Data Structures c

Using

Second Edition

Reema Thareja

*Assistant Professor*

*Department of Computer Science*

*Shyama Prasad Mukherjee College for Women*

*University of Delhi*

3

Oxford University Press is a department of the University of Oxford. It furthers the University’s objective of excellence in research, scholarship, and education by publishing worldwide. Oxford is a registered trade mark of Oxford University Press in the UK and in certain other countries.

Published in India by

Oxford University Press

YMCA Library Building, 1 Jai Singh Road, New Delhi 110001, India © Oxford University Press 2011, 2014

The moral rights of the author/s have been asserted.

First Edition published in 2011

Second Edition published in 2014

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the prior permission in writing of Oxford University Press, or as expressly permitted by law, by licence, or under terms agreed with the appropriate reprographics rights organization. Enquiries concerning reproduction outside the scope of the above should be sent to the Rights Department, Oxford University Press, at the address above.

You must not circulate this work in any other form

and you must impose this same condition on any acquirer.

ISBN-13: 978-0-19-809930-7

ISBN-10: 0-19-809930-4

Typeset in Times New Roman

by Pee-Gee Graphics, New Delhi

Printed in India by Radha Press, New Delhi 110031

*I dedicate this book to my family and*

*my uncle Mr B.L. Thareja*

Preface to the First Edition

A data structure is defined as a group of data elements used for organizing and storing data. In order to be effective, data has to be organized in a manner that adds to the efficiency of an algorithm, and data structures such as stacks, queues, linked lists, heaps, and trees provide different capabilities to organize data.

While developing a program or an application, many developers find themselves more interested in the type of algorithm used rather than the type of data structure implemented. However, the choice of data structure used for a particular algorithm is always of the utmost importance. Each data structure has its own unique properties and is constructed to suit various kinds of applications. Some of them are highly specialized to carry out specific tasks. For example, B-trees with their unique ability to organize indexes are well-suited for the implementation of databases. Similarly, stack, a linear data structure which provides ‘last-in-first-out’ access, is used to store and track the sequence of web pages while we browse the Internet. Specific data structures are essential components of many efficient algorithms, and make possible the management of large amounts of data, such as large databases and Internet indexing services. C, as we all know, is the most popular programming language and is widespread among all the computer architectures. Therefore, it is not only logical but also fundamentally essential to start the introduction and implementation of various data structures through C. The course *data structures* is typically taught in the second or third semester of most engineering colleges and across most engineering disciplines in India. The aim of this course is to help students master the design and applications of various data structures and use them in writing effective programs.

**About the Book**

This book is aimed at serving as a textbook for undergraduate engineering students of computer science and postgraduate level courses of computer applications. The objective of this book is to introduce the concepts of data structures and apply these concepts in problem solving. The book provides a thorough and comprehensive coverage of the fundamentals of data structures and the principles of algorithm analysis. The main focus has been to explain the principles required to select or design the data structure that will best solve the problem.

A structured approach is followed to explain the process of problem solving. A theoretical description of the problem is followed by the underlying technique. These are then ably supported by an example followed by an algorithm, and finally the corresponding program in C language. The salient features of the book include:

∑ Explanation of the concepts using diagrams

∑ Numerous solved examples within the chapters

∑ Glossary of important terms at the end of each chapter

∑ Comprehensive exercises at the end of each chapter

∑ Practical implementation of the algorithms using tested C programs

∑ Objective type questions to enhance the analytical ability of the students

*Preface to the First Edition* **ix**

∑ Annexures to provide supplementary information to help generate further interest in the subject

The book is also useful as a reference and resource to young researchers working on efficient data storage and related applications, who will find it to be a helpful guide to the newly established techniques of a rapidly growing research field.

**Acknowledgements**

The writing of this textbook was a mammoth task for which a lot of help was required from many people. Fortunately, I have had the fine support of my family, friends, and fellow members of the teaching staff at the Institute of Information Technology and Management (IITM). My special thanks would always go to my father Mr Janak Raj Thareja and mother Mrs Usha Thareja, my brother Pallav and sisters Kimi and Rashi who were a source of abiding inspiration and divine blessings for me. I am especially thankful to my son Goransh who has been very patient and cooperative in letting me realize my dreams. My sincere thanks go to my uncle Mr B.L. Thareja for his inspiration and guidance in writing this book.

I would also like to thank my students and colleagues at IITM who had always been there to extend help while designing and testing the algorithms. Finally, I would like to thank the editorial team at Oxford University Press for their help and support.

Comments and suggestions for the improvement of the book are welcome. Please send them to me at reemathareja@gmail.com

**Reema Thareja**

Preface to the Second Edition

A data structure is the logical or mathematical arrangement of data in memory. It considers not only the physical layout of the data items in the memory but also the relationships between these data items and the operations that can be performed on these items. The choice of appropriate data structures and algorithms forms the fundamental step in the design of an efficient program. Thus, a thorough understanding of data structure concepts is essential for students who wish to work in the design and implementation of software systems. C, a general-purpose programming language, having gained popularity in both academia and industry serves as an excellent choice for learning data structures.

This second edition of *Data Structures Using C* has been developed to provide a comprehensive and consistent coverage of both the abstract concepts of data structures as well as the implementation of these concepts using C language. The book utilizes a systematic approach wherein the design of each of the data structures is followed by algorithms of different operations that can be performed on them, and the analysis of these algorithms in terms of their running times.

**New to the Second Edition**

Based on the suggestions from students and faculty members, this edition has been updated and revised to increase the clarity of presentation where required. Some of the prominent changes are as follows:

• New sections on omega and theta notations, multi-linked lists, forests, conversion of general trees into binary trees, 2-3 trees, binary heap implementation of priority queues, interpolation search, jump search, tree sort, bucket hashing, cylinder surface indexing

• Additional C programs on header linked lists, parentheses checking, evaluation of prefix expressions, priority queues, multiple queues, tree sort, file handling , address calculation sort

• New appendices on dynamic memory allocation, garbage collection, backtracking, Johnson’s problem

• Stacks and queues and multi-way search trees are now covered in separate chapters with a more comprehensive explanation of concepts and applications

**Extended Material**

*Chapter 1*—This chapter has been completely restructured and reorganized so that it now provides a brief recapitulation of C constructs and syntax. Functions and pointers which were included as independent chapters in the first edition have now been jointly included in this chapter. *Chapter 2*—New sections on primitive and non-primitive data structures, different approaches to designing algorithms, omega, theta, and little notations have been included. A number of new examples have also been added which show how to find the complexity of different functions. *Chapter 5*—This chapter now includes brief sections on unions, a data type similar to structures. *Chapter 6*—This chapter has been expanded to include topics on multi-linked lists, multi-linked list implementation of sparse matrices, and a C program on header linked lists.

**vi** *Preface to the Second Edition*

*Chapter 7*—New C programs on parenthesis checking and evaluation of prefix expressions have been added. Recursion, which is one of the most common applications of stacks, has been moved to this chapter.

*Chapter 8*—New C programs on priority queues and multiple queues have been included. *Chapter 9*—This chapter now includes sections on general trees, forests, conversion of general trees into binary trees, and constructing a binary tree from traversal results.

*Chapter 10*—An algorithm for in-order traversal of a threaded binary tree has been added. *Chapter 11*—A table summarizing the differences between B and B+ trees and a section on 2-3 trees have been included.

*Chapter 12*—A brief section on how binary heaps can be used to implement priority queues has been added.

*Chapter 13*—This chapter now includes a section which shows the adjacency multi-list representation of graphs.

*Chapter 14*—As a result of organization, the sections on linear and binary search have been moved from Chapter 3 to this chapter. New search techniques such as interpolation search, jump search, and Fibonacci search have also been included. The chapter also extends the concept of sorting by including sections on practical considerations for internal sorting, sorting on multiple keys, and tree sort.

*Chapter 15*—New sections on bucket hashing and rehashing have been included. *Chapter 16*—This chapter now includes a section on cylinder surface indexing which is one of the widely used indexing structures for files stored in hard disks.

**Content and Coverage**

This book is organized into 16 chapters.

*Chapter 1*, *Introduction to C* provides a review of basic C constructs which helps readers to familiarize themselves with basic C syntax and concepts that will be used to write programs in this book.

*Chapter 2, Introduction to Data Strctures and Algorithms* introduces data structures and algorithms which serve as building blocks for creating efficient programs. The chapter explains how to calculate the time complexity which is a key concept for evaluating the performance of algorithms.

From *Chapter 3* onwards, every chapter discusses individual data structures in detail. *Chapter 3, Arrays* provides a detailed explanation of arrays that includes one-dimensional, two dimensional, and multi-dimensional arrays. The operations that can be performed on such arrays are also explained.

*Chapter 4, Strings* discusses the concept of strings which are also known as character arrays. The chapter not only focuses on reading and writing strings but also explains various operations that can be used to manipulate the character arrays.

*Chapter 5, Structures and Unions* deals with structures and unions. A structure is a collection of related data items of different types which is used for implementing other data structures such as linked lists, trees, graphs, etc. We will also read about unions which is also a collection of variables of different data types, except that in case of unions, we can only store information in one field at any one time.

*Chapter 6, Linked Lists* discusses different types of linked lists such as singly linked lists, doubly linked lists, circular linked lists, doubly circular linked lists, header linked lists, and multi-linked lists. Linked list is a preferred data structure when it is required to allocate memory dynamically.

*Preface to the Second Edition* **vii**

*Chapter 7, Stacks* focuses on the concept of last-in, first-out (LIFO) data structure called stacks. The chapter also shows the practical implementation of these data structures using arrays as well as linked lists. It also shows how stacks can be used for the evaluation of arithmetic expressions. *Chapter 8, Queues* deals with the concept of first-in, first-out (FIFO) data structure called queues. The chapter also provides the real-world applications of queues.

*Chapter 9, Trees* focuses on binary trees, their traversal schemes and representation in memory. The chapter also discusses expression trees, tournament trees, and Huffman trees, all of which are variants of simple binary trees.

*Chapter 10,Efficient Binary Trees* broadens the discussion on trees taken up in *Chapter 9* by going one step ahead and discussing efficient binary trees. The chapter discusses binary search trees, threaded binary trees, AVL trees, red-black trees, and splay trees.

Chapter 11, *Multi-way Search Trees* explores trees which can have more than one key value in a single node, such as M-way search trees, B trees, B+ trees, tries, and 2-3 trees. *Chapter 12, Heaps* discusses three types of heaps—binary heaps, binomial heaps, and Fibonacci heaps. The chapter not only explains the operations on these data structures but also makes a comparison, thereby highlighting the key features of each structure.

*Chapter 13, Graphs* contains a detailed explanation of non-linear data structure called graphs. It discusses the memory representation, traversal schemes, and applications of graphs in the real world.

*Chapter 14, Searching and Sorting* covers two of the most common operations in computer science, i.e. searching and sorting a list of values. It gives the technique, complexity, and program for different searching and sorting techniques.

*Chapter 15, Hashing and Collision* deals with different methods of hashing and techniques to resolve collisions.

*Chapter 16,* the last chapter of the book, *Files and Their Organization*, discusses the concept related to file organization. It explains the different ways in which files can be organized on the hard disk and the indexing techniques that can be used for fast retrieval of data.

The book also provides a set of seven appendices.

Appendix A introduces the concept of dynamic memory allocation in C programs. Appendix B provides a brief discussion of garbage collection technique which is used for automatic memory management.

Appendix C explains backtracking which is a recursive algorithm that uses stacks. Appendix D discusses Johnson’s algorithm which is used in applications where an optimal order of execution of different activities has to be determined.

Appendix E includes two C programs which show how to read and write binary files. Appendix F includes a C program which shows how to sort a list of numbers using address calculation sort.

Appendix G provides chapter-wise answers to all the objective questions. **Reema Thareja**

Brief Contents

*Preface to the Second Edition v*

*Preface to the First Edition viii*

**1. Introduction to C 1 2. Introduction to Data Structures and Algorithms 43 3. Arrays 66 4. Strings 115 5. Structures and Unions 138 6. Linked Lists 162 7. Stacks 219 8. Queues 253 9. Trees 279 10. Efficient Binary Trees 298 11. Multi-way Search Trees 344 12. Heaps 361 13. Graphs 383 14. Searching and Sorting 424 15. Hashing and Collision 464 16. Files and Their Organization 489** *Appendix A: Memory Allocation in C Programs 505*

*Appendix B: Garbage Collection 512*

*Appendix C: Backtracking 514*

*Appendix D: Josephus Problem 516*

*Appendix E: File Handling in C 518*

*Appendix F: Address Calculation Sort 520*

*Appendix G: Answers 522*

*Index 528*

**xii** *Detailed Contents*

Detailed Contents

*Preface to the Second Edition v*

*Preface to the First Edition viii*

**1. Introduction to C 1** 1.1 Introduction *1*

1.2 Identifiers and Keywords *2*

1.3 Basic Data Types *2*

1.4 Variables and Constants *3*

1.5 Writing the First C Program *5*

1.6 Input and Output Functions *6*

1.7 Operators and Expressions *9*

1.8 Type Conversion and Typecasting *16* 1.9 Control Statements *17*

1.9.1 Decision Control Statements *17* 1.9.2 Iterative Statements *22*

1.9.3 Break and Continue Statements *27* 1.10 Functions *28*

1.10.1 Why are Functions Needed? *29* 1.10.2 Using Functions *29*

1.10.3 Passing Parameters to

Functions *31*

1.11 Pointers *34*

1.11.1 Declaring Pointer Variables *35* 1.11.2 Pointer Expressions and Pointer Arithmetic *36*

1.11.3 Null Pointers *36*

1.11.4 Generic Pointers *36*

1.11.5 Pointer to Pointers *37*

1.11.6 Drawback of Pointers *38*

**2. Introduction to Data Structures and Algorithms 43**

2.1 Basic Terminology *43*

2.1.1 Elementary Data Structure

Organization *45*

2.2 Classification of Data Structures *45* 2.3 Operations on Data Structures *49* 2.4 Abstract Data Type *50*

2.5 Algorithms *50*

2.6 Different Approaches to Designing an Algorithm *51*

2.7 Control Structures Used in Algorithms *52* 2.8 Time and Space Complexity *54* 2.8.1 Worst-case, Average-case,

Best-case, and Amortized

Time Complexity *54*

2.8.2 Time–Space Trade-off *55*

2.8.3 Expressing Time and Space

Complexity *55*

2.8.4 Algorithm Efficiency *55*

2.9 Big O Notation *57*

2.10 Omega Notation (Ω) *60*

2.11 Theta Notation (Θ) *61*

2.12 Other Useful Notations *62*

**3. Arrays 66** 3.1 Introduction *66*

3.2 Declaration of Arrays *67*

3.3 Accessing the Elements of an Array *68* 3.3.1 Calculating the Address of Array Elements *68*

3.3.2 Calculating the Length of an

Array *69*

3.4 Storing Values in Arrays *69*

3.5 Operations on Arrays *71*

3.5.1 Traversing an Array *71*

3.5.2 Inserting an Element in an

Array *76*

3.5.3 Deleting an Element from an

Array *79*

3.5.4 Merging Two Arrays *82*

3.6 Passing Arrays to Functions *86* 3.6.1 Passing Individual Elements *86* 3.6.2 Passing the Entire Array *87*

3.7 Pointers and Arrays *90*

3.8 Arrays of Pointers *92*

3.9 Two-dimensional Arrays *93*

3.9.1 Declaring Two-dimensional

Arrays *93*

3.9.2 Initializing Two-dimensional

Arrays *95*

3.9.3 Accessing the Elements of

Two-dimensional Arrays *96* 3.10 Operations on Two-Dimensional Arrays *99*

3.11 Passing Two-dimensional Arrays to Functions *103*

3.12 Pointers and Two-dimensional

Arrays *105*

3.13 Multi-dimensional Arrays *107* 3.14 Pointers and Three-dimensional Arrays *109*

3.15 Sparse Matrices *110*

3.16 Applications of Arrays *111*

**4. Strings 115** 4.1 Introduction *115*

4.1.1 Reading Strings *117*

4.1.2 Writing Strings *118*

4.2 Operations on Strings *118*

4.3 Arrays of Strings *129*

4.4 Pointers and Strings *132*

**5. Structures and Unions 138**

5.1 Introduction *138*

5.1.1 Structure Declaration *138*

5.1.2 Typedef Declarations *139*

5.1.3 Initialization of Structures *140*

*Detailed Contents* **xiii**

5.1.4 Accessing the Members of a

Structure *141*

5.1.5 Copying and Comparing

Structures *142*

5.2 Nested Structures *144*

5.3 Arrays of Structures *146*

5.4 Structures and Functions *148*

5.4.1 Passing Individual Members *149* 5.4.2 Passing the Entire Structure *149* 5.4.3 Passing Structures through

Pointers *152*

5.5 Self-referential Structures *155*

5.6 Unions *155*

5.6.1 Declaring a Union *156*

5.6.2 Accessing a Member of a

Union *156*

5.6.3 Initializing Unions *156*

5.7 Arrays of Union Variables *157*

5.8 Unions Inside Structures *158*

**6. Linked Lists 162** 6.1 Introduction *162*

6.1.1 Basic Terminologies *162*

6.1.2 Linked Lists versus Arrays *164* 6.1.3 Memory Allocation and

De-allocation for a Linked

List *165*

6.2 Singly Linked Lists *167*

6.2.1 Traversing a Linked List *167*

6.2.2 Searching for a Value in a

Linked List *167*

6.2.3 Inserting a New Node in a

Linked List *168*

6.2.4 Deleting a Node from a

Linked List *172*

6.3 Circular Linked Lists *180*

6.3.1 Inserting a New Node in a

Circular Linked List *181*

6.3.2 Deleting a Node from a

Circular Linked List *182*

6.4 Doubly Linked Lists *188*

6.4.1 Inserting a New Node in a

Doubly Linked List *188*

**xiv** *Detailed Contents*

6.4.2 Deleting a Node from a

Doubly Linked List *191*

6.5 Circular Doubly Linked Lists *199* 6.5.1 Inserting a New Node in a

Circular Doubly Linked List *200*

6.5.2 Deleting a Node from a Circular Doubly Linked List *201*

6.6 Header Linked Lists *207*

6.7 Multi-linked Lists *210*

6.8 Applications of Linked Lists *211* 6.8.1 Polynomial Representation *211*

**7. Stacks 219** 7.1 Introduction to Stacks *219*

7.2 Array Representation of Stacks *220* 7.3 Operations on a Stack *221*

7.3.1 Push Operation *221*

7.3.2 Pop Operation *221*

7.3.3 Peek Operation *222*

7.4 Linked Representation of Stacks *224* 7.5 Operations on a Linked Stack *224* 7.5.1 Push Operation *224*

7.5.2 Pop Operation *225*

7.6 Multiple Stacks *227*

7.7 Applications of Stacks *230*

7.7.1 Reversing a List *230*

7.7.2 Implementing Parentheses

Checker *231*

7.7.3 Evaluation of Arithmetic

Expressions *232*

7.7.4 Recursion *243*

**8. Queues 253** 8.1 Introduction to Queues *253*

8.2 Array Representation of Queues *254* 8.3 Linked Representation of Queues *256* 8.4 Types of Queues *260*

8.4.1 Circular Queues *260*

8.4.2 Deques *264*

8.4.3 Priority Queues *268*

8.4.4 Multiple Queues *272*

8.5 Applications of Queues *275*

**9. Trees 279** 9.1 Introduction *279*

9.1.1 Basic Terminology *279*

9.2 Types of Trees *280*

9.2.1 General Trees *280*

9.2.2 Forests *280*

9.2.3 Binary Trees *281*

9.2.4 Binary Search Trees *285*

9.2.5 Expression Trees *285*

9.2.6 Tournament Trees *286*

9.3 Creating a Binary Tree from a

General Tree *286*

9.4 Traversing a Binary Tree *287*

9.4.1 Pre-order Traversal *287*

9.4.2 In-order Traversal *288*

9.4.3 Post-order Traversal *289*

9.4.4 Level-order Traversal *289*

9.4.5 Constructing a Binary Tree from Traversal Results *290*

9.5 Huffman’s Tree *290*

9.6 Applications of Trees *294*

**10. Efficient Binary Trees 298** 10.1 Binary Search Trees *298*

10.2 Operations on Binary Search Trees *300* 10.2.1 Searching for a Node in a

Binary Search Tree *300*

10.2.2 Inserting a New Node in a Binary Search Tree *301*

10.2.3 Deleting a Node from a Binary Search Tree *301*

10.2.4 Determining the Height of a Binary Search Tree *303*

10.2.5 Determining the Number of

Nodes *303*

10.2.6 Finding the Mirror Image of

a Binary Search Tree *305*

10.2.8 Finding the Smallest Node in a Binary Search Tree *305*

10.2.9 Finding the Largest Node in a Binary Search Tree *306*

10.3 Threaded Binary Trees *311*

10.3.1 Traversing a Threaded Binary Tree *314*

10.4 AVL Trees *316*

10.4.1 Operations on AVL Trees *317* Searching for a Node in an AVL Tree *317*

10.5 Red-Black Trees *327*

10.5.1 Properties of Red-Black Trees *328* 10.5.2 Operations on Red-Black

Trees *330*

10.5.3 Applications of Red-Black

Trees *337*

10.6 Splay Trees *337*

10.6.1 Operations on Splay Trees *338* 10.6.2 Advantages and Disadvantages of Splay Trees *340*

**11. Multi-way Search Trees 344** 11.1 Introduction to M-Way Search Trees *344* 11.2 B Trees *345*

11.2.1 Searching for an Element in a B Tree *346*

11.2.2 Inserting a New Element in a B Tree *346*

11.2.3 Deleting an Element from a B Tree *347*

11.2.4 Applications of B Trees *350*

11.3 B+ Trees *351*

11.3.1 Inserting a New Element in a B+ Tree *352*

11.3.2 Deleting an Element from a B+ Tree *352*

11.4 2-3 Trees *353*

11.4.1 Searching for an Element in a 2-3 Tree *354*

11.4.2 Inserting a New Element in a

2-3 Tree *354*

11.4.3 Deleting an Element from a

2-3 Tree *356*

11.5 Trie *358*

**12. Heaps 361** 12.1 Binary Heaps *361*

12.1.1 Inserting a New Element in a

Binary Heap *362*

*Detailed Contents* **xv**

12.1.2 Deleting an Element from a Binary Heap *364*

12.1.3 Applications of Binary Heaps *364* 12.2 Binomial Heaps *365*

12.2.1 Linked Representation of

Binomial Heaps *366*

12.2.2 Operations on Binomial

Heaps *366*

12.3 Fibonacci Heaps *373*

12.3.1 Structure of Fibonacci Heaps *373* 12.3.2 Operations on Fibonacci

Heaps *374*

12.4 Comparison of Binary, Binomial, and Fibonacci Heaps *379*

12.5 Applications of Heaps *379*

**13. Graphs 383** 13.1 Introduction *383*

13.2 Graph Terminology *384*

13.3 Directed Graphs *385*

13.3.1 Terminology of a Directed

Graph *385*

13.3.2 Transitive Closure of a Directed Graph *386*

13.4 Bi-connected Components *387* 13.5 Representation of Graphs *388*

13.5.1 Adjacency Matrix

Representation *388*

13.5.2 Adjacency List Representation *390* 13.5.3 Adjacency Multi-List

Representation *391*

13.6 Graph Traversal Algorithms *393* 13.6.1 Breadth-First Search

Algorithm *394*

13.6.2 Depth-first Search Algorithm *397* 13.7 Topological Sorting *400*

13.8 Shortest Path Algorithms *405*

13.8.1 Minimum Spanning Trees *405* 13.8.2 Prim’s Algorithm *407*

13.8.3 Kruskal’s Algorithm *409*

13.8.4 Dijkstra’s Algorithm *413*

13.8.5 Warshall’s Algorithm *414*

**xvi** *Detailed Contents*

13.8.6 Modified Warshall’s

Algorithm *417*

13.9 Applications of Graphs *419*

**14. Searching and Sorting 424**

14.1 Introduction to Searching *424*

14.2 Linear Search *424*

14.3 Binary Search *426*

14.4 Interpolation Search *428*

14.5 Jump Search *430*

14.6 Introduction to Sorting *433*

14.6.1 Sorting on Multiple Keys *433* 14.6.2 Practical Considerations for

Internal Sorting *434*

14.7 Bubble Sort *434*

14.8 Insertion Sort *438*

14.9 Selection Sort *440*

14.10 Merge Sort *443*

14.11 Quick Sort *446*

14.12 Radix Sort *450*

14.13 Heap Sort *454*

14.14 Shell Sort *456*

14.15 Tree Sort *458*

14.16 Comparison of Sorting Algorithms *460* 14.17 External Sorting *460*

**15. Hashing and Collision 464** 15.1 Introduction *464*

15.2 Hash Tables *465*

15.3 Hash Functions *466*

15.4 Different Hash Functions *467*

15.4.1 Division Method *467*

*Appendix A: Memory Allocation in C Programs 505*

*Appendix B: Garbage Collection 512 Appendix C: Backtracking 514*

*Appendix D: Josephus Problem 516 Appendix E: File Handling in C 518 Appendix F: Address Calculation Sort 520 Appendix G: Answers 522*

*Index 528*

15.4.2 Multiplication Method *467*

15.4.3 Mid-Square Method *468*

15.4.4 Folding Method *468*

15.5 Collisions *469*

15.5.1 Collision Resolution by Open Addressing *469*

15.5.2 Collision Resolution By

Chaining *481*

15.6 Pros and Cons of Hashing *485* 15.7 Applications of Hashing *485*

Real World Applications of Hashing *486*

**16. Files and Their**

**Organization 489** 16.1 Introduction *489*

16.2 Data Hierarchy *489*

16.3 File Attributes *490*

16.4 Text and Binary Files *491*

16.5 Basic File Operations *492*

16.6 File Organization *493*

16.6.1 Sequential Organization *493* 16.6.2 Relative File Organization *494* 16.6.3 Indexed Sequential File

Organization *495*

16.7 Indexing *496*

16.7.1 Ordered Indices *496*

16.7.2 Dense and Sparse Indices *497* 16.7.3 Cylinder Surface Indexing *497* 16.7.4 Multi-level Indices *498*

16.7.5 Inverted Indices *499*

16.7.6 B-Tree Indices *500*

16.7.7 Hashed Indices *501*

chapter 1

Introduction to C

**Learning Objective**

This book deals with the study of data structures through C. Before going into a

detailed analysis of data structures, it would be useful to familiarize ourselves with

the basic knowledge of programming in C. Therefore, in this chapter we will learn

about the various constructs of C such as identifiers and keywords, data types,

constants, variables, input and output functions, operators, control statements,

functions, and pointers.

**1.1 INTRODUCTION**

The programming language ‘C’ was developed in the early 1970s by Dennis Ritchie at Bell Laboratories. Although C was initially developed for writing system software, today it has become such a popular language that a variety of software programs are written using this language. The greatest advantage of using C for programming is that it can be easily used on different types of computers. Many other programming languages such as C++ and Java are also based on C which means that you will be able to learn them easily in the future. Today, C is widely used with the UNIX operating system.

***Structure of a C program***

A C program contains one or more functions, where a function is defined as a group of statements that perform a well-defined task. Figure 1.1 shows the structure of a C program. The statements in a function are written in a logical sequence to perform a specific task. The main() function is the most important function and is a part of every C program. Rather, the execution of a C program begins with this function.

From the structure given in Fig. 1.1, we can conclude that a C program can have any number of functions depending on the tasks that have to be performed, and each function can have any number

**2** *Data Structures Using C*

main()

{

Statement 1;

Statement 2;

............

............

Statement N;

}

Function1()

{

Statement 1;

Statement 2;

............

............

Statement N;

}

Function2()

{

Statement 1;

Statement 2;

............

............

Statement N;

}

..................

..................

FunctionN()

{

Statement 1;

Statement 2;

............

............

Statement N;

}

**Figure 1.1** Structure of a C program

of statements arranged according to specific meaningful sequence. Note that programmers can choose any name for functions. It is not mandatory to write Function1, Function2, etc., with an exception that every program must contain one function that has its name as main().

**1.2 IDENTIFIERS AND KEYWORDS**

Every word in a C program is either an identifier or a keyword.

***Identifiers***

Identifiers are basically names given to program elements such as variables, arrays, and functions. They are formed by using a sequence of letters (both uppercase and lowercase), numerals, and underscores.

Following are the rules for forming identifier names:

∑ Identifiers cannot include any special characters or punctuation marks (like #, $, ^, ?, ., etc.) except the underscore “\_”.

∑ There cannot be two successive underscores.

∑ Keywords cannot be used as identifiers.

∑ The case of alphabetic characters that form the identifier name is significant. For example, ‘FIRST’ is different from ‘first’ and ‘First’. ∑ Identifiers must begin with a letter or an underscore. However, use of underscore as the first character must be avoided because several complier-defined identifiers in the standard C library have underscore as their first character. So, inadvertently duplicated names may cause definition conflicts.

∑ Identifiers can be of any reasonable length. They should not contain more than 31 characters. (They can actually be longer than 31, but the compiler looks at only the first 31 characters of the name.)

***Keywords***

Like every computer language, C has a set of reserved words often known as keywords that cannot be used as an identifier. All keywords are basically a sequence of characters that have a fixed meaning. By convention, all keywords must be written in lower case letters. Table 1.1 contains the list

of keywords in C.

**Table 1.1** Keywords in C language

| break | case | char | const | continue | default |
| --- | --- | --- | --- | --- | --- |
| else | enum | extern | float | for | goto |
| long | register | return | short | signed | sizeof |
| switch | typedef | union | unsigned | void | volatile |

auto do double if int static struct while

**1.3 BASIC DATA TYPES**

Data type determines the set of values that a data item can take and the operations that can be performed on the item. C language provides four basic data types. Table 1.2 lists the data types, their size, range, and usage for a C programmer.

The char data type is of one byte and is used to store single characters. Note that C does not provide any data type for storing text. This is because text is made up of individual characters.You

*Introduction to C* **3**

might have been surprised to see that the range of char is given as –128 to 127. char is supposed to store characters not numbers, so why this range? The answer is that in the memory, characters are stored in their ASCII codes. For example, the character ‘A’ has the ASCII code of 65. In memory we will not store ‘A’ but 65 (in binary number format).

**Table 1.2** Basic data types in C

| **Size in Bytes** | **Range** |
| --- | --- |
| 1 | –128 to 127 |
| 2 | –32768 to 32767 |
| 4 | 3.4E–38 to 3.4E+38 |
| 8 | 1.7E–308 to 1.7E+308 |

**Data TypeUse**

char To store characters

int To store integer numbers float To store floating point numbers double To store big floating point numbers

In addition, C also supports four modifiers—two sign specifiers (signed and unsigned) and two size specifiers (short and long). Table 1.3 shows the variants of basic data types.

**Table 1.3** Basic data types and their variants

| **Size in Bytes** |
| --- |
| 1 |
| 1 |
| 1 |
| 2 |
| 2 |
| 2 |
| 2 |
| 2 |
| 2 |
| 4 |
| 4 |
| 4 |
| 4 |
| 8 |
| 10 |

**Data Type Range**

char –128 to 127 unsigned char 0 to 255 signed char –128 to 127 int –32768 to 32767 unsigned int 0 to 65535 signed int –32768 to 32767 short int –32768 to 32767 unsigned short int 0 to 65535 signed short int –32768 to 32767 long int –2147483648 to 2147483647 unsigned long int 0 to 4294967295 signed long int –2147483648 to 2147483647 float 3.4E–38 to 3.4E+38 double 1.7E–308 to 1.7E+308 long double 3.4E–4932 to 1.1E+4932

**Note** When the basic data type is omitted from a declaration, then automatically type int is assumed. For example,

long var; //int is implied

While the smaller data types take less memory, the larger data types incur a performance penalty. Although the data type we use for our variables does not have a big impact on the speed or memory usage of the application, we should always try to use int unless there is a need to use any other data type.

**1.4 VARIABLES AND CONSTANTS**

A variable is defined as a meaningful name given to a data storage location in the computer memory. When using a variable, we actually refer to the address of the memory where the data is stored. C language supports two basic kinds of variables.

**4** *Data Structures Using C*

***Numeric Variables***

Numeric variables can be used to store either integer values or floating point values. Modifiers like short, long, signed, and unsigned can also be used with numeric variables. The difference between signed and unsigned numeric variables is that signed variables can be either negative or positive but unsigned variables can only be positive. Therefore, by using an unsigned variable we can increase the maximum positive range. When we omit the signed/unsigned modifier, C language automatically makes it a signed variable. To declare an unsigned variable, the unsigned modifier must be explicitly added during the declaration of the variable.

***Character Variables***

Character variables are just single characters enclosed within single quotes. These characters could be any character from the ASCII character set—letters (‘a’, ‘A’), numerals (‘2’), or special characters (‘&’).

***Declaring Variables***

To declare a variable, specify the data type of the variable followed by its name. The data type indicates the kind of values that the variable can store. Variable names should always be meaningful and must reflect the purpose of their usage in the program. In C, variable declaration always ends with a semi-colon. For example,

int emp\_num;

float salary;

char grade;

double balance\_amount;

unsigned short int acc\_no;

In C, variables can be declared at any place in the program but two things must be kept in mind. First, variables should be declared before using them. Second, variables should be declared closest to their first point of use so that the source code is easier to maintain.

***Initializing Variables***

While declaring the variables, we can also initialize them with some value. For example, int emp\_num = 7;

float salary = 9800.99

char grade = ‘A’;

double balance\_amount = 100000000;

***Constants***

Constants are identifiers whose values do not change. While values of variables can be changed at any time, values of constants can never be changed. Constants are used to define fixed values like *pi* or the charge on an electron so that their value does not get changed in the program even by mistake.

***Declaring Constants***

To declare a constant, precede the normal variable declaration with const keyword and assign it a value.

const float pi = 3.14;

*Introduction to C* **5**

**1.5 WRITING THE FIRST C PROGRAM**

To write a C program, we first need to write the code. For that, open a text editor. If you are a Windows user, you may use Notepad and if you prefer working on UNIX/Linux, you can use emac or vi. Once the text editor is opened on your screen, type the following statements:

#include <stdio.h>

int main()

{

printf("\n Welcome to the world of C ");// prints the message on the screen return 0;// returns a value 0 to the operating system

}

After writing the code, select the directory of your choice and save the file as first.c.

**#include <stdio.h>** This is the first statement in our code that includes a file called stdio.h. This file has some in-built functions. By simply including this file in our code, we can use these functions directly. stdio basically stands for Standard Input/Output, which means it has functions for input and output of data like reading values from the keyboard and printing the results on the screen.

**int main()** Every C program contains a main() function which is the starting point of the program. int is the return value of the main function. After all the statements in the program have been executed, the last statement of the program will return an integer value to the operating system. The concepts will be clear to us when we read this chapter in toto. So even if you do not understand certain things, do not worry.

**{ }** The two curly brackets are used to group all the related statements of the main function.

**Table 1.4** Escape sequences

**SequencePurpose Escape**

\a Audible signal \b Backspace

\t Tab

\n New line

\v Vertical tab

\f New page\Clear screen \r Carriage return

**printf("\n Welcome to the world of C ");** The printf function is defined in the stdio.h file and is used to print text on the screen. The message that has to be displayed on the screen is enclosed within double quotes and put inside brackets.

\n is an escape sequence and represents a newline character. It is used to print the message on a new line on the screen. Other escape sequences supported by C language are shown in Table 1.4.

**return 0;** This is a return command that is used to return value 0 to the operating system to give an indication that there were no errors during the execution of the program.

**Note** Every statement in the main function ends with a semi-colon (;).

**first.c.** If you are a Windows user, then open the command prompt by clicking Start∅Run and typing “command” and clicking Ok. Using the command prompt, change to the directory in which you saved your file and then type:

C:\>tc first.c

In case you are working on UNIX/Linux operating system, then exit the text editor and type $cc first.c –ofirst

The –o is for the output file name. If you leave out the –o, then the file name a.out is used.

**6** *Data Structures Using C*

This command is used to compile your C program. If there are any mistakes in the program, then the compiler will tell you what mistake(s) you have made and on which line the error has occurred. In case of errors, you need to re-open your .c file and correct the mistakes. However, if everything is right, then no error(s) will be reported and the compiler will create an .exe file for your program. This .exe file can be directly run by typing

"first.exe" for Windows and "./first" for UNIX/Linux operating system

When you run the .exe file, the output of the program will be displayed on screen. That is, Welcome to the world of C

**Note** The printf and return statements have been indented or moved away from the left side. This is done to make the code more readable.

***Using Comments***

Comments are a way of explaining what a program does. C supports two types of comments. ∑ // is used to comment a single statement.

∑ /\* is used to comment multiple statements. A /\* is ended with \*/ and all statements that lie between these characters are commented.

Note that comment statements are not executed by the compiler. Rather, they are ignored by the compiler as they are simply added in programs to make the code understandable by programmers as well as other users. It is a good habit to always put a comment at the top of a program that tells you what the program does. This helps in defining the usage of the program the moment you open it.

***Standard Header Files***

Till now we have used printf() function, which is defined in the stdio.h header file. Even in other programs that we will be writing, we will use many functions that are not written by us. For example, to use the strcmp() function that compares two strings, we will pass string arguments and retrieve the result. We do not know the details of how these functions work. Such functions that are provided by all C compilers are included in standard header files. Examples of these standard header files include:

∑ string.h : for string handling functions

∑ stdlib.h : for some miscellaneous functions

∑ stdio.h : for standardized input and output functions

∑ math.h : for mathematical functions

∑ alloc.h : for dynamic memory allocation

∑ conio.h : for clearing the screen

All the header files are referenced at the start of the source code file that uses one or more functions from these files.

**1.6 INPUT AND OUTPUT FUNCTIONS**

The most fundamental operation in a C program is to accept *input* values from a standard input device and *output* the data produced by the program to a standard output device. As shown in Section 1.4, we can assign values to variables using the assignment operator ‘=’. For example, int a = 3;

What if we want to assign value to variable a that is inputted from the user at run-time? This is done by using the scanf function that reads data from the keyboard. Similarly, for outputting results of

*Introduction to C* **7**

the program, printf function is used that sends results to a terminal. Like printf and scanf, there are different functions in C that can carry out the input–output operations. These functions are collectively known as Standard Input/Output Library. A program that uses standard input/output functions must contain the following statement at the beginning of the program:

#include <stdio.h>

**scanf()**

The scanf()function is used to read formatted data from the keyboard. The syntax of the scanf() function can be given as,

scanf ("control string", arg1, arg2, arg3...argn);

The control string specifies the type and format of the data that has to be obtained from the keyboard and stored in the memory locations pointed by the arguments, arg1, arg2, ...,argn. The prototype of the control string can be given as,

%[\*][width][modifier]type

**\*** is an optional argument that suppresses assignment of the input field. That is, it indicates that data should be read from the stream but ignored (not stored in the memory location). **width** is an optional argument that specifies the maximum number of characters to be read. However, if the scanf function encounters a white space or an unconvertible character, input is terminated.

**modifier** is an optional argument (**h, l,** or **L)** , which modifies the type specifier. Modifier **h** is used for short int or unsigned short int, **l** is used for long int, unsigned long int, or double values. Finally, **L** is used for long double data values.

**type** specifies the type of data that has to be read. It also indicates how this data is expected to be read from the user. The type specifiers for scanf function are given in Table 1.5.

**Table 1.5** Type specifiers

**Type Qualifying Input**

%c For single characters

%d, %i For integer values

%e,%E,%f,%g,%G For floating point numbers

%o For octal numbers

%s For a sequence of (string of) characters

%u For unsigned integer values

%x,%X For hexadecimal values

The scanf function ignores any blank spaces, tabs, and newlines entered by the user. The function simply returns the number of input fields successfully scanned and stored. As we have not studied functions till now, understanding scanf function in depth will be a bit difficult here, but for now just understand that the scanf function is used to store values in memory locations associated with variables. For this, the function should have the address of the variables. The address of the variable is denoted by an & sign followed by the name of the variable. Look at the following code that shows how we can input value in a variable of int data type: int num;

scanf(" %4d ", &num);

The scanf function reads first four digits into the address or the memory location pointed by num.

**8** *Data Structures Using C*

**Note** In case of reading strings, we do not use the & sign in the scanf function.

**printf()**

The printf function is used to display information required by the user and also prints the values of the variables. Its syntax can be given as:

printf ("control string", arg1,arg2,arg3,...,argn);

After the control string, the function can have as many arguments as specified in the control string. The control string contains format specifiers which are arranged in the order so that they correspond with the arguments in the variable list. It may also contain text to be printed such as instructions to the user, identifier names, or any other text to make the text readable.

Note that there must be enough arguments because if there are not enough arguments, then the result will be completely unpredictable. However, if by mistake you specify more number of arguments, the excess arguments will simply be ignored. The prototype of the control string can be given as below:

%[flags][width][.precision][modifier]type

Each control string must begin with a % sign.

**flags** is an optional argument, which specifies output justification like decimal point, numerical sign, trailing zeros or octadecimal or hexadecimal prefixes. Table 1.6 shows different types of flags with their descriptions.

**Table 1.6** Flags in printf()

**Flags Description**

– Left–justify within the given field width

+ Displays the data with its numeric sign (either + or –)

# Used to provide additional specifiers like o, x, X, 0, 0x, or 0X for octal and hexadecimal values respectively for values different than zero

0 The number is left–padded with zeroes (0) instead of spaces

**width** is an optional argument which specifies the minimum number of positions that the output characters will occupy. If the number of output characters is smaller than the specified width, then the output would be right justified with blank spaces to the left. However, if the number of characters is greater than the specified width, then all the characters would be printed.

**precision** is an optional argument which specifies the number of digits to print after the decimal point or the number of characters to print from a string.

**modifier** field is same as given for scanf() function.

**type** is used to define the type and the interpretation of the value of the corresponding argument. The type specifiers for printf function are given in Table 1.5.

The most simple printf statement is

printf ("Welcome to the world of C language");

The function when executed prompts the message enclosed in the quotation to be displayed on the screen.

For float x = 8900.768, the following examples show output under different format specifications:

| 8 | 9 | 0 | 0 | . | 7 | 6 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- |

printf ("%f", x)

|  |  | 8 | 9 | 0 | 0 | . | 7 | 6 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

printf("%10f", x);

*Introduction to C* **9**

|  |  | 8 | 9 | 0 | 0 | . | 7 | 7 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |

printf("%9.2f", x);

| 8 | 9 | 0 | 0 | . | 7 | 6 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- |

printf("%6f", x);

**1.7 Operators AND EXPRESSIONS**

C language supports different types of operators, which can be used with variables and constants to form expressions. These operators can be categorized into the following major groups: ∑ Arithmetic operators ∑ Relational operators ∑ Equality operators ∑ Logical operators

∑ Unary operators ∑ Conditional operator ∑ Bitwise operators ∑ Assignment operators ∑ Comma operator ∑ Sizeof operator

We will now discuss all these operators.

***Arithmetic Operators***

Consider three variables declared as,

int a=9, b=3, result;

We will use these variables to explain arithmetic operators. Table 1.7 shows the arithmetic operators, their syntax, and usage in C language.

**Table 1.7** Arithmetic operators

| **Operator** | **Syntax** | **Comment** |
| --- | --- | --- |
| \* | a \* b | result = a \* b |
| / | a / b | result = a / b |
| + | a + b | result = a + b |
| – | a – b | result = a – b |
| % | a % b | result = a % b |

**OperationResult** Multiply 27 Divide 3 Addition 12 Subtraction 6 Modulus 0

In Table 1.7, a and b (on which the operator is applied) are called **operands**. Arithmetic operators can be applied to any integer or floating point number. The addition, subtraction, multiplication, and division (+, –, \*, and /) operators are the usual arithmetic operators, so you are already familiar with these operators.

However, the operator % might be new to you. The modulus operator (%) finds the remainder of an integer division. This operator can be applied only on integer operands and cannot be used on float or double operands.

While performing modulo division, the sign of the result is always the sign of the first operand (the dividend). Therefore,

16 % 3 = 1

–16 % 3 = –1

16 % –3 = 1

–16 % –3 = –1

When both operands of the division operator (/) are integers, the division is performed as an integer division. Integer division always results in an integer result. So, the result is always rounded-off by ignoring the remainder. Therefore,

9/4 = 2 and –9/4 = –3

**10** *Data Structures Using C*

**Note** It is not possible to divide any number by zero. This is an illegal operation that results in a run-time

division-by-zero exception thereby terminating the program.

tw

Except for modulus operator, all other arithmetic operators can accept a mix of integer and

floating point numbers. If both operands are integers, the result will be an integer; if one or both operands are floating point numbers then the result would be a floating point number. All the arithmetic operators bind from left to right. Multiplication, division, and modulus operators have higher precedence over addition and subtraction operators. Thus, if an arithmetic expression consists of a mix of operators, then multiplication, division, and modulus will be carried out first in a left to right order, before any addition and subtraction can be performed. For example,

3 + 4 \* 7

= 3 + 28

= 31

***Relational Operators***

A relational operator, also known as a comparison operator, is an operator that compares two values or expressions*.* Relational operators return true or false value, depending on whether the conditional relationship between the two

**Table 1.8** Relational operators

| **Meaning** |
| --- |
| Less than |
| Greater than |
| Less than or equal to |
| Greater than equal to |

**OperatorExample** < 3 < 5 gives 1 > 7 > 9 gives 0

<= 100 <= 100 gives 1 >= 50 >=100 gives 0

***Equality Operators***

operands holds or not.

For example, to test if x is less than y, relational operator < is used as x < y. This expression will return true value if x is less than y; otherwise the value of the expression will be false. C provides four relational operators which are illustrated in Table 1.8. These operators are evaluated from left to right.

C language also supports two equality operators to compare operands for strict equality or inequality. They are equal to (==) and not equal to (!=) operators. The equality operators have

**Table 1.9** Equality operators

**Operator Meaning**

== Returns 1 if both operands are equal, 0 otherwise

!= Returns 1 if operands do not have the same value, 0 otherwise

***Logical Operators***

lower precedence than the relational operators. The equal-to operator (**==**) returns true (1) if operands on both sides of the operator have the same value; otherwise, it returns false (0). On the contrary, the not equal-to operator (**!=**) returns true (1) if the operands do not have the same value; else it returns false (0). Table 1.9 summarizes equality operators.

C language supports three logical operators. They are logical AND (&&), logical OR (||), and logical NOT (!). As in case of arithmetic expressions, logical expressions are evaluated from left to right.

**Table 1.10** Truth table of logical AND

| **B** |
| --- |
| 0 |
| 1 |
| 0 |
| 1 |

**A A && B** 0 0

0 0

1 0

1 1

***Logical AND (&&)***

Logical AND is a binary operator, which simultaneously evaluates two values or relational expressions. If both the operands are true, then the whole expression is true. If both or one of the operands is false, then the whole expression evaluates to false. The truth table of logical AND operator is given in Table 1.10.

*Introduction to C* **11**

For example,

(a < b) && (b > c)

The whole expression is true only if both expressions are true, i.e., if b is greater than a and c. ***Logical OR (||)***

**Table 1.11** Truth table of logical OR

| **B** |
| --- |
| 0 |
| 1 |
| 0 |
| 1 |

**A A || B** 0 0

0 1

1 1

1 1

**Table 1.12** Truth table of logical NOT

**A ! A**

0 1

1 0

Logical OR returns a false value if both the operands are false. Otherwise it returns a true value. The truth table of logical OR operator is given in Table 1.11. For example,

(a < b) || (b > c)

The whole expression is true if either b is greater than a or b is greater than c or b is greater than both a and c.

***Logical NOT (!)***

The logical NOT operator takes a single expression and produces a zero if the expression evaluates to a non-zero value and produces a 1 if the expression produces a zero. The truth table of logical NOT operator is given in Table 1.12. For example,

int a = 10, b;

b = !a;

Now the value of b = 0. This is because value of a = 10. !a = 0. The value of !a is assigned to b, hence the result.

***Unary Operators***

Unary operators act on single operands. C language supports three unary operators. They are unary minus, increment, and decrement operators.

***Unary Minus (–)***

Unary minus operator negates the value of its operand. For example, if a number is positive then it becomes negative when preceded with a unary minus operator. Similarly, if the number is negative, it becomes positive after applying the unary minus operator. For example,

int a, b = 10;

a = –(b);

The result of this expression is a = –10, because variable b has a positive value. After applying unary minus operator (–) on the operand b, the value becomes –10, which indicates it has a negative value.

***Increment Operator (++) and Decrement Operator (––)***

The increment operator is a unary operator that increases the value of its operand by 1. Similarly, the decrement operator decreases the value of its operand by 1. For example, – –x is equivalent to writing x = x – 1.

The increment/decrement operators have two variants: *prefix* and *postfix*. In a prefix expression (++x or – –x), the operator is applied before the operand while in a postfix expression (x++ or x––), the operator is applied after the operand.

An important point to note about unary increment and decrement operators is that ++x is not same as x++. Similarly, – –x is not the same as x– –. Although, x++ and ++x both increment the value of x by 1, in the former case, the value of x is returned before it is incremented. Whereas in the latter case, the value of x is returned after it is incremented. For example,

**12** *Data Structures Using C*

int x = 10, y;

y = x++; is equivalent to writing

y = x;

x = x + 1;

Whereas y = ++x; is equivalent to writing

x = x + 1;

y = x;

The same principle applies to unary decrement operators. Note that unary operators have a higher precedence than the binary operators. And if in an expression we have more than one unary operator then they are evaluated from right to left.

***Conditional Operator***

The syntax of the conditional operator is

exp1 ? exp2 : exp3

exp1 is evaluated first. If it is true, then exp2 is evaluated and becomes the result of the expression, otherwise exp3 is evaluated and becomes the result of the expression. For example,

large = (a > b) ? a : b

The conditional operator is used to find the larger of two given numbers. First exp1, that is a > b, is evaluated. If a is greater than b, then large = a, else large = b. Hence, large is equal to either a or b, but not both.

Conditional operators make the program code more compact, more readable, and safer to use as it is easier to both check and guarantee the arguments that are used for evaluation. Conditional operator is also known as ternary operator as it takes three operands.

***Bitwise Operators***

As the name suggests, bitwise operators perform operations at the bit level. These operators include: bitwise AND, bitwise OR, bitwise XOR, and shift operators.

*Bitwise AND*

Like boolean AND (&&), bitwise AND operator (&) performs operation on bits instead of bytes, chars, integers, etc. When we use the bitwise AND operator, the bit in the first operand is ANDed with the corresponding bit in the second operand. The truth table is the same as we had seen in logical AND operation. The bitwise AND operator compares each bit of its first operand with the corresponding bit of its second operand. If both bits are 1, the corresponding bit in the result is 1 and 0 otherwise. For example,

10101010 & 01010101 = 00000000

*Bitwise OR*

When we use the bitwise OR operator (|), the bit in the first operand is ORed with the corresponding bit in the second operand. The truth table is the same as we had seen in logical OR operation. The bitwise OR operator compares each bit of its first operand with the corresponding bit of its second operand. If one or both bits are 1, the corresponding bit in the result is 1 and 0 otherwise. For example,

10101010 | 01010101 = 11111111

*Bitwise XOR*

When we use the bitwise XOR operator, the bit in the first operand is XORed with the corresponding

**Table 1.13** Truth table of bitwise XOR

| **B** |
| --- |
| 0 |
| 1 |
| 0 |
| 1 |

**A A ^ B** 0 0

0 1

1 1

1 0

*Introduction to C* **13**

bit in the second operand. The truth table of bitwise XOR operator is shown in Table 1.13. The bitwise XOR operator compares each bit of its first operand with the corresponding bit of its second operand. If one of the bits is 1, the corresponding bit in the result is 1 and 0 otherwise. For example,

10101010 ^ 01010101 = 11111111

*Bitwise NOT (~)*

The bitwise NOT or complement is a unary operator that performs

logical negation on each bit of the operand. By performing negation of each bit, it actually produces the one’s complement of the given binary value. Bitwise NOT operator sets the bit to 1 if it was initially 0 and sets it to 0 if it was initially 1. For example,

~10101011 = 01010100

*Shift Operators*

C supports two bitwise shift operators. They are shift left (<<) and shift right (>>). The syntax for a shift operation can be given as

operand op num

where the bits in the operand are shifted left or right depending on the operator (left, if the operator is << and right, if the operator is >>) by number of places denoted by num. For example, if we have

x = 0001 1101

then x << 1 produces 0011 1010

When we apply a left shift, every bit in x is shifted to the left by one place. So, the MSB (most significant bit) of x is lost, the LSB (least significant bit) of x is set to 0. Therefore, if we have x = 0001 1101, then

x << 3 gives result = 1110 1000

On the contrary, when we apply a right shift, every bit in x is shifted to the right by one place. So, the LSB of x is lost, the MSB of x is set to 0. For example, if we have x = 0001 1101, then x >> 1 gives result = 0000 1110

Similarly, if we have x = 0001 1101, then

x >> 4 gives result = 0000 0001

**Note** The expression x << y is equivalent to multiplication of x by 2y. And the expression x >> y is equivalent to division of x by 2y if x is unsigned or has a non-negative value.

***Assignment Operators***

In C language, the assignment operator is responsible for assigning values to the variables. While the equal sign (=) is the fundamental assignment operator, C also supports other assignment operators that provide shorthand ways to represent common variable assignments.

When an equal sign is encountered in an expression, the compiler processes the statement on the right side of the sign and assigns the result to the variable on the left side. For example,

int x;

x = 10;

assigns the value 10 to variable x. The assignment operator has right-to-left associativity, so the expression

**14** *Data Structures Using C*

**Table 1.14** Assignment operators

**Operator Example**

/= float a=9.0; float b=3.0;

a /= b;

\= int a= 9;

int b = 3;

a \= b;

\*= int a= 9;

int b = 3;

a \*= b;

+= int a= 9;

int b = 3;

a += b;

–= int a= 9;

int b = 3;

a –= b;

&= int a = 10;

int b = 20;

a &= b;

^= int a = 10;

int b = 20;

a ^= b;

<<= int a= 9;

int b = 3;

a <<= b;

>>= int a= 9;

int b = 3;

a >>= b;

a = b = c = 10;

is evaluated as

(a = (b = (c = 10)));

First 10 is assigned to c, then the value of c is assigned to b. Finally, the value of b is assigned to a. Table 1.14 contains a list of other assignment operators that are supported by C.

***Comma Operator***

The comma operator, which is also called the sequential-evaluation operator, takes two operands. It works by evaluating the first expression and discarding its value, and then evaluates the second expression and returns the value as the result of the expression. Comma-separated expressions when chained together are evaluated in left-to-right sequence with the right-most value yielding the result of the expression. Among all the operators, the comma operator has the lowest precedence.

Therefore, when a comma operator is used, the entire expression evaluates to the value of the right expression. For example, the following statement first increments a, then increments b, and then assigns the value of b to x.

int a=2, b=3, x=0;

x = (++a, b+=a);

Now, the value of x = 6.

**sizeof *Operator***

sizeof is a unary operator used to calculate the size of data types. This operator can be applied to all data types. When using this operator, the

keyword sizeof is followed by a type name, variable, or expression. The operator returns the size of the data type, variable, or expression in bytes. That is, the sizeof operator is used to determine the amount of memory space that the data type/variable/expression will take.

When a type name is used, it is enclosed in parentheses, but in case of variable names and expressions, they can be specified with or without parentheses. A sizeof expression returns an unsigned value that specifies the size of the space in bytes required by the data type, variable, or expression. For example, sizeof(char) returns 1, that is the size of a character data type. If we have,

int a = 10;

unsigned int result;

result = sizeof(a);

then result = 2, that is, space required to store the variable *a* in memory. Since a is an integer, it requires 2 bytes of storage space.

***Operator Precedence Chart***

Table 1.15 lists the operators that C language supports in the order of their *precedence* (highest to lowest). The *associativity* indicates the order in which the operators of equal precedence in an expression are evaluated.

**Table 1.15** Operators precedence chart **Operator Associativity**

*Introduction to C* **15**

***Examples of Expressions Using the Precedence Chart*** If we have the following variable declarations:

()

[]

.

—>

++(postfix)

––(postfix)

++(prefix)

––(prefix)

+(unary) – (unary) ! ~

(type)

\*(indirection) &(address)

sizeof

left–to–right

right–to–left right–to–left

int a = 0, b = 1, c = –1;

float x = 2.5, y = 0.0;

then,

**(a)** a && b = 0

**(b)** a < b && c < b = 1

**(c)** b + c || ! a

= ( b + c) || (!a)

= 0 ||1

= 1

**(d)** x \* 5 && 5 || ( b / c)

= ((x \* 5) && 5) || (b / c)

\* / % left–to–right + – left–to–right << >> left–to–right

= (12.5 && 5) || (1/–1) = 1

**(e)** a <= 10 && x >= 1 && b

< <= > >=

left–to–right

= ((a <= 10) && (x >= 1)) && b = (1 && 1) && 1

== != left–to–right & left–to–right ^ left–to–right | left–to–right

&& left–to–right || left–to–right ?: right–to–left

= 1

**(f)** !x || !c || b + c

= ((!x) || (!c)) || (b + c) = (0 || 0) || 0

= 0

**(g)** x \* y < a + b || c

= ((x \* y) < (a + b)) || c

=

+= –= \*= /= %= &= ^= |=

<<= >>=

right–to–left

= (0 < 1) || –1

= 1

**(h)** (x > y) + !a || c++ = ((x > y) + (!a)) || (c++) = (1 + 1) || 0

,(comma) left–to–right**Programming Example**

= 1

**1.** Write a program to calculate the area of a circle.

#include <stdio.h>

#include <conio.h>

int main()

{

float radius;

double area;

clrscr();

printf("\n Enter the radius of the circle : "); scanf("%f", &radius);

area = 3.14 \* radius \* radius;

printf(" \n Area = %.2lf", area);

return 0;

}

**Output**

Enter the radius of the circle : 7

Area = 153.86

**16** *Data Structures Using C*

**1.8 TYPE CONVERSION AND TYPECASTING**

Type conversion or typecasting of variables refers to changing a variable of one data type into another. While type conversion is done implicitly, casting has to be done explicitly by the programmer. We will discuss both these concepts here.

***Type Conversion***

Type conversion is done when the expression has variables of different data types. So to evaluate the expression, the data type is promoted from lower to higher level where the hierarchy of data types can be given as: double, float, long, int, short, and char. For example, type conversion is automatically done when we assign an integer value to a floating point variable. Consider the following code:

float x;

int y = 3;

x = y;

Now, x = 3.0, as integer value is automatically converted into its equivalent floating point representation.

***Typecasting***

Typecasting is also known as *forced conversion*. It is done when the value of one data type has to be converted into the value of another data type. The code to perform typecasting can be given as:

float salary = 10000.00;

int sal;

sal = (int) salary;

When floating point numbers are converted to integers, the digits after the decimal are truncated. Therefore, data is lost when floating point representations are converted to integral representations. As we can see in the code, typecasting can be done by placing the destination data type in parentheses followed by the variable name that has to be converted. Hence, we conclude that typecasting is done to make a variable of one data type to act like a variable of another type.

**Programming Example**

**2.** Write a program to convert an integer into the corresponding floating point number.

#include <stdio.h>

#include <conio.h>

int main()

{

float f\_num;

int i\_num;

clrscr();

printf("\n Enter any integer: ");

scanf("%d", &i\_num);

f\_num = (float)i\_num;

printf("\n The floating point variant of %d is = %f", i\_num, f\_num);

return 0;

}

**Output**

Enter any integer: 56

The floating point variant of 56 is = 56.000000

*Introduction to C* **17**

**1.9 CONTROL STATEMENTS**

Till now we know that the code in the C program is executed sequentially from the first line of the program to its last line. That is, the second statement is executed after the first, the third statement is executed after the second, so on and so forth. Although this is true, in some cases we want only selected statements to be executed. Control flow statements enable programmers to conditionally execute a particular block of code. There are three types of control statements: decision control (branching), iterative (looping), and jump statements. While branching means deciding what actions have to be taken, looping, on the other hand, decides how many times the action has to be taken. Jump statements transfer control from one point to another point.

**1.9.1 Decision Control Statements**

C supports decision control statements that can alter the flow of a sequence of instructions. These statements help to jump from one part of the program to another depending on whether a particular condition is satisfied or not. These decision control statements include:

(a) if statement, (b) if–else statement,

(c) if–else–if statement, and (d) switch–case statement.

**if *Statement***

if statement is the simplest decision control statement that is frequently used in decision making. The general form of a simple if statement is shown in Fig. 1.2.

**Syntax of if Statement**

if (test expression)

{

statement 1;

............

statement n;

}

statement x;

**Figure 1.2** if statement construct

Test

Expression

TRUE

Statement Block 1 Statement x

FALSE

The if block may include 1 statement or *n* statements enclosed within curly brackets. First the test expression is evaluated. If the test expression is true, the statements of the if block are executed, otherwise these statements will be skipped and the execution will jump to statement x.

The statement in an if block is any valid C language statement, and the test expression is any valid C language expression that evaluates to either true or false. In addition to simple relational expressions, we can also use compound expressions formed using logical operators. Note that there is no semi-colon after the test expression. This is because the condition and statement should be put together as a single statement.

**18** *Data Structures Using C*

#include <stdio.h>

int main()

{

int x=10;

if (x>0) x++;

printf("\n x = %d", x);

return 0;

}

In the above code, we take a variable x and initialize it to 10. In the test expression, we check if the value of x is greater than 0. As 10 > 0, the test expression evaluates to true, and the value of x is incremented. After that, the value of x is printed on the screen. The output of this program is

x = 11

Observe that the printf statement will be executed even if the test expression is false.

**Note** In case the statement block contains only one statement, putting curly brackets becomes optional. If there are more than one statement in the statement block, putting curly brackets becomes mandatory.

**if–else *Statement***

We have studied that using if statement plays a vital role in conditional branching. Its usage is very simple. The test expression is evaluated, if the result is true, the statement(s) followed by the expression is executed, else if the expression is false, the statement is skipped by the compiler.

What if you want a separate set of statements to be executed if the expression returns a false value? In such cases, we can use an if–else statement rather than using a simple if statement. The general form of simple if–else statement is shown in Fig. 1.3.

**Syntax of if-else**

**Statement**

if (test expression) {

statement block 1; }

else

{

statement block 2; }

statement x;

TRUE FALSE

Test

Expression

Statement Block 1 Statement Block 2 Statement x

**Figure 1.3** if–else statement construct

In the if–else construct, first the test expression is evaluated. If the expression is true, statement block 1 is executed and statement block 2 is skipped. Otherwise, if the expression is false, statement block 2 is executed and statement block 1 is ignored. In any case after the statement block 1 or 2 gets executed, the control will pass to statement x. Therefore, statement x is executed in every case.

*Introduction to C* **19**

**Programming Example**

**3.** Write a program to find whether a number is even or odd.

#include <stdio.h>

int main()

{

int a;

printf("\n Enter the value of a : ");

scanf("%d", &a);

if(a%2==0)

printf("\n %d is even", a);

else

printf("\n %d is odd", a);

return 0;

}

**Output**

Enter the value of a : 6

6 is even

**if–else–if *Statement***

C language supports if–else–if statements to test additional conditions apart from the initial test expression. The if–else–if construct works in the same way as a normal if statement. Its construct is given in Fig. 1.4.

**Syntax of if-else-if Statement**

if (test expression 1) {

statement block 1;

}

FALSE

TRUE

Statement

Test

~~Expression~~ 1

else if (test expression 2)

{

statement block 2;

}

...........................

else

{

statement block x;

}

statement y;

**Figure 1.4** if–else–if statement construct

Block 1

TRUE

Statement

Block 2

Test

Expression 2

Statement y

FALSE

Statement Block x

Note that it is not necessary that every if statement should have an else block as C supports simple if statements. After the first test expression or the first if branch, the programmer can have as many else–if branches as he wants depending on the expressions that have to be tested. For example, the following code tests whether a number entered by the user is negative, positive, or equal to zero.

#include <stdio.h>

int main()

{

**20** *Data Structures Using C*

int num;

printf("\n Enter any number : ");

scanf("%d", &num);

if(num==0)

printf("\n The value is equal to zero");

else if(num>0)

printf("\n The number is positive");

else

printf("\n The number is negative");

return 0;

}

Note that if the first test expression evaluates to a true value, i.e., num=0, then the rest of the statements in the code will be ignored and after executing the printf statement that displays ‘The value is equal to zero’, the control will jump to return 0 statement.

**switch–case *Statement***

A switch-case statement is a multi-way decision statement that is a simplified version of an if– else–if block. The general form of a switch statement is shown in Fig. 1.5.

TRUE

Value 1

**Syntax of Switch Statement**

switch (variable)

{

case value 1:

statement block 1;

break;

case value 2:

statement block 2;

break;

.....................

case value N:

statement block N;

break;

default:

statement block D;

break;

}

statement X;

**Figure 1.5** switch–case statement construct

Statement Block 1 TRUE

Statement Block 2

TRUE

Statement Block N

FALSE

Value 2

FALSE

FALSE

Value N

FALSE

Statement Block D Statement X

The power of nested if–else–if statements lies in the fact that it can evaluate more than one expression in a single logical structure. switch statements are mostly used in two situations:

∑ When there is only one variable to evaluate in the expression

∑ When many conditions are being tested for

*Introduction to C* **21**

When there are many conditions to test, using the if and else–if constructs becomes complicated and confusing. Therefore, switch case statements are often used as an alternative to long if statements that compare a variable to several ‘integral’ values (integral values are those values that can be expressed as an integer, such as the value of a char). Switch statements are also used to handle the input given by the user.

We have already seen the syntax of the switch statement. The switch case statement compares the value of the variable given in the switch statement with the value of each case statement that follows. When the value of the switch and the case statement matches, the statement block of that particular case is executed.

Did you notice the keyword default in the syntax of the switch case statement? Default is the case that is executed when the value of the variable does not match with any of the values of the case statements. That is, default case is executed when no match is found between the values of switch and case statements and thus there are no statements to be executed. Although the default case is optional, it is always recommended to include it as it handles any unexpected case.

In the syntax of the switch–case statement, we have used another keyword break. The break statement must be used at the end of each case because if it is not used, then the case that matched and all the following cases will be executed. For example, if the value of switch statement matched with that of case 2, then all the statements in case 2 as well as the rest of the cases including default will be executed. The break statement tells the compiler to jump out of the switch case statement and execute the statement following the switch–case construct. Thus, the keyword break is used to break out of the case statements.

*Advantages of Using a* switch–case *Statement*

Switch–case statement is preferred by programmers due to the following reasons: ∑ Easy to debug

∑ Easy to read and understand

∑ Ease of maintenance as compared to its equivalent if–else statements

∑ Like if–else statements, switch statements can also be nested

∑ Executes faster than its equivalent if–else construct

**Programming Example**

**4.** Write a program to determine whether the entered character is a vowel or not.

#include <stdio.h>

int main()

{

char ch;

printf("\n Enter any character : ");

scanf("%c", &ch);

switch(ch)

{

case ‘A’:

case ‘a’:

printf("\n %c is VOWEL", ch);

break;

case ‘E’:

case ‘e’:

printf("\n %c is VOWEL", ch);

break;

case ‘I’:

case ‘i’:

**22** *Data Structures Using C*

printf("\n %c is VOWEL", ch);

break;

case ‘O’:

case ‘o’:

printf("\n %c is VOWEL", ch);

break;

case ‘U’:

case ‘u’:

printf("\n %c is VOWEL", ch);

break;

default: printf("\n %c is not a vowel", ch);

}

return 0;

}

**Output**

Enter any character : j

j is not a vowel

Note that there is no break statement after case A, so if the character A is entered then control will execute the statements given in case a.

**1.9.2 Iterative Statements**

Iterative statements are used to repeat the execution of a sequence of statements until the specified expression becomes false. C supports three types of iterative statements also known as looping statements. They are

∑ while loop

∑ do–while loop

∑ for loop

In this section, we will discuss all these statements.

**while *loop***

The while loop provides a mechanism to repeat one or more statements while a particular condition is true. Figure 1.6 shows the syntax and general form of a while loop.

Statement x

**Syntax of While Loop**

statement x;

while (condition)

{

statement block;

}

statement y;

**Figure 1.6** While loop construct

Update the Condition Expression

Statement Block

TRUE

Condition

FALSE

Statement y

*Introduction to C* **23**

Note that in the while loop, the condition is tested before any of the statements in the statement block is executed. If the condition is true, only then the statements will be executed, otherwise if the condition is false, the control will jump to statement y, that is the immediate statement outside the while loop block.

In the flow diagram of Fig. 1.6, it is clear that we need to constantly update the condition of the while loop. It is this condition which determines when the loop will end. The while loop will execute as long as the condition is true. Note that if the condition is never updated and the condition never becomes false, then the computer will run into an infinite loop which is never desirable. For example, the following code prints the first 10 numbers using a while loop.

*#*include <stdio.h>

int main()

{

int i = 1;

while(i<=10)

{

printf("\n %d", i);

i = i + 1; // condition updated

}

return 0;

}

Note that initially i = 1 and is less than 10, i.e., the condition is true, so in the while loop the value of i is printed and its value is incremented by 1. When i=11, the condition becomes false and the loop ends.

**Programming Example**

**5.** Write a program to calculate the sum of numbers from m to n.

#include <stdio.h>

int main()

{

int n, m, i, sum =0;

printf("\n Enter the value of m : ");

scanf("%d", &m);

i=m;

printf("\n Enter the value of n : ");

scanf("%d", &n);

while(i<=n)

{

sum = sum + i;

i = i + 1;

}

printf("\n The sum of numbers from %d to %d = %d", m, n, sum);

return 0;

}

**Output**

Enter the value of m : 2

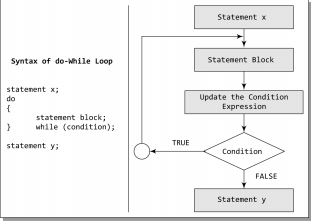
Enter the value of n : 10

The sum of numbers from 2 to 10 = 54

**do–while *Loop***

The do–while loop is similar to the while loop. The only difference is that in a do–while loop, the test condition is tested at the end of the loop. As the test condition is evaluated at the end, this means that the body of the loop gets executed at least one time (even if the condition is false). Figure 1.7 shows the syntax and the general form of a do–while loop.

**24** *Data Structures Using C*

**

**Figure 1.7** Do–while construct

Note that the test condition is enclosed in parentheses and followed by a semi-colon. The statements in the statement block are enclosed within curly brackets. The curly brackets are optional if there is only one statement in the body of the do–while loop.

The do–while loop continues to execute while the condition is true and when the condition becomes false, the control jumps to the statement following the do–while loop. The major disadvantage of using a do–while loop is that it always executes at least once, so even if the user enters some invalid data, the loop will execute. However, do–while loops are widely used to print a list of options for menu-driven programs. For example, consider the following code.

#include <stdio.h>

int main()

{

int i = 1;

do

{

printf("\n %d", i);

i = i + 1;

} while(i<=10);

return 0;

}

What do you think will be the output? Yes, the code will print numbers from 1 to 10. **Programming Example**

**6.** Write a program to calculate the average of first *n* numbers.

#include <stdio.h>

int main()

{

int n, i = 0, sum =0;

float avg = 0.0;

*Introduction to C* **25**

printf("\n Enter the value of n : ");

scanf("%d", &n);

do

{

sum = sum + i;

i = i + 1;

} while(i<=n);

avg = (float)sum/n;

printf("\n The sum of first %d numbers = %d",n, sum);

printf("\n The average of first %d numbers = %.2f", n, avg);

return 0;

}

**Output**

Enter the value of n : 20

The sum of first 20 numbers = 210

The average of first 20 numbers = 10.05

**for *Loop***

Like the while and do–while loops, the for loop provides a mechanism to repeat a task till a particular condition is true. The synax and general form of a for loop is given in Fig. 1.8.

Initialization of

Loop Variable

**Syntax of for Loop**

for (initialization; condition; increment/decrement/update) {

statement block;

}

statement y;

**Figure 1.8** for loop construct

Controlling

Condition for Loop Variable

TRUE

Statement Block

Update the

Loop Variable

Statement y

FALSE

When a for loop is used, the loop variable is initialized only once. With every iteration, the value of the loop variable is updated and the condition is checked. If the condition is true, the statement block of the loop is executed, else the statements comprising the statement block of the for loop are skipped and the control jumps to the statement following the for loop body.

In the syntax of the for loop, initialization of the loop variable allows the programmer to give it a value. Second, the condition specifies that while the conditional expression is true, the loop

**26** *Data Structures Using C*

should continue to repeat itself. Every iteration of the loop must make the condition to exit the loop approachable. So, with every iteration, the loop variable must be updated. Updating the loop variable may include incrementing the loop variable, decrementing the loop variable or setting it to some other value like, i +=2, where i is the loop variable.

Note that every section of the for loop is separated from the other with a semi-colon. It is possible that one of the sections may be empty, though the semi-colons still have to be there. However, if the condition is empty, it is evaluated as true and the loop will repeat until something else stops it.

The for loop is widely used to execute a single or a group of statements for a limited number of times. The following code shows how to print the first *n* numbers using a for loop.

#include <stdio.h>

int main()

{

int i, n;

printf("\n Enter the value of n :");

scanf("%d", &n);

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

printf("\n %d", i);

return 0;

}

In the code, i is the loop variable. Initially, it is initialized with 1. Suppose the user enters 10 as the value of n. Then the condition is checked, since the condition is true as i is less than n, the statement in the for loop is executed and the value of i is printed. After every iteration, the value of i is incremented. When i exceeds the value of n, the control jumps to the return 0 statement.

**Programming Example**

**7.** Write a program to determine whether a given number is a prime or a composite number.

#include <stdio.h>

#include <conio.h>

int main()

{

int flag = 0, i, num;

clrscr();

printf("\n Enter any number : ");

scanf("%d", &num);

for(i=2; i<num/2;i++)

{

if(num%i == 0)

{

flag =1;

break;

}

}

if(flag == 1)

printf("\n %d is a composite number", num);

else

printf("\n %d is a prime number", num);

return 0;

}

**Output**

Enter any number : 37

37 is a prime number

*Introduction to C* **27**

**1.9.3 Break and Continue Statements**

**break *Statement***

In C, the break statement is used to terminate the execution of the nearest enclosing loop in which it appears. We have already seen its use in the switch statement. The break statement is widely used with for, while, and do–while loops. When the compiler encounters a break statement, the control passes to the statement that follows the loop in which the break statement appears. Its syntax is quite simple, just type keyword break followed by a semi-colon.

**break;**

The example given below shows the manner in which break statement is used to terminate the loop in which it is embedded.

#include <stdio.h>

int main()

{

int i = 0;

while(i<=10)

{

if (i==5)

break;

printf("\t %d", i);

i = i + 1;

}

return 0;

}

**Output**

0 1 2 3 4

As soon as i becomes equal to 5, the break statement is executed and the control jumps to the statement following the while loop.

Hence, the break statement is used to exit a loop from any point within its body, bypassing its normal termination expression.

**continue *Statement***

Like the break statement, the continue statement can only appear in the body of a loop. When the compiler encounters a continue statement, then the rest of the statements in the loop are skipped and the control is unconditionally transferred to the loop-continuation portion of the nearest enclosing loop. Its syntax is quite simple, just type keyword continue followed by a semi-colon.

**continue;**

Again like the break statement, the continue statement cannot be used without an enclosing for, while, or do–while loop. When the continue statement is encountered in the while loop and in the do–while loop, the control is transferred to the code that tests the controlling expression. However, if placed within a for loop, the continue statement causes a branch to the code that updates the loop variable. For example, consider the following code:

#include <stdio.h>

int main()

{

int i;

for(i=0; i<= 10; i++)

{

if (i==5)

**28** *Data Structures Using C*

continue;

printf("\t %d", i);

}

return 0;

}

**Output**

0 1 2 3 4 6 7 8 9 10

Note that the code is meant to print numbers from 0 to 10. But as soon as i becomes equal to 5, the continue statement is encountered, so the printf() statement is skipped and the control passes to the expression that increments the value of i.

Hence, we conclude that the continue statement is somewhat the opposite of the break statement. It forces the next iteration of the loop to take place, skipping any code in between itself and the test condition of the loop. It is generally used to restart a statement sequence when an error occurs.

**1.10 FUNCTIONS**

C enables its programmers to break up a program into segments commonly known as *functions*, each of which can be written more or less independently of the others. Every function in the program is supposed to perform a well-defined task. Therefore, the program code of one function is completely insulated from the other functions.

Every function interfaces to the outside world

main()

{

........... ........... func1();

........... ........... return ; }

func1()

{

statement block; }

in terms of how information is transferred to it and how results generated by the function are transmitted back from it. This interface is basically specified by the function name. For example, look at Fig. 1.9 which explains how the main() function calls another function to perform a well-defined task.

In the figure, we can see that main() calls a function named func1(). Therefore, main() is

**Figure 1.9** main() calls func1()

known as the *calling function* and func1() is known as the *called function*. The moment the compiler

encounters a function call, the control jumps to the statements that are a part of the called function. After the called function is executed, the control is returned to the calling program. The main() function can call as many functions as it wants and as many times as it wants. For example, a function call placed within a for loop, while loop, or do–while loop may call the same function multiple times till the condition holds true.

Not only main(), any function can call any other function. For example, look at Fig. 1.10 which shows one function calling another, and the other function in turn calling some other function.

**main()**

{

......... ......... func1(); ......... ......... return 0; }

**func1()**

{

.........

.........

func2();

.........

.........

return;

}

**func2()**

{

.........

.........

func3();

.........

.........

return;

}

**func3()**

{

.........

.........

.........

.........

return;

}

**Figure 1.10** Function calling another function

*Introduction to C* **29**

**1.10.1 Why are Functions Needed?**

Let us analyse the reasons why segmenting a program into manageable chunks is an important aspect of programming.

∑ Dividing the program into separate well-defined functions facilitates each function to be written and tested separately. This simplifies the process of getting the total program to work. ∑ Understanding, coding, and testing multiple separate functions is easier than doing the same for one big function.

∑ If a big program has to be developed without using any function other than main(), then there will be countless lines in the main() function and maintaining that program will be a difficult task.

∑ All the libraries in C contain a set of functions that the programmers are free to use in their programs. These functions have been pre-written and pre-tested, so the programmers can use them without worrying about their code details. This speeds up program development, by allowing the programmer to concentrate only on the code that he has to write.

∑ Like C libraries, programmers can also write their own functions and use them from different points in the main program or any other program that needs its functionalities. ∑ When a big program is broken into comparatively smaller functions, then different programmers working on that project can divide the workload by writing different functions.

**1.10.2 Using Functions**

A function can be compared to a *black box* that takes in inputs, processes it, and then outputs the result. However, we may also have a function that does not take any inputs at all, or a function that does not return any value at all. While using functions, we will be using the following terminologies:

∑ A function *f* that uses another function *g* is known as the *calling function,* and *g* is known as the *called function*.

∑ The inputs that a function takes are known as *arguments.*

∑ When a called function returns some result back to the calling function, it is said to *return* that result.

∑ The calling function may or may not pass *parameters* to the called function. If the called function accepts arguments, the calling function will pass parameters, else not. ∑ *Function declaration* is a declaration statement that identifies a function’s name, a list of arguments that it accepts, and the type of data it returns.

∑ *Function definition* consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

***Function Declaration***

Before using a function, the compiler must know the number of parameters and the type of parameters that the function expects to receive and the data type of value that it will return to the calling program. Placing the function declaration statement prior to its use enables the compiler to make a check on the arguments used while calling that function.

The general format for declaring a function that accepts arguments and returns a value as result can be given as:

return\_data\_type function\_name(data\_type variable1, data\_type variable2,..);

Here, **function\_name** is a valid name for the function. Naming a function follows the same rules that are followed while naming variables. A function should have a meaningful name that must specify the task that the function will perform.

**30** *Data Structures Using C*

**return\_data\_type** specifies the data type of the value that will be returned to the calling function as a result of the processing performed by the called function.

**(data\_type variable1, data\_type variable2, ...)** is a list of variables of specified data types. These variables are passed from the calling function to the called function. They are also known as arguments or parameters that the called function accepts to perform its task.

**Note** A function having void as its return type cannot return any value. Similarly, a function having void as its parameter list cannot accept any value.

***Function Definition***

When a function is defined, space is allocated for that function in the memory. A function definition comprises of two parts:

∑ Function header

∑ Function body

The syntax of a function definition can be given as:

**return\_data\_type function\_name(data\_type variable1, data\_type variable2,..)** {

.............

statements

.............

return(variable);

}

Note that the number of arguments and the order of arguments in the function header must be the same as that given in the function declaration statement.

While **return\_data\_type function\_name(data\_type variable1, data\_type variable2,...)** is known as the function header, the rest of the portion comprising of program statements within the curly brackets { } is the function body which contains the code to perform the specific task.

Note that the function header is same as the function declaration. The only difference between the two is that a function header is not followed by a semi-colon.

***Function Call***

The function call statement invokes the function. When a function is invoked, the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function. A function call statement has the following syntax:

function\_name(variable1, variable2, ...);

The following points are to be noted while calling a function:

∑ Function name and the number and the type of arguments in the function call must be same as that given in the function declaration and the function header of the function definition. ∑ Names (and not the types) of variables in function declaration, function call, and header of function definition may vary.

∑ Arguments may be passed in the form of expressions to the called function. In such a case, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.

∑ If the return type of the function is not void, then the value returned by the called function may be assigned to some variable as given below.

**variable\_name = function\_name(variable1, variable2, ...);**

*Introduction to C* **31**

**Programming Example**

**8.** Write a program to find whether a number is even or odd using functions.

#include <stdio.h>

int evenodd(int); //FUNCTION DECLARATION

int main()

{

int num, flag;

printf("\n Enter the number : ");

scanf("%d", &num);

flag = evenodd(num); //FUNCTION CALL

if (flag == 1)

printf("\n %d is EVEN", num);

else

printf("\n %d is ODD", num);

return 0;

}

int evenodd(int a) // FUNCTION HEADER

{

// FUNCTION BODY

if(a%2 == 0)

return 1;

else

retun 0;

}

**Output**

Enter the number : 78

78 is EVEN

**1.10.3 Passing Parameters to Functions**

There are two ways in which arguments or parameters can be passed to the called function. **Call by value** The values of the variables are passed by the calling function to the called function.

**Call by reference** The addresses of the variables are passed by the calling function to the called function.

***Call by Value***

In this method, the called function creates new variables to store the value of the arguments passed to it. Therefore, the called function uses a copy of the actual arguments to perform its intended task. If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function, no change will be made to the value of the variables. This is because all the changes are made to the copy of the variables and not to the actual variables. To understand this concept, consider the code given below. The function add() accepts an integer variable num and adds 10 to it. In the calling function, the value of num = 2. In add(), the value of num is modified to 12 but in the calling function, the change is not reflected.

#include <stdio.h>

void add(int n);

int main()

{

int num = 2;

printf("\n The value of num before calling the function = %d", num); add(num);

**32** *Data Structures Using C*

printf("\n The value of num after calling the function = %d", num);

 return 0;

}

void add(int n)

{

n = n + 10;

printf("\n The value of num in the called function = %d", n);

}

**Output**

The value of num before calling the function = 2

The value of num in the called function = 12

The value of num after calling the function = 2

Following are the points to remember while passing arguments to a function using the call-by value method:

∑ When arguments are passed by value, the called function creates new variables of the same data type as the arguments passed to it.

∑ The values of the arguments passed by the calling function are copied into the newly created variables.

∑ Values of the variables in the calling functions remain unaffected when the arguments are passed using the call-by-value technique.

*Pros and cons*

The biggest advantage of using the call-by-value technique is that arguments can be passed as variables, literals, or expressions, while its main drawback is that copying data consumes additional storage space. In addition, it can take a lot of time to copy, thereby resulting in performance penalty, especially if the function is called many times.

***Call by Reference***

When the calling function passes arguments to the called function using the call-by-value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement. A better option is to pass arguments using the call-by-reference technique. In this method, we declare the function parameters as references rather than normal variables. When this is done, any changes made by the function to the arguments it received are also visible in the calling function.

To indicate that an argument is passed using call by reference, an asterisk (\*) is placed after the type in the parameter list.

Hence, in the call-by-reference method, a function receives an implicit reference to the argument, rather than a copy of its value. Therefore, the function can modify the value of the variable and that change will be reflected in the calling function as well. The following code illustrates this concept.

#include <stdio.h>

void add(int \*);

int main()

{

int num = 2;

printf("\n The value of num before calling the function = %d", num);

add(&num);

printf("\n The value of num after calling the function = %d", num);

return 0;

}

void add(int \*n)

*Introduction to C* **33**

{

\*n = \*n + 10;

printf("\n The value of num in the called function = %d", \*n);

}

**Output**

The value of num before calling the function = 2

The value of num in the called function = 12

The value of num after calling the function = 12

*Advantages*

The advantages of using the call-by-reference technique of passing arguments include: ∑ Since arguments are not copied into the new variables, it provides greater time and space efficiency.

∑ The function can change the value of the argument and the change is reflected in the calling function.

∑ A function can return only one value. In case we need to return multiple values, we can pass those arguments by reference, so that the modified values are visible in the calling function.

*Disadvantages*

However, the drawback of using this technique is that if inadvertent changes are caused to variables in called function then these changes would be reflected in calling function as original values would have been overwritten.

Consider the code given below which swaps the value of two integers. Note the value of integers in the calling function and called function.

//This function swaps the value of two variables

#include <stdio.h>

void swap\_call\_val(int, int);

void swap\_call\_ref(int \*, int \*);

int main()

{

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

printf("\n In main(), a = %d and b = %d", a, b);

swap\_call\_val(a, b);

printf("\n In main(), a = %d and b = %d", a, b);

printf("\n\n In main(), c = %d and d = %d", c, d);

swap\_call\_ref(&c, &d);

printf("\n In main(), c = %d and d = %d", c, d);

return 0;

}

void swap\_call\_val(int a, int b)

{

int temp;

temp = a;

a = b;

b = temp;

printf("\n In function (Call By Value Method) – a = %d and b = %d", a, b); }

void swap\_call\_ref(int \*c, int \*d)

{

int temp;

temp = \*c;

\*c = \*d;

\*d = temp;

printf("\n In function (Call By Reference Method) – c = %d and d = %d", \*c, \*d); }

**34** *Data Structures Using C*

**Output**

In main(), a = 1 and b = 2

In function (Call By Value Method) – a = 2 and b = 1

In main(), a = 1 and b = 2

In main(), c = 3 and d = 4

In function (Call By Reference Method) – c = 4 and d = 3

In main(), c = 4 and d = 3

**1.11 Pointers**

Every variable in C has a name and a value associated with it. When a variable is declared, a specific block of memory within the computer is allocated to hold the value of that variable. The size of the allocated block depends on the data type.

Consider the following statement.

int x = 10;

When this statement executes, the compiler sets aside 2 bytes of memory to hold the value 10. It also sets up a symbol table in which it adds the symbol x and the relative address in the memory where those 2 bytes were set aside.

(Note the size of integer may vary from one system to another. On 32 bit systems, integer variable is allocated 4 bytes while on 16 bit systems it is allocated 2 bytes.)

Thus, every variable in C has a value and also a memory location (commonly known as *address*) associated with it. We will use terms rvalue and lvalue for the value and the address of the variable, respectively.

The rvalue appears on the right side of the assignment statement (10 in the above statement) and cannot be used on the left side of the assignment statement. Therefore, writing 10 = k; is illegal. If we write,

int x, y;

x = 10;

y = x;

then, we have two integer variables x and y. The compiler reserves memory for the integer variable x and stores the rvalue 10 in it. When we say y = x, then x is interpreted as its rvalue since it is on the right hand side of the assignment operator =. Therefore, here x refers to the value stored at the memory location set aside for x, in this case 10. After this statement is executed, the rvalue

of y is also 10.

You must be wondering why we are discussing addresses and lvalues. Actually pointers are nothing but memory addresses. A pointer is a variable that contains the memory location of another variable. Therefore, a pointer is a variable that represents the location of a data item, such as a variable or an array element. Pointers are frequently used in C, as they have a number of useful applications. These applications include:

∑ Pointers are used to pass information back and forth between functions.

∑ Pointers enable the programmers to return multiple data items from a function via function arguments.

∑ Pointers provide an alternate way to access the individual elements of an array. ∑ Pointers are used to pass arrays and strings as function arguments. We will discuss this in subsequent chapters.

∑ Pointers are used to create complex data structures, such as trees, linked lists, linked stacks, linked queues, and graphs.

*Introduction to C* **35**

∑ Pointers are used for the dynamic memory allocation of a variable (refer Appendix A on memory allocation in C programs).

**1.11.1 Declaring Pointer Variables**

The general syntax of declaring pointer variables can be given as below.

data\_type \*ptr\_name;

Here, data\_type is the data type of the value that the pointer will point to. For example,

int \*pnum;

char \*pch;

float \*pfnum;

In each of the above statements, a pointer variable is declared to point to a variable of the specified data type. Although all these pointers (pnum, pch, and pfnum) point to different data types, they will occupy the same amount of space in the memory. But how much space they will occupy will depend on the platform where the code is going to run. Now let us declare an integer pointer variable and start using it in our program code.

int x= 10;

int \*ptr;

ptr = &x;

In the above statement, ptr is the name of the pointer variable. The \* informs the compiler that ptr is a pointer variable and the int specifies that it will store the address of an integer variable. An integer pointer variable, therefore, ‘points to’ an integer variable. In the last statement, ptr is assigned the address of x. The & operator retrieves the lvalue (address) of x, and copies that to the contents of the pointer ptr. Consider the memory cells given in Fig. 1.11.

10

1000 1001 1002 1003 1004 1005 1006 1007 1008 1009

**Figure 1.11** Memory representation

Now, since x is an integer variable, it will be allocated 2 bytes. Assuming that the compiler assigns it memory locations 1003 and 1004, the address of x (written as &x) is equal to 1003, that is the starting address of x in the memory. When we write, ptr = &x, then ptr = 1003.

We can ‘dereference’ a pointer, *i*.*e*., we can refer to the value of the variable to which it points by using the unary \* operator as in \*ptr. That is, \*ptr = 10, since 10 is the value of x. Look at the following code which shows the use of a pointer variable:

#include <stdio.h>

int main()

{

int num, \*pnum;

pnum = &num;

printf("\n Enter the number : ");

scanf("%d", &num);

printf("\n The number that was entered is : %d", \*pnum);

return 0;

}

**Output**

Enter the number : 10

The number that was entered is : 10

What will be the value of \*(&num)? It is equivalent to simply writing num.

**36** *Data Structures Using C*

**1.11.2 Pointer Expressions and Pointer Arithmetic**

Like other variables, pointer variables can also be used in expressions. For example, if ptr1 and ptr2 are pointers, then the following statements are valid:

int num1 = 2, num2 = 3, sum = 0, mul = 0, div = 1;

int \*ptr1, \*ptr2;

ptr1 = &num1;

ptr2 = &num2;

sum = \*ptr1 + \*ptr2;

mul = sum \* (\*ptr1);

\*ptr2 += 1;

div = 9 + (\*ptr1)/(\*ptr2) – 30;

In C, the programmer may add integers to or subtract integers from pointers as well as subtract one pointer from the other. We can also use shorthand operators with the pointer variables as we use them with other variables.

C also allows comparing pointers by using relational operators in the expressions. For example, p1>p2, p1==p2 and p1!=p2 are all valid in C.

Postfix unary increment (++) and decrement (––) operators have greater precedence than the dereference operator (\*). Therefore, the expression \*ptr++ is equivalent to \*(ptr++), as ++ has greater operator precedence than \*. Thus, the expression will increase the value of ptr so that it now points to the next memory location. This means that the statement \*ptr++ does not do the intended task. Therefore, to increment the value of the variable whose address is stored in ptr, you should write (\*ptr)++.

**1.11.3 Null Pointers**

So far, we have studied that a pointer variable is a pointer to a variable of some data type. However, in some cases, we may prefer to have a *null pointer* which is a special pointer value and does not point to any value. This means that a null pointer does not point to any valid memory address.

To declare a null pointer, you may use the predefined constant NULL which is defined in several standard header files including <stdio.h>, <stdlib.h>, and <string.h>. After including any of these files in your program, you can write

int \*ptr = NULL;

You can always check whether a given pointer variable stores the address of some variable or contains NULL by writing,

if (ptr == NULL)

{

Statement block;

}

You may also initialize a pointer as a null pointer by using the constant 0

int \*ptr,

ptr = 0;

This is a valid statement in C as NULL is a preprocessor macro, which typically has the value or replacement text 0. However, to avoid ambiguity, it is always better to use NULL to declare a null pointer. A function that returns pointer values can return a null pointer when it is unable to perform its task.

**1.11.4 Generic Pointers**

A generic pointer is a pointer variable that has *void* as its data type. The *void pointer*, or the generic pointer, is a special type of pointer that can point to variables of any data type. It is declared like a normal pointer variable but using the void keyword as the pointer’s data type. For example,

*Introduction to C* **37**

void \*ptr;

In C, since you cannot have a variable of type void, the void pointer will therefore not point to any data and, thus, cannot be dereferenced. You need to cast a void pointer to another kind of pointer before using it.

Generic pointers are often used when you want a pointer to point to data of different types at different times. For example, take a look at the following code.

#include <stdio.h>

int main()

{

int x=10;

char ch = ‘A’;

void \*gp;

gp = &x;

printf("\n Generic pointer points to the integer value = %d", \*(int\*)gp); gp = &ch;

printf("\n Generic pointer now points to the character= %c", \*(char\*)gp); return 0;

}

**Output**

Generic pointer points to the integer value = 10

Generic pointer now points to the character = A

It is always recommended to avoid using void pointers unless absolutely necessary, as they effectively allow you to avoid type checking.

**Programming Example**

**9.** Write a program to add two integers using pointers and functions.

#include <stdio.h>

void sum (int\*, int\*, int\*);

int main()

{

int num1, num2, total;

printf("\n Enter the first number : ");

scanf("%d", &num1);

printf("\n Enter the second number : ");

scanf("%d", &num2);

sum(&num1, &num2, &total);

printf("\n Total = %d", total);

return 0;

}

void sum (int \*a, int \*b, int \*t)

{

\*t = \*a + \*b;

}

**Output**

Enter the first number : 23

Enter the second number : 34

Total = 57

**1.11.5 Pointer to Pointers**

In C, you can also use pointers that point to pointers. The pointers in turn point to data or even to other pointers. To declare pointers to pointers, just add an asterisk \* for each level of reference.

**38** *Data Structures Using C*

For example, consider the following code: 

int x=10;

int \*px, \*\*ppx;

px = &x;

ppx = &px;

**Figure 1.12** Pointer to pointer

Now if we write,

printf("\n %d", \*\*ppx);

Let us assume, the memory locations of these variables are as shown in Fig. 1.12.

Then, it would print 10, the value of x.

**1.11.6 Drawbacks of Pointers**

Although pointers are very useful in C, they are not free from limitations. If used incorrectly, pointers can lead to bugs that are difficult to unearth. For example, if you use a pointer to read a memory location but that pointer is pointing to an incorrect location, then you may end up reading a wrong value. An erroneous input always leads to an erroneous output. Thus however efficient your program code may be, the output will always be disastrous. Same is the case when writing a value to a particular memory location.

Let us try to find some common errors when using pointers.

int x, \*px;

x=10;

\*px = 20;

*Error:* Un-initialized pointer. px is pointing to an unknown memory location. Hence it will overwrite that location’s contents and store 20 in it.

int x, \*px;

x=10;

px = x;

*Error:* It should be px = &x;

int x=10, y=20, \*px, \*py;

px = &x, py = &y;

if(px<py)

printf("\n x is less than y");

else

printf("\n y is less than x");

*Error*: It should be if(\*px< \*py)

**Points to Remember**

• C was developed in the early 1970s by Dennis Ritchie at Bell Laboratories.

• Every word in a C program is either an identifier or a keyword. Identifiers are the names given to program elements such as variables and functions. Keywords are reserved words which cannot be used as identifiers.

• C provides four basic data types: char, int, float, and double.

• A variable is defined as a meaningful name given to a data storage location in computer memory. • Standard library function scanf() is used to input data in a specified format.printf()function is used to output data of different types in a specified format. • C supports different types of operators which can be classified into following categories: arithmetic, relational, equality, logical, unary, conditional, bitwise, assignment, comma, and sizeof operators.

• Modulus operator (%) can only be applied on integer operands, and not on float or double operands. • Equality operators have lower precedence than relational operators.

• Like arithmetic expressions, logical expressions are evaluated from left to right.

• Both x++ and ++x increment the value of x, but in the former case, the value of x is returned before it is incremented. Whereas in the latter case, the value of x is returned after it is incremented.

• Conditional operator is also known as ternary operator as it takes three operands.

• Bitwise NOT or complement produces one’s complement of a given binary number.

• Among all the operators, comma operator has the lowest precedence.

• sizeof is a unary operator used to calculate the size of data types. This operator can be applied to all data types.

• While type conversion is done implicitly, typecasting has to be done explicitly by the programmer. Typecasting is done when the value of one data type has to be converted into the value of another data type.

• C supports three types of control statements: decision control statements, iterative statements, and jump statements.

• In a switch statement, if the value of the variable does not match with any of the values of case statements, then default case is executed.

• Iterative statements are used to repeat the execution of a list of statements until the specified expression becomes false.

• The break statement is used to terminate the execution of the nearest enclosing loop in which it appears.

• When the compiler encounters a continue statement, then the rest of the statements in the loop are skipped and the control is unconditionally transferred to the loop-continuation portion of the nearest enclosing loop.

• A C program contains one or more functions, where each function is defined as a group of statements that perform a specific task.

• Every C program contains a main() function which is the starting point of the program. It is the function that is called by the operating system when the user runs the program.

• Function declaration statement identifies a function’s name and the list of arguments that it accepts and the type of data it returns.

*Introduction to C* **39**

• Function definition, on the other hand, consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function. When a function is defined, space is allocated for that function in the memory.

• The moment the compiler encounters a function call, the control jumps to the statements that are a part of the called function. After the called function is executed, the control is returned back to the calling function.

• Placing the function declaration statement prior to its use enables the compiler to make a check on the arguments used while calling that function.

• A function having void as its return type cannot return any value. Similarly, a function having void as its parameter list cannot accept any value.

• Call by value method passes values of the variables to the called function. Therefore, the called function uses a copy of the actual arguments to perform its intended task. This method is used when the function does not need to modify the values of the original variables in the calling function.

• In call by reference method, addresses of the variables are passed by the calling function to the called function. Hence, in this method, a function receives an implicit reference to the argument, rather than a copy of its value. This allows the function to modify the value of the variable and that change is reflected in the calling function as well.

• A pointer is a variable that contains the memory address of another variable.

• The & operator retrieves the address of the variable. • We can ‘dereference’ a pointer, i.e., refer to the value of the variable to which it points by using unary \* operator.

• Null pointer is a special pointer variable that does not point to any variable. This means that a null pointer does not point to any valid memory address. To declare a null pointer we may use the predefined constant NULL.

• A generic pointer is pointer variable that has void as its data type. The generic pointer can point to variables of any data type.

• To declare pointer to pointers, we need to add an asterisk (\*) for each level of reference.

**40** *Data Structures Using C*

**Exercises**

**Review Questions**

**1.** Discuss the structure of a C program.

**2.** Differentiate between declaration and definition. **3.** How is memory reserved using a declaration statement?

**4.** What do you understand by identifiers and keywords?

**5.** Explain the terms variables and constants. How many types of variables are supported by C? **6.** What does the data type of a variable signify? **7.** Write a short note on basic data types that the C language supports.

**8.** Why do we include <stdio.h> in our programs? **9.** What are header files? Explain their significance. **10.** Write short notes on printf and scanf functions. **11.** Write a short note on operators available in C language.

**12.** Draw the operator precedence chart.

**13.** Differentiate between typecasting and type conversion.

**14.** What are decision control statements? Explain in detail.

**15.** Write a short note on the iterative statements that C language supports.

**16.** When will you prefer to work with a switch statement?

**17.** Define function. Why are they needed? **18.** Differentiate between function declaration and function definition.

**19.** Why is function declaration statement placed prior to function definition?

**20.** Explain the concept of making function calls. **21.** Differentiate between call by value and call by reference using suitable examples.

**22.** Write a short note on pointers.

**23.** Explain the difference between a null pointer and a void pointer.

**24.** How are generic pointers different from other pointer variables?

**25.** Write a short note on pointers to pointers.

**Programming Exercises**

**1.** Write a program to read 10 integers. Display these numbers by printing three numbers in a line separated by commas.

**2.** Write a program to print the count of even numbers between 1–200. Also print their sum.

**3.** Write a program to count the number of vowels in a text.

**4.** Write a program to read the address of a user. Dis play the result by breaking it in multiple lines. **5.** Write a program to read two floating point

numbers. Add these numbers and assign the result to an integer. Finally, display the value of all the three variables.

**6.** Write a program to read a floating point number. Display the rightmost digit of the integral part of the number.

**7.** Write a program to calculate simple interest and compound interest.

**8.** Write a program to calculate salary of an employee given his basic pay (to be entered by the user), HRA = 10% of the basic pay, TA = 5% of basic pay. Define HRA and TA as constants and use them to calculate the salary of the employee.

**9.** Write a program to prepare a grocery bill. Enter the name of the items purchased, quantity in which it is purchased, and its price per unit. Then display the bill in the following format:

\*\*\*\*\*\*\*\*\*\*\*\* B I L L \*\*\*\*\*\*\*\*\*\*\*\*

Item Quantity Price Amount

––––––––––––––––––––––––––––––––––––––––––––––––

––––––––––––––––––––––––––––––––––––––––––––––––– Total Amount to be paid

––––––––––––––––––––––––––––––––––––––––––––––––

**10.** Write a C program using printf statement to print BYE in the following format:

BBB Y Y EEEE

B B Y Y E

BBB Y EEEE

B B Y

**11.** Write a program to read an integer. Display the value of that integer in decimal, octal, and hexadecimal notation.

**12.** Write a program that prints a floating point value in exponential format with the following specifications:

(a) correct to two decimal places;

(b) correct to four decimal places; and

(c) correct to eight decimal places.

**13.** Write a program to find the smallest of three integers using functions.

**14.** Write a program to calculate area of a triangle using function.

**15.** Write a program to find whether a number is divisible by two or not using functions.

**16.** Write a program to print ‘Programming in C is Fun’ using pointers.

**17.** Write a program to read a character and print it. Also print its ASCII value. If the character is in lower case, print it in upper case and vice versa. Repeat the process until a ‘\*’ is entered.

**18.** Write a program to add three floating point numbers. The result should contain only two digits after the decimal.

**19.** Write a program to take input from the user and then check whether it is a number or a character. If it is a character, determine whether it is in upper case or lower case. Also print its ASCII value.

**20.** Write a program to display sum and average of numbers from 1 to n. Use for loop.

**21.** Write a program to print all odd numbers from m to n.

**22.** Write a program to print all prime numbers from m to n.

**23.** Write a program to read numbers until –1 is entered and display whether it is an Armstrong number or not.

**24.** Write a program to add two floating point numbers using pointers and functions.

**25.** Write a program to calculate area of a triangle using pointers.

**Multiple-choice Questions**

**1.** The operator which compares two values is (a) Assignment (b) Relational

(c) Unary (d) Equality

**2.** Ternary operator operates on how many operands? (a) 1 (b) 2

(c) 3 (d) 4

**3.** Which operator produces the one’s complement of the given binary value?

(a) Logical AND (b) Bitwise AND (c) Logical OR (d) Bitwise NOT **4.** Which operator has the lowest precedence? (a) Sizeof (b) Unary

(c) Assignment (d) Comma

*Introduction to C* **41**

**5.** Which of the following is the conversion character associated with short integer?

(a) %c (b) %h

(c) %e (d) %f

**6.** Which of the following is not a character constant? (a) ‘A’ (b) “A”

(c) ‘ ’ (d) ‘\*’

**7.** Which of the following is a valid variable name? (a) Initial.Name (b) A+B

(c) $amt (d) Floats

**8.** Which operator cannot be used with floating point numbers?

(a) + (b) –

(c) % (d) \*

**9.** Identify the erroneous expression.

(a) X=y=2, 4; (b) res = ++a \* 5; (c) res = /4; (d) res = a++ –b \*2 **10.** Function declaration statement identifies a function with its

(a) Name

(b) Arguments

(c) Data type of return value

(d) All of these

**11.** Which return type cannot return any value to the calling function?

(a) int (b) float

(c) void (d) double

**12.** Memory is allocated for a function when the function is

(a) declared (b) defined

(c) called (d) returned

**13.** \*(&num) is equivalent to writing

(a) &num (b) \*num

(c) num (d) None of these **14.** Which operator retrieves the lvalue of a variable? (a) & (b) \*

(c) –> (d) None of these **15.** Which operator is used to dereference a pointer? (a) & (b) \*

(c) –> (d) None of these

**True or False**

**1.** We can have only one function in a C program. **2.** Keywords are case sensitive.

**3.** Variable ‘first’ is the same as ‘First’.

**4.** Signed variables can increase the maximum positive range.

**5.** Comment statements are not executed by the compiler.

**42** *Data Structures Using C*

**6.** Equality operators have higher precedence than the relational operators.

**7.** Shifting once to the left multiplies the number by 2. **8.** Decision control statements are used to repeat the execution of a list of statements.

**9.** printf("%d", scanf("%d", &num)); is a valid C statement.

**10.** 1,234 is a valid integer constant.

**11.** A printf statement can generate only one line of output.

**12.** stdio.h is used to store the source code of the program.

**13.** The closing brace of main() is the logical end of the program.

**14.** The declaration section gives instructions to the computer.

**15.** Any valid printable ASCII character can be used for a variable name.

**16.** Underscore can be used anywhere in the variable name.

**17.** void is a data type in C.

**18.** All arithmetic operators have same precedence. **19.** The modulus operator can be used only with integers.

**20.** The calling function always passes parameters to the called function.

**21.** The name of a function is global.

**22.** No function can be declared within the body of another function.

**23.** The & operator retrieves the lvalue of the variable. **24.** Unary increment and decrement operators have greater precedence than the dereference operator. **25.** On 32-bit systems, an integer variable is allocated 4 bytes.

**Fill in the Blanks**

**1.** C was developed by \_\_\_\_\_\_.

**2.** The execution of a C program begins at \_\_\_\_\_\_. **3.** In the memory, characters are stored as \_\_\_\_\_\_.

**4.** return 0 returns 0 to the \_\_\_\_\_\_.

**5.** \_\_\_\_\_\_ finds the remainder of an integer division. **6.** sizeof is a \_\_\_\_\_\_ operator used to calculate the sizes of data types.

**7.** \_\_\_\_\_\_ is also known as forced conversion. **8.** \_\_\_\_\_\_ is executed when the value of the variable does not match with any of the values of the case statement.

**9.** \_\_\_\_\_\_ function prints data on the monitor. **10.** A C program ends with a \_\_\_\_\_\_.

**11.** \_\_\_\_\_\_ causes the cursor to move to the next line. **12.** A variable can be made constant by declaring it with the qualifier \_\_\_\_\_\_ at the time of initializa tion.

**13.** \_\_\_\_\_\_ operator returns the number of bytes occupied by the operand.

**14.** The \_\_\_\_\_\_ specification is used to read/write a short integer.

**15.** The \_\_\_\_\_\_ specification is used to read/write a hexadecimal integer.

**16.** To print the data left-justified, \_\_\_\_\_\_ specif ication is used.

**17.** After the function is executed, the control passes back to the \_\_\_\_\_\_.

**18.** A function that uses another function is known as the \_\_\_\_\_\_.

**19.** The inputs that the function takes are known as \_\_\_\_\_\_.

**20.** Function definition consist of \_\_\_\_\_\_ and \_\_\_\_\_\_. **21.** In \_\_\_\_\_\_ method, address of the variable is passed by the calling function to the called function. **22.** Size of character pointer is \_\_\_\_\_\_.

**23.** \_\_\_\_\_\_ pointer does not point to any valid memory address.

**24.** The \_\_\_\_\_\_ appears on the right side of the assignment statement.

**25.** The \_\_\_\_\_\_ operator informs the compiler that the variable is a pointer variable.

CHAPTER 2

Introduction to

Data Structures

and Algorithms

**LEARNING OBJECTIVE**

In this chapter, we are going to discuss common data structures and algorithms

which serve as building blocks for creating efficient programs. We will also discuss

different approaches to designing algorithms and different notations for evaluating

the performance of algorithms.

**2.1 BASIC TERMINOLOGY**

We have already learnt the basics of programming in C in the previous chapter and know how to write, debug, and run simple programs in C language. Our aim has been to design good programs, where a good program is defined as a program that

∑ runs correctly

∑ is easy to read and understand

∑ is easy to debug *and*

∑ is easy to modify.

A program should undoubtedly give correct results, but along with that it should also run efficiently. A program is said to be efficient when it executes in minimum time and with minimum memory space. In order to write efficient programs we need to apply certain data management concepts.

The concept of data management is a complex task that includes activities like data collection, organization of data into appropriate structures, and developing and maintaining routines for quality assurance.

Data structure is a crucial part of data management and in this book it will be our prime concern. A *data structure* is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

**44** *Data Structures Using C*

Data structures are used in almost every program or software system. Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables. Data structures are widely applied in the following areas:

∑ Compiler design ∑ Operating system

∑ Statistical analysis package ∑ DBMS

∑ Numerical analysis ∑ Simulation

∑ Artificial intelligence ∑ Graphics

When you will study DBMS as a subject, you will realize that the major data structures used in the Network data model is graphs, Hierarchical data model is trees, and RDBMS is arrays. Specific data structures are essential ingredients of many efficient algorithms as they enable the programmers to manage huge amounts of data easily and efficiently. Some formal design methods and programming languages emphasize data structures and the algorithms as the key organizing factor in software design. This is because representing information is fundamental to computer science. The primary goal of a program or software is not to perform calculations or operations but to store and retrieve information as fast as possible.

Be it any problem at hand, the application of an appropriate data structure provides the most efficient solution. A solution is said to be efficient if it solves the problem within the required resource constraints like the total space available to store the data and the time allowed to perform each subtask. And the best solution is the one that requires fewer resources than known alternatives. Moreover, the cost of a solution is the amount of resources it consumes. The cost of a solution is basically measured in terms of one key resource such as time, with the implied assumption that the solution meets the other resource constraints.

Today computer programmers do not write programs just to solve a problem but to write an efficient program. For this, they first analyse the problem to determine the performance goals that must be achieved and then think of the most appropriate data structure for that job. However, program designers with a poor understanding of data structure concepts ignore this analysis step and apply a data structure with which they can work comfortably. The applied data structure may not be appropriate for the problem at hand and therefore may result in poor performance (like slow speed of operations).

Conversely, if a program meets its performance goals with a data structure that is simple to use, then it makes no sense to apply another complex data structure just to exhibit the programmer’s skill. When selecting a data structure to solve a problem, the following steps must be performed.

1. Analysis of the problem to determine the basic operations that must be supported. For example, basic operation may include inserting/deleting/searching a data item from the data structure. 2. Quantify the resource constraints for each operation.

3. Select the data structure that best meets these requirements.

This three-step approach to select an appropriate data structure for the problem at hand supports a data-centred view of the design process. In the approach, the first concern is the data and the operations that are to be performed on them. The second concern is the representation of the data, and the final concern is the implementation of that representation.

There are different types of data structures that the C language supports. While one type of data structure may permit adding of new data items only at the beginning, the other may allow it to be added at any position. While one data structure may allow accessing data items sequentially, the other may allow random access of data. So, selection of an appropriate data structure for the problem is a crucial decision and may have a major impact on the performance of the program.

*Introduction to Data Structures and Algorithms* **45**

**2.1.1 Elementary Data Structure Organization**

Data structures are building blocks of a program. A program built using improper data structures may not work as expected. So as a programmer it is mandatory to choose most appropriate data structures for a program.

The term *data* means a value or set of values. It specifies either the value of a variable or a constant (e.g., marks of students, name of an employee, address of a customer, value of *pi*, etc.). While a data item that does not have subordinate data items is categorized as an elementary item, the one that is composed of one or more subordinate data items is called a group item. For example, a student’s name may be divided into three sub-items—first name, middle name, and last name—but his roll number would normally be treated as a single item.

A *record* is a collection of data items. For example, the name, address, course, and marks obtained are individual data items. But all these data items can be grouped together to form a record.

A *file* is a collection of related records. For example, if there are 60 students in a class, then there are 60 records of the students. All these related records are stored in a file. Similarly, we can have a file of all the employees working in an organization, a file of all the customers of a company, a file of all the suppliers, so on and so forth.

Moreover, each record in a file may consist of multiple data items but the value of a certain data item uniquely identifies the record in the file. Such a data item K is called a *primary key*, and the values K1, K2 ... in such field are called keys or key values. For example, in a student’s record that contains roll number, name, address, course, and marks obtained, the field roll number is a primary key. Rest of the fields (name, address, course, and marks) cannot serve as primary keys, since two or more students may have the same name, or may have the same address (as they might be staying at the same place), or may be enrolled in the same course, or have obtained same marks.

This organization and hierarchy of data is taken further to form more complex types of data structures, which is discussed in Section 2.2.

**2.2 CLASSIFICATION OF DATA STRUCTURES**

Data structures are generally categorized into two classes: *primitive* and *non-primitive* data structures.

***Primitive and Non-primitive Data Structures***

Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms ‘data type’, ‘basic data type’, and ‘primitive data type’ are often used interchangeably.

Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs. Non-primitive data structures can further be classified into two categories: *linear* and *non-linear* data structures.

***Linear and Non-linear Structures***

If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues. Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.

**46** *Data Structures Using C*

However, if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

C supports a variety of data structures. We will now introduce all these data structures and they would be discussed in detail in subsequent chapters.

***Arrays***

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an *index* (also known as the *subscript*).

In C, arrays are declared using the following syntax:

type name[size];

For example,

int marks[10];

The above statement declares an array marks that contains 10 elements. In C, the array index starts from zero. This means that the array marks will contain 10 elements in all. The first element will be stored in marks[0], second element in marks[1], so on and so forth. Therefore, the last element, that is the 10th element, will be stored in marks[9]. In the memory, the array will be stored as shown in Fig. 2.1.

1

st 2

nd 3

rd 4

th 5

th 6

th 7

th 8

th 9

th

10

th

element

element

element

element

element

element

element

element

element

element

marks[0] marks[1] marks[2] marks[3] marks[4] marks[5] marks[6] marks[7] marks[8] marks[9] **Figure 2.1** Memory representation of an array of 10 elements

Arrays are generally used when we want to store large amount of similar type of data. But they have the following limitations:

∑ Arrays are of fixed size.

∑ Data elements are stored in contiguous memory locations which may not be always available. ∑ Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

However, these limitations can be solved by using linked lists. We will discuss more about arrays in Chapter 3.

***Linked Lists***

A linked list is a very flexible, dynamic data structure in which elements (called *nodes*) form a sequential list. In contrast to static arrays, a programmer need not worry about how many elements will be stored in the linked list. This feature enables the programmers to write robust programs which require less maintenance.

In a linked list, each node is allocated space as it is added to the list. Every node in the list points to the next node in the list. Therefore, in a linked list, every node contains the following two types of data:

∑ The value of the node or any other data that corresponds to that node

∑ A pointer or link to the next node in the list

*Introduction to Data Structures and Algorithms* **47**

The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list. Since the memory for a node is dynamically allocated when it is added to the list, the total number of nodes that may be added to a list is limited only by the amount of memory available. Figure 2.2 shows a linked list of seven nodes.

1 2 3 4 5 6 7 X

**Figure 2.2** Simple linked list

**Note** *Advantage*: Easier to insert or delete data elements

*Disadvantage*: Slow search operation and requires more memory space

***Stacks***

A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack. Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.

In the computer’s memory, stacks can be implemented using arrays or linked lists. Figure 2.3 shows the array implementation of a stack. Every stack has a variable top associated with it. top is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted. There is another variable MAX, which is used to store the maximum number of elements that the stack can store.

If top = NULL, then it indicates that the stack is empty and if top = MAX–1, then the stack is full.

A AB ABC ABCD ABCDE

0 1 2 3 top=4 5678 9

**Figure 2.3** Array representation of a stack

In Fig. 2.3, top = 4, so insertions and deletions will be done at this position. Here, the stack can store a maximum of 10 elements where the indices range from 0–9. In the above stack, five more elements can still be stored.

A stack supports three basic operations: push, pop, and peep. The push operation adds an element to the top of the stack. The pop operation removes the element from the top of the stack. And the peep operation returns the value of the topmost element of the stack (without deleting it).

However, before inserting an element in the stack, we must check for overflow conditions. An overflow occurs when we try to insert an element into a stack that is already full. Similarly, before deleting an element from the stack, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a stack that is already empty.

***Queues***

A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the rear and removed from the other end called the front. Like stacks, queues can be implemented by using either arrays or linked lists.

Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively. Consider the queue shown in Fig. 2.4.

**48** *Data Structures Using C*

Front Rear

12 9 7 18 14 36

0 1 2 3 4 5678 9

**Figure 2.4** Array representation of a queue

Here, front = 0 and rear = 5. If we want to add one more value to the list, say, if we want to add another element with the value 45, then the rear would be incremented by 1 and the value would be stored at the position pointed by the rear. The queue, after the addition, would be as shown in Fig. 2.5.

Here, front = 0 and rear = 6. Every time a new element is to be added, we will repeat the same procedure.

Front Rear

12 9 7 18 14 36

45

0 1 2 3 4 5678 9

**Figure 2.5** Queue after insertion of a new element

Now, if we want to delete an element from the queue, then the value of front will be incremented. Deletions are done only from this end of the queue. The queue after the deletion will be as shown in Fig. 2.6.

Front

9 7 18 14 36

Rear 45

0

2 3 4 5 7 8 9 1 6

**Figure 2.6** Queue after deletion of an element

However, before inserting an element in the queue, we must check for overflow conditions. An overflow occurs when we try to insert an element into a queue that is already full. A queue is full when rear = MAX–1, where MAX is the size of the queue, that is MAX specifies the maximum number of elements in the queue. Note that we have written MAX–1 because the index starts from 0.

Similarly, before deleting an element from the queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If front = NULL and rear = NULL, then there is no element in the queue.

***Trees***

A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order. One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.

The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees. Each node contains a data element, a left pointer which points to the left sub-tree, and a right pointer which points to the right sub-tree. The root element is the topmost node which is pointed by a ‘root’ pointer. If root = NULL then the tree is empty.

Figure 2.7 shows a binary tree, where R is the root node and T1 and T2 are the left and right sub trees of R. If T1 is non-empty, then T1 is said to be the left successor of R. Likewise, if T2 is non-empty, then it is called the right successor of R.

T1

R

1

T2

2 3

*Introduction to Data Structures and Algorithms* **49**

In Fig. 2.7, node 2 is the left child and node 3 is the right child of the root node 1. Note that the left sub-tree of the root node consists of the nodes 2, 4, 5, 8, and 9. Similarly, the right sub-tree of the root node consists of the nodes 3, 6, 7, 10, 11, and 12.

4 5 6 7

8 9 10 11 12 **Figure 2.7** Binary tree

**Note** *Advantage*: Provides quick search, insert, and delete operations *Disadvantage*: Complicated deletion algorithm

***Graphs***

A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices. A graph is often viewed

as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist. In a tree structure, nodes can have any number of children but only one parent, a graph on the other hand relaxes all such kinds of restrictions. Figure 2.8 shows a graph with five nodes. A node in the graph may represent a city and the edges connecting the nodes can represent roads. A graph can also be used to represent a computer network where the nodes are workstations and the edges are the network connections. Graphs have so many applications in computer science and mathematics that several algorithms have been written to perform the standard graph operations, such as searching the graph and finding the shortest path between the nodes of a graph. Note that unlike trees, graphs do not have any root node. Rather, every node

A B C

D E

**Figure 2.8** Graph

in the graph can be connected with every another node in the graph. When two nodes are connected via an edge, the two nodes are known as *neighbours*. For example, in Fig. 2.8, node A has two neighbours: B and D.

**Note** *Advantage*: Best models real-world situations

*Disadvantage*: Some algorithms are slow and very complex

**2.3 OPERATIONS ON DATA STRUCTURES**

This section discusses the different operations that can be performed on the various data structures previously mentioned.

***Traversing*** It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.

***Searching*** It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.

***Inserting*** It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.

***Deleting*** It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

***Sorting*** Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants’ scores in descending order and then extracting the top three.

***Merging*** Lists of two sorted data items can be combined to form a single list of sorted data items.

**50** *Data Structures Using C*

Many a time, two or more operations are applied simultaneously in a given situation. For example, if we want to delete the details of a student whose name is X, then we first have to search the list of students to find whether the record of X exists or not and if it exists then at which location, so that the details can be deleted from that particular location.

**2.4 ABSTRACT DATA TYPE**

An *abstract data type* (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job. For example, stacks and queues are perfect examples of an ADT. We can implement both these ADTs using an array or a linked list. This demonstrates the ‘abstract’ nature of stacks and queues.

To further understand the meaning of an abstract data type, we will break the term into ‘data type’ and ‘abstract’, and then discuss their meanings.

***Data type*** Data type of a variable is the set of values that the variable can take. We have already read the basic data types in C include int, char, float, and double.

When we talk about a primitive type (built-in data type), we actually consider two things: a data item with certain characteristics and the permissible operations on that data. For example, an int variable can contain any whole-number value from –32768 to 32767 and can be operated with the operators +, –, \*, and /. In other words, the operations that can be performed on a data type are an inseparable part of its identity. Therefore, when we declare a variable of an abstract data type (e.g., stack or a queue), we also need to specify the operations that can be performed on it.

***Abstract*** The word ‘abstract’ in the context of data structures means *considered apart from the detailed specifications or implementation*.

In C, an abstract data type can be a structure considered without regard to its implementation. It can be thought of as a ‘description’ of the data in the structure with a list of operations that can be performed on the data within that structure.

The end-user is not concerned about the details of how the methods carry out their tasks. They are only aware of the methods that are available to them and are only concerned about calling those methods and getting the results. They are not concerned about how they work.

For example, when we use a stack or a queue, the user is concerned only with the type of data and the operations that can be performed on it. Therefore, the fundamentals of how the data is stored should be invisible to the user. They should not be concerned with how the methods work or what structures are being used to store the data. They should just know that to work with stacks, they have push() and pop() functions available to them. Using these functions, they can manipulate the data (insertion or deletion) stored in the stack.

***Advantage of using ADTs***

In the real world, programs *evolve* as a result of new requirements or constraints, so a modification to a program commonly requires a change in one or more of its data structures. For example, if you want to add a new field to a student’s record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program’s efficiency. In such a scenario, rewriting every procedure that uses the changed structure is not desirable. Therefore, a better alternative is to *separate* the use of a data structure from the details of its implementation. This is the principle underlying the use of abstract data types.

**2.5 ALGORITHMS**

The typical definition of algorithm is ‘a formally defined procedure for performing some calculation’. If a procedure is formally defined, then it can be implemented using a formal language,

*Introduction to Data Structures and Algorithms* **51**

and such a language is known as a *programming language*. In general terms, an algorithm provides a blueprint to write a program to solve a particular problem. It is considered to be an effective procedure for solving a problem in finite number of steps. That is, a well-defined algorithm always provides an answer and is guaranteed to terminate.

Algorithms are mainly used to achieve *software reuse*. Once we have an idea or a blueprint of a solution, we can implement it in any high-level language like C, C++, or Java. An algorithm is basically a set of instructions that solve a problem. It is not uncommon to have multiple algorithms to tackle the same problem, but the choice of a particular algorithm must depend on the time and space complexity of the algorithm.

**2.6 DIFFERENT APPROACHES TO DESIGNING AN ALGORITHM**

Algorithms are used to manipulate the data contained in data structures. When working with data structures, algorithms are used to perform operations on the stored data.

A complex algorithm is often divided into smaller units called modules. This process of dividing an algorithm into modules is called modularization. The key advantages of modularization are as follows:

∑ It makes the complex algorithm simpler to design and implement.

∑ Each module can be designed independently. While designing one module, the details of other modules can be ignored, thereby enhancing clarity in design which in turn simplifies implementation, debugging, testing, documenting, and maintenance of the overall algorithm.

There are two main approaches to design an algorithm—top-down approach and bottom-up approach, as shown in Fig. 2.9.

Top-down approach

Complex algorithm

Module 1 Module 2 Module n Each module can be divided into one or more sub-modules

Bottom-up approach

**Figure 2.9** Different approaches of designing an algorithm

***Top-down approach*** A top-down design approach starts by dividing the complex algorithm into one or more modules. These modules can further be decomposed into one or more sub-modules, and this process of decomposition is iterated until the desired level of module complexity is achieved. Top-down design method is a form of stepwise refinement where we begin with the topmost module and incrementally add modules that it calls.

Therefore, in a top-down approach, we start from an abstract design and then at each step, this design is refined into more concrete levels until a level is reached that requires no further refinement.

***Bottom-up approach*** A bottom-up approach is just the reverse of top-down approach. In the bottom-up design, we start with designing the most basic or concrete modules and then proceed towards designing higher level modules. The higher level modules are implemented by using the operations performed by lower level modules. Thus, in this approach sub-modules are grouped together to form a higher level module. All the higher level modules are clubbed together to form even higher level modules. This process is repeated until the design of the complete algorithm is obtained.

**52** *Data Structures Using C*

***Top-down vs bottom-up approach*** Whether the top-down strategy should be followed or a bottom-up is a question that can be answered depending on the application at hand. While top-down approach follows a stepwise refinement by decomposing the algorithm into manageable modules, the bottom-up approach on the other hand defines a module and then groups together several modules to form a new higher level module.

Top-down approach is highly appreciated for ease in documenting the modules, generation of test cases, implementation of code, and debugging. However, it is also criticized because the sub-modules are analysed in isolation without concentrating on their communication with other modules or on reusability of components and little attention is paid to data, thereby ignoring the concept of information hiding.

Although the bottom-up approach allows information hiding as it first identifies what has to be encapsulated within a module and then provides an abstract interface to define the module’s boundaries as seen from the clients. But all this is difficult to be done in a strict bottom-up strategy. Some top-down activities need to be performed for this.

All in all, design of complex algorithms must not be constrained to proceed according to a fixed pattern but should be a blend of top-down and bottom-up approaches.

**2.7 CONTROL STRUCTURES USED IN ALGORITHMS**

An algorithm has a finite number of steps. Some steps may involve decision-making and repetition. Broadly speaking, an algorithm may employ one of the following control structures: (a) sequence, (b) decision, and (c) repetition.

***Sequence***

Step 1: Input first number as A Step 2: Input second number as B Step 3: SET SUM = A+B

Step 4: PRINT SUM

Step 5: END

**Figure 2.10** Algorithm to add two numbers

By sequence, we mean that each step of an algorithm is executed in a specified order. Let us write an algorithm to add two numbers. This algorithm performs the steps in a purely sequential order, as shown in Fig. 2.10.

***Decision***

Decision statements are used when the execution of a process depends on the outcome of some condition. For

example, if x = y, then print EQUAL. So the general form of IF construct can be given as: IF *condition* Then *process*

A condition in this context is any statement that may evaluate to either a true value or a false value. In the above example, a variable x can be either equal to y or not equal to y. However, it cannot be both true and false. If the condition is true, then the process is executed. A decision statement can also be stated in the following manner:

IF *condition*

Then *process1*

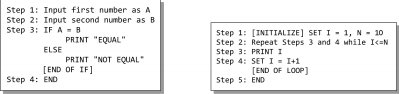
ELSE *process2*

This form is popularly known as the IF–ELSE construct. Here, if the condition is true, then process1 is executed, else process2 is executed. Figure 2.11 shows an algorithm to check if two numbers are equal.

***Repetition***

Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as while, do–while, and for loops. These loops execute one or more steps until some condition is true. Figure 2.12 shows an algorithm that prints the first 10 natural numbers.

*Introduction to Data Structures and Algorithms* **53**

** Figure 2.11** Algorithm to test for equality of **Figure 2.12** Algorithm to print the first 10 natural of two numbers

**Programming Examples**

**1.** Write an algorithm for swapping two values.

Step 1: Input first number as A

Step 2: Input second number as B

Step 3: SET TEMP = A

Step 4: SET A = B

Step 5: SET B = TEMP

Step 6: PRINT A, B

Step 7: END

**2.** Write an algorithm to find the larger of two numbers.

Step 1: Input first number as A

Step 2: Input second number as B

Step 3: IF A>B

PRINT A

ELSE

IF A<B

PRINT B

ELSE

PRINT "The numbers are equal"

[END OF IF]

[END OF IF]

Step 4: END

**3.** Write an algorithm to find whether a number is even or odd.

Step 1: Input number as A

Step 2: IF A%2 =0

PRINT "EVEN"

ELSE

PRINT "ODD"

[END OF IF]

Step 3: END

**4.** Write an algorithm to print the grade obtained by a student using the following rules.

Step 1: Enter the Marks obtained as M Step 2: IF M>75

PRINT O

Step 3: IF M>=60 AND M<75

PRINT A

Step 4: IF M>=50 AND M<60

PRINT B

Step 5: IF M>=40 AND M<50

PRINT C

ELSE

PRINT D

**Marks Grade** Above 75 O 60–75 A

50–59 B

40–49 C

Less then 40 D

**54** *Data Structures Using C*

[END OF IF]

Step 6: END

**5.** Write an algorithm to find the sum of first N natural numbers.

Step 1: Input N

Step 2: SET I = 1, SUM = 0

Step 3: Repeat Step 4 while I <= N

Step 4: SET SUM = SUM + I

SET I = I + 1

[END OF LOOP]

Step 5: PRINT SUM

Step 6: END

**2.8 TIME AND SPACE COMPLEXITY**

Analysing an algorithm means determining the amount of resources (such as time and memory) needed to execute it. Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity.

The *time complexity* of an algorithm is basically the running time of a program as a function of the input size. Similarly, the *space complexity* of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.

In other words, the number of machine instructions which a program executes is called its time complexity. This number is primarily dependent on the size of the program’s input and the algorithm used.

Generally, the space needed by a program depends on the following two parts: ∑ *Fixed part*: It varies from problem to problem. It includes the space needed for storing instructions, constants, variables, and structured variables (like arrays and structures). ∑ *Variable part*: It varies from program to program. It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

However, running time requirements are more critical than memory requirements. Therefore, in this section, we will concentrate on the running time efficiency of algorithms.

**2.8.1 Worst-case, Average-case, Best-case, and Amortized Time Complexity *Worst-case running time*** This denotes the behaviour of an algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

***Average-case running time*** The average-case running time of an algorithm is an estimate of the running time for an ‘average’ input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.

***Best-case running time*** The term ‘best-case performance’ is used to analyse an algorithm under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list. However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance. It is always recommended to improve the average performance and the worst-case performance of an algorithm.

***Amortized running time*** Amortized running time refers to the time required to perform a sequence of (related) operations averaged over all the operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.

*Introduction to Data Structures and Algorithms* **55**

**2.8.2 Time–Space Trade-off**

The best algorithm to solve a particular problem at hand is no doubt the one that requires less memory space and takes less time to complete its execution. But practically, designing such an ideal algorithm is not a trivial task. There can be more than one algorithm to solve a particular problem. One may require less memory space, while the other may require less CPU time to execute. Thus, it is not uncommon to sacrifice one thing for the other. Hence, there exists a

time–space trade-off among algorithms.

So, if space is a big constraint, then one might choose a program that takes less space at the cost of more CPU time. On the contrary, if time is a major constraint, then one might choose a program that takes minimum time to execute at the cost of more space.

**2.8.3 Expressing Time and Space Complexity**

The time and space complexity can be expressed using a function f(n) where n is the input size for a given instance of the problem being solved. Expressing the complexity is required when

∑ We want to predict the rate of growth of complexity as the input size of the problem increases. ∑ There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.

The most widely used notation to express this function f(n) is the Big O notation. It provides the upper bound for the complexity.

**2.8.4 Algorithm Efficiency**

If a function is linear (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains. However, if an algorithm contains loops, then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm.

Let us consider different cases in which loops determine the efficiency of an algorithm.

***Linear Loops***

To calculate the efficiency of an algorithm that has a single loop, we need to first determine the number of times the statements in the loop will be executed. This is because the number of iterations is directly proportional to the loop factor. Greater the loop factor, more is the number of iterations. For example, consider the loop given below:

for(i=0;i<100;i++)

statement block;

Here, 100 is the loop factor. We have already said that efficiency is directly proportional to the number of iterations. Hence, the general formula in the case of linear loops may be given as

f(n) = n

However calculating efficiency is not as simple as is shown in the above example. Consider the loop given below:

for(i=0;i<100;i+=2)

statement block;

Here, the number of iterations is half the number of the loop factor. So, here the efficiency can be given as

f(n) = n/2

**56** *Data Structures Using C*

***Logarithmic Loops***

We have seen that in linear loops, the loop updation statement either adds or subtracts the loop-controlling variable. However, in logarithmic loops, the loop-controlling variable is either multiplied or divided during each iteration of the loop. For example, look at the loops given below:

for(i=1;i<1000;i\*=2) for(i=1000;i>=1;i/=2)

statement block; statement block;

Consider the first for loop in which the loop-controlling variable i is multiplied by 2. The loop will be executed only 10 times and not 1000 times because in each iteration the value of i doubles. Now, consider the second loop in which the loop-controlling variable i is divided by 2. In this case also, the loop will be executed 10 times. Thus, the number of iterations is a function of the number by which the loop-controlling variable is divided or multiplied. In the examples discussed, it is 2. That is, when n = 1000, the number of iterations can be given by log 1000 which is approximately equal to 10.

Therefore, putting this analysis in general terms, we can conclude that the efficiency of loops in which iterations divide or multiply the loop-controlling variables can be given as

f(n) = log n

***Nested Loops***

Loops that contain loops are known as *nested loops*. In order to analyse nested loops, we need to determine the number of iterations each loop completes. The total is then obtained as the product of the number of iterations in the inner loop and the number of iterations in the outer loop.

In this case, we analyse the efficiency of the algorithm based on whether it is a linear logarithmic, quadratic, or dependent quadratic nested loop.

***Linear logarithmic loop*** Consider the following code in which the loop-controlling variable of the inner loop is multiplied after each iteration. The number of iterations in the inner loop is log 10. This inner loop is controlled by an outer loop which iterates 10 times. Therefore, according to the formula, the number of iterations for this code can be given as 10 log 10.

for(i=0;i<10;i++)

for(j=1; j<10;j\*=2)

statement block;

In more general terms, the efficiency of such loops can be given as f(n) = n log n. ***Quadratic loop*** In a quadratic loop, the number of iterations in the inner loop is equal to the number of iterations in the outer loop. Consider the following code in which the outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times. Therefore, the efficiency here is 100.

for(i=0;i<10;i++)

for(j=0; j<10;j++)

statement block;

The generalized formula for quadratic loop can be given as f(n) = n2.

***Dependent quadratic loop*** In a dependent quadratic loop, the number of iterations in the inner loop is dependent on the outer loop. Consider the code given below:

for(i=0;i<10;i++)

for(j=0; j<=i;j++)

statement block;

In this code, the inner loop will execute just once in the first iteration, twice in the second iteration, thrice in the third iteration, so on and so forth. In this way, the number of iterations can be calculated as

*Introduction to Data Structures and Algorithms* **57**

1 + 2 + 3 + ... + 9 + 10 = 55

If we calculate the average of this loop (55/10 = 5.5), we will observe that it is equal to the number of iterations in the outer loop (10) plus 1 divided by 2. In general terms, the inner loop iterates (n + 1)/2 times. Therefore, the efficiency of such a code can be given as

f(n) = n (n + 1)/2

**2.9 BIG O NOTATION**

In today’s era of massive advancement in computer technology, we are hardly concerned about the efficiency of algorithms. Rather, we are more interested in knowing the generic order of the magnitude of the algorithm. If we have two different algorithms to solve the same problem where one algorithm executes in 10 iterations and the other in 20 iterations, the difference between the two algorithms is not much. However, if the first algorithm executes in 10 iterations and the other in 1000 iterations, then it is a matter of concern.

We have seen that the number of statements executed in the program for n elements of the data is a function of the number of elements, expressed as f(n). Even if the expression derived for a function is complex, a dominant factor in the expression is sufficient to determine the order of the magnitude of the result and, hence, the efficiency of the algorithm. This factor is the Big O, and is expressed as O(n).

The Big O notation, where O stands for ‘order of’, is concerned with what happens for very large values of n. For example, if a sorting algorithm performs n2 operations to sort just n elements, then that algorithm would be described as an O(n2) algorithm.

When expressing complexity using the Big O notation, constant multipliers are ignored. So, an O(4n) algorithm is equivalent to O(n), which is how it should be written.

If f(n) and g(n) are the functions defined on a positive integer number n, then

f(n) = O(g(n))

That is, f of n is Big–O of g of n if and only if positive constants c and n exist, such that f(n)≤cg(n). It means that for large amounts of data, f(n) will grow no more than a constant factor than g(n). Hence, g provides an upper bound. Note that here c is a constant which depends on the following factors:

∑ the programming language used,

∑ the quality of the compiler or interpreter,

∑ the CPU speed,

∑ the size of the main memory and the access time to it,

∑ the knowledge of the programmer, and

∑ the algorithm itself, which may require simple but also time-consuming machine instructions. We have seen that the Big O notation provides a strict upper bound for f(n). This means that the function f(n) can do better but not worse than the specified value. Big O notation is simply written as f(n) ∈ O(g(n)) or as f(n) = O(g(n))*.*

Here, n is the problem size and O(g(n)) = {h(n): ∃ positive constants c, n0 such that 0 ≤ h (n) ≤ cg(n), ∀ n ≥ n0}. Hence, we can say that O(g(n)) comprises a set of all the functions h(n) that are less than or equal to cg(n) for all values of n ≥ n0.

If f(n) ≤ cg(n), c > 0, ∀ n ≥ n0, then f(n) = O(g(n)) and g(n) is an asymptotically tight upper bound for f(n).

Examples of functions in O(n3) include: n2.9, n3, n3 + n, 540n3 + 10.

**58** *Data Structures Using C*

Examples of functions not in O(n3) include: n3.2, n2, n2 + n, 540n + 10, 2n

To summarize,

• Best case O describes an upper bound for all combinations of input. It is possibly lower than the worst case. For example, when sorting an array the best case is when the array is already correctly sorted.

• Worst case O describes a lower bound for worst case input combinations. It is possibly greater than the best case. For example, when sorting an array the worst case is when the array is sorted in reverse order.

**Table 2.1** Examples of f(n) and g(n) **g(n) f(n) = O(g(n))** 10 O(1) 2n3 + 1 O(n3) 3n2 + 5 O(n2) 2n3 + 3n2 + 5n – 10 O(n3)

***Categories of Algorithms***

• If we simply write O, it means same as worst case O. Now let us look at some examples of g(n) and f(n). Table 2.1 shows the relationship between g(n) and f(n). Note that the constant values will be ignored because the main purpose of the Big O notation is to analyse the algorithm in a general fashion, so the anomalies that appear for small input sizes are simply ignored.

According to the Big O notation, we have five different categories of algorithms:

∑ Constant time algorithm: running time complexity given as O(1)

∑ Linear time algorithm: running time complexity given as O(n)

∑ Logarithmic time algorithm: running time complexity given as O(log n) ∑ Polynomial time algorithm: running time complexity given as O(nk) where k > 1 ∑ Exponential time algorithm: running time complexity given as O(2n) Table 2.2 shows the number of operations that would be performd for various values of n.

**Table 2.2** Number of operations for different functions of n

| **O(1)** | **O(log n)** | **O(n)** | **O(n log n)** | **O(n2)** |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 2 | 2 | 4 |
| 1 | 2 | 4 | 8 | 16 |
| 1 | 3 | 8 | 24 | 64 |
| 1 | 4 | 16 | 64 | 256 |

**n O(n3)** 1 1 2 8 4 64 8 512 16 4096

**Example 2.1** Show that 4n2 = O(n3).

***Solution*** By definition, we have

0 ≤ h(n) ≤ cg(n)

Substituting 4n2 as h(n) and n3 as g(n), we get

0 ≤ 4n2 ≤ cn3

Dividing by n3

0/n3 ≤ 4n2/n3 ≤ cn3/n3

0 ≤ 4/n ≤ c

Now to determine the value of c, we see that 4/n is maximum when n=1. Therefore, c=4. To determine the value of n0,

0 ≤ 4/n0 ≤ 4

0 ≤ 4/4 ≤ n0

*Introduction to Data Structures and Algorithms* **59**

0 ≤ 1 ≤ n0

This means n0=1. Therefore, 0 ≤ 4n2 ≤ 4n3, ∀ n ≥ n0=1.

**Example 2.2** Show that 400n3 + 20n2 = O(n3).

***Solution*** By definition, we have

0 ≤ h(n) ≤ cg(n)

Substituting 400n3 + 20n2 as h(n) and n3 as g(n), we get

0 ≤ 400n3 + 20n2 ≤ cn3

Dividing by n3

0/n3 ≤ 400n3/n3 + 20n2/n3 ≤ cn3/n3

0 ≤ 400 + 20/n ≤ c

Note that 20/n → 0 as n → ∞, and 20/n is maximum when n = 1. Therefore,

0 ≤ 400 + 20/1 ≤ c

This means, c = 420

To determine the value of n0,

0 ≤ 400 + 20/n0 ≤ 420

–400 ≤ 400 + 20/n0 – 400 ≤ 420 – 400

–400 ≤ 20/n0 ≤ 20

–20 ≤ 1/n0 ≤ 1

–20 n0 ≤ 1 ≤ n0. This implies n0 = 1.

Hence, 0 ≤ 400n3 + 20n2 ≤ 420n3 ∀ n ≥ n0=1.

**Example 2.3** Show that n = O(nlogn).

***Solution*** By definition, we have

0 ≤ h(n) ≤ cg(n)

Substituting n as h(n) and nlogn as g(n), we get

0 ≤ n ≤ c n log n

Dividing by nlogn, we get

0/n log n ≤ n/n log n ≤ c n log n/ n log n

0 ≤ 1/log n ≤ c

We know that 1/log n → 0 as n → ∞

To determine the value of c, it is clearly evident that 1/log n is greatest when n=2. Therefore, 0 ≤ 1/log 2 ≤ c = 1. Hence c = 1.

To determine the value of n0, we can write

0 ≤ 1/log n0 ≤ 1

0 ≤ 1≤ log n0

Now, log n0 = 1, when n0 = 2.

Hence, 0 ≤ n ≤ cn log n when c= 1 and ∀ n ≥ n0=2.

**Example 2.4** Show that 10n3 + 20n ≠ O(n2).

***Solution*** By definition, we have

0 ≤ h(n) ≤ cg(n)

Substituting 10n3 + 20n as h(n) and n2 as g(n), we get

0 ≤ 10n3 + 20n ≤ cn2

Dividing by n2

0/n2 ≤ 10n3/n2 + 20n/n2 ≤ cn2/n2

0 ≤ 10n + 20/n ≤ c

0 ≤ (10n2 + 20)/n ≤ c

Hence, 10n3 + 20n ≠ O2(n2)

**60** *Data Structures Using C*

***Limitations of Big O Notation***

There are certain limitations with the Big O notation of expressing the complexity of algorithms. These limitations are as follows:

∑ Many algorithms are simply too hard to analyse mathematically.

∑ There may not be sufficient information to calculate the behaviour of the algorithm in the average case.

∑ Big O analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not consider the programming effort.

∑ It ignores important constants. For example, if one algorithm takes O(n2) time to execute and the other takes O(100000n2) time to execute, then as per Big O, both algorithm have equal time complexity. In real-time systems, this may be a serious consideration.

**2.10 OMEGA NOTATION (**Ω**)**

The Omega notation provides a tight lower bound for f(n). This means that the function can never do better than the specified value but it may do worse.

Ω notation is simply written as, f(n) ∈ Ω(g(n)), where n is the problem size and Ω(g(n)) = {h(n): ∃ positive constants c > 0, n0 such that 0 ≤ cg(n) ≤ h(n), ∀ n ≥ n0}. Hence, we can say that Ω(g(n)) comprises a set of all the functions h(n) that are greater than or equal to cg(n) for all values of n ≥ n0.

If cg(n) ≤ f(n), c > O, ∀ n ≥ nO, then f(n) ∈ Ω(g(n)) and g(n) is an asymptotically tight lower bound for f(n).

Examples of functions in Ω(n2) include: n2, n2.9, n3 + n2, n3

Examples of functions not in Ω(n3) include: n, n2.9, n2

To summarize,

• Best case Ω describes a lower bound for all combinations of input. This implies that the function can never get any better than the specified value. For example, when sorting an array the best case is when the array is already correctly sorted.

• Worst case Ω describes a lower bound for worst case input combinations. It is possibly greater than best case. For example, when sorting an array the worst case is when the array is sorted in reverse order.

• If we simply write Ω, it means same as best case Ω.

**Example 2.5** Show that 5n2 + 10n = Ω(n2).

***Solution*** By the definition, we can write

0 ≤ cg(n) ≤ h(n)

0 ≤ cn2 ≤ 5n2 + 10n

Dividing by n2

0/n2 ≤ cn2/n2 ≤ 5n2/n2 + 10n/n2

0 ≤ c ≤ 5 + 10/n

Now, lim*n*∅•5 +10/n = 5.

Therefore, 0 ≤ c ≤ 5.

Hence, c = 5

Now to determine the value of n0

0 ≤ 5 ≤ 5 + 10/n0

–5 ≤ 5 – 5 ≤ 5 + 10/n0 – 5

*Introduction to Data Structures and Algorithms* **61**

–5 ≤ 0 ≤ 10/n0

So n0 = 1 as lim*n*∅•1/n = 0

Hence, 5n2 + 10n = Ω(n2) for c=5 and ∀ n ≥ nO=1.

**Example 2.6** Show that 7n ≠ Ω(n2).

***Solution*** By the definition, we can write

0 ≤ cg(n) ≤ h(n)

0 ≤ cn2 ≤ 7n

Dividing by n2, we get

0/n2 ≤ cn2/n2 ≤ 7n/n2

0 ≤ c ≤ 7/n

Thus, from the above statement, we see that the value of c depends on the value of n. There does not exist a value of n0 that satisfies the condition as n increases. This could fairly be possible if c = 0 but it is not allowed as the definition by itself says that lim*n*∅• 1 / n = 0.

**2.11 THETA NOTATION (**Θ**)**

Theta notation provides an asymptotically tight bound for f(n). Θ notation is simply written as, f(n) ∈ Θ(g(n)), where n is the problem size and

Θ(g(n)) = {h(n): ∃ positive constants c1, c2, and n0 such that 0 ≤ c1g(n) ≤ h(n) ≤ c2g(n), ∀ n ≥ n0}. Hence, we can say that Θ(g(n)) comprises a set of all the functions h(n) that are between c1g(n) and c2g(n) for all values of n ≥ n0.

If f(n) is between c1g(n) and c2g(n), ∀ n ≥ n0,then f(n) ∈ Θ(g(n)) and g(n) is an asymptotically tight bound for f(n) and f(n) is amongst h(n) in the set.

To summarize,

• The best case in Θ notation is not used.

• Worst case Θ describes asymptotic bounds for worst case combination of input values. • If we simply write Θ, it means same as worst case Θ.

**Example 2.7** Show that n2/2 – 2n = Θ(n2).

***Solution*** By the definition, we can write

c1g(n) ≤ h(n) ≤ c2g(n)

c1n2 ≤ n2/2 – 2n ≤ c2n2

Dividing by n2, we get

c1n2/n2 ≤ n2/2n2 – 2n/n2 ≤ c2n2/n2

c1 ≤ 1/2 – 2/n ≤ c2

This means c2 = 1/2 because lim*n*∅•1/2 – 2/n = 1/2 (Big O notation)

To determine c1 using Ω notation, we can write

0 < c1 ≤ 1/2 – 2/n

We see that 0 < c1 isminimum when n = 5. Therefore,

0 < c1 ≤ 1/2 – 2/5

Hence, c1 = 1/10

Now let us determine the value of n0

1/10 ≤ 1/2 – 2/n0 ≤ 1/2

2/n0 ≤ 1/2 – 1/10 ≤ 1/2

2/n0 ≤ 2/5 ≤ 1/2

**62** *Data Structures Using C*

n0 ≥ 5

You may verify this by substituting the values as shown below.

c1n2 ≤ n2/2 – 2n ≤ c2n2

c1 = 1/10, c2 = 1/2 and n0 = 5

1/10(25) ≤ 25/2 – 20/2 ≤ 25/2

5/2 ≤ 5/2 ≤ 25/2

Thus, in general, we can write, 1/10n2 ≤ n2/2 – 2n ≤ 1/2n2 for n ≥ 5.

**2.12 OTHER USEFUL NOTATIONS**

There are other notations like little o notation and little ω notation which have been discussed below.

***Little o Notation***

This notation provides a non-asymptotically tight upper bound for f(n). To express a function using this notation, we write

f(n) ∈ o(g(n)) where

o(g(n)) = {h(n) : ∃ positive constants c, n0 such that for any c > 0, n0 > 0, and 0 ≤ h(n) ≤ cg(n), ∀ n ≥ n0}.

This is unlike the Big O notation where we say for some c > 0 (not any). For example, 5n3 = O(n3) is asymptotically tight upper bound but 5n2 = o(n3) is non-asymptotically tight bound for f(n). Examples of functions in o(n3) include: n2.9, n3 / log n, 2n2

Examples of functions not in o(n3) include: 3n3, n3, n3 / 1000

**Example 2.8** Show that n3 / 1000 ≠ o(n3).

***Solution*** By definition, we have

0 ≤ h(n) < cg(n), for any constant c > 0

0 ≤ n3 / 1000 ≤ cn3

This is in contradiction with selecting any c < 1/1000.

An imprecise analogy between the asymptotic comparison of functions f(n) and g(n) and the relation between their values can be given as:

f(n) = O(g(n)) ≈ f(n) ≤ g(n) f(n) = o(g(n)) ≈ f(n) < g(n) f(n) = Θ(g(n)) ≈ f(n) = g(n)

***Little Omega Notation (***ω***)***

This notation provides a non-asymptotically tight lower bound for f(n). It can be simply written as, f(n) ∈ ω(g(n)), where

ω(g(n)) = {h(n) : ∃ positive constants c, n0 such that for any c > 0, n0 > 0, and 0 ≤ cg(n) < h(n),∀ n ≥ n0}. This is unlike the Ω notation where we say for some c > 0 (not any). For example, 5n3 = Ω(n3) is asymptotically tight upper bound but 5n2 = ω(n3) is non-asymptotically tight bound for f(n). Example of functions in ω(g(n)) include: n3 = ω(n2), n3.001 = ω(n3), n2logn = ω(n2) Example of a function not in ω(g(n)) is 5n2 ≠ ω(n2) (just as 5≠5)

**Example 2.9** Show that 50n3/100 ≠ ω(n3).

***Solution*** By definition, we have

0 ≤ cg(n) < h(n) , for any constant c > 0

0 ≤ cn3 < 50n3/100

Dividing by n3, we get

*Introduction to Data Structures and Algorithms* **63**

0 ≤ c < 50/100

This is a contradictory value as for any value of c as it cannot be assured to be less than 50/100 or 1/2.

An imprecise analogy between the asymptotic comparison of functions f(n) and g(n) and the relation between their values can be given as:

f(n) = Ω(g(n)) ≈ f(n) ≥ g(n) f(n) = ω(g(n)) ≈ f(n) > g(n) **Points to Remember**

• A data structure is a particular way of storing and organizing data either in computer’s memory or on the disk storage so that it can be used efficiently.

• There are two types of data structures: primitive and non-primitive data structures. Primitive data structures are the fundamental data types which are supported by a programming language. Non

primitive data structures are those data structures which are created using primitive data structures. • Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.

• If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. However, if the elements of a data structure are not stored in sequential order, then it is a non-linear data structure.

• An array is a collection of similar data elements which are stored in consecutive memory locations. • A linked list is a linear data structure consisting of a group of elements (called nodes) which together represent a sequence.

• A stack is a last-in, first-out (LIFO) data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack.

• A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first to be taken out. The elements in a queue are added at one end called the rear and removed from the other end called the front.

**Exercises**

**Review Questions**

• A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical tree structure.

• The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub trees, where both sub-trees are also binary trees.

• A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships can exist between the nodes.

• An abstract data type (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job.

• An algorithm is basically a set of instructions that solve a problem.

• The time complexity of an algorithm is basically the running time of the program as a function of the input size.

• The space complexity of an algorithm is the amount of computer memory required during the program execution as a function of the input size.

• The worst-case running time of an algorithm is an upper bound on the running time for any input. • The average-case running time specifies the expected

behaviour of the algorithm when the input is randomly drawn from a given distribution. • Amortized analysis guarantees the average perfor mance of each operation in the worst case. • The efficiency of an algorithm is expressed in terms of the number of elements that has to be processed and the type of the loop that is being used.

**1.** Explain the features of a good program. **2.** Define the terms: data, file, record, and primary key.

**64** *Data Structures Using C*

**3.** Define data structures. Give some examples. **4.** In how many ways can you categorize data structures? Explain each of them.

**5.** Discuss the applications of data structures. **6.** Write a short note on different operations that can be performed on data structures.

**7.** Compare a linked list with an array.

**8.** Write a short note on abstract data type. **9.** Explain the different types of data structures. Also discuss their merits and demerits.

**10.** Define an algorithm. Explain its features with the help of suitable examples.

**11.** Explain and compare the approaches for designing an algorithm.

**12.** What is modularization? Give its advantages. **13.** Write a brief note on trees as a data structure. **14.** What do you understand by a graph?

**15.** Explain the criteria that you will keep in mind while choosing an appropriate algorithm to solve a particular problem.

**16.** What do you understand by time–space trade-off? **17.** What do you understand by the efficiency of an algorithm?

**18.** How will you express the time complexity of a given algorithm?

**19.** Discuss the significance and limitations of the Big O notation.

**20.** Discuss the best case, worst case, average case, and amortized time complexity of an algorithm. **21.** Categorize algorithms based on their running time complexity.

**22.** Give examples of functions that are in Big O notation as well as functions that are not in Big O notation.

**23.** Explain the little o notation.

**24.** Give examples of functions that are in little o notation as well as functions that are not in little o notation.

**25.** Differentiate between Big O and little o notations. **26.** Explain the Ω notation.

**27.** Give examples of functions that are in Ω notation as well as functions that are not in Ω notation. **28.** Explain the Θ notation.

**29.** Give examples of functions that are in Θ notation as well as functions that are not in Θ notation. **30.** Explain the ω notation.

**31.** Give examples of functions that are in ω notation as well as functions that are in ω notation.

**32.** Differentiate between Big omega and little omega notations.

**33.** Show that n2 + 50n = O(n2).

**34.** Show that n2+n2+n2 = 3n2 = O(n3).

**35.** Prove that n3 ≠ O(n2).

**36.** Show that √n = Ω(lg n).

**37.** Prove that 3n + 5 ≠ Ω(n2).

**38.** Show that ½n2 – 3n ∈ Θ(n2).

**Multiple-choice Questions**

**1.** Which data structure is defined as a collection of similar data elements?

(a) Arrays (b) Linked lists (c) Trees (d) Graphs

**2.** The data structure used in hierarchical data model is

(a) Array (b) Linked list

(c) Tree (d) Graph

**3.** In a stack, insertion is done at

(a) Top (b) Front

(c) Rear (d) Mid

**4.** The position in a queue from which an element is deleted is called as

(a) Top (b) Front

(c) Rear (d) Mid

**5.** Which data structure has fixed size?

(a) Arrays (b) Linked lists (c) Trees (d) Graphs

**6.** If TOP = MAX–1, then that the stack is

(a) Empty (b) Full

(c) Contains some data (d) None of these **7.** Which among the following is a LIFO data structure?

(a) Stacks (b) Linked lists (c) Queues (d) Graphs

**8.** Which data structure is used to represent complex relationships between the nodes?

(a) Arrays (b) Linked lists (c) Trees (d) Graphs

**9.** Examples of linear data structures include (a) Arrays (b) Stacks

(c) Queues (d) All of these **10.** The running time complexity of a linear time algorithm is given as

(a) O(1) (b) O(n)

(c) O(n log n) (d) O(n2)

**11.** Which notation provides a strict upper bound for f(n)?

(a) Omega notation (b) Big O notation (c) Small o notation (d) Theta Notation **12.** Which notation comprises a set of all functions

h(n) that are greater than or equal to cg(n) for all values of n ≥ n0?

(a) Omega notation (b) Big O notation (c) Small o notation (d) Theta Notation **13.** Function in o(n2) notation is

(a) 10n2 (b) n1.9

(c) n2/100 (d) n2

**True or False**

**1.** Trees and graphs are the examples of linear data structures.

**2.** Queue is a FIFO data structure.

**3.** Trees can represent any kind of complex relationship between the nodes.

**4.** The average-case running time of an algorithm is an upper bound on the running time for any input. **5.** Array is an abstract data type.

**6.** Array elements are stored in continuous memory locations.

**7.** The pop operation adds an element to the top of a stack.

**8.** Graphs have a purely parent-to-child relationship between their nodes.

**9.** The worst-case running time of an algorithm is a lower bound on the running time for any input. **10.** In top-down approach, we start with designing the most basic or concrete modules and then proceed towards designing higher-level modules.

**11.** o(g(n)) comprises a set of all functions h(n) that are less than or equal to cg(n) for all values of n ≥ n0.

**12.** Simply Ω means same as best case Ω. **13.** Small omega notation provides an asymptotically tight bound for f(n).

**14.** Theta notation provides a non-asymptotically tight lower bound for f(n).

**15.** n3.001 ≠ ω(n3).

**Fill in the Blanks**

**1.** \_\_\_\_\_\_ is an arrangement of data either in the computer’s memory or on the disk storage.

*Introduction to Data Structures and Algorithms* **65**

**2.** \_\_\_\_\_\_ are used to manipulate the data contained in various data structures.

**3.** In \_\_\_\_\_\_, the elements of a data structure are stored sequentially.

**4.** \_\_\_\_\_\_ of a variable specifies the set of values that the variable can take.

**5.** A tree is empty if \_\_\_\_\_\_.

**6.** Abstract means \_\_\_\_\_\_.

**7.** The time complexity of an algorithm is the running time given as a function of \_\_\_\_\_\_.

**8.** \_\_\_\_\_\_ analysis guarantees the average perfor mance of each operation in the worst case. **9.** The elements of an array are referenced by an \_\_\_\_\_\_.

**10.** \_\_\_\_\_\_ is used to store the address of the topmost element of a stack.

**11.** The \_\_\_\_\_\_ operation returns the value of the topmost element of a stack.

**12.** An overflow occurs when \_\_\_\_\_\_.

**13.** \_\_\_\_\_\_ is a FIFO data structure.

**14.** The elements in a queue are added at \_\_\_\_\_\_ and removed from \_\_\_\_\_\_.

**15.** If the elements of a data structure are stored sequentially, then it is a \_\_\_\_\_\_.

**16.** \_\_\_\_\_\_ is basically a set of instructions that solve a problem.

**17.** The number of machine instructions that a pro gram executes during its execution is called its \_\_\_\_\_\_.

**18.** \_\_\_\_\_\_ specifies the expected behaviour of an algorithm when an input is randomly drawn from a given distribution.

**19.** The running time complexity of a constant time algorithm is given as \_\_\_\_\_\_.

**20.** A complex algorithm is often divided into smaller units called \_\_\_\_\_\_.

**21.** \_\_\_\_\_ design approach starts by dividing the complex algorithm into one or more modules. **22.** \_\_\_\_\_\_\_ case is when the array is sorted in reverse order.

**23.** \_\_\_\_\_\_\_\_ notation provides a tight lower bound for f(n).

**24.** The small o notation provides a \_\_\_\_\_\_\_\_\_ tight upper bound for f(n).

**25.** 540n2 + 10 \_\_\_\_ Ω (n2).

chapter 3

Arrays

**Learning Objective**

In this chapter, we will discuss arrays. An array is a user-defined data type that

stores related information together. All the information stored in an array belongs

to the same data type. So, in this chapter, we will learn how arrays are defined,

declared, initialized, and accessed. We will also discuss the different operations

that can be performed on array elements and the different types of arrays such as

two-dimensional arrays, multi-dimensional arrays, and sparse matrices.

**3.1 INTRODUCTION**

We will explain the concept of arrays using an analogy. Consider a situation in which we have 20 students in a class and we have been asked to write a program that reads and prints the marks of all the 20 students. In this program, we will need 20 integer variables with different names, as shown in Fig. 3.1.

Now to read the values of these 20 variables, we must have 20 read statements. Similarly, to print the value of these variables, we need 20 write statements. If it is just a matter of 20 variables, then it might be acceptable for the user to follow this approach. But would it be possible to follow this approach if we have to read and print the marks of students,

∑ in the entire course (say 100 students)

∑ in the entire college (say 500 students)

∑ in the entire university (say 10,000 students)

The answer is no, definitely not! To process a large amount of data, we need a data structure known as *array*.