------------------------------------------------------
  Name          Rollno    Sex     Course      Semester       Rank    Amount
-----------------------------------------------------------------------
 Suseekaran     20071      M    B.Tech(CSE)     First          1      30000
-----------------------------------------------------------------------
```

12.9 MEMBER ACCESS CONTROL

It is well known that the C++ class is more than an enhanced version of the C structure. The class sets up an environment in which the software designer not only can create objects that contain their own methods in the form of member functions, but also can exercise almost total control over the access of class implementations. In this section, access control and its mechanism to access the individual members of a class as well as the derived class are explained. It has already been stated that the access mechanism of the individual members of a class is based on the use of the keywords public, private and protected.

12.9.1 Accessing the Public Data

The public members of a class can be accessed by the following:

- Member functions of the class
- Nonmember functions of the class
- Member function of a friend class
- Member function of a derived class if it has been derived publicly.

For example, the following are valid declarations for accessing the public data member of a class:

(1) The member function of the class can access the public data.

```
class sample {
   public :
      int value;
      void getdata();
      void display()
      {
        ++ value;  // valid
      }
};
```

(2) The nonmember function of the class can access the public data.

```
class sample {
   public :
      int a;
};
```

```
int main()
{
   sample obj;
   obj.a++; // valid
}
```

(3) The member function of the derived class can access the public data of the base class if it has been derived publicly.

```
class base {
   public :
      int value;
      void getdata()
};
class derived : public base {
   public :
      void display()
      {
          ++value;   // valid
      }
};
```

The following declaration for accessing the public data member is invalid as it has been declared as private inheritance.

(1)

```
class base {
   public :
      int value;
      void getdata()
};
class derived : private base {
   public :
      void display()
      {
          ++value;   // invalid
      }
};
```

Note that the member function of the derived class cannot access the public data of the base class if it has been derived privately.

PROGRAM 12.10

A program to illustrate how a data member of a class is accessed by the member function.

```
#include <iostream>
using namespace std;
class sample {
    public:
       int a;
       void setdata();
       void display();
};
void sample :: setdata()
{
    a = 10;
}
void sample :: display()
```

```
{
    cout << " ++a = " << ++a << endl;
}
int main()
{
    sample obj;
    obj.setdata();
    obj.display();
    return 0;
}
```

Output of the above program

```
++a = 11
```

PROGRAM 12.11

A program to show how a public data member is accessed by a nonmember function of the class.

```
#include <iostream>
using namespace std;
class sample {
    public:
       int a;
       void setdata();
};
void sample :: setdata()
{
    a = 10;
}
int main()
{
    sample obj;
    obj.setdata();
    cout << "++a = " << ++(obj.a) << endl;
    return 0;
}
```

Output of the above program

```
++a = 11
```

PROGRAM 12.12

A program to demonstrate how a member function of a derived class can access the public data of a base class in which a derived class has been derived publicly.

```
//accessing public data by derived class
#include <iostream>
using namespace std;
class base {
    public :
       int value;
       inline void getdata()
       {
           cout << " enter a number " << endl;
           cin >> value;
       }
}; // end of class de nition
class derived : public base {
```

```
    public :
       void display()
       {
           ++value;
       }
};// end of class de nition

int main()
{
    derived  obj;
    obj.getdata();
    obj.display();
    cout << " value in derived class = " << obj.value;
    return 0;
}
```

Output of the above program

```
enter a number
10
value in derived class = 11
```

The following program will not be compiled as the member function of the derived class tries to access the public data of the base class, where the derived class has been inherited privately.

```
//accessing public data by derived class
//private derivation
#include <iostream>
using namespace std;
class base {
    public :
        int value;
        inline void getdata()
        {
           cout << " enter a number " << endl;
           cin >> value;
        }
 }; // end of class de nition
class derived : private base {
    public :
        void display()
        {
           ++value;
        }
 };// end of class de nition

int main()
{
    derived  obj;
    obj.getdata();  //  base::getdata() is not accessible
    obj.display();  //  base::value is not accessible
    cout << " value in derived class = " << obj.value;
    return 0;
}
```

12.9.2 Accessing the Private Data

The private members of a class can be accessed only by the following:

- The member function of the class, and
- The member functions of the friend class in which it is declared.

In other words, the public member function of a derived class cannot access the private data member of the base class irrespective of whether the derived class has been inherited publicly or privately.

For example, in the following program the declaration for accessing the private data members of base class by the public members of the derived class is invalid:

(1) A derived class with private derivation.

```
class baseA {
    private :
        int value;
};
class derivedB : private baseA {
    public :
      void f()
      {
         ++ value;  // error, baseA:: value is not accessible
      }
};
```

(2) Even if the derived class has been derived publicly from the base class, the public member of a derived class cannot access the private member of the base class.

```
class baseA {
    private :
        int value;
};
class derivedB : public baseA {
    public :
       void f()
       {
          ++ value;  // error, baseA:: value is not accessible
       }
};
```

Note that a derived class public member function cannot access the private member of a base class irrespective of whether the derived class has been derived publicly or privately.

The following program shows an error message indicating that baseA::value is not accessible.

```
//accessing public data by derived class
//private derivation
#include <iostream>
using namespace std;
class base {
   public :
      int value;
      inline void getdata()
      {
           cout << " enter a number " << endl;
           cin >> value;
      }
}; // end of class de nition
class derived : private base {
   public :
      void display()
      {
         ++value;
      }
```

```
};// end of class de nition

int main()
{
    derived  obj;
    obj.getdata();
    obj.display();
    cout << " value in derived class = " << obj.value;
    return 0;
}
```

Compile time error

```
void base :: getdata() is inaccessible due to private inheritance
int base :: value is inaccessible due to private inheritance.
```

PROGRAM 12.13

A program to demonstrate how the private data member cannot be accessed by the public member function of the derived class, even though the derived class has been inherited publicly.

```
//demonstration of private member
#include <iostream>
using namespace std;
class baseA{
    private :
       int value;
    public :
       baseA ()
       {
          cout << " enter a number : ";
          cin >> value;
    }
}; // end of base declaration
class derivedB: public baseA {
    public :
       void display ()
       {
          ++value;
          cout << " value in derivedB  = " << value << endl;
       }
}; // end of derived class de nition
int main()
{
    derivedB objB;
    objB.display();
    return 0;
}
```

Compile time error

```
void baseA :: value is a private category and inaccessible.
```

12.9.3 Accessing the Protected Data

The protected data members of a class can be accessed by the following:

- The member function,
- The friends of the class in which it is declared, and
- The member functions of the derived class irrespective of whether the derived class has been derived privately or publicly.

For example, the following program segments illustrate how a protected data member of a base is accessed by the public member function of the derived class:

(1) The protected data member of a class can be accessed by the public member function even if it has been derived privately, provided the derived class have a direct base.

```
class baseA {
   protected :
      int value;
};
class derivedB : private baseA {
   public :
      void f()
      {
           ++value;  // valid
      }
};
```

(2) The protected data member of a class cannot be accessed by the public member function of the derivedD because the derivedD has not been derived from the direct base of baseA.

```
class baseA {
   protected :
      int value;
};
class derivedB : private baseA {
      -------
      -------
};
class derivedC : private derivedB {
      -------
      -------
};
class derivedD : private derivedC {
   public :
      void f()
      {
          ++value;  // error, baseA::value is not accessible
      }
};
```

(3) The protected data of the base class can access the public member function of a derived class via public derivation.

```
class baseA {
   protected :
      int value;
};
class derivedB : public baseA {
   public :
      void f()
      {
           ++value;  // valid
      }
};
```

(4) The protected data member of the base class can be accessed by the public member functions of the derived class, even if it has a direct base or indirect base, provided it has been derived via public derivation.

```
class baseA {
   protected:
      int value;
};
class derivedB: public baseA {
   -------
   -------
};
class derivedC: public derivedB {
   -------
   -------
};
class derivedD: public derivedC {
   public :
      void f()
      {
         ++value;  // valid
      }
};
```

PROGRAM 12.14

A program to demonstrate how the protected data member of a base class is accessed by the public member function of the derived class if it has been derived privately.

```
//demonstration of accessing a protected member
#include <iostream>
using namespace std;
class baseA{
    protected:
       int value;
    public:
       baseA ()
       {
         cout << " enter a number : ";
         cin >> value;
       }
}; // end of base declaration

class derivedB: private baseA {
    public:
       void display ()
       {
         ++value;
         cout << " content of value  = " << value << endl;
       }
}; // end of derived class de nition

int main()
{
    derivedB obj1;
    obj1.display();
    return 0;
}
```

Output of the above program

```
enter a number :  100
content of value  = 101
```

PROGRAM 12.15

A program to demonstrate how the protected data member of a base class is accessed by the public member function of the derived class irrespective of whether the derived class has a direct base or indirect base and has been derived publicly.

```
//demonstration of accessing a protected data
#include <iostream>
using namespace std;
class baseA{
    protected :
       int value;

    public :
       baseA ()
       {
          cout << " enter a number : ";
          cin >> value;
       }
}; // end of base declaration

class derivedB: public baseA {
    public :
       void display ()
       {
          ++value;
          cout << " value in derivedB  = " << value << endl;
       }
}; // end of derived class de nition

class derivedC: public derivedB {
    public :
       void display ()
       {
          ++value;
          cout << " value in derivedC  = " << value << endl;
       }
}; // end of derived class de nition

class derivedD: public derivedC {
    public :
       void display ()
       {
          ++value;
          cout << " value in derivedD  = " << value << endl;
       }
}; // end of derived class de nition

int main()
{
    derivedD objD;
    objD.display();
    return 0;
}
```

Output of the above program

```
enter a number: 200
value in derivedD  = 201
```

The following program shows an error message and will not be compiled. The protected data of a base class cannot access the member function of the derived class due to indirect base via private derivation.

```
//demonstration of public access
#include <iostream>
using namespace std;
class baseA{
   protected:
      int value;
   public:
      baseA ()
      {
          cout << " enter a number: ";
          cin >> value;
       }
}; // end of base declaration

class derivedB: private baseA {
   public :
      void display ()
      {
          ++value;
          cout << " value in derivedB  = " << value << endl;
      }
}; // end of derived class de nition

class derivedC: private derivedB {
   public:
      void display ()   // error, baseA::value is not accessible
      {
          ++value;
          cout << " value in derivedC  = " << value << endl;
      }
}; // end of derived class de nition

int main()
{
   derivedC objC;
   objC.display();
   return 0;
}
```

Compile time error

```
int derivedB :: value is a protected
int derivedC :: value is a protected
```

In private inheritance, the protected data cannot be accessed by the public member functions of the derived class and hence, it gives compile time error.

12.9.4 Accessing Private Member by Friend Class

A public member function of a friend class can access the private member of the base class irrespective of whether a friend function has been derived privately or publicly.

(1) A friend class derived privately.

```
class baseA {
    friend class derivedB;
        private:
           int value ;
```

```
};
class derivedB: private baseA {
    public:
        void f()
        {
           ++value; // valid
        }
};
```

(2) A friend class derived publicly.

```
class baseA {
    friend class derivedB;
        private:
           int value ;
};
class derivedB: public baseA {
    public:
       void f()
       {
          ++value; // valid
       }
};
```

(3) A friend class derived publicly.

```
class baseA {
   friend class derivedB;
        private:
           int value ;
};
class derivedB: private baseA {
    public:
       void g()
       {
          ++value; // valid
       }
};

class derivedC: public baseA {
     public:
       void f()
       {
         ++value; // error, baseA:: value is not accessible
       }
};
```

No other public member function can access the private member of the base class other than the friend class public member function. Note that the derived class derivedC is not a friend of baseA and hence, it cannot access the private data of the base class, even though it has been derived publicly.

(4) A friend class derived publicly.

```
class baseA {
    friend class derivedB;
```

```
    private:
        int value ;
};
class derivedB: private baseA {
    -------
    -------
};
class derivedC: private derivedB {
    public:
        void f()
        {
            ++value; // error, baseA:: value is not accessible
        }
};
```

Note that the derived class `derivedC` cannot access the private data of the base class baseA even though it has been derived indirectly via derivedB. It is because, baseA extends its friendship only with derivedB but not with other classes.

(5) No other public member function can access the private member of the base class other than the friend class public member function, even though the friend class has been inherited publicly.

```
class baseA{
    friend class derivedB;
         private:
            int value;
};
class derivedB: public baseA {
    -------
    -------
};
class derivedC: public derivedB {
    public:
        void g()
        {
            ++value;  //error,baseA:: value is not accessible
        }
};
```

PROGRAM 12.16

A program to demonstrate how a private data of a base class is accessed by the public member function of the derived class through friend class declaration. The derived class has been inherited via public inheritance.

```
//demonstration of friend class
// public inheritance
#include <iostream>
using namespace std;
class baseA{
    friend class derivedB;
    private:
        int value;
    public:
        baseA ()
        {
            cout << " enter a number: ";
            cin >> value;
```

```
        }
}; // end of base declaration

class derivedB: public baseA {
    public:
        void display ()
        {
            ++value;
            cout << " value in derivedB  = " << value << endl;
        }
}; // end of derived class de nition

int main()
{
    derivedB objB;
    objB.display();
    return 0;
}
```

Output of the above program

```
enter a number: 10
value in derivedB  = 11
```

PROGRAM 12.17

A program to demonstrate how a private data of a base class is accessed by the public member function of the derived class through friend class declaration. The derived class has been inherited via private inheritance.

```
//demonstration of friend class
//private inheritance
#include <iostream>
using namespace std;
class baseA{
    friend class derivedB;
    private:
        int value;
    public:
        baseA ()
        {
            cout << " enter a number: ";
            cin >> value;
        }
}; // end of base declaration

class derivedB: private baseA {
    public:
        void display ()
        {
            ++value;
            cout << " value in derivedB  = " << value << endl;
        }
}; // end of derived class de nition

int main()
{
    derivedB objB;
    objB.display();
    return 0;
}
```

Output of the above program

```
enter a number: 5
value in derivedB  = 6
```

PROGRAM 12.18

The following program shows an error message due to accessing a private data of the base class by the public member function of the derived class via private inheritance. Because of the private derivation, the base class members cannot be accessed by the public members of the derived class.

```
//error baseA:: value is not accessible
//demonstration of friend class
#include <iostream>
using namespace std;
class baseA{
    friend class derivedB;
    private:
       int value;
    public:
       baseA ()
       {
           cout << " enter a number: ";
           cin >> value;
       }
}; // end of base declaration

class derivedB: private baseA {
    public:
       void display ()
       {
           ++value;
           cout << " value in derivedB  = " << value << endl;
       }
}; // end of derived class de nition

class derivedC: private derivedB {
    public:
       void display ()
       {
           ++value;
           cout << " value in derivedC  = " << value << endl;
       }
}; // end of derived class de nition

int main()
{
    derivedC objC;
    objC.display();
    return 0;
}
```

Compile time error

`int baseA :: value` is a private data member and it cannot be accessed by the public member functions of a derived class and hence it gives error.

PROGRAM 12.19

The following program tries to access the private data of the base class by the public member function of the derived class via public inheritance through friendship.

```
//demonstration of friend class
#include <iostream>
using namespace std;
class baseA{
    friend class derivedB;
    private:
       int value;
    public:
       baseA ()
       {
           cout << " enter a number: ";
           cin >> value;
       }
}; // end of base declaration

class derivedB: public baseA {
    public:
       void display ()
       {
           ++value;
           cout << " value in derivedB  = " << value << endl;
       }
}; // end of derived class de nition

class derivedC: public derivedB {
    public:
       void display ()
       {
           ++value;
           cout << " value in derivedC  = " << value << endl;
       }
}; // end of derived class de nition

int main()
{
    derivedC objC;
    objC.display();
    return 0;
}
```

Compile time error

`int baseA :: value` is a private data member and it cannot be accessed by the public member functions of a derived class and hence it gives error.

PROGRAM 12.20

A program to demonstrate how a protected data of a base class is accessed by the public member function of the derived class through friend class declaration. The derived class has been inherited via public inheritance.

```
//demonstration of friend class
#include <iostream>
using namespace std;
class baseA{
    friend class derivedB;
    protected:
       int value;
    public:
       baseA ()
       {
```

```
            cout << " enter a number: ";
            cin >> value;
        }
}; // end of base declaration

class derivedB: public baseA {
    public:
       void display ()
       {
           ++value;
           cout << " value in derivedB  = " << value << endl;
       }
}; // end of derived class de nition

class derivedC: public derivedB {
    public:
       void display ()
       {
           ++value;
           cout << " value in derivedC  = " << value << endl;
       }
}; // end of derived class de nition

int main()
{
    derivedC objC;
    objC.display();
    return 0;
}
```

Output of the above program

```
enter a number: 10
value in derivedC = 11
```

PROGRAM 12.21

The following program shows that an error is encountered as a result of an attempt to access the protected data of the base class by the public member function of the derived class via private inheritance. Because of the private derivation, the base class members cannot be accessed by the public members of the derived class even though it has a friendship with base class.

```
//demonstration of friend class
#include <iostream>
using namespace std;
class baseA{
    friend class derivedB;
    protected:
       int value;
    public:
       baseA ()
       {
           cout << " enter a number : ";
           cin >> value;
       }
}; // end of base declaration

class derivedB: private baseA {
    public:
       void display ()
       {
           ++value;
```

```
            cout << " value in derivedB  = " << value << endl;
        }
}; // end of derived class de nition

class derivedC: private derivedB {
    public:
       void display ()
       {
            ++value;
            cout << " value in derivedC  = " << value << endl;
       }
}; // end of derived class de nition

int main()
{
    derivedC objC;
    objC.display();
    return 0;
}
```

Compile time error

`int baseA:: value` is protected data member and it cannot be accessed by the public member functions of the derived class. Hence it gives error.

12.10 SUMMARY OF THE INHERITANCE ACCESS SPECIFIER

Access specifier	Accessible from own class	Accessible from derived calss	Accessible from objects outside class
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

REVIEW QUESTIONS

1. What is meant by "inheritance" in the OOP paradigm?
2. What are the advantages and disadvantages of declaring inheritance?
3. What is the difference between the base and derived classes?
4. How is a direct base class different from the indirect base class declaration in C++?
5. Explain how a data member and member functions of a base class can be accessed by the derived class member functions.
6. What are the rules governing the declaration of a class of single inheritance?
7. Define multiple inheritance.
8. List the merits and demerits of single inheritance over multiple inheritance.
9. What are the rules to be followed to declare a multiple inheritance class data type?
10. In what way a container class is different from a nested class object?
11. Explain how a private data of a base class can be accessed by a publicly derived class.
12. Explain the merits and demerits of private derivation over the public derivation.
13. What is a friend class? What are the access control a friend class has over the member function?
14. What is a public derivation? Explain how a private member of a base class can be accessed by the public member function of the derived class through public inheritance?

15. What is a private inheritance? What are the merits and demerits of private inheritance.
16. What is a protected inheritance? Explain how a protected data of the base class can be accessed by the public member function of the derived class.
17. What is a container class? List the pros and cons of declaring container classes.
18. How does an array of class objects declared with single inheritance?
19. How does an array of class objects declared with multiple inheritance?
20. What are the syntactic rules to be followed to avoid ambiguity in single inheritance?
21. What are the syntactic rules to be followed to avoid ambiguity in multiple inheritance?
22. Explain the relationship among the terms superclass, subclass, base class and derived class.
23. Explain a class hierarchy in which a base class has multiple base classes.
24. Explain a class hierarchy in which a derived class has multiple base classes.
25. Explain the following with syntactic rules:
(i) Public inheritance
(ii) Protected inheritance
(iii) Private inheritance

CONCEPT REVIEW PROBLEMS

1. Determine the output of each of the following programs when it is executed.
(a)

```
#include <iostream>
using namespace std;
class abc {
    public:
        void display();
};
class derivedB: public abc
{
    public:
        void display();
};

void abc :: display()
{
    cout << "members of abc \n";
}
void derivedB :: display()
{
    abc::display();
    cout << "members of derivedB \n";
}
int main()
{
    abc *obj;
    obj->display();
    return 0;
}
```

(b)

```
#include <iostream>
```

```
using namespace std;
class abc {
    public:
       void display();
};
class derivedB: public abc
{
    public:
       void display();
};
void abc :: display()
{
    cout << "members of abc \n";
}
void derivedB :: display()
{
    cout << "members of derivedB \n";
}
int main()
{
    abc *obj;
    obj->display();
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
class abc {
    public:
       void display();
};
class derivedB: public abc
{
    public:
       void display();
};
class derivedC: public derivedB
{
    public:
       void display();
};
void abc :: display()
{
    cout << "members of abc \n";
}
void derivedB :: display()
{
    cout << "members of derivedB \n";
}
void derivedC :: display()
{
    abc::display();
    cout << "members of derivedC \n";
}
int main()
```

```
{
    abc *obj;
    obj->display();
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
class abc {
    public:
        void display();
};
class derivedB: public abc
{
    public:
        void display();
};
void abc :: display()
{
    cout << "members of abc \n";
}
void derivedB :: display()
{
    abc::display();
    cout << "members of derivedB \n";
}
int main()
{
    derivedB *obj;
    obj->display();
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
class abc {
    public:
        void display();
};
class derivedB: public abc
{
    public:
        void display();
};

class derivedC: public derivedB
{
    public:
        void display();
};
class derivedD: public derivedC
{
    public:
        void display();
};
```

```
void abc :: display()
{
    cout << "members of abc \n";
}
void derivedB :: display()
{
    abc::display();
    cout << "members of derivedB \n";
}
void derivedC :: display()
{
    derivedB::display();
    cout << "members of derivedC \n";
}
void derivedD :: display()
{
    derivedC::display();
    cout << "members of derivedD \n";
}

int main()
{
    derivedD *obj;
    obj->display();
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
class abc {
    public:
       void dispabc();
};
class xyz {
    public:
       void dispxyz();
};

class derivedC: abc,xyz
{
    public:
       void display();
};
class derivedD: derivedC
{
    public:
       void display();
};
void abc :: dispabc()
{
    cout << "members of abc \n";
}
void xyz :: dispxyz()
{
    cout << "members of xyz \n";
```

```
}
void derivedC :: display()
{
    abc::dispabc();
    xyz::dispxyz();
    cout << "members of derivedC \n";
}
void derivedD :: display()
{
    derivedC::display();
    cout << "members of derivedD \n";
}

int main()
{
    derivedD *obj;
    obj->display();
    return 0;
}
```

2. What will be the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
class abc {};
class xyz: public abc
{
    public:
       int i;
       void display(int a);
};
void xyz :: display(int a)
{
    cout << "a = " << a << "\n";
}

int main()
{
    xyz obj;
    obj.display(10);
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
class abc;
class xyz: public abc
{
    public:
       int i;
       void display(int a);
};
void xyz :: display(int a)
{
    cout << "a = " << a << "\n";
```

```
}

int main()
{
    xyz obj;
    obj.display(10);
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
class abc { };
class xyz { };
class  nalC: public abc,public xyz
{
    public:
       int i;
       void display(int a);
};
void  nalC :: display(int a)
{
    cout << "a = " << a << "\n";
}

int main()
{
     nalC obj;
    obj.display(10);
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
struct abc { };
union xyz { };
class  nalC: public abc,public xyz
{
    public:
       int i;
       void display(int a);
};
void  nalC :: display(int a)
{
    cout << "a = " << a << "\n";
}

int main()
{
     nalC obj;
    obj.display(10);
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
```

```
struct abc {
    public:
       int a;
};
class  nalC: public abc
{
    public:
       void display();
};
void  nalC :: display()
{
    a = 10;
    cout << "a = " << a << "\n";
}

int main()
{
     nalC obj;
    obj.display();
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
struct abc {
    protected:
       int a;
};
class  nalC: public abc
{
    public:
       void display();
};
void  nalC :: display()
{
    a = 10;
    cout << "a = " << a << "\n";
}

int main()
{
     nalC obj;
    obj.display();
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
union abc {
    int a;
};
class  nalC: public abc
{
    public:
       void display();
```

```
};
void  nalC :: display()
{
    a = 10;
    cout << "a = " << a << "\n";
}

int main()
{
     nalC obj;
    obj.display();
    return 0;
}
```

(h)

```
#include <iostream>
using namespace std;
struct abc {
    int a;
};
union  nalC: public abc
{
    public:
       void display();
};
void  nalC :: display()
{
    a = 10;
    cout << "a = " << a << "\n";
}

int main()
{
     nalC obj;
    obj.display();
    return 0;
}
```

(i)

```
#include <iostream>
using namespace std;
struct abc {
    int a;
};
struct  nalC: abc
{
    public:
       void display();
};
void  nalC :: display()
{
    a = 10;
    cout << "a = " << a << "\n";
}

int main()
{
```

```
     nalC obj;
    obj.display();
    return 0;
}
```

(j)

```
#include <iostream>
using namespace std;
class abc {
    int a;
};
struct  nalC: abc
{
    public:
       void display();
};
void  nalC :: display()
{
    a = 10;
    cout << "a = " << a << "\n";
}

int main()
{
     nalC obj;
    obj.display();
    return 0;
}
```

3. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
class A {
    public:
       int a;
       void displayA(int a);
};

class B: public A
{
    public:
       int b;
       void displayB(int b);
};

class C: public B
{
    public:
       int c;
       void displayC(int c);
};

void A :: displayA(int a)
{
    ++a;
    cout << " value of a = " << a << endl;
```

```
}

void B :: displayB(int b)
{
    A::displayA(10);
    ++b;
    cout << " value of b = " << b << endl;
}

void C :: displayC(int c)
{
    B::displayB(20);
    ++c;
    cout << " value of c = " << c << endl;
}
int main()
{
    C cobj;
    cobj.displayC(30);
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
class A {
    public:
       int a;
       void displayA(int a);
};

class B:  private A
{
    public:
       int b;
       void displayB(int b);
};

class C: private B
{
    public:
       int c;
       void displayC(int c);
};

void A :: displayA(int a)
{
    ++a;
    cout << " value of a = " << a << endl;
}

void B :: displayB(int b)
{
    A::displayA(10);
    ++b;
    cout << " value of b = " << b << endl;
```

```
}

void C :: displayC(int c)
{
    B::displayB(20);
    ++c;
    cout << " value of c = " << c << endl;
}
int main()
{
    C cobj;
    cobj.displayC(30);
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
class A {
    public:
       int a;
       void displayA(int a);
};

class B: A
{
    public:
       int b;
       void displayB(int b);
};

class C: B
{
    public:
       int c;
       void displayC(int c);
};

void A :: displayA(int a)
{
    ++a;
    cout << " value of a = " << a << endl;
}

void B :: displayB(int b)
{
    A::displayA(10);
    ++b;
    cout << " value of b = " << b << endl;
}

void C :: displayC(int c)
{
    B::displayB(20);
    ++c;
    cout << " value of c = " << c << endl;
```

```
}
int main()
{
    C cobj;
    cobj.displayC(30);
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
class A {
    public:
       int a;
       void displayA(int a);
};

class B: protected A
{
    public:
       int b;
       void displayB(int b);
};

class C: protected B
{
    public:
       int c;
       void displayC(int c);
};

void A :: displayA(int a)
{
    ++a;
    cout << " value of a = " << a << endl;
}

void B :: displayB(int b)
{
    A::displayA(10);
    ++b;
    cout << " value of b = " << b << endl;
}

void C :: displayC(int c)
{
    B::displayB(20);
    ++c;
    cout << " value of c = " << c << endl;
}
int main()
{
    C cobj;
    cobj.displayC(30);
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
class A {
    private:
       int a;
    public:
       void displayA(int a);
};

class B : public A
{
    private:
       int b;
    public:
       void displayB(int b);
};

void A :: displayA(int a)
{
    ++a;
    cout << " value of a = " << a << endl;
}

void B :: displayB(int b)
{
    A::displayA(100);
    ++b;
    cout << " value of b = " << b << endl;
}
int main()
{
    B bobj;
    bobj.displayB(300);
    return 0;
}
```

PROGRAMMING EXERCISES

1.(a) Develop an object-oriented program in C++ to read the following information from the keyboard in which the base class consists of: employee name, code and designation. The derived class contains the data members, viz. years of experience and age.

Employee name
Employee code
Designation
Years of experience
Age

(b). Construct an object-oriented data base to carry out the following methods:

(i) Build a master table
(ii) List a table

(iii) Insert a new entry
(iv) Delete old entry
(v) Edit entry
(vi) Search for a record that is to be printed
(vii) Sort entries

2. Develop an object-oriented program in C++ to create a data base of the following items of the derived class.

Name of the patient
Sex
Age
Ward number
Bed number
Nature of the illness
Date of admission

Design a base class consisting of the data members, viz. name of the patient, sex and age and another base class consisting of ward number, bed number and nature of the illness. The derived class consists of the data member viz. date of admission.

Your program should have the facilites as mentioned in 1(b).

3. Develop an object-oriented program in C++ to create a pay roll system of an organisation with the following information read from the keyboard:

Employee name
Employee code
Designation
Account number
Date of joining
Basic pay
DA, HRA and CCA
Deductions like PPF, GPF, CPF, LIC, NSS, NSC, etc.

Design a base class consisting of employee name, employee code and designation and another base class consisting of the data member, such as account number and date of joining. The derived class consists of the data member of basic pay plus other earnings and deductions.

Your program should have the facilities as enumerated in 1(b).

4. Develop an object-oriented program in C++ to prepare the mark sheet of an university examination with the following items read from the keyboard:

Name of the student
Roll number
Subject name
Subject code
Internal marks
External marks

Design a base class consisting of the data members such as name of the student, roll number and subject name. The derived class consists of the data members, viz. subject code, internal marks and external marks.

Your program should have the facilities as listed in 1(b).

5. Develop an object-oriented program in C++ to create a library information system containing the following for all books in the library:

Accession number
Name of the author
Title of the book
Year of publication
Publisher's name
Cost of the book

Design a base class with the data members, accession number, name of the author and title of the book, and another base class consisting of year of publication and publisher's name. The derived class consists of the data member viz., cost of the book.

Your program should have the facilities as listed in 1(b).

6. Develop an object-oriented program in C++ to create a data base of the personnel information system containing the following information:

Name
Date of birth
Blood group
Height
Weight
Insurance policy number
Contact address
Telephone number
Driving licence number, etc.

Design a base class with name, date of birth, blood group, and another base class consisting of the data members such as height and weight. Design one more base class consisting of the insurance policy number and contact address. The derived class contains the data members viz. telephone number and driving licence number.

Your program should have the facilities as mentioned in 1(b).

Overloading Functions and Operators

Chapter 13

This chapter presents how the concepts of overloading of an object-oriented programming (OOP) can be incorporated in C++. In addition to data hiding, data encapsulation and inheritance, 'overloading' is a noteworthy feature of object oriented programming paradigm. Focus is on how overloading of functions and operators can be declared, defined and called in a user-defined program.

13.1 FUNCTION OVERLOADING

Other than the concept of classes and objects, overloading of functions and operators are the most noteworthy features of C++. The C++ language allows the user to create new abstract data types, which is one of the major advantages of the OOP.

Function overloading is a logical method of calling several functions with different arguments and data types that perform basically identical things by the same name. The main advantages of using function overloading are:

- eliminating the use of different function names for the same operation.
- helps to understand, debug and grasp easily.
- easy maintainability of the code.
- better understanding of the relation between the program and the outside world.

Function overloading is an easy concept in C++ and generally it is used within the class concept for member functions and constructors. The compiler classifies the overloaded function by its name and the number and type of arguments in the function declaration. The function declaration and definition is essential for each function with the same function name but with different arguments and data types.

For example, consider the following `swap( )` function which is used to swap the different data types such as `int`, `oat` and `char`. In order to carry out the swapping of these data types, it is required to write

three distinct functions with different names, i.e., one function for swapping integer values, one for floating point numbers and another one for character data types. The function declaration, definition and calling of each function is also different. Each function swap ( ) must have a different definition such that the compiler can choose the correct function.

(1)

A function for swapping two integers is,

```
void swap_int (int &a,int &b)
{
     int temp;
     temp = a;
     a = b;
     b = temp;
}
```

(2)

A function for swapping two floating point numbers is,

```
void swap_ oat ( oat &a, oat &b)
{
      oat temp;
     temp = a;
     a = b;
     b = temp;
}
```

(3)

A function for swapping two character data types is,

```
void swap_char (char &a,char &b)
{
     char temp;
     temp = a;
     a = b;
     b = temp;
}
```

The following program segment shows how various swapping functions can be declared and called with parameters.

```
#include <iostream>
using namespace std;
int main()
{
    void swap_int (int &ix , int &iy);
    void swap_ oat ( oat &fx, oat &fy);
    void swap_char (char &cx,char &cy); // function declaration
    -------
    -------
    swap_int(x,y);
    swap_ oat(fx,fy);
    swap_char(cx,cy);
    return 0;
}
```

PROGRAM 13.1

A program to perform the swapping of two data items of integer, floating point numbers and character types without function overloading.

```
// functions without overloading
//swapping two items
#include <iostream>
#include <string>
using namespace std;
int main()
{
    void swap_int (int &ix , int &iy);
    void swap_ oat ( oat &fx, oat &fy);
    void swap_char (char &cx,char &cy); // function declaration
    void swap_string (string &str1, string &str2);
    void menu();
    int ix,iy;
     oat fx,fy;
    char cx,cy,ch;
    string str1,str2;
    menu();
    while (( ch = cin.get()) != 'q') {
        switch(ch) {
           case 'i':
                // swapping on integers
                cout << " enter any two integers \n";
                cin >> ix >> iy ;
                cout << " swapping of integers \n";
                cout << " ix = " << ix << " iy = " << iy << endl;
                swap_int(ix,iy);
                cout << " after swapping \n";
                cout << " ix = " << ix << " iy = " << iy << endl;
                break;
           case 'f':
                //  oating point numbers
                cout << " enter any two  oating point numbers\n";
                cin >> fx >> fy;
                cout << " swapping of  oating point numbers \n";
                cout << " fx = " << fx << " fy = " << fy << endl;
                swap_ oat(fx,fy);
                cout << " after swapping \n";
                cout << " fx = " << fx << " fy = " << fy << endl;
                break;
           case 'c':
                //swapping characters
                cout << " enter any two characters\n";
                cin >> cx >> cy;
                cout << " swapping of characters \n";
                cout << " cx = " << cx << " cy = " << cy << endl;
                swap_char(cx,cy);
                cout << " after swapping \n";

                cout << " cx = " << cx << " cy = " << cy << endl;
                break;
           case 's':
                //swapping strings
                cout << " enter any two strings\n";
                cin >> str1 >> str2;
                cout << " swapping of characters \n";
                cout << " str1 = " << str1 << " str2 = " << str2;
                cout << endl;
                swap_string(str1,str2);
                cout << " after swapping \n";
```

```
                cout << " str1 = " << str1 << " str2 = " << str2;
                cout << endl;
                break;
           case 'm':
                menu();
                break;
        }
    } //end of while statement
    return 0;
}

void menu()
{
    cout << "Swapping two data without function overloading\n";
    cout << " i -> integer swapping \n";
    cout << " f -> real number swapping \n";
    cout << " c -> character swapping \n";
    cout << " s -> string swapping \n";
    cout << " m -> menu \n";
    cout << " q -> quit \n";
}

void swap_int (int &a,int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void swap_ oat ( oat &a, oat &b)
{
     oat temp;
    temp = a;
    a = b;
    b = temp;
}


void swap_char (char &a,char &b)
{
    char temp;
    temp = a;
    a = b;
    b = temp;
}

void swap_string (string &a,string &b)
{
    string temp;
    temp = a;
    a = b;
    b = temp;
}
```

Output of the above program

```
Swapping two data without function overloading
  i -> integer swapping
  f -> real number swapping
  c -> character swapping
  s -> string swapping
  m -> menu
  q -> quit

  i
```

```
enter any two integers
10   20
swapping of integers
ix = 10 iy = 20
after swapping
ix = 20 iy = 10

f
enter any two  oating point numbers
1.1   -2.2
swapping of  oating point numbers
fx = 1.1 fy = -2.2
after swapping
fx = -2.2 fy = 1.1

c
enter any two characters
a   b
swapping of characters
cx = a cy = b
after swapping
cx = b cy = a

s
enter any two strings
Windows    Linux
swapping of characters
str1 = Windows str2 = Linux
after swapping
str1 = Linux str2 = Windows

q
```

13.1.1 Function Overloading with Various Data Types

The function overloading allows to use the same function name for the various data types. The function declaration, definition and calling of these functions are done with the same function name but with different data arguments. By function overloading, one can achieve greater flexibility in program.

The following program segment illustrates how the function declaration is done and called for swapping two data items for the various arguments using the concept of function overloading.

```
#include <iostream>
using namespace std;
int main()
{
    void swap (int &ix , int &iy);
    void swap ( oat &fx, oat &fy);
    void swap (char &cx,char &cy); // functions are declared
    -------
    -------
    swap(ix,iy); // functions are called with same name
    swap(fx,fy);
    swap(cx,xy);
    return 0;
}
```

The following functions are defined with the same name for the various data and arguments for performing the swapping of two quantities.

(1)

A function definition for swapping two integers.

```
void swap (int &a,int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

(2)

A function definition for swapping two floating point numbers with the same function name.

```
void swap ( oat &a,  oat &b)
{
     oat temp;
    temp = a;
    a = b;
    b = temp;
}
```

(3) A function definition for swapping two characters of the same function name.

```
void swap (char &a,char &b)
{
    char temp;
    temp = a;
    a = b;
    b = temp;
}
```

When the same name is used, the correct function will be selected by the C++ compiler by comparing the types of actual arguments with the types of formal arguments.

For example, the following program segment illustrates how to declare, define and call the same function name with different data items.

```
#include <iostream>
using namespace std;
int main()
{
    void swap (int &ix , int &iy);
    void swap ( oat &fx, oat &fy);
    void swap (char &cx,char &cy); // functions are declared
    -------
    -------
    swap(ix,iy); // actual arguments being compared with formal arguments
    return 0;
}

void swap (char &a,char &b) // formal arguments
{
    -------
    -------
}
void swap (int &a,int &b)
{
    -------
    -------
```

```
}
void swap ( oat &a, oat &b)
{
    -------
    -------
}
```

PROGRAM 13.2

A program to demonstrate how function overloading is carried out for swapping of two variables of the various data types, namely integers, floating point numbers and character types.

```
// functions overloading
//swapping two items
#include <iostream>
#include <string>
using namespace std;
int main()
{
    void swap (int &ix , int &iy);
    void swap ( oat &fx, oat &fy);
    void swap (char &cx,char &cy); // function declaration
    void swap (string &str1, string &str2);
    void menu();
    int ix,iy;
     oat fx,fy;
    char cx,cy,ch;
    string str1,str2;
    menu();
    while (( ch = cin.get()) != 'q') {
        switch(ch) {
            case 'i':
                // swapping on integers
                cout << " enter any two integers \n";
                cin >> ix >> iy ;
                cout << " swapping of integers \n";
                cout << " ix = " << ix << " iy = " << iy << endl;
                swap(ix,iy);
                cout << " after swapping \n";
                cout << " ix = " << ix << " iy = " << iy << endl;
                break;
            case 'f':
                //  oating point numbers
                cout << " enter any two  oating point numbers\n";
                cin >> fx >> fy;
                cout << " swapping of  oating point numbers \n";
                cout << " fx = " << fx << " fy = " << fy << endl;
                swap(fx,fy);
                cout << " after swapping \n";
                cout << " fx = " << fx << " fy = " << fy << endl;
                break;
            case 'c':
                //swapping characters
                cout << " enter any two characters\n";
                cin >> cx >> cy;
                cout << " swapping of characters \n";
                cout << " cx = " << cx << " cy = " << cy << endl;
                swap(cx,cy);
                cout << " after swapping \n";
                cout << " cx = " << cx << " cy = " << cy << endl;
                break;
            case 's':
                //swapping strings
```

```
                cout << " enter any two strings\n";
                cin >> str1 >> str2;
                cout << " swapping of characters \n";
                cout << " str1 = " << str1 << " str2 = " << str2;
                cout << endl;
                swap(str1,str2);
                cout << " after swapping \n";
                cout << " str1 = " << str1 << " str2 = " << str2;
                cout << endl;
                break;
            case 'm':
                menu();
                break;
        }
    } //end of while statement
    return 0;
}

void menu()
{
    cout << "Swapping two data using function overloading\n";
    cout << " i -> integer swapping \n";
    cout << " f -> real number swapping \n";
    cout << " c -> character swapping \n";
    cout << " s -> string swapping \n";
    cout << " m -> menu \n";
    cout << " q -> quit \n";
    cout << " code, please ?\n";
}

void swap (int &a,int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void swap ( oat &a, oat &b)
{
     oat temp;
    temp = a;
    a = b;
    b = temp;
}

void swap (char &a,char &b)
{
    char temp;
    temp = a;
    a = b;
    b = temp;
}

void swap (string &a,string &b)
{
    string temp;
    temp = a;
    a = b;
    b = temp;
}
```

Output of the above program

```
Swapping two data using function overloading
 i -> integer swapping
 f -> real number swapping
```

```
c -> character swapping
s -> string swapping
m -> menu
q -> quit
code, please?

i
enter any two integers
100   200
swapping of integers
ix = 100 iy = 200
after swapping
ix = 200 iy = 100

f
enter any two  oating point numbers
3.3   -5.6
swapping of  oating point numbers
fx = 3.3 fy = -5.6
after swapping
fx = -5.6 fy = 3.3

c
enter any two characters
x   y
swapping of characters
cx = x cy = y
after swapping
cx = y cy = x

s
enter any two strings
C++   Java
swapping of characters
str1 = C++ str2 = Java
after swapping
str1 = Java str2 = C++

q
```

PROGRAM 13.3

A program to demonstrate how to use the function overloading for displaying different parameters of the member functions.

```
#include <iostream>
#include <string>
using namespace std;
class abc {
    private:
       int a,b;
        oat fa,fb;
       string str;
    public:
       void display();
       void display (int a);
```

```
        void display (int a,int b);
        void display ( oat fa, oat fb);
        void display (string msg);
        void menu();
};
void abc :: display()
{
    cout <<"Calling function without arguments\n";
}
void abc :: display(int a)
{
    cout <<"Calling function with one int argument \n";
    cout << "a = " << a;
}
void abc :: display(int a, int b)
{
    cout <<"Calling function with two int arguments \n";
    cout << "a = " << a << " b = " << b << endl;
}
void abc :: display( oat fa,  oat fb)
{
    cout <<"Calling function with  oating arguments \n";
    cout << "fa = " << fa << " fb = " << fb << endl;
}
void abc :: display(string msg)
{
    cout <<"Calling function with string argument\n";
    cout << "Message = " << msg << endl;
}
void abc :: menu()
{
    cout <<"Function overloading\n";
    cout <<" 1 -> calling display()\n";
    cout <<" 2 -> calling display(int a) \n";
    cout <<" 3 -> calling display(int a, int b) \n";
    cout <<" 4 -> calling display( oat fa,  oat fb) \n";
    cout <<" 5 -> calling display (string msg) \n";
    cout <<" m -> menu() \n";
    cout <<" q -> quit \n";
}
int main()
{
    abc obj;
    void menu();
    int n1,n2;
     oat x,y;
    string str;
    char ch;
    obj.menu();
    while (( ch = cin.get()) != 'q') {
        switch (ch) {
             case '1':
                  obj.display();
                  break;
             case '2':
                  cout <<"enter an integer \n";
                  cin >> n1;
                  obj.display(n1);
                  break;
             case '3':
                  cout <<"enter any two integers \n";
                  cin >> n1 >> n2;
                  obj.display(n1,n2);
                  break;
             case '4':
                  cout <<"enter any two real numbers \n";
                  cin >> x >> y;
```

```
                    obj.display(x,y);
                    break;
               case '5':
                    cout <<"enter a string \n";
                    cin >> str;
                    obj.display(str);
                    break;
               case 'm':
                    obj.menu();
                    break;
          }
     }
     return 0;
}
```

Output of the above program

```
Function overloading
1 -> calling display ()
2 -> calling display (int a)
3 -> calling display (int a, int b)
4 -> calling display ( oat fa,  oat fb)
5 -> calling display (string msg)
m -> menu()
q -> quit

1
Calling function without arguments

2
enter an integer
10
Calling function with one int argument
a = 10

3
enter any two integers
100 200
Calling function with two int arguments
a = 100 b = 200

4
enter any two real numbers
1.1 2.2
Calling function with  oating arguments
fa = 1.1 fb = 2.2

5
enter a string
Hello,C++
Calling function with string argument
Message = Hello,C++
q
```

PROGRAM 13.4

A program to find the square of a given number belonging to the three data types, namely integers, floating point and double precision numbers without overloading of functions.

```
// function without overloading
#include <iostream>
using namespace std;
class abc {
   public:
      int square_int (int);
       oat square_ oat ( oat);
      double square_double (double);
      void menu();
};

int abc :: square_int (int a)
{
    return(a*a);
}

 oat abc :: square_ oat ( oat a)
{
    return(a*a);
}

double abc :: square_double (double a)
{
    return(a*a);
}
void abc :: menu()
{
    cout << "Finding the square of a given number\n";
    cout << " i -> integers\n";
    cout << " f -> real numbers \n";
    cout << " d -> double precision numbers\n";
    cout << " m -> menu \n";
    cout << " q -> quit \n";
    cout << " code, please ?\n";
}

int main()
{
    abc obj;
    int x,xsq;
     oat y,ysq;
    double z,zsq;
    char ch;
    obj.menu();
    while (( ch = cin.get()) != 'q') {
        switch (ch) {
            case 'i':
                cout << " enter an integer " << endl;
                cin >> x;
                xsq = obj.square_int (x);
                cout << " x = " << x;
                cout << " and its square = " << xsq << endl;
                break;
            case 'f':
                cout << " enter a  oating point number\n";
                cin >> y;
                ysq = obj.square_ oat (y);
                cout << " y = " << y ;
                cout << " and its square = " << ysq <<endl;
                break;
            case 'd':
                cout << " enter a double  " << endl;
                cin >> z;
                zsq = obj.square_double (z);
                cout << " z = " << z;
                cout << " and its square = " << zsq << endl;
                break;
```

```
            case 'm':
                obj.menu();
                break;
        }
    }
    return 0;
}
```

Output of the above program

```
Finding the square of a given number
i -> integers
f -> real numbers
d -> double precision numbers
m -> menu
q -> quit
code, please?

i
enter an integer
12
x = 12 and its square = 144

f
enter a  oating point number
3.4
y = 3.4 and its square = 11.56

d
enter a double
100.34
z = 100.34 and its square = 10068.1
q
```

PROGRAM 13.5

A program to find the square of a given number belonging to the three data types, namely integers, floating point and double precision numbers using overloading of functions.

```
// function overloading
#include <iostream>
using namespace std;
class abc {
   public:
      int square (int);
       oat square ( oat);
      double square (double);
      void menu();
};

int abc :: square (int a)
{
    return(a*a);
}

 oat abc :: square ( oat a)
{
    return(a*a);
}
```

```
double abc :: square (double a)
{
    return(a*a);
}
void abc :: menu()
{
    cout << "Finding the square of a given number\n";
    cout << " i -> integers\n";
    cout << " f -> real numbers \n";
    cout << " d -> double precision numbers\n";
    cout << " m -> menu \n";
    cout << " q -> quit \n";
    cout << " code, please ?\n";
}

int main()
{
    abc obj;
    int x,xsq;
     oat y,ysq;
    double z,zsq;
    char ch;
    obj.menu();
    while (( ch = cin.get()) != 'q') {
        switch (ch) {
             case 'i':
                  cout << " enter an integer " << endl;
                  cin >> x;
                  xsq = obj.square (x);
                  cout << " x = " << x;
                  cout << " and its square = " << xsq << endl;
                  break;
             case 'f':
                  cout << " enter a  oating point number\n";
                  cin >> y;
                  ysq = obj.square (y);
                  cout << " y = " << y ;
                  cout << " and its square = " << ysq <<endl;
                  break;
             case 'd':
                  cout << " enter a double  " << endl;
                  cin >> z;
                  zsq = obj.square (z);
                  cout << " z = " << z;
                  cout << " and its square = " << zsq << endl;
                  break;
             case 'm':
                  obj.menu();
                  break;
        }
    }
    return 0;
}
```

Output of the above program

```
Finding the square of a given number
 i -> integers
 f -> real numbers
 d -> double precision numbers
 m -> menu
 q -> quit
 code, please?

 i
 enter an integer
```

```
2
x = 2 and its square = 4

f
enter a  oating point number
4.4
y = 4.4 and its square = 19.36

d
enter a double
6.6
z = 6.6 and its square = 43.56

q
```

13.1.2 Function Overloading with Arguments

So far, we have seen how a function could be overloaded for the various data types. The functions can be overloaded not only for the different data types but also for a number of arguments in the function call. The following program segment shows how a function can be declared, defined and called for different arguments for overloading of functions.

```
#include <iostream>
using namespace std;
int main()
{

    int square (int );
    int square (int, int,int);
    -------
    -------
    asq = square (a);    // function call with a single argument
    bsq = square (x,y,z); //function call with various arguments
    -------
    -------
    return 0;
}
```

(1) A function definition to find the square of a single argument is,

```
int square (int a)
{
   return(a*a);
}
```

(2) A function definition to find the square of three arguments is,

```
int square (int a,int b, int c)
{
   int temp;
   temp = a+b+c;  // sum of three numbers
   return(temp*temp);
}
```

PROGRAM 13.6

A program to find the square of a given number with different arguments using function overloading.

```
// overloading of functions
// nding the square of the numbers
// number of arguments are also a factor
#include <iostream>
using namespace std;
int square (int );
int square (int, int,int);
int main()
{
    int a,x,y,z,asq,bsq;
    cout << " enter an integer " << endl;
    cin >> a;
    cout << " enter any three numbers \n";
    cin >> x >> y >> z;
    asq = square (a);
    cout << " a = " << a << " and its square = " <<asq << endl;
    bsq = square (x,y,z);
    cout << " x = " << x;
    cout << " ,y = " << y;
    cout << " ,z = " << z ;
    cout << " and its sum of square = " << bsq <<endl;
    return 0;
}

int square (int a)
{
    return(a*a);
}

int square (int a,int b , int c)
{
    int temp;
    temp = a+b+c;  // sum of three numbers
    return(temp*temp);
}
```

Output of the above program

```
enter an integer
2
enter any three numbers
1 2   3
a = 2 and its square = 4
x = 1, y = 2, z = 3 and its sum of square = 36
```

PROGRAM 13.7

A program to find the sum of the elements of a two-dimensional array of integers and floating point numbers without function overloading.

```
#include <iostream>
using namespace std;
int main()
{
    int sum_funct1 (int a[3][3],int n);
     oat sum_funct2 ( oat b[3][3], int n);
    int n = 3, sum1;
     oat sum2;
    static int a[3][3] = {
                          {1,2,3},
                          {4,5,6},
                          {7,8,9}
                          };
```

```
    static  oat b[3][3] = {
                          {1.1,2.2,3.3},
                          {4.4,5.5,6.6},
                          {7.7,8.8,9.9}
                          };
    sum1 = sum_funct1(a,n);
    cout << " sum of integers = " << sum1;
    cout << endl;
    sum2 = sum_funct2(b,n);
    cout << " sum of  oating point numbers = " << sum2;
    cout << endl;
    return 0;
}

int sum_funct1 ( int a[3][3], int n)
{
    int temp = 0;
    for ( int i =0; i<= n-1; ++i) {
        for (int j = 0; j<=n-1; ++j)
            temp = temp+a[i][j];
    }
    return(temp);
}

 oat sum_funct2 (  oat b[3][3], int n)
{
     oat temp = 0.0;
    for ( int i = 0; i<= n-1; ++i) {
        for (int j = 0 ; j<=n-1; ++j)
            temp = temp+b[i][j];
    }
    return(temp);
}
```

Output of the above program

```
sum of integers = 45
sum of  oating point numbers = 49.5
```

PROGRAM 13.8

A program to find the sum of the elements of a two-dimensional array of integers and floating point numbers with function overloading.

```
//function overloading
#include <iostream>
using namespace std;
int sum (int a[3][3],int n);
double sum (double b[3][3], int n);
int main()
{
    int n = 3, sum1;
    double sum2;
    static int a[3][3] = {
                          {1,2,3},
                          {4,5,6},
                          {7,8,9}
                          };
    static double b[3][3] = {
                             {1.1,2.2,3.3},
                             {4.4,5.5,6.6},
                             {7.7,8.8,9.9}
                             };
    sum1 = sum(a,n);
    cout << " sum of integers = " << sum1;
```

```
    cout << endl;
    sum2 = sum(b,n);
    cout << " sum of  oating point numbers = " << sum2;
    cout << endl;
    return 0;
}

int sum (int a[3][3], int n)
{
    int temp = 0;
    for ( int i = 0; i <= n-1; ++i) {
        for (int j = 0; j<=n-1; ++j)
            temp = temp+a[i][j];
    }
    return(temp);
}

double sum (double b[3][3], int n)
{
    double temp = 0.0;
    for (int i = 0; i <= n-1; ++i) {
        for (int j = 0 ; j<=n-1; ++j)
            temp = temp+b[i][j];
    }
    return(temp);
}
```

Output of the above program

```
sum of integers = 45
sum of  oating point numbers = 49.5
```

13.1.3 Scoping Rules for Function Overloading

By definition, overloading is a process of defining the same function name to carry out similar types of activities with various data items or with different arguments. The overloading mechanism is acceptable only within the same scope of the function declaration. Sometimes, one can declare the same function name for different scopes of the classes or with global and local declaration, but it does not come under the technique of function overloading.

For example, the following program segment illustrates how a function cannot be considered under the mechanism of function overloading since it is being declared for a different scope.

```
#include <iostream>
using namespace std;
class  rst {
    public:
       void display();
       -------
       -------
};
class second {
    public :
       void display();
       -------
       -------
};
int main()
{
     rst obj1;
    second obj2;
    -------
```

```
        -------
        obj1.display(); // no function overloading takes place
        obj2.display();
        return 0;
    }
```

PROGRAM 13.9

A program to demonstrate how function overloading is not incorporated though the same name has been used to declare the member functions in different classes.

```
#include <iostream>
using namespace std;
class  rst {
    private :
        int x;
    public :
        void getdata();
        void display();
};

class second {
    private :
        int y;
    public :
        void getdata();
        void display();
};

void  rst :: getdata()
{
    cout << " enter a value for x \n";
    cin >> x;
}

void  rst :: display()
{
    cout << " Entered number is x = " << x;
    cout << endl;
}

void second :: getdata()
{
    cout << " enter a value for y \n";
    cin >> y;
}

void second :: display()
{
    cout << " Entered number is y = " << y;
    cout << endl;
}

int main()
{
     rst obj1;
    second obj2;
    obj1.getdata();
    Obj2.getdata();
    obj1.display();
    obj2.display();
    return 0;
}
```

Output of the above program

```
enter a value for x
10
enter a value for y
20
Entered number is x = 10
Entered number is y = 20
```

PROGRAM 13.10

A program to demonstrate how to define the same function name within a global declaration and a class. Even though the same name is used to define the functions, in the different scopes, no function overloading takes place in the program.

```
#include <iostream>
using namespace std;
void display ()
{
    cout << " global display \n";
    cout << endl;
}

class sample {
    private :
        int x;
    public :
        void getdata();
        void display();
};

void sample :: getdata()
{
    cout << " enter a value for x \n";
    cin >> x;
}

void sample :: display()
{
    cout << " Entered number is :" << x;
    cout << endl;
}

int main()
{
    sample obj;
    obj.getdata();
    cout << " calling a member function display()\n";
    obj.display();
    cout << " calling a non-member function display()\n";
    ::display();
    return 0;
}
```

Output of the above program

```
enter a value for x
10
calling a member function display()
Entered number is: 10
calling a non-member function display()
global display
```

13.1.4 Special Features of Function Overloading

It is known that function overloading is the process of defining two or more functions with the same name, which differ only by return type and parameters. Some of the special features of the function overloading are discussed in this section.

(1) The function arguments must be sufficiently different since the compiler cannot distinguish which function to be called when and where.

The following program segement is invalid usage of function overloading because the arguments declared in the functions are the same.

```
#include <iostream>
using namespace std;
int main()
{
    int funct1 (int);
    int funct1 (int &a);
    int x;
    -------
    -------
    funct1(x);
    return 0;
}

int funct1 (int)
{
    -------
    -------
}

int funct1 (int &a) //error,both the arguments are same int and int&
{
    -------
    -------
}
```

The above function arguments are not sufficiently different unabling the compiler to distinguish between these functions, and hence displays an error message.

(2) While `typedef` is used for declaring a user-defined name for functions and variables, it is not a separate type but only a synonym for another type.

For example, the following function declaration cannot be used for function overloading.

```
#include <iostream>
using namespace std;
int main()
{
    typedef  oat real;
    void funct1( real);
    void funct1( oat);
    -------
    -------
    funt1(x);
    return 0;
}

void funct1(real)  // error
{
```

```
    -------
    -------
}
void funct1( oat )  // error, function arguments are same
{
    -------
    -------
}
```

(3) Even though the values of enumerated data types are integers, they are distinguished from the standard data type of `int`. So, whenever a function is declared with a function argument of `int` and an enumerated data type, it is valid in C++ for function overloading.

For example, the following function declaration is valid:

```
#include <iostream>
using namespace std;
int main()
{
    enum  day {mon,tue,wed};
    void funct1(int i);
    void funct1(day);
    -------
    -------
    return 0;
}
void funct1(int)
{
    -------
    -------
}

void funct1(day)   // valid, even the value of day is integer quantity
{
    -------
    -------
}
```

(4) The pointer arguments of a pointer variable and an array type are identical.

```
funct1(char *);
funct1(char []);  // both are same
```

For example, the following function declaration for overloading is invalid:

```
#include <iostream>
using namespace std;
int main()
{
    void funct1(char *, char *);
    void funct1 (char [], char []);
    -------
    -------
    funct1(a,b);
    return 0;
}

void funct1(char *a, char *b)
{
```

```
        -------
        -------
    }

    void funct1(char a[], char b[])   // error,
    {
        -------
        -------
    }
```

The compiler will display the error message as 'redeclaration of the function funct1 ()'.

For example, the following function declaration of a character array is invalid for function overloading even though the array sizes are different.

```
    funct1(char [20]);   // same as funct1(char *);
    funct1(char [10]);   // same as funct1(char *);
```

The following program segments are illegal construction of the function overloading.

```
#include <iostream>
using namespace std;
int main()
{
    void funct1(char [20], char [20]);
    void funct1(char [10], char [10]);
    -------
    -------
    funct1(a,b);
    return 0;
}

void funct1(char a[20], char b[20])
{
    -------
    -------
}

void funct1(char a[10], char b[10])   // error,
{
    -------
    -------
}
```

The compiler will display the error message as 'redeclaration of the function funct1 ()'.

The following function declarations are valid even though the character data types are different.

```
    funct1(char);
    funct1( unsigned char );
    funct1(signed char );
    funct1(char *);
    funct1(unsigned char *);
    funct1( signed char *);
```

13.1.5 Function Overloading Considerations

Although functions can be distinguished on the basis of return type, they cannot be overloaded on this basis. The following Table 13.1 is a partial list of considerations for function overloading in C++.

Table 13.1

Function Declaration Element	*Used for Overloading*
Function return type	No
Number of arguments	Yes
Type of arguments	Yes
Presence or absence of ellipsis	Yes
Use of typedef names	No
Unspecified array bounds	No
`const` or `volatile`	Yes

13.2 OPERATOR OVERLOADING

In the previous section, we have seen how C++ enables one to define several functions having the same name as long as they have different argument lists, known as function overloading or functional polymorphism. Its purpose is to use the same function for the basic operation.

This section explores how operators can be redefined and used in a program. Operator overloading is another example of C++ polymorphism. In fact, some form of operator overloading is available in all high level languages. For example, in BASIC the + operator can be used to carry out three different operations such as adding integers, adding two real numbers and concatenating two strings. Operator overloading is one of the most challenging and exciting features of C++. The main advantages of using overloading operators in a program are that it is much easier to read and debug.

Operators which already exist in the language, can only be overloaded. Overloading cannot alter either the basic template of an operator, nor its place in the order of precedence. It seems that the main idea of using operator overloading in a C++ program is to make it more natural to read and write. Even debugging such codes are much easier. As C++ is mostly used to develop a large software package in an easy way, due care must be taken while implementing the operator overloading in a program, otherwise, it leads to confusion while debugging and testing the codes. Operator overloading is accomplished by means of a special kind of function. Operator overloading can be carried out by means of either member functions or friend functions.

The general syntax of operator overloading is,

```
return_type operator operator_to_be_overloaded (parameters);
```

The keyword '`operator`' must be preceded by the return type of a function which gives information to the compiler that overloading of operator is to be carried out. Only those operators that are predefined in the C++ compiler are allowed to be overloaded.

Following are some examples for operator overloading of functions, their declaration and their equivalent conventional declaration.

(1)

```
void operator++ (); is equal to
void increment();
```

(2)

```
int sum (int x,int y); is equal to
int operator+ ( int x,int y);
```

Rules for overloading operators

Rule 1 Only those operators that are predefined in the C++ compiler can be used. Users cannot create new operators such as $, @ etc.

The following method of declaring an operator overloading is invalid.

```
class  newoperator {
     private :
        int x;
        int y;
     public :
        int operator $(); // error
        int operator @[]; // error
        int operator ::(); // error
};
```

Rule 2 Users cannot change operator templates. Each operator in C++ comes with its own template which defines certain aspects of its use, such as whether it is a binary operator or a unary operator and its order of precedence. This template is fixed and cannot be altered by overloading.

For example, ++ and –– cannot be used except in unary operators. During overloading, the prefixed incrementer/decrementer and the postfix incrementer/decrementer are not distinguished.

For an example,

```
++ operator ()
operator ++()
```

There is no difference between writing either prefix incrementer or postfix incrementer.

Rule 3 Overloading an operator never gives a meaning which is radically different from its natural meaning.

For example, the operator * may be overloaded to add the objects of two classes but the code becomes unreadable.

```
// error
class sample {
     private :
        int x;
        int y;
     public :
        int operator *(); // adding two objects
        int operator /(); // subtracting two objects
    };

int sample :: operator *()  // the code becomes unreadable
{
    return (x+y);
}

void main (void)
{
    -------
    -------
}
```

Rule 4 Unary operators overloaded by means of a member function take no explicit arguments and return no explicit values. When they are overloaded by means of a friend function, they take no reference argument, namely, the name of the relevant class.

13.2.1 Overloading Assignment Operator

The assignment operator is a unary operator because it is connected only to the entity on the right, which is the operand. Whenever an assignment operator is overloaded in a base class, it cannot be inherited in

a derived class. Hence the assignment operator which is to be overloaded must be either a method (or a member function) or a class object with arguments.

The following program segment illustrates the overloading of an assignment operator in a class.

```
//overloading an assignment operator
#include <iostream>
using namespace std;
class sample {
    private :
       int x;
        oat y;
    public :
       sample(int, oat);
       void operator= (sample abc);
       void display();
};

void sample :: operator= (sample abc)
{
    x = abc.x;
    y = abc.y;
}

int main()
{
    sample obj1;
    sample obj2;
    -------
    -------
    obj1 = obj2;
    obj2.display();
    return 0;
}
```

PROGRAM 13.11

A program to create a class of objects, namely, `obja` and `objb`. The contents of object `obja` is assigned to the object `objb` using the conventional assignment technique.

```
//using an assignment operator
//without overloading
#include <iostream>
using namespace std;
class sample {
    private :
        int x;
         oat y;
    public :
        sample(int, oat);
        void display();
};
sample :: sample ( int one,  oat two)
{
    x = one;
    y = two;
}
void sample :: display()
```

```
{
    cout << " integer number (x) = " << x << endl;
    cout << "  oating value (y) = " << y << endl;
    cout << endl;
}

int main()
{
    sample obj1(10,-22.55);
    sample obj2(20,-33.44);
    obj2 = obj1;
    cout << " contents of the  rst object \n";
    obj1.display();
    cout << " contents of the second object \n";
    obj2.display();
    return 0;
}
```

Output of the above program

```
contents of the  rst object
integer number (x) = 10
 oating value (y) = -22.55

contents of the second object
integer number (x) = 10
 oating value (y) = -22.55
```

PROGRAM 13.12

A program to create a class of objects, namely, obja and objb. The contents of object obja is assigned to the object objb using the operator overloading technique.

```
//overloading an assignment operator
#include <iostream>
using namespace std;
class sample {
    private :
        int x;
         oat y;
    public :
        sample(int, oat);
                //overloading assignment operator
        void operator= (sample abc);
        void display();
};

sample :: sample (int one,  oat two)
{
    x = one;
    y = two;
}

void sample :: operator= (sample abc)
{
    x = abc.x;
    y = abc.y;
}

void sample :: display()
{
    cout << " integer number (x) = :" << x << endl;
    cout << "  oating value (y) = :" << y << endl;
```

```
    cout << endl;
}

int main()
{
    sample obj1(10,-22.55);
    sample obj2(20,-33.44);
    obj1.operator =(obj2);
    cout << " contents of the  rst object \n";
    obj1.display();
    cout << " contents of the second object \n";
    obj2.display();
    return 0;
}
```

Output of the above program

```
contents of the  rst object
integer number (x) = :20
 oating value (y) = :-33.44

contents of the second object
integer number (x) = :20
 oating value (y) = :-33.44
```

PROGRAM 13.13

A program to create a class of objects, namely, obja and objb. The contents of object obja is assigned to the object objb using the operator overloading technique and the conventional method to access the overloaded operator function.

```
//overloading an assignment operator
#include <iostream>
using namespace std;
class sample {
    private :
        int x;
         oat y;
    public :
        sample(int, oat);
        void operator= (sample abc);
        void display();
};

sample :: sample (int one,  oat two)
{
    x = one;
    y = two;
}
void sample :: operator= (sample abc)
{
    x = abc.x;
    y = abc.y;
}

void sample :: display()
{
    cout << " integer number (x) = :" << x << endl;
    cout << "  oating value (y) = :" << y << endl;
    cout << endl;
}
```

```
int main()
{
    sample obj1(10,-22.55);
    sample obj2(20,-33.44);
    obj1 = obj2;
    cout << " contents of the  rst object \n";
    obj1.display();
    cout << " contents of the second object \n";
    obj2.display();
    return 0;
}
```

Output of the above program

```
contents of the  rst object
integer number (x) = :20
 oating value (y) = :-33.44

contents of the second object
integer number (x) = :20
 oating value (y) = :-33.44
```

13.3 OVERLOADING OF BINARY OPERATORS

Binary operators overloaded by means of member functions take one formal argument which is the value to the right of the operator. Binary operators, overloaded by means of friend functions take two arguments.

13.3.1 Overloading Arithmetic Operators

As arithmetic operators are binary operators, they require two operands to perform the operation. Whenever an arithmetic operator is used for overloading, the operator overloading function is invoked with single class objects.

PROGRAM 13.14

A program to perform overloading of a plus operator for finding the sum of the two given class objects.

```
//overloading arithmetic operators
#include <iostream>
using namespace std;
class sample {
    private :
        int value;
    public :
        sample ();
        sample (int one);
        sample operator+ (sample objb);
        void display();
};
sample ::sample ()
{
    value = 0;
}

sample :: sample (int one)
{
    value = one;
}
```

```
sample sample :: operator+ (sample objb)
{
    sample objsum;
    objsum.value = value+objb.value;
    return(objsum);
}

void sample :: display()
{
    cout << "value = " << value << endl;
}

int main()
{
    sample obj1(10);
    sample obj2(20);
    sample objsum;
    objsum = obj1 + obj2;
    obj1.display();
    obj2.display();
    objsum.display();
    return 0;
}
```

Output of the above program

```
value = 10
value = 20
value = 30
```

PROGRAM 13.15

A program to perform simple arithmetic operations of two complex numbers using operator overloading.

```
// complex number operations
//using operator overloading
struct complex {
      oat real;
      oat imag;
};
complex operator + (complex a, complex b);
complex operator - (complex a, complex b);
complex operator * (complex a, complex b);
complex operator / (complex a, complex b);
#include <iostream>
using namespace std;
#include <stdio.h>
int main()
{
    complex a,b,c;
    int ch;
    void menu(void);
    cout << "enter a  rst complex number \n";
    cin >> a.real >> a.imag;
    cout << " enter a second complex number \n";
    cin >> b.real >> b.imag;
    cout << "  rst complex number \n";
    cout << a.real;
    if (a.imag < 0)
        cout << "-i" << (-1)*a.imag << endl;
    else
        cout << "+i"<< a.imag << endl;
    cout << " second complex number \n";
    cout << b.real;
```

```
    if (b.imag < 0)
        cout << "-i" << (-1)*b.imag << endl;
    else
        cout << "+i"<< b.imag << endl;
    menu();
    while (( ch = getchar()) != 'q') {
        switch (ch) {
            case 'a' :
                c = a+b;
                cout << " Addition of two complex numbers \n";
                cout << c.real;
                if (c.imag < 0)
                    cout << "-i" << (-1)*c.imag << endl;
                else
                    cout << "+i"<< c.imag << endl;
                break;
            case 's' :
                c =a-b;
                cout << " Subtraction of two complex numbers \n";
                cout << c.real;
                if (c.imag < 0)
                    cout << "-i" << (-1)*c.imag << endl;
                else
                    cout << "+i"<< c.imag << endl;
                break;
            case 'm' :
                c = a*b;
                cout << " Multiplication of two complex numbers \n";
                cout << c.real;
                if (c.imag < 0)
                    cout << "-i" << (-1)*c.imag << endl;
                else
                    cout << "+i"<< c.imag << endl;
                break;
            case 'd' :
                c = a/b;
                cout << " Division of two complex numbers \n";
                cout << c.real;
                if (c.imag < 0)
                    cout << "-i" << (-1)*c.imag << endl;
                else
                    cout << "+i"<< c.imag << endl;
                break;
            case 't':
                menu();
                break;
        } // end of switch
    }
    return 0;
} // end of main program

void menu(void)
{
    cout << " complex number operations \n";
    cout << " t -> menu () \n";
    cout << " a -> addition \n";
    cout << " s -> subtraction \n";
    cout << " m -> multiplication \n";
    cout << " d -> division \n";
    cout << " q -> quit \n";
    cout << " option, please ? \n";
}

complex operator + ( struct complex a, struct complex b)
{
    complex c;
```

```
    c.real = a.real+b.real;
    c.imag = a.imag+b.imag;
    return(c);
}

complex operator - (struct complex a, struct complex b)
{
    complex c;

    c.real = a.real-b.real;
    c.imag = a.imag-b.imag;
    return(c);
}

complex operator * (struct complex a, struct complex b)
{
    complex c;

    c.real = (a.real*b.real)-(a.imag*b.imag);
    c.imag = (a.real*b.imag)+(a.imag*b.real);
    return(c);
}
complex operator / (struct complex a, struct complex b)
{
    complex c;
     oat temp;
    temp = (b.real*b.real)+(b.imag*b.imag);
    c.real = ((a.real*b.real)+(a.imag*b.imag))/temp;
    c.imag = ((b.real*a.imag)-(a.real*b.imag))/temp;
    return(c);
}
```

Output of the above program

```
enter a  rst complex number
1   1
enter a second complex number
2   2
 rst complex number
1+i1
second complex number
2+i2
complex number operations
t -> menu ()
a -> addition
s -> subtraction
m -> multiplication
d -> division
q -> quit
option, please?

a
Addition of two complex numbers
3+i3

s
Subtraction of two complex numbers
-1-i1

m
Multiplication of two complex numbers
0+i4
```

```
d
Division of two complex numbers
 0.5+i0

 q
```

13.3.2 Overloading of Comparison Operators

Comparison and logical operators are binary operators that require two objects to be compared and hence the result will be one of these: (Table 13.2)

Table 13.2

Operator	*Meaning*
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

The return value of the operator function is an integer. Operator overloading accepts an object on its right as a parameter and the object on the left is passed by the this pointer.

PROGRAM 13.16

A program to demonstrate how to overload a less than operator in a program to compare two classes of objects. The operator overloading returns either 1 or 0 as the case may be.

```
//overloading comparison operators
#include <iostream>
using namespace std;
class sample {
    private:
        int value;
    public :
        sample();
        sample(int one);
        void display();
        int operator < (sample obj);
};

sample :: sample ()
{
    value = 0;
}
sample :: sample (int one)
{
    value = one;
}

void sample :: display()
{
    cout << " value " << value << endl;
}
int sample :: operator < (sample obja)
{
    return (value < obja.value);
}
```

```
int main()
{
     sample obja(20);
     sample objb(100);
     cout << (obja < objb) << endl;
     cout << (objb < obja) << endl;
     return 0;
}
```

Output of the above program

```
1
0
```

13.4 OVERLOADING OF UNARY OPERATORS

Unary operators overloaded by member functions take no formal arguments, whereas when they are overloaded by friend functions they take a single argument.

Overloading of incrementer and decrementer It is well known that C++ supports very unusual notation or operator that is used for incrementing and decrementing by 1. These operators can be used as either perfix or postfix. In general, overloading of these operators cannot be distinguished between prefix or postfix operation. However, whenever a postfix operation is overloaded, it takes a single argument along a member function of a class object.

The following program segment illustrates the overloading of an incrementer operator with prefix operation.

```
//overloading pre x ++ incrementer operator
#include <iostream>
using namespace std;
class  bonacci {
    public:
       void operator++();
       -------
       -------
};

void  bonacci :: operator++()
{
       -------
       -------
}
int main()
{
     bonacci obj;
    -------
    -------
    ++obj;
    return 0;
}
```

PROGRAM 13.17

A program to generate a Fibonacci series by overloading a prefix operator.

```
//overloading pre x ++ incrementer operator
#include <iostream>
#include <iomanip>
using namespace std;
struct  bonacci {
    public:
        unsigned long int f0,f1, b;
         bonacci(); // constructor
        void operator++();
        void display();
};

 bonacci ::  bonacci ()
{
    f0 = 0;
    f1 = 1;
     b = f0+f1;
}

void  bonacci :: display()
{
    cout << setw(4) <<  b;
}

void  bonacci :: operator++()
{
    f0 = f1;
    f1 =  b;
     b = f0+f1;
}

int main()
{
     bonacci obj;
    int n;
    cout << " How many  bonacci numbers are to be displayed ? \n";
    cin >>n;
    cout << obj.f0 << setw(4) << obj.f1;
    for (int i = 2; i <= n-1; ++i) {
        obj.display();
        ++obj;
    }
    cout << endl;
    return 0;
}
```

Output of the above program

```
How many  bonacci numbers are to be displayed?
8
0   1   1   2   3   5   8   13
```

Postfix incrementer Unlike overloading a prefix operator, overloading a postfix operator always takes a single argument so as to distinguish between the prefix and the postfix operations.

The following program segment illustrates the overloading of an incrementer operator with postfix operation.

```
//using operator overloading of post x incrementer
#include <iostream>
using namespace std;
class  bonacci {
   public :
        bonacci operator++(int);
};
```

```
 bonacci  bonacci :: operator++ (int x)
{
   -------
   -------
   return *this;
}
int main()
{
     bonacci obj;
    -------
    -------
    obj++;
    return 0;
}
```

PROGRAM 13.18

A program to generate a Fibonacci series by overloading a postfix operator.

```
//generation of  bonacci numbers
//using operator overloading of post x incrementer
#include <iostream>
#include <iomanip>
using namespace std;
class  bonacci {
    public :
       unsigned long int f0,f1, b;
        bonacci(); // constructor
        bonacci operator++(int);
       void display();
};

 bonacci ::  bonacci ()
{
    f0 = 0;
    f1 = 1;
     b = f0+f1;
}

void  bonacci :: display()
{
    cout << setw(4) <<  b;
}

 bonacci  bonacci :: operator++ (int x)
{
    f0 = f1;
    f1 =  b;
     b = f0+f1;
    return *this;
}
int main()
{
     bonacci obj;
    int n;
    cout << " How many  bonacci numbers are to be displayed ? \n";
    cin >>n;
    cout << obj.f0 << setw(4) << obj.f1;
    for (int i = 2; i<= n-1; ++i) {
        obj.display();
        obj++;
```

```
    }
    cout << endl;
    return 0;
}
```

Output of the above program

```
How many  bonacci numbers are to be displayed?
9
0    1    1    2    3    5    8    13   21
```

A summary of C++ operators that can be overloaded are given below.

Type 1 Binary operators The following binary operators can be overloaded. (Table 13.3)

Table 13.3

Operator	*Meaning*
`[]`	Array element reference
`()`	Function call
`new`	New operator
`delete`	Delete operator
`*`	Multiplication
`/`	Division
`%`	Modulus
`+`	Addition
`-`	Subtraction
`<<`	Left shift
`>>`	Right shift
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`==`	Equality
`!=`	Inequality
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`&&`	Logical AND
`||`	Logical OR
`=`	Assignment
`*=`	Mulitply and assign
`/=`	Divide and assign
`%=`	Modulus and assign
`+=`	Add and assign
`-=`	Subtract and assign
`<<=`	Shift left and assign
`>>=`	Shift right and assign
`&=`	Bitwise AND and assign
`|=`	Bitwise OR and assign
`^=`	Bitwise one's complement and assign
`,`	Comma

Type 2 Unary operators Unary operators that can be overloaded are: (Table 13.4)

Table 13.4

Operator	*Meaning*
`->`	Indirect member operator
`!`	Logical negation
`*`	Pointer reference (indirection)
`&`	Address
`~`	Ones complement
`->*`	Indirect pointer to member
`+`	Addition
`-`	Subtraction
`++`	Incrementer
`--`	Decrementer
`-`	Unary minus

Type 3 Operators common to unary and binary forms The following operators are used both for unary and binary forms: (Table 13.5)

Table 13.5

Operator	*Meaning*
`+`	addition
`-`	subtraction and unary minus
`*`	multiplication and pointer reference (indirection)
`&`	bitwise AND and address of

Type 4 Operators that cannot be overloaded The following operators cannot be used for overloading purposes: (Table 13.6)

Table 13.6

Operator	*Meaning*
`.`	Direct member or class member operator
`.*`	Direct pointer to member (Class member dereference)
`::`	Scope resolution operator
`?:`	Conditional operator
`sizeof`	Size in bytes operator
`#, ##`	Preprocessing symbols

REVIEW QUESTIONS

1. What is meant by overloading in object-oriented programming paradigm?
2. What is function overloading?
3. What are the syntactic rules governing the definition of a function overloading?
4. List the merits and demerits of function overloading over the conventional functional usages.

5. What are the scope rules governing the function overloading?
6. What is meant by operator overloading?
7. What are the C++ operators that cannot be overloaded? Explain.
8. List the C++ operators that can be overloaded for binary usages.
9. List the C++ operators that can be overloaded for unary usages.
10. What are the C++ operators that can be used both for binary and unary applications?
11. Can the operator precedence and syntax be changed through overloading? Explain.
12. What are the operators that may not require to be defined either in a method or a class object among arguments in overloading?
13. Explain how memory management operators are used for overloading in C++.
14. Explain the syntax rules that govern the overloading of increment or decrement operators.
15. Explain how the preincrementer and postincrementer are taken care of while declaring operator overloading.
16. Explain the operation of overloading of an assignment operator.
17. Explain how logical operators can be used for overloading in C++?
18. Explain how the predecrementer and postdecrementer are taken care of while declaring operator overloading.

CONCEPT REVIEW PROBLEMS

1. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
int main()
{
    int min(int, int);
     oat min( oat, oat);
    double min(double,double);
    int a = 10,b = 20;
     oat fa = 1.1, fb = -2.2;
    double da = 100,db = 300;
    cout << "\n Minimum = " << min(a,b);
    cout << "\n Minimum = " << min(fa,fb);
    cout << "\n Minimum = " << min(da,db);
    return 0;
}

int min(int a,int b)
{
    return (a < b ? a : b);
}

 oat min( oat a,  oat b)
{
    return (a < b ? a : b);
}

double min(double a, double b)
{
```

```
    return (a < b ? a : b);
}
```

(b)

```
#include <iostream>
using namespace std;
int main()
{
    void display();
    void display (int);
    void display(int, int);
    void display(int, int, int);
    int a = 10, b = 20, c = 30;
    if ( a++ > ++b)
       display(a);
    else if (b++ > ++c)
       display(b,c);
    else if ( b > ++c || a > c)
       display(a,b,c);
    else
       display();
    return 0;
}

void display(int a)
{
    cout << " a = " << a << endl;
}

void display(int b, int c)
{
    cout << " b = " << b << endl;
    cout << " c = " << c << endl;
}

void display(int a, int b, int c)
{
    cout << " a = " << a << endl;
    cout << " b = " << b << endl;
    cout << " c = " << c << endl;
}

void display()
{
    cout << " Hello, C++ world \n";
}
```

(c)

```
#include <iostream>
using namespace std;
int main()
{
    void display();
    void display (int);
    void display(int, int);
    void display(int, int, int);
    int a = 10, b = 20, c = 30;
    if (a++ > ++b)
```

```
            display(a);
        else if (b++ > ++c)
            display(b,c);
        else if ((b < ++c) && (++a < ++c))
            display(a,b,c);
        else
            display();
        return 0;
    }

    void display(int a)
    {
        cout << " a = " << a << endl;
    }

    void display(int b, int c)
    {
        cout << " b = " << b << endl;
        cout << " c = " << c << endl;
    }

    void display(int a, int b, int c)
    {
        cout << " a = " << a << endl;
        cout << " b = " << b << endl;
        cout << " c = " << c << endl;
    }

    void display()
    {
        cout << " Hello, C++ world \n";
    }
```

(d)

```
    #include <iostream>
    #include <string>
    using namespace std;
    int main()
    {
        void display();
        void display (string);
        void display(string, string);
        void display(string, string, string);
        string str1 = "Hello";
        string str2 = "C++";
        string str3 = "world";
        int x = 10, y = 3;
        int temp = ++x % ++y;
        switch (temp) {
            case 10:
                display();
                break;
            case 3:
                display(str1,str2,str3);
                break;
            case 4:
                display(str1);
```

```
                break;
            case 11:
                display (str1,str2);
                break;
        }
        return 0;
    }

    void display()
    {
        cout << "Nothing to display \n";
    }

    void display(string a)
    {
        cout << " str1 = " << a << endl;
    }

    void display(string b, string c)
    {
        cout << " str1 = " << b << endl;
        cout << " str2 = " << c << endl;
    }

    void display(string a, string b, string c)
    {
        cout << " str1 = " << a << endl;
        cout << " str2 = " << b << endl;
        cout << " str3 = " << c << endl;
    }
```

(e)

```
    #include <iostream>
    #include <string>
    using namespace std;
    class abc {
        public:
            void display();
            void display (string);
            void display(string, string);
            void display(string, string, string);
    };
    int main()
    {
        abc obj;
        string str1 = "Hello";
        string str2 = "C++";
        string str3 = "world";
        int x = 11, y = 3;
        int temp = ++x % y++;
        switch (temp) {
            case 10:
                obj.display();
                break;
            case 3:
                obj.display(str1,str2,str3);
```

```
                break;
            case 4:
               obj.display(str1);
               break;
            case 0:
               obj.display (str1,str2);
               break;
        }
        return 0;
    }

    void abc :: display()
    {
        cout << "Nothing to display \n";
    }

    void abc :: display(string a)
    {
        cout << " str1 = " << a << endl;
    }

    void abc :: display(string b, string c)
    {
        cout << " str1 = " << b << endl;
        cout << " str2 = " << c << endl;
    }

    void abc :: display(string a, string b, string c)
    {
        cout << " str1 = " << a << endl;
        cout << " str2 = " << b << endl;
        cout << " str3 = " << c << endl;
    }
```

(f)

```
    #include <iostream>
    #include <string>
    using namespace std;
    class abc {
        public:
            void seta(int a[]);
            void setfa( oat fa[]);
            int sum(int a[]);
             oat sum ( oat fa[]);
    };

    void abc :: seta(int a[])
    {
        for (int i = 0; i <= 9; ++i)
           a[i] = i;
    }

    void abc :: setfa( oat fa[])
    {
        for (int i = 0; i <= 9; ++i)
           fa[i] = i+0.1;
```

```
}

int abc :: sum(int a[])
{
    int temp = 0;
    for (int i = 0; i <= 9; ++i)
       temp += a[i];
    return (temp);
}

 oat abc :: sum( oat a[])
{
     oat temp = 0;
    for (int i = 0; i <= 9; ++i)
       temp += a[i];
    return (temp);
}

int main()
{
    abc obj;
    int a[10];
     oat fa[10];
    obj.seta(a);
    obj.setfa(fa);
    cout << " sum a[] = " << obj.sum(a) << endl;
    cout << " sum fa[] = " << obj.sum(fa) << endl;
    return 0;
}
```

2. What will be the output of each of the following program when it is executed?

(a)

```
#include <iostream>
using namespace std;
class abc {
    private :
       int x;
    public :
       abc(int);
       void display();
};
abc :: abc (int value)
{
    x = value;
}
void abc :: display()
{
    cout << " x = " << x << endl;
}

int main()
{
    abc obj1(10);
    abc obj2(20);
    obj2 = obj1;
    cout << " contents of the  rst object \n";
```

```
    obj1.display();
    cout << " contents of the second object \n";
    obj2.display();
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
class abc {
   private :
      int x;
   public :
      abc(int);
      void operator= (abc operabc);
      void display();
};

abc :: abc(int value)
{
    x = value;
}

void abc :: operator= (abc operabc)
{
     x = ++(operabc.x);
}

void abc :: display()
{
    cout << " x = " << x << endl;
}

int main()
{
    abc obj1(100);
    abc obj2(20);
    obj1.operator= (obj2);
    cout << " contents of the  rst object \n";
    obj1.display();
    cout << " contents of the second object \n";
    obj2.display();
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
class abc {
   private :
      int x;
   public :
      abc(int);
      void operator= (abc operabc);
      void display();
};
```

```
abc :: abc( int value)
{
    x = value;
}

void abc :: operator= (abc operabc)
{
     x = ++(operabc.x);
}

void abc :: display()
{
    cout << " x = " << x << endl;
}

int main()
{
    abc obj1(100);
    abc obj2(20);
    obj1.operator= (obj2);
    cout << " contents of the  rst object \n";
    obj1.display();
    obj2.operator= (obj2);
    cout << " contents of the second object \n";
    obj2.display();
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
class abc{
    private:
       int x;
    public:
       abc();
       void operator++ (int);
       void display();
};
abc :: abc()
{
    x = 0;
}
void abc ::operator ++(int x)
{
    x++;
}
void abc ::display()
{
    cout << " x = " << x << endl;
}
int main()
{
    abc obj;
    obj.display();
    obj.operator ++(10);
```

```
    obj.display();
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
class abc{
    private:
       int x;
    public:
       abc();
       abc operator++();
       void display();
};
abc :: abc()
{
    x = 0;
}
abc abc ::operator ++()
{
    ++x;
    return *this;
}
void abc ::display()
{
    cout << " x = " << x << endl;
}
int main()
{
    abc obj;
    obj.display();
    obj++;
    obj.display();
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
class abc{
    private:
       int x;
    public:
       abc();
       abc operator++();
       void display();
};
abc :: abc()
{
    x = 0;
}
```

```
abc abc ::operator ++()
{
    x++;
    return *this;
}
void abc ::display()
{
    cout << " x = " << x << endl;
}
int main()
{
    abc obj;
    obj.display();
    obj++;
    obj.display();
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
class abc {
    private:
       int value;
    public :
       abc();
       abc(int );
       int operator >= (abc obj);
};

abc :: abc()
{
    value = 0;
}
abc :: abc( int one)
{
    value = one;
}

int abc :: operator >= ( abc obja)
{
    return ( value >= obja.value);
}

int main()
{
    abc obja(20);
    abc objb(100);
    cout << (obja >= objb) << endl;
    cout << (objb >= obja) << endl;
    return 0;
}
```

PROGRAMMING EXERCISES

1. Write a program in C++ to read a set of integers up to `n`, where `n` is defined by the user and stored in a one-dimensional array. Also, read a set of floating point numbers of the same size and store it into another array and print the contents of these two arrays separately, using the function overloading technique.
2. Write a program in C++ to perform the following using function overloading concepts:
   (i) To read a set of integers,
   (ii) To read a set of floating point numbers,
   (iii) To read a set of double numbers individually, and find out the average of the non-negative integer and also to calculate the deviation of the numbers.
3. Write a program in C++ using function overloading method to read a set of integers and floating point numbers separately and to store it in the corresponding arrays. Again read a number 'd' from the keyboard and check whether the number 'd' is present in the arrays. If it is so, print out how many times the number 'd' is repeated in the array.
4. Write a program in C++ using function overloading to read two matrices of different data types such as integers and floating point numbers. Find out the sum of the above two matrices separately and display the total sums of these arrays individually.
5. Develop a program in C++ using function overloading for adding two given integer matrices; two floating point number matrices and double precision value matrices, separately.
6. Develop a program in C++ using function overloading for subtracting two given integer matrices; two floating point number matrices and double precision value matrices separately.
7. Develop a program in C++ using function overloading for multiplying two given integer matrices; two floating point number matrices and double precision value matrices separately.
8. Write a program in C++ using operator overloading for the binary numbers to perform simple arithmetic operations that has the functions to add, subtract, multiply and divide.
9. Write a program in C++ using operator overloading to find a factorial of a given number.
10. Write a program in C++ to check whether a given number is prime or not, using operator overloading.
11. Write a program in C++ to perform the following using operator overloading:
    (i) Area of a circle
    (ii) Circumference of a circle
    (iii) Area of a rectangle
    (iv) Area of a triangle
    (v) Perimeter of a square
12. Write a program in C++ to find the simple and compound interest (I) of a given principal (P), rate of interest (R) and number of years (N), using operator overloading.

Polymorphism and Virtual Functions

Chapter 14

This chapter focusses on the implementation of the concept of polymorphism using the keyword 'virtual'. The emphasis is on how to realise the virtual functions; pure virtual functions; virtual destructors; and virtual base classes in C++. The early binding and the late binding of compilers are discussed with suitable examples.

14.1 POLYMORPHISM

A true object-oriented programming paradigm must consist of three items: *data abstraction* and *encapsulation*, *inheritance* and *polymorphism*. Data abstraction and encapsulation are the processes of defining some new data types. This involves the internal representation of the data members and the methods of declaration, definition and uses in a program. Inheritance involves the creation of a new type from an existing type in some hierarchical fashion. Data hiding, encapsulation and inheritance are explained in the earlier chapters.

The word 'poly' originated from a Greek word meaning `many` and 'morphism' from a Greek word meaning `form`, and thus 'polymorphism' means many forms. In object-oriented programming, polymorphism refers to identically named methods (member functions) that have different behaviour depending on the type of object they refer.

Polymorphism is the process of defining a number of objects of different classes into a group and call the methods to carry out the operation of the objects using different function calls. In other words, polymorphism means 'to carry out different processing steps by functions having same messages'. It treats objects of related classes in a generic manner. The keyword `virtual` is used to perform the polymorphism concept in C++. Polymorphism refers to the run-time binding to a pointer to a method.

14.2 EARLY BINDING

Choosing a function in normal way, during compilation time is called as early binding or static binding or static linkage. During compilation time, the C++ compiler determines which function is used based on the parameters passed to the function or the function's return type. The compiler then substitutes the correct function for each invocation. Such compiler-based substitutions are called static linkage. Whatever functions discussed so far in the earlier chapters, are based on static binding only.

By default, C++ follows early binding. With early binding, one can achieve greater efficiency. Function calls are faster in this case because all the information necessary to call the function are hard coded.

The purest object-oriented programming language like Small talk permits only run-time binding of the methods, whereas C++ allows both compile-time binding and run-time binding. In that sense, C++ is a hybrid language as it generates the code of both procedural and object-oriented programming paradigm.

PROGRAM 14.1

A program to demonstrate the operation of the static binding. In this program, a message is given to access the methods of the derived class from the base class members through pointer techniques.

```
//demonstration of static binding
#include <iostream>
using namespace std;
class square {
    protected:
        int x;
    public:
        void getdata();
        void display();
        int area();
};

class rectangle : public square
{
    protected:
        int y;
    public:
        void getdata();
        void display();
        int area();
};

void square:: getdata()
{
    cout << " enter the value of side x? \n";
    cin >> x;
}
void square:: display()
{
    cout << " value of x = y = " << x << endl;
    cout << " Area of the square = " << area();
    cout << endl;
}
int square :: area()
{
    int temp = x*x;
    return(temp);
}
void rectangle :: getdata()
{
```

```
    cout << " enter the value of sides x and y ? \n";
    cin >> x >> y;
}

void rectangle:: display()
{   cout << " value of x = " << x << " and y = " << y << endl;
    cout << " Area of the rectangle = " << area();
    cout << endl;
}
int rectangle  :: area()
{
    int temp = x*y;
    return(temp);
}

int main()
{
    square sqobj;
    rectangle rectobj;
    square *ptr;
    ptr = &sqobj;
    ptr = &rectobj;
    ptr->getdata();
    ptr->area();
    ptr->display();
    return 0;
}
```

Output of the above program

```
enter the value of side x?
10
value of x = y = 10
Area of the square = 100
```

The derived class rectangle is inherited from the base class square through public derivation. It is known that an object of a derived class not only inherits characteristics from the base class but also has characteristics that are specific to the derived class.

For example, the following program segment illustrates the use of message calling from the derived class objects to the base class objects.

```
int main()
{
    square sqobj;
    rectangle rectobj;
    square *ptr;
    ptr = &sqobj;
    ptr = &rectobj;
    ptr->getdata();
    ptr->area();
    ptr->display();
    return 0;
}
```

An object of the derived can be accessed by the pointer of a base class. The following way of declaration and assignment of the pointer to the base class is permitted in C++.

```
square  *ptr;        //pointer to base class object
rectangle rectobj;   //object of the derived class is created
ptr = &rectobj;      //indirect reference to the pointer base
```

Through the pointer of the base class, the derived class data members and member functions can be accessed. Therefore, the following member reference statement is valid.

```
ptr->getdata();
ptr->area();
ptr->display();
```

The function call is made from the base class pointer to access members of the derived class. Even though the specific function call is made to access the members of the derived class, the function call (message) has not reached the derived class members. So the base class members have been displayed but not the derived class members. It is because the member functions of the base class and the derived class are declared non-virtual and C++ compiler takes by default only a static binding. The function call through static binding is shown in Fig. 14.1.

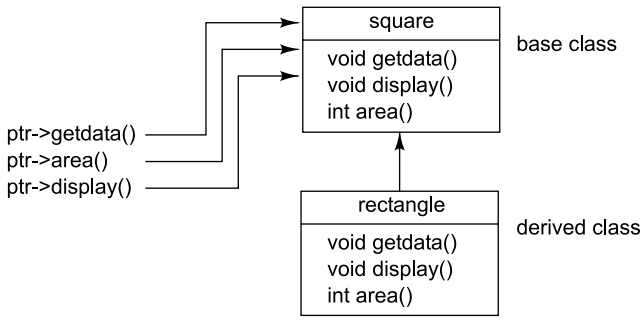


Fig. 14.1 *Function Call through Static Binding*

C++ supports polymorphism through virtual methods and pointers. In the previous example, the following methods have been declared as virtual in the base class.

```
virtual void getdata();
virtual void area();
virtual void display();
```

The derived class contains the same methods with non-virtual members.

```
void getdata();
void area();
void display();
```

For non-virtual functions with the same name, the system determines at compile time the function to be invoked. For virtual methods with the same name, the system determines at run time the methods to invoke. The invoked methods are determined by the type of object to which the pointer points.

For example, if `ptr` currently points to an object of a class square, then the members of the virtual methods of class square are invoked by

```
ptr->getdata();   // virtual methods of the base class
ptr->area();      // virtual methods of the base class
ptr->display();   // virtual methods of the base class
```

If `ptr` currently points to objects of a class rectangle, then the members of the class rectangle are invoked by the same message of the function call.

```
ptr->getdata();   // methods of the derived class
ptr->area();      // methods of the derived class
ptr->display();   // methods of the derived class
```

By now, one can understand that polymorphism defines identically named methods (member functions) that have different behaviours by the same message (function call). In other words, polymorphism refers to the run-time binding of a pointer to a method. The keyword virtual is used to carry out the run-time binding of a pointer which points to a member function of an object.

The function call through dynamic binding is shown in Fig.14.2.

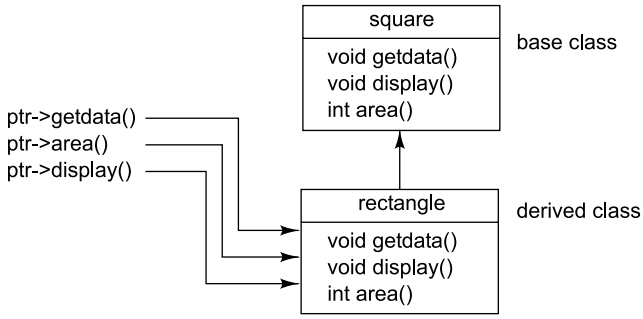


Fig. 14.2 *Function Call through Dynamic Binding*

PROGRAM 14.2

A program to demonstrate the compile time binding of the member functions of the class. The same message is given to access the derived class member functions from the array of pointers. As functions are declared as non-virtual, the C++ compiler invokes only the static binding.

```
//nonvirtual function
//demonstration of compile time binding using
//array of pointers
#include <iostream>
using namespace std;
class baseA {
    public :
        void display() {
            cout << "One \n";
        }
};
class derivedB: public baseA
{
    public:
        void display(){
            cout << " Two \n";
        }
};
class derivedC: public derivedB
{
    public:
        void display(){
            cout << " Three \n";
        }
};
int main()
{
    //define three objects
    baseA obja;
    derivedB objb;
    derivedC objc;
    baseA *ptr[3]; //define an array of pointers to baseA
    ptr[0] = &obja;
    ptr[1] = &objb;
    ptr[2] = &objc;
    for (int i = 0; i<=2; i++)
```

```
        ptr[i]->display(); //same message for all objects
    return 0;
}
```

Output of the above program

```
One
One
One
```

The function call through static binding is shown in Fig. 14.3.

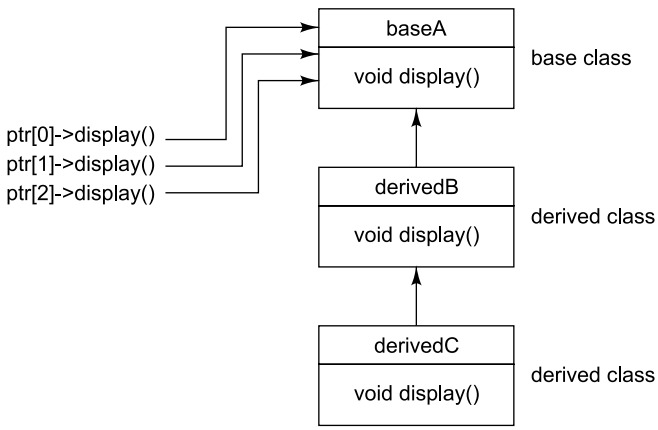


Fig. 14.3 *Function Call through Static Binding*

14.3 POLYMORPHISM WITH POINTERS

Pointers are also central to polymorphism in C++. To enable polymorphism, C++ allows a pointer in a base class to point to either a base class object or to any derived class object. The following program segment illustrates how a pointer is assigned to point to the object of the derived class.

```
class baseA {
    -------
    -------
};
class derivedD : public baseA
{   -------
    -------
};
int main()
{
    baseA  *ptr;// pointer to baseA
    derivedD  objd;
    ptr = &objd; //indirect reference objd to the pointer
    -------
    -------
    return 0;
}
```

The pointer `ptr` points to an object of the derived class `objd`.

By contrast, a pointer to a derived class object may not point to a base class object without explicit casting. For example, the following assignment statements are invalid:

```
int main()
{
    baseA  obja
    derivedD  *ptr;
    ptr = &obja; // invalid
    -------
    -------
    return 0;
}
```

Note that a derived class pointer cannot point to base class objects. But, the above code can be corrected by using explicit casting.

```
int main()
{
    square sqobj;
    rectangle *ptr;              //pointer of the derived class
    ptr = (rectangle*) &sqobj;   //explicit casting
    ptr->display();
    -------
    -------
    return 0;
}
```

The following program segment illustrates how a different assignment of a pointer to an object of a base class be given to an object of a derived class.

(1) A base class pointer can point to the object of the same class or a derived class.

```
class baseA {
    -------
    -------
};
class derivedD: public baseA
{
    -------
    -------
};
int main()
{
    baseA obja;
    derivedD  objd;
    baseA  *ptr;
    ptr = &obja; //valid, pointer to a same class
    ptr = &objd; //valid, pointer to a derived class
    -------
    -------
    return 0;
}
```

(2) A derived class pointer cannot point to an object of a base class but it can point to the same class object.

```
class baseA {
    -------
    -------
};
```

```
class derivedD: public baseA
{
    -------
    -------
};
int main()
{
    baseA obja;
    derivedD  objd;
    derivedD  *ptr;
    ptr = &obja;      //error, cannot point to base class object
    ptr = &objd;      //valid,pointer to a same class object
    -------
    -------
    return 0;
 }
```

Note that a derived class pointer cannot point to an object of a base class.

PROGRAM 14.3

A program to illustrate how to assign the pointer of the derived class to the object of a base class using explicit casting.

```
//demonstration of run time binding
//pointer of the derived class points to a base class object
//using explicit casting
#include <iostream>
using namespace std;
class square {
    protected:
        int x;
    public:
        virtual void getdata();
        virtual void display();
        virtual int area();
};
class rectangle: public square
{
    protected:
        int y;
    public :
        void getdata();
        void display();
        int area();
};
void square:: getdata()
{
        cout << " enter the value of side x? \n";
        cin >> x;
}
void square:: display()
{
    cout << " value of x = y = " << x << endl;
    cout << " Area of the square = " << area();
    cout << endl;
}
int square :: area()
{
    int temp = x*x;
    return(temp);
```

```
}
void rectangle :: getdata()
{
    cout << " enter the value of sides x and y ? \n";
    cin >> x >> y;
}
void rectangle:: display()
{

    cout << " value of x = " << x << " and y = " << y << endl;
    cout << " Area of the rectangle = " << area();
    cout << endl;
}
int rectangle  :: area()
{
    int temp = x*y;
    return(temp);
}

int main()
{
    square sqobj;
    rectangle *ptr;              //pointer of the derived class
    ptr = (rectangle*)&sqobj;    //explicit casting
    ptr->getdata();
    ptr->area();
    ptr->display();
    return 0;
}
```

Output of the above program

```
enter the value of side x?
value of x = y = 12
Area of the square = 144
```

14.4 VIRTUAL FUNCTIONS

A virtual function is one that does not really exist but it appears real in some parts of a program. Virtual functions are advanced features of the object-oriented programming concept and they are not necessary for each C++ program. This section presents how the polymorphic features are incorporated using the virtual functions.

The general syntax of the virtual function declaration is:

```
class user_defined_name {
    private:
      -------
      -------
    public:
      virtual return_type function_name1(arguments);
      virtual return_type function_name2(arguments);
      virtual return_type function_name3(arguments);
      -------
      -------
};
```

To make a member function virtual, the keyword `virtual` is used in the methods while it is declared in the class definition but not in the member function definition. The keyword virtual should be preceded by a return type of the function name. The compiler gets information from the keyword virtual that it is a virtual function and not a conventional function declaration.

The syntax diagram of the virtual method declaration is given in Fig. 14.4.

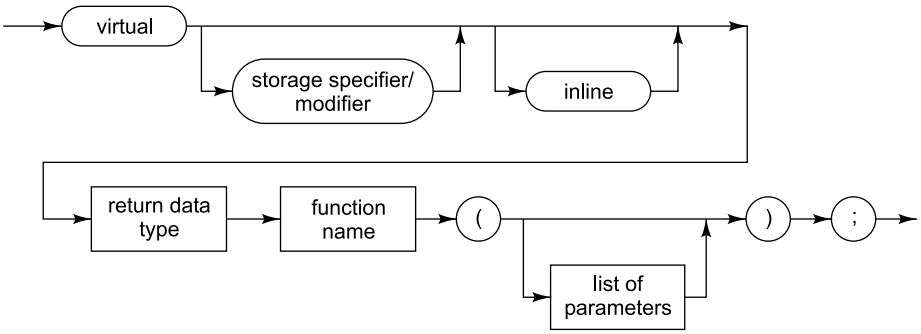


Fig. 14.4 *Syntax Diagram of Virtual Method Declaration*

For example, the following declaration of the virtual function is valid.

```
class sample {
    private:
       int x;
       float y;
    public:
       virtual void display();
       virtual int sum();
};
```

Some of the invalid declarations of the virtual functions in C++ are given below:

(1) The keyword virtual should not be repeated in the definition if the definition occurs outside the class declaration. The use of a function specifier virtual in the function definition is invalid.

```
class sample {
    private :
       int x;
       float y;
    public :
       virtual void display();
};
virtual void sample:: display() //error
{
    -------
    -------
}
```

(2) A virtual function cannot be a static member because a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.

```
class sample {
    private:
       int x;
       float y;
    public:
       virtual static int sum(); //error, due to static
};
int sample:: sum()
{
```

```
    -------
    -------
}
```

(3) A virtual function cannot have a constructor member function but it can have the destructor member function.

```
class sample {
    private:
       int x;
       float y;
    public:
       virtual sample (int xx, float yy); //constructor
       void getdata();
       void display();
};
```

Note that it is an error to make a constructor virtual type.

(4) A destructor member function does not take any argument and no return type can be specified for it not even void.

```
class sample {
    private :
       int x;
       float y;
    public :
       virtual ~ sample (int xx, float yy); //invalid
       void getdata();
       void display();
};
```

Note that the destructor member function can be virtual even though it does not take any argument.

(5) It is an error to redefine a virtual method with a change of return data type in the derived class with the same parameter types as those of a virtual method in the base class.

```
class baseA {
    private :
       int x;
       float y;
    public:
       virtual int sum (int xx,float yy);  //error
};
class derivedD: public baseA {
    private:
       int z;
    public:
       virtual float sum (int xx,float yy);
};
```

The above declarations of two virtual functions are invalid due to the different type of return value. Even though these functions take identical arguments, the return data types are different.

The following manner, one can correct the above said error

```
virtual float sum (int xx,float yy); // base class
virtual float sum (int xx,float yy); // derived class
```

Both the above functions can be written with int data types in the base class as well as in the derived class as

```
virtual int sum (int xx,float yy); // base class
virtual int sum (int xx,float yy); // derived class
```

(6) Only a member function of a class can be declared as virtual. It is an error to declare a non-member function (non-method) of a class virtual.

```
virtual void display() //error, nonmember function
{
    -------
    -------
}
```

14.5 LATE BINDING

Choosing functions during execution time is called late binding or dynamic binding or dynamic linkage. Late binding requires some overhead but provides increased power and flexibility. The late binding is implemented through virtual functions. An object of a class must be declared either as a pointer to a class or a reference to a class.

For example, the following declaration of the virtual function shows how a late binding or run-time binding can be carried out:

```
class sample {
    private:
       int x;
       float y;
    public:
       virtual void display();
       int sum();
};
class derivedD: public baseA
{
    private:
       int x;
       float y;
    public:
       void display();   //virtual
       int sum();
};
int main()
{
    baseA *ptr;
    derivedD  objd;
    ptr = &objd;
    -------
    -------
    ptr-> display();  //run time binding
    ptr-> sum();     // compile time binding
    return 0;
}
```

The keyword virtual must be followed by a return type of a member function if a run time is to be bound. Otherwise the compile time binding will be effected as usual. In the above program segment, only the `display()` function has been declared as virtual in the base class, whereas the `sum()` is non-virtual.

Even though the message is given from the pointer of the base class to the objects of the derived class, it will not access the `sum()` function of the derived class as it has been declared as non-virtual. The `sum()` function compiles only the static binding.

PROGRAM 14.4

A program to demonstrate the run-time binding of the member functions of a class. The same message is given to access the derived class member functions from the array of pointers. As functions are declared as virtual, the C++ compiler invokes the dynamic binding.

```
//virtual function
//demonstration of run time binding using
//array of pointers
#include <iostream>
using namespace std;
class baseA {
    public:
        virtual  void display() {
            cout << " One \n";
        }
};

class derivedB : public baseA
{
    public:
        virtual  void display(){
            cout << " Two \n";
        }
};

class derivedC : public derivedB
{
    public:
        virtual  void display(){
            cout << " Three \n";
        }
};

int main()
{
    //define three objects
    baseA obja;
    derivedB objb;
    derivedC objc;
    baseA *ptr[3]; //define an array of pointers to baseA
    ptr[0] = &obja;
    ptr[1] = &objb;
    ptr[2] = &objc;
    for (int i = 0; i<=2; i++)
        ptr[i]->display(); //same message for all objects
    return 0;
}
```

Output of the above program

```
One
Two
Three
```

The function call through dynamic binding is shown in the following Fig. 14.5.

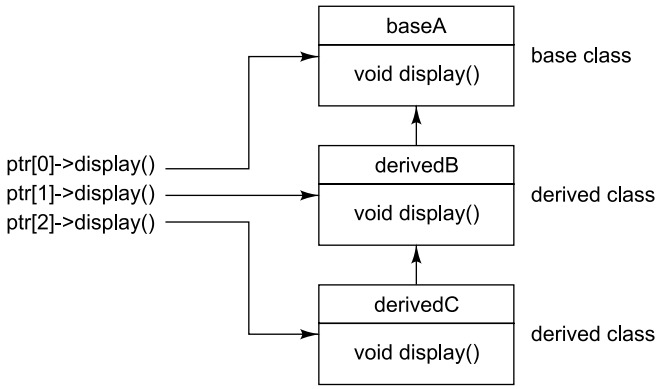


Fig. 14.5 *Function Call through Dynamic Binding*

PROGRAM 14.5

A program to illustrate the static binding of the member functions of a class.

```
//accessing member functions with pointers
#include <iostream>
using namespace std;
class base {
    private:
        int x;
        float y;
    public:
        void getdata();
        void display();
};
class derivedB : public base {
    private :
        int rollno;
        char name[20];
    public :
        void getdata();
        void display();
};

void base :: getdata()
{
    cout << " enter an integer \n";
    cin >> x;
    cout << " enter a real number \n";
    cin >> y;
}
void base :: display()
{
    cout << " entered numbers are x = " << x << " and y = " << y;
    cout << endl;
}

void derivedB :: getdata()
{
    cout << " enter roll number of a student ? \n";
    cin >> rollno;
    cout << " enter name of a student ? \n";
```

```
    cin >> name;
}

void derivedB:: display()
{
    cout << " Roll number           student's name \n";
    cout << rollno  << '\t' << name << endl;
}

int main()
{
    base *ptr;
    derivedB obj;
    ptr = &obj;
    ptr->getdata();
    ptr->display();
    return 0;
}
```

Output of the above program

```
enter an integer
10
enter a real number
2.2
entered numbers are x = 10 and y = 2.2
```

PROGRAM 14.6

A program to illustrate the dynamic binding of member functions of a class.

```
//accessing member functions with pointers
#include <iostream>
using namespace std;
class base {
    private:
        int x;
        float y;
    public:
        virtual void getdata();
        virtual void display();
};
class derivedB : public base {
    private:
        long int rollno;
        char name[20];
    public:
        void getdata();
        void display();
};

void base :: getdata()
{
    cout << " enter an integer \n";
    cin >> x;
    cout << " enter a real number \n";
    cin >> y;
}
void base :: display()
{
    cout << " entered numbers are x = " << x << " and y = " << y;
    cout << endl;
}
void derivedB :: getdata()
{
```

```
    cout << " enter roll number of a student ? \n";
    cin >> rollno;
    cout << " enter name of a student ? \n";
    cin >> name;
}

void derivedB :: display()
{
    cout << " Roll number   student's name \n";
    cout << rollno  << '\t' << name << endl;
}

int main()
{
    base *ptr;
    derivedB obj;
    ptr = &obj;
    ptr->getdata();
    ptr->display();
    return 0;
}
```

Output of the above program

```
enter roll number of a student?
20071
enter name of a student?
Suhail
Roll number      student's name
20071            Suhail
```

Virtual function with inline code substitution Though virtual functions can be declared as an `inline` code, being the run-time binding of the compiler, the inline code does not affect much of the programming efficiency. For inline code subtitution, the compiler must get information about the functions, like from where they have to be invoked etc. These must be defined during the compilation time.

The general syntax of the virtual function with inline code substitution is:

```
class base {
    private:
        -------
        -------
    public:
        virtual inline void getdata();
        virtual inline void display();
};
int main()
{
    base obja;
    obja->getdata();
    obja->display();
    return 0;
}
```

PROGRAM 14.7

A program to demonstrate how to define virtual functions with inline code substitution for run-time binding of the member functions of a class.

```
//accessing member functions with pointers
// virtual functions with inline code
#include <iostream>
using namespace std;
class base {
    private:
        int x;
        float y;
    public:
        virtual inline void getdata();
        virtual inline void display();
};
class derivedB: public base {
    private:
        long int rollno;
        char name[20];
    public:
        void getdata();
        void display();
};

inline void base :: getdata()
{
    cout << " enter an integer \n";
    cin >> x;
    cout << " enter a real number \n";
    cin >> y;
}
inline void base :: display()
{
    cout << " entered numbers are x = " << x << " and y = " << y;
    cout << endl;
}

void derivedB :: getdata()
{
    cout << " enter roll number of a student ? \n";
    cin >> rollno;
    cout << " enter name of a student ? \n";
    cin >> name;
}

void derivedB :: display()
{
    cout << " Roll number   student's name \n";
    cout << rollno  << '\t' << name << endl;
}

int main()
{
    base *ptr;
    derivedB obj;
    ptr = &obj;
    ptr->getdata();
    ptr->display();
    return 0;
}
```

Output of the above program

```
enter roll number of a student?
20072
enter name of a student?
Ahmed
Roll number      student's name
20072            Ahmed
```

PROGRAM 14.8

A program to access the members of the derived class objects through an array of pointers. In this program, only the base class member functions are preceded by the keyword virtual.

```
//accessing member functions with array of pointers
#include <iostream>
using namespace std;
class base {
    private:
        int x;
        float y;
    public:
        virtual void getdata();
        virtual void display();
};
class derivedB: public base {
    private:
        long int rollno;
        char name[20];
    public:
        void getdata();
        void display();
};

class derivedC: public base {
    private:
        float height;
        float weight;
    public:
        void getdata();
        void display();
};

void base :: getdata()
{
    cout << " enter an integer \n";
    cin >> x;
    cout << " enter a real number \n";
    cin >> y;
}
void base :: display()
{
    cout << " entered numbers are x = " << x << " and y = " << y;
    cout << endl;
}

void derivedB :: getdata()
{
    cout << " enter roll number of a student ? \n";
    cin >> rollno;
    cout << " enter name of a student ? \n";
    cin >> name;
}

void derivedB :: display()
{
    cout << " Roll number   student's name \n";
    cout << rollno  << '\t' << name << endl;
}

void derivedC :: getdata()
{
    cout << " enter height of student ? \n";
```

```
    cin >> height;
    cout << " enter weight of student ? \n";
    cin >> weight;
}

void derivedC :: display()
{
    cout << " Height and weight of the student's \n";
    cout << height  << '\t' << weight << endl;
}


int main()
{
    base *ptr[3];
    derivedB objb;
    derivedC objc;
    ptr[0] = &objb;
    ptr[1] = &objc;
    ptr[0]->getdata();
    ptr[1]->getdata();
    ptr[0]->display();
    ptr[1]->display();
    return 0;
}
```

Output of the above program

```
enter roll number of a student?
20073
enter name of a student?
Kandasamy
enter height of student?
173
enter weight of student?
78
Roll number   student's name
20073          Kandasamy
Height and weight of the student's
173     78
```

PROGRAM 14.9

A program to access the members of the derived class objects through an array of pointers. In this program, both the base class and the derived class member functions are preceded by the keyword virtual.

```
//accessing member functions with array of pointers
#include <iostream>
using namespace std;
class base {
    private:
        int x;
        float y;
    public:
        virtual void getdata();
        virtual void display();
};
class derivedB: public base {
    private:
        long int rollno;
        char name[20];
    public:
```

```
        virtual void getdata();
        virtual void display();
};

class derivedC: public base {
    private:
        float height;
        float weight;
    public:
        virtual void getdata();
        virtual void display();
};

void base :: getdata()
{
    cout << " enter an integer \n";
    cin >> x;
    cout << " enter a real number \n";
    cin >> y;
}
void base :: display()
{
    cout << " entered numbers are x = " << x << " and y = " << y;
    cout << endl;
}

void derivedB :: getdata()
{
    cout << " enter roll number of a student ? \n";
    cin >> rollno;
    cout << " enter name of a student ? \n";
    cin >> name;
}

void derivedB :: display()
{
    cout << " Roll number   student's name \n";
    cout << rollno  << '\t' << name << endl;
}

void derivedC :: getdata()
{
    cout << " enter height of student ? \n";
    cin >> height;
    cout << " enter weight of student ? \n";
    cin >> weight;
}

void derivedC :: display()
{
    cout << " Height and weight of the student's \n";
    cout << height  << '\t' << weight << endl;
}

int main()
{
    base *ptr[3];
    derivedB objb;
    derivedC objc;
    ptr[0] = &objb;
    ptr[1] = &objc;
    ptr[0]->getdata();
    ptr[1]->getdata();
    ptr[0]->display();
    ptr[1]->display();
    return 0;
}
```

Output of the above program

```
enter roll number of a student?
20074
enter name of a student?
Suhail
enter height of student?
167
enter weight of student?
87.5
Roll number   student's name
20074              Suhail
Height and weight of the student's
167     87.5
```

14.6 PURE VIRTUAL FUNCTIONS

A pure virtual function is a type of function which has only a function declaration. It does not have the function definition. The following program segment illustrates how to declare a pure virtual function:

Case 1

```
//pure virtual functions
#include <iostream>
using namespace std;
class base {
    private:
       int x;
       float y;
    public:
       virtual void getdata();
       virtual void display();
};
class derivedB: public base {
    -------
    -------
};
void base :: getdata()  //pure virtual function definition
{
    // empty statements
}
void base :: display() //pure virtual function definition
{
     // empty statements
}
```

The syntax diagram of the pure virtual method declaration is given in Fig. 14.6.

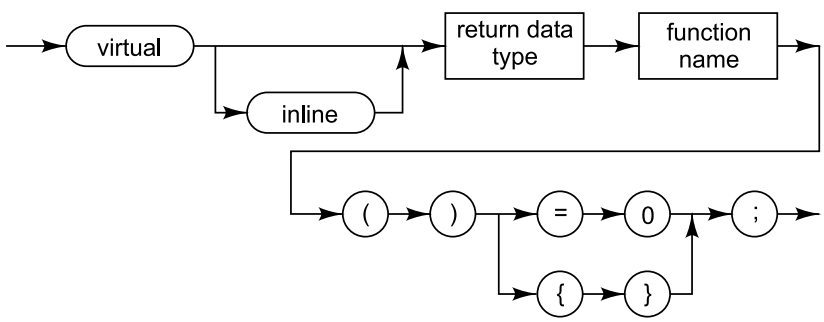


Fig. 14.6 *Syntax Diagram of Pure Virtual Method Declaration*

PROGRAM 14.10

A program to demonstrate how a pure virtual function is defined, declared and invoked from the object of a derived class through the pointer of the base class.

```
//pure virtual functions
#include <iostream>
using namespace std;
class base {
    private:
        int x;
        float y;
    public:
        virtual  void getdata();
        virtual  void display();
};
class derivedB: public base {
    private:
        long int rollno;
        char name[20];
    public:
        void getdata();
        void display();
};
void base :: getdata() { }  //pure virtual function
void base :: display() { }  //pure virtual function

void derivedB :: getdata()
{
    cout << " enter roll number of a student ? \n";
    cin >> rollno;
    cout << " enter name of a student ? \n";
    cin >> name;
}
void derivedB :: display()
{
    cout << " Roll number   student's name \n";
    cout << rollno  << '\t' << name << endl;
}
int main()
{
    base *ptr;
    derivedB obj;
    ptr = &obj;
    ptr->getdata();
    ptr->display();
    return 0;
}
```

Output of the above program

```
enter roll number of a student?
20075
enter name of a student?
Antony
Roll number   student's name
20075               Antony
```

Case 2 A pure virtual function can also have the following format, when a virtual function is declared within the class declaration itself. The virtual function may be equated to zero if it does not have a function definition.

```
//pure virtual functions
#include <iostream> using namespace std;
  class base {
      private:
         int x;
         float y;
      public:
         virtual void getdata() = 0;
         virtual void display() = 0;
  };
  class derivedB: public base {
       -------
       -------
  };
```

PROGRAM 14.11

A program to illustrate how to declare a pure virtual function and equate it to zero as it does not have any function parts.

```
//pure virtual functions
#include <iostream>
using namespace std;
class base {
    private:
        int x;
        float y;
    public:
        virtual inline void getdata() = 0;
        virtual inline void display() = 0;
};
class derivedB: public base {
    private:
        long int rollno;
        char name[20];
    public:
        void getdata();
        void display();
};
void derivedB :: getdata()
{
    cout << " enter roll number of a student ? \n";
    cin >> rollno;
    cout << " enter name of a student ? \n";
    cin >> name;
}
void derivedB :: display()
{
    cout << " Roll number   student's name \n";
    cout << rollno  << '\t' << name << endl;
}
int main()
{
    base *ptr;
    derivedB obj;
    ptr = &obj;
    ptr->getdata();
    ptr->display();
    return 0;
}
```

Output of the above program

```
enter roll number of a student?
20076
enter name of a student?
Ratanakumar
Roll number  student's name
20076        Ratanakumar
```

When an object of the derived class tries to access through the pointer of the base class members, the function invoking message will reach only the derived class members but not to the base class members as the base class member function may not have a function definition.

14.7 ABSTRACT BASE CLASSES

A class which consists of pure virtual functions is called an abstract base class. In the previous section, it has been discussed that a function may be defined without any statement or the function declaration may be equated to zero if it does not have the function definition part.

PROGRAM 14.12

A program to demonstrate how to define an abstract base class with pure virtual functions in which the function definition part has been defined without any statement. The members of the derived class objects are accessed through the base class objects through pointer technique.

```
//demonstration of abstract base classes
#include <iostream>
using namespace std;
class base {
    private:
        int x;
        float y;
    public:
        virtual void getdata();
        virtual void display();
};
class derivedB: public base {
    private:
        long int rollno;
        char name[20];
    public:
        void getdata();
        void display();
};
class derivedC: public base {
    private:
        float height;
        float weight;
    public:
        void getdata();
        void display();
};
void base :: getdata() {}  //pure virtual function
void base :: display() {} // pure virtual function
void derivedB :: getdata()
{
    cout << " enter a roll number of student ? \n";
    cin >> rollno;
```

```
    cout << " enter a name of student ? \n";
    cin >> name;
}
void derivedB :: display()
{
    cout << " Roll number   student's name \n";
    cout << rollno  << '\t' << name << endl;
}
void derivedC :: getdata()
{
    cout << " enter height of student ? \n";
    cin >> height;
    cout << " enter weight of student ? \n";
    cin >> weight;
}
void derivedC :: display()
{
    cout << " Height and weight of the student's \n";
    cout << height  << '\t' << weight << endl;
}
int main()
{
    base *ptr[3];
    derivedB objb;
    derivedC objc;
    ptr[0] = &objb;
    ptr[1] = &objc;
    ptr[0]->getdata();
    ptr[1]->getdata();
    ptr[0]->display();
    ptr[1]->display();
    return 0;
}
```

Output of the above program

```
enter a roll number of student?
20075
enter a name of student?
Ramu
enter height of student?
167.56
enter weight of student?
64
Roll number   student's name
20075              Ramu
Height and weight of the student's
167.56       64
```

PROGRAM 14.13

A program to demonstrate how to define an abstract base class with pure virtual functions in which the function declaration is equated to zero as it does not have the function definition part.

```
//demonstration of abstract base classes
//case -2
#include <iostream>
using namespace std;
class base {
    private:
        int x;
```

```
        float y;
    public:
        virtual inline void getdata() = 0;
        virtual inline void display() = 0;
};
class derivedB: public base {
    private:
        long int rollno;
        char name[20];
    public:
        void getdata();
        void display();
};
class derivedC: public base {
    private:
        float height;
        float weight;
    public:
        void getdata();
        void display();
};
void derivedB :: getdata()
{
    cout << " enter a roll number of student ? \n";
    cin >> rollno;
    cout << " enter a name of student ? \n";
    cin >> name;
}
void derivedB :: display()
{
    cout << " Roll number   student's name \n";
    cout << rollno  << '\t' << name << endl;
}
void derivedC :: getdata()
{
    cout << " enter height of student ? \n";
    cin >> height;
    cout << " enter weight of student ? \n";
    cin >> weight;
}
void derivedC :: display()
{
    cout << " Height and weight of the student's \n";
    cout << height  << '\t' << weight << endl;
}
int main()
{
    base *ptr[3];
    derivedB objb;
    derivedC objc;
    ptr[0] = &objb;
    ptr[1] = &objc;
    ptr[0]->getdata();
    ptr[1]->getdata();
    ptr[0]->display();
    ptr[1]->display();
    return 0;
}
```

Output of the above program

```
enter a roll number of student?
20071
enter a name of student?
Velusamy.K
enter height of student?
```

```
175.6
enter weight of student?
78
Roll number   student's name
20071           Velusamy.K
Height and weight of the student's
175.6     78
```

14.8 CONSTRUCTORS UNDER INHERITANCE

It has already been pointed out that whenever an object of a class is created, a constructor member function is invoked automatically and when an object of the derived class is created, the constructor for that object is called. This is due to the object of the derived class which contains the members of the base class also. Since the base class is also part of the derived class, it is not logical to call the constructors of the base class.

The firing order of constructors under inheritance is shown in Fig. 14.7.

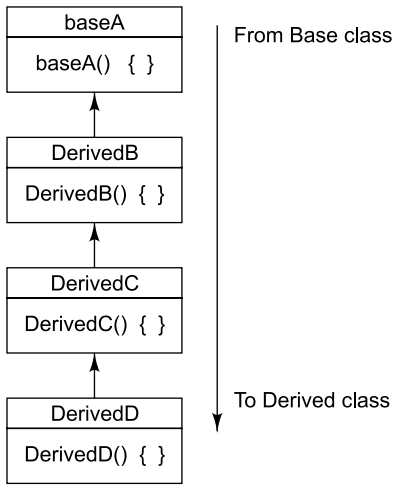


Fig. 14.7 *Firing Order of Constructors under Inheritance*

PROGRAM 14.14

A program to demonstrate how to define and declare a constructor member function in the base class as well as in the derived class under the inheritance mechanism.

```
//constructors under inheritance
#include <iostream>
using namespace std;
class baseA {
    public:
       baseA(); //constructor
};
class derivedB: public baseA {
    public :
```

```
        derivedB(); //constructor
};
baseA :: baseA()
{
    cout << " base class constructor \n";
}
derivedB :: derivedB()
{
    cout << " derived class constructor \n";
}
int main()
{
    derivedB objb;
    return 0;
}
```

Output of the above program

```
base class constructor
derived class constructor
```

The above program consists of two constructors `baseA()` and `derivedB()`. Whenever an object objb is created, the constructors of the base classes are automatically invoked. C++ fires the constructor from the lowest to the highest class. Before creating `derivedB()`, C++ executes `baseA()`. The constructor of base class is therefore executed before the constructor of the derived class.

PROGRAM 14.15

A program to illustrate how to define and declare a constructor member function under multiple inheritance technique.

```
//constructors under inheritance
#include <iostream>
using namespace std;
class baseA {
    public:
        baseA(); //constructor
};
class derivedB: public baseA {
    public:
        derivedB(); //constructor
};
class derivedC: public derivedB {
    public :
        derivedC(); //constructor
};
class derivedD: public derivedC{
    public:
        derivedD(); //constructor
};
baseA :: baseA()
{
    cout << " base class\n";
}
derivedB :: derivedB()
{
    cout << " derivedB class \n";

}
derivedC :: derivedC()
{
    cout << " derivedC class \n";
```

```
}
derivedD :: derivedD()
{
    cout << " derivedD class \n";
}
int main()
{
    derivedD objd;
    return 0;
}
```

Output of the above program

```
base class
derivedB class
derivedC class
derivedD class
```

14.9 DESTRUCTORS UNDER INHERITANCE

It has been seen that destructor is a special member function. It is invoked automatically to free the memory space which was allocated by the constructor functions. Whenever an object of the class is getting destroyed, the destructors are used to free the heap area so that the free memory space may be used subsequently. In the previous section, it has been demonstrated that constructors in an inheritance hierarchy fire from a base class to a derived class. Destructors in an inheritance hierarchy fire from a derived class to a base class order, i.e., they fire in the reverse order of that of the constructors.

The firing order of destructors under inheritance is shown in Fig. 14.8.

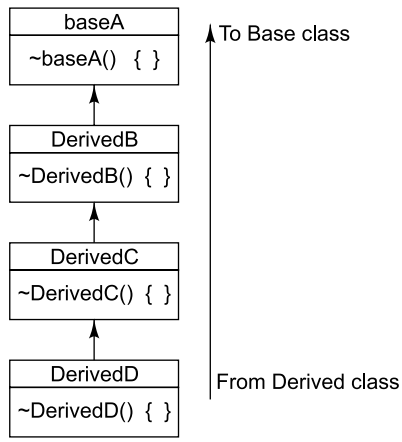


Fig. 14.8 *Firing Order of Destructors under Inheritance*

PROGRAM 14.16

A program to demonstrate how the destructor member function gets fired from the derived class objects to the base class objects through pointers.

```
//destructors under inheritance
#include <iostream>
using namespace std;
class baseA {
    public:
       ~baseA(); //destructor
};
class derivedB: public baseA {
    public:
       ~derivedB(); //destructor
};
baseA :: ~baseA()
{
```

```
    cout << " base class  destructor\n";
}
derivedB :: ~derivedB()
{
    cout << " derivedB class destructor \n";
}
int main()
{
    derivedB objb;
    return 0;
}
```

Output of the above program

```
derivedB class destructor
base class  destructor
```

PROGRAM 14.17

A program to demonstrate how to define, declare and invoke the destructor member functions under multiple inheritance.

```
//destructors under inheritance
#include <iostream>
using namespace std;
class baseA {
    public :
       ~baseA(); //destructor
};
class derivedB: public baseA {
    public:
       ~derivedB(); //destructor
};
class derivedC: public derivedB {
    public:
       ~derivedC(); //destructor
};
class derivedD: public derivedC{
    public:
       ~derivedD(); //destructor
};
baseA :: ~baseA()
{
    cout << " base class\n";
}
derivedB :: ~derivedB()
{
    cout << " derivedB class \n";
}
derivedC :: ~derivedC()
{
    cout << " derivedC class \n";
}
derivedD :: ~derivedD()
{
    cout << " derivedD class \n";
}
int main()
{
    derivedD objd;
    return 0;
}
```

Output of the above program

```
derivedD class
derivedC class
derivedB class
base class
```

PROGRAM 14.18

A program to display the message of both constructors and destructors of a base class and a derived class.

```
//destructor under inheritance
#include <iostream>
using namespace std;
class baseA {
    public:
       baseA() { //baseA's constructor
           cout << " base class constructor\n";
       }
       ~baseA() { //baseA's destructor
           cout << " base class destructor\n";
       }
};
class derivedD: public baseA {
    public:
       derivedD() { //derivedD's constructor
           cout << " derived class constructor\n";
       }
       ~derivedD() { //derivedD's destructor
           cout << " derived class destructor\n";
       }
};
int main()
{
    derivedD obj;
    return 0;
}
```

Output of the above program

```
base class constructor
derived class constructor
derived class destructor
base class destructor
```

A destructor's main job is to free the storage a computer dynamically allocates. By firing in the reverse order of constructors, destructors ensure that the most recently allocated storage is the first storage to be freed. The following program explains how destructror member functions free the memory space which was allocated by the new operator.

PROGRAM 14.19

A program to allocate the heap memory area using a constructor member function and to free the memory storage by the destructors under multiple inheritance system.

```
// destructor member function under inheritance
#include <iostream>
using namespace std;
class baseA {
```

```
    private:
       char *ptrbase;
    public:
       baseA();
       ~baseA();
};
class derivedD: public baseA {
    private:
       char *ptrderived;
    public:
       derivedD();
       ~derivedD();
};
baseA:: baseA()  //baseA's constructor
{
    ptrbase = new char[5];
    cout << " Base class allocates 5 bytes \n";
}
baseA :: ~baseA()  //baseA's destructor
{
    delete[] ptrbase;
    cout << " base class frees 5 bytes \n";
}
derivedD::derivedD()  //derivedD's constructor
{
    ptrderived = new char[100];
    cout << " Derived class allocates 100 bytes \n";
}
derivedD :: ~derivedD()
{
    delete[] ptrderived;
    cout << " Derived class frees 100 bytes \n";
}
int main()
{
    baseA *ptr = new derivedD;
    delete ptr;
    return 0;
}
```

Output of the above program

```
Base class allocates 5 bytes
Derived class allocates 100 bytes
base class frees 5 bytes
```

14.10 VIRTUAL DESTRUCTORS

It is known that the destructor member function is invoked to free the memory storage by the C++ compiler automatically. But the destructor member function of the derived class is not invoked to free the memory storage which was allocated by the constructor member function of the derived class. It is because the destructor member functions are non-virtual and the message will not reach the destructor member functions under late binding. So it is better to have a destructor member function as virtual and the virtual destructors are essential in a program to free the memory space effectively under late binding method.

The following program segment illustrates how to define a virtual destructor in a program.

```
// virtual destructor
class baseA {
    public:
       baseA();
                 //constructor cannot have virtual
```

```
        virtual ~baseA();
};
class derivedD: public baseA {
    -------
    -------
};
baseA :: ~baseA()  //baseA's destructor
{
    delete[] ptrbase;
    cout << " base class frees 5 bytes \n";
}
int main()
{
    baseA *ptr = new derivedD;
    delete ptr;
}
```

Note that whenever instances are created at run time on the heap through the new operator, constructor member functions are called automatically. When the delete operator is used, the destructors are automatically called to release the space occupied by the instance itself. As a derived class instance always contains a base class instance, it is necessary to invoke destructors of both the classes in order to ensure that all the space on the heap is released.

PROGRAM 14.20

A program to illustrate how a block of memory space which are allocated by new operators and released when destructor member functions are called.

```
// virtual destructor
#include <iostream>
using namespace std;
class baseA {
    private:
        char *ptrbase;
    public:
        baseA();          //constructor cannot have virtual
        virtual ~baseA();
};
class derivedD: public baseA {
    private:
        char *ptrderived;
    public:
        derivedD();
        ~derivedD();
};
baseA:: baseA()  //baseA's constructor
{
    ptrbase = new char[5];
    cout << " Base class allocates 5 bytes \n";
}
baseA :: ~baseA()  //baseA's destructor
{
    delete[] ptrbase;
    cout << " base class frees 5 bytes \n";
}
derivedD::derivedD()  //derivedD's constructor
{
    ptrderived = new char[100];
```

```
    cout << " Derived class allocates 100 bytes \n";
}
derivedD :: ~derivedD()
{
    delete[] ptrderived;
    cout << " Derived class frees 100 bytes \n";
}
int main()
{
    baseA *ptr = new derivedD;
    delete ptr;
    return 0;
}
```

Output of the above program

```
Base class allocates 5 bytes
Derived class allocates 100 bytes
Derived class frees 100 bytes
base class frees 5 bytes
```

PROGRAM 14.21

A program to demonstrate how to define, declare and invoke the virtual destructor member function in multiple inheritance using the polymorphic technique.

```
//non-virtual destructor function
#include <iostream>
using namespace std;
class base {
    public:
       void display();
       ~base();
};
class derived: public base
{
    public:
       void display();
       ~derived();
};
void base :: display()
{
    cout << "  Base class member function";
    cout << endl;
}
base :: ~base()
{
    cout << " base class destructor is called ";
    cout << endl;
}
void derived :: display()
{
    cout << "  Derived class member function";
    cout << endl;
}
derived :: ~derived()
{
    cout << " derived class destructor is called ";
    cout << endl;
}
int main()
{
    base *ptr = new derived;
```

```
    ptr->display(); // same method is called for both base class
                    // and derived class
    delete ptr;
    return 0;
}
```

Output of the above program

```
Base class member function
base class destructor is called
```

PROGRAM 14.22

A program to illustrate how to define and declare a class which consists of both virtual members and virtual destructors under multiple inheritance.

```
//virtual destructor function
#include <iostream>
using namespace std;
class base {
    public:
       virtual void display ();
       virtual ~base();        //destructor member function
};
class derived: public base
{
    public:
       virtual void display ();
       virtual ~derived();    //destructor member function
};
void base :: display()
{
    cout << "  Base class member function";
    cout << endl;
}
base :: ~base()
{
    cout << " base class destructor is called ";
    cout << endl;
}
void derived :: display()
{
    cout << "  Derived class member function";
    cout << endl;
}
derived :: ~derived()
{
    cout << " derived class destructor is called ";
    cout << endl;
}
int main()
{
    base *ptr = new derived;
    ptr->display();  // same method is called for both base class
                     // and derived class
    delete ptr;
    return 0;
}
```

Output of the above program

```
Derived class member function
derived class destructor is called
base class destructor is called
```

14.11 VIRTUAL BASE CLASSES

We have already learned in Chapter 12 on "Single and Multiple Inheritance" that inheritance is a process of creating a new class which is derived from more than one base classes. Multiple inheritance hierarchies can be complex, which may lead to a situation in which a derived class inherits multiple times from the same indirect base class.

For example, the following program segment illustrates how a base class can be derived twice from the derived class in different way.

```
class baseA {
    protected:
        int x;
        -------
        -------
};
class derivedB: public baseA {  //path 1, through derivedB
    protected:
        -------
        -------
};
class derivedD: public baseA { //path 2, through derivedD
    protected:
        -------
        -------
};
class abc : public derivedB,public derivedD
{
        //the data member x comes twice
        -------
        -------
};
```

The data member `x` is inherited twice in the derived class abc, once through the derived class `derivedB` and again through `derivedC`. This is wasteful and confusing.

The multiple repetition of base class under multiple inheritance is shown in Fig. 14.9.

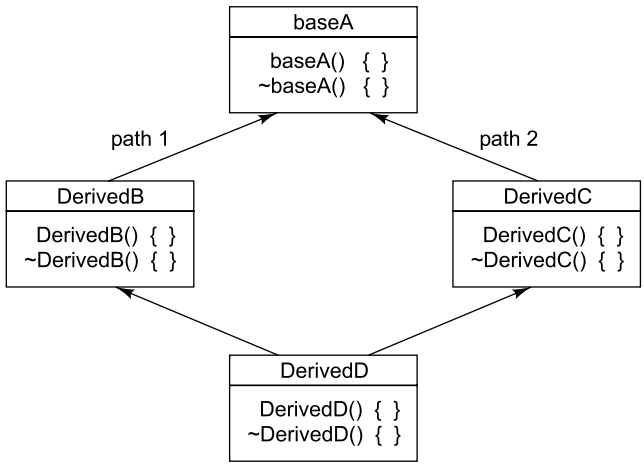


Fig. 14.9 *Repetition of Base Class under Multiple Inheritance*

The above multiple repetition of the data member can be corrected by changing the derived class `DerivedB` and `DerivedD` into virtual base classes. Any base class which is declared using the keyword virtual is called a virtual base class. Virtual base classes are useful method to avoid unnecessary repetition of the same data member in the multiple inheritance hierarchies. For example, the following program segment illustrates how a base class is derived only once from the derived classes via virtual base classes.

```
class baseA {
    protected:
         int x;
         -------
         -------
};
class derivedB: public virtual baseA {  //path 1, through derivedB
    protected:
         -------
         -------
};
class derivedD: public virtual baseA { //path 2, through derivedD
    protected:
         -------
         -------
};
class abc : public derivedB,public derivedD
{
         //the data member x comes only once
         -------
         -------
};
```

By making `derivedB` and `derivedD` into virtual base classes for `abc`, a copy of the data member *x* is available only once.

A class may be both an ordinary and a virtual base in the same inheritance structure. For example, the following program segment illustrates how to define a derived class with virtual and nonvirtual class objects.

```
class baseA {
    protected:
         int x;
         -------
         -------

};
class derivedB: public virtual baseA {  //path 1, through derivedB
    protected:
         -------
         -------
};
class derivedC : public virtual baseA { //path 2, through derivedD
    protected:
         -------
         -------
};
class derivedD : public derivedB,public derivedD
{
```

```
        protected:
              -------
              -------
    };
    class derivedE : public baseA
    {
        protected:
              -------
              -------
    };
    class abc : public derivedE,public derivedD
    {
              -------
              -------
    };
```

From the above illustration, it can be inferred that an object of derived class `abc` will contain two baseA class objects, one virtual and another nonvirtual.

PROGRAM 14.23

A program to define multiple derived classes which are accessing the same base class data members through indirect base references.

```
//using nonvirtual base classes
#include <iostream>
using namespace std;
class baseA {
    protected:
        int x;
    public:
        void getdata();
        void display();
};
class derivedB : public baseA {  //path 1, through derivedB
    protected:
        float y;
    public:
        void getdata();
        void display();
};
class derivedD : public baseA { //path 2, through derivedD
    protected:
        char name[20];
    public:
        void getdata();
        void display();
};
class abc : public derivedB,public derivedD
{
    public:
        void getdata();
        void display();
};
void baseA :: getdata()
{
    cout << " enter an integer \n";
    cin >> x;
}
void baseA :: display()
```

```
{
    cout << " integer : " << x << endl;
}
void derivedB :: getdata()
{
    baseA::getdata();
    cout << " enter a floating point value \n";
    cin >> y;
}
void derivedB :: display()
{
    baseA :: display();
    cout << " real number :" << y << endl;
}
void derivedD :: getdata()
{
    baseA :: getdata();
    cout << " enter a string \n";
    cin >> name;
}
void derivedD :: display()
{
    baseA:: display();
    cout << " string : " << name << endl;
}
void abc:: getdata()
{
    derivedB :: getdata();
    derivedD :: getdata();
}
void abc:: display()
{
    derivedB :: display();
    derivedD :: display();
}
int main()
{
    abc obj;
    obj.getdata();
    obj.display();
    return 0;

}
```

Output of the above program

```
enter an integer
10
enter a floating point value
-2.2
enter an integer
20
enter a string
C++, world

integer : 10
real number :-2.2
integer : 20
string : C++,world
```

The protected data member `x` is accessed twice by the two derived classes through `derivedB` and `derivedD`. The content of the data variable is copied into two memory variables in the heap. It is certainly a repetition with different data and confusing if it is intended to process the same item of the base class.

PROGRAM 14.24

A program to define multiple derived classes which is accessing the same base class data members through indirect base references using virtual base class.

```
//using virtual base classes
#include <iostream>
using namespace std;
class baseA {
    protected:
        int x;
    public:
        void getdata();
        void display();
};
class derivedB : public virtual baseA {  //path 1, through derivedB
    protected:
        float y;
    public:
        void getdata();
        void display();
};
class derivedD : public virtual baseA { //path 2, through derivedD
    protected:
        char name[20];
    public:
        void getdata();
        void display();
};
class abc : public derivedB,public derivedD
{
    public:
        void getdata();
        void display();
};
void baseA :: getdata()
{
    cout << " enter an integer \n";
    cin >> x;
}
void baseA :: display()
{
    cout << " integer : " << x << endl;
}
void derivedB :: getdata()
{
    baseA::getdata();
    cout << " enter a floating point value \n";
    cin >> y;
}
void derivedB :: display()
{
    baseA :: display();
    cout << " real number :" << y << endl;
}
void derivedD :: getdata()
{
    baseA :: getdata();
    cout << " enter a string \n";
    cin >> name;
}
void derivedD :: display()
{
```

```
    baseA:: display();
    cout << " string : " << name << endl;
}
void abc:: getdata()
{
    derivedB :: getdata();
    derivedD :: getdata();
}
void abc:: display()
{
    derivedB :: display();
    derivedD :: display();
}
int main()
{
    abc obj;
    obj.getdata();
    obj.display();
    return 0;
}
```

Output of the above program

```
enter an integer
20
enter a floating point value
3.3
enter an integer
55
enter a string
Hello, C++

integer: 55
real number: 3.3
integer: 55
string: Hello,C++
```

Even though the protected data member `x` is accessed by two derived classes through `derivedB` and `derivedD`, the content of the variable `x` is the same.

REVIEW QUESTIONS

1. What is polymorphism? List the pros and cons of using polymorphism in object-oriented programming.
2. In what way a polymorphic technique increases the efficiency of the software design in the modern world?
3. Explain the concept of static binding, compile linkage or early binding.
4. Explain the technique of dynamic binding, run-time linkage or dynamic binding.
5. What is a virtual function and what are the advantages of declaring a virtual function in a program?
6. What are the syntactic rules to be observed while for defining the keyword virtual?
7. Explain how polymorphism can be incorporated using the keyword virtual.
8. Explain how the message call is given to member functions of the derived classes.
9. Explain the merits and demerits of the run-time binding over the compile time binding.
10. Explain how the data members of a base class can be initialised under multiple inheritance.

11. Explain how the data members of a derived class can be initialised under multiple inheritance.
12. Explain why a constructor member function cannot be a virtual method (member function).
13. What are the pros and cons of declaring destructor member functions under multiple inheritance?
14. Explain how a destructor member function can be invoked in the derived class from the base class objects.
15. What is the firing order of the constructors under multiple inheritance?
16. What is the firing order of the destructors under multiple inheritance?
17. What is a virtual destructor? Give a few of its applications in real-time applications.
18. What is the use of declaring the virtual destructor under multiple inheritance?
19. Explain how the virtual base class is different from the conventional base classes of the OOPs.
20. Explain the syntactic rules of the virtual base class in C++.
21. What is an abstract base class and what are the advantages of using it in a program?
22. What is a pure virtual function? What are the merits and demerits of defining and declaring a pure virtual function in a program?
23. Explain the various techniques of defining a pure virtual function.

CONCEPT REVIEW PROBLEMS

1. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
class baseA {
   public:
      int a;
      void setdata();
      void display();
};
class derivedB : baseA
{
   public:
      int b;
      void setdata();
      void display();
};
void baseA :: setdata()
{
      a = 10;
}
void baseA :: display()
{
      cout << "++a = " << ++a;
      cout << endl;
}

void derivedB :: setdata()
{
      b = 20;
}
```

```
void derivedB :: display()
{
      cout << "++b = " << ++b;
      cout << endl;
}
int main()
{
   derivedB obj;
   obj.setdata();
   obj.display();
   return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
class baseA {
   public:
      int a;
      virtual void setdata();
      virtual void display();
};
class derivedB : baseA
{
   public:
      int b;
      virtual void setdata();
      virtual void display();
};
void baseA :: setdata()
{
      a = 10;
}
void baseA :: display()
{
      cout << "++a = " << ++a;
      cout << endl;
}

void derivedB :: setdata()
{
      b = 20;
}
void derivedB :: display()
{
      cout << "++b = " << ++b;
      cout << endl;
}
int main()
{
   derivedB obj;
   obj.setdata();
   obj.display();
   return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
class baseA {
   public:
      int a;
      virtual void setdata();
      virtual void display();
};
class derivedB : public baseA
{
   public:
      int b;
      void setdata();
      void display();
};
void baseA :: setdata()
{
      a = 10;
}
void baseA :: display()
{
      cout << "++a = " << ++a;
      cout << endl;
}
void derivedB :: setdata()
{
      b = 20;
}
void derivedB :: display()
{
      cout << "++b = " << ++b;
      cout << endl;
}
int main()
{
   baseA *ptr;
   derivedB obj;
   ptr = &obj;
   ptr->setdata();
   ptr->display();
   return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
class baseA {
   public:
      int a;
      void setdata();
      void display();
};
class derivedB : public baseA
{
```

```
   public:
      int b;
      void setdata();
      void display();
};
void baseA :: setdata()
{
      a = 10;
}
void baseA :: display()
{
      cout << "++a = " << ++a;
      cout << endl;
}

void derivedB :: setdata()
{
      b = 20;
}
void derivedB :: display()
{
      cout << "++b = " << ++b;
      cout << endl;
}
int main()
{
   baseA *ptr;
   derivedB obj;
   ptr = &obj;
   ptr->setdata();
   ptr->display();
   return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
struct A {
    void display()
    {
         cout << "A" << endl;
    }
};
struct B : A
{
    void display()
    {
         cout << "B" << endl;
    }
};

struct C : B
{
    void display()
    {
```

```
            cout << "C" << endl;
        }
};

struct D : C
{
        void display()
        {
            cout << "D" << endl;
        }
};
int main()
{
        A objA;
        B objB;
        C objC;
        D objD;
        A *ptr[3];
        ptr[0] = &objA;
        ptr[1] = &objB;
        ptr[2] = &objC;
        ptr[3] = &objD;
        for (int i = 0; i <= 3; ++i) {
            ptr[i]->display();
        }
        return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
struct A {
        virtual void display()
        {
            cout << "A" << endl;
        }
};
struct B : A
{
        virtual void display()
        {
            cout << "B" << endl;
        }
};

struct C : B
{
        virtual void display()
        {
            cout << "C" << endl;
        }
};

struct D : C
{
```

```
    virtual void display()
    {
        cout << "D" << endl;
    }
};
int main()
{
    A objA;
    B objB;
    C objC;
    D objD;
    A *ptr[3];
    ptr[0] = &objA;
    ptr[1] = &objB;
    ptr[2] = &objC;
    ptr[3] = &objD;
    for (int i = 0; i <= 3; ++i) {
        ptr[i]->display();
    }
    return 0;
}
```

2. What will be the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
struct A {
   A()
   {
      cout << "A" << endl;
   }
};
struct B : A
{
   B()
   {
      cout << "B" << endl;
   }
};

struct C : B
{
   C()
   {
      cout << "C" << endl;
   }
};

struct D : C
{
   D()
   {
      cout << "D" << endl;
   }
};
```

```
int main()
{
   D obj;
   return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
struct A {
   ~A()
   {
      cout << "~A" << endl;
   }
};
struct B : A
{
   ~B()
   {
      cout << "~B" << endl;
   }
};

struct C : B
{
   ~C()
   {
      cout << "~C" << endl;
   }
};

struct D : C
{
   ~D()
   {
      cout << "~D" << endl;
   }
};
int main()
{
   D obj;
   return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
struct A {
    A();
   ~A();
};
struct B : A
{
    B();
   ~B();
};
```

```
struct C : B
{
    C();
   ~C();
};

struct D : C
{
    D();
   ~D();
};

A :: A()
{
    cout << "A" << endl;
}
A :: ~A()
{
    cout << "~A" << endl;
}
B :: B()
{
    cout << "B" << endl;
}
B :: ~B()
{
    cout << "~B" << endl;
}
C :: C()
{
    cout << "C" << endl;
}
C :: ~C()
{
    cout << "~C" << endl;
}
D :: D()
{
    cout << "D" << endl;
}
D :: ~D()
{
    cout << "~D" << endl;
}
int main()
{
    D obj;
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
class A {
```

```
    public:
        void display()
        {
            cout << "A" << endl;
        }
};
class B : public A
{
    public:
        void display()
        {
            A::display();
            cout << "B" << endl;
        }
};
class C : public A
{
    public:
        void display()
        {
            A::display();
            cout << "C" << endl;
        }
};
class D : public B, public C
{
    public:
        void display()
        {
            B::display();
            C::display();
            cout << "D" << endl;
        }
};
int main()
{
    D *obj;
    obj->display();
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
class A {
   public:
      virtual void display()
      {
         cout << "A" << endl;
      }
};
class B : public virtual A
{
   public:
      void display()
```

```
        {
            A::display();
            cout << "B" << endl;
        }
};
class C : public virtual A
{
    public:
        void display()
        {
            A::display();
            cout << "C" << endl;
        }
};
class D : public B, public C
{
    public:
        void display()
        {
            B::display();
            C::display();
            cout << "D" << endl;
        }
};
int main()
{
    D obj;
    obj.display();
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
struct A {
    virtual A();
    virtual ~A();
};
struct B : A
{
    virtual B();
    virtual ~B();
};
A :: A()
{
    cout << "A" << endl;
}
A :: ~A()
{
    cout << "~A" << endl;
}
B :: B()
{
    cout << "B" << endl;
}
B :: ~B()
```

```
{
    cout << "~B" << endl;
}
int main()
{
    B obj;
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
struct A {
    inline A();
    inline virtual ~A();
};
struct B : A
{
    inline B();
    inline virtual ~B();
};
A :: A()
{
    cout << "A" << endl;
}
A :: ~A()
{
    cout << "~A" << endl;
}
B :: B()
{
    cout << "B" << endl;
}
B :: ~B()
{
    cout << "~B" << endl;
}
int main()
{
    B obj;
    return 0;
}
```

(h)

```
#include <iostream>
using namespace std;
struct A {
    inline A();
    inline virtual ~A();
};
struct B : virtual A
{
    inline B();
    inline virtual ~B();
};
A :: A()
{
    cout << "A" << endl;
}
```

```
A :: ~A()
{
    cout << "~A" << endl;
}
B :: B()
{
    cout << "B" << endl;
}
B :: ~B()
{
    cout << "~B" << endl;
}
int main()
{
    B obj;
    return 0;
}
```

PROGRAMMING EXERCISES

1. Write an object-oriented program in C++ using polymorphic technique that prints either the number and its square or the number and its cube from 0 to 100.
2. Modify the above program so that it prints the number, square and cubes of only even numbers from 0 to 100 with the same output.
3. Write a program in C++ to generate the following series of numbers using polymorphism.

(i)

1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1
7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1

(ii)

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9

(iii)

9 8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
7 6 5 4 3 2 1
6 5 4 3 2 1
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1

(iv)

1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

4. Write a program in C++ that prints the factorial of a given number using dynamic binding.
5. Write a program in C++ that determines whether a given number is a prime number or not and then prints the result, using polymorphism.

   (Hint: Prime number is a number which is divisible only by 1 and by itself. 3 is a prime number since it is divisible by 1 and 3. Whereas 6 is not a prime number as it is divisible by 1, 2 and 3.)
6. Write a program in C++ to solve the general quadratic equation.

   $ax^2 + bx + c = 0$

   using the polymorphic technique.
7. Write a program in C++ to generate a Fibonacci series of 'n' numbers (where n is defined by the user) using run time binding.

   (The series should be: 1 1 2 3 5 8 13 21 32, and so on.)
8. (a) Develop an object-oriented program in C++ to read the following information from the keyboard in which the base class consists of employee name, code and designation, and the derived class containing the data members, viz. years of experience and age.

```
Employee name
Employee code
Designation
Years of experience
Age
```

   Design a virtual base class for the item employee name and code.

   (b) Construct an object-oriented data base to carry out the following:

   (i) build a master table
   (ii) list a table
   (iii) insert a new entry
   (iv) delete old entry
   (v) edit an entry
   (vi) search for a record that is to be printed
   (vii) sort entries
9. Develop an object-oriented program in C++ to create a data base of the following items of the derived class.

```
Name of the patient
Sex
Age
Ward number
Bed number
Nature of the illness
Date of admission
```

   Design a base class consisting of the data members namely, name of the patient, sex and age. Another base class consists of ward number, bed number and nature of the illness. The derived class consists of the data member date of admission.

   Design a virtual base class for the data member, namely, name of the patient, sex and age.

   The program should have the facilities as mentioned in 8(b).
10. Develop an object-oriented program in C++ to create a pay roll system of an organisation with the following information read from the keyboard:

```
Employee name
Employee code
Designation
Account number
Date of joining
Basic pay
DA, HRA and CCA
Deductions like PPF, GPF, CPF, LIC, NSS, NSC, etc
```

Design a base class consisting of employee name, employee code and designation. Another base class consists of data member, such as account number and date of joining. The derived class consists of the data member basic pay plus other earnings and deductions. Construct a base class for the item account number and date of joining.

The program should have the facilities as listed in 8(b).

11. Develop an object-oriented program in C++ to prepare the mark sheet of a university examination with the following items read from the keyboard:

```
Name of the student
Roll number
Subject name
Subject code
Internal assessment (IA) marks
University examination (UE) marks
```

Design a base class consisting of the data members such as name of the student, roll number and subject name. The derived class consists of the data members, subject code, internal assessment and university examination marks. Construct a virtual base class for the item name of the student and roll number.

The program should have the facilities as mentioned in 8(b).

12. Develop an object-oriented program in C++ to create a library information system containing the following for all the books in the library:

```
Accession number
Name of the author
Title of the book
Year of publication
Publisher's name
Cost of the book
```

Design a base class with the data members accession number, name of the author and title of the book. Another base class consists of year of publication and publisher's name. The derived class consists of the data member cost of the book. Construct a virtual base class for the accession number.

The program should have the facilities as mentioned in 8(b).

13. Develop an object-oriented program in C++ to create a data base for a personnel information system containing the following information:

```
Name
Date of birth
Blood group
Height
Weight
Insurance policy number
```

```
Contact address
Telephone number
Driving licence number, etc
```

Design a base class with name, date of birth and blood group. Another base class consists of the data members such as height and weight. One more base class consists of the insurance policy number, contact address. The derived class contains the data members, telephone number and the driving licence number.

Construct a virtual base class for the name and blood group.

The program should have the facilities as listed in 8(b).

Templates, Namespace and Exception Handling

Chapter 15

This chapter presents the most recent features of the object-oriented programming such as function templates, class templates and exception handling with suitable illustrative examples. This chapter also covers the importance of namespace which is one of the additions to the ANSI/ISO C++ standard.

15.1 FUNCTION TEMPLATE

Template is a method for writing a single function or class for a family of similar functions or classes in a generic manner. When a single function is written for a family of similar functions, it is called as a 'function template'. In this function at least one formal argument is generic.

In the previous chapter, it has been explained how a function can be overloaded for a similar type of operation to be executed and how these functions are defined and called in a program. In function overloading, though the same name is used for all functions which are defined as overloading, yet the codes are to be repeated for every function. Only the function names are same, but the function definition and declaration are repeated.

For example,

```
swap ( char *, char *) // swapping two character data types
{
     -------
     -------
}

swap ( int, int)  // swapping two integer quantities
{
```

```
    -------
    -------
}

swap ( oat,  oat) // swapping two  oating point numbers
{
    -------
    -------
}
```

C++ provides certain features with the capability to define a single function for a group of similar functions. When a single function is written for a family of similar functions, they are called as function templates. The main advantage of using function template is avoiding unnecessary repetition of the source code. The object code becomes more compact and efficient than the conventional way of declaring and defining the functions. Secondly, full data type checking is carried out.

The function template does not specify the actual data types of the arguments that a function accepts but it uses a generic or parameterised data type. In a function template at least one formal argument is generic.

The general syntax for declaring a function template in C++ is,

```
template<class T> T function_name  (T formal arguments)
{
    -------
    -------
    return(T);
}
```

where the `template` and `class` are keywords in C++ and the function template must start with 'template' and the `T` is a parameterized data type.

The above declaration of a function template may be written in the following format also.

```
template<class T>
T function_name (T formal arguments)
{
    -------
    -------
    return(T);
}
```

Users may be anxious to know how the argument type is determined. The return type of the function is never considered for the actual parameterized data types to be processed. The actual data type of the function matches with the formal arguments of the function declaration whenever parameterized arguments are inferred in the function template.

Few examples high lighting the declaration of a function template are given below:

(1) A function template is defined to find the sum of the given array of elements such as int, float or double etc.

```
template<class T>
T sum (T array [], int n)
{
    T temp = 0;
    for ( int i = 0; i<= n-1; ++i)
        temp = temp+array[i];
    return(temp);
 }
```

(2) A function template is defined to swap two given items of different data like int, float, double or a character.

```
template<class T>
T swap (T & rst, T &second)
{
    T temp;
    temp =  rst;
     rst = second;
    second = temp;
    return(0);
}
```

PROGRAM 15.1

A program to define a function template for summing an array of integers and an array of floating point numbers.

```
//using function template
#include <iostream>
using namespace std;
template<class T> T  sum (T array[], int n)
{
    T temp = 0;
    for (int i = 0; i<= n-1; ++i)
    temp = temp+array[i];
    return(temp);
}
int main()
{
    int n = 3, sum1;
     oat sum2;
    static int a[3] = {1,2,3};
    static double b[3] = {1.1,2.2,3.3};
    sum1 = sum(a,n);
    cout << " sum of the integers = " << sum1 << endl;
    sum2 = sum(b,n);
    cout << " sum of the  oating point numbers = " << sum2;
    cout << endl;
    return 0;
}
```

Output of the above program

```
sum of the integers = 6
sum of the  oating point numbers = 6.6
```

In the above program, the function template sum () has been defined as a generic function for summing an array of values of the size up to n elements where the number of elements are passed by the function call from the calling portion of the program. The sum () function template is called twice — once to find the sum of the integer array and again to find the sum of floating point numbers.

PROGRAM 15.2

A program to define the function template for swapping two items of the various data types, namely, integers, floating point numbers, characters and string based on object oriented programming approach.

```
//swapping based on OOPs
#include <iostream>
#include <string>
```

```
using namespace std;
class sample {
    public:
        int a,b;
         oat fa,fb;
        char ch1,ch2;
        string str1,str2;
        template <class T> void swap (T &a, T &b);
        void menu();
};

template <class T>
void sample :: swap(T &a, T &b)
{
    T temp = a;
    a = b;
    b = temp;
}
void sample :: menu()
{
    cout <<"Welcome to parameterized template programming \n";
    cout <<" i -> integer swapping \n";
    cout <<" f ->  oating point number swapping\n";
    cout <<" c -> character swapping \n";
    cout <<" s -> string swapping \n";
    cout <<" m -> menu() \n";
    cout <<" q -> quit \n";
    cout <<" option, please ? \n";
}
int main()
{
    sample obj;
    char code;
    obj.menu();
    while ((code = cin.get()) != 'q') {
        switch (code) {
            case 'i':
                cout <<"enter any two integers\n";
                cin >> obj.a >> obj.b;
                cout <<" Before swapping \n";
                cout << "a = " << obj.a << " ,b = " << obj.b << endl;
                obj.swap(obj.a,obj.b);
                cout <<" After swapping \n";
                cout << "a = " << obj.a << " ,b = " << obj.b << endl;
                break;
            case 'f':
                cout <<"enter two  oating point numbers\n";
                cin >> obj.fa >> obj.fb;
                cout <<" Before swapping \n";
                cout << "fa = " << obj.fa << " ,fb = " << obj.fb << endl;
                obj.swap(obj.fa,obj.fb);
                cout <<" After swapping \n";
                cout << "fa = " << obj.fa << " ,fb = " << obj.fb << endl;
                break;
            case 'c':
                cout <<"enter any two characters\n";
                cin.ignore();
                obj.ch1 = cin.get();
                cin.ignore(); //skip any whitespace like new line
                obj.ch2 = cin.get();
                cout <<" Before swapping \n";
                cout << " ch1 = " << obj.ch1;
                cout << " ,ch2 = " << obj.ch2;
                cout << endl;
                obj.swap(obj.ch1,obj.ch2);
                cout <<" After swapping \n";
                cout << " ch1 = " << obj.ch1;
```

```
                cout << " ,ch2 = " << obj.ch2;
                cout << endl;
                break;
            case 's':
                cout <<"enter any two words\n";
                cin >> obj.str1;
                cin.ignore(); //skip whitespace, if any
                cin >> obj.str2;
                cout <<" Before swapping \n";
                cout << "str1 = " << obj.str1 << " ,str2 = " << obj.str2;
                cout << endl;
                obj.swap(obj.str1,obj.str2);
                cout <<" After swapping \n";
                cout << "str1 = " << obj.str1 << " ,str2 = " << obj.str2;
                cout << endl;
                break;
            case 'm':
                obj.menu();
                break;
        } // end of switch-case structure
    } //end of while statement
    return 0;
}
```

Output of the above program

```
Welcome to parameterised template programming
i -> integer swapping
f ->  oating point number swapping
c -> character swapping
s -> string swapping
m -> menu()
q -> quit
option, please?

i
enter any two integers
10 20
Before swapping
a = 10, b = 20
After swapping
a = 20, b = 10

f
enter two  oating point numbers
1.1   2.2
Before swapping
fa = 1.1, fb = 2.2
After swapping
fa = 2.2, fb = 1.1

c
enter any two characters
a b
Before swapping
ch1 = a, ch2 = b
After swapping
ch1 = b, ch2 = a

s
```

```
enter any two words
Hyderabad   Bangalore
Before swapping
str1 = Hyderabad, str2 = Bangalore
After swapping
str1 = Bangalore, str2 = Hyderabad
q
```

PROGRAM 15.3

A program to demonstrate how a function template is constructed to find square of a given number with different data types such as integers, floating point numbers and double.

```
//using function template
#include <iostream>
using namespace std;
template<class T> T  square ( T n)
{
    return(n*n);
}

int main()
{
    int x,xsq;
     oat y,ysq;
    double z,zsq;
    cout << "enter an integer \n";
    cin >> x;
    cout << " enter a  oating point number ? \n";
    cin >> y;
    cout << " enter a double precision number \n";
    cin >> z;
    xsq = square (x);
    cout << " x = " << x << "  and its square = " << xsq << endl;
    ysq = square (y);
    cout << " y = " << y << "  and its square = " << ysq << endl;
    zsq = square (z);
    cout << " z = " << z << "  and its square = " << zsq << endl;
    return 0;
}
```

Output of the above program

```
enter an integer
2
enter a  oating point number?
2.2
enter a double precision number
2.34544
x = 2  and its square = 4
y = 2.2  and its square = 4.84
z = 2.34544  and its square = 5.50111
```

15.2 CLASS TEMPLATE

In addition to function templates, C++ also supports the concept of class templates. By definition, a class template is a class definition that describes a family of related classes. C++ offers the user the ability to create a class that contains one or more data types that are generic or parameterised.

The manner of declaring the class template is the same as that of a function template. The keyword template must be inserted as a first word for defining a class template.

The general syntax of the class template is,

```
template <class T>
class user_de ned_name
{
    private:
        -------
        -------
    public:
        -------
        -------
};
```

The following program segment illustrates how to define and declare a class template in C++.

```
#include <iostream>
using namespace std;
template <class T>
class sample {
    private:
       T value, value1, value2;
    public:
       void getdata();
       void sum();
};
int main()
{
    sample <int> obj1;
    sample < oat> obj2;
    -------
    -------
    return 0;
}
```

Once the class template has been defined, it is required to instantiate a class object using a specific primitive or user-defined type to replace the parameterised types.

A member function of a class template also contains the keyword `template` whenever it is declared outside the scope of a class definition. For example, the following program segment shows how to declare a member function for the class template.

```
#include <iostream>
using namespace std;
template <class T>
class sample {
   private:
       T value,value1,value2;
   public:
       void getdata();
       void sum();
};
template <class T>
void sample <T> :: getdata()
{
   cin >> value1 >> value2;
}
```

```
template <class T>
void sample <T> :: sum()
{
    T value;
    value = value1+value2;
    cout << " sum of = " << value << endl;
}
```

PROGRAM 15.4

A program to illustrate how to define and declare a class template to find the sum of the given two data items.

```
//adding two parameterised data types
#include <iostream>
using namespace std;
template <class T>
class sample {
    private:
        T value, value1, value2;
    public:
        void getdata();
        void sum();
};

template <class T>
void sample <T> :: getdata()
{
    cin >> value1 >> value2;
}

template <class T>
void sample <T> :: sum()
{
    T value;
    value = value1+value2;
    cout << " sum = " << value << endl;
}

int main()
{
    sample <int> obj1;
    sample < oat> obj2;
    cout << " enter any two integers :" << endl;
    obj1.getdata();
    obj1.sum();
    cout << " enter any two  oating point numbers :" << endl;
    obj2.getdata();
    obj2.sum();
    return 0;
}
```

Output of the above program

```
enter any two integers:
10   20
sum = 30

enter any two  oating point numbers:
10.11   20.22
sum = 30.33
```

PROGRAM 15.5

A program to read a set of numbers from the user input and find the sum and average of given numbers using a class template.

```
// nding sum and average of given numbers
#include <iostream>
using namespace std;
template <class T>
class sample{
    private:
        T a;
    public:
        void sum (int n);
};

template <class T>
void sample <T> :: sum(int n)
{
    T temp = 0;
    for (int i = 0; i <= n-1; ++i) {
        cout << "enter a value \n";
        cin >> a;
        temp += a;
    }
    cout << "sum = " << temp;
    T ave = temp/ n;
    cout << " ,and Average = " << ave << endl;
}

int main()
{
    int n;
    cout << "How many numbers ?\n";
    cin >> n;
    cout <<" for integers \n";
    sample <int> obj1;
    obj1.sum(n);
    cout << "  oating point numbers\n";
    sample <double> obj2;
    obj2.sum(n);
    return 0;
}
```

Output of the above program

```
How many numbers?
5
for integers
enter a value
1
enter a value
2
enter a value
3
enter a value
4
enter a value
5
sum = 15, and Average = 3
```

```
 oating point numbers
enter a value
1.1
enter a value
2.2
enter a value
3.3
enter a value
4.4
enter a value
5.5
sum = 16.5, and Average = 3.3
```

15.3 OVERLOADING OF FUNCTION TEMPLATE

We have already seen that a function can be overloaded in C++. It is well known that function templates are used for constructing a generic and parameterised data types. C++ also supports to construct a program for overloading these function templates. One can implement the function templates for overloading either as a stand alone function template or member functions of a class.

PROGRAM 15.6

A program to demonstrate how to perform the function overloading using a function template.

```
//overloading of function template
#include <iostream>
using namespace std;
template <class T>
void display(T a)
{
    cout << "calling function template \n";
    cout << " a = " << a << endl;
}

void display(int n)
{
    cout << "calling conventional display \n";
    cout << " n = " << n << endl;
}

int main()
{
    display(10);
    display(1.1);
    return 0;
}
```

Output of the above program

```
calling conventional display
n = 10
calling function template
a = 1.1
```

PROGRAM 15.7

A program to demonstrate how to perform the function overloading using a function template. The function templates are defined as the member functions of a class.

```
//overloading of template methods
#include <iostream>
using namespace std;
template <class T>
class sample {
    public:
        void display (T a);
        void display (T a, T b);
        void display (T a, T b, T c);
};

template <class T>
void sample <T> :: display (T a)
{
    cout << " a = " << a << endl;
}

template <class T>
void sample <T> :: display (T a, T b)
{
    cout << " a = " << a << " b = " << b << endl;
}

template <class T>
void sample <T> :: display (T a, T b, T c)
{
    cout << " a = " << a << " b = " << b << " c = " << c;
    cout << endl;
}

int main()
{
    sample <int> obj1;
    obj1.display(10);
    obj1.display(10,20);
    obj1.display(10,20,30);
    sample <double> obj2;
    obj2.display(1.1);
    obj2.display(1.1,2.2);
    obj2.display(1.1,2.2,3.3);
    return 0;
}
```

Output of the above program

```
a = 10
a = 10 b = 20
a = 10 b = 20 c = 30
a = 1.1
a = 1.1 b = 2.2
a = 1.1 b = 2.2 c = 3.3
```

PROGRAM 15.8

A program to demonstrate how to define and declare a class template with default constructor.

```
//using class template
#include <iostream>
using namespace std;
template <class T>
class sample {
    private:
        T value;
    public:
        sample (T = 0) //default constructor
        {
            cout << " default constructor is called \n";
            cout << " value = " << value << endl;
        }
};
int main()
{
    sample <int> obj1;
    sample < oat> obj2;
    return 0;
}
```

Output of the above program

```
default constructor is called
value = 1073828704
default constructor is called
value = 36.7598
```

Since automatic variable is not initialised by the programmer, compiler displays a garbage value. Hence the above output.

PROGRAM 15.9

A program to demonstrate how to define and declare a class template with a constructor and a member function.

```
//using class template
#include <iostream>
using namespace std;
template <class T>
class sample {
    public :
        sample ();
        inline void display();
};

template <class T>
sample<T> :: sample()
{
    cout <<"class template - constructor \n";
}

template <class T>
void sample<T> :: display()
{
    cout << "this is a member function \n";
}

int main()
{
    sample <int> obj1;
    obj1.display();
```

```
    sample < oat> obj2;
    obj2.display();
    return 0;
}
```

Output of the above program

```
class template - constructor
this is a member function
class template - constructor
this is a member function
```

PROGRAM 15.10

A program to demonstrate how to define and declare a class template with a special member function, constructor and destructor.

```
//using class template
//de ning constructor and destructor
#include <iostream>
using namespace std;
template < class T>
class sample {
    private :
        T value;
    public :
        sample ();
        ~sample ();
        inline void display ();
};
template <class T>
sample <T> :: sample()
{
    cout <<"constructor \n";
}

template <class T>
sample <T> :: ~sample()
{
    cout << "destructor \n";
}

template < class T>
void sample <T> :: display()
{   cout <<"member function \n";
}

int main()
{
    cout << "Calling class object for integer \n";
    sample <int> obj1;
    obj1.display();
    cout << "Calling class object for  oating point \n";
    sample < oat> obj2;
    obj2.display();
    return 0;
}
```

Output of the above program

```
Calling class object for integer
constructor
member function
```

```
Calling class object for  oating point
constructor
member function
destructor
destructor
```

PROGRAM 15.11

A program to demonstrate how to define and declare a class template with a special member function, constructor and destructor. The constructor contains a single argument.

```
// constructor with single argument
// and destructor of class template
#include <iostream>
using namespace std;
template <class T>
class sample {
    private :
        T value;
    public :
        sample (T n) : value (n) {}; // constructor
        ~sample (){} // destructor
        void display ()
        {
            cout << " content of the value = " << value << endl;
        }
};

int main()
{
    sample <int> obj1(10);
    cout << " integer :" << endl;
    obj1.display();
    sample < oat> obj2(-22.12345);
    cout << " Floating point number :" << endl;
    obj2.display();
    return 0;
}
```

Output of the above program

```
integer:
content of the value = 10
Floating point number:
content of the value = -22.1234
```

PROGRAM 15.12

A program to demonstrate how to define and declare a class template with a special member function, constructor and destructor. The constructor contains a single argument with a different format.

```
// constructor with single argument
// and destructor of class template
#include <iostream>
using namespace std;
template <class T>
class sample {
    private:
```

```
        T value;
    public:
        sample ( T n) ; // constructor
       ~sample (); // destructor
        void display ();
};
template <class T>
sample <T> :: sample (T n) : value (n) { }

template <class T>
sample <T> :: ~sample(){}

template <class T>
void sample <T> :: display()
{
    cout << " content of the value = " << value << endl;
}

int main()
{
    sample <int> obj1(10);
    cout << " integer :" << endl;
    obj1.display();
    sample < oat> obj2(-22.12345);
    cout << " Floating point number :" << endl;
    obj2.display();
    sample <double> obj3(12345678L);
    cout << " Double precision number :" << endl;
    obj3.display();
    return 0;
}
```

Output of the above program

```
integer:
content of the value = 10
Floating point number:
content of the value = -22.1234
Double precision number:
content of the value = 1.23457e+07
```

15.4 EXCEPTION HANDLING

This section describes the C++ error handling mechanism which is generally referred to as exception handling. This section covers not only the syntax and semantics of the exception handling keywords but it also presents how to define, declare and implement those keywords in a program.

15.4.1 An Overview

An exception is an error or an unexpected event. The exception handler is a set of codes that executes when an exception occurs. Exception handling is one of the most recently added features and perhaps it may not be supported by much earlier versions of the C++ compilers. The main purpose of exception handling used in a program is to detect and manage runtime errors. The word exception comes from the exceptional program flow that occurs during a runtime error. C++ makes use of classes and objects in handling exceptions.

Whenever a caller of a function detects an error without exception handling, it is very difficult to trace, check and handle it in a complex and big software. The program must be developed with exception handling in such a way that it determines the possible errors the program might encounter and then include codes to handle them. For example, in the program which performs the input/output operation in file processing, it

is essential to check whether a file has been opened successfully or not and to display the appropriate error message if any unexpected event occurs.

The ANSI/ISO C++ language defines a standard for exception handling. An exception handling is a type of error handling mechanism whenever a program encounters an abnormal situation or runtime errors. In order to implement and realise the exception handling, the following keywords are used in C++:

```
try
catch
throw
```

15.4.2 The `Try` Block

A `try` block protects any code within the block either directly or indirectly. A try block is like a sentinel that guards some section of code, shielding it from errors. Only the code inside a try block can detect or handle exceptions. The try block can also be nested like any other C++ code.

The general syntax of the try block is,

```
try
{
    /* the C++ code  one wants to protect or
       shielding it from errors  */
}
```

If an exception occurs, the program flow is interrupted.

15.4.3 The Throw Expression

The throw statement actually throws an exception of the specified type. When an exception occurs, the throw expression initialises a temporary object which is to match the type of argument it used in throw statement.

The general syntax of the throw statement is,

```
throw argument;
```

where `argument` is sent to the corresponding catch handler.

Different types of throw expression are given below:

(1) `throw " An error occurred ";`
This example specifies that an error message is to be displayed.

(2) `throw object;`
This example specifies that object is to be passed on to a handler.

(3) `throw;`
This example simply specifies that the last exception thrown is to be thrown again.

15.4.4 The Catch Block

The catch block receives the thrown exception. The exception handler is indicated by the catch keyword. The general syntax of the catch block is as follows:

```
catch (parameter)
{
     //handles error here
}
```

The parameter, which can be named or unnamed, denotes the type of exception the clause handles. This parameter can be any valid C++ data type including a structure or a class. The catch handler is known by the data type given in the parameter. A catch handler is used to catch any type of exception object.

15.4.5 The Layout of Exception Handling

The general structure and layout of an exception handling is presented in this section. Usually, a program will have exception handlers defined for several types of errors that might occur. For example, the layout of a typical try block and its associated catch clause is:

```
try
{
    // code to protect
}

catch (int x)
{
    // handler int errors
}

catch (char *str)
{
    // handler char * errors
}

catch ( oat dx)
{
    // handler  oat errors
}
```

This try block has three catch handlers. The first one that catches integer exceptions, the second one for character pointer and the third one catches float exceptions. One can define as many catch handlers as one wants to cover all the bases.

PROGRAM 15.13

A program to demonstrate how to detect a divide by zero error using the exception handling technique.

```
#include <iostream>
#include <string>
using namespace std;
class sample {
    private:
         oat a,b;
    public:
        void getdata();
        void divide();
};
void sample ::getdata()
{
    cout<< "enter any two  oating point numbers\n";
    cin >> a >> b;
}
void sample :: divide()
{
     string str;
     try {
          if (b == 0)
             throw str;
          else
          {
              oat temp = a/b;
             cout << "Quotient = " << temp << endl;
```

```
            }
        }
        catch (string str) {
            cout << " Exception - Divide by zero \n";
        }
}
int main()
{
    sample obj;
    obj.getdata();
    obj.divide();
    return 0;
}
```

Output of the above program

```
enter any two  oating point numbers
1   2
Quotient = 0.5

1   0
enter any two  oating point numbers
Exception - Divide by zero
```

PROGRAM 15.14

A program to demonstrate how to define, declare and use an exception handling technique for detecting a memory out of range check error.

```
//memory out of range check
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
class sample {
    private:
        int a[10];
        string str;
    public:
        void getdata();
        void display();
};
void sample :: getdata()
{
    try {
        for (int i = 0; i < 9; ++i) {
            if (i > 5)
                throw str;
            else
            {
                cout <<"enter an element \n";
                cin >> a[i];
            }
        }
    }
    catch (string str) {
        cout << "\n Exception - Memory out of range";
    }
}

void sample ::display ()
{
```

```
    cout << "\n entered elements are \n";
    for (int i = 0; i <= 5; ++i)
        cout << setw(4) << a[i];
}
int main()
{
    sample obj;
    obj.getdata();
    obj.display();
    return 0;
}
```

Output of the above program

```
enter an element
10
enter an element
20
enter an element
30
enter an element
40
enter an element
50
enter an element
60
Exception - Memory out of range
entered elements are
10 20 30 40 50 60
```

PROGRAM 15.15

A program to illustrate how to use the multiple catch statements for throwing the different types of data for handling exception.

```
//using multiple catch
#include <iostream>
using namespace std;
class sample {
    private:
        int a;
    public:
        void getdata();
        void display();
};
void sample :: getdata()
{
    cout << "enter anyone (0,1 or -1) \n";
    cin >> a;
}
void sample :: display()
{
    int n;
    char ch;
    string str_name;
    try {
        if (a == 0) throw n;
        else if (a == 1) throw ch;
        else if (a == -1) throw str_name;
    }
    catch (int n) {
```

```
        cout << "Exception - integer \n";
    }
    catch (char ch) {
        cout << "Exception - Character \n";
    }
    catch (string str_name) {
        cout << "Exception - String \n";
    }
}
int main()
{
    sample obj;
    obj.getdata();
    obj.display();
    return 0;
}
```

Output of the above program

```
enter anyone (0,1 or -1)
0
Exception - integer

enter anyone (0,1 or -1)
1
Exception - Character

enter anyone (0,1 or -1)
-1
Exception - String
```

PROGRAM 15.16

A program to demonstrate how to implement catch all exceptions using the catch (...) statement.

```
//catch all exceptions
#include <iostream>
using namespace std;
class abc {
    public:
        void display(int x);
};
void abc :: display(int x)
{
    try
    {
        if (x == 0) throw 'x';   // handling char
        if (x == 1) throw x;     // handling int
        if (x == 2) throw 1.1f;  // handling  oat
        if (x == 3) throw 1000L; // handling long
    }
    catch (...)
    {
        cout << "Exception occurred \n";
    }
}
int main()
{
    abc obj;
    obj.display(0);
    obj.display(1);
```

```
    obj.display(2);
    obj.display(3);
    return 0;
}
```

Output of the above program

```
Exception occurred
Exception occurred
Exception occurred
Exception occurred
```

PROGRAM 15.17

A program to illustrate how to realise rethrowing an exception in C++ using throw statement without any argument.

```
// rethrowing an exception
#include <iostream>
using namespace std;
class abc {
    public:
        void display(int x, int y);
};
void abc :: display(int x, int y)
{
    try
        {
        if (y == 0)
           throw y;
        else
           cout << " x/y = " << (double) x/ (double) y << endl;
    }
    catch (int)
        {
        cout << "Exception occurred \n";
        throw;
    }
}
int main()
{
    abc obj;
    try {
        obj.display(1,0);
    }
    catch (int)
    {
        cout <<"Exception occurred in main \n";
    }
    return 0;
}
```

Output of the above program

```
Exception occurred
Exception occurred in main
```

15.4.6 Standard Exceptions

The C++ library defines a number of common exceptions. The standard exceptions are defined in the following header file.

```
#include <stdexcept>
```

In general, exception handling mechanism can be classified into two categories, namely, logic error and runtime error.

(a) Logic Error The abnormal program termination can be caused due to the following error states:

- domain_error
- invalid_argument
- length_error
- out_of_range

(b) Runtime_error The run-time error can be occurred due to the following conditions.

- range_error
- overflow_error
- underflow_error

15.5 NAMESPACE

This section deals with the importance of namespace which is one of the salient features added in the ANSI/ISO C++ standard. This section also explains how to use and implement a standard namespace; how to realise a namespace alias; how to accomplish and construct a program using unnamed namespace and nested namespace in C++.

15.5.1 Namespace Declaration

A namespace is a technique of expressing logical grouping of all elements into a single translation unit in order to avoid the name collision or conflict. A namespace is a scope. In general, local scopes, global scopes and classes are namespaces.

There is a possibility that a global object or function uses the same identifier as another one, causing redefinition of identifiers. Name collision is a common thing in a big and bulky software systems. The main objective and purpose of using the namespace declaration in a program is to avoid the redefinition errors caused by the name collision.

A namespace declaration identifies and assigns a unique name to a user-declared namespace. Such namespaces are used to solve the problem of name collision in large programs and libraries. Programmers can use namespaces to develop new software components and libraries without causing naming conflicts with existing components. Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

The general syntax of namespace declaration is,

```
namespace identi er
{
   // entities
}
```

where `identi er` is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace.

For example,

```
namespace X
{
    int a;
}
namespace Y
{
    double a;
```

```
}
```

In this case, the variable `int a` is normal variable declared within a namespace called X and `double a` is within a namespace called `Y`. In order to access these variables from outside the `X` and `Y` namespace, one has to use the scope resolution operator `::`. The variables `int a` and `double a` can be accessed in the following manner.

```
X :: a;
Y :: a;
```

A namespace declaration, whether it involves a new namespace, an unnamed namespace, or an extended namespace definition, must be accompanied by a namespace body enclosed within curly braces. The namespace body may contain declarations or definitions of variables, functions, objects, templates, and nested namespaces. A list of these declarations are said to be members of the namespace.

The following namespace declaration statement causes syntax error and that is an invalid form of declaration.

```
namespace X;   //invalid
```

One can declare namespace statement without any members. Even though the following namespace declaration is a valid statement, it is meaningless.

```
namespace X   //valid, but meaningless
{
}
```

For example, the following program illustrates how the namespace declaration avoids the name collision of the identifiers of the same names.

```
//namespace
#include <iostream>
using namespace std;
namespace  rst
{
     int abc = 5;
}
namespace second
{
     double abc = 3.1416;
}
int main ()
{
     cout <<  rst::abc << endl;
     cout << second::abc << endl;
     return 0;
}
```

Output of the above program

```
5
3.1416
```

In the above program, there are two global variables with the same name, abc. One is defined within the namespace first and the other one in second. No redefinition errors happen due to namespace declaration.

PROGRAM 15.18

A program to illustrate how to declare, define and realise a namespace mechanism.

```
#include <iostream>
using namespace std;
namespace A {
    int a = 10;
    void display();
}
namespace B {
    int a = 100;
    void display();
}
void A:: display()
{
    cout << "A :: display \n";
    cout << A::a << endl;
}

void B :: display()
{
    cout << "B :: display\n";
    cout << B::a << endl;
}

int main()
{
    A::display();
    B::display();
    return 0;
}
```

Output of the above program

```
A :: display
10
B :: display
100
```

15.5.2 Defining Namespace Members

Members of a namespace may be defined within that namespace. A namespace member can contain not only the simple data types but also the functions and aggregated data types. Members of a namespace can be declared and introduced in the following form:

```
namespace identi er
{
    //declarations and de nitions of members
}
```

The members can be declared within a namespace definition. These members can be referred and accessed using the scope resolution operator in the following form:

```
namespace_name :: members_name;
```

For example the following namespace declaration shows how members of a namespace can be defined.

```
namespace X
{
      int i = 10;
      void f()
}
namespace Y
{
      int i = 20;
       oat a = 1.1f;
      void g();
```

```
}
namespace Z
{
    int i = 30;
    double dx = 2.2;
    void d();
}
```

Unlike other declarative regions, the definition of a namespace can be split over several parts of a single translation unit. For example, the following program segment shows how to perform and realise the extension of namespace declaration:

```
namespace A
{
    // declare namespace A variables
    int i;
    int j;
}
namespace B
{
}
namespace C
{
}
namespace D
{
}
namespace A
{
    // declare namespace A functions
    void func(void);
    int int_func(int i);
}
namespace B
{
    //add some members for B
}
int main()
{
}
```

When a namespace is continued in this manner, after its initial definition, the continuation is called an extension namespace definition.

PROGRAM 15.19

A program to show how to construct an extension namespace definition.

```
#include <iostream>
using namespace std;
namespace A
{
    // declare namespace A variables
    int i = 10;
}
namespace B
```

```
{
    int i = 100;
}
namespace A
{
    // declare namespace A functions
    void func();
}
void A::func()
{
    cout << "++A::i = " << ++A::i << endl;
}
namespace B
{
    int j = 200;
    void func();
}

void B::func()
{
    cout << "++B::i = " << ++B::i << endl;
    cout << " B::j = "  << B::j << endl;
}
int main()
{
    A::func();
    B::func();
    return 0;
}
```

Output of the above program

```
++A::i = 11
++B::i = 101
B::j = 200
```

Members of a named namespace can be defined outside the namespace in which they are declared by explicit qualification of the name being defined. However, the entity being defined must already be declared in the namespace. In addition, the definition must appear after the point of declaration in a namespace that encloses the declaration's namespace.

The following program segment shows how an error is caused due to improper definition of namespace members where the entities are defined before namespace declaration.

```
namespace X
{
     void f();
}
void X::f() { }   //ok

void Y::g() { }   //error, the entity must be declared in the
                  // namespace before it is de ned.
namespace Y
{
     void g();
}
```

The corrected form of the above member definition is given below:

```
namespace X
{
     void f();
}
```

```
void X::f() { }

namespace Y
{
     void g();
}
void Y::g() { }
```

PROGRAM 15.20

A program to demonstrate how an error is caused due to improper definition of namespace members.

```
#include <iostream>
using namespace std;
namespace A
{
    int i = 10;
    void f();
}
void A::f()
{
    cout << "++i = " << ++i << endl;
}
void V::g()
{
    cout << " ++j = " << ++j << endl;
}
namespace V
{
    int j = 20;
    void g();
}
int main()
{
    A::f();
    V::g();
    return 0;
}
```

Compile time error The function `V::g()` and the `int j` are undefined. Member functions of `namespace V` are defined before the namespace declaration.

15.5.3 Namespace Alias

A namespace alias is a mechanism or technique in which one can declare alternate names for existing namespaces. In other words, a namespace alias definition substitutes a short name for a long and lengthy namespace name. The identifier is a synonym for the qualified namespace specifier and becomes a namespace alias.

The namespace alias is declared in the following format:

```
namespace new_name = current_name;
```

For example,

```
namespace Very_Long_Namespace_Name
{
    int a;
    double x;
    void f();
}
```

```
namespace ABC = Very_Long_Namespace_Name
```

Now ABC is a namespace alias for Very_Long_Namespace_Name. A namespace name cannot be identical to any other entity in the same declarative region. In addition, a global namespace name cannot be the same as any other global entity name in a given program.

PROGRAM 15.21

A program to demonstrate how to declare, define and construct a namespace alias.

```
//namespace alias
#include <iostream>
using namespace std;
namespace Indian_Institute_of_Technology_Kharagpur {
    string cse = "Prof Ghosh";
    string ece = "Prof Banerjee";
}
namespace IIT_KGP = Indian_Institute_of_Technology_Kharagpur;
int main()
{
    cout << " Name 1 = " << IIT_KGP::cse << endl;
    cout << " Name 2 = " << IIT_KGP::ece << endl;
    return 0;
}
```

Output of the above program

```
Name 1 = Prof Ghosh
Name 2 = Prof Banerjee
```

15.5.4 Nested Namespace

A namespace definition can be nested within another namespace definition. Every namespace definition must appear either at file scope or immediately within another namespace definition.

```
namespace outer
{
    entities_of_outer
    namespace inner
    {
        entities_of_inner
        namespace innermost
        {
            entities_of_innermost
        }
    }
}
```

The scope resolution operator is used to access the entities of each of the namespace items. For example, the following program segment shows how to construct and realise a nested namespace:

```
namespace A
{
    int i = 10;
    void f();
    namespace B
    {
```

```
            int j = 20;
            void g();
            namespace C
            {
                int k = 30;
                void h();
            }
        }
    }
```

Each items of the namespace can be accessed using the scope resolution operator in the following manner:

(1) To access the members of the namespace A,
```
A::i;
A::f();
```

(2) To access the members of the namespace B,
```
A::B::j;
A::B::g();
```

(3) To access the members of the namespace C,
```
A::B::C::k;
A::B::C::h();
```

PROGRAM 15.22

A program to declare, define and perform the nested namespace.

```
#include <iostream>
using namespace std;
namespace X {
    int a = 10;
    namespace Y {
         oat a = 20.2f;
        namespace Z {
            char a = 'a';
        }
    }
}
int main()
{
    cout << "X::a = " << X::a << '\n';
    cout << "X::Y::a = " << X::Y::a << '\n';
    cout << "X::Y::Z::a = " << X::Y::Z::a << '\n';
    return 0;
}
```

Output of the above program

```
X::a = 10
X::Y::a = 20.2
X::Y::Z::a = a
```

PROGRAM 15.23

A program to demonstrate how a nested namespace is realised.

```
//using nested namespace
#include <iostream>
```

```
using namespace std;
namespace A {
    int a = 10;
    void display();
    namespace B {
        int a = 100;
        void display();
    }
}
void A:: display()
{
    cout << "A :: display \n";
    cout << ++(A::a) << endl;
}

void A::B:: display()
{

    cout << "A::B :: display\n";
    cout << ++A::B::a << endl;
}

int main()
{
    A::display();
    A::B::display();
    return 0;
}
```

Output of the above program

```
A:: display
11
A::B :: display
101
```

The following program segment shows how an error is caused due to improper definition of namespace members where the entities are defined before namespace declaration.

```
namespace Q {
    namespace V {
        void f();
    }
    void V::f() { }  // ok
    void V::g() { }  // g() is not yet a member of V

    namespace V {
        void g();
    }
}
```

PROGRAM 15.24

A program to illustrate how to declare, define and accomplish the extension of namespace within a nested namespace.

```
#include <iostream>
using namespace std;
namespace Q {
    namespace V {
        int i = 10;
```

```
        void f();
    }
    void V::f()
    {
        cout << "++i = " << ++i << endl;
    }
    namespace V {
        int j = 20;
        void g();
    }
    void V::g()
    {
        cout << "++j = " << ++j << endl;
    }
}

int main()
{
    Q::V::f();
    Q::V::g();
    return 0;
}
```

Output of the above program

```
++i = 11
++j = 21
```

15.5.5 Unnamed namespace

It is often useful to wrap a set of declarations in a namespace simply to protect against the possibility of name clashes without using any namespace name. A namespace is declared without a user-defined name is known as an unnamed namespace. An unnamed namespace has an implied using directive. One can declare an unnamed namespace as a superior alternative to the use of global static variable declarations.

The general syntax of the unnamed namespace is given below:

```
namespace
{
   //  namespace body
}
```

An unnamed namespace definition having the syntax shown above behaves as if it were replaced by:

```
namespace unique
{
    //  namespace body
}
using namespace unique;
```

Each unnamed namespace has an identifier, assigned and maintained by the program and represented here by unique, that differs from all other identifiers in the entire program.

For example, the following program segment shows how to declare an unnamed namespace:

```
namespace
{
    int i;
    void f();
    void g();
}
```

Unnamed namespaces are a superior replacement for the static declaration of variables. They allow variables and functions to be visible within an entire translation unit, yet not visible externally. Although

entities in an unnamed namespace might have external linkage, they are effectively qualified by a name unique to their translation unit and therefore can never be seen from any other translation unit.

PROGRAM 15.25

A program to show how to use the unnamed namespace declaration.

```
//using unnamed namespace
#include <iostream>
using namespace std;
namespace {
    int a = 10;
}
namespace {
    int b = 100;
}
void display1()
{
    cout << "display1 = ";
    cout << a++ << endl;
}

void display2()
{
    cout << "display2 = ";
    cout << ++b << endl;
}

int main()
{
    display1();
    display2();
    return 0;
}
```

Output of the above program

```
display1 = 10
display2 = 101
```

PROGRAM 15.26

A program to illustrate how to declare and perform the unnamed namespace in a nested namespace declaration.

```
//using unnamed namespace
#include <iostream>
using namespace std;
namespace {
    int a = 10;
    namespace {
        int b = 20;
        namespace {
            int c = 30;
            namespace {
                int d = 40;
            }
        }
    }
}
```

```
void display()
{
    cout << " ++a = " << ++a << endl;
    cout << " ++b = " << ++b << endl;
    cout << " ++c = " << ++c << endl;
    cout << " ++d = " << ++d << endl;

}
int main()
{
    display();
    return 0;
}
```

Output of the above program

```
++a = 11
++b = 21
++c = 31
++d = 41
```

15.5.6 The Keyword Using

The keyword `using` is used to introduce a name from a namespace into the current declarative region. The general syntax of the keyword using is given below:

```
using userde ned_name :: member;
```

The following program segment shows how to use the keyword using:

```
namespace A
{
    int x = 5;
}
int main ()
{
    using A::x;
    cout << ++x << endl;
}
```

PROGRAM 15.27

A program to illustrate how to use the keyword using for namespace declaration.

```
// The keyword using
#include <iostream>
using namespace std;
namespace  rst
{
    int x = 5;
    int y = 10;
}

namespace second
{
    double x = 3.1416;
    double y = 2.7183;
}

int main ()
{
```

```
    using  rst::x;
    using second::y
    cout << x << endl;
    cout << y << endl;
    cout <<  rst::y << endl;
    cout << second::x << endl;
    return 0;
}
```

Output of the above program

```
5
2.7183
10
3.1416
```

Notice how in this code, `x` (without any name qualifier) refers to `rst::x` whereas `y` refers to `second::y`, exactly as our using declarations have specified. We still have access to `rst::y` and `second::x` using their fully qualified names. The keyword using can also be used as a directive to introduce an entire namespace:

```
// The keyword using
#include <iostream>
using namespace std;
namespace  rst
{
    int x = 5;
    int y = 10;
}

namespace second
{
    double x = 3.1416;
    double y = 2.7183;
}

int main ()
{
    using namespace  rst;
    cout << x << endl;
    cout << y << endl;
    cout << second::x << endl;
    cout << second::y << endl;
    return 0;
}
```

Output of the above program

```
5
10
3.1416
2.7183
```

In this case, since we have declared that we were `using namespace` first, all direct uses of `x` and `y` without name qualifiers were referring to their declarations in namespace first.

The keyword 'using' and 'using namespace' have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope. For example, if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

```
// using namespace example
#include <iostream>
using namespace std;

namespace  rst
{
     int x = 5;
}

namespace second
{
     double x = 3.1416;
}

int main ()
{
    {
       using namespace  rst;
       cout << x << endl;
    }
    {
       using namespace second;
       cout << x << endl;
    }
    return 0;
}
```

Output of the above program

```
5
3.1416
```

15.5.7 The Using Directive

The `using directive` allows the names in a namespace to be used without the namespace name as an explicit qualifier. Of course, the complete, qualified name can still be used to improve readability.

Note the difference between the using directive and the using declaration: the using declaration allows an individual name to be used without qualification, the using directive allows all the names in a namespace to be used without qualification.

```
#include <iostream>
int main()
{
    std::cout << "Hello ";
    using namespace std;
    cout << "C++ World" << endl;
}
```

Output of the above program

```
Hello C++ World
```

If a local variable has the same name as a namespace variable, the namespace variable is hidden. It is an error to have a namespace variable with the same name as a global variable.

15.5.8 Namespace std

All the files in the C++ standard library declare all of its entities within the `std namespace`. That is why we have generally included the `using namespace std;` statement in all programs that used any

entity defined in iostream. The ANSI/ISO C++ standard requires to explicitly declare the namespace in the standard library.

Whenever the header file <iostream> is used in a program, one has to specify the namespace of `cout` in one of the following ways:

```
std::cout (explicitly)
using std::cout (using declaration)
using namespace std (using directive)
```

15.5.9 Explicit Qualification

Namespace members can be accessed using an explicit qualifier and the scope resolution operator. The following program demonstrates how to access the global identifier using explicit qualification:

```
int i;
namespace A
{
    int a, b, c;
    namespace B
    {
       int i, j, k;
    }
}

int main()
{
    A::a++;
    A::B::i++;   // B's i
    ::i++;   // the global i
}
```

The statement `::i++` accesses the `i` that is declared in the first statement of the example. Such usage of the scope resolution operator without a preceding qualifier invokes the global namespace.

Usage of explicit qualification might be cumbersome with longer names or in large programs. The using declaration, using directive, and namespace aliases provide more straightforward ways to reference namespace members.

REVIEW QUESTIONS

1. What is a template? List the merits and demerits of using a template in C++.
2. In what way a template increases the efficiency of designing a program?
3. What are the disadvantages of using a function template in C++?
4. Define a function template.
5. Explain how a function template is defined and declared in a program.
6. What are the rules to be followed while defining the definition of a function template?
7. What is a class template? List a few applications of defining a class with a parameterised data type.
8. Explain the syntactic rules of declaring the following with class templates:
   (i) constructor
   (ii) destructor
   (iii) default constructor
9. What is exception handler? What are the keywords used to handle the exception in C++?

10. Explain how an exception handler is defined and invoked in a program.
11. List the merits and demerits of defining an exception handler in an object-oriented programming.
12. Explain the importance of namespace declaration in C++.
13. What is meant by namespace std?
14. Explain how a nested namespace is used in C++.
15. What are the advantages of using unnamed namespace?

CONCEPT REVIEW PROBLEMS

1. What will be the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
template<class T > T  sum (T array[3][3], int n)
{
    T temp = 0;
    for (int i =0; i<= n-1; ++i)
       for (int j = 0; j <= n-1; ++j)
          temp = temp+array[i][j];
    return(temp);
}

int main()
{
    int n = 3, sum1;
     oat sum2;
    static int a[3][3] = {{1,2,3},
                          {4,5,6},
                          {7,8,9}
                         };
    static double b[3][3] = {{1.1,2.2,3.3},
                             {4.4,5.5,6.6},
                             {7.7,8.8,9.9}
                            };
    sum1 = sum(a,n);
    cout << " sum of the integers = " << sum1 << endl;
    sum2 = sum(b,n);
    cout << " sum of the  oating point numbers = " << sum2;
    cout << endl;
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
template <class T>
class sample {
    private:
        T a,b;
    public:
        T  nd_min (T a, T b);
};
```

```
template <class T>
T sample <T> ::  nd_min (T a, T b)
{
    return ( a < b ? a : b);
}

int main()
{
    int a = 10, b = 20;
    sample <int> obj1;
    int temp1 = obj1. nd_min(a,b);
    cout << "a = " << a << " b = " << b;
    cout << " minimum = " << temp1 << endl;
    double x = 1.1, y = 2.2;
    sample <double>  obj2;
    double temp2 = obj2. nd_min(x,y);
    cout << "x = " << x << " y = " << y;
    cout << " minimum = " << temp2 << endl;
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
template <class T>
class abc {
    private:
       T value;
    public:
       abc() {
          cout << "default constructor \n";
       }
       abc (int a) {
          cout << "constructor with int argument \n";
       }
       abc ( oat af) {
          cout << "constructor with  oating point \n";
       }
};
int main()
{
    abc<int> obj1(10);
    abc< oat> obj2(-21.22f);
    return 0;
}
```

(d)

```
//using class template
#include <iostream>
using namespace std;
template <class T>
class sample {
    public :
       sample ();
       inline void display();
};
```

```
template <class T>
sample<T> :: sample()
{
    cout <<"class template - constructor \n";
}

template <class T>
void sample<T> :: display()
{
    cout << "this is a member function \n";
}

int main()
{
    sample <long int> obj1;
    sample < oat> obj2;
    sample <double> obj3;
    sample <signed char> obj4;
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
template <class T>
class sample {
    private:
        T value;
    public:
        sample (T n) : value (n)
        {
              cout << "constructor\n";
        };
       ~sample ()
        {
          cout << "destructor \n";
        }
        void display ()
        {
           cout << " content of the value = " << value << endl;
        }
};

int main()
{
    sample <int> obj1(10);
    cout << " integer :" << endl;
    obj1.display();
    return 0;
}
```

2. Determine the output of each of the following program when it is executed.

(a)

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

```
using namespace std;
int main()
{
    string str;
     oat a = 1,b = 0.0001,quotient;
    try {
        if (b == 0)
           throw str;
        else
        {
           quotient = a/b;
           cout << "a = " << a << " b = " << b;
           cout <<" Quotient = " << quotient << endl;
        }
    }
    catch (string str) {
        cout << "divide by zero";
    }
    return 0;
}
```

(b)

```
#include <iostream>
#include <string>
using namespace std;
int a[10] = {1,2,3,4,5,6,7,8,9,10};
int main()
{
    string str;
    int sum = 0;
    try {
        for (int i = 0; i <= 9; ++i) {
          if (i >= 5)
             throw str;
          else
          {
             cout <<"element[" << i << "] = " << a[i];
             cout << endl;
             sum += a[i];
          }
        }
    }
    catch (string str) {
         cout << "Memory out of range\n";
    }
    cout << "sum = " << sum << endl;
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
class abc {
    public:
       void display(int x, int y);
};
```

```
void abc :: display(int x, int y)
{
    try
    {
        if (y == 0)
            throw y;
        else
            cout << " x/y = " << (double) x/ (double) y << endl;
    }
    catch (int)
    {
           cout << "Exception occurred \n";
           throw;
    }
}
int main()
{
    abc obj;
    try {
        obj.display(1,2);
    }
    catch (int)
    {
        cout <<"Exception occurred in main \n";
    }
    return 0;
}
```

3. What will be the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
int a = 10;
namespace X {
    int a = 20;
}
namespace Y {
    int a = 30;
}
int main()
{
    ++::a;
    ++X::a;
    ++Y::a;
    cout << "::a = " << ::a << '\n';
    cout << "X::a = " << X::a << '\n';
    cout << "Y::a = " << Y::a << '\n';
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
int a = 10;
namespace X {
    int a = 20;
```

```
}
namespace Y {
    int a = 30;
}
int main()
{
    cout << "::a = " << ::a-- << '\n';
    cout << "X::a = " << X::a-- << '\n';
    cout << "Y::a = " << Y::a-- << '\n';
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
int a = 10;
namespace X {
    int a = 20;
}
namespace Y {
    int a = 30;
}
int main()
{
    cout << "::a = " << --::a << '\n';
    cout << "X::a = " << ++X::a << '\n';
    cout << "Y::a = " << Y::a++ << '\n';
    return 0;
}
```

(d)

```
#include <iostream>
using namespace std;
namespace {
    int a = 20;
}
namespace {
    int *ptr;
}
int main()
{
    ptr = &a;
    cout << "*ptr = " << *ptr << '\n';
    return 0;
}
```

(e)

```
#include <iostream>
using namespace std;
namespace {
    int a = 20;
}
namespace {
    int *ptr;
}
int main()
{
```

```
    ptr = &a;
    cout << "++*ptr = " << ++(*ptr) << '\n';
    return 0;
}
```

(f)

```
#include <iostream>
using namespace std;
namespace X {
    int a = 20;
}
namespace Y {
    int *ptr;
}
int main()
{
    Y::ptr = &(++X::a);
    cout << "++*ptr = " << *(Y::ptr) << '\n';
    return 0;
}
```

(g)

```
#include <iostream>
using namespace std;
namespace X {
    int a;
    namespace Y {
        int a;
   }
}
int main()
{
    X::a = 10;
    Y::a = 20;
    cout << "X::a = " << X::a << '\n';
    cout << "Y::a = " << Y::a << '\n';
    return 0;
}
```

(h)

```
#include <iostream>
using namespace std;
namespace  {
    int a;
    namespace  {
        int b;
    }
}
int main()
{
    a = 10;
    b = 20;
    cout << "a = " << a << '\n';
    cout << "b = " << b << '\n';
    return 0;
}
```

4. Determine the output of each of the following program when it is executed.

(a)

```
#include <iostream>
using namespace std;
namespace  {
    int a;
    namespace  {
        int a;
    }
}
int main()
{
    a = 10;
    cout << "a = " << a << '\n';
    return 0;
}
```

(b)

```
#include <iostream>
using namespace std;
namespace A {
    int a = 10;
    void display();
}
namespace B {
    int a = 100;
    void display();
}
void A:: display()
{
    cout << "A :: display \n";
    cout << ++(A::a) << endl;
}

void B:: display()
{
    cout << "B :: display\n";
    cout << B::a++ << endl;
}

int main()
{
    A::display();
    B::display();
    return 0;
}
```

(c)

```
#include <iostream>
using namespace std;
namespace X {
    int a = 10;
    namespace Y {
        int a = 20;
    }
}
```

```
int main()
{
    cout << "X::a = " << ++X::a << '\n';
    cout << "X::Y::a = " << ++X::Y::a << '\n';
    return 0;
}
```

(d)

```
//using unnamed namespace
#include <iostream>
using namespace std;
namespace {
    int a = 10;
}
namespace {
    int b = 100;
}
void display1()
{
    cout << "display1 = ";
    cout << ++a << endl;
}

void display2()
{
    cout << "display2 = ";
    cout << ++b << endl;
}

int main()
{
    display1();
    display2();
    return 0;
}
```

(e)

```
//using unnamed namespace
#include <iostream>
using namespace std;
namespace {
   int a = 10;
   namespace {
       int a = 20;
       namespace {
          int a = 30;
          namespace {
             int a = 40;
          }
       }
   }
}
void display()
{
    cout << " ++a = " << ++a << endl;
```

```
}
int main()
{
    display();
    return 0;
}
```

(f)

```
// using
#include <iostream>
using namespace std;
namespace  rst
{
    int x = 5;
    int y = 10;
}
namespace second
{
    double x = 3.1416;
    double y = 2.7183;
}
int main ()
{
    using  rst::x;
    using second::y
    cout << x << endl;
    cout << y << endl;
    cout <<  rst::y << endl;
    cout << second::x << endl;
    return 0;
}
```

(g)

```
// using namespace
#include <iostream>
using namespace std;
namespace  rst
{
    int x = 5;
}
namespace second
{
    double x = 3.1416;
}
int main ()
{
  {
    using namespace  rst;
    cout << x << endl;
  }
  {
    using namespace second;
    cout << x << endl;
  }
  return 0;
}
```

PROGRAMMING EXERCISES

1. Write a program in C++ to read a set of integers up to `n`, where `n` is defined by the user and store it in a one-dimensional array. Also read a set of floating point numbers of the same size and store it into another array and print the contents of these arrays separately using the function template technique.
2. Write a program in C++ to perform the following using the function template concepts:
   (i) to read a set of integers
   (ii) to read a set of floating point numbers
   (iii) to read a set of double numbers individually.
   Find out the average of the nonnegative integers and also calculate the deviation of the numbers.
3. Write a program in C++ using the function template concept to read a set of integers and floating point numbers separately and store it in the corresponding arrays. Again read a number 'd' from the keyboard and check whether the number 'd' is present in the arrays. If it is so, print how many times the number d is repeated in the array.
4. Write a program in C++ using function template to read two matrices of different data types such as integers and floating point values and find the sum of the matrices of integers and floating point numbers separately, and display the total sums of these arrays individually.
5. Develop a program in C++ using function template to perform matrix addition of two given integer matrices, two floating point number matrices and double precision value matrices separately.
6. Develop a program in C++ using function template to perform matrix subtraction of two given integer matrices, two floating point number matrices and double precision value matrices separately.
7. Develop a program in C++ using function template to perform matrix multiplication of two given integer matrices, two floating point number matrices and double precision value matrices separately.
8. Write a program in C++ using operator template for the binary numbers to perform a simple arithmetic operations such as add, subtract, multiply and divide.
9. Write an object-oriented program in C++ using a class template to read any five parameterised data type such as float and integer, and print the average.
10. Write an object-oriented program in C++ to read a set of numbers up to `n`, where `n` is defined by the user and print the contents of the array in the reverse order using a class template.

Data File Operations

Chapter 16

In this chapter, creation and accession of files from secondary storage devices using the C++ language are explained. C++ supports different types of levels to access a file from the diskette, depending upon the nature of data. This chapter gives the complete information on file operations using C++.

16.1 OPENING AND CLOSING OF FILES

File is a collection of data or a set of characters or may be a text or a program. Basically, there are two types of files in the C++: *sequential files* and *random access files*. The sequential files are very easy to create than random access files. In sequential files the data or text will be stored or read back sequentially. In random access files, data can be accessed and processed randomly.

The header file, <fstream> supports the highly sophisticated input/output stream processing techniques and to implement input/output for the advanced language features such as classes, derived classes, function overloading, virtual function and multiple inheritance.

This section describes how to open and close a data file using the header file <fstream>. The C++ Input and Output (I/O) class package handles file I/O as much as it handles standard input and output. The following methods are used in C++ to read and write files:

`ifstream` - to read a stream of object from a specified file
`ofstream` - to write a stream of object on a specified file
`fstream` - both to read and write a stream of objects on a specified file

The header file <fstream> is a new class which consists of basic file operation routines and functions. The fstream, ifstream and ofstream are called as derived class as these class objects are already defined in the basic input and output class namely <iostream>.

16.1.1 Opening a File

The following examples illustrate how files can be opened for reading and writing in C++. The member function open () is used to create a file pointer for opening a file in the disk.

(a) ifstream The header file <ifstream> is a derived class from the base class of istream and is used to read a stream of objects from a file. The following program segment shows how a file is opened to read a class of stream objects from a specified file:

```
#include <fstream>
using namespace std;
int main()
{
    ifstream  in le;
    in le.open("data_ le"); // opening a  le
    -------
    -------
    return 0;
}
```

(b) ofstream The header file <ofstream> is derived from the base class of ostream and is used to write a stream of objects in a file. The following program segment illustrates how a file is opened to write a class of stream objects on a specified file:

```
#include <fstream>
using namespace std;
int main()
{
    ofstream  in le;
    in le.open("data_ le");
    -------
    -------
    return 0;
}
```

(c) fstream The header file <fstream> is a derived class from the base class of iostream and is used for both reading and writing a stream of objects on a file. The include <fstream> automatically includes the header file <iostream>. The following program segement shows how a file is opened for both reading and writing a class of stream objects from a specified file.

```
#include <fstream>
using namespace std;
int main()
{
    fstream  in le;
    in le.open("data_ le", ios::in | ios::out);
    -------
    -------
    return 0;
}
```

When a file is opened for both reading and writing, the I/O streams keep track of two file pointers — one for input operation and the other for output operation.

Note that for an instance istream (input), the default mode is ios:: in; for ofstream instance, the default mode is ios:: out . However, for an fstream (input/output) instance, there is no default mode. The bitwise OR operator is used to declare more than one mode. Following table 16.1 is the list of member functions used as file attributes for the various kinds of file opening operation:

Table 16.1

Name of the member function	*Meaning*
`ios:: in`	open a file for reading
`ios:: out`	open a file for writing
`ios:: app`	append at the end of a file
`ios:: ate`	seek to the end of a file upon opening instead of beginning
`ios:: trunc`	delete a file if it exists and recreate it
`ios:: nocreate`	open a file if a file does not exist
`ios:: replace`	open a file if a file does exist
`ios:: binary`	open a file for binary mode; default is text

16.1.2 Closing a File

The member function close () is used to close a file which has been opened for file processing such as to read, to write and for both to read and write. The close () member function is called automatically by the destructor functions. However, one may call this member function to close the file explicitly. The close member function will not contain any arguments. The general syntax of the close () member function is:

```
#include <fstream>
using namespace std;
int main()
{
    fstream  in le;
    in le.open("data_ le", ios::in | ios::out);
    -------
    -------
    in le.close (); // calling to close the  le
    return 0;
}
```

16.2 STREAM STATE MEMBER FUNCTIONS

In C++, file stream classes inherit a stream state member from the ios class. The stream state member functions give the information status like end of file has been reached or file open failure and so on. The following stream state member functions are used for checking the open failure if any, when one attempts to open a file from the diskette.

```
eof()
fail()
bad()
good()
```

(a) eof() The eof() stream state member function is used to check whether a file pointer has reached the end of a file character or not. If it is successful, eof() member function returns a nonzero, otherwise returns 0. The general syntax of the eof() stream state member function is:

```
#include <fstream>
using namespace std;
```

```
int main()
{
    ifstream  in le;
    in le.open("text");
    while (!in le.eof()) {
        -------
        -------
    }
    return 0;
}
```

(b) `fail()` The fail() stream state member function is used to check whether a file has been opened for input or output successfully, or any invalid operations are attempted or there is an unrecoverable error. If it fails, it returns a nonzero character. The general syntax of the fail () stream state member function is:

```
#include <fstream>
using namespace std;
int main()
{
    ifstream  in le;
    in le.open("text");
    while (!in le.fail()) {
        cout << " couldn't open a  le " << endl;
        continue;
        -------
        -------
    }
    return 0;
}
```

(c) `bad()` The bad() stream state member function is used to check whether any invalid file operations has been attempted or there is an unrecoverable error. The bad() member function returns a nonzero if it is true; otherwise returns a zero. The general syntax of the bad() stream state member function is:

```
#include <fstream>
#include <cstdlib>
using namespace std;
int main()
{
    ifstream  in le;
    in le.open("text");
    if (in le.bad() ) {
        cerr << " open failure " << endl;
        exit(1);
    }
    -------
    -------
}
```

(d) `good()` The good() stream state member function is used to check whether the previous file operation has been successful or not. The good() returns a nonzero if all stream state bits are zero. The general syntax of the good() stream state member function is:

```
#include <fstream>
#include <cstdlib>
using namespace std;
int main()
```

```
{
    ifstream  in le;
    in le.open("text");
    if (in le.good()) {
        -------
        -------
    }
}
```

16.3 READING/WRITING A CHARACTER FROM A FILE

The following member functions are used for reading and writing a character from a specified file.

```
get()
put()
```

(a) get() The get() member function is used to read an alphanumeric character from a specified file. The general syntax of the get() function is:

```
#include <fstream>
using namespace std;
int main()
{
    ifstream in le;
    char ch;
    in le.open ("text");
    -------
    -------
    while (( !in le.eof()) {
        ch = in le.get()
        -------
        -------
    } // end of while loop
}
```

(b) put() The put() member function is used to write a character to a specified file or a specified output stream. The general syntax of the put() member function is:

```
#include <fstream>
using namespace std;
int main()
{
    ofstream out le;
    char ch;
    out le.open ("text");
    -------
    -------
    while (( !iout le.eof()) {
        ch = out le.get()
        cout.put(ch) // display a character onto a screen
        -------
        -------
    }
}
```

PROGRAM 16.1

A program to demonstrate the writing of a set of lines to a specified file, namely, "text.dat".

```
//storing a text on a  le
#include <fstream>
using namespace std;
int main()
{
    ofstream out le;
    out le.open("text.dat");
    out le << " this is a test \n";
    out le << " program to store \n";
    out le << " a set of lines onto a  le \n";
    out le.close();
    return 0;
}
```

The contents of the "text.dat" file

```
this is a test
program to store
a set of lines onto a  le
```

PROGRAM 16.2

A program to illustrate the writing of a set of lines to a user-defined file where name of the file is specified in the user-defined mode.

```
//storing a text on a speci ed  le
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream out le;
    char fname[10];
    cout << " enter a  le name to be opened ?\n";
    cin >> fname;
    out le.open(fname);
    out le << " this is a test \n";
    out le << " program to store \n";
    out le << " a set of lines onto a  le \n";
    out le.close();
    return 0;
}
```

Output of the above program

```
enter a  le name to be opened
data

The contents of the  le called "data"
this is a test
program to store
a set of lines onto a  le
```

PROGRAM 16.3

A program to read a set of lines from the keyboard and to store it on a specified file.

```
//reading a text and store it on a speci ed  le
#include <iostream>
#include <fstream>
using namespace std;
const int MAX = 2000;
int main()
{
    ofstream out le;
    char fname[10],line[MAX];
    cout << " enter a  le name to be opened ?\n";
    cin >> fname;
    out le.open(fname);
    cout << " enter a set of lines and terminate with @\n";
    cin.get(line,MAX,'@');
    cout << " given input \n";
    cout << line;
    cout << " storing onto a  le ....\n";
    out le << line;
    out le.close();
    return 0;
}
```

Output of the above program

```
enter a  le name to be opened?
  data
enter a set of lines and terminate with @
  this
  is a
  test program
  by Ravich
  @
given input
  this
  is a
  test program
  by Ravich
storing onto a  le ....
```

PROGRAM 16.4

A program to demonstrate how to read a text file and to display the contents on the screen.

```
// reading and displaying a text  le
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
using namespace std;
int main()
{
    ifstream in le;
    char fname1[10];
    char ch;
    cout << " enter a  le name  ? \n";
```

```
    cin >> fname1;
    in le.open(fname1);
    if (in le.fail()) {
        cerr << " No such a  le exists \n";
        exit(1);
    }
    while ( !in le.eof()) {
        ch = (char)in le.get();
        cout.put(ch);
    }
    in le.close();
    return 0;
}
```

PROGRAM 16.5

A program to copy the contents of a text file into another.

```
//  le copy
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
using namespace std;
int main()
{
    ofstream out le;
    ifstream in le;
    char fname1[10],fname2[10];
    char ch;
    cout << " enter a  le name to be copied ? \n";
    cin >> fname1;
    cout << " new  le name ? \n";
    cin >> fname2;
    in le.open(fname1);
    if (in le.fail()) {
        cerr << " No such a  le exists \n";
        exit(1);
    }
    out le.open(fname2);
    if (out le.fail()) {
        cerr << " unable to create a  le \n";
        exit(1);
    }
    while ( !in le.eof()) {
        ch = (char)in le.get();
        out le.put(ch);
    }
    in le.close();
    out le.close();
    return 0;
}
```

Output of the above program

```
enter a  le name to be copied?
data
new  le name?
tempdata
```

PROGRAM 16.6

A program to perform the deletion of white spaces such as horizontal tab, vertical tab, space, line feed, new line and carriage return, from a text file and to store the contents of the file without white spaces on another file.

```
// deleting white spaces from a text  le
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
using namespace std;
int main()
{
    ofstream out le;
    ifstream in le;
    char fname1[10],fname2[10];
    char ch;
    cout << " enter a  le name to be copied ? \n";
    cin >> fname1;
    cout << " new  le name ? \n";
    cin >> fname2;
    in le.open(fname1);
    if (in le.fail()) {
        cerr << " No such a  le exists \n";
        exit(1);
    }
    out le.open(fname2);
    if (out le.fail()) {
        cerr << " unable to create a  le \n";
        exit(1);
    }
    while ( !in le.eof()) {
        ch = (char)in le.get();
        if (ch == ' ' || ch == '\t' || ch == '\n')
            ;
        else
            out le.put(ch);
    }
    in le.close();
    out le.close();
    return 0;
}
```

Output of the above program

```
content of the input  le
this is a
test program
by Sampath K Reddy

content of the output  le

thisisatestprogrambySampathKReddy
```

PROGRAM 16.7

A program to convert a lower case character to an upper case character of a text file.

```
// converting a lower case to upper case letters
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
#include <cctype>
using namespace std;
int main()
{
    ofstream out le;
    ifstream in le;
    char fname1[10],fname2[10];
    char ch,uch;
    cout << " enter a  le name to be copied ? \n";
    cin >> fname1;
    cout << " new  le name ? \n";
    cin >> fname2;
    in le.open(fname1);
    if (in le.fail()) {
        cerr << " No such a  le exists \n";
        exit(1);
    }
    out le.open(fname2);
    if (out le.fail()) {
        cerr << " unable to create a  le \n";
        exit(1);
    }
    while ( !in le.eof()) {
        ch = (char)in le.get();
        uch = toupper(ch);
        out le.put(uch);
    }
    in le.close();
    out le.close();
    return 0;
}
```

Input file

```
this is a
test program
by Sampath K Reddy
```

Output file

```
THIS IS A
TEST PROGRAM
BY SAMPATH K REDDY
```

16.4 BINARY FILE OPERATIONS

In C++, by default the file stream operations are performed in text mode but supports binary file operations also. A binary file is a sequential access file in which data are stored and read back one after another in the binary format instead of ASCII characters. For example, a binary file contains integer, floating point number, array of structures, etc. Binary file processing is well suited for the design and development of a complex data base or to read and write a binary information.

The text file created by C++ can be edited by an ordinary editor or by a word processor. The text file can easily be transferred from one computer system to another. On the other hand, a binary file is more accurate for numbers because it stores the exact internal representation of a value. There are no conversion errors or round off errors. Saving data in binary format can be faster as there is no conversion taking place while storing data to a file. The binary format data file normally takes less space.

However, binary format data file cannot be easily transferred from one computer system to another due to variations in the internal representation of the data from one computer to another. In order to open a binary file, it is required to use the following mode:

```
in le ("data", ios:: binary);
```

The following program segment shows how to open a binary file in C++:

```
#include <fstream>
using namespace std;
int main()
{
    ofstream out le;
    out le ("data", ios:;binary);
    -------
    -------
}
```

PROGRAM 16.8

A program to open a binary file for storing a set of numbers on a specified file.

```
// storing data on a  le
// using binary  le operations
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    ofstream out le;
    char fname[10];
     oat  x,y,temp;
    cout << " enter a  le name ? \n";
    cin >> fname;
    out le.open(fname,ios::out | ios::binary);
    x = 1.5;
    y = 10.5;
    cout << "x              temp " << endl;
    while ( x <= y) {
        temp = x*x;
        out le << x << '\t' << temp << endl;
        cout << x << '\t' << temp << endl;
        x = x+1.5;
    }
    out le.close();
    return 0;
}
```

Output of the above program

```
enter a  le name?
data
The content of the  le called "data"
1.5       2.25
3         9
4.5       20.25
6         36
7.5       56.25
9         81
10.5      110.25
```

16.5 CLASSES AND FILE OPERATIONS

Since C++ is an OOP language, it is reasonable to study how objects can be read and written onto a file. This section presents how objects can be written to and read from the external device, normally a disk. The header file `<fstream>` must be included for handling the file input and output operations. The mode of file operations such as to read, to write and both to read and write should be defined. The binary file operations required to handle the input and output are carried out using the member functions `get()` and `put()` for insertion and extraction operators. The member functions `read()` and `write()` are used to read and write a stream of objects from a specified file respectively.

(a) Reading an Object from a File The read() member function is used to get data for the stream of object from a specified file. The general syntax of the read() member function is

```
in le.read(( char *) &obj, sizeof(obj));
```

The following program segment shows how to read a class of objects from a file using read () member function:

```
// reading an object from a  le
#include <fstream>
using namespace std;
class student_info {
    protected:
        char name[20];
        int age;
        char sex;
    public:
        void getdata();
        void display();
};
int main()
{
    student_info obj;
    fstream in le;
    in le.open( "data", ios:: in );
    in le.read ((char*) &obj,sizeof(obj));
    -------
    -------
    in le.close();
}
```

(b) Writing an Object to a File The write() member function is used to save the stream of objects on a specified file. The general syntax of the write() member function is:

```
in le.write(( char *) &obj, sizeof(obj));
```

The following program segment illustrates how to write an object to a file using write() member function.

```
//writing an object on a  le
#include <fstream>
using namespace std;
class student_info {
    protected:
        char name[20];
        int age;
```

```
        char sex;
    public:
        void getdata();
        void display();
};
int main()
{
    student_info obj;
    fstream out le;
    out le.open( "data", ios::out);
    out le.write ((char*) &obj,sizeof(obj));
    -------
    -------
    out le.close();
}
```

PROGRAM 16.9

A program to read the class object of `student_info` such as name, age, sex, height and weight from the keyboard and to store them on a specified file using `read()` and `write()` member functions. Again, the same file is opened for reading and displaying the contents of the file on the screen.

```
// classes and  le operations using read and write
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;
class student_info {
    protected :
        char name[20];
        int age;
        char sex;
         oat height;
         oat weight;
    public :
        void getdata();
        void display();
};
void student_info :: getdata()
{
    cout << " Enter the following information \n";
    cout << " name :";
    cin >> name;
    cout << " age :";
    cin >> age;
    cout << " sex :";
    cin >> sex;
    cout << " Height :";
    cin >> height ;
    cout << " Weight :";
    cin >> weight;
}
void student_info :: display()
{
    cout << name << setw(5) << age << setw(10) << sex
        << setw(5) << height << setw(5) << weight << endl;
}

int main()
{
```

```
    student_info obj;
    fstream in le;
    char fname[10];
    cout << " enter a  le name to be stored ? \n";
    cin >> fname;
    in le.open(fname, ios:: in | ios::out);
    // reading from the keyboard
    obj.getdata();
    // storing onto the  le
    in le.open(fname, ios::out);
    cout << " storing onto the  le......\n";
    in le.write ((char*) &obj,sizeof(obj));
    in le.close();
    // reading from the  le
    in le.open(fname, ios::in);
    cout << " reading from the  le......\n";
    in le.read ((char*) &obj,sizeof(obj));
    obj.display();
    in le.close();
    return 0;
}
```

Output of the above program

```
enter a  le name to be stored?
data
Enter the following information
name: Ahmed.K
age: 19
sex: M
Height: 156
Weight: 50

storing onto the  le......
reading from the  le......
Ahmed.K   19   M   156   50
```

PROGRAM 16.10

A program to read a set of text from a specified file using OOPs technique.

```
// read and display a text  le
// using OOPs technique
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
using namespace std;
class abc {
    public:
        void  leread();
};

void abc::  leread()
{
    ifstream in le;
    char fname[10];
    char ch;
    cout << " enter a  le name to be opened ?\n";
    cin >> fname;
    in le.open(fname);
```

```
    if (in le.fail()) {
        cerr << " No such a  le exists \n";
        exit(1);
    }
    cout << " reading from the  le ...\n";
    while (!in le.eof()) {
        ch = (char)in le.get();
        cout.put(ch);
    }
    in le.close();
}

int main()
{
    abc obj;
    obj. leread();
    return 0;
}
```

Output of the above program

```
enter a  le name to be opened?
data.txt
reading from the  le ...
this is a test
program by Ahmed
```

PROGRAM 16.11

A program to read a set of lines from the keyboard and to store it on a specified file using OOPs technique. The same file is used to read and display its contents on the video screen.

```
// read and display a text  le
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
using namespace std;
class abc {
    public:
        char a[2000];
        void getdata();
        void display();
        void  leread();
        void  lesave();
};
void abc :: getdata()
{
    char ch;
    int i = 0;
    cout <<"enter a line of text and terminate with @\n";
    while ((ch = cin.get()) != '@') {
        a[i++] = ch;
    }
    a[i++] = '\0';
}
void abc :: display()
{
    cout << "contents of a character array \n";
    for (int i = 0; a[i] != '\0'; ++i)
        cout.put(a[i]);
}
```

```
void abc::  leread()
{
    ifstream in le;
    char fname[10];
    char ch;
    cout << " enter a  le name to be opened ?\n";
    cin >> fname;
    in le.open(fname);
    if (in le.fail()) {
        cerr << " No such a  le exists \n";
        exit(1);
    }
    cout << " reading from the  le ...\n";
    while (!in le.eof()) {
        ch = (char)in le.get();
        cout.put(ch);
    }
    in le.close();
}

void abc::  lesave()
{
    ofstream out le;
    char fname[10],ch;
    cout << " enter a  le name to be saved ?\n";
    cin >> fname;
    out le.open(fname);
    if (out le.fail()) {
        cerr << " unable to create a  le \n";
        exit(1);
    }
    cout << " saving onto the  le ...\n";
    for (int i = 0; a[i] != '\0'; ++i) {
        ch = (char)a[i];
        out le.put(ch);
    }
    out le.close();
}

int main()
{
    abc obj;
    obj.getdata();
    obj.display();
    obj. lesave();
    obj. leread();
    return 0;
}
```

Output of the above program

```
enter a line of text and terminate with @
this is
a test
program by
Sampath K Reddy
@

contents of a character array
this is
a test
program by
Sampath K Reddy
```

```
enter a  le name to be saved?
data

saving onto the  le ...
enter a  le name to be opened?
data

reading from the  le ...
this is
a test
program by
Sampath K Reddy
```

PROGRAM 16.12

A program to read a text file and find out the number of characters and lines that are stored in the given file using OOPs technique.

```
//  nding number of characters and lines
// of a given text  le
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
using namespace std;
class abc {
    public:
        void  leread();
};

void abc::  leread()
{
    ifstream in le;
    char fname[10];
    char ch;
    int nch,nln;
    cout << " enter a  le name to be opened ?\n";
    cin >> fname;
    in le.open(fname);
    if (in le.fail()) {
        cerr << " No such a  le exists \n";
        exit(1);
    }
    nch = 0; nln = 0;
    cout << " reading from the  le ...\n";
    while (!in le.eof()) {
        ch = (char)in le.get();
        if (ch == '\n')
            nln++;
        ++nch;
        cout.put(ch);
    }
    cout << "Characters = " << nch-1 << ", Lines = " << nln-1;
    cout << '\n';
    in le.close();
}

int main()
{
    abc obj;
    obj. leread();
```

```
    return 0;
}
```

Output of the above program

```
enter a  le name to be opened?
data

reading from the  le ...
this is
a test

Characters = 16, Lines = 2
```

16.6 STRUCTURES AND FILE OPERATIONS

It has already been stated that a structure is a user-defined data type whose elements are heterogeneous types. This section lays emphasis on how a structure data can be read and written from a specified file. An array of structures can be stored and accessed using file handling commands. Sometimes, it may be required to store collective structure elements and retrieve them in the similar format. The following program segment illustrates how a file is opened for reading and writing a structure data type.

PROGRAM 16.13

A program to read a data for the structure elements such as name, age, sex, height and weight from the keyboard and to store them on a specified file using read () and write () member functions. Again, the same file is opened for reading and displaying the contents of the file on the screen.

```
// structures and  le operations
#include <iostream>
#include <iomanip>
#include <fstream>
const int MAX = 200;
using namespace std;
struct school {
    char name[20];
    int age;
    char sex;
     oat height;
     oat weight;
};
int main()
{
    struct school student[MAX];
    fstream in le;
    char fname[10];
    int i,n;
    cout << " enter a  le name to be stored ? \n";
    cin >> fname;
    in le.open(fname, ios:: in | ios::out);
    // reading from the keyboard
    cout << " How many records are to be stored ? \n";
    cin >> n;
    cout << " enter the following information \n";
    for ( i=0; i<= n-1; ++i) {
        cout << " name : " ;
        cin >> student[i].name;
        cout << " age : ";
```

```
        cin >> student[i].age;
        cout << " sex : ";
        cin >> student[i].sex;
        cout << " height : ";
        cin >> student[i].height;
        cout << " weight : ";
        cin >> student[i].weight;
    }
    // storing onto the  le
    in le.open(fname, ios::out);
    cout << " storing onto the  le......\n";
    for (i=0; i<=n-1; ++i ){
        in le.write ((char*) &student[i],sizeof(student[i]));
    }
    in le.close();
    // reading from the  le
    in le.open(fname, ios::in);
    cout << " reading from the  le......\n";
    for (i=0; i<=n-1; ++i ){
        in le.read ((char*) &student[i],sizeof(student[i]));
        cout << student[i].name << setw(5) << student[i].age
             << setw(10) << student[i].sex << setw(5) <<
                student[i].height << setw(5) << student[i].weight
             << endl;
    }
    in le.close();
    return 0;
}
```

Output of the above program

```
enter a  le name to be stored?
data

How many records are to be stored?
2
enter the following information
name: Antony
age: 23
sex: M
height: 167
weight: 56

name: Velusamy
age: 22
sex: M
height: 178
weight: 67
storing onto the  le......
reading from the  le......
Antony     23    M    167    56
Velusamy   22    M    178    67
```

16.7 ARRAY OF CLASS OBJECTS AND FILE OPERATIONS

In Chapter 10 on "Classes and Objects", how to define and declare an array of class objects in C++ has been explained. In this section, how to read and write a class of objects from a specified file is detailed. It is well known that an array is a user-defined data type whose elements are homogeneous and stored in consecutive memory locations. For practical applications, an array of class objects are essential to construct

complex data base systems and hence it is meaningful to study how an array of class objects are read and written on a file.

The following program segment illustrates how to read and write an array of class objects from a file.

```
#include <fstream>
using namespace std;
int const max = 200;
class employee_info {
    protected:
        char name[20];
        int age;
    public:
        void getdata();
        void display();
};

int main()
{
    student_info obj[max];
    fstream in le;
    in le.open( "data", ios:: in | ios::out);
    // storing onto the  le
    in le.open(fname, ios::out);
        -------
        -------
    cout << " storing onto the  le......\n";
    for (i=0; i<=n-1; ++i ){
        in le.write ((char*) &obj[i],sizeof(obj[i]));
    }
        -------
        -------
    // reading from the  le
    in le.open(fname, ios::in);
    cout << " reading from the  le......\n";
    for (i=0; i<=n-1; ++i ){
        in le.read ((char*) &obj[i],sizeof(obj[i]));
        obj[i].display();
    }
    in le.close();
}
```

PROGRAM 16.14

A program to read an array of class object of student_info such as name, age, sex, height and weight from the keyboard and to store them on a specified file using read () and write () member functions. Again, the same file is opened for reading and displaying the contents of the file on the screen.

```
// array of class objects and  le operations
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX = 200;
class student_info {
```

```
    protected:
        char name[20];
        int age;
        char sex;
         oat height;
         oat weight;
    public:
        void getdata();
        void display();
};
void student_info :: getdata()
{
    cout << " name:";
    cin >> name;
    cout << " age:";
    cin >> age;
    cout << " sex:";
    cin >> sex;
    cout << " Height:";
    cin >> height ;
    cout << " Weight:";
    cin >> weight;
}

void student_info :: display()
{
    cout << name << setw(5) << age << setw(10) << sex
         << setw(5) << height << setw(5) << weight << endl;
}
int main()
{
    student_info obj[MAX];
    fstream in le;
    char fname[10];
    int i,n;
    cout << " enter a  le name to be stored ? \n";
    cin >> fname;
    in le.open(fname, ios:: in | ios::out);
    cout << " How many objects are to be stored ? \n";
    cin >>n;
    // reading from the keyboard
    cout << " Enter the following information \n";
    for (i =0; i <= n-1; ++i) {
        int j = i;
        cout << endl;
        cout << " object = " << j+1 << endl;
        obj[i].getdata();
    }
    // storing onto the  le
    in le.open(fname, ios::out);
    cout << " storing onto the  le......\n";
    for (i=0; i<=n-1; ++i){
        in le.write ((char*) &obj[i],sizeof(obj[i]));
    }
    in le.close();
    // reading from the  le
    in le.open(fname, ios::in);
    cout << " reading from the  le......\n";
    for (i=0; i<=n-1; ++i){
        in le.read ((char*) &obj[i],sizeof(obj[i]));
        obj[i].display();
    }
    in le.close();
    return 0;
}
```

Output of the above program

```
enter a  le name to be stored?
data
How many objects are to be stored?
2
Enter the following information

object = 1
name: Madasamy.K
age: 24
sex: M
Height: 189
Weight: 90

object = 2
name: Mary.L
age: 22
sex: F
Height: 156
Weight: 45

storing onto the  le......
reading from the  le......
Madasamy.K   24    M    189    90
Mary.L       22    F    156    45
```

16.8 NESTED CLASSES AND FILE OPERATIONS

In Chapter 10 on " Classes and Objects ", it has already been stated that a class can be a member of another class. When a class is declared as a member of another class, then it is called as a nested class or a class within class. When a class is declared as a member of another class, it contains only the scoping of another class. The object of the outer class does not contain the object of the inner class. In this section, the reading and writing of the nested class objects from a file is described in detail.

The following program segment illustrates how to read and write a nested class of objects from a file.

```
// array of nested class objects using  le operations
#include <fstream>
using namespace std;
class student_info {
    private:
        char name[20];
    public:
        void getbase();
        void display();
        class date {
            private:
               int year;
            public:
               void getdate();
               void show_date();
               class age_class {
                   private:
                      int age;
```

```
                    public:
                       void getage ();
                       void show_age();
                  }; // end of age_class;
          };  // end of date class
};// end of student_info class declaration

int main()
{
    student_info obj1;
    student_info::date obj2;
    student_info::date::age_class obj3;
    fstream in le;
    in le.open(fname, ios:: in | ios::out);
        -------
        -------
    // storing onto the  le
    in le.open(fname, ios::out);
    cout << " storing onto the  le......\n";
    for (i=0; i<=n-1; ++i ){
        in le.write ((char*) &obj1,sizeof(obj1));
        in le.write ((char*) &obj2,sizeof(obj2));
        in le.write ((char*) &obj3,sizeof(obj3));
    }
    in le.close();
    -------
    -------
    // reading from the  le
    in le.open(fname, ios::in);
    cout << " reading from the  le......\n";
    for (i=0; i<=n-1; ++i ){
        in le.read ((char*) &obj1,sizeof(obj1));
        in le.read ((char*) &obj2,sizeof(obj2));
        in le.read ((char*) &obj3,sizeof(obj3));
        obj1.display();
        obj2.show_date();
        obj3.show_age();
    }
    in le.close();
}
```

PROGRAM 16.15

A program to demonstrate how to read data from an array of nested class objects from the keyboard and to write it in a specified file.

```
// array of nested class objects using  le operations
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX = 100;
class student_info {
    private:
        char name[20];
```

```
        long int rollno;
        char sex;
    public:
        void getbase();
        void display();
        class date {
            private:
                int day;
                int month;
                int year;
            public:
                void getdate();
                void show_date();
                class age_class {
                    private:
                        int age;
                    public:
                        void getage ();
                        void show_age();
                }; // end of age_class;
        };  // end of date class declaration
};// end of student_info class declaration

void student_info :: getbase()
{
    cout << " enter a name: ";
    cin >> name;
    cout << " roll no:";
    cin >> rollno;
    cout << " sex:";
    cin >> sex;
}
void student_info::date :: getdate()
{
    cout << " enter a date of birth\n";
    cout << " day:";
    cin >> day;
    cout << " month:";
    cin >> month;
    cout << " year:";
    cin >> year;
}
void student_info::date ::age_class:: getage ()
{
    cout << " enter an age:";
    cin >>age;
}
void student_info:: display()
{
    cout << name << "     " <<'\t';
    cout << rollno << "     ";
    cout << sex << "     ";
}
void student_info::date::show_date()
{
    cout << day << '/' << month << '/' << year << '\t';
}
void student_info::date::age_class::show_age()
{
    cout << '\t'<< age << endl;
}

int main()
{
    student_info obj1[MAX];
    student_info::date obj2[MAX];
```

```
    student_info::date::age_class obj3[MAX];
    int n,i;
    fstream in le;
    char fname[10];
    cout << " enter a  le name to be stored ? \n";
    cin >> fname;
    in le.open(fname, ios:: in | ios::out);
    cout << " how many students ?\n";
    cin >> n;
    // reading from the keyboard
    cout << " enter the following information \n";
    for (i=0; i<= n-1; ++i) {
         int j = i+1;
         cout << " \n object : " << j << endl;
         obj1[i].getbase();
         obj2[i].getdate();
         obj3[i].getage();
    }
    // storing onto the  le
    in le.open(fname, ios::out);
    cout << " storing onto the  le......\n";
    for (i=0; i<=n-1; ++i){
         in le.write ((char*) &obj1[i],sizeof(obj1[i]));
         in le.write ((char*) &obj2[i],sizeof(obj2[i]));
         in le.write ((char*) &obj3[i],sizeof(obj3[i]));
    }
    in le.close();
    // reading from the  le
    in le.open(fname, ios::in);
    cout << " reading from the  le......\n";
    cout << "\n\n\n" << endl;
    cout << " Contents of the array of nested classes \n";
    cout << " ------------------------------------------------\n";
    cout << " student's_name   Roll_no  sex  date_of_birth   age \n";
    cout << " ------------------------------------------------\n";
    for (i=0; i<=n-1; ++i){
         in le.read ((char*) &obj1[i],sizeof(obj1[i]));
         in le.read ((char*) &obj2[i],sizeof(obj2[i]));
         in le.read ((char*) &obj3[i],sizeof(obj3[i]));
         obj1[i].display();
         obj2[i].show_date();
         obj3[i].show_age();
    }
    cout << " ------------------------------------------------\n";
    in le.close();
    return 0;
}
```

Output of the above program

```
enter a  le name to be stored?
data
how many students?
3
enter the following information

object: 1
enter a name: Antony
roll no: 27001
sex: M
enter a date of birth
day: 21
month: 12
year: 1980
```

```
enter an age: 26

object: 2
enter a name:  Ahmed.M
roll no: 27002
sex: M
enter a date of birth
day: 12
month: 11
year: 1981
enter an age: 25

object: 3
enter a name: Kuppusamy
roll no: 27004
sex: M
enter a date of birth
day: 10
month: 7
year: 1981
enter an age: 25

storing onto the  le......
reading from the  le......
```

Contents of the array of nested classes

```
-------------------------------------------------------------
student's_name     Roll_no    sex     date_of_birth     age
-------------------------------------------------------------
Antony               27001      M       21/12/1980       26
Ahmed.M              27002      M       12/11/1981       25
Kuppusamy            27004      M        10/7/1981       25
-------------------------------------------------------------
```

16.9 RANDOM ACCESS FILE PROCESSING

So far, how a sequential file could be declared and accessed in C++ has been explained. A sequential access file is very easy to create than a random access file. In the sequential access file, data are stored and retrieved one after another. The file pointer always move from the starting of the file to the end of file. On the other hand, a random access file need not necessarily start from the beginning of the file and move towards end of the file. Random access means moving the file pointer directly to any location in the file instead of moving it sequentially. The random access approach is often used with data base files.

In order to perform both reading and modifying an object of a data base, a file should be opened with mode of access for both to read and to write. The header file `< fstream>` is required to declare a random access file. As stated in the previous section that `fstream` is a class which is based on both the classes of `ifstream` and `ofstream`. The `fstream` inherits two file pointers, one for the input buffer and the other for the output buffer for handling a random access file both for reading and writing.

Declaring a Random Access File The random access file must be opened with the following mode of access (Table 16.2):

Table 16.2

Mode of Access	*Meaning*
`ios:: in`	in order to read a file
`ios:: out`	in order to write a file
`ios:: ate`	in order to append
`ios:: binary`	binary format

The following program segment shows how a random access file is opened for both reading and writing.

```
#include <fstream>
using namespace std;
int main()
{
    fstream  le;
     le.open(fname, ios:: in | ios:: out | ios:: ate | ios:: binary);
    -------
    -------
}
```

It is essential to open a random access file with the following mode of access in order to perform read, write and append. The file should be declared as a binary status as the data members of a class is stored in a binary format. The `fstream` inherits the following member functions in order to move the file pointer in and around the data base (Table 16.3).

Table 16.3

Enumerated Value	*File Position*
`ios::beg`	from the beginning of the file
`ios::cur`	from the current file pointer position
`ios::end`	from the end of the file

The following member functions are used to process a random access file.

(a) seekg() The seekg() member function is used to position file operations for random input operations.

For example, the following program segment shows the positioning of the file operation for a random access file

```
#include <fstream>
using namespace std;
int main()
{
    fstream in le;
    -------
    -------
    in le.seekg(40);            // goto byte number 40
    in le.seekg(40,ios::beg);   // same as the above
    in le.seekg(0,ios::end);    // goto the end of  le
    in le.seekg(0);             // goto start of the  le
    in le.seekg(-1, ios::cur);  // the  le pointer is moved
                                // back end by one byte
}
```

(b) seekp() The seekp() member function is used to position file operations for random output operations.

(c) tellg() The tellg() member function is used to check the current position of the input stream.

(d) tellp() The tellp() member function is used to check the current position of the output stream.

PROGRAM 16.16

A program to demonstrate how to read a character from a random access file using seekg() method.

```
//reading from the  le
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    fstream in le;
    char fname[20];
    char ch;
    cout << "enter a  le name ?\n";
    cin >> fname;
    in le.open(fname, ios::in | ios :: out);
    in le.seekg(5L,ios::beg);
    in le.get(ch);
    cout << "after 5 characters from beginning = " << ch;
    cout << '\n';
    in le.seekg(-10L,ios::end);
    in le.get(ch);
    cout << "10 characters before end = " << ch;
    cout << '\n';
    in le.seekg(0,ios::cur);
    in le.get(ch);
    cout << "current character = " << ch;
    in le.close();
    return 0;
}
```

Output of the above program

The file called "data.txt" consists of the following contents:

```
abcdefghijklmnopqrstuvwxyz

enter a  le name?
data.txt
after 5 characters from beginning = f
10 characters before end = r
current character = s
```

PROGRAM 16.17

A program to read a class object of student_info such as name, roll number, sex, age, height and weight from the keyboard and to store them on a specified file using write() member functions.

```
//storing records into the random access  le
#include <iostream>
#include <fstream>
using namespace std;
struct student_info {
    char name[25];
    long int rollnumber;
    char sex;
    int age;
     oat height;
     oat weight;
```

```
};
int main()
{
    fstream in le;
    student_info obj[100];
    char fname[20];
    int n;
    cout << "enter a  le name ?\n";
    cin >> fname;
    in le.open(fname, ios::out | ios :: binary);
    if (!in le)
    {
        cout << "Error in opening  le \n";
        return 0;
    }
    cout << " How many records ? \n";
    cin >> n;
    for (int i = 0; i <= n-1; ++i) {
        cout << " Name         : \n";
        cin >> obj[i].name;
        cout << " Roll Number : \n";
        cin >> obj[i].rollnumber;
        cout << " Sex          : \n";
        cin >> obj[i].sex;
        cout << " Age          : \n";
        cin >> obj[i].age;
        cout << " Height       : \n";
        cin >> obj[i].height;
        cout << " Weight       : \n";
        cin >> obj[i].weight;
    }
    cout << " storing data onto the  le ...\n";
    for (int i = 0; i <= n-1; ++ i) {
        in le.write((char *) &obj[i],sizeof(obj[i]));
    }
    in le.close();
    return 0;
}
```

Output of the above program

```
enter a  le name?
data
How many records?
1
Name          : Velusamy
Roll Number   : 27001
Sex           : M
Age           : 24
Height        : 167
Weight        : 65
storing data onto the  le ...
```

PROGRAM 16.18

A program to demonstrate how to read a class object of student_info such as name, roll number, sex, age, height and weight from a random access file using seekg() method.

```
//reading records from the random access  le
#include <iostream>
#include <fstream>
using namespace std;
```

```
struct student_info {
    char name[25];
    long int rollnumber;
    char sex;
    int age;
     oat height;
     oat weight;
};
//function prototype
long bytesize(int recordnumber);
void display (student_info obj);

int main()
{
    fstream in le;
    student_info obj;
    char fname[20];
    char ch;
    cout << "enter a  le name ?\n";
    cin >> fname;
    in le.open(fname, ios::in | ios :: binary);
    if (!in le)
    {
        cout << "Error in opening  le \n";
        return 0;
    }
    cout << "The  rst Record : " << '\n';
    in le.seekg(bytesize(0),ios::beg);
    in le.read((char *) &obj,sizeof(obj));
    display(obj);
    cout << "The second Record : " << '\n';
    in le.seekg(bytesize(1),ios::beg);
    in le.read((char *) &obj,sizeof(obj));
    display(obj);

    cout << "The Last Record : " << '\n';
    in le.seekg(bytesize(-1),ios::end);
    in le.read((char *) &obj,sizeof(obj));
    display(obj);
    in le.close();
    return 0;
}

long bytesize(int recordnumber)
{
    return (sizeof(student_info) *recordnumber);
}

void display(student_info obj)
{
    cout << "Name :" << obj.name << '\n';
    cout << "Roll Number:" << obj.rollnumber;
    cout << '\n';
    cout << "Sex :" << obj.sex << '\n';
    cout << "Age :" << obj.age << '\n';
    cout << "Height :" << obj.height << '\n';
    cout << "Weight :" << obj.weight << '\n';
}
```

Output of the above program

```
enter a  le name?
data

The  rst Record:
Name         : Ravich
```

```
Roll Number  : 27001
Sex          : M
Age          : 21
Height       : 167
Weight       : 78

The second Record:
Name         : Ahmed
Roll Number  : 27002
Sex          : M
Age          : 23
Height       : 145
Weight       : 67

The Last Record:
Name         : Antony
Roll Number  : 27004
Sex          : M
Age          : 22
Height       : 182
Weight       : 89
```

REVIEW QUESTIONS

1. Explain the salient features of the `<fstream>` header file in C++.
2. Explain the various functions involved in opening and closing a sequential file in C++.
3. Explain the following character input and output file functions.
   (i) `get()` (ii) `put()`
4. How are the string handling features supported in the `<fstream>` header file in C++?
5. Explain the syntactic rules for the following functions.
   (i) `fstream` (ii) `ofstream` (iii) `ifstream`
6. Explain the following member functions.
   (i) `eof()` (ii) `fail()`
   (iii) `bad()` (iv) `good()`
7. Explain the syntactic rules for the following functions.
   (i) `open()` (ii) `close()`
   (iii) `read()` (iv) `write()`
8. Explain the merits and demerits of random access file processing in C++.
9. Explain the syntactic rules for the following random access file member functions.
   (i) `tellg()` (ii) `teelp()`
   (iii) `seekp()` (iv) `seekg();`
10. What is a binary file? List the merits and demerits of the binary file usage in C++.
11. Explain how a random access file is defined and processed.
12. Explain how an array of class objects can be stored and retrieved from a file.
13. Describe how a class object can be written on a file.
14. What are the syntactic rules for reading an object from the file?
15. Explain how nested class objects can be defined and accessed using file commands.
16. What are the scoping rules for the nested class objects when using a file?

PROGRAMMING EXERCISES

1. Write a program in C++ to read a file and to (i) display the contents of the file on to the screen, (ii) display the number of characters; and (iii) the number of lines in the file.
2. Write a program in C++ to read a file and to display the contents of the file on the screen with line numbers.
   For example,
   ```
   1   this is a test
   2   program
   3   -------
   4   -------
   ```
3. Write a program in C++ to merge two files into a one file heading.
4. Write a program in C++ to read a file and to transfer the contents of the file to the printer with the line numbers.
   For example,
   ```
   1   this is a test
   2   program
   3   -------
   4   -------
   ```
5. Write a program in C++ to read students' record such as name, sex, roll number, height, and weight from the specified file and to display in a sorted order (name is the key for sorting).
6. Write a program in C++ using a random access file function to create a database of the student's information such as name, roll number, sex, address and the program should have the following facilities:
   (i) to list the entire data base
   (ii) to display only a particular record
   (iii) to update a record
   (iv) to delete a record
   (v) to sort a record (name is a key reference)
7. Write a program in C++ using a random access file function to create a data base for a reservation system (bus/railway/air) using the information such as
   name of the passenger
   sex
   age
   starting place of the journey
   destination etc.
   The program should have the following facilities:
   (i) to display an entire passenger's list
   (ii) to display only a particular record
   (iii) to update a record
   (iv) to delete a record
   (v) to sort a record (name is a key for sorting)

STL–Containers Library

Chapter 17

This chapter deals with the C++ Standard Template Library (STL) which is now a built-in part of ANSI C++ compilers. The STL represents a main thrust of generic programming in C++. This chapter focusses mainly on dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority-queue), linked lists (list), trees (set), associative arrays (map) which are very commonly used in programming.

17.1 INTRODUCTION

A container is a holder object that stores a collection other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements. The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers). The STL containers are classified into three major categories:

- Sequence containers
- Associative containers
- Container adapters

(a) Sequence Containers A sequence container is one of the STL-containers that organises a finite set of objects of the same type. The elements of the sequence containers are arranged into a strictly linear fashion. STL provides three basic kinds of sequence containers: Vectors, Lists and Deques where Deque is an abbreviation for Double Ended Queue.

A vector provides a sequence of items implemented as an array that can automatically grow as needed during program execution. A deque provides a sequence of items that has a front and back: items can be efficiently added or removed from the front and back. A list gives a sequence of items that allows quick additions and removals from any position.

```
    Types of
Sequence containers
        -|-- vector
         |
         |-- deque
         |
         |-- list
```

(b) Container Adaptors Container adaptors are not full container classes because they do not provide the actual data structure implementation in which elements can be stored. The STL container adaptors do not support iterators. The container adaptors can be classified into three types, namely, stack, queue and priority_queue.

A stack provides a last-in-first-out (LIFO) data structure. A queue is a first-in-first-out (FIFO) data structure. A priority_queue gives a first-in-first-out (FIFO) data structure with the highest priority item always at the front of the priority_queue.

```
    Types of
Container adaptors
        -|-- stack
         |
         |-- queue
         |
         |-- priority_queue
```

(c) Associative Containers An associative container is non-sequential but uses a key to access elements. The keys, typically a number or a string, are used by the container to arrange the stored elements in a specific order. For example in a dictionary the entries are ordered alphabetically.

The associative containers can be classified into five types such as set, multiset, map, multimap and bitset. A map supports a collection of keys and values associated with the keys. The key and values can be of different types. A multimap is also a map that can support multiple values for each key. A set supports a sorted set of unique members. A multilist is also a set that can support multiple values per key. A bitset is a data construct that allows the user to set, reset, and check individual bits.

```
    Types of
Associative containers
        -|-- set
         |
         |-- multiset
         |
         |-- map
         |
         |-- multimap
         |
         |-- bitset
```

17.2 VECTOR CLASS

The STL vector class is a template class of sequence containers that arrange elements of a given type in a linear arrangement and allow fast random access to any element. They should be the preferred container for a sequence when random-access performance is at a premium. Vectors allow constant time insertions and deletions at the end of the sequence. Inserting or deleting elements in the middle of a vector requires linear time.

The following header file must be included to use class vector.

```
Header: #include <vector>
```

The following program segment shows how to create a vector class object for an integer elements.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector <int> v1; // v1 is an empty vector class object

    vector <int> v2(10); // v2 is a vector class object which consists of 10
                            integer
                         // elements and default with 0

    vector <int> v3(10,2); // v3 is a vector class object which contains the
                           // elements of 10 integers and is initialized with 2
    -------
    -------
    return 0;
}
```

The following examples illustrate how to create a vector class objects for different data types.

```
vector <double> v1,v2,v3;
vector <string> str1;
vector <char> a;
vector < oat> coordinate(2);
```

The following statements are equal in C++:

`vector <int> a(10); // STL` form of vector declaration for storing 10 integers

`int a[10]; // C++` form of creating an array for storing 10 integer elements.

PROGRAM 17.1

A program to demonstrate how to create a vector class objects and fill it and display its contents on the screen.

```
//using vector class
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
int main()
{
    vector <int> a(10); // create a 10 element vector
    for (int i = 0; i <= 9; ++i)
        a[i] = i*i;
    cout << "contents of the vector class objects \n";
    for (int i = 0; i <= 9; ++i)
        cout << setw(4) << a[i];
    cout << endl;
    return 0;
}
```

Output of the above program

```
contents of the vector class objects
0   1   4   9   16   25   36   49   64   81
```

PROGRAM 17.2

A program to illustrate how to create a vector class object and intialize with a certain value during the vector declaration.

```
//using vector class
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
int main()
{
    vector <int> a(10,2); // create a 10 element vector and  ll it with 2
    cout << "contents of the vector class objects \n";
    for (int i = 0; i <= 9; ++i)
        cout << setw(4) << a[i];
    cout << endl;
    return 0;
}
```

PROGRAM 17.3

A program to illustrate how to use an iterator in a vector class object.

```
//using vector class and iterator
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
int main()
{
    vector <int> a(10,2); // create a 10 element vector and  ll it with 2
    vector <int> ::iterator i;
    cout << "contents of the vector class objects \n";
    for (i = a.begin(); i != a.end(); i++)
        cout << setw(4) << *i;
    cout << endl;
    return 0;
}
```

Output of the above program

```
contents of the vector class objects
2   2   2   2   2   2   2   2   2   2
```

17.2.1 Vector Member Functions

(a) Special Member functions

`(constructor)`	Construct vector
`(destructor)`	Vector destructor
`operator=`	Copy vector content

(b) Iterators

`begin`	Return iterator to beginning
`end`	Return iterator to end
`rbegin`	Return reverse iterator to reverse beginning
`rend`	Return reverse iterator to reverse end

(c) Capacity

`size`	Return size
`max_size`	Return maximum size
`resize`	Change size
`capacity`	Return size of allocated storage capacity
`empty`	Test whether vector is empty
`reserve`	Request a change in capacity

(d) Element Access

`operator[]`	Access element
`at`	Access element
`front`	Access first element
`back`	Access last element

(e) Modifiers

`assign`	Assign vector content
`push_back`	Add element at the end
`pop_back`	Delete last element
`insert`	Insert elements
`erase`	Erase elements
`swap`	Swap content
`clear`	Clear content

(f) Allocator

`get_allocator`	Get allocator (public member function)

17.3 DOUBLE ENDED QUEUE (DEQUE) CLASS

The `deque` is one of the sequence containers of STL that arranges elements of a given type in a linear arrangement. It also allows fast random access to any element and efficient insertion and deletion at the back of the container. However, unlike a vector, the `deque` class also supports efficient insertion and deletion at the front of the container. The deque (usually pronounced "deck") is an irregular acronym of a double-ended queue. Deque sequences have the following properties:

- Individual elements can be accessed by their position index.
- Iteration over the elements can be performed in any order.
- Elements can be efficiently added and removed from any of its ends (either the beginning or the end of the sequence).

The following header file must be included to use class deque.

```
Header: #include <deque>
```

For example, the following program illustration shows how to declare the various forms of `deque` class objects in STL.

```
#include <iostream>
#include <deque>
using namespace std;
int main()
{
    deque <int> d1;  // creates an empty deque class object d1

    deque <int> d2(10); // creates deque class object d2 which stores 10]
                        // integers with default value of 0
```

```
    deque <int> d3(10,30); // creates deque class object d3 which stores 10
                           // integers with the initial value of 30
    -------
    -------
    return 0;
}
```

The following declaration shows how to create a deque class object for the various data types:

```
deque < oat> f1;
deque <char> ch1,ch2;
deque <string> str1(100);
```

PROGRAM 17.4

A program to demonstrate how to create a deque class objects and fill it and display its contents on the screen.

```
#include <iostream>
#include <iomanip>
#include <deque>
using namespace std;
int main()
{
    deque <int> d1(10,20);
    cout <<"contents of the deque\n";
    for (int i = 0; i <= 9; ++i)
        cout << setw(4) << d1[i];
    cout << endl;
    return 0;
}
```

Output of the above program

```
20   20   20   20   20   20   20   20   20   20
```

PROGRAM 17.5

A program to illustrate how to use the different member functions in the deque class object.

```
//using assign() member function
#include <deque>
#include <iostream>
using namespace std;
int main( )
{

    deque <int> d1,d2;
    deque <int>::const_iterator ptr;
    d1.push_back(10);
    d1.push_back(20);
    d1.push_back(30);
    d2.push_back(40);
    d2.push_back(50);
    d2.push_back(60);

    cout << "\n contents of deque d1 = ";
    for (ptr = d1.begin(); ptr != d1.end(); ptr++)
        cout << " " << *ptr;
```

```
    cout << endl;

    d1.assign(d2.begin(), d2.end());
    cout << "\n contents of d1  (after copy from d2) = ";
    for (ptr = d1.begin(); ptr != d1.end(); ptr++)
        cout << " " << *ptr;
    cout << endl;

    d1.assign(6,3);
    cout << "\n contents of d1 = ";
    for (ptr = d1.begin(); ptr != d1.end(); ptr++)
        cout << " " << *ptr;
    cout << endl;
    return 0;
}
```

Output of the above program

```
contents of deque d1 = 10   20   30
contents of d1 (after copy from d2) = 40   50   60
contents of d1 = 3   3   3   3   3   3
```

17.3.1 Deque Member Functions

(a) Special Member Functions

`(constructor)`	Construct deque container
`(destructor)`	Deque destructor
`operator=`	Copy container content

(b) Iterators

`begin`	Return iterator to beginning
`end`	Return iterator to end
`rbegin`	Return reverse iterator to reverse beginning
`rend`	Return reverse iterator to reverse end

(c) Capacity

`size`	Return size
`max_size`	Return maximum size
`resize`	Change size
`empty`	Test whether container is empty

(d) Element Access

`operator[]`	Access element
`at`	Access element
`front`	Access first element
`back`	Access last element

(e) Modifiers

`assign`	Assign container content
`push_back`	Add element at the end
`push_front`	Insert element at beginning
`pop_back`	Delete last element
`pop_front`	Delete first element
`insert`	Insert elements

`erase`	Erase elements
`swap`	Swap content
`clear`	Clear content

(f) Allocator

`get_allocator`	Get allocator

17.4 LIST CLASS

The list class is one of sequence containers that maintain their elements in a linear arrangement and allows efficient insertions and deletions at any location within the sequence. The sequence is stored as a bidirectional linked list of elements, each containing a member of some type Type. List containers are implemented as doubly-linked lists. Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept by the association to each element of a link to the element preceding it and a link to the element following it.

The list container provides the following advantages over vector or deque classes:

- Insertion and removal of elements can be done anywhere in the container (constant time).
- Moving elements and block of elements within the container or even between different containers (constant time).
- Iterating over the elements in forward or reverse order (linear time).

The main drawback of lists compared to these other sequence containers is that they lack direct access to the elements by their position; For example, to access the sixth element in a list one has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between them. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

The following header file must be included to use class list.

```
Header: #include <list>
```

For example, the following program segment illustrates how to create a list objects in different ways:

```
//using list class object
#include <iostream>
#include <iomanip>
#include <list>
using namespace std;
int main()
{
    list <int> abc;  //creates an empty list class object abc

    list <int> xy(10);// creates a list class object that contains 10
                      integers.
          // by default all of its elements are initialized to zero

    list <int> ab(10,3); // creates a list class object that contains 10
                         // integers and that all elements are initialized by 3
   -------
   -------
    return 0;
}
```

PROGRAM 17.6

A program to demonstrate how to create a list class objects and fill it and display its contents on the screen.

```
//using list class object
#include <iostream>
#include <iomanip>
#include <list>
using namespace std;
int main()
{
    list <int> a(10,3);
    cout <<"contents of the list \n";
    for (int i = 0; i <= 9; ++i)
        cout << setw(4) << a.back();
    cout << endl;
    return 0;
}
```

Output of the above program

```
3   3   3   3   3   3   3   3   3   3
```

PROGRAM 17.7

A program to illustrate how to use the different member functions in the list class object.

```
//using list class object
#include <iostream>
#include <iomanip>
#include <list>
using namespace std;
int main()
{
    list <int> a;
    a.push_back(10);
    a.push_back(20);
    cout << "contents of the list \n";
    cout << a.back() << setw(4);
    cout << a.front() << endl;

    return 0;
}
```

Output of the above program

```
20    10
```

17.4.1 List Member Functions

(a) Special Member Functions

`(constructor)`	Construct list
`(destructor)`	List destructor
`operator=`	Copy container content

(b) Iterators

`begin`	Return iterator to beginning
`end`	Return iterator to end
`rbegin`	Return reverse iterator to reverse beginning
`rend`	Return reverse iterator to reverse end

(c) Capacity

`empty`	Test whether container is empty
`size`	Return size
`max_size`	Return maximum size
`resize`	Change size

(d) Element access

`front`	Access first element
`back`	Access last element

(d) Modifiers

`assign`	Assign new content to container
`push_front`	Insert element at beginning
`pop_front`	Delete first element
`push_back`	Add element at the end
`pop_back`	Delete last element
`insert`	Insert elements
`erase`	Erase elements
`swap`	Swap content
`clear`	Clear content

(e) Operations

`splice`	Move elements from list to list
`remove`	Remove elements with specific value
`remove_if`	Remove elements fulfilling condition
`unique`	Remove duplicate values
`merge`	Merge sorted lists
`sort`	Sort elements in container
`reverse`	Reverse the order of elements

(f) Allocator

`get_allocator`	Get allocator

17.5 STACK CLASS

The stack class is one of the types of a template container adaptor class that provides a restriction of functionality limiting access to the element most recently added to some underlying container type. In other words, the stack class is a standard template library which supports a last-in, first-out (LIFO) data structure. Elements may be inserted, inspected, or removed only from the top of the stack, which is the last element at the end of the base container. The restriction to accessing only the top element is the reason for using the stack class.

The following header file is to be included whenever one uses the template class stack member functions or their operators in STL.

```
Header: #include <stack>
```

For example, the following ways the stack () constructor is used in STL.

```
#include <stack>
#include <vector>
#include <list>
#include <iostream>
using namespace std;
int main( )
{

    stack <int>  s1; // s1 is an empty stack class object for int data
    stack < oat> f1; // stack manipulation with  oat data
    stack <char> ch1;
    stack <bool> b1;
    -------
    -------
    return 0;
}
```

Different types of stack() construction

(1) To create a stack with deque base container, the `stack()` constructor is declared in the following form:

```
stack <char, deque<char> > abc;
```

(2) To declare a stack with vector base containers, the `stack()` constructor is used in the following form:

```
stack <int, vector<int> > va;
```

(3) To declare a stack with list base container, the `stack()` constructor is used in the following form:

```
stack <int, list<int> > lsi;
```

(4) To copy elements from a container of the base class object at the time of `stack()` declaration:

```
vector<int> v1;
v1.push_back(10);
stack <int, vector<int> > vs (v1);
```

where `vs` is the stack object which copies all elements that are stored in the vector class object `v1`.

17.5.1 Stack Member Functions

The following member functions of a stack class are used to handle the different types of operations.

```
empty()
size()
pop()
stack()
push()
top()
```

(a) `stack::push()` The push() member function of a stack class object is used to add an element to the top end of the stack. The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

The general syntax of the push() member function is:

```
void push(val);
```

where `val` is the element added to the top of the stack.

For example,

```
stack <int> s1;
s1.push(10);
s1.push(20);
s1.push(30);
```

(b) stack::pop() The pop() member function of a stack class object is used to remove the element from the top of the stack. The stack must be nonempty to apply the member function. The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

The general syntax of the pop () member function is:

```
void pop();
```

For example,

```
stack <int> s1;
int temp;
s1.push(10);
s1.push(20);
s1.push(30);
temp = s1.pop();
```

where the content of the `temp` is 30 because the stack class performs based on the LIFO (Last In First Out) method.

(c) stack::empty() The empty() member function is used to test if a stack is empty. The return value of the empty() member function of a stack class is true if the stack is empty; false if the stack is nonempty.

The general syntax of the empty() member function is:

```
bool empty() const;
```

(d) stack::size() The size() member function is used to return the number of elements in the stack. The return value of the size() member function is the current length of the stack. The general syntax of the size() member function is:

```
size_type size() const;
```

For example,

```
stack <int> s1;
stack <int>::size_type i;
s1.push(10);
s1.push(20);
s1.push(30);
i = s1.size();
```

The size() member returns the number of elements that are stored in the stack and hence the value of `i` is 3.

(e) stack::top() The top() member function is used to return a reference to an element at the top of the stack. The return value of a reference to the last element in the container at the top of the stack. The stack must be nonempty to apply the member function. The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

If the return value of top() member function is assigned to a const_reference, the stack object cannot be modified. If the return value of top() member function is assigned to a reference, the stack object can be modified.

The general syntax of the top() member function is:

```
value_type& top();
```

or

```
const value_type& top()const;
```

(f) ***stack::stack()*** The stack() constructor is used to construct a stack that is empty or that is a copy of a base container class. The general syntax of the stack () constructor is:

```
stack();
```

or

```
explicit stack(const container_type value);
```

where `value` is the container of which the constructed stack is to be a copy.

PROGRAM 17.8

A program to illustrate how to construct the stack class and use the various member functions for handling stack operations.

```
#include <stack>
#include <iostream>
using namespace std;
stack <int> s1;
stack <int> ::size_type tmp;
int main( )
{
    void menu();
    void disp_top();
    int temp;
    char ch;
    menu();
    while (( ch = cin.get()) != 'q') {
         switch (ch) {
             case 'a':
                 cout << "\n enter an integer to push into the stack = ";
                 cin >> temp;
                 s1.push(temp);
                 break;
             case 'd':
                 if (s1.empty())
                    cout <<"\n stack is empty and nothing can be popped \n";
                 else
                 {
                     cout <<"\n popped item from the stack =" << s1.top();
                     s1.pop();
                 }
                 break;
             case 's':
                 tmp = s1.size();
                 cout <<"\n number of elements in the stack = " << tmp;
                 break;
             case 'e':
                 if (s1.empty())
                     cout << "\n stack is empty \n";
                 else
                     cout << "\n stack is not empty \n";
                 break;
             case 't':
                 disp_top();
                 break;
             case 'm':
```

```
                menu();
                break;
            }
        }
}

void disp_top()
{
    if (s1.empty())
     cout << "stack is empty \n";
    else
    {
     int& i = s1.top();
     cout << "\n Top element of the stack = " << i;
     cout << "\n modi ed value of ++i = " << ++i;
     const int& j = s1.top();
     cout<< "\n Top element of the stack = " << j;
     cout << "\n ++j is not permitted due to const\n";
     s1.pop();
    }
}

void menu()
{
    cout <<"stack implementation using STL\n";
    cout << " a -> push() element into the stack \n";
    cout << " d -> pop() element from the stack \n";
    cout << " s ->  nd the number of elements in the stack\n";
    cout << " e -> check whether the stack is empty \n";
    cout << " t -> display the top element from the stack \n";
    cout << " m -> menu() \n";
    cout << " q -> quit \n";
    cout << " option, please ?\n";
}
```

17.5.2 Summary of Stack Member Functions

`(constructor)`	Construct stack
`empty`	Test whether container is empty
`size`	Return size
`top`	Access next element
`push`	Add element
`pop`	Remove element

17.6 QUEUE CLASS

The queue class is a template container adaptor class that provides a restriction of functionality limiting access to the front and back elements of some underlying container type. In other words, the queue class supports a first-in, first-out (FIFO) data structure. A good analogue to keep in mind would be people lining up for a bank teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected. The restriction to accessing only the front and back elements in this way is the reason for using the queue class.

The following header file is to be included whenever one uses the template class `queue` member functions or their operators in STL.

`Header: #include <queue>`

For example, the following ways the `queue()` constructor is used in STL.

```
#include <queue>
using namespace std;
int main( )
{

    queue <int>  s1; // s1 is an empty queue class object for int data
    queue < oat> f1; // for  oat data
    queue <char> ch1;
    queue <bool> b1;
    -------
    -------
    return 0;
}
```

17.6.1 Queue Member Functions

The following are the queue member functions which are used to handle the different operations on the queue container class object.

```
back()          push()
empty()         queue()
front()         size()
pop()
```

(a) queue::back() The back() member function of a queue container class is used to return a reference to the last and most recently added element at the back of the queue. The return value of the back() member function of a queue container is the last element of the queue. If the queue is empty, the return value is undefined.

If the return value of back is assigned to a const_reference, the queue object cannot be modified. If the return value of back is assigned to a reference, the queue object can be modified. The general syntax of the back() member function is:

```
value_type& back();
```

or

```
const value_type& back() const;
```

For example, the following program segment illustrates how to use the `back()` member function of a `queue` container class.

```
queue <int> q1;
q1.push(10);
q1.push(11);
int& i = q1.back();
```

(b) queue::empty() The empty() member function of a queue container class is used to test if a queue is empty. The return value of the empty() member function is true if the queue is empty; false if the queue is nonempty. The general syntax of the empty() member function is:

```
bool empty() const;
```

For example, the following program segment illustrates how to use the empty() member function of a queue container class.

```
queue <int> q1
q1.push(10);
if (q1.empty())
    cout << "The queue q1 is empty\n";
else
```

```
cout << "The queue q1 is not empty\n";
```

(c) queue::front() The front() member function of a queue container class is used to return a reference to the first element at the front of the queue. The return value of the front() member function is the last element of the queue. If the queue is empty, the return value is undefined.

If the return value of front() member function is assigned to a const_reference, the queue object cannot be modified. If the return value of front() member function is assigned to a reference, the queue object can be modified. The member function returns a reference to the first element of the controlled sequence, which must be nonempty. The general syntax of the front() member function is:

```
value_type& front();
```

or

```
const value_type& front() const;
```

For example, the following program segment illustrates how to use the `front()` member function of a queue container class.

```
queue <int> q1;
queue <int>::size_type i;
q1.push(10);
q1.push(20);
q1.push(30);
int& i = q1.front();
cout << "Element at the front of queue " << i;
```

(d) queue::pop The pop() member function of a queue container class is used to remove an element from the front of the queue.

The queue must be nonempty to apply the member function. The top of the queue is the position occupied by the most recently added element and is the last element at the end of the container. The general syntax of the pop() member function is:

```
void pop();
```

For example, the following program segment illustrates how to use the pop() member function of a queue container class.

```
queue <int> q1;
queue <int>::size_type i;
q1.push(10);
q1.push(20);
q1.push(30);
i = q1.size();
cout << "The queue length " << i << endl;
q1.pop();
i = q1.size();
cout << "After a pop, the queue length is " << i;
```

(e) queue::push The push() member function of a queue container class is used to add an element to the back of the queue. The top of the queue is the position occupied by the most recently added element and is the last element at the end of the container.

The general syntax of the push () member function is:

```
void push(val)
```

where `val` is the element added to the top of the queue.

For example, the following program segment illustrates how to use the push() member function of a queue container class.

```
queue <int> q1;
queue <int>::size_type i;
q1.push(10);
i = q1.size();
cout << "The queue length is " << i << endl;
q1.push(20);
q1.push(30);
i = q1.size();
cout << "The queue length is " << i << endl;
```

(f) queue::queue The queue constructor is used to construct a queue that is empty or that is a copy of a base container object.

```
queue();
```

or

```
explicit queue(const container_type& value);
```

where `value` is the const container of which the constructed queue is to be a copy.

The default base container for queue is deque. To declare a queue with default deque base container is given in the following form:

```
queue <char> q1;
```

(g) queue::size The size() member function of a queue container class is used to return the number of elements in the queue. The return value of the size member function of a queue container class is the current length of the queue. The general syntax of the size() member function is:

```
size_type size() const;
```

For example, the following program segment illustrates how to use the size() member function of a queue container class.

```
queue <int> q1;
queue <int>::size_type i;
q1.push(10);
q1.push(20);
q1.push(30);
i = q1.size();
cout << "The queue length is " << i << endl;
```

PROGRAM 17.9

A program to illustrate how to construct the queue class and use the various member functions for handling queue operations.

```
#include <queue>
#include <iostream>
using namespace std;
queue <int> q1;
queue <int> ::size_type tmp;
int main()
{
    void menu();
    int temp;
    char ch;
    menu();
    while (( ch = cin.get()) != 'q') {
        switch (ch) {
```

```
            case 'a':
                cout << "\n enter an integer to push into the queue = ";
                cin >> temp;
                q1.push(temp);
                break;
            case 'f':
                if (q1.empty())
                    cout <<"\n queue is empty and nothing can be popped \n";
                else
                {
                    cout <<"\n element at the front of the queue =" << q1.front();
                    q1.pop();
                }
                break;
            case 'b':
                if (q1.empty())
                    cout <<"\n queue is empty and nothing can be popped \n";
                else
                    cout <<"\n element at the back of the queue =" << q1.back();
                break;
            case 's':
                tmp = q1.size();
                cout <<"\n number of elements in the queue = " << tmp;
                break;
            case 'e':
                if (q1.empty())
                    cout << "\n queue is empty \n";
                else
                    cout << "\n queue is not empty \n";
                break;
            case 'm':
                menu();
                break;
            }
        }
        return 0;
}

void menu()
{
     cout <<"queue implementation using STL\n";
     cout << " a -> push() element into the queue \n";
     cout << " f -> pop() element at front of the queue \n";
     cout << " b -> pop() element at back of the queue \n";
     cout << " s ->  nd the number of elements in the queue\n";
     cout << " e -> check whether the queue is empty \n";
     cout << " m -> menu() \n";
     cout << " q -> quit \n";
     cout << " option, please?\n";
}
```

17.6.2 Summary of Queue Member Functions

`(constructor)`	Construct queue
`empty`	Test whether container is empty
`size`	Return size
`front`	Access next element
`back`	Access last element
`push`	Insert element
`pop`	Delete next element

17.7 PRIORITY_QUEUE CLASS

The `priority_queue` class orders its elements so that the largest element is always at the top position. It supports insertion of an element and the inspection and removal of the top element. A good analogue to keep in mind would be people lining up where they are arranged by age, height, or some other criterion.

A template container adaptor class that provides a restriction of functionality limiting access to the top element of some underlying container type, which is always the largest or of the highest priority. New elements can be added to the priority_queue and the top element of the priority_queue can be inspected or removed. The only requirement is that it must be accessible through random access iterators and it must support the following operations:

```
front()
push_back()
pop_back()
```

Therefore, the standard container class templates vector and deque can be used. By default, if no container class is specified for a particular priority_queue class, the standard container class template vector is used.

This context is similar to a heap where only the max heap element can be retrieved (the one at the top in the priority queue) and elements can be inserted indefinitely.

```
Header: #include <queue>
```

17.7.1 Priority_queue Member Functions

The following are the priority_queue member functions which are used to handle the different operations on the priority_queue container class object.

`empty()`	`push()`
`pop()`	`size()`
`priority_queue()`	`top()`

(a) Priority_queue::empty() The empty() member function of a priority_queue container class is used to test if a priority_queue is empty. The return value of the empty() member function is true if the priority_queue is empty; false if the priority_queue is nonempty.

The general syntax of the `empty()` member function is:

```
bool empty() const;
```

For example, the following program segment illustrates how to use the empty () member function of a priority_queue container class.

```
priority_queue <int> q1;
q1.push(10);
if (q1.empty())
   cout << "The priority_queue q1 is empty" << endl;
else
   cout << "The priority_queue q1 is not empty" << endl;
```

(b) Priority_queue::pop() The pop() member function of a priority_queue container class is used to remove the largest element of the priority_queue from the top position. The priority_queue must be nonempty to apply the member function. The top of the priority_queue is always occupied by the largest element in the container.

The general syntax of the `pop()` member function is:

```
void pop();
```

For example, the following program segment illustrates how to use the `pop()` member function of a `priority_queue` container class.

```
priority_queue <int> q1;
priority_queue <int>::size_type i;
q1.push(10);
q1.push(20);
q1.push(30);
i = q1.size();
cout << "The priority_queue length is " << i;
q1.pop( );
i = q1.size( );
cout << "After a pop, the priority_queue length is " << i << endl;
```

(c) Priority_queue::priority_queue() The priority_queue() is used to construct a priority_queue that is empty or that is a copy of a range of a base container object or of another priority_queue.

The general syntax of the priority_queue() member function is:

```
priority_queue();
```

For example, the following program segment illustrates how to use the priority_queue() constructor in a container class.

```
priority_queue <int> q2;
q2.push(5);
q2.push(15);
q2.push(10);
```

(d) Priority_queue::size() The size() member function of a priority_queue container class is used to return the number of elements in the priority_queue. In other words, the return value of the size () member function is the current length of the priority_queue. The general syntax of the size () member function is:

```
size_type size() const;
```

For example, the following program segment illustrates how to use the `size()` member function of a `priority_queue` container class.

```
priority_queue <int> q1, q2;
priority_queue <int>::size_type i;
q1.push(10);
i = q1.size();
cout << "The priority_queue length is " << i << endl;
q1.push(20);
i = q1.size();
cout << "The priority_queue length is now " << i << endl;
```

(e) Priority_queue::top() The top() member function of a priority_queue container class is used to return the largest element at the top of the priority_queue. The priority_queue must be nonempty to apply the member function. The return value is A const reference to the largest element, as determined by the Traits function, object of the priority_queue.

The general syntax of the `top ()` member function is:

```
const value_type& top() const;
```

For example, the following program segment illustrates how to use the `top()` member function of a `priority_queue` container class.

```
priority_queue<int> q1;
priority_queue<int>::size_type i;
q1.push(10);
```

```
q1.push(30);
q1.push(20);
i = q1.size( );
cout << "The priority_queue length is " << i << endl;
const int& j = q1.top();
cout << "The element at the top of the priority_queue is " << j << endl;
```

(f) Priority_queue::push() The push() member function of a priority_queue container class is used to add an element to the top of the priority_queue. The top of the priority_queue is the position occupied by the largest element in the container. The general syntax of the push() member function is:

```
void push(val);
```

where `val` is the element added to the top of the priority_queue.

For example, the following program segment illustrates how to use the push() member function of a priority_queue container class.

```
priority_queue<int> q1;
priority_queue<int>::size_type i;
q1.push(10);
q1.push(30);
q1.push(20);
i = q1.size( );
cout << "The priority_queue length is " << i << endl;
q1.push(-10);
const int& j = q1.top();
cout << "The element at the top of the priority_queue is " << ii << endl;
```

17.7.2 Summary of `priority_queue` Member Functions

`(constructor)`	Construct priority queue
`empty`	Test whether container is empty
`size`	Return size
`top`	Access top element
`push`	Insert element
`pop`	Remove top element

17.8 SET

`Set` is one of the associative container classes which is used to associate a key with each item stored and then use the key to retrieve the stored item. `Sets` are a kind of associative containers that stores unique elements, and in which the elements themselves are the keys. In other words, a `set` is used to store a `set` of keys as elements but no duplicate values are allowed.

Associative containers are containers especially designed to be efficient accessing its elements by their key (unlike sequence containers, which are more efficient accessing elements by their relative or absolute position). Internally, the elements in a set are always sorted from lower to higher following a specific strict weak ordering criterion set on container construction. Sets are typically implemented as binary search trees.

Therefore, the main characteristics of set as an associative container are:

- Unique element values
- The element value is the key itself
- Elements follow a strict weak ordering at all times.
- This container class supports bidirectional iterators.

The set class template is defined in header `<set>`.

17.8.1 Set Member Functions

(a) Special Member Functions

`(constructor)`	Construct set
`(destructor)`	Set destructor
`operator=`	Copy container content

(b) Iterators

`begin`	Return iterator to beginning
`end`	Return iterator to end
`rbegin`	Return reverse iterator to reverse beginning
`rend`	Return reverse iterator to reverse end

(c) Capacity

`empty`	Test whether container is empty
`size`	Return container size
`max_size`	Return maximum size

(d) Modifiers

`insert`	Insert element
`erase`	Erase elements
`swap`	Swap content
`clear`	Clear content

(e) Observers

`key_comp`	Return comparison object
`value_comp`	Return comparison object

(f) Operations

`nd`	Get iterator to element
`count`	Count elements with a specific key
`lower_bound`	Return iterator to lower bound
`upper_bound`	Return iterator to upper bound
`equal_range`	Get range of equal elements

(g) Allocator

`get_allocator`	Get allocator

17.9 MULTISET

Multisets are associative containers with the same properties as set containers, but allowing for multiple keys with equal values. In other words, multiset is used to store a set of keys as elements and duplicates of elements are also allowed.

The multiset object uses this expression to determine the position of the elements in the container. All elements in a multiset container are ordered following this rule at all times. The multiset class template is defined in header `<set>`.

17.9.1 Multiset Member Functions

(a) Special Member Functions

`(constructor)`	Construct multiset
`(destructor)`	Multiset destructor
`operator=`	Copy container content

(b) Iterators

`begin`	Return iterator to beginning
`end`	Return iterator to end
`rbegin`	Return reverse iterator to reverse beginning
`rend`	Return reverse iterator to reverse end

(c) Capacity

`empty`	Test whether container is empty
`size`	Return container size
`max_size`	Return maximum size

(d) Modifiers

`insert`	Insert element
`erase`	Erase elements
`swap`	Swap content
`clear`	Clear content

(e) Observers

`key_comp`	Return comparison object
`value_comp`	Return comparison object

(f) Operations

`nd`	Get iterator to element
`count`	Count elements with a specific key
`lower_bound`	Return iterator to lower bound
`upper_bound`	Return iterator to upper bound
`equal_range`	Get range of equal elements

(g) Allocator

`get_allocator`	Get allocator

17.10 MAP

Maps are a kind of associative containers that store elements formed by the combination of a key value and a mapped value. In a map, the key value is generally used to uniquely identify the element, while the mapped value is some sort of value associated to this key. Types of key and mapped value may differ. In other words, map is used to store a set of keys to data elements. Each key is associated with a unique data element and duplicate keys are not permitted.

For example, a typical example of a map is a telephone guide where the name is the key and the telephone number is the mapped value. Internally, the elements in the map are sorted from lower to higher key value following a specific strict weak ordering criterion set on construction.

The map class template is defined in header `<map>`.
The main characteristics of a map as an associative container are given below:

- Unique key values: no two elements in the map have keys that compare equal to each other.
- Each element is composed of a key and a mapped value.
- Elements follow a strict weak ordering at all times.
- Maps are also unique among associative containers in that these implement the direct access operator (operator[]) which allows for direct access of the mapped value.

17.10.1 Map Member Functions

(a) Special Member Functions

`(constructor)`	Construct map
`(destructor)`	Map destructor
`operator=`	Copy container content

(b) Iterators

`begin`	Return iterator to beginning
`end`	Return iterator to end
`rbegin`	Return reverse iterator to reverse beginning
`rend`	Return reverse iterator to reverse end

(c) Capacity

`empty`	Test whether container is empty
`size`	Return container size
`max_size`	Return maximum size

(d) Element access

`operator[]`	Access element

(e) Modifiers

`insert`	Insert element
`erase`	Erase elements
`swap`	Swap content
`clear`	Clear content

(f) Observers

`key_comp`	Return key comparison object
`value_comp`	Return value comparison object

(g) Operations

`nd`	Get iterator to element
`count`	Count elements with a specific key
`lower_bound`	Return iterator to lower bound
`upper_bound`	Return iterator to upper bound
`equal_range`	Get range of equal elements

(h) Allocator

`get_allocator`	Get allocator

17.11 MULTIMAP

The multimap is meant for multiple-key map. We have already seen that maps are a kind of associative containers that store elements formed by the combination of a key value and a mapped value, much like map containers, but allowing different elements to have the same key value.

In other words, multimap is used to map a set of keys to data elements. The same key may be associated with multiple values.

The multimap class template is defined in header `<map>`.

17.11.1 Multimap Member Functions

(a) Special Member Functions

`(constructor)`	Construct multimap
`(destructor)`	Multimap destructor
`operator=`	Copy container content

(b) Iterators

`begin`	Return iterator to beginning
`end`	Return iterator to end
`rbegin`	Return reverse iterator to reverse beginning
`rend`	Return reverse iterator to reverse end

(c) Capacity

`empty`	Test whether container is empty
`size`	Return container size
`max_size`	Return maximum size

(d) Modifiers

`insert`	Insert element
`erase`	Erase elements
`swap`	Swap content
`clear`	Clear content

(e) Observers

`key_comp`	Return key comparison object
`value_comp`	Return value comparison object

(f) Operations

`nd`	Get iterator to element
`count`	Count elements with a specific key
`lower_bound`	Return iterator to lower bound
`upper_bound`	Return iterator to upper bound
`equal_range`	Get range of equal elements

(g) Allocator

`get_allocator`	Get allocator

17.12 BITSET

A bitset is a special container class that is designed to store bits (elements with only two possible values: 0 or 1, true or false). The class is very similar to a regular array, but optimizing for space allocation: each element occupies only one bit (which is eight times less than the smallest elemental type in C++: char). Each element (each bit) can be accessed individually:

For example, for a given bitset named mybitset, the expression mybitset[3] accesses its fourth bit, just like a regular array accesses its elements.

17.12.1 Bitset Member Functions

(a) Special Member Functions

`(constructor)`	Construct bitset
`applicable operators`	Bitset operators

(b) Bit access

`operator[]`	Access bit

(c) Bit operations

`set`	Set bits
`reset`	Reset bits
`ip`	Flip bits

(d) Bitset operations

`to_ulong`	Convert to unsigned long integer
`to_string`	Convert to string
`count`	Count bits set
`size`	Return size
`test`	Return bit value
`any`	Test if any bit is set
`none`	Test if no bit is set

REVIEW QUESTIONS

1. What is a container? What are the different types of containers used in STL?
2. Explain the different types of sequence containers used in C++.
3. What is a container adaptor? What are the different types of container adaptors used in STL?
4. Elucidate the different types of associative containers used in C++.
5. Discuss how a vector class object for an integer elements is created.
6. Summarise the vector member functions used in STL.
7. Explain how a deque class object is constructed in C++.
8. Discuss the various deque member functions used in STL.
9. What is a list container? Explain how a list container is constructed in C++.
10. Summarise the list member functions used in STL.
11. What is a stack? Explain how a stack is simulated in STL.

12. Explain the following stack member functions used in C++.
(a) `empty()` (b) `size()` (c) `pop()`
(d) `stack()` (e) `push()` (f) `top()`
13. What is a queue? Explain how a queue is simulated in STL.
14. Discuss the following queue member functions used in C++.
(a) `back()` (b) `push()` (c) `empty()`
(d) `queue()` (e) `front()` (f) `size()`
(g) `pop()`
15. Explain how a priority_queue class is realised in STL.
16. Summarise the various priority_queue member functions used in C++.
17. What is a set? Explain how a set class is accomplished in STL.
18. Explain the various set member functions used in C++.
19. What is a multiset? Explain how a multiset associative container is used in C++.
20. Elucidate the various multiset member functions used in STL.
21. What is a map? Explain how a map container is used in STL.
22. Summarise the map member functions used in C++.
23. What is a multimap? Explain how a multimap container is accomplished in C++.
24. Discuss the multimap member functions used in STL.
25. What is a bitset? What are the advantages of using a bitset in a program?
26. Explain the various bitset member functions used in STL.

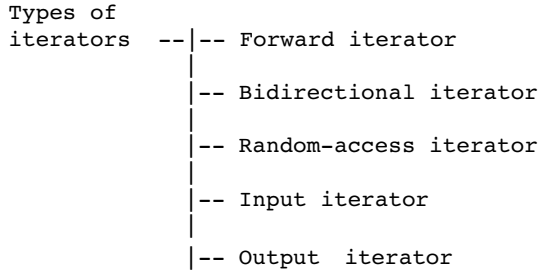
STL–Iterators and Allocators

Chapter 18

This chapter deals with the functions and characteristics of STL, namely iterators and allocators which are used to access items stored in containers.

18.1 INTRODUCTION

The ANSI C++ provides STL iterators and allocators which are mainly used to construct a generic algorithm. Elements of the containers can be accessed easily and efficiently through iterators. Iterators are objects that behave a lot like pointers. A typical iterator is an object of a class declared inside of a container class. The iterator overloads pointer operators such as the increment operator ++, the decrement operator ––, and the dereferencing operator * in order to provide pointer-like behaviour. The STL provides five iterator types, namely, forward, bidirectional, random-access, input and output.



Header: #include <iterator> The header `<iterator>` file defines the iterator primitives, predefined iterators and streamiterators, as well as several supporting templates. The predefined iterators include insert and reverse adaptors. There are three classes of insert iterator adaptors: front, back, and general.

18.2 TYPES OF ITERATORS

(1) Output Output iterators are iterators especially designed for sequential output operations, where each element pointed by the iterator is written a value only once and then the iterator is incremented. The output iterator is used forward moving and may store but not retrieve values. It is provided by `ostream` and inserter.

(2) Input Input iterators are iterators especially designed for sequential input operations, where each value pointed by the iterator is read only once and then the iterator is incremented. The input iterator is used forward moving and may retrieve but not store values. It is provided by `istream`.

(3) Forward Forward iterators are iterators especially designed for sequential access, where the algorithm passes through all the elements in the range from the beginning to the end. The forward iterator is used not only forward moving but also may store and retrieve values.

(4) Bidirectional Bidirectional iterators are iterators especially designed for sequential access in both directions — towards the end and towards the beginning. The bidirectional iterator is used not only forward and backward moving but also may store and retrieve values. It is provided by `list`, `set`, `multiset`, `map`, and `multimap`.

(5) Random Access Random access iterators are the most complete iterators in terms of functionality. Elements can be accessed in any order. The random access iterator is used both for storing and retrieving values. It is provided by `vector, deque, string,` and `array`.

18.3 <ITERATOR> MEMBER FUNCTIONS

The `<iterator>` member function provides the same functionality as standard pointers in C++. The STL iterator member functions are given below:

```
advance()
back_inserter()
distance()
front_inserter()
inserter()
```

18.3.1 advance ()

The `advance()` member function is used to increment an iterator by a specified number of positions. The function template of the advance() member function is given below:

```
template<class InputIterator, class Distance>
void advance(InputIterator& LPOS, incr);
```

where `LPOS` is the iterator that is to be incremented and that must satisfy the requirements for an input iterator; `incr` is an integral type that is convertible to the iterator's difference type and that specifies the number of increments the position of the iterator is to be advanced.

PROGRAM 18.1

A program to illustrate how to use the `advance()` member function of the iterator class in a list operations.

```
//using advance() member function
#include <iterator>
#include <list>
#include <iostream>
using namespace std;
int main()
{
    list<int> L;
    list <int>::iterator j, LPOS = L.begin();
    for (int i = 1; i < 9; ++i)
    {
        L.push_back (i*10);
    }

    cout << "Contents of the list L \n";
    for (j = L.begin(); j != L.end(); j++)
        cout << *j << " ";

    advance (LPOS, 5); //point to the 5th element
    cout << "\nAdvanced 5 steps forward to point to the fifth element = "
         << *LPOS << endl;

    return 0;
}
```

Output of the above program

```
Contents of the list L
10   20   30   40   50   60   70   80
Advanced 5 steps forward to point to the fifth element = 50
```

18.3.2 Back_inserter()

The `back_inserter()` member function is used to create an iterator that can insert elements at the back of a specified container. The function template of the back_inserter() member function is given below:

```
template<class Container>
    back_insert_iterator<Container>

    back_inserter(Container& v);
```

where `v` is the container into which the back insertion is to be executed.

PROGRAM 18.2

A program to demonstrate how to insert elements at the back of a vector container using the `back_inserter()` member function of the iterator class.

```
//using back_inserter() member function
#include <iterator>
#include <vector>
#include <iostream>
using namespace std;
int main()
{
```

```
    vector<int> v;
    vector <int>::iterator j;
    for (int i = 1; i <= 3; ++i)
    {
        v.push_back (i);
    }

    cout << "The initial vector v \n";
    for (j = v.begin(); j != v.end(); j++)
      cout << *j << "\t";
    back_inserter (v) = -40;
    back_inserter (v) = -50;
    back_inserter (v) = -60;
    cout << "\n After the back insertions, the vector v \n";
    for (j = v.begin(); j != v.end(); j++)
        cout << *j << "\t";
}
```

Output of the above program

```
The initial vector v
1   2   3
After the back insertions, the vector v
1   2   3   -40   -50   -60
```

18.3.3 Distance()

The `distance()` member function is used to determine the number of increments between the positions addressed by two iterators. The function template of the distance() member function is given below:

```
template<class InputIterator>
   typename iterator_traits<InputIterator>::difference_type
      distance(
         InputIterator _First,
         InputIterator _Last
      );
```

where `_First` is the first iterator whose distance from the second is to be determined; `_Last` is the second iterator whose distance from the first is to be determined. The return value is the number of times that `_First` must be incremented until it equals `_Last`.

PROGRAM 18.3

A program to demonstrate how to use the `distance()` member function of the iterator class in a list operations.

```
//using distance() member function
#include <iterator>
#include <list>
#include <iostream>
using namespace std;
int main()
{

    list<int> L;
    list <int>::iterator j;
    for (int i = 1; i < 9; ++i)
    {
        L.push_back (i*10);
    }
    cout << "Content of the list L \n";
```

```
    for (j = L.begin(); j != L.end(); j++)
        cout << *j << "\t";

    list<int>::difference_type Ldiff ;

    Ldiff = distance (L.begin(), L.end());
    cout << "\n The distance from L.begin( ) to L.end()= "
         << Ldiff << endl;
}
```

Output of the above program

```
Content of the list L
10   20   30   40   50   60   70   80
The distance from L.begin( ) to L.end() = 8
```

18.3.4 Front_inserter()

The `front_inserter()` member function is used to create an iterator that can insert elements at the front of a specified container. The function template of the front_inserter() member function is given below:

```
template<class Container>

    front_insert_iterator<Container>

    front_inserter(Container& v);
```

where v is the container object whose front is having an element inserted.

PROGRAM 18.4

A program to demonstrate how to insert elements at the front of a list container using the `front_inserter()` member function of the iterator class.

```
//using front_inserter() member function
#include <iterator>
#include <list>
#include <iostream>
using namespace std;
int main()
{
    list<int> L;
    list <int>::iterator j;
    for (int i = 1; i <= 5; ++i)
    {
        L.push_back (i*10);
    }
    cout << "Contents of the list L \n";
    for (j = L.begin(); j != L.end(); j++)
        cout << *j << "\t";

    front_inserter (L) = -11;
    front_inserter (L) = -22;
    front_inserter (L) = -33;

    cout << "\nAfter the front insertions, the list L \n";
    for (j = L.begin(); j != L.end(); j++)
        cout << *j << "\t";
}
```

Output of the above program

```
Contents of the list L
```

```
10   20   30   40   50
After the front insertions, the list L
-33   -22   -11   10   20   30   40   50
```

18.3.5 Inserter ()

The `inserter()` member function is an iterator adaptor that is used to add a new element to a container at a specified point of insertion. The function template of the inserter () member function is given below:

```
template<class Container, class Itererator>
   insert_iterator<Container>
   inserter(Container& v,Iterator ptr);
```

where `v` is the container to which new elements are to be added; `ptr` is an iterator locating the point of insertion.

PROGRAM 18.5

A program to demonstrate how to add a new element to a container at a specified point of insertion using `inserter ()` member function of the iterator class.

```
//using inserter() member function
#include <iterator>
#include <list>
#include <iostream>
using namespace std;
int main()
{

    list<int> L;
    list <int>::iterator j;

    for (int i = 1; i <= 5; ++i)
    {
        L.push_back (i);
    }

    cout << "Contents of the list L \n";
    for (j = L.begin(); j != L.end(); j++)
        cout << *j << "\t";

    inserter (L, L.begin()) = -60;

    cout << "\nAfter the insertions, the list L \n";
    for ( j = L.begin(); j != L.end(); j++)
        cout << *j << "\t";
}
```

Output of the above program

```
Contents of the list L
1    2    3    4    5
After the insertions, the list L
-60   1    2    3    4   5
```

18.4 OPERATORS

(a) `operator!=` operator!= is used to test if the iterator object on the left side of the operator is not equal to the iterator object on the right side.

(b) `operator==` operator == is used to test if the iterator object on the left side of the operator is equal to the iterator object on the right side.

(c) `operator<` operator< is used to test if the iterator object on the left side of the operator is less than the iterator object on the right side.

(d) `operator<=` operator<= is used to test if the iterator object on the left side of the operator is less than or equal to the iterator object on the right side.

(e) `operator>` operator> is used to test if the iterator object on the left side of the operator is greater than the iterator object on the right side.

(f) `operator>=` operator>= is used to test if the iterator object on the left side of the operator is greater than or equal to the iterator object on the right side.

(g) `operator+` operator+ is used to add an offset to an iterator and return the new reverse_iterator addressing the inserted element at the new offset position.

(h) `operator-` operator– is used to subtract one iterator from another and return the difference.

18.5 TYPES OF ITERATOR CLASSES

(a) `back_insert_iterator` The back_insert_iterator class is the template class which describes an output iterator object. It inserts elements into a container of type container, which it accesses through the protected pointer object it stores called container.

(b) `bidirectional_iterator_tag` A class that provides a return type for an iterator_category function that represents a bidirectional iterator.

(c) `front_insert_iterator` The front_insert_iterator class is the template class which describes an output iterator object. It inserts elements into a container of type container, which it accesses through the protected pointer object it stores called container.

(d) `forward_iterator_tag` A class that provides a return type for an iterator_category function that represents a forward iterator.

(e) `input_iterator_tag` A class that provides a return type for an iterator_category function that represents a bidirectional iterator.

(f) `insert_iterator` The insert_iterator class is the template class which describes an output iterator object. It inserts elements into a container of type container, which it accesses through the protected pointer object it stores called container.

(g) `istream_iterator` The istream_iterator is the template class which describes an input iterator object. It extracts objects of class from an input stream, which it accesses through an object it stores, of type pointer to basic_istream.

(h) `istreambuf_iterator` The istreambuf_iterator is the template class which describes an output iterator object. It inserts elements of class into an output stream buffer, which it accesses through an object it stores, of type pointer to basic_streambuf.

(i) `iterator` The iterator is the template class that is used as a base type for all iterators.

(j) `iterator_traits` A template helper class providing critical types that are associated with different iterator types so that they can be referred to in the same way.

(k) `ostream_iterator` The ostream_iterator is the template class which describes an output iterator object. It inserts objects of class into an output stream, which it accesses through an object it stores, of type pointer to basic_ostream.

(l) `ostreambuf_iterator` The ostreambuf_iterator is the template class which describes an output iterator object. It inserts elements of class into an output stream buffer, which it accesses through an object it stores, of type pointer to basic_streambuf.

(m) `output_iterator_tag` A class that provides a return type for iterator_category function that represents an output iterator.

(n) `random_access_iterator_tag` A class that provides a return type for iterator_category function that represents a random-access iterator.

(o) `reverse_iterator` The template class describes an object that behaves like a random-access iterator, only in reverse.

18.6 SUMMARY OF ITERATOR CLASSES

(a) Base Class

`iterator`	Iterator base class
`iterator_traits`	Iterator traits

(b) Iterator Operations

`advance`	Advance iterator
`distance`	Return distance between iterators

(c) Inserters

`back_inserter`	Construct a back insert iterator
`front_inserter`	Construct a front insert iterator
`inserter`	Construct an insert iterator

(d) Predefined iterators

`reverse_iterator`	Reverse iterator

(e) Inserter iterators

`back_insert_iterator`	Back insert iterator
`front_insert_iterator`	Front insert iterator
`insert_iterator`	Insert iterator

(f) Input/Output iterators

`istream_iterator`	Istream iterator
`ostream_iterator`	Ostream iterator
`istreambuf_iterator`	Input stream buffer iterator
`ostreambuf_iterator`	Output stream buffer iterator

REVIEW QUESTIONS

1. What is an iterator? Explain how an iterator is defined and used in C++.
2. What are the different types of iterators used in STL?
3. Explain the various iterator member functions used in C++.
4. Elucidate how an advance () member function is accomplished in C++.
5. Explain how a back_inserter() member function is defined and used in STL.
6. Explain how a distance () member function is used in STL.
7. Discuss how a front_inserter() member function is defined and used in STL.
8. Elucidate how a inserter() member function is used in STL.
9. What is a forward iterator?
10. What is a bidirectional iterator?
11. What is a random-access iterator? Give a suitable example.
12. Explain how an input iterator is defined in C++.
13. Explain how an output iterator is used in STL.
14. Explain the different types of iterator operators used in STL.
15. Summarise the various types of iterator classes used in C++.

STL–Algorithms and Function Objects

Chapter 19

This chapter presents an overview of complete STL, such as algorithms and functional objects that are implemented in ANSI C++ compiler. The Standard C++ Library provides a wide assortment of generic algorithms designed to be efficient and work with a wide variety of data types.

19.1 INTRODUCTION

The algorithms provided by the STL are implemented as function templates and perform various operations on elements of containers. There are many algorithms already implemented and tested in the STL. These algorithms use iterators to perform the tasks. The main purpose of using these STL-algorithms can cut not only the programming time but also by providing plug-in solutions.

The header `<algorithm>` defines a collection of functions especially designed to be used on ranges of elements. A range is any sequence of objects that can be accessed through iterators or pointers, such as an array or an instance of some of the STL containers.

```
Types of STL
  Algorithms ——|—— Non-mutating algorithms
               |
               |—— Mutating algorithms
               |
               |—— Sorting algorithms
               |
               |—— Set algorithms
               |
               |—— Heap operations
               |
               |—— Relational algorithms
               |
               |—— Permutations algorithms
               |
               |—— Numeric algorithms
```

(a) Non-mutating Algorithms The non-mutating algorithms are also called *non-modifying sequence operations*. These algorithms operate on a container without changing its contents.

(b) Mutating Algorithms The mutating algorithms are also called *modifying sequence operations.* These algorithms operate on a container and modify its contents.

(c) Sorted Sequence Algorithms These algorithms include sorting and merging, binary searches and operations on sorted container sequences.

(d) Heap Operation Algorithms These algorithms make it easy to sort a sequence when needed.

(e) Comparison Algorithms These algorithms allow element selection based on comparisons.

(f) Permutation Algorithms These algorithms provide ways of permutating a sequence.

(g) Generalised Numeric Algorithms One can explore these kinds of algorithms for numerics.

The following header file must be included in order to perform various operations on STL algorithms.

```
Header: #include <algorithm>
```

19.2 NON-MODIFYING SEQUENCE ALGORITHMS

The Standard C++ Library provides generic non-modifying algorithms that search for specific container elements, count container elements meeting certain criteria and check container elements for equality—all activities that do not modify the container itself. Table 19.1 gives the summary of non-modifying sequence alogorithms.

Table 19.1

Algorithm	*Description*
`for_each`	Apply function to range
`find`	Find value in range
`find_if`	Find element in range
`find_end`	Find last subsequence in range
`find_first_of`	Find element from set in range
`adjacent_find`	Find equal adjacent elements in range
`count`	Count appearances of value in range
`count_if`	Return number of elements in range satisfying condition
`mismatch`	Return first position where two ranges differ
`equal`	Test whether the elements in two ranges are equal
`search`	Find subsequence in range
`search_n`	Find succession of equal values in range

The non-modifying algorithms can be classified into the following subcategories:

- counting algorithms
- find algorithms
- search algorithms
- sequence comparison algorithms

19.2.1 Counting Algorithms

The counting algorithms provide generic counting of elements meeting specified values. The counting algorithms are:

```
count()
count_if()
```

(a) `count()` The count () algorithm counts the number of times a value appears in a sequence.

(b) `count_if()` The count_if() algorithm counts the number of times a value appears in a sequence for which the given predicate is true.

19.2.2 Find Algorithms There are several find algorithms, each targeting a certain method of finding elements in a sequence. The find algorithms are:

```
adjacent_find()
find()
find_first_of()
find_if()
for_each()
```

(a) `adjacent_find()` The adjacent_find() algorithm searches a sequence for adjacent pairs of equal elements.

(b) `find()` The find() algorithm searches a container for the first occurrence of an element.

(c) `find_first_of()` The find_first_of() algorithm finds a value from one sequence in another sequence.

(d) `find_if()` The find_if() algorithm searchs a container for the first occurrence of an element for which the given predicate is true.

(e) `for_each()` The for_each() algorithm applies a function object to each element in a sequence.

19.2.3 Search Algorithms

The Standard library includes three search algorithms:

```
find_end()
search()
search_n()
```

(a) `find_end()` The find_end() algorithm finds the last occurrence of a specified sequence as a sequence.

(b) `search()` The search () algorithm finds the first occurrence of a specified sequence as a subsequence.

(c) `search_n()` The search_n() finds the n^{th} occurrence of an element in a sequence.

19.2.4 Sequence Comparison Algorithm

The Standard library includes two sequence comparison algorithms:

```
equal()
mismatch()
```

(a) `equal()` The equal() algorithm compares two ranges of elements. It returns true only the a corresponding pairs of elements in both ranges are equal.

(b) `mismatch()` The mismatch() algorithm compares two ranges of elements. It returns a pair of iterators that are the first corresponding positions at which unequal elements occur in a pair of sequences.

19.3 MODIFYING SEQUENCE ALGORITHMS

The Standard C++ library provides generic modifying algorithms that copy, paste, remove, rotate and transform specific container elements—all activities that modify the container's contents. Table 19.2 presents the summary of modifying sequence algorithms that are supported by STL.

Table 19.2

Algorithm	*Description*
`copy`	Copy range of elements
`copy_backward`	Copy range of elements backwards
`swap`	Exchange values of two objects
`swap_ranges`	Exchange values of two ranges
`iter_swap`	Exchange values of objects pointed by two iterators
`transform`	Apply function to range
`replace`	Replace value in range
`replace_if`	Replace values in range
`replace_copy`	Copy range replacing value
`replace_copy_if`	Copy range replacing value
`fill`	Fill range with value
`fill_n`	Fill sequence with value
`generate`	Generate values for range with function
`generate_n`	Generate values for sequence with function
`remove`	Remove value from range
`remove_if`	Remove elements from range
`remove_copy`	Copy range removing value
`remove_copy_if`	Copy range removing values
`unique`	Remove consecutive duplicates in range
`unique_copy`	Copy range removing duplicates
`reverse`	Reverse range
`reverse_copy`	Copy range reversed
`rotate`	Rotate elements in range
`rotate_copy`	Copy rotated range
`random_shuffle`	Rearrange elements in range randomly
`partition`	Partition range in two
`stable_partition`	Partition range in two - stable ordering

With one exception (transform), one can further categorise the modifying algorithms by operation, giving the following subcategories:

- copy algorithms
- fill and generate algorithms
- replace algorithms
- remove algorithms
- reverse and rotate algorithms
- swap algorithms

19.3.1 Copy Algorithms

The copy algorithm copy elements from one sequence to another. Each algorithm performs a different function based on different criteria. The standard C++ supports the following two functions to perform copy algorithms.

```
copy()
copy_backward()
```

(a) `copy()` The copy() algorithm is used to copy elements from one sequence to another.

(b) `copy_backward()` The copy_backward() algorithm is used to copy elements from one sequence to another sequence starting with the last element.

19.3.2 Fill and Generate Algorithms

The fill and generate algorithms are used to systematically assign values to sequence elements. These algorithms include the following:

```
fill()
fill_n()
generate()
generate_n()
```

(a) `fill()` The fill() algorithm is used to replace each element in a sequence with a given value.

(b) `fill_n()` The fill_n() algorithm is used to replace the first *n* elements in a sequence with a given value.

(c) `generate()` The generate() algorithm is used to replace each element in a sequence with the value returned by an operation.

(d) `generate_n()` The generate_n() algorithm is used to replace the first n elements in a sequence with the value returned by an operation.

19.3.3 Replace Algorithms

The replace algorithms are used to replace values in existing sequence elements. These algorithms include the following:

```
replace()
replace_copy()
replace_copy_if()
replace_if()
```

(a) `replace()` The replace() algorithm replaces with another value each element in a sequence that matches some specified value.

(b) `replace_copy()` The replace_copy() algorithm copies a sequence to another sequence, replacing with another value each element in the sequence that matches some specified value.

(c) `replace_copy_if()` The replace_copy_if() algorithm copies a sequence to another sequence, replacing with another value each element in the sequence that matches some specified value only if a predicate returns true.

(d) `replace_if()` The replace_if() algorithm replaces with another value each element in a sequence that matches some specified value only if a predicate returns true.

19.3.4 Remove Algorithms

The remove algorithms are used to remove existing elements from a sequence. These algorithms include the following:

```
remove()
remove_copy()
remove_copy_if()
remove_if()
```

(a) `remove()` The remove() algorithm removes elements equal to a specified value.

(b) `remove_copy()` The remove_copy() copies a sequence, removing elements equal to a specified value in the new sequence.

(c) `remove_copy_if()` The remove_copy_if() copies a sequence, removing elements equal to a specified value in the new sequence if a predicate returns true.

(d) `remove_if()` The remove_if() removes elements equal to a specified value if a predicate returns true.

19.3.5 Reverse and Rotate Algorithms

The reverse and rotate algorithms are used to reord the sequence of elements in a container.

```
random_shuffle()
reverse()
reverse_copy()
rotate()
rotate_copy()
```

(a) `random_shuffle()` The random_shuffle() reorders elements into a uniformly pseudo-random order.

(b) `reverse()` The reverse() reverses the order of the elements in a sequence.

(c) `reverse_copy()` The reverse_copy() copies one sequence into another, reversing the order of the elements in the new sequence.

(d) `rotate()` The rotate() performs a circular shift of all elements in a sequence.

(e) `rotate_copy()` The rotate_copy() copies one sequence into another, performing a circular shift of all elements in the new sequence.

19.3.6 Swap Algorithms

The swap algorithms move elements from one location to another in a container. These algorithms include the following:

```
iter_swap()
swap()
swap_ranges()
```

(a) `iter_swap()` The iter_swap() algorithm swaps two elements using iterators.

(b) `swap()` The swap() algorithm exchanges two sequence elements.

(c) `swap_ranges()` The swap_ranges() algorithm exchanges elements of one sequence with those of another sequence using a specified range.

19.3.7 The `transform` Algorithm

The `transform()` algorithm performs a specified operation on every element in a sequence. This algorithm is implemented in the `transform()` function declared in the standard header `<algorithm>`.

19.3.8 Unique Algorithms

The unique algorithms remove duplicate adjacent elements within a sequence. These algorithms include the following:

```
unique()
unique_copy()
```

(a) `unique()` The unique() algorithm removes duplicate elements of a sequence if they are adjacent.

(b) `unique_copy()` The unique_copy() copies a sequence, removing duplicate elements from the new sequence if they are adjacent.

19.4 SORTED SEQUENCE ALGORITHMS

The standard C++ library provides several generic algorithms related to sorted sequences and sort operations. The following Table 19.3 briefly describes each of the sorting and searching algorithms.

Table 19.3

Algorithm	*Description*
`binary_search`	finds a value in a sorted sequence using repeated bisection.
`equal_range`	searches a sorted sequence for a subsequence having a specified value.
`inplace_merge`	merges two consecutive sorted sequences.
`lower_bound`	finds the first occurence of a value in a sorted sequence.
`includes`	returns true if a specified subsequence is present in a sequence.
`merge`	merges two sorted sequences.
`nth_element`	places a sequence element into the position it would occupy if the sequence were sorted.
`partial_sort`	sorts the first part of a sequence
`partial_sort_copy`	makes a copy of a sequence, sorting the first part of the new sequence.
`partition`	places elements matching a predicate first in the sort order.
`set_difference`	compares two sequences, creating a sorted third sequence that contains elements found in the first sequence but not in the second.
`set_intersection`	compares two sequences, creating a sorted third sequence that contains elements found in both of the first two sequences.
`set_symmetric_difference`	compares two sequences, creating a sorted third sequence that contains all elements found in the first but not the second, and all elements found in the second but not the first.
`set_union`	creates a new sequence that contains elements found in two sequences, eliminating duplicates.
`sort`	sorts a sequence
`stable_partition`	sorts a sequence, placing elements that match a predicate first while maintaining relative order.
`stable_sort`	sorts a sequence, maintaining relative order of equal elements.
`upper_bound`	finds the last occurrence of a value in a sorted sequence.

We can further categorise the sorted sequence algorithms by operation, giving the following subcategories:

- Binary search algorithms
- Merge algorithms
- Partition algorithms
- Set operation algorithms
- Sorting algorithms

19.4.1 Binary Search Algorithms

The binary search algorithms make searching sorted sequences orders of magnitude faster than linear searches on unsorted sequences. These algorithms include the following:

```
binary_search()
equal_range()
lower_bound()
upper_bound()
```

(a) `binary_search()` The binary_search() algorithm finds a value in a sorted sequence using repeated bisection.

(b) `equal_range()` The equal_range() algorithm searches a sorted sequence for a subsequence having a specified value.

(c) `lower_bound()` The lower_bound() algorithm finds the first occurrence of a value in a sorted sequence.

(d) `upper_bound()` The upper_bound() algorithm finds the last occurrence of a value in a sorted sequence.

19.4.2 Merge Algorithms

The merge algorithms combine sequences in various ways. These algorithms include `inplace_merge()` and `merge()`.

(a) `inplace_merge` The inplace_merge() algorithm merges two consecutive sorted sequences.

(b) `merge()` The merge() algorithm merges two sorted sequences.

19.4.3 `partition` Algorithms

Partition algorithms place every element that satisfies a predicate requirement before every element that does not satisfy that requirement, rearranging a sequence as needed. The partition algorithm includes partition() and stable_partition()

(a) `partition` The partition() algorithm places elements matching a predicate first in the sort order.

(b) `stable_partition` The stable_partition() algorithm sorts a sequence, placing elements that match a predicate first while maintaining relative order.

19.4.4 `set` Operation Algorithms

The algorithms that operate on sets work only with sorted sequences because typical `set` operations such as intersect and difference are terribly slow on unsorted sequences. The `set` algorithms include the following:

```
includes()
set_difference()
set_intersection()
set_symmetric_difference()
set_union()
```

(a) `includes()` The includes() algorithm returns true if a specified subsequence is present in a sequence.

(b) `set_difference()` The set_difference() algorithm compares two sequences, creating a sorted third sequence that contains elements found in the first sequence but not in the second.

(c) `set_intersection()` The set_intersection() algorithm compares two sequences, creating a sorted third sequence that contains elements found in both of the first two sequences.

(d) `set_symmetric_difference()` The set_symmetric_difference() algorithm compares two sequences, creating a sorted third sequence that contains all elements found in first the but not the second, and all elements found in the second but not the first.

(e) `set_union()` The set_union() algorithm creates a new sequence that contains elements found in two sequences, eliminating duplicates.

19.4.5 Sorting Algorithms

The algorithms that actually sort sequences include the following:

```
nth_element()
partial_sort()
partial_sort_copy()
sort()
stable_sort()
```

(a) `nth_element()` The nth_element() algorithm places a sequence element into the position it would occupy if the sequence were sorted.

(b) `partial_sort()` The partial_sort() algorithm sorts the first part of a sequence

(c) `partial_sort_copy()` The partial_sort_copy() makes a copy of a sequence, sorting the first part of the new sequence.

(d) `sort()` The sort() algorithm sorts a sequence.

(e) `stable_sort()` The stable_sort() algorithm sorts a sequence, maintaining relative order of equal elements.

19.5 HEAP OPERATION ALGORITHMS

The standard C++ library provides generic heap operation algorithms to work with heaps. The following Table 19.4 briefly describes each of the heap operation algorithms.

Table 19.4

Algorithm	*Description*
`push_heap`	Push element into heap range
`pop_heap`	Pop element from heap range
`make_heap`	Make heap from range
`sort_heap`	Sort elements of heap

(a) `make_heap()` The make_heap() algorithm creates a sequence to be used as a heap.

(b) `pop_heap()` The pop_heap() algorithm removes an element from a heap.

(c) `push_heap()` The push_heap() algorithm adds an element to a heap.

(d) `sort_heap()` The sort_heap() algorithm sorts a heap.

19.6 COMPARISON ALGORITHMS

The standard C++ library provides generic algorithms to compare sequences and elements. The following Table 19.5 briefly describes each of the comparison algorithms.

Table 19.5

Algorithm	*Description*
`min`	Return the lesser of two arguments
`max`	Return the greater of two arguments
`min_element`	Return smallest element in range
`max_element`	Return largest element in range
`lexicographical_compare`	Performs a lexicographical comparison of two sequences to determine which is first

(a) `lexicographical_compare()` The lexicographical_compare() algorithm performs a lexicographical comparison of two sequences to determine which is first.

(b) `max()` The max() algorithm returns the larger of two arguments.

(c) `max_element()` The max_element() algorithm returns an iterator to the largest value in a sequence.

(d) `min()` The min() algorithm returns the smaller of two arguments.

(e) `min_element()` The min_element() algorithm returns an iterator to the smallest value in a sequence.

19.7 PERMUTATION ALGORITHM

The standard C++ library provides two generic algorithms to perform permutations on sequences. The following Table 19.6 briefly describes both of these algorithms:

Table 19.6

Algorithm	*Description*
`next_permutation`	permutes a sequence into the next lexicographical ordering of permutations
`prev_permutation`	permutes a sequence into the previous lexicographical ordering of permutations

(a) `next_permutation()` The next_permutation() algorithm permutes a sequence into the next lexicographical ordering of permutations.

(b) `prev_permutation()` The prev_permutation() algorithm permutes a sequence into the previous lexicographical ordering of permutations.

19.8 NUMERIC ALGORITHMS

Numeric algorithms are special types of container template functions that perform algorithms provided for numerical processing. The algorithms are similar to the Standard Template Library (STL) algorithms and are fully compatible with the STL but are part of the C++ Standard Library rather than the STL. Like the STL algorithms, they are generic because they can operate on a variety of data structures.

The following header file must be included in order to perform various operations on numeric algorithms.

```
Header: #include <numeric>
```

19.8.1 The Numeric Members Functions

The standard C++ library provides four generic algorithms to perform numeric operation on sequences. The following Table 19.7 briefly describes these algorithms.

Table 19.7

Algorithm	*Description*
`accumulate()`	Accumulate values in range
`adjacent_difference()`	Compute adjacent difference of range
`inner_product()`	Compute cumulative inner product of range
`partial_sum()`	Compute partial sums of range

(a) `accumulate()` The accumulate() member function is used to compute the sum of all the elements in a specified range including some initial value by computing successive partial sums or compute the result of successive partial results similarly obtained from using a specified binary operation other than the sum.

(b) `adjacent_difference()` The adjacent_difference() member function is used to compute the successive differences between each element and its predecessor in an input range and outputs the results to a destination range or compute the result of a generalised procedure where the difference operation is replaced by another, specified binary operation.

(c) `inner_product()` The inner_product() member function is used to compute the sum of the element-wise product of two ranges and adds it to a specified initial value or compute the result of a generalized procedure where the sum and product binary operations are replaced by other specified binary operations.

(d) `partial_sum()` The partial_sum() member function is used to compute a series of sums in an input range from the first element through the ith element and stores the result of each such sum in ith element of a destination range or compute the result of a generalised procedure where the sum operation is replaced by another specified binary operation.

19.9 FUNCTION OBJECTS

```
Header: #include <functional>
```

The header file <functional> defines Standard Template Library (STL) functions that help construct function objects, also known as functors, and their binders. In other words, function objects are objects specifically designed to be used with a syntax similar to that of functions. In C++, this is achieved by defining member function operator () in their class.

A function object and function pointers can be passed as a predicate to an algorithm, but function objects are also adaptable and increase the scope, flexibility, and efficiency of the STL.

19.10 THE FUNCTIONAL MEMBERS

(a) Base Classes Algorithms require two types of function objects: unary and binary. Unary function objects require one argument, and binary function objects require two arguments. The Table 19.8 gives the summary of the functional members.

Table 19.8

Function Objects	*Description*
`unary_function`	Unary function object base class
`binary_function`	Binary function object base class

19.10.1 Operator Classes

The function objects of the operator classes can be classified into three groups, namely, (a) arithmetic operations, (b) Comparison operations, and (c) Logical operations.

(a) Arithmetic Operations (Table 19.9)

(i) `plus()` The class provides a predefined function object that performs the arithmetic operation of addition on elements of a specified value type.

(ii) `minus()` The class provides a predefined function object that performs the arithmetic operation of subtraction on elements of a specified value type.

(iii) `multiplies()` The class provides a predefined function object that performs the arithmetic operation of multiplication on elements of a specified value type.

(iv) `divides()` The class provides a predefined function object that performs the arithmetic operation of division on elements of a specified value type.

(v) `modulus()` The class provides a predefined function object that performs the arithmetic operation of modulus on elements of a specified value type.

(vi) `negate()` The class provides a predefined function object that returns the negative of an element value.

Table 19.9

Function Objects	*Description*
`plus`	Addition function object class
`minus`	Subtraction function object class
`multiplies`	Multiplication function object class
`divides`	Division function object class
`modulus`	Modulus function object class
`negate`	Negative function object class

(b) Comparison Operations (Table 19.10)

(i) `equal_to()`A binary predicate that tests whether a value of a specified type is equal to another value of that type.

(ii) `not_equal_to()` A binary predicate that tests whether a value of a specified type is not equal to another value of that type.

(iii) `greater()` A binary predicate that tests whether a value of a specified type is greater than another value of that type.

(iv) `less()` A binary predicate that tests whether a value of a specified type is less than another value of that type.

(v) `greater_equal()` A binary predicate that tests whether a value of a specified type is greater than or equal to another value of that type.

(vi) `less_equal()` A binary predicate that tests whether a value of a specified type is less than or equal to another value of that type.

Table 19.10

Function Objects	*Description*
`equal_to`	Function object class for equality comparison
`not_equal_to`	Function object class for non-equality comparison
`greater`	Function object class for greater-than inequality comparison
`less`	Function object class for less-than inequality comparison
`greater_equal`	Function object class for greater-than-or-equal-to comparison
`less_equal`	Function object class for less-than-or-equal-to comparison

(c) Logical Operations (Table 19.11)

(i) `logical_and()` The class provides a predefined function object that performs the logical operation of conjunction on elements of a specified value type and tests for the truth or falsity of the result.

(ii) `logical_or()` The class provides a predefined function object that performs the logical operation of disjunction on elements of a specified value type and tests for the truth or falsity of the result.

(iii) `logical_not()` The class provides a predefined function object that performs the logical operation of negation on elements of a specified value type and tests for the truth or falsity of the result.

Table 19.11

Function Objects	*Description*
`logical_and`	Logical AND function object class
`logical_or`	Logical OR function object class
`logical_not`	Logical NOT function object class

19.10.2 Adaptor and Conversion Functions

(a) Negators (Table 19.12)

(i) `not1()` The not1() negator is used to return the complement of a unary predicate.

(ii) `not2()` The not2() negator is used to return the complement of a binary predicate.

Table 19.12

Function Objects	*Description*
`not1`	Return negation of unary function object
`not2`	Return negation of binary function object

(b) Parameter Binders (Table 19.13)

(i) `binder1st` A template class providing a constructor that converts a binary function object into a unary function object by binding the first argument of the binary function to a specified value.

(ii) `binder2nd` A template class providing a constructor that converts a binary function object into a unary function object by binding the second argument of the binary function to a specified value.

Table 19.13

Function Objects	*Description*
`bind1st`	Return function object with first parameter binded
`bind2nd`	Return function object with second parameter binded

(c) Conversors (Table 19.14)

(i) `ptr_fun()` The ptr_fun() is a helper template function that is used to convert unary and binary function pointers, respectively, into unary and binary adaptable functions.

(ii) `mem_fun_ref()` The mem_fun_ref() is a helper template function that is used to construct function object adaptors for member functions when initialised with reference arguments.

(iii) `mem_fun()` The mem_fun() is a helper template function that is used to construct function object adaptors for member functions when initialised with pointer arguments.

Table 19.14

Function Objects	*Description*
`ptr_fun`	Convert function pointer to function object
`mem_fun`	Convert member function to function object (pointer version)
`mem_fun_ref`	Convert member function to function object (reference version)

(d) Instrumental Types (Table 19.15)

(i) `unary_negate()` A template class providing a member function that negates the return value of a specified unary function.

(ii) `binary_negate()` A template class providing a member function that negates the return value of a specified binary function.

(iii) `bind1st()` A helper template function that creates an adaptor to convert a binary function object into a unary function object by binding the first argument of the binary function to a specified value.

(iv) `bind2nd()` A helper template function that creates an adaptor to convert a binary function object into a unary function object by binding the second argument of the binary function to a specified value.

(v) `pointer_to_unary_function()` The pointer_to_unary_function() is used to to convert a unary function pointer into an adaptable unary function.

(vi) `pointer_to_binary_function()` The pointer_to_binary_function() is used to convert a binary function pointer into an adaptable binary function.

(vii) `mem_fun_t()` The mem_fun_t() is an adapter class that allows a non_const member function and that takes no arguments to be called as a unary function object when initialised with a pointer argument.

(viii) `mem_fun1_t()` The mem_fun1_t () is an adapter class that allows a `non_const` member function and that takes a single argument to be called as a binary function object when initialised with a pointer argument.

(ix) `const_mem_fun_t()` The const_mem_fun_t() is an adapter class that allows a const member function and that takes no arguments to be called as a unary function object when initialised with a pointer argument.

(x) `const_mem_fun1_t()` The const_mem_fun1_t() is an adapter class that allows a const member function and that takes a single argument to be called as a binary function object when initialised with a pointer argument.

(xi) `mem_fun_ref_t()` The mem_fun_ref_t() is an adapter class that allows a `non_const` member function and that takes no arguments to be called as a unary function object when initialised with a reference argument.

(xii) `mem_fun1_ref_t()` The mem_fun1_ref_t() is an adapter class that allows a `non_const` member function and that takes a single argument to be called as a binary function object when initialised with a reference argument.

(xiii) `const_mem_fun_ref_t()` The const_mem_fun_ref_t() is an adapter class that allows a `const` member function and that takes no arguments to be called as a unary function object when initialised with a reference argument.

(xiv) `const_mem_fun1_ref_t()` The const_mem_fun1_ref_t() is an adapter class that allows a const member function and that takes a single argument to be called as a binary function object when initialised with a reference argument.

Table 19.15

Instrumental types	*Description*
`unary_negate`	Generate negation of unary function object class
`binary_negate`	Generate negation of binary function object class
`binder1st`	Generate function object class with 1st parameter binded
`binder2nd`	Generate function object class with 2nd parameter binded
`pointer_to_unary_function`	Generate unary function object class from pointer
`pointer_to_binary_function`	Generate binary function object class from pointer
`mem_fun_t`	Generate function object class from parameterless member (pointer version)
`mem_fun1_t`	Generate function object class from single-parameter member (pointer version)
`const_mem_fun_t`	Generate function object class from const parameterless member (pointer version)
`const_mem_fun1_t`	Generate function object class from single-parameter const member (pointer version)
`mem_fun_ref_t`	Generate function object class from parameterless member (reference version)
`mem_fun1_ref_t`	Generate function object class from single-parameter member (reference version)
`const_mem_fun_ref_t`	Generate function object class from const parameterless member (reference version)
`const_mem_fun1_ref_t`	Generate function object class from single-parameter const member (reference version)

REVIEW QUESTIONS

1. Explain how the STL-algorithms are useful for constructing a generic programming.
2. What are the different types of STL-algorithms used in C++?
3. Explain the importance of non-mutating algorithms.
4. Explain the following counting algorithm with a suitable example.
   (a) `count()` (b) `count_if()`
5. Elucidate how the following STL-algorithm is used in C++.
   (a) `adjacent_find()` (b) `find()` (c) `find_first_of()`
   (e) `find_if()` (f) `for_each()`
6. Discuss the importance of the following algorithms.
   (a) `find_end()` (b) `search()` (c) `search_n()`
7. Explain how a sequence comparison algorithm is constructed in C++.
8. Summarise the list of mutating algorithms used in STL.
9. Explain the following copy algorithm with a suitable example.
   (a) `copy()` (b) `copy_backward()`

10. Explain how the following algorithm is accomplished in STL.
(a) `fill()` (b) `fill_n()`
(c) `generate()` (d) `generate_n()`

11. Discuss the importance of the following replace algorithms.
(a) `replace()` (b) `replace_copy()`
(c) `replace_copy_if()` (d) `replace_if()`

12. Explain the following remove algorithms with a suitable example.
(a) `remove()` (b) `remove_copy()`
(c) `remove_copy_if()` (d) `remove_if()`

13. Elucidate how the reverse and rotate algorithms are used to reorder the sequence of elements in a container.

14. Summarise the list of swap algorithms used in STL.

15. Discuss the following unique algorithms with a suitable example.
(a) `unique()` (b) `unique_copy()`

16. Explain how the following algorithm is accomplished in STL.
(a) `binary_search()` (b) `equal_range()`
(c) `lower_bound()` (d) `upper_bound()`

17. Summarise the list of set operation algorithms used in STL.

18. Explain the following sort algorithms with a suitable example.
(a) `nth_element()` (b) `partial_sort()` (c) `partial_sort_copy()`
(d) `sort()` (e) `stable_sort()`

19. Describe how the heap operation algorithms are constructed in C++.

20. Summarise the list of comparison algorithms used in STL.

21. Explain the following numeric algorithm with a suitable example.
(a) `accumulate()` (b) `adjacent_difference()`
(c) `inner_product()` (d) `partial_sum()`

22. Explain the importance of function objects in C++.

Appendix

A-1 SOLUTIONS TO THE CONCEPT REVIEW PROBLEMS

Chapter 3

(1) (a) invalid (no space between identifiers)
(b) invalid (special character '+')
(c) invalid (special character '')
(d) valid
(e) invalid (first character should be an alphabet)
(f) invalid (keywords cannot be used as a user-defined identifier)
(g) invalid (keywords cannot be used as a user-defined identifier)
(h) invalid (special character " ")
(i) valid
(j) invalid (no space between identifiers)
(k) valid (in ANSI/ISO C++)
(l) valid

(2) (a) invalid (comma is not allowed)
(b) invalid (decimal point not permitted)
(c) invalid (special symbols not allowed)
(d) invalid (usage of e is for real number)
(e) invalid (usage of e is for real number)
(f) valid
(g) invalid (not numeral)
(h) valid
(i) valid
(j) valid
(k) valid
(l) valid
(m) valid
(n) valid
(o) invalid (usage of f is for real number)
(p) invalid (usage of e is for real number)

(3) (a) invalid (letter 'o' is not allowed for a radix representation)
(b) valid
(c) invalid (0X missing)
(d) valid
(e) invalid (not a numeral)
(f) valid
(g) valid
(h) invalid (88 is not an octal number)
(i) invalid (0X missing)
(j) invalid (LU is not a radix representation)
(k) valid
(l) valid
(m) valid
(n) valid
(o) invalid (not a numeral)
(p) invalid (not a numeral)

(4) (a) invalid (special characters Rs are not permitted)
(b) valid
(c) valid
(d) invalid (comma is not allowed with numerals)
(e) valid
(f) valid
(g) valid
(h) invalid (decimal point is not permitted after E)
(i) valid
(j) valid
(k) valid
(l) valid
(m) valid
(n) invalid (both f and l is not permitted for real numbers)
(o) valid
(p) valid

(5) (a) valid
(b) invalid (multicharacters)
(c) invalid (string)
(d) invalid (missing single quote)
(e) invalid (string)
(f) invalid (muticharacters)
(g) invalid (only a single character with ‘’)
(h) invalid (only a single character with ‘’)
(i) invalid (string)
(j) invalid (string)
(k) valid
(l) valid

(6) (a) a/b +10
(b) (5/9) * (f-32)
(c) (a/b)* (c/d)
(d) a * (a+v)/(b–1)

(7) (a) no space between pay bill (single identifier)
(b) left hand side cannot have an expression
(c) two operators (/-2.5) appearing together
(d) special symbols prefix 100
(e) operator is missing => s* (s–a) *(s–b)*(s–c)
(f) missing assignment operator and semicolon
(g) missing lvalue

(8) (a) $g = \dfrac{x(x^2 + y^2)}{(x^2 - y^2 + 2)}$
(b) sum = $\dfrac{x^2}{(a / b)}$
(c) $y = x + \dfrac{x}{y} + x - 2.0$
(d) root = $\dfrac{b^2 - 4ac}{2a}$
(e) interest = $\dfrac{pnr}{100}$

(9) (a) v1 = 1.5
(b) v2 = 60
(c) v3 = 13.5
(d) v4 = 53
(e) v5 = –10
(f) v6 = –55
(g) v7 = –2.875

(10) (a) sum1 = –14
(b) sum2 = 1
(c) sum3 = 10
(d) sum4 = –5
(e) sum5 = –2
(f) sum6 = 4
(g) sum7 = –7

(11) (a) exp1 = 0
(b) exp2 = 1
(c) exp3 = 0
(d) exp4 = 0
(e) exp5 = 1
(f) exp6 = 0
(g) exp7 = 1
(h) exp8 = 1
(i) exp9 = 0
(j) exp10 = 0
(k) exp11 = 1

(12) (a) f1 = 1
(b) f2 = 1
(c) f3 = 0
(d) f4 = 0
(e) f5 = 1
(f) f6 = 0
(g) f7 = 1
(h) f8 = 1

(13) (a) a = 3
(b) b = 2
(c) c = 28
(d) d = 0
(e) a = 4
(f) b = 12

Chapter 4

(1) (a) value of x = 10
(b) x = 10 y = 20 sum = 30
(c) x = 10 y = 20 sum = 30
(d) 0
(e) 1.23e+100
1.23E+100

a
A

(f) 100
0×64
0144
100
64
144

(g) 1
true
1
true

(h) 0

(i) 1

(2) (a) 0 (b) 2 (c) 0 (d) 2

(e) a = 0
b = 1
c = a + b = 1

(f) a = 1
b = 1
c = a + b = 2

(g) a = 1
b = 1
c = a + b = 1

(h) a = 0
b = 1
c = a + b = 1

(i) a = 1
b = 0
x = a–b = 1

(j) ABC

(k) A
B
C

(l) Hello
C++
program

Chapter 5

(1) (a) b = 20 c = 30
In a short circuit evaluation, the logical AND (&&) will not evaluate the second condition, if the first condition is false, due to the result will be false. Hence the above output.

(b) a = 10 b = 21
In a short circuit evaluation, the logical AND (&&) will evaluate the second condition only if the first condition is true. Hence the above output.

(c) b = 20 c = 30

In a short circuit evaluation, the logical OR (||) will not evaluate the second condition if the first condition is true, due to the result will be true. Hence the above output.

(d) b = 21 c = 31

In a short circuit evaluation, the logical OR (||) will evaluate the second condition only if the first condition is false. Hence the above output.

(e) a + b + c = 60

a = 10, b = 20, c = 30 and hence a + b + c = 60

(f) a + b + c = 62

a = 11, b = 20, c = 31 and hence a + b + c = 62

(g) value of a*b = 210

a = 10, b = 21 and hence a*b = 210

(h) value of a+b = 34

a = 13, b = 21 and hence a+b = 34

(2) (a) default case (b) inner case 2

(c) inner case 3 (d) x = 4

x = 9

No break statement is used in the case '4' and hence the program is executed even after the case '4'.

(e) x = 4

x = 10

No break statement is used within the case labels and hence the program is executed from case '4' to case '3'.

(f) y = 1

x = 32

x = 2 i = 1 (x += x that is x = x + x, 1+1 = 2)

x = 4 i = 2 (x += x , 2 + 2 = 4), and so on

x = 8 i = 3

x = 16 i = 4

x = 32 i = 5

(g) a = 7

grade - C

For an integer division, the digits will be dropped after a decimal point. (Here, only 7 is assigned and 0.5 is dropped).

(3)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 75;
    a = a % 10;
    cout <<"a = " << a;
    if (a == 9)
        cout <<"grade - A \n";
    else if (a ==  8)
        cout <<"grade - B \n";
    else if (a == 7)
        cout <<"grade - C \n";
```

```
        else if (a == 6)
            cout <<"grade - D \n";
        else
            cout <<" grade - fail \n";
        return 0;
    }
```

(4)

```
    #include <iostream>
    using namespace std;
    int main()
    {
        char grade;
        grade = 'E';
        switch (grade) {
            case 'A':
                cout <<" Excellent \n";
                break;
            case 'B':
            case 'C':
                cout <<" Good \n";
                break;
            case 'D':
            case 'E':
                cout <<" Poor \n";
                break;
            default:
                cout <<" Grade - Fail \n";
                break;
        }
    }
```

(5) (a) r = 2
p = (10 % 2) /3, that is 10 % 2 = 0 and 0/3 gives 0.

(b) r = 2
p = (10 % 2) /3, that is 10 % 2 = 0 and 0/3 gives 0.
In the post incrementation, the value of `p` is assigned for comparison before increment. Hence the above output.

(c) s = 3
if (++`p` == 0), here `p` is incremented and compared with 0. Hence the value of `p` = 1, the if statement fails to execute the `if-block`. Then it checks for another `else-if` statement.
`else if` (p++ == 1), the value of `p` is compared with 1 before incrementation due to the post increment. Hence the above output.

(d) p = 6
if `( (++p)++ == 0)`, here `p` is incremented and compared with 0 due to parenthese which is higher precedence. Hence the value of `p` = 1, the `if` statement fails to execute the `if-block`. However, the value of `p` is incremented again that is `p` = 2, due to post increment operator.
It checks for another `else-if` statement.
`else if` (`p++ == 1`), the value of `p = 2`, is compared with 1 before incrementation due to the post increment. `else if` satement blocks also fail to execute but the value of `p` is incremented by 1.

Now p = 3
It checks for another else-if statement.
```
else if ( ( p++ == 2) || ( p++ == 3) || ( p++ == 4))
```
The value of p is not matched with any of the conditions above.
However, each ++ operation, the value is incremented by one.
Hence the above output.

(6) (a) No Output
As the test condition of the for statement checks whether the given condition is 0 or 1. In this case, the given test condition is 0 and hence the for statement will not be repeated at all.

(b) The test condition of the for statement checks whether the given condition is 0 or 1. In this case, the given test condition is 1 and the for statement will be repeated. There is no provision to change of the test condition from 1 to 0 and the for statement will be repeated for ever.

(c) –6 –5 –4 –3 –2 –1
In C++, the for statement checks whether the given test condition is 0 or 1. The value of j increments from –6 by 1. When it reaches the value to 0, the for statement fails to repeat.

(d) No output as the for loop terminates with semicolon.

(e) The for loop repeats indefinitely as there is no provision to check for the test condition within the for statement. Hence the value of j will be incremented from –6 by one, to indefinitely.

(f) The for loop repeats indefinitely as there is no provision to check for the test condition within the for statement. Hence the value of j will be incremented from –6 by one, to indefinitely.

(g) ABCDEFGHIJKLMNOPQRSTUVWXYZ

(h) j = 1 j = 3 j = 5 j = 7 j = 9

(i) j = 0 j = 3 j = 6 j = 9

(j) j = 0 j = 0.2 j = 0.4 j = 0.6 j = 0.8 j = 1

(7) (a) i = 1
i = 2
i = 3
and so on
The given test condition is true and there is no provision to change the given test condition from true to false within the for statement. Hence, the for loop iterates for ever until the user terminates from external command like prss Cntrl + C keys together.

(b) No output will be displayed.
The given test condition is false. Therefore the for statement will not be repeated at all.

(c) No output.
The for statement is repeated for ever due to the given condition for (;;). The for (); is known for the null statement. Hence, cout << " i = " << i << "\n" will not be executed.

(d) The for statement will be repeated for ever.
```
i = 1
i = 2
```
and so on until the user terminates through external command like pressing Cntrl + C keys.

(e) The given test condition is true and hence the for statement will be repeated for ever until the user terminates through external command like pressing Cntrl + keys.
```
a = -1.1
a = -0.6
a = -0.1
a = 0.4
```

```
a = 0.9
```
and so on.

(f) The `for` statement will be repeated for ever and displaying all the ASCII characters from 0 to 255.
```
ch = A
ch = B
```
and so on.

(g) `i` = 1
`i` = 2
`i` = 3 and so on.
There is no test condition to terminate the for statement and therefore the `for` statement will be executed for ever displaying the value of `i` incrementing by one.

(h) There is no test condition to terminate the `for` statement and therefore the `for` statement will be executed for ever displaying the value of `i` = 0.

(8) (a) `i` = 0
`i` = 1
`i` = 2 and so on.
The given test condition for the `while` tatement is true and hence the `while` statement will be repeated for ever displaying the following contents. Due to the post increment operator, the value of `i` starts from 0.

(b) i = 1
i = 2
i = 3 and so on.
The given test condition for the `while` statement is true and hence the `while` statement will be repeated for ever displaying the following contents. Due to the preincrement operator, the value of i starts from 1.

(c) No output due to the given test condition value is false. Hence the while statement will not be repeated.

(d) No output due to the test condition value is zero. As long as the test condition value is true (non-zero), the while loop will repeat. The given test condition value is zero (false) and hence the while loop is not entered.

(e) Compile time error.
ISO C++ forbids an empty while statement.

(f) The while loop repeats for ever displaying the values from 1 2 3 and so on due to the test condition value is one (nonzero). There is no provision to change the given test condition in this program. Hence it repeats for ever until the user terminates through external commands like pressing Cntrl +C.

(g) –6 –5 –4 –3 –2 –1

(h)
```
Komputer
Komputer
 i = 2
Komputer
Komputer
 i = 4
Komputer
```

Komputer
i = 6
Komputer
Komputer
i = 8
Komputer
Komputer
i = 10
No of iterations = 5

(i)

Komputer
Hello world
Komputer
Hello world
Komputer
Hello world
No of iterations of the inner loop = 3
No of iterations of the outer loop = 3

(9) (a) The `do-while` statement repeats for ever displaying the `flag = 1` until the user terminates through external command like pressing Cntrl + C. As long as the test condition value is true (one), the `do-while` statement repeats for ever.

(b) `flag = 1`

(c) The `do-while` statement repeats for ever displaying the flag = 1. The new value of the flag that is !(flag), is not stored into the variable flag.

(d) "Hello C++ world" will be displayed 5 times

(e) "Hello C++ world" will be displayed 3 times

(f) "Hello C++ world" will be displayed 6 times

(g) "Hello C++ world" will be displayed 2 times and i = 2

Chapter 6

(1) (a) 1 5 1 (b) 1 5 1 10 1 (c) 1 5 10 5 1
(d) 1 6 6 (e) 1 6 16 6 (f) 10 11 11
(g) 10 11 11 (h) 10 2 2 (i) 10 2 10

(2) (a) i = 10 j = 20 total = 200

(b) Compile time error. The return statement with no value is defined but in function declaration, a non-void return type is defined. Hence, it gives error.

(c) total = 40 (d) sum = 10 (e) sum = 10

(f) sum = 13 (g) sum = 0 (h) sum = 0

(3) (a) Innermost ...
now inside the function 2
Within a function 1
In main ...

(b) j = 11 b = 12

(c) i = 10 a = 10 (returned value is not assigned (in the function call)

(d) i = 11 a = 12

(e) i = 10 a = 10 (due to post-increment operation)

(f) sum = 1108544021. An automatic variable will be initialised with a garbage value by default, if it is uninitialised by the user.
(g) sum = 1. A static variable is initialised to zero by default, if it is uninitialised. Hence above the output.
(h) sum = 0
(i) sum = 3
(j) value of `a` (in main) = 10
value of `a` (inside function) = 10
value of `a` (after function call) = 10
(k) `a` = 10

Chapter 7

(1) (a) 1 2 3 4 0 0 (b) 0 1 4 9 16
(c) Compilation time error. Array is declared as a const data type. Whenever any items are declared with const modifier, it is meant for read only purpose. Altering the const data items is illegal. Hence it gives error.
(d) contents of the array in main
0 1 4 9 16
contents of the array in function
0 1 4 9 16
(e) contents of the array in main
1 0 1 0 1
contents of the array in function
0 1 0 1 0
(f) Contents of the array
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5

(2) (a) Contents of the array
1 0 0
0 1 0
0 0 1
(b) Contents of the array
1 0 0
0 1 0
0 0 1
Sum of all elements in the array = 3
(c) Contents of the array
1 2 13
4 15 6
17 8 9
Sum of all diagonal elements in the array = 45
(d) Contents of the array
1 2 13

4 15 6
17 8 9
Sum of all diagonal elements in the array = 25

(e) contents of the array in main
1 2 3 4 0
sum of all elements = 10

Chapter 8

(1) (a) ++(*ptr1) = 11 (b) (*ptr1)++ = 10
(c) *ptr1++ = 10 (d) ++*ptr1 = 11
(e) *ptr1 = 11
(f) &(a++)—it is an illegal form of declaring unary (&) operator. The compiler displays the error message as "non-lvalue in unary (&)"
(g) *ptr1 = 0 (h) *ptr1 = 1
(i) *ptr = 10

(2) (a) 10 10 11
(b) 10 11 11
(c) 10 11 12 12
(d) 10 11 12 12 12
(e) 10 11 11 12 12
(f) 10 11 10 10 10
(g) Compilation error, In the case of call by reference, the actual parameter list must have only the variables. It cannot have constants.

(3) (a) New Delhi
Chennai
Mumbai
Hyderabad
(b) Hyderabad
Mumbai
Chennai
New Delhi
(c) contents of ptr1 = 101
contents of ptr2 = 101
(d) contents of ptr1 = 10
contents of ptr2 = 10
contents of ptr3 = 10
(e) 2 3 5
10 20 30
(f) *ptr = 10
*ptr = 11
(g) *ptr1 = 10
*ptr1 = 11
*ptr2 = 11

Chapter 9

(1) (a) Contents of x = 101

Contents of y = -199

(b) Contents of x = 10
Contents of y = -20

(c) Contents of x = 11
Contents of y = -19

(d) Contents of x = 11
Contents of y = -19

(e) Contents of x = 11
Contents of y = -19

(2) (a) Contents of x = 11
Contents of y = -19

(b) Contents of x = 11
Contents of y = -19

(c) Contents of x = -1073748600
Contents of y = 1107383668
(ptr++)->x; /* pointer is incremented,not its contents */
(ptr++)->y; /* pointer is incremented,not its contents */
Hence, garbage values are displayed due to the random address selection

(d) Contents of x = 10
Contents of y = -20

(e) Contents of x = -1073747112
Contents of y = 10
abc->ptr1++; /* address is incremented but not its contents */
abc->ptr2++; /* address is incremented but not its contents */
Hence, garbage values are displayed due to the random address selection

(f) Contents of x = -1073745960
Contents of y = 10
++(obj->ptr1); /* address is incremented but not its contents */
++(obj->ptr2); /* address is incremented but not its contents */
Hence, garbage values are displayed due to the random address selection

Chapter 10

(1) (a)
```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

(b)
```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
```

(c)
```
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

(d)
```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

(e) total = 30 (f) total = 25

(g) Compile time error due to conflicts with declaration of variables. The `'int a'` has been declared in both places as a private and a public category. Hence, it gives error.

(2) (a) Hello C++ world

(b) Hello
C++
world

(c) Compile time error due to member functions of `struct::abc` is private.

Nested class members should be defined in a public category.

(d) Compile time error due to member functions of `struct abc :: xyz` is private.

(e) `Hello`

(f) Compile time error due to `void abc:: dispabc()` is private category. Member functions of a class object should be declared as public category.

(g) value of a = 11

(h) Compile time error due to the storage class modifier 'static'. The keyword 'static' can only be applied for objects and functions. The storage class modifier 'static' cannot be declared for class declaration. Hence it gives error.

(3) (a) `i` = 0. The static variable will be initialised to zero automatically by the compiler, if it is uninitialised.

(b) `i` = 0. The volatile variable will be initialised to zero automatically by the compiler, if it is uninitialised.

(c) Compile time error due to the variable `'i'` is declared as a const data type that is read only purpose. The const value cannot be changed in a program. Hence it gives compilation error stating that the variable `i` is read only.

(d) Compile time error due to uninitialisation of the `'const int i'`. Whenever a `const` data type is declared, it should be initialised with data. Therefore, it gives error.

(e) `i` = 1073828704 (garbage). An auto variable will be initialised automatically by the C++ compiler with some garbage if it is uninitialised explicitly.

(f) `class tag`

Chapter 11

(1) (a) Compile time error due to all of its members in a class are private.
When a constructor is defined in a class, it should be a public type.
Constructor cannot be defined as a private or protected type.

(b) Compile time error due to a constructor is defined as a protected type.
Constructor cannot be defined as a private or protected type.

(c) Compile time error due return type specification for a constructor is defined.
Constructor cannot be defined with return data type and not even as a void category.

(d) Compile time error due to return 0 from a constructor.

(e) Calling constructor
The return statement does not return any data type and not even void type.
The C++ compiler just skips this statement and does not display any error message or warning.
Hence the above output will be displayed.

(f) Compile time error due to a constructor is defined as a static modifier.

(g) Compile time error. The storage type, namely, static, register, extern, const and voltaile are not used to define a constructor. Hence it gives error.

(2) (a) `Calling constructor`
By default, all of its members in a union or struct are public. Hence the above output.

(b) `Calling constructor`
By default, all of its members in a `union` or `struct` are public.

(c) `Calling destructor`

(d) `Calling destructor`
The `return` statement does not return any data type and not even void type.
The C++ compiler just skips this statement and does not display any error message or warning.

Hence the above output will be displayed.

(e) `Calling destructor`
Destructor can be declared as virtual whereas the contructor cannot be declared.

(f) Compile time error due to a constructor is declared as virtual.

(g) It will not display any specific error during compile time or run-time error. But no output will be displayed due to the struct sample object is not instantiated.

(h) Compile time error due to `abc::sample:: sample()` is not permitted in ISO standard de laration.

(3) (a) abc - class constructor
(b) x - class constructor
(c) z - class destructor
y - class destructor
x - class destructor
abc - class destructor
(d) z - class destructor
y - class destructor
x - class destructor
abc - class destructor

Chapter 12

(1) (a) members of abc
(b) members of abc
(c) members of abc
(d) members of abc
members of derivedB
(e) members of abc
members of derivedB
members of derivedC
members of derivedD
(f) members of abc
members of xyz
members of derivedC
members of derivedD

(2) (a) a = 10
(b) Compile time error. A class has been named but not declared. This type of class is not permitted to use as a base class in a derived class. The C++ compiler gives error message stating that a base class 'abc' has incomplete type.
(c) a = 10
(d) Compile time error. A class 'xyz' fails to be a struct or class type.
(e) a = 10 (f) a = 10
(g) Compile time error. A base class cannot be declared with union type. Base class is supposed to be a struct or class type. Hence it gives error.
(h) Compile time error. A union type cannot be specified as a derived class. Hence it gives error.
(i) a = 10
(j) Compile time error. int abc::a is a private data type. The members of a derived class cannot access the private data member of a base class. Hence it gives error.

(3) (a) value of a = 11
value of b = 21
value of c = 31
(b) value of a = 11
value of b = 21
value of c = 31
(c) value of a = 11
value of b = 21
value of c = 31
(d) value of a = 11
value of b = 21
value of c = 31
(e) value of a = 101
value of b = 301

Chapter 13

(1) (a) Minimum = 10
Minimum = -2.2
Minimum = 100
(b) Hello, C++ world
(c) a = 12
b = 22
c = 33
(d) str1 = Hello
str2 = C++
str3 = world
(e) str1 = Hello
str2 = C++
(f) sum a[] = 45
sum fa[] = 46

(2) (a) contents of the first object
x = 10
contents of the second object
x = 10
(b) contents of the first object
x = 21
contents of the second object
x = 20
(c) contents of the first object
x = 21
contents of the second object
x = 21
(d) x = 0
x = 0
(e) x = 0
x = 1

(f) x = 0
x = 1
(g) 0
1

Chapter 14

(1) (a) ++b = 21 (b) ++b = 21 (c) ++b = 21 (d) ++a = 11
(e) A
A
A
A
(f) A
B
C
D
(2) (a) A
B
C
D
(b) ~D
~C
~B
~A
(c) A
B
C
D
~D
~C
~B
~A
(d) A
B
A
C
D
(e) A
B
A
C
D
(f) Compile time error. Constructors cannot be declared as virtual whereas destructors can be declared as virtual. Hence it gives error.
(g) A
B
~B
~A
(h) A
B
~B
~A

Chapter 15

(1) (a) sum of the integers = 45
sum of the floating point numbers = 49.5
(b) a = 10 b = 20 minimum = 10
x = 1.1 y = 2.2 minimum = 1.1
(c) constructor with int argument
constructor with floating point
(d) class template - constructor
class template - constructor
class template - constructor
class template - constructor

(e) constructor
integer:
content of the value = 10
destructor

(2) (a) a = 1 b = 1e-04 Quotient = 10000
(b) element[0] = 1
element[1] = 2
element[2] = 3
element[3] = 4
element[4] = 5
Memory out of range
sum = 15
(c) x/y = 0.5

(3) (a) ::a = 11
X::a = 21
Y::a = 31
(b) ::a = 10
X::a = 20
Y::a = 30
(c) ::a = 9
X::a = 21
Y::a = 30
(d) *ptr = 20 (e) ++*ptr = 21 (f)++*ptr = 21
(g) Compile time error. The scope of `Y::a` is undefined.
Hence it gives error.
(h)
```
a = 10
b = 20
```

(4) (a) Compile time error. In a unanimous namespace, each variable should be distinct. The `int` a is ambiguous and hence, it gives error.
(b) A :: display
11
B :: display
100
(c) X::a = 11
X::Y::a = 21
(d) display1 = 11
display2 = 101
(e) Compile time error. The identifier `'a'` is an ambiguous in the unnamed namespace declaration.
Hence, compile time error is displayed.
(f) 5
2.7183
10
3.14 1 16
(g) 5
3.1416

Bibliography

1. *Borland, C++ User's manual*, ver. 3.0, Scotts Valley, California, 1992.
2. *Borland Inc, Turbo C++ ver. 3.1, User's Guide,* Scotts Valley, California, 1992.
3. Andrews, M., *Visual C++ Object Oriented Programming*, SAMS, A Division of Prentice Hall Publ Inc., 1993.
4. Barton, J.J. and Nackman, L.R., *Scientific and Engineering C++, An Introduction with Advanced Techniques and examples,* Addison Wesley Publishing Co., Reading, Mass, 1994.
5. Berry, J.T., *C++ Programming*, *The Waite group's*, Prentice Hall of India, New Delhi, 1994.
6. Budd, T.A., *Classic Data Structures in C++*, Addison Wesley Publishing Co., Reading, Mass, 1994.
7. Cargill, T., *C++ Programming Style*, Addison Wesley Publishing Co., Reading, Mass, 1992.
8. Chriatian, K., *Microsoft Guide to C++ Programming*, Microsoft, 1992.
9. Chirlian, P.M., *Programming in C++*, CBS Publishers and Distributors, New Delhi, 1992.
10. Cline, M.P., and Lomow, G.A., *C++ FAGS - Frequently Asked Questions*, Addison Wesley Publishing Co., Reading, Mass, 1995.
11. Coplien, J.O., *Advanced C++*, *Programming Styles and Idioms*, Addison Wesley Publishing Co., Reading, Mass, 1992.
12. Ellis, M.A., and Stroustrup, B., *The Annotated C++ Reference Manual*, Addison Wesley Publishing Co., Reading, Mass, 1990.
13. Faison, T., *Borland C++ version 3, Object Oriented Programming,* SAMS, A Division of Prentice Hall Publishing Inc., 1992.
14. Graham, N., *Learning C++*, McGraw Hill Inc., New York, 1991.
15. Gray, N.A.B., Programming with Class, John Wiley and Sons. Inc., Chichester, 1994.
16. Gurewich, N. and Gurewich,O., *Master C++ from C to C++ in 2 weeks*, BPB Publications, New Delhi, 1994.
17. Hansen, T.L., *The C++ Answer Book*, Addison Wesley Publishing Co., Reading, Mass, 1990.
18. Holmes, M and Flanders, B., *PC Magazines*, *C++ Communications Utilities*, BPB Publications, New Delhi, 1994.
19. Huckert, E., *Programming in C++*, Galgotia Publications, New Delhi, 1992.
20. Hughes, J.M., *Programming* in *Zortech C++ with ver. 2.0*, Galgotia Publications, New Delhi, 1990.
21. Jamsa, K., *C/C++ Tips*, Galgotia Publications, New Delhi, 1994.
22. ----------, *Rescued by C++*, *The easiest way to learn C++*, Galgotia Publications, New Delhi, 1994.
23. ----------, *Success with C++,* Galgotia Publications, New Delhi, 1994.
24. Johnsonbaugh, R. and Kalin, M., *Object Oriented Programming in C++,* Prentice Hall Inc., Englewood Cliffs, NJ, 1995.

25. Ladd, S.R., *C++ Techniques and Applications*, M&T Books, A Division of M&T Publishing Inc, Redwood city, CA, 1990.
26. ----------, *Turbo C++, Techniques and Applications*, BPB Publications, New Delhi, 1993.
27. ----------, *Applying C++*, BPB Publications, New Delhi, 1993.
28. ----------, *C++ Components and Algorithms*, BPB Publications, New Delhi, 1993.
29. Lafore, R., *Object Oriented Programming in Microsoft C++*, Galgotia Publications, New Delhi, 1993.
30. ----------, *Object Oriented Programming in Turbo C++*, Galgotia Publications, New Delhi, 1992.
31. Lippman, S.B., *C++ Primer*, Addison Wesley Publishing Co., Reading, Mass, 1995, 2 edn.
32. Masters, T., *Advanced Algorithms for Neural Networks – A C++ source book*, John Wiley & Sons, Inc., New York, 1995.
33. Mayers, S., *Effective C++: 50 specific ways to improve your programs and designs*, Addison Wesley Publishing Co., Reading, Mass, 1993.
34. ----------, *Effective C++,* Addison Wesley Publishing Co., Reading, Mass, 1992.
35. McCord, J.W., *Developing Window Applications with Borland C++ 3.1*, SAMS, A Division of Prentice Hall Computer Publishing, Indiana, USA, 1994.
36. Murray, R.B., *C++ Strategies and Tactics*, Addison Wesley Publishing Co., Reading, Mass,1993.
37. Nagler, E., *Learning C++ A Hands on Approach*, Jaico Publishing House, Bombay, 1994.
38. Ouallive, S., Windows Programming with Borland C++ Covers Through Version 4.0, BPB Publications, New Delhi, 1994.
39. Prata, S., *The Waite Group's*, *C++ Primer Plus*, Galgotia Publications, New Delhi, 1992.
40. Peterson, M., *Borland C++, Developer's Bible,* Galgotia Publications, New Delhi, 1993.
41. Riley, C., *Programming online Help with C++*, BPB Publications, New Delhi, 1994.
42. Schildt, H., Using Turbo C++, Osborne McGraw Hill Publishing Co., New York, 1992.
43. Sengupta, S., and Korobkin, C.P., *C++ Object Oriented Data structures*, Springer-Verlag, New York, 1994.
44. Shammas, N.O., *What Every Borland C++ version 4.0 Programmers Should Know*, SAMS, A Divison of Prentice Hall publishing Inc., 1994.
45. Smith, N.E., *Illustrated Borland C++* BPB Publications, New Delhi, 1993.
46. ----------, *Object Oriented Programming Using Turbo C++*, BPB Publications, New Delhi, 1992.
47. Stroustrup, B., *The C++ Programming Language,* Addison Wesley Publishing Co., Reading, Mass, 1986.
48. Traister, R.J., *Clean Coding in Borland C++*, BPB Publications, New Delhi, 1993.
49. Vijay Mukhi's, *Dyanmic Data Exchange Under Windows 3.1 Using Borland C++ 3.1*, Vision Books, New Delhi, 1994.
50. ----------, The C Odyssey C++ Graphics, BPB Publications, New Delhi, 1992.
51. ----------, *Last Word on C++*, Vision Books, New Delhi, 1994.
52. Weiner, R.S., and Pinson. L.J., *An Introduction to Object Oriented Programming and C++*, Addison Wesley, Reading, Mass, 1988.
53. Weiss, M.A., *Data structures and Algorithms and Analysis in C++*, The Benjamin/Cummings Publishing Co., Inc., Redwood city,1994.
54. Wilson, D.A., Rossenstein, L.S. and Shafer, D., *C++ Programming with MacAPP,* Addison Wesley Publishing Co., Reading, Mass, 1991.
55. Young, M.J., *Mastering Microsoft Visual C++ Programming,* Sybix, USA, 1993.

Index

M

N

O

P