

OBJECT ORIENTED PROGRAMMING AND DATA STRUCTURES

MEET THE AUTHOR



E Balagurusamy, former Vice Chancellor, Anna University, Chennai, and Member, Union Public Service Commission, New Delhi, is currently the Chairman of EBG Foundation, Coimbatore. He is a teacher, trainer, and consultant in the fields of Information Technology and Management. He holds an ME (Hons) in Electrical Engineering and PhD in Systems Engineering from the Indian Institute of Technology, Roorkee. His areas of interest include *Object-Oriented Software Engineering*, *E-Governance*, *Technology Management*, *Business Process Re-engineering*, and *Total Quality Management*.

A prolific writer, Dr Balagurusamy has authored a large number of research papers and several books. His best-selling books, among others include

- *Fundamentals of Computers*
- *Computing Fundamentals and C Programming*
- *Programming in ANSI C*, 6e
- *Programming in Java*, 4e
- *Programming in BASIC*, 3e
- *Programming in C#*, 3e
- *Numerical Methods*
- *Reliability Engineering*

A recipient of numerous honors and awards, Dr Balagurusamy has been listed in the *Directory of Who's Who of Intellectuals* and in the *Directory of Distinguished Leaders in Education*.

OBJECT ORIENTED PROGRAMMING AND DATA STRUCTURES

E Balagurusamy

*Chairman
EBG Foundation
Coimbatore*



McGraw Hill Education (India) Private Limited

NEW DELHI

McGraw Hill Education 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



McGraw Hill Education (India) Private Limited

Published by McGraw Hill Education (India) Private Limited,
P-24, Green Park Extension, New Delhi 110 016

Object Oriented Programming and Data Structures

Copyright © 2014, by McGraw Hill Education (India) Private Limited.

No part of this publication may be reproduced or distributed in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise or stored in a database or retrieval system without the prior written permission of the publishers. The program listings (if any) may be entered, stored and executed in a computer system, but they may not be reproduced for publication.

This edition can be exported from India only by the publishers,
McGraw Hill Education (India) Private Limited.

ISBN (13 digits): 978-9-33-920551-5

ISBN (10 digits): 9-33-920551-0

Managing Director: *Kaushik Bellani*

Head—Higher Education Publishing and Marketing: *Vibha Mahajan*

Senior Publishing Manager—SEM & Tech Ed.: *Shalini Jha*

Editorial Executive—Acquisitions: *S.Vamsi Deepak*

Manager—Production Systems: *Satinder S Baveja*

Asst. Manager Editorial Services: *Sohini*

Production Executive: *Anuj Kr Shrivastava*

Assistant General Manager—Higher: *Ed.: Vijay Sarathi*

Asst. Product Manager—SEM & Tech Ed.: *Tina Jajoriya*

Senior Graphic Designer: *Meenu Raghav*

General Manager—Production: *Rajender P Ghansela*

Production Manager: *Reji Kumar*

Information contained in this work has been obtained by McGraw Hill Education (India), from sources believed to be reliable. However, neither McGraw Hill Education (India) nor its authors guarantee the accuracy or completeness of any information published herein, and neither McGraw Hill Education (India) nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that McGraw Hill Education (India) and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

Typeset at Print-O-World, 2579, Mandir Lane, Shadipur, New Delhi 110 008 and printed at Gopsons, A - 2 & 3, Sector - 64, Noida, U.P. - 201 301

Cover: Gopsons

CONTENTS

Preface

xiii

Unit-I DATA ASTRACTING & OVERLOADING 1–187

1. Principles of Object-Oriented Programming 3

- 1.1 Software Crisis 4
- 1.2 Software Evolution 5
- 1.3 A Look at Procedure-Oriented Programming 6
- 1.4 Object-Oriented Programming Paradigm 8
- 1.5 Basic Concepts of Object-Oriented Programming 9
- 1.6 Benefits of OOP 13
- 1.7 Object-Oriented Languages 14
- 1.8 Applications of OOP 15
- Summary 16
- Key Terms 17
- Review Questions 17

2. Beginning with C++ 19

- 2.1 What is C++? 20
- 2.2 Applications of C++ 20
- 2.3 A Simple C++ Program 20
- 2.4 More C++ Statements 25
- 2.5 An Example with Class 27
- 2.6 Structure of a C++ Program 28
- 2.7 Creating the Source File 29
- 2.8 Compiling and Linking 29
- Summary 30
- Key Terms 30
- Review Questions 30
- Debugging Exercises 31
- Programming Exercises 32

3. Tokens, Expressions and Control Structures 33

- 3.1 Introduction 34
- 3.2 Tokens 34
- 3.3 Keywords 34
- 3.4 Identifiers and Constants 35

3.5	Basic Data Types	36
3.6	User-Defined Data Types	38
3.7	Storage Classes	41
3.8	Derived Data Types	42
3.9	Symbolic Constants	43
3.10	Type Compatibility	44
3.11	Declaration of Variables	45
3.12	Dynamic Initialization of Variables	45
3.13	Reference Variables	46
3.14	Operators in C++	48
3.15	Scope Resolution Operator	48
3.16	Member Dereferencing Operators	50
3.17	Memory Management Operators	51
3.18	Manipulators	55
3.19	Type Cast Operator	57
3.20	Expressions and Their Types	58
3.21	Special Assignment Expressions	60
3.22	Implicit Conversions	61
3.23	Operator Overloading	62
3.24	Operator Precedence	63
3.25	Control Structures	63
	<i>Summary</i>	68
	<i>Key Terms</i>	69
	<i>Review Questions</i>	69
	<i>Debugging Exercises</i>	70
	<i>Programming Exercises</i>	72

4. Classes and Objects

74

4.1	Introduction	75
4.2	C Structures Revisited	75
4.3	Specifying a Class	76
4.4	Defining Member Functions	80
4.5	A C++ Program with Class	82
4.6	Making an Outside Function Inline	83
4.7	Nesting of Member Functions	84
4.8	Private Member Functions	85
4.9	Arrays within a Class	86
4.10	Memory Allocation for Objects	90
4.11	Static Data Members	90
4.12	Static Member Functions	93
4.13	Arrays of Objects	95
4.14	Objects as Function Arguments	97
4.15	Friendly Functions	99
4.16	Returning Objects	104
4.17	const Member Functions	106

- 4.18 Pointers to Members 107
- 4.19 Local Classes 109
 - Summary* 109
 - Key Terms* 110
 - Review Questions* 110
 - Debugging Exercises* 111
 - Programming Exercises* 114

5. Functions in C++

116

- 5.1 Introduction 117
- 5.2 The Main Function 117
- 5.3 Function Prototyping 118
- 5.4 Call by Reference 119
- 5.5 Return by Reference 120
- 5.6 Inline Functions 121
- 5.7 Default Arguments 123
- 5.8 const Arguments 125
- 5.9 Recursion 125
- 5.10 Function Overloading 127
- 5.11 Friend and Virtual Functions 129
- 5.12 Math Library Functions 129
 - Summary* 131
 - Key Terms* 132
 - Review Questions* 132
 - Debugging Exercises* 133
 - Programming Exercises* 134

6. Constructors and Destructors

136

- 6.1 Introduction 137
- 6.2 Constructors 137
- 6.3 Parameterized Constructors 138
- 6.4 Multiple Constructors in a Class 141
- 6.5 Constructors with Default Arguments 143
- 6.6 Dynamic Initialization of Objects 144
- 6.7 Copy Constructor 146
- 6.8 Dynamic Constructors 148
- 6.9 Constructing Two-Dimensional Arrays 149
- 6.10 const Objects 151
- 6.11 Destructors 151
 - Summary* 155
 - Key Terms* 155
 - Review Questions* 155
 - Debugging Exercises* 156
 - Programming Exercises* 158

7. Operator Overloading and Type Conversions	160
7.1 Introduction	161
7.2 Defining Operator Overloading	161
7.3 Overloading Unary Operators	162
7.4 Overloading Binary Operators	164
7.5 Overloading Binary Operators Using Friends	167
7.6 Manipulation of Strings Using Operators	170
7.7 Some Other Operator Overloading Examples	173
7.8 Rules for Overloading Operators	175
7.9 Type Conversions	176
Summary	183
Key Terms	183
Review Questions	184
Debugging Exercises	184
Programming Exercises	186
Unit-II INHERITANCE AND POLYMORPHISM	189–269
8. Inheritance: Extending Classes	179
8.1 Introduction	192
8.2 Defining Derived Classes	193
8.3 Single Inheritance	194
8.4 Making a Private Member Inheritable	199
8.5 Multilevel Inheritance	202
8.6 Multiple Inheritance	206
8.7 Hierarchical Inheritance	211
8.8 Hybrid Inheritance	212
8.9 Virtual Base Classes	215
8.10 Abstract Classes	218
8.11 Constructors in Derived Classes	220
8.12 Member Classes: Nesting of Classes	225
Summary	227
Key Terms	227
Review Questions	228
Debugging Exercises	228
Programming Exercises	232
9. Pointers, Virtual Functions and Polymorphism	235
9.1 Introduction	236
9.2 Pointers	237
9.3 Pointers to Objects	247
9.4 this Pointer	251
9.5 Pointers to Derived Classes	253
9.6 Virtual Functions	255
9.7 Pure Virtual Functions	260

9.8 Virtual Constructors and Destructors 262

Summary 263

Key Terms 264

Review Questions 264

Debugging Exercises 264

Programming Exercises 269

Unit-III LINEAR DATA STRUCTURES

271–394

10. Introduction to Algorithm and Data Structures

273

10.1 Introduction 274

10.2 Algorithms 274

10.3 Asymptotic Notation 277

10.4 Introduction To Data Structures 282

10.5 Types Of Data Structures 282

10.6 Data Structure Operations 286

Summary 287

Key Terms 287

Review Questions 288

11. Arrays

289

11.1 Introduction 290

11.2 Types of Arrays 290

11.3 Representation of one-Dimensional Array in Memory 291

11.4 Array Traversal 291

11.5 Insertion and Deletion 293

11.6 Sorting and Searching 298

11.7 Representation of Multi-Dimensional Array in Memory 301

11.8 Realizing Matrices Using Two-Dimensional Arrays 303

11.9 Matrix Operations 305

Summary 314

Key Terms 314

Review Questions 314

Programming Exercises 314

12. Linked Lists

315

12.1 Introduction 316

12.2 Linked Lists—Basic Concept 316

12.3 Linked List Implementation 317

12.4 Types of Linked Lists 328

12.5 Circular Linked List 328

12.6 Doubly Linked List 337

12.7 Some More Programs 344

Summary 345

Key Terms 345

Review Questions 345

Programming Exercises 346

13. Stacks	347
13.1 Introduction	348
13.2 Stacks	348
13.3 Stack Operations	349
13.4 Stack Implementation	351
<i>Summary</i>	364
<i>Key Terms</i>	365
<i>Review Questions</i>	365
<i>Programming Exercises</i>	366
14. Queues	367
14.1 Introduction	368
14.2 Queues—Basic Concept	368
14.3 Queue Operations	370
14.4 Queue Implementation	372
14.5 Circular Queues	383
14.6 Priority Queues	384
14.7 Double-ended Queues	384
14.8 Polynomial Manipulation Representing Polynomials using Linked Lists	386
14.9 Arithmetic Expression Infix to Postfix Conversion	390
<i>Summary</i>	393
<i>Key Terms</i>	393
<i>Review Questions</i>	393
<i>Programming Exercises</i>	394
UNIT-IV NON-LINEAR DATA STRUCTURE	395–447
15. Trees	386
15.1 Introduction	398
15.2 Basic Concept	398
15.3 Binary Tree	399
15.4 Binary Tree Representation	401
15.5 Binary Tree Traversal	406
15.6 Binary Search Tree	412
15.7 Tree Variants	420
<i>Summary</i>	428
<i>Key Terms</i>	429
<i>Review Questions</i>	429
<i>Programming Exercises</i>	430
16. Graphs	431
16.1 Introduction	432
16.2 Basic Concept	432
16.3 Graph Terminology	432
16.4 Graph Implementation	434
16.5 Shortest Path Algorithm	441

16.6	Graph Traversal	445
	<i>Summary</i>	446
	<i>Key Terms</i>	447
	<i>Review Questions</i>	447
	<i>Programming Exercises</i>	447

UNIT-V SORTING AND SEARCHING **447–487**

17. Sorting and Searching	447
17.1 Introduction	452
17.2 Sorting Techniques	452
17.3 Searching Techniques	477
	<i>Summary</i> 486
	<i>Key Terms</i> 487
	<i>Review Questions</i> 487
	<i>Programming Exercises</i> 487

Model Test Paper	489

PREFACE

Object-Oriented Programming (OOP) and data structures has become the preferred programming approach by the software industries, as it offers a powerful way to cope with the complexity of real-world problems. Among the OOP languages today, C++ is by far the most widely used language.

Since its creation by Bjarne Stroustrup in the early 1980s, C++ has undergone many changes and improvements. The language was standardized in 1998 by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) by incorporating not only the new features but also the changes suggested by the user groups. The present edition completely follows the specifications of ANSI/ISO Standards.

Object Oriented Programming with C++ is for the programmers who wish to know all about C++ language and object-oriented programming. It explains in a simple and easy-to-understand style the what, why and how of object-oriented programming with C++.

The book has detailed coverage on various data structure concepts using object oriented programming.

The book provides numerous examples, illustrations and complete programs. The sample programs are meant to be both simple and educational. Wherever necessary, pictorial descriptions of concepts have been included to improve clarity and facilitate better understanding. The book also presents the concept of object-oriented approach and discusses important elements of object-oriented analysis and design of systems.

FIRST EDITION

With each edition, I have worked hard to provide a book that is useful, interesting, and engaging to students while introducing them to the core concepts. Keeping this in mind, the existing topics have been strengthened in the present edition with the enhancement of the pedagogical topics, based on the readers' suggestions. Corrections to typographical errors have also been incorporated.

- 20 New Programs and Review Questions (As per University Pattern)
- Refreshed Minor and Major Projects
- Revived and enriched pedagogy includes
 - *168 Programs*
 - *130 Review Questions*

SALIENT FEATURES OF THE BOOK

- Simple and easy-to-understand language

- Extended coverage to topics like *Overloading, Abstract classes, Typecasting, Dynamic Memory Allocation, Structures and Unions, and RTTI*
- Projects to enhance programming skills
- **Refreshed** C++ proficiency test *covering latest technical interview questions along with answers*
- **Refreshed and enhanced** pedagogy which includes over *500 pedagogical aids like programs, review questions, exercise with answers, programming exercises, and special boxes with notes*

CHAPTER ORGANIZATION

The book is organized into 5 units and 17 chapters. **Unit 1** is on Data Abstraction and Overloading which is covered in detail from Chapters 1-7. **Unit -2** is on Inheritance and Polymorphism which is dealt in detail in chapters 8 and 9.

Unit 3 explains linear data structures which is covered in chapters 10-14 while **Unit 4** deals with Non-Linear data structures which is covered in chapters 15 and 16. **Unit 5** deals with Sorting and searching which is present in Chapter 17.

Each chapter ends with a set of summary, key terms, review questions (the frequently asked universities' questions earmarked with  icon), debugging exercises and programming problems. For further assistance, students can also refer to the website (<http://www.mhhe.com/balagurusamy/oopds/au14>) for solutions to questions containing the  icon. In addition, two new projects – PR2PO (Major) and Credit CalC (Minor) are included in the present edition. Executing Turbo C++, Executing C++ under Windows, Glossary of ANSI C++ Keywords, C++ Operator Precedence, Points to Remember, Glossary of Important C++ and OOP Terms, and C++ Proficiency Test are provided in the appendices; helping students understand and implement the concepts in a better way and proving to be valuable source and tool in their study.

ACKNOWLEDGEMENTS

My sincere thanks to Mr. Deepak Manoj from University of Florida, Mrs. Bhuvanalakshmi from Adhiparasakthi Engineering College and publishing professionals from McGraw-Hill Education for their keen interest and support in bringing out this edition in its present form.

E Balagurusamy

FEEDBACK

Remember to write to us. We look forward to receiving your feedback, comments and ideas to enhance the quality of this book. You can reach us at info.india@mheducation.com. Please mention the title and the author's name as the subject.

In case you spot piracy of this book, in any form, please do let us know.

UNIT 1

Data Abstraction and Overloading

- **Chapter 1:** Principles of Object-Oriented Programming
- **Chapter 2:** Beginning with C++
- **Chapter 3:** Tokens, Expressions and Control Structures
- **Chapter 4:** Classes and Objects
- **Chapter 5:** Functions in C++
- **Chapter 6:** Constructors and Destructors
- **Chapter 7:** Operator Overloading and Type Conversions

1

PRINCIPLES OF OBJECT-ORIENTED PROGRAMMING

C H A P T E R O U T L I N E

- 1.1 Software Crisis
- 1.2 Software Evolution
- 1.3 A Look at Procedure-Oriented Programming
- 1.4 Object-Oriented Programming Paradigm
- 1.5 Basic Concepts of Object-Oriented Programming
 - 1.5.1 Objects
 - 1.5.2 Classes
 - 1.5.3 Data Abstraction and Encapsulation
 - 1.5.4 Inheritance
 - 1.5.5 Polymorphism
 - 1.5.6 Dynamic Binding
 - 1.5.7 Message Passing
- 1.6 Benefits of OOP
- 1.7 Object-Oriented Languages
- 1.8 Applications of OOP
- Summary*
- Key Terms*





1.1 SOFTWARE CRISIS

Developments in software technology continue to be dynamic. New tools and techniques are announced in quick succession. This has forced the software engineers and industry to continuously look for new approaches to software design and development, and they are becoming more and more critical in view of the increasing complexity of software systems as well as the highly competitive nature of the industry. These rapid advances appear to have created a situation of crisis within the industry. The following issues need to be addressed to face this crisis:

- How to represent real-life entities of problems in system design?
- How to design systems with open interfaces?
- How to ensure reusability and extensibility of modules?
- How to develop modules that are tolerant to any changes in future?
- How to improve software productivity and decrease software cost?
- How to improve the quality of software?
- How to manage time schedules?
- How to industrialize the software development process?

Many software products are either not finished, or not used, or else are delivered with major errors. Figure 1.1 shows the fate of the US defence software projects undertaken in the 1970s. Around 50% of the software products were never delivered, and one-third of those which were delivered were never used. It is interesting to note that only 2% were used as delivered, without being subjected to any changes. This illustrates that the software industry has a remarkably bad record in delivering products.

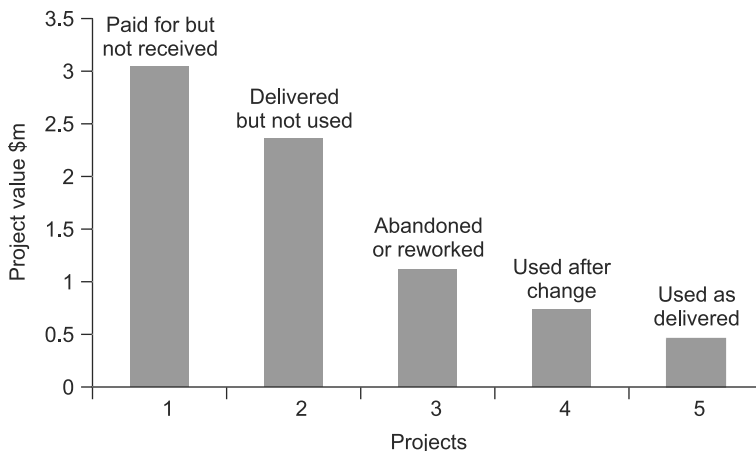


Fig. 1.1 *The state of US defence projects (according to the US government)*

Changes in user requirements have always been a major problem. Another study (Fig. 1.2) shows that more than 50% of the systems required modifications due to changes in user requirements and data formats. It only illustrates that, in a changing world with a dynamic business environment, requests for change are unavoidable and, therefore, systems must be adaptable and tolerant to changes.

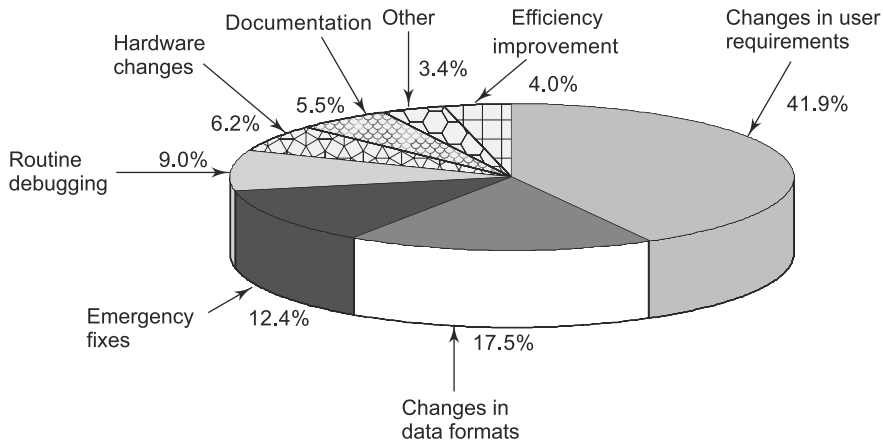


Fig. 1.2 Breakdown of maintenance costs

These studies and other reports on software implementation suggest that software products should be evaluated carefully for their quality before they are delivered and implemented. Some of the quality issues that must be considered for critical evaluation are

1. Correctness
2. Maintainability
3. Reusability
4. Openness and interoperability
5. Portability
6. Security
7. Integrity
8. User friendliness

Selection and use of proper software tools would help resolve some of these issues.



1.2 SOFTWARE EVOLUTION

Ernest Tello, a well-known writer in the field of artificial intelligence, compared the evolution of software technology to the growth of a tree. Like a tree, the software evolution has had distinct phases or “layers” of growth. These layers were built up one by one over the last five decades as shown in Fig. 1.3, with each layer representing an improvement over the previous one. However, the analogy fails if we consider the life of these layers. In software systems, each of the layers continues to be functional, whereas in the case of trees, only the uppermost layer is functional.

Alan Kay, one of the promoters of the object-oriented paradigm and the principal designer of Smalltalk, has said: “As *complexity increases, architecture dominates the basic material*”. To build today’s complex software, it is just not enough to put together a sequence of programming statements and sets of procedures and modules; we need to incorporate sound construction techniques and program structures that are easy to comprehend, implement and modify.

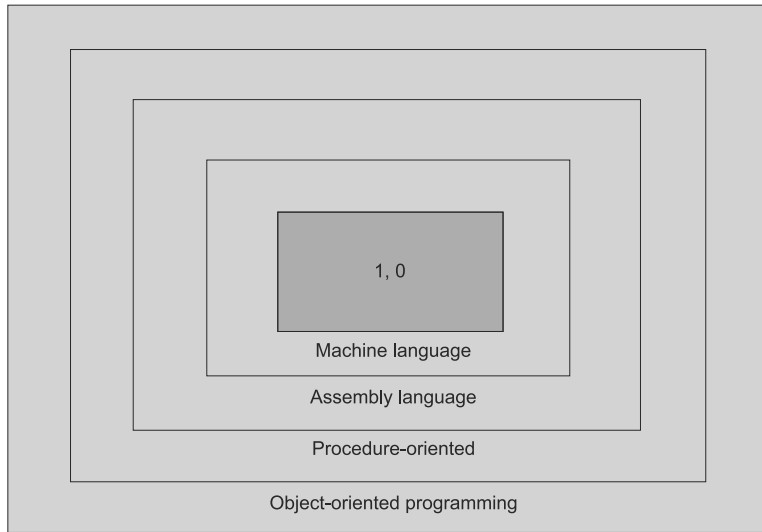


Fig. 1.3 Layers of computer software

Since the invention of the computer, many programming approaches have been tried. These include techniques such as *modular programming*, *top-down programming*, *bottom-up programming* and *structured programming*. The primary motivation in each has been the concern to handle the increasing complexity of programs that are reliable and maintainable. These techniques have become popular among programmers over the last two decades.

With the advent of languages such as C, structured programming became very popular and was the main technique of the 1980s. Structured programming was a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs.

Object-Oriented Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily.



1.3 A LOOK AT PROCEDURE-ORIENTED PROGRAMMING

Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as *procedure-oriented programming (POP)*. In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions. A typical program structure for procedural programming is shown in Fig. 1.4. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

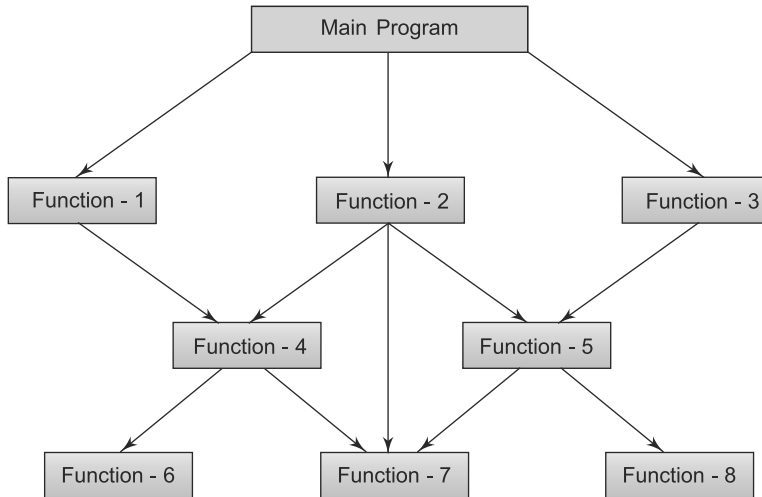


Fig. 1.4 Typical structure of procedure-oriented programs

Procedure-oriented programming basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as *functions*. We normally use a *flowchart* to organize these actions and represent the flow of control from one action to another. While we concentrate on the development of functions, very little attention is given to the data that are being used by various functions. What happens to the data? How are they affected by the functions that work on them?

In a multi-function program, many important data items are placed as *global* so that they may be accessed by all the functions. Each function may have its own *local data*. Figure 1.5 shows the relationship of data and functions in a procedure-oriented program.

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

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

Some characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs *top-down* approach in program design.

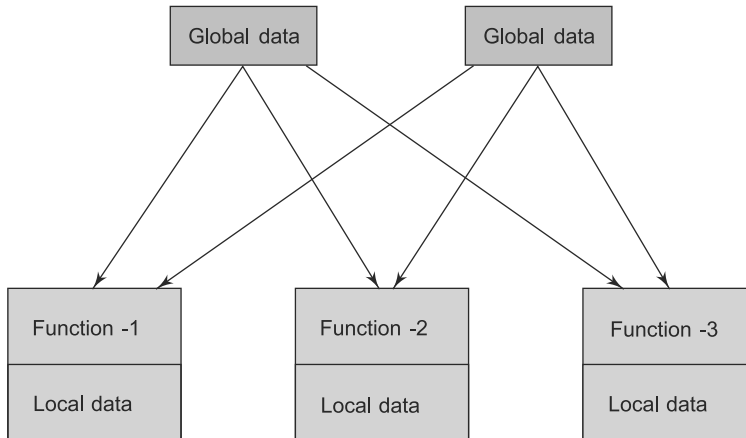


Fig. 1.5 Relationship of data and functions in procedural programming



1.4 OBJECT-ORIENTED PROGRAMMING PARADIGM

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called *objects* and then builds data and functions around these objects. The organization of data and functions in object-oriented programs is shown in Fig. 1.6. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

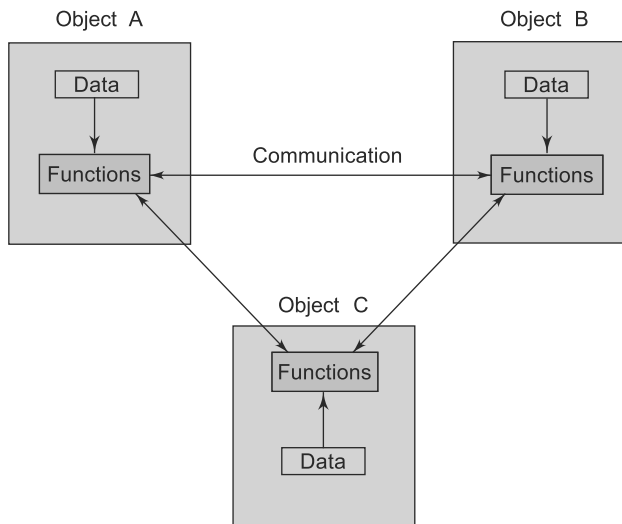


Fig. 1.6 Organization of data and functions in OOP

Some of the striking features of object-oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- *Follows bottom-up approach in program design.*

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people. It is therefore important to have a working definition of object-oriented programming before we proceed further. We define “object-oriented programming as *an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.*” Thus, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.



1.5 BASIC CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

We shall discuss these concepts in some detail in this section.

1.5.1 Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in C.

When a program is executed, the objects interact by sending messages to one another. For example, if “customer” and “account” are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data, and code to manipulate

the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted, and the type of response returned by the objects. Although different authors represent them differently, Fig. 1.7 shows two notations that are popularly used in object-oriented analysis and design.

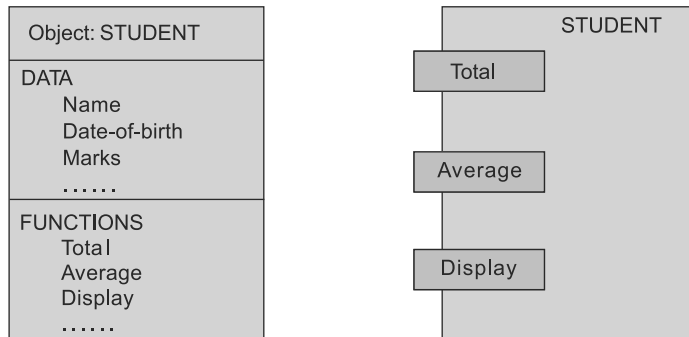


Fig. 1.7 Two ways of representing an object

1.5.2 Classes

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a *class*. In fact, objects are variables of the type *class*. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language. The syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

```
fruit mango;
```

will create an object **mango** belonging to the class **fruit**.

1.5.3 Data Abstraction and Encapsulation

The wrapping up of data and functions into a single unit (called class) is known as *encapsulation*. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding* or *information hiding*.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract *attributes* such as size, weight and cost, and *functions* to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called *data members* because they hold information. The functions that operate on these data are sometimes called *methods* or *member functions*.

Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT).

1.5.4 Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of *hierarchical classification*. For example, the bird 'robin' is a part of the class 'flying bird' which is again a part of the class 'bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in Fig. 1.8.

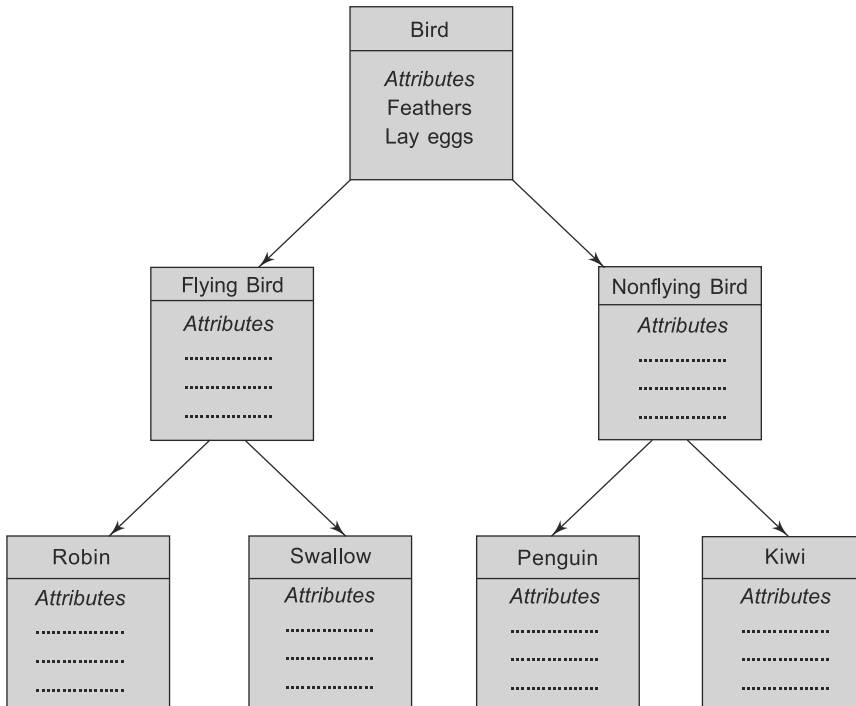


Fig. 1.8 Property inheritance

In OOP, the concept of inheritance provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of the classes.

Note that each sub-class defines only those features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

1.5.5 Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the

operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as *operator overloading*.

Figure 1.9 illustrates that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context. Using a single function name to perform different types of tasks is known as *function overloading*.

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

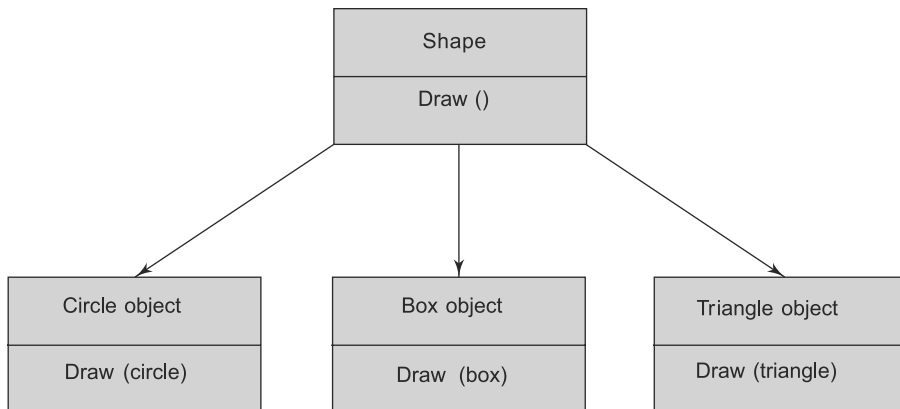


Fig. 1.9 Polymorphism

1.5.6 Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. *Dynamic binding* (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure “draw” in Fig. 1.9. By inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

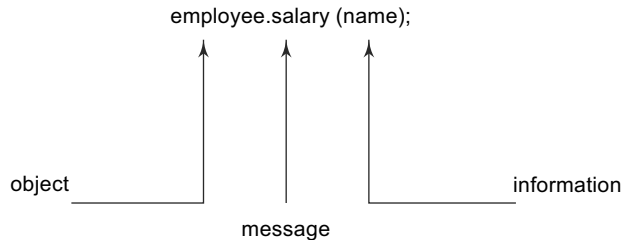
1.5.7 Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behavior,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

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



Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.



1.6 BENEFITS OF OOP

OOP offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

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

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. There are a number of issues that need to be tackled to reap some of the benefits stated above. For instance, object libraries must be available for reuse. The technology is still developing and current products may be superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.

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



1.7 OBJECT-ORIENTED LANGUAGES

Object-oriented programming is not the right of any particular language. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially designed to support the OOP concepts makes it easier to implement them.

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

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

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object-based programming are:

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

Object-oriented programming incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statement:

Object-based features + inheritance + dynamic binding

Languages that support these features include C++, Smalltalk, Object Pascal and Java. There are a large number of object-based and object-oriented programming languages. Table 1.1 lists some popular general purpose OOP languages and their characteristics.

Table 1.1 Characteristics of some OOP languages

Characteristics	Simula *	Smalltalk *	Objective C	C++	Ada **	Object Pascal	Turbo Pascal	Eiffel *	Java *
Binding (early or late)	Both	Late	Both	Both	Early	Late	Early	Early	Both
Polymorphism	✓	✓	✓	✓	✓	✓	✓	✓	✓
Data hiding	✓	✓	✓	✓	✓	✓	✓	✓	✓
Concurrency	✓	Poor	Poor	Poor	Difficult	No	No	Promised	✓
Inheritance	✓	✓	✓	✓	No	✓	✓	✓	✓
Multiple Inheritance	No	✓	✓	✓	No	---	---	✓	No

(Contd.)

Table 1.1 (Contd.)

Characteristics	Simula *	Smalltalk *	Objective C	C++	Ada **	Object Pascal	Turbo Pascal	Eiffel *	Java *
Garbage Collection	✓	✓	✓	✓	No	✓	✓	✓	✓
Persistence	No	Promised	No	No	like 3GL	No	No	Some Support	✓
Genericity	No	No	No	✓	✓	No	No	✓	No
Object Libraries	✓	✓	✓	✓	Not much	✓	✓	✓	✓

* Pure object-oriented languages

** Object-based languages

Others are extended conventional languages

As seen from Table 1.1, all languages provide for polymorphism and data hiding. However, many of them do not provide facilities for concurrency, persistence and genericity. Eiffel, Ada and C++ provide generic facility which is an important construct for supporting reuse. However, persistence (a process of storing objects) is not fully supported by any of them. In Smalltalk, though the entire current execution state can be saved to disk, yet the individual objects cannot be saved to an external file.

Simula is one of the oldest object oriented programming languages, though it has spent most of its life in a research environment. Commercially, C++ and Java are the most prominent object oriented programming languages that have found extensive usage in application development.

Use of a particular language depends on characteristics and requirements of an application, organizational impact of the choice, and reuse of the existing programs. C++ has now become the most successful, practical, general-purpose OOP language, and is widely used in industry today.



1.8 APPLICATIONS OF OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as windows. Hundreds of windowing systems have been developed, using the OOP techniques.

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem. The promising areas for application of OOP include:

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext, hypermedia and experttext
- AI and expert systems
- Neural networks and parallel programming

- Decision support and office automation systems
- CIM/CAM/CAD systems

The richness of OOP environment has enabled the software industry to improve not only the quality of software systems but also its productivity. Object-oriented technology is certainly changing the way the software engineers think, analyze, design and implement systems.

SUMMARY

- ☐ Software technology has evolved through a series of phases during the last five decades.
- ☐ The most popular phase till recently was procedure-oriented programming (POP).
- ☐ POP employs *top-down* programming approach where a problem is viewed as a sequence of tasks to be performed. A number of functions are written to implement these tasks.
- ☐ POP has two major drawbacks, viz. (1) data move freely around the program and are therefore vulnerable to changes caused by any function in the program, and (2) it does not model very well the real-world problems.
- ☐ Object-oriented programming (OOP) was invented to overcome the drawbacks of the POP. It employs the *bottom-up* programming approach. It treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it in a data structure called **class**. This feature is called *data encapsulation*.
- ☐ In OOP, a problem is considered as a collection of a number of entities called **objects**. Objects are instances of classes.
- ☐ Insulation of data from direct access by the program is called *data hiding*.
- ☐ *Data abstraction* refers to putting together essential features without including background details.
- ☐ *Inheritance* is the process by which objects of one class acquire properties of objects of another class.
- ☐ *Polymorphism* means one name, multiple forms. It allows us to have more than one function with the same name in a program. It also allows overloading of operators so that an operation can exhibit different behaviors in different instances.
- ☐ *Dynamic binding* means that the code associated with a given procedure is not known until the time of the call at run-time.
- ☐ *Message passing* involves specifying the name of the object, the name of the function (message) and the information to be sent.
- ☐ Object-oriented technology offers several benefits over the conventional programming methods—the most important one being the reusability.
- ☐ Applications of OOP technology has gained importance in almost all areas of computing including real-time business systems.
- ☐ There are a number of languages that support object-oriented programming paradigm. Popular among them are C++, Smalltalk and Java. C++ has become an industry standard language today.

Key Terms

Ada | assembly language | bottom-up programming | C++ | classes | concurrency | data abstraction | data encapsulation | data hiding | data members | dynamic binding | early binding | Eiffel | flowcharts | function overloading | functions | garbage collection | global data | hierarchical classification | inheritance | Java | late binding | local data | machine language | member functions | message passing | methods | modular programming | multiple inheritance | object libraries | Object Pascal | object-based programming | Objective C | object-oriented languages | object-oriented programming | objects | operator overloading | persistence | polymorphism | procedure-oriented programming | reusability | Simula | Smalltalk | structured programming | top-down programming | Turbo Pascal

Review Questions

- 1.1 What do you think are the major issues facing the software industry today?
- 1.2 Briefly discuss the software evolution during the period 1950 – 1990.
- 1.3 What is procedure-oriented programming? What are its main characteristics?
- 1.4 Discuss an approach to the development of procedure-oriented programs.
- 1.5 Describe how data are shared by functions in a procedure-oriented program.
- 1.6 What is object-oriented programming? How is it different from the procedure-oriented programming?
- 1.7 How are data and functions organized in an object-oriented program?
- 1.8 What are the unique advantages of an object-oriented programming paradigm?
- 1.9 Distinguish between the following terms:
 - (a) Objects and classes
 - (b) Data abstraction and data encapsulation 
 - (c) Inheritance and polymorphism
 - (d) Dynamic binding and message passing
- 1.10 What kinds of things can become objects in OOP?
- 1.11 Describe inheritance as applied to OOP.
- 1.12 What do you mean by dynamic binding? How is it useful in OOP?
- 1.13 How does object-oriented approach differ from object-based approach?
- 1.14 List a few areas of application of OOP technology.
- 1.15 State whether the following statements are TRUE or FALSE.
 - (a) In procedure-oriented programming, all data are shared by all functions.
 - (b) The main emphasis of procedure-oriented programming is on algorithms rather than on data.
 - (c) One of the striking features of object-oriented programming is the division of programs into objects that represent real-world entities.
 - (d) Wrapping up of data of different types into a single unit is known as encapsulation.

- (e) One problem with OOP is that once a class is created it can never be changed.
- (f) Inheritance means the ability to reuse the data values of one object by another object.
- (g) Polymorphism is extensively used in implementing inheritance.
- (h) Object-oriented programs are executed much faster than conventional programs.
- (i) Object-oriented systems can scale up better from small to large.
- (j) Object-oriented approach cannot be used to create databases.

BEGINNING WITH C++

- 2.1 What is C++?
- 2.2 Application of C++
- 2.3 A Simple C++ Program
 - 2.3.1 Program Features
 - 2.3.2 Comments
 - 2.3.3 Output Operator
 - 2.3.4 The Iostream File
 - 2.3.5 The Type of main()
- 2.4 More C++ Statements
 - 2.4.1 Variables
 - 2.4.2 Input Operator
 - 2.4.3 Cascading of I/O Operators
- 2.5 An Example with Class
- 2.6 Structure of A C++ Program
- 2.7 Creating the Source File
- 2.8 Compiling and Linking
 - 2.8.1 Unix AT&T C++
 - 2.8.2 Turbo C++ and Borland C++
 - 2.8.3 Visual C++

Summary

Key Terms





2.1 WHAT IS C++?

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.

During the early 1990's the language underwent a number of improvements and changes. In November 1997, the ANSI/ISO standards committee standardised these changes and added several new features to the language specifications.

C++ is a superset of C. Most of what we already know about C applies to C++ also. Therefore, almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ compiler. We shall see these differences later as and when they are encountered.

The most important facilities that C++ adds on to C are classes, inheritance, function overloading, and operator overloading. These features enable creation of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

The object-oriented features in C++ allow programmers to build large programs with clarity, extensibility and ease of maintenance, incorporating the spirit and efficiency of C. The addition of new features has transformed C from a language that currently facilitates top-down, structured design, to one that provides bottom-up, object-oriented design.



2.2 APPLICATIONS OF C++

C++ is a versatile language for handling very large programs. It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real-life application systems.

- Since C++ allows us to create hierarchy-related objects, we can build special object-oriented libraries which can be used later by many programmers.
- While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine-level details.
- C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.
- It is expected that C++ will replace C as a general-purpose language in the near future.



2.3 A SIMPLE C++ PROGRAM

Let us begin with a simple example of a C++ program that prints a string on the screen.

Program 2.1 Printing a String

```
#include <iostream>                                // include header file

using namespace std;
int main()
{
    cout << "C++ is better than C.\n";  // C++ statement

    return 0;

}                                                    // End of example
```

This simple program demonstrates several C++ features.

2.3.1 Program Features

Like C, the C++ program is a collection of functions. The above example contains only one function, **main()**. As usual, execution begins at **main()**. Every C++ program must have a **main()**. C++ is a free-form language. With a few exceptions, the compiler ignores carriage returns and white spaces. Like C, the C++ statements terminate with semicolons.

2.3.2 Comments

C++ introduces a new comment symbol **//** (double slash). Comments start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line, and whatever follows till the end of the line is ignored. Note that there is no closing symbol.

The double slash comment is basically a single line comment. Multiline comments can be written as follows:

```
// This is an example of
// C++ program to illustrate
// Some of its features
```

The C comment symbols **/*, */** are still valid and are more suitable for multiline comments. The following comment is allowed:

```
/* This is an example of
   C++ program to illustrate
   some of its features
*/
```

We can use either or both styles in our programs. Since this is a book on C++, we will use only the C++ style. However, remember that we can not insert a **//** style comment within the text of a program line. For example, the double slash comment cannot be used in the manner as shown below:

```
for(j=0; j<n; /* loops n times */ j++)
```

2.3.3 Output Operator

The only statement in Program 2.1 is an output statement. The statement

```
cout << "C++ is better than C.";
```

causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, `cout` and `<<`. The identifier `cout` (pronounced as 'C out') is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices. We shall later discuss streams in detail.

The operator `<<` is called the *insertion or put* to operator. It inserts (or sends) the contents of the variable on its right to the object on its left (Fig. 2.1).

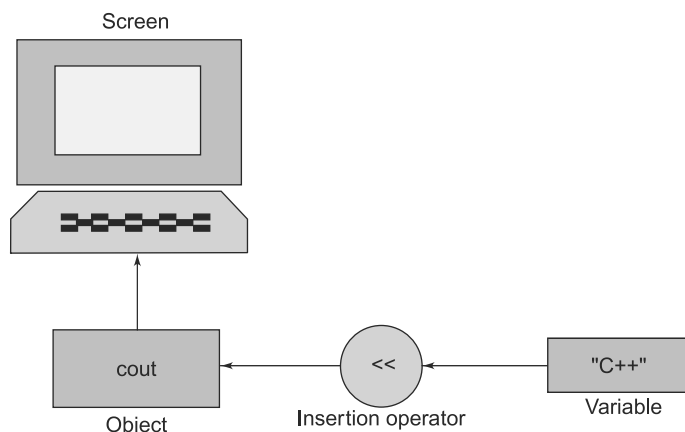


Fig. 2.1 Output using insertion operator

The object **cout** has a simple interface. If `string` represents a string variable, then the following statement will display its contents:

```
cout << string;
```

You may recall that the operator `<<` is the bit-wise left-shift operator and it can still be used for this purpose. This is an example of how one operator can be used for different purposes, depending on the context. This concept is known as *operator overloading*, an important aspect of polymorphism. Operator overloading is discussed in detail in Chapter 7.

It is important to note that we can still use `printf()` for displaying an output. C++ accepts this notation. However, we will use `cout <<` to maintain the spirit of C++.

2.3.4 The `iostream` File

A C++ program typically contains pre-processor directive statements at the beginning. Such statements are preceded with a `#` symbol to indicate the presence of a pre-processor directive to the compiler, which in turn lets the pre-processor handle the `#` directive statement. All C++ programs begin with a `#include` directive that includes the specified header file contents into the main program.

We have used the following `#include` directive in the program:

```
#include <iostream>
```

This directive causes the preprocessor to add the contents of the `iostream` file to the program. It contains declarations for the identifier **cout** and the operator `<<`. Some old versions of C++ use a header file called `iostream.h`. This is one of the changes introduced by ANSI C++. (We should use `iostream.h` if the compiler does not support ANSI C++ features.)

The header file **`iostream`** should be included at the beginning of all programs that use input/output statements. Note that the naming conventions for header files may vary. Some implementations use **`iostream.hpp`**; yet others **`iostream.hxx`**. We must include appropriate header files depending on the contents of the program and implementation.

Tables 2.1 and 2.2 provide lists of C++ standard library header files that may be needed in C++ programs. The header files with `.h` extension are “old style” files which should be used with old compilers. Table 2.1 also gives the version of these files that should be used with the ANSI standard compilers.

Table 2.1 *Commonly used old-style header files*

Header file	Contents and purpose	New version
<code><assert.h></code>	Contains macros and information for adding diagnostics that aid program debugging	<code><cassert></code>
<code><ctype.h></code>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.	<code><cctype></code>
<code><float.h></code>	Contains the floating-point size limits of the system.	<code><cfloat></code>
<code><limits.h></code>	Contains the integral size limits of the system.	<code><climits></code>
<code><math.h></code>	Contains function prototypes for math library functions.	<code><cmath></code>
<code><stdio.h></code>	Contains function prototypes for the standard input/output library functions and information used by them.	<code><cstdio></code>
<code><stdlib.h></code>	Contains function prototypes for conversion of numbers to text, text to numbers, memory allocation, random numbers, and various other utility functions.	<code><cstdlib></code>
<code><string.h></code>	Contains function prototypes for C-style string processing functions.	<code><cstring></code>
<code><time.h></code>	Contains function prototypes and types for manipulating the time and date.	
<code><iostream.h></code>	Contains function prototypes for the standard input and standard output functions.	<code><iostream></code>
<code><iomanip.h></code>	Contains function prototypes for the stream manipulators that enable formatting of streams of data.	<code><iomanip></code>
<code><fstream.h></code>	Contains function prototypes for functions that perform input from files on disk and output to files on disk.	<code><fstream></code>

Table 2.2 *New header files included in ANSI C++*

Header file	Contents and purpose
<code><utility></code>	Contains classes and functions that are used by many standard library header files.
<code><vector></code> , <code><list></code> , <code><deque></code> <code><queue></code> , <code><set></code> , <code><map></code> , <code><stack></code> , <code><bitset></code>	The header files contain classes that implement the standard library containers. Containers store data during a program's execution. We discuss these header files in Chapter 14.

(Contd.)

Table 2.2 (Contd.)

Header file	Contents and purpose
<functional>	Contains classes and functions used by algorithms of the standard library.
<memory>	Contains classes and functions used by the standard library to allocate memory to the standard library containers.
<iterator>	Contains classes for manipulating data in the standard library containers.
<algorithm>	Contains functions for manipulating data in the standard library containers.
<exception>, <stdexcept>	These header files contain classes that are used for exception handling.
<string>	Contains the definition of class string from the standard library. Discussed in Chapter 15
<sstream>	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory.
<locale>	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.)
<limits>	Contains a class for defining the numerical data type limits on each computer platform.
<typeinfo>	Contains classes for run-time type identification (determining data types at execution time).

Apart from `#include`, the other pre-processor directives such as `#define`, `#error`, etc., work the same way as they would in C. But, their usage in C++ is quite limited due to the availability of other means for achieving the same functionality.

Namespace

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the **namespace** scope we must include the using directive, like

```
using namespace std;
```

Here, **std** is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in **std** to the current global scope. **using** and **namespace** are the new keywords of C++. Namespaces are discussed in detail in Chapter 16.

2.3.5 Return Type of main()

In C++, `main()` returns an integer type value to the operating system. Therefore, every `main()` in C++ should end with a `return(0)` statement; otherwise a warning or an error might occur. Since **main()** returns an integer type value, return type for `main()` is explicitly specified as **int**. Note that the default return type for all functions in C++ is **int**. The following **main** without type and return will run with a warning:

```
main()
{
    .....
    .....
}
```



2.4 MORE C++ STATEMENTS

Let us consider a slightly more complex C++ program. Assume that we would like to read two numbers from the keyboard and display their average on the screen. C++ statements to accomplish this is shown in Program 2.2.

Program 2.2 Average of Two Numbers

```
#include <iostream>

using namespace std;

int main()
{
    float number1, number2, sum, average;
    cout << "Enter two numbers: "; // prompt
    cin  >> number1;               // Reads numbers
    cin  >> number2;               // from keyboard

    sum = number1 + number2;
    average = sum/2;

    cout << "Sum = " << sum << "\n";
    cout << "Average = " << average << "\n";

    return 0;
}
```

The output of Program 2.2 would be

```
Enter two numbers: 6.5 7.5
Sum = 14
Average = 7
```

2.4.1 Variables

The program uses four variables: number1, number2, sum, and average. They are declared as type float by the statement.

```
float number1, number2, sum, average;
```

All variables must be declared before they are used in the program.

2.4.2 Input Operator

The statement

```
cin >> number1;
```

is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable `number1`. The identifier **cin** (pronounced 'C in') is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator `>>` is known as *extraction or get from* operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right (Fig. 2.2). This corresponds to the familiar `scanf()` operation. Like `<<`, the operator `>>` can also be overloaded.

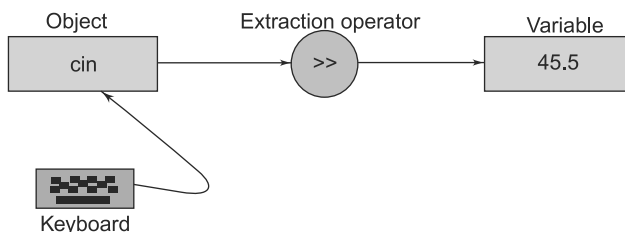


Fig. 2.2 Input using extraction operator

2.4.3 Cascading of I/O Operators

We have used the *insertion operator* `<<` repeatedly in the last two statements for printing results.

The statement

```
cout << "Sum = " << sum << "\n";
```

first sends the string "Sum =" to `cout` and then sends the value of `sum`. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of `<<` in one statement is called *cascading*. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

```
cout << "Sum = " << sum << "\n"
    << "Average = " << average << "\n";
```

This is one statement but provides two lines of output. If you want only one line of output, the statement will be

```
cout << "Sum = " << sum << ", "
    << "Average = " << average << "\n";
```

The output will be

```
Sum = 14, Average = 7
```

We can also cascade input operator `>>` as shown below:

```
cin >> number1 >> number2;
```

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to `number1` and 20 to `number2`.



2.5 AN EXAMPLE WITH CLASS

One of the major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Like structures in C, classes are user-defined data types.

Program 2.3 shows the use of class in a C++ program.

Program 2.3 Use of Class

```
#include <iostream>
using namespace std;
class person
{
    char name[30];
    int age;

    public:
        void getdata(void);
        void display(void);
};
void person :: getdata(void)
{
    cout << "Enter name: ";
    cin >> name;
    cout << "Enter age: ";
    cin >> age;
}
void person :: display(void)
{
    cout << "\nName: " << name;
    cout << "\nAge: " << age;
}

int main()
{
    person p;
    p.getdata();
    p.display();

    return 0;
}
```

The output of Program 2.3 would be

Enter Name: Ravinder

Enter Age: 30

Name: Ravinder

Age: 30



Note *cin can read only one word and therefore we cannot use names with blank spaces.*

The program defines **person** as a new data of type class. The class **person** includes two basic data type items and two functions to operate on that data. These functions are called **member functions**. The main program uses **person** to declare variables of its type. As pointed out earlier, class variables are known as *objects*. Here, **p** is an object of type **person**. Class objects are used to invoke the functions defined in that class. More about classes and objects is discussed in Chapter 5.



2.6 STRUCTURE OF A C++ PROGRAM

As it can be seen from the Program 2.3, a typical C++ program would contain four sections as shown in Fig. 2.3. These sections may be placed in separate code files and then compiled independently or jointly.

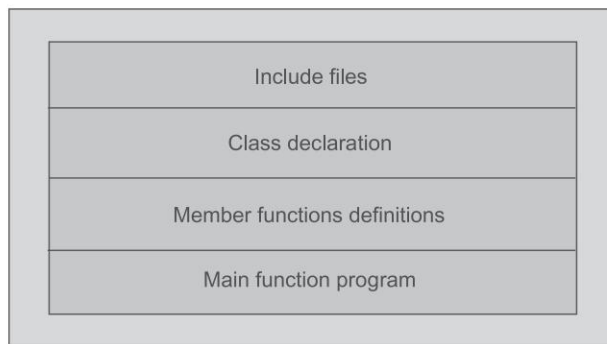


Fig. 2.3 *Structure of a C ++ program*

It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification of the interface (class definition) from the implementation details (member functions definition). Finally, the main program that uses the class is placed in a third file which “includes” the previous two files as well as any other files required.

This approach is based on the concept of client-server model as shown in Fig. 2.4. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

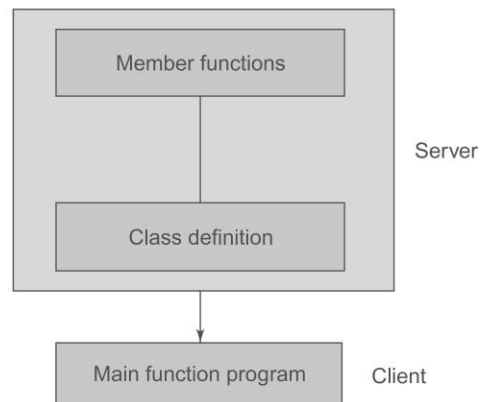


Fig. 2.4 *The client-server model*



2.7 CREATING THE SOURCE FILE

Like C programs, C++ programs can be created using any text editor. For example, on the UNIX, we can use *vi* or *ed* text editor for creating and editing the source code. On the DOS system, we can use *edlin* or any other editor available or a word processor system under non-document mode.

Some systems such as Turbo C++ provide an integrated environment for developing and editing programs. Appropriate manuals should be consulted for complete details.

The file name should have a proper file extension to indicate that it is a C++ program file. C++ implementations use extensions such as *.c*, *.C*, *.cc*, *.cpp* and *.cxx*. Turbo C++ and Borland C++ use *.c* for C programs and *.cpp* (C plus plus) for C++ programs. Zortech C++ system uses *.cxx* while UNIX AT&T version uses *.C* (capital C) and *.cc*. The operating system manuals should be consulted to determine the proper file name extensions to be used.



2.8 COMPILING AND LINKING

The process of compiling and linking again depends upon the operating system. A few popular systems are discussed in this section.

2.8.1 Unix AT&T C++

The process of implementation of a C++ program under UNIX is similar to that of a C program. We should use the "CC" (uppercase) command to compile the program. Remember, we use lowercase "cc" for compiling C programs. The command

```
CC example.C
```

at the UNIX prompt would compile the C++ program source code contained in the file **example.C**. The compiler would produce an object file **example.o** and then automatically link with the library functions to produce an executable file. The default executable filename is **a.out**.

A program spread over multiple files can be compiled as follows:

```
CC file1.C file2.o
```

The statement compiles only the file **file1.C** and links it with the previously compiled **file2.o** file. This is useful when only one of the files, needs to be modified. The files that are not modified need not be compiled again.

2.8.2 Turbo C++ and Borland C++

Turbo C++ and Borland C++ provide an integrated program development environment under MS DOS. They provide a built-in editor and a menu bar which includes options such as File, Edit, Compile and Run.

We can create and save the source files under the **File** option, and edit them under the **Edit** option. We can then compile the program under the **Compile** option and execute it under the **Run** option. The **Run** option can be used without compiling the source code. In this case, the **RUN** command causes the system to compile, link and run the program in one step. Turbo C++ being the most popular compiler, creation and execution of programs under Turbo C++ system are discussed in detail in Appendix B.

2.8.3 Visual C++

It is a Microsoft application development system for C++ that runs under Windows. Visual C++ is a visual programming environment in which basic program components can be selected through menu choices, buttons, icons, and other predetermined methods.

SUMMARY

- ☐ C++ is a superset of C language.
- ☐ C++ adds a number of object-oriented features such as objects, inheritance, function overloading and operator overloading to C. These features enable building of programs with clarity, extensibility and ease of maintenance.
- ☐ C++ can be used to build a variety of systems such as editors, compilers, databases, communication systems, and many more complex real-life application systems.
- ☐ C++ supports interactive input and output features and introduces a new comment symbol `//` that can be used for single line comments. It also supports C-style comments.
- ☐ Like C programs, execution of all C++ programs begins at **main()** function and ends at **return()** statement. The header file **iostream** should be included at the beginning of all programs that use input/output operations.
- ☐ All ANSI C++ programs must include **using namespace std** directive.
- ☐ A typical C++ program would contain four basic sections, namely, include files section, class declaration section, member function section and main function section.
- ☐ Like C programs, C++ programs can be created using any text editor.
- ☐ Most compiler systems provide an integrated environment for developing and executing programs. Popular systems are UNIX AT&T C++, Turbo C++ and Microsoft Visual C++.

Key Terms

#include | **a.out** | Borland C++ | cascading | **cin** | class | client | comments | **cout** | edlin | extractionoperator | **float** | free-form | get from operator | input operator | insertion operator | **int** | **iostream** | **iostream.h** | keyboard | **main()** | member functions | MS-DOS | **namespace** | object | operating systems | operator overloading | output operator | put to operator | return () | screen | server | Simula67 | text editor | Turbo C++ | Unix AT&T C++ | **using** | Visual C++ | Windows | Zortech C++

Review Questions

- 2.1 State whether the following statements are TRUE or FALSE.
- (a) Since C is a subset of C++, all C programs will run under C++ compilers.
 - (b) In C++, a function contained within a class is called a member function.

- (c) Looking at one or two lines of code, we can easily recognize whether a program is written in C or C++.
- (d) In C++, it is very easy to add new features to the existing structure of an object.
- (e) The concept of using one operator for different purposes is known as operator overloading.
- (f) The output function printf() cannot be used in C++ programs.

2.2 Why do we need the preprocessor directive #include <iostream> ?

2.3 How does a main() function in C++ differ from main() in C?

2.4 What do you think is the main advantage of the comment // in C++ as compared to the old C type comment?

2.5 Describe the major parts of a C++ program.

Debugging Exercises

2.1 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
    int i = 0;
    i = i + 1;
    cout << i << " ";
    /*comment\*/i = i + 1;
    cout << i;
}
```

2.2 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
    short i=2500, j=3000;
    cout >> "i + j = " >> -(i+j);
}
```

2.3 What will happen when you run the following program?

```
#include <iostream.h>
void main()
{
    int i=10, j=5;
    int modResult=0;
    int divResult=0;

    modResult = i%j;
    cout << modResult << " ";

    divResult = i/modResult;
    cout << divResult;
}
```

2.4 Find errors, if any, in the following C++ statements.

- (a) `cout << "x=" x;`
- (b) `m = 5; // n = 10; // s = m + n;`
- (c) `cin >>x; >>y;`
- (d) `cout << \n "Name:" << name;`
- (e) `cout <<"Enter value:"; cin >> x;`
- (f) `/*Addition*/ z = x + y;`

Programming Exercises

2.1 Write a program to display the following output using a single `cout` statement.

```
Maths = 90
Physics = 77
Chemistry = 69
```

2.2 Write a program to read two numbers from the keyboard and display the larger value on the screen. WEB

2.3 Write a program that inputs a character from keyboard and displays its corresponding ASCII value on the screen. WEB

2.4 Write a program to read the values of `a`, `b` and `c` and display the value of `x`, where

$$x = a / b - c$$

Test your program for the following values:

(a) `a = 250, b = 85, c = 25`

(b) `a = 300, b = 70, c = 70` WEB

2.5 Write a C++ program that will ask for a temperature in Fahrenheit and display it in Celsius.

WEB

2.6 Redo Exercise 2.5 using a class called **temp** and member functions. WEB

TOKENS, EXPRESSIONS AND CONTROL STRUCTURES

3.1	Introduction	3.20	Expressions and Their Types
3.2	Tokens	3.20.1	Constant Expressions
3.3	Keywords	3.20.2	Integral Expressions
3.4	Identifiers and Constants	3.20.3	Float Expressions
3.5	Basic Data Types	3.20.4	Pointer Expressions
3.6	User-Defined Data Types	3.20.5	Relational Expressions
3.6.1	Structures and Unions	3.20.6	Logical Expressions
3.6.2	Classes	3.20.7	Bitwise Expressions
3.6.3	Enumerated Data Type	3.21	Special Assignment Expressions
3.7	Storage Classes	3.21.1	Chained Assignment
3.8	Derived Data Types	3.21.2	Embedded Assignment
3.8.1	Arrays	3.21.3	Compound Assignment
3.8.2	Functions	3.22	Implicit Conversions
3.8.3	Pointers	3.23	Operator Overloading
3.9	Symbolic Constants	3.24	Operator Precedence
3.10	Type Compatibility	3.25	Control Structures
3.11	Declaration of Variables	3.25.1	The if statement
3.12	Dynamic Initialization of Variables	3.25.2	The switch statement
3.13	Reference Variables	3.25.3	The do-while statement
3.14	Operators in C++	3.25.4	The while statement
3.15	Scope Resolution Operator	3.25.4	The for statement
3.16	Member Dereferencing Operators		<i>Summary</i>
3.17	Memory Management Operators		<i>Key Terms</i>
3.18	Manipulators		
3.19	Type Cast Operator		





3.1 INTRODUCTION

As mentioned earlier, C++ is a superset of C and therefore most constructs of C are legal in C++ with their meaning unchanged. However, there are some exceptions and additions. In this chapter, we shall discuss these exceptions and additions with respect to tokens and control structures.



3.2 TOKENS

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

- Keywords
- Identifiers
- Constants
- Strings
- Operators

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



3.3 KEYWORDS

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements.

Table 3.1 gives the complete set of C++ keywords. Many of them are common to both C and C++. The ANSI C keywords are shown in boldface. Additional keywords have been added to the ANSI C keywords in order to enhance its features and make it an object-oriented language. ANSI C++ standards committee has added some more keywords to make the language more versatile. These are shown separately.

Table 3.1 C++ keywords

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef
class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual

(Contd.)

default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while
<i>Added by ANSI C++</i>			
bool	export	reinterpret_cast	typename
const_cast	false	static_cast	using
dynamic_cast	mutable	true	wchar_t
explicit	namespace	typeid	

Note: The ANSI C keywords are shown in boldface.



3.4 IDENTIFIERS AND CONSTANTS

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

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

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

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

Constants refer to fixed values that do not change during the execution of a program.

Like C, C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings. Literal constant do not have memory locations. Examples:

```
123          // decimal integer
12.34        // floating point integer
037          // octal integer
0X2          // hexadecimal integer
"C++"        // string constant
'A'          // character constant
L'ab'        // wide-character constant
```

The **wchar_t** type is a wide-character literal introduced by ANSI C++ and is intended for character sets that cannot fit a character into a single byte. Wide-character literals begin with the letter L.

C++ also recognizes all the backslash character constants available in C.



Note C++ supports two types of string representation — the C-style character string and the string class type introduced with Standard C++. Although the use of the string class type is recommended, it is advisable to understand and use C-style strings in some situations. The string class type strings support many features and are discussed in detail in Chapter 15.



3.5 BASIC DATA TYPES

Data types in C++ can be classified under various categories as shown in Fig. 3.1.

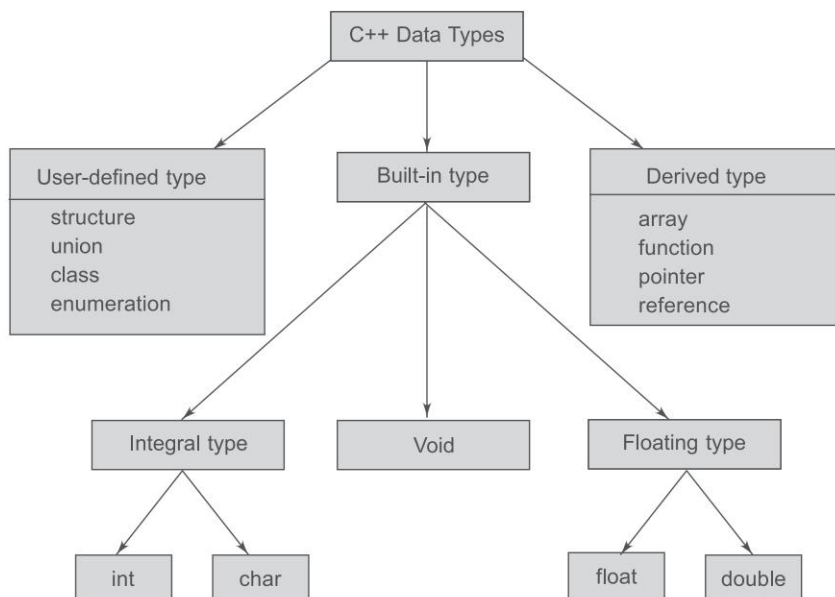


Fig. 3.1 Hierarchy of C++ data types

Both C and C++ compilers support all the built-in (also known as *basic* or *fundamental*) data types. With the exception of **void**, the basic data types may have several *modifiers* preceding them to serve the needs of various situations. The modifiers **signed**, **unsigned**, **long**, and **short** may be applied to character and integer basic data types. However, the modifier **long** may also be applied to **double**. Data type representation is machine specific in C++. Table 3.2 lists all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine.

ANSI C++ committee has added two more data types, **bool** and **wchar_t**. They are discussed in Chapter 16.

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

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

Table 3.2 Size and range of C++ basic data types

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

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

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

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

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

are valid statements. But, the statement,

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

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

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

```
void *ptr1;
char *ptr2;
ptr2 = ptr1;
```

are all valid statements in ANSI C but not in C++. A **void** pointer cannot be directly assigned to other type pointers in C++. We need to use a cast operator as shown below:

```
ptr2 = (char *)ptr1;
```




3.6 USER-DEFINED DATA TYPES

3.6.1 Structures and Unions

Standalone variables of primitive types are not sufficient enough to handle real world problems. It is often required to group logically related data items together. While arrays are used to group together similar type data elements, structures are used for grouping together elements with dissimilar types.

The general format of a structure definition is as follows:

```
struct name
{
    data_type member1;
    data_type member2;
    . . . . .
    . . . . .
};
```

Let us take the example of a book, which has several attributes such as title, number of pages, price, etc. We can realize a book using structures as shown below:

```
struct book
{
    chartitle[25];
    charauthor[25];
    int pages;
    float price;
};
struct book book1, book2, book3;
```

Here book1, book2 and book3 are declared as variables of the user-defined type book. We can access the member elements of a structure by using the dot (.) operator, as shown below:

```
book1.pages=550;
book2.price=225.75;
```

Unions are conceptually similar to structures as they allow us to group together dissimilar type elements inside a single unit. But there are significant differences between structures and unions as far as their implementation is concerned. The size of a structure type is equal to the sum of the sizes of individual member types. However, the size of a union is equal to the size of its largest member element. For instance, consider the following union declaration:

```
union result
{
    int marks;
    char grade;
    float percent;
};
```


The union result will occupy four bytes in memory as its largest size member element is the floating type variable per cent. However, if we had defined result as a structure then it would have occupied seven bytes in memory that is, the sum of the sizes of individual member elements. Thus, in case of unions the same memory space is used for representing different member elements. As a result, union members can only be manipulated exclusive of each other. In simple words, we can say that unions are memory-efficient alternatives of structures particularly in situations where it is not required to access the different member elements simultaneously.

In C++, structures and unions can be used just like they are used in C. However, there is an interesting side to C++ structures that we will study in Chapter 4.

Table 3.3 *Difference between structures and unions*

Structure	Union
A structure is defined with 'struct' keyword.	A union is defined with 'union' keyword.
All members of a structure can be manipulated simultaneously.	The members of a union can be manipulated only one at a time.
The size of a structure object is equal to the sum of the individual sizes of the member objects.	The size of a union object is equal to the size of largest member object.
Structure members are allocated distinct memory locations.	Union members share common memory space for their exclusive usage.
Structures are not considered as memory efficient in comparison to unions.	Unions are considered as memory efficient particularly in situations when the members are not required to be accessed simultaneously.
Structures in C++ behave just like a class. Almost everything that can be achieved with a class can also be done with structures.	Unions retain their core functionality in C++ with slight add-ons like declaration of anonymous unions.

3.6.2 Classes

C++ also permits us to define another user-defined data type known as **class** which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming. Other details about classes are discussed in Chapter 4.

3.6.3 Enumerated Data Type

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

```
enum shape{circle, square, triangle};
enum colour{red, blue, green, yellow};
enum position{off, on};
```

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

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

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

```
colour background = blue;           // allowed
colour background = 7;              // Error in C++
colour background = (colour) 7;     // OK
```

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

```
int c = red;           // valid, colour type promoted to int
```

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

```
enum colour{red, blue=4, green=8};
enum colour{red=5, blue, green};
```

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

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

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

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

```
int switch_1 = off;
int switch_2 = on;
```

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

```
enum shape
{
    circle,
    rectangle,
    triangle
};
int main()
{
    cout << "Enter shape code:";
    int code;
    cin >> code;
```

```

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

return 0;
}

```

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



3.7 STORAGE CLASSES

In addition to the data type, a variable, also has a storage class associated with it. The storage class of a variable specifies the lifetime and visibility of a variable within the program. Lifetime refers to the longevity of a variable or the duration till which a variable remains active during program execution. Visibility, on the other hand, signifies the scope of a variable, i.e., in which program modules the variable is accessible. There are four types of storage classes, as explained below:

Automatic It is the default storage class of any type of variable. Its visibility is restricted to the function in which it is declared. Further, its lifetime is also limited till the time its container function is executing. That is, it is created as soon as its declaration statement is encountered and is destroyed as soon the program control leaves its container function block.

External As the name suggests, an external variable is declared outside of a function but is accessible inside the function block. Also called global variable, its visibility is spread all across the program that means, it is accessible by all the functions present in the program. The lifetime of an external variable is same as the lifetime of a program.

Static A static variable has the visibility of a local variable but the lifetime of an external variable. That means, once declared inside a function block, it does not get destroyed after the function is executed, but retains its value so that it can be used by future function calls.

Register Similar in behavior to an automatic variable, a register variable differs in the manner in which it is stored in the memory. Unlike, automatic variables that are stored in the primary memory, the register variables are stored in CPU registers. The objective of storing a variable in registers is to increase its access speed, which eventually makes the program run faster. However, it may be noted that it is not an obligation for a compiler to store a register type variable only in CPU registers, but if there are no registers vacant to accommodate the variable then it is stored just like any other automatic variable.

Table 3.4 gives a summary of the four storage classes:

Table 3.4 Storage classes

	<i>Automatic</i>	<i>External</i>	<i>Static</i>	<i>Register</i>
Lifetime	Function block	Entire program	Entire program	Function block
Visibility	Local	Global	Local	Local
Initial Value	Garbage	0	0	Garbage
Storage	Stack segment	Data segment	Data segment	CPU Registers
Purpose	Local variables used by a single function	Global variables used throughout the program	Local variables retaining their values throughout the program	Variables using CPU registers for storage purpose
Keyword	auto	extern	static	Register



3.8 DERIVED DATA TYPES

3.8.1 Arrays

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

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

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

```
char string[4] = "xyz"; // O.K. for C++
```

3.8.2 Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these modifications and improvements were driven by the requirements of the object-oriented concept of C++. Some of these were introduced to make the C++ program more reliable and readable. All the features of C++ functions are discussed in Chapter 5.

3.8.3 Pointers

Pointers are declared and initialized as in C. Examples:

```
int *ip;    // int pointer
ip = &x;    // address of x assigned to ip
*ip = 10;   // 10 assigned to x through indirection
```

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

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

We cannot modify the address that **ptr1** is initialized to.

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

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

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

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

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

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



3.9 SYMBOLIC CONSTANTS

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

- Using the qualifier **const**, and
- Defining a set of integer constants using **enum** keyword.

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

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

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

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

```
const size = 10;
```

means

```
const int size = 10;
```

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

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

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

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

Another method of naming integer constants is by enumeration as under;

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

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

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

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

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

Such values can be any integer values. Enumerated data type has been discussed in detail in Section 3.6.



3.10 TYPE COMPATIBILITY

C++ is very strict with regard to type compatibility as compared to C. For instance, C++ defines **int**, **short int**, and **long int** as three different types. They must be cast when their values are assigned to one another. Similarly, **unsigned char**, **char**, and **signed char** are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility, or else, a cast must be applied. These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

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

```
sizeof ('x')
```

is equivalent to

```
sizeof(int)
```

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

```
sizeof ('x')
```

equals

```
sizeof(char)
```



3.11 DECLARATION OF VARIABLES

We know that, in C, all variables must be declared before they are used in executable statements. This is true with C++ as well. However, there is a significant difference between C and C++ with regard to the place of their declaration in the program. C requires all the variables to be defined at the beginning of a scope. When we read a C program, we usually come across a group of variable declarations at the beginning of each scope level. Their actual use appears elsewhere in the scope, sometimes far away from the place of declaration. Before using a variable, we should go back to the beginning of the program to see whether it has been declared and, if so, of what type.

C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use.

The example below illustrates this point.

```
int main()
{
    float x;                // declaration
    float sum = 0;

    for(int i=1; i<5; i++) // declaration
    {
        cin >> x;
        sum = sum +x;
    }
    float average;          // declaration
    average = sum/(i-1);
    cout << average;
    return 0;
}
```

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



3.12 DYNAMIC INITIALIZATION OF VARIABLES

In C, a variable must be initialized using a constant expression, and the C compiler would fix the initialization code at the time of compilation. C++, however, permits initialization of the variables at run time. This is referred to as *dynamic initialization*. In C++, a variable can be initialized at run time using expressions at the place of declaration. For example, the following are valid initialization statements:

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

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

```
float average; // declare where it is necessary
average = sum/i;
```

can be combined into a single statement:

```
float average = sum/i; // initialize dynamically at run time
```

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



3.13 REFERENCE VARIABLES

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

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

Example:

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

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

```
cout << total;
```

and

```
cout << sum;
```

both print the value 100. The statement

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

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

```
sum = 0;
```

will change the value of both the variables to zero.

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

C++ assigns additional meaning to the symbol **&**. Here, **&** is not an address operator. The notation **float &** means reference to **float**. Other examples are:

```
int n[10];
int & x = n[10];           // x is alias for n[10]
char & a = '\n';           // initialize reference to a literal
```


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

The following references are also allowed:

```
i.  int x;
    int *p = &x;
    int & m = *p;
ii. int & n = 50;
```

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

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

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

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

```
int & x = m;
```

Thus, **x** becomes an alias of **m** after executing the statement

```
f(m);
```

Such function calls are known as *call by reference*. This implementation is illustrated in Fig. 3.2. Since the variables **x** and **m** are aliases, when the function increments **x**, **m** is also incremented. The value of **m** becomes 20 after the function is executed. In traditional C, we accomplish this operation using pointers and dereferencing techniques.

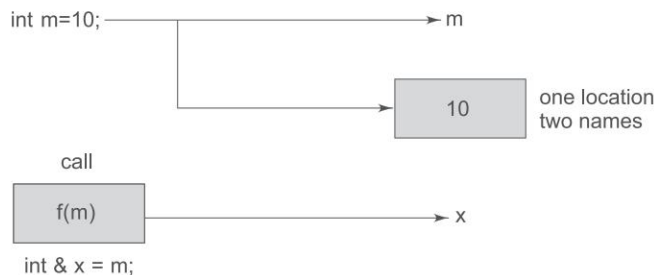


Fig. 3.2 Call by reference mechanism

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



3.14 OPERATORS IN C++

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. We have already seen two such operators, namely, the insertion operator `<<`, and the extraction operator `>>`. Other new operators are:

<code>::</code>	Scope resolution operator
<code>::*</code>	Pointer-to-member declarator
<code>->*</code>	Pointer-to-member operator
<code>.*</code>	Pointer-to-member operator
<code>delete</code>	Memory release operator
<code>endl</code>	Line feed operator
<code>new</code>	Memory allocation operator
<code>setw</code>	Field width operator

In addition, C++ also allows us to provide new definitions to some of the built-in operators. That is, we can give several meanings to an operator, depending upon the types of arguments used. This process is known as *operator overloading*.



3.15 SCOPE RESOLUTION OPERATOR

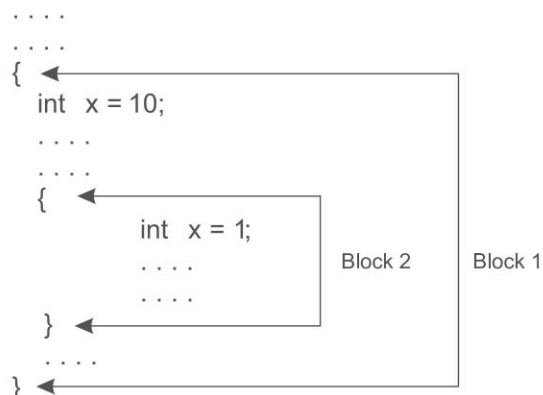
Like C, C++ is also a block-structured language. Blocks and scopes can be used in constructing programs. We know that the same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to be local to that block. Consider the following segment of a program

```

.....
.....
{
    int x = 10;
    .....
    .....
}
.....
.....
{
    int x = 1;
    .....
    .....
}

```

The two declarations of `x` refer to two different memory locations containing different values. Statements in the second block cannot refer to the variable `x` declared in the first block, and vice versa. Blocks in C++ are often nested. For example, the following style is common:



Block2 is contained in block1. Note that a declaration in an inner block *hides* a declaration of the same variable in an outer block and, therefore, each declaration of `x` causes it to refer to a different data object. Within the inner block, the variable `x` will refer to the data object declared therein.

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator `::` called the *scope resolution operator*. This can be used to uncover a hidden variable. It takes the following form:

```
:: variable-name
```

This operator allows access to the global version of a variable. For example, `::count` means the global version of the variable `count` (and not the local variable `count` declared in that block). Program 3.1 illustrates this feature.

Program 3.1 Scope Resolution Operator

```

#include <iostream>

using namespace std;

int m = 10;           // global m

int main()
{
    int m = 20;       // m redeclared, local to main

    {
        int k = m;
        int m = 30;   // m declared again
                      // local to inner block
    }
}

```

(Contd.)

```

    cout << "we are in inner block \n";
    cout << "k = " << k << "\n";
    cout << "m = " << m << "\n";
        cout << "::m = " << ::m << "\n";

    }

    cout << "\nWe are in outer block \n";
    cout << "m = " << m << "\n";
    cout << "::m = " << ::m << "\n";

    return 0;
}

```

The output of Program 3.1 would be

```

We are in inner block
k = 20
m = 30
::m = 10

We are in outer block
m = 20
::m = 10

```

In the above program, the variable **m** is declared at three places, namely, outside the **main()** function, inside the **main()**, and inside the inner block.



Note It is to be noted **::m** will always refer to the global **m**. In the inner block, **::m** refers to the value 10 and not 20.

A major application of the scope resolution operator is in the classes to identify the class to which a member function belongs. This will be dealt in detail later when the classes are introduced.



3.16 MEMBER DEREFERENCING OPERATORS

As you know, C++ permits us to define a class containing various types of data and functions as members. C++ also permits us to access the class members through pointers. In order to achieve this, C++ provides a set of three pointer-to-member operators. Table 3.5 shows these operators and their functions.

Table 3.5 Member dereferencing operators

Operator	Function
::*	To declare a pointer to a member of a class
*	To access a member using object name and a pointer to that member
->*	To access a member using a pointer to the object and a pointer to that member

Further details on these operators will be meaningful only after we discuss classes, and therefore we defer the use of member dereferencing operators until then.



3.17 MEMORY MANAGEMENT OPERATORS

C uses **malloc()** and **calloc()** functions to allocate memory dynamically at run time. Similarly, it uses the function **free()** to free dynamically allocated memory. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as *free store* operators.

An object can be created by using **new**, and destroyed by using **delete**, as and when required. A data object created inside a block with **new**, will remain in existence until it is explicitly destroyed by using **delete**. Thus, the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The **new** operator can be used to create objects of any type. It takes the following general form:

```
pointer-variable = new data-type;
```

Here, *pointer-variable* is a pointer of type *data-type*. The **new** operator allocates sufficient memory to hold a data object of type *data-type* and returns the address of the object. The *data-type* may be any valid data type. The *pointer-variable* holds the address of the memory space allocated.

Examples:

```
p = new int;
q = new float;
```

where **p** is a pointer of type **int** and **q** is a pointer of type **float**. Here, **p** and **q** must have already been declared as pointers of appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

```
int *p = new int;
float *q = new float;
```

Subsequently, the statements

```
*p = 25;
*q = 7.5;
```

assign 25 to the newly created **int** object and 7.5 to the **float** object.

We can also initialize the memory using the **new** operator. This is done as follows:

```
pointer-variable = new data-type(value);
```

Here, *value* specifies the initial value. Examples:

```
int *p = new int(25);
float *q = new float(7.5);
```

As mentioned earlier, **new** can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes. The general form for a one-dimensional array is

```
pointer-variable = new data-type[size];
```

Here, size specifies the number of elements in the array. For example, the statement

```
int *p = new int[10];
```

creates a memory space for an array of 10 integers. **p[0]** will refer to the first element, **p[1]** to the second element, and so on.

When creating multi-dimensional arrays with **new**, all the array sizes must be supplied.

```
array_ptr = new int[3][5][4]; // legal
array_ptr = new int[m][5][4]; // legal
array_ptr = new int[3][5][ ]; // illegal
array_ptr = new int[ ][5][4]; // illegal
```

The first dimension may be a variable whose value is supplied at runtime. All others must be constants.

Similar to the basic data type objects, the **new** operator can also be used for dynamically creating class objects. The size of the class object is automatically determined and the memory space is allocated accordingly. The general form of allocating class objects using **new** operator is

```
class sample
{
    private:
        data-type d1;
        data-type d2;
        .
        .
    public:
        .
        .
};
main()
{
    .
    .
    sample *ptr = new sample;
    .
}
```

Here, the size of the sample class object will be automatically determined and the corresponding memory space reserved by the **new** operator.

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is

```
delete pointer-variable;
```

The *pointer-variable* is the pointer that points to a data object created with **new**. Examples:

```
delete p;  
delete q;
```

If we want to free a dynamically allocated array, we must use the following form of **delete**:

```
delete [size] pointer-variable;
```

The *size* specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent versions of C++ do not require the size to be specified. For example,

```
delete [] p;
```

will delete the entire array pointed to by **p**.

Program 3.2 demonstrates the use of new and delete operators for dynamic allocation and deallocation of memory.

Program 3.2 Use of new and delete Operators

```
#include <iostream>  
#include<conio.h>  
using namespace std;  
  
void main()  
{  
    int *arr;  
    int size;  
  
    cout<<"Enter the size of the integer array: ";  
    cin>>size;  
  
    cout<<"Creating an array of size "<<size<<"..";  
    arr = new int[size];  
  
    cout<<"\nDynamic allocation of memory for array arr is  
    successful.";  
  
    delete arr;  
    getch();  
}
```

The output of Program 3.2 would be

```
Enter the size of the integer array: 5
Creating an array of size 5..
Dynamic allocation of memory for array arr is successful.
```

What happens if sufficient memory is not available for allocation? In such cases, like **malloc()**, **new** returns a null pointer. Therefore, it may be a good idea to check for the pointer produced by **new** before using it. It is done as follows:

```
.....
.....
p = new int;
if(!p)
{
    cout << "allocation failed \n";
}
.....
.....
```

If you are using Standard C++, then instead of null pointer the **bad_alloc** exception will be thrown. It is a system generated erroneous condition indicating memory allocation failure. To handle this situation, adequate exception handling code must be included in the program.

Program 3.3 illustrates the use of **new** and **delete** operators for dynamic allocation of class objects. The program uses **bad_alloc** exception instead of null pointer for handling allocation failure.

Program 3.3 Use of bad_alloc Exception

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

class sample
{
    private:
        int data1;
        char data2;
    public:
        void set (int i,char c)
        {
            data1=i;
            data2=c;
        }

        void display(void)
        {
            cout<<"Data1 = "<<data1;
            cout<<"\nData2 = "<<data2;
        }
}
```

(Contd.)


```

};
int main()
{
    sample *s_ptr;
    try
    {
        s_ptr = new sample;
    }

    catch (bad_alloc ba)
    {
        cout<<"Bad Allocation occurred...the program will terminate
        now..";
        return 1;
    }

    s_ptr->set(25, 'A');
    s_ptr->display();

    delete s_ptr;
    getch();
}

```

The **new** operator offers the following advantages over the function **malloc()**.

1. It automatically computes the size of the data object. We need not use the operator **sizeof**.
2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, **new** and **delete** can be overloaded.



3.18 MANIPULATORS

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw**.

The **endl** manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character `"\n"`. For example, the statement

```

.....
.....
cout << "m = " << m << endl
    << "n = " << n << endl
    << "p = " << p << endl;
.....
.....

```

would cause three lines of output, one for each variable. If we assume the values of the variables as 2597, 14, and 175 respectively, the output will appear as follows:

m =

2	5	9	7

```

n = 

|   |   |
|---|---|
| 1 | 4 |
|---|---|


p = 

|   |   |   |
|---|---|---|
| 1 | 7 | 5 |
|---|---|---|


```

It is important to note that this form is not the ideal output. It should rather appear as under:

```

m = 2597
n =   14
p =  175

```

Here, the numbers are *right-justified*. This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

```
cout << setw(5) << sum << endl;
```

The manipulator **setw(5)** specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below:

		3	4	5

Program 3.4 illustrates the use of **endl** and **setw**.

Program 3.4 Use of Manipulators

```

#include <iostream>
#include <iomanip> // for setw

using namespace std;

int main()
{
    int Basic = 950, Allowance = 95, Total = 1045;

    cout << setw(10) << "Basic" << setw(10) << Basic << endl
         << setw(10) << "Allowance" << setw(10) << Allowance << endl
         << setw(10) << "Total" << setw(10) << Total << endl;
    return 0;
}

```

The output of Program 3.4 would be

```

Basic          950
Allowance      95
Total         1045

```



Note Character strings are also printed right-justified.

We may also control the precision of floating point numbers appearing in the output by using the `fixed` and `setprecision()` manipulators, as shown below:

```
cout<<fixed<<setprecision(2)<<average;
```

The above statement will display the value of `average` variable on the screen restricted to a precision of two.

We can also write our own manipulators as follows:

```
#include <iostream>
ostream & symbol(ostream & output)
{
    return output << "\\tRs";
}
```

The **symbol** is the new manipulator which represents **Rs**. The identifier **symbol** can be used whenever we need to display the string **Rs**.



3.19 TYPE CAST OPERATOR

C++ permits explicit type conversion of variables or expressions using the type cast operator.

Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternative. The following two versions are equivalent:

```
(type-name) expression // C notation
type-name (expression) // C++ notation
```

Examples:

```
average = sum/(float)i; // C notation
average = sum/float(i); // C++ notation
```

A type-name behaves as if it is a function for converting values to a designated type. The function-call notation usually leads to simplest expressions. However, it can be used only if the type is an identifier. For example,

```
p = int * (q);
```

is illegal. In such cases, we must use C type notation.

```
p = (int *) q;
```

Alternatively, we can use **typedef** to create an identifier of the required type and use it in the functional notation.

```
typedef int * int_pt;
p = int_pt(q);
```

Program 3.5 demonstrates explicit type casting in C++:

Program 3.5 Explicit Type Casting

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

int main()
{
    int intvar=25;
    float floatvar=35.87;

    cout<<"intvar = "<<intvar;
    cout<<"\nfloatvar = "<<floatvar;
    cout<<"\nfloat(intvar) = "<<float(intvar);
    cout<<"\nint(floatvar) = "<<int(floatvar);

    getch();
}
```

The output of Program 3.5 would be

```
intvar = 25
floatvar = 35.87
float(intvar) = 25
int(floatvar) = 35
```

ANSI C++ adds the following new cast operators:

- `const_cast`
- `static_cast`
- `dynamic_cast`
- `reinterpret_cast`



3.20 EXPRESSIONS AND THEIR TYPES

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands, and zero or more operators to produce a value. Expressions may be of the following seven types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as *compound expressions*.

3.20.1 Constant Expressions

Constant expressions consist of only constant values. Examples:

```
15
20 + 5 / 2.0
'x'
```

3.20.2 Integral Expressions

Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

```
m
m * n - 5
m * 'x'
5 + int(2.0)
```

where **m** and **n** are integer variables.

3.20.3 Float Expressions

Float expressions are those which, after all conversions, produce floating-point results. Examples:

```
x + y
x * y / 10
5 + float(10)
10.75
```

where **x** and **y** are floating-point variables.

3.20.4 Pointer Expressions

Pointer expressions produce address values. Examples:

```
&m
ptr
ptr + 1
"xyz"
```

where **m** is a variable and **ptr** is a pointer.

3.20.5 Relational Expressions

Relational expressions yield results of type **bool** which takes a value **true** or **false**. Examples:

```
x <= y
a+b == c+d
m+n > 100
```

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as *Boolean expressions*.

3.20.6 Logical Expressions

Logical expressions combine two or more relational expressions and produces **bool** type results. Examples:

```
a > b && x == 10
x == 10 || y == 5
```

3.20.7 Bitwise Expressions

Bitwise expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. Examples:

```
x << 3 // Shift three bit position to left
y >> 1 // Shift one bit position to right
```

Shift operators are often used for multiplication and division by powers of two.

ANSI C++ has introduced what are termed as *operator keywords* that can be used as alternative representation for operator symbols.



3.21 SPECIAL ASSIGNMENT EXPRESSIONS

3.21.1 Chained Assignment

```
x = (y = 10);
or
x = y = 10;
```

First 10 is assigned to y and then to x.

A chained statement cannot be used to initialize variables at the time of declaration. For instance, the statement

```
float a = b = 12.34; // wrong
```

is illegal. This may be written as

```
float a=12.34, b=12.34 // correct
```

3.21.2 Embedded Assignment

```
x = (y = 50) + 10;
```

(y = 50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50+10 = 60 is assigned to x. This statement is identical to

```
y = 50;
x = y + 10;
```

3.21.3 Compound Assignment

Like C, C++ supports a *compound assignment operator* which is a combination of the assignment operator with a binary arithmetic operator. For example, the simple assignment statement

```
x = x + 10;
```

may be written as

```
x += 10;
```

The operator += is known as *compound assignment operator* or *short-hand assignment operator*. The general form of the compound assignment operator is

```
variable1 op= variable2;
```

where *op* is a binary arithmetic operator. This means that

```
variable1 = variable1 op variable2;
```



3.22 IMPLICIT CONVERSIONS

We can mix data types in expressions. For example,

```
m = 5+2.75;
```

is a valid statement. Wherever data types are mixed in an expression, C++ performs the conversions automatically. This process is known as *implicit* or *automatic conversion*.

When the compiler encounters an expression, it divides the expressions into sub-expressions consisting of one operator and one or two operands. For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the “smaller” type is converted to the “wider” type. For example, if one of the operand is an **int** and the other is a **float**, the **int** is converted into a **float** because a **float** is wider than an **int**. The “waterfall” model shown in Fig. 3.3 illustrates this rule.

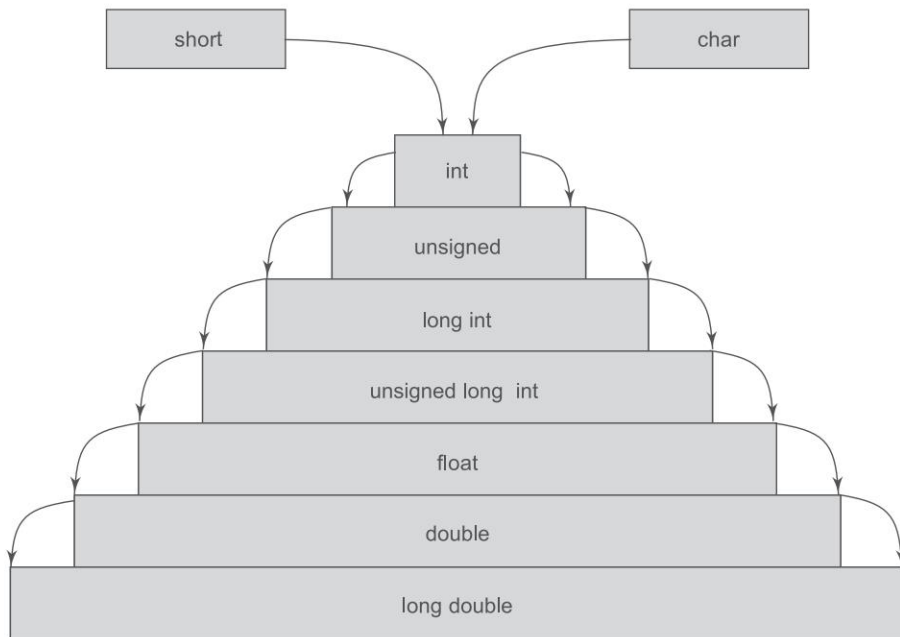


Fig. 3.3 Waterfall model of type conversion

Whenever a **char** or **short int** appears in an expression, it is converted to an **int**. This is called *integral widening conversion*. The implicit conversion is applied only after completing all integral widening conversions.

Table 3.6 Results of mixed-mode operations

LHO \ RHO							
	char	short	int	long	float	double	long double
char	int	int	int	long	float	double	long double
short	int	int	int	long	float	double	long double
int	int	int	int	long	float	double	long double
long	long	long	long	long	float	double	long double
float	float	float	float	float	float	double	long double
double	double	double	double	double	double	double	long double
long double	long	long	long	long	long	long	long double
	double	double	double	double	double	double	

RHO – Right-hand operand

LHO – Left-hand operand



3.23 OPERATOR OVERLOADING

As stated earlier, overloading means assigning different meanings to an operation, depending on the context. C++ permits overloading of operators, thus allowing us to assign multiple meanings to operators. Actually, we have used the concept of overloading in C also. For example, the operator ***** when applied to a pointer variable, gives the value pointed to by the pointer. But it is also commonly used for multiplying two numbers. The number and type of operands decide the nature of operation to follow.

The input/output operators **<<** and **>>** are good examples of operator overloading. Although the built-in definition of the **<<** operator is for shifting of bits, it is also used for displaying the values of various data types. This has been made possible by the header file *iostream* where a number of overloading definitions for **<<** are included. Thus, the statement

```
cout << 75.86;
```

invokes the definition for displaying a **double** type value, and

```
cout << "well done";
```

invokes the definition for displaying a **char** value. However, none of these definitions in *iostream* affect the built-in meaning of the operator.

Similarly, we can define additional meanings to other C++ operators. For example, we can define **+** operator to add two structures or objects. Almost all C++ operators can be overloaded with a few exceptions such as the member-access operators (**.** and **.***), conditional operator (**?:**), scope resolution operator (**::**) and the size operator (**sizeof**). Definitions for operator overloading are discussed in detail in Chapter 7.



3.24 OPERATOR PRECEDENCE

Although C++ enables us to add multiple meanings to the operators, yet their association and precedence remain the same. For example, the multiplication operator will continue having higher precedence than the add operator. Table 3.7 gives the precedence and associativity of all the C++ operators. The groups are listed in the order of decreasing precedence. The labels *prefix* and *postfix* distinguish the uses of ++ and --. Also, the symbols +, -, *, and & are used as both unary and binary operators.

Table 3.7 Operator precedence and associativity

Operator	Associativity
::	left to right
-> . () [] postfix ++ postfix --	left to right
prefix ++ prefix -- ~ ! unary + unary - unary *	
unary & (type) sizeof new delete	right to left
-> **	left to right
* / %	left to right
+ -	left to right
<< >>	left to right
<< = >> =	left to right
= = !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
?:	left to right
= * = / = % = + = - =	right to left
<< = >> = & = ^ = =	left to right
, (comma)	

Note: The unary operations assume higher precedence.



3.25 CONTROL STRUCTURES

In C++, a large number of functions are used that pass messages, and process the data contained in objects. A function is set up to perform a task. When the task is complex, many different algorithms can be designed to achieve the same goal. Some are simple to comprehend, while others are not. Experience has also shown that the number of bugs that occur is related to the format of the program. The format

should be such that it is easy to trace the flow of execution of statements. This would help not only in debugging but also in the review and maintenance of the program later. One method of achieving the objective of an accurate, error-resistant and maintainable code is to use one or any combination of the following three control structures:

1. Sequence structure (straight line)
2. Selection structure (branching)
3. Loop structure (iteration or repetition)

Figure 3.4 shows how these structures are implemented using *one-entry, one-exit* concept, a popular approach used in modular programming.

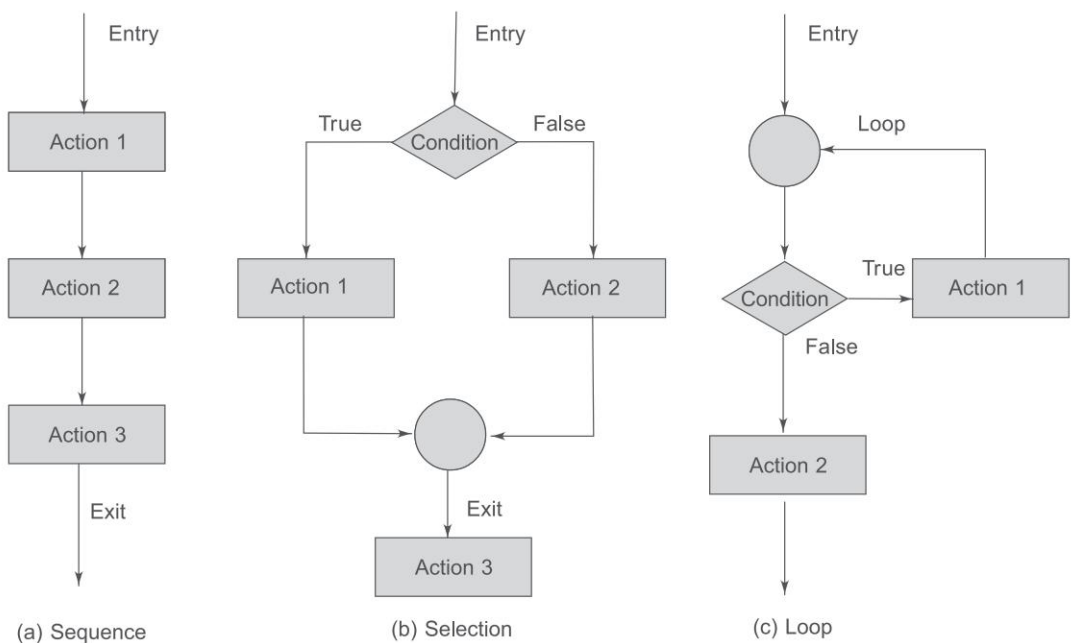
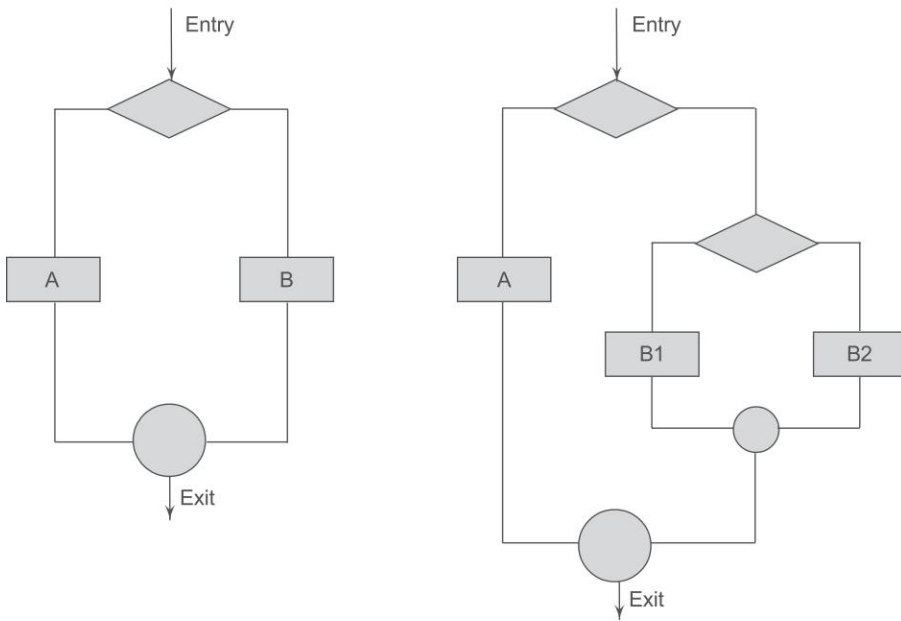


Fig. 3.4 Basic control structures

It is important to understand that all program processing can be coded by using only these three logic structures. The approach of using one or more of these basic control constructs in programming is known as *structured programming*, an important technique in software engineering.

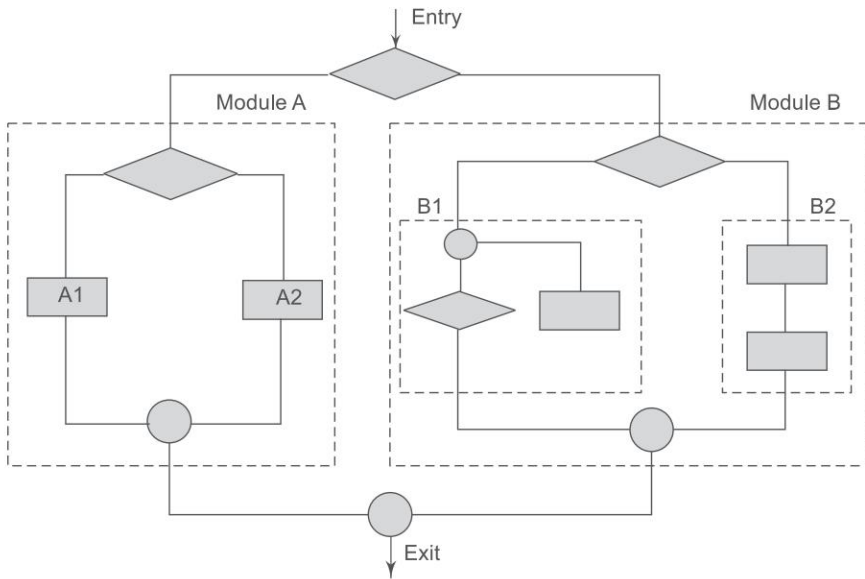
Using these three basic constructs, we may represent a function structure either in detail or in summary form as shown in Figs. 3.5 (a), (b) and (c).

Like C, C++ also supports all the three basic control structures, and implements them using various control statements as shown in Fig. 3.6. This shows that C++ combines the power of structured programming with the object-oriented paradigm.



(a) First level of abstraction

(b) Second level of abstraction



(c) Detailed flow chart

Fig. 3.5 Different levels of abstraction

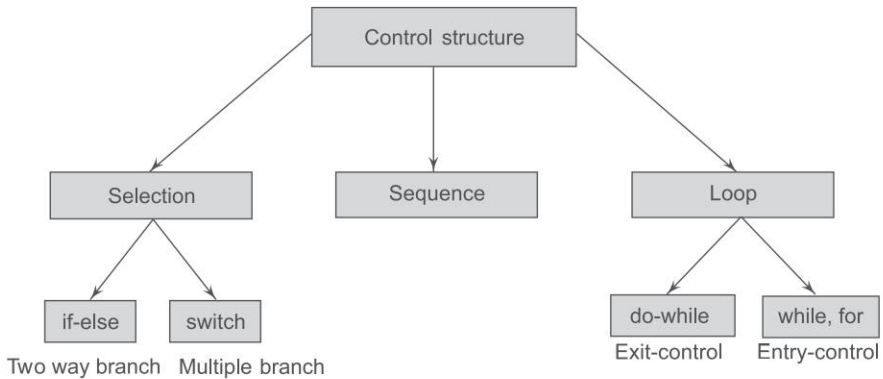


Fig. 3.6 C++ statements to implement in two forms

3.25.1 The if statement

The **if** statement is implemented in two forms:

- Simple **if** statement
- **if...else** statement

Examples:

Form 1

```

if(expression is true)
{
    action1;
}
action2;
action3;
  
```

Form 2

```

if(expression is true)
{
    action1;
}
else
{
    action2;
}
action3;
  
```

3.25.2 The switch statement

This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points. This is implemented as follows:

```

switch(expression)
{
    case1:
  
```

```
    {
    action1;
    }
    case2:
    {
    action2;
    }
    case3:
    {
    action3;
    }
    default:
    {
    action4;
    }
}
action5;
```

3.25.3 The do-while statement

The **do-while** is an *exit-controlled* loop. Based on a condition, the control is transferred back to a particular point in the program. The syntax is as follows:

```
do
{
    action1;
}
while(condition is true);
action2;
```

3.25.4 The while statement

This is also a loop structure, but is an *entry-controlled* one. The syntax is as follows:

```
while(condition is true)
{
    action1;
}
action2;
```

3.25.4 The for statement

The **for** is an *entry-entrolled* loop and is used when an action is to be repeated for a predetermined number of times. The syntax is as follows:

```
for(initial value; test; increment)
{
    action1;
}
action2;
```

The syntax of the control statements in C++ is very much similar to that of C and therefore they are implemented as and when they are required.

SUMMARY

- ☐ C++ provides various types of tokens that include keywords, identifiers, constants, strings, and operators.
- ☐ Identifiers refer to the names of variables, functions, arrays, classes, etc.
- ☐ C++ provides an additional use of **void**, for declaration of generic pointers.
- ☐ The enumerated data types differ slightly in C++. The tag names of the enumerated data types become new type names. That is, we can declare new variables using these tag names.
- ☐ In C++, the size of character array should be one larger than the number of characters in the string.
- ☐ C++ adds the concept of constant pointer and pointer to constant. In case of constant pointer, we can not modify the address that the pointer is initialized to. In case of pointer to a constant, contents of what it points to cannot be changed.
- ☐ Pointers are widely used in C++ for memory management and to achieve polymorphism.
- ☐ C++ provides a qualifier called **const** to declare named constants which are just like variables except that their values can not be changed. A **const** modifier defaults to an **int**.
- ☐ C++ is very strict regarding type checking of variables. It does not allow to equate variables of two different data types. The only way to break this rule is type casting.
- ☐ C++ allows us to declare a variable anywhere in the program, as also its initialization at run time, using the expressions at the place of declaration.
- ☐ A reference variable provides an alternative name for a previously defined variable. Both the variables refer to the same data object in the memory. Hence, change in the value of one will also be reflected in the value of the other variable.
- ☐ A reference variable must be initialized at the time of declaration, which establishes the correspondence between the reference and the data object that it names.
- ☐ A major application of the scope resolution (::) operator is in the classes to identify the class to which a member function belongs.
- ☐ In addition to **malloc()**, **calloc()** and **free()** functions, C++ also provides two unary operators, **new** and **delete** to perform the task of allocating and freeing the memory in a better and easier way.
- ☐ C++ also provides manipulators to format the data display. The most commonly used manipulators are **endl** and **setw**.
- ☐ C++ supports seven types of expressions. When data types are mixed in an expression, C++ performs the conversion automatically using certain rules.
- ☐ C++ also permits explicit type conversion of variables and expressions using the type cast operators.
- ☐ Like C, C++ also supports the three basic control structures namely, sequence, selection and loop, and implements them using various control statements such as, **if**, **if...else**, **switch**, **.do..while**, **while** and **for**.

Key Terms

array | associativity | automatic conversion | backslash character | bitwise expression | **bool** | boolean expression | branching | call by reference | **calloc()** | character constant | chained assignment | **class** | compound assignment | compound expression | **const** | constant | constant expression | control structure | data types | decimal integer | declaration | **delete** | dereferencing | derived-type | **do...while** | embedded assignment | **endl** | entry control | enumeration | exit control | explicit conversion | expression | float expression | floating point integers | **for** | formatting | free store | **free()** | function | hexadecimal integer | identifier | **if** | **if...else** | implicit conversion | initialization | integer constant | integral expression | integral widening | iteration | keyword | literal | logical expression | loop | loop structure | **malloc()** | manipulator | memory | named constant | **new** | octal integer | operator | operator keywords | operator overloading | operator precedence | pointer | pointer expression | pointer variable | reference | reference variable | relational expression | repetition | scope resolution | selection | selection structure | sequence | sequence structure | **setw** | short-hand assignment | **sizeof()** | straight line | **string** | string constant | **struct** | structure | structured programming | switch | symbolic constant | token | type casting | type compatibility | **typedef** | **union** | user-defined type | variable | **void** | water-fall model | **wchar_t** | **while** | wide-character

Review Questions

- 3.1 Enumerate the rules of naming variables in C++. How do they differ from ANSI C rules?
- 3.2 An **unsigned int** can be twice as large as the **signed int**. Explain how?
- 3.3 Why does C++ have type modifiers?
- 3.4 What are the applications of **void** data type in C++?
- 3.5 Can we assign a **void** pointer to an **int** type pointer? If not, why? How can we achieve this?
- 3.6 Describe, with examples, the uses of enumeration data types.
- 3.7 Describe the differences in the implementation of **enum** data type in ANSI C and C++.
- 3.8 Why is an array called a derived data type?
- 3.9 The size of a **char** array that is declared to store a string should be one larger than the number of characters in the string. Why?
- 3.10 The **const** was taken from C++ and incorporated in ANSI C, although quite differently. Explain.
- 3.11 How does a constant defined by **const** differ from the constant defined by the preprocessor statement **#define**?
- 3.12 In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?
- 3.13 What do you mean by dynamic initialization of a variable? Give an example.
- 3.14 What is a reference variable? What is its major use?
- 3.15 List at least four new operators added by C++ which aid OOP.
- 3.16 What is the application of the scope resolution operator **::** in C++?
- 3.17 What are the advantages of using **new** operator as compared to the function **malloc()**?
- 3.18 Which manipulator is used to control the precision of floating point numbers? Support your answer with the help of an example.

- 3.19** Illustrate with an example, how the **setw** manipulator works.
- 3.20** How do the following statements differ?
- (a) `char * const p;`
 - (b) `char const *p;`

Debugging Exercises

- 3.1** What will happen when you execute the following code?

```
#include <iostream.h>
void main()
{
    int i=0;
    i=400*400/400;
    cout << i;
}
```

- 3.2** Identify the error in the following program.

```
#include <iostream.h>
void main()
{
    int num[]={1,2,3,4,5,6};
    num[1]==[1]num ? cout<<"Success" : cout<<"Error";
}
```

- 3.3** Identify the errors in the following program.

```
#include <iostream.h>
void main()
{
    int i=5;
    while(i)
    {
        switch(i)
        {
            default:
            case 4:
            case 5:

                break;

            case 1:
                continue;

                case 2:
                case 3:
                    break;

        }
        i--;
    }
}
```


3.4 Identify the error in the following program.

```
#include <iostream.h>
#define pi 3.14

int squareArea(int &);
int circleArea(int &);

void main()
{
    int a=10;
    cout << squareArea(a) << " ";
    cout << circleArea(a) << " ";
    cout << a << endl;
}

int squareArea(int &a)
{
    return a *== a;
}

int circleArea(int &r)
{
    return r = pi * r * r;
}
```

3.5 Identify the error in the following program.

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

char* allocateMemory();

void main()
{
    char* str;
    str = allocateMemory();
    cout << str;
    delete str;
    str = "    ";
    cout << str;
}

char* allocateMemory()
{
    str = "Memory allocation test, ";
    return str;
}
```

3.6 Find errors, if any, in the following C++ statements.

- (a) long float x;
- (b) char *cp = vp; // vp is a void pointer
- (c) int code = three; // three is an enumerator
- (d) int *p = new; // allocate memory with new

- (e) enum (green, yellow, red);
- (f) int const *p = total;
- (g) const int array_size;
- (h) for (i=1; int i<10; i++) cout << i << "\n";
- (i) int & number = 100;
- (j) float *p = new int [10];
- (k) int public = 1000;
- (l) char name[3] = "USA";

Programming Exercises

3.1 Write a function using reference variables as arguments to swap the values of a pair of integers. W E B

3.2 Write a function that creates a vector of user-given size **M** using **new** operator.

3.3 Write a program to print the following output using **for** loops.

```

1
22
333
4444
55555
..... W E B

```

3.4 Write a program to evaluate the following investment equation

$$V = P(1 + r)^n$$

and print the tables which would give the value of V for various combination of the following values of P , r and n :

P : 1000, 2000, 3000, ..., 10,000

r : 0.10, 0.11, 0.12, ..., 0.20

n : 1, 2, 3, ..., 10

Hint: P is the principal amount and V is the value of money at the end of n years. This equation can be recursively written as

$$V = P(1 + r)$$

$$P = V$$

In other words, the value of money at the end of the first year becomes the principal amount for the next year, and so on. W E B

3.5 An election is contested by five candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and **count** the votes cast for each candidate using an array variable count. In case, a number read is outside the range 1 to 5, the ballot should be considered as a 'spoilt ballot', and the program should also count the number of spoilt ballots. W E B

3.6 A cricket team has the following table of batting figures for a series of test matches:

Player's name	Runs	Innings	Times not out
Sachin	8430	230	18
Saurav	4200	130	9
Rahul	3350	105	11
.	.	.	.
.	.	.	.

Write a program to read the figures set out in the above form, to calculate the batting averages and to print out the complete table including the averages.

3.7 Write programs to evaluate the following functions to 0.0001% accuracy.

$$(a) \sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad (b) \text{SUM} = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \dots$$

$$(c) \cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

3.8 Write a program to print a table of values of the function $y = e^{-x}$ for x varying from 0 to 10 in steps of 0.1. The table should appear as follows.

Table for $Y = \text{EXP} [-X]$

X	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.0									
1.0									
.									
.									
9.0									

3.9 Write a program to calculate the variance and standard deviation of N numbers.

$$\text{Variance} = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2$$

$$\text{Standard Deviation} = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2}$$

$$\text{where } \bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$$

3.10 An electricity board charges the following rates to domestic users to discourage large consumption of energy:

For the first 100 units – 60P per unit
 For next 200 units – 80P per unit
 Beyond 300 units – 90P per unit

All users are charged a minimum of Rs. 50.00. If the total amount is more than Rs. 300.00 then an additional surcharge of 15% is added.

Write a program to read the names of users and number of units consumed and print out the charges with names. W E B

4

CLASSES AND OBJECTS

C H A P T E R O U T L I N E

- 4.1 Introduction
- 4.2 C Structures Revisited
 - 4.2.1 Limitations of C Structure
 - 4.2.2 Extensions to Structures
- 4.3 Specifying a Class
 - 4.3.1 A Simple Class Example
 - 4.3.2 Creating Objects
 - 4.3.3 Accessing Class Members
- 4.4 Defining Member Functions
 - 4.4.1 Outside the Class Definition
 - 4.4.2 Inside the Class Definition
- 4.5 A C++ Program with Class
- 4.6 Making an Outside Function Inline
- 4.7 Nesting of Member Functions
- 4.8 Private Member Functions
- 4.9 Arrays within a Class
- 4.10 Memory Allocation for Objects
- 4.11 Static Data Members
- 4.12 Static Member Functions
- 4.13 Arrays of Objects
- 4.14 Objects as Function Arguments
- 4.15 Friendly Functions
- 4.16 Returning Objects
- 4.17 const Member Functions
- 4.18 Pointers to Members
- 4.19 Local Classes

Summary

Key Terms





4.1 INTRODUCTION

The most important feature of C++ is the “class”. Its significance is highlighted by the fact that Stroustrup initially gave the name “C with classes” to his new language. A class is an extension of the idea of structure used in C. It is a new way of creating and implementing a user-defined data type. We shall discuss, in this chapter, the concept of class by first reviewing the traditional structures found in C and then the ways in which classes can be designed, implemented and applied.



4.2 C STRUCTURES REVISITED

We know that one of the unique features of the C language is structures. They provide a method for packing together data of different types. A structure is a convenient tool for handling a group of logically related data items. It is a user-defined data type with a *template* that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations. For example, consider the following declaration:

```
struct student
{
    char    name[20];
    int     roll_number;
    float   total_marks;
};
```

The keyword **struct** declares **student** as a new data type that can hold three fields of different data types. These fields are known as *structure members* or *elements*. The identifier **student**, which is referred to as *structure name* or *structure tag*, can be used to create variables of type **student**. Example:

```
struct student A; // C declaration
```

A is a variable of type **student** and has three member variables as defined by the template. Member variables can be accessed using the *dot* or *period operator* as follows:

```
strcpy(A.name, "John");
A.roll_number = 999;
A.total_marks = 595.5;
Final_total = A.total_marks + 5;
```

Structures can have arrays, pointers or structures as members.

4.2.1 Limitations of C Structure

The standard C does not allow the struct data type to be treated like built-in types. For example, consider the following structure:

```
struct complex
{
    float x;
    float y;
};
struct complex c1, c2, c3;
```

The complex numbers `c1`, `c2`, and `c3` can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other. For example,

```
c3 = c1 + c2;
```

is illegal in C.

Another important limitation of C structures is that they do not permit *data hiding*. Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In other words, the structure members are public members.

4.2.2 Extensions to Structures

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of OOP. *Inheritance*, a mechanism by which one type can inherit characteristics from other types, is also supported by C++.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions.

In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword `struct` can be omitted in the declaration of structure variables. For example, we can declare the student variable `A` as

```
student A; // C++ declaration
```

Remember, this is an error in C.

C++ incorporates all these extensions in another user-defined type known as **class**. There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since `class` is a specially introduced data type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions. Therefore, we will not discuss structures any further.



Note

The only difference between a structure and a class in C++ is that, by default, the members of a class are private, while, by default, the members of a structure are public.



4.3 SPECIFYING A CLASS

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new *abstract data type* that can be treated like any other built-in data type. Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is

```
class class_name
{
    private:
        variable declarations;
        function declarations;
    public:
        variable declarations;
        function declaration;
};
```

The **class** declaration is similar to a **struct** declaration. The keyword **class** specifies, that what follows is an abstract data of type *class_name*. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called *class members*. They are usually grouped under two sections, namely, *private* and *public* to denote which of the members are *private* and which of them are *public*. The keywords **private** and **public** are known as visibility labels. Note that these keywords are followed by a colon.

The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword private is optional. By default, the members of a class are **private**. If both the labels are missing, then, by default, all the members are **private**. Such a class is completely hidden from the outside world and does not serve any purpose.

The variables declared inside the class are known as *data members* and the functions are known as *member functions*. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class. This is illustrated in Fig. 4.1. The binding of data and functions together into a single class-type variable is referred to as *encapsulation*.

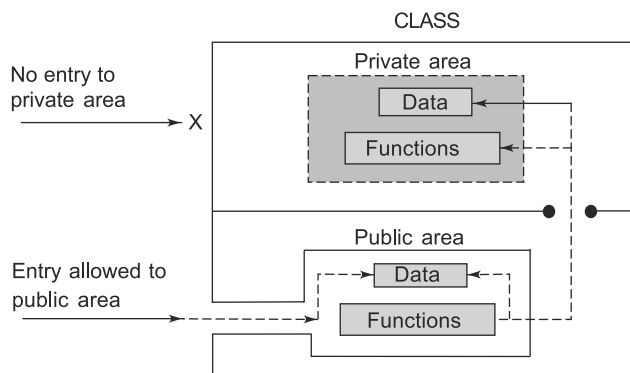


Fig. 4.1 Data hiding in classes

4.3.1 A Simple Class Example

A typical class declaration would look like

```
class item
{
    int number;           // variables declaration
    float cost;           // private by default
public:
    void getdata(int a, float b); // functions declaration
    void putdata(void);          // using prototype
};                          // ends with semicolon
```

We usually give a class some meaningful name, such as **item**. This name now becomes a new type identifier that can be used to declare instances of that class type. The class **item** contains two data members and two function members. The data members are private by default while both the functions are public by declaration. The function **getdata()** can be used to assign values to the member variables **number** and **cost**, and **putdata()** for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of the class **item**. Note that the functions are declared, not defined. Actual function definitions will appear later in the program. The data members are usually declared as **private** and the member functions as **public**. Figure 4.2 shows two different notations used by the OOP analysts to represent a class.

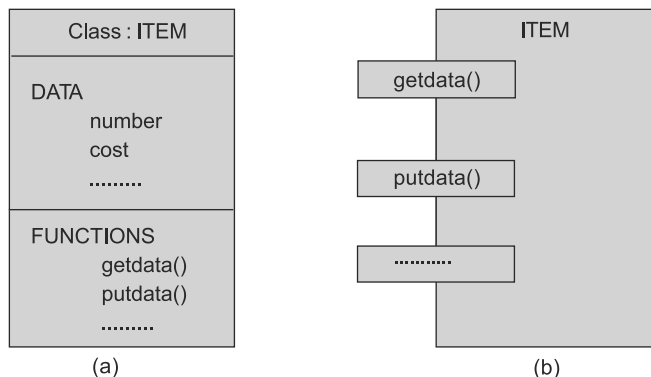


Fig. 4.2 Representation of a class

4.3.2 Creating Objects

Remember that the declaration of **item** as shown above does not define any objects of **item** but only specifies what they will contain. Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). For example,

```
item x; // memory for x is created
```

creates a variable **x** of type **item**. In C++, the class variables are known as *objects*. Therefore, **x** is called an object of type **item**. We may also declare more than one object in one statement. Example:

```
item x, y, z;
```


The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification, like a structure, provides only a *template* and does not create any memory space for the objects.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition

```
class item
{
    .....
    .....
    .....
} x,y,z;
```

would create the objects **x**, **y** and **z** of type **item**. This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

4.3.3 Accessing Class Members

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The **main()** cannot contain statements that access **number** and **cost** directly. The following is the format for calling a member function:

```
object-name.function-name (actual-arguments);
```

For example, the function call statement

```
x.getdata(100,75.5);
```

is valid and assigns the value 100 to **number** and 75.5 to **cost** of the object **x** by implementing the **getdata()** function. The assignments occur in the actual function. Please refer Sec. 4.4 for further details.

Similarly, the statement

```
x.putdata();
```

would display the values of data members. Remember, a member function can be invoked only by using an object (of the same class). The statement like

```
getdata(100,75.5);
```

has no meaning. Similarly, the statement

```
x.number = 100;
```

is also illegal. Although **x** is an object of the type **item** to which **number** belongs, the number (declared private) can be accessed only through a member function and not by the object directly.

It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member functions. For example,

```
x.putdata();
```

sends a message to the object **x** requesting it to display its contents.

A variable declared as public can be accessed by the objects directly. Example:

```
class xyz
{
    int x;
    int y;
    public:
        int z;
};

.....
.....
xyz p;
p.x = 0;           // error, x is private
p.z = 10           // OK, z is public
.....
.....
```

Note *The use of data in this manner defeats the very idea of data hiding and therefore should be avoided.*



4.4 DEFINING MEMBER FUNCTIONS

Member functions can be defined in two places:

- Outside the class definition.
- Inside the class definition.

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However, there is a subtle difference in the way the function header is defined. Both these approaches are discussed in detail in this section.

4.4.1 Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body. Since C++ does not support the old version of function definition, the *ANSI prototype* form must be used for defining the function header.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which **class** the function belongs to. The general form of a member function definition is

```
return-type class-name :: function-name (argument declaration)
{
    Function body
}
```

The membership label `class-name ::` tells the compiler that the function *function-name* belongs to the class *class-name*. That is, the scope of the function is restricted to the *class-name* specified in the header line. The symbol `::` is called the *scope resolution operator*.

For instance, consider the member functions **getdata()** and **putdata()** as discussed above. They may be coded as follows:

```
void item :: getdata(int a, float b)
{
    number = a;
    cost = b;
}
void item :: putdata(void)
{
    cout << "Number :" << number << "\n";
    cout << "Cost   :" << cost   << "\n";
}
```

Since these functions do not return any value, their return-type is `void`. Function arguments are declared using the ANSI prototype.

The member functions have some special characteristics that are often used in the program development. These characteristics are:

- Several different classes can use the same function name. The 'membership label' will resolve their scope.
- Member functions can access the private data of the class. A nonmember function cannot do so. (However, an exception to this rule is a *friend* function discussed later.)
- A member function can call another member function directly, without using the dot operator.

4.4.2 Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the `item` class as follows:

```
class item
{
    int number;
    float cost;
public:
    void getdata(int a, float b); // declaration
    // inline function
    void putdata(void)           // definition inside the class
    {
        cout << number << "\n";
        cout << cost   << "\n";
    }
};
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restrictions and limitations that apply to an **inline** function are also applicable here. Normally, only small functions are defined inside the class definition.



4.5 A C++ PROGRAM WITH CLASS

All the details discussed so far are implemented in Program 4.1.

Program 4.1 Class Implementation

```
#include <iostream>

using namespace std;

class item
{
    int number;    // private by default
    float cost;    // private by default
public:
    void getdata(int a, float b);           // prototype declaration,
                                           // to be defined

    // Function defined inside class
    void putdata(void)
    {
        cout << "number :" << number << "\n";
        cout << "cost :" << cost << "\n";
    }
};

//.....Member Function Definition .....
void item :: getdata(int a, float b)       // use membership label
{
    number = a;                           // private variables
    cost = b;                             // directly used
}

//.....Main Program.....

int main()
{
    item x;                               // create object x

    cout << "\nobject x " << "\n";

    x.getdata(100, 299.95);               // call member function
    x.putdata();                          // call member function

    item y;                               // create another object

    cout << "\nobject y" << "\n";

    y.getdata(200, 175.50);
    y.putdata();

    return 0;
}
```

This program features the class **item**. This class contains two private variables and two public functions. The member function **getdata()** which has been defined outside the class supplies values to both the variables. Note the use of statements such as

```
number = a;
```

in the function definition of **getdata()**. This shows that the member functions can have direct access to private data items.

The member function **putdata()** has been defined inside the class and therefore behaves like an **inline** function. This function displays the values of the private variables **number** and **cost**.

The program creates two objects, x and y in two different statements. This can be combined in one statement.

```
item x, y;          // creates a list of objects
```

Here is the output of Program 4.1:

```
object x
number   :100
cost     :299.95

object y
number   :200
cost     :175.5
```

For the sake of illustration we have shown one member function as **inline** and the other as an 'external' member function. Both can be defined as **inline** or external functions.



4.6 MAKING AN OUTSIDE FUNCTION INLINE

One of the objectives of OOP is to separate the details of implementation from the class definition. It is therefore good practice to define the member functions outside the class.

We can define a member function outside the class definition and still make it inline by just using the qualifier **inline** in the header line of function definition. Example:

```
class item
{
    ....
    ....
    public:
        void getdata(int a, float b);           // declaration
};
inline void item :: getdata(int a, float b)    // definition
{
    number = a;
    cost = b;
}
```



4.7 NESTING OF MEMBER FUNCTIONS

We just discussed that a member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as *nesting of member functions*. Program 4.2 illustrates this feature.

Program 4.2 Nesting of Member Functions

```
#include<iostream>
#include<conio.h>
#include<string>
using namespace std;

class binary
{
    string s;

public:
    void read(void)
    {
        cout<<"Enter a binary number:";
        cin>>s;
    }

    void chk_bin(void)
    {
        for(int i=0;i<s.length();i++)
        {
            if(s.at(i)!='0' && s.at(i)!='1')
            {
                cout<<"\nIncorrect binary number format...the program will
                quit";
                getch();
                exit(0);
            }
        }
    }

    void ones(void)
    {
        chk_bin(); //calling member function
        for(int i=0;i<s.length();i++)
        {
            if(s.at(i)=='0')
                s.at(i)='1';
            else
```

(Contd.)

```
        s.at(i)='0';
    }
}

void displayones(void)
{
    ones(); //calling member function
    cout<<"\nThe 1's compliment of the above binary number is:
    "<<s;
}

};

int main()
{
    binary b;
    b.read();
    b.displayones();
    getch();
    return 0;
}
```

The output of Program 4.2 would be the following:

Output 1:

```
Enter a binary number: 110101
The 1's compliment of the above binary number is: 001010
```

Output 2:

```
Enter a binary number: 1101210
Incorrect binary number format...the program will quit
```



Note

The above program uses the built-in class, string for storing the binary number. We will explore the string class in more detail in Chapter 15.



4.8 PRIVATE MEMBER FUNCTIONS

Although it is normal practice to place all the data items in a private section and all the functions in public, some situations may require certain functions to be hidden (like private data) from the outside calls. Tasks such as deleting an account in a customer file, or providing increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator. Consider a class as defined below:

```
class sample
{
    int m;
    void read(void); // private member function
public:
    void update(void);
    void write(void);
};
```

If **s1** is an object of **sample**, then

```
s1.read();           // won't work; objects cannot access
                     // private members
```

is illegal. However, the function **read()** can be called by the function **update()** to update the value of **m**.

```
void sample :: update(void)
{
    read(); // simple call; no object used
}
```



4.9 ARRAYS WITHIN A CLASS

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size=10; // provides value for array size

class array
{
    int a[size]; // 'a' is int type array
public:
    void setval(void);
    void display(void);
};
```

The array variable **a[]** declared as a private member of the class **array** can be used in the member functions, like any other array variable. We can perform any operations on it. For instance, in the above class definition, the member function **setval()** sets the values of elements of the array **a[]**, and **display()** function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

Let us consider a shopping list of items for which we place an order with a dealer every month. The list includes details such as the code number and price of each item. We would like to perform operations such as adding an item to the list, deleting an item from the list and printing the total value of the order. Program 4.3 shows how these operations are implemented using a class with arrays as data members.

Program 4.3 Processing Shopping List

```
#include <iostream>
using namespace std;
const m=50;
class ITEMS
{
    int itemCode[m];
    float itemPrice[m];
    int count;
public:
    void CNT(void){count = 0;}           // initializes count to 0
    void getitem(void);
    void displaySum(void);
    void remove(void);
    void displayItems(void);
};
//=====
void ITEMS :: getitem(void)             // assign values to data
                                        // members of item
{
    cout << "Enter item code :";
    cin >> itemCode[count];

    cout << "Enter item cost :";
    cin >> itemPrice[count];
    count++;
}
void ITEMS :: displaySum(void)          // display total value of
                                        // all items
{
    float sum = 0;
    for(int i=0; i<count; i++)
        sum = sum + itemPrice[i];
    cout << "\nTotal value :" << sum << "\n";
}
void ITEMS :: remove(void)             // delete a specified item
{
    int a;
    cout << "Enter item code :";
    cin >> a;

    for(int i=0; i<count; i++)
        if(itemCode[i] == a)
            itemPrice[i] = 0;
}
void ITEMS :: displayItems(void)       // displaying items
{
    cout << "\nCode Price\n";
```

(Contd.)

```

        for(int i=0; i<count; i++)
        {
            cout <<"\n" << itemCode[i];
            cout <<" " << itemPrice[i];
        }
        cout << "\n";
    }
    //=====

int main()
{
    ITEMS order;
    order.CNT();
    int x;
    do                                     // do...while loop
    {
        cout << "\nYou can do the following;"
              << "Enter appropriate number \n";
        cout << "\n1 : Add an item ";
        cout << "\n2 : Display total value";
        cout << "\n3 : Delete an item";
        cout << "\n4 : Display all items";
        cout << "\n5 : Quit";
        cout << "\n\nWhat is your option?";

        cin >> x;

        switch(x)
        {
            case 1 : order.getitem(); break;
            case 2 : order.displaySum(); break;
            case 3 : order.remove(); break;
            case 4 : order.displayItems(); break;
            case 5 : break;
            default : cout << "Error in input; try again\n";
        }
    } while(x != 5);                      // do...while ends

    return 0;
}

```

The output of Program 4.3 would be

```

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?1
Enter item code :111
Enter item cost :100

```

You can do the following; Enter appropriate number

- 1 : Add an item
- 2 : Display total value
- 3 : Delete an item
- 4 : Display all items
- 5 : Quit

What is your option?1

Enter item code :222

Enter item cost :200

You can do the following; Enter appropriate number

- 1 : Add an item
- 2 : Display total value
- 3 : Delete an item
- 4 : Display all items
- 5 : Quit

What is your option?1

Enter item code :333

Enter item cost :300

You can do the following; Enter appropriate number

- 1 : Add an item
- 2 : Display total value
- 3 : Delete an item
- 4 : Display all items
- 5 : Quit

What is your option?2

Total value :600

You can do the following; Enter appropriate number

- 1 : Add an item
- 2 : Display total value
- 3 : Delete an item
- 4 : Display all items
- 5 : Quit

What is your option?3

Enter item code :222

You can do the following; Enter appropriate number

- 1 : Add an item
- 2 : Display total value
- 3 : Delete an item
- 4 : Display all items
- 5 : Quit

What is your option?4

Code	Price

111	100

222	0

333	300

You can do the following; Enter appropriate number

```
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit
```

What is your option?5



Note

*The program uses two arrays, namely **itemCode[]** to hold the code number of items and **itemPrice[]** to hold the prices. A third data member **count** is used to keep a record of items in the list. The program uses a total of four functions to implement the operations to be performed on the list. The statement*

```
const int m = 50;
```

defines the size of the array members.

The first function **CNT()** simply sets the variable **count** to zero. The second function **getitem()** gets the item code and the item price interactively and assigns them to the array members **itemCode[count]** and **itemPrice[count]**. Note that inside this function **count** is incremented after the assignment operation is over. The function **displaySum()** first evaluates the total value of the order and then prints the value. The fourth function **remove()** deletes a given item from the list. It uses the item code to locate it in the list and sets the price to zero indicating that the item is not 'active' in the list. Lastly, the function **displayItems()** displays all the items in the list.

The program implements all the tasks using a menu-based user interface.



4.10 MEMORY ALLOCATION FOR OBJECTS

We have stated that the memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects. This is shown in Fig. 4.3.



4.11 STATIC DATA MEMBERS

A data member of a class can be qualified as static. The properties of a **static** member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are :

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

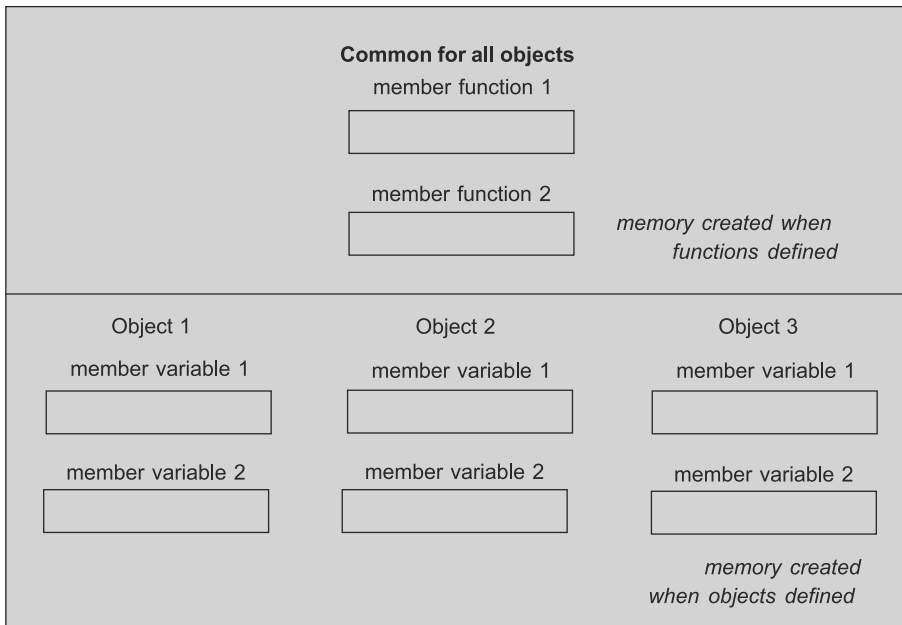


Fig. 4.3 Object of memory

- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects. Program 4.4 illustrates the use of a static data member.

Program 4.4 Static Class Member

```
#include <iostream>

using namespace std;

class item
{
    static int count;
    int number;
public:
    void getdata(int a)
    {
        number = a;
        count ++;
    }
    void getcount(void)
```

(Contd.)

```
{
    cout << "count: ";
    cout << count << "\n";
}
};

int item :: count;

int main()
{
    item a, b, c;           // count is initialized to zero
    a.getcount();           // display count
    b.getcount();
    c.getcount();

    a.getdata(100);         // getting data into object a
    b.getdata(200);         // getting data into object b
    c.getdata(300);         // getting data into object c

    cout << "After reading data" << "\n";

    a.getcount();           // display count
    b.getcount();
    c.getcount();
    return 0;
}
```

The output of the Program 4.4 would be

```
count: 0
count: 0
count: 0
After reading data
count: 3
count: 3
count: 3
```

**Note**

Notice the following statement in the program:

```
int item :: count; // definition of static data member
```

Note that the type and scope of each **static** member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as *class variables*.

The **static** variable **count** is initialized to zero when the objects are created. The count is incremented whenever the data is read into an object. Since the data is read into objects three times, the

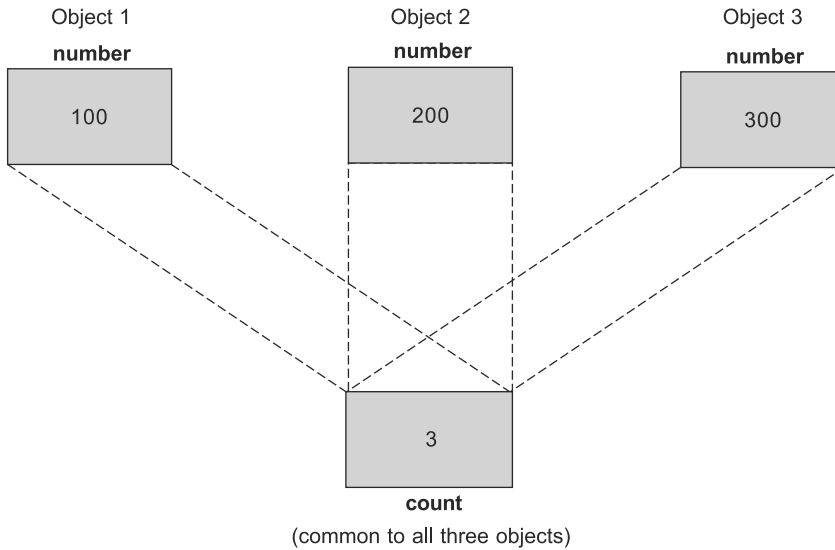


Fig. 4.4 *Sharing of a static data member*

variable `count` is incremented three times. Because there is only one copy of `count` shared by all the three objects, all the three output statements cause the value 3 to be displayed. Figure 4.4 shows how a static variable is used by the objects.

Static variables are like non-inline member functions as they are declared in a class declaration and defined in the source file. While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives `count` the initial value 10.

```
int item :: count = 10;
```



4.12 STATIC MEMBER FUNCTIONS

Like **static** member variable, we can also have **static** member functions. A member function that is declared **static** has the following properties:

- A **static** function can have access to only other static members (functions or variables) declared in the same class.
- A **static** member function can be called using the class name (instead of its objects) as follows:

```
class-name :: function-name;
```

Program 4.5 illustrates the implementation of these characteristics. The **static** function **showcount()** displays the number of objects created till that moment. A count of number of objects created is maintained by the **static** variable `count`.

The function **showcode()** displays the code number of each object.

Program 4.5 Statis Member Function

```
#include <iostream>

using namespace std;

class test
{
    int code;
    static int count;           // static member variable
public:
    void setcode(void)
    {
        code = ++count;
    }
    void showcode(void)
    {
        cout << "object number: " << code << "\n";
    }
    static void showcount(void) // static member function
    {
        cout << "count: " << count << "\n";
    }
};

int test :: count;
int main()
{
    test t1, t2;

    t1.setcode();
    t2.setcode();

    test :: showcount();       // accessing static function

    test t3;
    t3.setcode();

    test :: showcount();

    t1.showcode();
    t2.showcode();
    t3.showcode();

    return 0;
}
```

The output of Program 4.5 would be

```
count: 2
count: 3
object number: 1
object number: 2
object number: 3
```


**Note**

Note that the statement

```
code = ++count;
```

*is executed whenever **setcode()** function is invoked and the current value of **count** is assigned to **code**. Since each object has its own copy of **code**, the value contained in **code** represents a unique number of its object.*

Remember, the following function definition will not work:

```
static void showcount()
{
    cout << code;    // code is not static
}
```



4.13 ARRAYS OF OBJECTS

We know that an array can be of any data type including **struct**. Similarly, we can also have arrays of variables that are of the type **class**. Such variables are called *arrays of objects*. Consider the following class definition:

```
class employee
{
    char name[30];
    float age;
public:
    void getdata(void);
    void putdata(void);
};
```

The identifier **employee** is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example:

```
employee manager[3];        // array of manager
employee foreman[15];       // array of foreman
employee worker[75];        // array of worker
```

The array **manager** contains three objects(managers), namely, **manager[0]**, **manager[1]** and **manager[2]**, of type **employee** class. Similarly, the **foreman** array contains 15 objects (foremen) and the **worker** array contains 75 objects(workers).

Since an array of objects behaves like any other array, we can use the usual array-accessing methods to access individual elements, and then the dot member operator to access the member functions. For example, the statement

```
manager[i].putdata();
```

will display the data of the *i*th element of the array **manager**. That is, this statement requests the object **manager[i]** to invoke the member function **putdata()**.

An array of objects is stored inside the memory in the same way as a multi-dimensional array. The array **manager** is represented in Fig. 4.5. Note that only the space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.

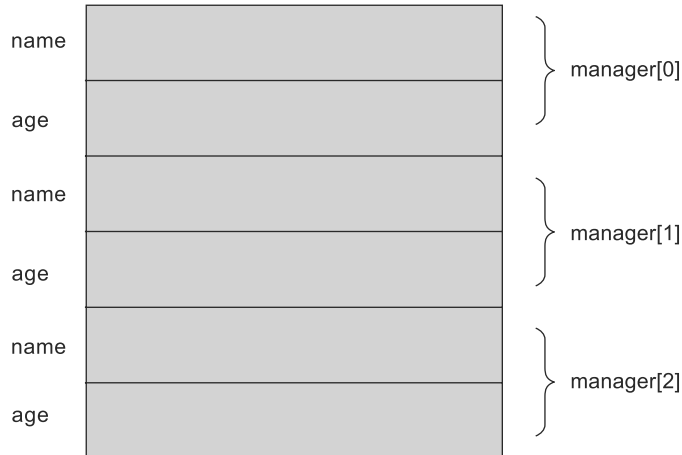


Fig. 4.5 Storage of data items of an object array

Program 4.6 illustrates the use of object arrays.

Program 4.6 Arrays of Objects

```
#include <iostream>

using namespace std;

class employee
{
    char name[30];    // string as class member
    float age;
public:
    void getdata(void);
    void putdata(void);
};

void employee :: getdata(void)
{
    cout << "Enter name: ";
    cin >> name;
    cout << "Enter age: ";
    cin >> age;
}

void employee :: putdata(void)
{
    cout << "Name: " << name << "\n";
    cout << "Age: " << age << "\n";
}

const int size=3;
int main()
{
```

(Contd.)

```

employee manager[size];
for(int i=0; i<size; i++)
{
    cout << "\nDetails of manager" << i+1 << "\n";
    manager[i].getdata();
}
cout << "\n";
for(i=0; i<size; i++)
{
    cout << "\nManager" << i+1 << "\n";
    manager[i]. putdata();
}
return 0;
}

```

This being an interactive program, the input data and the program output are shown below:

```

Interactive input
Details of manager1
Enter name: xxx
Enter age: 45

Details of manager2
Enter name: yyy
Enter age: 37

Details of manager3
Enter name: zzz
Enter age: 50

```

```

Program output
Manager1
Name: xxx
Age: 45

Manager2
Name: yyy
Age: 37

Manager3
Name: zzz
Age: 50

```



4.14 OBJECTS AS FUNCTION ARGUMENTS

Like any other data type, an object may be used as a function argument. This can be done in two ways:

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

The first method is called *pass-by-value*. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The

second method is called *pass-by-reference*. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Program 4.7 illustrates the use of objects as function arguments. It performs the addition of time in the hour and minutes format.

Program 4.7 Objects as Arguments

```
#include <iostream>

using namespace std;

class time
{
    int hours;
    int minutes;
public:
    void gettime(int h, int m)
    { hours = h; minutes = m; }
    void puttime(void)
    {
        cout << hours << " hours and ";
        cout << minutes << " minutes " << "\n";
    }
    void sum(time, time); // declaration with objects as arguments
};

void time :: sum(time t1, time t2)    // t1, t2 are objects
{
    minutes = t1.minutes + t2.minutes;
    hours = minutes/60;
    minutes = minutes%60;
    hours = hours + t1.hours + t2.hours;
}

int main()
{
    time T1, T2, T3;

    T1.gettime(2,45); // get T1
    T2.gettime(3,30); // get T2

    T3.sum(T1,T2); // T3=T1+T2

    cout << "T1 = "; T1.puttime();        // display T1
    cout << "T2 = "; T2.puttime();        // display T2
    cout << "T3 = "; T3.puttime();        // display T3

    return 0;
}
```

The output of Program 4.7 would be:

```
T1 = 2 hours and 45 minutes
T2 = 3 hours and 30 minutes
T3 = 6 hours and 15 minutes
```



Note

Since the member function `sum()` is invoked by the object **T3**, with the objects **T1** and **T2** as arguments, it can directly access the hours and minutes variables of **T3**. But, the members of **T1** and **T2** can be accessed only by using the dot operator (like **T1.hours** and **T1.minutes**). Therefore, inside the function `sum()`, the variables `hours` and `minutes` refer to **T3**, **T1.hours** and **T1.minutes** refer to **T1**, and **T2.hours** and **T2.minutes** refer to **T2**.

Figure 4.6 illustrates how the members are accessed inside the function `sum()`.

An object can also be passed as an argument to a non-member function. However, such functions can have access to the **public member** functions only through the objects passed as arguments to it. These functions cannot have access to the private data members.

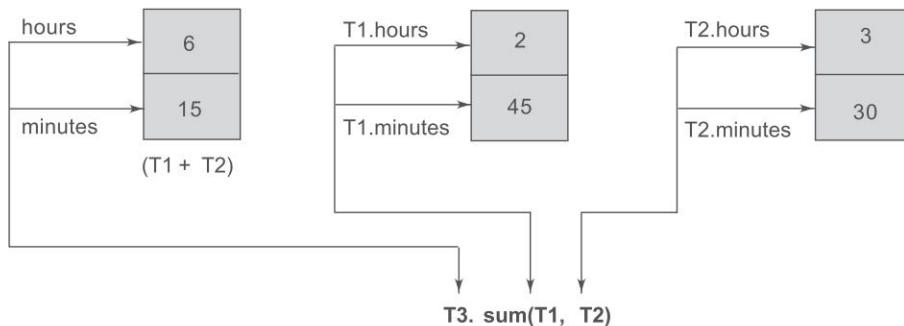


Fig. 4.6 Accessing members of objects within a called function



4.15 FRIENDLY FUNCTIONS

We have been emphasizing throughout this chapter that the private members cannot be accessed from outside the class. That is, a nonmember function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. For example, consider a case where two classes, **manager** and **scientist**, have been defined. We would like to use a function `income_tax()` to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

To make an outside function “friendly” to a class, we have to simply declare this function as a **friend** of the class as shown below:

```

class ABC
{
    .....
    .....
    public:
        .....
        .....
        friend void xyz(void); // declaration
};

```

The function declaration should be preceded by the keyword **friend**. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or the scope operator `::`. The functions that are declared with the keyword **friend** are known as *friend functions*. A function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as **friend**.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.(e.g., A.x).
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

The friend functions are often used in operator overloading which will be discussed later.

Program 4.8 illustrates the use of a friend function.

Program 4.8 Friend Function

```

#include <iostream>

using namespace std;

class sample
{
    int a;
    int b;
    public:
        void setvalue() {a=25; b=40; }
        friend float mean(sample s);
};

float mean(sample s)
{
    return float(s.a + s.b)/2.0;
}

```

(Contd.)

```

int main()
{
    sample X; // object X
    X.setvalue();
    cout << "Mean value = " << mean(X) << "\n";

    return 0;
}

```

The output of Program 4.8 would be

```
Mean value = 32.5
```



Note

The friend function accesses the class variables **a** and **b** by using the dot operator and the object passed to it. The function call **mean(X)** passes the object **X** by value to the friend function.

Member functions of one class can be **friend** functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

```

class X
{
    .....
    .....
    int fun1();                // member function of X
    .....
};

class Y
{
    .....
    .....
    friend int X :: fun1();      // fun1() of X
                                // is friend of Y
    .....
};

```

The function **fun1()** is a member of **class X** and a **friend** of class **Y**.

We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**. This can be specified as follows:

```

class Z
{
    .....
    friend class X;            // all member functions of X are
                                // friends to Z
};

```

Program 4.9 demonstrates how friend functions work as a bridge between the classes.

Program 4.9 Using Friend Function to Add Data Objects of two Different Classes

```
#include <iostream>
using namespace std;

class ABC; //Forward declaration
//-----//
class XYZ
{
    int data;
public:
    void setvalue(int value)
    {
        data=value;
    }
    friend void add(XYZ, ABC); //Friend function declaration
};
//-----//
class ABC
{
    int data;
public:
    void setvalue(int value)
    {
        data=value;
    }
    friend void add(XYZ, ABC); //Friend function declaration
};
//-----//
void add(XYZ obj1, ABC obj2) //Friend function definition
{
    cout<<"Sum of data values of XYZ and ABC objects using friend
    function = "<<obj1.data+obj2.data;
}
//-----//
int main()
{
    XYZ X;
    ABC A;
    X.setvalue(5);
    A.setvalue(50);
    add(X, A); //Calling friend function
    return 0;
}
```

The output of Program 4.9 would be

Sum of data values of XYZ and ABC objects using friend function = 55

**Note**

The function `add()` has arguments from both *XYZ* and *ABC*. When the function `add()` is declared as a friend in *XYZ* for the first time, the compiler will not acknowledge the presence of *ABC* unless its name is declared in the beginning as

```
class ABC;
```

This is known as 'forward' declaration.

As pointed out earlier, a friend function can be called by reference. In this case, local copies of the objects are not made. Instead, a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.

This method can be used to alter the values of the private members of a class. Remember, altering the values of private members is against the basic principles of data hiding. It should be used only when absolutely necessary.

Program 4.10 shows how to use a common friend function to exchange the private values of two classes. The function is called by reference.

Program 4.10 Swapping Private Data of Classes

```
#include <iostream>

using namespace std;

class class_2;

class class_1
{
    int value1;
public:
    void indata(int a) {value1 = a;}
    void display(void) {cout << value1 << "\n";}
    friend void exchange(class_1 &, class_2 &);
};

class class_2
{
    int value2;
public:
    void indata(int a) {value2 = a;}
    void display(void) {cout << value2 << "\n";}
    friend void exchange(class_1 &, class_2 &);
};

void exchange(class_1 & x, class_2 & y)
{
```

(Contd.)

```

        int temp = x.value1;
        x.value1 = y.value2;
        y.value2 = temp;
    }

    int main()
    {
        class_1 C1;
        class_2 C2;

        C1.indata(100);
        C2.indata(200);

        cout << "Values before exchange" << "\n";
        C1.display();
        C2.display();
        exchange(C1, C2); // swapping

        cout << "Values after exchange " << "\n";
        C1.display();
        C2.display();

        return 0;
    }

```

The objects **x** and **y** are aliases of **C1** and **C2** respectively. The statements

```

int temp = x.value1
x.value1 = y.value2;
y.value2 = temp;

```

directly modify the values of **value1** and **value2** declared in **class_1** and **class_2**.

The output of Program 4.10 would be

```

Values before exchange
100
200
Values after exchange
200
100

```



4.16 RETURNING OBJECTS

A function cannot only receive objects as arguments but also can return them. The example in Program 4.11 illustrates how an object can be created (within a function) and returned to another function.

Program 4.11 **Returning Objects**

```
#include<iostream>
#include<conio.h>
using namespace std;
class matrix
{
    int m[3][3];
public:
    void read(void)
    {
        cout<<"Enter the elements of the 3X3 matrix:\n";
        int i,j;
        for(i=0;i<3;i++)
            for(j=0;j<3;j++)
            {
                cout<<"m["<<i<<"["<<j<<" = ";
                cin>>m[i][j];
            }
    }

    void display (void)
    {
        int i,j;
        for(i=0;i<3;i++)
        {
            cout<<"\n";
            for(j=0;j<3;j++)
            {
                cout<<m[i][j]<<"\t";
            }
        }
    }

    friend matrix trans(matrix);
};

matrix trans(matrix m1)
{
    matrix m2; //creating an object
    int i,j;

    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            m2.m[i][j]=m1.m[j][i];

    return(m2); //returning an object
}
```

(Contd.)

```

int main()
{
    matrix mat1,mat2;
    mat1.read();
    cout<<"\nYou entered the following matrix:";
    mat1.display();

    mat2 = trans(mat1);
    cout<<"\nTransposed matrix:";
    mat2.display();

    getch();
    return 0;
}

```

Upon execution, Program 4.11 would generate the following output:

Enter the elements of the 3×3 matrix:

```

m[0][0] = 1
m[0][1] = 2
m[0][2] = 3
m[1][0] = 4
m[1][1] = 5
m[1][2] = 6
m[2][0] = 7
m[2][1] = 8
m[2][2] = 9

```

You entered the following matrix:

```

1   2   3
4   5   6
7   8   9

```

Transposed matrix:

```

1   4   7
2   5   8
3   6   9

```

The program finds the transpose of a given 3×3 matrix and stores it in a new matrix object. The display member function displays the matrix elements.



4.17 const MEMBER FUNCTIONS

If a member function does not alter any data in the class, then we may declare it as a **const** member function as follows:

```

void mul(int, int) const;
double get_balance() const;

```

The qualifier **const** is appended to the function prototypes (in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.



4.18 POINTERS TO MEMBERS

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator **&** to a “fully qualified” class member name. A class member pointer can be declared using the operator **::*** with the class name. For example, given the class

```
class A
{
    private:
        int m;
    public:
        void show();
};
```

We can define a pointer to the member **m** as follows:

```
int A::* ip = &A :: m;
```

The **ip** pointer created thus acts like a class member in that it must be invoked with a class object. In the statement above, the phrase **A::*** means “pointer-to-member of **A** class”. The phrase **&A::m** means the “address of the **m** member of **A** class”.

Remember, the following statement is not valid:

```
int *ip = &m;           // won't work
```

This is because **m** is not simply an **int** type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer **ip** can now be used to access the member **m** inside member functions (or friend functions). Let us assume that **a** is an object of **A** declared in a member function. We can access **m** using the pointer **ip** as follows:

```
cout << a.*ip;           // display
cout << a.m;             // same as above
```

Now, look at the following code:

```
ap = &a;                 // ap is pointer to object a
cout << ap -> *ip;        // display m
cout << ap -> m;          // same as above
```

The *dereferencing operator* **->*** is used to access a member when we use pointers to both the object and the member. The *dereferencing operator* ***** is used when the object itself is used with the member pointer. Note that ***ip** is used like a member name.

We can also design pointers to member functions which, then, can be invoked using the dereferencing operators in the **main** as shown below:

```
(object-name .* pointer-to-member function) (10);
(pointer-to-object ->* pointer-to-member function) (10)
```

The precedence of () is higher than that of .* and ->*, so the parentheses are necessary.

Program 4.12 illustrates the use of dereferencing operators to access the class members.

Program 4.12 Dereferencing Operator

```
#include<iostream>

using namespace std;

class M
{
    int x;
    int y;
public:
    void set_xy(int a, int b)
    {
        x = a;
        y = b;
    }
    friend int sum(M m);
};

int sum(M m)
{
    int M::* px = &M::x;
    int M::* py = &M::y;
    M *pm = &m;
    int S = m.*px + pm->*py;
    return S;
}

int main()
{
    M n;
    void (M::*pf)(int,int) = &M::set_xy;
    (n.*pf)(10,20);
    cout << "SUM = " << sum(n) << "\n";

    M *op = &n;
    (op->*pf)(30,40);
    cout << "SUM = " << sum(n) << "\n";

    return 0;
}
```

The output of Program 4.12 would be

```
sum = 30
sum = 70
```



4.19 LOCAL CLASSES

Classes can be defined and used inside a function or a block. Such classes are called local classes. Examples:

```
void test(int a)                // function
{
    .....
    .....
    class student              // local class
    {
        .....
        .....                // class definition
        .....
    };
    .....
    .....
    student s1(a);             // create student object
    .....                     // use student object
}
```

Local classes can use global variables (declared above the function) and static variables declared inside the function but cannot use automatic local variables. The global variables should be used with the scope operator (::).

There are some restrictions in constructing local classes. They cannot have static data members and member functions must be defined inside the local classes. Enclosing function cannot access the private members of a local class. However, we can achieve this by declaring the enclosing function as a friend.

SUMMARY

- ☐ A class is an extension to the structure data type. A class can have both variables and functions as members.
- ☐ By default, members of the class are private whereas that of structure are public.
- ☐ Only the member functions can have access to the private data members and private functions. However the public members can be accessed from outside the class.
- ☐ In C++, the class variables are called objects. With objects we can access the public members of a class using a dot operator.
- ☐ We can define the member functions inside or outside the class. The difference between a member function and a normal function is that a member function uses a membership 'identity' label in the header to indicate the class to which it belongs.
- ☐ The memory space for the objects is allocated when they are declared. Space for member variables is allocated separately for each object, but no separate space is allocated for member functions.
- ☐ A data member of a class can be declared as a **static** and is normally used to maintain values common to the entire class.
- ☐ The static member variables must be defined outside the class.

- ☐ A static member function can have access to the static members declared in the same class and can be called using the class name.
- ☐ C++ allows us to have arrays of objects.
- ☐ We may use objects as function arguments.
- ☐ A function declared as a **friend** is not in the scope of the class to which it has been declared as friend. It has full access to the private members of the class.
- ☐ A function can also return an object.
- ☐ If a member function does not alter any data in the class, then we may declare it as a **const** member function. The keyword **const** is appended to the function prototype.
- ☐ It is also possible to define and use a class inside a function. Such a class is called a local class

Key Terms

abstract data type | arrays of objects | **class** | class declaration | class members | class variables | const member functions | data hiding | data members | dereferencing operator | dot operator | elements | encapsulation | **friend** functions | inheritance | **inline** functions | local class | member functions | nesting of member functions | objects | pass-by-reference | pass-by-value | period operator | **private** | proto-type | **public** | scope operator | scope resolution | static data members | static member functions | static variables | **struct** | structure | structure members | structure name | structure tag | template

Review Questions

- 4.1 How do structures in C and C++ differ?
- 4.2 What is a class? How does it accomplish data hiding?
- 4.3 How does a C++ structure differ from a C++ class?
- 4.4 What are objects? How are they created?
- 4.5 How is a member function of a class defined?
- 4.6 Can we use the same function name for a member function of a class and an outside function in the same program file? If yes, how are they distinguished? If no, give reasons.
- 4.7 Describe the mechanism of accessing data members and member functions in the following cases:
 - (a) Inside the **main** program.
 - (b) Inside a member function of the same class.
 - (c) Inside a member function of another class.
- 4.8 When do we declare a member of a class **static**? Justify the need of static members. F A Q
- 4.9 What is a friend function? What are the merits and demerits of using friend functions?
- 4.10 Can we pass class objects as function arguments? Explain with the help of an example.
- 4.11 State whether the following statements are TRUE or FALSE.

- (a) Data items in a class must always be private.
- (b) A function designed as private is accessible only to member functions of that class.
- (c) A function designed as public can be accessed like any other ordinary functions.
- (d) Member functions defined inside a class specifier become inline functions by default.
- (e) Classes can bring together all aspects of an entity in one place.
- (f) Class members are public by default.
- (g) Friend functions have access to only public members of a class.
- (h) An entire class can be made a friend of another class.
- (i) Functions cannot return class objects.
- (j) Data members can be initialized inside class specifier.

Debugging Exercises

4.1 Identify the error in the following program.

```
#include <iostream.h>
struct Room
{
    int width;
    int length;
    void setValue(int w, int l)
    {
        width = w;
        length = l;
    }
};
void main()
{
    Room objRoom;
    objRoom.setValue(12, 1,4);
}
```

4.2 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
    int width, height;
    void setValue(int w, int h)
    {
        width = w;
        height = h;
    }
};
void main()
{
    Room objRoom;
    objRoom.width = 12;
}
```

4.3 Identify the error in the following program.

```
#include <iostream.h>
class Item
{
private:
    static int count;
public:
    Item()
    {
        count++;
    }
    int getCount()
    {
        return count;
    }
    int* getCountAddress()
    {
        return count;
    }
};
int Item::count = 0;

void main()
{
    Item objItem1;
    Item objItem2;

    cout << objItem1.getCount() << ' ';
    cout << objItem2.getCount() << ' ';

    cout << objItem1.getCountAddress() << ' ';
    cout << objItem2.getCountAddress() << ' ';
}
```

4.4 Identify the error in the following program.

```
#include <iostream.h>
class staticFunction
{
    static int count;
public:
    static void setCount()
    {
        count++;
    }
    void displayCount()
    {
        cout << count;
    }
};
```

```
int staticFunction::count = 10;
void main()
{
    staticFunction obj1;
    obj1.setCount(5);
    staticFunction::setCount();
    obj1.displayCount();
}
```

4.5 Identify the error in the following program.

```
#include <iostream.h>
class Length
{
    int feet;
    float inches;
public:
    Length()
    {
        feet = 5;
        inches = 6.0;
    }
    Length(int f, float in)
    {
        feet = f;
        inches=in;
    }
    Length addLength(Length l)
    {
        l.inches += this->inches;
        l.feet += this->feet;
        if(l.inches>12)
        {
            l.inches-=12;
            l.feet++;
        }
        return l;
    }
    int getFeet()
    {
        return feet;
    }
    float getInches()
    {
        return inches;
    }
};
void main()
{
    Length objLength1;
    Length objLength1(5, 6.5);
    objLength1 = objLength1.addLength(objLength2);
}
```

```

        cout << objLength1.getFeet() << ' ';
        cout << objLength1.getInches() << ' ';
    }

```

4.6 Identify the error in the following program.

```

#include <iostream.h>
class Room;
void Area()
{
    int width, height;
    class Room
    {
        int width, height;
    public:
        void setValue(int w, int h)
        {
            width = w;
            height = h;
        }
        void displayValues()
        {
            cout << (float)width << ' ' << (float)height;
        }
    };
    Room objRoom1;
    objRoom1.setValue(12, 8);
    objRoom1.displayValues();
}

void main()
{
    Area();
    Room objRoom2;
}

```

Programming Exercises

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

Data members

- (a) Name of the depositor
- (b) Account number
- (c) Type of account
- (d) Balance amount in the account

Member functions

- (a) To assign initial values
- (b) To deposit an amount
- (c) To withdraw an amount after checking the balance

(d) To display name and balance

Write a main program to test the program. **W E B**

- 4.2** Write a class to represent a vector (a series of float values). Include member functions to perform the following tasks:

- (a) To create the vector
- (b) To modify the value of a given element
- (c) To multiply by a scalar value
- (d) To display the vector in the form (10, 20, 30, ...)

Write a program to test your class.

- 4.3** Modify the class and the program of Exercise 4.11 for handling 10 customers. **W E B**

- 4.4** Modify the class and program of Exercise 4.12 such that the program would be able to add two vectors and display the resultant vector. (Note that we can pass objects as function arguments.)

- 4.5** Create two classes **DM** and **DB** which store the value of distances. **DM** stores distances in metres and centimetres and **DB** in feet and inches. Write a program that can read values for the class objects and add one object of **DM** with another object of **DB**. **W E B**

Use a friend function to carry out the addition operation. The object that stores the results may be a **DM** object or **DB** object, depending on the units in which the results are required.

The display should be in the format of feet and inches or metres and centimetres depending on the object on display.

- 4.6** Refer to Program 4.11 and write a function that receives two matrix objects as arguments and returns a new matrix object containing their multiplication result. **W E B**

5

FUNCTIONS IN C++

C H A P T E R O U T L I N E

- 5.1 Introduction
- 5.2 The Main Function
- 5.3 Function Prototyping
- 5.4 Call by Reference
- 5.5 Return by Reference
- 5.6 Inline Functions
- 5.7 Default Arguments
- 5.8 Const Arguments
- 5.9 Recursion
- 5.10 Function Overloading
- 5.11 Friend and Virtual Functions
- 5.12 Math Library Functions

Summary

Key Terms





5.1 INTRODUCTION

We know that functions play an important role in C program development. Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

Recall that we have used a syntax similar to the following in developing C programs.

```
void show();           /* Function declaration */
main()
{
    .....
    show();           /* Function call */
    .....
}
void show()            /* Function definition */
{
    .....
    .....           /* Function body */
    .....
}
```

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered. C++ is no exception. Functions continue to be the building blocks of C++ programs. In fact, C++ has added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ function can be overloaded to make it perform different tasks depending on the arguments passed to it. Most of these modifications are aimed at meeting the requirements of object-oriented facilities.

In this chapter, we shall briefly discuss the various new features that are added to C++ functions and their implementation.



5.2 THE MAIN FUNCTION

C does not specify any return type for the **main()** function which is the starting point for the execution of a program. The definition of **main()** would look like this:

```
main()
{
    // main program statements
}
```

This is perfectly valid because the **main()** in C does not return any value.

In C++, the **main()** returns a value of type **int** to the operating system. C++, therefore, explicitly defines **main()** as matching one of the following prototypes:

```
int main();
int main(int argc, char * argv[]);
```

The functions that have a return value should use the **return** statement for termination. The **main()** function in C++ is, therefore, defined as follows:

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

Since the return type of functions is **int** by default, the keyword **int** in the **main()** header is optional. Most C++ compilers will generate an error or warning if there is no **return** statement. Turbo C++ issues the warning

Function should return a value

and then proceeds to compile the program. It is good programming practice to actually return a value from **main()**.

Many operating systems test the return value (called **exit value**) to determine if there is any problem. The normal convention is that an exit value of zero means the program ran successfully, while a nonzero value means there was a problem. The explicit use of a **return(0)** statement will indicate that the program was successfully executed.



5.3 FUNCTION PROTOTYPING

Function prototyping is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. With function prototyping, a *template* is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exist in the conventional C functions.

Remember, C also uses prototyping. But it was introduced first in C++ by Stroustrup and the success of this feature inspired the ANSI C committee to adopt it. However, there is a major difference in prototyping between C and C++. While C++ makes the prototyping essential, ANSI C makes it optional, perhaps, to preserve the compatibility with classic C.

Function prototype is a *declaration statement* in the calling program and is of the following form:

```
type function-name (argument-list);
```

The *argument-list* contains the types and names of arguments that must be passed to the function.

Example:

```
float volume(int x, float y, float z);
```


Note that each argument variable must be declared independently inside the parentheses. That is, a combined declaration like

```
float volume(int x, float y, z);
```

is illegal.

In a function declaration, the names of the arguments are *dummy* variables and therefore, they are optional. That is, the form

```
float volume(int, float, float);
```

is acceptable at the place of declaration. At this stage, the compiler only checks for the type of arguments when the function is called.

In general, we can either include or exclude the variable names in the argument list of prototypes. The variable names in the prototype just act as placeholders and, therefore, if names are used, they don't have to match the names used in the *function call* or *function definition*.

In the function definition, names are required because the arguments must be referenced inside the function. Example:

```
float volume(int a, float b, float c)
{
    float v = a*b*c;
    ....
    ....
}
```

The function **volume()** can be invoked in a program as follows:

```
float cube1 = volume(b1,w1,h1); // Function call
```

The variable **b1**, **w1**, and **h1** are known as the actual parameters which specify the dimensions of **cube1**. Their types (which have been declared earlier) should match with the types declared in the prototype. Remember, the calling statement should not include type names in the argument list.

We can also declare a function with an *empty argument list*, as in the following example:

```
void display( );
```

In C++, this means that the function does not pass any parameters. It is identical to the statement

```
void display(void);
```

However, in C, an empty parentheses implies any number of arguments. That is, we have foregone prototyping. A C++ function can also have an 'open' parameter list by the use of ellipses in the prototype as shown below:

```
void do_something(...);
```



5.4 CALL BY REFERENCE

In traditional C, a function call passes arguments by value. The *called function* creates a new set of variables and copies the values of arguments into them. The function does not have access to the actual

variables in the calling program and can only work on the copies of values. This mechanism is fine if the function does not need to alter the values of the original variables in the calling program. But, there may arise situations where we would like to change the values of variables in the *calling program*. For example, in bubble sort, we compare two adjacent elements in the list and interchange their values if the first element is greater than the second. If a function is used for *bubble sort*, then it should be able to alter the values of variables in the calling function, which is not possible if the call-by-value method is used.

Provision of the *reference variables* in C++ permits us to pass parameters to the functions by reference. When we pass arguments by reference, the 'formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. Consider the following function:

```
void swap(int &a,int &b)    // a and b are reference variables
{
    int t = a;             // Dynamic initialization
    a = b;
    b = t;
}
```

Now, if *m* and *n* are two integer variables, then the function call

```
swap(m, n);
```

will exchange the values of *m* and *n* using their aliases (reference variables) *a* and *b*. Reference variables have been discussed in detail in Chapter 3. In traditional C, this is accomplished using *pointers* and *indirection* as follows:

```
void swap1(int *a, int *b) /* Function definition */
{
    int t;
    t = *a;                /* assign the value at address a to t */
    *a = *b;               /* put the value at b into a */
    *b = t;                /* put the value at t into b */
}
```

This function can be called as follows:

```
swap1(&x, &y);            /* call by passing */
                          /* addresses of variables */
```

This approach is also acceptable in C++. Note that the call-by-reference method is neater in its approach.



5.5 RETURN BY REFERENCE

A function can also return a reference. Consider the following function:

```
int & max(int &x, int &y)
{
    if (x > y)
```

```

        return x;
    else
        return y;
}

```

Since the return type of **max()** is **int &**, the function returns reference to **x** or **y** (and not the values). Then a function call such as **max(a, b)** will yield a reference to either **a** or **b** depending on their values. This means that this function call can appear on the left-hand side of an assignment statement. That is, the statement

```
max(a,b) = -1;
```

is legal and assigns **-1** to **a** if it is larger, otherwise **-1** to **b**.



5.6 INLINE FUNCTIONS

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as *macros*. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called *inline function*. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code (something similar to macros expansion). The inline functions are defined as follows:

```

inline function-header
{
    function body
}

```

Example:

```

inline double cube(double a)
{
    return(a*a*a);
}

```

The above inline function can be invoked by statements like

```

c = cube(3.0);
d = cube(2.5+1.5);

```

On the execution of these statements, the values of **c** and **d** will be 27 and 64 respectively. If the arguments are expressions such as **2.5 + 1.5**, the function passes the value of the expression, 4 in this case. This makes the inline feature far superior to macros.

It is easy to make a function inline. All we need to do is to prefix the keyword **inline** to the function definition. All inline functions must be defined before they are called.

We should exercise care before making a function **inline**. The speed benefits of **inline** functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of **inline** functions may be lost. In such cases, the use of normal functions will be more meaningful. Usually, the functions are made inline when they are small enough to be defined in one or two lines. Example:

```
inline double cube(double a) {return(a*a*a);}
```

Remember that the inline keyword merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a **switch**, or a **goto** exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain **static** variables.
4. If **inline** functions are recursive.



Note *Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. However, it makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called. So, a trade-off becomes necessary.*

Program 5.1 illustrates the use of inline functions.

Program 5.1 Inline Functions

```
#include <iostream>

using namespace std;

inline float mul(float x, float y)
{
    return(x*y);
}

inline double div(double p, double q)
{
    return(p/q);
}

int main()
{
    float a = 12.345;
```

(Contd.)

```

float b = 9.82;

cout << mul(a,b) << "\n";
cout << div(a,b) << "\n";

return 0;
}

```

The output of Program 5.1 would be

```

121.228
1.25713

```



5.7 DEFAULT ARGUMENTS

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a *default value* to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype (i.e., function declaration) with default values:

```
float amount(float principal,int period,float rate=0.15);
```

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument **rate**. A subsequent function call like

```
value = amount(5000,7);           // one argument missing
```

passes the value of 5000 to **principal** and 7 to **period** and then lets the function use default value of 0.15 for **rate**. The call

```
value = amount(5000,5,0.12);      // no missing argument
```

passes an explicit value of 0.12 to **rate**.

A default argument is checked for type at the time of declaration and evaluated at the time of call. One important point to note is that only the trailing arguments can have default values and therefore we must add defaults from *right to left*. We cannot provide a default value to a particular argument in the middle of an argument list.

Some examples of function declaration with default values are

```

int mul(int i, int j=5, int k=10);           // legal
int mul(int i=5, int j);                     // illegal
int mul(int i=0, int j, int k=10);           // illegal
int mul(int i=2, int j=5, int k=10);         // legal

```

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using default arguments, a programmer can use only

those arguments that are meaningful to a particular situation. Program 5.2 illustrates the use of default arguments.

Program 5.2 Default Arguments

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

using namespace std;
int main()
{
    float amount;
    float value(float p, int n, float r=0.15); //prototype
    void printline(char ch='*',int len=40);    //prototype

    printline();                               //use default values for arguments

    amount = value (5000.00,5);                //default for 3rd argument
    cout<<"\n Final Value = "<<amount<<"\n\n";

    amount = value (10000.00,5,0.30); //pass all arguments explicitly

    cout<<"\n Final Value = "<<amount<<"\n\n";

    printline('=');                            //use default value for second argument

    getch();
    return 0;
}

float value (float p, int n, float r)
{
    int year = 1;
    float sum = p;

    while(year<=n)
    {
        sum = sum*(1+r);
        year = year+1;
    }
    return(sum);
}

void printline(char ch, int len)
{
    for(int i=1;i<=len;i++)
        printf("%c",ch);
    printf("\n");
}
```

The output of Program 5.2 would be

```
*****
Final Value = 10056.8
Final Value = 37129.3
=====
```

Advantages of providing the default arguments are:

1. We can use default arguments to add new parameters to the existing functions.
2. Default arguments can be used to combine similar functions into one.



5.8 const ARGUMENTS

In C++, an argument to a function can be declared as `const` as shown below.

```
int strlen(const char *p);
int length(const string &s);
```

The qualifier **const** tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.



5.9 RECURSION

Recursion is a situation where a function calls itself meaning, one of the statements in the function definition makes a call to the same function in which it is present. It may sound like an infinite looping condition but just as a loop has a conditional check to take the program control out of the loop, a recursive function also possesses a base case which returns the program control from the current instance of the function to call back to the calling function. For example, in a series of recursive calls to compute the factorial of a number, the base case would be a situation where factorial of 0 is to be computed. Let us consider a few examples (Program 5.3 and 5.4) to understand how the recursive approach works.

Program 5.3 Calculating Factorial of a Number

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

long fact(int n)
{
    if (n == 0)                //base case
        return 1;

    return (n * fact(n-1)); //recursive function call
}
```

(Contd.)

```

int main()
{
    int num;

    cout << "Enter a positive integer: ";
    cin >> num;

    cout << "Factorial of " << num << " is " << fact(num);

    getch();
    return 0;
}

```

The output of Program 5.3 would be

```

Enter a positive integer: 10
Factorial of 10 is 3628800

```

Program 5.4 Solving Tower of Hanoi Problem

```

#include <iostream>
#include <conio.h>
using namespace std;

void TOH(int d,char tower1, char tower2, char tower3)
{
    if (d==1) //base case
    {
        cout << "\nShift top disk from tower " << tower1 << " to tower "
            << tower2;
        return;
    }

    TOH(d-1,tower1, tower3, tower2); //recursive function call
    cout << "\nShift top disk from tower " << tower1 << " to tower "
        << tower2;

    TOH(d-1,tower3, tower2, tower1); //recursive function call
}

int main()
{
    int disk;

    cout << "Enter the number of disks: ";
    cin >> disk;

    if (disk < 1)
        cout << "\nThere are no disks to shift";
}

```

(Contd.)


```

else
    cout << "\nThere are " << disk << " disks in tower 1\n";

TOH(disk, '1', '2', '3');
cout << "\n\n" << disk << " disks in tower 1 are shifted to
tower 2";

getch();
return 0;
}

```

The output of Program 5.4 would be

```

Enter the number of disks: 3
There are 3 disks in tower 1
Shift top disk from tower 1 to tower 2
Shift top disk from tower 1 to tower 3
Shift top disk from tower 2 to tower 3
Shift top disk from tower 1 to tower 2
Shift top disk from tower 3 to tower 1
Shift top disk from tower 3 to tower 2
Shift top disk from tower 1 to tower 2

3 disks in tower 1 are shifted to tower 2

```



5.10 FUNCTION OVERLOADING

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

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

```

// Declarations
int add(int a, int b);           // prototype 1
int add(int a, int b, int c);    // prototype 2
double add(double x, double y);  // prototype 3
double add(int p, double q);     // prototype 4
double add(double p, int q);     // prototype 5

// Function calls
cout << add(5, 10);               // uses prototype 1
cout << add(15, 10.0);           // uses prototype 4
cout << add(12.5, 7.5);          // uses prototype 3
cout << add(5, 10, 15);          // uses prototype 2
cout << add(0.75, 5);            // uses prototype 5

```

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

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

```
char to int
float to double
```

to find a match.

3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

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

A function call such as

```
square(10)
```

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

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

Program 5.5 illustrates function overloading.

Program 5.5 Function Overloading

```
//Function area() is overloaded three times
#include<iostream>

//Declaration of Function Prototypes
int area (int);
int area (int, int);
float area (float);

int main()
{
    cout<<"Calling the area() function for computing the area of a
    square (side = 5) - "<<area(5)<<"\n";
    cout<<"Calling the area() function for computing the area of a
    rectangle (length = 5, breadth = 10) - "<<area(5,10)<<"\n";
    cout<<"Calling the area() function for computing the area of a
    circle (radius = 5.5) - "<<area(5.5);
```

(Contd.)

```
return 0;
}

int area (int side)                //Area of square
{
    return(side*side);
}

int area (int length, int breadth) //Area of rectangle
{
    return(length*breadth);
}

float area (float radius)          //Area of circle
{
    return(3.14*radius*radius);
}
```

The output of Program 5.5 would be

```
Calling the area() function for computing the area of a square
(side = 5) - 25
Calling the area() function for computing the area of a rectangle (length
= 5, breadth = 10) - 50
Calling the area() function for computing the area of a circle
(radius = 5.5) - 94.99
```

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

Overloaded functions are extensively used for handling class objects. They will be illustrated later when the classes are discussed in the next chapter.



5.11 FRIEND AND VIRTUAL FUNCTIONS

C++ introduces two new types of functions, namely, friend function and virtual function. They are basically introduced to handle some specific tasks related to class objects. Therefore, discussions on these functions have been reserved until after the class objects are discussed. The friend functions have been discussed in Sec. 4.15 and virtual functions in Sec. 9.5 of Chapter 9.



5.12 MATH LIBRARY FUNCTIONS

The standard C++ supports many math functions that can be used for performing certain commonly used calculations. Most frequently used math library functions are summarized in Table 5.1.

Table 5.1 Commonly used math library functions

Function	Purposes
ceil(x)	Rounds x to the smallest integer not less than x $\text{ceil}(8.1) = 9.0$ and $\text{ceil}(-8.8) = -8.0$
cos(x)	Trigonometric cosine of x (x in radians)
exp(x)	Exponential function e^x .
fabs(x)	Absolute value of x. If $x > 0$ then $\text{abs}(x)$ is x If $x = 0$ then $\text{abs}(x)$ is 0.0 If $x < 0$ then $\text{abs}(x)$ is $-x$
floor(x)	Rounds x to the largest integer not greater than x $\text{floor}(8.2) = 8.0$ and $\text{floor}(-8.8) = -9.0$
log(x)	Natural logarithm of x (base e)
log10(x)	Logarithm of x (base 10)
pow(x,y)	x raised to power y (x^y)
sin(x)	Trigonometric sine of x (x in radians)
sqrt(x)	Square root of x
tan(x)	Trigonometric tangent of x (x in radians)



Note The argument variables *x* and *y* are of type **double** and all the functions return the data type **double**.

To use the math library functions, we must include the header file **math.h** in conventional C++ and **cmath** in ANSI C++.

Program 5.6 demonstrates the use of math functions

Program 5.6 Use of Math Functions

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

using namespace std;

int main()
{
    cout<<fixed<<setprecision(2);
    cout<<"sin (100) = "<<sin(100.00)<<"\n"; //Computing sine value
    cout<<"cos (100) = "<<cos(100.00)<<"\n"; //Computing cosine
    value
    cout<<"tan (100) = "<<tan(100.00)<<"\n"; //Computing tangent
```

(Contd.)

```

value
cout<<"7 to the power of 6 = "<<pow(7.0,6.0)<<"\n"; //Computing
power of one value raised to the other
cout<<"log10 (10) = "<<log10(10.00)<<"\n"; //Computing log to
the base of 10 value
cout<<"sqrt (2) = "<<sqrt(2.00)<<"\n";      //Computing the
square root of a value

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

The output of Program 5.6 would be

```

sin (100) = -0.51
cos (100) = 0.86
tan (100) = -0.59
7 to the power of 6 = 117649.00
log10 (10) = 1.00
sqrt (2) = 1.41
```

SUMMARY

- ☐ It is possible to reduce the size of program by calling and using functions at different places in the program.
- ☐ In C++ the main() returns a value of type **int** to the operating system. Since the return type of functions is **int** by default, the keyword **int** in the main() header is optional. Most C++ compilers issue a warning, if there is no return statement.
- ☐ Function prototyping gives the compiler the details about the functions such as the number and types of arguments and the type of return values.
- ☐ Reference variables in C++ permit us to pass parameters to the functions by reference. A function can also return a reference to a variable.
- ☐ When a function is declared inline the compiler replaces the function call with the respective function code. Normally, a small size function is made as **inline**.
- ☐ The compiler may ignore the inline declaration if the function declaration is too long or too complicated and hence compile the function as a normal function.
- ☐ C++ allows us to assign default values to the function parameters when the function is declared. In such a case we can call a function without specifying all its arguments. The defaults are always added from right to left.
- ☐ In C++, an argument to a function can be declared as **const**, indicating that the function should not modify the argument.
- ☐ C++ allows function overloading. That is, we can have more than one function with the same name in our program. The compiler matches the function call with the exact function code by checking the number and type of the arguments.
- ☐ C++ supports two new types of functions, namely **friend** functions and **virtual** functions.
- ☐ Many mathematical computations can be carried out using the library functions supported by the C++ standard library.

Key Terms

actual arguments | argument list | bubble sort | call by reference | call by value | called function | calling program | calling statement | **cmath** | **const** arguments | declaration statement | default arguments | default values | dummy variables | ellipses | empty argument list | exit value | formal arguments | **friend** functions | function call | function definition | function overloading | function polymorphism | function prototype | indirection | inline | inline functions | macros | **main()** | math library | **math.h** | overloading | pointers | polymorphism | prototyping | reference variable | return by reference | **return** statement | return type | **return()** | template | virtual functions

Review Questions

5.1 State whether the following statements are TRUE or FALSE.

- (a) A function argument is a value returned by the function to the calling program.
- (b) When arguments are passed by value, the function works with the original arguments in the calling program.
- (c) When a function returns a value, the entire function call can be assigned to a variable.
- (d) A function can return a value by reference.
- (e) When an argument is passed by reference, a temporary variable is created in the calling program to hold the argument value.
- (f) It is not necessary to specify the variable name in the function prototype.

5.2 Describe the different styles of writing prototypes.

5.3 Find errors, if any, in the following function prototypes.

- (a) float average(x,y);
- (b) int mul(int a,b);
- (c) int display(...);
- (d) void Vect(int? &V, int & size);
- (e) void print(float data[], size = 20);

5.4 What is the main advantage of passing arguments by reference?

5.5 What is the most significant advantage that you see in using references instead of pointers?

5.6 When will you make a function **inline**? Why?

5.7 How does an **inline** function differ from a preprocessor macro?

5.8 When do we need to use default arguments in a function?

5.9 What is the significance of an empty parenthesis in a function declaration?

5.10 What do we mean by overloading of a function? Explain with an example when do we use this concept? **F A Q**

5.11 How do we achieve function overloading? On what basis, the compiler distinguishes between a set of overloaded functions having the same name?

5.12 Comment on the following function definitions:

```
(a) int *f( )
    {
        int m = 1;
        ....
        ....
        return(&m);
    }
```

```
(b) double f( )
    {
        ....
        ....
        return(1);
    }
```

```
(c) int & f()
    {
        int n = 10;
        ....
        ....
        return(n);
    }
```

Debugging Exercises

5.1 Identify the error in the following program.

```
#include <iostream.h>
int fun()
{
    return 1;
}
float fun()
{
    return 10.23;
}
void main()
{
    cout << (int)fun() << ' ';
    cout << (float)fun() << ' ';
}
```

5.2 Identify the error in the following program.

```
#include <iostream.h>
void display(const int const1=5)
{
    const int const2=5;
    int array1[const1];
    int array2[const2];
}
```

```

    for(int i=0; i<5; i++)
    {
        array1[i] = i;
        array2[i] = i*10;
        cout << array1[i] << ' ' << array2[i] << ' ';
    }
}

void main()
{
    display(5);
}

```

5.3 Identify the error in the following program.

```

#include <iostream.h>
int gValue=10;
void extra()
{
    cout << gValue << ' ';
}

void main()
{
    extra();
    {
        int gValue = 20;
        cout << gValue << ' ';
        cout << :: gValue << ' ';
    }
}

```

5.4 Find errors, if any, in the following function definition for displaying a matrix: void display (int A[][], int m, int n)

```

{
    for(i=0; i<m; i++)
    for(j=0; j<n; j++)
        cout << " " << A[i][j];
    cout << "\n";
}

```

Programming Exercises

- 5.1** Write a function to read a matrix of size m x n from the keyboard.
- 5.2** Write a program to read a matrix of size m x n from the keyboard and display the same on the screen using functions. W E B
- 5.3** Rewrite the program of Exercise 5.2 to make the row parameter of the matrix as a default argument. W E B
- 5.4** The effect of a default argument can be alternatively achieved by overloading. Discuss with an example.
- 5.5** Write a macro that obtains the largest of three numbers.

- 5.6** Redo Exercise 5.5 using inline function. Test the function using a **main** program.
- 5.7** Write a function `power()` to raise a number **m** to a power **n**. The function takes a **double** value for **m** and **int** value for **n**, and returns the result correctly. Use a default value of 2 for **n** to make the function to calculate squares when this argument is omitted. Write a **main** that gets the values of **m** and **n** from the user to test the function. `W E B`
- 5.8** Write a function that performs the same operation as that of Exercise 5.7 but takes an **int** value for **m**. Both the functions should have the same name. Write a **main** that calls both the functions. Use the concept of function overloading. `W E B`
- 5.9** Write a program to compute the area of a triangle and a circle by overloading the `area()` function. `W E B`

6

CONSTRUCTORS AND DESTRUCTORS

- 6.1 Introduction
- 6.2 Constructors
- 6.3 Parameterized Constructors
- 6.4 Multiple Constructors in a Class
- 6.5 Constructors with Default Arguments
- 6.6 Dynamic Initialization of Objects
- 6.7 Copy Constructor
- 6.8 Dynamic Constructors
- 6.9 Constructing Two-Dimensional Arrays
- 6.10 const Objects
- 6.11 Destructors

Summary

Key Terms

C H A P T E R O U T L I N E





6.1 INTRODUCTION

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as **putdata()** and **setvalue()** to provide initial values to the private member variables. For example, the following statement

```
A.input();
```

invokes the member function **input()**, which assigns the initial values to the data items of object **A**. Similarly, the statement

```
x.getdata(100,299.95);
```

passes the initial values as arguments to the function **getdata()**, where these values are assigned to the private variables of object **x**. All these 'function call' statements are used with the appropriate objects that have already been created. These functions cannot be used to initialize the member variables at the time of creation of their objects.

Providing the initial values as described above does not conform with the philosophy of C++ language. We stated earlier that one of the aims of C++ is to create user-defined data types such as **class**, that behave very similar to the built-in types. This means that we should be able to initialize a **class** type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

```
int m = 20;
float x = 5.75;
```

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. It is therefore clear that some more features of classes need to be explored that would enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects. It also provides another member function called the *destructor* that destroys the objects when they are no longer required.



6.2 CONSTRUCTORS

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
// class with a constructor

class integer
{
    int m, n;
public:
```

```

integer(void);                // constructor declared
.....
.....
};
integer :: integer(void)      // constructor defined
{
    m = 0; n = 0;
}

```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```

integer int1;                // object int1 created

```

not only creates the object **int1** of type **integer** but also initializes its data members **m** and **n** to zero. There is no need to write any statement to invoke the constructor function (as we do with the normal member functions). If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately. This would be very inconvenient, if there are a large number of objects.

A constructor that accepts no parameters is called the *default constructor*. The default constructor for **class A** is **A::A()**. If no such constructor is defined, then the compiler supplies a default constructor. Therefore a statement such as

```

A a;

```

invokes the default constructor of the compiler to create the object **a**.

The constructor functions have some special characteristics:

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**. (Meaning of virtual will be discussed later in Chapter 9.)
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the operators **new** and **delete** when memory allocation is required.

Remember, when a constructor is declared for a class, initialization of the class objects becomes mandatory.



6.3 PARAMETERIZED CONSTRUCTORS

The constructor **integer()**, defined above, initializes the data members of all the objects to zero. However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called *parameterized constructors*.

The constructor **integer()** may be modified to take arguments as shown below:

```
class integer
{
    int m, n;
    public:
        integer(int x, int y);    // parameterized constructor
        .....
        .....
};
integer :: integer(int x, int y)
{
    m = x; n = y;
}
```

When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer(0,100);    // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);    // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. Program 6.1 demonstrates the passing of arguments to the constructor functions.

The constructor functions can also be defined as **inline** functions. Example:

```
class integer
{
    int m, n;
    public:
        integer(int x, int y)    // Inline constructor
        {
            m = x; y = n;
        }
        .....
        .....
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class A
{
    .....
    .....
public:
    A(A);
};
```

is illegal.

However, a constructor can accept a *reference* to its own class as a parameter. Thus, the statement

```
Class A
{
    .....
    .....
public:
    A(A&);
};
```

is valid. In such cases, the constructor is called the *copy constructor*.

Program 6.1 Example of Parameterized Constructor

```
//This program defines a class called Point that stores the x and y
coordinates of a point. The class uses parameterized constructor for
initializing the class objects
```

```
#include <iostream>
```

```
class Point
{
```

```
    int x, y;
```

```
public:
```

```
    Point(int a, int b) //inline parameterized constructor
    definition
```

```
    {
```

```
        x=a;
```

```
        y=b;
```

```
    }
```

```
    void display()
```

```
    {
```

```
        cout<<"("<<x<<" "<<y<<")\n";
```

```
    }
```

```
};
```

```
int main()
```

(Contd.)

```

{
    Point p1(1,1); //invokes parameterized constructor
    Point p2(5,10);
    cout<<"Point p1 = ";
    p1.display();
    cout<<"Point p2 = ";
    p2.display();
    return 0;
}

```

The output of Program 6.1 would be

```

Point p1 = (1,1)
Point p2 = (5,10)

```



6.4 MULTIPLE CONSTRUCTORS IN A CLASS

So far we have used two kinds of constructors. They are

```

integer();                // No arguments
integer(int, int);        // Two arguments

```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from **main()**. C++ permits us to use both these constructors in the same class. For example, we could define a class as follows:

```

class integer
{
    int m, n;
public:
    integer(){m=0; n=0;}           // constructor 1
    integer(int a, int b)         // constructor 2
    {m = a; n = b;}
    integer(integer & i)          // constructor 3
    {m = i.m; n = i.n;}
};

```

This declares three constructors for an **integer** object. The first constructor receives no arguments, the second receives two **integer** arguments and the third receives one integer object as an argument. For example, the declaration

```
integer I1;
```

would automatically invoke the first constructor and set both **m** and **n** of **I1** to zero. The statement

```
integer I2(20,40);
```

would call the second constructor which will initialize the data members **m** and **n** of **I2** to 20 and 40 respectively. Finally, the statement

```
integer I3(I2);
```

would invoke the third constructor which copies the values of **I2** into **I3**. In other words, it sets the value of every data element of **I3** to the value of the corresponding data element of **I2**. As mentioned earlier, such a constructor is called the *copy constructor*. We learned in Chapter 4 that the process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class, we say that the constructor is overloaded.

Program 6.2 shows the use of overloaded constructors.

Program 6.2 Overloaded Constructors

```
#include <iostream>

using namespace std;

class complex
{
    float x, y;
public:
    complex(){ } // constructor no arg
    complex(float a) {x = y = a;} // constructor-one arg
    complex(float real, float imag) // constructor-two args
    {x = real; y = imag;}

    friend complex sum(complex, complex);
    friend void show(complex);
};

complex sum(complex c1, complex c2) // friend
{
    complex c3;
    c3.x = c1.x + c2.x;
    c3.y = c1.y + c2.y;
    return(c3);
}

void show(complex c) // friend
{
    cout << c.x << " + j" << c.y << "\n";
}

int main()
{
    complex A(2.7, 3.5); // define & initialize
    complex B(1.6); // define & initialize
    complex C; // define

    C = sum(A, B); // sum() is a friend
    cout << "A = "; show(A); // show() is also friend
    cout << "B = "; show(B);
    cout << "C = "; show(C);
```

(Contd.)


```
// Another way to give initial values (second method)
    complex P,Q,R;           // define P, Q and R
    P = complex(2.5,3.9);    // initialize P
    Q = complex(1.6,2.5);    // initialize Q
    R = sum(P,Q);

    cout << "\n";
    cout << "P = "; show(P);
    cout << "Q = "; show(Q);
    cout << "R = "; show(R);

    return 0;
}
```

The output of Program 6.2 would be

```
A = 2.7 + j3.5
B = 1.6 + j1.6
C = 4.3 + j5.1

P = 2.5 + j3.9
Q = 1.6 + j2.5
R = 4.1 + j6.4
```



Note

There are three constructors in the class **complex**. The first constructor, which takes no arguments, is used to create objects which are not initialized; the second, which takes one argument, is used to create objects and initialize them; and the third, which takes two arguments, is also used to create objects and initialize them to specific values. Note that the second method of initializing values looks better.

Let us look at the first constructor again.

```
complex() { }
```

It contains the empty body and does not do anything. We just stated that this is used to create objects without any initial values. Remember, we have defined objects in the earlier examples without using such a constructor. Why do we need this constructor now? As pointed out earlier, C++ compiler has an *implicit constructor* which creates objects, even though it was not defined in the class.

This works fine as long as we do not use any other constructors in the class. However, once we define a constructor, we must also define the “do-nothing” implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.



6.5 CONSTRUCTORS WITH DEFAULT ARGUMENTS

It is possible to define constructors with default arguments. For example, the constructor `complex()` can be declared as follows:

```
complex(float real, float imag=0);
```

The default value of the argument **imag** is zero. Then, the statement

```
complex C(5.0);
```

assigns the value 5.0 to the **real** variable and 0.0 to **imag** (by default). However, the statement

```
complex C(2.0, 3.0);
```

assigns 2.0 to **real** and 3.0 to **imag**. The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor **A::A()** and the default argument constructor **A::A(int = 0)**. The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

```
A a;
```

The ambiguity is whether to 'call' **A::A()** or **A::A(int = 0)**.



6.6 DYNAMIC INITIALIZATION OF OBJECTS

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment. Program 6.3 illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

Program 6.3 Dynamic Initialization of Objects

```
// Long-term fixed deposit system
#include <iostream>
using namespace std;
class Fixed_deposit
{
    long int P_amount;    // Principal amount
    int     Years;        // Period of investment
    float   Rate;         // Interest rate
    float   R_value;      // Return value of amount
public:
    Fixed_deposit(){ }
    Fixed_deposit(long int p, int y, float r=0.12);
    Fixed_deposit(long int p, int y, int r);
    void display(void);
};

Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
```

(Contd.)

```

        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for(int i = 1; i <= y; i++)
            R_value = R_value * (1.0 + r);
    }

Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
    P_amount = p;
    Years = y;
    Rate = r;
    R_value = P_amount;

    for(int i=1; i<=y; i++)
        R_value = R_value*(1.0+float(r)/100);
}

void Fixed_deposit :: display(void)
{
    cout << "\n"
        << "Principal Amount = " << P_amount << "\n"
        << "Return Value      = " << R_value << "\n";
}

int main()
{
    Fixed_deposit FD1, FD2, FD3;           // deposits created

    long int p;                           // principal amount

    int    y;                             // investment period, years
    float  r;                             // interest rate, decimal form
    int    R;                             // interest rate, percent form

    cout << "Enter amount, period, interest rate(in percent)" << "\n";
    cin >> p >> y >> R;
    FD1 = Fixed_deposit(p,y,R);

    cout << "Enter amount, period, interest rate(decimal form)" << "\n";
    cin >> p >> y >> r;
    FD2 = Fixed_deposit(p,y,r);

    cout << "Enter amount and period" << "\n";
    cin >> p >> y;
    FD3 = Fixed_deposit(p,y);

    cout << "\nDeposit 1";
    FD1.display();

    cout << "\nDeposit 2";
    FD2.display();

    cout << "\nDeposit 3";
    FD3.display();

    return 0;
}

```

The output of Program 6.3 would be

```
Enter amount,period,interest rate(in percent)
10000 3 18
Enter amount,period,interest rate(in decimal form)
10000 3 0.18
Enter amount and period
10000 3

Deposit 1
Principal Amount = 10000
Return Value      = 16430.3

Deposit 2
Principal Amount = 10000
Return Value      = 16430.3

Deposit 3
Principal Amount = 10000
Return Value      = 14049.3
```

The program uses three overloaded constructors. The parameter values to these constructors are provided at run time. The user can provide input in one of the following forms:

1. Amount, period and interest in decimal form.
2. Amount, period and interest in percent form.
3. Amount and period.



Note Since the constructors are overloaded with the appropriate parameters, the one that matches the input values is invoked. For example, the second constructor is invoked for the forms (1) and (3), and the third is invoked for the form (2). Note that, for form (3), the constructor with default argument is used. Since input to the third parameter is missing, it uses the default value for *r*.



6.7 COPY CONSTRUCTOR

We briefly mentioned about the copy constructor in Sec. 6.3. We used the copy constructor

```
integer(integer &i);
```

in Sec. 6.4 as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare and initialize an object from another object. For example, the statement

```
integer I2(I1);
```

would define the object I2 and at the same time initialize it to the values of I1. Another form of this statement is

```
integer I2 = I1;
```

The process of initializing through a copy constructor is known as *copy initialization*. Remember, the statement

```
I2 = I1;
```

will not invoke the copy constructor. However, if **I1** and **I2** are objects, this statement is legal and simply assigns the values of **I1** to **I2**, member-by-member. This is the task of the overloaded assignment operator(=). We shall see more about this later.

A copy constructor takes a reference to an object of the same class as itself as an argument. Let us consider a simple example of constructing and using a copy constructor as shown in Program 6.4.

Program 6.4 Copy Constructor

```
#include <iostream>

using namespace std;

class code
{
    int id;
public:
    code(){ }           // constructor
    code(int a) { id = a;} // constructor again
    code(code & x)       // copy constructor
    {
        id = x.id;      // copy in the value
    }
    void display(void)
    {
        cout << id;
    }
};

int main()
{
    code A(100);        // object A is created and initialized
    code B(A);          // copy constructor called
    code C = A;         // copy constructor called again

    code D;             // D is created, not initialized
    D = A;              // copy constructor not called

    cout << "\n id of A: "; A.display();
    cout << "\n id of B: "; B.display();
    cout << "\n id of C: "; C.display();
    cout << "\n id of D: "; D.display();

    return 0;
}
```

The output of Program 6.4 would be

```
id   of A: 100
id   of B: 100
id   of C: 100
id   of D: 100
```



Note *A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor.*

When no copy constructor is defined, the compiler supplies its own copy constructor.



6.8 DYNAMIC CONSTRUCTORS

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator.

Program 6.5 Constructors with new

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

using namespace std;

class String
{
    char *name;
    int length;
public:
    String()                                // constructor-1
    {
        length = 0;
        name = new char[length + 1];
    }
    String(char *s)                        // constructor-2
    {
        length = strlen(s);
        name = new char[length + 1];      // one additional
                                            // character for \0

        strcpy(name, s);
    }

    void display(void)
    {cout << name << "\n";}
    void join(String &a, String &b);
};
```

(Contd.)

```

void String :: join(String &a, String &b)
{
    length = a.length + b.length;
    delete name;
    name = new char[length+1]; // dynamic allocation

    strcpy(name, a.name);
    strcat(name, b.name);
};

int main()
{
    char *first = "Joseph ";
    String name1(first), name2("Louis "), name3 ("Lagrange"),
        s1,s2;

    s1.join(name1, name2);
    s2.join(s1, name3);
    name1.display();
    name2.display();
    name3.display();
    s1.display();
    s2.display();
    return 0;
}

```

The output of Program 6.5 would be

```

Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrange

```

Note This Program uses two constructors. The first is an empty constructor that allows us to declare an array of strings. The second constructor initializes the **length** of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end-of-string character '\0'.

The member function **join()** concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string functions **strcpy()** and **strcat()**. Note that in the function **join()**, **length** and **name** are members of the object that calls the function, while **a.length** and **a.name** are members of the argument object **a**. The **main()** function program concatenates three strings into one string. The output is as shown below:

```
Joseph Louis Lagrange
```



6.9 CONSTRUCTING TWO-DIMENSIONAL ARRAYS

We can construct matrix variables using the class type objects. The example in Program 6.6 illustrates how to construct a matrix of size $m \times n$.

Program 6.6 Constructing Matrix Objects

```
#include <iostream>

using namespace std;

class matrix
{
    int **p; // pointer to matrix
    int d1,d2; // dimensions
public:
    matrix(int x, int y);
    void get_element(int i, int j, int value)
    {p[i][j]=value;}
    int & put_element(int i, int j)
    {return p[i][j];}
};

matrix :: matrix(int x, int y)
{
    d1 = x;
    d2 = y;
    p = new int *[d1]; // creates an array pointer
    for(int i = 0; i < d1; i++)
        p[i] = new int[d2]; // creates space for each row
}

int main()
{
    int m, n;

    cout << "Enter size of matrix: ";
    cin >> m >> n;
    matrix A(m,n); // matrix object A constructed

    cout << "Enter matrix elements row by row \n";
    int i, j, value;

    for(i = 0; i < m; i++)
        for(j = 0; j < n; j++)
        {
            cin >> value;
            A.get_element(i,j,value);
        }
    cout << "\n";
    cout << A.put_element(1,2);

    return 0;
};
```

The output of Program 6.6 would be

```
Enter size of matrix: 3 4
Enter matrix elements row by row
```

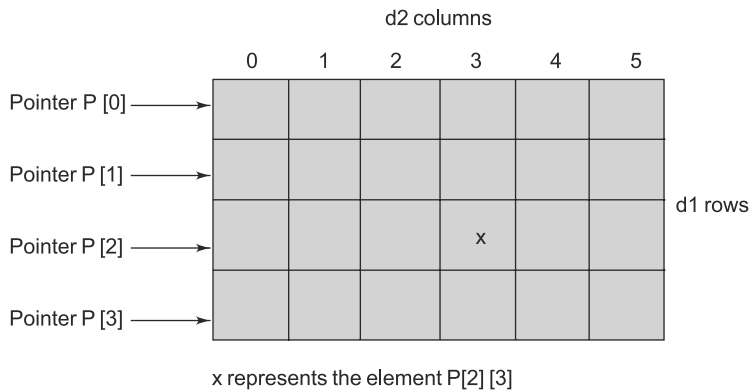


```

11 12 13 14
15 16 17 18
19 20 21 22
17

```

17 is the value of the element (1, 2).



The constructor first creates a vector pointer to an **int** of size **d1**. Then, it allocates, iteratively an **int** type vector of size **d2** pointed at by each element **p[i]**. Thus, space for the elements of a **d1 x d2** matrix is allocated from free store as shown above.



6.10 const OBJECTS

We may create and use constant objects using **const** keyword before object declaration. For example, we may create **X** as a constant object of the class **matrix** as follows:

```
const matrix X(m,n); // object X is constant
```

Any attempt to modify the values of **m** and **n** will generate compile-time error. Further, a constant object can call only **const** member functions. As we know, a **const** member is a function prototype or function definition where the keyword **const** appears after the function's signature.

Whenever **const** objects try to invoke **nonconst** member functions, the compiler generates errors.



6.11 DESTRUCTORS

A *destructor*, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor for the class **integer** can be defined as shown below:

```
~integer() { }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that

is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory. For example, the destructor for the **matrix** class discussed above may be defined as follows:

```
matrix :: ~matrix()
{
    for(int i=0; i<d1; i++)
        delete p[i];
    delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

The example below illustrates that the destructor has been invoked implicitly by the compiler.

Program 6.7 Implementation of Destructors

```
#include<iostream>
using namespace std;
int count=0;
class test
{
public:
    test()
    {
        count++;
        cout<<"\n\nConstructor Msg: Object number "<<count<<
        "created..";
    }
    ~test()
    {
        cout<<"\n\nDestructor Msg: Object number "<<count<<
        "destroyed..";
        count--;
    }
};

int main()
{
    cout<<"Inside the main block..";
    cout<<"\n\nCreating first object T1..";

    test T1;

    { //Block 1
```

(Contd.)

```

        cout<<"\n\nInside Block 1..";
        cout<<"\n\nCreating two more objects T2 and T3..";
        test T2,T3;
        cout<<"\n\nLeaving Block 1..";
    }

    cout<<"\n\nBack inside the main block..";
    return 0;
}

```

The output of Program 6.7 would be

```

Inside the main block..
Creating first object T1..
Constructor Msg: Object number 1 created..
Inside Block 1..
Creating two more objects T2 and T3..
Constructor Msg: Object number 2 created..
Constructor Msg: Object number 3 created..
Leaving Block 1..
Destructor Msg: Object number 3 destroyed.
Destructor Msg: Object number 2 destroyed.
Back inside the main block..
Destructor Msg: Object number 1 destroyed..

```



Note

A class constructor is called everytime an object is created. Similarly, as the program control leaves the current block the objects in the block start getting destroyed and destructors are called for each one of them. Note that the objects are destroyed in the reverse order of their creation. Finally, when the main block is exited, destructors are called corresponding to the remaining objects present inside main.

Similar functionality as depicted in Program 6.7 can be attained by using static data members with constructors and destructors. We can declare a static integer variable count inside a class to keep a track of the number of its object instantiations. Being static, the variable will be initialized only once, i.e., when the first object instance is created. During all subsequent object creations, the constructor will increment the count variable by one. Similarly, the destructor will decrement the count variable by one as and when an object gets destroyed. To realize this scenario, the code in Program 6.7 will change slightly, as shown below:

```

#include <iostream>
using namespace std;

class test
{
private:
    static int count=0;
public:
    .
    .

```

```
}  
test ()  
{  
.  
count++;  
}  
  
~test ()  
{  
.  
count--;  
}
```

The primary use of destructors is in freeing up the memory reserved by the object before it gets destroyed. Program 6.8 demonstrates how a destructor releases the memory allocated to an object:

Program 6.8 Memory Allocation to an Object using Destructor

```
#include<iostream>  
#include<conio.h>  
using namespace std;  
class test  
{  
    int *a;  
public:  
    test(int size)  
    {  
        a = new int[size];  
        cout<<"\n\nConstructor Msg: Integer array of size "<<size<<"  
        created..";  
    }  
    ~test()  
    {  
        delete a;  
        cout<<"\n\nDestructor Msg: Freed up the memory allocated for  
        integer array";  
    }  
};  
int main()  
{  
    int s;  
    cout<<"Enter the size of the array..";  
    cin>>s;  
    cout<<"\n\nCreating an object of test class..";  
    test T(s);  
    cout<<"\n\nPress any key to end the program..";  
    getch();  
  
    return 0;  
}
```

The output of program 6.8 would be

```
Enter the size of the array..5
Creating an object of test class..
Constructor Msg: Integer array of size 5 created..
Press any key to end the program..
Destructor Msg: Freed up the memory allocated for integer array
```

SUMMARY

- ☐ C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects.
- ☐ A constructor has the same name as that of a class.
- ☐ Constructors are normally used to initialize variables and to allocate memory.
- ☐ Similar to normal functions, constructors may be overloaded.
- ☐ When an object is created and initialized at the same time, a copy constructor gets called.
- ☐ We may make an object **const** if it does not modify any of its data values.
- ☐ C++ also provides another member function called the destructor that destroys the objects when they are no longer required.

Key Terms

automatic initialization | **Const** | Constructor | constructor overloading | copy constructor | copy initialization | default argument | default constructor | **Delete** | Destructor | dynamic construction | dynamic initialization | explicit call | implicit call | implicit constructor | initialization | **new** | parameterized constructor | reference | shorthand method | **strcat()** | **strcpy()** | **strlen()** | **virtual**

Review Questions

- 6.1 What is a constructor? Is it mandatory to use constructors in a class?
- 6.2 How do we invoke a constructor function?
- 6.3 List some of the special properties of the constructor functions.
- 6.4 What is a parameterized constructor?
- 6.5 Can we have more than one constructors in a class? If yes, explain the need for such a situation.
- 6.6 What do you mean by dynamic initialization of objects? Why do we need to do this?
- 6.7 How is dynamic initialization of objects achieved?
- 6.8 Explain the function of default constructor. **F A Q**
- 6.9 Distinguish between the following two statements:

```
time T2(T1);
time T2 = T1;
```

T1 and T2 are objects of **time** class.

6.10 Describe the importance of destructor. Explain its use with the help of an example. F A Q

6.11 State whether the following statements are TRUE or FALSE.

- (a) Constructors, like other member functions, can be declared anywhere in the class.
- (b) Constructors do not return any values.
- (c) A constructor that accepts no parameter is known as the default constructor.
- (d) A class should have at least one constructor.
- (e) Destructors never take any argument.
- (f) The objects are always destroyed in the reverse order of their creation.

Debugging Exercises

6.1 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
    int length;
    int width;
public:
    Room(int l, int w=0):
        width(w),
        length(l)
    {
    }
};
void main()
{
    Room objRoom1;
    Room objRoom2(12, 8);
}
```

6.2 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
    int length;
    int width;
public:
    Room()
    {
        length = 0;
        width = 0;
    }
    Room(int value=8)
    {
        length = width = 8;
    }
}
```

```
void display()
{
    cout << length << ' ' << width;
}

};

void main()
{
    Room objRoom1;
    objRoom1.display();
}
```

6.3 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
    int width;
    int height;
    static int copyConsCount;
public:
    void Room()
    {
        width = 12;
        height = 8;
    }

    Room(Room& r)
    {
        width = r.width;
        height = r.height;
        copyConsCount++;
    }

    void dispCopyConsCount()
    {
        cout << copyConsCount;
    }
};

int Room::copyConsCount = 0;

void main()
{
    Room objRoom1;
    Room objRoom2(objRoom1);
    Room objRoom3 = objRoom1;
    Room objRoom4;
    objRoom4 = objRoom3;
    objRoom4.dispCopyConsCount();
}
```

6.4 Identify the error in the following program.

```

#include <iostream.h>

class Room
{
    int width;
    int height;
    static int copyConsCount;
public:
    Room()
    {
        width = 12;
        height = 8;
    }
    Room(Room& r)
    {
        width = r.width;
        height = r. height;
        copyConsCount++;
    }

    void disCopyConsCount()
    {
        cout << copyConsCount;
    }
};

int Room::copyConsCount = 0;

void main()
{
    Room objRoom1;
    Room objRoom2 (objRoom1);
    Room objRoom3 = objRoom1;
    Room objRoom4;
    objRoom4 = objRoom3;

    objRoom4.dispCopyConsCount();
}

```

Programming Exercises

6.1 Design constructors for the classes designed in Programming Exercises 5.1 through 5.5 of Chapter 5.

6.2 Define a class **String** that could work as a user-defined string type. Include constructors that will enable us to create an uninitialized string

```
String s1; // string with length 0
```

and also to initialize an object with a string constant at the time of creation like

```
String s2("Well done!");
```


Include a function that adds two strings to make a third string. Note that the statement

```
s2 = s1;
```

will be perfectly reasonable expression to copy one string to another.

Write a complete program to test your class to see that it does the following tasks:

- (a) Creates uninitialized string objects.
- (b) Creates objects with string constants.
- (c) Concatenates two strings properly.
- (d) Displays a desired string object. **W E B**

- 6.3** A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise the message "Required copies not in stock" is displayed. **W E B**

Design a system using a class called **books** with suitable member functions and constructors. Use new operator in constructors to allocate memory space required.

- 6.4** Improve the system design in Exercise 6.3 to incorporate the following features:
- (a) The price of the books should be updated as and when required. Use a private member function to implement this.
 - (b) The stock value of each book should be automatically updated as soon as a transaction is completed.
 - (c) The number of successful and unsuccessful transactions should be recorded for the purpose of statistical analysis. Use **static** data members to keep count of transactions.
- 6.5** Modify the program of Exercise 6.4 to demonstrate the use of pointers to access the members.

7

OPERATOR OVERLOADING AND TYPE CONVERSIONS

- 7.1 Introduction
- 7.2 Defining Operator Overloading
- 7.3 Overloading Unary Operators
- 7.4 Overloading Binary Operators
- 7.5 Overloading Binary Operators Using Friends
- 7.6 Manipulation of Strings Using Operators
- 7.7 Some Other Operator Overloading Examples
 - 7.7.1 Overloading the Subscript Operator []
 - 7.7.2 Overloading the Pointer-to-member (->) Operator
- 7.8 Rules for Overloading Operators
- 7.9 Type Conversions
 - 7.9.1 Basic to Class Type
 - 7.9.2 Class to Basic Type
 - 7.9.3 One Class to Another Class Type
 - 7.9.4 A Data Conversion Example

Summary

Key Terms

C H A P T E R O U T L I N E





7.1 INTRODUCTION

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

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can overload (give additional meaning to) all the C++ operators except the following:

- Class member access operators (`.`, `.*`).
- Scope resolution operator (`::`).
- Size operator (**`sizeof`**).
- Conditional operator (`?:`).

The reason why we cannot overload these operators may be attributed to the fact that these operators take names (example class name) as their operand instead of values, as is the case with other normal operators.

Although the *semantics* of an operator can be extended, we cannot change its *syntax*, the grammatical rules that govern its use such as the number of operands, precedence and associativity. For example, the multiplication operator will enjoy higher precedence than the addition operator. Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator `+`, which has been overloaded to add two vectors, can still be used to add two integers.



7.2 DEFINING OPERATOR OVERLOADING

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

```
return type classname :: operator op(arglist)
{
    Function body // task defined
}
```

where *return type* is the type of value returned by the specified operation and *op* is the operator being overloaded. **operator** *op* is the function name, where **operator** is a keyword.

Operator functions must be either member functions (nonstatic) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with **friend** functions. Arguments may be

passed either by value or by reference. Operator functions are declared in the class using prototypes as follows:

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

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

The process of overloading involves the following steps:

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

Overloaded operator functions can be invoked by expressions such as

op x or x op

for unary operators and

x op y

for binary operators. *op x* (or *x op*) would be interpreted as

operator op (x)

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

x.operator op (y)

in case of member functions, or

operator op (x,y)

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



7.3 OVERLOADING UNARY OPERATORS

Let us consider the unary minus operator. A minus operator when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each of its data items.

Program 7.1 shows how the unary minus operator is overloaded.

Program 7.1 Overloading Unary Minus

```
#include <iostream>

using namespace std;

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

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

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

void space :: operator-()
{
    x = -x;
    y = -y;
    z = -z;
}

int main()
{
    space S;
    S.getdata(10, -20, 30);
    cout << "S : ";
    S.display();

    -S;                               // activates operator-() function
    cout << "-S : ";
    S.display();

    return 0;
}
```

The output of Program 7.1 would be

```
S : x = 10 y = -20 z = 30
-S : x = -10 y = 20 z = -30
```



Note

The function **operator-**() takes no argument. Then, what does this operator function do? It changes the sign of data members of the object *S*. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Remember, a statement like

```
S2 = -S1;
```

will not work because, the function **operator-**() does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a friend function as follows:

```
friend void operator-(space &s);    // declaration
void operator-(space &s)           // definition
{
    s.x = -s.x;
    s.y = -s.y;
    s.z = -s.z;
}
```



Note

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to **operator-**(). Therefore, the changes made inside the operator function will not reflect in the called object.



7.4 OVERLOADING BINARY OPERATORS

We have just seen how to overload an unary operator. The same mechanism can be used to overload a binary operator. In Chapter 6, we illustrated, how to add two complex numbers using a friend function. A statement like

```
C = sum(A, B); // functional notation.
```

was used. The functional notation can be replaced by a natural looking expression

```
C = A + B; // arithmetic notation
```

by overloading the + operator using an **operator+()** function. Program 7.2 illustrates how this is accomplished.

Program 7.2 Overloading + Operator

```
#include <iostream>
using namespace std;

class complex
{
    float x;                // real part
    float y;                // imaginary part
public:
    complex(){ }            // constructor 1
    complex(float real, float imag) // constructor 2
    { x = real; y = imag; }
    complex operator+(complex);
    void display(void);
};

complex complex :: operator+(complex c)
{
    complex temp;           // temporary
    temp.x = x + c.x;       // these are
    temp.y = y + c.y;       // float additions
    return(temp);
}

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

int main()
{
    complex C1, C2, C3;     // invokes constructor 1
    C1 = complex(2.5, 3.5); // invokes constructor 2
    C2 = complex(1.6, 2.7);
    C3 = C1 + C2;

    cout << "C1 = "; C1.display();
    cout << "C2 = "; C2.display();
    cout << "C3 = "; C3.display();

    return 0;
}
```

The output of Program 7.2 would be

```
C1 = 2.5 + j3.5
C2 = 1.6 + j2.7
C3 = 4.1 + j6.2
```

**Note**

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

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

We should note the following features of this function:

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

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from? Now let us look at the statement that invokes this function:

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

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

```
C3 = C1.operator+(C2);  // usual function call syntax
```

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

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

c.x refers to the object **C2** and **x** refers to the object **C1**. **temp.x** is the real part of **temp** that has been created specially to hold the results of addition of **C1** and **C2**. The function returns the complex **temp** to be assigned to **C3**. Figure 7.1 shows how this is implemented.

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

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

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

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object. Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another object. Using *temporary objects* can make the code shorter, more efficient and better to read.

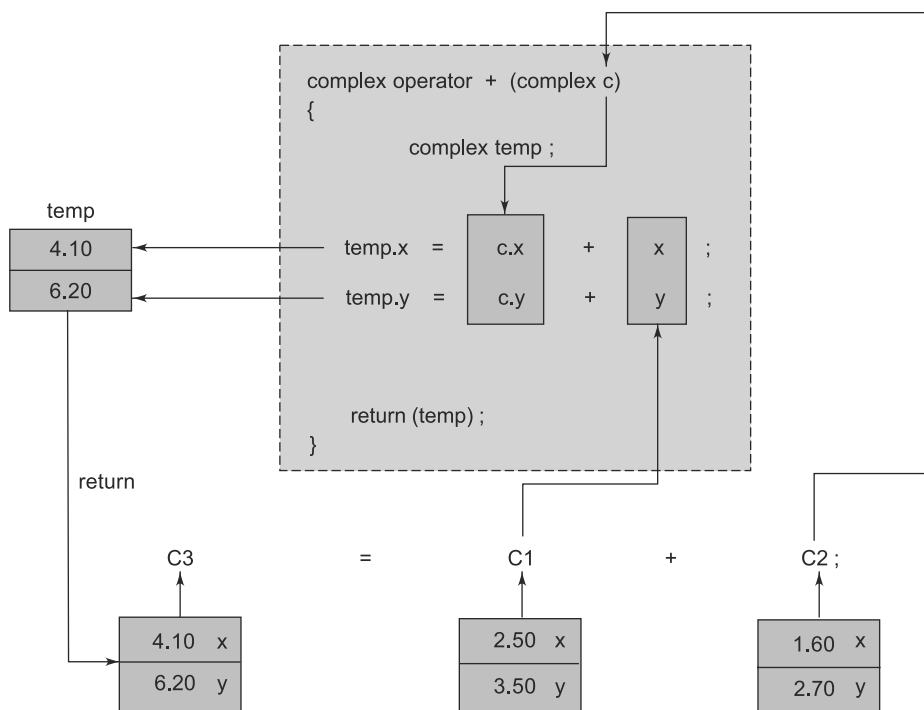


Fig. 7.1 Implementation of the overloaded `+` operator



7.5 OVERLOADING BINARY OPERATORS USING FRIENDS

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

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

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

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

2. Redefine the operator function as follows:

```

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

```

In this case, the statement

```
C3 = C1 + C2;
```

is equivalent to

```
C3 = operator+(C1, C2);
```

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

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

where **A** and **B** are objects of the same class. This will work for a member function but the statement

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

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

It may be recalled that an object need not be used to invoke a **friend** function but can be passed as an argument. Thus, we can use a friend function with a built-in type data as the *left-hand* operand and an object as the *right-hand* operand. Program 7.3 illustrates this, using scalar *multiplication* of a vector. It also shows how to overload the input and output operators >> and <<.

Program 7.3 Overloading Operators using Friends

```
#include <iostream.h>

const size = 3;
class vector
{
    int v[size];
public:
    vector();           // constructs null vector
    vector(int *x);     // constructs vector from array
    friend vector operator *(int a, vector b); // friend 1
    friend vector operator *(vector b, int a); // friend 2
    friend istream & operator >> (istream &, vector &);
    friend ostream & operator << (ostream &, vector &);
};

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

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

vector operator *(int a, vector b)
{
```

(Contd.)

```

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

vector operator *(vector b, int a)
{
    vector c;

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

istream & operator >> (istream &din, vector &b)
{
    for(int i=0; i<size; i++)
        din >> b.v[i];
    return(din);
}

ostream & operator << (ostream &dout, vector &b)
{
    dout << "(" << b.v [0];
    for(int i=1; i<size; i++)
        dout << ", " << b.v[i];
    dout << ")";
    return(dout);
}

int x[size] = {2,4,6};

int main()
{
    vector m;                                // invokes constructor 1
    vector n = x;                            // invokes constructor 2

    cout << "Enter elements of vector m " << "\n";
    cin >> m;                                // invokes operator>>() function
    cout << "\n";
    cout << "m = " << m << "\n";           // invokes operator <<()

    vector p, q;

    p = 2 * m;                               // invokes friend 1
    q = n * 2;                               // invokes friend 2

    cout << "\n";
    cout << "p = " << p << "\n";           // invokes operator<<()
    cout << "q = " << q << "\n";

    return 0;
}

```

The output of Program 7.3 would be

```
Enter elements of vector m
5 10 15

m = (5, 10, 15)
p = (10, 20, 30)
q = (4, 8, 12)
```

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

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

The program and its output are largely self-explanatory. The first constructor

```
vector();
```

constructs a vector whose elements are all zero. Thus,

```
vector m;
```

creates a vector `m` and initializes all its elements to 0. The second constructor

```
vector(int &x);
```

creates a vector and copies the elements pointed to by the pointer argument `x` into it. Therefore, the statements

```
int x[3] = {2, 4, 6};
vector n = x;
```

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



Note

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

```
friend istream & operator>>(istream &, vector &);
friend ostream & operator<<(ostream &, vector &);
```

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



7.6 MANIPULATION OF STRINGS USING OPERATORS

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

Although these limitations exist in C++ as well, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers. (Recently, ANSI C++ committee has added a new class called **string** to the C++ class library that supports all kinds of string manipulations. String manipulations using the **string** class are discussed in Chapter 15.

For example, we shall be able to use statements like

```
string3 = string1 + string2;
if(string1 >= string2) string = string1;
```

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

```
class string
{
    char *p;                // pointer to string
    int len;                // length of string
public:
    .....                  // member functions
    .....                  // to initialize and
    .....                  // manipulate strings
};
```

We shall consider an example to illustrate the application of overloaded operators to strings. The example shown in Program 7.4 overloads two operators, + and <= just to show how they are implemented. This can be extended to cover other operators as well.

Program 7.4 Mathematical Operations on Strings

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

class string
{
    char *p;
    int len;
public:
    string() {len = 0; p = 0;} // create null string
    string(const char * s);    // create string from arrays
    string(const string & s);  // copy constructor
    ~ string(){delete p;}      // destructor

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

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

(Contd.)

```

        friend void show(const string s);
};
string :: string(const char *s)
{
    len = strlen(s);
    p = new char[len+1];
    strcpy(p,s);
}

string :: string(const string & s)
{
    len = s.len;
    p = new char[len+1];
    strcpy(p,s.p);
}

// overloading + operator
string operator+(const string &s, const string &t)
{
    string temp;
    temp.len = s.len + t.len;
    temp.p = new char[temp.len+1];
    strcpy(temp.p,s.p);
    strcat(temp.p,t.p);
    return(temp);
}

// overloading <= operator
int operator<=(const string &s, const string &t)
{
    int m = strlen(s.p);
    int n = strlen(t.p);

    if(m <= n) return(1);

    else return(0);
}

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

int main()
{
    string s1 = "New ";
    string s2 = "York";
    string s3 = "Delhi";
    string string1, string2, string3;
    string1 = s1;
    string2 = s2;
    string3 = s1+s3;

    cout <<  "\nstring1 = "; show(string1);
    cout <<  "\nstring2 = "; show(string2);

```

(Contd.)

```

    cout <<  "\n";
    cout <<  "\nstring3 = "; show(string3);
    cout <<  "\n\n";
    if(string1 <= string3)
    {
        show(string1);
        cout << " smaller than ";
        show(string3);
        cout << "\n";
    }
    else
    {
        show(string3);
        cout << " smaller than ";
        show(string1);
        cout << "\n";
    }

    return 0;
}

```

The output of Program 7.4 would be

```

string1 = New
string2 = York

string3 = New Delhi

New smaller than New Delhi

```



7.7 SOME OTHER OPERATOR OVERLOADING EXAMPLES

7.7.1 Overloading the Subscript Operator []

The subscript operator is normally used to access and modify a specific element in an array. Program 7.5 demonstrates the overloading of the subscript operator to customize its behavior:

Program 7.5 Overloading of the Subscript Operator

```

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

using namespace std;

class arr
{
    int a[5];
public:
    arr(int *s)
    {

```

(Contd.)

```

        int i;
        for(i=0;i<5;i++)
            a[i]=s[i];
    }
    int operator [] (int k) //Overloading the subscript operator
    {
        return (a[k]);
    }
};

int main()
{
    int x[5] = {1,2,3,4,5};
    arr A(x);
    int i;
    for(i=0;i<5;i++)
    {
        cout<<x[i]<<"\t"; //Using subscript operator to access the
        private array elements
    }

    getch();
    return 0;
}

```

The output of Program 7.5 would be

```
1 2 3 4 5
```

As can be seen in the above program, we have used the subscript operator along with the object name to access the private array elements of the object.

7.7.2 Overloading the Pointer-to-member (->) Operator

The pointer-to-member operator (->) is normally used in conjunction with an object pointer to access any of the object's members. Program 7.6 demonstrates the overloading of the -> operator:

Program 7.6 Overloading of Pointer-to-member Operator

```

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

using namespace std;

class test
{
public:
    int num;
    test(int j)
    {

```

(Contd.)


```

        num=j;
    }

    test* operator -(void)
    {
        return this;
    }
};

int main()
{
    test T(5);
    test *Ptr = &T;

    cout<<"T.num = "<<T.num;           //Accessing num normally
    cout<<"\nPtr->num = "<<Ptr->num; //Accessing num using normal
                                     object pointer
    cout<<"\nT->num = "<<T->num;       //Accessing num using
                                     overloaded -> operator

    getch();
    return 0;
}

```

The output of Program 7.6 would be

```

T.num = 5
Ptr->num = 5
T->num = 5

```

The above program demonstrates both the normal (Ptr->num) as well as the overloaded (T->num) behaviour of the -> operator.

The statement,

```
return this;
```

returns a pointer to itself; that is a pointer to the test class object.



7.8 RULES FOR OVERLOADING OPERATORS

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

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
4. There are some operators that cannot be overloaded. (See Table 7.1.)
5. We cannot use **friend** functions to overload certain operators. (See Table 7.2.) However, member functions can be used to overload them.

6. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
7. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
8. When using binary operators overloaded through a member function, the left-hand operand must be an object of the relevant class.
9. Binary arithmetic operators such as +, −, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Table 7.1 *Operators that cannot be overloaded*

<i>Sizeof</i>	<i>Size of operator</i>
.	Membership operator
.*	Pointer-to-member operator
::	Scope resolution operator
?:	Conditional operator

Table 7.2 *Where a friend cannot be used*

=	Assignment operator
()	Function call operator
[]	Subscripting operator
->	Class member access operator



7.9 TYPE CONVERSIONS

We know that when constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left. For example, the statements

```
int m;
float x = 3.14159;
m = x;
```

convert **x** to an integer before its value is assigned to **m**. Thus, the fractional part is truncated. The type conversions are automatic as long as the data types involved are built-in types.

What happens when they are user-defined data types?

Consider the following statement that adds two objects and then assigns the result to a third object.

```
v3 = v1 + v2;    // v1, v2 and v3 are class type objects
```

When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. We have seen, in the case of class

objects, that the values of all the data members of the right-hand object are simply copied into the corresponding members of the object on the left-hand. What if one of the operands is an object and the other is a built-in type variable? Or, what if they belong to two different classes?

Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types. We must, therefore, design the conversion routines by ourselves, if such operations are required.

Three types of situations might arise in the data conversion between incompatible types:

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

We shall discuss all the three cases in detail.

7.9.1 Basic to Class Type

The conversion from basic type to class type is easy to accomplish. It may be recalled that the use of constructors was illustrated in a number of examples to initialize objects. For example, a constructor was used to build a vector object from an **int** type array. Similarly, we used another constructor to build a string type object from a **char*** type variable. These are all examples where constructors perform a *defacto* type conversion from the argument's type to the constructor's class type.

Consider the following constructor:

```
string :: string(char *a)
{
    length = strlen(a);
    P = new char[length+1];
    strcpy(P,a);
}
```

This constructor builds a **string** type object from a **char*** type variable **a**. The variables **length** and **p** are data members of the class **string**. Once this constructor has been defined in the string class, it can be used for conversion from **char*** type to string type. Example:

```
string s1, s2;
char* name1 = "IBM PC";
char* name2 = "Apple Computers";
s1 = string(name1);
s2 = name2;
```

The statement

```
s1 = string(name1);
```

first converts **name1** from **char*** type to **string** type and then assigns the string type values to the object **s1**. The statement

```
s2 = name2;
```

also does the same job by invoking the constructor implicitly.

Let us consider another example of converting an **int** type to a **class** type.

```
class time
{
    int hrs;
    int mins;
public:
    ....
    ....
    time(int t)                // constructor
    {
        hrs = t/60; // t in minutes
        mins = t%60;
    }
};
```

The following conversion statements can be used in a function:

```
time T1;                // object T1 created
int duration = 85;
T1 = duration;           // int to class type
```

After this conversion, the **hrs** member of **T1** will contain a value of 1 and **mins** member a value of 25, denoting 1 hours and 25 minutes.



Note

The constructors used for the type conversion take a single argument whose type is to be converted.

In both the examples, the left-hand operand of = operator is always a **class** object. Therefore, we can also accomplish this conversion using an overloaded = operator.

7.9.2 Class to Basic Type

The constructors did a fine job in type conversion from a basic to class type. What about the conversion from a class to basic type? The constructor functions do not support this operation. Luckily, C++ allows us to define an overloaded *casting operator* that could be used to convert a class type data to a basic type. The general form of an overloaded casting operator function, usually referred to as a *conversion function*, is

```
operator typename()
{
    ....
    .... (Function statements)
    ....
}
```

This function converts a class type data to *typename*. For example, the **operator double()** converts a class object to type **double**, the **operator int()** converts a class type object to type **int**, and so on.

Consider the following conversion function:

```
vector :: operator double()
{
    double sum = 0;
    for(int i=0; i<size; i++)
        sum = sum + v[i] * v[i];
    return sqrt(sum);
}
```

This function converts a vector to the corresponding scalar magnitude. Recall that the magnitude of a vector is given by the square root of the sum of the squares of its components. The operator **double()** can be used as follows:

```
double length = double(V1);
    or
double length = V1;
```

where **V1** is an object of type **vector**. Both the statements have exactly the same effect. When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Since it is a member function, it is invoked by the object and, therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

In the string example described in the previous section, we can do the conversion from string to **char*** as follows:

```
string :: operator char*()
{
    return(p);
}
```

7.9.3 One Class to Another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But there are situations where we would like to convert one class type data to another class type.

Example:

```
objX = objY; // objects of different types
```

objX is an object of class **X** and **objY** is an object of class **Y**. The **class Y** type data is converted to the **class X** type data and the converted value is assigned to the **objX**. Since the conversion takes place from **class Y to class X**, **Y** is known as the *source* class and **X** is known as the *destination* class.

Such conversions between objects of different classes can be carried out by either a constructor or a conversion function. The compiler treats them the same way. Then, how do we decide which form to use? It depends upon where we want the type-conversion function to be located in the source class or in the destination class.

We know that the casting operator function

```
operator typename()
```

converts the class object of which it is a *member to typename*. The *typename* may be a built-in type or a user-defined one (another class type). In the case of conversions between objects, *typename* refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used (i.e., source class). The conversion takes place in the source class and the result is given to the destination class object.

Now consider a single-argument constructor function which serves as an instruction for converting the *argument's type* to the class type of which it is a member. This implies that the argument belongs to the *source* class and is passed to the *destination* class for conversion. This makes it necessary that the conversion constructor be placed in the destination class. Figure 7.2 illustrates these two approaches. Table 7.3 provides a summary of all the three conversions. It shows that the conversion from a class to any other type (or any other class) should make use of a casting operator in the source class. On the other hand, to perform the conversion from any other type/class to a class type, a constructor should be used in the destination class.

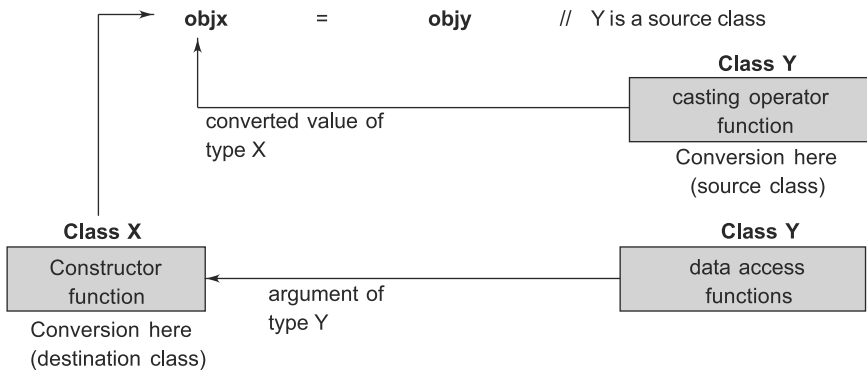


Fig. 7.2 Conversion between object

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destination class.

Table 7.3 Type conversions

Conversion required	Conversion takes place in	
	Source class	Destination class
Basic → class	Not applicable	Constructor
Class → basic	Casting operator	Not applicable
Class → class	Casting operator	Constructor

7.9.4 A Data Conversion Example

Let us consider an example of an inventory of products in store. One way of recording the details of the products is to record their code number, total items in the stock and the cost of each item. Another approach is to just specify the item code and the value of the item in the stock. The example shown in Program 7.7 uses two classes and shows how to convert data of one type to another.

Program 7.7 Data Conversions

```
#include <iostream>

using namespace std;

class invent2;      // destination class declared

class invent1      // source class
{
    int code;       // item code
    int items;      // no. of items
    float price;    // cost of each item
public:
    invent1(int a, int b, float c)
    {
        code = a;
        items = b;
        price = c;
    }
    void putdata()
    {
        cout << "Code: " << code << "\n";
        cout << "Items: " << items << "\n";
        cout << "Value: " << price << "\n";
    }
    int getcode() {return code;}
    int getitems() {return items;}
    float getprice() {return price;}
    operator float() {return(items * price);}

    /* operator invent2()          // invent1 to invent2
    {
        invent2 temp;
        temp.code = code;
        temp.value = price * items;
        return temp;
    } */
}; // End of source class

class invent2      // destination class
{
    int code;
    float value;
public:
    invent2()
```

(Contd.)

```

    {
        code = 0; value = 0;
    }
    invent2(int x, float y)        // constructor for initialization
    {
        code = x;
        value = y;
    }
    void putdata()
    {
        cout << "Code: " << code << "\n";
        cout << "Value: " << value << "\n\n";
    }
    invent2(invent1 p)            // constructor for conversion
    {
        code = p.getcode();
        value = p.getitems() * p.getprice();
    }
};                                // End of destination class

int main()
{
    invent1 s1(100,5,140.0);
    invent2 d1;
    float total_value;

    /* invent1 To float */
    total_value = s1;

    /* invent1 To invent2 */
    d1 = s1;

    cout << "Product details - invent1 type" << "\n";
    s1.putdata();

    cout << "\nStock value" << "\n";
    cout << "Value = " << total_value << "\n\n";

    cout << "Product details-invent2 type" << "\n";
    d1.putdata();

    return 0;
}

```

The output of Program 7.7 would be

```

Product details-invent1 type
Code: 100
Items: 5
Value: 140
Stock value
Value = 700
Product details-invent2 type
Code: 100
Value: 700

```


**Note**

We have used the conversion function

```
operator float( )
```

*in the class **invent1** to convert the **invent1** type data to a **float**. The constructor*

```
invent2 (invent1)
```

*is used in the class **invent2** to convert the **invent1** type data to the **invent2** type data.*

Remember that we can also use the casting operator function

```
operator invent2()
```

*in the class **invent1** to convert **invent1** type to **invent2** type. However, it is important that we do not use both the constructor and the casting operator for the same type conversion, since this introduces an ambiguity as to how the conversion should be performed.*

SUMMARY

- ☐ Operator overloading is one of the important features of C++ language that enhances its exhaustibility.
- ☐ Using overloading feature we can add two user defined data types such as objects, with the same syntax, just as basic data types.
- ☐ We can overload almost all the C++ operators except the following:
 - class member access operators(., *)
 - scope resolution operator (::)
 - size operator(sizeof)
 - conditional operator(?:)
- ☐ Operator overloading is done with the help of a special function, called operator function, which describes the special task to an operator.
- ☐ There are certain restrictions and limitations in overloading operators. Operator functions must either be member functions (nonstatic) or friend functions. The overloading operator must have at least one operand that is of user-defined type.
- ☐ The compiler does not support automatic type conversions for the user defined data types. We can use casting operator functions to achieve this.
- ☐ The casting operator function should satisfy the following conditions:
 - It must be a class member.
 - It must not specify a return type.
 - It must not have any arguments.

Key Terms

arithmetic notation | binary operators | casting | casting operator | constructor | conversion function | destination class | friend | friend function | functional notation | manipulating strings | operator | operator function | operator overloading | scalar multiplication | semantics | sizeof | source class | syntax | temporary object | type conversion | unary operators

Review Questions

- 7.1 What is type conversion? F A Q
- 7.2 What is operator overloading?
- 7.3 Why is it necessary to overload an operator?
- 7.4 Name the operators that can not be overloaded in C++.
- 7.5 What is an operator function? Describe the syntax of an operator function.
- 7.6 How many arguments are required in the definition of an overloaded unary operator?
- 7.7 A class alpha has a constructor as follows:
 alpha(int a, double b);
 Can we use this constructor to convert types?
- 7.8 A friend function cannot be used to overload the assignment operator =. Explain why?
- 7.9 When is a friend function compulsory? Give an example. F A Q
- 7.10 We have two classes X and Y. If **a** is an object of X and b is an object of Y and we want to say **a = b**; What type of conversion routine should be used and where?
- 7.11 State whether the following statements are TRUE or FALSE.
- (a) Using the operator overloading concept, we can change the meaning of an operator.
 - (b) Operator overloading works when applied to class objects only.
 - (c) Friend functions cannot be used to overload operators.
 - (d) When using an overloaded binary operator, the left operand is implicitly passed to the member function.
 - (e) The overloaded operator must have at least one operand that is user-defined type.
 - (f) Operator functions never return a value.
 - (g) Through operator overloading, a class type data can be converted to a basic type data.
 - (h) A constructor can be used to convert a basic type to a class type data.

Debugging Exercises

- 7.1 Identify the error in the following program.

```
#include <iostream.h>
class Space
{
    int mCount;
public:
    Space()
    {
        mCount = 0;
    }
    Space operator ++()
    {
```

```

        mCount++;
        return Space(mCount);
    }
};
void main()
{
    Space objSpace;
    objSpace++;
}

```

7.2 Identify the error in the following program.

```

#include <iostream.h>
enum WeekDays
{
    mSunday,
    mMonday,
    mTuesday,
    mWednesday,
    mThursday,
    mFriday,
    mSaturday
};
bool op==(WeekDays& w1, WeekDays& w2)
{
    if(w1== mSunday && w2 == mSunday)
        return 1;
    else if(w1== mSunday && w2 == mSunday)
        return 1;
    else if(w1== mSunday && w2 == mSunday)
        return 1;
    else if(w1== mSunday && w2 == mSunday)
        return 1;
    else if(w1== mSunday && w2 == mSunday)
        return 1;
    else if(w1== mSunday && w2 == mSunday)
        return 1;
    else if(w1== mSunday && w2 == mSunday)
        return 1;
    else
        return 0;
}
void main()
{
    WeekDays w1 = mSunday, w2 = mSunday;
    if(w1==w2)
        cout << "Same day";
    else
        cout << "Different day";
}

```

7.3 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
    float mWidth;
    float mLength;
public:
    Room()
    {
    }
    Room(float w, float h)
        :mWidth(w), mLength(h)
    {
    }
    operator float()
    {
        return (float)mWidth * mLength;
    }

    float getWidth()
    {
    }

    float getLength()
    {
        return mLength;
    }
};

void main()
{
    Room objRoom1(2.5, 2.5);
    float fTotalArea;
    fTotalArea = objRoom1;
    cout << fTotalArea;
}
```

Programming Exercises

NOTE: For all the exercises that follow, build a demonstration program to test your code.

7.1 Create a class **FLOAT** that contains one float data member. Overload all the four arithmetic operators so that they operate on the objects of **FLOAT**. WEB

7.2 Design a class **Polar** which describes a point in the plane using polar coordinates **radius** and **angle**. A point in polar coordinates is shown in Fig. 7.3.

Use the overloaded + operator to add two objects of Polar.

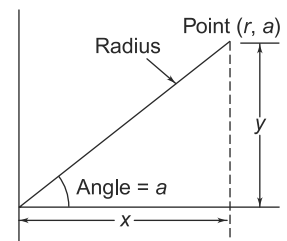


Fig. 7.3 Polar coordinates of a point

Note that we cannot add polar values of two points directly. This requires first the conversion of points into rectangular coordinates, then adding the corresponding rectangular coordinates and finally converting the result back into polar coordinates. You need to use the following trigonometric formulae:

```
x = r * cos(a);  
y = r * sin(a);  
a = atan(y/x); // arc tangent  
r = sqrt(x*x + y*y);
```

- 7.3** Create a class **MAT** of size **m** x **n**. Define all possible matrix operations for **MAT** type objects.
- 7.4** Define a class **String**. Use overloaded **==** operator to compare two strings.
- 7.5** Define two classes **Polar** and **Rectangle** to represent points in the polar and rectangle systems. Use conversion routines to convert from one system to the other.

UNIT 2

Inheritance and Polymorphism

- **Chapter 8:** Inheritance : Extending Classes
- **Chapter 9:** Pointers, Virtual Functions and Polymorphism

8

INHERITANCE: EXTENDING CLASSES

CHAPTER OUTLINE

- 8.1 Introduction
- 8.2 Defining Derived Classes
- 8.3 Single Inheritance
- 8.4 Making a Private Member Inheritable
- 8.5 Multilevel Inheritance
- 8.6 Multiple Inheritance
 - 8.6.1 Ambiguity Resolution in Inheritance
- 8.7 Hierarchical Inheritance
- 8.8 Hybrid Inheritance
- 8.9 Virtual Base Classes
- 8.10 Abstract Classes
- 8.11 Constructors in Derived Classes
- 8.12 Member Classes: Nesting of Classes
- Summary*
- Key Terms*





8.1 INTRODUCTION

Reusability is yet another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save time and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of *reusability*. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called *inheritance* (or *derivation*). The old class is referred to as the *base class* and the new one is called the *derived class* or *subclass*.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class, is called *single inheritance* and one with several base classes is called *multiple inheritance*. On the other hand, the traits of one class may be inherited by more than one class. This process is known as *hierarchical inheritance*. The mechanism of deriving a class from another 'derived class' is known as *multilevel inheritance*. Figure 8.1 shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance. (Some authors show the arrow in opposite direction meaning "inherited from".)

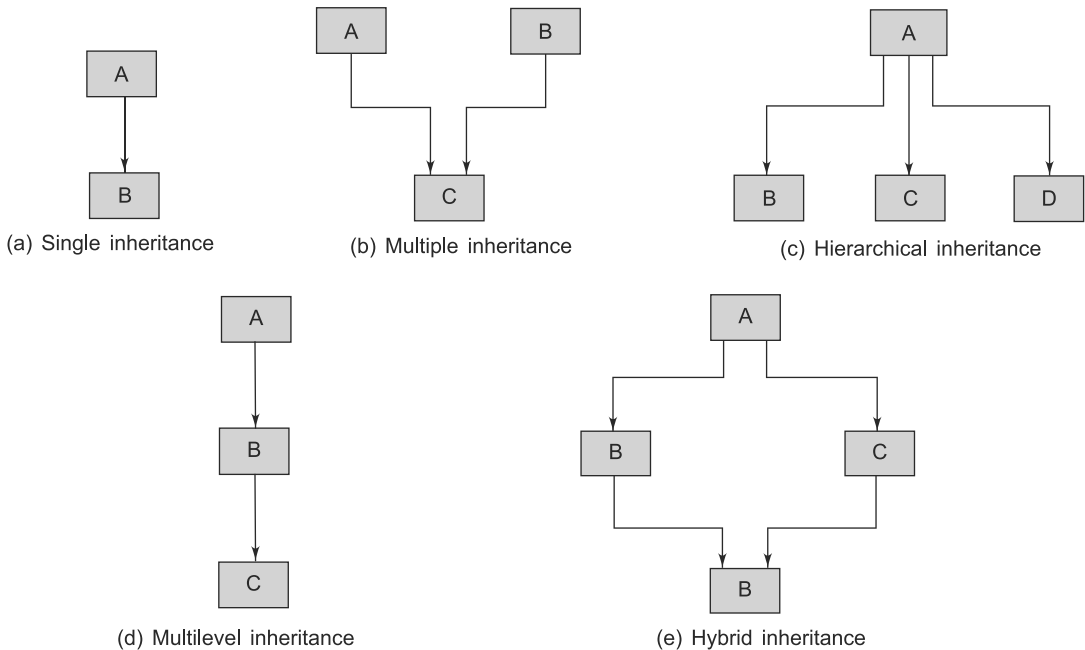


Fig. 8.1 Forms of inheritance



8.2 DEFINING DERIVED CLASSES

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is

```
class derived-class-name : visibility-mode base-class-name
{
    ..... //
    ..... // members of derived class
    ..... //
};
```

The colon indicates that the *derived-class-name* is derived from the *base-class-name*. The *visibility-mode* is optional and, if present, may be either **private** or **public**. The default visibility-mode is **private**. Visibility mode specifies whether the features of the base class are *privately derived* or *publicly derived*.

Examples:

```
class derived: private base    // private derivation
{
    members of derived
};
class derived: public base    // public derivation
{
    members of derived
};
class derived: base           // private derivation by default
{
    members of derived
};
```

When a base class is *privately inherited* by a derived class, 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. Remember, a public member of a class can be accessed by its own objects using the *dot operator*. The result is that no member of the base class is accessible to the objects of the derived class.

On the other hand, when the base class is *publicly inherited*, 'public members' of the base class become 'public members' of the derived class and therefore they are accessible to the objects of the derived class. In both the cases, the *private members* are not inherited and therefore, the private members of a base class will never become the members of its derived class.

In inheritance, some of the base class data elements and member functions are 'inherited' into the derived class. We can add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.



8.3 SINGLE INHERITANCE

Let us consider a simple example to illustrate inheritance. Program 8.1 shows a base class **B** and a derived class **D**. The class **B** contains one private data member, one public data member, and three public member functions. The class **D** contains one private data member and two public member functions.

Program 8.1 Single Inheritance: Public

```
#include <iostream>

using namespace std;

class B
{
    int a;    // private; not inheritable
public:
    int b;    // public; ready for inheritance
    void set_ab();
    int get_a(void);
    void show_a(void);
};

class D : public B    // public derivation
{
    int c;
public:
    void mul(void);
    void display(void);
};

//-----
void B :: set_ab(void)
{
    a = 5; b = 10;
}

int B :: get_a()
{
    return a;
}

void B :: show_a()
{
    cout << "a = " << a << "\n";
}

void D :: mul()
{
    c = b * get_a();
}

void D :: display()
{
    cout << "a = " << get_a() << "\n";
```

(Contd.)

```

        cout << "b = " << b << "\n";
        cout << "c = " << c << "\n\n";
    }
    //-----
int main()
{
    D d;

    d.set_ab();
    d.mul();
    d.show_a();
    d.display();

    d.b = 20;
    d.mul();
    d.display();

    return 0;
}

```

The output of Program 8.1 would be

```

a = 5
a = 5
b = 10
c = 50

a = 5
b = 20
c = 100

```

The class **D** is a public derivation of the base class **B**. Therefore, **D** inherits all the **public** members of **B** and retains their visibility. Thus, a **public** member of the base class **B** is also a public member of the derived class **D**. The private members of **B** cannot be inherited by **D**. The class **D**, in effect, will have more members than what it contains at the time of declaration as shown in Fig. 8.2.

The program illustrates that the objects of class **D** have access to all the public members of **B**. Let us have a look at the functions **show_a()** and **mul()**:

```

void show_a()
{
    cout << "a = " << a << "\n";
}

void mul()
{
    c = b * get_a();      // c = b * a
}

```

Although the data member **a** is private in **B** and cannot be inherited, objects of **D** are able to access it through an inherited member function of **B**.

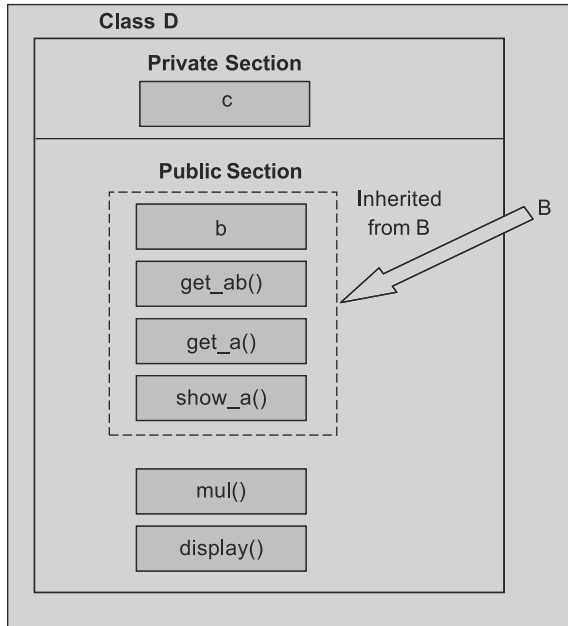


Fig. 8.2 Adding more members to a class (by public derivation)

Let us now consider the case of private derivation.

```
class B
{
    int a;
public:
    int b;
    void get_ab();
void get_a();
    void show_a();
};

class D : private B                // private derivation
{
    int c;
public:
    void mul();
    void display();
};
```

The membership of the derived class **D** is shown in Fig. 8.3. In **private** derivation, the **public** members of the base class become **private** members of the derived class. Therefore, the objects of **D** can not have direct access to the public member functions of **B**.

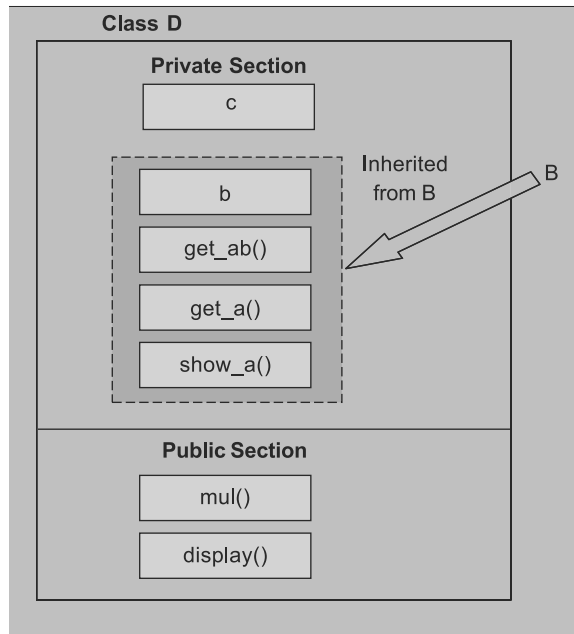


Fig. 8.3 Adding more members to a class (by private derivation)

The statements such as

```
d.get_ab();           // get_ab() is private
d.get_a();           // so also get_a()
d.show_a();          // and show_a()
```

will not work. However, these functions can be used inside **mul()** and **display()** like the normal functions as shown below:

```
void mul()
{
    get_ab();
    c = b * get_a();
}
void display()
{
    show_a();           // outputs value of 'a'
    cout << "b = " << b << "\n"
        << "c = " << c << "\n\n";
}
```

Program 8.2 incorporates these modifications for private derivation. Please compare this with Program 8.1.

Program 8.2 Single Inheritance : Private

```
#include <iostream>

using namespace std;

class B
{
    int a;                                // private; not inheritable
public:
    int b;                                // public; ready for inheritance
    void get_ab();
    int get_a(void);
    void show_a(void);
};

class D : private B                       // private derivation
{
    int c;
public:
    void mul(void);
    void display(void);
};

// -----

void B :: get_ab(void)
{
    cout << "Enter values for a and b: ";
    cin  >> a >> b;
}

int B :: get_a()
{
    return a;
}

void B :: show_a()
{
    cout << "a = " << a << "\n";
}

void D :: mul()
{
    get_ab();
    c = b * get_a();                     // 'a' cannot be used directly
}

void D :: display()
{
```

(Contd.)

```

    show_a(); // outputs value of 'a'
    cout << "b = " << b << "\n"
         << "c = " << c << "\n\n";
}

// -----
int main()
{
    D d;

    // d.get_ab(); WON'T WORK
    d.mul();
    // d.show_a(); WON'T WORK
    d.display();
    // d.b = 20; WON'T WORK; b has become private
    d.mul();
    d.display();

    return 0;
}

```

The output of Program 8.2 would be

```

Enter values for a and b: 5 10
a = 5
b = 10
c = 50
Enter values for a and b: 12 20
a = 12
b = 20
c = 240

```

Suppose a base class and a derived class define a function of the same name. What will happen when a derived class object invokes the function? In such cases, the derived class function supersedes the base class definition. However, if the derived class does not redefine the function, then the base class function will be called.



8.4 MAKING A PRIVATE MEMBER INHERITABLE

We have just seen how to increase the capabilities of an existing class without modifying it. We have also seen that a private member of a base class cannot be inherited and therefore it is not available for the derived class directly. What do we do if the **private** data needs to be inherited by a derived class? This can be accomplished by modifying the visibility limit of the **private** member by making it **public**. This would make it accessible to all the other functions of the program, thus taking away the advantage of data hiding.

C++ provides a third *visibility modifier*, **protected**, which serve a limited purpose in inheritance. A member declared as **protected** is accessible by the member functions within its class and any class

immediately derived from it. It *cannot* be accessed by the functions outside these two classes. A class can now use all the three visibility modes as illustrated below:

```
class alpha
{
    private:                // optional
        .....              // visible to member functions
        .....              // within its class
    protected:
        .....              // visible to member functions
        .....              // of its own and that of derived class
    public:
        .....              // visible to all functions
        .....              // in the program
};
```

When a **protected** member is inherited in **public** mode, it becomes **protected** in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A **protected** member, inherited in the **private** mode derivation, becomes **private** in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since **private** members cannot be inherited). Figure 8.4 is the pictorial representation for the two levels of derivation.

The keywords **private**, **protected**, and **public** may appear in any order and any number of times in the declaration of a class. For example,

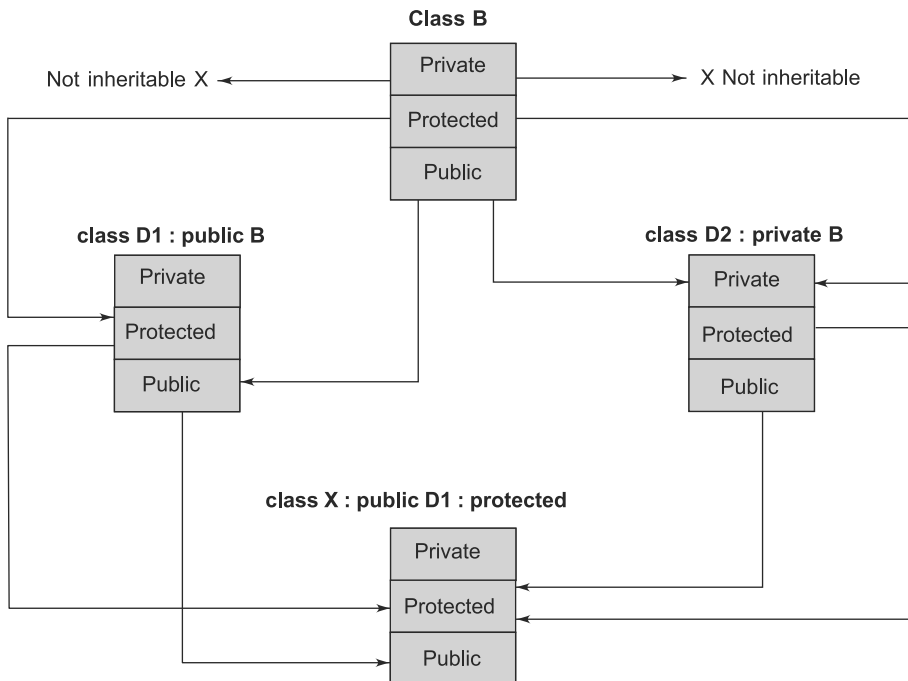


Fig. 8.4 Effect of inheritance on the visibility of members

```

class beta
{
    protected:
        .....
    public:
        .....
    private:
        .....
    public:
        .....
};

```

is a valid class definition.

However, the normal practice is to use them as follows:

```

class beta
{
    .....
    .....
    protected:
        .....
    public:
        .....
}

```

// private by default

It is also possible to inherit a base class in **protected** mode (known as *protected derivation*). In protected derivation, both the **public** and **protected** members of the base class become **protected** members of the derived class. Table 8.1 summarizes how the visibility of base class members undergoes modifications in all the three types of derivation.

Table 8.1 *Visibility of inherited members*

Base class visibility		Derived class visibility		
		Public derivation	Private derivation	Protected derivation
Private	→	Not inherited	Not inherited	Not inherited
Protected	→	Protected	Private	Protected
Public	→	Public	Private	Protected

Now let us review the access control to the **private** and **protected** members of a class. What are the various functions that can have access to these members? They could be

1. A function that is a friend of the class.
2. A member function of a class that is a friend of the class.
3. A member function of a derived class.

While the friend functions and the member functions of a friend class can have direct access to both the **private** and **protected** data, the member functions of a derived class can directly access only the

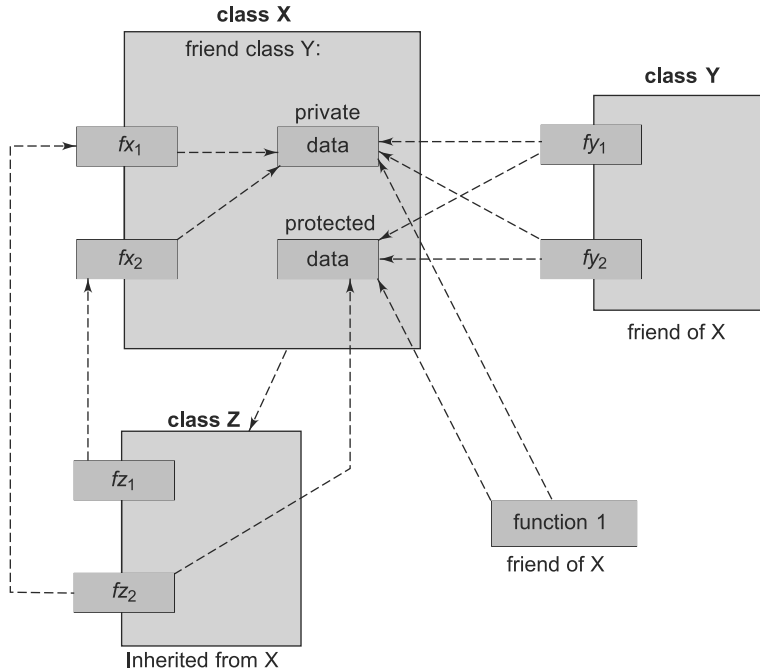


Fig. 8.5 Access mechanism in classes

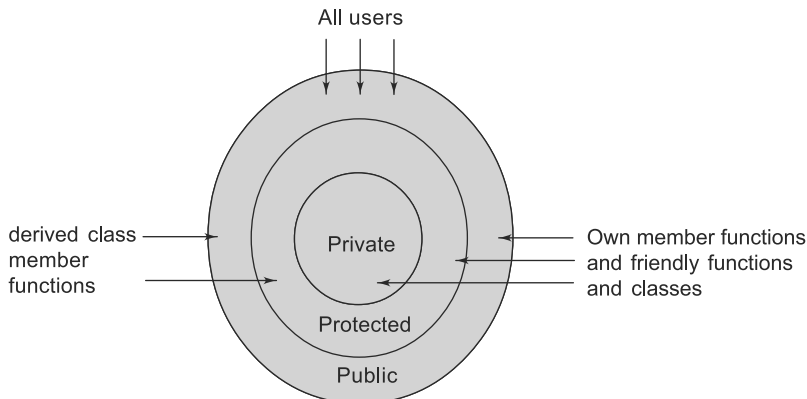


Fig. 8.6 A simple view of access control to the members of a class

protected data. However, they can access the **private** data through the member functions of the base class. Figure 8.5 illustrates how the access control mechanism works in various situations. A simplified view of access control to the members of a class is shown in Fig. 8.6.



8.5 MULTILEVEL INHERITANCE

It is not uncommon that a class is derived from another derived class as shown in Fig. 8.7. The class **A** serves as a base class for the derived class **B**, which in turn serves as a base class for the derived class

C. The class **B** is known as *intermediate* base class since it provides a link for the inheritance between **A** and **C**. The chain **ABC** is known as inheritance *path*.

A derived class with multilevel inheritance is declared as follows:

```
class A{.....};           // Base class
class B: public A {.....}; // B derived from A
class C: public B {.....}; // C derived from B
```

This process can be extended to any number of levels.

Let us consider a simple example. Assume that the test results of a batch of students are stored in three different classes. Class **student** stores the roll-number, class **test** stores the marks obtained in two subjects and class **result** contains the **total** marks obtained in the test. The class **result** can inherit the details of the marks obtained in the test and the roll-number of students through multilevel inheritance. Example:

```
class student
{
    protected:
        int roll_number;
    public:
        void get_number(int);
        void put_number(void);
};
void student :: get_number(int a)
{
    roll_number = a;
}
void student :: put_number()
{
    cout << "Roll Number: " << roll_number << "\n";
}

class test : public student           // First level derivation
{
    protected:
        float sub1;
        float sub2;
    public:
        void get_marks(float, float);
        void put_marks(void);
};
void test :: get_marks(float x, float y)
{
    sub1 = x;
    sub2 = y;
}
void test :: put_marks()
```

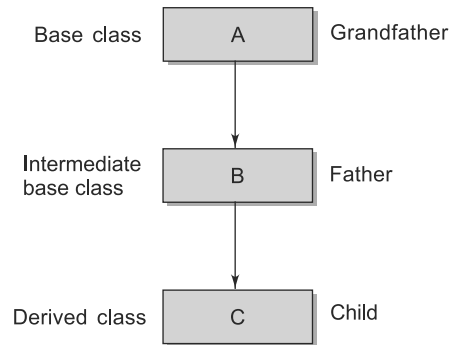


Fig. 8.7 Multilevel inheritance

```

{
    cout << "Marks in SUB1 = " << sub1 << "\n";
    cout << "Marks in SUB2 = " << sub2 << "\n";
}
class result : public test                // Second level derivation
{
    float total;                          // private by default
public:
    void display(void);
};

```

The class **result**, after inheritance from 'grandfather' through 'father', would contain the following members:

```

private:
    float total;                          // own member
protected:
    int roll_number;                     // inherited from student
                                           // via test
    float sub1;                          // inherited from test
    float sub2;                          // inherited from test
public:
    void get_number(int);                 // from student via test
    void put_number(void);                 // from student via test
    void get_marks(float, float);         // from test
    void put_marks(void);                 // from test
    void display(void);                   // own member

```

The inherited functions **put_number()** and **put_marks()** can be used in the definition of **display()** function:

```

void result :: display(void)
{
    total = sub1 + sub2;
    put_number();
    put_marks();
    cout << "Total = " << total << "\n";
}

```

Here is a simple **main()** program:

```

int main()
{
    result student1;    // student1 created
    student1.get_number(111);
    student1.get_marks(75.0, 59.5);
    student1.display();

    return 0;
}

```

This will display the result of **student1**. The complete program is shown in Program 8.3.

Program 8.3 Multilevel Inheritance

```
#include <iostream>

using namespace std;

class student
{
    protected:
        int roll_number;
    public:
        void get_number(int);
        void put_number(void);
};

void student :: get_number(int a)
{
    roll_number = a;
}
void student :: put_number()
{
    cout << "Roll Number: " << roll_number << "\n";
}
class test : public student           // First level derivation
{
    protected:
        float sub1;
        float sub2;
    public:
        void get_marks(float, float);
        void put_marks(void);
};

void test :: get_marks(float x, float y)
{
    sub1 = x;
    sub2 = y;
}

void test :: put_marks()
{
    cout << "Marks in SUB1 = " << sub1 << "\n";
    cout << "Marks in SUB2 = " << sub2 << "\n";
}

class result : public test           // Second level derivation
{
    float total;                     // private by default
```

(Contd.)

```
public:
    void display(void);
};

void result :: display(void)
{
    total = sub1 + sub2;
    put_number();
    put_marks();
    cout << "Total = " << total << "\n";
}

int main()
{
    result student1;           // student1 created

    student1.get_number(111);

    student1.get_marks(75.0, 59.5);

    student1.display();

    return 0;
}
```

The output of Program 8.3 would be

```
Roll Number: 111
Marks in SUB1 = 75
Marks in SUB2 = 59.5
Total = 134.5
```



8.6 MULTIPLE INHERITANCE

A class can inherit the attributes of two or more classes as shown in Fig. 8.8. This is known as *multiple inheritance*. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another.

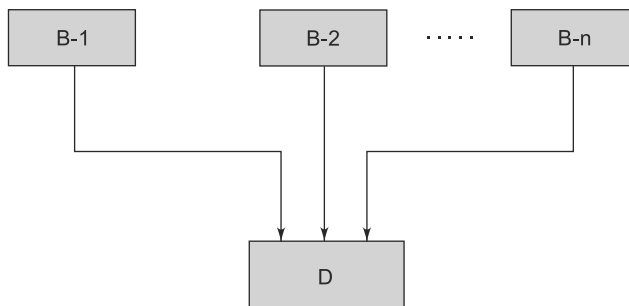


Fig. 8.8 Multiple inheritance

The syntax of a derived class with multiple base classes is as follows:

```
class D: visibility B-1, visibility B-2 ...
{
    .....
    ..... (Body of D)
    .....
};
```

where, *visibility* may be either **public** or **private**. The base classes are separated by commas.

Example:

```
class P : public M, public N
{
    public:
        void display(void);
};
```

Classes **M** and **N** have been specified as follows:

```
class M
{
    protected:
        int m;
    public:
        void get_m(int);
};
void M :: get_m(int x)
{
    m = x;
}
class N
{
    protected:
        int n;
    public:
        void get_n(int);
};
void N :: get_n(int y)
{
    n = y;
}
```

The derived class **P**, as declared above, would, in effect, contain all the members of **M** and **N** in addition to its own members as shown below:

```
class P
{
    protected:
```


int m;	// from M
int n;	// from N

```

public:
    void get_m(int);           // from M
    void get_n(int);           // from N
    void display(void);        // own member
};

```

The member function `display()` can be defined as follows:

```

void P :: display(void)
{
    cout << "m = " << m << "\n";
    cout << "n = " << n << "\n";
    cout << "m*n =" << m*n << "\n";
};

```

The `main()` function which provides the user-interface may be written as follows:

```

main()
{
    P p;
    p.get_m(10);
    p.get_n(20);
    p.display();
}

```

Program 8.4 shows the entire code illustrating how all the three classes are implemented in multiple inheritance mode.

Program 8.4 Multiple Inheritance

```

#include <iostream>

using namespace std;

class M
{
    protected:
        int m;
    public:
        void get_m(int);
};

class N
{

```

(Contd.)

```
protected:
    int n;
public:
    void get_n(int);
};

class P : public M, public N
{
    public:
        void display(void);
};

void M :: get_m(int x)
{
    m = x;
}

void N :: get_n(int y)
{
    n = y;
}

void P :: display(void)
{
    cout << "m = " << m << "\n";
    cout << "n = " << n << "\n";
    cout << "m*n = " << m*n << "\n";
}

int main()
{
    P p;

    p.get_m(10);
    p.get_n(20);
    p.display();

    return 0;
}
```

The output of Program 8.4 would be

```
m = 10
n = 20
m*n = 200
```

Ambiguity Resolution in Inheritance

Occasionally, we may face a problem in using the multiple inheritance, when a function with the same name appears in more than one base class. Consider the following two classes.

```

class M
{
    public:
        void display(void)
        {
            cout << "Class M\n";
        }
};

class N
{
    public:
        void display(void)
        {
            cout << "Class N\n";
        }
};

```

Which **display()** function is used by the derived class when we inherit these two classes? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function as shown below:

```

class P : public M, public N
{
    public:
        void display(void)    // overrides display() of M and N
        {
            M :: display(); }
};

```

We can now use the derived class as follows:

```

int main()
{
    P p;
    p.display();
}

```

Ambiguity may also arise in single inheritance applications. For instance, consider the following situation:

```

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

class B : public A
{
    public:
        void display()
        {

```

```

        cout << "B\n";
    }
};

```

In this case, the function in the derived class overrides the inherited function and, therefore, a simple call to **display()** by **B** type object will invoke function defined in **B** only. However, we may invoke the function defined in **A** by using the scope resolution operator to specify the class.

Example:

```

int main()
{
    B b;                                // derived class object
    b.display();                        // invokes display() in B
    b.A::display();                    // invokes display() in A
    b.B::display();                    // invokes display() in B

    return 0;
}

```

This will produce the following output:

```

B
A
B

```



8.7 HIERARCHICAL INHERITANCE

We have discussed so far how inheritance can be used to modify a class when it did not satisfy the requirements of a particular problem on hand. Additional members are added through inheritance to extend the capabilities of a class. Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.

As an example, Fig. 8.9 shows a hierarchical classification of students in a university. Another example could be the classification of accounts in a commercial bank as shown in Fig. 8.10. All the students have certain things in common and, similarly, all the accounts possess certain common features.

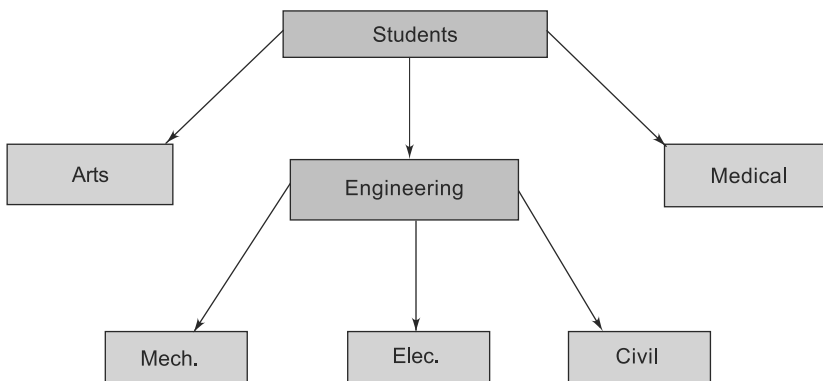


Fig. 8.9 Hierarchical classification of students

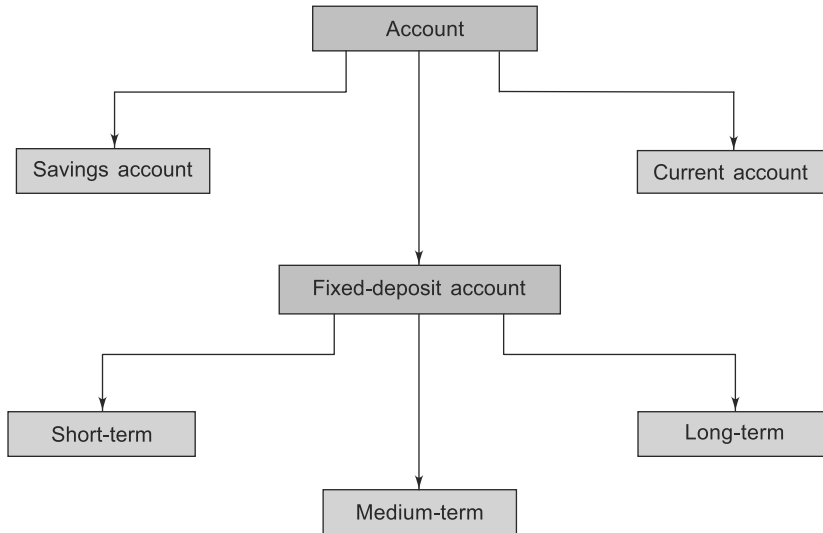


Fig. 8.10 *Classification of bank accounts*

In C++, such problems can be easily converted into class hierarchies. The base class will include all the features that are common to the subclasses. A *subclass* can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.



8.8 HYBRID INHERITANCE

There could be situations where we need to apply two or more types of inheritance to design a program. For instance, consider the case of processing the student results discussed in Sec. 8.5. Assume that we have to give weightage for sports before finalising the results. The weightage for sports is stored in a separate class called **sports**. The new inheritance relationship between the various classes would be as shown in Fig. 8.11.

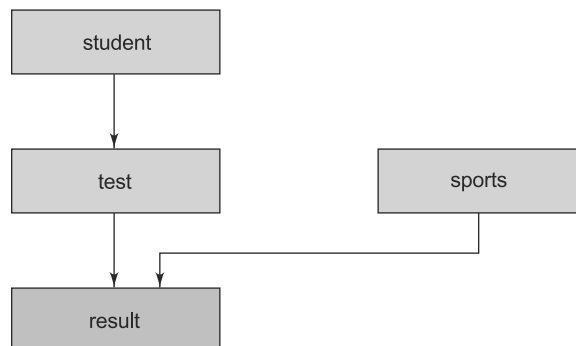


Fig. 8.11 *Multilevel, multiple inheritance*

The **sports** class might look like

```
class sports
{
    protected:
        float score;
    public:
        void get_score(float);
        void put_score(void);
};
```

The result will have both the multilevel and multiple inheritances and its declaration would be as follows:

```
class result : public test, public sports
{
    .....
    .....
};
```

Where test itself is a derived class from student. That is,

```
class test : public student
{
    .....
    .....
};
```

Program 8.5 illustrates the implementation of both multilevel and multiple inheritance.

Program 8.5 Hybrid Inheritance

```
#include <iostream>

using namespace std;

class student
{
    protected:
        int roll_number;
    public:
        void get_number(int a)
        {
            roll_number = a;
        }
        void put_number(void)
        {
            cout << "Roll No: " << roll_number << "\n";
        }
};
```

(Contd.)

```

class test : public student
{
    protected:
        float part1, part2;
    public:
        void get_marks(float x, float y)
        {
            part1 = x; part2 = y;
        }
        void put_marks(void)
        {
            cout << "Marks obtained: " << "\n"
                << "Part1 = " << part1 << "\n"
                << "Part2 = " << part2 << "\n";
        }
};

class sports
{
    protected:
        float score;
    public:
        void get_score(float s)
        {
            score = s;
        }
        void put_score(void)
        {
            cout << "Sports wt: " << score << "\n\n";
        }
};

class result : public test, public sports
{
    float total;
    public:
        void display(void);
};

void result :: display(void)
{
    total = part1 + part2 + score;

    put_number();
    put_marks();
    put_score();

    cout << "Total Score: " << total << "\n";
}

int main()

```

(Contd.)

```

{
    result student_1;
    student_1.get_number(1234);
    student_1.get_marks(27.5, 33.0);
    student_1.get_score(6.0);
    student_1.display();

    return 0;
}

```

The output of Program 8.5 would be

```

Roll No: 1234
Marks obtained:
Part1 = 27.5
Part2 = 33
Sports wt: 6

Total Score: 66.5

```



8.9 VIRTUAL BASE CLASSES

We have just discussed a situation which would require the use of both the multiple and multilevel inheritance. Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in Fig. 8.12. The 'child' has two *direct base classes* 'parent1' and 'parent2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as *indirect base class*.

Inheritance by the 'child' as shown in Fig. 8.12 might pose some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. This means, 'child' would have *duplicate* sets of the members inherited from 'grandparent'. This introduces ambiguity and should be avoided.

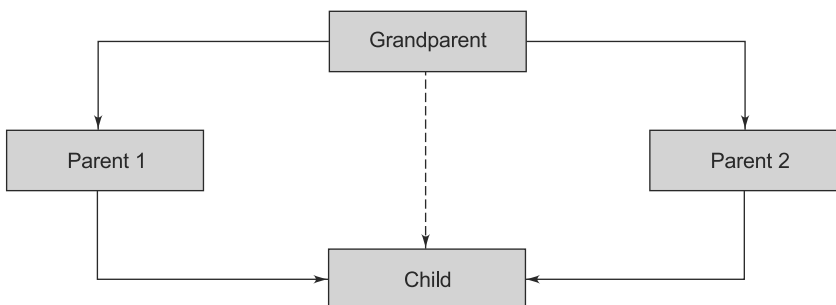


Fig. 8.12 Multipath inheritance

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as *virtual base class* while declaring the direct or intermediate base classes as shown below:

```
class A                                // grandparent
{
    .....
    .....
};
class B1 : virtual public A            // parent1
{
    .....
    .....
};
class B2 : public virtual A           // parent2
{
    .....
    .....
};
class C : public B1, public B2        // child
{
    .....                            // only one copy of A
    .....                            // will be inherited
};
```

When a class is made a **virtual** base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.



Note The keywords *virtual* and *public* may be used in either order.

For example, consider again the student results processing system discussed in Sec. 8.8. Assume that the class **sports** derives the **roll_number** from the class **student**. Then, the inheritance relationship will be as shown in Fig. 8.13.

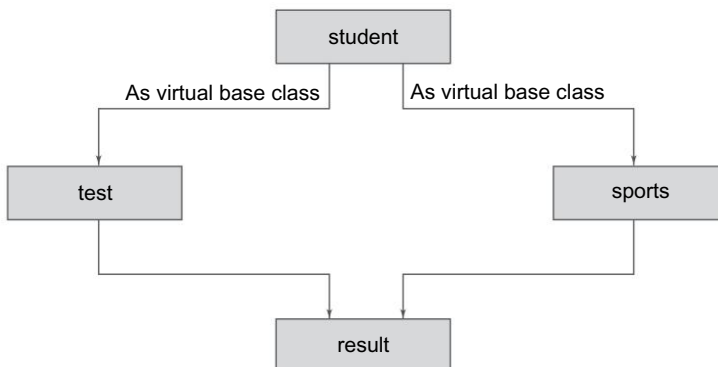


Fig. 8.13 Virtual base class

A program to implement the concept of virtual base class is illustrated in Program 8.6.

Program 8.6 Virtual Base Class

```
#include <iostream>
using namespace std;
class student
{
    protected:
        int roll_number;
    public:
        void get_number(int a)
        {
            roll_number = a;
        }
        void put_number(void)
        {
            cout << "Roll No: " << roll_number << "\n";
        }
};

class test : virtual public student
{
    protected:
        float part1, part2;
    public:
        void get_marks(float x, float y)
        {
            part1 = x; part2 = y;
        }
        void put_marks(void)
        {
            cout << "Marks obtained: " << "\n"
                << "Part1 = " << part1 << "\n"
                << "Part2 = " << part2 << "\n";
        }
};

class sports : public virtual student
{
    protected:
        float score;
    public:
        void get_score(float s)
        {
            score = s;
        }
        void put_score(void)
        {
            cout << "Sports wt: " << score << "\n\n";
        }
};
```

(Contd.)

```

    }
};

class result : public test, public sports
{
    float total;
public:
    void display(void);
};

void result :: display(void)
{
    total = part1 + part2 + score;

    put_number();
    put_marks();
    put_score();

    cout << "Total Score: " << total << "\n";
}

int main()
{
    result student_1;
    student_1.get_number(678);
    student_1.get_marks(30.5, 25.5);
    student_1.get_score(7.0);
    student_1.display();

    return 0;
}

```

The output of Program 8.6 would be

```

Roll No: 678
Marks obtained:
Part1 = 30.5
Part2 = 25.5
Sport wt: 7
Total Score: 63

```

In the above example, only one copy of **roll_number** variable is inherited by the result class from the student class. Thus, any changes made to **roll_number** by any of the derived classes are made to this single instance. This ensures data integrity.



8.10 ABSTRACT CLASSES

An *abstract class* is one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes). It is a design concept in program development and provides a base upon which other classes may be built.

In real programming scenarios, the concept of abstract base classes holds great significance. They are deliberately used in a program for creating derived classes and are not meant for creating instance

objects as demonstrated in **student** class in Program 8.6. The concept can be further explained through an example of an abstract base class **vehicle** which may be declared in a program for deriving **LMV** (light motor vehicle), **HMV** (heavy motor vehicle) and **TW** (two wheeler) derived classes. A pure virtual function **spec()** may set the specifications of **LMV** as per its four wheels and smaller engine and that of **HMV** as per its eight or more wheels and relatively larger engine.

While the use of the abstract base class **vehicle** may look to be optional here, it holds significance as far as logical program design is concerned. Also, we may include some of the common characteristics of all the derived classes in the abstract base class itself. For example, speed, mileage, rpm, etc., are some of the characteristics that are common across all derived classes of the vehicle base class. However, one must never forget the fact that by definition a class can only be considered as an abstract class if it has at least one pure virtual function.

The general form of using abstract class is shown below

```
class vehicle //abstract base class
{
    private:
        data-type d1;
        data-type d2;

    public:
        virtual void spec()=0; //pure virtual function
};

class LMV : public vehicle
{
    public:
        void spec()
        {
            //LMV definition of spec function
        }
};

class HMV : public vehicle
{
    public:
        void spec()
        {
            //HMV definition of spec function
        }
};

class TW : public vehicle
{
    public:
        void spec()
        {
            //TW definition of spec function
        }
};
```



8.11 CONSTRUCTORS IN DERIVED CLASSES

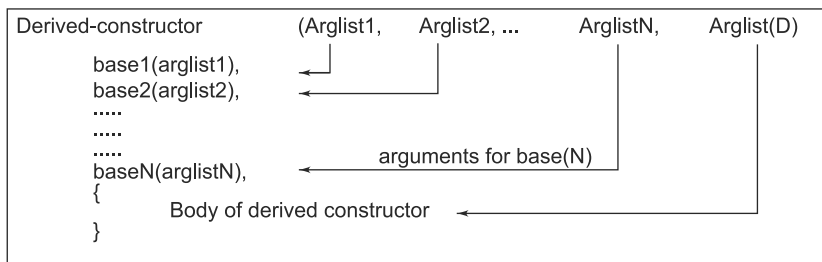
As we know, the constructors play an important role in initializing objects. We did not use them earlier in the derived classes for the sake of simplicity. One important thing to note here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. Remember, while applying inheritance we usually create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base classes are constructed *in the order in which they appear in the declaration of the derived class*. Similarly, in a multilevel inheritance, the constructors will be executed *in the order of inheritance*.

Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. How are they passed to the base class constructors so that they can do their job? C++ supports a special argument passing mechanism for such situations.

The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor.

The general form of defining a derived constructor is:



The header line of *derived-constructor* function contains two parts separated by a colon(:). The first part provides the declaration of the arguments that are passed to the *derived-constructor* and the second part lists the function calls to the base constructors.

base1(arglist1), base2(arglist2) ... are function calls to base constructors **base1()**, **base2()**, ... and therefore *arglist1, arglist2, ...* etc. represent the actual parameters that are passed to the base constructors. *Arglist1* through *ArglistN* are the argument declarations for base constructors *base1* through *baseN*. *ArglistD* provides the parameters that are necessary to initialize the members of the derived class.

Example:

```

D(int a1, int a2, float b1, float b2, int d1):
A(a1, a2),      /* call to constructor A */
B(b1, b2)      /* call to constructor B */
{
    d = d1;      // executes its own body
}
  
```

A(a1, a2) invokes the base constructor **A()** and B(b1, b2) invokes another base constructor **B()**. The constructor **D()** supplies the values for these four arguments. In addition, it has one argument of its own. The constructor **D()** has a total of five arguments. **D()** may be invoked as follows:

```
.....
D objD(5, 12, 2.5, 7.54, 30);
.....
```

These values are assigned to various parameters by the constructor **D()** as follows:

```
5      →    a1
12     →    a2
2.5    →    b1
7.54   →    b2
30     →    d1
```

The constructors for virtual base classes are invoked before any nonvirtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared. Any non-virtual bases are then constructed before the derived class constructor is executed. See Table 8.2.

Table 8.2 *Execution of base class constructors*

Method of inheritance	Order of execution
Class B: public A { }; class A : public B, public C { }; class A : public B, virtual public C { };	A () ; base constructor B () ; derived constructor B () ; base(first) C () ; base(second) A () ; derived C () ; virtual base B () ; ordinary base A () ; derived

Program 8.7 illustrates how constructors are implemented when the classes are inherited.

Program 8.7

Constructors in Derived Class

```
#include <iostream>

using namespace std;

class alpha
{
    int x;
public:
    alpha(int i)
    {
        x = i;
        cout << "alpha initialized \n";
    }
}
```

(Contd.)

```
        void show_x(void)
        { cout << "x = " << x << "\n"; }
};

class beta
{
    float y;
public:
    beta(float j)
    {
        y = j;
        cout << "beta initialized \n";
    }
    void show_y(void)
    { cout << "y = " << y << "\n"; }
};

class gamma: public beta, public alpha
{
    int m, n;
public:
    gamma(int a, float b, int c, int d):
        alpha(a), beta(b)
    {
        m = c;
        n = d;
        cout << "gamma initialized \n";
    }
    void show_mn(void)
    {
        cout << "m = " << m << "\n"
              << "n = " << n << "\n";
    }
};

int main()
{
    gamma g(5, 10.75, 20, 30);
    cout << "\n";
    g.show_x();
    g.show_y();
    g.show_mn();

    return 0;
}
```

The output of Program 8.7 would be

```
beta initialized
alpha initialized
gamma initialized
```

```
x = 5
y = 10.75
m = 20
n = 30
```



Note ***beta** is initialized first, although it appears second in the derived constructor. This is because it has been declared first in the derived class header line. Also, note that **alpha(a)** and **beta(b)** are function calls. Therefore, the parameters should not include types.*

C++ supports another method of initializing the class objects. This method uses what is known as initialization list in the constructor function. This takes the following form:

```
constructor (arglist) : initialization-section
{
    assignment-section
}
```

The *assignment section* is nothing but the body of the constructor function and is used to assign initial values to its data members. The part immediately following the colon is known as the *initialization section*. We can use this section to provide initial values to the base constructors and also to initialize its own class members. This means that we can use either of the sections to initialize the data members of the constructors class. The initialization section basically contains a list of initializations separated by commas. This list is known as *initialization list*. Consider a simple example:

```
class XYZ
{
    int a;
    int b;
public:
    XYZ(int i, int j) : a(i), b(2 * j) { }
};

main()
{
    XYZ x(2, 3);
}
```

This program will initialize **a** to 2 and **b** to 6. Note how the data members are initialized, just by using the variable name followed by the initialization value enclosed in the parenthesis (like a function call). Any of the parameters of the argument list may be used as the initialization value and the items in the list may be in any order. For example, the constructor **XYZ** may also be written as:

```
XYZ(int i, int j) : b(i), a(i + j) { }
```

In this case, **a** will be initialized to 5 and **b** to 2. Remember, the data members are initialized in the order of declaration, independent of the order in the initialization list. This enables us to have statements such as

```
XYZ(int i, int j) : a(i), b(a * j) { }
```


Here **a** is initialized to 2 and **b** to 6. Remember, **a** which has been declared first is initialized first and then its value is used to initialize **b**. However, the following will not work:

```
XYZ(int i, int j) : b(i), a(b * j) { }
```

because the value of **b** is not available to **a** which is to be initialized first.

The following statements are also valid:

```
XYZ(int i, int j) : a(i) {b = j;}
XYZ(int i, int j) { a = i; b = j;}
```

We can omit either section, if it is not needed. Program 8.8 illustrates the use of initialization lists in the base and derived constructors.

Program 8.8 Initialization List in Constructors

```
#include <iostream>

using namespace std;

class alpha
{
    int x;
public:
    alpha(int i)
    {
        x = i;
        cout << "\n alpha constructed";
    }

    void show_alpha(void)
    {
        cout << " x = " << x << "\n";
    }
};

class beta
{
    float p, q;
public:
    beta(float a, float b): p(a), q(b+p)
    {
        cout << "\n beta constructed";
    }
    void show_beta(void)
    {
        cout << " p = " << p << "\n";
        cout << " q = " << q << "\n";
    }
};

class gamma : public beta, public alpha
```

(Contd.)

```

{
    int u,v;
public:
    gamma(int a, int b, float c):
        alpha(a*2), beta(c,c), u(a)
    { v = b; cout << "\n gamma constructed";}

    void show_gamma(void)
    {
        cout << " u = " << u << "\n";
        cout << " v = " << v << "\n";
    }
};

int main()
{
    gamma g(2, 4, 2.5);

    cout << "\n\n Display member values " << "\n\n";

    g.show_alpha();
    g.show_beta();
    g.show_gamma();

    return 0;
};

```

The output of Program 8.8 would be

```

beta constructed
alpha constructed
gamma constructed

Display member values

x = 4
p = 2.5
q = 5
u = 2
v = 4

```



Note The argument list of the derived constructor **gamma** contains only three parameters **a**, **b** and **c** which are used to initialize the five data members contained in all the three classes.



8.12 MEMBER CLASSES: NESTING OF CLASSES

Inheritance is the mechanism of deriving certain properties of one class into another. We have seen in detail how this is implemented using the concept of derived classes. C++ supports yet another way of

inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its members as shown below:

```
class alpha {...};
class beta {...};
class gamma
{
    alpha a;           // a is an object of alpha class
    beta b;           // b is an object of beta class
    .....
};
```

All objects of **gamma** class will contain the objects **a** and **b**. This kind of relationship is called *contain-ership* or *nesting*. Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then the other members are created. This means, constructors of all the member objects should be called before its own constructor body is executed. This is accomplished using an initialization list in the constructor of the nested class.

Example:

```
class gamma
{
    .....
    alpha a;           // a is object of alpha
    beta b;           // b is object of beta
public:
    gamma(arglist): a(arglist1), b(arglist2)
    {
        // constructor body
    }
};
```

arglist is the list of arguments that is to be supplied when a **gamma** object is defined. These parameters are used for initializing the members of **gamma**. *arglist1* is the argument list for the constructor of **a** and *arglist2* is the argument list for the constructor of **b**. *arglist1* and *arglist2* may or may not use the arguments from *arglist*. Remember, **a(arglist1)** and **b(arglist2)** are function calls and therefore the arguments do not contain the data types. They are simply variables or constants.

Example:

```
gamma(int x, int y, float z) : a(x), b(x,z)
{
    Assignment section(for ordinary other members)
}
```

We can use as many member objects as are required in a class. For each member object, we add a constructor call in the initializer list. The constructors of the member objects are called in the order in which they are declared in the nested class.

SUMMARY

- ☐ The mechanism of deriving a new class from an old class is called inheritance. Inheritance provides the concept of reusability. The C++ classes can be reused using inheritance.
- ☐ The derived class inherits some or all of the properties of the base class.
- ☐ A derived class with only one base class is called single inheritance.
- ☐ A class can inherit properties from more than one class which is known as multiple inheritance.
- ☐ A class can be derived from another derived class which is known as multilevel inheritance.
- ☐ When the properties of one class are inherited by more than one class, it is called hierarchical inheritance.
- ☐ A private member of a class cannot be inherited either in public mode or in private mode.
- ☐ A protected member inherited in public mode becomes protected, whereas inherited in private mode becomes private in the derived class.
- ☐ A public member inherited in public mode becomes public, whereas inherited in private mode becomes private in the derived class.
- ☐ The friend functions and the member functions of a friend class can directly access the private and protected data.
- ☐ The member functions of a derived class can directly access only the protected and public data. However, they can access the private data through the member functions of the base class.
- ☐ Multipath inheritance may lead to duplication of inherited members from a 'grandparent' base class. This may be avoided by making the common base class a virtual base class.
- ☐ In multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.
- ☐ In multilevel inheritance, the constructors are executed in the order of inheritance.
- ☐ A class can contain objects of other classes. This is known as containership or nesting.

Key Terms

abstract class | access control | access mechanism | ancestor class | assignment section | base class | base constructor | child class | common base class | containership | derivation | derived class | derived constructor | direct base class | dot operator | duplicate members | father class | **friend** | grandfather class | grandparent class | hierarchical inheritance | hybrid inheritance | indirect base class | inheritance | inheritance path | initialization list | initialization section | intermediate base | member classes | multilevel inheritance | multiple inheritance | nesting | **private** | private derivation | private members | privately derived | **protected** | protected members | **public** | public derivation | public members | publicly derived | reusability | single inheritance | subclass | virtual base class | visibility mode | visibility modifier

Review Questions

- 8.1 What does inheritance mean in C++?
- 8.2 What are the different forms of inheritance? Give an example for each.
- 8.3 Describe the syntax of the single inheritance in C++.
- 8.4 We know that a private member of a base class is not inheritable. Is it anyway possible for the objects of a derived class to access the private members of the base class? If yes, how? Remember, the base class cannot be modified.
- 8.5 How do the properties of the following two derived classes differ?
- (a) `class D1: private B{...};`
 - (b) `class D2: public B{...};`
- 8.6 When do we use the protected visibility specifier to a class member?
- 8.7 Describe the syntax of multiple inheritance. When do we use such an inheritance?
- 8.8 What are the implications of the following two definitions?
- (a) `class A: public B, public C{...};`
 - (b) `class A: public C, public B{...};`
- 8.9 What is a virtual base class?
- 8.10 When do we make a class virtual?
- 8.11 What is an abstract class?
- 8.12 In what order are the class constructors called when a derived class object is created?
- 8.13 Class D is derived from class B. The class D does not contain any data members of its own. Does the class D require constructors? If yes, why?
- 8.14 What is containership? How does it differ from inheritance?
- 8.15 Describe how an object of a class that contains objects of other classes created?
- 8.16 State whether the following statements are TRUE or FALSE:
- (a) Inheritance helps in making a general class into a more specific class.
 - (b) Inheritance aids data hiding.
 - (c) One of the advantages of inheritance is that it provides a conceptual framework.
 - (d) Inheritance facilitates the creation of class libraries.
 - (e) Defining a derived class requires some changes in the base class.
 - (f) A base class is never used to create objects.
 - (g) It is legal to have an object of one class as a member of another class.
 - (h) We can prevent the inheritance of all members of the base class by making base class virtual in the definition of the derived class.

Debugging Exercises

- 8.1 Identify the error in the following program.

```
#include <iostream.h>
class Student {
```

```

        char* name;
        int rollNumber;
private:
    Student() {
        name = "AlanKay";
        rollNumber = 1025;
    }
    void setNumber(int no)
    {
        rollNumber = no;
    }
    int getRollNumber() {
        return rollNumber;
    }
};

class AnualTest: Student {
    int mark1, mark2;
public:
    AnualTest(int m1, int m2)
        :mark1(m1), mark2(m2) {
    }
    int getRollNumber() {
        return Student::getRollNumber();
    }
};

void main()
{
    AnualTest test1(92, 85);
    cout << test1.getRollNumber();
}

```

8.2 Identify the error in the following program.

```

#include <iostream.h>
class A
{
public:
    A()
    {
        cout << "A";
    }
};

class B: public A
{
public:
    B()
    {
        cout << "B";
    }
};

class C: public B

```

```

{
public:
    C()
    {
        cout << "C";
    }
};
class D
{
public:
    D()
    {
        cout << "D";
    }
};
class E: public C, public D
{
public:
    E()
    {
        cout << "D";
    }
};
class F: B, virtual E
{
public:
    F()
    {
        cout << "F";
    }
};
void main()
{
    F f;
}

```

8.3 Identify the error in the following program.

```

#include <iostream.h>
class A
{
    int i;
};
class AB: virtual A
{
    int j;
};
class AC: A, ABAC
{
    int k;
};
class ABAC: AB, AC
{

```

```

        int l;
    };
void main()
{
    ABAC abac;
    cout << "sizeof ABAC:" << sizeof(abac);
}

```

8.4 Find errors in the following program. State reasons.

```

// Program test
#include <iostream.h>

class X
{
    private:
        int x1;
    protected:
        int x2;
    public:
        int x3;
};

class Y: public X
{
    public:
        void f()
        {
            int y1,y2,y3;
            y1 = x1;
            y2 = x2;
            y3 = x3;
        }
};

class Z: X
{
    public:
        void f()
        {
            int z1,z2,z3;
            z1 = x1;
            z2 = x2;
            z3 = x3;
        }
};

main()
{
    int m,n,p;
    Y y;
}

```



```

        m = y.x1;
        n = y.x2;
        p = y.x3;
        Z z;
        m = z.x1;
        n = z.x2;
        p = z.x3;
    }

```

8.5 Debug the following program.

```

// Test program
#include <iostream.h>

class B1
{
    int b1;
public:
    void display();
    {
        cout << b1 <<"\n";
    }
};

class B2
{
    int b2;
public:
    void display();
    {
        cout << b2 <<"\n";
    }
};

class D: public B1, public B2
{
    // nothing here
};

main()
{
    D d;
    d.display()
    d.B1::display();
    d.B2::display();
}

```

Programming Exercises

- 8.1** Assume that a bank maintains two kinds of accounts for customers, one called as savings account and the other as current account. The savings account provides compound interest and withdrawal facilities but no cheque book facility. The current account provides cheque book facility but no interest. Current account holders should also maintain a minimum balance and if the balance falls below this level, a service charge is imposed.

Create a class `account` that stores customer name, account number and type of account. From this, derive the classes `cur_acct` and `sav_acct` to make them more specific to their requirements. Include necessary member functions in order to achieve the following tasks:

- (a) Accept deposit from a customer and update the balance.
- (b) Display the balance.
- (c) Compute and deposit interest.
- (d) Permit withdrawal and update the balance.
- (e) Check for the minimum balance, impose penalty, necessary, and update the balance.

Do not use any constructors. Use member functions to initialize the class members.

8.2 Modify the program of Exercise 8.1 to include constructors for all the three classes.

8.3 An educational institution wishes to maintain a database of its employees. The database is divided into a number of classes whose hierarchical relationships are shown in Fig. 8.14. The figure also shows the minimum information required for each class. Specify all the classes and define functions to create the database and retrieve individual information as and when required. W E B

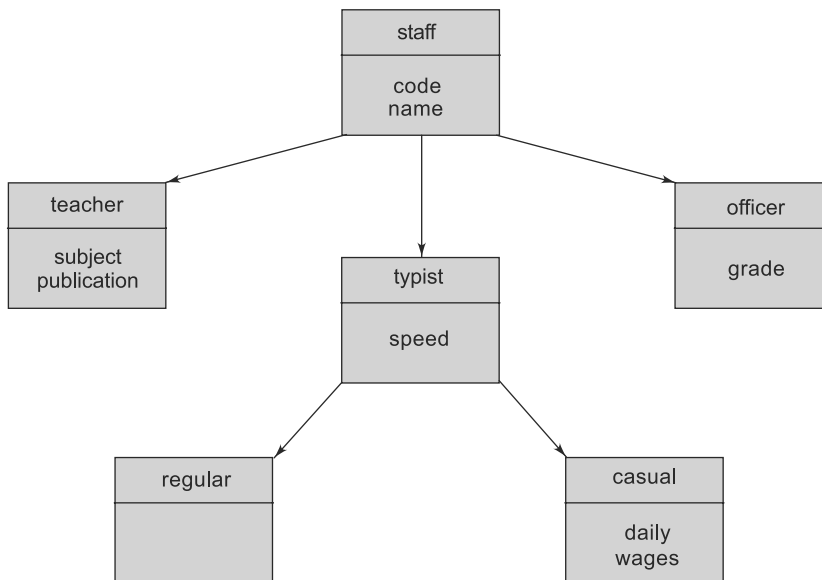


Fig. 8.14 Class relationships (for Exercise 8.3)

8.4 The database created in Exercise 8.3 does not include educational information of the staff. It has been decided to add this information to teachers and officers (and not for typists) which will help the management in decision making with regard to training, promotion, etc. Add another data class called **education** that holds two pieces of educational information, namely, highest qualification in general education and highest professional qualification. This class should be inherited by the classes **teacher** and **officer**. Modify the program of Exercise 8.19 to incorporate these additions. W E B

8.5 Consider a class network of Fig. 8.15. The class **master** derives information from both `account` and `admin` classes which in turn derive information from the class **person**. Define all the four classes and write a program to create, update and display the information contained in **master** objects.

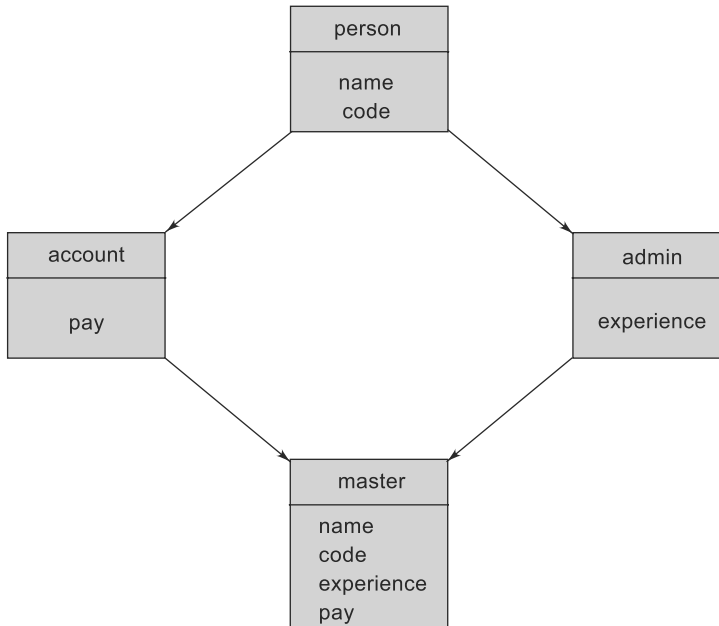


Fig. 8.15 *Multipath inheritance (for Exercise 8.5)*

- 8.6** In Exercise 8.3, the classes **teacher**, **officer**, and **typist** are derived from the class **staff**. As we know, we can use container classes in place of inheritance in some situations. Redesign the program of Exercise 8.3 such that the classes **teacher**, **officer**, and **typist** contain the objects of **staff**.
- 8.7** We have learned that OOP is well suited for designing simulation programs. Using the techniques and tricks learned so far, design a program that would simulate a simple real-world system familiar to you.

9

POINTERS, VIRTUAL FUNCTIONS AND POLYMORPHISM

C H A P T E R O U T L I N E

9.1 Introduction

9.2 Pointers

9.2.1 Declaring and Initializing Pointers

9.2.2 Manipulation of Pointers

9.2.3 Pointer Expressions and Pointer Arithmetic

9.2.4 Using Pointers with Arrays and Strings

9.2.5 Arrays of Pointers

9.2.6 Pointers and Strings

9.2.7 Pointers to Functions

9.3 Pointers to Objects

9.4 this Pointer

9.5 Pointers to Derived Classes

9.6 Virtual Functions

9.6.1 Rules for Virtual Functions

9.7 Pure Virtual Functions

9.8 Virtual Constructors and Destructors

Summary

Key Terms





9.1 INTRODUCTION

Polymorphism is one of the crucial features of OOP. It simply means 'one name, multiple forms'. We have already seen how the concept of *polymorphism* is implemented using the overloaded functions and operators. The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called *early binding* or *static binding* or *static linking*. Also known as *compile time polymorphism*, early binding simply means that an object is bound to its function call at compile time.

Now let us consider a situation where the function name and prototype is the same in both the base and derived classes. For example, consider the following class definitions:

```
class A
{
    int x;
public:
    void show() {...}    // show() in base class
};
class B: public A
{
    int y;
public:
    void show() {...}    // show() in derived class
};
```

How do we use the member function **show()** to print the values of objects of both the classes **A** and **B**? Since the prototype of **show()** is the same in both the places, the function is not overloaded and therefore static binding does not apply. We have seen earlier that, in such situations, we may use the class resolution operator to specify the class while invoking the functions with the derived class objects.

It would be nice if the appropriate member function could be selected while the program is running. This is known as *run time polymorphism*. How could it happen? C++ supports a mechanism known as *virtual function* to achieve run time polymorphism. Please refer Fig. 9.1.

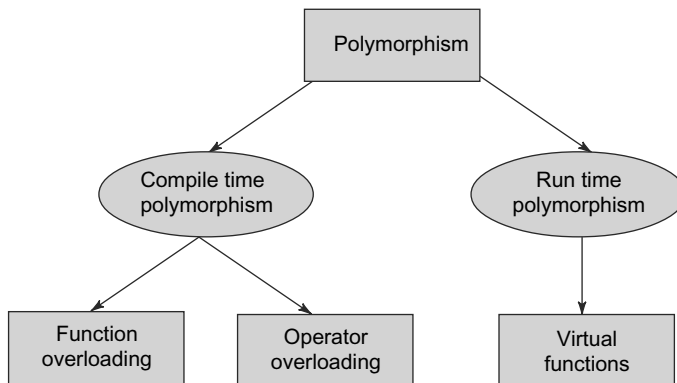


Fig. 9.1 Achieving polymorphism

At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as *late binding*. It is also known as *dynamic binding* because the selection of the appropriate function is done dynamically at run time.

Dynamic binding is one of the powerful features of C++. This requires the use of pointers to objects. We shall discuss in detail how the object pointers and virtual functions are used to implement dynamic binding.



9.2 POINTERS

Pointers is one of the key aspects of C++ language similar to that of C. As we know, pointers offer a unique approach to handle data in C and C++. We have seen some of the applications of pointers in Chapters 3 and 5. In this section, we shall discuss the rudiments of pointers and the special usage of them in C++.

We know that a pointer is a derived data type that refers to another data variable by storing the variable's memory address rather than data. A pointer variable defines where to get the value of a specific data variable instead of defining actual data.

Like C, a pointer variable can also refer to (or point to) another pointer in C++. However, it often points to a data variable. Pointers provide an alternative approach to access other data objects.

9.2.1 Declaring and Initializing Pointers

As discussed in Chapter 3, we can declare a pointer variable similar to other variables in C++. Like C, the declaration is based on the data type of the variable it points to. The declaration of a pointer variable takes the following form:

```
data-type *pointer-variable;
```

Here, *pointer-variable* is the name of the pointer, and the *data-type* refers to one of the valid C++ data types, such as int, char, float, and so on. The *data-type* is followed by an asterisk (*) symbol, which distinguishes a pointer variable from other variables to the compiler.



Note We can locate asterisk (*) immediately before the pointer variable, or between the data type and the pointer variable, or immediately after the data type. It does not cause any effect in the execution process.

As we know, a pointer variable can point to any type of data available in C++. However, it is necessary to understand that a pointer is able to point to only one data type at the specific time. Let us declare a pointer variable, which points to an integer variable, as follows:

```
int *ptr;
```

Here, **ptr** is a pointer variable and points to an integer data type. The pointer variable, ptr, should contain the memory location of any integer variable. In the same manner, we can declare pointer variables for other data types also.

Like other programming languages, a variable must be initialized before using it in a C++ program. We can initialize a pointer variable as follows:

```
int *ptr, a;           // declaration
ptr=&a;                // initialization
```

The pointer variable, **ptr**, contains the address of the variable **a**. Like C, we use the 'address of' operator or reference operator, i.e., '&' to retrieve the address of a variable. The second statement assigns the address of the variable **a** to the pointer **ptr**.

We can also declare a pointer variable to point to another pointer, similar to that of C. That is, a pointer variable contains address of another pointer. Program 9.1 explains how to refer to a pointer's address by using a pointer in a C++ program.

Program 9.1 Example of Using Pointers

```
#include <iostream.h>
#include <conio.h>
void main()
{
    int a, *ptr1, **ptr2;
    clrscr();
    ptr1 = &a;
    ptr2=&ptr1;
    cout << "The address of a : " << ptr1 << "\n";
    cout << "The address of ptr1 : " << ptr2;
    cout << "\n\n";
    cout << "After incrementing the address values:\n\n";
    ptr1+=2;
    cout << "The address of a : " << ptr1 << "\n";
    ptr2+=2;
    cout << "The address of ptr1 : " << ptr2 << "\n";
}
```

The memory location is always addressed by the operating system. The output may vary depending on the system. Output of Program 9.1 would look like

```
The address of a:      0x8fb6fff4
The address of ptr1: 0x8fb6fff2
After incrementing the address values:
The address of a:      0x8fb6fff8
The address of a:      0x8fb6fff6
```

We can also use *void pointers*, known as generic pointers, which refer to variables of any data type. Before using void pointers, we must type cast the variables to the specific data types that they point to.



Note The pointers, which are not initialized in a program, are called Null pointers. Pointers of any data type can be assigned with one value, i.e., '0' called null address.

9.2.2 Manipulation of Pointers

As discussed earlier, we can manipulate a pointer with the indirection operator, i.e., '*' which is also known as dereference operator. With this operator, we can indirectly access the data variable content. It takes the following general form:

`*pointer_variable`

As we know, dereferencing a pointer allows us to get the content of the memory location that the pointer points to. After assigning address of the variable to a pointer, we may want to change the content of the variable. Using the dereference operator, we can change the contents of the memory location.

Let us consider an example that illustrates how to dereference a pointer variable. Program 9.2 illustrates the use of dereference operator to access and alter the contents of the variable being pointed by the pointer variable.

Program 9.2 Manipulation of Pointers

```
#include<iostream.h>

#include<conio.h>

int main()
{
    int a=10;
    int *ptr;
    clrscr();

    ptr=&a;
    cout<<"The value of a is: "<<*ptr;
    *ptr=*ptr+a;
    cout<<"\nThe revised value of a is "<<a;
    getch();
    return(0);
}
```

The output of Program 9.2 would be

```
The value of a is : 10
The revised value of a is : 20
```



Caution Before dereferencing a pointer, it is essential to assign a value to the pointer. If we attempt to dereference an uninitialized pointer, it will cause runtime error by referring to any other location in memory.

9.2.3 Pointer Expressions and Pointer Arithmetic

As discussed in Chapter 3, there are a substantial number of arithmetic operations that can be performed with pointers. C++ allows pointers to perform the following arithmetic operations:

- A pointer can be incremented (++) (or) decremented (--)
- Any integer can be added to or subtracted from a pointer
- One pointer can be subtracted from another

Example:

```
int a[6];
int *aptr;
aptr=&a[0];
```

Obviously, the pointer variable, **aptr**, refers to the base address of the variable **a**. We can increment the pointer variable, shown as follows:

```
aptr++ (or) ++aptr
```

This statement moves the pointer to the next memory address. Similarly, we can decrement the pointer variable, as follows:

```
aptr-- (or) --aptr
```

This statement moves the pointer to the previous memory address. Also, if two pointer variables point to the same array they can be subtracted from each other.

We cannot perform pointer arithmetic on variables which are not stored in contiguous memory locations. Program 9.3 illustrates the arithmetic operations that we can perform with pointers.

Program 9.3 Arithmetic Operations on Pointers

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

void main()
{
    int num[]={56,75,22,18,90};
    int *ptr;
    int i;
    clrscr();
    cout << "The array values are:\n";
    for(i=0;i<5;i++)
        cout<< num[i]<<"\n";
    /*Allocating the base address of num to ptr*/
    ptr = num;
    /* Printing the value in the array */
    cout << "\nValue of ptr : "<< *ptr;
```

(Contd.)

```

        cout << "\n";
        ptr++;
        cout<<"\nValue of ptr++ : "<< *ptr;
        cout << "\n";
        ptr--;
        cout<<"\nValue of ptr-- : "<< *ptr;
        cout << "\n";
        ptr=ptr+2;
        cout<<"\nValue of ptr+2 : "<< *ptr;
        cout << "\n";
        ptr=ptr-1;
        cout <<"\nValue of ptr-1: "<< *ptr;
        cout << "\n";
        ptr+=3;
        cout<<"\nValue of ptr+=3: "<< *ptr;
        ptr-=2;
        cout << "\n";
        cout<<"\nValue of ptr-=2: "<< *ptr;
        cout << "\n";
        getch();
    }

```

The output of Program 9.3 would be

```

The array values are:
56
75
22
18
90
Value of ptr      : 56
Value of ptr++    : 75
Value of ptr--    : 56
Value of ptr+2    : 22
Value of ptr-1    : 75
Value of ptr+=3   : 90
Value of ptr-=2   : 22

```

9.2.4 Using Pointers with Arrays and Strings

Pointer is one of the efficient tools to access elements of an array. Pointers are useful to allocate arrays dynamically, i.e., we can decide the array size at run time. To achieve this, we use the functions, namely **malloc()** and **calloc()**, which we already discussed in Chapter 3. Accessing an array with pointers is simpler than accessing the array index.

In general, there are some differences between pointers and arrays; arrays refer to a block of memory space, whereas pointers do not refer to any section of memory. The memory addresses of arrays cannot be changed, whereas the content of the pointer variables, such as the memory addresses that it refers to, can be changed.

Even though there are subtle differences between pointers and arrays, they have a strong relationship between them.



Note *There is no error checking of array bounds in C++. Suppose we declare an array of size 25. The compiler issues no warnings if we attempt to access 26th location. It is the programmer's task to check the array limits.*

We can declare the pointers to arrays as follows:

```
int *nptr;
nptr=number[0];
```

Or

```
nptr=number;
```

Here, **nptr** points to the first element of the integer array, number[0]. Also, consider the following example:

```
float *fptr;
fptr=price[0];
```

Or

```
fptr=price;
```

Here, **fptr** points to the first element of the array of float, price[0]. Program 9.4 demonstrates how linear search is performed in an array using pointers.

Program 9.4 Pointers with Arrays

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

using namespace std;

void main()
{
    int arr[10] = {1,99,4,37,88,3,19,45,62,87};
    int i,num,*ptr;

    /*Assigning the base address of array arr to ptr*/
    ptr=arr;

    cout<<"Enter the element to be searched: ";
    cin>>num;

    for(i=0;i<10;i++)
    {
        if(*ptr==num)
```

(Contd.)

```

    {
        cout<<"\n"<<num<<" is present in the array";
        break;
    }

    else if(i==9)
        cout<<"\n"<<num<<" is not present in the array";
        ptr++; //Incrementing the pointer to make it point
                //to the next array element
    }
    getch();
}

```

The output of Program 9.4 would be

```

Enter the element to be searched: 87
87 is present in the array

```

9.2.5 Arrays of Pointers

Similar to other variables, we can create an array of pointers in C++. The array of pointers represents a collection of addresses. By declaring array of pointers, we can save a substantial amount of memory space.

An array of pointers point to an array of data items. Each element of the pointer array points to an item of the data array. Data items can be accessed either directly or by dereferencing the elements of pointer array. We can reorganize the pointer elements without affecting the data items.

We can declare an array of pointers as follows:

```
int *inarray[10];
```

This statement declares an array of 10 pointers, each of which points to an integer. The address of the first pointer is inarray[0], and the second pointer is inarray[1], and the final pointer points to inarray[9]. Before initializing, they point to some unknown values in the memory space. We can use the pointer variable to refer to some specific values. Program 9.5 explains the implementation of array of pointers.

Program 9.5 Arrays of Pointers

```

#include <iostream.h>

#include <conio.h>
#include <string.h>
#include <ctype.h>

void main()
{
    int i=0;
    char *ptr[10] = {

```

(Contd.)

```

        "books",
        "television",
        "computer",
        "sports"
    };
    char str[25];
    clrscr();
    cout << "\n\n\nEnter your favorite leisure pursuit: " ;
    cin >> str;
    for(i=0; i<4; i++)
    {
        if(!strcmp(str, *ptr[i]))
        {
            cout << "\n\nYour favorite pursuit " << " is available
            here"
            << endl;
            break;
        }
    }
    if(i==4)
    cout << "\n\nYour favorite " << " not available here" <<
    endl;
    getch();
}

```

The output of Program 9.5 would be

```

Enter your favorite leisure pursuit: books
Your favorite pursuit is available here

```

9.2.6 Pointers and Strings

We have seen the usage of pointers with one dimensional array elements. However, pointers are also efficient to access two dimensional and multi-dimensional arrays in C++. There is a definite relationship between arrays and pointers. C++ also allows us to handle the special kind of arrays, i.e., strings with pointers.

We know that a string is one dimensional array of characters, which start with the index 0 and ends with the null character '\0' in C++. A pointer variable can access a string by referring to its first character. As we know, there are two ways to assign a value to a string. We can use the character array or variable of type char *. Let us consider the following string declarations:

```

char num[]="one";
const char *numptr= "one";

```

The first declaration creates an array of four characters, which contains the characters, 'o','n','e','\0', whereas the second declaration generates a pointer variable, which points to the first character, i.e., 'o' of the string. There are numerous string handling functions available in C++. All of these functions are available in the header file <cstring>.

Program 9.6 demonstrates how to search a substring within a main string

Program 9.6 Searching a Substring within a Main String

```
#include<iostream>
#include<conio.h>
#include<string>
using namespace std;
void main()
{
    int i,j;
    char str[] = "\nC++ is better than C";
    int len = strlen(str);
    char *substr = new char[len];
    cout<<"The main string is: "<<str;
    cout<<"\n\nEnter the substring to be searched: ";
    cin>>substr;

    int k,len2=strlen(substr);
    for(i=0;i<len;i++)
    {
        k=i;
        for(j=0;j<len2;j++)
        {
            if(str[k]==substr[j])
            {
                if(j==len2-1)
                {
                    cout<<"\nThe substring is present in the main
                        string";
                    getch();
                    exit(0);
                }
                k++;
            }
            else
                break;
        }
    }

    if(i==len)
        cout<<"\nThe substring is not present in the main string";
    getch();
}
```

The output of Program 9.6 would be

```
The main string is:
C++ is better than C
Enter the substring to be searched: bet
The substring is present in the main string
```

9.2.7 Pointers to Functions

Even though pointers to functions (or function pointers) are introduced in C, they are widely used in C++ for dynamic binding, and event-based applications. The concept of pointer to function acts as a base for pointers to members, which we have discussed in Chapter 5.

The pointer to function is known as callback function. We can use these function pointers to refer to a function. Using function pointers, we can allow a C++ program to select a function dynamically at run time. We can also pass a function as an argument to another function. Here, the function is passed as a pointer. The function pointers cannot be dereferenced. C++ also allows us to compare two function pointers.

C++ provides two types of function pointers; function pointers that point to static member functions and function pointers that point to non-static member functions. These two function pointers are incompatible with each other. The function pointers that point to the non-static member function requires hidden argument.

Like other variables, we can declare a function pointer in C++. It takes the following form:

```
data_type(*function_name)();
```

As we know, the `data_type` is any valid data types used in C++. The `function_name` is the name of a function, which must be preceded by an asterisk (*). The `function_name` is any valid name of the function.

Example:

```
int (*num_function(int x));
```

Remember that declaring a pointer only creates a pointer. It does not create actual function. For this, we must define the task, which is to be performed by the function. The function must have the same return type and arguments. Program 9.7 explains how to declare and define function pointers in C++.

Program 9.7 Pointers to Functions

```
#include <iostream.h>

typedef void (*FunPtr)(int, int);

void Add(int i, int j)
{
    cout << i << " + " << j << " = " << i + j;
}

void Subtract(int i, int j)
{
    cout << i << " - " << j << " = " << i - j;
}

void main()
{
    FunPtr ptr;
    ptr = &Add;
    ptr(1,2);
    cout << endl;
    ptr = &Subtract;
    ptr(3,2);
}
```

The output of Program 9.7 would be

```
1 + 2 = 3
3 - 2 = 1
```



9.3 POINTERS TO OBJECTS

We have already seen how to use pointers to access the class members. As stated earlier, a pointer can point to an object created by a class. Consider the following statement:

```
item x;
```

where **item** is a class and **x** is an object defined to be of type **item**. Similarly we can define a pointer **it_ptr** of type **item** as follows:

```
item *it_ptr;
```

Object pointers are useful in creating objects at run time. We can also use an object pointer to access the public members of an object. Consider a class **item** defined as follows:

```
class item
{
    int code;
    float price;

public:
    void getdata(int a, float b)
    {
        code = a;
        price = b;
    }

    void show(void)
    {
        cout << "Code : " << code << "\n";
        << "Price: " << price << "\n\n";
    }
};
```

Let us declare an **item** variable **x** and a pointer **ptr** to **x** as follows:

```
item x;
item *ptr = &x;
```

The pointer **ptr** is initialized with the address of **x**.

We can refer to the member functions of **item** in two ways, one by using the *dot operator* and the *object*, and another by using the *arrow operator* and the *object pointer*. The statements

```
x.getdata(100, 75.50);
x.show();
```

are equivalent to

```
ptr->getdata(100, 75.50);
ptr->show();
```


Since ***ptr** is an alias of **x**, we can also use the following method:

```
(*ptr).show();
```

The parentheses are necessary because the dot operator has higher precedence than the *indirection operator* *****.

We can also create the objects using pointers and **new** operator as follows:

```
item *ptr = new item;
```

This statement allocates enough memory for the data members in the object structure and assigns the address of the memory space to **ptr**. Then **ptr** can be used to refer to the members as shown below:

```
ptr -> show();
```

If a class has a constructor with arguments and does not include an empty constructor, then we must supply the arguments when the object is created.

We can also create an array of objects using pointers. For example, the statement

```
item *ptr = new item[10];           // array of 10 objects
```

creates memory space for an array of 10 objects of **item**. Remember, in such cases, if the class contains constructors, it must also contain an empty constructor.

Program 9.8 illustrates the use of pointers to objects.

Program 9.8 Pointers to Objects

```
#include <iostream>
using namespace std;
class item
{
    int code;
    float price;
public:
    void getdata(int a, float b)
    {
        code = a;
        price = b;
    }

    void show(void)
    {
        cout << "Code : " << code << "\n";
        cout << "Price: " << price << "\n";
    }
};

const int size = 2;

int main()
```

(Contd.)

```

{
    item *p = new item [size];
    item *d = p;
    int x, i;
    float y;

    for(i=0; i<size; i++)
    {
        cout << "Input code and price for item" << i+1;
        cin >> x >> y;
        p->getdata(x,y);
        p++;
    }

    for(i=0; i<size; i++)
    {
        cout << "Item:" << i+1 << "\n";
        d->show();
        d++;
    }
    return 0;
}

```

The output of Program 9.8 would be

```

Input code and price for item1 40 500
Input code and price for item2 50 600
Item:1
Code : 40
Price: 500
Item:2
Code : 50
Price: 600

```

In Program 9.8, we created space dynamically for two objects of equal size. But this may not be the case always. For example, the objects of a class that contain character strings would not be of the same size. In such cases, we can define an array of pointers to objects that can be used to access the individual objects. This is illustrated in Program 9.9.

Program 9.9 Array of Pointers to Objects

```

#include <iostream>
#include <cstring>

using namespace std;

class city
{
    protected:
        char *name;

```

(Contd.)

```

        int len;
    public:
        city()
        {
            len = 0;
            name = new char[len+1];
        }
        void getname(void)
        {
            char *s;
            s = new char[30];

            cout << "Enter city name:";
            cin >> s;
            len = strlen(s);
            name = new char[len + 1];
            strcpy(name, s);
        }
        void printname(void)
        {
            cout << name << "\n";
        }
};

int main()
{
    city *cptr[10]; // array of 10 pointers to cities

    int n = 1;
    int option;

    do
    {
        cptr[n] = new city; // create new city
        cptr[n]->getname();
        n++;
        cout << "Do you want to enter one more name?\n";
        cout << "(Enter 1 for yes 0 for no):";
        cin >> option;
    }
    while(option);

    cout << "\n\n";
    for(int i=1; i<=n; i++)
    {
        cptr[i]->printname();
    }

    return 0;
}

```

The output of Program 9.9 would be

```
Enter city name:Hyderabad
Do you want to enter one more name?
(Enter 1 for yes 0 for no);1
Enter city name:Secunderabad
Do you want to enter one more name?
(Enter 1 for yes 0 for no);1
Enter city name:Malkajgiri
Do you want to enter one more name?
(Enter 1 for yes 0 for no);0
```

```
Hyderabad
Secunderabad
Malkajgiri
```



9.4 this POINTER

C++ uses a unique keyword called **this** to represent an object that invokes a member function. **this** is a pointer that points to the object for which *this* function was called. For example, the function call **A.max()** will set the pointer **this** to the address of the object **A**. The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer **this** acts as an *implicit* argument to all the member functions. Consider the following simple example:

```
class ABC
{
    int a;
    ....
    ....
};
```

The private variable 'a' can be used directly inside a member function, like

```
a = 123;
```

We can also use the following statement to do the same job:

```
this->a = 123;
```

Since C++ permits the use of shorthand form **a = 123**, we have not been using the pointer **this** explicitly so far. However, we have been implicitly using the pointer **this** when overloading the operators using member function.

Recall that, when a binary operator is overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer **this**. One important application of the pointer **this** is to return the object it points to. For example, the statement

```
return *this;
```

inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare two or more objects inside a member function and return the invoking object as a result. Example:

```
person & person :: greater(person & x)
{
    if x.age > age
        return x;           // argument object
    else
        return *this;       // invoking object
}
```

Suppose we invoke this function by the call

```
max = A.greater(B);
```

The function will return the object **B** (argument object) if the age of the person **B** is greater than that of **A**, otherwise, it will return the object **A** (invoking object) using the pointer **this**. Remember, the dereference operator ***** produces the contents at the address contained in the pointer. A complete program to illustrate the use of **this** is given in Program 9.10.

Program 9.10 **this** Pointer

```
#include <iostream>
#include <cstring>

using namespace std;

class person
{
    char name[20];
    float age;
public:
    person(char *s, float a)
    {
        strcpy(name, s);
        age = a;
    }
    person & person :: greater(person & x)
    {
        if(x.age >= age)
            return x;
        else
            return *this;
    }
    void display(void)
    {
        cout << "Name: " << name << "\n"
              << "Age: " << age << "\n";
    }
}
```

(Contd.)

```

    }
};

int main()
{
    person P1("John", 37.50),
           P2("Ahmed", 29.0),
           P3("Hebber", 40.25);

    person P = P1.greater(P3);          // P3.greater(P1)
    cout << "Elder person is: \n";
    P.display();

    P = P1.greater(P2);                 // P2.greater(P1)
    cout << "Elder person is: \n";
    P.display();

    return 0;
}

```

The output of Program 9.10 would be

```

Elder person is:
Name: Hebber
Age: 40.25
Elder person is:
Name: John
Age: 37.5

```



9.5 POINTERS TO DERIVED CLASSES

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type-compatible with pointers to objects of a derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes. For example, if **B** is a base class and **D** is a derived class from **B**, then a pointer declared as a pointer to **B** can also be a pointer to **D**. Consider the following declarations:

```

B *cptr;    // pointer to class B type variable
B b;        // base object
D d;        // derived object
cptr = &b;  // cptr points to object b

```

We can make **cptr** to point to the object **d** as follows:

```

cptr = &d;  // cptr points to object d

```

This is perfectly valid with C++ because **d** is an object derived from the class **B**.

However, there is a problem in using **cptr** to access the public members of the derived class **D**. Using **cptr**, we can access only those members which are inherited from **B** and not the members that

originally belong to **D**. In case a member of **D** has the same name as one of the members of **B**, then any reference to that member by **cptr** will always access the base class member.

Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer *declared* as pointer to the derived type.

Program 9.11 illustrates how pointers to a derived object are used.

Program 9.11 Pointers to Derived Objects

```
#include <iostream>

using namespace std;

class BC
{
    public:
        int b;
        void show()
        { cout << "b = " << b << "\n"; }
};

class DC : public BC
{
    public:
        int d;
        void show()
        { cout << "b = " << b << "\n"
          << "d = " << d << "\n";
        }
};

int main()
{
    BC *bptr;                // base pointer
    BC base;
    bptr = &base;            // base address

    bptr->b = 100;            // access BC via base pointer
    cout << "bptr points to base object \n";
    bptr -> show();
    // derived class
    DC derived;
    bptr = &derived;         // address of derived object
    bptr -> b = 200;         // access DC via base pointer

    /* bptr -> d = 300;*/    // won't work
    cout << "bptr now points to derived object \n";
    bptr -> show();         // bptr now points to derived object
```

(Contd.)

```

/* accessing d using a pointer of type derived class DC */

DC *dptr;                // derived type pointer
dptr = &derived;
dptr->d = 300;

cout << "dptr is derived type pointer\n";
dptr -> show();

cout << "using ((DC *)bptr)\n";
((DC *)bptr) -> d = 400;
((DC *)bptr) -> show();

return 0;
}

```

The output of Program 9.11 would be

```

bptr points base object
b = 100
bptr now points to derived object
b = 200
dptr is derived type pointer
b = 200
d = 300
using ((DC *)bptr)
b = 200
d = 400

```



Note We have used the statement

```
bptr -> show();
```

two times. First, when **bptr** points to the base object, and second when **bptr** is made to point to the derived object. But, both the times, it executed **BC::show()** function and displayed the content of the base object. However, the statements

```

dptr -> show();
((DC *) bptr) -> show();    // cast bptr to DC type

```

display the contents of the **derived** object. This shows that, although a base pointer can be made to point to any number of derived objects, it cannot directly access the members defined by a derived class.



9.6 VIRTUAL FUNCTIONS

As mentioned earlier, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes. This

necessitates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the derived objects. But, we just discovered that a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. How do we then achieve polymorphism? It is achieved using what is known as 'virtual' functions.

When we use the same function name in both the base and derived classes, the function in base class is declared as *virtual* using the keyword **virtual** preceding its normal declaration. When a function is made **virtual**, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the **virtual** function. Program 9.12 illustrates this point.

Program 9.12 Virtual Functions

```
#include <iostream>
using namespace std;
class Base
{
    public:
        void display() {cout << "\n Display base ";}
        virtual void show() {cout << "\n show base";}
};
class Derived : public Base
{
    public:
        void display() {cout << "\n Display derived";}
        void show() {cout << "\n show derived";}
};

int main()
{
    Base B;
    Derived D;
    Base *bptr;

    cout << "\n bptr points to Base \n";
    bptr = &B;
    bptr -> display();           // calls Base version
    bptr -> show();             // calls Base version

    cout << "\n\n bptr points to Derived\n";
    bptr = &D;
    bptr -> display();           // calls Base version
    bptr -> show();             // calls Derived version

    return 0;
}
```

The output of Program 9.12 would be

```

bptr points to Base

Display base
Show base

bptr points to Derived

Display base
Show derived

```



Note When *bptr* is made to point to the object **D**, the statement

```
bptr -> display();
```

calls only the function associated with the **Base** (i.e., **Base::display()**), whereas the statement

```
bptr -> show();
```

calls the **Derived** version of **show()**. This is because the function **display()** has not been made **virtual** in the **Base** class.

One important point to remember is that, we must access **virtual** functions through the use of a pointer declared as a pointer to the base class. Why can't we use the object name (with the dot operator) the same way as any other member function to call the virtual functions? We can, but remember, run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

Let us take an example where **virtual** functions are implemented in practice. Consider a book shop which sells both books and video-tapes. We can create a class known as **media** that stores the title and price of a publication. We can then create two derived classes, one for storing the number of pages in a book and another for storing the playing time of a tape. Figure 9.2 shows the class hierarchy for the book shop.

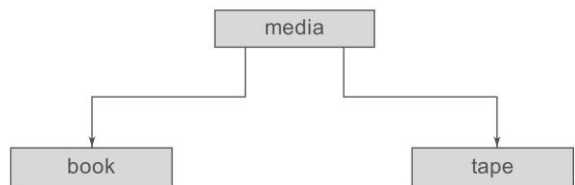


Fig. 9.2 The class hierarchy for the book shop

The classes are implemented in Program 9.13. A function **display()** is used in all the classes to display the class contents. Notice that the function **display()** has been declared **virtual** in **media**, the base class.

In the **main** program we create a heterogeneous list of pointers of type **media** as shown below:

```
media *list[2] = {&book1, &tape1};
```

The base pointers **list[0]** and **list[1]** are initialized with the addresses of objects **book1** and **tape1** respectively.

Program 9.13 Runtime Polymorphism

```
#include <iostream>
#include <cstring>

using namespace std;

class media
{
    protected:
        char title[50];
        float price;
    public:
        media(char *s, float a)
        {
            strcpy(title, s);
            price = a;
        }

        virtual void display() { } // empty virtual function
};

class book: public media
{
    int pages;
    public:
        book(char *s, float a, int p):media(s,a)
        {
            pages = p;
        }
        void display();
};

class tape :public media
{
    float time;
    public:
        tape(char * s, float a, float t):media(s, a)
        {
            time = t;
        }
        void display();
};

void book :: display()
{
    cout << "\n Title: " << title;
    cout << "\n Pages: " << pages;
    cout << "\n Price: " << price;
}
```

(Contd.)

```

void tape :: display()
{
    cout << "\n Title: " << title;
    cout << "\n play time: " << time << "mins";
    cout << "\n price: " << price;
}

int main()
{
    char * title = new char[30];
    float price, time;
    int pages;

    // Book details
    cout << "\n ENTER BOOK DETAILS\n";
    cout << " Title: "; cin >> title;
    cout << " Price: "; cin >> price;
    cout << " Pages: "; cin >> pages;

    book book1(title, price, pages);

    // Tape details
    cout << "\n ENTER TAPE DETAILS\n";
    cout << " Title: "; cin >> title;
    cout << " Price: "; cin >> price;
    cout << " Play time (mins): "; cin >> time;
    tape tapel(title, price, time);

    media* list[2];
    list[0] = &book1;
    list[1] = &tapel;

    cout << "\n MEDIA DETAILS";

    cout << "\n ..... BOOK ..... ";
    list[0] -> display(); // display book details

    cout << "\n ..... TAPE ..... ";

    list[1] -> display(); // display tape details

    result 0;
}

```

The output of Program 9.13 would be

```

ENTER BOOK DETAILS
Title:Programming_in_ANSI_C
Price: 88
Pages: 400

```

```

ENTER TAPE DETAILS
Title: Computing_Concepts
Price: 90
Play time (mins): 55

MEDIA DETAILS
..... BOOK .....
Title: Programming_in_ANSI_C
Pages: 400
Price: 88

..... TAPE .....
Title: Computing_Concepts
Play time: 55mins
Price: 90

```

Rules for Virtual Functions

When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements:

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.



9.7 PURE VIRTUAL FUNCTIONS

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a *placeholder*. For example, we have not defined any object of class media and therefore the function display() in the base class has been defined 'empty'. Such functions are called "do-nothing" functions.

A "do-nothing" function may be defined as follows:

```
virtual void display() = 0;
```

Such functions are called *pure virtual* functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called *abstract base classes*. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Program 9.14 demonstrates the use of pure virtual function

Program 9.14 Pure Virtual Function

```
#include <iostream>

using namespace std;

class Balagurusamy           //base class
{
public:
    virtual void example()=0;    //Denotes pure virtual Function Definition
};

class C:public Balagurusamy    //derived class 1
{
public:
    void example()
    {
        cout<<"C text Book written by Balagurusamy";
    }
};

class oops:public Balagurusamy //derived class 2
{
public:
    void example()
    {
        cout<<"C++ text Book written by Balagurusamy";
    }
};

void main()
{
    Exforsys* arra[2];

    C e1;
    oops e2;
    arra[0]=&e1;
    arra[1]=&e2;

    arra[0]->example();
    arra[1]->example();
}
```

The output of Program 9.14 would be

```
C text Book written by Balagurusamy
C++ text Book written by Balagurusamy
```



9.8 VIRTUAL CONSTRUCTORS AND DESTRUCTORS

“A constructor can not be virtual”. There are some valid reasons that justify this statement. First, to create an object the constructor of the object class must be of the same type as the class. But, this is not possible with a virtually implemented constructor. Second, at the time of calling a constructor, the virtual table would not have been created to resolve any virtual function calls. Thus, a virtual constructor itself would not have anywhere to look up to. As a result, it is not possible to declare a constructor as virtual.

A virtual destructor however, is pretty much feasible in C++. In fact, its use is often promoted in certain situations. One such situation is when we need to make sure that the different destructors in an inheritance hierarchy are called in order, particularly when the base class pointer is referring to a derived type object. It must be noted here that the order of calling of destructors in an inheritance hierarchy is opposite to that of constructors.

Consider the following class declarations that make use of destructors:

```
class A
{
    public:
        ~A()
        {
            //Base class destructor
        }
};

class B : public A
{
    public:
        ~B()
        {
            //Derived class destructor
        }
};

main()
{
    A *ptr = new B();
    .
    .
    delete ptr;
}
```

In the above class declarations, both base class A and the derived class B have their own destructors. Now, an A class pointer has been allocated a B class object. When this object pointer is deleted using the delete operator, it will trigger the base class destructor and the derived class destructor won't be called at all. This may lead to a memory leak situation. To make sure that the derived class destructor is mandatorily called, we must declare the base class destructor as virtual, as shown below.

```

class A
{
    public:
    virtual ~A()
    {
        //Base class destructor
    }
};
.
.

```

SUMMARY

- ☐ Polymorphism simply means one name having multiple forms.
- ☐ There are two types of polymorphism, namely, compile time polymorphism and run time polymorphism.
- ☐ Functions and operators overloading are examples of compile time polymorphism. The overloaded member functions are selected for invoking by matching arguments, both type and number. The compiler knows this information at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early or static binding or static linking. It means that an object is bound to its function call at compile time.
- ☐ In run time polymorphism, an appropriate member function is selected while the program is running. C++ supports run time polymorphism with the help of virtual functions. It is called late or dynamic binding because the appropriate function is selected dynamically at run time. Dynamic binding requires use of pointers to objects and is one of the powerful features of C++.
- ☐ Object pointers are useful in creating objects at run time. It can be used to access the public members of an object, along with an arrow operator.
- ☐ A **this** pointer refers to an object that currently invokes a member function. For example, the function call **a.show()** will set the pointer 'this' to the address of the object 'a'.
- ☐ Pointers to objects of a base class type are compatible with pointers to objects of a derived class. Therefore, we can use a single pointer variable to point to objects of base class as well as derived classes.
- ☐ When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. By making the base pointer to point to different objects, we can execute different versions of the virtual function.
- ☐ Run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. It cannot be achieved using object name along with the dot operator to access virtual function.
- ☐ We can have virtual destructors but not virtual constructors.
- ☐ If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, the respective calls will invoke the base class function.
- ☐ A virtual function, equated to zero is called a pure virtual function. It is a function declared in a base class that has no definition relative to the base class. A class containing such pure function is called an abstract class.

Key Terms

abstract base classes | 'address of' operator | argument object | arrays of pointers | arrow operator | base address | base object | base pointer | call back function | class hierarchy | compile time | compile time polymorphism | dereference operator | derived object | do-nothing function | dot operator | dynamic binding | early binding | function overloading | function pointer | implicit argument | indirection operator | invoking object | late binding | **new** operator | null pointers | object pointer | operator overloading | placeholder | pointers | pointer arithmetic | pointers to functions | polymorphism | pure virtual function | run time | run time polymorphism | static binding | static linking | **this** pointer | virtual constructors | virtual destructors | virtual function | void pointers

Review Questions

- 9.1 What does polymorphism mean in C++ language?
- 9.2 How is polymorphism achieved at (a) compile time, and (b) run time?
- 9.3 Discuss the different ways by which we can access public member functions of an object.
- 9.4 What is pointer to a function? Explain the situations in which such a concept is used.
- 9.5 Explain, with an example, how you would create space for an array of objects using pointers.
- 9.6 What does **this** pointer point to?
- 9.7 What are the applications of **this** pointer?
- 9.8 Can a pointer of base class type point to an object of the derived class? Explain.
- 9.9 What is a virtual function?
- 9.10 Why do we need virtual functions?
- 9.11 When do we make a virtual function "pure"? What are the implications of making a function a pure virtual function? F A Q
- 9.12 State which of the following statements are TRUE or FALSE.
 - (a) Virtual functions are used to create pointers to base classes.
 - (b) Virtual functions allow us to use the same function call to invoke member functions of objects of different classes.
 - (c) A pointer to a base class cannot be made to point to objects of derived class.
 - (d) **this** pointer points to the object that is currently used to invoke a function.
 - (e) **this** pointer can be used like any other pointer to access the members of the object it points to.
 - (f) **this** pointer can be made to point to any object by assigning the address of the object.
 - (g) Pure virtual functions force the programmer to redefine the virtual function inside the derived classes.
- 9.13 Distinguish between early binding and late binding. F A Q

Debugging Exercises

9.1 Identify the error in the following program.

```
#include <iostream.h>
class Info
{
    char *name;
    int number;
public:
    void getInfo()
    {
        cout << "Info::getInfo ";
        getName();
    }

    void getName()
    {
        cout << "Info::getName ";
    }
};
class Name: public Info
{
    char *name;
public:
    void getName()
    {
        cout << "Name::getName ";
    }
};

void main()
{
    Info *p;
    Name n;
    p = n;
    p->getInfo();
}
/*
```

9.2 Identify the error in the following program.

```
#include <iostream.h>
class Person
{
    int age;
public:
    Person()
    {
    }
    Person(int age)
```

```

    {
        this.age = age;
    }
    Person& operator < (Person &p)
    {
        return age < p.age ? p: *this;
    }
    int getAge()
    {
        return age;
    }
};
Void main ()
{
    Person P1 (15);
    Person P2 (11);
    Person P3;
    //if p1 is less than p2
    p3 = p1 < p2; p1. lessthan(p2)
    cout << p3.getAge();
}
/*

```

9.3 Identify the error in the following program.

```

#include "iostream.h"

class Human
{
public:
    Human()
    {
    }
    virtual ~Human()
    {
        cout << "Human::~~Human";
    }
};

class Student: public Human
{
public:
    Student()
    {
    }
    ~Student()
    {
        cout << "Student::~~Student()";
    }
};

```

```

void main()
{
    Human *H = new Student();
    delete H;
}

```

9.4 Correct the errors in the following program.

```

class test
{
    private:
        int m;
    public:
        void getdata()
        {
            cout << "Enter number:";
            cin >> m;
        }
        void display()
        {
            cout << m;
        }
};

main()
{
    test T;
    T->getdata();
    T->display();

    test *p;
    p = new test;
    p.getdata();
    (*p).display();
}

```

9.5 Debug and run the following program. What will be the output?

```

#include <iostream.h>
class A
{
    protected:
        int a,b;
    public:
        A(int x = 0, int y)
        {
            a = x;
            b = y;
        }
        virtual void print();
};
class B: public A

```

```

{
    private:
        float p,q;
    public:
        B(int m, int n, float u, float v)
        {
            p = u;
            q = v;
        }
        B() {p = q = 0;}
        void input(float u, float v);
        virtual void print(float);
};
void A::print(void)
{
    cout << A values: << a <<" "<< b <<"\n";
}
void B::print(float)
{
    cout <<B values:<< u <<" "<< v <<"\n";
}
void B::input(float x, float y)
{
    p = x;
    q = y;
}
main()
{
    A a1(10,20), *ptr;
    B b1;
    b1.input(7.5,3.142);
    ptr = &a1;
    ptr->print();
    ptr = &b1;
    ptr->print();
}

```

9.6 Test the output of the following program:

```

#include<iostream>
using namespace std;
class BC
{
    public:
        void show(void){cout<<"\nI am in base class..";}
};
class DC: public BC
{
    public:

```

```

        void show(void){cout<<"\nI am in derived class..";}
    };

    void main()
    {
        BC *ptr;
        DC dobj;
        ptr=&dobj;
        ptr->show();
    }

```

What would you change in the above program to trigger the show() method of the derived class instead of the base class?

Programming Exercises

- 9.1** Create a base class called **shape**. Use this class to store two **double** type values that could be used to compute the area of figures. Derive two specific classes called **triangle** and **rectangle** from the base **shape**. Add to the base class, a member function **get_data()** to initialize base class data members and another member function **display_area()** to compute and display the area of figures. Make **display_area()** as a virtual function and redefine this function in the derived classes to suit their requirements.

Using these three classes, design a program that will accept dimensions of a triangle or a rectangle interactively, and display the area.

Remember the two values given as input will be treated as lengths of two sides in the case of rectangles, and as base and height in the case of triangles, and used as follows:

Area of rectangle = $x * y$
 Area of triangle = $1/2 * x * y$ WEB

- 9.2** Extend the above program to display the area of circles. This requires addition of a new derived class 'circle' that computes the area of a circle. Remember, for a circle we need only one value, its radius, but the **get_data()** function in the base class requires two values to be passed. (Hint: Make the second argument of **get_data()** function as a default one with zero value.)
- 9.3** Run the above program with the following modifications:
- (a) Remove the definition of **display_area()** from one of the derived classes.
 - (b) In addition to the above change, declare the **display_area()** as **virtual** in the base class **shape**.

Comment on the output in each case.

- 9.4** Using the concept of pointers write a function that swaps the private data values of two objects of the same class type. WEB

UNIT 3

Linear Data Structures

- **Chapter 10:** Introduction to Algorithm and Data Structures
- **Chapter 11:** Arrays
- **Chapter 12:** Linked Lists
- **Chapter 13:** Stacks
- **Chapter 14:** Queues

INTRODUCTION TO ALGORITHM AND DATA STRUCTURES

- 10.1 Introduction
 - 10.2 Algorithms
 - 10.2.1 Characteristics of an Algorithm
 - 10.2.2 Representation of an Algorithm
 - 10.2.3 Efficiency of an Algorithm
 - 10.3 Asymptotic Notation
 - 10.3.1 Big-Oh Notation
 - 10.3.2 Omega Notation
 - 10.3.3 Theta Notation
 - 10.4 Introduction to Data Structures
 - 10.4.1 Characteristics of Data Structure
 - 10.5 Types of Data Structures
 - 10.5.1 Arrays
 - 10.5.2 Linked Lists
 - 10.5.3 Stacks
 - 10.5.4 Queues
 - 10.5.5 Trees
 - 10.5.6 Graphs
 - 10.6 Data Structure Operations
 - 10.6.1 Data Structure Efficiency
- Summary*
- Key Terms*





10.1 INTRODUCTION

In the last two chapters, we learnt the basics of C programming language and its various programming constructs. In this chapter, we will focus on one of the main starting points to developing programming applications, i.e., algorithms. An algorithm is a set of instructions that defines the complete solution to a given problem. It uses simple English language for writing the solution steps. It is quite possible to have multiple algorithmic solutions for the same problem. In such cases, performance becomes the sole criterion for choosing a specific solution. We can use various asymptotic notations, such as big-oh and omega for assessing the performance or running time of an algorithm.

In this chapter, we will also get a brief introduction to data structures. Data structure is a collection of data and the associated operations. Some of the commonly used data structures are arrays, stacks, queues, linked lists, etc.



10.2 ALGORITHMS

An algorithm can be defined as a step-by-step procedure that provides solution to a given problem. It comprises of a well-defined set of finite number of steps or rules that are executed sequentially to obtain the desired solution.

To understand algorithms in a better way, let us consider a simple problem of identifying the smallest number from a given list of numbers. Following is the algorithm for this problem:

Select the first number in the list and tag it as the smallest-so-far element.

1. For each subsequent element in the list.
2. Replace the smallest-so-far number with the list element if the latter is smaller.
3. Once all the numbers have been compared, the smallest-so-far number is considered as the smallest number in the list.

In computing terms, an algorithm is described a little differently. It is defined as a hierarchy of steps used for computational procedures, which usually starts with an input value and generates the desired output. While defining an algorithm, you must consider two primary factors, the time it requires to solve the problem and the required memory space. For instance, if an algorithm takes hours to solve a problem, then it is of no use. Similarly, if an algorithm requires gigabytes of computer memory, then also it is not considered an ideal algorithm.

Figure 10.1 shows a simple illustration of how algorithms are used for solving computational problems.

An algorithm solves only a single problem at a time. However, the same problem can be solved using multiple algorithms. The benefit of using multiple algorithms to solve the same problem is purely situational. One algorithm could be more efficient for a particular set of inputs or for a specific variation of the problem while another algorithm could be more efficient for some different set of inputs or for some different variation of the problem. The use of multiple

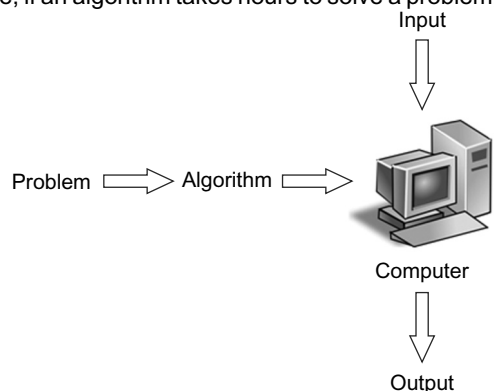


Fig. 10.1 Use of algorithms for solving computational problems

algorithms is more evident while solving sorting problems. One sorting algorithm could be efficient for sorting a large collection of integers while another sorting algorithm could be more efficient for sorting a large collection of strings. Thus, in this case the choice of a particular algorithm solely depends on the type of input values.

10.2.1 Characteristics of an Algorithm

There are certain key characteristics that an algorithm must possess. These characteristics are:

1. An algorithm must comprise of a **finite** number of steps.
2. It should have zero or more valid and clearly defined **input** values.
3. It should be able to generate at least a single valid **output** based on a valid input.
4. It must be **definite**, i.e., each instruction in the algorithm should be defined clearly.
5. It should be **correct**, i.e., it should be able to perform the desired task of generating correct output from the given input.
6. There should be no **ambiguity** regarding the order of execution of algorithm steps.
7. It should be able to **terminate** on its own, i.e., it should not go into an infinite loop.

10.2.2 Representation of an Algorithm

You can represent an algorithm in a number of ways, right from normal English language phrases to graphical representation using flow charts. However, such representations are mainly useful when the algorithm is simple and small.

Another way of representing an algorithm is the pseudocode. Pseudocode is an informal representation of the algorithm that provides a complete outline of a program so that the programmers can easily understand it and transform it into a program using the programming language of their choice. The structure and syntax of pseudocode is quite similar to typical programming language constructs, thus it is easy to transform it into a program. Since there are no tight syntactical constraints associated with developing a pseudocoded algorithm, the programmer has the liberty to focus only on getting the solution logic right.

While representing an algorithm in pseudocode form, you must use certain conventions consistently throughout the algorithm. This helps in easy understanding of the algorithm. Following are some of the general conventions that are followed while writing pseudocode:

1. Provide a valid name for the algorithm written using pseudocode.
2. For each line of instruction, specify a line number.
3. Always begin an identifier name with English alphabet.
4. It is not necessary to explicitly specify the data type of the variables.
5. Always indent the statements present inside a block structure appropriately.
6. Use **read** and **write** instructions to specify input and output operations respectively.
7. Use **if** or **if else** constructs for conditional statements. You must end an **if** statement with the corresponding **end if** statement. Further, each **if** construct should be vertically aligned, depicted as follows:

```
If (conditional expression)
    Statement
end-if
```

Or

```

if (conditional expression)
    Statement
else
    Statement
end-if

```

8. For looping or iterative statements, you can use **for** or **while** looping constructs. A **for** loop must end with an **end for** statement while a **while** loop must end with an **end while** statement, as depicted below:

```

for i = 1 to 10 do
{
    Statement 1
    .
    .
    Statement n
}
end-for

```

```

while (conditional expression) do
{
    Statement 1
    .
    .
    Statement n
}
end-while

```

9. Use logical and relational operators whenever logical or relational operations are to be performed. For example,

```

i = j
i < j
i ≥ j

```

10. Represent an array or list element by specifying the name of the array followed by its index within square brackets. For instance, $A[i]$ will represent the i^{th} element of the array A.

Let us now go through some examples of algorithms created using pseudocode.

Example 10.1 Algorithm to Interchange Two Numbers

Interchange (X, Y)

```

Step 1: Begin
Step 2: Set X = X + Y
Step 3: Set Y = X - Y
Step 4: Set X = X - Y
Step 5: Write (X, Y)
Step 6: End

```

Example 10.2 Algorithm to Calculate Average of 15 Numbers**Average (avg, sum)**

```
Step 1: Begin
Step 2: Set avg = 0.0 and sum = 0
Step 3: for i = 1 to 15 do
Step 4: Read (a)
Step 5: sum = sum + a
Step 6: end-for
Step 7: avg = sum/15
Step 8: Write (avg)
Step 9: End
```

Example 10.3 Algorithm to Sort n Numbers

```
Sort (a, n)
Step 1: Begin
Step 2: Read (n)
Step 3: for i = n to 2 do
Step 4: for j = 1 to i-1 do
Step 5: if a[j]>a[j+1] then
Step 6: Interchange a[j] and a[j+1]
Step 7: end-if
Step 8: end-for
Step 9: end-for
Step 10: End
```

10.2.3 Efficiency of an Algorithm

Whenever we refer the term efficiency in the context of algorithms, it points at two aspects: one, whether the algorithm runs faster; and two, whether it uses lesser amount of memory space. Thus, an efficient algorithm will always create the best possible tradeoff between its running time and memory space consumption.

The function that derives the running time of an algorithm and its memory space requirements for a given set of inputs is referred as algorithm complexity. Time complexity is the measure of the running time of an algorithm for a given set of inputs. Space complexity is the measure of the amount of memory space required by an algorithm for its complete execution, for a given set of inputs.

Time complexity is typically measured by counting the number of primitive or elementary steps performed by the algorithm for its complete execution. These steps are machine independent and their count is directly dependent on the size of input data set. The representation or expression of time complexity is done asymptotically, as we shall see in the subsequent section.



10.3 ASYMPTOTIC NOTATION

Asymptotic notation is the most simple and easiest way of describing the running time of an algorithm. It represents the efficiency and performance of an algorithm in a systematic and meaningful manner.

Asymptotic notations describe time complexity in terms of three common measures, best case (or 'fastest possible'), worst case (or 'slowest possible'), and average case (or 'average time').

The three most important asymptotic notations are

1. Big-Oh notation
2. Omega notation
3. Theta notation

10.3.1 Big-Oh Notation

The big-oh notation is a method that is used to express the upper bound of the running time of an algorithm. It is denoted by 'O'. Using this notation, we can compute the maximum possible amount of time that an algorithm will take for its completion.

Definition Consider $f(n)$ and $g(n)$ to be two positive functions of n , where n is the size of the input data. Then, $f(n)$ is big-oh of $g(n)$, if and only if there exists a positive constant C and an integer n_0 , such that

$$f(n) \leq Cg(n) \text{ and } n > n_0$$

Here, $f(n) = O(g(n))$

Figure 10.2 shows the graphical representation of big-oh notation.

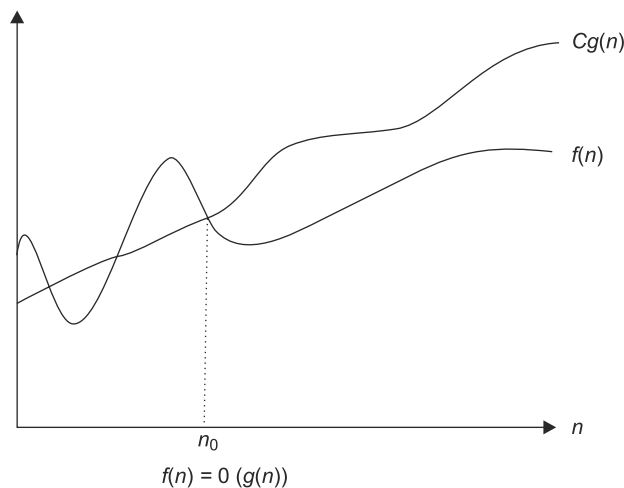


Fig. 10.2 Graphical representation of big-oh notation

Some of the typical complexities (computing time) represented by big-oh notation are

1. $O(1) \rightarrow$ Constant
2. $O(n) \rightarrow$ Linear
3. $O(n^2) \rightarrow$ Quadratic
4. $O(n^3) \rightarrow$ Cubic
5. $O(2^n) \rightarrow$ Exponential
6. $O(\log n) \rightarrow$ Logarithmic

Example 10.4 Derive the big-oh notation, if $f(n) = 8n + 7$ and $g(n) = n$

To show $f(n)$ is $O(g(n))$, we must consider positive constants C and integer n_0 , such that

$$f(n) \leq Cg(n) \text{ for all } n > n_0$$

$$\text{or } 8n + 7 \leq Cn \text{ for all } n > n_0$$

Let $C = 15$.

Now, we must show that $8n + 7 \leq 15n$.

$$\text{or } 7 \leq 7n$$

$$\text{or } 1 \leq n$$

Therefore, $f(n) = 8n + 7 \leq 15n$ for all $n \geq 1$, where $C = 15$ and $n_0 = 1$.

Hence, $f(n) = O(g(n))$.

Example 10.5 Derive the big-oh notation, if $f(n) = 2n + 2$ and $g(n) = n^2$

Given, $f(n) = 2n + 2$ and $g(n) = n^2$.

For $n = 1$,

$$f(n) = 2(1) + 2$$

$$= 4$$

$$g(n) = (1)^2$$

$$= 1$$

i.e., $f(n) > g(n)$

For $n = 2$,

$$f(n) = 2(2) + 2$$

$$= 6$$

$$g(n) = (2)^2$$

$$= 4$$

i.e., $f(n) > g(n)$

For $n = 3$,

$$f(n) = 2(3) + 2$$

$$= 8$$

$$g(n) = (3)^2$$

$$= 9$$

i.e., $f(n) < g(n)$

Therefore, $f(n) \leq Cg(n)$ is true if $n > 2$.

10.3.2 Omega Notation

The omega notation is a method that is used to express the lower bound of the running time of an algorithm. Omega notation is denoted by ' Ω '. Using this notation, you can compute the minimum amount of time that an algorithm will take for its completion.

Definition Consider $f(n)$ and $g(n)$ to be two positive functions of n , where n is the size of the input data. Then, $f(n)$ is omega of $g(n)$, if and only if there exists a positive constant C and an integer n_0 , such that

$$f(n) \leq Cg(n) \text{ and } n > n_0$$

Here, $f(n) = \Omega(g(n))$

Figure 10.3 shows the graphical representation of omega notation.

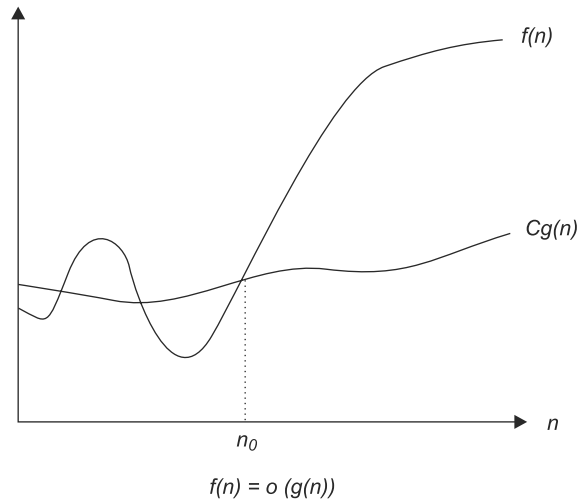


Fig. 10.3 Graphical representation of Omega notation

Example 10.6 Deduce the omega notation if $f(n) = 2n^2 + 4$ and $g(n) = 6n$.

Given, $f(n) = 2n^2 + 4$ and $g(n) = 6n$.

For $n = 0$,

$$\begin{aligned} f(n) &= 2(0)^2 + 4 \\ &= 4 \end{aligned}$$

$$\begin{aligned} g(n) &= 6(0) \\ &= 0 \end{aligned}$$

i.e., $f(n) > g(n)$

For $n = 1$,

$$\begin{aligned} f(n) &= 2(1)^2 + 4 \\ &= 2 + 4 \\ &= 6 \end{aligned}$$

$$\begin{aligned} g(n) &= 6(1) \\ &= 6 \end{aligned}$$

i.e., $f(n) = g(n)$

For $n = 2$,

$$\begin{aligned} f(n) &= 2(2)^2 + 4 \\ &= 8 + 4 \\ &= 12 \end{aligned}$$

$$\begin{aligned} g(n) &= 6(2) \\ &= 12 \end{aligned}$$

i.e., $f(n) = g(n)$

(Contd.)

For $n = 3$,
 $f(n) = 2(3)^2 + 4$
 $= 18 + 4$
 $= 22$
 $g(n) = 6(3)$
 $= 18$
 i.e., $f(n) > g(n)$
 Therefore, we can say that $f(n) > Cg(n)$, if $n > 2$.

Example 10.7 Deduce the omega notation if $f(n) = 2n + 6$ and $g(n) = 2n$.

Given, $f(n) = 2n + 6$ and $g(n) = 2n$.
 For $n = 0$,
 $f(n) = 2(0) + 6$
 $= 6$
 $g(n) = 2(0)$
 $= 0$
 i.e., $f(n) > g(n)$
 For $n = 1$,
 $f(n) = 2(1) + 6$
 $= 2 + 6$
 $= 8$
 $g(n) = 2(1)$
 $= 2$
 i.e., $f(n) > g(n)$
 Therefore, we can say that $f(n) > Cg(n)$, for $n > 1$.

10.3.3 Theta Notation

The theta notation is a method that is used to express the running time of an algorithm between the lower and upper bounds. Theta notation is denoted by ' θ '. Using this notation, we can compute the average time that an algorithm will take for its completion.

Definition Consider $f(n)$ and $g(n)$ to be two positive functions of n , where n is the size of the input data. Then, $f(n)$ is theta of $g(n)$, if and only if there exists two positive constants C_1 and C_2 , such that,

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

Here, $f(n) = \theta(g(n))$.

Figure 3.4 shows the graphical representation of theta notation.

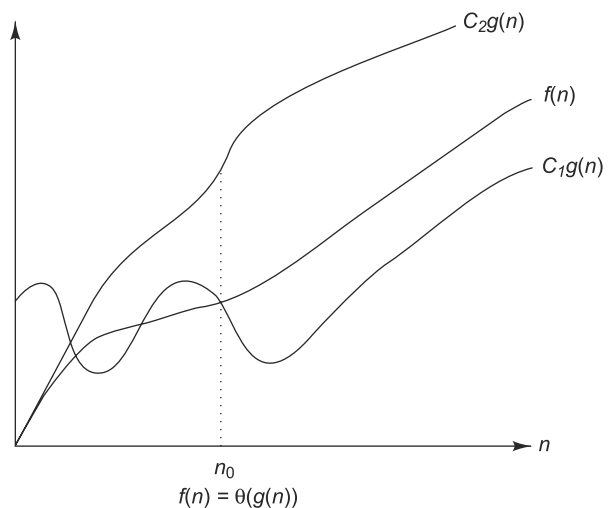


Fig. 10.4 Graphical representation of Theta notation

Example 10.8 Deduce the theta notation if $f(n) = 2n + 8$.

Let $f(n) = 2n + 8 > 5n$ where $n \geq 2$

Similarly, $f(n) = 2n + 8 > 6n$ where $n \geq 2$

and $f(n) = 2n + 8 < 7n$ where $n \geq 2$

Thus, $5n < 2n + 8 < 7n$, for $n \geq 2$,

Here $C_1 = 5$ and $C_2 = 7$

Hence, $f(n) = 2n + 8 = \theta(n)$



10.4 INTRODUCTION TO DATA STRUCTURES

In simple terms, data structure can be defined as a representation of data along with its associated operations. It is the way of organizing and storing data in a computer system so that it can be used efficiently. This organization can be in the form of a group of data elements stored under one name. Here, the data elements are referred as members of the data structure.

Depending on the type of data structures, the members can be of different types and lengths. Some of the examples of data structures include arrays, linked lists, binary trees, stacks, etc. Algorithms are used to manipulate the data structures in a number of different ways, like sorting the data elements or searching a particular data item.

The design and implementation of a typical data structure is associated with the definition of the operations that can be performed on the data structure. The specification of these data structure operations is done with the help of algorithms.

10.4.1 Characteristics of Data Structure

Data structures help in storing, organizing, and analyzing the data in a logical manner. The following points highlight the need of data structures in computer science:

1. It depicts the logical representation of data in computer memory.
2. It represents the logical relationship between the various data elements.
3. It helps in efficient manipulation of stored data elements.
4. It allows the programs to process the data in an efficient manner.



10.5 TYPES OF DATA STRUCTURES

Data structures are primarily divided into two classes, primitive and non-primitive. Primitive data structures include all the fundamental data structures that can be directly manipulated by machine-level instructions. Some of the common primitive data structures include integer, character, real, boolean, etc. Non-primitive data structures, on the other hand, refer to all those data structures that are derived from one or more primitive data structures. The objective of creating non-primitive data structures is to form sets of homogeneous or heterogeneous data elements.

Non-primitive data structures are further categorized into two types: linear and non-linear. In linear data structures, all the data elements are arranged in a linear or sequential fashion. Examples of linear data structures include arrays, stacks, queues, linked lists, etc. In non-linear data structures, there is no definite order or sequence in which data elements are arranged. For instance, a non-linear data structure could arrange data elements in a hierarchical fashion. Examples of non-linear data structures are trees and graphs.

Figure 10.5 shows the classification of different types of data structures.

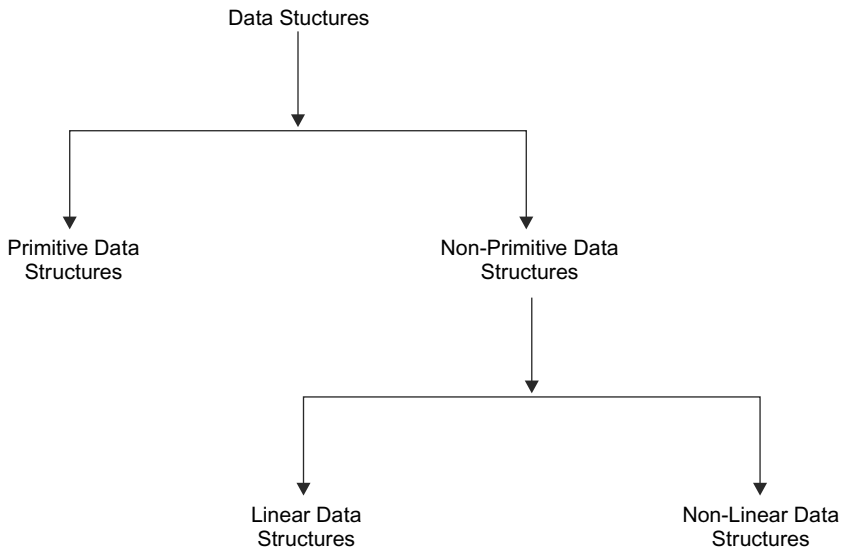


Fig. 10.5 Types of data structures

The subsequent sections give a brief overview of some of the important data structures. Each of these data structures will be covered in detail in later chapters.

10.5.1 Arrays

An array is a collection of similar type data elements stored at consecutive locations in the memory. Typical examples of arrays include list of integers, group of names, etc. The group of array elements is referred with a common name called *array name*. Access to individual array elements is provided with the help of an index identifier. In C language, array index starts with 0. For example, list[5] refers to the 6th element of the array 'list'.

Figure 10.6 shows the logical representation of arrays.

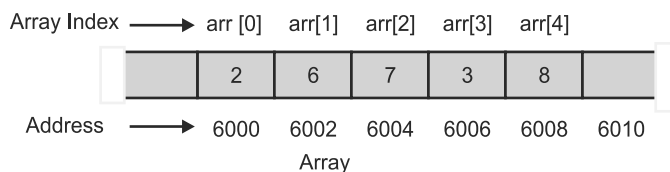


Fig. 10.6 Logical representation of arrays

Application of Arrays Arrays are particularly used in programs that require storing large collection of similar type data elements.

Note For more information on arrays, refer to Chapter 11.

10.5.2 Linked Lists

Linked list is a data structure used for storing data in the form of a list. It comprises of multiple nodes connected to each other through pointers. Each node comprises of two parts. One part contains the data value while the other part contains a pointer to the next node in the list.

Linked lists eliminate one of the main disadvantages associated with arrays, that is inefficient utilization of memory space. A linked list blocks only that much amount of memory space as is required for storing its constituent data elements. Every time a new element is to be inserted into the linked list, a corresponding new node is created. This is in contrast to arrays, which block a fixed amount of memory space irrespective of their precise requirement.

Figure 10.7 shows the logical representation of a linked list.

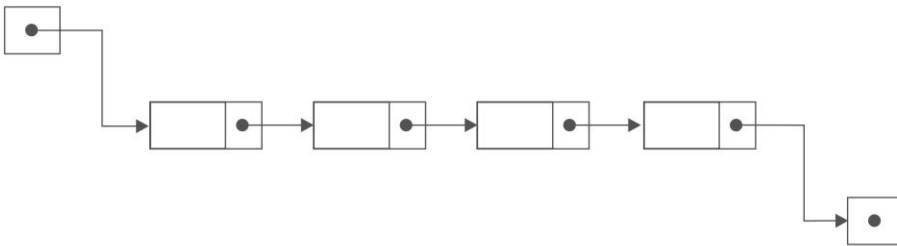


Fig. 10.7 Logical representation of linked lists

Application of Linked Lists Linked lists are used in situations where there is a need for dynamic memory allocation. For instance, a number of data structures like stacks, queues, trees, etc., are implemented with the help of linked lists.

Note For more information on linked lists, refer to Chapter 12.

10.5.3 Stacks

Stack is a linear data structure that maintains a list of elements in such a manner that elements can be inserted or deleted only from one end of the list. This end is referred as top of the stack. Stack is based on the Last-In-First-Out (LIFO) principle, which means the element that is last added to the stack is the one that is first removed from the stack. A stack of books can be considered similar to a stack data structure as it allows the books to be added or removed only from the top end of the stack and not from the middle.

Figure 10.8 shows the logical representation of a stack.

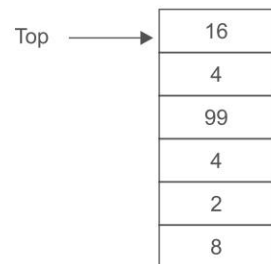


Fig. 10.8 Logical representation of stacks

Application of Stacks Stacks are typically used in the implementation of system processes, such as compilation and program control.

Note For more information on stacks, refer to Chapter 13.

10.5.4 Queues

Queue is a linear data structure that maintains a list of elements in such a manner that elements are inserted from one end of the queue (called *rear*) and deleted from the other end (called *front*). Queue is based on the First-In-First-Out (FIFO) principle, which means the element that is first added to the queue is also the one that is first removed from the queue. A queue of people standing at a bus stop can be considered similar to a queue data structure as people join the queue at the back and leave the queue from the front.

Figure 10.9 shows the logical representation of a queue.

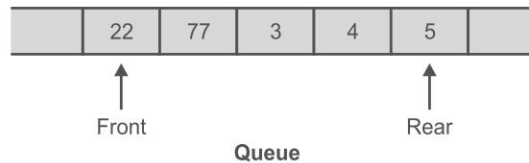


Fig. 10.9 Logical representation of queues

Application of Queues Queues are typically used in the implementation of key system processes such as CPU scheduling, resource sharing, etc.

Note For more information on queues, refer to Chapter 14.

10.5.5 Trees

Tree is a linked data structure that arranges its nodes in the form of a hierarchical tree structure. Each node comprises of zero or more child nodes. The node present at the top of the tree structure is referred as root node. Data is accessed from the tree data structure through various tree traversal methods.

Figure 10.10 shows the representation of a tree.

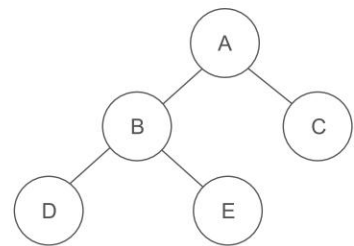


Fig. 10.10 Tree

Application of Trees Tree data structure is typically used for storing hierarchical data, implementing search trees, and maintaining sorted data.

Note For more information on trees, refer to Chapter 15.

10.5.6 Graphs

Graph is a linked data structure that comprises of a group of vertices called *nodes* and a group of edges. An edge is nothing but a pair of vertices. Graph data structure realizes the mathematical concept of graphs. The edges of a graph are typically associated with certain values also called weights. This helps to compute the cost of traversing the graph through a certain path.

Figure 10.11 shows the representation of a graph.

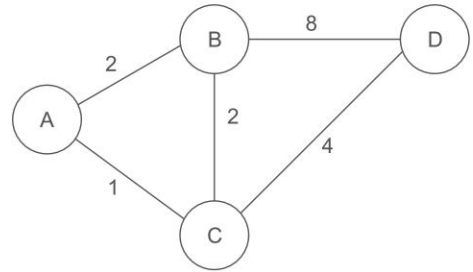


Fig. 10.11 Graph

Application of Graphs One of the typical application areas of graphs is in the modelling of networking systems. It helps to compute the cost of transmitting data from a particular network path.



Note

For more information on trees, refer to Chapter 16.



10.6 DATA STRUCTURE OPERATIONS

There are several common operations associated with data structures that are used for manipulating the stored data. While defining a data structure, you also need to define these associated operations. The following operations are most frequently performed on any data structure type:

1. **Traversing** It is the process of accessing each record of a data structure exactly once.
2. **Searching** It is the process of identifying the location of a record that contains a specific key value.
3. **Inserting** It is the process of adding a new record in to a data structure.
4. **Deleting** It is the process of removing an existing record from a data structure.

Apart from these typical data structure operations, there are some other operations associated with data structures, such as

1. **Sorting** It is the process of arranging the records of a data structure in a specific order, such as alphabetical, ascending, or descending.
2. **Merging** It is the process of combining the records of two different sorted data structures to produce a single sorted data set.

10.6.1 Data Structure Efficiency

Table 10.1 lists the efficiency of various data structure types for different operations.

Table 10.1 Efficiency of various data structure types

Data Structure	Insert	Search	Delete
Array	$O(n)$	$O(n)$	$O(n)$
Linked List	$O(1)$	$O(n)$	$O(n)$

(Contd.)

Data Structure	Insert	Search	Delete
Stack	$O(1)$	–	$O(1)$
Queue	$O(1)$	–	$O(1)$
Tree (Sorted)	$O(\log n)$	$O(\log n)$	$O(\log n)$

SUMMARY

- ☐ An algorithm is a well-defined set of finite number of steps or rules that are executed sequentially to obtain the desired solution.
- ☐ Algorithms can be represented in the form of pseudocode or flowcharts.
- ☐ The function that derives the running time of an algorithm and its memory space requirements for a given set of inputs is referred as algorithm complexity.
- ☐ Asymptotic notations describe time complexity in terms of three common measures, best case, worst case, and average case.
- ☐ The various types of asymptotic notations are big-oh, omega, and theta.
- ☐ Data structure is the way of organizing and storing data in a computer system.
- ☐ Data structures are primarily divided into two classes, primitive and non-primitive.
- ☐ Primitive data structures include all the fundamental data structures such as integer, character, real, etc.
- ☐ Non-primitive data structures are the ones that are derived from one or more primitive data structures. Examples of non-primitive data structures include arrays, linked lists, stacks, queues, etc.
- ☐ An array is a collection of similar type data elements stored at consecutive locations in the memory.
- ☐ Linked list is a collection of nodes connected to each other through pointers.
- ☐ Stack is a linear data structure that stores the data elements based on Last-In-First-Out (LIFO) principle, which means the element that is last added to the stack is the one that is first removed from the stack.
- ☐ Queue is a linear data structure that stores the data elements based on First-In-First-Out (FIFO) principle, which means the element that is first added to the stack is also the one that is first removed from the stack.
- ☐ Tree is a linked data structure that arranges its nodes in the form of a hierarchical tree structure.
- ☐ Graph is a linked data structure that comprises of a group of vertices called nodes and a group of edges.

Key Terms

pseudocode | time complexity | space complexity | big-oh | omega | theta | LIFO | FIFO | root node |

Review Questions

- 10.1** What is an algorithm? Explain with the help of an example.
- 10.2** List the characteristics of an algorithm.
- 10.3** How are algorithms represented? Explain with the help of an example.
- 10.4** What is algorithm complexity?
- 10.5** What is an asymptotic notation? Explain the various types of asymptotic notations.
- 10.6** What is a data structure? What are its various characteristics?
- 10.7** What are the different types of data structures?
- 10.8** Explain any two non-primitive data structures.
- 10.9** What is a tree? Why is it used?
- 10.10** List the typical operations associated with derived data structure types.

11

ARRAYS

CHAPTER OUTLINE

- 11.1 Introduction
- 11.2 Types of Arrays
- 11.3 Representation of one-dimensional Array in Memory
- 11.4 Array Traversal
- 11.5 Insertion and Deletion
 - 11.5.1 Insertion
 - 11.5.2 Deletion
- 11.6 Sorting and Searching
 - 11.6.1 Sorting
 - 11.6.2 Searching
- 11.7 Representation of Multi-dimensional Array in Memory
- 11.8 Realizing Matrices using Two-dimensional Arrays
- 11.9 Matrix Operations
 - 11.9.1 Addition
 - 11.9.2 Subtraction
 - 11.9.3 Multiplication
 - 11.9.4 Transpose

Summary

Key Terms





11.1 INTRODUCTION

In Chapter 4, we briefly introduced one of the most important and commonly used derived data types, called *array*. In this chapter, we will observe how array is used as a data structure in different programming situations. We will also get familiar with the logical representation of arrays in memory.

An array is regarded as one of the most fundamental entities for storing logical groups of data. It also forms the basis for implementing some advanced data structures, like stacks and queues, as we shall see in the later chapters.

Typically, an array is defined as a collection of same type elements. That means, it can store a group of integers, characters, strings, structures, and so on. However, an array cannot store different type elements like a group of integers and fractions, or a group of strings and integers. Following are some of the characteristics associated with arrays:

1. It uses a single name for referencing all the array elements. The differentiation between any two elements is made on the basis of index value.
2. It stores the different elements at consecutive memory locations.
3. Its name can be used as a pointer to the first array element.
4. Its size is always a constant expression and not a variable.
5. It does not perform bound checking on its own. It has to be implemented programmatically.

Before we delve further into exploring array as a data structure, let us first identify the different types of arrays.



11.2 TYPES OF ARRAYS

As already explained, an array is a logical grouping of same type data elements. Now, it is quite possible that each of these elements is itself an array. Further, each of the elements of the sub array could also be an array. This forms the basis of characterizing an array into different types, as depicted in Table 11.1.

Table 11.1 *Types of arrays*

<i>Array Type</i>	<i>Description</i>	<i>C Representation</i>	<i>Example</i>
One-dimensional array	It is a group of same type data elements, such as integers, floats, or characters.	<code>array[]</code>	A group of integers. {2, 5, 6, 1, 9}
Multi-dimensional array	It is a group of data elements, where each element is itself an array.	<code>array[][]..[]</code>	A group of strings. {"Ajay", "Vikas", "Amit", "Sumit"}

The various instances of multi-dimensional arrays are two-dimensional (2D) array, three-dimensional (3D) array, four-dimensional (4D) array, and so on. The choice of a particular multi-dimensional array depends on the programming situation at hand. For instance, if we are required to realize a 3×3 matrix programmatically, then we can do so by declaring a two dimensional array, say `M[3][3]`. Here, each dimension of the array *M* signifies the row and column of the matrix respectively. Multi-dimensional arrays are covered in greater detail later in this chapter. For now, let us focus on implementing and manipulating one-dimensional array.



11.3 REPRESENTATION OF ONE-DIMENSIONAL ARRAY IN MEMORY

The elements of a one-dimensional array are stored at consecutive locations in memory. Each of the locations is accessed with the help of array index identifier to retrieve the corresponding element.

Consider the following integer array:

```
arr[5] = {2, 6, 7, 3, 8}
```

Here, **arr** is a five-element integer array. Figure 11.1 shows the representation of array **arr** in memory:

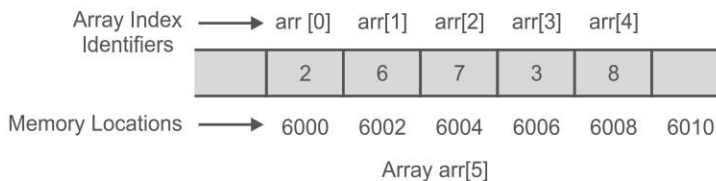


Fig. 11.1 Array representation in memory

As shown in Fig. 11.1, each array element is stored at consecutive memory locations, i.e., 6000, 6002, 6004, and so on. The location of the first element, i.e., 6000 is also referred as the base address of the array.

If we know the base address of an array, then we can find the location of its individual elements by using a simple formula, which is

$$\text{Address of } A[k] = B + W * k$$

Here,

A[] is the array.

B is the base address, i.e., the address of the first element.

W is the word size or the size of an array element.

k is the index identifier.

For instance, the address of the third element of array **arr** stored at index location 2 would be

$$\begin{aligned} \text{Address of } arr[2] &= 6000 + 2 * 2 \\ &= 6000 + 4 \\ &= 6004 \end{aligned}$$



Note The word size of a data type is decided by the programming language being used and the hardware specifications.



11.4 ARRAY TRAVERSAL

While working with arrays, it is often required to access the array elements; that is, reading values from the array. This is achieved with the help of array traversal. It involves visiting the array elements and storing or retrieving values from it. Some of the typical situations where array traversal may be required are:



Check Point

1. What is a base address?

Ans. It is the memory address of the first element of an array.

2. How are array elements stored in memory?

Ans. The elements of an array are stored at consecutive locations in memory.



Mind Jog

What is 'array index out of bound'?

It is a runtime error that is encountered when a program tries to reference an address location outside of the defined array limits.

Printing array elements,
Searching an element in the array,
Sorting an array, and so on

Example 11.1 Algorithm to Sequentially Traverse an Array

```
traverse(arr[], size)
Step 1: Start
Step 2: Set i = 0
Step 3: Repeat Steps 4-5 while i < size
Step 4: Access arr[i]
Step 5: Set i = i + 1
Step 6: Stop
```

Program 11.1 performs array traversal and prints the array elements as output. It uses the algorithm depicted in Example 11.1.

Program 11.1 Traverse Array

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

void transverse (int*, int); //Function prototype for array transversal
int main()
{
    intarr[5] = {2, 6, 7, 3, 8};
    int N = 5;
    system("cls");
    cout<< "Press any key to perform array transversal and display its
    elements : \n\n";
```

(Contd.)

```

        getch();
        transverse (arr,N); //Calling transverse fuction
        getch();
    }

void transverse(int *array, int size)
{
    inti = 0;
    for(i = 0; i< size; i++)
    {
        cout<< "arr[" <<i<< "] = " << array[i] <<endl; //Accessing Array
        elements and printing it
    }
}

```

Output

Press any key to perform array transversal and display its elements :

```

arr[0] = 2
arr[1] = 6
arr[2] = 7
arr[3] = 3
arr[4] = 8

```

**Tip**

While traversing an array, the index identifier should be updated carefully so that array out of bound situation does not arise. In this situation, the program tries to access a location outside of the reserved memory block, which is an illegal operation.



11.5 INSERTION AND DELETION

Insertion is the task of adding an element into an existing array while deletion is the task of removing an element from the array. The point of insertion or deletion that is the position where an element is to be inserted or deleted holds vital importance, as we shall see in the subsequent sections.

11.5.1 Insertion

If an element is to be inserted at the end of the array, then it can be simply achieved by storing the new element one position to the right of the last element. However, the array must have vacant positions at the end for this to be feasible. Alternatively, if an element is required to be inserted at the middle, then this will require all the subsequent elements to be moved one place to the right. Figure 11.2 depicts the insertion of an element into an array.

**Check Point****1. What is array traversal?**

Ans. It is the task of visiting the array elements and storing or retrieving values from it.

2. What is the need for array traversal?

Ans. It is required in almost all array related operations, such as sorting, searching, printing, etc.

	-1	3	5	22	77			
	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	
Initial Array A								
	-1	3	5	22	77	4		
	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	
Array contents after element 4 is inserted at the end								
	-1	3	4	5	22	77		
	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	
Array contents after element 4 is inserted at index location 2								

Fig. 11.2 Array insertion

Example 11.2 Write an Algorithm to Perform Array Insertion.

The following algorithm inserts an element P at index location k in the array $A[N]$, where $k \leq N$.

```

insert (A[N], k, P)
Step 1: Start
Step 2: Set  $i = N$ 
Step 3: Repeat Steps 4-5 while  $i \geq k$ 
Step 4: Set  $A[i+1] = A[i]$ 
Step 5: Set  $i = i - 1$ 
Step 6: Set  $A[k] = P$ 
Step 7: Set  $N = N + 1$ 
Step 8: Stop

```

Program 11.2 performs array insertion and prints the updated array elements as output. It uses the algorithm depicted in Example 11.2.

Program 11.2 Array Insertion

```

#include<iostream>
#include<conio.h>
using namespace std;

int main()
{
    int array[10] = {-1, 3, 5, 22, 77};
    int i, k, N, P;
    system("cls");
    N = 5;

```

(Contd.)

```
cout<< "The contents of the array are:\n";
for(i = 0; i< N; i++)
    cout<< "array[" <<i<< "] = " << array[i] <<endl; //Printing
    Array elements

cout<<endl;

cout<< "Enter the element to be inserted:\n";
cin>> P;

cout<< "Enter the location where "<< P <<" is to be inserted:\n";
cin>> k;

cout<<endl;

for(i = N; i>= k; i--)
    array[i+1] = array [i];

array[k] = P;
N = N+1;

cout<< "The contents of the array after insertion are:\n";

for(i = 0; i< N; i++)
    cout<< "array[" <<i<< "] = " << array[i] <<endl; //Printing
    array elements after insertion

getch();
}
```

Output

The contents of the array are:

```
array[0] = -1
array[1] = 3
array[2] = 5
array[3] = 22
array[4] = 77
```

Enter the element to be inserted : 19

Enter the location where 19 is to be inserted : 1

The contents of the array after insertion are:

```
array[0] = -1
array[1] = 19
array[2] = 3
array[3] = 5
array[4] = 22
array[5] = 77
```

11.5.2 Deletion

The deletion of elements follows a similar procedure as insertion. The deletion of element from the end is quite simple and can be achieved by mere updation of index identifier. However, to remove an element from the middle, one must move all the elements present to the right of the point of deletion, one position to the left. Figure 11.3 depicts the deletion of an element from an array.

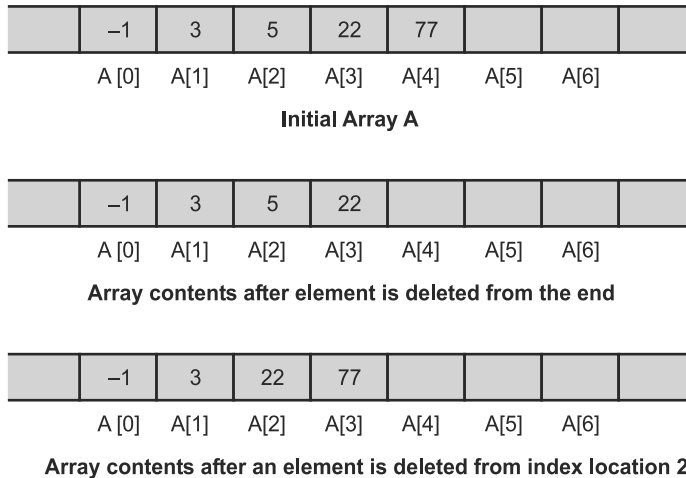


Fig. 11.3 Array deletion

Example 11.3 Write an Algorithm to Perform Array Deletion.

The following algorithm deletes the element at index location k in the array $A[N]$, where $k \leq N$.

```
delete (A[N], k)
Step 1: Start
Step 2: Set D = A[k]
Step 3: Set i = k
Step 4: Repeat Steps 5-6 while i <= N-1
Step 5: Set A[i] = A[i+1]
Step 6: Set i = i + 1
Step 7: Set N = N - 1
Step 8: Stop
```

Program 11.3 performs array deletion and prints the updated array elements as output. It uses the algorithm depicted in Example 11.3.

Program 11.3 Array Deletion

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

(Contd.)


```
int main()
{
    int array[10] = {-1, 3, 5, 22, 77};
    int i, k, N, D;

    system("cls");

    N = 5;

    cout<< "The contents of the array are:\n";
    for(i = 0; i< N; i++)
        cout<< "array[" <<i<< "] = " << array[i] <<endl; //Printing array
        elements

    cout<<endl;

    cout<< "Enter the location from where element is to be deleted:\n";
    cin>> k;

    cout<<endl;

    D = array[k];

    for(i = k; i<= N-2; i++)
        array[i] = array [i+1];

    N = N - 1;

    cout<< D << " is the array element that was deleted from index location
    " << k <<endl;

    cout<< "\nThe contents of the array after deletion are:\n";

    for(i = 0; i< N; i++)
        cout<< "array[" <<i<< "] = " << array[i] <<endl; //Printing array
        elements after deletion

    getch();
}
```

Output

The contents of the array are:

```
array[0] = -1
array[1] = 3
array[2] = 5
array[3] = 22
array[4] = 77
```

Enter the location from where element is to be deleted : 3

22 is the array element that was deleted from index location 3

The contents of the array after deletion are:

```
array[0] = -1
array[1] = 3
array[2] = 5
array[3] = 77
```

Note

For large-sized arrays, inserting an element at the middle could be a considerable programming overhead as it would require the other elements to be moved from their current positions.



11.6 SORTING AND SEARCHING

Sorting and searching are two of the most common operations performed on arrays. The sorting operation arranges the elements of an array in a specific order or sequence. Searching, on the other hand, locates a specific element in the array.

11.6.1 Sorting

Sorting involves comparing the array elements with each other and shuffling them until all the elements are sorted. There are a number of sorting techniques that are applied to sort an array in an efficient manner. We shall explore these sorting techniques in Chapter 17. Here, let us focus on one of the most basic sorting techniques called *bubble sort*.

Bubble sort operates on an array in such a manner that the largest element is brought to the end of the array with each successive iteration. It repeatedly compares two consecutive elements and moves the largest amongst them to the right. This process is repeated for each pair of elements until the current iteration moves the largest element to the end.

Consider the following integer array:

```
arr[5] = {5, 4, 3, 2, 1}
```

Here, **arr** is a five-element integer array. It will take four iterations or passes to sort this five-element array. Each pass will bring the largest element to the end of the array. Here's a snapshot of the array contents after each of the four passes:

```
Initial Array - arr[5] = {5, 4, 3, 2, 1}
Pass 1         - arr[5] = {4, 3, 2, 1, 5}
Pass 2         - arr[5] = {3, 2, 1, 4, 5}
Pass 3         - arr[5] = {2, 1, 3, 4, 5}
Pass 4         - arr[5] = {1, 2, 3, 4, 5}
```

As shown above, the fourth pass produces the sorted array.

Algorithm Refer Section 17.2.3 for the algorithm on applying bubble sorting technique to sort an array.

Program 11.4 implements bubble sorting technique to sort an array of five elements.



Check Point

1. What is array insertion and deletion?

Ans. The task of adding an element into an existing array is called array insertion while the task of deleting an existing element from the array is called array deletion.

2. What is a point of insertion?

Ans. It is the location in the array where a new element is to be inserted.

Program 11.4 Array Sorting

```

#include<iostream>
#include<conio.h>
using namespace std;
// Sorting array using Bubble Sort
int main()
{
    int array[5] = {5, 4, 3, 2, 1};
    int i, k, j, temp;
    system("cls");
    cout<< "\n\nThe initial Array Elements are :\n";
    for(i=0; i<5; i++)
        cout<< "array[" <<i<< "] = " << array[i] <<endl; //Printing
        initial array elements
    for(i=5; i>1; i--) //Outer loop for controlling the number of
    passes
    {
        for(j=0; j<i-1; j++) // Inner loop for controlling the number
        of comparisons made in a pass
        {
            if(array[j] > array[j+1])
            {
                //Swapping Adjacent Elements
                temp = array[j+1];
                array[j+1] = array[j];
                array[j] = temp;
            }
        }
    }
    cout<< "\n\nThe contents of the Sorted array are:\n";
    for(i = 0; i < 5; i++)
        cout<< "array[" <<i<< "] = " << array[i] <<endl; //Printing
        array elements after deletion
    getch();
}

```

Output

```

The initial Array Elements are :
array[0] = 5
array[1] = 4
array[2] = 3
array[3] = 2
array[4] = 1
The contents of the Sorted array are:
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5

```



Note Just like an integer array, sorting can also be applied to an array of floats, characters, structures, and so on.

11.6.2 Searching

Searching is the process of traversing an array to find out if a specific element is present in the array or not. If the search is successful, the index location of the element is returned. There are various searching mechanisms that can be employed to search an element in an array. We shall explore these searching techniques in Chapter 8. Here, let us focus on one of the most basic searching techniques called *linear search*.

The linear search technique traverses an array sequentially to search the desired element. It starts the search with the first element and moves towards the end in a step-by-step fashion. At each step, it matches the element to be searched with the array element, and if there is a match, the location of the array element is returned.

Consider the following integer array:

```
arr[5] = {22, 19, 4, 8, 10}
```

Here, **arr** is a five element integer array. If we apply linear search to the array **arr** to search element 4, then the search will be successful as element 4 is present in the array. The search module will return index location 2 as the search result because element 4 is the third element in the array.

Algorithm Refer to Section 18.3.1 for the algorithm on applying linear search on an array.

Program 11.5 applies linear searching technique on an array of five elements.



Check Point

1. What is sorting?

Ans. It is the task of arranging the elements of an array in a sequence.

2. How many passes does bubble sorting technique require to sort an array of n elements?

Ans. $n-1$.

Program 11.5 Array Linear Search

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
    int array[5] = {22, 19, 4, 8, 10};
    int i, k, flag, index;
    system("cls");
    flag = 0;
    cout<< "The contents of the array are:\n";
    for(i = 0; i < 5; i++)
        cout<< "array[" <<i<< "] = " << array[i] <<endl; //Printing
        array elements
    cout<<endl;
```

(Contd.)

```
cout<< "Enter the element to be searched:\n";
cin>> k;

for(i = 0; i< 5; i++)//Scanning array elements one by one
{
    if(k == array[i])
    {
        flag = 1; //Successful search
        index = i;
        break;
    }
}

if (flag == 1)
    cout<< "Search is Successful! Array Element " << k << " is
    present at the index location " << index << " in the array"
    <<endl;
else //Unsuccessful search
    cout<< "Search is Unsuccessful!" <<endl;

getch();
}
```

Output

The contents of the array are:

array[0] = 22

array[1] = 19

array[2] = 4

array[3] = 8

array[4] = 10

Enter the element to be searched : 4

Search is Successful!!! Array Element 4 is present at the index location 2 in the array

**Note**

Just like an integer array, searching can also be performed on an array of floats, characters, structures, and so on.



11.7 REPRESENTATION OF MULTI-DIMENSIONAL ARRAY IN MEMORY

Let us recall that a multi-dimensional array is an array of arrays. Unlike one-dimensional arrays which have only one subscript, a multidimensional array has multiple subscripts. For example, a two-dimensional array, one of the most widely used instances of multi-dimensional arrays, has two subscripts. It is

used to programmatically realize a matrix with its first subscript representing the row and the second subscript representing the column of a matrix.

The representation of a two-dimensional array in memory is not like the grid-like structure of a matrix. Instead, it is same as the one-dimensional array representation in memory. It either stores the array elements row by row (*row major order*) or column by column (*column major order*). Figure 11.4 illustrates these representations:

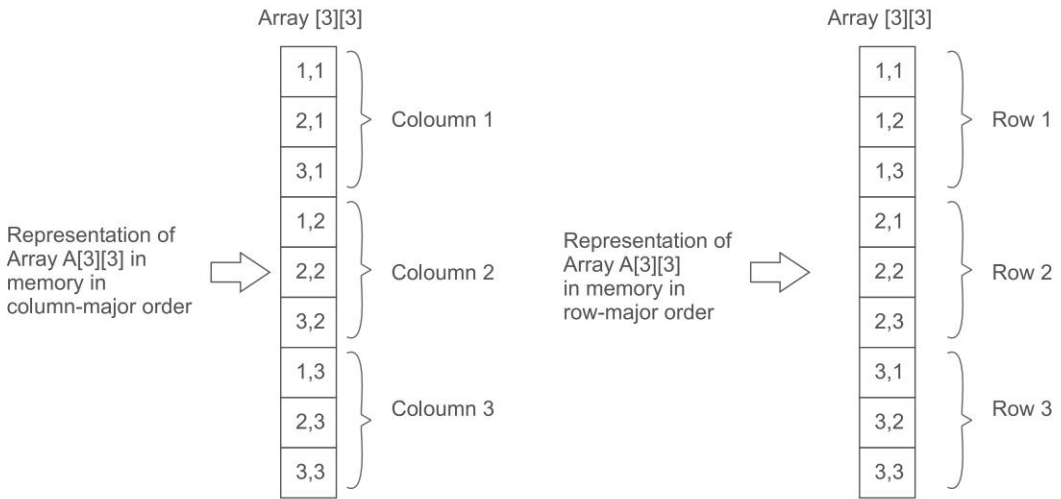


Fig. 11.4 Representation of two-dimensional array in memory

As shown in Fig. 11.4, the elements of a two-dimensional array are stored at consecutive memory locations. The only difference is in the order in which these elements are stored in memory. In column-major order, the elements are stored column-by-column while in row-major order the elements are stored row-by-row. Both these memory representations are intrinsic to a programming language and the programmer does not have a choice of selecting a particular representation format for storing array elements.

The formula for computing the address location of a multi-dimensional array element in row major implementation is given below:

$$\text{Address of } A[i,j] = B + W (n (i - \text{LBR}) + (j - \text{LBC}))$$

Here,

1. **A[][]** is the multidimensional array.
2. **B** is the base address.
3. **W** is the word size or the size of an array element.
4. **n** is the number of columns.
5. **i, j** are the index identifiers.
6. **LBR** is the lower bound of row index.
7. **LBC** is the lower bound of column index.



Check Point

1. What is searching?

Ans. Searching is the process of determining if a specific element is present in the array or not.

2. At a maximum, how many elements would the searching technique require to traverse in an n -element array?

Ans. n

Similarly, the formula for computing the address location of a multi-dimensional array element in column major implementation is given below:

$$\text{Address of } A[i, j] = B + W (m (j - LBC) + (i - LBR))$$

Here, m represents the number of rows.

Example 11.4 A 10×12 matrix is implemented using array $A[10][12]$. If the base address of the array is 200 and the word size is 2 then compute the address of the element $A[4, 7]$ in

(a) Row major order

(b) Column major order

Assume that the lower bound of both row and column indices is 1.

(a) Row major order

$$\text{Address of } A[i, j] = B + W (n (i - LBR) + (j - LBC))$$

$$\begin{aligned} \text{Address of } A[4, 7] &= 200 + 2 (12 (4 - 1) + (7 - 1)) \\ &= 200 + 2 (42) \\ &= 284 \end{aligned}$$

(b) Column major order

$$\text{Address of } A[i, j] = B + W (m (j - LBC) + (i - LBR))$$

$$\begin{aligned} \text{Address of } A[4, 7] &= 200 + 2 (10 (7 - 1) + (4 - 1)) \\ &= 200 + 2 (63) \\ &= 326 \end{aligned}$$



Check Point

1. What is row-major order?

Ans. It is the memory representation of a two-dimensional array in row-by-row fashion.

2. What is column-major order?

Ans. It is the memory representation of a two-dimensional array in column-by-column fashion.



11.8 REALIZING MATRICES USING TWO-DIMENSIONAL ARRAYS

Two-dimensional arrays are most commonly used for realizing matrices. The first subscript signifies the rows of a matrix while the second subscript signifies the columns. Operation on these array-represented matrices can be performed through simple programming.

Figure 11.5 depicts the realization of a matrix through a two-dimensional array:



Mind Jog

What is a square matrix?

It is the matrix that has equal number of rows and columns.

$$M[3][4] \rightarrow \left\{ \begin{array}{cccc} 0,0 & 0,1 & 0,2 & 0,3 \\ 1,0 & 1,1 & 1,2 & 1,3 \\ 2,0 & 2,1 & 2,2 & 2,3 \end{array} \right\}$$

Fig. 11.5 Matrix represented by two-dimensional array

Figure 11.5 shows the subscript values for each of the elements of the matrix $M[3][4]$.

Program 11.6 realizes a 3×3 matrix using a two-dimensional array.

Program 11.6 3x3 Matrix using Two Dimensional Array

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
    inti, j, M[3][3];

    system("cls");

    //Reading Matrix Elements

    cout<< "Enter the Elements of the 3 X 3 matrix:\n\n";
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            cout<< "M[" <<i<< j << "] = ";
            cin>> M[i][j];
        }
    }

    //Printing Matrix Elements
    cout<< "\nThe matrix represented by the 3 X 3 2D array is: \n";

    for(i = 0; i < 3; i++)
    {
        cout<<"\n\t\t";
        for(j = 0; j < 3; j++)
            cout<< M[i][j] << "\t";
    }

    cout<<endl;

    getch();
}
```

(Contd.)

Output

Enter the Elements of the 3 X 3 matrix:

```

M[00] = 1
M[01] = 2
M[02] = 3
M[10] = 4
M[11] = 5
M[12] = 6
M[20] = 7
M[21] = 8
M[22] = 9

```

The matrix represented by the 3 X 3 2D array is:

1	2	3
4	5	6
7	8	9



MATRIX OPERATIONS

The various operations associated with matrices are

1. Addition
2. Subtraction
3. Multiplication
4. Transpose

11.9.1 Addition

Addition is the task of adding individual elements of two matrices. For instance,

	a1	a2	a3
If matrix A =	a4	a5	a6
	a7	a8	a9

	b1	b2	b3
And, matrix B =	b4	b5	b6
	b7	b8	b9

	a1+b1	a2+b2	a3+b3
Then, A + B =	a4+b4	a5+b5	a6+b6
	a7+b7	a8+b8	a9+b9

Program 11.7 Adding on two 3×3 matrices

Program 11.7 Adding on two 3 × 3 Matrices

```

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

using namespace std;

int main()
{
    inti, j, A[3][3], B[3][3], C[3][3];
    system("cls");

    //Reading Matrix Elements

    cout<< "\nEnter the elements of the 3 X 3 matrix A:\n\n";
    for(i = 0; i< 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            cout<< "A[" <<i<< j << "] = ";
            cin>> A[i][j]; //Reading Elements of the 1st Matrix.
        }
    }

    cout<< "\nEnter the elements of the 3 X 3 matrix B:\n\n";
    for(i = 0; i< 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            cout<< "B[" <<i<< j << "] = ";
            cin>> B[i][j]; //Reading Elements of the 2nd Matrix.
        }
    }

    //Printing Matrix Elements
    cout<< "\nThe entered matrices are: \n";
    for(i = 0; i< 3; i++)
    {
        cout<< "\n\t\t";
        for(j = 0; j < 3; j++)
            cout<< A[i][j] << "\t"; // Displaying elements of Matrix A
        cout<< "\n\t\t";
        for(j = 0; j < 3; j++)
            cout<< B[i][j] << "\t"; // Displaying elements of Matrix B
    }

    for(i = 0; i< 3; i++)
        for(j = 0; j < 3; j++)
            C[i][j] = A[i][j] + B[i][j]; //Computing the sum of two matrices
    cout<< "\n\nSum of A and B is shown below: \n";

```

(Contd.)

```
for(i = 0; i < 3; i++)
{
    cout<<"\n\t\t";
    for(j = 0; j < 3; j++)
        cout<< C[i][j] << "\t";//Displaying the result
    }

    cout<<endl;

    getch();
}
```

Output

Enter the elements of the 3 X 3 matrix A:

```
A[00] = 1
A[01] = 1
A[02] = 1
A[10] = 1
A[11] = 1
A[12] = 1
A[20] = 1
A[21] = 1
A[22] = 1
```

Enter the elements of the 3 X 3 matrix B:

```
B[00] = 2
B[01] = 2
B[02] = 2
B[10] = 2
B[11] = 2
B[12] = 2
B[20] = 2
B[21] = 2
B[22] = 2
```

The entered matrices are:

1	1	1	2	2	2
1	1	1	2	2	2
1	1	1	2	2	2

Sum of A and B is shown below:

3	3	3
3	3	3
3	3	3

11.9.2 Subtraction

Subtraction is the task of subtracting individual elements of two matrices. For instance,

		a1	a2	a3
If matrix A	=	a4	a5	a6
		a7	a8	a9
		b1	b2	b3
And, matrix B	=	b4	b5	b6
		b7	b8	b9
		a1-b1	a2-b2	a3-b3
Then, A - B	=	a4-b4	a5-b5	a6-b6
		a7-b7	a8-b8	a9-b9

A C program to perform matrix subtraction will be same as matrix addition (see Example 11.11). We just need to replace the +sign with a -sign.

11.9.3 Multiplication

Matrix multiplication is not as simple as matrix addition or subtraction. It uses a certain formula to generate multiplication result. For instance,

		a1	a2	a3
If matrix A	=	a4	a5	a6
		a7	a8	a9
		b1	b2	b3
And, matrix B	=	b4	b5	b6
		b7	b8	b9
	a1b1+a2b4+a3b7	a1b2+a2b5+a3b8	a1b3+a2b6+a3b9	
Then, A × B	= a4b1+a5b4+a6b7	a4b2+a5b5+a6b8	a4b3+a5b6+a6b9	
	a7b1+a8b4+a9b7	a7b2+a8b5+a9b8	a7b3+a8b6+a9b9	

For two non-square matrices, multiplication is feasible only if the number of columns in the left matrix is equal to the number of rows in the right matrix. Thus, if an $M \times N$ matrix is multiplied with a $N \times P$ matrix, then the resultant matrix would be a $M \times P$ matrix.

Program 11.8 Multiplying on Two 3×3 Matrices

Program 11.8 Adding on Two 3×3 Matrices

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

using namespace std;

int main()
{
    inti, j, k, A[3][3], B[3][3], C[3][3];
```

(Contd.)

```
system("cls");
//Reading Matrix Elements
cout<< "\nEnter the Elements of the 3 X 3 matrix A:\n\n";
for(i = 0; i< 3; i++)
{
    for(j = 0; j < 3; j++)
    {
        cout<< "A[" <<i<< j << "] = ";
        cin>> A[i][j]; //Reading Elements of the 1st Matrix.
    }
}

cout<< "\nEnter the Elements of the 3 X 3 matrix B:\n\n";
for(i = 0; i< 3; i++)
{
    for(j = 0; j < 3; j++)
    {
        cout<< "B[" <<i<< j << "] = ";
        cin>> B[i][j]; //Reading Elements of the 2nd Matrix.
    }
}

//Printing Matrix Elements
cout<< "\nThe entered Matrices are: \n";
for(i = 0; i< 3; i++)
{
    cout<< "\n\t\t";
    for(j = 0; j < 3; j++)
        cout<< A[i][j] << "\t"; // Displaying elements of Matrix A
    cout<< "\n\t\t";
    for(j = 0; j < 3; j++)
        cout<< B[i][j] << "\t"; // Displaying elements of Matrix B
}

// Multiplying the two Matrices
for(i = 0; i< 3; i++)
{
    for(j = 0; j < 3; j++)
    {
        C[i][j] = 0;
        for(k = 0; k < 3; k++)
        {
            C[i][j] = C[i][j] + (A[i][k] * B[k][j]);
        }
    }
}

cout<< "\n\nThe product of the two matrices A X B is shown below: \n";
for(i = 0; i< 3; i++)
{
```

(Contd.)

```

        cout<<"\n\t\t";
        for(j = 0; j < 3; j++)
            cout<< C[i][j] << "\t";//Displaying the Result
    }
    cout<<endl;
    getch();
}

```

Output

Enter the Elements of the 3 X 3 matrix A:

```

A[00] = 1
A[01] = 2
A[02] = 3
A[10] = 4
A[11] = 5
A[12] = 6
A[20] = 7
A[21] = 8
A[22] = 9

```

Enter the Elements of the 3 X 3 matrix B:

```

B[00] = 9
B[01] = 8
B[02] = 7
B[10] = 6
B[11] = 5
B[12] = 4
B[20] = 3
B[21] = 2
B[22] = 1

```

The entered Matrices are:

1	2	3	9	8	7
4	5	6	6	5	4
7	8	9	3	2	1

The product of the two Matrices A X B is shown below:

30	24	18
84	69	54
138	114	90

11.9.4 Transpose

In simple words, transposing a matrix refers to the task of changing the rows into columns and columns into rows. For instance,

		a1	a2	a3
If matrix A	=	a4	a5	a6
		a7	a8	a9
		a1	a4	a7
Then, transpose(A)	=	a2	a5	a8
		a3	a6	a9

Program 11.9 C Program to Transpose a 3×3 Matrix**Program 11.9** Transpose 3×3 Matrix

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

using namespace std;

int main()
{
    inti, j, A[3][3], T[3][3];
    system("cls");

    //Reading Matrix Elements

    cout<< "\nEnter the Elements of the 3 X 3 matrix:\n\n";
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            cout<< "A[" <<i<< j << "] = ";
            cin>> A[i][j]; //Reading Elements of the Matrix.
        }
    }

    //Printing Matrix Elements

    cout<< "\nThe entered Matrix is: \n";
    for(i = 0; i < 3; i++)
    {
        cout<< "\n\t\t";
        for(j = 0; j < 3; j++)
            cout<< A[i][j] << "\t"; // Displaying Elements of the Matrix
    }

    for(i = 0; i < 3; i++)
        for(j = 0; j < 3; j++)
            T[i][j] = A[j][i]; //Computing the Matrix Transpose

    cout<< "\n\nThe Transpose of the Matrix is shown below: \n";
    for(i = 0; i < 3; i++)
    {
        cout<< "\n\t\t";
```

(Contd.)

```

        for(j = 0; j < 3; j++)
            cout<< T[i][j] << "\\t";//Displaying the Result
    }
    cout<<endl;
    getch();
}

```

Output

Enter the Elements of the 3 X 3 matrix:

```

A[00] = 1
A[01] = 2
A[02] = 3
A[10] = 4
A[11] = 5
A[12] = 6
A[20] = 7
A[21] = 8
A[22] = 9

```

The entered Matrix is:

1	2	3
4	5	6
7	8	9

The Transpose of the Matrix is shown below:

1	4	7
2	5	8
3	6	9



Note

Adequate checks must be included in a program to ensure that non-compatible matrices are not operated.



Check Point

1. What is matrix addition?

Ans. It is the task of adding the relative elements of two matrices.

2. What is matrix transpose?

Ans. It is the task of changing the rows into columns and columns into rows.

SOLVED PROBLEMS

Problem 11.1 Consider the following array of integers:

35 18 7 12 5 23 16 3 1

Create a snapshot of the above array for the following operations:

Inserting element 99 at index location 2.

Deleting the first element of the array.

Solution

Array contents after insertion: 35 18 99 7 12 5 23 16 3 1

Array contents after deletion: 18 7 12 5 23 16 3 1

Problem 11.2 Consider the following array of integers:

74 39 35 32 97 84

Create a snapshot of the above array after the sorting operation is performed on it.

Solution

Initial array 74 39 35 32 97 84

Sorted array 32 35 39 74 84 97

Problem 11.3 Consider the following array of integers:

74 39 35 32 97 84

How many elements would need to be traversed before search operation is completed on the following items:

32

83

Solution

4

6

Problem 11.4 Consider the following array of integers::

35 54 12 18 23 15 45 38

Deduce the address of the 4th element (index location 3), if the base address is 3000. Assume that the word size is 2.

Solution

$$\begin{aligned}\text{Address of arr}[3] &= 3000 + 2 * 3 \\ &= 3000 + 6 \\ &= 3006\end{aligned}$$

Problem 11.5 A two-dimensional array A[5][10] is implemented in row order manner in the memory. Deduce the address of the A[3][5] element, if the base address of the array is 3000 and the word size is 2. Assume the lower bound of row and column indices to be 1.

Solution

$$\begin{aligned}\text{Address of } A[i, j] &= B + W (n(i - \text{LBR}) + (j - \text{LBC})) \\ \text{Address of } A[3, 5] &= 3000 + 2 (10 (3 - 1) + (5 - 1)) \\ &= 3000 + 2 (24) \\ &= 3048\end{aligned}$$

Problem 11.6 Solve Problem 5 in case of column order implementation.

Solution

$$\begin{aligned}\text{Address of } A[i, j] &= B + W (m(j - \text{LBC}) + (i - \text{LBR})) \\ \text{Address of } A[3, 5] &= 3000 + 2 (5 (3 - 1) + (5 - 1)) \\ &= 3000 + 2 (14) \\ &= 3028\end{aligned}$$

SUMMARY

- ☐ Arrays are characterized as one-dimensional and multi-dimensional arrays.
- ☐ One-dimensional arrays are stored at consecutive locations in memory.
- ☐ Array traversal involves visiting the array elements and storing or retrieving values from it.
- ☐ Insertion is the task of adding an element into an existing array while deletion is the task of removing an element from the array.
- ☐ Sorting involves arranging the elements of an array in a specific order or sequence.
- ☐ Searching involves locating a specific element in an array.
- ☐ Multi-dimensional array either stores the array elements in row major order or column major order.
- ☐ Two-dimensional arrays are most commonly used for realizing matrices.
- ☐ Common operations performed on matrices are: addition, subtraction, multiplication, transpose.

Key Terms

Array | One-dimensional array | Multi-dimensional array | Array subscript | Base address | Sorting | Searching | Row major order | Column major order

Review Questions

- 11.1 What is an array? What are its various types?
- 11.2 Explain the representation of a one-dimensional array in memory with the help of an illustration.
- 11.3 What is array traversal? Why is it used?
- 11.4 What are the typical operations associated with arrays? Explain.
- 11.5 Write an algorithm for deleting an element at index location k in the array $A[N]$.
- 11.6 What is the difference between sorting and searching?
- 11.7 Explain the representation of a two-dimensional array in memory with the help of an illustration.
- 11.8 Explain with the help of an illustration how a 2×2 matrix is stored in memory using column major order representation.
- 11.9 What is matrix multiplication? Explain with the help of an example.
- 11.10 What is the significance of transposing a matrix?

Programming Exercises

- 11.1 Write a C program to find the smallest element in an integer array.
- 11.2 Write a C program to read a value and insert it at the middle of an integer array.
- 11.3 Write a C program to sort an array of 10 integers.
- 11.4 Write a C program to demonstrate searching on an array of ten integers.
- 11.5 Write a C program to show how matrices are realized using two-dimensional arrays.
- 11.6 Write a C program to perform matrix subtraction.
- 11.7 Write a C program to perform transpose of a matrix.

12

LINKED LISTS

CHAPTER OUTLINE

- 12.1 Introduction
- 12.2 Linked Lists—Basic Concept
 - 12.2.1 Representation of Linked Lists
 - 12.2.2 Advantages of Linked Lists
 - 12.2.3 Disadvantages of Linked Lists
- 12.3 Linked List Implementation
 - 12.3.1 Linked List Node Declaration
 - 12.3.2 Linked List Operations
 - 12.3.3 Linked List Implementation
- 12.4 Types of Linked Lists
- 12.5 Circular Linked List
 - 12.5.1 Circular Linked List Operations
 - 12.5.2 Circular Linked List Implementation
- 12.6 Doubly Linked List
 - 12.6.1 Doubly Linked List Node Declaration
 - 12.6.2 Doubly Linked List Operations
 - 12.6.3 Doubly Linked List Implementation
- 12.7 Some More Programs
 - Summary*
 - Key Terms*





12.1 INTRODUCTION

In the previous chapter, we learnt about arrays and how they are used for storing same type data elements in memory. While arrays are a good way of grouping same data together, they also have a key limitation associated with them. An array is allocated fixed amount of memory space before a program is executed. Thus, if there is a need at run time to store more data in the array than its actual capacity, then there is no way of doing this. This is where a linked list becomes more useful. It allows for dynamic allocation of memory space at run time. Thus, there is no need to block memory space at compile time.

Linked list is a collection of nodes or data elements logically connected to each other. Whenever there is a need to add a new element to the list, a new node is created and appended at the end of the list. In this chapter, we will learn how a linked list is implemented and how common operations like insertion and deletion are performed on it. We will also learn about linked list variants, that is circular linked list and doubly linked list.



12.2 LINKED LISTS—BASIC CONCEPT

Linked list is a collection of data elements stored in such a manner that each element points at the next element in the list. The elements of a linked list are also referred as *nodes*. Each node has two parts: INFO and NEXT. The INFO part contains the data element while the NEXT part contains the address of the next node. The NEXT part of the last node of the list contains a NULL value indicating the end of the list. The beginning of the list is indicated with the help of a special pointer called FIRST. Similarly, the end of the list is indicated by a pointer called LAST.

12.2.1 Representation of Linked Lists

Unlike arrays, the nodes of a linked list need not occupy contiguous locations in memory. Instead, they can be stored at discrete memory locations, logically connected with each other through node NEXT.

Figure 12.1 depicts the logical representation of a linked list.

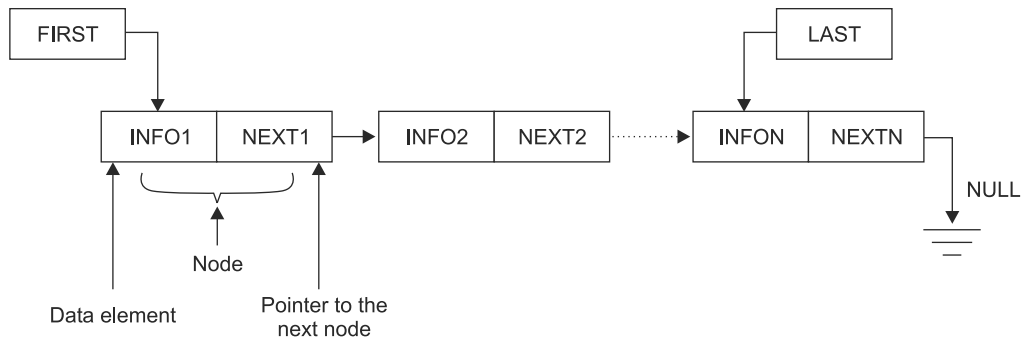


Fig. 12.1 Logical representation of a linked list

As shown in the above representation, the first and last nodes of the list are indicated by two distinct pointers, FIRST and LAST.

12.2.2 Advantages of Linked Lists

Some of the key advantages of linked lists are the following:

1. Linked lists facilitate dynamic memory management by allowing elements to be added or deleted at any time during program execution.
2. The use of linked lists ensures efficient utilization of memory space as only that much amount of memory space is reserved as is required for storing the list elements.
3. It is easy to insert or delete elements in a linked list, unlike arrays, which require shuffling of other elements with each insert and delete operation.

12.2.3 Disadvantages of Linked Lists

Apart from the advantages, linked lists also possess certain limitations, which are the following:

1. A linked list element requires more memory space in comparison to an array element because it has to also store the address of the next element in the list.
2. Accessing an element is a little more difficult in linked lists than arrays because unlike arrays, there is no index identifier associated with each list element. Thus, to access a linked list element, it is mandatory to traverse all the preceding elements.



12.3 LINKED LIST IMPLEMENTATION

The implementation of a linked list involves two tasks:

1. Declaring the list node
2. Defining the linked list operations

12.3.1 Linked List Node Declaration

Since a linked list node contains two parts, INFO and NEXT, a structure construct is best suited for its realization. The following structure declaration defines a linked list node:

```
struct node
{
    int INFO;
    struct node *NEXT;
};
typedef struct node NODE;
```

The above structure declaration defines a new data type called NODE that represents a linked list node. The node structure contains two members, INFO for storing integer data values and NEXT for storing address of the next node.

The statement, *struct node *NEXT*, indicates that the pointer NEXT points at same structure type i.e. node. Such structures that contain pointer references to their own types are called as self-referential structures.

12.3.2 Linked List Operations

The typical operations performed on a linked list are:

1. Insert
2. Delete
3. Search
4. Print

1. Insert The insert operation adds a new element to the linked list. The following tasks are performed while adding the new element:

- (a) Memory space is reserved for the new node.
- (b) The element is stored in the INFO part of the new node.
- (c) The new node is connected to the existing nodes in the list.

Depending on the location where the new node is to be added, there are three scenarios possible, which are

- (a) Inserting the new element at the beginning of the list
- (b) Inserting the new element at the end of the list
- (c) Inserting the new element somewhere at the middle of the list

Inserting a new element at the beginning or end of the list is easy as it only requires resetting the respective NEXT fields. However, if the new element is to be added somewhere at the middle of the list then a search operation is required to be performed to identify the point of insertion.

Figures 12.2 (a) and (b) show the insertion of a new element between two existing elements of a linked list.

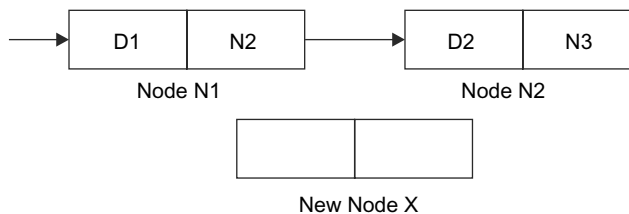


Fig 12.2(a) *Creating a new element*

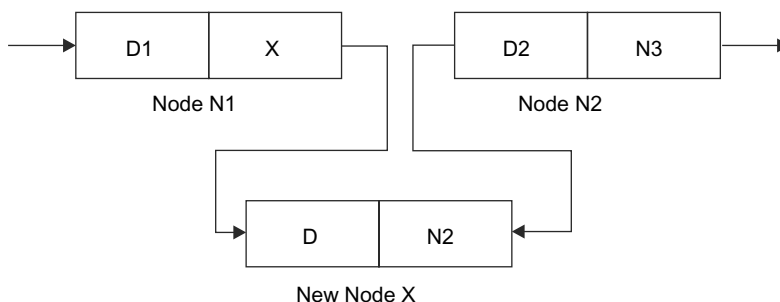


Fig 12.2(b) *Inserting the newly created element*

Example 12.1 Algorithm to Insert an Element at the End of a Linked List

```

insert (value)
Step 1: Start
Step 2: Set PTR = addressof (New Node)
        //Allocate a new node and assign its address to the pointer PTR
Step 3: Set PTR->INFO = value;
        //Store the element value to be inserted in the INFO part of the new node
Step 4: If FIRST = NULL, then goto Step 5 else goto Step 7
        //Check whether the existing list is empty
Step 5: Set FIRST=PTR and LAST=PTR
        //Update the FIRST and LAST pointers
Step 6: Set PTR->NEXT = NULL and goto Step 8
Step 7: Set LAST->NEXT=PTR, PTR->NEXT=NULL and LAST=PTR
        //Link the newly created node at the end of the list
Step 8: Stop

```

2. Delete The delete operation removes an existing element from the linked list. The following tasks are performed while deleting an existing element:

- The location of the element is identified.
- The element value is retrieved. In some cases, the element value is simply ignored.
- The link pointer of the preceding node is reset.

Depending on the location from where the element is to be deleted, there are three scenarios possible, which are

- Deleting an element from the beginning of the list.
- Deleting an element from the end of the list.
- Deleting an element somewhere from the middle of the list.

Deleting an element from the beginning or end of the list is easy as it only requires resetting the first and last pointers. However, if an element is to be deleted from within the list then a search operation is required to be performed for locating that element. Figures 12.3 (a) and (b) show the deletion of an element that is present between two existing elements of a linked list.

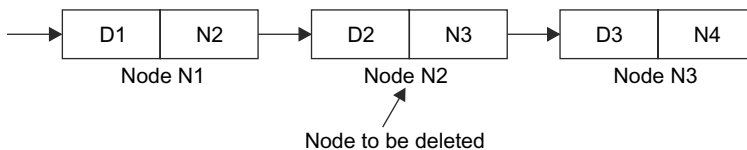


Fig 12.3(a) Identifying the node to be deleted

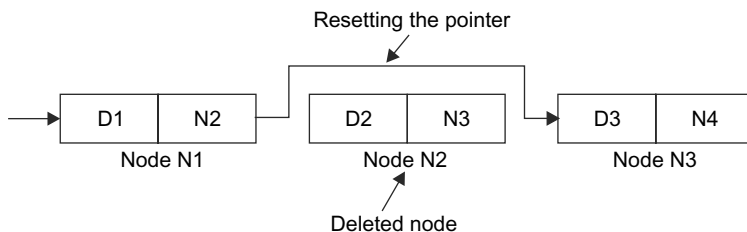


Fig 12.3(b) Deleting the node

Example 12.2 Algorithm to Delete a Specific Element from a Linked List

```

delete (value)
Step 1: Start
Step 2: Set LOC = search (value)
        //Call the search module to search the location of the node to be
deleted and assign it to LOC pointer
Step 3: If LOC=NULL goto Step 4 else goto Step 5
Step 4: Return ("Delete operation unsuccessful: Element not present") and
Stop
Step 5: If LOC=FIRST goto Step 6 else goto Step 10
        //Check if the element to be deleted is the first element in the list
Step 6: If FIRST=LAST goto Step 7 else goto Step 8
        //Check if there is only one element in the list
Step 7: Set FIRST=LAST=NULL and goto Step 9
Step 8: Set FIRST=FIRST->NEXT
Step 9: Return ("Delete operation successful") and Stop
Step 10: Set TEMP=LOC-1
        //Assign the location of the node present before LOC to temporary
pointer TEMP
Step 11: Set TEMP->NEXT=LOC->NEXT
        //Link the TEMP node with the node being currently pointed by LOC
Step 12: If LOC=LAST goto Step 13 else goto Step 14
        //Check if the element to be deleted is currently the last element in
the list
Step 13: Set LAST=TEMP
Step 14: Return ("Delete operation successful")
Step 15: Stop

```

3. Search The search operation helps to find an element in the linked list. The following tasks are performed while searching an element:

- (a) Traverse the list sequentially starting from the first node.
- (b) Return the location of the searched node as soon as a match is found.
- (c) Return a search failure notification if the entire list is traversed without any match.

The NEXT pointers help in traversing the linked list from start till end.

Example 12.3 Algorithm to Search a Specific Element in the Linked List

```

search (value)
Step 1: Start
Step 2: If FIRST=NULL goto Step 3 else goto Step 4
        //Check if the linked list is empty
Step 3: Return ("Search unsuccessful: Element not present") and Stop
Step 4: Set PTR=FIRST
Step 5: Repeat Steps 6-8 until PTR!=LAST
        //Repeat Steps 6-8 until PTR is not equal to LAST
Step 6: If PTR->INFO=value goto Step 7 else goto Step 8
Step 7: Return ("Search successful", PTR) and Stop

```

(Contd.)


```

Step 8: Set PTR=PTR->NEXT
Step 9: If LAST->INFO=value goto Step 10 else goto Step 11
      //Check if the element to be searched is the last element in the list
Step 10: Return ("Search successful", LAST) and Stop
Step 11: Return ("Search unsuccessful: Element not present")
Step 12: Stop

```

4. Print The print operation prints or displays the linked list elements on the screen. To print the elements, the linked list is traversed from start till end using NEXT pointers.

Example 12.4 Algorithm to Print all the Linked List Elements

```

print ()
Step 1: Start
Step 2: If FIRST=NULL goto Step 3 else goto Step 4
      //Check if the linked list is empty
Step 3: Display ("Empty List") and Stop
Step 4: If FIRST=LAST goto Step 5 else goto Step 6
      //Check if the list has only one element
Step 5: Display (FIRST->INFO) and Stop
Step 6: Set PTR=FIRST
Step 7: Repeat Steps 8-9 until PTR!=LAST
      //Repeat Steps 8-9 until PTR is not equal to LAST
Step 8: Display (PTR->INFO)
      //Displaying list elements
Step 9: Set PTR=PTR->NEXT
Step 10: Display (LAST->INFO)
      //Displaying last element
Step 11: Stop

```

12.3.3 Linked List Implementation

Linked list implementation involves declaring its structure and defining its operations. The following example shows how a linked list is implemented using C language.

Program 12.1 implements a linked list in C. It uses the insert (Example 12.1), delete (Example 12.2), search (Example 12.3) and print (Example 12.4) algorithms for realizing the common linked list operations.

Program 12.1 Linked List Implementation

```

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

using namespace std;

// Linked List Declaration
struct node
{

```

(Contd.)

```

    int INFO;
    struct node *NEXT;
};

// Declaring pointers to first and last node of the linked list
struct node *FIRST = NULL;
struct node *LAST = NULL;

//Declaring the function prototypes for the linked list operations
void insert (int);
int remove (int);
void print (void);
struct node *search (int);

int main()
{
    int num1, num2, choice;
    struct node *location;

    //Displaying a menu of choices for performing linked list operations
    while(1)
    {
        system("cls");
        cout<< "\n\n Select an option \n";
        cout<< "\n1 - Insert";
        cout<< "\n2 - Delete";
        cout<< "\n3 - Search";
        cout<< "\n4 - Print";
        cout<< "\n5 - Exit";

        cout<< "\n\nEnter your choice: ";
        cin>> choice;

        switch(choice)
        {
            case 1:
            {
                cout<< "\nEnter the Element to be inserted into the Linked
                List : ";
                cin>> num1;
                insert(num1); // Calling the insert() Function
                cout<< "\n" << num1 << " is successfully inserted into the
                Linked List";
                getch();
                break;
            }

            case 2:
            {
                cout<< "\nEnter the element to be deleted from the Linked List: ";
                cin>> num1;
                num2 = remove (num1); //Calling the remove() Function
                if(num2 == -9999)

```

(Contd.)

```

        cout<< "\n\t" << num1 << " is not present in the linked
        list\n\t";
    else
        cout<< "\n\tElement " << num2 << " is successfully deleted
        from the Linked List\n\t";
    getch();
    break;
}

case 3:
{
    cout<< "\nEnter the Element to be searched: ";
    cin>> num1;
    location = search (num1); //Calling the search() function
    if (location == NULL)
        cout<< "\n\t" << num1 <<" is not present in the Linked
        List\n\t";
    else
    {
        if (location == LAST)
            cout<< "\n\tElement " << num1 << " is the last element
            in the list";
        else
            cout<< "\n\tElement " << num1 << " is present before
            element " << (location-> NEXT)-> INFO << " in the linked
            list";
        }
    getch();
    break;
}

case 4:
{
    print(); //Printing the Linked List Elements;
    getch();
    break;
}

case 5:
{
    exit(1);
    break;
}

default:
{
    cout<< "\nIncorrect Choice. Please Try Again!!!\n";
    getch();
    break;
}
}

```

(Contd.)

```

    }
}
//Insert Function

void insert(int value)
{
    //Creating a New Node
    struct node *PTR = (struct node*)malloc(sizeof(struct node));

    //Storing the Element to be inserted in the List
    PTR -> INFO = value;

    //Linking the New Node to the Linked List
    if(FIRST == NULL)
    {
        FIRST = LAST = PTR;
        PTR -> NEXT = NULL;
    }
    else
    {
        LAST -> NEXT = PTR;
        PTR -> NEXT = NULL;
        LAST = PTR;
    }
}

// remove Function
int remove (int value)
{
    struct node *LOC, *TEMP;
    inti;
    i = value;

    LOC = search (i); //Calling the search() Function

    if(LOC == NULL) //Element Not Found
        return(-9999);

    if(LOC == FIRST)
    {
        if (FIRST == LAST)
            FIRST = LAST = NULL;
        else
            FIRST = FIRST -> NEXT;
        return(value);
    }

    for(TEMP = FIRST; TEMP -> NEXT != LOC; TEMP = TEMP -> NEXT)
    ;
    TEMP -> NEXT = LOC -> NEXT;
}

```

(Contd.)

```

    if(LOC == LAST)
        LAST = TEMP;
    return(LOC -> INFO);
}

// Search Function

struct node *search (int value)
{
    struct node *PTR;

    if(FIRST == NULL) //Checking for the Empty List
        return(NULL);

    // Searching the Linked List
    for(PTR = FIRST; PTR != LAST; PTR = PTR -> NEXT)
    {
        if(PTR -> INFO == value)
            return(PTR); // Returning the Location of the Searched Element
    }

    if(LAST -> INFO == value)
        return(LAST);
    else
        return(NULL); // Returning NULL value indicating unsuccessful search
}

// Print Function

void print()
{
    struct node *PTR;

    if(FIRST == NULL) //Checking whether the Linked List is Empty
    {
        cout<< "\n\tEmpty List!!\n";
        return;
    }

    cout<< "Linked List Elements :\n";
    if(FIRST == LAST) //Checking if there is only one element in the list
    {
        cout<< "\t" << FIRST -> INFO;
        return;
    }

    //Printing the List Elements
    for(PTR = FIRST; PTR != LAST; PTR = PTR -> NEXT)
        cout<< "\t" << PTR -> INFO;
}

```

(Contd.)

```
    cout<< "\\t" << LAST -> INFO;  
    return;  
}
```

Output

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 4

Empty List!!

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 1

Enter the Element to be inserted into the Linked List : 1

1 is successfully inserted into the Linked List

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 1

Enter the Element to be inserted into the Linked List : 2

2 is successfully inserted into the Linked List

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 1

Enter the Element to be inserted into the Linked List : 3

3 is successfully inserted into the Linked List

Select an option

- 1 - Insert
- 2 - Delete

(Contd.)

- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 3

Enter the Element to be searched: 5

5 is not present in the Linked List

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 3

Enter the Element to be searched: 2

Element 2 is present before element 3 in the linked list

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 2

Enter the element to be deleted from the Linked List: 2

Element 2 is successfully deleted from the Linked List

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 4

Linked List Elements :

1 3

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 5



12.4 TYPES OF LINKED LISTS

Depending on the manner in which its nodes are interconnected with each other, linked lists are categorized into the following types:

1. **Singly linked list** In this type of linked list, each node points at the successive node. Thus, the list can only be traversed in the forward direction. The linked list implementation that we saw in the previous section is an example of singly linked list.
2. **Circular list** In this type of linked list, the first and the last node are logically connected with each other, thus giving the impression of a circular list formation. Actually, the NEXT part of the last node contains the address of the FIRST node, thus connecting the rear of the list to its front.
3. **Doubly linked list** In this type of linked list, a node points at both its preceding as well as succeeding nodes. Thus, the list can be traversed in both forward as well as backward directions.



12.5 CIRCULAR LINKED LIST

The only difference between singly linked list and circular linked list is that the last node of singly linked list points at NULL while the last node of circular linked list points at the first list element. That means, the NEXT part of the last node of a circular linked list contains the address of its FIRST node. One of the main advantages of circular linked list is that it allows traversal of the complete list from any of its node, which is not possible with singly or doubly linked lists.

Figure 12.4 depicts the logical representation of a circular linked list.

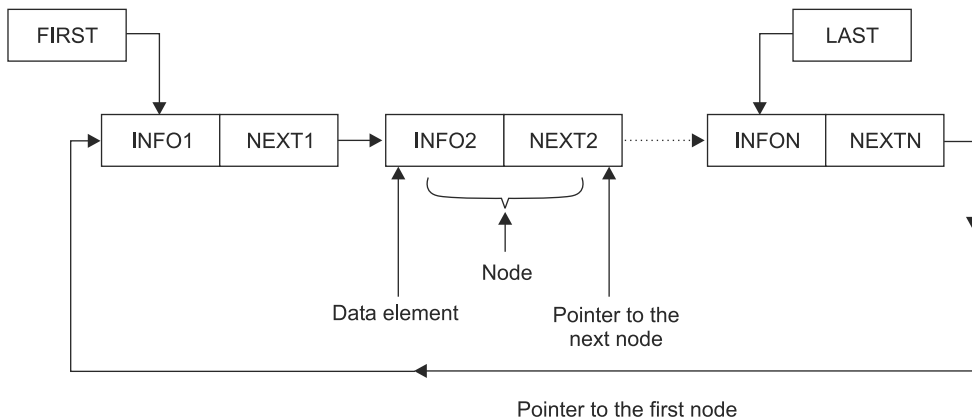


Fig. 12.4 Logical representation of a circular linked list

The implementation of a circular linked list involves two tasks:

1. Declaring the list node
2. Defining the list operations

The declaration of the circular linked list node is similar to the declaration of the singly linked list node. However, the definition of certain operations of a circular linked list is slightly different than that of the singly linked list.

12.12.1 Circular Linked List Operations

The typical operations performed on a circular linked list are:

1. Insert
2. Delete
3. Search
4. Print

1. Insert The insert operation in a circular list is performed in the same manner as a singly linked list. The only exception is when the element is inserted at the end of the list. In such a case, the NEXT pointer of the newly inserted node is assigned the address of the first element in the list, thus ensuring that the list stays circular.

Example 12.5 Algorithm to Insert an Element at the End of a Circular Linked List

```
insert (value)
Step 1: Start
Step 2: Set PTR = addressof (New Node)
        //Allocate a new node and assign its address to the pointer PTR
Step 3: Set PTR->INFO = value;
        //Store the element value to be inserted in the INFO part of the new node
Step 4: If FIRST = NULL, then goto Step 5 else goto Step 7
        //Check whether the existing list is empty
Step 5: Set FIRST=PTR and LAST=PTR
        //Update the FIRST and LAST pointers
Step 6: Set PTR->NEXT = FIRST and goto Step 8
        //Create a circular link
Step 7: Set LAST->NEXT=PTR, PTR->NEXT=FIRST and LAST=PTR
        //Add the newly created node at the end of the list and link it with
the first node
Step 8: Stop
```

2. Delete The delete operation in a circular list is performed in the same manner as a singly linked list. The only exception is when the element to be deleted is at the end of the list. In such a case, the NEXT pointer of the second last node in the list is assigned the address of the first element to ensure that the list stays circular.

Example 12.6 Algorithm to Delete an Element from a Circular Linked List

```
delete (value)
Step 1: Start
Step 2: Set LOC = search (value)
        //Call the search module to search the location of the node to be
deleted and assign it to LOC pointer
Step 3: If LOC=NULL goto Step 4 else goto Step 5
Step 4: Return ("Delete operation unsuccessful: Element not present")
and Stop
```

(Contd.)

```

Step 5: If LOC=FIRST goto Step 6 else goto Step 11
        //Check if the element to be deleted is the first element in the list
Step 6: If FIRST=LAST goto Step 7 else goto Step 8
        //Check if there is only one element in the list
Step 7: Set FIRST=LAST=NULL and goto Step 10
Step 8: Set FIRST=FIRST->NEXT
        //Reset the FIRST pointer
Step 9: Set Last->NEXT=FIRST
        //Link the last node with the updated FIRST pointer
Step 10: Return ("Delete operation successful") and Stop
Step 11: Set TEMP=LOC-1
        //Assign the location of the node present before LOC to temporary
pointer TEMP
Step 11: Set TEMP->NEXT=LOC->NEXT
        //Link the TEMP node with the node being currently pointed by LOC
Step 12: If LOC=LAST goto Step 13 else goto Step 15
        //Check if the element to be deleted is currently the last element in
the list
Step 13: Set LAST=TEMP
Step 14: Set TEMP->NEXT=FIRST
        //Create circular link
Step 15: Return ("Delete operation successful")
Step 16: Stop

```

3. Search The search operation in a circular linked list is performed in the same manner as a singly linked list. The circular list also provides the additional flexibility of starting the search from anywhere in the list. An unsuccessful search is signified when the same node is reached from where the search was started.

4. Print The print operation in a circular list is performed in the same manner as a singly linked list. The circular nature of the list allows us to start the print operation from anywhere in the list.

12.5.2 Circular Linked List Implementation

The implementation of circular linked list involves declaring its structure and defining its operations. The following example shows how a circular linked list is implemented in C.

Program 12.2 implements a circular linked list in C. It uses the insert (Example 12.5) and delete (Example 12.6) algorithms for realizing the insert and delete operations on the circular linked list. For performing the search and print operations, the same algorithms (Example 12.3 and Example 12.4) have been used that were earlier used for implementing a singly linked list.

Program 12.2 Implementation of a Circular Linked List

```

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

using namespace std;

```

(Contd.)

```

// Circular Linked List Declaration
structcl_node
{
    int INFO;
    structcl_node *NEXT;
};

// Declaring pointers to First and Last node of the Linked List
structcl_node *FIRST = NULL;
structcl_node *LAST = NULL;

//Declaring the function prototypes for the List Operations
void insert (int);
int remove (int);
void print (void);
structcl_node *search (int);

int main()
{
    int num1, num2, choice;
    structcl_node *location;

    //Displaying a menu of choices for performing List operations
    while(1)
    {
        system("cls");
        cout<< "\n\n Select an option \n";
        cout<< "\n1 - Insert";
        cout<< "\n2 - Delete";
        cout<< "\n3 - Search";
        cout<< "\n4 - Print";
        cout<< "\n5 - Exit";

        cout<< "\n\nEnter your choice: ";
        cin>> choice;

        switch(choice)
        {
            case 1:
            {
                cout<< "\nEnter the Element to be inserted into the
                Circular Linked List : ";
                cin>> num1;
                insert(num1); // Calling the insert() Function
                cout<< "\n" << num1 << " is successfully inserted into
                the Circular Linked List";
                getch();
                break;
            }

            case 2:
            {

```

(Contd.)

```

        cout<< "\nEnter the element to be deleted from the
        Circular Linked List: ";
        cin>> num1;
        num2 = remove (num1); //Calling the remove()
        Function
        if(num2 == -9999)
            cout<< "\n\t" << num1 << " is not present in the
            list\n\t";
        else
            cout<< "\n\tElement " << num2 << " is successfully
            deleted from the Circular Linked List\n\t";
        getch();
        break;
    }

    case 3:
    {
        cout<< "\nEnter the Element to be searched: ";
        cin>> num1;
        location = search (num1); //Calling the search() function
        if (location == NULL)
            cout<< "\n\t" << num1 <<" is not present in the List\n\t";
        else
            cout<< "\n\tElement " << num1 << " is present before
            element " << (location -> NEXT) -> INFO << " in
            the circular linked list";
        getch();
        break;
    }

    case 4:
    {
        print(); //Printing the Linked List Elements;
        getch();
        break;
    }

    case 5:
    {
        exit(1);
        break;
    }

    default:
    {
        cout<< "\nIncorrect Choice. Please Try Again!!!\n";
        getch();
        break;
    }
}
}
}

```

(Contd.)

```

//Insert Function

void insert(int value)
{
    //Creating a New Node
    structcl_node *PTR = (structcl_node*)malloc(sizeof(structcl_node));

    //Storing the Element to be inserted in the new node
    PTR -> INFO = value;

    //Linking the New Node to the Circular Linked List
    if(FIRST == NULL)
    {
        FIRST = LAST = PTR;
        PTR -> NEXT = NULL;
    }
    else
    {
        LAST -> NEXT = PTR;
        PTR -> NEXT = NULL;
        LAST = PTR;
    }
}

// Remove Function
int remove (int value)
{
    structcl_node *LOC, *TEMP;
    inti;
    i = value;

    LOC = search (i); //Calling the search() Function

    if(LOC == NULL) //Element Not Found
        return(-9999);

    if(LOC == FIRST)
    {
        if (FIRST == LAST)
            FIRST = LAST = NULL;
        else
        {
            FIRST = FIRST -> NEXT;
            LAST -> NEXT = FIRST;
        }
        return(value);
    }

    for(TEMP = FIRST; TEMP -> NEXT != LOC; TEMP = TEMP -> NEXT)
    ;
}

```

(Contd.)

```

    if(LOC == LAST)
    {
        LAST = TEMP;
        TEMP -> NEXT = FIRST;
    }
    else
        TEMP -> NEXT = LOC -> NEXT;

    return(LOC -> INFO);
}

// Search Function
structcl_node *search (int value)
{
    structcl_node *PTR;

    if(FIRST == NULL) //Checking for the Empty List
        return(NULL);

    if(FIRST == LAST && FIRST -> INFO == value) //Checking if there is
only one element in the List
        return(FIRST);

    // Searching the Linked List
    for(PTR = FIRST; PTR != LAST; PTR = PTR -> NEXT)
    {
        if(PTR -> INFO == value)
            return(PTR); // Returning the Location of the Searched Element
    }

    if(LAST -> INFO == value)
        return(LAST);
    else
        return(NULL); // Returning NULL value indicating unsucessful search
}

// Print Function
void print()
{
    structcl_node *PTR;

    if(FIRST ==NULL) //Checking wether the Linked List is Empty
    {
        cout<< "\n\tEmpty List!!\n";
        return;
    }

    cout<< "Circular Linked List Elements :\n";
    if(FIRST == LAST) //Checking if there is only one element in the list
    {
        cout<< "\t" << FIRST -> INFO;
        return;
    }
}

```

(Contd.)

```
//Printing the List Elements
for(PTR = FIRST; PTR != LAST; PTR = PTR -> NEXT)
    cout<< "\\t" << PTR -> INFO;

cout<< "\\t" << LAST -> INFO;
return;
}
```

Output

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 4

Empty List!!

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 1

Enter the Element to be inserted into the Circular Linked List : 1

1 is successfully inserted into the Linked List

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 1

Enter the Element to be inserted into the Circular Linked List : 2

2 is successfully inserted into the Linked List

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 1

Enter the Element to be inserted into the Circular Linked List : 3

(Contd.)

```
3 is successfully inserted into the Linked List
Select an option
1 - Insert
2 - Delete
3 - Search
4 - Print
5 - Exit
Enter your choice: 3
Enter the Element to be searched: 5
    5 is not present in the Circular Linked List
Select an option
1 - Insert
2 - Delete
3 - Search
4 - Print
5 - Exit
Enter your choice: 3
Enter the Element to be searched: 3
    Element 3 is present before element 1 in the Circular Linked list
Select an option
1 - Insert
2 - Delete
3 - Search
4 - Print
5 - Exit
Enter your choice: 2
Enter the element to be deleted from the Circular Linked List: 2
    Element 2 is successfully deleted from the Circular Linked List
Select an option
1 - Insert
2 - Delete
3 - Search
4 - Print
5 - Exit
Enter your choice: 4
Linked List Elements :
    1      3
Select an option
1 - Insert
2 - Delete
3 - Search
4 - Print
5 - Exit
Enter your choice: 5
```




12.6 DOUBLY LINKED LIST

Each node of a doubly linked list has three parts: INFO, NEXT, and PREVIOUS. The INFO part contains the data element while the NEXT and PREVIOUS parts contain the address of the next and previous nodes respectively. The NEXT part of the last node of the list contains a NULL value indicating the end of the list. The beginning of the list is indicated with the help of a special pointer called FIRST.

The main advantage of a doubly linked list is that it allows both forward and backward traversal. Figure 12.5 depicts the logical representation of a doubly linked list:

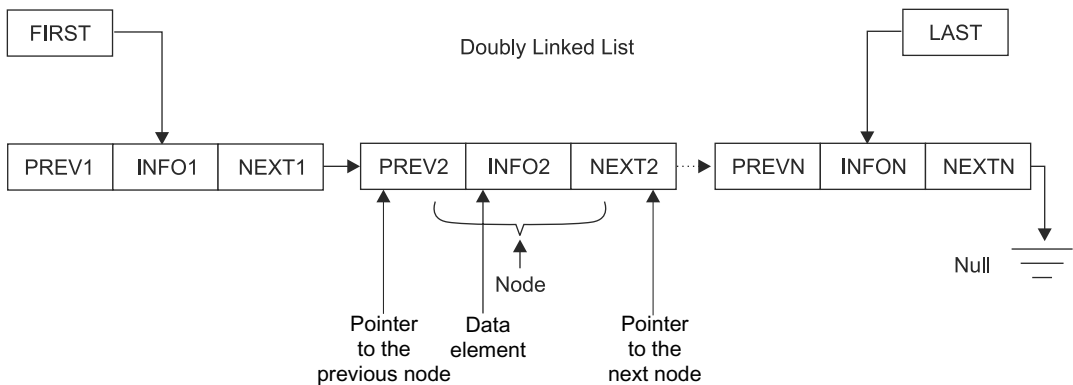


Fig. 12.5 Logical representation of a doubly linked list

The implementation of a doubly linked list involves two tasks:

1. Declaring the list node
2. Defining the list operations

12.6.1 Doubly Linked List Node Declaration

The following structure declaration defines the node of a doubly linked list:

```
struct node
{
    int INFO;
    struct node *NEXT;
    struct node *PREVIOUS;
};
typedef struct node NODE;
```

The above structure declaration defines a new data type called NODE that represents a doubly linked list node. The node structure contains three members, INFO for storing integer data values, NEXT for storing address of the next node, and PREVIOUS for storing the address of the previous node.

12.6.2 Doubly Linked List Operations

The typical operations performed on a doubly linked list are

1. Insert
2. Delete
3. Search
4. Print

1. Insert The insert operation in a doubly linked list is performed in the same manner as a singly linked list. The only exception is that the additional node pointer PREVIOUS is also required to be updated for the new node at the time of insertion.

Example 12.7 Algorithm to Insert an Element at the End of a Doubly Linked List

```
insert (value)
Step 1: Start
Step 2: Set PTR = addressof (New Node)
        //Allocate a new node and assign its address to the pointer PTR
Step 3: Set PTR->INFO = value;
        //Store the element value to be inserted in the INFO part of the new
node
Step 4: If FIRST = NULL, then goto Step 5 else goto Step 7
        //Check whether the existing list is empty
Step 5: Set FIRST=PTR and LAST=PTR
        //Update the FIRST and LAST pointers
Step 6: Set PTR->NEXT = PTR -> PREVIOUS = NULL and goto Step 8
Step 7: Set LAST->NEXT=PTR, PTR->PREVIOUS = LAST, PTR->NEXT=NULL, and
LAST=PTR
        //Link the newly created node at the end of the list
Step 8: Stop
```

2. Delete The delete operation in a doubly linked list is performed in the same manner as a singly linked list. The only exception is that the additional node pointer PREVIOUS of the adjacent node is also required to be updated at the time of deletion.

Example 12.8 Algorithm to Delete an Element from a Doubly Linked List

```
delete (value)
Step 1: Start
Step 2: Set LOC = search (value)
        //Call the search module to search the location of the node to be
deleted and assign it to LOC pointer
Step 3: If LOC=NULL goto Step 4 else goto Step 5
Step 4: Return ("Delete operation unsuccessful: Element not present")
and Stop
Step 5: If LOC=FIRST goto Step 6 else goto Step 10
        //Check if the element to be deleted is the first element in the list
Step 6: If FIRST=LAST goto Step 7 else goto Step 8
        //Check if there is only one element in the list
Step 7: Set FIRST=LAST=NULL and goto Step 9
Step 8: Set FIRST->NEXT->PREVIOUS=NULL and FIRST=FIRST->NEXT
```

(Contd.)

```

Step 9: Return ("Delete operation successful") and Stop
Step 10: Set TEMP=LOC-1
        //Assign the location of the node present before LOC to temporary
pointer TEMP
Step 11: If LOC=LAST goto Step 12 else goto Step 13
Step 12: Set LAST=TEMP, TEMP->NEXT=NULL and goto Step 15
Step 13: Set TEMP->NEXT=LOC->NEXT
Step 14: Set LOC->NEXT->PREVIOUS=TEMP
        //Delete the LOC node and set the adjacent NEXT and PREVIOUS pointers
Step 15: Return ("Delete operation successful")
Step 16: Stop

```

3. Search The search operation in a doubly linked list is performed in the same manner as a singly linked list. The doubly linked list also provides the additional flexibility of starting the search from the end and moving backwards towards the front.

4. Print The print operation in a doubly linked list is performed in the same manner as a singly linked list. The doubly linked list also allows you to print the list elements in reverse order by starting from the end and moving backwards towards the front.

12.6.3 Doubly Linked List Implementation

The implementation of doubly linked list involves declaring its structure and defining its operations. The following example shows how a doubly linked list is implemented in C.

Program 12.3 implements a doubly linked list in C. It uses the insert (Example 12.7) and delete (Example 12.8) algorithms for realizing the insert and delete operations on the doubly linked list. For performing the search and print operations, the same algorithms (Example 12.3 and Example 12.4) have been used that were earlier used for implementing a singly linked list.

Program 12.3 Doubly Linked List Implementation

```

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

using namespace std;

// Doubly Linked List Declaration
struct dl_node
{
    int INFO;
    struct dl_node *NEXT;
    struct dl_node *PREVIOUS;
};

// Declaring pointers to First and Last node of the Doubly Linked List
struct dl_node *FIRST = NULL;
struct dl_node *LAST = NULL;

//Declaring the function prototypes for the List Operations
void insert (int);

```

(Contd.)

```

int remove (int);
void print (void);
struct dl_node *search (int);

int main()
{
    int num1, num2, choice;
    struct dl_node *location;

    //Displaying a menu of choices for performing List operations
    while(1)
    {
        system("cls");
        cout<< "\n\n Select an option \n";
        cout<< "\n1 - Insert";
        cout<< "\n2 - Delete";
        cout<< "\n3 - Search";
        cout<< "\n4 - Print";
        cout<< "\n5 - Exit";

        cout<< "\n\nEnter your choice: ";
        cin>> choice;

        switch(choice)
        {

            case 1:
            {

                cout<< "\nEnter the Element to be inserted into the Doubly
                Linked List : ";
                cin>> num1;
                insert(num1); // Calling the insert() Function
                cout<< "\n" << num1 << " is successfully inserted into
                the Doubly Linked List";
                getch();
                break;
            }

            case 2:
            {

                cout<< "\nEnter the element to be deleted from the Doubly
                Linked List: ";
                cin>> num1;
                num2 = remove (num1); //Calling the remove() Function
                if(num2 == -9999)
                    cout<< "\n\t" << num1 << " is not present in the list\n\t";
                else
                    cout<< "\n\tElement " << num2 << " is successfully
                    deleted from the Doubly Linked List\n\t";
            }
        }
    }
}

```

(Contd.)

```

        getch();
        break;
    }

    case 3:
    {
        cout<< "\nEnter the Element to be searched: ";
        cin>> num1;
        location = search (num1); //Calling the search() function
        if (location == NULL)
            cout<< "\n\t" << num1 <<" is not present in the List\n\t";
        else
        {
            if (location == LAST)
                cout<< "\n\tElement " << num1 << " is the last
                element in the list";
            else
                cout<< "\n\tElement " << num1 << " is present before
element " << (location -> NEXT) -> INFO << " in the doubly linked list";
        }
        getch();
        break;
    }

    case 4:
    {
        print(); //Printing the Doubly Linked List Elements;
        getch();
        break;
    }

    case 5:
    {
        exit(1);
        break;
    }

    default:
    {
        cout<< "\nIncorrect Choice. Please Try Again!!!\n";
        getch();
        break;
    }
}

}

}

//Insert Function

void insert(int value)

```

(Contd.)

```

{
    //Creating a New Node
    structdl_node *PTR = (structdl_node*)malloc(sizeof(structdl_node));

    //Storing the Element to be inserted in the new node
    PTR -> INFO = value;

    //Linking the New Node to the Doubly Linked List
    if(FIRST == NULL)
    {
        FIRST = LAST = PTR;
        PTR -> NEXT = NULL;
        PTR -> PREVIOUS = NULL;
    }
    else
    {
        LAST -> NEXT = PTR;
        PTR -> NEXT = NULL;
        PTR -> PREVIOUS = LAST;
        LAST = PTR;
    }
}

// Remove Function
int remove (int value)
{
    structdl_node *LOC, *TEMP;
    inti;
    i = value;

    LOC = search (i); //Calling the search() Function

    if(LOC == NULL) //Element Not Found
        return(-9999);

    if(LOC == FIRST)
    {
        if (FIRST == LAST)
            FIRST = LAST = NULL;
        else
        {
            FIRST -> NEXT -> PREVIOUS = NULL;
            FIRST = FIRST -> NEXT;
        }
        return(value);
    }

    for(TEMP = FIRST; TEMP -> NEXT != LOC; TEMP = TEMP -> NEXT)
    ;
    if(LOC == LAST)

```

(Contd.)

```

    {
        LAST = TEMP;
        TEMP -> NEXT = NULL;
    }
    else
    {
        TEMP -> NEXT = LOC -> NEXT;
        LOC -> NEXT ->PREVIOUS = TEMP;
    }

    return(LOC -> INFO);
}

// Search Function

structdl_node *search (int value)
{
    structdl_node *PTR;

    if(FIRST == NULL) //Checking for the Empty List
        return(NULL);

    if(FIRST == LAST && FIRST -> INFO == value) //Checking if there is only
one element in the List
        return(FIRST);

    // Searching the Linked List
    for(PTR = FIRST; PTR != LAST; PTR = PTR -> NEXT)
    {
        if(PTR -> INFO == value)
            return(PTR); // Returning the Location of the Searched Element
    }

    if(LAST -> INFO == value)
        return(LAST);
    else
        return(NULL); // Returning NULL value indicating unsucessful search
}

// Print Function

void print()
{
    structdl_node *PTR;

    if(FIRST ==NULL) //Checking wether the Linked List is Empty
    {
        cout<< "\n\tEmpty List!!\n";
        return;
    }
}

```

(Contd.)

```

cout<< "Doubly Linked List Elements :\n";
if(FIRST == LAST) //Checking if there is only one element in the list
{
    cout<< "\t" << FIRST -> INFO;
    return;
}

//Printing the List Elements
for(PTR = FIRST; PTR != LAST; PTR = PTR -> NEXT)
    cout<< "\t" << PTR -> INFO;

cout<< "\t" << LAST -> INFO;
return;
}

```

Output

The Output of this program is same as Program 12.1 (Single Linked List Implementation)



12.7 SOME MORE PROGRAMS

Program 12.4 Declaring the Node of a Singly Linked List that Stores Student

```

struct student
{
    char name[30];
    introllno;
    float percentage;
};

struct node
{
    struct student S;
    struct node *NEXT;
};

typedef struct node NODE;

```

Program 12.5 Function to Print the Elements of a Doubly Linked List in Reverse Order

```

// Print Function

void print()
{
    struct node *PTR;

    if(FIRST ==NULL) //Checking whether the Linked List is Empty
    {

```

(Contd.)


```

        cout<< "\n\tEmpty List!!\n";
        return;
    }

    cout<< "Linked List Elements :\n";
    if(FIRST == LAST) //Checking if there is only one element in the list
    {
        cout<< "\t" << FIRST -> INFO;
        return;
    }

    //Printing the List Elements in the reverse order
    for(PTR = LAST; PTR != FIRST; PTR = PTR -> PREVIOUS)
        cout<< "\t" << PTR -> INFO;

    cout<< "\t" << FIRST -> INFO;
    return;
}

```

SUMMARY

- ☐ Linked list is a collection of nodes or data elements logically connected to each other.
- ☐ Each node of a linked list has two parts: INFO and NEXT. The INFO part contains the data element while the NEXT part contains the address of the next node in the list.
- ☐ The implementation of a linked list involves declaring the list node and defining the list operations.
- ☐ The typical operations performed on a linked list are: insert, delete, search and print.
- ☐ The various types of linked lists are: singly linked list, doubly linked list, and circular linked list.
- ☐ In a circular linked list, the first and last nodes are logically connected with each other through the NEXT pointer.
- ☐ In a doubly linked list, a node points at both its preceding as well as succeeding nodes.

Key Terms

Singly linked list | Circular list | Doubly linked list | INFO | NEXT | PREVIOUS | FIRST | LAST | Insert | Delete | Search | Print

Review Questions

- 12.1** What is a linked list? What are its various types?
- 12.2** Explain the representation of a linked list in memory with the help of an illustration.
- 12.3** Explain the typical operations that are performed on a linked list.
- 12.4** Explain the key advantages and disadvantages of linked lists.
- 12.5** What is a circular linked list? How is it different from a normal linked list?

- 12.6 What is a doubly linked list? Why is it used?
- 12.7 Write the algorithm for searching an element in a singly linked list.
- 12.8 Write the algorithm for inserting an element in a circular linked list.

Programming Exercises

- 12.1 Write a code snippet for declaring the node of a doubly linked list.
- 12.2 Write a C function to delete a node from a singly linked list.
- 12.3 Write a C function to insert a new node at the end of a circular linked list.
- 12.4 Write a C function to print the elements of a linked list.
- 12.5 Write a C function to print the elements of a doubly linked list in both forward and backward directions.

13

STACKS

CHAPTER OUTLINE

- 13.1 Introduction
 - 13.2 Stacks
 - 13.2.1 Stack Representation in Memory
 - 13.2.2 Arrays vs. Stacks
 - 13.3 Stack Operations
 - 13.3.1 Push
 - 13.3.2 Pop
 - 13.3.3 An Example of Stack Operations
 - 13.4 Stack Implementation
 - 13.4.1 Array Implementation of Stacks
 - 13.4.2 Linked Implementation of Stacks
- Summary*
- Key Terms*





13.1 INTRODUCTION

In the previous chapters, we learnt how arrays are used for implementing linear data structures. Arrays provide the flexibility of adding or removing elements anywhere in the list. But there are certain linear data structures that permit the insertion and deletion operations only at the beginning or end of the list, but not in the middle. Such data structures have significant importance in systems processes such as compilation and program control.

Stack is one such data structure which is in fact one of the very first data structures that students get familiar with while studying this subject.



13.2 STACKS

Stack is a linear data structure in which items are added or removed only at one end, called *top of the stack*. Thus, there is no way to add or delete elements anywhere else in the stack. A stack is based on Last-In-First-Out (LIFO) principle that means the data item that is inserted last into the stack is the first one to be removed from the stack. We can relate a stack to certain real-life objects and situations, as shown in Figs. 13.1 (a) and (b).

As we can see in Fig. 13.1, one can add a new book to an existing stack of books only at its top and nowhere else. Similarly, a plate cannot be added at the middle of the plates stack; one has to first remove all the plates above the insertion point for the new plate to be added there. Another apt example of a stack is a set of bangles worn by Indian women on their arms. A bangle can only be worn from one side of the hand and to remove a bangle from the middle one has to first remove all the prior bangles.

The concept of stack in data structures follows the same analogy as the stack of books or the stack of plates. We may use a stack in data structures to store built-in or user-defined type elements depending upon our programming requirements. Irrespective of the type of elements stored, each stack implementation follows similar representation in memory, as explained next.

A stack of books



Fig. 13.1(a) Stack of books

A plate can not be added at the middle



Fig. 13.1(b) Stack of plates



Mind Jog

Who discovered stacks?

Stack was first proposed in 1957 by a German computer scientist Friedrich L. Bauer.

13.2.1 Stack Representation in Memory

Just like their real world counterparts, stacks appear as a group of elements stored at contiguous locations in memory. Each successive insert or delete operation adds or removes an item from the group. The top location of the stack or the point of addition or deletion is maintained by a pointer called *top*. Figure 13.2 shows the logical representation of stacks in memory.

As we can see in Fig. 13.2, there are six elements in the stack with element 16 being at the top of the stack.



Note The logical representation of stacks showing stack elements stored at contiguous memory location might be true in case of their array implementation but the same might not be true in case of their linked implementation, as we shall study later in this chapter.

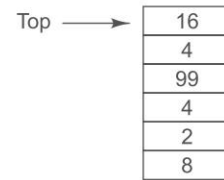


Fig. 13.2 Logical representation of stacks

13.2.2 Arrays vs. Stacks

While both arrays and stacks may look to be similar in their logical representation, they are different in several aspects, as explained in Table 13.1.

Table 13.1 Arrays vs. Stacks

Arrays	Stacks
Arrays provide the flexibility of adding or removing data elements anywhere in the list, i.e., at the beginning, end or anywhere in the middle. While this flexibility may seem to be a boon in certain situations, the same may not be true in situations where frequent insertions or deletions are required. This is because; each insertion or deletion in arrays requires the adjoining elements to be shifted to new locations, which is an overhead.	Stacks restrict the insertion or deletion of elements to only one place in the list i.e. the top of the stack. Thus, there are no associated overheads of shifting other elements to new locations.
By using arrays, a programmer can realize common scenarios where grouping of records is required, for example inventory management, employee records management, etc.	Stacks find their usage as vital in solutions to advanced systems problems such as recursion control, expression evaluation, etc.



Check Point

1. What is a stack?

Ans. Stack is a linear data structure in which items are added or removed only at one end, called top.

2. What is LIFO?

Ans. Last-In-First-Out (LIFO) principle specifies that the data item that is inserted last into the stack is the first one to be removed from the stack.



13.3 STACK OPERATIONS

There are two key operations associated with the stack data structure: push and pop. Adding an element to the stack is referred as push operation while reading or deleting an element from the stack is referred as pop operation. Figures 13.3 (a) and (b) depict the push and pop operations on a stack.

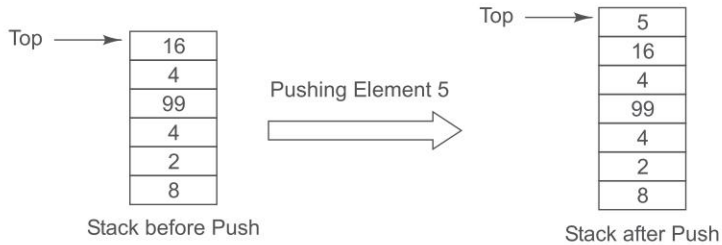


Fig. 13.3(a) Push operation

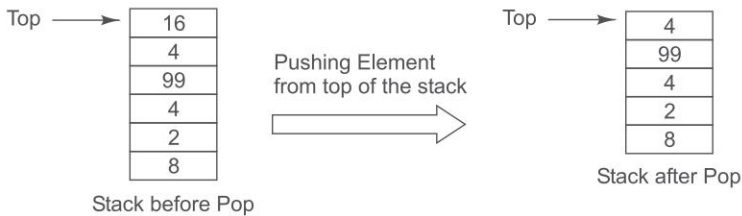


Fig. 13.3(b) Pop operation

Note *Top is the cornerstone of the stack data structure as it points at the entry/exit gateway of the stack.*

13.3.1 Push

As we can see in Fig. 13.3 (a), the push operation involves the following subtasks:

1. Receiving the element to be inserted
2. Incrementing the stack pointer, top
3. Storing the received element at new location of top

Thus, the programmatic realization of the push operation requires implementation of the above mentioned subtasks, as we shall see later in this chapter.

Tip *What happens if the stack is full and there is no more room to push any new element? Such a condition is referred as stack overflow. It is always advisable to implement appropriate overflow handling mechanisms in a program to counter any unexpected results.*

13.3.2 Pop

As we can see in Fig. 13.3 (b), the pop operation involves the following subtasks:

- Retrieving or removing the element at the top of the stack.
- Decrementing the stack pointer, top.

Thus, the programmatic realization of the pop operation requires implementation of the above mentioned subtasks, as we shall see later in this chapter.

Some pop implementations require the popped element to be returned back to the calling function while others may simply focus on updating the stack pointer and ignore the popped element all together. The choice of a particular type of implementation depends solely on the programming situation at hand.



Mind Jog

Which programming language provide built-in support for stacks?

LISP and Python



Tip

What happens if the stack is empty and we want to perform the pop operation? Such a condition is referred as stack underflow. It is always advisable to implement appropriate underflow handling mechanisms in a program to counter any unexpected results.

13.3.3 An Example of Stack Operations

Figure 13.4 shows how the stack contents change after a series of push and pop operations.

We can see in this figure how stack contents change by the push/pop operations occurring at one end of the stack, i.e., its top.



13.4 STACK IMPLEMENTATION

Stack implementation involves choosing the data storage mechanism for storing stack elements and implementing methods for performing the two stack operations, push and pop. A typical implementation of the push operation checks if there is any room left in the stack, and if there is any, it increments the stack counter by one and inserts the received item at the top of the stack. Similarly, the implementation of the pop operation checks whether or not the stack is already empty, if it is not, it removes the top element of the stack and decrements the stack counter by one.

We can implement stacks by using arrays or linked lists. The advantages or disadvantages of array or linked implementations of stacks are the same that are associated with such types of data structures. However, both implementation types have their own usage in specific situations.

13.4.1 Array Implementation of Stacks

The array implementation of stacks involves allocation of fixed size array in the memory. Both stack operations (push and pop) are made on this array with a constant check being made to ensure that the array does not go out of bounds.



Check Point

1. What is a push operation?

Ans. Adding an element into the stack is referred as push operation.

2. What is a pop operation?

Ans. Reading or deleting an element from the stack is referred as pop operation.



Mind Jog

Is stack a restricted data structure?

Yes, because limited operations can be performed on it.

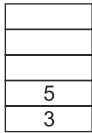
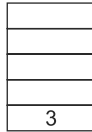
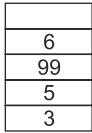
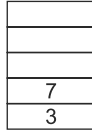
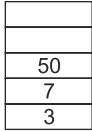
Initial Empty Stack	Top → Null  Empty Stack
Push (3), Push (5)	Top →  Stack Contents
Pop ()	Top →  Stack Contents
Push(7), Push (99), Push (6)	Top →  Stack Contents
Pop (), Pop ()	Top →  Stack Contents
Push (50)	Top →  Stack Contents
Pop (), Pop (), Pop ()	Top → Null  Empty Stack

Fig 13.4 Stack operations

Push Operation The push operation involves checking whether or not the stack pointer is pointing at the upper bound of the array. If it is not, the stack pointer is incremented by 1 and the new item is pushed (inserted) at the top of the stack.

Example 13.1 Algorithm to Implement Push Operation under Array Representation of Stacks

```
push(stack[MAX], element)
```

```
Step 1: Start
```

```
Step 2: If top = MAX-1 goto Step 3 else goto Step 4
```

```
Step 3: Display message "Stack Full" and exit
```

```
Step 4: top = top + 1
```

```
Step 5: stack[top] = element
```

```
Step 6: Stop
```

The above algorithm inserts an element at the top of a stack of size MAX.

Pop Operation The pop operation involves checking whether or not the stack pointer is already pointing at NULL (empty stack). If it is not, the item that is being currently pointed is popped (removed) from the stack (array) and the stack pointer is decremented by 1.

Example 13.2 Algorithm to Implement Pop Operation under Array Representation of Stacks

```
pop(stack[MAX], element)
```

```
Step 1: Start
```

```
Step 2: If top = -1 goto Step 3 else goto Step 4
```

```
Step 3: Display message "Stack Empty" and exit
```

```
Step 4: Return stack[top] and set top = top - 1
```

```
Step 5: Stop
```

The above algorithm removes the element at the top of the stack.

Implementation

Program 13.1 implements a stack using arrays in C. It uses the push (Example 13.1) and pop (Example 13.2) algorithms for realizing the common stack operations.

Program 13.1 Implementing a Stack Using Arrays

```
// Program for demonstrating implementation of stacks using arrays
```

```
#include<iostream>
```

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

```
using namespace std;
```

```
int stack[100]; //Declaring a 100 element stack array
```

```
int top = -1; // Declaring and initializing the stack pointer
```

(Contd.)

```

void push(int); // Declaring a function prototype for inserting an element
into the stack

int pop(); // Declaring a fuction prototype for removing an element from
the stack
void display(); // Declaring a fuction prototype for displaying the elements
of a stack

int main()
{
    int choice;
    int num1 = 0, num2 = 0;

    while(1)
    {
        system("cls");
        // Creating an interactive interface for performing stack operation

        cout<< "\nSelect a choice from the following:\n";
        cout<< "[1] Push an Element into the Stack\n";
        cout<< "[2] Pop out an Element from the Stack\n";
        cout<< "[3] Display the Stack Elements\n";
        cout<< "[4] Exit \n\n";
        cout<< "\n\tYour Choice : ";
        cin>> choice;

        switch(choice)
        {
            case 1:
            {
                cout<< "\n\tEnter the Element to be pushed into the Stack : ";
                cin>> num1;
                push(num1); //Inserting an Element
                break;
            }

            case 2:
            {
                num2 = pop(); // Removing an Element
                cout<< "\n\t" << num2 << " element popped out of the Stack \n\t";
                getch();
                break;
            }

            case 3:
            {
                display(); // Displaying Stack elements
                getch();
            }
        }
    }
}

```

(Contd.)

```

        break;
    }

    case 4:
    {
        exit(1);
        break;
    }

    default:
    {
        cout<< "\nInvalid Choice!!!\n";
        break;
    }
}
}

//Push Fuction
void push(int element)
{
    if(top == 99) //Checking whether stack is full
    {
        cout<< "Stack is Full.\n";
        getch();
        exit(1);
    }

    top = top + 1; // Incrementing the Stack Pointer
    stack[top] = element; // Inserting the new Element
}

//Pop Fuction
int pop()
{
    if(top == -1) // Checking whether the Stack is Empty
    {
        cout<< "\n\tStack is Empty.\n";
        getch();
        exit(1);
    }

    return(stack[top--]); //Returning the top Element and decrementing
    the Stack Pointer
}

//Display Function

```

(Contd.)

```
void display()
{
    inti;
    if(top == -1)
    {
        cout<< "\n\tStack is Empty.\n";
        getch();
    }
    else
        cout<< "\n\tThe various Stack Elements are : \n";
    for(i = top; i>= 0; i--)
        cout<< "\t" << stack[i] << "\n";
}
```

Output

Select a choice from the following:

- [1] Push an Element into the Stack
- [2] Pop out an Element from the Stack
- [3] Display the Stack Elements
- [4] Exit

Your Choice : 1

Enter the Element to be pushed into the Stack : 1

Select a choice from the following:

- [1] Push an Element into the Stack
- [2] Pop out an Element from the Stack
- [3] Display the Stack Elements
- [4] Exit

Your Choice : 1

Enter the Element to be pushed into the Stack : 2

Select a choice from the following:

- [1] Push an Element into the Stack
- [2] Pop out an Element from the Stack
- [3] Display the Stack Elements
- [4] Exit

Your Choice : 1

Enter the Element to be pushed into the Stack : 3

Select a choice from the following:

- [1] Push an Element into the Stack
- [2] Pop out an Element from the Stack
- [3] Display the Stack Elements
- [4] Exit

(Contd.)

```
Your Choice : 3

The various Stack Elements are :
3
2
1

Select a choice from the following:
[1] Push an Element into the Stack
[2] Pop out an Element from the Stack
[3] Display the Stack Elements
[4] Exit

Your Choice : 2

Element 3 is popped out of the Stack

Select a choice from the following:
[1] Push an Element into the Stack
[2] Pop out an Element from the Stack
[3] Display the Stack Elements
[4] Exit

Your Choice : 3

The various Stack Elements are :
2
1

Select a choice from the following:
[1] Push an Element into the Stack
[2] Pop out an Element from the Stack
[3] Display the Stack Elements
[4] Exit

Your Choice : 4
```

**Tip**

The above code shows storage of an integer type element into the stack. However, we may store other built-in or user-defined type elements in the stack as per our own requirements.

13.4.2 Linked Implementation of Stacks

The linked implementation of stacks involves dynamically allocating memory space at run time while performing stack operations. Since, the allocation of memory space is dynamic, the stack consumes only that much amount of space as is required for holding its data elements. This is contrary to array-implemented stacks which continue to occupy a fixed memory space even if there are no elements present. Thus, linked implementation of stacks based on dynamic memory allocation technique prevents wastage of memory space.

**Note**

The linked implementation of stacks is based on dynamic memory management techniques, which allow allocation and deallocation of memory space at runtime.

Push Operation The push operation under linked implementation of stacks involves the following tasks:

1. Reserving memory space of the size of a stack element in memory
2. Storing the pushed (inserted) value at the new location
3. Linking the new element with existing stack
4. Updating the stack pointer

Example 13.3 Algorithm to Implement Push Operation under Linked Representation of Stacks

```
push(structure stack, element, next, value)
```

Step 1: Start

Step 2: Set `ptr=(struct stack*)malloc(sizeof(struct stack))`, to reserve a block of memory for the new stack node and assign its address to pointer `ptr`

Step 3: Set `ptr->element=value`, to copy the inserted value into the new node

Step 4: Set `ptr->next=top`, to link the new node to the current top node

Step 5: Set `top = ptr` to designate the new node as the top node

Step 6: Return

Step 7: Stop

The above algorithm inserts an element at the top of the stack.

Pop Operation The pop operation under linked implementation of stacks involves the following tasks:

1. Checking whether the stack is empty
2. Retrieving the top element of the stack
3. Updating the stack pointer
4. Returning the retrieved (popped) value

Example 13.4 Algorithm in C to Implement Pop Operation under Linked Representation of Stacks

```
pop(structure stack, element, next)
```

Step 1: Start

Step 2: If `top = NULL` goto Step 3 else goto Step 4

Step 3: Display message "Stack Empty" and exit

Step 4: Set `temp=top->element`, to retrieve the element at top node of the stack

Step 5: Set `top=top->next`, to designate the next stack node as the top node

Step 6: Return `temp`

Step 7: Stop

The above algorithm removes the element at the top of the stack.

Implementation

Program 13.2 implements a stack using linked lists in C. It uses the push (Example 13.4) and pop (Example 13.5) algorithms for realizing the common stack operations.

Program 13.2 Stack Implementation using Linked List

```

// Program for demonstrating implementation of stacks using Linked List
#include<iostream>
#include<stdlib.h>
#include<conio.h>

using namespace std;
struct stack //Declaring the structure for Stack Elements
{
    int element;
    struct stack *next; // Stack Element pointing to another Stack Element
} *top;

void push(int); // Declaring a function prototype for inserting an element
into the stack
int pop(); // Declaring a fuction prototype for removing an element from
the stack
void display(); // Declaring a fuction prototype for displaying the elements
of a stack

int main()
{
    int choice;
    int num1 = 0, num2 = 0;

    while(1)
    {
        system("cls");
        // Creating an interactive interface for performing stack operation

        cout<< "\nSelect a choice from the following:\n";
        cout<< "[1] Push an Element into the Stack\n";
        cout<< "[2] Pop out an Element from the Stack\n";
        cout<< "[3] Display the Stack Elements\n";
        cout<< "[4] Exit \n\n";
        cout<< "\n\tYour Choice : ";
        cin>> choice;

        switch(choice)
        {
            case 1:
            {
                cout<< "\n\tEnter the Element to be pushed into
                the Stack : ";
                cin>> num1;
                push(num1); //Inserting an Element
                break;
            }
        }
    }
}

```

(Contd.)

```

        case 2:
        {
            num2 = pop(); // Removing an Element
            cout<< "\n\tElement " << num2 << " is popped out
            of the Stack \n\t";
            getch();
            break;
        }
        case 3:
        {
            display(); // Displaying Stack elements
            getch();
            break;
        }

        case 4:
        {
            exit(1);
            break;
        }

        default:
        {
            cout<< "\nInvalid Choice!!!\n";
            break;
        }
    }
}

//Push Fuction
void push(int value)
{
    struct stack *ptr = (struct stack*)malloc(sizeof(struct stack)); //
    Dynamically allocating memory space to store Stack Elements

    ptr -> element = value; //Assigning value to the newly allocated Stack
    Element

    //Updating Stack Pointers
    ptr -> next = top;
    top = ptr;
    return;
}

//Pop Fuction
int pop()
{

```

(Contd.)


```

    if(top == NULL) // Checking whether the Stack is Empty
    {
        cout<< "\n\tStack is Empty.\n";
        getch();
        exit(1);
    }
    else
    {
        int temp = top -> element; // Retrieving the Top Element
        top = top -> next; // Updating the Stack Pointer
        return(temp); //Returning the Popped Value
    }
}

//Display Function

void display()
{
    struct stack *ptr1 = NULL;
    ptr1 = top;
    cout<< "\nThe Various Stack Elements are : \n";
    while(ptr1 != NULL)
    {
        cout<< "\t" << ptr1 -> element << "\n";
        ptr1 = ptr1 -> next;
    }
}

Output

```

The Output of this program is same as the Program 13.1



Tip

It is advisable to check the overflow condition even with linked implementation of stacks, as in certain situations the available memory may also run out of space.



Tip

It is a good programming practice to release unused memory space so as to ensure efficient memory space utilization.



Check Point

1. What is array implementation of stacks?

Ans. It involves allocation of fixed size array in the memory for storing stack elements. Both push and pop operations are performed on this array-implemented stack.

2. What is linked implementation of stacks?

Ans. It involves dynamic allocation of memory space at run time while performing stack operations.

SOLVED PROBLEMS

Problem 13.1 The contents of a stack S are as follows:

Stack (S)	99	2	44	8				
Index	0	1	2	3↑	4	5	6	7

The stack can store a maximum of eight elements and the top pointer currently points at index 3.

Show the stack contents and indicate the position of the top pointer after each of the following stack operations:

- (a) Push (S, 5) (b) Push (S, 7) (c) Pop (S)
 (d) Pop (S) (e) Pop (S) (f) Push (S, -1)

Solution

Push (S, 5)

Step 1 $\text{Top} = \text{Top} + 1 = 3 + 1 = 4$

Step 2 $S[\text{Top}] = S[4] = 5$

Stack contents

Stack (S):	99	2	44	8	5			
Index:	0	1	2	3	4↑	5	6	7

Push (S, 7)

Step 1 $\text{Top} = \text{Top} + 1 = 4 + 1 = 5$

Step 2 $S[\text{Top}] = S[5] = 7$

Stack contents

Stack (S):	99	2	44	8	5	7		
Index:	0	1	2	3	4	5↑	6	7

Pop (S)

Step 1 $\text{Item} = S[\text{Top}] = S[5] = 7$

Step 2 $\text{Top} = \text{Top} - 1 = 5 - 1 = 4$

Stack contents

Stack (S)	99	2	44	8	5			
Index	0	1	2	3	4↑	5	6	7

Pop (S)

Step 1 $\text{Item} = S[\text{Top}] = S[4] = 5$

Step 2 $\text{Top} = \text{Top} - 1 = 4 - 1 = 3$

Stack contents

Stack (S)	99	2	44	8				
Index	0	1	2	3↑	4	5	6	7

Pop (S)

Step 1 $\text{Item} = S[\text{Top}] = S[3] = 8$

Step 2 $\text{Top} = \text{Top} - 1 = 3 - 1 = 2$

Stack contents

Stack (S)	99	2	44					
Index	0	1	2↑	3	4	5	6	7

Push (S, -1)

Step 1 Top = Top + 1 = 2 + 1 = 3

Step 2 S [Top] = S [3] = -1

Stack contents

Stack (S)	99	2	44	-1				
Index	0	1	2	3↑	4	5	6	7

Problem 13.2 Consider the following two states of a stack S:

State 1

Stack (S)	99	2	44	8	4			
Index	0	1	2	3	4↑	5	6	7

State 2

Stack (S)	99	5	-1	6				
Index	0	1	2	3↑	4	5	6	7

Write the series of push and pop operations that will transition the stack S from State 1 to State 2.

Solution

Step 1 Pop (S)

Step 2 Pop (S)

Step 3 Pop (S)

Step 4 Pop (S)

Step 5 Push (S, 5)

Step 6 Push (S, -1)

Step 7 Push (S, 6)

Problem 13.3 Consider the following stack S:

Stack (S)	99	2	44	8		
Index	0	1	2	3↑	4	5

What will be result of the following statements?

```

i = 0;
while (i != 5)
{
    push (S, i);
    i = i + 1;
}

```

Solution

Step 1 push (S, 0) → Top = 4, S [4] = 0

Step 2 push (S, 1) → Top = 5, S [5] = 1

Step 3 push (S, 2) → Top = 6 → Stack Overflow

Problem 13.4 Consider the following stack S:

Stack (S)	99	2	44	8		
Index	0	1	2	3↑	4	5

What will be result of the following statements?

```
i = 0;
while (i != 5)
{
    item = pop (S);
    i = i + 1;
}
```

Solution

Step 1 pop (S) item = 8, Top = 2

Step 2 pop (S) item = 44, Top = 1

Step 3 pop (S) item = 2, Top = 0

Step 4 pop (S) → item = 99, Top = NULL

Step 5 pop (S) → Stack Underflow

Problem 13.5 The linked implementation of stacks eliminates the limitation of array implementation that restricts the number of stack elements below the array upper bound. So, is it true to say that a stack overflow condition can never occur with linked implementation of stacks? Justify your answer.

Solution

No, it is not correct to say that a stack overflow condition can never occur with linked implementation of stacks. This is because; the system memory is also available till a certain extent. If we continue to push elements into a stack then a situation will arise when the system memory will run out of space causing the malloc function to return a NULL pointer. In this situation, the stack would be considered to be in an overflow state.

Problem 13.6 Identify and correct the logical error in the following statement that performs pop operation on a stack S.
item = S[--top];

Solution

The statement,

item = S[--top];

contains the prefix operator --, which decrements the value of top by one. The new value now pointed by top is then popped and allocated to the variable item. However, this is not the top value of the stack.

To pop out the top value of the stack we must use -- as the postfix operator, as shown below.

item = S[top--];

The above statement will first retrieve the top value of the stack and then decrement the top pointer by one.

SUMMARY

- ☐ A stack is a linear list in which elements are added and removed only from one end called top of the stack.
- ☐ Stacks are based on Last-In-First-Out or LIFO principle that means, the element added last into the list is the first one to be removed.
- ☐ Inserting an element into a stack is referred as push operation while removing an element from the stack is referred as pop operation.
- ☐ Stacks can be implemented through arrays or linked lists.

- ☐ The array implementation of stacks reserves a fixed amount of memory space in the form of an array for storing stack elements.
- ☐ The linked implementation of stacks uses dynamic memory management techniques for allocating the memory space for storing a new stack element at run time.
- ☐ Since linked implementation of stacks is based on dynamic memory allocation it is more efficient as compared to array-based implementation.
- ☐ The various application areas of stacks are expression evaluation, program control, recursion control, etc.

Key Terms

Stack | Stack top | LIFO | Push | Pop | Array implementation | Lined implementation

Review Questions

- 13.1 What is a stack? Explain with examples.
- 13.2 Briefly describe the LIFO principle.
- 13.3 What is a top pointer? Explain its significance.
- 13.4 What are the different application areas of stack data structure?
- 13.5 Give any four real-life examples that principally resemble the stack data structure.
- 13.6 Explain the logical representation of stacks in memory with the help of an example.
- 13.7 Explain push and pop operations with the help of examples.
- 13.8 Deduce the contents of an empty stack after the execution of the following operations in sequence:
 - Push (6)
 - Push (8)
 - Push (-1)
 - Pop ()
 - Push (7)
 - Pop ()
 - Pop ()
- 13.9 What will happen if we keep on pushing elements into a stack one after another?
- 13.10 What will happen if we continue to pop out elements from a stack one after another?
- 13.11 How are stacks implemented?
- 13.12 What is the advantage of linked implementation of stacks over array implementation?
- 13.13 What role does dynamic memory management techniques play in linked implementation of stacks?
- 13.14 Briefly explain the overflow and underflow conditions along with their remedies.
- 13.15 Can an overflow situation occur even with linked implementation of stacks that uses dynamic memory allocation techniques? Explain.

Programming Exercises

- 13.1** Write a function in C to perform the push operation on an array-based stack that can store a maximum of 50 elements. Make sure that the overflow condition is adequately handled.
- 13.2** Write a function in C to perform the pop operation on a linked implementation of stack. Make sure that the underflow condition is adequately handled.
- 13.3** A stack contains N elements in it with *TOP* pointing at the top of the stack. It is required to reverse the order of occurrence of the N elements and store them in the same stack. Write a C program to achieve the same.
- 13.4** Modify the C program solution of Question 13.3 to store the N elements in sorted fashion with the largest element stored at the *TOP*.
- 13.5** A linked list implemented stack containing unknown number of elements is given. You are required to count the number of elements present in the stack. Write a function *count ()* in C that uses the pop operation to count the number of elements in the stack but does not actually remove the elements from the stack.
- 13.6** The Tower of Hanoi problem comprises of three towers with discs initially stacked on to the first tower. The requirement is to replicate the initial stack of discs into another tower while adhering to the following conditions:
- (a) A larger disk can not be placed on a smaller disk
 - (b) Only one disc can be shifted at a time
- Write a C program to find a solution to the above problem using stacks.
- 13.7** An input text string comprises the following:
- (a) Letters
 - (b) Digits
 - (c) Special Characters
- Write a program in C that accepts a text string from the user and stores its individual characters in three different stacks, i.e., L (for storing letters), D (for storing digits) and SC (for storing special characters). The program should terminate as soon as a '~' symbol is encountered.

- 13.8** A stack is represented by the following structure declaration:

```
struct STACK
{
    int ELEMENT[100];
    int TOP;
};
```

Write the push () and pop () functions in C for the above stack.

14

QUEUES

CHAPTER OUTLINE

- 14.1 Introduction
- 14.2 Wueues—Basic Concept
 - 14.2.1 Logical Representation of Queues
- 14.3 Queue Operation
- 14.4 Queue Implementation
 - 14.4.1 Array Implementation of Queues
 - 14.4.2 Linked Implementation of Queues
- 14.5 Circular Queues
- 14.6 Priority Queues
- 14.7 Double-ended Queues
- 14.8 Arithmetic Expression Infix to Postfix Conversion
- 14.9 Polynomial Manipulation Representing Polynomials using Linked Lists
 - Summary*
 - Key Terms*





14.1 INTRODUCTION

In Chapter 13, we learnt how stacks are different from arrays and how they store the data in memory. In this chapter, we will learn about another linear data structure called queues. While stacks allow insertion and deletion of data only at one end, queues restrict the insertion and deletion of data at two distinct ends. Just like stacks, queues also hold great significance in the implementation of key system processes such as CPU scheduling, resource sharing, etc.



14.2 QUEUES—BASIC CONCEPT

Queue is a linear data structure in which items are inserted at one end called 'Rear' and deleted from the other end called 'Front'. Queues are based on the First-In-First-Out (FIFO) principle that means the data item that is inserted first in the queue is also the first one to be removed from the queue. We can relate queues to certain real-life objects and situations, as shown in Figs. 14.1 (a) and (b).

Queue of people

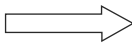


Fig. 14.1(a) Queue of people

A new object can not be added at the middle of an assembly line

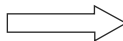


Fig. 14.1(b) An assembly line

As we can see in Fig. 14.1(a), a person can join a queue of waiting people only at its tail end while the person who joined the queue first becomes the first one to leave the queue. Likewise, the objects in an assembly line (Fig. 14.1(b)) also follow the same analogy. Another example of queue is the line up of vehicles at the toll booth. The vehicle that comes first to the toll booth leaves the booth first while

the vehicle that comes last leaves at the last; thus, observing FIFO principle. The concept of queue in data structures follows the same analogy as the queue of people or the queue of vehicles at toll booth.

An instance of queue implementation is a system of networked computers and resources where there are multiple users sharing one common printer amongst them. When a user on the network sends a print request, the request is added to the print queue. When the request reaches at the front it gets executed and is removed from the print queue. This ensures orderly execution of users' print requests. Figure 14.2 depicts this scenario.

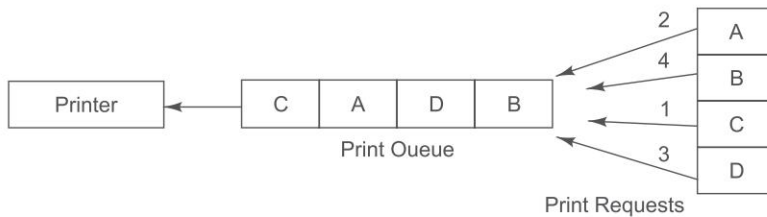


Fig. 14.2 Queue implementation

As we can see in the Fig. 14.2, four users, A, B, C and D share a single printer on the network. When a user sends a print request, it gets added to the print queue. User C sends the first print request, thus it gets added at the front of the print queue. Similarly, print requests from other users are also added in the queue as per their request order. Now, based on FIFO analogy, the printer will first process print request of user C, followed by A, D and user B at the last.



Mind Jog

What is the meaning of enqueue and dequeue?

All queue insertions are termed as enqueue while all queue deletions are termed as dequeue.

14.2.1 Logical Representation of Queues

Just like their real world counterparts, queues appear as a group of elements stored at contiguous locations in memory. Each successive insert operation adds an element at the rear end of the queue while each delete operation removes an element from the front end of the queue. The location of the front and rear ends are marked by two distinct pointers called *front* and *rear*.

Figure 14.3 shows the logical representation of queues in memory.

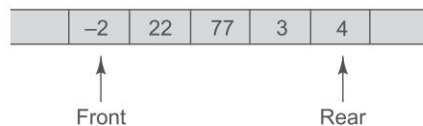


Fig. 14.3 Logical representation of queues

As we can see in the above figure, there are five elements in the queue with -2 at the front and 4 at the rear.



Note

The logical representation of queues showing queue elements stored at contiguous memory location might be true in case of their array implementation but the same might not be true in case of their linked implementation, as we shall study later in this chapter.



14.3 QUEUE OPERATIONS

There are two key operations associated with the queue data structure: insert and delete. The insert operation adds an element at the rear end of the queue while the delete operation removes an element from the front end of the queue. Figures 14.4 (a) and (b) depict the insert and delete operations on a queue.



Check Point

1. What is a queue?

Ans: Queue is a linear data structure in which items are inserted at one end called 'Rear' and deleted from the other end called 'Front'.

2. What is FIFO?

Ans: First-In-First-Out (FIFO) principle specifies that the data item that is inserted first in the queue is also the first one to be removed from the queue.

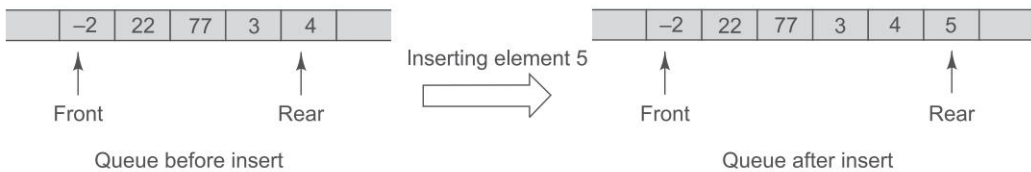


Fig. 14.4(a) Insert operation

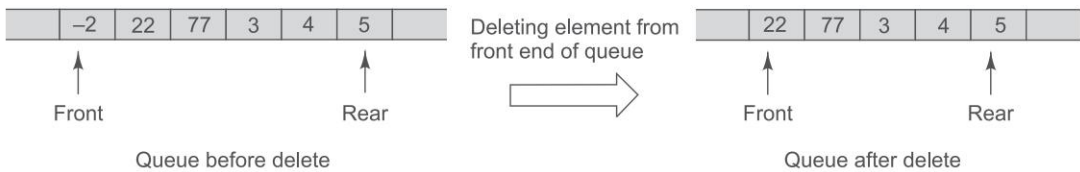


Fig. 14.4(b) Delete operation



Note The front and rear indicators are quite significant in queue's context as they point at entry and exit gateways of the queue.

1. Insert As we can see in Fig. 14.4(a), the insert operation involves the following subtasks:

- Receiving the element to be inserted.
- Incrementing the queue pointer, rear.
- Storing the received element at new location of rear.

Thus, the programmatic realization of the insert operation requires implementation of the above mentioned subtasks, as we shall see later in this chapter.



Tip Before inserting a new element, it needs to be checked whether the queue is already full. If the queue is already full then a new element cannot be added at its rear end. Such a situation is termed as queue overflow.

2. Delete As we can see in Fig. 14.4 (b), the delete operation involves the following subtasks:
Retrieving or removing the element from the front end of the queue.

Incrementing the queue pointer, *front*, to make it point to the next element in the queue.

Thus, the programmatic realization of the delete operation requires implementation of the above mentioned subtasks, as we shall see later in this chapter.

Some queue implementations require the deleted element to be returned back to the calling function while others may simply focus on updating the front pointer and ignore the deleted element all together. The choice of a particular type of implementation depends solely on the programming situation at hand.

An example of queue operations

Figure 14.5 shows how the queue contents change after a series of insert and delete operations.

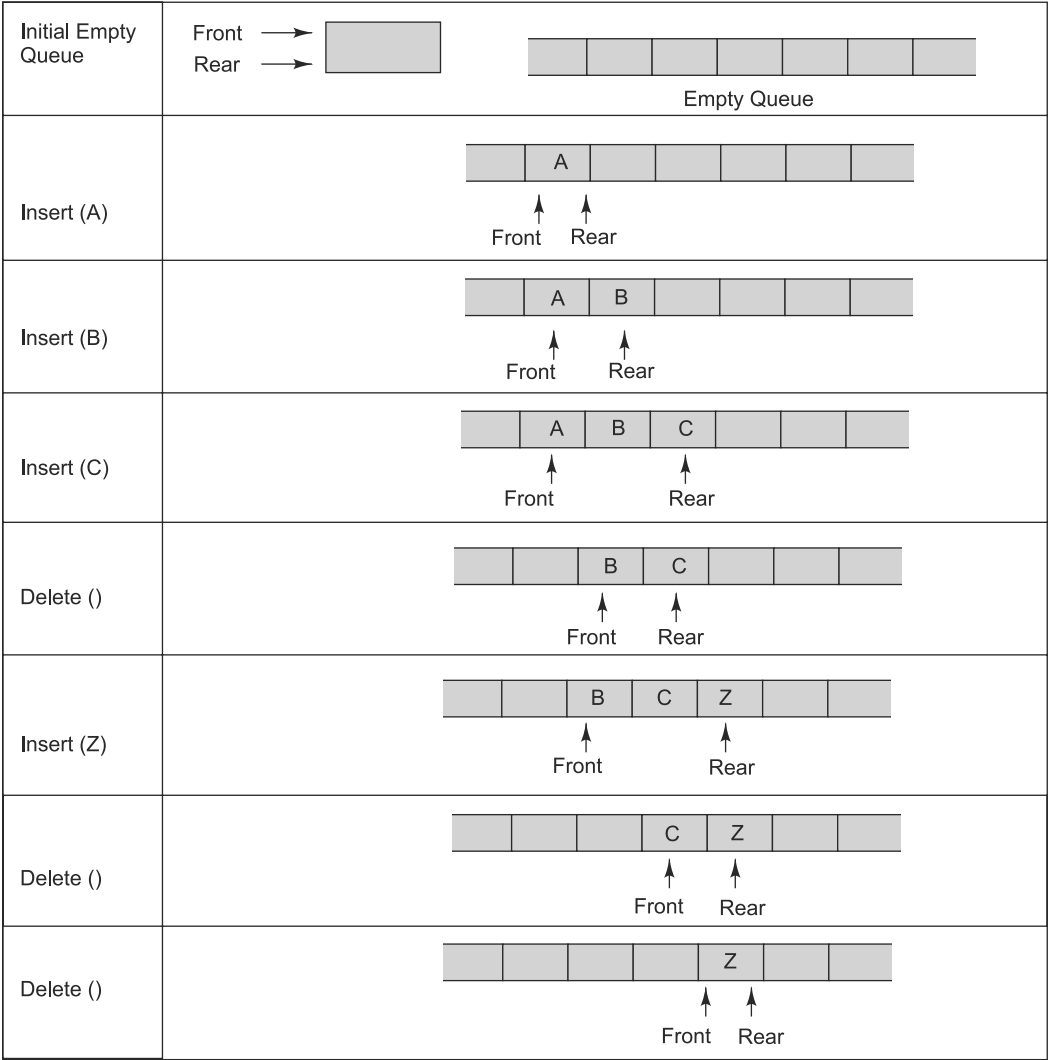


Fig. 14.5 Queue operations

**Tip**

Before deleting an element, it needs to be checked whether the queue is already empty. If the queue is already empty then there is nothing to be deleted. Such a situation is termed as queue underflow.

We can see in Fig. 14.5 how queue contents change with insert and delete operations occurring at rear and front ends respectively.



14.4 QUEUE IMPLEMENTATION

Queue implementation involves choosing the data storage mechanism for storing queue elements and implementing methods for performing the two queue operations, insert and delete. Like stacks, we can implement queues by using arrays or linked lists. The advantages or disadvantages of array or linked implementations of queues are the same that are associated with such types of data structures. However, both types of implementation have their own usage in specific situations.

**Mind Jog****What is a bounded queue?**

It is a queue restricted to a fixed number of elements.

14.4.1 Array Implementation of Queues

The array implementation of queues involves allocation of fixed size array in the memory. Both queue operations (insert and delete) are performed on this array with a constant check being made to ensure that the array does not go out of bounds.

Insert Operation The insert operation involves checking whether or not the queue pointer *rear* is pointing at the upper bound of the array. If it is not, *rear* is incremented by 1 and the new item is added at the end of the queue.

**Check Point****1. What is a queue insert operation?**

Ans. The queue insert operation adds an element at the rear end of the queue.

2. What is a queue delete operation?

Ans. The queue delete operation removes an element from the front end of the queue.

Example 14.1 Algorithm to Realize Insert Operation under Array Implementation of Queues

```
insert(queue[MAX], element, front, rear)
Step 1: Start
Step 2: If front = NULL goto Step 3 else goto Step 6
Step 3: front = rear = 0
Step 4: queue[front]=element
Step 5: Goto Step 10
Step 6: if rear = MAX-1 goto Step 7 else goto Step 8
Step 7: Display the message, "Queue is Full" and goto Step 10
Step 8: rear = rear +1
Step 9: queue[rear] = element
Step 10: Stop
```

 Tip

The above code shows insertion of an integer type element into the queue. However, we may store other built-in or user-defined type elements in the queue as per our own requirements.

Delete Operation The delete operation involves checking whether or not the queue pointer *front* is already pointing at NULL (empty queue). If it is not, the item that is being currently pointed is removed from the queue (array) and the *front* pointer is incremented by 1.

Example 14.2 Algorithm to Realize Delete Operation under Array Implementation of Queues

```
delete(queue[MAX], front, rear)
Step 1: Start
Step 2: If front = NULL and rear = NULL goto Step 3 else goto Step 4
Step 3: Display the message, "Queue is Empty" and goto Step 10
Step 4: if front != NULL and front = rear goto Step 5 else goto Step 8
Step 5: Set i = queue[front]
Step 6: Set front = rear = -1
Step 7: Return the deleted element i and goto Step 10
Step 8: Set i = queue[front]
Step 9: Return the deleted element i
Step 10: Stop
```

Implementation

Program 14.1 implements a queue using arrays in C. It uses the insert (Example 14.1) and delete (Example 14.2) functions for realizing the common queue operations.

Program 14.1 Implementation of Queue using Array

```
// Program for demonstrating implementation of queues using arrays

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

using namespace std;

int queue[100]; //Declaring a 100 element Queue array
int front = -1; // Declaring and initializing the Front Queue pointer
int rear = -1; // Declaring and initializing the Rear Queue pointer

void insert(int); // Declaring a function prototype for inserting an
element into the Queue
int del(); // Declaring a fuction prototype for removing an element from
the Queue
void display(); // Declaring a fuction prototype for displaying the elements
of a Queue
```

(Contd.)

```

int main()
{
    int choice;
    int num1 = 0, num2 = 0;

    while(1)
    {
        system("cls");

        // Creating an interactive interface for performing queue operation

        cout<< "\nSelect a choice from the following:\n";
        cout<< "[1] Add an Element into the Queue\n";
        cout<< "[2] Remove out an Element from the Queue\n";
        cout<< "[3] Display the Queue Elements\n";
        cout<< "[4] Exit \n\n";
        cout<< "\n\tYour Choice : ";
        cin>> choice;

        switch(choice)
        {
            case 1:
            {
                cout<< "\n\tEnter the Element to be added into the Queue : ";
                cin>> num1;
                insert(num1); //Adding an Element
                break;
            }

            case 2:
            {
                num2 = del(); // Removing an Element
                if(num2 == -9999)
                    cout<< "\n\tQueue is Empty!\n";
                else
                    cout<< "\n\t\tElement " << num2 << " is removed from
                    the Queue \n\t";
                getch();
                break;
            }

            case 3:
            {
                display(); // Displaying Queue elements
                getch();
                break;
            }

            case 4:

```

(Contd.)

```

        {
            exit(1);
            break;
        }

        default:
        {
            cout<< "\nInvalid Choice!!!\n";
            break;
        }
    }
}

//Insert Fuction
void insert(int element)
{
    if(front == -1) // Adding Element to an Empty Queue
    {
        front = rear = front +1;
        queue[front] = element;
        return;
    }

    if(front == 99) //Checking whether Queue is full
    {
        cout<< "Queue is Full!\n";
        getch();
        return;
    }
    rear = rear + 1; // Incrementing Rear Pointer
    queue[rear] = element; // Inserting the New Element
}

//Delete Fuction

int del()
{
    inti;
    if(front == -1 && rear == -1) // Checking whether the queue is Empty
    {
        cout<< "\n\tQueue is Empty!\n";
        getch();
        return(-9999);
    }

    if(front != -1 && rear == front) // Checking whether the queue has
    only one element
    {

```

(Contd.)

```

        i = queue[front];
        front = -1;
        rear = -1;
        return(i);
    }

    return(queue[front++]); //Returning the Frontmost Element and
    incrementing the Front Queue Pointer
}

//Display Function
void display()
{
    inti;
    if(front == -1)
    {
        cout<< "\n\tQueue is Empty!\n";
        // getch();
        return;
    }
    else
        cout<< "\n\tThe various queue Elements are : \n\n\t";

    for(i = front; i<= rear; i++)
        cout<< "\t" << queue[i]; //Printing Queue Elements
}

```

Output

Select a choice from the following:

- [1] Add an Element into the Queue
- [2] Remove out an Element from the Queue
- [3] Display the Queue Elements
- [4] Exit

Your Choice : 3

Queue is Empty!

Select a choice from the following:

- [1] Add an Element into the Queue
- [2] Remove out an Element from the Queue
- [3] Display the Queue Elements
- [4] Exit

Your Choice : 1

Enter the Element to be added into the Queue : 1

Select a choice from the following:

(Contd.)

- [1] Add an Element into the Queue
- [2] Remove out an Element from the Queue
- [3] Display the Queue Elements
- [4] Exit

Your Choice : 1

Enter the Element to be added into the Queue : 2

Select a choice from the following:

- [1] Add an Element into the Queue
- [2] Remove out an Element from the Queue
- [3] Display the Queue Elements
- [4] Exit

Your Choice : 1

Enter the Element to be added into the Queue : 3

Select a choice from the following:

- [1] Add an Element into the Queue
- [2] Remove out an Element from the Queue
- [3] Display the Queue Elements
- [4] Exit

Your Choice : 3

The various queue Elements are :

1 2 3

Select a choice from the following:

- [1] Add an Element into the Queue
- [2] Remove out an Element from the Queue
- [3] Display the Queue Elements
- [4] Exit

Your Choice : 2

Element 1 is removed from the Queue

Select a choice from the following:

- [1] Add an Element into the Queue
- [2] Remove out an Element from the Queue
- [3] Display the Queue Elements
- [4] Exit

Your Choice : 2

Element 2 is removed from the Queue

(Contd.)

```
Select a choice from the following:
[1] Add an Element into the Queue
[2] Remove out an Element from the Queue
[3] Display the Queue Elements
[4] Exit

Your Choice : 2

Element 3 is removed from the Queue

Select a choice from the following:
[1] Add an Element into the Queue
[2] Remove out an Element from the Queue
[3] Display the Queue Elements
[4] Exit

Your Choice : 2

Queue is Empty!

Select a choice from the following:
[1] Add an Element into the Queue
[2] Remove out an Element from the Queue
[3] Display the Queue Elements
[4] Exit

Your Choice : 4
```

14.4.2 Linked Implementation of Queues

The linked implementation of queues involves dynamically allocating memory space at run time while performing queue operations. Since, the allocation of memory space is dynamic, the queue consumes only that much amount of space as is required for holding its data elements. This is contrary to array-implemented queues which continue to occupy a fixed memory space even if there are no elements present. Thus, linked implementation of queues based on dynamic memory allocation technique prevents wastage of memory space.



Note *The linked implementation of queues is based on dynamic memory management techniques, which allow allocation and de-allocation of memory space at runtime.*

Insert Operation The insert operation under linked implementation of queues involves the following tasks:

1. Reserving memory space of the size of a queue element in memory
2. Storing the added (inserted) value at the new location
3. Linking the new element with existing queue
4. Updating the *rear* pointer

Example 14.4 Algorithm to Realize Insert Operation under Linked Implementation of Queues

```
insert(structure queue, value, front, rear)
```

Step 1: Start

Step 2: Set ptr=(struct queue*)malloc(sizeof(struct queue)), to reserve a block of memory for the new queue node and assign its address to pointer ptr

Step 3: Set ptr->element=value, to copy the inserted value into the new node

Step 4: if front = NULL goto Step 5 else goto Step 7

Step 5: Set front = rear = ptr

Step 6: Set ptr->next=NULL and goto Step 10

Step 7: Set rear->next=ptr

Step 8: Set ptr->next=NULL

Step 9: Set rear = ptr

Step 10: Stop

Delete Operation The delete operation under linked implementation of queues involves the following tasks:

1. Checking whether the queue is empty.
2. Retrieving the front most element of the queue.
3. Updating the *front* pointer.
4. Returning the retrieved (removed) value

Example 14.5 Algorithm to Realize Delete Operation under Linked Implementation of Queues

```
delete(structure queue, front, rear)
```

Step 1: Start

Step 2: if front = NULL goto Step 3 else goto Step 4

Step 3: Display message, "Queue is Empty" and goto Step 7

Step 4: Set i = front->element

Step 5: Set front = front->next

Step 6: Return the deleted element i

Step 7: Stop

**Tip**

It is a good programming practice to release unused memory space so as to ensure efficient memory space utilization.

Implementation

Program 14.2 implements a queue using linked lists in C. It uses the insert (Example 14.4) and delete (Example 14.5) functions for realizing the common queue operations.

Program 14.2 Implementation of Queue using Linked List

```
// Program for demonstrating implementation of queues using arrays
#include<iostream>
#include<stdlib.h>
```

(Contd.)

```

#include<conio.h>
using namespace std;

struct queue // Declaring the Structure for Queue Elements
{
    int element;
    struct queue *next; // Queue Element pointing to another Queue Element
};

struct queue *front = NULL;
struct queue *rear = NULL;

void insert(int); // Declaring a function prototype for inserting an
element into the Queue
int del(); // Declaring a fuction prototype for removing an element from
the Queue
void display(); // Declaring a fuction prototype for displaying the elements
of a Queue

int main()
{
    int choice;
    int num1 = 0, num2 = 0;

    while(1)
    {
        system("cls");

        // Creating an interactive interface for performing queue operation

        cout<< "\nSelect a choice from the following:\n";
        cout<< "[1] Add an Element into the Queue\n";
        cout<< "[2] Remove out an Element from the Queue\n";
        cout<< "[3] Display the Queue Elements\n";
        cout<< "[4] Exit \n\n";
        cout<< "\n\tYour Choice : ";
        cin>> choice;

        switch(choice)
        {
            case 1:
            {
                cout<< "\n\tEnter the Element to be added into the Queue : ";
                cin>> num1;
                insert(num1); //Adding an Element
                break;
            }

            case 2:
            {
                num2 = del(); // Removing an Element
            }
        }
    }
}

```

(Contd.)

```

        if(num2 == -9999)
            cout<< "\n\tQueue is Empty!\n";
        else
            cout<< "\n\t\tElement " << num2 << " is removed from
            the Queue \n\t";
        getch();
        break;
    }

    case 3:
    {
        display(); // Displaying Queue elements
        getch();
        break;
    }

    case 4:
    {
        exit(1);
        break;
    }

    default:
    {
        cout<< "\nInvalid Choice!!!\n";
        break;
    }
}

}

//Insert Fuction
void insert(int value)
{
    struct queue *ptr = (struct queue*)malloc(sizeof(struct queue)); //
    Dynamically allocating memory space to store Queue Elements

    ptr -> element = value; //Assigning value to the newly allocated Queue
    Element

    if(front == NULL)
    {
        front = rear = ptr;
        ptr -> next = NULL;
    }

    else //Updating Stack Pointers
    {
        rear -> next = ptr;
        ptr -> next = NULL;
    }
}

```

(Contd.)

```

        rear = ptr;
    }
}

//Delete Fuction

int del()
{
    inti;
    if(front == NULL) // Checking whether the queue is Empty
    {
        return(-9999);
    }

    else
    {
        i = front -> element; //Removing Element from the Start
        front = front -> next;
        return(i);
    }
}

//Display Function

void display()
{
    inti;
    struct queue *ptr = front;

    if(front == NULL) // Checking whether the queue is Empty
    {
        return;
    }
    else
        cout<< "\n\tThe various queue Elements are : \n\n\t";

    //Printing Queue Elements

    while(ptr != rear)
    {
        cout<< "\t" <<ptr -> element;
        ptr = ptr -> next;
    }

    cout<< "\t" << rear -> element;
}

```

Output

The Output of this program is same as that of Program 14.2



14.5 CIRCULAR QUEUES

A circular queue is a queue whose start and end locations are logically connected with each other. That means, the start location comes after the end location. If we continue to add elements in a circular queue till its end location, then after the end location has been filled, the next element will be added at the beginning of the queue. Circular queues remove one of the main disadvantages of array implemented queues in which a lot of memory space is wasted due to inefficient utilization.

Figure 14.6 shows the logical representation of a circular queue.

As we can see in Fig. 14.6, the start location of the queue comes after its end location. Thus, if the queue is filled till its capacity, i.e., the end location, then the start location will be checked for space, and if it is empty, the new element will be added there. Figure 14.7 shows the different states of a circular queue during insert and delete operations.



Check Point

1. What is array implementation of queues?

Ans. It involves allocation of fixed size array in the memory for storing queue elements. Both insert and delete operations are performed on this array.

2. What is linked implementation of queues?

Ans. It involves dynamic allocation of memory space at run time while performing queue operations.

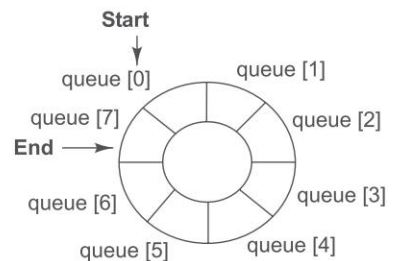


Fig. 14.6 Circular queue

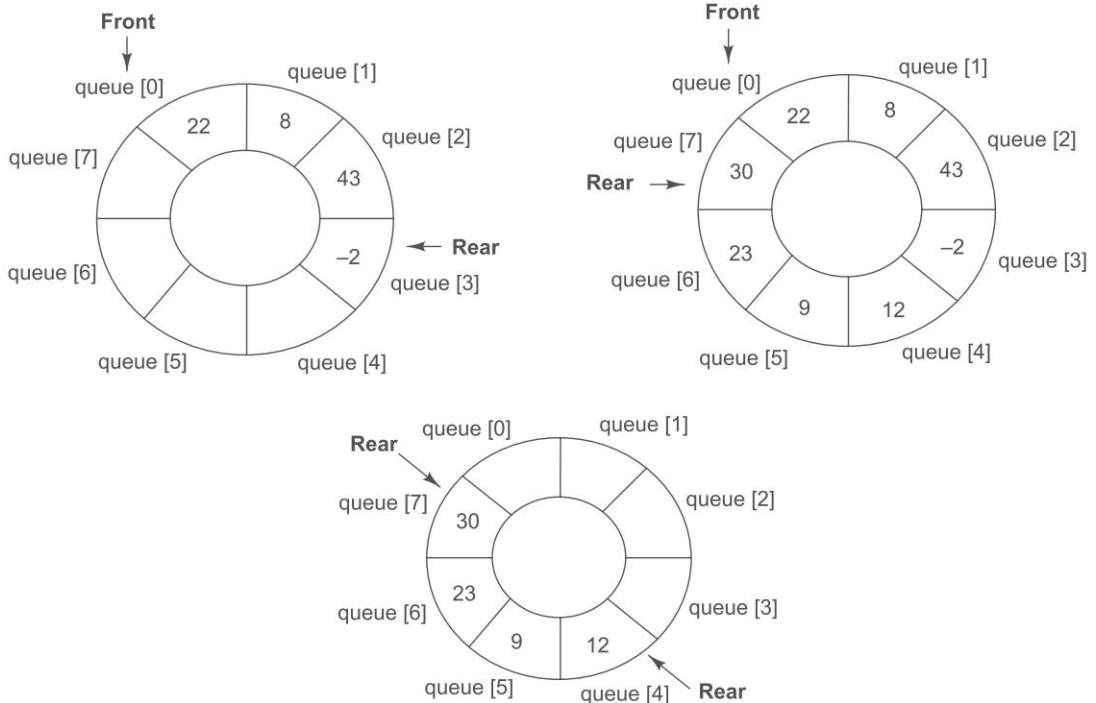


Fig. 14.7 Inserting and deleting elements in a circular queue

Insert Operation The insert operation for array implemented circular queues involves the following tasks:

1. Checking whether the queue is already full.
2. Updating the *rear* pointer.
 - (a) If the queue is empty, set *front* and *rear* to point to the first location in the queue.
 - (b) If *rear* is pointing at the last location of the queue, set *rear* to point to the first location in the queue.
 - (c) If none of the above situations exist, simply increment the *rear* pointer by 1.
3. Inserting the new element at the *rear* location.



14.6 PRIORITY QUEUES

Priority queue is a type of queue in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities. The order of processing or deletion of elements in a priority queue is decided by the following rules:

1. An element with highest priority is deleted before all other elements of lower priority.
2. If two elements have the same priority then they are deleted as per the order in which they were added into the queue (i.e., First-In-First-Out).

The implementation of priority queues may follow different approaches. For instance, elements may be added arbitrarily into the queue and deleted as per their priority values or, the elements may be sorted as per their priorities at the time of their insertion itself, and deleted in a sequential fashion. We'll be following the later approach for implementing priority queues.

The structure of a priority queue needs to be defined in such a manner that each queue node is able to store both its value as well as its priority information. The following C structure defines the node of a priority queue:

```
struct queue /*Node of a priority queue*/
{
    int element;
    int priority;
    struct queue *next; /*Pointer to the next queue node*/
};
```



14.7 DOUBLE-ENDED QUEUES

A double-ended queue is a special type of queue that allows insertion and deletion of elements at both ends, i.e., front and rear. In simple words, a double-ended queue can be referred as a linear list of elements in which insertion and deletion of elements takes place at its two ends but not in the middle. This is the reason why it is termed as double-ended queue or deque.



Check Point

1. What is a circular queue?

Ans. A circular queue is a queue whose start and end locations are logically connected with each other.

2. What is the advantage of circular queue?

Ans. The implementation of circular queues ensures efficient utilization of memory space in comparison to normal queues.



Check Point

1. What is a priority queue?

Ans. It is a type of queue in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities.

2. What is order of deletion if two or more elements in a priority queue have same priorities?

Ans. If two or more elements have the same priority then they are deleted as per the order in which they were added into the queue (i.e. First-In-First-Out).

Based on the type of restrictions imposed on insertion and deletion of elements, a double-ended queue is categorized into two types:

- 1. Input-restricted deque** It allows deletion from both the ends but restricts the insertion at only one end.
- 2. Output-restricted deque** It allows insertion at both the ends but restricts the deletion at only one end.

Figure 14.8 shows the logical representation of a deque.

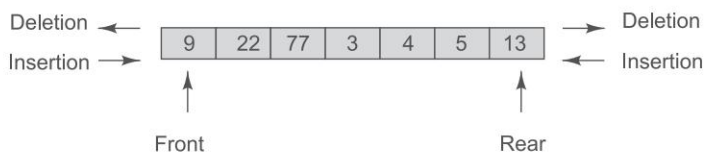


Fig. 14.8 Double-ended queue

As shown in Fig. 14.8, insertion and deletion of elements is possible at both front and rear ends of the queue. As a result, the following four operations are possible for a double-ended queue:

- 1. i_front** Insertion at front end of the queue.
- 2. d_front** Deletion from front end of the queue.
- 3. i_rear** Insertion at rear end of the queue.
- 4. d_rear** Deletion from rear end of the queue.



Mind Jog

In which situation is a deque used?

A deque is used for implementing A-Steal job scheduling algorithm. This algorithm helps perform task scheduling for multiple processors.



Check Points

1. What is a double-ended queue?

Ans: A double-ended queue is a special type of queue that allows insertion and deletion of elements at both ends, i.e., front and rear.

2. What are the different types of double-ended queues?

Ans: The two types of double-ended queues are input-restricted deque (insertion at one end, deletion at both ends) and output-restricted deque (insertion at both ends, deletion at one end).



14.8 POLYNOMIAL MANIPULATION REPRESENTING POLYNOMIALS USING LINKED LISTS

Linked lists can be used for representing polynomial expressions.

A linked list can be used for representing a polynomial expression by storing polynomial terms in the form of interconnected list nodes. Each node contains the following:

1. Variable
2. Exponent
3. Coefficient

Addition of two polynomials is carried out by adding the nodes having equivalent exponents and storing the result in a separate linked list.

The following structure declaration represents a polynomial term:

```
struct p_term
{
    int coeff;
    int pow;

    struct p_term* next;
}
```



14.9 ARITHMETIC EXPRESSION INFIX TO POSTFIX CONVERSION

Program 14.3 in C that uses the stack data structure to convert an infix expression into postfix expression. The program also evaluates the value of the resultant postfix expression.

Program 14.3 Evaluating Arithmetic Expression using Stack

```
// Program for demonstrating solving Arithmetic Expression using Stacks

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

using namespace std;

int stack[100]; //Declaring a 100 element stack array
int top = -1; // Declaring and initializing the stack pointer

void in2post(char[]); // Function Prototype for converting infix to postfix
expression
void push(int); // Function Prototype for pushing an element into the Stack
int pop(); // Function Prototype for removing an Element from a Stack
intprec(char); // Function Prototype for determining precedence of Operators.
```

(Contd.)

```
intcal(char []); // Function Prototype for Calculating the value of a
Postfix Expression
```

```
int main()
{
    char in[100], post[100];

    system("cls");

    cout<< "\nEnter the infix Expression : ";
    cin>> in;
    in2post(in);
    getch();
}

void in2post(char in_exp[])
{
    int x, y, z, result;
    char a, c, post_exp[100];
    char t;

    x = y = result = 0;
    push('\0');
    t = in_exp[x];
    while(t != '\0')
    {
        if(isalnum(t))
        {
            post_exp[y] = t;
            y++;
        }
        else if(t == '(')
            push('(');
        else if(t == ')')
        {
            while(stack[top] != '(')
            {
                c = pop();
                post_exp[y] = c;
                y++;
            }
            c = pop();
        }
        else
        {
            while(prec(stack[top]) >= prec(t))
            {
                c=pop();
                post_exp[y] = c;
                y++;
            }
        }
    }
}
```

(Contd.)

```

        }
        push(t);
    }
    x++;
    t = in_exp[x];
}
while(top != -1)
{
    c = pop();
    post_exp[y] = c;
    y++;
}

cout<< "\nThe Equivalent Postfix Expression is : ";

for(z=0; z<y; z++)
    cout<<post_exp[z];

cout<< "\n\nDo you want to evaluate the Postfix Expression? (Y/N) : ";
cin>> a;

if(a == 'y' || a == 'Y')
{
    result = cal(post_exp);
    cout<< "\nResult = " << result << "\n";
    getch();
}

else if (a == 'n' || a == 'N')
    exit(0);
}

intcal(char post[])
{
    int m, n, x, y, len, j;
    j = 0;
    len = strlen(post);

    while(j <len)
    {
        if(isdigit(post[j]))
        {
            x = post[j] - '0';
            push(x);
        }

        else
        {
            m = pop();
            n = pop();

```

(Contd.)

```

        switch(post[j])
        {
            case '+' :
                x = n + m;
                break;
            case '-' :
                x = n - m;
                break;
            case '*' :
                x = n * m;
                break;
            case '/' :
                x = n / m;
                break;
        }
        push(x);
    }
    j++;
}
if(top > 0)
{
    cout<< "\nDiscrepancy between number of Operators and Operands!!!\n";
    exit(0);
}

else
{
    y = pop();
    return(y);
}
return 0;
}

intprec(char t)
{
    switch(t)
    {
        case '(':
            return(10);
        case ')':
            return(9);
        case '+':
            return(7);
        case '-':
            return(7);
        case '*':
            return(8);
        case '/':
            return(8);
    }
}

```

(Contd.)

```

        case '\0':
            return(0);
        default:
            cout<< "\nInvalid Expression\n";
            break;
    }
    return 0;
}

void push(int n)
{
    if(top == 99)
    {
        cout<< "\n\nStack is Full!!!\n";
        getch();
        exit(1);
    }
    stack[++top] = n;
}

int pop()
{
    if(top == -1)
        getch();
    return(stack[top--]);
}

```

Output

Enter the infix Expression : 1+2*3

The Equivalent Postfix Expression is : 123*+

Do you want to evaluate the Postfix Expression? (Y/N) : y

Result = 7

SOLVED PROBLEMS

Problem 14.1 The contents of a queue Q are as follows:

Queue (Q)	4	5	—9	66				
Index	0	1	2	3	4	5	6	7

F?

R?

The queue can store a maximum of eight elements and the front (F) and rear (R) pointers currently point at index 0 and 3 respectively.

Show the queue contents and indicate the position of the front and rear pointers after each of the following queue operations:

- (a) Insert (Q, 16), (b) Delete (Q), (c) Delete (Q), (d) Insert (Q, 7), (e) Delete (Q), (f) Insert(Q,—2)

Solution

- (a) Insert (Q, 16)

Step 1 $R = R + 1 = 3 + 1 = 4$ Step 2 $Q[R] = Q[4] = 16$ **Queue contents**

Queue (Q)	4	5	-9	66	16			
Index	0	1	2	3	4	5	6	7

 $F \uparrow$ $R \uparrow$

- (b) Delete (Q)

Step 1 Item = $Q[F] = Q[0] = 4$ Step 2 $F = F + 1 = 0 + 1 = 1$ **Queue contents**

Queue (Q)		5	-9	66	16			
Index	0	1	2	3	4	5	6	7

 $F \uparrow$ $R \uparrow$

- (c) Delete (Q)

Step 1 Item = $Q[F] = Q[1] = 5$ Step 2 $F = F + 1 = 1 + 1 = 2$ **Queue contents**

Queue (Q)			-9	66	16			
Index	0	1	2	3	4	5	6	7

 $F \uparrow$ $R \uparrow$

- (d) Insert (Q, 7)

Step 1 $R = R + 1 = 4 + 1 = 5$ Step 2 $Q[R] = Q[5] = 7$ **Queue contents**

Queue (Q)			-9	66	16	7		
Index	0	1	2	3	4	5	6	7

 $F \uparrow$ $R \uparrow$

- (e) Delete (Q)

Step 1 Item = $Q[F] = Q[2] = -9$ Step 2 $F = F + 1 = 2 + 1 = 3$ **Queue contents**

Queue (Q)				66	16	7		
Index	0	1	2	3	4	5	6	7

 $F \uparrow$ $R \uparrow$

- (f) Insert (Q, -2)

Step 1 $R = R + 1 = 5 + 1 = 6$

Step 2 $Q[R] = Q[6] = -2$

Queue contents

Queue (Q)			-9	66	16	7	-2	
Index	0	1	2	3	4	5	6	7

$F \uparrow$ $R \uparrow$

Problem 14.2 Consider the following two states of a queue Q:

State 1

Queue (Q)		4	-1	8				
Index	0	1	2	3	4	5	6	7

$F \uparrow$ $R \uparrow$

State 2

Queue (Q)				3	11	22	33	
Index	0	1	2	3	4	5	6	7

$F \uparrow$ $R \uparrow$

Write the series of insert and delete operations that will transition the queue Q from State 1 to State 2.

Solution

Step 1 Delete (Q)

Step 2 Delete (Q)

Step 3 Insert (Q, 11)

Step 4 Insert (Q, 22)

Step 5 Insert (Q, 33)

Problem 14.3 Is there any limitation associated with array implemented queues?

Solution One of the key limitations of array implemented queue is that it may lead to an overflow condition even when a number of its preceding locations are empty. Such a situation can be easily avoided by implementing the queue in a circular fashion, which logically connects its front and rear ends.

Problem 14.4 Identify the error in the following structure declaration of a priority queue node:

```
struct queue /*Node of a priority queue*/
{
    int element;
    int priority;
    }*next;
```

Solution In the linked implementation of a queue, the next pointer should be associated with each node of the queue. Hence, next should be declared inside the structure declaration and not outside, as shown below:

```
struct queue /*Node of a priority queue*/
{
    int element;
    int priority;
    *next; /*Pointer to the next queue node*/
}
```


SUMMARY

- ☐ Queue is a linear data structure in which items are inserted at one end called 'Rear' and deleted from the other end called 'Front'.
- ☐ Queues are based on the First-In-First-Out (FIFO) principle that means the data item that is inserted first in the queue is also the first one to be removed from the queue.
- ☐ There are two key operations associated with the queue data structure: insert and delete.
- ☐ Queues can be implemented through arrays or linked lists.
- ☐ The array implementation of queues reserves a fixed amount of memory space in the form of an array for storing queue elements.
- ☐ The linked implementation of queues uses dynamic memory management techniques for allocating the memory space for storing a new queue element at run time.
- ☐ Since linked implementation of queues is based on dynamic memory allocation it is more efficient as compared to array-based implementation.
- ☐ A circular queue is one whose start and end locations are logically connected with each other.
- ☐ Circular queues remove one of the main disadvantages of array implemented queues in which a lot of memory space is wasted due to inefficient utilization.
- ☐ Priority queue is a type of queue in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities.
- ☐ A double-ended queue is a special type of queue that allows insertion and deletion of elements at both ends, i.e., front and rear.
- ☐ A double-ended queue can also be referred as a linear list of elements in which insertion and deletion of elements takes place at its two ends but not in the middle.
- ☐ A double-ended queue is categorized into two types: input-restricted deque and output-restricted deque.

Key Terms

Queue | Front | Rear | FIFO | Insert | Delete | Array implementation | Lined implementation | Circular queue | Priority queue | Double-ended queue

Review Questions

- 14.1 What is a queue? Explain with examples.
- 14.2 Briefly describe the FIFO principle.
- 14.3 What are front and rear pointers? Explain their significance.
- 14.4 What are the different application areas of queue data structure?
- 14.5 Give any three real-life examples that principally resemble the queue data structure.
- 14.6 Explain the logical representation of queue in memory with the help of an example.

- 14.7** Explain insert and delete queue operations with the help of examples.
- 14.8** Deduce the contents of an empty queue after the execution of the following operations in sequence:
- Insert (9)
 - Insert (-7)
 - Delete ()
 - Insert (4)
 - Delete ()
 - Insert (18)
 - Delete ()
- 14.9** What is a priority queue? How is it different from a normal queue?
- 14.10** Explain the significance of double-ended queue.
- 14.11** How are queues implemented?
- 14.12** What is the advantage of linked implementation of queues over array implementation?
- 14.13** What is the objective of implementing a queue in circular fashion?
- 14.14** List the differences between stack and queue data structures.
- 14.15** How is a double-ended queue implemented?

Programming Exercises

- 14.1** Write a C function to print the elements of a queue implemented using linked list.
- 14.2** Write a C function to perform the delete operation on an array-implemented circular queue.
- 14.3** Write a C function to insert an element into a priority queue as per priority.
- 14.4** Write a C function to perform the delete operation at the end of a double-ended queue.
- 14.5** Write a C function to remove elements from a queue and store them in a stack. Also, display the contents of the resultant stack.

UNIT 4

Non-linear Data Structures

- **Chapter 15:** Trees
- **Chapter 16:** Graphs

15

TREES

CHAPTER OUTLINE

- 15.1 Introduction
- 15.2 Basic concept
 - 15.2.1 Tree Terminology
- 15.3 Binary Tree
 - 15.3.1 Binary Tree Concepts
- 15.4 Binary Tree Representation
 - 15.4.1 Array Representation
 - 15.4.2 Linked Representation
- 15.5 Binary Tree traversal
- 15.6 Binary search Tree
- 15.7 Tree variants
 - 15.7.1 Expression Trees
 - 15.7.2 Threaded Binary Trees
 - 15.7.3 Balanced Trees
 - 15.7.4 Splay Trees
 - 15.7.5 m -way Trees

Summary

Key Terms





15.1 INTRODUCTION

Till now, we focussed only on linear data structures such as stacks, queues and linked lists. But, in real-world situations, data relationships are not always linear. Tree is one such non-linear data structure which stores the data elements in a hierarchical manner. Each node of the tree stores a data value, and is linked to other nodes in a hierarchical fashion.

In this chapter, we will learn about the different types of trees and their related operations. Most importantly, we will focus on binary tree and its variants, which are widely used in the field of computer science.



15.2 BASIC CONCEPT

A tree is defined as a finite set of elements or nodes, such that

1. One of the nodes present at the top of the tree is marked as root node.
2. The remaining elements are partitioned across multiple subtrees present below the root node.

Figure 15.1 shows a sample tree T.

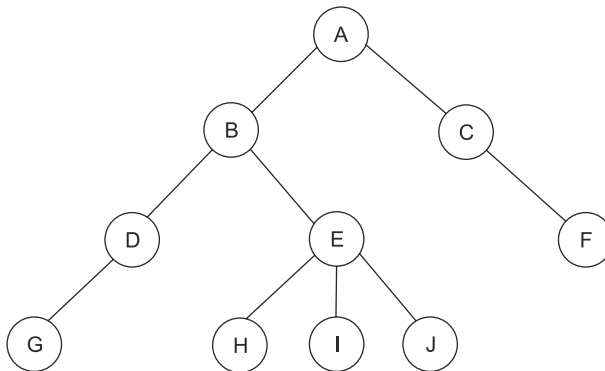


Fig. 15.1 Tree T

Here, T is a simple tree containing ten nodes with A being the root node. The node A contains two subtrees. The left subtree starts at node B while the right subtree starts at node C. Both the subtrees further contain subtrees below them, thus indicating recursive nature of the tree data structure. Each node in the tree has zero or more child nodes.

15.2.1 Tree Terminology

There are a number of key terms associated with trees. Table 15.1 lists some of the important key terms.

Table 15.1 Tree terminology

Key Term	Description	Example (Refer to Fig. 15.1)
Node	It is the data element of a tree. Apart from storing a value, it also specifies links to the other nodes.	A, B, C, D

Key Term	Description	Example (Refer to Fig. 15.1)
Root	It is the top node in a tree.	A
Parent	A node that has one or more child nodes present below it is referred as parent node.	B is the parent node of D and E
Child	All nodes in a tree except the root node are child nodes of their immediate predecessor nodes.	H, I and J are child nodes of E
Leaf	It is the terminal node that does not have any child nodes.	G, H, I, J and F are leaf nodes
Internal node	All nodes except root and leaf nodes are referred as internal nodes.	B, C, D and E are internal nodes
Sibling	All the child nodes of a parent node are referred as siblings.	D and E are siblings
Degree	The degree of a node is the number of subtrees coming out of the node.	Degree of A is 2 Degree of E is 3
Level	All the tree nodes are present at different levels. Root node is at level 0, its child nodes are at level 1, and so on.	A is at level 0 B and C are at level 1 G, H, I, J are at level 3
Depth or Height	It is the maximum level of a node in the tree.	Depth of tree T is 3
Path	It is the sequence of nodes from source node till destination node.	A–B–E–J



15.3 BINARY TREE

Binary tree is one of the most widely used non-linear data structures in the field of computer science. It is a restricted form of a general tree. The restriction that it applies to a general tree is that its nodes can have a maximum degree of 2. That means, the nodes of a binary tree can have zero, one or two child nodes but not more than that. Figure 15.2 shows a binary tree.

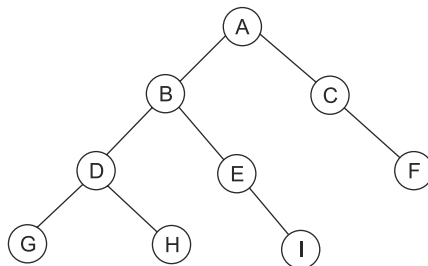


Fig. 15.2 Binary tree

As shown in the above binary tree, all nodes have a maximum degree of 2. The maximum number of nodes that can be present at level n is 2^n .

15.3.1 Binary Tree Concepts

Before we learn how binary trees are represented in memory, let us discuss some of the key concepts associated with binary trees. Table 15.2 lists these key concepts.

Table 15.2 *Binary tree concepts*

Concept	Description	Example
Strictly binary tree	A binary tree is called strictly binary if all its nodes barring the leaf nodes contain two child nodes.	
Complete binary tree	A binary tree of depth d is called complete binary tree if all its levels from 0 to $d - 1$ contain maximum possible number of nodes and all the leaf nodes present at level d are placed towards the left side.	
Perfect binary tree	A binary tree is called perfect binary tree if all its leaf nodes are at the lowest level and all the non-leaf nodes contain two child nodes.	
Balanced binary tree	A binary tree is called balanced binary tree if the depths of the subtrees of all its nodes do not differ by more than 1.	



15.4 BINARY TREE REPRESENTATION

The sequential representation of binary trees is done by using arrays while the linked representation is done by using linked lists.

15.4.1 Array Representation

In the array representation of binary trees, one-dimensional array is used for storing the node elements. The following rules are applied while storing the node elements in the array:

1. The root node is stored at the first position in the array while its left and right child nodes are stored at the successive positions.
2. If a node is stored at index location i then its left child node will be stored at location $2i + 1$ while the right child node will be stored at location $2i + 2$.

Let us consider a binary tree T_1 , as shown in Fig. 15.3.

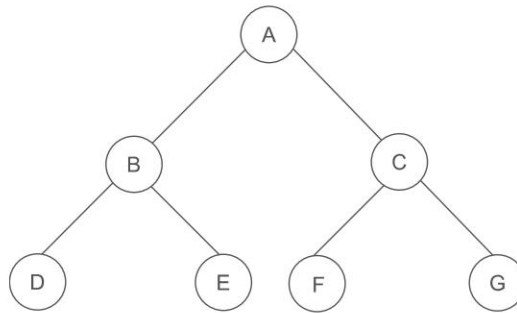


Fig. 15.3 Binary tree T_1

Here, T_1 is a binary tree containing seven nodes with A being the root node. B and C are the left and right child nodes of A respectively. Let us apply the rules explained earlier to arrive at the array representation of binary tree T_1 . Figure 15.4 shows the array representation.

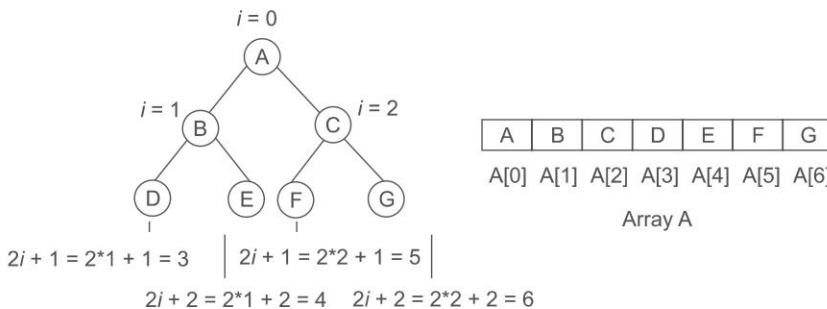


Fig. 15.4 Array representation of binary tree T_1



Check Point

1. What is a leaf?

Ans. It is the terminal node in a tree that does not have any child nodes.

2. What is a balanced binary tree?

Ans. A binary tree is called balanced binary tree if the depths of the subtrees of all its nodes do not differ by more than 1.

Figure 15.4 shows the array index values for each of the tree nodes. Array A is used for storing the node values.

Now, let us modify the binary tree T_1 a little by deleting nodes E and F. The revised array representation of T_1 is shown in Fig. 15.5.

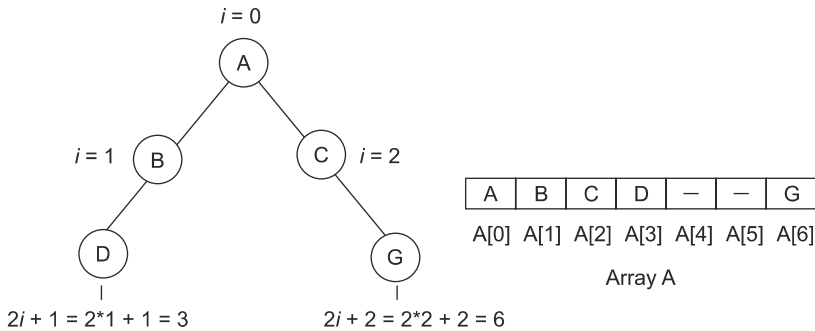


Fig. 15.5 Revised array representation of binary tree T_1

As we can see in Fig. 15.5, even after removing two elements from the tree, it still requires the same number of memory locations for storing the node elements. This is the main disadvantage of array representation of binary trees. It efficiently utilises the memory space only when the tree is a complete binary tree. Otherwise, there are always some memory locations lying vacant in the array.

15.4.2 Linked Representation

To avoid the disadvantages associated with array representation, linked representation is used for implementing binary trees. It uses a linked list for storing the node elements. Each tree node is represented with the help of the linked list node comprising of the following fields:

1. **INFO** Stores the value of the tree node.
2. **LEFT** Stores a pointer to the left child.
3. **RIGHT** Stores a pointer to the right child.

In addition, there is a special pointer that points at the root node. Figure 15.6 shows how linked list is used for representing a binary tree in memory.

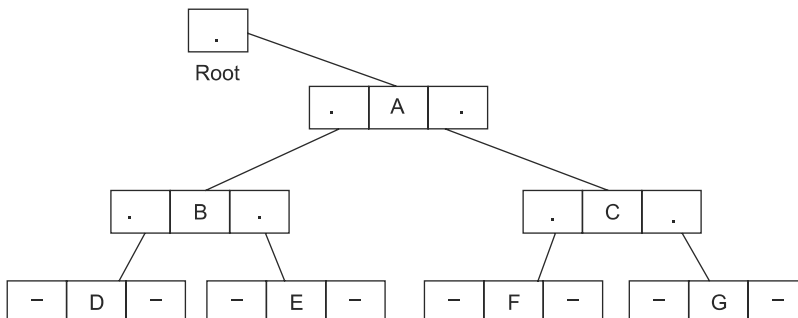


Fig. 15.6 Linked representation of binary tree

The linked representation of binary tree uses dynamic memory allocation technique for adding new nodes to the tree. It reserves only that much amount of memory space as is required for storing its node values. Thus, linked representation is more efficient as compared to array representation.

Program 15.1 Binary Tree Implementation

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

using namespace std;

struct node
{
    int INFO;
    struct node *LEFT, *RIGHT;
};

node *insert(node *, int); // Function prototype for inserting a new node
void display (node*); // Function prototype for displaying the tree nodes
int count = 1; // Counter for ascertaining left to right position for the new node

int main()
{
    node *root = NULL;
    int element, choice;

    system("cls");

    // Displaying a menu of choices

    while(1)
    {
        system("cls");

        cout << "\nSelect an Option\n";
        cout << "\n1 - Insert";
        cout << "\n2 - Display";
        cout << "\n3 - Exit";

        cout << "\n\nEnter your Choice : ";
        cin >> choice;

        switch(choice)
        {
            case 1:
            {
                cout << "\nEnter the node value : ";
                cin >> element;
```

(Contd.)

```

        root = insert(root, element); // Calling the insert function
for inserting a new Element into the Tree
        break;
    }

    case 2:
    {
        display(root); // Calling the display function for printing
the node values
        getch();
        break;
    }

    case 3:
    {
        exit(1);
        break;
    }

    default :
    {
        cout << "\n\n Incorrect Choice!!! Please Try Again...";
        getch();
        break;
    }
}

}

}

node *insert(node *r, int n)
{
    if(r == NULL)
    {
        r = (node*) malloc (sizeof(node));
        r -> LEFT = r -> RIGHT = NULL;
        r -> INFO = n;
        count = count + 1;
    }

    else
    {
        if(count % 2 == 0)
            r -> LEFT = insert(r -> LEFT,n);
        else
            r -> RIGHT = insert(r -> RIGHT,n);
    }
    return(r);
}

void display(node *r)

```

(Contd.)

```
{
    if(r -> LEFT != NULL)
        display(r -> LEFT);
    cout << r -> INFO << endl;
    if(r -> RIGHT != NULL)
        display(r -> RIGHT);
}
```

Output

Select an Option

1 - Insert
2 - Display
3 - Exit

Enter your Choice : 1

Enter the node value : 1

Enter your Choice : 1

Enter the node value : 2

Enter your Choice : 1

Enter the node value : 3

Enter your Choice : 1

Enter the node value : 4

Enter your Choice : 1

Enter the node value : 5

Enter your Choice : 1

Enter the node value : 6

Select an Option

1 - Insert
2 - Display
3 - Exit

Enter your Choice : 2

6

4

2

1

3

5

Select an Option

1 - Insert
2 - Display
3 - Exit

Enter your Choice : 3



15.5 BINARY TREE TRAVERSAL

Traversal is the process of visiting the various elements of a data structure. Binary tree traversal can be performed using three methods:

1. Preorder
2. Inorder
3. Postorder

1. Preorder The preorder traversal method performs the following operations:

- (a) Process the root node (N).
- (b) Traverse the left subtree of N (L).
- (c) Traverse the right subtree of N (R).

2. Inorder The inorder traversal method performs the following operations:

- (a) Traverse the left subtree of N (L).
- (b) Process the root node (N).
- (c) Traverse the right subtree of N (R).

3. Postorder The postorder traversal method performs the following operations:

- (a) Traverse the left subtree of N (L).
- (b) Traverse the right subtree of N (R).
- (c) Process the root node (N).

Figure 15.7 shows an illustration of the different binary tree traversal methods.

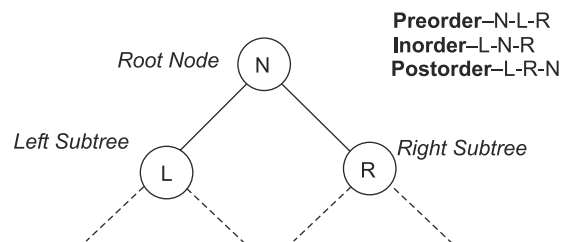
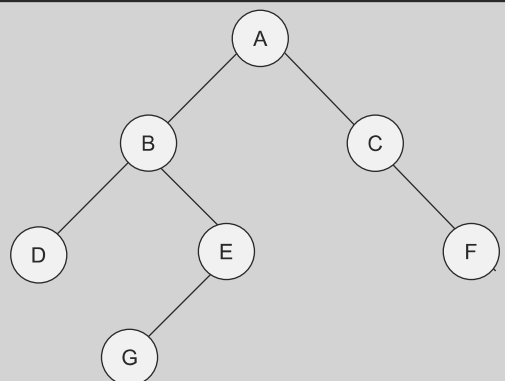


Fig. 15.7 Binary tree traversal

Example 15.1 Consider the following binary tree:
 For the above binary tree, deduce the following:
 (a) Preorder traversal sequence
 (b) Inorder traversal sequence
 (c) Postorder traversal sequence

- (a) Preorder traversal sequence
A–B–D–E–G–C–F
- (b) Inorder traversal sequence
D–B–G–E–A–C–F
- (c) Postorder traversal sequence
D–G–E–B–F–C–A



Example 15.2 Write algorithms for the following:

- (a) Preorder traversal
- (b) Inorder traversal
- (c) Postorder traversal

(a) Preorder

preorder(root)

Step 1: Start

Step 2: Display root

Step 3: Function Call preorder(root->LEFT)

Step 4: Function Call preorder(root->RIGHT)

Step 5: Stop

(b) Inorder

inorder(root)

Step 1: Start

Step 2: Function Call inorder(root->LEFT)

Step 3: Display root

Step 4: Function Call inorder(root->RIGHT)

Step 5: Stop

(c) Postorder

postorder(root)

Step 1: Start

Step 2: Function Call postorder(root->LEFT)

Step 3: Function Call postorder(root->RIGHT)

Step 4: Display root

Step 5: Stop

Program 15.2 Binary Tree Traversal

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

using namespace std;

struct node
{
    int INFO;
    struct node *LEFT, *RIGHT;
};

node *insert(node *, int); // Function prototype for Inserting a New Node
void preorder (node*); // Function prototype for displaying Preorder Taversal Path
void inorder (node*); // Function prototype for displaying Inorder Traversal Path
void postorder (node*); // Function prototype for displaying Postorder Traversal Path

int count = 1; // Counter for ascertaining left to right position for the New Node

int main()
{
```

(Contd.)

```

node *root = NULL;
int element, choice;

system("cls");

// Displaying a menu of choices
while(1)
{
    system("cls");

    cout << "\nSelect an Option\n";
    cout << "\n1 - Insert";
    cout << "\n2 - Preorder";
    cout << "\n3 - Inorder";
    cout << "\n4 - Postorder";
    cout << "\n5 - Exit";

    cout << "\n\nEnter your Choice : ";
    cin >> choice;

    switch(choice)
    {
        case 1:
        {
            cout << "\nEnter the node value : ";
            cin >> element;
            root = insert(root, element); // Calling the insert function
for inserting a new element into the tree
            break;
        }

        case 2:
        {
            preorder(root); // Calling the preorder function
            getch();
            break;
        }

        case 3:
        {
            inorder(root); // Calling the inorder function
            getch();
            break;
        }

        case 4:
        {
            postorder(root); // Calling the postorder function
            getch();
            break;
        }
    }
}

```

(Contd.)


```
        case 5:
        {
            exit(1);
            break;
        }

        default :
        {
            cout << "\n\n Incorrect Choice!!! Please Try Again...";
            getch();
            break;
        }
    }
}

node *insert(node *r, int n)
{
    if(r == NULL)
    {
        r = (node*) malloc (sizeof(node));
        r -> LEFT = r -> RIGHT = NULL;
        r -> INFO = n;
        count = count + 1;
    }

    else
    {
        if(count % 2 == 0)
            r -> LEFT = insert(r -> LEFT,n);
        else
            r -> RIGHT = insert(r -> RIGHT,n);
    }
    return(r);
}

void preorder(node *r)
{
    if(r != NULL)
    {
        cout << r->INFO << endl;
        preorder(r->LEFT);
        preorder(r->RIGHT);
    }
}

void inorder(node *r)
{
    if(r != NULL)
    {
```

(Contd.)

```
        inorder(r->LEFT);
        cout << r->INFO << endl;
        inorder(r->RIGHT);
    }
}

void postorder(node *r)
{
    if(r != NULL)
    {
        postorder(r->LEFT);
        postorder(r->RIGHT);
        cout << r->INFO << endl;
    }
}
```

Output

Select an Option

- 1 - Insert
- 2 - Preorder
- 3 - Inorder
- 4 - Postorder
- 5 - Exit

Enter your Choice : 1

Enter the node value : 1

Enter your Choice : 1

Enter the node value : 2

Enter your Choice : 1

Enter the node value : 3

Enter your Choice : 1

Enter the node value : 4

Enter your Choice : 1

Enter the node value : 5

Enter your Choice : 1

Enter the node value : 6

Select an Option

- 1 - Insert
- 2 - Preorder
- 3 - Inorder
- 4 - Postorder
- 5 - Exit

(Contd.)

Enter your Choice : 2

1
2
4
6
3
5

Select an Option

1 - Insert
2 - Preorder
3 - Inorder
4 - Postorder
5 - Exit

Enter your Choice : 3

6
4
2
1
3
5

Select an Option

1 - Insert
2 - Preorder
3 - Inorder
4 - Postorder
5 - Exit

Enter your Choice : 4

6
4
2
5
3
1

Select an Option

1 - Insert
2 - Preorder
3 - Inorder
4 - Postorder
5 - Exit

Enter your Choice : 5



15.6 BINARY SEARCH TREE

A binary tree is referred as a binary search tree if for any node n in the tree:

1. the node elements in the left subtree of n are lesser in value than n .
2. the node elements in the right subtree of n are greater than or equal to n .

Thus, binary search tree arranges its node elements in a sorted manner. As the name suggests, the most important application of a binary search tree is searching. The average running time of searching an element in a binary search tree is $O(\log n)$, which is better than other data structures like array and linked lists.

Figure 15.8 shows a sample binary search tree.

As we can see in the figure, all the nodes in the left subtree are less than the nodes in the right subtree.

The various operations performed on a binary search tree are:

1. Insert
2. Search
3. Delete

1. Insert The insert operation involves adding an element into the binary tree. The location of the new element is determined in such a manner that insertion does not disturb the sort order of the tree.

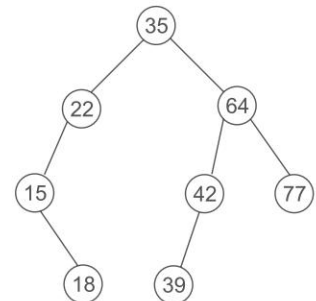


Fig. 15.8 Binary search tree

Example 15.3 Insert into Binary Tree

```

node *insert(node *r, int n)
{
    if(r == NULL)
    {
        r = (node*) malloc (sizeof(node));
        r -> LEFT = r -> RIGHT = NULL;
        r -> INFO = n;
    }

    else if(n < r -> INFO)
        r -> LEFT = insert(r -> LEFT,n);
    else if(r > r -> INFO)
        r -> RIGHT = insert(r -> RIGHT,n);
    else if(r == r -> INFO)
        cout << "\nInsert Operation Failed : Duplicate Entry!!!"
    return(r);
}
  
```



Check Point

1. Which node in a binary tree does not have a parent node?

Ans. Root

2. Which tree traversal method processes the root node first and then the left and right subtrees?

Ans. Preorder

- 2. Search** The search operation involves traversing the various nodes of the binary tree to search the desired element. The sorted nature of the tree greatly benefits the search operation as with each iteration, the number of nodes to be searched gets reduced. For example, if the value to be searched is less than the root value then the remainder of the search operation will only be performed in the left subtree while the right subtree will be completely ignored.

Example 15.4 Searching in Binary Tree

```
void search(node *r, int n)
{
    if(r == NULL)
    {
        cout << "\n" << n << " is not present in the Tree!!!";
        return;
    }

    else if(n == r -> INFO)
        cout << "\nElement " << n << " is present in the Tree!!!";
    else if(n < r -> INFO)
        search(r -> LEFT, n);
    else
        search(r -> RIGHT, n);
}
```

- 3. Delete** The delete operation involves removing an element from the binary search tree. It is important to ensure that after the element is removed from the tree, the other elements are shuffled in such a manner that the sort order of the tree is regained. The delete operation is depicted in Fig. 15.9.

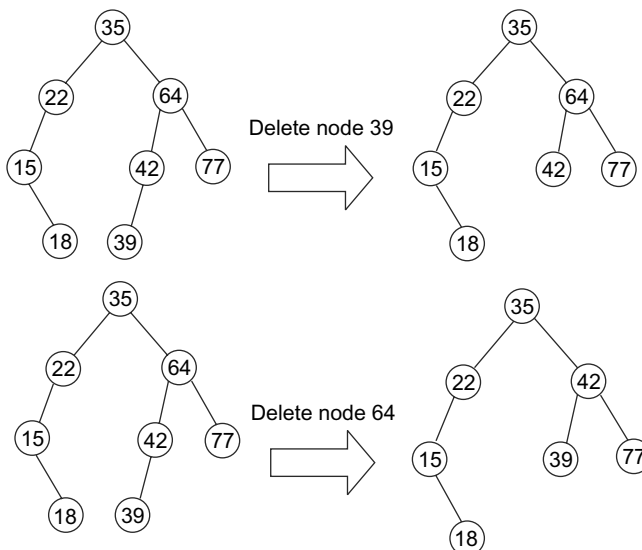


Fig. 15.9 Deleting an element from binary search tree

As we can see in Fig. 15.9, if the node to be deleted is a leaf node, then it is simply deleted without requiring any shuffling of other nodes. However, if the node to be deleted is an internal node then appropriate shuffling is required to ensure that the tree regains its sort order.

Example 15.5 Function for Deleting Element from a Binary Search Tree

```
int del(node *r, int n)
{
    node *ptr;

    if(r == NULL)
        return(0);

    else if(n < r -> INFO)
        return (del(r -> LEFT, n));
    else if(n > r -> INFO)
        return (del(r -> RIGHT, n));
    else
    {
        if(r -> LEFT == NULL)
        {
            ptr = r;
            r = r -> RIGHT;
            free(ptr);
            return(1);
        }

        else if(r -> RIGHT == NULL)
        {
            ptr = r;
            r = r -> LEFT;
            free(ptr);
            return(1);
        }

        else
        {
            ptr = r -> LEFT;
            while(ptr -> RIGHT != NULL)
                ptr = ptr -> RIGHT;
            r -> INFO = ptr -> INFO;
            return(del(r -> LEFT, ptr -> INFO));
        }
    }
}
```

4. Implementation The implementation of a binary search tree requires implementing the insert, search, and delete operations.

Program 15.3 uses the insert (Example 15.3), search (Example 15.4), and delete (Example 15.5) functions.

Program 15.3 Implementation of a Binary Tree

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

using namespace std;

struct node
{
    int INFO;
    struct node *LEFT, *RIGHT;
};

node *insert(node *, int); // Function prototype for Inserting a New Node
void search(node *, int n); // Function prototype for searching a Tree Nodes
int del(node *r, int n); // Function prototype for deleting a Tree Node
void display (node*); // Function prototype for displaying the Tree Nodes

int main()
{
    node *root = NULL;
    int element, choice, num, flag;

    system("cls");

    // Displaying a menu of choices

    while(1)
    {
        //system("cls");

        cout << "\nSelect an Option\n";
        cout << "\n1 - Insert";
        cout << "\n2 - Search";
        cout << "\n3 - Delete";
        cout << "\n4 - Display";
        cout << "\n5 - Exit";

        cout << "\n\nEnter your Choice : ";
        cin >> choice;

        switch(choice)
        {
            case 1:
            {
                cout << "\nEnter the node value : ";
                cin >> element;
```

(Contd.)

```

        root = insert(root, element); // Calling the insert
function for inserting a new Element into the Tree
        break;
    }

    case 2:
    {
        cout << "\nEnter the Element to be Searched : ";
        cin >> num;
        search(root, num); // Calling the Search function for
searching an Element in the Tree
        break;
    }

    case 3:
    {
        cout << "\nEnter the Element to be deleted : ";
        cin >> num;
        flag = del(root, num); // Calling the delete function for
deleting an Element in the Tree
        if(flag == 1)
            cout << "\n Element " << num << " is deleted from the Tree";
        else
            cout << "\n Element " << num << " is not present in
the Tree";
        break;
    }

    case 4:
    {
        display(root); // Calling the display function for printing
the node values
        getch();
        break;
    }

    case 5:
    {
        exit(1);
        break;
    }

    default :
    {
        cout << "\n\n Incorrect Choice!!! Please Try Again...";
        getch();
        break;
    }
}
}

```

(Contd.)


```
}

void display(node *r)
{
    if(r -> LEFT != NULL)
        display(r -> LEFT);
    cout << r -> INFO << endl;
    if(r -> RIGHT != NULL)
        display(r -> RIGHT);
}

int del(node *r, int n)
{
    node *ptr;

    if(r == NULL)
        return(0);

    else if(n < r -> INFO)
        return (del(r -> LEFT, n));
    else if(n > r -> INFO)
        return (del(r -> RIGHT, n));
    else
    {
        if(r -> LEFT == NULL)
        {
            ptr = r;
            r = r -> RIGHT;
            free(ptr);
            return(1);
        }
        else if(r -> RIGHT == NULL)
        {
            ptr = r;
            r = r -> LEFT;
            free(ptr);
            return(1);
        }
        else
        {
            ptr = r -> LEFT;
            while(ptr -> RIGHT != NULL)
                ptr = ptr -> RIGHT;
            r -> INFO = ptr -> INFO;
            return(del(r -> LEFT, ptr -> INFO));
        }
    }
}

void search(node *r, int n)
```

(Contd.)

```

{
    if(r == NULL)
    {
        cout << "\n" << n << " is not present in the Tree!!!";
        return;
    }

    else if(n == r -> INFO)
        cout << "\nElement " << n << " is present in the Tree!!!";
    else if(n < r -> INFO)
        search(r -> LEFT, n);
    else
        search(r -> RIGHT, n);
}

node *insert(node *r, int n)
{
    if(r == NULL)
    {
        r = (node*) malloc (sizeof(node));
        r -> LEFT = r -> RIGHT = NULL;
        r -> INFO = n;
    }

    else if(n < r -> INFO)
        r -> LEFT = insert(r -> LEFT,n);
    else if(n > r -> INFO)
        r -> RIGHT = insert(r -> RIGHT,n);
    else if(n == r -> INFO)
        cout << "\nInsert Operation Failed : Duplicate Entry!!!";
    return(r);
}

```

Output

Select an Option

- 1 - Insert
- 2 - Search
- 3 - Delete
- 4 - Display
- 5 - Exit

Enter your Choice : 1

Enter the node value : 6

Enter your Choice : 1

Enter the node value : 1

Enter your Choice : 1

(Contd.)

```
Enter the node value : 5
Enter your Choice : 1
Enter the node value : 2
Enter your Choice : 1
Enter the node value : 4
Enter your Choice : 1
Enter the node value : 3
Select an Option
1 - Insert
2 - Search
3 - Delete
4 - Display
5 - Exit

Enter your Choice : 4
1
2
3
4
5
6
Select an Option
1 - Insert
2 - Search
3 - Delete
4 - Display
5 - Exit

Enter your Choice : 2
Enter the Element to be Searched : 7
7 is not present in the Tree!!!
Enter your Choice : 2
Enter the Element to be Searched : 4
Element 4 is present in the Tree!!!
Select an Option
1 - Insert
2 - Search
3 - Delete
4 - Display
5 - Exit

Enter your Choice : 3
Enter the Element to be deleted : 8
Element 8 is not present in the Tree
Enter your Choice : 3
```

(Contd.)

```

Enter the Element to be deleted : 4
Element 4 is deleted from the Tree
Select an Option
1 - Insert
2 - Search
3 - Delete
4 - Display
5 - Exit

Enter your Choice : 5

```



15.7 TREE VARIANTS

Based on the concept of trees, binary trees and binary search trees various tree variants have been deduced. Each of these variants possesses distinct characteristics and serves specific purposes. For example, balanced binary trees balance their nodes in such a way that the height of the tree is always kept to a minimum, thus ensuring better average case performance at the time of searching.

In the subsequent sections, we will learn about the various tree variants.

15.7.1 Expression Trees

Expression tree is nothing but a binary tree containing mathematical expression. The internal nodes of the tree are used to store operators while the leaf or terminal nodes are used to store operands. Various compilers and parsers use expression trees for evaluating arithmetic and logical expressions.

Consider the following expression:

The expression tree for the above expression is shown in Fig. 15.10.

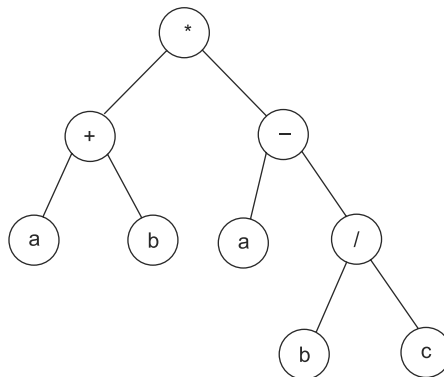


Fig. 15.10 Expression tree

As shown in the above tree, the internal nodes store the operators while the leaf nodes store the operands. While constructing a binary tree from a given expression, the following precedence rules are followed:

1. Parentheses are evaluated first.
2. The exponential expressions are evaluated next.

3. Then, division and multiplication operations are evaluated.
4. Finally, addition and subtraction operations are evaluated.

Representing an expression using a binary tree has another key advantage. By applying the various traversal methods we can deduce the other representations of an expression. For example, the preorder traversal of an expression tree derives its prefix notation.

Table 15.3 shows the various expression notations deduced after traversing the expression tree shown in Fig. 15.10.

Table 15.3 Expression notations

Expression Notation	Traversal Method	Example (Refer to Fig. 15.10)
Prefix	Preorder	$* + ab - a/bc$
Infix	Inorder	$a + b*a - a/c$
Postfix	Postorder	$ab + abc/-*$

15.7.2 Threaded Binary Trees

Let us recall the structure declaration of a tree node described during the linked implementation of a binary tree:

```
struct bin_tree
{
    int INFO;
    struct node *LEFT, *RIGHT;
};
```

As we can see in the above declaration, each node in a binary tree has two pointer nodes associated with it, i.e., LEFT and RIGHT. Now, in case of leaf nodes, these pointers contain NULL values. Considering the number of leaf nodes that are there in a typical binary tree, this leads to a lot of memory space getting wasted. Threaded binary trees offer an innovative alternate to avoid this memory wastage.

In a threaded binary tree, all nodes that do not have a right child contain a pointer or a thread to its inorder successor. The address of the inorder successor node is stored in the RIGHT pointer. But, how do we distinguish between a normal pointer and a thread pointer? This is done with the help of a Boolean variable, as shown in the below node declaration of a threaded binary tree:

```
struct t_tree
{
    int INFO;
    struct node *LEFT, *RIGHT;
    boolean LThread, RThread;
};
```



Note

Just like a right thread points at the inorder successor, we can also make the left thread to point at the postorder successor so as to deduce the postorder traversal sequence.

Figure 15.11 shows a threaded binary tree.

As shown in the figure, nodes D, E, F and H contain threads to point at their inorder successors.

Now, what is the advantage of a threaded binary tree representation? Try to recall the algorithm for inorder traversal of a binary tree. The algorithm uses recursive function calls to determine the inorder traversal path. The execution of recursive function calls requires the use of stack and consumes both memory as well as time. The threaded tree traversal allows us to determine the inorder sequence using an iterative approach instead of a recursive approach.

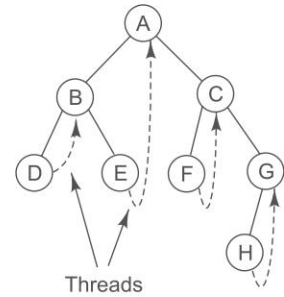


Fig. 15.11 Threaded binary tree

Example 15.6 Algorithm for Traversal of a Threaded Binary Tree to Generate Inorder Sequence

inorder (node)

```

Step 1: Start
Step 2: Set current = leftmost (node)
//current refers to the current node
//leftmost function returns the left most node value in a subtree
Step 3: while current != NULL repeat Steps 4-7
Step 4: Display current
Step 5: If current->RThread != NULL goto Step 6 else goto Step 7
Step 6: Set current = current->RIGHT
Step 7: Set current = leftmost (current->RIGHT)
Step 8: Stop

```

leftmost (node)

```

Step 1: Start
Step 2: Set ptr = node
Step 3: if ptr = NULL goto Step 4 else goto Step 5
Step 4: Return NULL and goto Step 8
Step 5: while ptr->LEFT != NULL repeat Step 6
Step 6: Set ptr = ptr->LEFT
Step 7: Return ptr
Step 8: Stop

```

If we apply the above algorithm on the threaded binary tree shown in Fig. 15.11, then we will obtain the following inorder sequence:

D-B-E-A-F-C-H-G



Check Point

1. In an expression tree, the internal nodes contain _____ while the leaf nodes contain _____.

Ans. operators, operands

2. In a threaded binary tree, a RIGHT thread points at the _____ successor of a node.

Ans. Inorder

15.7.3 Balanced Trees

In the previous sections, we saw how nodes are added to a binary search tree. With each addition of a node in a tree, there is a possibility that the height of the tree may also get changed. The height of a tree has a direct affect on its efficiency to perform the search operation. For instance, consider the binary search trees shown in Fig. 15.12.

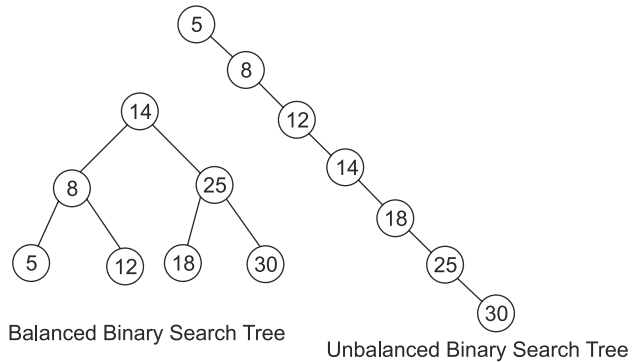


Fig. 15.12 Binary search trees

Both the binary search trees shown in the above figure contain the same nodes however the height of the first tree is 2 while that of the second tree is 6. To search element 30 in the above trees, we need to dig a lot deeper in the second tree as compared to the first tree. Thus, while implementing binary trees, it is important to keep the height of the tree in check.

There are various binary search trees that keep the tree balanced whenever a new node is added by shuffling the tree nodes appropriately. These are:

1. AVL tree
2. Red-Black tree

1. AVL tree AVL tree, also called *height-balanced tree* was defined by mathematicians Adelson, Velskii and Landis in the year 1962. The main characteristic of an AVL tree is that for all its nodes, the height of the left subtree and the height of the right subtree never differ by more than

At any point of time, an AVL tree node is in any one of the following states:

- (a) *Balanced* The height of left subtree is equal to the height of right subtree.
- (b) *Left heavy* The height of left subtree is one more than the height of right subtree.
- (c) *Right heavy* The height of right subtree is one more than the height of left subtree.

Figure 15.13 shows an AVL tree.

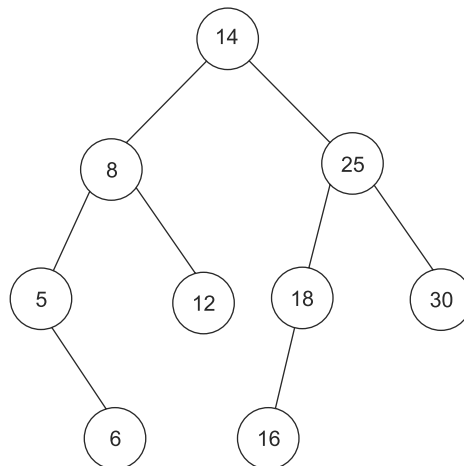


Fig. 15.13 AVL tree

As shown in Fig. 15.13, the height of left and right subtrees of each node differs by not more than 1.

Now, how is an AVL tree created and maintained? This is done by associating a balance factor (BF) with each node that keeps a track of the height balance for that particular node. BF for a node is calculated by using the following formula:

$$\text{BF} = \text{Height of Left Subtree} - \text{Height of Right Subtree}$$

Let us apply the above formula to calculate the balance factor for each node of the AVL tree shown in Fig. 15.13. Figure 15.14 shows the updated AVL tree.

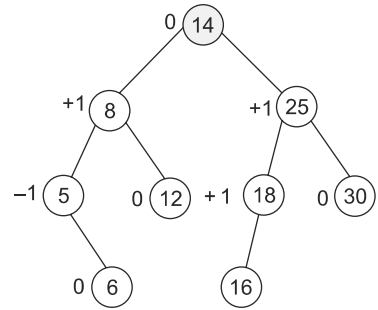


Fig. 15.14 AVL tree with balance factors

As shown in Fig. 15.14, the balance factors of all the nodes are not more than 1, which is the key characteristic of an AVL tree.

The structure declaration of an AVL tree node contains an additional field for storing the balance factor, as shown below:

```
struct avl_node
{
    int INFO;
    struct node *LEFT, *RIGHT;
    int BF;
};
```

Whenever a new node is inserted in an AVL tree, a slight disbalance is created at the point of insertion which reflects in the balance factors of the nodes in its preceding path till the root node. To restore the balance of the tree, left and right rotations are carried out to move the nodes towards the right or left. This is repeated until the balance factors of all the nodes are reduced below 1.

Figure 15.15 shows the insertion of node value 15 into the AVL tree shown in Fig. 15.14. It depicts how the disbalance resulting out of the insert operation is corrected.

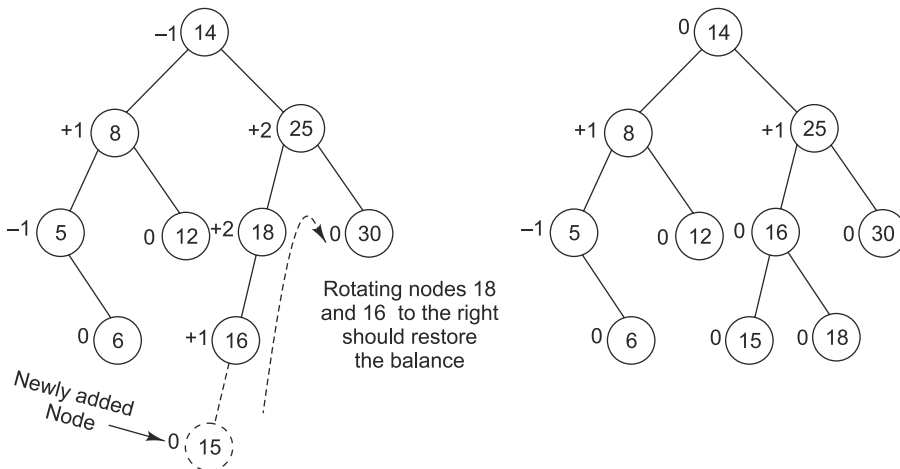


Fig. 15.15 Inserting an element in an AVL tree

The delete operation follows a similar approach. A left or right rotation may need to be carried out if a node is deleted from an AVL tree.

2. Red-Black tree Red-Black tree is a self-balancing binary search tree that has an average running time of $O(\log n)$ for insert, delete and search operations. As the name suggests, the red-black tree associates a color attribute with each node, which can possess only two values, red or black. That means each node in a red-black tree is either red or black colored. Apart from possessing the properties of a typical binary search tree, a red-black tree possesses the following properties:

- Each node is either red or black in color.
- The root node is black colored.
- The leaf nodes are black colored. It includes the NULL children.
- The child nodes of all red-colored nodes are black.
- Each path from a given node to any of its leaf nodes contains equal number of black nodes. The number of such black nodes is also referred as black-height (bh) of the node.

The above properties ensure that the length of the longest path from the root node to a leaf node is less than roughly twice of the shortest path. This ensures that the balance of the tree is always kept under check. The key advantage of a red-black tree is that its worst case running time is better than most of the other binary search trees.

Figure 15.16 shows a red-black tree.

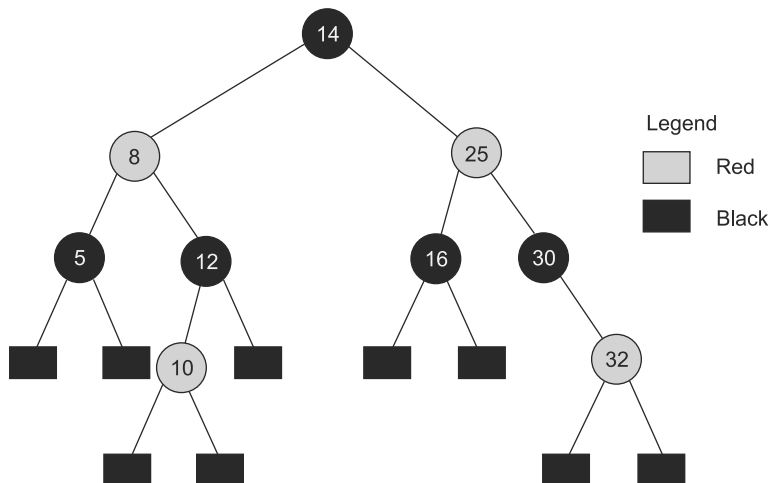


Fig. 15.16 Red-Black tree

The insert and delete operations on a red-black tree require small number of rotations as well as change of colors of some of the nodes so that the tree complies with all the properties of a red-black tree. However, the average running time of these operations is $O(\log n)$.

15.7.4 Splay Trees

The concept of splay trees is based on the assumption that when a particular element is accessed from a binary search tree then there are high chances that the same element would be accessed again in future. Now, if the element is placed deep in the tree then all such repetitive accesses would be inefficient. To make the repetitive accesses of a node efficient, splay tree shifts the accessed node towards the root

two levels at a time. This shifting is done through splay rotations. Table 15.4 shows the various types of splay rotations along with an illustration.

Table 15.4 Splay rotations

Splay Rotation	Occurrence	Illustration
Zig	When root node P is the parent of the node N being accessed.	
Zigzag	When node N is the right child of parent P, which itself is the left child of grandparent G. Or, when node N is the left child of parent P, which itself is the right child of grandparent G.	
Zigzig	When both node N and parent P are left or right child of grandparent G.	

Splay rotations ensure that all future accesses of a node are efficient as compared to its first time access. Let us now discuss what happens when typical tree-related operations are performed on a splay tree:

- 1. Insert** New element is inserted at the root.
- 2. Search** There are two possibilities:
 - (a) *Successful search* The searched node is moved to the root position.
 - (b) *Unsuccessful search* The last node accessed during the unsuccessful search operation is moved to the root position.
- 3. Delete** The element to be deleted is first brought to the root position. After deleting the root node, the largest node in the left subtree is moved to the root position.

15.7.5 *m*-way Trees

Binary search trees are more suitable for smaller data sets where the data is static. However, for large data sets which require dynamic access (example file storage); binary search trees are not exactly suitable. For such cases, the nodes of the tree are required to store large amounts of data. This is achieved with the help of *m*-way trees.

m-way search trees are an extension of binary search trees having the following properties:

1. Each node of the tree stores 1 to $m - 1$ number of keys.
2. The keys are stored in a sorted manner inside the node.

3. A node containing k values can have a maximum of $k + 1$ subtrees.
4. The subtree pointed by pointer T_i has values less than the key value of k_{i+1} .
5. All the subtrees are m -way trees.

Figure 15.17 shows a sample m -way tree.

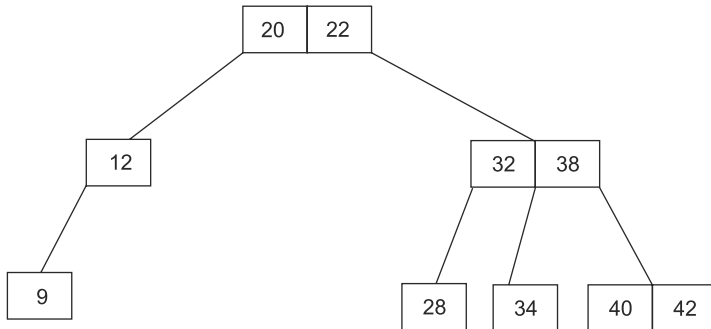


Fig. 15.17 Sample m -way tree

1. B tree To ensure efficiency while searching an m -way tree it is important to control its height. This is achieved with the help of B tree. A B tree is nothing but a height balanced m -way search tree.

A B tree of order m has the following properties:

- i. Root node is either a leaf node or it contains child nodes ranging from 2 to m .
- ii. All internal nodes contain a maximum of $m - 1$ keys.
- iii. Number of children of internal nodes ranges from $m/2$ to m .
- iv. Number of keys stored in the leaf nodes ranges from $(m - 1)/2$ to $m - 1$. All the keys are stored in a sorted manner.
- v. All leaf nodes are at the same depth.

Figure 15.18 shows a sample B tree.

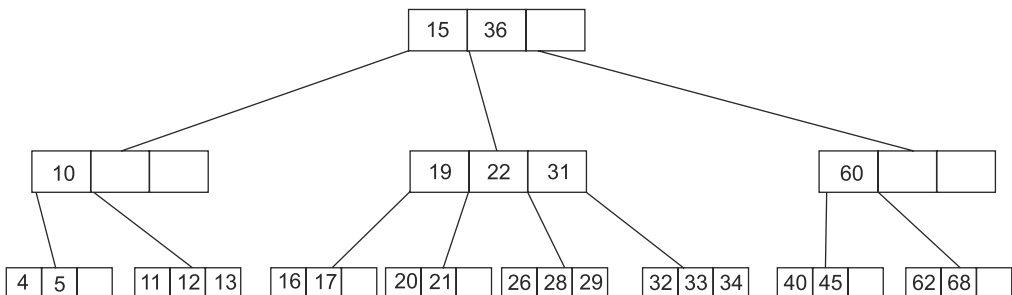


Fig. 15.18 Sample B tree

An element is inserted in a B tree by first identifying the location where the new node should be inserted. If the existing node is not full, the new element is inserted within the existing node and an appropriate pointer is created linking it with the parent node. However, if the existing node is full then it is split into three parts. The middle part is accommodated with the parent node while the new element is inserted in one of the child nodes.

Similarly, deletion of an element from a B tree is done by first removing the element from a node and then carrying out appropriate redistributions to ensure that the tree stays true to its properties.

2. B⁺ tree B⁺ tree is a variant of B tree that is mainly used for implementing index sequential access of records. The main difference between B⁺ tree and B tree is that in B⁺ tree data records are only stored in the leaf nodes. The internal nodes of a B⁺ tree are only used for storing the key values. The key values help in performing the search operation. If the target element is less than a key value then the search proceeds towards its left pointer. Similarly, if the target element is greater than a key value then the search proceeds towards its right pointer.

A B⁺ tree of order m has the following properties:

- i. The internal nodes contain up to $m - 1$ keys.
- ii. The number of children of internal nodes lies between $m/2$ and m .
- iii. The subtree between keys k_1 and k_2 contains values v such that $k_1 \leq v < k_2$.
- iv. All leaf nodes are at the same level.
- v. All the leaf nodes are sequentially connected through a linked list.

B⁺ tree is typically used for implementing index sequential file organization in a database. The internal nodes are used for representing index values through which data records in the sequence set are accessed.

Figure 15.19 shows a sample B⁺ tree.

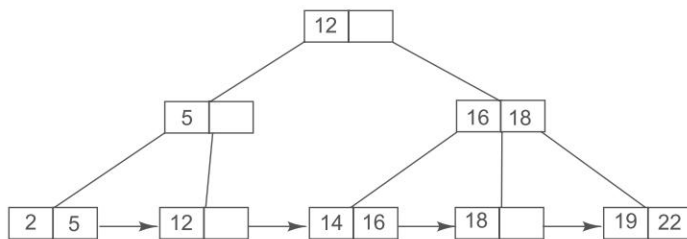


Fig. 15.19 Sample B⁺ tree



Check Point

1. In which situation is a zig-zig splay rotation performed?

Ans. When both node N and parent P are left or right child of grandparent G .

2. B⁺ tree implementation helps in performing _____ search.

Ans. Index-sequential search.

SUMMARY

- ☐ Tree is a non-linear data structure which stores the data elements in a hierarchical manner.
- ☐ The top node of a tree is marked as a root node while the remaining nodes are partitioned across the subtrees present under the root node.

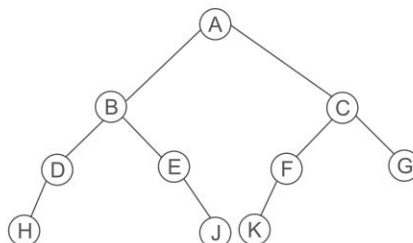
- ☐ A binary tree is a restricted form of a general tree that can have zero, one or two child nodes but not more than that.
- ☐ Traversal is the process of visiting the various elements of a data structure. Binary tree traversal can be performed using three methods: preorder, inorder and postorder.
- ☐ If N represents the parent node, L represents the left subtree and R represents the right subtree then
 - o *preorder sequence* N—L—R
 - o *inorder sequence* L—N—R
 - o *postorder sequence* L—R—N
- ☐ A binary search tree arranges its node elements in a sorted manner. The node elements in the left subtree are less than the parent node while the node elements in the right subtree are greater than or equal to the parent node.
- ☐ Expression tree is a binary tree whose internal nodes store operators while the leaf or terminal nodes store the operands.
- ☐ A threaded binary tree uses the empty NULL pointers of nodes to create threads to their inorder successors. This increases the inorder traversal efficiency by preventing the use of recursive function calls.
- ☐ In an AVL tree, the height of the left subtree and the height of the right subtree differ by not more than 1. Keeping the height of the tree in check ensures that the search efficiency is optimized.
- ☐ Red-Black tree is a self-balancing binary search tree that has an average running time of $O(\log n)$ for insert, delete and search operations. Each node in a red-black tree is colored either red or black.
- ☐ *m*-way search trees are a generalized form of binary search trees that are used for storing large amounts of data. The two types of *m*-way trees are: B tree and B⁺ tree.

Key Terms

Root | Leaf | Depth or Height | INFO | LEFT | RIGHT | Preorder | Inorder | Postorder | Thread | Balance factor

Review Questions

- 15.1 What is a tree? Explain any five key terms associated with a tree.
- 15.2 What is the difference between complete binary tree and perfect binary tree?
- 15.3 What are the two types of balanced binary trees? Explain with the help of an illustration.
- 15.4 What is the advantage of linked implementation of a binary tree over array implementation?
- 15.5 What are the different types of tree traversal methods? Explain with the help of an example.
- 15.6 Deduce the preorder and postorder sequences for the following binary tree:



- 15.7** What is a binary search tree? Explain with the help of an example.
- 15.8** What is an expression tree? Explain with the help of an example.
- 15.9** What is a splay tree? Explain the different types of splay rotations.
- 15.10** What is an m-way tree? Explain the two instances of m-way trees.

Programming Exercises

- 15.1** Write a function in C to count the number of nodes in a binary tree.
- 15.2** Write a function in C to display the elements of a binary search tree in ascending order.
- 15.3** Write a function in C that displays all the leaf nodes of a binary tree.
- 15.4** Write a function in C that returns the degree of a binary tree node.
- 15.5** Write a C function that transforms a given binary tree into a binary search tree.

16

GRAPHS

CHAPTER OUTLINE

- 16.1 Introduction
 - 16.2 Basic Concept
 - 16.3 Graph Terminology
 - 16.4 Graph Implementation
 - 16.4.1 Implementing Graphs Using Adjacency Matrix
 - 16.4.2 Implementing Graphs Using Path Matrix
 - 16.4.3 Implementing Graphs Using Adjacency List
 - 16.5 Shortest Path Algorithm
 - 16.6 Graph Traversal
 - 16.6.1 Breadth First Search
 - 16.6.2 Depth First Search
- Summary*
- Key Terms*





16.1 INTRODUCTION

Till now, we have learnt about different types of data structures, such as arrays, linked lists, trees, etc. In this chapter, we will learn about another important data structure called graph. It is similar to the mathematical graph structure, which comprises of a set of vertices connected with each other through edges. Some of the typical operations performed on a graph data structure include finding possible paths between two nodes and finding the shortest possible path.

Graph data structure finds its application in varied domains, such as computer network analysis, travel application, chip designing, gaming and so on.

In this chapter, we will learn how a graph data structure is represented and what algorithms are used for graph traversal. We will also learn about the shortest path algorithm that allows us to find the shortest path between two nodes.



16.2 BASIC CONCEPT

A graph G consists of the following elements:

- A set V of vertices or nodes where $V = \{v_1, v_2, v_3, \dots, v_n\}$
- A set E of edges also called arcs where $E = \{e_1, e_2, e_3, \dots, e_n\}$

Here, $G = (V, E)$.

Figure 16.1 shows a sample graph G .

In Fig. 16.1, e_1 is an edge between v_1 and v_2 vertices while e_2 is an edge between v_2 and v_3 vertices. Thus, we can generically represent an edge e as $e = (u, v)$ where e connects both u and v vertices.

Now, $e = (u, v)$ means the same thing as $e = (v, u)$. This means that the ordering of the vertices has no significance here. Thus, we can call the graph G as **undirected graph**.

If we replace each edge of the Graph G with arrows, then it will become a **directed graph** or **digraph**, as shown in Fig. 16.2.

In Fig. 16.2 graph, the set of vertices and edges are:

$$V(G) = \{v_1, v_2, v_3, v_4, v_5\}$$

$$E(G) = \{(v_1, v_2), (v_2, v_3), (v_1, v_4), (v_4, v_3), (v_3, v_5)\}$$

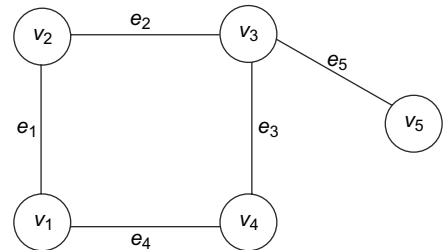


Fig. 16.1 Graph G

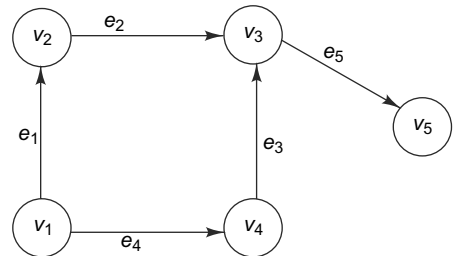


Fig. 16.2 Directed graph



16.3 GRAPH TERMINOLOGY

There are a number of key terms associated with the concept of graphs. Table 16.1 explains some of these important key terms.

Table 16.1 Graph terminology

Key Terms	Description
Adjacent node	If $e(u, v)$ represents an edge between u and v vertices then both u and v are called adjacent to each other. That means, u is adjacent to v and v is adjacent to u .
Predecessor node	If $e(u, v)$ represents a directed edge from u to v then u is a predecessor node of v .
Successor node	If $e(u, v)$ represents a directed edge from u to v then v is a successor node of u .
Degree	Degree of a vertex is the number of edges connected to a vertex. For example, in the graph shown in Fig. 16.1, the degree of vertex v_3 is 3.
Indegree	In a directed graph, indegree of a vertex is the number of edges ending at the vertex.
Outdegree	In a directed graph, outdegree of a vertex is the number of edges beginning at the vertex.
Path	A path is a sequence of vertices each adjacent to the next. For example, in the graph shown in Fig. 16.2, the path between the vertices v_1 and v_5 is $v_1-v_2-v_3-v_5$.
Cycle	It is a path that starts and ends at the same vertex.
Loop	It is an edge whose endpoints are same that is, $e = (u, u)$.
Weight	It is a non-negative number assigned to an edge. It is also called length.
Order	Order of a graph is the number of the vertices contained in the graph.
Labeled Graph	It is a graph that has labeled edges.
Weighted Graph	It is a graph that has weights assigned to each of its edges.
Connected Graph	It is an undirected graph in which there is a path between each pair of nodes.
Strongly Connected Graph	It is a directed graph in which there is a route between each pair of nodes.
Complete Graph	It is an undirected graph in which there is a direct edge between each pair of nodes.
Tree	It is a connected graph with no cycles.

Note There are no standards defined related to the use of graph terminology. Thus, you may find the same concept being referred with different names at different places. For instance, a graph edge could also be referred as an arc or a link.



16.4 GRAPH IMPLEMENTATION

Graphs are nothing but a collection of nodes and edges. Thus, while representing graphs in memory the only focus is on capturing details related to the different vertices and edges. Graphs can be implemented using the following methods:

1. Adjacency matrix
2. Path matrix
3. Adjacency list



Check Point

1. In a weighted directed graph, what does the term weighted and directed signify?

Ans. The term weighted signifies that all the edges of the graph are assigned an integer number called weight. The term directed signifies that each edge of the graph is a pointed arrow that points from one vertex to the other.

2. What is indegree?

Ans. Indegree of a vertex is the number of edges incident on the vertex.

16.4.1 Implementing Graphs Using Adjacency Matrix

Consider a graph $G = (V, E)$ having N nodes. The adjacency matrix of graph G is defined as an $N \times N$ matrix A , where:

1. $A_{i,j} = 1$, if there is an edge from vertex v_i to v_j
2. and $A_{i,j} = 0$, if there is no edge from vertex v_i to v_j

Let us try and understand the concept of adjacency matrix with the help of an example: Consider the graph shown in Fig. 16.3.

The adjacency matrix of the above graph will be

Here, 0s represent that there is no directed edge between the corresponding vertices while 1s represent the presence of a directed edge.

$A_{i,j}$	1	2	3	4	5
1	1	1	0	1	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	0	1	0	0
5	0	0	0	0	0

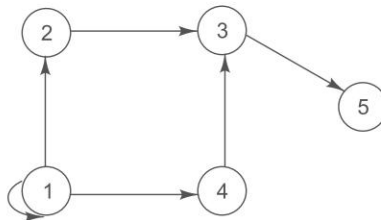


Fig. 9.3 Graph

Program 16.1 represents the directed graph shown in Fig. 16.3 with the help of adjacency matrix.

Program 16.1 Adjacency Matrix

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

using namespace std;
int main()
{
    int A[5][5];
    int i, j;

    system("cls");

    for(i = 0; i < 5; i++)
        for(j = 0; j < 5; j++)
            A[i][j] = 0; // Initializing the Array A

    // Creating the Adjacency Matrix

    A[0][0] = 1;
    A[0][1] = 1;
    A[0][3] = 1;
    A[1][2] = 1;
    A[2][4] = 1;
    A[3][2] = 1;

    // Printing the Adjacency Matrix
    cout << "Adjacency Matrix : \n";
    for(i = 0; i < 5; i++)
    {
        cout << endl << "\t";
        for(j = 0; j < 5; j++)
            cout << A[i][j] << "\t";
    }
    getch();
}
```

Output

Adjacency Matrix :

1	1	0	1	0
0	0	1	0	0
0	0	0	0	1
0	0	1	0	0
0	0	0	0	0

Advantages and Disadvantages

The advantage of adjacency matrix representation of a graph is that it is simple to implement. However it also has certain disadvantages, such as the following:

1. It requires $O(n^2)$ memory space even if the adjacency matrix is sparse.
2. It proves to be an inefficient representation for graphs that have large number of vertices.

16.4.2 Implementing Graphs Using Path Matrix

Consider a digraph $G = (V, E)$ having N nodes. The path matrix of graph G is defined as an $N \times N$ matrix P , where

1. $P_{i,j} = 1$, if there is a path from vertex v_i to v_j , and
2. $P_{i,j} = 0$, if there is no path from vertex v_i to v_j .

Now, the path matrix P can be deduced using the adjacency matrix of G , as depicted below:

$$P_N = A + A^2 + A^3 + \dots + A^N$$

Here, A^2 is the square of the adjacency matrix A , A^3 is the cube of A , and so on. All the non-zero entries resulting from the addition operation above are replaced by 1 to arrive at the path matrix.

This method of deriving the path matrix by computing powers of adjacency matrix is not very efficient, as it requires performing a number of matrix multiplication operations. Warshall has suggested a more simplified method of deriving the path matrix from the adjacency matrix. Warshall's method determines the presence of a path between v_i and v_j by

1. identifying a direct path from v_i to v_j , and
2. identifying an indirect path from v_i and v_j that is, a path from v_i to v_k and v_k to v_j .

That is, $P_{i,j} = P_{i,j}$ OR ($P_{i,k}$ AND $P_{k,j}$)

Here, OR represents the logical OR operation and AND represents the logical AND operation.

Example 16.1 Warshall's Algorithm for Deriving the Path Matrix of a Digraph G

```
path_matrix(Adjacency Matrix A[], N)
Step 1: Start
Step 2: Set P[] = A[]
Step 3: Set i = j = k = 1
Step 4: Repeat Steps 5-10 while k <=N
Step 5: Repeat Steps 6-9 while i <=N
Step 6: Repeat Steps 7-8 while j <=N
Step 7: P[i,j] = P[i,j] OR (P[i,k] AND P[k,j])
Step 8: j = j + 1
Step 9: i = i + 1
Step 10: k = k + 1
Step 11: Display path matrix P[]
Step 12: Stop
```

Program 16.2 uses Warshall's algorithm to derive the path matrix of the digraph shown in Fig. 16.3.

Program 16.2 Warshall's Algorithm for Deriving the Path Matrix of a Digraph G

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

using namespace std;

int AND(int, int); // Function Prototype for performing Logical And
Operation
int OR(int, int); // Function Prototype for performing Logical OR Operation

int main()
{
    int A[5][5], P[5][5];
    int i, j, k;

    system("cls");

    for(i = 0; i < 5; i++)
        for(j = 0; j < 5; j++)
            A[i][j] = 0; // Initializing the Array A

    // Creating the Adjacency Matrix

    A[0][0] = 1;
    A[0][1] = 1;
    A[0][3] = 1;
    A[1][2] = 1;
    A[2][4] = 1;
    A[3][2] = 1;

    // Printing the Adjacency Matrix
    cout << "Adjacency Matrix : \n";
    for(i = 0; i < 5; i++)
    {
        cout << endl << "\t";
        for(j = 0; j < 5; j++)
            cout << A[i][j] << "\t";
    }

    for(i = 0; i < 5; i++)
        for(j = 0; j < 5; j++)
            P[i][j] = A[i][j];

    // Creating Path Matrix
    for(k=0; k<5; k++)
        for(i = 0; i < 5; i++)
            for(j = 0; j < 5; j++)
                P[i][j] = OR(P[i][j], AND(P[i][k], P[k][j]));
}
```

(Contd.)

```

// Printing Path Matrix
cout << "\n\nPath Matrix : \n";
for(i = 0; i < 5; i++)
{
    cout << endl << "\t";
    for(j = 0; j < 5; j++)
        cout << P[i][j] << "\t";
}
getch();
}

int AND(int x, int y)
{
    return(x*y);
}

int OR(int x, int y)
{
    if(x == 0 && y ==0)
        return(0);
    else
        return(1);
}

```

Output

Adjacency Matrix :

1	1	0	1	0
0	0	1	0	0
0	0	0	0	1
0	0	1	0	0
0	0	0	0	0

Path Matrix :

1	1	1	1	1
0	0	1	0	1
0	0	0	0	1
0	0	1	0	1
0	0	0	0	0

16.4.3 Implementing Graphs Using Adjacency List

Adjacency list is a linked representation of a graph. It consists of a list of graph nodes with each node itself consisting of a linked list of its neighboring nodes. Figure 16.4 shows the adjacency list of the directed graph shown in Fig. 16.3.

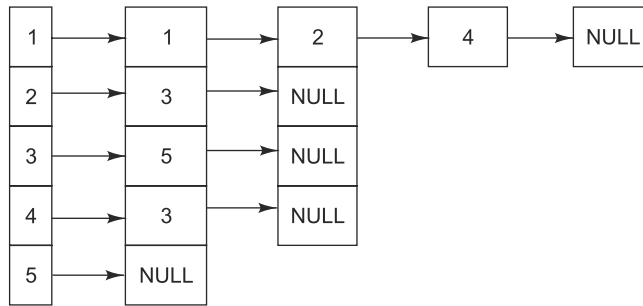


Fig. 16.4 Adjacency list

The adjacency list shown above contains five nodes with each node pointing towards its successor nodes.

Program 16.3 represents a directed graph using adjacency list.

Program 16.3 Graph using Adjacency List

```

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

using namespace std;

struct vertex
{
    struct vertex *edge[10];
    int id;
} node[10];

void display(int);

int main()
{
    int i, j, N;
    char ch;

    system("cls");

    i = j = N = 0;
    cout << "\nEnter the number of Graph Vertices : ";
    cin >> N;
    cout << endl;

    for(i=0; i<N; i++)
    {
        node[i].id = i;

        fflush(stdin);
    }
}

```

(Contd.)

```

        for(j=0; j<N; j++)
        {
            fflush(stdin);
            cout << "Edge from " << i+1 << " to " << j+1 << "? (y/n) : ";
            cin >> ch;

            if(ch == 'y' || ch == 'Y')
                node[i].edge[j] = &node[j];
            else
                node[i].edge[j] = NULL;
        }
    }

    display(N);
    getch();
}

void display(int num)
{
    int i, j;
    cout << endl;

    for(i =0; i<num; i++)
    {
        cout << "Edges of the node[" << i+1 << "] are : ";
        for(j=0; j<num; j++)
        {
            if(node[i].edge[j] == NULL)
                continue;
            cout << " (" << i+1 << "-" << node[i].edge[j] -> id+1
                << ") ";
        }
        cout << endl;
    }
}

```

Output

Enter the number of Graph Vertices : 3

```

Edge from 1 to 1? (y/n) : n
Edge from 1 to 2? (y/n) : y
Edge from 1 to 3? (y/n) : y
Edge from 2 to 1? (y/n) : y
Edge from 2 to 2? (y/n) : n
Edge from 2 to 3? (y/n) : y

```

(Contd.)

Edge from 3 to 1? (y/n) : n
 Edge from 3 to 2? (y/n) : y
 Edge from 3 to 3? (y/n) : y

Edges of the node[1] are : (1-2) (1-3)
 Edges of the node[2] are : (2-1) (2-3)
 Edges of the node[3] are : (3-2) (3-3)



Note Adjacency matrix and path matrix are both examples of sequential representation of graphs. Adjacency list is an example of linked representation of graphs.



16.5 SHORTEST PATH ALGORITHM

One of the most common problems associated with graphs is to find the shortest path from one node to the other. It finds its relevance in a number of real-life applications. For example, consider a scenario where a freight carrier departing from Delhi is required to drop consignments at Lucknow, Jaipur, Ahemadabad, Pune, and Hyderabad airports. In this situation, the route taken by the carrier is determined by assessing the distance between each of these cities. The final route undertaken should be the shortest path that covers all these cities.

We can relate this scenario with a graph data structure where each city represents a graph node while the distance between the cities represents the weight of the edges.

Consider the weighted digraph shown in Fig. 16.5.

The weight matrix of the above graph will be

Here, $W_{i,j}$ represents the weight of the edge from node v_i to v_j . The value 0 signifies that there is no direct edge between the corresponding nodes. Now, a modification of the Warshall's algorithm can be applied to the weight matrix to derive the shortest path matrix SP that represents the weight of the shortest possible path between any two nodes of a graph.

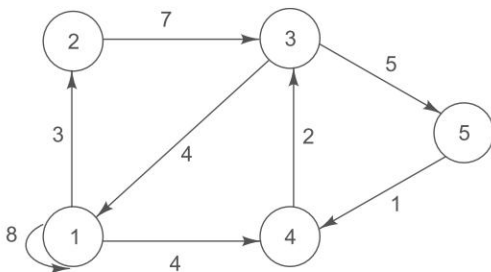


Fig. 9.5 Weighted digraph



Check Point

1. Write the Warshall's relation for computing a path matrix?

Ans. $P[i,j] = P[i,j]$ OR $(P[i,k] \text{ AND } P[k,j])$

2. The end of adjacency list of a graph node is signified by _____.

Ans. NULL pointer

$W_{i,j}$	1	2	3	4	5
1	8	3	0	4	0
2	0	0	7	0	0
3	4	0	0	0	5
4	0	0	2	0	0
5	0	0	0	1	0

It begins with replacing all 0's in the weight matrix with ∞ , as shown below.

$SP_{i,j}$	1	2	3	4	5
1	8	3	8	4	8
2	8	8	7	8	8
3	4	8	8	8	5
4	8	8	2	8	8
5	8	8	8	1	8

Now, the following relation is applied to arrive at the shortest path matrix:

$$SP_{i,j} = \text{Minimum of } (SP_{i,j}, SP_{i,k} + SP_{k,j})$$

The shortest path matrix obtained after applying the above relation for each graph node is

$SP_{i,j}$	1	2	3	4	5
1	8	3	6	4	11
2	11	14	7	13	12
3	4	7	8	6	5
4	6	9	2	8	7
5	7	10	3	1	8

Example 16.2 Modified Warshall's Algorithm for Deriving the Shortest Path Matrix of a Digraph G

```

shortest_path_matrix(Path Matrix P[], N)
Step 1: Start
Step 2: Set i = j = 1
Step 3: Repeat Steps 4-9 while i<=N
Step 4: Repeat Steps 5-8 while j<=N
Step 5: if P[i,j]=0 goto Step 6 else goto Step 7
Step 6: Set SP[i,j]= 8
Step 7: Set SP[i,j]=P[i,j]
Step 8: j = j + 1
Step 9: i = i + 1
Step 10: Set i = j = k = 1
Step 11: Repeat Steps 12-17 while k<=N
Step 12: Repeat Steps 13-16 while i <=N
Step 13: Repeat Steps 14-15 while j <=N
Step 14: SP[i,j] = MINIMUM(SP[i,j], SP[i,k]+SP[k,j])
Step 15: j = j + 1
Step 16: i = i + 1
Step 17: k = k + 1
Step 18: Display shortest path matrix SP[]
Step 19: Stop

```

Program 16.4 uses modified Warshall's algorithm to derive the shortest path matrix of the weighted digraph shown in Fig. 16.5.

Program 16.4 Modified Warshall's Algorithm for Deriving the Shortest Path Matrix of a Digraph G

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

using namespace std;

int MIN(int, int); // Function prototype for computing the Minimum among
2 Integers

int main()
{
    int P[5][5], SP[5][5];
    int i, j, k;

    system("cls");

    for(i = 0; i < 5; i++)
        for(j = 0; j < 5; j++)
            P[i][j] = 0; // Initializing the Array P

    P[0][0] = 8;
    P[0][1] = 3;
    P[0][3] = 4;
    P[1][2] = 7;
    P[2][0] = 4;
    P[2][4] = 5;
    P[3][2] = 2;
    P[4][3] = 1;

    cout << "\n\nPath Matrix : \n";

    for(i = 0; i < 5; i++)
    {
        cout << endl << "\t";
        for(j = 0; j < 5; j++)
            cout << P[i][j] << "\t";
    }

    for(i = 0; i < 5; i++)
    {
        for(j = 0; j < 5; j++)
        {
            if(P[i][j] == 0)
                SP[i][j] = 999;
```

(Contd.)

```

        else
            SP[i][j] = P[i][j];
    }
}

for(k = 0; k < 5; k++)
    for(i = 0; i < 5; i++)
        for(j = 0; j < 5; j++)
            SP[i][j] = MIN(SP[i][j], (SP[i][k] + SP[k][j]));

cout << "\n\nShortest Path Matrix : \n";

for(i = 0; i < 5; i++)
{
    cout << endl << "\t";
    for(j = 0; j < 5; j++)
        cout << SP[i][j] << "\t";
}

getch();
}

int MIN(int x, int y)
{
    if(x <= y)
        return(x);
    else
        return(y);
}

```

Output

Path Matrix :

8	3	0	4	0
0	0	7	0	0
4	0	0	0	5
0	0	2	0	0
0	0	0	1	0

Shortest Path Matrix :

8	3	6	4	11
11	14	7	13	12
4	7	8	6	5
6	9	2	8	7
7	10	3	1	8



16.6 GRAPH TRAVERSAL

One of the common tasks associated with graphs is to traverse or visit the graph nodes and edges in a systematic manner. There are two methods of traversing a graph:

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

Both these methods consider the graph nodes to be in one of the following states at any given point of time:

1. Ready state
2. Waiting state
3. Processed state

The state of a node keeps on changing as the graph traversal progresses. Once the state of a node becomes processed, it is considered as traversed or visited.

16.6.1 Breadth First Search

The BFS method begins with analyzing the starting node and then progresses by analysing its adjacent or neighbouring nodes. Once all the neighbouring nodes of the starting node are analyzed, the algorithm starts analyzing the neighboring nodes of each of the analyzed neighboring nodes. This method of graph traversal requires frequent backtracking to the already analyzed nodes. As a result, a data structure is required for storing information related to the neighboring nodes. The BFS method uses the queue data structure for storing the nodes data.

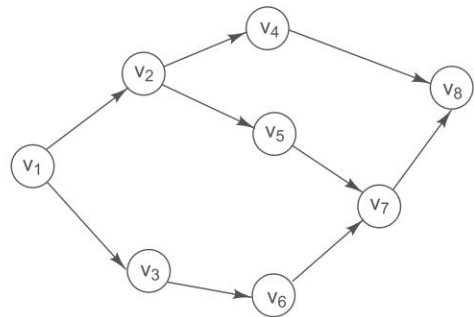


Fig. 16.6 Graph traversal

Consider the graph shown in Fig. 16.6.

The BFS traversal sequence for the above graph will be: $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$. Another BFS traversal sequence can be: $v_1, v_3, v_2, v_6, v_5, v_4, v_7, v_8$.

Example 16.3 Algorithm for the BFS Graph Traversal Method

BFS(adj[], status[], queue[], N)

Step 1: Start

Step 2: Set status[] = 1

Step 3: Push(queue, v1)

Step 4: Set status[v1]=2

Step 5: Repeat Step 6-11 while queue[] is not empty

Step 6: $V = \text{Pop}(\text{queue})$ ←

Here, the pop operation signifies the processing of a graph node.

Step 7: status[V]=3

Step 8: Repeat Step 9-11 while adj(V) is not empty

Step 9: If adj(V) = 1 goto step 10 else goto step 8

Step 10: Push(queue, adj(V))

Step 11: Set adj[v]=2

Step 12: Stop



Note *Pop means removing an element from the queue while push means inserting an element into the queue.*

16.6.2 Depth First Search

Unlike the BFS traversal method, which visits the graph nodes level by level, the DFS method visits the graph nodes along the different paths. It begins analyzing the nodes from the start to the end node and then proceeds along the next path from the start node. This process is repeated until all the graph nodes are visited.

The DFS method also requires frequent backtracking to the already analyzed nodes. It uses the stack data structure for storing information related to the previous nodes.

Let us again consider the graph shown in Fig. 16.6. The DFS traversal sequence for this graph will be: $v_1, v_2, v_4, v_8, v_5, v_7, v_3, v_6$. Another DFS traversal sequence can be: $v_1, v_3, v_6, v_7, v_8, v_2, v_5, v_4$.

Example 16.4 Algorithm for the DFS Graph Traversal Method

DFS(adj[], status[], stack[], N)

Step 1: Start

Step 2: Set status[] = 1

Step 3: Push(stack, v1)

Step 4: Set status[v1]=2

Step 5: Repeat Step 6-11 while stack[] is not empty

Step 6: $V = \text{Pop}(\text{stack})$

Step 7: status[V]=3

Step 8: Repeat Step 9-11 while adj(V) is not empty

Step 9: If adj(V) = 1 goto step 10 else goto step 8

Step 10: Push(stack, adj(V))

Step 11: Set adj[v]=2

Step 12: Stop

Here, the pop operation signifies the processing of a graph node.



Check Point

1. What is BFS?

Ans. It is the method of traversing a graph in such a manner that all the vertices at a particular level are visited first before proceeding onto the next level.

2. What is DFS?

Ans. It is the method of traversing a graph in such a manner that all the vertices in a given path (starting from the first node) are visited first before proceeding onto the next path.

SUMMARY

- ☐ A graph $G(V, E)$ consists of the following elements:
 - ☐ A set V of vertices or nodes where $V = \{v_1, v_2, v_3, \dots, v_n\}$
 - ☐ A set E of edges also called arcs where $E = \{e_1, e_2, e_3, \dots, e_n\}$
- ☐ A graph can be implemented in three ways: adjacency matrix, path matrix, and adjacency list.
- ☐ Adjacency matrix and path matrix are the sequential methods of representing a graph. Adjacency matrix signifies whether there is an edge between any two vertices of the graph. Path matrix signifies whether there is a path between any two vertices of the graph.

- ☐ Adjacency list is a linked representation of a graph. It consists of a list of graph nodes with each node itself consisting of a linked list of its neighboring nodes.
- ☐ Breadth First Search or BFS is the method of traversing a graph in such a manner that all the vertices at a particular level are visited first before proceeding onto the next level.
- ☐ Depth First Search or DFS is the method of traversing a graph in such a manner that all the vertices in a given path (starting from the first node) are visited first before proceeding onto the next path.

Key Terms

Weighted graph | Directed | Adjacency matrix | Path matrix | Adjacency list

Review Questions

- 16.1 What is a graph? Explain with an example.
- 16.2 List and explain any five key terms associated with graphs.
- 16.3 What are the different methods of representing a graph?
- 16.4 What is an adjacency matrix? How can you derive a path matrix from an adjacency matrix?
- 16.5 Explain adjacency list implementation of a graph with the help of an example.
- 16.6 What is the significance of computing the shortest path in a graph? Explain with the help of an example.
- 16.7 Write the modified Warshall's algorithm for computing the shortest path between two nodes of a graph.
- 16.8 What is BFS? Explain with the help of an example.
- 16.9 What is DFS? Explain with the help of an example.

Programming Exercises

- 16.1 Write a C function to deduce the adjacency matrix for a given directed graph G.
- 16.2 Write a C function that takes as input the adjacency matrix and applies Warshall's algorithm to generate the corresponding path matrix.
- 16.3 Write a C program to implement a 3-node directed graph using adjacency list.
- 16.4 Write a C function that takes as input the path matrix and applies the shortest path algorithm to generate the corresponding shortest path matrix.

UNIT 5

Sorting and Searching

➤ **Chapter 17:** Sorting and Searching

SORTING AND SEARCHING

- 17.1 Introduction
- 17.2 Sorting Techniques
 - 17.2.1 Selection Sort
 - 17.2.2 Insertion Sort
 - 17.2.3 Bubble Sort
 - 17.2.4 Quick Sort
 - 17.2.5 Merge Sort
 - 17.2.6 Bucket Sort
- 17.3 Searching Techniques
 - 17.3.1 Linear Search
 - 17.3.2 Binary Search
 - 17.3.3 Hashing

Summary

Key Terms





17.1 INTRODUCTION

Sorting and searching are two of the most common operations performed by computers all around the world. The sorting operation arranges the numerical and alphabetical data present in a list, in a specific order or sequence. Searching, on the other hand, locates a specific element across a given list of elements. At times, a list may require sorting before the search operation can be performed on it.

A telephone directory is one such example where both sorting and searching techniques are applied. The names of telephone subscribers are first alphabetically sorted and then posted on to the telephone directory. If one needs to search the telephone number of a particular subscriber in the telephone directory then it can be easily achieved by looking up the directory on the basis of the subscriber name. Now, consider the same scenario in the absence of a sorted list of subscribers. It would become very tough and painstaking to search the subscriber name in a directory where names are posted in a random fashion without any definite order.

There are a number of sorting techniques that can be employed to sort a given list of data elements. The suitability of a specific technique in a specific situation depends on a number of factors, such as

1. size of the data structure,
2. algorithm efficiency, and
3. programmer's knowledge of the technique.

While all the sorting methods produce the same result, that is a list of sorted elements, it is one or more of the above factors that play an important role in choosing a specific sorting technique in a given situation.

In this chapter, we will discuss the various searching and sorting methods.



17.2 SORTING TECHNIQUES

Consider a list L containing n elements, as shown below.

$$L_1, L_2, L_3, \dots, L_n$$

Now, there are $n!$ ways in which the elements can be arranged within the list. We can apply a sorting technique to the list L to arrange the elements in either ascending or descending order.

If we sort the list in ascending order, then

$$L_1 \leq L_2 \leq L_3 \dots \leq L_n$$

Alternatively, if we arrange the list in descending order, then

$$L_1 \geq L_2 \geq L_3 \dots \geq L_n$$

Example 17.1 Consider an Array A containing Five Elements, as shown below

	22	77	3	-1	5	
	A[0]	A[1]	A[2]	A[3]	A[4]	

What would be the resultant array if it is sorted in
ascending order
descending order

Solution Array A sorted in ascending order.

	-1	3	5	22	77	
	A[0]	A[1]	A[2]	A[3]	A[4]	

Array A sorted in descending order.

	77	22	5	3	-1	
	A[0]	A[1]	A[2]	A[3]	A[4]	

As already explained, there are a number of methods that can be used to sort a given list of elements. We will discuss the following sorting methods in the forthcoming sections:

1. Selection sort
2. Insertion sort
3. Bubble sort
4. Quick sort
5. Merge sort
6. Bucket sort



Mind Jog

What is the internal sorting?

All sorting techniques which require the data set to be present in the main memory are referred as internal sorting techniques.



Note The application of a sorting technique is not restricted to an array or a list alone. In fact, we may apply sorting to other data structures such as structures or linked lists provided there is a subelement in the data structure based on which sorting can be performed.

17.2.1 Selection Sort

Selection sort is one of the most basic sorting techniques. It works on the principle of identifying the smallest element in the list and moving it to the beginning of the list. This process is repeated until all the elements in the list are sorted.

Let us consider an example where a list L contains five integers stored in a random fashion, as shown in Fig. 17.1.

18	3	2	33	21
List L				

Fig. 17.1 List of integers

Now, if the list L is sorted using selection sort technique then first of all the first element in the list, i.e., 18 will be selected and compared with all the remaining elements in the list. The element which is found to be the lowest amongst the remaining set of elements will be swapped with the first element. Then, the second element will be selected and compared with the remaining elements in the list. This process is repeated until all the elements are rearranged in a sorted manner. Table 17.1 illustrates the sorting of list L in ascending order using selection sort.

Table 17.1 Selection sort

Pass	Comparison	Resultant Array
1		
2		
3		
4		
→ denotes the currently selected element → denotes the smallest element identified in the current pass		

A single iteration of the selection sorting technique that brings the smallest element at the beginning of the list is called a pass. As we can see in the above table, four passes were required to sort a list of five elements. Hence, we can say that selection sort requires $n-1$ passes to sort an array of n elements.

Example 17.2 Algorithm to Perform Selection Sort on a Given Array of Integers

selection(arr[], size)

Step 1: Start

Step 2: Set $i = 0$, $loc = 0$ and $temp = 0$

Step 3: Repeat Steps 4-6 while $i < size$

Step 4: Set $loc = \text{Min}(\text{arr}, i, size)$

Step 5: Swap the elements stored at $\text{arr}[i]$ and $\text{a}[loc]$ by performing the following steps

I Set $temp = a[loc]$

II Set $a[loc] = a[i]$

III Set $a[i] = temp$

Step 6: Set $i = i + 1$

Step 8: Stop

Min(array[], LB, UB)

Step 1: Start

Step 2: Set $m = LB$

Step 3: Repeat Steps 4-6 while $LB < UB$

Step 4: if $\text{array}[LB] < \text{array}[m]$ goto Step 5 else goto Step 6

Step 5: Set $m = LB$

Step 6: Set $LB = LB + 1$

Step 7: Return m

Step 8: Stop

Program 17.1 implements selection sorting technique in C. It uses the algorithm depicted in Example 17.2.

Program 17.1 Selection Sort

```

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

using namespace std;

void selection(int*, int); // Function Prototype for performing Selection Sort
int Min(int*, int, int); // Function Prototype for finding Minimum Element
in the Array

int main()
{
    int *arr;
    inti, N;

    system("cls");

    cout<< "\nEnter the number of elements in the Array : ";
    cin>> N;

    arr = (int*) malloc (sizeof(int)); //Dynamic allocation of
memory for the array

    cout<< "\nEnter the " << N << " elements to Sort :\n";
    for(i = 0; i< N; i++)
        cin>>arr[i]; //Reading Array Elements

    selection(arr,N); //Calling the Selection function

    cout<< "\n The Sorted Elements are :\n\n";
    for(i = 0; i< N; i++)
        cout<<arr[i] <<endl; //Printing Insertion Sorted Array

    getch();
}

void selection(int *a, int size)
{
    inti, loc, temp;
    i = loc = temp = 0;
    for(i = 0; i< size; i++)
    {
        loc = Min(a, i, size); // Calling the Min Function

        //Swapping the Array Elements
        temp = a[loc];
        a[loc] = a[i];
        a[i] = temp;
    }
}

```

(Contd.)

```

    }
}

int Min(int *array, int LB, int UB)
{
    int m = LB;

    //Finding the Smallest Element

    while(LB < UB)
    {
        if(array[LB] < array [m])
            m = LB;

        LB++;
    }

    return(m);
}

```

Output

Enter the number of elements in the Array : 5

Enter the 5 elements to Sort :

18
3
2
33
21

The Sorted Elements are :

2
3
18
21
33

Efficiency of Selection Sort Assume that an array containing n elements is sorted using selection sort technique.

Now, the number of comparisons made during first pass = $n - 1$

Number of comparisons made during second pass = $n - 2$

Number of comparisons made during last pass = 1

So, total number of comparisons = $(n - 1) + (n - 2) + \dots + 1$
 $= n * (n - 1) / 2$
 $= O(n^2)$

Thus, efficiency of selection sort = $O(n^2)$

Advantages and Disadvantages Some of the key advantages of selection sorting technique are:

1. It is one of the simplest of sorting techniques.
2. It is easy to understand and implement.
3. It performs well in case of smaller lists.
4. It does not require additional memory space to perform sorting.

The disadvantages associated with selection sort that prevent the programmers from using it often are as follows:

1. The efficiency of $O(n^2)$ is not well suited for large sized lists.
2. It does not leverage the presence of any existing sort pattern in the list.



Note

Selection sort is an internal sorting technique and requires the entire data structure to be present in the main memory while performing sorting. As a result, it is not well suited for sorting large sized data structures.

17.2.2 Insertion Sort

As the name suggests, insertion sort method sorts a list of elements by inserting each successive element in the previously sorted sublist. Such insertion of elements requires the other elements to be shuffled as required.

To understand the insertion sorting method, consider a scenario where an array A containing n elements needs to be sorted. Now, each pass of the insertion sorting method will insert the element $A[j]$ into its appropriate position in the previously sorted subarray, i.e., $A[1], A[2], \dots, A[j-1]$. The following list describes the tasks performed in each of the passes:

Pass 1 $A[2]$ is compared with $A[1]$ and inserted either before or after $A[1]$. This makes $A[1], A[2]$ a sorted sub array.

Pass 2 $A[3]$ is compared with both $A[1]$ and $A[2]$ and inserted at an appropriate place. This makes $A[1], A[2], A[3]$ a sorted sub array.

Pass $n-1$ $A[n]$ is compared with each element in the sub-array $A[1], A[2], A[3], \dots, A[n-1]$ and inserted at an appropriate position. This eventually makes the entire array A sorted.

Let us revisit the list L containing five integers stored in a random fashion, as shown in Fig. 17.1.

Now, if the list L is sorted using insertion sort technique then first of all the second element in the list, i.e., 3 will be selected and compared with the first element, i.e., 18. Since 3 is less than 18, the two elements will be interchanged. This process is repeated until all the elements are rearranged in a sorted manner. Table 17.2 illustrates the sorting of list L in ascending order using insertion sort technique.



Check Point

1. How many passes are required by the selection sort technique to sort an array of N elements?

Ans. $N-1$

2. What is the most significant disadvantage of selection sort?

Ans. One of the most critical disadvantages of selection sort is that its efficiency of $O(n^2)$ does not make it suitable for large sized lists.

Table 17.2 Insertion sort

Pass	Comparison	Resultant Array
1		
2		
3		
4		
→ denotes the previously sorted sub array → denotes the current selection		

As we can see in the above illustration, four passes were required to sort a list of five elements. Hence, we can say that insertion sort requires $n-1$ passes to sort an array of n elements.

Example 17.3 Algorithm to Perform Insertion Sort on a Given Array of Integers

```
insertion(arr[], size)
```

```
Step 1: Start
```

```
Step 2: Set i = 1, j = 0 and temp = 0
```

```
Step 3: Repeat Steps 4-12 while i < size
```

```
Step 4: Set temp = arr[i]
```

```
Step 5: Set j = i-1
```

```
Step 6: Repeat Steps 7-10 while j >= 0
```

```
Step 7: if arr[j] > temp goto Step 8 else goto Step 9
```

```
Step 8: Set arr[j+1] = arr[j]
```

```
Step 9: Branch out and go to Step 11
```

```
Step 10: Set j = j-1
```

```
Step 11: Set arr[j+1] = temp
```

```
Step 12: Set i = i + 1
```

```
Step 13: Stop
```

Program 17.2 implements insertion sorting technique in C. It uses the algorithm depicted in Example 17.3.

Program 17.2 Insertion Sort

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

using namespace std;

void insertion(int [], int); // Function Prototype for performing Insertion Sort

int main()
{
    int *arr;
```

(Contd.)

```

    inti, N;
    system("cls");
    cout<< "\nEnter the number of elements in the Array : ";
    cin>> N;

    arr = (int*) malloc (sizeof(int)); //Dynamic allocation of
    memory for the array

    cout<< "\nEnter the " << N << " elements to Sort :\n";
    for(i = 0; i< N; i++)
        cin>>arr[i]; //Reading Array Elements

    insertion(arr,N); //Calling the insertion function

    cout<< "\n The Sorted Elements are :\n\n";
    for(i = 0; i< N; i++)
        cout<<arr[i] <<endl; //Printing Insertion Sorted Array

    getch();
}

void insertion(int array[], int size)
{
    inti, j, temp;

    for(i = 0; i< size; i++)
    {
        temp = array[i]; //Selecting the Next Element to be Inserted
        //Inserting the element in the previously sorted Sub-Array
        for(j = i-1; j >= 0; j--)
        {
            if(array[j] > temp)
                array[j+1] = array[j];
            else
                break;
        }
        array[j+1] = temp;
    }
}

```

Output

```

Enter the number of elements in the Array : 5
Enter the 5 elements to Sort :
18
3
2
33
21

The Sorted Elements are :
2
3
18
21
33

```

Efficiency of Insertion Sort Assume that an array containing n elements is sorted using insertion sort technique.

The minimum number of elements that must be scanned = $n - 1$

For each of the elements the maximum number of shifts possible = $n - 1$

Thus, efficiency of insertion sort = $O(n^2)$

Advantages and Disadvantages Some of the key advantages of insertion sorting technique are:

1. It is one of the simplest sorting techniques that is easy to implement.
2. It performs well in case of smaller lists.
3. It leverages the presence of any existing sort pattern in the list, thus resulting in better efficiency.

The disadvantages associated with insertion sorting technique are as follows.

1. The efficiency of $O(n^2)$ is not well suited for large sized lists.
2. It requires large number of elements to be shifted.



Tip

Insertion sorting technique should not be used with lists containing lengthy records as the worst case of $O(n^2)$ may result in inefficient performance.

17.2.3 Bubble Sort

Bubble sort is one of the oldest and simplest of sorting techniques. It focuses on bringing the largest element to the end of the list with each successive pass. Unlike selection sort, it does not perform a search to identify the largest element; instead it repeatedly compares two consecutive elements and moves the largest amongst them to the right. This process is repeated for all pairs of elements until the current iteration moves the largest element to the end of the list.

To understand the bubble sorting method, consider a scenario where an array A containing n elements needs to be sorted. In the first pass, elements $A[1]$ and $A[2]$ are compared and if $A[1]$ is larger than $A[2]$ then the two values are swapped. Next, $A[2]$ and $A[3]$ are compared. The last comparison of the first pass between $A[n - 1]$ and $A[n]$ brings the largest element of the list to the end. The second pass repeats this process for the remaining $n - 1$ elements. Finally, the last pass compares only the first two elements i.e., $A[1]$ and $A[2]$ to generate the sorted list.

Let us revisit the list L containing five integers stored in a random fashion, as shown in Fig. 17.1.

Table 17.3 illustrates the sorting of list L in ascending order using bubble sort:



Check Point

1. What is the efficiency of insertion sort?

Ans. The efficiency of insertion sort is $O(n^2)$.

2. What is the advantage of using insertion sort?

Ans. The advantage of using insertion sort technique is that it is easy to implement and it performs well for small sized lists.

Table 17.3 *Bubble sort*

Pass	Comparison	Resultant Array
1		
2		
3		
4		

As we can see in the above illustration, four passes were required to sort a list of five elements. Hence, we can say that bubble sort requires $n - 1$ passes to sort an array of n elements.

Example 17.4 Algorithm to Perform Bubble Sort on a Given Array of Integers

```
bubble(arr[], size)
```

Step 1: Start

Step 2: Set $i = \text{size}$, $j = 0$ and $\text{temp} = 0$

Step 3: Repeat Steps 4-9 while $i > 1$

Step 4: Set $j = 0$

Step 5: Repeat Steps 6-8 while $j < i-1$

Step 6: if $\text{arr}[j] > \text{arr}[j+1]$ goto Step 7 else goto Step 8

Step 7: Swap the elements stored at $\text{arr}[j]$ and $\text{arr}[j+1]$ by performing the following steps

I Set $\text{temp} = \text{arr}[j+1]$

II Set $\text{arr}[j+1] = \text{arr}[j]$

III Set $\text{arr}[j] = \text{temp}$

Step 8: Set $j = j + 1$

Step 9: Set $i = i - 1$

Step 10: Stop

Program 17.3 implements bubble sorting technique in C. It uses the algorithm depicted in Example 17.4.

Program 17.3 Bubble Sort

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

using namespace std;

void bubble(int [], int); // Function Prototype for performing Bubble Sort

int main()
{
    int *arr;
    int i, N;

    system("cls");

    cout<< "\nEnter the number of elements in the Array : ";
    cin>> N;

    arr = (int*) malloc (sizeof(int)); //Dynamic allocation of
    memory for the array

    cout<< "\nEnter the " << N << " elements to Sort :\n";
    for(i = 0; i< N; i++)
        cin>>arr[i]; //Reading Array Elements

    bubble(arr,N); //Calling the bubble function

    cout<< "\n The Sorted Elements are :\n\n";
    for(i = 0; i< N; i++)
        cout<<arr[i] <<endl; //Printing Insertion Sorted Array

    getch();
}

void bubble(int array[], int size)
{
    int i, j, temp;

    for(i = size; i> 0; i--)
    {
        for(j = 0; j < i-1; j++)
        {
            if(array[j] > array[j+1])
            {
                //Swapping adjacent Elements
```

(Contd.)

```

        temp = array[j+1];
        array[j+1] = array[j];
        array[j] = temp;
    }
}
}

```

Output

Enter the number of elements in the Array : 5

Enter the 5 elements to Sort :

18
3
2
33
21

The Sorted Elements are :

2
3
18
21
33

Efficiency of Bubble Sort Assume that an array containing n elements is sorted using bubble sort technique.

Number of comparisons made in first pass = $n - 1$

Number of comparisons made in second pass = $n - 2$

Number of comparisons made in last pass = 1

Total number of comparisons made = $(n - 1) + (n - 2) + \dots + 1$
 $= n * (n - 1) / 2$
 $= O(n^2)$

Thus, efficiency of bubble sort = $O(n^2)$

Advantages and Disadvantages Some of the key advantages of bubble sorting technique are:

1. It is easy to understand and implement.
2. It leverages the presence of any existing sort pattern in the list, thus resulting in better efficiency.

The disadvantages associated with bubble sorting technique are given below.

1. The efficiency of $O(n^2)$ is not well suited for large sized lists.
2. It requires large number of elements to be shifted.
3. It is slow in execution as large elements are moved towards the end of the list in a step-by-step fashion.



Note

Bubble sort leverages any existing sort pattern in a list quite well. Its best case efficiency on an already sorted list is $O(n)$ which is better than a number of other sorting techniques.

17.2.4 Quick Sort

As the name suggests, quick sort is one of the fastest sorting methods that is based on divide and conquer strategy. It divides the given list into a number of sub lists and then works on each of the sub lists to obtain the sorted output. It first chooses one of the list elements as a key value and then tries to place the key value at its final position in the list. Once, the key value is positioned correctly, the two sub lists to the left and right of the key value are processed in the similar fashion until the entire list becomes sorted.



Check Point

1. Why is the bubble sorting technique slow in execution?

Ans. Bubble sorting technique is considered as slow because it moves the elements to the end of the list in a step-by-step fashion.

2. How many passes are required by bubble sort to sort an array of N elements?

Ans. $N - 1$



Note

The key value is also called as pivot element.

Consider an array containing six elements, as shown below.

34 99 5 2 57 40

Initially, the first list element i.e., 34 is chosen as the pivot element. Now, the list is scanned from right to left to identify the first element that is less than 34. This element is 2. So, both the elements are swapped and the list becomes:

2 99 5 34 57 40

Now, the list is scanned from left to right till the place where 34 is stored and the control stops at the first element that is greater than 34. This element is 99. So, both the elements are swapped and the list becomes:

2 34 5 99 57 40

Now, the list is again scanned from right to left starting with element 99 and ending at element 34. Element 5 is found to be lesser than 34, thus both the elements are swapped. Now, the list becomes:

2 5 34 99 57 40

Now, there are no elements present between 5 and 34, thus we can assume that 34 has attained its final position in the list.

Now, the two sublists to the left and right of the pivot element are identified, as shown below:

2 5 (34) 99 57 40

Now, each of these lists is processed in the same fashion and eventually all the elements are placed at appropriate positions in the final sorted list.

Example 17.5 Algorithm to Perform Quick Sort on a given Array of Integers

quick(arr[], LB, UB)

Step 1: Start

Step 2: Set pivot=0, nxt_pvt=0, left=LB, right=UB

Step 3: Set pivot = arr[left] to select the first element as the pivot element

Step 4: Repeat Steps 5-14 while LB < UB

(Contd.)

```

Step 5: Repeat Step 6 while arr[UB] >= pivot and LB < UB
Step 6: Set UB = UB - 1
Step 7: if LB is not equal to UB goto Step 8 else goto Step 10
Step 8: Set arr[LB]=arr[UB]
Step 9: Set LB = LB + 1
Step 10: Repeat Step 11 while arr[LB] <= pivot and LB < UB
Step 11: Set LB = LB + 1
Step 12: if LB is not equal to UB goto Step 13 else goto Step 15
Step 13: Set arr[UB]=arr[LB]
Step 14: Set UB = UB - 1
Step 15: Set arr[LB]= pivot
Step 16: Set nxt_pvt = LB
Step 17: Set LB = left and UB = right
Step 18: if LB < nxt_pvt goto Step 19 else goto Step 20
Step 19: Apply quick sort in the left sub list by calling module quick(arr,
LB, nxt_pvt-1)
Step 20: if UB > nxt_pvt goto Step 21 else goto Step 22
Step 21: Apply quick sort in the right sub list by calling module quick(arr,
nxt_pvt+1, UB)
Step 22: Stop

```

Program 17.4 implements quick sorting technique in C. It uses the algorithm depicted in Example 17.5.

Program 17.4 Quick Sort

```

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

using namespace std;

void quick(int[], int, int); // Function Prototype for performing Insertion Sort

int main()
{
    int *arr;
    inti, N;

    system("cls");

    cout<< "\nEnter the number of elements in the Array : ";
    cin>> N;

    arr = (int*) malloc (sizeof(int)); //Dynamic allocation of memory for
    the array

    cout<< "\nEnter the " << N << " elements to Sort :\n";
    for(i = 0; i< N; i++)
        cin>>arr[i]; //Reading Array Elements

```

(Contd.)


```

    quick(arr, 0, N-1); //Calling the Quick Sort function

    cout<< "\n The Sorted Elements are :\n\n";
    for(i = 0; i< N; i++)
        cout<<arr[i] <<endl; //Printing Quick Sorted Array

    getch();
}

void quick(int array[], int LB, int UB)
{
    int pivot, nxt_pvt, left, right;
    left = LB;
    right = UB;
    pivot = array[left];

    while(LB < UB)
    {
        // Scanning the list from left to right to identify the element
        // lesser than the pivot element

        while((array[UB] >= pivot) && (LB < UB))
            UB--;

        if(LB != UB)
        {
            array[LB] = array[UB]; //Shuffling the Pivot Element
            LB++;
        }

        // Scanning the list from left to right to identify the element
        // greater than the pivot element

        while((array[LB] <= pivot) && (LB < UB))
            LB++;

        if(LB != UB)
        {
            array[UB] = array[LB]; //Shuffling the Pivot Element
            UB--;
        }
    }

    array[LB] = pivot;
    nxt_pvt = LB;
    LB = left;
    UB = right;

    if(LB < nxt_pvt)

```

(Contd.)

```

        quick(array, LB, nxt_pvt-1);
    if(UB >nxt_pvt)
        quick(array, nxt_pvt+1, UB);

}

```

Output

Enter the number of elements in the Array : 6

Enter the 6 elements to Sort :

34
99
5
2
57
40

The Sorted Elements are :

2
5
34
40
57
99

Efficiency of Quick Sort Assume that an array containing n elements is to be sorted using quick sort technique. Let us analyze the efficiency of quick sort in best case and worst case scenarios.

Best Case In the best case, the pivot element always divides the list in to two equal halves. Here, we are assuming that the number of elements in the list is a power of 2. That means, $n = 2^m$ or $m = \log_2 n$

Number of comparisons made in first pass = n

Number of comparisons made in second pass = $2*(n / 2)$

Number of comparisons made in the third pass = $4*(n / 4)$

Number of comparisons made in the fourth pass = $8*(n / 8)$

Number of comparisons made in the k^{th} pass = $k*(n / k)$

Now, total number of comparisons = $O(n) + O(n) + O(n) + \dots + m \text{ terms}$
 $= O(n * m)$
 $= O(n \log n)$

Thus, efficiency of quick sort in best case scenario = $O(n \log n)$

Worst Case It may happen that the pivot element divides the lists in unequal partitions. In the worst case, there would be no element in one of the lists while the other list will contain all the elements.

In such a case, number of comparisons made in first pass = $n - 1$

Number of comparisons made in second pass = $n - 2$

Number of comparisons made in last pass = 1

$$\begin{aligned}
 \text{Total number of comparisons} &= (n-1) + (n-2) + \dots + 1 \\
 &= n * (n-1) / 2 \\
 &= O(n^2)
 \end{aligned}$$

Thus, efficiency of quick sort in worst case scenario = $O(n^2)$

Advantages and Disadvantages Some of the key advantages of quick sorting technique are:

1. It is one of the fastest sorting algorithms.
2. Its implementation does not require any additional memory.

The disadvantages associated with quick sorting technique are as follows.

1. The worst case efficiency of $O(n^2)$ is not well suited for large sized lists.
2. Its algorithm is considered as a little more complex in comparison to some other sorting techniques.



Note

The choice of the pivot element may have a direct impact on the performance of the quick sort algorithm, considering that there could be some pre-existing sort order present in the input list. As a result, different implementations of the quick sorting technique use first, last, middle or at times some randomly chosen element as the pivot element.

17.2.5 Merge Sort

Merge sort is another sorting technique that is based on divide-and-conquer approach. It divides a list into several sub lists of equal sizes and sorts them individually. It then merges the various sub lists in pairs to eventually form the original list, while ensuring that the sort order is not disturbed.

Consider a list L containing n elements on which merge sort is to be performed. Initially, the n elements of the list L are considered as n different sublists of one element each. Since, a list having one element is sorted in itself, thus there is no further action required on these sublists. Now, each of the sublists is merged in pairs to form $n/2$ sublists having two elements each. While merging two lists the elements are compared and placed in a sorted fashion in the new sublist. This process is repeated until the original list is formed with elements arranged in a sorted fashion.

Consider an array containing six elements, as shown below:

34 99 5 2 57 40 8 29

Figure 17.2 shows how merge sort is performed on the above list.

As we can see in the above illustration, the sorted sublists are progressively merged in each pass to eventually generate the original list, sorted in ascending order.



Check Point

1. What is the best case efficiency of quick sort?

Ans. $O(n \log n)$.

2. What is a pivot element?

Ans. It is a key value that is selected and shuffled continuously as per the quick sort algorithm until it attains its final position in the list.

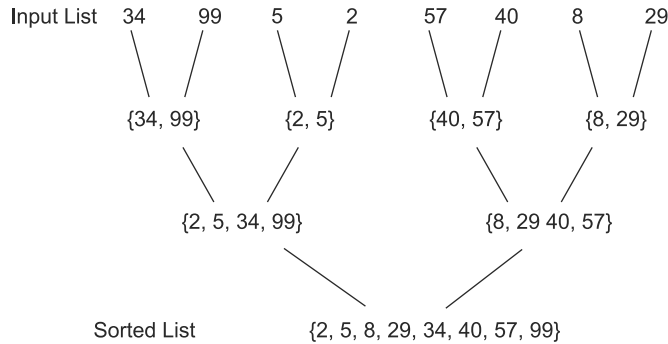


Fig. 17.2 Merge sort

Example 17.5 Algorithm to Perform Merge Sort on a Given Array of Integers

```
mergesort(arr[], size)
```

```
Step 1: Start
```

```
Step 2: Set mid = 0
```

```
Step 3: if size = 1 goto Step 4 else goto Step 5
```

```
Step 4: Stop and return back to the calling module
```

```
Step 5: Set mid = size / 2
```

```
Step 6: Call module mergesort(arr, mid)
```

```
Step 7: Call module mergesort(arr+mid, size-mid)
```

```
Step 8: Call module merge(arr, mid, arr+mid, size-mid)
```

```
Step 9: Stop
```

```
merge(a[], size1, b[], size2)
```

```
Step 1: Start
```

```
Step 2: Initialize a temporary array, temp_array[size1+size2]
```

```
Step 3: Set i=0, j=0, k=0
```

```
Step 4: Repeat Step 5-9 while i < size1 and j < size2
```

```
Step 5: If a[i] < b[j] goto Step 6 else goto Step 8
```

```
Step 6: Set temp_array[k] = a[i]
```

```
Step 7: Set k = k + 1 and i = i + 1
```

```
Step 8: Set temp_array[k] = b[j]
```

```
Step 9: Set k = k + 1 and j = j + 1
```

```
Step 10: Repeat Step 11-12 while i < size1
```

```
Step 11: Set temp_array[k] = a[i]
```

```
Step 12: Set k = k + 1 and i = i + 1
```

```
Step 13: Repeat Step 14-15 while j < size2
```

```
Step 14: Set temp_array[k] = b[j]
```

```
Step 15: Set k = k + 1 and j = j + 1
```

```
Step 16: Set a[] = temp_array[]
```

```
Step 17: Stop
```

Program 17.5 implements merge sorting technique in C. It uses the algorithm depicted in Example 17.6.

Program 17.5 Merge Sort

```

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

using namespace std;

void mergesort(int*, int); // Function Prototype for performing Merge Sort
void merge(int*, int, int*, int); // Function Prototype for performing Merge Two Arrays

int main()
{
    int *arr;
    int i, N;

    system("cls");

    cout<< "\nEnter the number of elements in the Array : ";
    cin>> N;

    arr = (int*) malloc (sizeof(int)*N); //Dynamic allocation of memory
    for the array

    cout<< "\nEnter the " << N << " elements to Sort :\n";
    for(i = 0; i< N; i++)
        cin>>arr[i]; //Reading Array Elements

    mergesort(arr, N); //Calling the Merge Sort Function

    cout<< "\n The Sorted Elements are :\n\n";
    for(i = 0; i< N; i++)
        cout<<arr[i] <<endl; //Printing Sorted Array

    getch();
}

void mergesort(int *array, int size)
{
    int mid;
    if (size == 1)
        return;
    else
    {
        mid = size / 2;

        // Making Recursive Calls to Mergesort Function
        mergesort(array, mid);
        mergesort(array+mid, size-mid);
        merge(array, mid, array+mid, size-mid); // Calling Merge Function
    }
}

void merge(int *a, int s1, int *b, int s2)

```

(Contd.)

```
{
    inti, j, k, *temp_arr;

    temp_arr = (int*) malloc((s2+s1) * sizeof(int)); //Dynamic allocation
    of memory for the array
    i = j = k = 0;

    while(i< s1 && j < s2)
    {
        if(a[i] < b[j])
            temp_arr[k++] = a[i++];
        else
            temp_arr[k++] = b[j++];
    }

    while(i< s1)
        temp_arr[k++] = a[i++];

    while(j < s2)
        temp_arr[k++] = b[j++];

    for(i = 0; i< k; i++)
        a[i] = temp_arr[i];

    free(temp_arr);
}
```

Output

Enter the number of elements in the Array : 8

Enter the 8 elements to Sort :

34
99
5
2
57
40
8
29

The Sorted Elements are :

2
5
8
29
34
40
57
99

Efficiency of Merge Sort Assume that an array containing n elements is sorted using merge sort technique. As we have already seen, merge sort is divided into two submodules. One module keeps on dividing the list until n different sublists of one element each are generated. Alternatively, the other module keeps on appending pairs of sublists until the main list with n elements is generated.

Now, the number of steps required for dividing the list = $\log_2 n$

The number of steps required for merging the lists = $\log_2 n$

Total number of steps = $2\log_2 n$

Now, in each step all n list elements are compared.

Thus, total number of comparisons made = $n \cdot 2\log_2 n$

$$= 2n\log_2 n$$

$$= O(n\log_2 n)$$

Thus, efficiency of merge sort = $O(n\log_2 n)$

Advantages and Disadvantages Some of the key advantages of merge sorting technique are:

1. It is a fast and stable sorting method.
2. It always ensures an efficiency of $O(n\log n)$.

The disadvantages associated with merge sorting technique are as follows.

1. It requires additional memory space to perform sorting. The size of the additional space is in direct proportion to the size of the input list.
2. Even though the number of comparisons made by merge sort are nearly optimal, its performance is slightly lesser than that of quick sort.



Check Point

1. What is the efficiency of merge sort?

Ans. $O(n\log n)$.

2. What is the most significant advantage of merge sort?

Ans. It always ensures an efficiency of $O(n\log n)$.

17.2.6 Bucket Sort

Bucket sort distributes the list of elements across different buckets in such a way that any bucket m contains elements greater than the elements of bucket $m-1$ but less than the elements of bucket $m+1$. The elements within each bucket are sorted individually either by using some alternate sorting technique or by recursively applying bucket sort technique. In the end, elements of all the buckets are merged to generate the sorted list. This technique is particularly effective for smaller range of data series.

Consider a list containing ten integers stored in a random fashion, as shown in Fig. 17.3.

2	27	13	18	21	43	42	39	31	4

Fig. 17.3 List of integers

In the above list, all elements are between the range of 0 to 50. So, let us create five buckets for storing ten elements each. Figure 17.4 shows how these buckets are used for sorting the list.

As we can see in the above illustration, the list elements are first distributed as per their values across different buckets. Then, each of the buckets are individually sorted and later merged to generate the original sorted list.



Mind Jog

What is a stable sort?

It is that sorting methodology which preserves the original order of the duplicate values in the final sorted list.

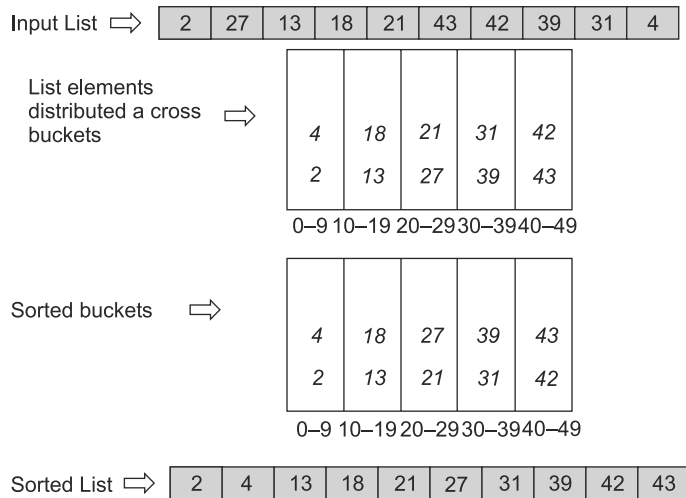


Fig. 17.4 Bucket sort

Example 17.7 Algorithm to Perform Bucket Sort on a Given Array of Integers

Assumption: The input list elements are within the range of 0 to 49.

bucket(arr[], size)

Step 1: Start

Step 2: Set $i = 0$, $j = 0$ and $k = 0$

Step 3: Initialize an array $c[5]$ and set all its values to 0; it keeps track of the number of elements in each of the five buckets

Step 4: Create five buckets by initializing a 2-D array $b[5][10]$ and set all its values to 0

Step 5: Now, distribute the input list elements across different buckets. To do this, repeat Steps 6-16 while $i < \text{size}$

Step 6: if $0 \leq \text{arr}[i] \leq 9$ then goto Step 7 else goto Step 8

Step 7: Set $b[0][c[0]] = \text{arr}[i]$ and $c[0] = c[0] + 1$

Step 8: if $10 \leq \text{arr}[i] \leq 19$ then goto Step 9 else goto Step 10

Step 9: Set $b[1][c[1]] = \text{arr}[i]$ and $c[1] = c[1] + 1$

Step 10: if $20 \leq \text{arr}[i] \leq 29$ then goto Step 11 else goto Step 12

Step 11: Set $b[2][c[2]] = \text{arr}[i]$ and $c[2] = c[2] + 1$

Step 12: if $30 \leq \text{arr}[i] \leq 39$ then goto Step 13 else goto Step 14

Step 13: Set $b[3][c[3]] = \text{arr}[i]$ and $c[3] = c[3] + 1$

Step 14: if $40 \leq \text{arr}[i] \leq 49$ then goto Step 15 else goto Step 16

Step 15: Set $b[4][c[4]] = \text{arr}[i]$ and $c[4] = c[4] + 1$

Step 16: Set $i = i + 1$

Step 17: Sort each of the buckets $b[i][j]$ by calling insertion sort module `insertion(&b[i][j], c[i])`

Step 18: Merge all the buckets together into the main array by setting `array[j] = b[i][j]`

Step 19: Stop

Program 17.6 implements bucket sorting technique in C. It uses the algorithms depicted in Example 17.4 and Example 17.7.

Program 17.6 Bucket Sort

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

using namespace std;

void insertion(int *, int); // Function Prototype for performing Insertion Sort
void bucket(int *, int); // Function prototype for performing Bucket Sort

int main()
{
    int *arr;
    inti, N;

    system("cls");

    cout<< "\nEnter the number of elements in the Array : ";
    cin>> N;

    arr = (int*) malloc (sizeof(int)); //Dynamic allocation of
    memory for the array

    cout<< "\nEnter the " << N << " elements to Sort :\n";
    for(i = 0; i< N; i++)
        cin>>arr[i]; //Reading Array Elements

    bucket(arr,N); //Calling the insertion function

    cout<< "\n The Sorted Elements are :\n\n";
    for(i = 0; i< N; i++)
        cout<<arr[i] <<endl; //Printing Insertion Sorted Array
    getch();
}

// Insertion Sort Function for Sorting Elements in a Bucket

void insertion(int *array, int size)
{
    inti, j, temp;
    i = j = temp = 0;

    for(i = 0; i< size; i++)
    {
        temp = array[i]; //Selecting the Next Element to be Inserted

        //Inserting the element in the previously sorted Sub-Array
```

(Contd.)

```

        for(j = i-1; j >= 0; j--)
        {
            if(array[j] > temp)
                array[j+1] = array[j];
            else
                break;
        }

        array[j+1] = temp;
    }
}

void bucket(int *array, int size)
{
    inti, j, k, b[5][10];
    int c[5];

    for(i=0; i<5; i++)
        c[i] = 0;

    // Distributing Elements across Different Buckets

    for(i=0; i<size; i++)
    {
        if(array[i] >= 0 && array[i] <=9)
            b[0][c[0]++] = array[i];
        if(array[i] >= 10 && array[i] <=19)
            b[1][c[1]++] = array[i];
        if(array[i] >= 20 && array[i] <=29)
            b[2][c[2]++] = array[i];
        if(array[i] >= 30 && array[i] <=39)
            b[3][c[3]++] = array[i];
        if(array[i] >= 40 && array[i] <=49)
            b[4][c[4]++] = array[i];
    }

    // Sorting elements in each Bucket using Insertion Sort
    for(i=0; i<5; i++)

    {
        if(c[i] != 0)
            insertion(&b[i][0], c[i]); //Calling Insert Function
    }

    // Merging Buckets to form the Original List

    i = 0;
    k = 0;

```

(Contd.)

```

while(i< 5)
{
    if(c[i] == 0)
    {
        i = i + 1;
        continue;
    }

    for(j=0; j<c[i]; j++)
        array[k++] = b[i][j];
    i = i + 1;
}
}

```

Output

Enter the number of elements in the Array : 10

Enter the 10 elements to Sort :

2
27
13
18
21
43
42
39
31
4

The Sorted Elements are :

2
4
13
18
21
27
31
39
42
43



Note

The bucket sort algorithm is particularly suited for lists having elements within a specific range. The above program is based on the assumption that the elements in the input list are within the range of 0 to 49.

Efficiency of Bucket Sorting Assume that an array containing n elements is sorted using bucket sort technique.

In the worst case, all elements of the list will be placed in a single bucket.

Now, each bucket is sorted using insertion sort, whose efficiency = $O(n^2)$

Thus, worst case efficiency of bucket sort = $O(n^2)$

Advantages and Disadvantages Some of the key advantages of bucket sorting technique are:

1. It preserves the order of repetitive values in the list.
2. It performs well for large size lists having elements in a smaller range.

The disadvantages of bucket sort are as follows:

1. It works only if the range of input values is fixed.
2. It requires additional space to perform the sorting operation.



17.3 SEARCHING TECHNIQUES

Searching refers to determining whether an element is present in a given list of elements or not. If the element is found to be present in the list then the search is considered as successful, otherwise it is considered as an unsuccessful search. The search operation returns the location or address of the element found.

There are various searching methods that can be employed to perform search on a data set. The choice of a particular searching method in a given situation depends on a number of factors, such as

1. order of elements in the list, i.e., random or sorted
2. size of the list

Let us explore the various searching methods one by one.

17.3.1 Linear Search

It is one of the conventional searching techniques that sequentially searches for an element in the list. It typically starts with the first element in the list and moves towards the end in a step-by-step fashion. In each iteration, it compares the element to be searched with the list element, and if there is a match, the location of the list element is returned.

Consider an array of integers A containing n elements. Let k be the value that needs to be searched. The linear search technique will first compare $A[0]$ with k to find out if they are same. If the two values are found to be same then the index value, i.e., 0 will be returned as the location containing k . However, if the two values are not same then k will be compared with $A[1]$. This process will be repeated until the element is not found. If the last comparison between k and $A[n-1]$ is also negative then the search will be considered as unsuccessful.

Figure 17.5 depicts the linear search technique performed on an array of integers.



Check Point

1. What is the worst case efficiency of bucket sort?

Ans. $O(n^2)$.

2. What is the distinctive advantage of using bucket sort technique?

Ans. It preserves the order of duplicate elements in the final sorted list.

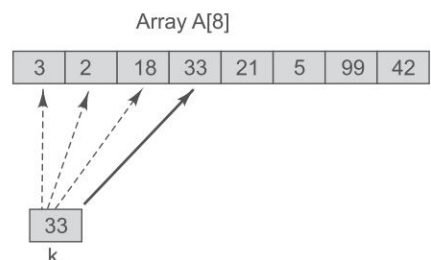


Fig. 17.5 Linear search

As shown in Fig. 17.5, the value k is repeatedly compared with each element of the array A . As soon as the element is found, the corresponding index location is returned and the search operation is terminated.

Example 17.8 Algorithm to Perform Linear Search on a Given Array of Integers

```
linear(arr[], size, k)
Step 1: Start
Step 2: Set i = 0
Step 3: Repeat Steps 4-6 while i < size
Step 4: if k = arr[i] goto Step 5 else goto Step 6
Step 5: Return i and goto Step 9
Step 6: Set i = i + 1
Step 7: If i = size goto Step 8 else goto Step 9
Step 8: Return NULL and goto Step 9
Step 9: Stop
```

Program 17.7 implements linear search technique in C. It uses the algorithm depicted in Example 17.8.

Program 17.7 Linear Search

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

using namespace std;

int linear(int [], int, int); // Function Prototype for performing Binary Search

int main()
{
    int *arr;
    inti, N, k, index;

    system("cls");

    cout<< "\nEnter the number of elements in the Array : ";
    cin>> N;

    arr = (int*) malloc (sizeof(int)); //Dynamic allocation of
    memory for the array

    cout<< "\nEnter the " << N << " elements of the Array :\n";
    for(i = 0; i< N; i++)
        cin>>arr[i]; //Reading Array Elements

    cout<< "\n\nEnter the Element to be Searched by Linear Search : ";
    cin>> k;

    index = linear(arr, N, k); //Calling the Linear function

    //Printing Search Results

    if(index == -999)
```

(Contd.)

```

        cout<< "\n Element " << k << " is not present in the Array\n";
    else
        cout<< "\n Element " << k << " is stored at the index
        location " << index << " of the Array";

    getch();
}

int linear(int array[], int size, intnum)
{
    inti;
    for(i = 0; i< size; i++) // Scanning the Array Elements one by one
    {
        if(num == array[i])
            return (i); // Successful Search
        if(i == size)
            return(-999); // Unsuccessful Search
    }
}

```

Output

```

Enter the number of elements in the Array : 8
Enter the 8 elements of the Array :
3
2
18
33
21
5
99
42

Enter the Element to be Searched by Linear Search : 33
Element 33 is stored at the index location 3 of the Array

```

Efficiency of Linear Search Assume that an array containing n elements is to be searched for the value k . In the best case, k would be the first element in the list, thus requiring only one comparison. In the worst case, it would be last element in the list, thus requiring n comparisons.

To compute the efficiency of linear search we can add all the possible number of comparisons and divide it by n .

$$\begin{aligned}
 \text{Thus, efficiency of linear search} &= (1 + 2 + \dots + n) / n \\
 &= n(n+1) / 2n \\
 &= O(n)
 \end{aligned}$$

Advantages and Disadvantages Some of the key advantages of linear search technique are:

1. It is a simple searching technique that is easy to implement.
2. It does not require the list to be sorted in a particular order.

**Check Point**

1. What is the efficiency of linear search?

Ans. $O(n)$.

2. What is the key advantage of linear search?

Ans. It does not require the list to be sorted in a particular order.

The disadvantages associated with it are as follows:

1. It is quite inefficient for large sized lists.
2. It does not leverage the presence of any pre-existing sort order in a list.

17.3.2 Binary Search

Binary search technique has a prerequisite – it requires the elements of a data structure (list) to be already arranged in a sorted manner before search can be performed in it. It begins by comparing the element that is present at the middle of the list. If there is a match then the search ends immediately and the location of the middle element is returned. However, if there is a mismatch then it focuses the search either in the left or the right sub list depending on whether the target element is lesser than or greater than middle element. The same methodology is repeatedly followed until the target element is found.

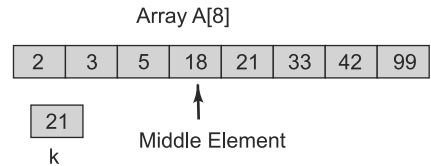


Fig. 17.6 Binary search

Binary search follows the same analogy as that of a telephone directory that we had discussed earlier. One needs to keep focusing on a smaller subset of directory pages every time there is a mismatch. However, such a search would not have been possible had the directory entries were not already sorted.

Consider an array of integers A containing eight elements, as shown in Fig. 17.6. Let $k = 21$ be the value that needs to be searched.

As we can see in Fig. 17.6, the array A on which binary search is to be performed is already sorted. The following steps describe how binary search is performed on array A to search for value k :

1. First of all, the middle element in the array A is identified, which is 18.
2. Now, k is compared with 18. Since k is greater than 18, the search is focused on the right sub list.
3. The middle element in the right sub list is 33. Since k is less than 33, the search is focused on the left sublist, which is {21, 33}.
4. Now, again k is compared with the middle element of {21, 33}, which is 21. Thus, it matches with k .
5. The index value of 21, i.e., 4 is returned and the search is considered as successful.

Example 17.9 Algorithm to Perform Binary Search on a Given Array of Integers

```
binary(arr[], size, num)
```

```
Step 1: Start
```

```
Step 2: Set  $i = 0$ ,  $j = \text{size}$ ,  $k = 0$ 
```

```
Step 3: Repeat Steps 4-9 while  $i \leq j$ 
```

```
Step 4: Set  $k = (i + j) / 2$ 
```

```
Step 5: If  $\text{arr}[k] = \text{num}$  goto Step 6 else goto Step 7
```

```
Step 6: return  $k$  and goto Step 11
```

```
Step 7: If  $\text{arr}[k] < \text{num}$  goto Step 8 else goto Step 9
```

```
Step 8:  $i = k + 1$ 
```

```
Step 9:  $j = k - 1$ 
```

```
Step 10: Return NULL and goto Step 11
```

```
Step 11: Stop
```

Program 17.8 implements binary search technique in C. It uses the algorithm depicted in Example 17.9.

Program 17.8 Binary Search

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

using namespace std;

int binary(int [], int, int); // Function Prototype for performing Binary Search

int main()
{
    int *arr;
    inti, N, k, index;

    system("cls");

    cout<< "\nEnter the number of elements in the Array : ";
    cin>> N;

    arr = (int*) malloc (sizeof(int)); //Dynamic allocation of
    memory for the array

    cout<< "\nEnter the " << N << " elements of the Array in the
    Sorted Format :\n";
    for(i = 0; i< N; i++)
        cin>>arr[i]; //Reading Array Elements

    cout<< "\n\n Enter the Element to be Searched by Binary Search
    : ";
    cin>> k;

    index = binary(arr, N, k); //Calling the Binary function

    //Printing Search Results

    if(index == -999)
        cout<< "\n Element " << k << " is not present in the Array\n";
    else
        cout<< "\n Element " << k << " is stored at the index
        location " << index << " of the Array";

    getch();
}

int binary(int array[], int size, intnum)
{
    inti, j, k;
    i = 0;
    j = size;
```

(Contd.)


```

while(i<= j)
{
    k = (i + j) / 2;
    if(array[k] == num)
        return (k); // Successful Search
    else if (array[k] < num)
        i = k + 1;
    else
        j = k- 1;
}
return(-999); // Unsuccessful Search
}

```

Output

```

Enter the number of elements in the Array : 8
Enter the 8 elements of the Array in the Sorted Format :
2
3
5
18
21
33
42
99

Enter the Element to be Searched by Binary Search : 21
Element 21 is stored at the index location 4 of the Array

```

Efficiency of Binary Search

Best Case The best case for a binary search algorithm occurs when the element to be searched is present at the middle of the list. In this case, only one comparison is made to perform the search operation.

Thus, efficiency = $O(1)$

Worst Case The worst case for a binary search algorithm occurs when the element to be searched is not present in the list. In this case, the list is continuously divided until only one element is left for comparison.

Let n be the number of list elements and c be the total number of comparisons made in the worst case.

Now, after every single comparison, the number of list elements left to be searched is reduced by 2.

Thus, $c = \log_2 n$

Hence, efficiency = $O(\log_2 n)$

Advantages and Disadvantages Some of the key advantages of binary search technique are:

1. It requires lesser number of iterations.
2. It is a lot faster than linear search.

The disadvantages associated with it are as follows:

1. Unlike linear search, it requires the list to be sorted before search can be performed.
2. In comparison to linear search, the binary search technique may seem to be a little difficult to implement.

17.3.3 Hashing

So far we have learnt two of the most fundamental searching techniques, i.e., linear and binary search. Both these searching techniques find their usage across varied programming situations. However, in case of large databases, an altogether different searching technique is widely used. This technique is called hashing.

Hashing finds the location of an element in a data structure without making any comparisons. In contrast to the other comparison-based searching techniques, like linear and binary search, hashing uses a mathematical function to determine the location of an element. This mathematical function called *hash function* accepts a value, known as *key*, as input and generates an output known as *hash key*. The hash function generates hash keys and stores elements corresponding to each hash key in the hash table. The keys that hash function accepts as input could be a digit associated with the input elements. In other words, we can say that a hash table is a data structure, which is implemented by a hash function and used for searching elements in quick time. In a hash table, hash keys act as the addresses of the elements. Figure 17.7 depicts the hash functionality.

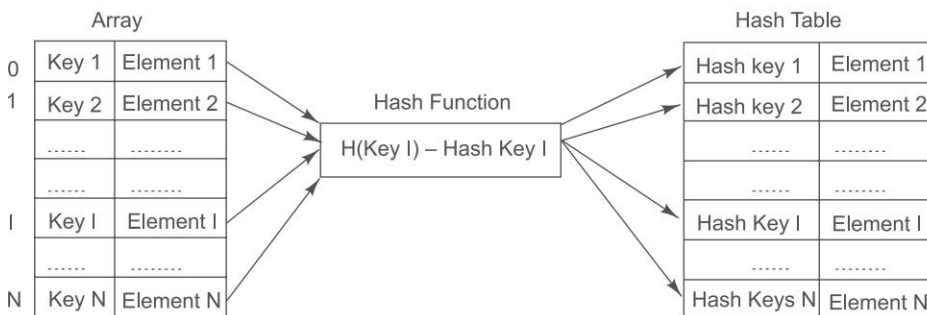


Fig. 17.7 Hashing

Let us consider a simple example of a file containing information for five employees of a company. Each record in that file contains the name and a three digit numeric Employee ID of the employee. In this case, the hash function will implement a hash table of five slots using Employee IDs as the keys. That means, the hash function will take Employee IDs as input and generate the hash keys, as shown in Fig. 17.8.

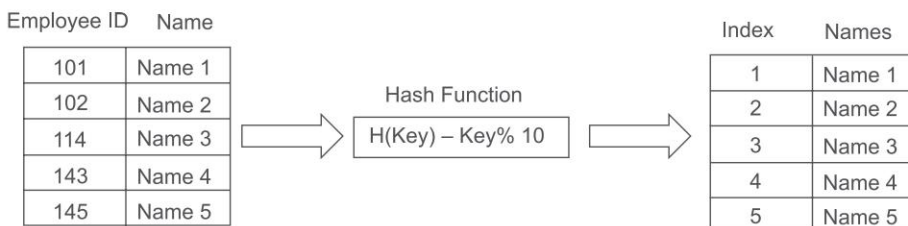


Fig. 17.8 Generating hash keys



Check Point

1. What is the worst case efficiency of binary search?

Ans. $O(\log 2n)$.

2. What is the key prerequisite for performing binary search?

Ans. The key prerequisite for performing binary search is that the input list must already be sorted.

In the hash table generated in the above example, the hash function is *Employee ID%17*. Thus, for Employee ID 101, hash key will be calculated as 1. Therefore, Name1 will be stored at position 1 in the hash table. For Employee ID 102, hash key will be 2, hence Name2 will be stored at position 2 in the hash table. Similarly, Name3, Name4, and Name5 will be stored at position 4, 3, and 5 respectively, as shown in Fig. 17.5. Later, whenever an employee record is searched using the Employee ID, the hash function will indicate the exact position of that record in the hash table.

Collision As we have already learnt, the hash function takes some key values as input, performs some mathematical calculation, and generates hash key to ascertain the position in the hash table where the record corresponding to the key will be stored. However, it is quite possible that the hash function generates same hash keys for two different key values. That means, two different records are indicated to be stored at the same position in the hash table. This situation is termed as *collision*. As a result, a hash function must be designed in such a way that the possibility of a collision is negligible. Various techniques such as, linear probing, chaining without replacement, and chaining with replacement are used to evade the chances of a collision.

Perfect Hashing Perfect hashing ensures that there is no possibility of collision occurrence. It can be achieved only when the set of input keys are known beforehand. As a result, collision prevention measures are programmatically included while developing the hash function.

Program 17.9 Hash Function

```
#include<iostream>
#include<conio.h>
using namespace std;
int hash(int); // Function Prototype for generating Hash Keys
int main()
{
    int key, hk;
    system("cls");
    cout<< "Enter the next Key : ";
    cin>> key;

    hk = hash(key);

    cout<< "\n\nThe Hash key generated for the key " << key << " is " <<hk;
    getch();
}

int hash(int k)
{
    return(k-10000);
}
```

Output

Enter the next Key : 10765

The Hash key generated for the key 10765 is 765

SOLVED PROBLEMS

Problem 17.1 Consider the following array of integers:

35 18 7 12 5 23 16 3 1

Create a snapshot of the above array at each pass if the bubble sorting technique is applied on it.

Solution

Initial array	35	18	7	12	5	23	16	3	1
Pass 1	1	35	18	12	7	23	16	5	3
Pass 2	1	3	35	18	12	23	16	7	5
Pass 3	1	3	5	35	18	23	16	12	7
Pass 4	1	3	5	7	35	23	18	16	12
Pass 5	1	3	5	7	12	35	23	18	16
Pass 6	1	3	5	7	12	16	35	23	18
Pass 7	1	3	5	7	12	16	18	35	23
Pass 8	1	3	5	7	12	16	18	23	35 (sorted array)



Check Point

1. What is a hash table?

Ans. It is a data structure, which is implemented by a hash function and used for searching elements in quick time.

2. What is perfect hashing?

Ans. Perfect hashing reduces the possibility of collision occurrence to zero.

Problem 17.2 Consider the following array of integers:

74 39 35 32 97 84

Create a snapshot of the above array at each pass if the selection sorting technique is applied on it.

Solution

Initial array	74	39	35	32	97	84
Pass 1	32	39	35	74	97	84
Pass 2	32	35	39	74	97	84
Pass 3	32	35	39	74	97	84
Pass 4	32	35	39	74	97	84
Pass 5	32	35	39	74	84	97 (sorted array)

Problem 17.3 Consider the following array of integers:

35 54 12 18 23 15 45 38

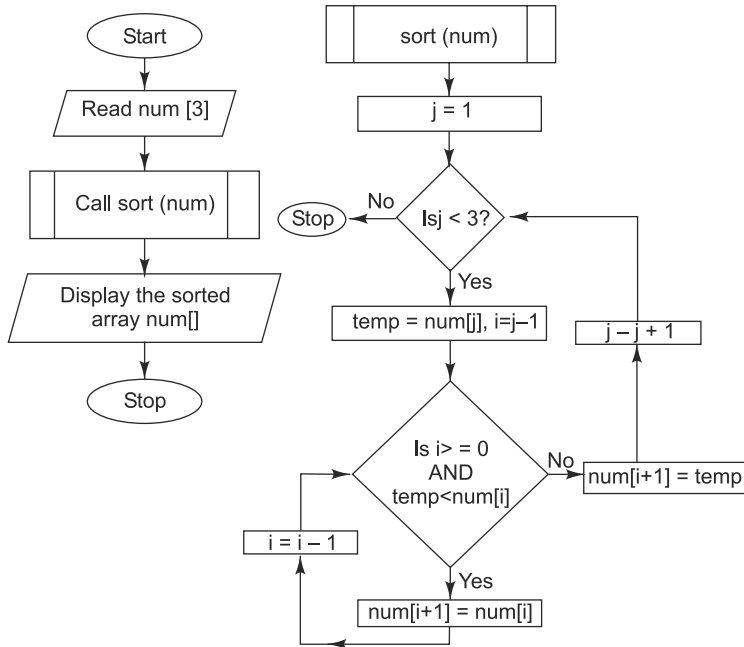
Create a snapshot of the above array at each pass if the quick sorting technique is applied on it.

Solution

Initial Array	35	54	12	18	23	15	45	38
Pass 1	18	54	12	35	23	15	45	38
Pass 2	18	15	12	35	23	54	45	38
Pass 3	12	15	18	35	23	54	45	38
Pass 4	12	15	18	35	23	54	45	38
Pass 5	12	15	18	35	23	54	45	38
Pass 6	12	15	18	54	23	35	45	38
Pass 7	15	18	38	23	35	45	54	
Pass 8	15	18	23	38	35	45	54	
Pass 9	15	18	23	38	35	45	54	
Pass 10	12	15	18	23	35	38	45	54
Pass 11	12	15	18	23	35	38	45	54
Pass 12	12	15	18	23	35	38	45	54
Pass 13	12	15	18	23	35	38	45	54 (sorted array)

Problem 17.4 Draw a flowchart for sorting three integers using insertion sort technique.

FLOWCHART



SUMMARY

- ☐ Sorting is the process of arranging the numerical and alphabetical data present in a list, in a specific order or sequence.
- ☐ Searching is the process of locating a specific element across a given list of elements.
- ☐ Selection sort works on the principle of identifying the smallest element in the list and moving it to the beginning of the list.
- ☐ Insertion sort method sorts a list of elements by inserting each successive element in the previously sorted sublist. The insertion of elements requires the other elements to be shuffled appropriately.
- ☐ Bubble sort works by bringing the largest element to the end of the list with each successive pass.
- ☐ Quick sort is one of the fastest sorting methods that is based on divide and conquer approach. It revolves around a key element called pivot to perform the sorting operation.
- ☐ Merge sort divides a list into several sublists of equal sizes and sorts them individually. The sorted sublists are later merged to form the original list.
- ☐ Bucket sort distributes the list of elements across different buckets and sorts each of the buckets individually. These buckets are later merged to form the original sorted list.
- ☐ Linear search is one of the conventional searching techniques that sequentially searches for an element in the list

(Contd.)

- ☐ Binary search works on an already sorted list to perform the search operation. It repetitively looks for the middle element of the list until the target element is found.
- ☐ Hashing finds the location of an element in a data structure without making any comparisons. It uses the hash function to determine the location of an element.
- ☐ Collision is a situation where the hash function generates same hash keys for two different key values.
- ☐ Perfect hashing ensures that there is no possibility of collision occurrence.

Key Terms

Pivot | Bucket | Hash | Hash table

Review Questions

- 17.1 What is sorting? List the various sorting techniques.
- 17.2 What is searching? List the various searching techniques.
- 17.3 Why quick sort is considered the fastest sorting technique?
- 17.4 What is a pivot element? Where is it used?
- 17.5 Deduce the worst case efficiency of binary search technique.
- 17.6 What is bubble sorting? What is it considered to be slow?
- 17.7 Explain the insertion and selection sorting techniques with examples.
- 17.8 What is hashing? Explain with example.
- 17.9 What is a collision? How can it be prevented?
- 17.10 Explain the role of buckets in bucket sorting technique.

Programming Exercises

- 17.1 Write a program in C that takes as input five integers and displays them in a sorted sequence.
- 17.2 Write a C function that applies the bubble sorting technique to sort a set of alphanumeric characters as per their ASCII values.
- 17.3 Write a C program that uses the insertion sorting technique to sort an array of structures. The sorting must be performed on the basis of one of the structure members.
- 17.4 Write a C function that performs linear search on an array of real values.
- 17.5 Write a C program that sorts the given set of integers and performs binary search on them.

EC 6301 OBJECT ORIENTED PROGRAMMING AND DATA STRUCTURES

MODEL QUESTION PAPER 1

PART A

(10 X 2 = 20 Marks)

1. Define function overloading.
2. What is call by reference?
3. Define multi-level inheritance.
4. Define 'this' pointer.
5. What are different types of linked lists?
6. Define a queue.
7. What is a binary tree?
8. Explain about adjacency matrix.
9. What is the advantage of quick sort?
10. What is sorting?

PART B

(5 X 16 = 80 Marks)

- 11 (a) (i) Explain the elements of object-oriented programming. (8)
(ii) What are the differences between reference variables and normal variables? (8)
OR
(b) Write a program to implement nested classes using C++. (16)
- 12 (a) Explain the various types of inheritance with suitable example programs. (16)
OR
(b) Discuss virtual function and polymorphism with examples. (16)
- 13 (a) Write a C ++ program to implement stack and its operations PUSH and POP. (16)
OR
(b) Define double-linked list. Explain the various operations of double-linked list with algorithms. (16)
- 14 (a) Write a program to convert the expression $((A + B) * C - (D - E) ^ (F + G))$ to equivalent prefix and postfix notations. (16)

OR

- (b) What are the graph traversal methods? Explain with example. (16)
- 15 (a) Explain the algorithm of quick sort by sorting the following set of numbers as an example: (16)
- 42 47 52 57 62 37 32 27 22
- OR
- (b) Describe the divide- and- conquer technique with the help of merge sort. (16)

MODEL QUESTION PAPER 2

PART A

(10 X 2 = 20 Marks)

1. What are the features of object oriented programming?
2. What are the input and output operators used in C++?
3. What are the types of inheritance?
4. What is a virtual function?
5. Define infix, prefix and postfix notations.
6. Define a stack.
7. What is meant by traversing?
8. What is a graph?
9. What are the two main classifications of sorting based on the source of data?
10. What is searching? List a few techniques.

PART B

(5 X 16 = 80 Marks)

- 11 (a) (i) Give the syntax and usage of the reserved word *inline* with two examples. (8)
- (ii) Explain the importance of constructors and destructors with examples. (8)
- OR
- (b) What is operator overloading? Overload the numerical operators '+' and '/' for complex numbers "addition" and "division" respectively. (16)
- 12 (a) What are abstract classes? Explain with an example program. (16)
- OR
- (b) (i) Explain the concept of inheritance by considering an example of "vehicle". (8)
- (ii) Explain the operators used for dynamic memory allocation with examples. (8)
- 13 (a) Explain the implementation of stack using a linked list. (16)
- OR
- (b) Discuss in detail the array implementation of queue. (16)
- 14 (a) Draw the binary search tree for the following input list: 60, 25, 75, 15, 50, 66, 33, 44. (16)
- Trace an algorithm to delete the nodes 25, 75, 44 from the tree.
- OR
- (b) Explain the shortest-path algorithm with an example. (16)
- 15 (a) Sort the sequence 3, 1, 4, 7, 5, 9, 2, 6, 5 using insertion sort. (16)
- OR
- (b) What is linear search? Write a program explaining the same. (16)