

As per
JNTU-Hyderabad
Syllabus regulation
2013

Computer Programming

First Edition


About the Author

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

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

- *Computing Fundamentals and C Programming*
- *Fundamentals of Computers*
- *Programming in C#, 3/e*
- *Programming in Java, 4/e*
- *Object-Oriented Programming with C++, 5/e*
- *Programming in BASIC, 3/e*
- *Programming in ANSI C, 5/e*
- *Numerical Methods*
- *Reliability Engineering*

A recipient of numerous honours and awards, he has been listed in the Directory of Who's Who of Intellectuals and in the Directory of Distinguished Leaders in Education.



As per
JNTU-Hyderabad
Syllabus regulation
2013

Computer Programming

First Edition

E Balagurusamy

Chairman

EBG Foundation

Coimbatore



McGraw Hill Education (India) Private Limited

NEW DELHI

McGraw Hill Education Offices

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



McGraw Hill Education (India) Private Limited

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

Computer Programming, 1e

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

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

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

ISBN (13): 978-93-5134-2939

ISBN (10): 93-5134-293X

Vice President and Managing Director: *Ajay Shukla*

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

Publishing Manager (SEM & Tech. Ed.): *Shalini Jha*

Editorial Researcher—Acquisitions: *S. Vamsi Deepak*

Manager—Production Systems: *Satinder S Baveja*

Assistant Manager—Editorial Services: *Sohini Mukherjee*

Senior Production Executive: *Suhaib Ali*

Assistant General Manager—Higher Education (Marketing): *Vijay Sarathi*

Senior Product Specialist: *Tina Jajoriya*

Senior Graphic Designer—Cover: *Meenu Raghav*

General Manager—Production: *Rajender P Ghansela*

Production Manager—*Reji Kumar*

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

Typeset at Text-o-Graphics, B-1/56, Aravali Apartment, Sector-34, Noida 201 301, and printed at

Cover Printer:

Content

Preface

xv

Roadmap to the Syllabus

xix

1. Introduction to Computers

1.1-1.50

- 1.1 Introduction 1.1
- 1.2 Computer Systems 1.2
 - 1.2.1 Input Devices 1.3
 - 1.2.2 CPU 1.5
 - 1.2.3 Output Devices 1.6
 - 1.2.4 Memory 1.8
- 1.3 Programming Languages 1.12
 - 1.3.1 Machine Language 1.12
 - 1.3.2 Assembly Language 1.12
 - 1.3.3 High-Level Languages 1.13
 - 1.3.4 Procedure-oriented Languages 1.14
 - 1.3.5 Problem-oriented Languages 1.14
 - 1.3.6 Natural Languages 1.14
- 1.4 Programming Environment 1.15
- 1.5 Creating and Running Programs 1.15
 - 1.5.1 Structured Programming 1.15
 - 1.5.2 System Development Tools 1.16
 - 1.5.3 Developing a Program 1.18
 - 1.5.4 Running a Program 1.21
- 1.6 Software Development Method 1.22
 - 1.6.1 Analysing the Requirements 1.23
 - 1.6.2 Feasibility Analysis 1.23
 - 1.6.3 Creating the Design 1.24
 - 1.6.4 Developing Code 1.24
 - 1.6.5 Testing the Software 1.25
 - 1.6.6 Deploying the Software 1.25
 - 1.6.7 Maintaining the Software 1.25

1.7	Applying Software Development Method	1.25
1.8	Problem Solving	1.27
1.9	Algorithms	1.27
1.9.1	Characteristics of Algorithms	1.28
1.9.2	Advantages of Algorithms	1.28
1.9.3	Disadvantages of Algorithms	1.28
1.10	Flowcharts	1.30
1.10.1	Flowchart Design Rules	1.33
1.10.2	Advantages of Flowcharts	1.33
1.10.3	Disadvantages of Flowcharts	1.33
1.11	Pseudocodes	1.36
1.11.1	Pseudocode Rules	1.37
1.11.2	Advantages of Pseudocodes	1.37
1.11.3	Disadvantages of Pseudocodes	1.37
1.12	Problem Solving Examples	1.40
	Just Remember	1.46
	Multiple Choice Questions	1.47
	Review Questions	1.48
	Key Terms	1.49

2 Introduction to C

2.1-2.147

2.1	Introduction	2.1
2.2	Overview of C	2.2
2.2.1	History of C	2.2
2.2.2	Characteristics of C	2.4
2.2.3	Sample Program 1: Printing a Message	2.4
2.2.4	Sample Program 2: Adding Two Numbers	2.7
2.2.5	Sample Program 3: Interest Calculation	2.8
2.2.6	Sample Program 4: Use of Subroutines	2.10
2.2.7	Sample Program 5: Use of Math functions	2.11
2.3	Basic Structure of C Programs	2.13
2.4	Programming Style	2.13
2.5	Executing A 'C' Program	2.14
2.6	UNIX System	2.15
2.7	C Character Set	2.17
2.7.1	Trigraph Characters	2.19
2.8	C Tokens	2.19
2.9	Keywords and Identifiers	2.20
2.10	Constants	2.21
2.10.1	Integer Constants	2.21
2.10.2	Real Constants	2.22
2.10.3	Single Character Constants	2.23
2.10.4	String Constants	2.24
2.11	Variables	2.24
2.12	Data Types	2.25

2.12.1	Integer Types	2.26
2.12.2	Floating Point Types	2.27
2.12.3	Void Types	2.27
2.12.4	Character Types	2.27
2.13	Declaration of Variables	2.27
2.13.1	Primary Type Declaration	2.28
2.13.2	User-defined Type Declaration	2.29
2.14	Declaration of Storage Class	2.30
2.15	Assigning Values to Variables	2.31
2.15.1	Assignment Statement	2.32
2.15.2	Reading Data from Keyboard	2.34
2.15.3	Declaring a Variable as a Constant	2.36
2.15.4	Declaring a Variable as Volatile	2.36
2.16	Case Studies	2.37
2.16.1	Calculation of Average of Numbers	2.37
2.16.2	Temperature Conversion Problem	2.38
2.17	Managing Input and Output Operations	2.39
2.17.1	Reading a Character	2.39
2.17.2	Writing a Character	2.42
2.17.3	Formatted Input	2.43
2.17.4	Points to Remember while Using scanf	2.50
2.17.5	Formatted Output	2.50
2.18	Case Studies	2.55
2.19	Operators and expressions	2.59
2.19.1	Arithmetic Operators	2.59
2.19.2	Relational Operators	2.61
2.19.3	Logical Operators	2.62
2.19.4	Assignment Operators	2.63
2.19.5	Increment and Decrement Operators	2.65
2.19.6	Conditional Operator	2.66
2.19.7	Bitwise Operators	2.67
2.19.8	Special Operators	2.67
2.19.9	Operator Precedence	2.69
2.19.10	Precedence of Arithmetic Operators	2.70
2.19.11	Some Computational Problems	2.72
2.19.12	Type Conversions in Expressions	2.73
2.19.13	Operator Precedence and Associativity	2.75
2.20	Case Studies	2.78
2.21	Decision making and branching	2.79
2.21.1	Decision Making with if Statement	2.80
2.21.2	The switch Statement	2.90
2.21.3	The goto Statement	2.92
2.22	Case Studies	2.94
2.23	Decision making and looping	2.98
2.23.1	The while Statement	2.99
2.23.2	The do Statement	2.100

2.23.3	The for Statement	2.102
2.23.4	Jumps in Loops	2.107
2.24	Jumping out of the Program	2.112
2.25	Structured Programming	2.112
2.26	Case Studies	2.113
	Just Remember	2.142
	Review Questions	2.144
	Multiple Choice Questions	2.145
	Programming Exercise	2.146

3A. Functions

3A.1-3A.45

3A.1	Introduction	3A.1
3A.2	Need for User-defined Functions	3A.1
3A.3	A Multi-function Program	3A.2
3A.4	Elements of user-defined Functions	3A.4
3A.5	Definition of Functions	3A.4
3A.5.1	Function Header	3A.5
3A.5.2	Name and Type	3A.5
3A.5.3	Formal Parameter List	3A.5
3A.5.4	Function Body	3A.6
3A.6	Return Values and their Types	3A.7
3A.7	Function Calls	3A.8
3A.8	Function Declaration	3A.9
3A.9	Category of Functions	3A.11
3A.10	No Arguments and no Return Values	3A.11
3A.11	Arguments but no Return Values	3A.13
3A.12	Arguments with Return Values	3A.15
3A.13	No Arguments but Returns a Value	3A.18
3A.14	Functions that Return Multiple Values	3A.19
3A.15	Nesting of Functions	3A.20
3A.16	Recursion	3A.21
3A.17	The Scope, Visibility and Lifetime of Variables	3A.22
3A.17.1	Automatic Variables	3A.23
3A.17.2	External Variables	3A.24
3A.17.3	Static Variables	3A.328
3A.17.4	Register Variables	3A.29
3A.17.5	Nested Blocks	3A.30
3A.18	Multifile Programs	3A.31
3A.19	Preprocessor Commands	3A.32
3A.19.1	Macro Substitution Directives	3A.33
3A.19.2	File Inclusion Directive	3A.35
3A.19.3	Conditional Inclusion Directives	3A.35
3A.19.4	Additional Preprocessor Directives	3A.36
	Just Remember	3A.37
	Multiple Choice Questions	3A.38

Case Study	3A.39
Review Questions	3A.41
Programming Exercises	3A.44
Key Terms	3A.45

3B. Arrays

3B.1-3B.79

3B.1	Introduction	3B.1
3B.2	One-Dimensional Arrays	3B.2
3B.3	Declaration of one-dimensional Arrays	3B.3
3B.4	Initialization of one-dimensional Arrays	3B.5
3B.4.1	Compile Time Initialization	3B.6
3B.4.2	Run Time Initialization	3B.6
3B.5	Two-Dimensional Arrays	3B.14
3B.6	Initializing Two-Dimensional Arrays	3B.18
3B.7	Multi-Dimensional Arrays	3B.21
3B.8	Dynamic Arrays	3B.22
3B.9	Passing Arrays to Functions	3A.22
3B.9.1	One-Dimensional Arrays	3A.22
3B.9.2	Two-Dimensional Arrays	3A.26
3B.10	Passing Strings to Functions	3A.27
3B.11	More About Arrays	3B.27
3B.12	C Programming Examples – Built-in Functions	3B.58
	Just Remember	3B.63
	Multiple Choice Questions	3B.63
	Case Study	3B.64
	Review Questions	3B.75
	Programming Exercises	3B.77
	Key Terms	3B.79

4A. Pointers

4A.1-4A.35

4A.1	Introduction	4A.1
4A.2	Understanding Pointers	4A.1
4A.3	Accessing the Address of a Variable	4A.3
4A.4	Declaring Pointer Variables	4A.4
4A.5	Initialization of Pointer Variables	4A.5
4A.6	Accessing a Variable Through its Pointer	4A.6
4A.7	Chain of Pointers	4A.8
4A.8	Pointer Expressions	4A.9
4A.9	Pointer Increments and Scale Factor	4A.10
4A.10	Pointers and Arrays	4A.11
4A.11	Pointers and Character Strings	4A.14
4A.12	Array of Pointers	4A.16
4A.13	Pointers as Function Arguments	4A.16
4A.14	Functions Returning Pointers	4A.19

- 4A.15 Pointers to Functions 4A.19
- 4A.16 Introduction 4A.22
- 4A.17 Dynamic Memory Allocation 4A.23
- 4A.18 Allocating a Block of Memory: Malloc 4A.23
- 4A.19 Allocating Multiple Blocks of Memory: Calloc 4A.25
- 4A.20 Releasing the Used Space: Free 4A.26
 - Just Remember 4A.26
 - Multiple Choice Questions 4A.27
 - Case Study 4A.28
 - Review Questions 4A.33
 - Programming Exercises 4A.34
 - Key Terms 4A.35

4B. Strings

4B.1-4B.30

- 4B.1 Introduction 4B.1
- 4B.2 Declaring and Initializing String Variables 4B.2
- 4B.3 Reading Strings From Terminal 4B.3
 - 4B.3.1 Using scanf Function 4B.3
 - 4B.3.2 Reading a Line of Text 4B.5
 - 4B.3.3 Using *getchar* and *gets* Functions 4B.5
- 4B.4 Writing Strings to Screen 4B.7
 - 4B.4.1 Using printf Function 4B.7
 - 4B.4.2 Using putchar and puts Functions 4B.10
- 4B.5 Arithmetic Operations on Characters 4B.11
- 4B.6 Putting Strings Together 4B.12
- 4B.7 Comparison of Two Strings 4B.14
- 4B.8 String-Handling Functions 4B.14
 - 4B.8.1 strcat() Function 4B.14
 - 4B.8.2 strcmp() Function 4B.15
 - 4B.8.3 strcpy() Function 4B.15
 - 4B.8.4 strlen() Function 4B.16
 - 4B.8.5 strncpy() Function 4B.17
 - 4B.8.6 strncmp() Function 4B.17
 - 4B.8.7 strncat() Function 4B.18
 - 4B.8.8 strstr() Function 4B.18
- 4B.9 Table of Strings 4B.19
- 4B.10 Other Features of Strings 4B.20
- 4B.11 String / Data Conversion 4B.20
 - Just Remember 4B.22
 - Multiple Choice Questions 4B.22
 - Case Study 4B.23
 - Review Questions 4B.27
 - Programming Exercises 4B.29
 - Key Terms 4B.30

5. Structures and Unions**5.1-5.59**

- 5.1 Introduction 5.1
- 5.2 Defining a Structure 5.1
- 5.3 Declaring Structure Variables 5.2
- 5.4 Accessing Structure Members 5.4
- 5.5 Structure Initialization 5.5
- 5.6 Copying and Comparing Structure Variables 5.7
- 5.7 Operations On Individual Members 5.8
- 5.8 Arrays of Structures 5.9
- 5.9 Arrays Within Structures 5.11
- 5.10 Structures Within Structures 5.12
- 5.11 Pointers and Structures 5.14
- 5.12 Structures and Functions 5.17
- 5.13 Unions 5.20
- 5.14 Size of Structures 5.21
- 5.15 Bit Fields 5.22
- 5.16 Typedef 5.24
- 5.17 Enum 5.24
- 5.18 Command Line Arguments 5.38
- 5.19 Application of Command Line Arguments 5.48
 - Just Remember 5.50
 - Multiple Choice Questions 5.51
 - Case Study 5.52
 - Review Questions 5.55
 - Programming Exercises 5.58
 - Key Terms 5.59

6. File Management in C**6.1-6.19**

- 6.1 Introduction 6.1
- 6.2 Types of Files 6.2
- 6.3 Defining and Opening a File 6.3
- 6.4 Closing a File 6.4
- 6.5 Input/Output Operations on Files 6.4
 - 6.5.1 The *getc* and *putc* Functions 6.5
 - 6.5.2 The *getw* and *putw* Functions 6.6
 - 6.5.3 The *fprintf* and *fscanf* Functions 6.8
- 6.6 Error Handling During I/O Operations 6.10
- 6.7 Random Access to Files 6.12
 - Just Remember 6.16
 - Multiple Choice Questions 6.17
 - Review Questions 6.18
 - Programming Exercises 6.19
 - Key Terms 6.19

7. Sorting and Searching Techniques **7.1-7.23**

- 7.1 Introduction 7.1
- 7.2 Sorting 7.1
 - 7.2.1 Sorting Efficiency 7.2
 - 7.2.2 Exchange Sorting—Bubble Sort 7.3
 - 7.2.3 Exchange Sorting—Quick Sort 7.5
 - 7.2.4 Selection Sort 7.8
 - 7.2.5 Merge Sort 7.10
 - 7.2.6 Simple Insertion Sort 7.12
 - 7.2.7 Shell Sort 7.13
- 7.3 Searching 7.15
 - 7.3.1 Linear Search 7.15
 - 7.3.2 Binary Search 7.16
 - 7.3.3 Indexed Sequential Search 7.19
- Just Remember 7.22
- Multiple Choice Questions 7.22
- Review Questions 7.23

8. Data Structures **8.1-8.52**

- 8.1 Introduction 8.1
- 8.2 Abstract Data Types 8.2
- 8.3 Linear List 8.2
 - 8.3.1 Linked Lists Operations 8.5
 - 8.3.2 Implementation 8.10
 - 8.3.3 Linked List with Header 8.16
- 8.4 Stacks 8.17
 - 8.4.1 Stack Operations 8.18
- 8.5 Stack Implementation 8.18
 - 8.5.1 Array Implementation of Stacks 8.19
 - 8.5.2 Linked Implementation of Stacks 8.23
- 8.6 Applications of Stacks 8.27
 - 8.6.1 Infix Notation 8.27
 - 8.6.2 Prefix Notation 8.27
 - 8.6.3 Conversion from Infix to Prefix Notation 8.28
 - 8.6.4 Evaluation of Prefix Expression 8.31
 - 8.6.5 Postfix Notation 8.32
 - 8.6.6 Conversion from Infix to Postfix Notation 8.32
 - 8.6.7 Evaluation of Postfix Expression 8.36
- 8.7 Queues 8.36
 - 8.7.1 Queue Operations 8.37
- 8.8 Queue Implementation 8.38
 - 8.8.1 Array Implementation of Queues 8.38
 - 8.8.2 Linked Implementation of Queues 8.44

Just Remember	8.49
Multiple Choice Questions	8.49
Review Questions	8.50
Programming Exercises	8.52

Appendix A

A.1-A.12

Solved Question Papers

June 2011 Set 1	<i>1.1–1.6</i>
June 2011 Set 2	<i>2.1–2.8</i>
June 2011 Set 3	<i>3.1–3.7</i>
June 2011 Set 4	<i>4.1–4.11</i>
June 2012	SQP SET 1.1–1.8

Preface

The developments in digital electronics and related technologies during the last few decades have ushered in the second Industrial Revolution that is popularly referred to as the Information Revolution. Computer technology plays an ever-increasing role in this new revolution. Today, the application of computers is all pervasive in everybody's life. A sound knowledge of how computers process data and information has, therefore, become indispensable for anyone who seeks employment not only in the area of IT but also in any other field.

COMPUTER PROGRAMMING

Computer programming is dedicated to writing, testing and maintaining programs that computers follow to perform their functions. To create programs that control the behaviour of a machine, we need programming languages. This book enables students to master the necessary skills for computer programming with C language and shows them how to use these skills wisely with data structures and algorithms.

THE C PROGRAMMING LANGUAGE

C is a general-purpose structured programming language that is powerful, efficient and compact. C combines the features of a high-level language with the elements of the assembler and is thus close to both man and machine. The growth of C during the last few years has been phenomenal. It has emerged as the language of choice for most applications due to its speed, portability and compactness of code. It has now been implemented on virtually every sort of computer, from micro to mainframe.

Rightly so, many institutions and universities in India have introduced a subject covering Computer Programming. **This book is specially designed for the first-year students studying the foundation course in the first semester of Jawaharlal Nehru Technological University (JNTU).**

DATA STRUCTURES

Data structures are prevalent in almost every program, since they provide specialised formats of storing and organising data in a computer. This book presents the design and implementation of data structures using standard ANSI C programming language.

WHY IS THIS BOOK A WINNER?

- This book is completely in sync with the new syllabus prescribed by JNTU, effective August 2009.
 - It offers comprehensive coverage of topics related to Data Structures, Linked Lists, Queues, Stacks (important as per the new syllabus).
 - New chapters on Fundamentals of Computers and Computing Concepts cover two new units in the revised syllabus. These chapters clearly explain the basics of computers, thereby providing a strong foundation for mastering advanced topics on programming and data structures given in successive chapters.
- This book incorporates all the features of ANSI C that are essential for a C programmer. The ANSI standards are explained in detail at appropriate places (noted in the text by a special mention for the convenience of our readers).
- From a learner to a skilled C programmer, this book ensures smooth and successful transition. The concept of ‘learning by example’ has been stressed throughout the book. Every important feature of the language is treated in depth followed by a complete program example to illustrate its use. Case studies at the end of chapters not only describe the common ways in which C features are put together but also show real-life applications. Wherever necessary, pictorial descriptions of concepts are included to facilitate better understanding. Thus, this book succeeds at presenting a contemporary approach to programming with a unique combination of theory and practice.
- It highly appeals to the students because of focused coverage of syllabus, variety of programs and simple theory.

PEDAGOGICAL FEATURES

- **Bottom-up approach** of explaining concepts.
- **Algorithms and flowcharts** are covered extensively.
- **Codes and comments** illustrate the use of various features of the language.
- **Special boxes feature supplementary information and notes** that complement the text.
- **Case studies in relevant chapters** demonstrate real-life applications.
- **Just remember** section lists out helpful hints and possible problem areas.
- **Review questions (132)** provide ample opportunities to test the conceptual understanding.
- **Programming exercises (135)** simulate interest to practice programming applications.
- **Solved examples and programs (200)** have been tested using compilers compatible to both UNIX and MS-DOS operating systems and, wherever appropriate, the nature of output has been discussed. These programs also demonstrate the general principles of a good programming style.
- **Multiple choice questions (179)** help students test their conceptual understanding of the subject.
- **Lab assignments** are given as per the new syllabus.
- Updated with information on **compiler C-99** supported by numerous programs (12) using C-99.
- **Solutions to 2011 exam question papers (4 sets)** and 2012 exam question paper.

HOW IS THE BOOK ORGANISED

The content is spread over 10 chapters. **Chapter 1** introduces computer systems, programming languages and environment, software development method, algorithms, flowcharts and pseudocodes. **Chapter 2** gives an overview of C and explains keywords, identifiers, constants, variables, data types and various case studies on these. **Chapters 3A** and **3B** discuss arrays and functions respectively, whereas pointers and strings are covered in **Chapters 4A** and **4B**. **Chapter 5** presents structures and unions. **Chapter 6** deals with file management in C. Sorting and searching techniques are covered in **Chapter 7**. Finally, **Chapter 8** presents data structures.

There is also an Appendix and Solved Question Papers of June 2011 and June 2012 at the end of the book.

CD RESOURCES

The book is accompanied with a CD which provides the following resources:

- All lab assignments
- Previous years' solved university question papers
- 208 additional objective/ review/ debugging type questions pertaining to each unit
- 5 Model Question Papers

The supplementary CD helps students master the programming language and write their own programs using Computer programming concepts and data structures.

ACKNOWLEDGEMENTS

The author is grateful to *Mrs Nagaratna Parameshwar Hegde* of Vasavi College of Engineering, Hyderabad for her useful comments and suggestions.

E Balagurusamy

PUBLISHER'S NOTE

McGraw Hill Education (India) looks forward to receiving from teachers and students their valuable views, comments and suggestions for improvements, all of which may be sent to tmh.corefeedback@gmail.com (mentioning the title and author's name). Also, please inform any observations on piracy related issues.

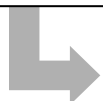
Roadmap to the Syllabus

COMPUTER PROGRAMMING

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

Module 1: Introduction to Computers—Computer Systems, Computing Environments, Computer Languages, Creating and running programs, Program Development.

Introduction to the C Language—Background, C Programs, Identifiers, Types, Variables, Constants, Input I Output, Operators (Arithmetic, relational, logical, bitwise etc.), Expressions, Precedence and Associativity, Expression Evaluation, Type conversions, Statements—Selection Statements (making decisions)—if and switch statements, Repetition statements (loops)-while, for, do-while statements, Loop examples, other statements related to looping break, continue, goto, Simple C Program examples.



GO TO:

CHAPTER 1. Introduction to Computers

CHAPTER 2. Introduction to C

Module 2: Functions-Designing Structured Programs, Functions, user defined functions, inter function communication, Standard functions, Scope, Storage classes-auto, register, static, extern, scope rules, type qualifiers, recursion-recursive functions, Limitations of recursion, example C programs, Preprocessor commands.

Arrays—Concepts, using arrays in C, inter function communication, array applications, two-dimensional arrays, multidimensional arrays, C program examples.



GO TO:

CHAPTER 3A. Functions

CHAPTER 3B. Arrays

Module 3: Pointers—Introduction (Basic Concepts), Pointers for inter function communication, pointers to pointers, compatibility, Pointer Applications-Arrays and Pointers, Pointer Arithmetic and arrays, Passing an array to a function, memory allocation functions, array of pointers, programming applications, pointers to void, pointers to functions.

Strings—Concepts, C Strings, String Input I Output functions, arrays of strings, string manipulation functions, string I data conversion, C program examples.



GO TO:

CHAPTER 4A. Pointers

CHAPTER 4B. Strings

Module 4: Enumerated, Structure, and Union Types—The Type Definition (typedef), Enumerated types, Structures—Declaration, initialization, accessing structures, operations on structures, Complex structures, structures and functions, Passing structures through pointers, self referential structures, unions, bit fields, C programming examples, command-line arguments, Input and Output—Concept of a file, streams, text files and binary files, Differences between text and binary files, State of a file, Opening and Closing files, file input 1 output functions (standard library input 1 output functions for files), file status functions (error handling), Positioning functions, C program examples.

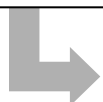


GO TO:

CHAPTER 5. Structures and Unions

CHAPTER 6. File Management in C

Module 5: Searching and Sorting—Sorting-selection sort, bubble sort, Searching-linear and binary search methods. Lists- Linear list—singly linked list implementation, insertion, deletion and searching operations on linear list, StacksPush and Pop Operations, Queues—Enqueue and Dequeue operations.



GO TO:

CHAPTER 7. Sorting and Searching Techniques

CHAPTER 8. Data Structures

1 Introduction to Computers

1.1 INTRODUCTION

The term *computer* is derived from the word *compute*. A computer is an electronic device that takes data and instructions as an *input* from the user, *processes* data, and provides useful information known as *output*. This cycle of operation of a computer is known as the *input–process–output* cycle and is shown in Fig. 1.1 The electronic device is known as *hardware* and the set of instructions is known as *software*.

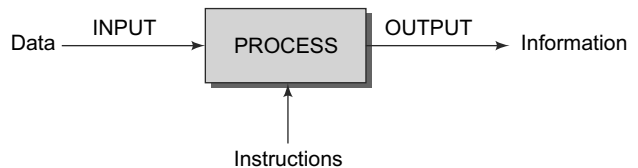


Fig. 1.1 Input–process–output concept

A computer consists of various components that function as an integrated system to perform computational tasks. These components include the following:

- **Central Processing Unit (CPU)** It is the brain of the computer that is responsible for controlling and executing program instructions.
- **Monitor** It is a display screen, which shows information in visual form.
- **Keyboard and Mouse** These are the peripheral devices used by the computer for receiving inputs from the user.

Figure 1.2 shows the various components of a computer.

The unique capabilities and characteristics of a computer have made it very popular among its various users, including engineers, managers, accountants, teachers, students, etc.

Some of the key characteristics of a modern digital computer include, among others the following:

- **Speed** The computer is a fast electronic device that can solve large and complex problems in few seconds. The speed of a computer generally depends upon its hardware configuration.
- **Storage capacity** A computer can store huge amounts of data in many different formats. The storage area of a computer system is generally divided into two categories, main memory and secondary storage.

1.2 Computer Programming

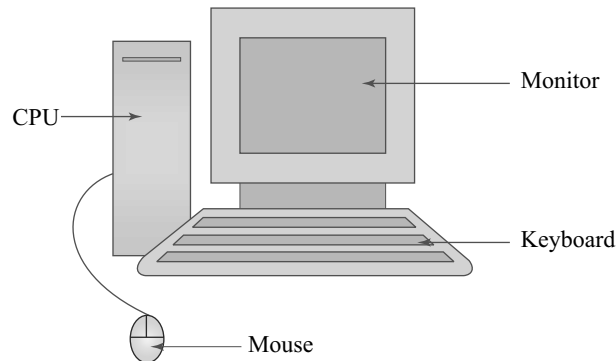


Fig. 1.2 *The components of a computer*

- **Accuracy** A computer carries out calculations with great accuracy. The accuracy achieved by a computer depends upon its hardware configuration and the specified instructions.
- **Reliability** A computer produces results with no error. Most of the computer-generated errors are in actuality human errors that are instigated by the user itself. Therefore, computers are regarded as quite trustworthy machines.
- **Versatility** Computers are versatile machines. They can perform varied tasks and can be used for many different purposes.
- **Diligence** Computers can perform repetitive calculations any number of times with the same level of accuracy.

These capabilities of computers have enabled us to use them for a variety of tasks. Application areas may broadly be classified into the following major categories.

1. Data processing (commercial use)
2. Numerical computing (scientific use)
3. Text (word) processing (office and educational use)
4. Message communication (e-mail)
5. Image processing (animation and industrial use)
6. Voice recognition (multimedia)

1.2 COMPUTER SYSTEMS

A computer system comprises of **hardware** and **software** components. Hardware refers to the physical parts of the computer system and software is the set of instructions or programs that are necessary for the functioning of a computer to perform certain tasks. Hardware includes the following components:

- **Input devices** They are used for accepting the data on which the operations are to be performed. The examples of input devices are keyboard, mouse and track ball.
- **Processor** Also known as CPU, it is used to perform the calculations and information processing on the data that is entered through the input device.
- **Output devices** They are used for providing the output of a program that is obtained after performing the operations specified in a program. The examples of output devices are monitor and printer.
- **Memory** It is used for storing the input data as well as the output of a program that is obtained after performing the operations specified in a program. Memory can be primary memory as well

as secondary memory. Primary memory includes Random Access Memory (RAM) and secondary memory includes hard disks and floppy disks.

Software supports the functioning of a computer system internally and cannot be seen. It is stored on secondary memory and can be an **application software** as well as **system software**. The application software is used to perform a specific task according to requirements and the system software is mandatory for running application software. The examples of application software include Excel and MS Word and the examples of system software include operating system and networking system.

All the hardware components interact with each other as well as with the software. Similarly, the different types of software interact with each other and with the hardware components. The interaction between various hardware components is illustrated in Fig. 1.3.

1.2.1 Input Devices

Input devices can be connected to the computer system using cables. The most commonly used input devices among others are:

- Keyboard
- Mouse
- Scanner

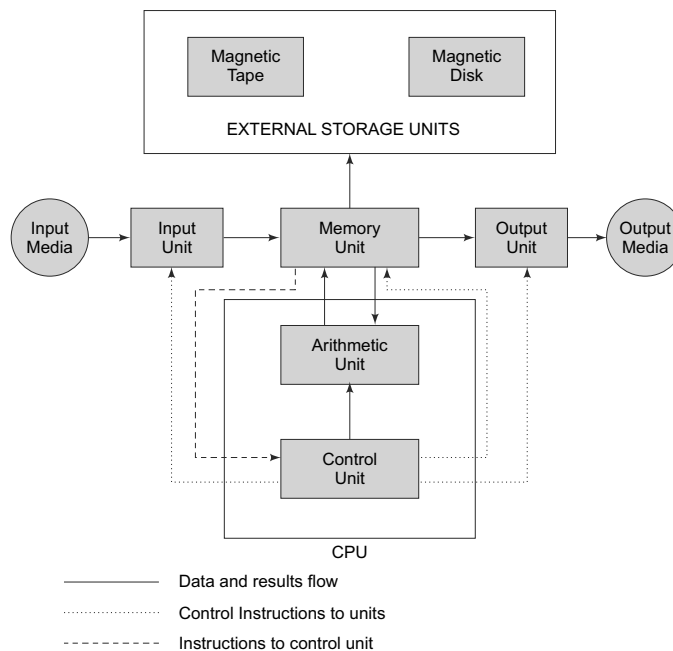


Fig. 1.3 *Interaction among hardware components*

Keyboard

A standard keyboard includes alphanumeric keys, function keys, modifier keys, cursor movement keys, spacebar, escape key, numeric keypad, and some special keys, such as Page Up, Page Down, Home, Insert, Delete and End. The alphanumeric keys include the number keys and the alphabet keys. The function keys are the keys that help perform a specific task such as searching a file or refreshing a Web page. The modifier

1.4 Computer Programming

keys such as Shift and Control keys modify the casing style of a character or symbol. The cursor movement keys include up, down, left and right keys and are used to modify the direction of the cursor on the screen. The spacebar key shifts the cursor to the right by one position. The numeric keypad uses separate keypads for numbers and mathematical operators. A keyboard is shown in Fig. 1.4.

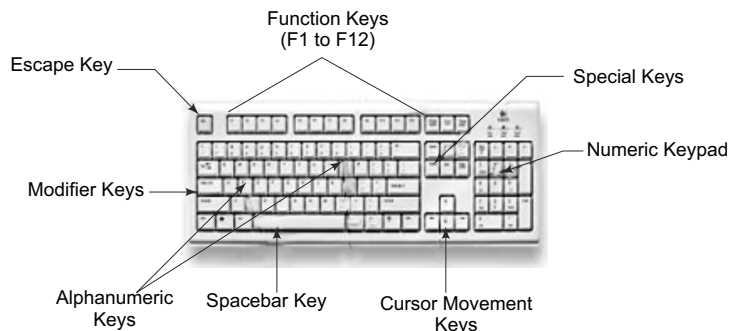


Fig. 1.4 Keyboard

Mouse

The mouse allows the user to select elements on the screen, such as tools, icons, and buttons, by pointing and clicking them. We can also use a mouse to draw and paint on the screen of the computer system. The mouse is also known as a pointing device because it helps change the position of the pointer or cursor on the screen.

The mouse consists of two buttons, a wheel at the top and a ball at the bottom of the mouse. When the ball moves, the cursor on the screen moves in the direction in which the ball rotates. The left button of the mouse is used to select an element and the right button, when clicked, displays the special options such as **open** and **explore** and **shortcut** menus. The wheel is used to scroll down in a document or a Web page. A mouse is shown in Fig. 1.5.

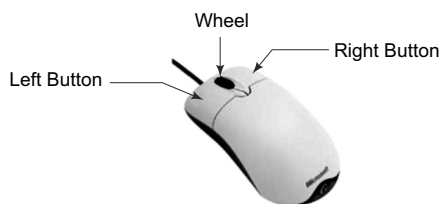


Fig. 1.5 Mouse

Scanner

A scanner is an input device that converts documents and images as the digitized images understandable by the computer system. The digitized images can be produced as black and white images, gray images, or colored images. In case of colored images, an image is considered as a collection of dots with each dot representing a combination of red, green, and blue colors, varying in proportions. The proportions of red, green, and blue colors assigned to a dot are together called as *color description*. The scanner uses the color description of the dots to produce a digitized image. Fig. 1.6 shows a scanner.



Fig. 1.6 Scanner

There are the following types of scanners that can be used to produce digitized images:

- **Flatbed scanner** It contains a scanner head that moves across a page from top to bottom to read the page and converts the image or text available on the page in digital form. The flatbed scanner is used to scan graphics, oversized documents, and pages from books.

- **Drum scanner** In this type of scanner, a fixed scanner head is used and the image to be scanned is moved across the head. The drum scanners are used for scanning prepress materials.
- **Slide scanner** It is a scanner that can scan photographic slides directly to produce files understandable by the computer.
- **Handheld scanner** It is a scanner that is moved by the end user across the page to be scanned. This type of scanner is inexpensive and small in size.

1.2.2 CPU

The CPU consists of Control Unit (CU) and ALU. CU stores the instruction set, which specifies the operations to be performed by the computer. CU transfers the data and the instructions to the ALU for an arithmetic operation. ALU performs arithmetical or logical operations on the data received. The CPU registers store the data to be processed by the CPU and the processed data also. Apart from CU and ALU, CPU seeks help from the following hardware devices to process the data:

Motherboard

It refers to a device used for connecting the CPU with the input and output devices. The components on the motherboard are connected to all parts of a computer and are kept insulated from each other. Some of the components of a motherboard are:

- **Bus** Electrical pathways that transfer data and instructions among different parts of the computer. For example, the data bus is an electrical pathway that transfers data among the microprocessor, memory and input/output devices connected to the computer. The address bus is connected among the microprocessor, RAM and Read Only Memory (ROM), to transfer addresses of RAM and ROM locations that is to be accessed by the microprocessor.
- **System clock** It is a clock used for synchronizing the activities performed by the computer. The electrical signals that are passed inside a computer are timed, based on the tick of the clock. As a result, the faster the system clock, the faster is the processing speed of the computer.
- **Microprocessor** CPU component that performs the processing and controls the activities performed by the different parts of the computer. The microprocessor is plugged to the CPU socket placed on the motherboard.
- **ROM** Chip that contains the permanent memory of the computer that stores information, which cannot be modified by the end user.

Random Access Memory (RAM)

It refers to primary memory of a computer that stores information and programs, until the computer is used. RAM is available as a chip that can be connected to the RAM slots in the motherboard.

Video Card/Sound card

The video card is an interface between the monitor and the CPU. Video cards also include their own RAM and microprocessors that are used for speeding up the processing and display of a graphic. These video cards are placed on the expansion slots, as these slots allow us to connect the high-speed graphic display cards to the motherboard. A sound card is a circuit board placed on the motherboard and is used to enhance the sound capabilities of a computer. The sound cards are plugged to the Peripheral Component Interconnect (PCI) slots. The PCI slots also enable the connection of networks interface card, modem cards and video cards, to the motherboard.

1.6 Computer Programming



Fig. 1.7 A motherboard

1.2.3 Output Devices

The data, processed by the CPU, is made available to the end user by the output devices. The most commonly used output devices are:

- Monitor
- Printer
- Speaker
- Plotter

Monitor

A monitor is the most commonly used output device that produces visual displays generated by the computer. The monitor, also known as a screen, is connected as an external device using cables or connected either as a part of the CPU case. The monitor connected using cables, is connected to the video card placed on the expansion slot of the motherboard. The display device is used for visual presentation of textual and graphical information.

The monitors can be classified as cathode ray tube (CRT) monitors or liquid crystal display (LCD) monitors. The CRT monitors are large, occupy more space in the computer, whereas LCD monitors are thin, light weighted, and occupy lesser space. Both the monitors are available as monochrome, gray scale and color models. However, the quality of the visual display produced by the CRT is better than that produced by the LCD.

A monitor can be characterized by its monitor size and resolution. The monitor size is the length of the screen that is measured diagonally. The resolution of the screen is expressed as the number of picture



Fig. 1.8 Monitor

elements or pixels of the screen. The resolution of the monitor is also called the dot pitch. The monitor with a higher resolution produces a clearer image.

Printer

The printer is an output device that is used to produce a hard copy of the electronic text displayed on the screen, in the form of paper sheets that can be used by the end user. The printer is an external device that is connected to the computer system using cables. The computer needs to convert the document that is to be printed to data that is understandable by the printer. The *printer driver software* or the *print driver software* is used to convert a document to a form understandable by the computer. When the computer components are upgraded, the upgraded printer driver software needs to be installed on the computer.

The performance of a printer is measured in terms of *dots per inch (DPI)* and *pages per minute (PPM)* produced by the printer. The greater the DPI parameter of a printer, the better is the quality of the output generated by it. The higher PPM represents higher efficiency of the printer. Printers can be classified based on the technology they use to print the text and images:

- **Dot matrix printers** Dot matrix printers are impact printers that use perforated sheet to print the text. The process to print a text involves striking a pin against a ribbon to produce its impression on the paper.
- **Inkjet printers** Inkjet printers are slower than dot matrix printers and are used to generate high quality photographic prints. Inkjet printers are not impact printers. The ink cartridges are attached to the printer head that moves horizontally, from left to right.
- **Laser printers** The laser printer may or may not be connected to a computer, to generate an output. These printers consist of a microprocessor, ROM and RAM, which can be used to store the textual information. The printer uses a cylindrical drum, a toner and the laser beam.



Fig. 1.9 Printer

Speaker

The speaker is an electromechanical transducer that converts an electrical signal into sound. They are attached to a computer as output devices, to provide audio output, such as warning sounds and Internet audios. We can have built-in speakers or attached speakers in a computer to warn end users with error audio messages and alerts. The audio drivers need to be installed in the computer to produce the audio output. The sound card being used in the computer system decides the quality of audio that we listen using music CDs or over the Internet. The computer speakers vary widely in terms of quality and price. The sophisticated computer speakers may have a subwoofer unit, to enhance bass output.



Fig. 1.10 Speakers

Plotter

The plotter is another commonly used output device that is connected to a computer to print large documents, such as engineering or constructional drawings. Plotters use multiple ink pens or inkjets with color cartridges

1.8 Computer Programming

for printing. A computer transmits binary signals to all the print heads of the plotter. Each binary signal contains the coordinates of where a print head needs to be positioned for printing. Plotters are classified on the basis of their performance, as follows:

- **Drum plotter** They are used to draw perfect circles and other graphic images. They use a drawing arm to draw the image. The drum plotter moves the paper back and forth through a roller and the drawing arm moves across the paper.
- **Flat-bed plotter** A flat bed plotter has a flat drawing surface and the two drawing arms that move across the paper sheet, drawing an image. The plotter has a low speed of printing and is large in size.
- **Inkjet plotter** Spray nozzles are used to generate images by spraying droplets of ink onto the paper. However, the spray nozzles can get clogged and require regular cleaning, thus resulting in a high maintenance cost.
- **Electrostatic plotter** As compared to other plotters, an electrostatic plotter produces quality print with highest speed. It uses charged electric wires and special dielectric paper for drawing.

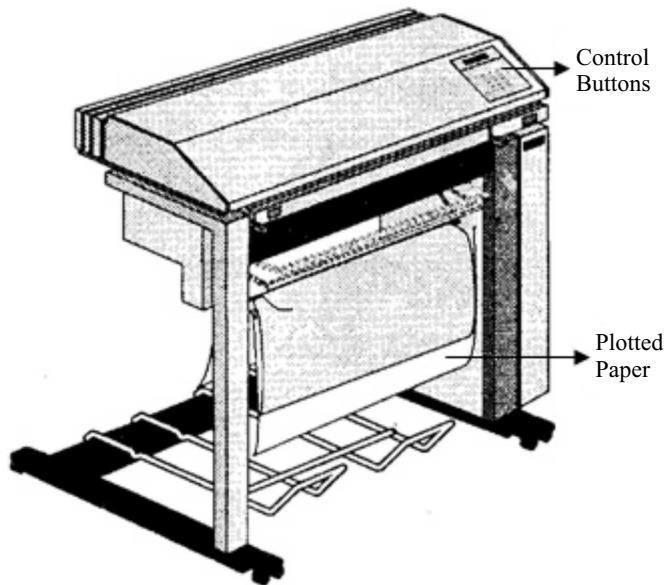


Fig. 1.11 An ink-jet plotter.

1.2.4 Memory

The memory unit of a computer is used to store data, instructions for processing data, intermediate results of processing and the final processed information. The memory units of a computer are classified as primary memory and secondary memory. Figure 1.12 shows the memory categorization in a computer system.

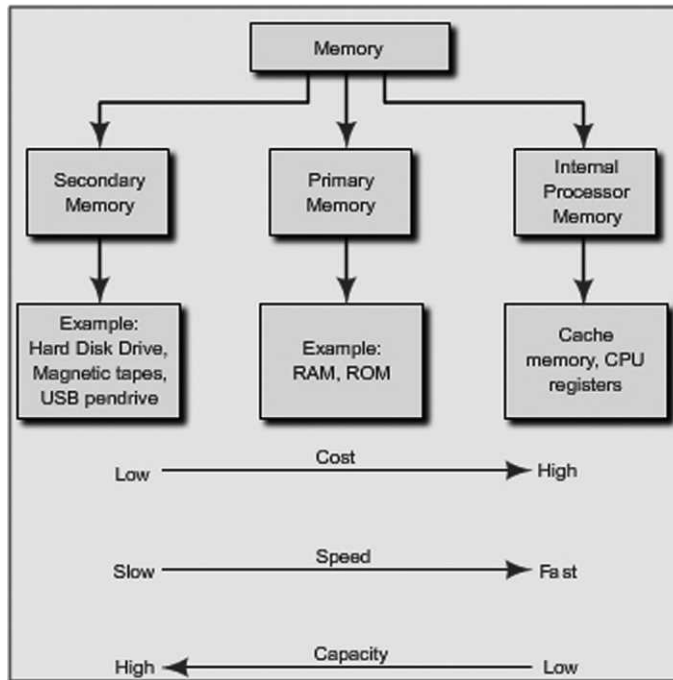


Fig. 1.12 Categorization of Memory Devices

Primary Memory

The primary memory is available in the computer as a built-in unit of the computer. The primary memory is represented as a set of locations with each location occupying 8 bits. Each bit in the memory is identified by a unique address. The data is stored in the machine-understandable binary form in these memory locations. The commonly used primary memories are:

- **ROM** ROM represents Read Only Memory that stores data and instructions, even when the computer is turned off. It is the permanent memory of the computer where the contents cannot be modified by an end user. ROM is a chip that is inserted into the motherboard. It is generally used to store the Basic Input/Output system (BIOS), which performs the Power On Self Test (POST).
- **RAM** RAM is the read/write memory unit in which the information is retained only as long as there is a regular power supply. When the power supply is interrupted or switched off, the information stored in the RAM is lost. RAM is a volatile memory that temporarily stores data and applications as long as they are in use. When the use of data or the application is over, the content in RAM is erased.
- **Cache memory** Cache memory is used to store the data and the related application that was last processed by the CPU. When the processor performs processing, it first searches the cache memory and then the RAM, for an instruction. The cache memory is always placed between CPU and the main memory of the computer system.

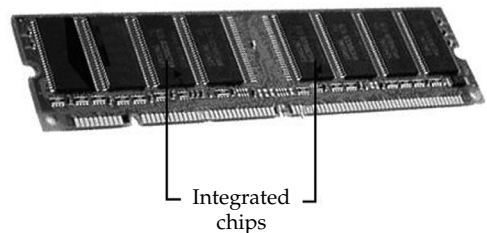


Fig. 1.13 RAM

1.10 Computer Programming

Table 1.1 depicts some of the key differences between RAM and ROM:

TABLE 1.1

RAM	ROM
It is a read/write memory	It is a read only memory
It is volatile storage device	It is a permanent storage device
Data is erased as soon as power supply is turned off	Data remains stored even after power supply has been turned off
It is used as the main memory of a computer system	It is used to store Basic input output system (BIOS).

Secondary Memory

Secondary memory represents the external storage devices that are connected to the computer. They provide a non-volatile memory source used to store information that is not in use currently. A storage device is either located in the CPU casing of the computer or is connected externally to the computer. The secondary storage devices can be classified as:

- **Magnetic storage device** The magnetic storage devices store information that can be read, erased and rewritten a number of times. These include floppy disk, hard disk and magnetic tapes.
- **Optical storage device** The optical storage devices are secondary storage devices that use laser beams to read the stored data. These include CD-ROM, rewritable compact disk (CD-RW), and digital video disks with read only memory (DVD-ROM).

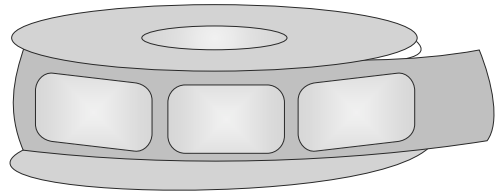


Fig. 1.14 Magnetic tape

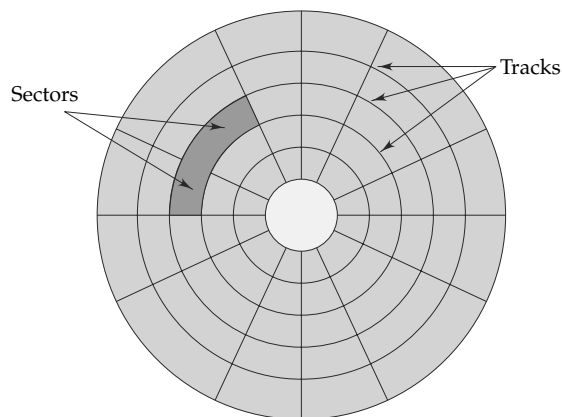


Fig. 1.15 Magnetic disk

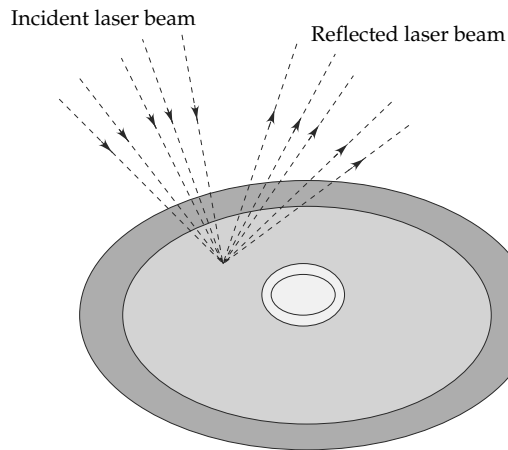


Fig. 1.16 Optical Disk

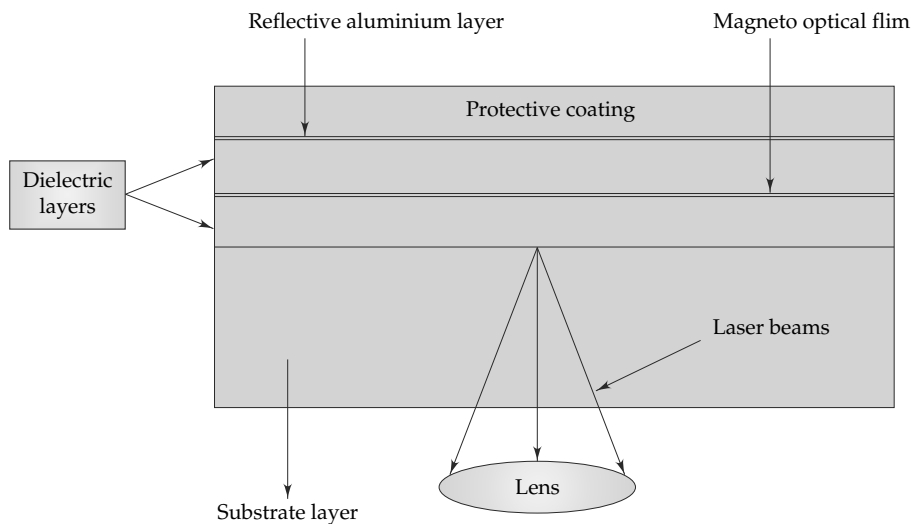


Fig. 1.17 Magneto-optical disk

- Magneto-optical storage device** The magneto-optical devices are generally used to store information, such as large programs, files and backup data. The end user can modify the information stored in magneto-optical storage devices multiple times. These devices provide higher storage capacity as they use laser beams and magnets for reading and writing data to the device. Examples of magneto-optical devices include Sony MiniDisc, Maxoptix T5-2600, etc.
- Universal serial bus (USB) drive** USB drive or commonly known as pen drive is a removable storage device that is interfaced on the USB port of a computer system. It is pretty fast and compact



Fig. 1.18 USB drive

1.12 Computer Programming

in comparison to other storage devices like CD and floppy disk. One of the most important advantages of a USB drive is that it is larger in capacity as compared to other removable storage devices. Off late, it has become very popular amongst computer users.

1.3 PROGRAMMING LANGUAGES

The operations of a computer are controlled by a set of instructions (called *a computer program*). These instructions are written to tell the computer:

1. what operation to perform
2. where to locate data
3. how to present results
4. when to make certain decisions

The communication between two parties, whether they are machines or human beings, always needs a common language or terminology. The language used in the communication of computer instructions is known as the programming language. The computer has its own language and any communication with the computer must be in its language or translated into this language.

Three levels of programming languages are available. They are:

1. machine languages (low level languages)
2. assembly (or symbolic) languages
3. procedure-oriented languages (high level languages)

1.3.1 Machine Language

As computers are made of two-state electronic devices they can understand only pulse and no-pulse (or '1' and '0') conditions. Therefore, all instructions and data should be written using *binary codes* 1 and 0. The binary code is called the *machine code* or *machine language*.

Computers do not understand English, Hindi or Tamil. They respond only to machine language. Added to this, computers are not identical in design, therefore, each computer has its own machine language. (However, the script 1 and 0, is the same for all computers). This poses two problems for the user.

First, it is difficult to understand and remember the various combinations of 1's and 0's representing numerous data and instructions. Also, writing error-free instructions is a slow process.

Secondly, since every machine has its own machine language, the user cannot communicate with other computers (If he does not know its language). Imagine a Tamilian making his first trip to Delhi. He would face enormous obstacles as the language barrier would prevent him from communicating.

Machine languages are usually referred to as the *first generation* languages.

1.3.2 Assembly Language

The Assembly language, introduced in 1950s, reduced programming complexity and provided some standardization to build an application. The assembly language, also referred to as the *second-generation* programming language, is also a low-level language. In an assembly language, the 0s and 1s of machine language are replaced with abbreviations or mnemonic code.

The main advantages of an assembly language over a machine language are:

- As we can locate and identify syntax errors in assembly language, it is easy to debug it.
- It is easier to develop a computer application using assembly language in comparison to machine language.
- Assembly language operates very efficiently.

An assembly language program consists of a series of instructions and mnemonics that correspond to a stream of executable instructions. An assembly language instruction consists of a mnemonic code followed by zero or more operands. The mnemonic code is called the *operation code* or *opcode*, which specifies the operation to be performed on the given arguments. Consider the following machine code:

```
10110000 01100001
```

Its equivalent assembly language representation is:

```
mov al, 061h
```

In the above instruction, the opcode “move” is used to move the hexadecimal value 61 into the processor register named ‘al’. The following program shows the assembly language instructions to subtract two numbers:

```
ORG 500           /Origin of program is location 500
LDA SUB           /Load subtrahend to AC
CMA               /Complement AC
INC               /Increment AC
ADD MIN           /Add minuend to AC
STA DIF           /Store difference
HLT               /Halt computer
MIN, DEC 56       /Minuend
SUB, DEC -2        /Subtrahend
DIF, HEX 0 /Difference stored here
END /End of symbolic program
```

It should be noted that during execution, the assembly language program is converted into the machine code with the help of an *assembler*. The simple assembly language statements had one-to-one correspondence with the machine language statements. This one-to-one correspondence still generated complex programs. Then, macroinstructions were devised so that multiple machine language statements could be represented using a single assembly language instruction. Even today programmers prefer to use an assembly language for performing certain tasks such as:

- To initialize and test the system hardware prior to booting the operating system. This assembly language code is stored in ROM
- To write patches for disassembling viruses, in anti-virus product development companies
- To attain extreme optimization, for example, in an inner loop in a processor-intensive algorithm
- For direct interaction with the hardware
- In extremely high-security situations where complete control over the environment is required
- To maximize the use of limited resources, in a system with severe resource constraints

1.3.3 High-Level Languages

High level languages further simplified programming tasks by reducing the number of computer operation details that had to be specified. High level languages like COBOL, Pascal, FORTRAN, and C are more abstract, easier to use, and more portable across platforms, as compared to low-level programming languages. Instead of dealing with registers, memory addresses and call stacks, a programmer can concentrate more on the logic to solve the problem with help of variables, arrays or Boolean expressions. For example, consider the following assembly language code:

```
LOAD A
ADD B
```

1.14 Computer Programming

STORE C

Using FORTRAN, the above code can be represented as:

$C = A + B$

The above high-level language code is executed by translating it into the corresponding machine language code with the help of a compiler or interpreter.

High-level languages can be classified into the following three categories:

- Procedure-oriented languages (third generation)
- Problem-oriented languages (fourth generation)
- Natural languages (fifth generation)

1.3.4 Procedure-oriented Languages

High-level languages designed to solve general-purpose problems are called *procedural languages* or *third-generation languages*. These include BASIC, COBOL, FORTRAN, C, C++, and JAVA, which are designed to express the logic and procedure of a problem. Although, the syntax of these programming languages is different, they use English-like commands that are easy to follow. Another major advantage of third-generation languages is that they are portable. We can use the compiler (or interpreter) on any computer and create the object code. The following program represents the source code in the C language:

```
if( n>10)
{
    do
    {
        n++;
    }while ( n<50);
}
```

The third generation programming languages are considered as domain-specific programming languages because they are designed to develop software applications for a specific field. For example, the third generation programming language, COBOL, was designed to solve a large number of problems specific to the business field.

1.3.5 Problem-oriented Languages

Problem-oriented languages are used to solve specific problems and are known as the *fourth-generation languages*. These include query Languages, Report Generators and Application Generators which have simple, English-like syntax rules. Fourth-generation languages (4 GLs) have reduced programming efforts and overall cost of software development. These languages use either a visual environment or a text environment for program development similar to that of third-generation languages. A single statement in a fourth-generation language can perform the same task as multiple lines of a third-generation language. Further, the programmer just needs to drag and drop from the toolbar, to create various items like buttons, text boxes, labels, etc. Also, the programmer can quickly create the prototype of the software application.

These languages are typically used in the WYSIWYG (What You See Is What You Get) environment to facilitate faster and convenient application development. Visual Studio is one such environment that encompasses a number of programming tools as well multiple programming language support to ensure flexibility to the programmer during application development.

1.3.6 Natural Languages

Natural languages are designed to make a computer to behave like an expert and solve problems. The programmer just needs to specify the problem and the constraints for problem-solving. Natural languages

such as LISP and PROLOG are mainly used to develop artificial intelligence and expert systems. These languages are widely known as *fifth generation* languages.

The programming languages of this generation mainly focus on constraint programming, which is somewhat similar to declarative programming. It is a programming paradigm in which the programmer only needs to specify the solution to be found within the constraints rather than specifying the method of finding the desired solution.

The programming languages of this generation allow the users to communicate with the computer system in a simple and an easy manner. Programmers can use normal English words while interacting with the computer system.

1.4 PROGRAMMING ENVIRONMENT

A programming environment comprises all those components that facilitate the development of a program. These components are largely divided under two categories—programming tools and Application Programming Interfaces (APIs). They are regarded as the building blocks of any programming environment.

An API can be defined as a collection of data structures, classes, protocols, and pre-defined functions stored in the form of libraries. These libraries are included in the software packages of the programming languages like C, C++, etc. An API makes the development task easier for the programmers, as in-built API components are used again and again, ensuring reusability.

The software application which is used for the development, maintenance and debugging of a software program is known as programming tool. A good programming tool ensures that the programming activities are performed in an efficient manner. The following are some of the main categories of programming tools:

- **Integrated Development Environment (IDE)** It is the most commonly used tool that offers an integrated environment to the programmers for software development. It contains almost all the components for software development such as compiler, editor, debugger, etc.
- **Debugging tool** It is a specialized tool that helps the programmer to detect and remove bugs or errors from a program.
- **Memory usage tool** As the name suggests, memory usage tool helps the programmer to manage the memory resources in an efficient manner.

1.5 CREATING AND RUNNING PROGRAMS

A programmer should adopt standard methodologies and approaches to program development. Software or *program-development life cycle* is one such standard methodology that is applicable to all types of program development scenarios. It comprises of a number of interlinked phases with each phase serving a definite purpose. We will study the program development life cycle in more detail in the next section.

1.5.1 Structured Programming

Another important program development approach is structured programming, which is a subset of one of the key programming paradigms, i.e., procedural programming.

It helps in making a program easily understandable and debuggable. A program that is not based on the structured programming approach is very difficult to maintain, debug and understand.

Structured programming approach mainly focuses on the order of execution of the statements within a program. It suggests the use of sequential execution of statements in a program. Thus, structured programming approach suggests the use of mainly three types of control structures—sequential, repetitive and selective.

1.16 Computer Programming

Further, it suggests avoiding the use of **goto**, **break** and **continue** statements in a program as all these are unconditional branch statements.

1.5.2 System Development Tools

The successful development and execution of programs requires the usage of a number of tools. Some of these typical system development tools are

- Language translators
- Linkers
- Debuggers
- Editors

Language Translators

- **Assembler** An assembler is a computer program that translates assembly language statements into machine language codes. The assembler takes each of the assembly language statements from the source code and generates a corresponding bit stream using 0's and 1's. The output of the assembler in the form of sequence of 0's and 1's is called *object code* or *machine code*. This machine code is finally executed to obtain the results.

A modern assembler translates the assembly instruction mnemonics into opcodes and resolves symbolic names for memory locations and other entities to create the object code. Several sophisticated assemblers provide additional facilities that control the assembly process, facilitate program development, and aid debugging. The modern assemblers like Sun SPARC and MIPS based on RISC architectures, optimize instruction scheduling to attain efficient utilization of CPU. The modern assemblers generally include a macro facility and are called *macro assemblers*.

Assemblers can be classified as *single-pass assemblers* and *two-pass assemblers*. The single-pass assembler was the first assembler that processed the source code once to replace the mnemonics with the binary code. The single-pass assembler was unable to support advanced source-code optimization. As a result, the two-pass assembler was developed that read the program twice. During the first pass, all the variables and labels are read and placed into the symbol table. On the second pass, the label gaps are filled from the table by replacing the label name with the address. This helps to attain higher optimization of the source code. The translation process of an assembler consists of the following tasks:

- Replacing symbolic addresses like LOOP, by numeric addresses
- Replacing symbolic operation code by machine operation codes
- Reserving storage for the instructions and data
- Translating constants into their machine representation

- **Compiler** The compiler is a computer program that translates the source code written in a high-level language into the corresponding *object code* of the low-level language. This translation process is called *compilation*. The entire high-level program is converted into the executable machine code file. A program that translates from a low-level language to a high-level one is a decompiler. Compiled languages include COBOL, FORTRAN, C, C++, etc.

In 1952, Grace Hopper wrote the first compiler for the A-0 programming language. In 1957, John Backus at IBM introduced the first complete compiler. With the increasing complexity of computer architectures and expanding functionality supported by newer programming languages, compilers have become more

and more complex. Though early compilers were written in assembly languages, nowadays it has become common practice to implement a compiler in the language it compiles. Compilers are also classified as *single-pass compilers* and *multi-pass compilers*. Though single-pass compilers are generally faster than multi-pass compilers, for sophisticated optimization, multi-pass assemblers are required to generate high-quality code.

- **Interpreter** The interpreter is a translation program that converts each high-level program statement into the corresponding machine code. This translation process is carried out just before the program statement is executed. Instead of the entire program, one statement at a time is translated and executed immediately. The commonly used interpreted language are BASIC and PERL. Although, interpreters are easier to create as compared to compilers, the compiled languages can be executed more efficiently and are faster.

Linkers

Most of the high-level languages allow the developer to develop a large program containing multiple modules. Linker arranges the object code of all the modules that have been generated by the language translator into a single program. The execution unit of the computer system is incapable of linking all the modules at the execution time and therefore, linker is regarded as one of the important software because of its ability to combine all the modules into a single program. Linker assembles the various objects generated by the compiler in such a manner that all the objects are accepted as a single program during execution. Linker also includes the links of various objects, which are defined in the runtime libraries. In many cases, linker inserts the symbolic address of the objects in place of their real address. Figure 1.19 illustrates the working of a linker.

Debuggers

Debugger is the software that is used to detect the errors and bugs present in the programs. The debugger locates the position of the errors in the program code with the help of what is known as the Instruction Set

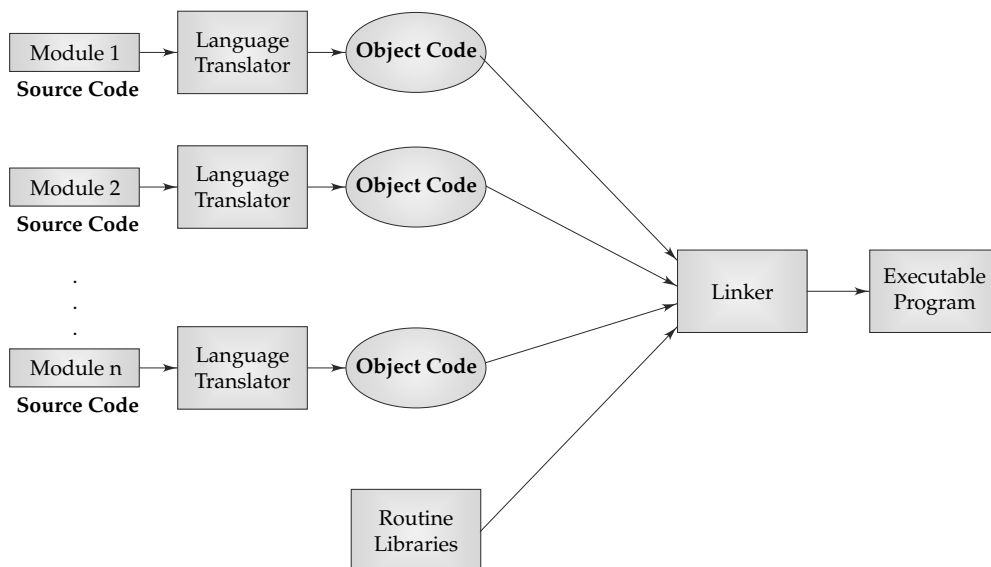


Fig. 1.19 Working of a linker

1.18 Computer Programming

Simulator (ISS) technique. ISS is capable of stopping the execution of a program at the point where an erroneous statement is encountered.

Debugger is divided into two types, namely machine-level debugger and symbolic debugger. The machine-level debugger debugs the object code of the program and shows all the lines where bugs are detected. On the other hand, the symbolic debugger debugs the original code, i.e., the high-level language code of the program. It shows the position of the bug in the original code of the program developed by the programmer.

While debugging a program, the debugger performs a number of functions other than debugging, such as inserting breakpoints in the original code, tracking the value of specific variables, etc. In order to debug the program, a debugger helps perform the following tasks:

- Step-by-step execution of a program
- Back tracking for checking the previous steps
- Stopping the execution of the program until the errors are corrected

Editors

Editor is a special program that allows the user to work with text in a computer system. It is used for the documentation purposes and enables us to edit the information present in an existing document or a file. The editor enables us to perform various editing operations such as copy, cut and paste while editing the text. On the basis of the content edited by the editors, they are divided into the following categories:

- **Text editor** It is used to edit plain text. An operating system always includes a text editor for updating the configuration files.
- **Digital audio editor** It is used to edit the information related to the audio components of a multimedia application. These editors are used in audio applications where editing the music and the sound signals is necessary.
- **Graphics editor** It is used to edit the information related to the graphical objects. These editors are generally used in the multimedia applications where the user is working with multiple animation objects.
- **Binary file editor** It is used to edit the digital data or the binary data, i.e., data having strings of 0s and 1s.
- **HTML editor** It is used to edit the information included in the Web pages.
- **Source code editor** It is used to edit the source code of a program written in a programming language such as C, C++ and Java.

1.5.3 Developing a Program

Developing a program refers to the process of writing the source code for the required application by following the syntax and the semantics of the chosen programming language. Syntax and semantics are the set of rules that a programmer needs to adhere while developing a program.

Before actually developing a program, the aim and the logic of the program should be very clear to the programmer. Therefore, the first stage in the development of a program is to carry out a detailed study of the program objectives. The objectives make the programmer aware of the purpose for which the program is being developed. After ascertaining the program objectives, the programmer needs to list down the set of steps to be followed for program development. This set of program development steps is called algorithm. The programmer may also use a graphical model known as flowchart to represent the steps defined in the program algorithm.

After the logic of the program has been developed either by an algorithm or a flowchart, the next step is to choose a programming language for actual development of the program code. There are a number of

factors that should be taken into consideration while selecting the target programming language, such as performance and efficiency of the programming language, programmer's prior experience with the language, etc.

A programming language is typically bundled together with an IDE containing the necessary tools for developing, editing, running and debugging a computer program. For instance, Turbo C is provided with a strong and powerful IDE to develop, compile, debug and execute the programs.

Figure 1.20 shows the IDE of C language.

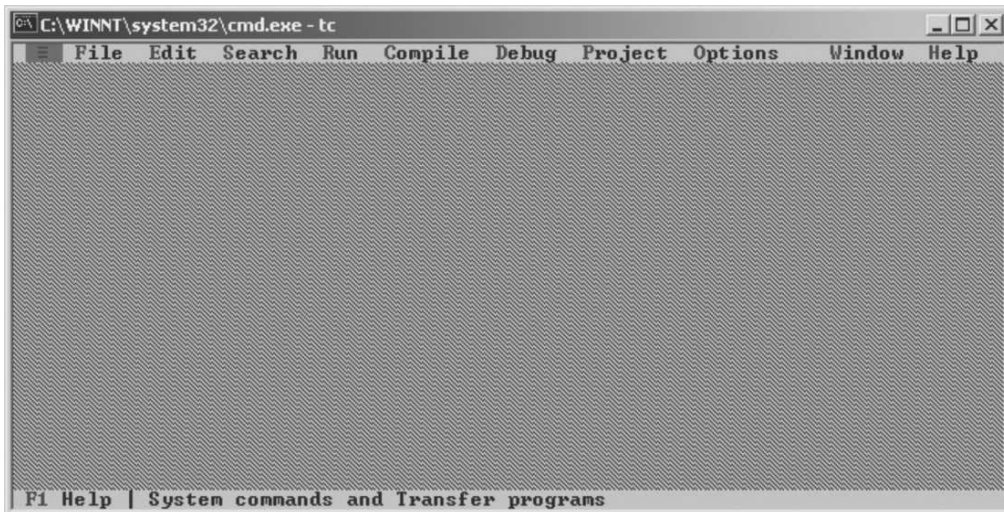


Fig. 1.20 The IDE of C Language.

Suppose we are required to develop a program for calculating the percentage of marks of two subjects for student and display the result. The first step in the development of a program for this problem is the preparation of an algorithm, as shown below:

```
Step 1 - Input the marks for first subject. (mark1)
Step 2 - Input the marks for second subject. (mark2)
Step 3 - Calculate the percentage.
         percentage = (mark1 + mark2)/200*100
Step 4 - If percentage > 40
Step 5 - Display Pass
Step 6 - Else
Step 7 - Display Fail
```

Figure 1.21 shows the flowchart for the above algorithm.

1.20 Computer Programming

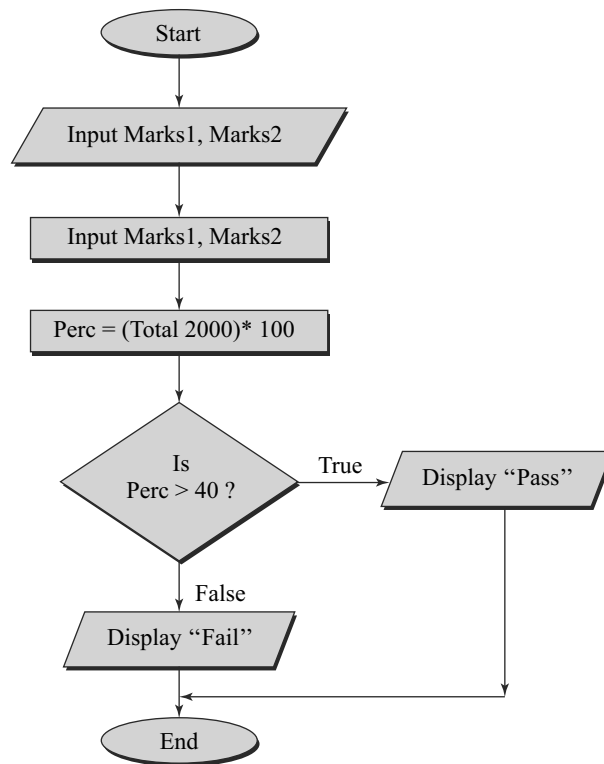


Fig. 1.21 Flowchart for calculating the percentage of marks and displaying the result.

After developing the algorithm and flowchart, the actual development of the program can be started in the source code editor of C language. The following code shows the C language program for calculating the percentage of marks in two different subjects for a student.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    float mark1,mark2;
    float percentage;
    clrscr();
    printf("\n Enter marks of first subject:");
    scanf("\n %f", &mark1);
    printf("\n Enter marks of second subject:");
    scanf("\n %f", &mark2);
    percentage =((mark1+mark2)/200)*100;
    if(percentage>40)
```



```
printf("\n The student is passed");
else
printf("\n The student is failed");
getch();
}
```

Figure 1.22 shows the program code in the source code editor of C language.

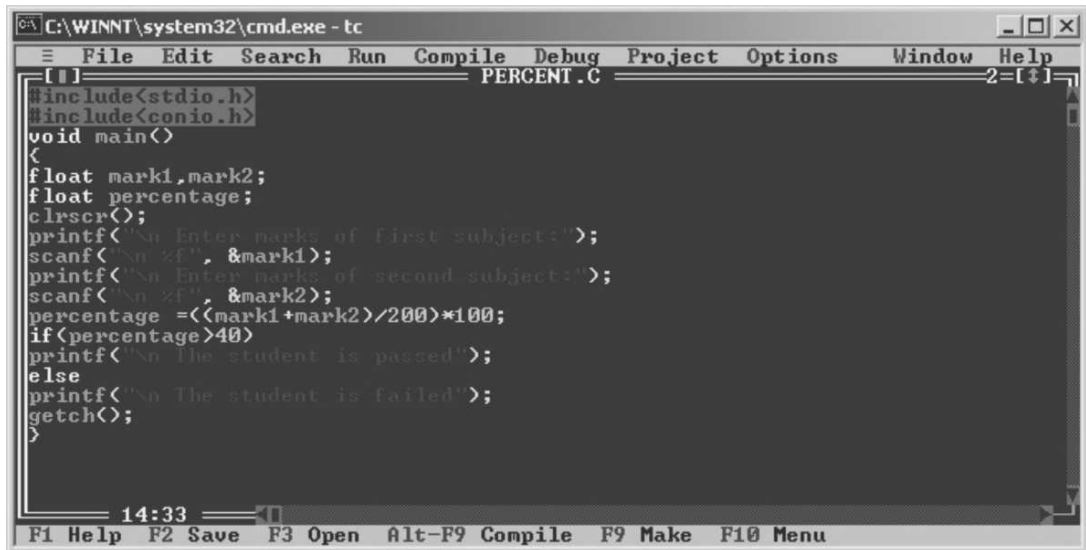


Fig. 1.22 Developing a program in the source code editor of C language.

1.5.4 Running a Program

After developing the program code, the next step is to compile the program. Program compilation helps identify any syntactical errors in the program code. If there are no syntax errors in the source code, then the compiler generates the target object code. It is the machine language code that the processor of the computer system can understand and execute.

Once the corresponding object code or the executable file is built by the compiler, the program can be run in order to check the logical correctness of the program and generate the desired output. The logical errors also called semantic errors might cause the program to generate undesired results. Programming languages provide various mechanisms such as exception handling for handling these logical errors. If the output generated by the program corresponding to the given inputs matches with the desired result, then the purpose of developing the program is served. Otherwise, the logic of the program should be checked again to obtain the correct solution for the given problem.

Figure 1.23 shows the output of the program developed in C language.

1.22 Computer Programming

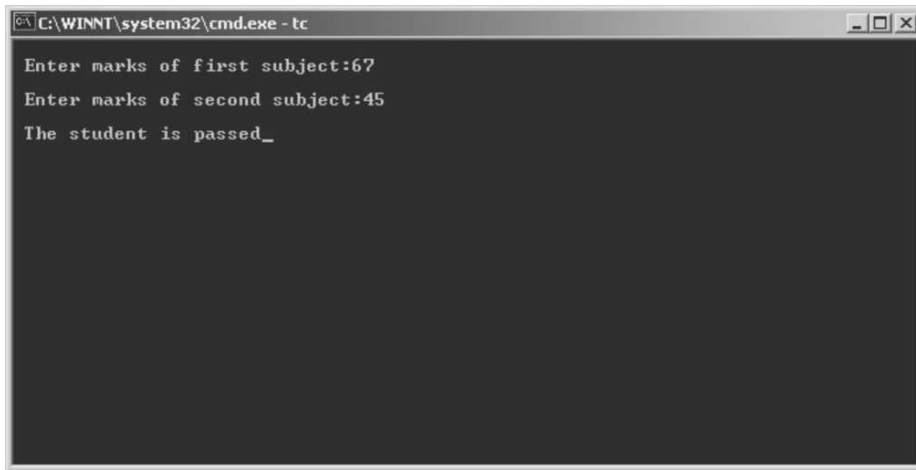


Fig. 1.23 Running a program.

The above figure shows the output generated by running the C program. We can run a program in C by either selecting Run → Run or by pressing the Alt and F9 keys simultaneously.

1.6 SOFTWARE DEVELOPMENT METHOD

The entire process of software development and implementation involves a series of steps. Each successive step is dependent on the outcome of the previous step. Thus, the team of software designers, developers and operators are required to interact with each other at each stage of software development so as to ensure that the end product is as per the client's requirements. Figure 1.24 shows the various software development steps:

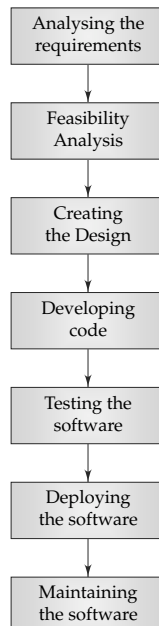


Fig. 1.24 Software development steps

1.6.1 Analysing the Requirements

In this step, the requirements related to the software, which is to be developed, are understood. Analysing the requirements or requirement analysis is an important step in the process of developing a software. If the requirements of the user are not properly understood, then the software is bound to fall short of the end user's expectations. Thus, requirement analysis is always the first step towards development of a software.

Software is abstract in nature; as a result, the users may not be able to provide the complete set of requirements pertaining to the desired software during the requirement analysis stage. Thus, there should be continuous interaction between the software development team and the end users. Moreover, the software development team also needs to take into account the fact that the requirements of the users may keep changing during the development process. Thus, proper analysis of user requirements is quite essential for developing the software within a given timeframe. It will not only help in controlling the software development cost but will also lead to faster and accurate development of a software.

The task of requirement analysis is typically performed by a business analyst. The person is a professional in this field who understands the requirements of the novice end user, and documents and shares it with the development team.

1.6.2 Feasibility Analysis

In this step, the feasibility of developing the software in terms of resources and cost is ascertained. In order to determine the feasibility of software development, the existing system of the user is analysed properly. Apart from studying the existing system, this step involves identifying the need of automation in the existing system. The analysis done in this step is documented in a standard document called feasibility report, which contains the observations and recommendations related to the task of software development. Some of the important activities performed during the feasibility analysis stage are as follows:

- **Determining development alternatives** This activity involves searching for the different alternatives that are available for the development of software. There are mainly four alternatives available for the development of a software. The first alternative is to allow the existing system to continue without developing a new software for automation. The second alternative can be to develop the new software using specific programming languages such as Java, C++, Visual Basic etc. The third alternative is to develop the software using the architectural technologies such as Java 2 Enterprise Edition (J2EE) and mainframe based with thin clients. The fourth development alternative is to buy an already developed software along with its source code from the market and customise it according to the client's requirements.
- **Analysing economic feasibility** This activity involves determining whether the development of a new software will be financially beneficial or not. This type of feasibility analysis is performed to determine the overall profit that can be earned from the development and implementation of the software. This feasibility analysis activity involves evaluating all the alternatives available for development and selecting the one which is most economical.
- **Assessing technical feasibility** The technical feasibility assessment involves analysing various factors such as performance of the technologies, ease of installation, ease of expansion or reduction in size, interoperability with other technologies, etc. The technical feasibility activity typically involves the study of the nature of technology as to how easily it can be learnt and the level of training required to understand the technology. This type of feasibility assessment greatly helps in selecting the appropriate technologies to be used for developing the software. The selection should be made after evaluating the requirement specification of the software. In addition, the advantages and disadvantages of each identified technology must also be evaluated during technical feasibility assessment.

1.24 Computer Programming

- **Analysing operational feasibility** Operational feasibility assessment involves studying the software on operational and maintenance fronts. The operational feasibility of any software is done on the basis of several factors, such as:
 - Type of tools needed for operating the software
 - Skill set required for operating the software
 - Documentation and other support required for operating the software

1.6.3 Creating the Design

After the feasibility analysis stage, the next step is creating the architecture and design of the new software. This step involves developing a logical model or basic structure of the new software. For example, if the new software is based on client–server technology then this step would involve determining and specifying the number of tiers to be used in the client–server design. This step also involves documenting the varied specifications pertaining to database and data structure design. The flow of the development process is mainly illustrated in this stage using a special language known as Unified Modelling Language (UML). UML uses pictorial representation methods for depicting the flow of data in the software. Some of the key features, which are considered while designing a software, are:

- **Extensibility** The design of the software should be extensible so that it allows the addition of some new options or modules in future. The architecture of the software should be flexible enough to not get disturbed with the addition of new functionality.
- **Modularity** The software should be modular in nature so that its working and data flow can be understood easily. Modularity also helps in parallel development of the various software modules, which are later integrated into a single software product.
- **Compatibility** Software should run correctly in the existing system with an older version or with other software. Thus, software should be compatible and work well in conjunction with other software.
- **Security** Software must be able to control unauthorised access. While designing a new software, it is ensured that there are proper security mechanisms incorporated in the product.
- **Fault tolerance** The software should be capable of handling exceptions or faults that may occur during its operation. The software must have the capability to recover from failures.
- **Maintainability** The design of the software should be created in a simple manner with appropriate details so that it is easy to maintain.

1.6.4 Developing Code

In this step, the code for the different modules of the new software is developed. The code for the different modules is developed according to the design specifications of each module. The programmers in the software development team use tools like compilers, interpreters and debuggers to perform tasks such as finding errors in the code and converting the code into machine language for its execution. The code can be written using programming languages such as C, C++ or Java. The choice of the programming language to be used for developing the code is made on the basis of the type of software that is to be developed. There are certain key points or conventions, which must be kept in mind while writing code; for instance:

- There should be proper indentation in the code.
- Proper naming conventions should be followed for naming the variables, methods and program files.
- Proper comments should be included to ensure ease of understanding during maintenance.

- The code for different modules of the new software must be simple so that it can be easily understood.
- The code must be logically correct so as to minimise logical errors in the program.

1.6.5 Testing the Software

Testing is basically performed to detect the prevalence of any errors in the new software and rectify those errors. One of the reasons for the occurrence of errors or defects in a new software is that the requirements of the users or client were not properly understood. Another reason for the occurrence of errors is the common mistakes committed by a programmer while developing the code. The two important activities that are performed during testing are verification and validation. Verification is the process of checking the software based on some pre-defined specifications, while validation involves testing the product to ascertain whether it meets the user's requirements. During validation, the tester inputs different values to ascertain whether the software is generating the right output as per the original requirements. The various testing methodologies include:

- Black box testing
- White box testing
- Gray box testing
- Nonfunctional testing
- Unit testing
- Integration testing
- System testing
- Acceptance testing

1.6.6 Deploying the Software

In this step, the newly developed and fully tested software is installed in its target environment. Software documentation is handed over to the users and some initial data are entered in the software to make it operational. The users are also given training on the software's interface and its other functions.

1.6.7 Maintaining the Software

Once the software has been deployed successfully, a continuous support is provided to it for ensuring its full-time availability. A corrupt file, a virus infection and a fatal error are some of the situations where the maintenance personnel are asked to fix the software and bring it back to its normal functioning. Further, a software may also be required to be modified if its environment undergoes a change. In order to successfully maintain the software, it is required that it should have been properly documented at the time of its development. This is because the maintenance person might not be the same who was originally involved in the development of the software. Thus, a good code documentation serves vital for the maintenance person to fix the software.

1.7 APPLYING SOFTWARE DEVELOPMENT METHOD

To understand how software development method is applied, consider a simple scenario where it is required to convert the temperature given in Fahrenheit to its corresponding Celsius value.

1.26 Computer Programming

Program Objective

To convert the temperature value from Fahrenheit to Celsius

Analysis

Input: Temperature value in Fahrenheit

Output: Temperature value in Celsius

Conversion method: The formula $C = (F - 32) / 1.8$ can be used to generate the desired output

Data elements: Real Variable F is used to store the input temperature value in Fahrenheit

Real Variable C is used to store the resultant temperature value in Celsius

Design

Algorithm

```
Step 1 - Read F
Step 2 - Compute C = (F-32) / 1.8
Step 3 - Display C
```

Development

Program

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

void main()
{
    float F, C;
    clrscr();

    printf("Enter the temperature value in Fahrenheit: ");
    scanf("%f",&F);

    C=(F-32.0)/1.8;

    printf("The equivalent temperature value in degrees Celsius is: %.2f",C);
    getch();
}
```

Testing

The program must be tested with multiple input values so as to ensure that there are no logical errors present in the code.

```
Enter the temperature value in Fahrenheit: 0
The equivalent temperature value in degrees Celsius is: -17.78
```

```

Enter the temperature value in Fahrenheit: 175
The equivalent temperature value in degrees Celsius is: 79.44

Enter the temperature value in Fahrenheit: 250
The equivalent temperature value in degrees Celsius is: 121.11

```

1.8 PROBLEM SOLVING

Problems that can be solved through a computer may range in size and complexity. Since computers do not possess any common sense and cannot make any unplanned decisions, the problem, whether it is simple or complex, has to be broken into a well-defined set of solution steps for the computer to implement.

Problem solving is the process of solving a problem in a computer system by following a sequence of steps, which include the following:

1. **Developing the algorithm** An algorithm is a sequence of steps written in the form of English phrases that specify the tasks that are performed while solving a problem. It involves identifying the variable names and types that would be used for solving the problem.
2. **Drawing the flowchart** A flowchart is the graphical representation of the flow of control and logic in the solution of a problem. The flowchart is a pictorial representation of an algorithm.
3. **Writing the Pseudocode** Pseudocode is pretty much similar to algorithms. It uses generic syntax for describing the steps that are to be performed for solving a problem. Along with the statements written using generic syntax, pseudocode can also use English phrases for describing an action.

1.9 ALGORITHMS

Algorithms help a programmer in breaking down the solution of a problem into a number of sequential steps. Corresponding to each step a statement is written in a programming language; all these statements are collectively termed as a program.

The following is an example of an algorithm to add two integers and display the result:

Algorithm

```

Step 1 - Accept the first integer as input from the user.
         (num1)
Step 2 - Accept the second integer as input from the user.
         (num2)
Step 3 - Calculate the sum of the two integers.
         (sum = num1 + num2)
Step 4 - Display sum as the result.

```

There is a time and space complexity associated with each algorithm. Time complexity specifies the amount of time required by an algorithm for performing the desired task. Space complexity specifies the amount of memory space required by the algorithm for performing the desired task. While solving a complex problem, it is possible to have multiple algorithms for obtaining the required solution. The algorithm that ensures best time and space trade off should be chosen for obtaining the desired solution.

1.28 Computer Programming

1.9.1 Characteristics of Algorithms

The typical characteristics that are necessary for a sequence of instructions to qualify as an algorithm are the following:

- The instructions must be in an ordered form.
- The instructions must be simple and concise. They must not be ambiguous.
- There must be an instruction (condition) for program termination.
- The repetitive programming constructs must possess an exit condition. Otherwise, the program might run infinitely.
- The algorithm must completely and definitely solve the given problem statement.

1.9.2 Advantages of Algorithms

Some of the key advantages of algorithms are the following:

- It provides the core solution to a given problem. This solution can be implemented on a computer system using any programming language of user's choice.
- It facilitates program development by acting as a design document or a blueprint of a given problem solution.
- It ensures easy comprehension of a problem solution as compared to an equivalent computer program.
- It eases identification and removal of logical errors in a program.
- It facilitates algorithm analysis to find out the most efficient solution to a given problem.

1.9.3 Disadvantages of Algorithms

Apart from the advantages, algorithms also possess certain limitations, which are

- In large algorithms, the flow of program control becomes difficult to track.
- Algorithms lack visual representation of programming constructs like flowcharts; thus, understanding the logic becomes relatively difficult.

Example 1.1 Write an algorithm to find out whether a given number is prime or not.

Solution

Algorithm

```
Step 1 - Start
Step 2 - Accept a number from the user (num)
Step 3 - Initialize looping counter i = 2
Step 4 - Repeat Step 5 while i < num
Step 5 - If remainder of num divided by i (num%i) is Zero then goto Step 6 else goto Step 4
Step 6 - Display "num is not a prime number" and break from the loop
Step 7 - If i = num then goto Step 8 Else goto Step 9
Step 8 - Display "num is a prime number"
Step 9 - Stop
```

Example 1.2 Write an algorithm to find the average of marks obtained by a student in three subjects.

Solution**Algorithm**

```
Step 1 - Start
Step 2 - Accept the marks in three subjects from the user (marks1, marks2, marks3)
Step 3 - Calculate average marks using formula, average = (marks1 + marks2 + marks3)/3
Step 4 - Display the computed average of three subject marks
Step 5 - Stop
```

Example 1.3 Write an algorithm to determine whether the given year is a leap year or not.

Solution**Algorithm**

```
Step 1 - Start
Step 2 - Accept an year value from the user (year)
Step 3 - If remainder of year value divided by 4 (year%4) is 0 then goto Step 4 else goto Step 5
Step 4 - Display "'year' is a leap year" and goto Step 6
Step 5 - Display "'year' is not a leap year"]
Step 6 - Stop
```

Example 1.4 Write an algorithm to find out whether a given number is even or odd.

Solution**Algorithm**

```
Step 1 - Start
Step 2 - Accept a number from the user (num)
Step 3 - If remainder of num divided by 2 (num/2) is Zero then goto Step 4 else goto Step 5
Step 4 - Display "num is an even number" and goto Step 6
Step 5 - Display "num is an odd number"
Step 6 - Stop
```

Example 1.5 Write an algorithm to determine whether a given string is a palindrome or not.

Solution

Algorithm

```

Step 1 - Start
Step 2 - Accept a string from the user (str)
Step 3 - Calculate the length of string str (len)
Step 4 - Initialize looping counters left=0, right=len-1 and chk = 't'
Step 5 - Repeat Steps 6-8 while left < right and chk = 't'
Step 6 - If str(left) = str(right) goto Step 8 else goto step 7
Step 7 - Set chk = 'f'
Step 8 - Set left = left + 1 and right = right - 1
Step 9 - If chk='t' goto Step 10 else goto Step 11
Step 10 - Display "The string is a palindrome" and goto Step 12
Step 11 - Display "The string is not a palindrome"
Step 12 - Stop

```

1.10 FLOWCHARTS

A flowchart can be defined as the pictorial representation of a process, which describes the sequence and flow of control and information within the process. The flow of information is represented inside the flowchart in a step-by-step form. This technique is mainly used for developing business workflows and solving problems using computers.

A flowchart uses different symbols for depicting different activities, which are performed at different stages of a process. The various symbols used in a flowchart are the following:

- **Start and end** It is represented by an oval or a rounded rectangle. It represents the starting and the ending of a process. Every process starts and ends at some point so a flowchart always contains one start as well as one end symbol. Figure 1.25 shows the start and the end symbols used in a flowchart.

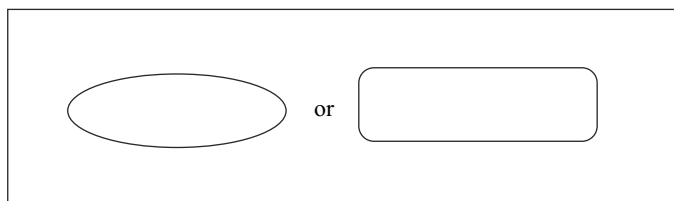


Fig. 1.25 Start and end symbol

- **Input or output** It is represented by a parallelogram. It represents the inputs given by the user to the process and the outputs given by the process to the user. Figure 1.26 shows the input or output symbol.



Fig. 1.26 Input or output symbol

- **Action or process** It is represented by a rectangle. It represents the actions, logics and calculations taking place in a process. Figure 1.27 shows the action or process symbol.



Fig. 1.27 Action or process symbol

- **Decision or condition** It is represented by a rhombus or a diamond shape. It represents the condition or the decision-making step in a flowchart. The result of the decision is a Boolean value, which is either true or false. Each of these values takes the flow of the program to a certain point, which is shown with the help of arrows. Figure 1.28 shows the decision or condition symbol.

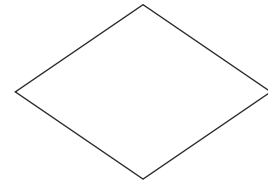


Fig. 1.28 Decision or condition symbol

- **Arrow** It is represented by a directed line. It represents the flow of process and the sequence of steps in a flowchart. It guides the process about the direction and the sequence, which is to be followed while performing the various steps in the process. Figure 1.29 shows the arrow symbol.

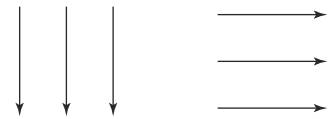


Fig. 1.29 Arrow symbol

- **Connector** It is represented by a circle in a flowchart. It represents the continuation of the flow of steps when a flowchart continues to the next page. A character such as an alphabet (a to z) or a symbol (α , β or χ), etc. can be placed in the circle at the position where the flow is broken and the same character is also placed in the circle at the position from where the flowchart continues. Figure 1.30 shows the connector symbol.

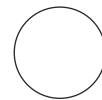
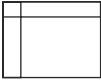
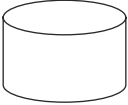








Fig. 1.30 Connector symbol

In addition to the above-mentioned basic flowchart symbols, certain other symbols are also used inside flowcharts for depicting advanced operations. Table 1.1 shows some of these advanced flowchart symbols:

Symbol	Description
	Represents the internal memory of a computer system, such as RAM and ROM
	Represents a database
	Represents a subroutine

(Contd.)

Symbol	Description
	Represents a set of documents
	Represents an idle or waiting state
	Merges multiple data sets into one
	Extracts individual sets of data items from a single data set
	Represents a combination of merge and extract actions. It extracts multiple data sets from multiple input data sets.

In order to understand how a flowchart represents the flow of information; consider an example of a flowchart for adding two numbers, as shown in Fig. 1.31.

Flowchart

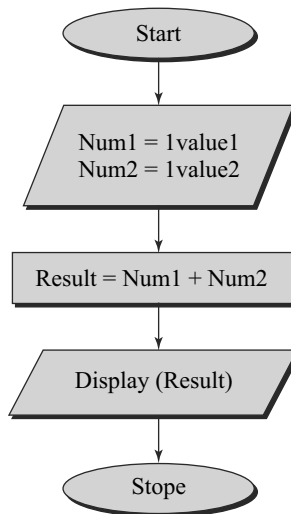


Fig. 1.31 Flowchart for addition of two numbers.

1.10.1 Flowchart Design Rules

Some of the standard guidelines or rules that must be followed while designing a flowchart are

- It must begin with a "Start" and end with a "Stop" symbol.
- The standard process flow should be either from top to bottom or from left to right.
- The instructions specified in the flowchart must be crisp and concise.
- The arrows must be aligned properly so as to clearly depict the flow of program control.
- The use of connectors should be generally avoided as they make the program look more complex.
- A process or action flowchart symbol must have only one input arrow and one output arrow.
- Two arrows must never intersect or cross each other; if such a need arises, then appropriate bridge or crossover symbols must be used.

1.10.2 Advantages of Flowcharts

Some of the key advantages of using a flowchart in program design are

- It helps to understand the flow of program control in an easy way.
- Developing program code by referring its flow chart is easier in comparison to developing the program code from scratch.
- It helps in avoiding semantic errors.
- Any concept is better understood with the help of visual representation. This fact also holds true for flowcharts. It is easier to understand the pictorial representation of a programming logic.
- A flowchart acts as documentation for the process or program flow.
- The use of flowcharts works well for small program design.

1.10.3 Disadvantages of Flowcharts

Flowcharts also have certain limitations, such as

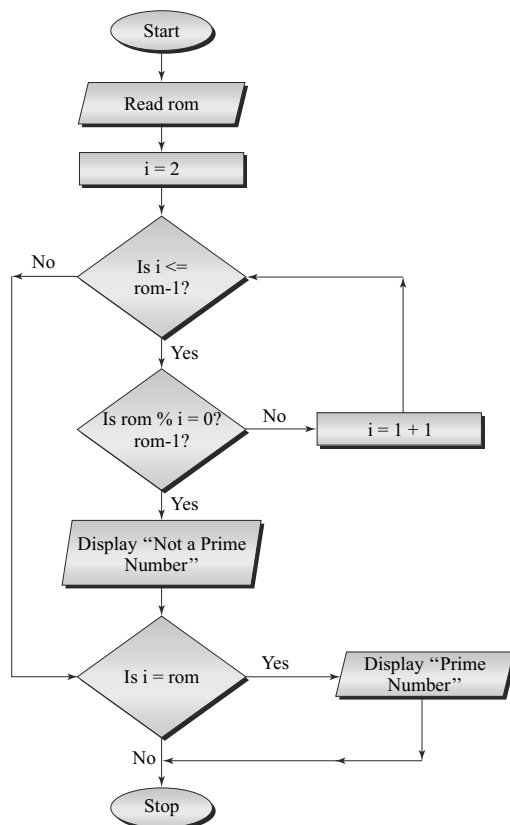
- For a large program, the flow chart might become very complex and confusing.
- Modification of a flowchart is difficult and requires almost an entire rework.
- Since flowcharts require pictorial representation of programming elements, it becomes a little tedious and time consuming to create a flowchart.
- Excessive use of connectors in a flowchart may at times confuse the programmers.

Example 1.6 Draw a flowchart for the problem statement given in Example 1.1.

Solution

Flowchart to find out whether a given number is prime or not:

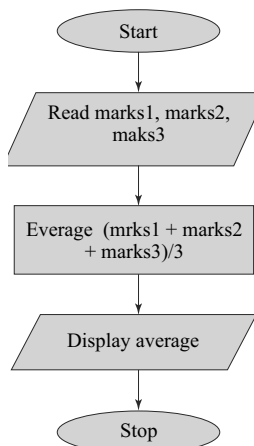
1.34 Computer Programming



Example 1.7 Draw a flowchart for the problem statement given in Example 1.2.

Solution

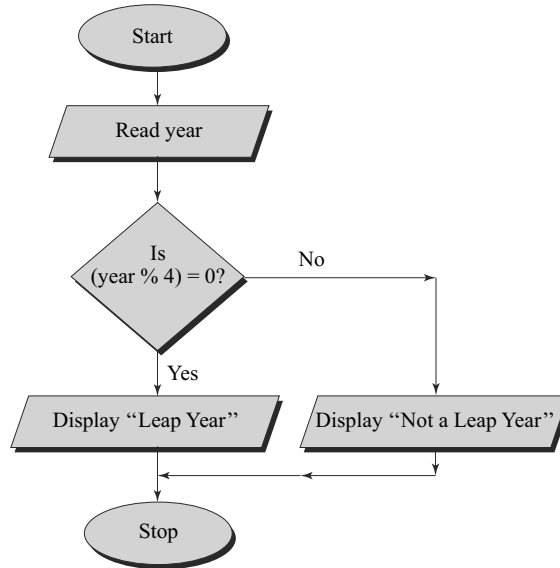
Flowchart to find the average of marks obtained by a student:



Example 1.8 Draw a flowchart for the problem statement given in Example 1.3.

Solution

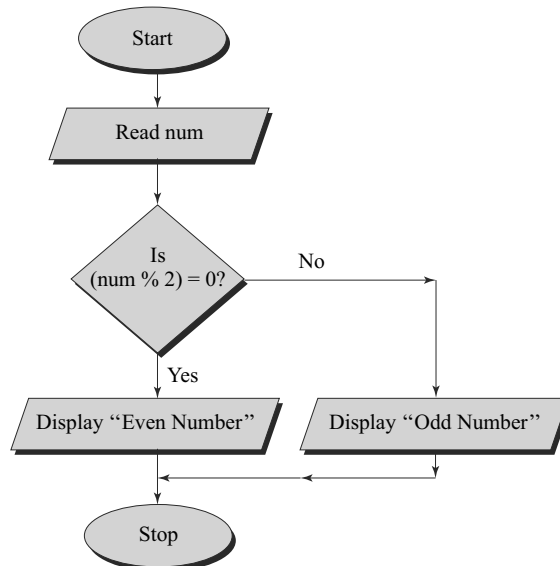
Flowchart to determine whether the given year is a leap year or not:



Example 1.9 Draw a flowchart for the problem statement given in Example 1.4.

Solution

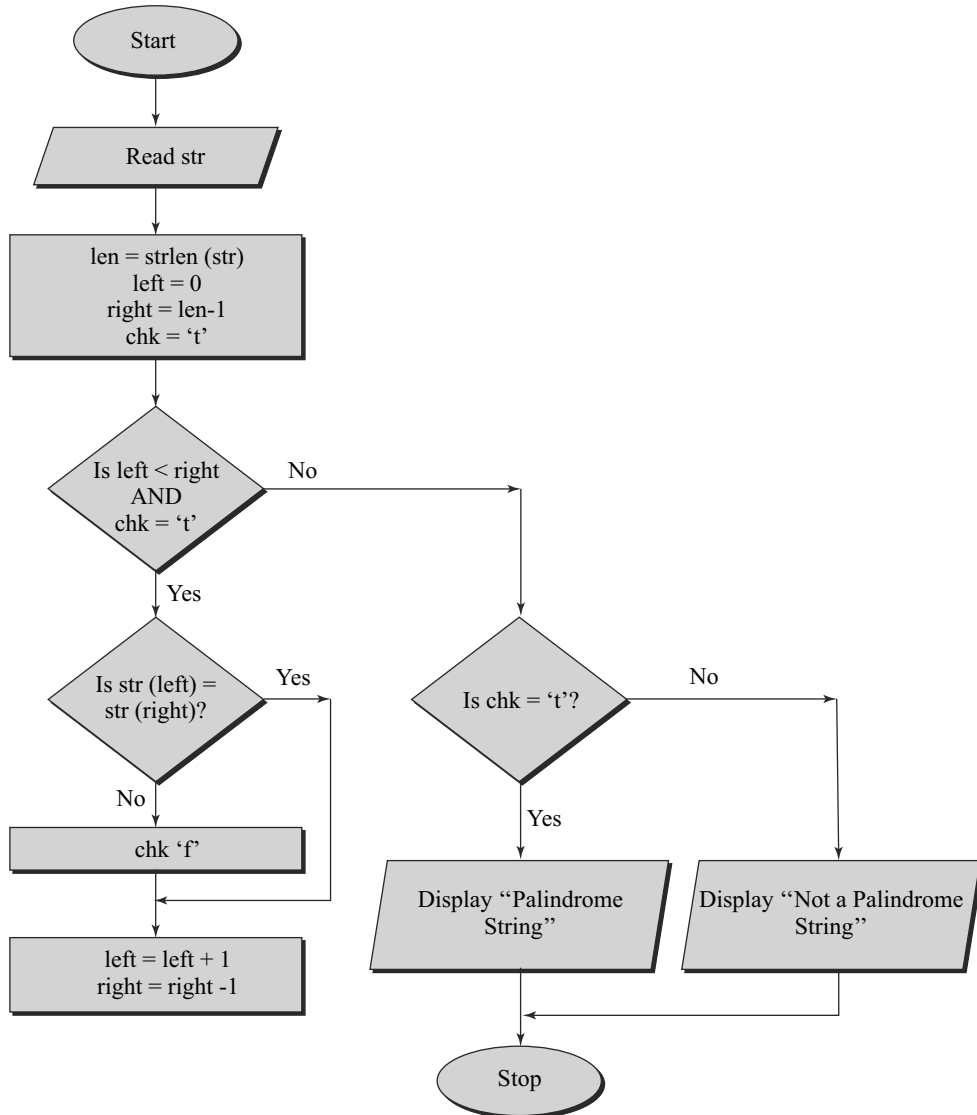
Flowchart to find out whether a given number is even or odd:



Example 1.10 Draw a flowchart for the problem statement given in Example 1.5.

Solution

Flowchart to determine whether a given string is a palindrome or not:



1.11 PSEUDOCODES

Analyzing a detailed algorithm before developing a program is very time consuming. Hence, there arises a need of a specification that only focuses on the logic of the program. Pseudocodes serve this purpose by specifying only the logic, which is used by the programmer for developing a computer program.

Pseudocode is not written using specific syntax of a programming language rather it is written with a combination of generic syntax and normal English language. It helps the programmer understand the basic logic of the program after which it is the programmer's choice to write the final code in any programming language.

The example of a pseudocode to add two numbers and display the result is shown below:

Pseudocode

```
DEFINE: Integer num1, num2, result
READ: Integer num1
READ: Integer num2
SET: result = num1 + num2
Display: result
```

After the pseudocode for a computer program has been written, it is used to develop the source code for the computer program. The source code is developed using a programming language, which can be an assembly language or a high level programming language.

1.11.1 Pseudocode Rules

Some of the standard guidelines or rules that must be followed while developing pseudocodes are given below.

- Instruction or operation code in a pseudocode statement must be kept all capitalized; for example READ (specifying input operation), PRINT (specifying print command), and so on.
- The ending of looping and decision-making constructs must be labelled appropriately with corresponding "END" keywords.
- The pseudocode instructions must be properly indented just like the statements are indented in a computer program.
- Convoluted programming instructions must be avoided by writing simple and clear instructions.

1.11.2 Advantages of Pseudocodes

Some of the key advantages of using a pseudocode in program design are the following:

- It is easy to comprehend as it uses English phrases for writing program instructions.
- Developing program code using pseudocode is easier in comparison to developing the program code from scratch.
- Developing program code using pseudocode is also easier in comparison to developing the program code from flowchart.
- The pseudocode instructions are easier to modify in comparison to a flowchart.
- The use of pseudocode works well for large program design.

1.11.3 Disadvantages of Pseudocodes

Pseudocodes also have certain limitations, as explained below:

- Since pseudocode does not use any kind of pictorial representations for program elements; it may at times become difficult to understand the program logic.

1.38 Computer Programming

- There is no standard format for developing a pseudocode. Therefore, it may become a challenge to use the same pseudocode by different programmers.
- Pseudocodes are at a disadvantage in comparison to flowcharts when it comes to understanding the flow of program control.

Example 1.11 Write the pseudocode for the problem statement given in Example 1.1.

Solution

Pseudocode to find out whether a given number is prime or not:

Pseudocode

```
BEGIN
DEFINE: Integer num, i
DISPLAY: "Enter a number: "
READ: num
FOR: i = 2 to num-1
    IF: num%i=0
        DISPLAY: "'num' is not a prime number"
        BREAK
    END IF
END FOR
IF: i=num
    DISPLAY: "'num' is a prime number"
END IF
END
```

Example 1.12 Write the pseudocode for the problem statement given in Example 1.2.

Solution

Pseudocode to find the average of marks obtained by a student:

Pseudocode

```
BEGIN
DEFINE: Integer marks1, marks2, marks3
DEFINE: Real average
DISPLAY: "Enter the marks in three subjects: "
READ: marks1, marks2, marks3
COMPUTE: average = (marks1 + marks2 + marks3)/3
DISPLAY: "The average value of marks is 'average'"
END
```

Example 1.13 Write the pseudocode for the problem statement given in Example 1.3.

Solution

Pseudocode to determine whether the given year is a leap year or not:

Pseudocode

```
BEGIN
DEFINE: Integer year
DISPLAY: "Enter the year value: "
READ: year
IF: year%4=0
    DISPLAY: "'year' is a leap year"
ELSE
    DISPLAY: "'year' is not a leap year"
END IF
END
```

Example 1.14 Write the pseudocode for the problem statement given in Example 1.4.

Solution

Pseudocode to find out whether a given number is even or odd:

Pseudocode

```
BEGIN
DEFINE: Integer num
DISPLAY: "Enter a number: "
READ: num
IF: num%2=0
    DISPLAY: "'num' is an even number"
ELSE
    DISPLAY: "'num' is an odd number"
END IF
END
```

Example 1.15 Write the pseudocode for the problem statement given in Example 1.5.

Solution

Pseudocode to determine whether a given string is a palindrome or not:

1.40 Computer Programming

Pseudocode

```
BEGIN
DEFINE: String str
DEFINE: Character chk
DEFINE: Integer left, right, len
SET: chk = 't'
DISPLAY: "Enter a string: "
READ: str
COMPUTE: len = strlen(str)
SET: left = 0
SET: right = len-1
REPEAT
    IF: str(left)=str(right)
        CONTINUE
    ELSE
        SET: chk = 'f'
    END IF
    COMPUTE: left = left + 1
    COMPUTE: right = right - 1
UNTIL: left<right AND chk='t'
IF: chk='t'
    DISPLAY: "'str' is a palindrome string"
ELSE
    DISPLAY: "'str' is not a palindrome string"
END IF
END
```

1.12 PROBLEM SOLVING EXAMPLES

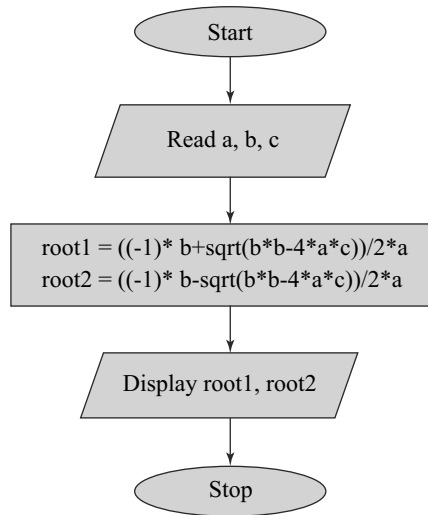
Example 1.16 Design and develop the solution for finding the roots of a quadratic equation.

Solution

Algorithm

```
Step 1 - Start
Step 2 - Accept three numbers (a, b, c) from the user for the quadratic equation  $ax^2 + bx + c$ 
Step 3 - Calculate  $root1 = ((-1)*b + \sqrt{b*b - 4*a*c}) / 2*a$ 
Step 4 - Calculate  $root2 = ((-1)*b - \sqrt{b*b - 4*a*c}) / 2*a$ 
Step 5 - Display the computed roots of the quadratic equation
Step 6 - Stop
```

Flowchart



Pseudocode

```

BEGIN
DEFINE: Integer a, b, c
DEFINE: Real root1, root2
DISPLAY: "Enter the values of a, b and c for the quadratic equation ax2 + bx + c: "
READ: a, b, c
COMPUTE: root1 = ((-1)*b + sqrt(b*b - 4*a*c))/2*a
COMPUTE: root2 = ((-1)*b - sqrt(b*b - 4*a*c))/2*a
DISPLAY: "The roots of the quadratic equation are 'root1' and 'root2'"
END
  
```

Program

```

#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
    int a,b,c;
    float root1,root2; /*Declaration of variables to store root values*/
    clrscr();
    printf("\nEnter the value a, b and c for the quadratic equation axx+bx+c: ");
  
```

1.42 Computer Programming

```
scanf("%d %d %d",&a, &b, &c); /*Reading values of a, b and c*/
root1=(-1)*b+sqrt(b*b-4*a*c))/2*a; /*Computing quadratic root1*/
root2=(-1)*b-sqrt(b*b-4*a*c))/2*a; /*Computing quadratic root2*/
printf("\n\nThe roots of the quadratic equation are: %.2f %.2f",root1, root2); /*Dis-
playing resultant roots*/
getch();
}
```

Output

```
Enter the value a, b and c for the quadratic equation axx+bx+c: 1
-7
12

The roots of the quadratic equation are: 4.00 3.00
```

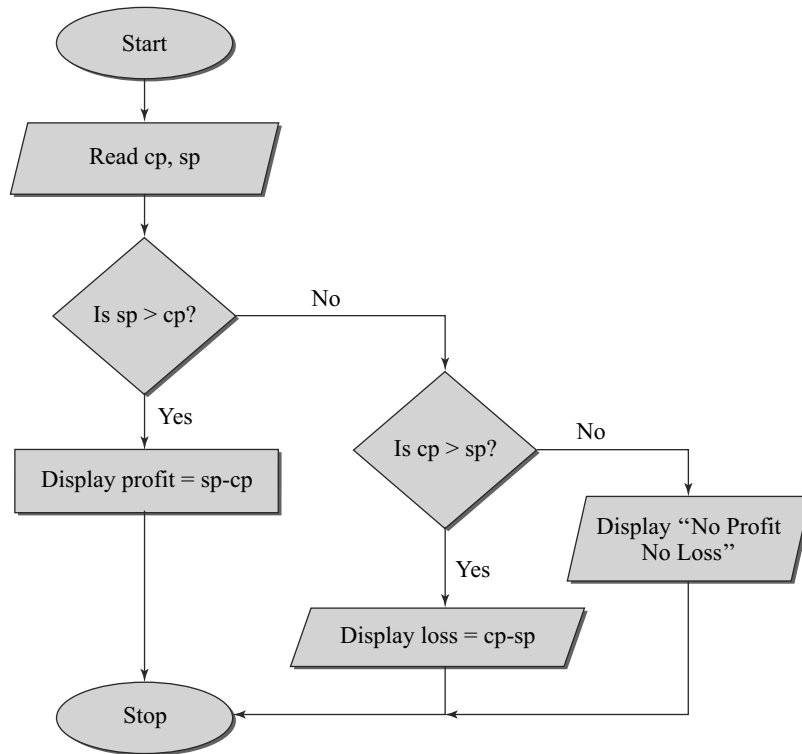
Example 1.17 Design and develop the solution for determining whether there is profit or loss during the selling of an item.

Solution

Algorithm

```
Step 1 - Start
Step 2 - Accept the cost price and selling price of an item from the user (cp, sp)
Step 3 - If sp>cp then goto step 4 else goto step 5
Step 4 - Display "There is a profit of (sp-cp)" and goto Step 8
Step 5 - If cp>sp then goto step 6 else goto step 7
Step 6 - Display "There is a loss of (cp-sp)"
Step 7 - Display "No profit no loss!"
Step 8 - Stop
```

Flowchart



Pseudocode

```

BEGIN
DEFINE: Long Integer cp, sp
DISPLAY: "Enter the cost price and selling price of an item: "
READ: cp, sp
IF: sp>cp
    DISPLAY: "There is a profit of 'sp-cp'"
ELSE
    IF: cp>sp
        DISPLAY: "There is a loss of 'cp-sp'"
    ELSE
        DISPLAY: "No profit no loss!"
    END IF
END IF
END IF
END
  
```

1.44 Computer Programming

Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
    long cp, sp;
    clrscr();
    printf("\nEnter the cost price and selling price of an item: ");
    scanf("%ld %ld",&cp, &sp); /*Reading CP and SP*/
    if(sp>cp) /*Checking condition of profit*/
        printf("\n\nThere is a profit of %ld units", sp-cp);
    else if(sp<cp) /*Checking condition of loss*/
        printf("\n\nThere is a loss of %ld units", cp-sp);
    else
        printf("\n\nNo Profit No Loss!");
    getch();
}
```

Output

Enter the cost price and selling price of an item: 24

30

There is a profit of 6 units

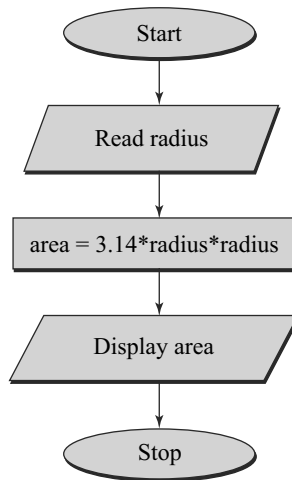
Example 1.18 Design and develop the solution for finding the area of a circle.

Solution

Algorithm

```
Step 1 - Start
Step 2 - Accept the radius of the circle from the user (radius)
Step 3 - Calculate area of the circle using formula area = 3.14 * radius * radius
Step 4 - Display the computed area of the circle
Step 5 - Stop
```


Flowchart



Pseudocode

```
BEGIN
DEFINE: Real radius, area
DISPLAY: "Enter the radius of the circle: "
READ: radius
COMPUTE: area = 3.14*radius*radius
DISPLAY: "The area of the circle is 'area'"
END
```

Program

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

void main()
{
    float radius, area;
    clrscr();
    printf("\nEnter the radius of the circle: ");

    scanf("%f",&radius); /*Reading value of radius*/
    area=3.14*radius*radius; /*Computing area of the circle*/
```

```
printf("\n\nThe area of the circle having radius of %.2f is %.2f",radius, area);  
/*Displaying result*/  
getch();  
}
```

Output

Enter the radius of the circle: 4

The area of the circle having radius of 4.00 is 50.24



Just Remember

- All the devices that expand the capabilities of a computer in some way are termed as peripheral devices. Examples: printer, plotter, disk drive, speaker, microphone, etc.
- Cache memory has the fastest access time followed by RAM, and secondary storage devices.
- High-level language is the most user-friendly, followed by assembly and machine language.
- Machine-level language is the most efficient followed by assembly and high-level language.
- It is always advisable to first create the algorithm and then the actual program. It helps to develop the code in a more systematic manner which is less error-prone.
- It is always advisable to test a program in varied test scenarios so as to ensure that there is no prevalence of any semantic error.
- Describing the process step-by-step is called as **algorithm**.
- Representing the various steps in the form of a diagram is called as **flow chart**.
- **Parallel computing** was developed to speed up things and ultimately try to imitate human actions in **artificial intelligence**.
- Sequence involves a series of steps that we perform one after the other.
- Selection involves making a choice from multiple available options.
- Iteration involves performing repetitive tasks.
- The above three basic categories of activities combined in different ways can form the basis for describing *any* algorithm.
- Programs without go to statements are easy to understand and therefore, easy to maintain.
- We can combine two or more conditions into a single **compound condition**.
- Putting an *if* within another *if* forms a **nested condition**.
- It is always a good practice to do a walkthrough of an algorithm with a variety of values.
- The process of removing a bug from an algorithm is called as **debugging**.

**Multiple Choice Questions**

1. Which of the following is used to perform computations on the entered data?
(a) Memory (b) Processor
(c) Input device (d) Output device
2. Which of the following is not an input device?
(a) Plotter (b) Scanner
(c) Keyboard (d) Mouse
3. Which of the following is not an output device?
(a) Plotter (b) Scanner
(c) Printer (d) Speaker
4. Which of the following is used as a primary memory of the computer?
(a) Magnetic storage device
(b) RAM
(c) Optical storage device
(d) Magneto-optical storage device
5. Which of the following is used as a secondary memory of the computer?
(a) Magnetic storage device
(b) RAM
(c) Cache memory
(d) ROM
6. Which of the following is defined as a computer program for performing a particular task on the computer system?
(a) Hardware (b) Software
(c) Processor (d) Memory
7. Which type of operating system allows more than one program to run at the same time in a computer system?
(a) Multi-threading operating system
(b) Interactive operating system
(c) Multi-user operating system
(d) Multi-tasking operating system
8. Which one of the following converts assembly language into machine language?
(a) Interpreter (b) Compiler
(c) Assembler (d) Algorithm
9. In which of the following languages, the instructions are written in the form of 0s and 1s?
(a) Assembly language
(b) Programming language
(c) High-level language
(d) Machine language
10. Which one of the following is known as the 'language of the computer'?
(a) Programming language
(b) High-level language
(c) Machine language
(d) Assembly language
11. What are the three main categories of high-level language?
(a) Low-level languages
(b) Procedure oriented languages
(c) Mechanical languages
(d) Natural languages
(e) Problem oriented languages
12. Which of the following is not a translator program?
(a) Linker (b) Assembler
(c) Compiler (d) Interpreter
13. Any C program
(a) Must contain at least one function
(b) Need not contain any function
(c) Needs input data
(d) None of the above



Review Questions

- 1.1 State whether the following statements are *true* or *false*.
 - (a) The alphanumeric keys are the keys that help perform a specific task such as searching a file or refreshing the Web pages.
 - (b) Dot matrix printers are slower than inkjet printers and are used to generate high quality photographic prints.
- 1.2 Fill in the blanks with appropriate words in the following statement.
 - (a) The _____ keys include the number keys and the alphabet keys.
- 1.3 What are input devices? Briefly explain some popular input devices.
- 1.4 What is the purpose of an output device? Explain various types of output devices.
- 1.5 What is assembly language? What are its main advantages?
- 1.6 What is high-level language? What are the different types of high-level languages?
- 1.7 What do we understand by a compiler and an assembler?
- 1.8 What is a flow chart? How is it different from an algorithm?
- 1.9 What are the functions of a flow chart?
- 1.10 Point down the differences between an algorithm and a flowchart.
- 1.11 Write an algorithm for withdrawing Rs. 1000 from the bank.
- 1.12 Draw a flowchart for the above.
- 1.13 Describe in detail the steps involved in testing.
- 1.14 Discuss sequence, selection and iteration in detail.
- 1.15 State whether the following statements are *true* or *false*.
 - (a) Describing the process step by step is called as flowchart.
 - (b) Algorithm involves very complex process.
 - (c) When we break up a big task into smaller steps, what we actually do is to create an algorithm.
 - (d) Each step in an algorithm can be called as an instruction.
 - (e) Parallel computing slows down the things.
 - (f) In general, the steps in an algorithm can be divided in five basic categories.
 - (g) Making a choice from multiple available options is called as a sequence.
 - (h) Performing repetitive tasks is called as iteration.
 - (i) The positioning of *End-if* can change the meaning in a process.
 - (j) Avoiding a *goto* in a program, makes it a *goto-less* or *top-down* or *structured*.
 - (k) *goto-less* programs are difficult to understand.
 - (l) We can combine two or more conditions into a single compound condition.
 - (m) If there is an *if* within another *if* then it is called as a compound condition.
 - (n) Checking to see if our algorithm gives the desired result is called as a test case.
 - (o) A group of test cases makes up the test data.
 - (p) Sequence, selection and iteration form the building blocks for writing any algorithm.



Key Terms

- **Computer:** It is an electronic device that takes data and instructions as input from the user, processes the data, and generates useful information as an output.
- **Vacuum tube:** It was used in the first generation computers for developing the circuitry. It comprised of glass and filaments.
- **Transistor:** It is a solid state device used in the second generation computers. It replaced vacuum tubes.
- **IC:** It is a silicon chip that embeds an electronic circuit comprising of several components, such as transistors, diodes, and resistors. It is used in third generation computers.
- **Microprocessor:** It is a processor chip used in fourth generation computers. It integrates thousands of components on a single chip.
- **LAN:** It is a network, where multiple computers in a local area, such as home, office, or small group of buildings, are connected and allowed to communicate among them.
- **WAN:** It is a network, which facilitates connection and communication of hundreds of computers located across multiple locations.
- **MAN:** It is a network that is used to connect the computers over a large geographical area, such as district or city.
- **GUI:** It is a user-friendly interface that provides icons and menus to interact with the various computer applications.
- **Microcomputer:** It is defined as a computer that has a microprocessor as its CPU.
- **Minicomputer:** It is a medium-sized computer that is designed to serve multiple users simultaneously.
- **Mainframe computer:** It is a computer, which helps in handling the information processing of various organizations like banks, insurance companies, hospitals and railways.
- **Supercomputer:** It is the most powerful and fastest computer. It is used for complex scientific applications.
- **Input devices:** Input devices accept the data from the end users on which the operations are to be performed.
- **Output devices:** Output devices are used for providing the output of a program that is obtained after performing the operations specified in a program.
- **CPU:** It is the heart of a computer that is used to process the data entered through the input device.
- **Memory:** It is used for storing the input data as well as the output of a program that is obtained after performing the operations in a program.
- **Scanner:** It is an input device that converts documents and images as the digitized images understandable by the computer system.
- **Motherboard:** It is a device used for connecting the CPU with the input and output devices.
- **RAM:** It is the primary memory of a computer that stores information and programs, until the computer is used.
- **Monitor:** It is an output device that produces visual displays generated by the computer.
- **Printer:** It is an output device that prints the computer generated information onto the paper sheets.
- **Speaker:** It is an electromechanical transducer that converts an electrical signal into sound.
- **Plotter:** It is an output device that is connected to a computer to print large documents, such as engineering and constructional drawings.
- **System software:** It refers to a computer program that manages and controls hardware components.
- **Application software:** It is a computer program that is designed and developed for performing specific utility tasks; it is also known as end-user program.
- **Operating System:** It is the system software that helps in managing the resources of a computer system. It also provides a platform

1.50 Computer Programming

for the application programs to run on the computer system.

- **Binary number system:** It is a numeral system that represents numeric values using only two digits, 0 and 1, known as bits.
- **ASCII:** It is a standard alphanumeric code that represents numbers, alphabetic characters, and symbols using a 7-bit format.
- **Logic gates:** These are the basic building blocks of a digital computer having two input signals and one output signal.
- **Assembler:** It is a computer program that translates assembly language statements into machine language codes.
- **Compiler:** It is a computer program that translates the source code written in a high-level language into the corresponding object code of the low-level language.
- **Interpreter:** It is a translation program that converts each high-level program statement into the corresponding machine code.
- **Algorithm:** It is a complete, detailed, and precise step-by-step method for solving a problem independently.
- **Flowchart:** It is a pictorial representation of an algorithm depicting the flow of various steps.

2 Introduction to C

2.1 INTRODUCTION

C is a powerful, portable and elegantly structured programming language. It combines the features of a high-level language with the elements of an assembler and, therefore, it is suitable for writing both system software and application packages. It is the most widely used general-purpose language today. In fact, C has been used for implementing systems such as operating systems, compilers, linkers, word processors and utility packages. Figure 2.1 shows the various features of C that make it so much popular among programmers.

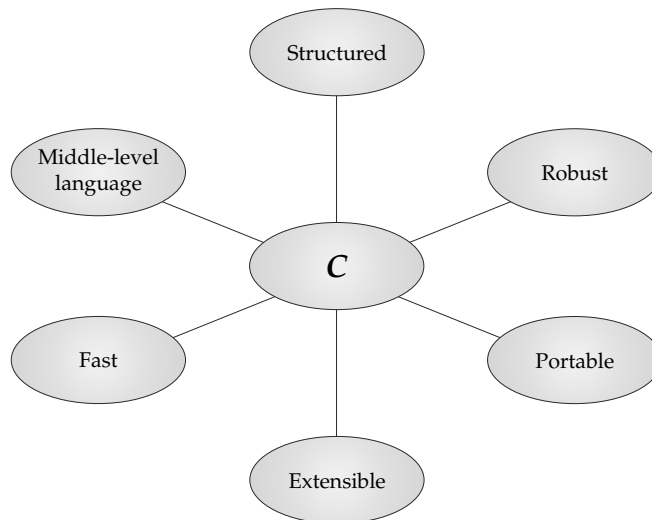


Fig. 2.1 *Features of C*

2.2 Computer Programming

To write programs in C, we need to understand various terms such as characters, identifiers, keywords, constants, variables, data types, programming constructs, etc. All this shall form the basis of this chapter along with actual programs written in C language.

2.2 OVERVIEW OF C

The increasing popularity of C is probably due to its many desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assembly language with the features of a high-level language and, therefore, it is well suited for writing both system software and business packages. In fact, many C compilers available in the market are written in C.

Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators. It is many times faster than BASIC. For example, a program to increment a variable from 0 to 15000 takes about one second in C, while it takes more than 50 seconds in an interpreter of BASIC.

There are only 32 keywords in ANSI C, and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs. C is highly portable. This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

C language is well suited for structured programming, thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would make a complete program. This modular structure makes program debugging, testing and maintenance easier.

Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to C library. With the availability of a large number of functions, the programming task becomes simple.

2.2.1 History of C

‘C’ seems a strange name for a programming language. But this strange-sounding language is one of the most popular computer languages today because it is a structured, high-level, machine-independent language. It allows software developers to develop programs without worrying about the hardware platforms where these will be implemented.

The root of all modern languages is ALGOL, introduced in the early 1960s. ALGOL was the first computer language to use a block structure. Although it never became popular in USA, it was widely used in Europe. ALGOL introduced the concept of structured programming to the computer science community. Computer scientists like Corrado Bohm, Guiseppe Jacopini and Edsger Dijkstra popularised this concept during 1960s. Subsequently, several languages were announced.

In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language) primarily for writing system software. In 1970, Ken Thompson created a language using many features of BCPL and called it simply B. B was used to create early versions of UNIX operating system at Bell Laboratories. Both BCPL and B were ‘typeless’ system programming languages.

C was evolved from ALGOL, BCPL and B by Dennis Ritchie at the Bell Laboratories in 1972. C uses many concepts from these languages and added the concept of data types and other powerful features. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell Laboratories, was coded almost entirely in C. UNIX is one of the most popular network operating systems in use today and the heart of the Internet data superhighway.

For many years, C was used mainly in academic environments, but eventually with the release of many C compilers for commercial use and the increasing popularity of UNIX, it started receiving widespread support among computer professionals. Today, C is running under a variety of operating system and hardware platforms.

During 1970s, C had evolved into what is now known as ‘traditional C’. The language became more popular after publication of the book *The C Programming Language* by Brian Kerningham and Dennis Ritchie in 1978. The book was so popular that the language came to be known as ‘K&R C’ among the programming community. The rapid growth of C led to the development of different versions of the language that were similar but often incompatible. This posed a serious problem for system developers.

To assure that the C language remains standard, in 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of C in December 1989, which is now known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990. This version of C is also referred as C99.

During 1990s, C++, a language entirely based on C, underwent a number of improvements and changes, and became an ANSI/ISO approved language in November 1997. C++ added several new features to C to make it not only a true object-oriented language but also a more versatile language. During the same period, Sun Microsystems of USA created a new language Java modelled on C and C++.

All popular computer languages are dynamic in nature. They continue to improve their power and scope by incorporating new features, and C is no exception. Although C++ and Java were evolved out of C, the standardization committee of C felt that a few features of C++/Java, if added to C, would enhance the usefulness of the language. The result was the 1999 standard for C. This version is usually referred to as C99. The history and development of C is illustrated in Fig. 2.2.

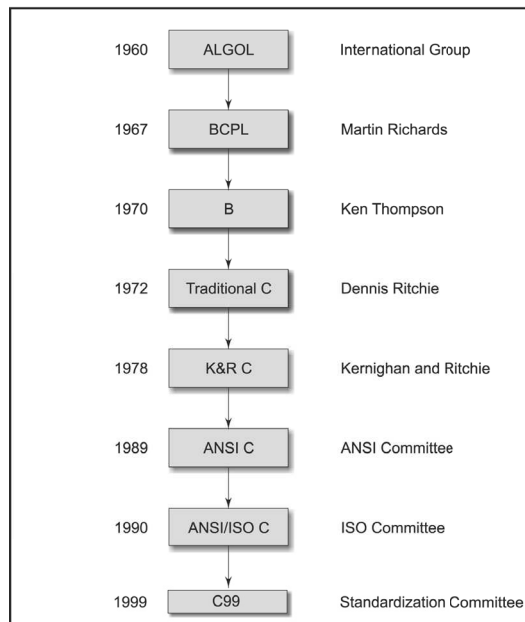


Fig. 2.2 History of ANSI C

2.4 Computer Programming

Although C99 is an improved version, still many commonly available compilers do not support all the new features incorporated in C99.

2.2.2 Characteristics of C

The increasing popularity of C is probably due to its many desirable qualities. Some of the important characteristics are:

- It is a highly structured language.
- It uses features of high-level languages.
- It can handle bit-level operations.
- C is a machine-independent language and, therefore, highly portable.
- It supports a variety of data types and a powerful set of operators.
- It supports dynamic memory management by using the concept of pointers.
- It enables the implementation of hierarchical and modular programming with the help of functions.
- C can extend itself by addition of functions to its library continuously.

Before discussing specific features of C, we shall look at some sample C programs, and analyze and understand how they work

2.2.3 Sample Program 1: Printing a Message

Consider a very simple program given in Fig. 2.3.

```
main( )
{
/*.....printing begins.....*/
    printf("I see, I remember");
/*.....printing ends.....*/
}
```

Fig. 2.3 A program to print one line of text

This program when executed will produce the following output:

```
I see, I remember
```

Let us have a close look at the program. The first line informs the system that the name of the program is main and the execution begins at this line. The main() is a special function used by the C system to tell the computer where the program starts. Every program must have exactly one main function. If we use more than one main function, the compiler cannot understand which one marks the beginning of the program.

The empty pair of parentheses immediately following main indicates that the function main has no arguments (or parameters). The concept of arguments will be discussed in chapter 5.

The opening brace “{” in the second line marks the beginning of the function main and the closing brace “}” in the last line indicates the end of the function. In this case, the closing brace also marks the end of the program. All the statements between these two braces form the function body. The function body contains a set of instructions to perform the given task.

In this case, the function body contains three statements out of which only the printf line is an executable statement. The lines beginning with /* and ending with */ are known as comment lines. These are used in a program to enhance its readability and understanding. Comment lines are not executable statements and

therefore anything between `/*` and `*/` is ignored by the compiler. In general, a comment can be inserted wherever blank spaces can occur—at the beginning, middle or end of a line—“but never in the middle of a word”.

Although comments can appear anywhere, they cannot be nested in C. That means, we cannot have comments inside comments. Once the compiler finds an opening token, it ignores everything until it finds a closing token. Thus the following comment line is not valid and will result in an error.

```
/* = = = = /* = = = = */ = = = = */
```

Since comments do not affect the execution speed and the size of a compiled program, we should use them liberally in our programs. They help the programmers and other users in understanding the various functions and operations of a program and serve as an aid to debugging and testing. We shall see the use of comment lines more in the examples that follow.

Let us now look at the `printf()` function, the only executable statement of the program, as shown under:

```
printf("I see, I remember");
```

`printf` is a predefined standard C function for printing output. Predefined means that it is a function that has already been written and compiled, and linked together with our program at the time of linking. The `printf` function causes everything between the starting and the ending quotation marks to be printed out. In this case, the output will be:

```
I see, I remember
```

Note that the print line ends with a semicolon. Every statement in C should end with a semicolon (;) mark. Suppose we want to print the above quotation in two lines as

```
I see,  
I remember!
```

This can be achieved by adding another `printf` function as shown below:

```
printf("I see, \n");  
printf("I remember !");
```

The information contained between the parentheses is called the argument of the function. This argument of the first `printf` function is “I see, \n” and the second is “I remember !”. These arguments are simply strings of characters to be printed out.

Notice that the argument of the first `printf` contains a combination of two characters `\` and `n` at the end of the string. This combination is collectively called the newline character. A newline character instructs the computer to go to the next (new) line. It is similar in concept to the carriage return key on a typewriter. After printing the character comma (,) the presence of the newline character `\n` causes the string “I remember !” to be printed on the next line. No space is allowed between `\` and `n`.

If we omit the newline character from the first `printf` statement, then the output will again be a single line as shown below:

2.6 Computer Programming

```
I see, I remember !
```

This is similar to the output of the program in Fig. 4.3.

It is also possible to produce two or more lines of output by one `printf` statement with the use of newline character at appropriate places. For example, the statement `printf("I see,\n I remember !");` will generate the following output:

```
I see,  
I remember !
```

However, the statement `printf("I\n.. see,\n... .. I\n... .. remember !");` will print out the following:

```
I  
.. see,  
... .. I  
... .. remember !
```

NOTE: Some authors recommend the inclusion of the statement `#include <stdio.h>` at the beginning of all programs that use any input/output library functions. However, this is not necessary for the functions `printf` and `scanf` which have been defined as a part of the C language.

Before we proceed to discuss further examples, we must note one important point. C does make a distinction between uppercase and lowercase letters. For example, `printf` and `PRINTF` are not the same. In C, everything is written in lowercase letters. However, uppercase letters are used for symbolic names representing constants. We may also use uppercase letters in output strings like "I SEE" and "I REMEMBER"

The above example that printed I see, I remember is one of the simplest programs. Figure 2.4 highlights the general format of such simple programs. All C programs need a main function.

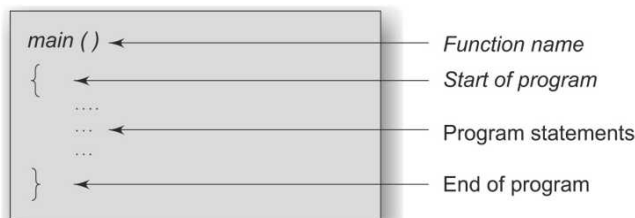


Fig. 2.4 Format of simple C programs

The main Function

The main is a part of every C program. C permits different forms of main statement. Following forms are allowed.

- main()
- int main()
- void main()
- main(void)
- void main(void)
- int main(void)

The empty pair of parentheses indicates that the function has no arguments. This may be explicitly indicated by using the keyword void inside the parentheses. We may also specify the keyword int or void before the word main. The keyword void means that the function does not return any information to the operating system and int means that the function returns an integer value to the operating system. When int is specified, the last statement in the program must be “return 0”. For the sake of simplicity, we use the first form in our programs.

2.2.4 Sample Program 2: Adding Two Numbers

Consider another program, which performs addition on two numbers and displays the result. The complete program is shown in Fig. 2.5:

```

/* Program ADDITION           line-1 */
/* Written by EBG             line-2 */
main()                        /* line-3 */
{                             /* line-4 */
    int number;               /* line-5 */
    float amount;             /* line-6 */
                                /* line-7 */
    number = 100;             /* line-8 */
                                /* line-9 */
    amount = 30.75 + 75.35; /* line-10 */
    printf(“%d\n”,number);    /* line-11 */
    printf(“%5.2f”,amount);    /* line-12 */
                                /* line-13 */
}

```

Fig. 2.5 Program to add two numbers

This program when executed will produce the following output:

```

100
106.10

```

The first two lines of the program are comment lines. It is a good practice to use comment lines in the beginning to give information such as name of the program, author, date, etc. Comment characters are also used in other lines to indicate line numbers.

The words **number** and **amount** are variable names that are used to store numeric data. The numeric data may be either in integer form or in real form. In C, all variables should be declared to tell the compiler what the variable names are and what type of data they hold.

2.8 Computer Programming

The variables must be declared before they are used. In lines 5 and 6, the declarations **int number**; and **float amount**; tell the compiler that **number** is an integer (int) and **amount** is a floating (float) point number. Declaration statements must appear at the beginning of the functions, as shown in Fig. 4.5. All declaration statements end with a semicolon.

The words such as **int** and **float** are called the keywords and cannot be used as variable names. Data is stored in a variable by assigning a data value to it. This is done in lines 8 and 10. In line 8, an integer value 100 is assigned to the integer variable **number** and in line 10, the result of addition of two real numbers 30.75 and 75.35 is assigned to the floating point variable **amount**. Thus, the following statements are called the assignment statements:

```
number = 100;
amount = 30.75 + 75.35;
```

Every assignment statement must have a semicolon at the end.

The next statement is an output statement that prints the value of **number**, as shown below:

```
printf("%d\n", number);
```

The above print statement contains two arguments. The first argument “%d” tells the compiler that the value of the second argument **number** should be printed as a decimal integer. Note that these arguments are separated by a comma. The newline character `\n` causes the next output to appear on a new line.

The last statement of the program i.e. `printf("%5.2f", amount);` prints out the value of **amount** in floating point format. The format specification **%5.2f** tells the compiler that the output must be in floating point, with five places in all and two places to the right of the decimal point.

2.2.5 Sample Program 3: Interest Calculation

The program in Fig. 2.6 calculates the value of money at the end of each year of investment, assuming an interest rate of 11 percent and prints the year, and the corresponding amount, in two columns. The output is shown in Fig. 2.7 for a period of 10 years with an initial investment of 5000.00. The program uses the following formula:

$$\text{Value at the end of year} = \text{Value at start of year} (1 + \text{interest rate})$$

In the program, the variable **value** represents the value of money at the end of the year while **amount** represents the value of money at the start of the year. The statement `amount = value;` makes the value at the end of the current year as the value at start of the next year.

```
/*————— INVESTMENT PROBLEM —————*/
#define PERIOD 10
#define PRINCIPAL 5000.00
/*————— MAIN PROGRAM BEGINS —————*/
main()
{ /*————— DECLARATION STATEMENTS —————*/
```

```

    int year;
    float amount, value, inrate;
/*----- ASSIGNMENT STATEMENTS -----*/
    amount = PRINCIPAL;
    inrate = 0.11;
    year = 0;
/*----- COMPUTATION STATEMENTS -----*/
/*----- COMPUTATION USING While LOOP -----*/
    while(year <= PERIOD)
    {
        printf("%2d %8.2f\n",year, amount);
        value = amount + inrate * amount;
        year = year + 1;
        amount = value;
    }
/*----- while LOOP ENDS -----*/
}
/*----- PROGRAM ENDS -----*/

```

Fig. 2.6 Program for investment problem

```

0    5000.00
1    5550.00
2    6160.50
3    6838.15
4    7590.35
5    8425.29
6    9352.07
7    10380.00
8    11522.69
9    12790.00
10   14197.11

```

Fig. 2.7 Output of the investment program

Let us consider the new features introduced in this program. The second and third lines begin with `#define` instructions. A **#define** instruction defines value to a symbolic constant for use in the program. Whenever a symbolic name is encountered, the compiler substitutes the value associated with the name automatically. To change the value, we have to simply change the definition. In this example, we have defined two symbolic constants `PERIOD` and `PRINCIPAL` and assigned values 10 and 5000.00 respectively. These values remain constant throughout the execution of the program.

The #define Directive

A `#define` is a preprocessor compiler directive and not a statement. Therefore, `#define` lines should not end with a semicolon. Symbolic constants are generally written in uppercase so that they are easily distinguished from lowercase variable names.

2.10 Computer Programming

#define instructions are usually placed at the beginning before the main() function. Symbolic constants are not declared in declaration section.

We must note that the defined constants are not variables. We may not change their values within the program by using an assignment statement. For example, the following statement is illegal:

```
PRINCIPAL = 10000.00;
```

The declaration section declares year as integer and amount, value and inrate as floating point variables. Note all the floating point variables are declared in one statement. They can also be declared as below:

```
float amount;  
float value;  
float inrate;
```

When two or more variables are declared in one statement, they are separated by a comma.

All computations and printing are accomplished in a while loop. while is a mechanism for evaluating repeatedly a statement or a group of statements. In this case as long as the value of year is less than or equal to the value of PERIOD, the four statements that follow while are executed. Note that these four statements are grouped by braces. We exit the loop when year becomes greater than PERIOD.

C supports the basic four arithmetic operators (−, +, *, /) along with several others.

2.2.6 Sample Program 4: Use of Subroutines

So far, we have used only printf function that has been provided for us by the C system. The program shown in Fig. 2.8 uses a user-defined function. A function defined by the user is equivalent to a subroutine in FORTRAN or subprogram in BASIC. Figure 2.8 presents a very simple program that uses a mul () function.

```
Program  
/*----- PROGRAM USING FUNCTION -----*/  
int mul (int a, int b); /*--- DECLARATION ---*/  
/*----- MAIN PROGRAM BEGINS -----*/  
    main ()  
    {  
        int a, b, c;  
        a = 5;  
        b = 10;  
        c = mul (a,b);  
        printf ("multiplication of %d and %d is %d",a,b,c);  
    }  
/*----- MAIN PROGRAM ENDS  
    MUL() FUNCTION STARTS -----*/
```



```

int mul (int x, int y)
int p;
{
    p = x*y;
    return(p);
}

/* ----- MUL ( ) FUNCTION ENDS ----- */

```

Fig. 2.8 A program using a user-defined function

In the above program, the **mul ()** function multiplies the values of x and y and the result is returned to the **main ()** function when it is called in the following statement:

```
c = mul (a, b);
```

The **mul ()** has two arguments x and y that are declared as integers. The values of a and b are passed on to x and y respectively when the function **mul ()** is called.

2.2.7 Sample Program 5: Use of Math functions

We often use standard mathematical functions such as **cos**, **sin**, **exp**, etc. We shall see now the use of a mathematical function in a program. The standard mathematical functions are defined and kept as a part of C math library. If we want to use any of these mathematical functions, we must add an **#include** instruction in the program. Like **#define**, it is also a compiler directive that instructs the compiler to link the specified mathematical functions from the library. The instruction takes the following form:

```
#include <math.h>
```

Here, **math.h** is the filename containing the required function. Fig. 2.9 illustrates the use of cosine function. The program calculates cosine values for angles 0, 10, 20.....180 and prints out the results with headings.

```

Program
/*----- PROGRAM USING COSINE FUNCTION ----- */
#include <math.h>
#define PI 3.1416
#define MAX 180
main ( )
{
    int angle;
    float x,y;
    angle = 0;
    printf("  Angle      Cos(angle)\n\n");
    while(angle <= MAX)

```

2.12 Computer Programming

```
{  
    x = (PI/MAX)*angle;  
    y = cos(x);  
    printf("%15d %13.4f\n", angle, y);  
    angle = angle + 10;  
}
```

Output

Angle	Cos(angle)
0	1.0000
10	0.9848
20	0.9397
30	0.8660
40	0.7660
50	0.6428
60	0.5000
70	0.3420
80	0.1736
90	-0.0000
100	-0.1737
110	-0.3420
120	-0.5000
130	-0.6428
140	-0.7660
150	-0.8660
160	-0.9397
170	-0.9848
180	-1.0000

Fig. 2.9 Program using a math function

The #include Directive

As mentioned earlier, C programs are divided into modules or functions. Some functions are written by users, like us, and many others are stored in the C library. Library functions are grouped category-wise and stored in different files known as header files. If we want to access the functions stored in the library, it is necessary to tell the compiler about the files to be accessed.

This is achieved by using the preprocessor directive `#include` as follows:

```
#include <filename>
```

Here, filename is the name of the library file that contains the required function definition. Preprocessor directives are placed at the beginning of a program.

2.3 BASIC STRUCTURE OF C PROGRAMS

The examples discussed so far illustrate that a C program can be viewed as a group of building blocks called functions. A function is a subroutine that may include one or more statements designed to perform a specific task. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in Fig. 2.10.

The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later. The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants.

There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the user-defined functions.

Every C program must have one `main()` function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon(`;`).

The subprogram section contains all the user-defined functions that are called in the main function. User-defined functions are generally placed immediately after the main function, although they may appear in any order.

All sections, except the main function section may be absent when they are not required.

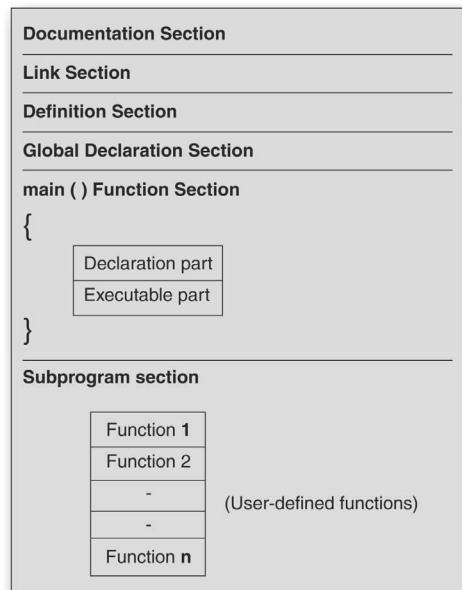


Fig. 2.10 Typical structure of a C program

2.4 PROGRAMMING STYLE

Unlike some other programming languages (COBOL, FORTRAN, etc.,) C is a free-form language. That is, the C compiler does not care, where on the line we begin typing. While this may be a licence for bad programming, we should try to use this fact to our advantage in developing readable programs. Although several alternative styles are possible, we should select one style and use it with total consistency.

First of all, we must develop the habit of writing programs in lowercase letters. C program statements are written in lowercase letters. Uppercase letters are used only for symbolic constants.

Braces, group program statements together and mark the beginning and the end of functions. A proper indentation of braces and statements would make a program easier to read and debug.

Since C is a free-form language, we can group statements together on one line. For example, consider the following statements

2.14 Computer Programming

```
a = b;  
x = y + 1;  
z = a + x;
```

The above statements can be written in one line as under:

```
a= b; x = y+1; z = a+x;
```

Similarly, consider the following program:

```
main( )  
{  
    printf("hello C");  
}
```

It can also be written in one line as under:

```
main( ) {printf("Hello C");}
```

However, this style makes the program more difficult to understand and should not be used.

2.5 EXECUTING A 'C' PROGRAM

Executing a program written in C involves a series of steps. These are:

1. Creating the program;
2. Compiling the program;
3. Linking the program with functions that are needed from the C library; and
4. Executing the program.

Figure 2.11 illustrates the process of creating, compiling and executing a C program. Although these steps remain the same irrespective of the operating system, system commands for implementing the steps and conventions for naming files may differ on different systems.

An operating system is a program that controls the entire operation of a computer system. All input/output operations are channeled through the operating system. The operating system, which is an interface between the hardware and the user, handles the execution of user programs.

The two most popular operating systems today are UNIX (for minicomputers) and MS-DOS (for microcomputers). We shall discuss briefly the procedure to be followed in executing C programs under both these operating systems in the following sections.

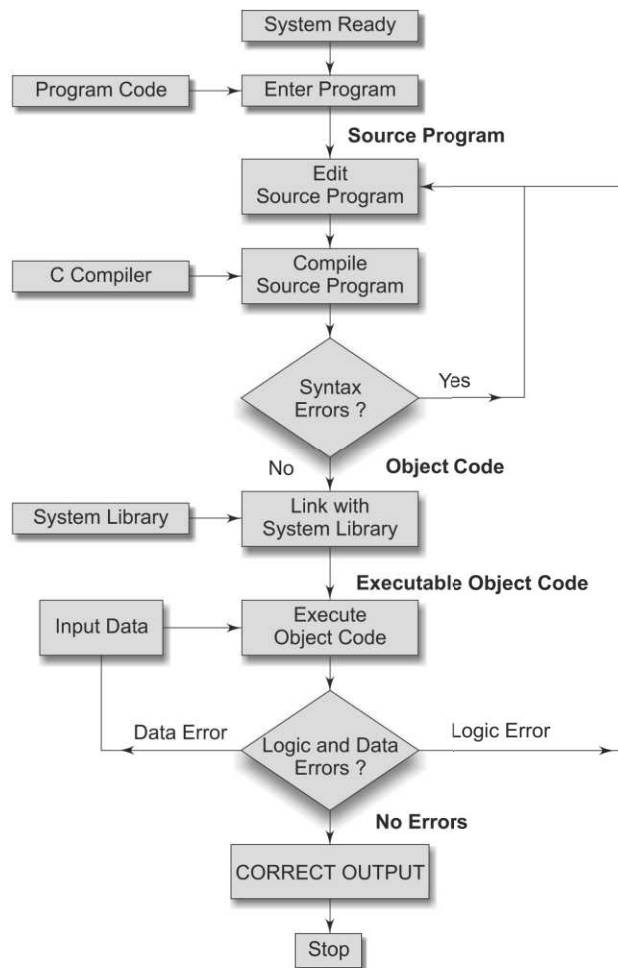


Fig. 2.11 Process of compiling and running a C program

2.6 UNIX SYSTEM

Creating the program Once we load the UNIX operating system into the memory, the computer is ready to receive program. The program must be entered into a file. The file name can consist of letters, digits and special characters, followed by a dot and a letter c. Examples of valid file names are:

```
hello.c
program.c
ebg1.c
```

The file is created with the help of a text editor, either ed or vi. The command for calling the editor and creating the file is

2.16 Computer Programming

```
ed filename
```

If the file existed before, it is loaded. If it does not yet exist, the file has to be created so that it is ready to receive the new program. Any corrections in the program are done under the editor.

When the editing is over, the file is saved on disk. It can then be referenced any time later by its file name. The program that is entered into the file is known as the source program, since it represents the original form of the program.

Compiling and Linking Let us assume that the source program has been created in a file named `ebg1.c`. Now the program is ready for compilation. The compilation command to achieve this task under UNIX is

```
cc ebg1.c
```

The source program instructions are now translated into a form that is suitable for execution by the computer. The translation is done after examining each instruction for its correctness. If everything is alright, the compilation proceeds silently and the translated program is stored on another file with the name `ebg1.o`. This program is known as object code.

Linking is the process of putting together other program files and functions that are required by the program. For example, if the program is using `exp()` function, then the object code of this function should be brought from the math library of the system and linked to the main program. Under UNIX, the linking is automatically done (if no errors are detected) when the `cc` command is used.

If any mistakes in the syntax and semantics of the language are discovered, they are listed out and the compilation process ends right there. The errors should be corrected in the source program with the help of the editor and the compilation is done again.

The compiled and linked program is called the executable object code and is stored automatically in another file named `a.out`.

Executing the Program Execution is a simple task. The command would load the executable object code into the computer memory and execute the instructions:

```
a.out
```

During execution, the program may request for some data to be entered through the keyboard. Sometimes the program does not produce the desired results. Perhaps, something is wrong with the program logic or data. Then it would be necessary to correct the source program or the data. In case the source program is modified, the entire process of compiling, linking and executing the program should be repeated.

Creating Our Own Executable File Note that the linker always assigns the same name `a.out`. When we compile another program, this file will be overwritten by the executable object code of the new program. If we want to prevent this from happening, we should rename the file immediately by using the following command:

```
mv a.out name
```

We may also achieve this by specifying an option in the `cc` command as follows:

```
cc -o name source-file
```

This will store the executable object code in the file name and prevent the old file a.out from being destroyed.

Multiple Source Files To compile and link multiple source program files, we must append all the files names to the cc command, as shown below:

```
cc filename-1.c ... filename-n.c
```

These files will be separately compiled into object files called filename-i.o and then linked to produce an executable program file a.out as shown in Fig. 2.12.

It is also possible to compile each file separately and link them later. For example, the following commands will compile the source files mod1.c and mod2.c into objects files mod1.o and mod2.o:

```
cc -c mod1.c
cc -c mod2.c
```

They can be linked together by the following command:

```
cc mod1.o mod2.o
```

We may also combine the source files and object files as follows:

```
cc mod1.c mod2.o
```

Only mod1.c is compiled and then linked with the object file mod2.o. This approach is useful when one of the multiple source files need to be changed and recompiled or an already existing object files is to be used along with the program to be compiled.

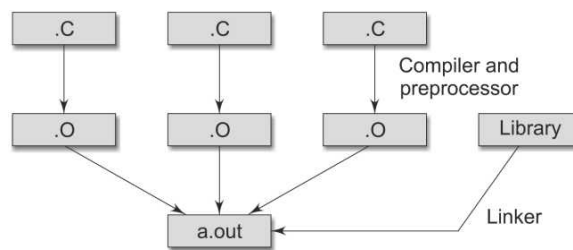


Fig. 2.12 Compilation of multiple files

2.7 C CHARACTER SET

A programming language is designed to help process certain kinds of data consisting of numbers, characters and strings and to provide useful output known as information. The task of processing of data is accomplished

2.18 Computer Programming

by executing a sequence of precise instructions called a program. These instructions are formed using certain symbols and words according to some rigid rules known as syntax rules (or grammar). Every program instruction must conform precisely to the syntax rules of the language.

In C, the characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. However, a subset of characters is available that can be used on most personal, micro, mini and mainframe computers. The characters in C are grouped into the following categories:

1. Letters
2. Digits
3. Special characters
4. White spaces

The entire character set is given in Table 2.1.

TABLE 2.1 C Character Set

Characters	Notations
Letters	<ul style="list-style-type: none">• Uppercase A.....Z• Lowercase a.....z
Digits	<ul style="list-style-type: none">• All decimal digits 09
Special characters	<ul style="list-style-type: none">• , comma• & ampersand• . period• ^ caret• ; semicolon• * asterisk• : colon• – minus sign• ? question mark• + plus sign• ‘ apostrophe• < opening angle bracket (or less than sign)• “ quotation mark• # number sign• ! exclamation mark• > closing angle bracket (or greater than sign)• vertical bar• / slash• (left parenthesis• \ backslash

White Spaces

-) right parenthesis
- ~ tilde
- [left bracket
- _ under score
-] right bracket
- \$ dollar sign
- { left brace
- % percent sign
- } right brace
- Blank space
- Horizontal tab
- Carriage return
- New line
- Form feed

The compiler ignores white spaces unless they are a part of a string constant. White spaces may be used to separate words, but are prohibited between the characters of keywords and identifiers.

2.7.1 Trigraph Characters

Many non-English keyboards do not support all the characters mentioned in Table 2.1. ANSI C introduces the concept of 'trigraph' sequences to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters (two question marks followed by another character) as shown in Table 2.2. For example, if a keyboard does not support square brackets, we can still use them in a program using the trigraph sequence ??(or ??).

TABLE 2.2 ANSI C Trigraph Sequences

Trigraph sequence	Translation
??=	# number sign
??([left bracket
??)] right bracket
??<	{ left brace
??>	} right brace
??!	vertical bar
??/	\ back slash
??-	~ tilde

2.8 C TOKENS

In a passage of text, individual words and punctuation marks are called tokens. Similarly, in a C program the smallest individual units are known as C tokens. C has six types of tokens as shown in Fig. 2.13. C programs are written using these tokens and the syntax of the language.

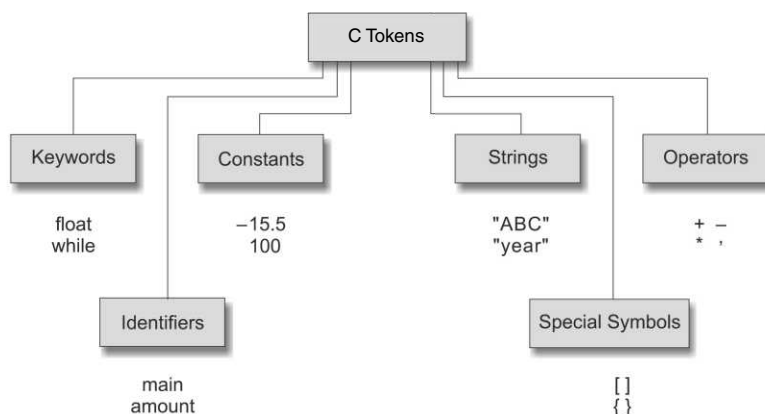


Fig. 2.13 C tokens and examples

2.9 KEYWORDS AND IDENTIFIERS

Every C word is classified as either a keyword or an identifier. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as the basic building blocks for program statements. The list of all keywords of ANSI C is shown in Table 2.3. All keywords must be written in lowercase. Some compilers may use additional keywords that must be identified from the C manual.

TABLE 2.3 ANSI C Keywords

Auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist a sequence of letters and digits, with a letter as the first character. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers. It is usually used as a link between two words in long identifiers. Rules for identifiers are as follows:

- The first character must be an alphabet (or underscore).
- It must consist only letters, digits or underscore.
- Only first 31 characters are significant in an identifier.
- It cannot be same as a keyword.
- It must not contain white space.

2.10 CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program. C supports several constants as illustrated in Fig. 2.14.

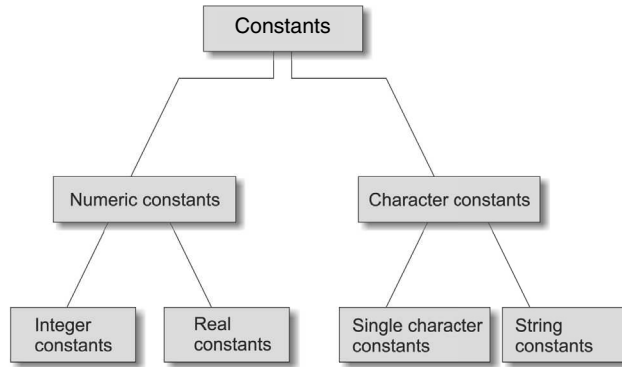


Fig. 2.14 Basic types of C constants

2.10.1 Integer Constants

An integer constant refers to a sequence of digits. There are three types of integers, namely, decimal integer, octal integer and hexadecimal integer. Decimal integers consist a set of digits, 0 through 9, preceded by an optional – or + sign. Examples of some valid decimal integer constants are:

123 –321 0 654321 +78

Embedded spaces, commas and nondigit characters are not permitted between digits. Examples of some invalid decimal integer constants are:

15 750 20,000 \$1000

An octal integer constant consists any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are:

037 0 0435 0551

A sequence of digits preceded by 0x or 0X is considered as a hexadecimal integer. They may also include alphabets A through F or a through f. The letters A through F represent the numbers 10 through 15. Following are the examples of valid hexa decimal integers:

0X2 0x9F 0Xbcd 0x

The largest integer value that can be stored is machine-dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. It is also possible to store larger integer constants on these machines by appending qualifiers such as U, L and UL to the constants. For example:

56789U	or 56789u	(unsigned integer)
987612347UL	or 98761234ul	(unsigned long integer)
9876543L	or 9876543l	(long integer)

Example 2.1 Representation of integer constants on a 16-bit computer.

The program in Fig. 2.15 illustrates the use of integer constants on a 16-bit machine. The output shows that the integer values larger than 32767 are not properly stored on a 16-bit machine. However, when they are qualified as long integer (by appending L), the values are correctly stored.

```

Program
    main()
    {
        printf("Integer values\n\n");
        printf("%d %d %d\n", 32767, 32767+1, 32767+10);
        printf("\n");
        printf("Long integer values\n\n");
        printf("%ld %ld %ld\n", 32767L, 32767L+1L, 32767L+10L);
    }
Output
Integer values
32767 -32768 -32759
Long integer values
32767 32768 32777

```

Fig. 2.15 Representation of integer constants on a 16-bit machine

2.10.2 Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called real (or floating point) constants. Some examples of real constants are:

0.0083 -0.75 435.36 +247.0

These numbers are shown in decimal notation, having a whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point, or digits after the decimal point. For example, the following are all valid real numbers.

215. .95 -.71 +.5

A real number may also be expressed in exponential (or scientific) notation. For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2 . The general form is:

mantissa e exponent

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign. The letter e separating the mantissa and the exponent can be written in either lowercase or uppercase. Since the exponent causes the decimal point to 'float', this notation is said to represent a real number in floating point form. Examples of legal floating-point constants are:

0.65e4 12e-2 1.5e+5 3.18E3 -1.2E-1

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, -0.000000368 is equivalent to -3.68E-7.

Floating-point constants are normally represented as double-precision quantities. However, the suffix `f` or `F` may be used to force single-precision and `l` or `L` to extend double precision further. Some examples of valid and invalid numeric constants are given in Table 2.4.

TABLE 2.4 Examples of numeric constants

Constant	Valid ?	Remarks
698354L	Yes	Represents long integer
25,000	No	Comma is not allowed
+5.0E3	Yes	(ANSI C supports unary plus)
3.5e-5	Yes	
7.1e 4	No	No white space is permitted
-4.5e-2	Yes	
1.5E+2.5	No	Exponent must be an integer
\$255	No	\$ symbol is not permitted
0X7B	Yes	Hexadecimal integer

2.10.3 Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of single quote marks. Example of character constants are:

`'5'` `'X'` `' '` `' '`

Note that the character constant `'5'` is not the same as the number 5. The last constant is a blank space. Character constants have integer values known as ASCII values. For example, the following statement would print the number 97, the ASCII value of the letter `a`.

```
printf("%d", 'a');
```

Similarly, the following statement would output the letter `'a'`.

```
printf("%c", '97');
```

NOTE: Since each character constant represents an integer value, it is also possible to perform arithmetic operations on character constants.

Backslash Character Constants C supports some special backslash character constants that are used in output functions. For example, the symbol `'\n'` stands for newline character. A list of such backslash character constants is given in Table 2.5. Note that each one of them represents one character, although they consist two characters. These character combinations are known as escape sequences.

TABLE 2.5 Backslash character constants

Constant	Meaning
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\"'	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

2.10.4 String Constants

A string constant is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, special characters and blank space. Some of the examples of string constants are:

“Hello!” “1987” “WELL DONE” “?...!” “5+3” “X”

It is important to note that a character constant (e.g., 'X') is not equivalent to the single character string constant (e.g., “X”). Further, a single character string constant does not have an equivalent integer value, while a character constant has an integer value. Character strings are often used in programs to make it meaningful.

2.11 VARIABLES

A variable is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution. A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program. Some examples of such names are:

```
Average
height
Total
Counter_1
class_strength
```

As mentioned earlier, variable names may consist letters, digits and the underscore (_) character, subject to the following conditions:

1. They must begin with a letter. Some systems permit underscore as the first character.
2. ANSI standard recognises a length of 31 characters. However, length should not be normally more than eight characters, since only the first eight characters are treated as significant by many compilers.

3. Uppercase and lowercase are significant. That is, the variable Total is not the same as total or TOTAL.
4. It should not be a keyword.
5. White space is not allowed.

Some examples of valid variable names are:

John	Value	T_raise
Delhi	x1	ph_value
mark	sum1	distance

Invalid examples include:

123 (area)
% 25th

Further examples of variable names and their correctness are given in Table 2.6.

TABLE 2.6 Examples of variable names

Variable name	Valid ?	Remark
First_tag	Valid	
char	Not valid	char is a keyword
Price\$	Not valid	Dollar sign is illegal
group one	Not valid	Blank space is not permitted
average_number	Valid	First eight characters are significant
int_type	Valid	Keyword may be part of a name

2.12 DATA TYPES

C language is rich in its data types. Storage representations and machine instructions to handle constants differ from machine to machine. The variety of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

ANSI C supports three classes of data types:

1. Primary (or fundamental) data types
2. Derived data types
3. User-defined data types

All C compilers support five fundamental data types, namely integer (int), character (char), floating point (float), double-precision floating point (double) and void. Many of them also offer extended data types such as long int and long double. Various data types and the terminology used to describe them are given in Fig. 2.16. The range of the basic four types are given in Table 2.7.

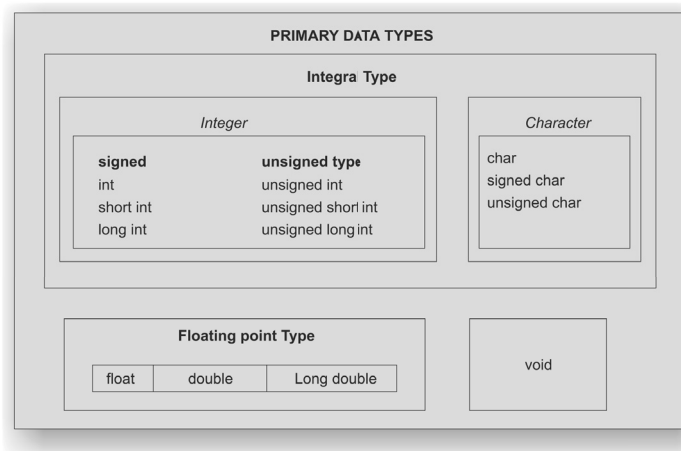


Fig. 2.16 Primary data types in C

TABLE 2.7 Size and range of basic data types on 16-bit machines

Data type	Range of values
char	−128 to 127
int	−32,768 to 32,767
float	3.4e−38 to 3.4e+38
double	1.7e−308 to 1.7e+308

2.12.1 Integer Types

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. If we use a 16-bit word length, the size of the integer value is limited to the range −32768 to +32767 (that is, −215 to +215−1). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32-bit word length can store an integer ranging from −2,147,483,648 to 2,147,483,647.

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely short int, int and long int, in both signed and unsigned forms. ANSI C defines these types so that they can be organised from the smallest to the largest, as shown in Fig. 2.17. For example, short int represents fairly small integer values and requires half the amount of storage as a regular int number uses. Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16-bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

We declare long and unsigned integers to increase the range of values. The use of qualifier signed on integers



Fig. 2.17 Integer types

is optional because the default declaration assumes a signed number. Table 2.8 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

2.12.2 Floating Point Types

Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword `float`. When the accuracy provided by a float number is not sufficient, the type `double` can be used to define the number.

TABLE 2.8 Size and range of data types on a 16-bit machine

Type	Size (bits)	Range
char or signed char	8	−128 to 127
unsigned char	8	0 to 255
int or signed int	16	−32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	−128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	−2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
Float	32	3.4E − 38 to 3.4E + 38
double	64	1.7E − 308 to 1.7E + 308
long double	80	3.4E − 4932 to 1.1E + 4932

A double data type number uses 64 bits giving a precision of 14 digits. These are known as double precision numbers. Remember that double type represents the same data type that float represents, but with a greater precision. To extend the precision further, we may use long double which uses 80 bits. The relationship among floating types is illustrated in Fig. 2.18.

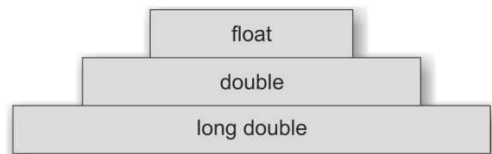


Fig. 2.18 Floating-point types

2.12.3 Void Types

The void type has no values. This is usually used to specify the type of functions. The type of a function is said to be void when it does not return any value to the calling function. It can also play the role of a generic type, meaning that it can represent any of the other standard types.

2.12.4 Character Types

A single character can be defined as a character (`char`) type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier signed or unsigned may be explicitly applied to `char`. While unsigned chars have values between 0 and 255, signed chars have values from −128 to 127.

2.13 DECLARATION OF VARIABLES

After designing suitable variable names, we must declare them to the compiler. Variable declaration basically does two things:

2.28 Computer Programming

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

The declaration of variables must be done before they are used in the program.

2.13.1 Primary Type Declaration

A variable can be used to store a value of any data type. That is, the name has nothing to do with its type. The syntax for declaring a variable is as follows:

```
data-type v1,v2,...vn ;
```

v_1, v_2, \dots, v_n are the names of variables. Variables are separated by commas. A declaration statement must end with a semicolon. For example, the following represent valid variable declarations:

```
int count;
int number, total;
double ratio;
```

`int` and `double` are the keywords to represent integer type and real type data values, respectively. Table 2.9 shows various data types and their keyword equivalents.

TABLE 2.9 Data types and their keywords

Data type	Keyword equivalent
Character	char
Unsigned character	unsigned char
Signed character	signed char
Signed integer	signed int (or int)
Signed short integer	signed short int (or short int or short)
Signed long integer	signed long int (or long int or long)
Unsigned integer	unsigned int (or unsigned)
Unsigned short integer	unsigned short int (or unsigned short)
Unsigned long integer	unsigned long int (or unsigned long)
Floating point	float
Double-precision floating point	double
Extended double-precision floating point	long double

The program given in Fig. 2.19 illustrates declaration of variables.

`main()` is the beginning of the program. The opening brace `{` signals the execution of the program. Declaration of variables is usually done immediately after the opening brace of the program. The variables can also be declared outside (either before or after) the main function.

```

main() /*.....Program Name..... */
{
    /*.....Declaration.....*/
    float    x, y;
    int      code;
    short int    count;
    long int    amount;
    double     deviation;
    unsigned    n;
    char       c;
    /*.....Computation..... */
    . . . .
    . . . .
    . . . .
} /*.....Program ends.....*/

```

Fig. 2.19 Declaration of variables

When an adjective (qualifier), short, long, or unsigned, is used without a basic data type specifier, C compilers treat the data type as an int. If we want to declare a character variable as unsigned, then we must do so using both the terms like unsigned char.

2.13.2 User-defined Type Declaration

C supports a feature known as ‘type definition’ that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables. It takes the general form as follows:

```
typedef type identifier;
```

where type refers to an existing data type and ‘identifier’ refers to the ‘new’ name given to the data type. The existing data type may belong to any class of type, including the user-defined ones. Remember that the new type is ‘new’ only in name, but not the data type. typedef cannot create a new type. Some examples of type definition are as follows:

```
typedef int units;
typedef float marks;
```

Here, units symbolises int and marks symbolises float. They can be later used to declare variables as follows:

```
units batch1, batch2;
marks name1[50], name2[50];
```

2.30 Computer Programming

Here `batch1` and `batch2` are declared as **int** variable and `name1[50]` and `name2[50]` are declared as 50 element floating point array variables. The main advantage of `typedef` is that we can create meaningful data type names for increasing the readability of the program.

Another user-defined data type is enumerated data type provided by ANSI standard. It is defined as follows:

```
enum identifier {value1, value2, ... value n};
```

where ‘`identifier`’ is a user-defined enumerated data type, which can be used to declare variables that can have one of the values enclosed within the braces (known as enumeration constants). After this definition, we can declare variables to be of this ‘`new`’ type as follows:

```
enum identifier v1, v2, ... vn;
```

The enumerated variables `v1`, `v2`, ... `vn` can only have one of the values `value1`, `value2`, ... `valuen`. The assignments of the following types are valid:

```
v1 = value3;  
v5 = value1;
```

Following is an example of using enum data type:

```
enum day {Monday, Tuesday, ... Sunday};  
enum day week_st, week_end;
```

```
week_st = Monday;  
week_end = Friday;  
if(week_st == Tuesday)  
    week_end = Saturday;
```

2.14 DECLARATION OF STORAGE CLASS

Variables in C can have not only data type but also storage class that provides information about their location and visibility. The storage class decides the portion of the program within which the variables are recognised. Consider the following example:

```
/* Example of storage classes */  
int m;  
main()  
{  
    int i;  
    float balance;  
    ....  
    ....  
    function1();  
}
```

```

function1()
{
    int i;
    float sum;
    ....
    ....
}

```

The variable *m* which has been declared before the main is called global variable. It can be used in all the functions in the program. It need not be declared in other functions. A global variable is also known as an external variable.

The variables *i*, *balance* and *sum* are called local variables because they are declared inside a function. Local variables are visible and meaningful only inside the functions in which they are declared. They are not known to other functions. Note that the variable *i* has been declared in both the functions. Any change in the value of *i* in one function does not affect its value in the other.

C provides a variety of storage class specifiers that can be used to declare explicitly the scope and lifetime of variables. Table 2.10 provide a list of the various storage class specifiers along with their meanings:

TABLE 2.10 Storage classes and their meaning

Storage class	Meaning
auto	Local variable known only to the function in which it is declared. Default is auto.
static	Local variable which exists and retains its value even after the control is transferred to the calling function.
extern	Global variable known to all functions in the file.
register	Local variable which is stored in the register.

2.15 ASSIGNING VALUES TO VARIABLES

Variables are created for use in program statements such as,

```

value = amount + inrate * amount;
while (year <= PERIOD)
{
    ....
    ....
    year = year + 1;
}

```

In the first statement, the numeric value stored in the variable *inrate* is multiplied by the value stored in *amount*, and the product is added to *amount*. The result is stored in the variable *value*. This process is possible only if the variables *amount* and *inrate* have already been given values. The variable *value* is called the target variable.

2.32 Computer Programming

While all the variables are declared for their type, the variables that are used in expressions (on the right side of equal (=) sign of a computational statement) must be assigned values before they are encountered in the program. Similarly, the variable `year` and the symbolic constant `PERIOD` in the while statement must be assigned values before this statement is encountered.

2.15.1 Assignment Statement

Values can be assigned to variables using the assignment operator `=` as follows:

`variable_name = constant;`

Examples are:

```
initial_value = 0;
final_value = 100;
balance = 75.84;
yes = 'x';
```

C permits multiple assignments in one line. For example:

```
initial_value = 0; final_value = 100;
```

An assignment statement implies that the value of the variable on the left of the 'equal sign' is set equal to the value of the quantity (or the expression) on the right. The following statement means that the 'new value' of `year` is equal to the 'old value' of `year` plus 1.

```
year = year + 1;
```

During assignment operation, C converts the type of value on the right-hand side to the type on the left. This may involve truncation when real value is converted to an integer.

It is also possible to assign a value to a variable at the time the variable is declared. This takes the following form:

```
int final_value = 100;
char yes = 'x';
double balance = 75.84;
```

The process of giving initial values to variables is called initialisation. C permits the initialisation of more than one variables in one statement using multiple assignment operators. For example:

```
p = q = s = 0;
x = y = z = MAX;
```

Here, the first statement initialises the variables `p`, `q`, and `s` to zero while the second initialises `x`, `y`, and `z` with `MAX`. Note that `MAX` is a symbolic constant defined at the beginning.

NOTE: Remember that external and static variables are initialised to zero by default. Automatic variables that are not initialised explicitly contain garbage values.

Example 2.2 The program in Fig. 2.20 shows typical declarations, assignments and values stored in various types of variables.

```

Program
main()
{
/*.....DECLARATIONS.....*/
    float    x, p ;
    double   y, q ;
    unsigned k ;
/*.....DECLARATIONS AND ASSIGNMENTS.....*/
    int      m = 54321 ;
    long int n = 1234567890 ;
/*.....ASSIGNMENTS.....*/
    x = 1.234567890000 ;
    y = 9.87654321 ;
    k = 54321 ;
    p = q = 1.0 ;
/*.....PRINTING.....*/
    printf("m = %d\n", m) ;
    printf("n = %ld\n", n) ;
    printf("x = %.12lf\n", x) ;
    printf("x = %f\n", x) ;
    printf("y = %.12lf\n", y) ;
    printf("y = %lf\n", y) ;
    printf("k = %u p = %f q = %.12lf\n", k, p, q) ;
}
Output
m = -11215
n = 1234567890
x = 1.234567880630
x = 1.234568
y = 9.876543210000
y = 9.876543
k = 54321 p = 1.000000 q = 1.000000000000

```

Fig. 2.20 Examples of assignments

In the program given in Fig. 2.20, the variables x and p have been declared as floating-point variables. Note that the way the value of 1.234567890000 that we assigned to x is displayed under different output formats. The value of x is displayed as 1.234567880630 under `%.12lf` format, while the actual value assigned is 1.234567890000. This is because the variable x has been declared as a float that can store values only up to six decimal places.

The variable m that has been declared as `int` is not able to store the value 54321 correctly. Instead, it contains some garbage. Since this program was run on a 16-bit machine, the maximum value that an `int`

2.34 Computer Programming

variable can store is only 32767. However, the variable k (declared as unsigned) has stored the value 54321 correctly. Similarly, the long int variable n has stored the value 1234567890 correctly.

The value 9.87654321 assigned to y declared as double has been stored correctly but the value is printed as 9.876543 under %lf format. Note that unless specified otherwise, the printf function will always display a float or double value to six decimal places.

2.15.2 Reading Data from Keyboard

Another way of giving values to variables is to input data through keyboard using the scanf function. It is a general input function available in C and is very similar in concept to the printf function. It works much like an INPUT statement in BASIC. The general format of scanf is as follows:

```
scanf("control string", &variable1,&variable2,...);
```

The control string contains the format of data being received. The ampersand symbol & before each variable name is an operator that specifies the variable name's address. We must always use this operator, otherwise unexpected results may occur. Let us look at an example:

```
scanf("%d", &number);
```

When this statement is encountered by the computer, the execution stops and waits for the value of the variable number to be typed in. Since the control string "%d" specifies that an integer value is to be read from the terminal, we have to type in the value in integer form. Once the number is typed in and the 'Return' Key is pressed, the computer then proceeds to the next statement. Thus, the use of scanf provides an interactive feature and makes the program 'user friendly'. The value is assigned to the variable number.

Example 2.3 The program in Fig. 2.21 illustrates the use of scanf function.

The first executable statement in the program is a printf, requesting the user to enter an integer number. This is known as 'prompt message' and appears on the screen like as:

```
Enter an integer number
```

As soon as the user types in an integer number, the computer proceeds to compare the value with 100. If the value typed in is less than 100, then the following message is printed on the screen.

```
Your number is smaller than 100
```

Otherwise, the following message is displayed

```
Your number contains more than two digits
```


Outputs of the program run for two different inputs are also shown in Fig. 2.21.

```

Program
main()
{
    int number;

    printf("Enter an integer number\n");
    scanf ("%d", &number);
    if ( number < 100 )
        printf("Your number is smaller than 100\n\n");
    else
        printf("Your number contains more than two digits\n");
}
Output
Enter an integer number
54
Your number is smaller than 100
Enter an integer number
108
Your number contains more than two digits

```

Fig. 2.21 Use of scanf function for interactive computing.

NOTE: The above program uses a decision statement if...else to decide whether the number is less than 100. We will learn about decision-making statements later in this chapter.

Example 2.4 Figure 2.22 shows a program that uses the scanf function to receive values from the end user.

```

Program
main()
{
    int year, period ;
    float amount, inrate, value ;

    printf("Input amount, interest rate, and period\n\n") ;
    scanf ("%f %f %d", &amount, &inrate, &period) ;
    printf("\n") ;
    year = 1 ;

    while( year <= period )
    {
        value = amount + inrate * amount ;
        printf("%2d Rs %8.2f\n", year, value) ;
        amount = value ;
        year = year + 1 ;
    }
}

```

2.36 Computer Programming

```
}  
Output  
Input amount, interest rate, and period  
10000 0.14 5  
1 Rs 11400.00  
2 Rs 12996.00  
3 Rs 14815.44  
4 Rs 16889.60  
5 Rs 19254.15  
Input amount, interest rate, and period  
20000 0.12 7  
1 Rs 22400.00  
2 Rs 25088.00  
3 Rs 28098.56  
4 Rs 31470.39  
5 Rs 35246.84  
6 Rs 39476.46  
7 Rs 44213.63
```

Fig. 2.22 *Interactive investment program.*

In the above program, the computer requests the user to input the values of the amount to be invested, interest rate and period of investment by printing the following prompt message and then waits for input values.

```
Input amount, interest rate, and period
```

As soon as we finish entering the three values corresponding to the three variables amount, inrate and period, the computer begins to calculate the amount at the end of each year, up to 'period' and produces output as shown above.

2.15.3 Declaring a Variable as a Constant

We may like the value of certain variables to remain constant during the execution of a program. We can achieve this by declaring the variable with the qualifier `const` at the time of initialisation. Here is an example:

```
const int class_size = 40;
```

Here, `const` is a new data type qualifier defined by ANSI standard. This tells the compiler that the value of the `int` variable `class_size` must not be modified by the program. However, it can be used on the right_hand side of an assignment statement like any other variable.

2.15.4 Declaring a Variable as Volatile

ANSI standard defines another qualifier `volatile` that could be used to tell explicitly the compiler that a variable's value may be changed at any time by some external sources (from outside the program). Here is the example:

```
volatile int date;
```

The value of `date` may be altered by some external factors even if it does not appear on the left-hand side of an assignment statement. When we declare a variable as `volatile`, the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value.

Remember that the value of a variable declared as `volatile` can be modified by its own program as well. If we wish that the value must not be modified by the program while it may be altered by some other process, then we may declare the variable as both `const` and `volatile` as shown further:

```
volatile const int location = 100;
```

2.16 CASE STUDIES

2.16.1 Calculation of Average of Numbers

A program to calculate the average of a set of N numbers is given in Fig. 2.23.

```
Program
#define N 10          /* SYMBOLIC CONSTANT */
main()
{
    int count;          /* DECLARATION OF */
    float sum, average, number; /* VARIABLES */
    sum = 0;            /* INITIALIZATION */
    count = 0;          /* OF VARIABLES */
    printf("\nEnter 10 numbers");
    while( count < N )
    {
        scanf("%f", &number);
        sum = sum + number;
        count = count + 1;
    }
    average = sum/N;
    printf("N = %d Sum = %f", N, sum);
    printf(" Average = %f", average);
}
```

Output

```
1
2.3
4.67
1.42
7
3.67
4.08
2.2
4.25
8.21
N = 10      Sum = 38.799999 Average = 3.880
```

Fig. 2.23 Average of N numbers

2.38 Computer Programming

The variable `number` is declared as `float` and therefore it can take both integer and real numbers. Since the symbolic constant `N` is assigned the value of 10 using the `#define` statement, the program accepts ten values and calculates their sum using the `while` loop. The variable `count` counts the number of values and as soon as it becomes 11, the `while` loop is exited and then the average is calculated.

Notice that the actual value of `sum` is 38.8 but the value displayed is 38.799999. In fact, the actual value that is displayed is quite dependent on the computer system. Such an inaccuracy is due to the way the floating point numbers are internally represented inside the computer.

2.16.2 Temperature Conversion Problem

The program presented in Fig. 2.24 converts the given temperature in fahrenheit to celsius using the following conversion formula:

$$C = \frac{F - 32}{1.8}$$

```
Program
#define F_LOW  0 /* - - - - - */
#define F_MAX  250 /* SYMBOLIC CONSTANTS */
#define STEP 25 /* - - - - - */
main()
{
    typedef float REAL; /* TYPE DEFINITION */
    REAL fahrenheit, celsius; /* DECLARATION */
    fahrenheit = F_LOW; /* INITIALIZATION */
    printf("Fahrenheit Celsius\n\n");
    while( fahrenheit <= F_MAX )
    {
        celsius = ( fahrenheit - 32.0 ) / 1.8;
        printf("%5.1f %7.2f\n", fahrenheit, celsius);
        fahrenheit = fahrenheit + STEP;
    }
}
```

Output

Fahrenheit	Celsius
0.0	-17.78
25.0	-3.89
50.0	10.00
75.0	23.89
100.0	37.78
125.0	51.67
150.0	65.56
175.0	79.44
200.0	93.33
225.0	107.22
250.0	121.11

Fig. 2.24 Temperature conversion–fahrenheit-celsius

The program prints a conversion table for reading temperature in celsius, given the fahrenheit values. The minimum and maximum values and step size are defined as symbolic constants. These values can be changed by redefining the `#define` statements. A user-defined data type name `REAL` is used to declare the variables `fahrenheit` and `celsius`.

The formation specifications `%5.1f` and `%7.2` in the second `printf` statement produces two-column output as shown.

2.17 MANAGING INPUT AND OUTPUT OPERATIONS

Reading, processing and writing of data are the three essential functions of a computer program. Most programs take some data as input and display the processed data, often known as information or results, on a suitable medium. So far we have seen two methods of providing data to the program variables. One method is to assign values to variables through the assignment statements such as $x = 5$; $a = 0$; and so on. Another method is to use the input function `scanf`, which can read data from a keyboard. We have used both the methods in most of our earlier example programs. For outputting results, we have used extensively the function `printf`, which sends results out to a terminal.

Unlike other high-level languages, C does not have any built-in input/output statements as part of its syntax. All input/output operations are carried out through function calls such as `printf` and `scanf`. There exist several functions that have more or less become standard for input and output operations in C. These functions are collectively known as the standard I/O library.

2.17.1 Reading a Character

The simplest of all input/output operations is reading a character from the ‘standard input’ unit (usually the keyboard) and writing it to the ‘standard output’ unit (usually the screen). Reading a single character can be done by using the function `getchar`. `getchar` takes the following form:

```
variable_name = getchar( );
```

Here, `variable_name` is a valid C name that has been declared as `char` type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to `getchar` function. Since `getchar` is used on the right-hand side of an assignment statement, the character value of `getchar` is in turn assigned to the variable name on the left. For example, the following statements will assign the character ‘H’ to the variable name when we press the key H on the keyboard.

```
char name;
name = getchar();
```

Example 2.5 The program in Fig. 2.25 shows the use of `getchar` function in an interactive environment.

```
Program
#include <stdio.h>
main()
{
    char answer;
```

2.40 Computer Programming

```
printf("Would you like to know my name?\n");
printf("Type Y for YES and N for NO: ");
answer = getchar(); /*...Reading a character...*/
if(answer == 'Y' || answer == 'y')
    printf("\n\nMy name is BUSY BEE\n");
else
    printf("\n\nYou are good for nothing\n");
}
```

Output

```
Would you like to know my name?
Type Y for YES and N for NO: Y
My name is BUSY BEE
```

```
Would you like to know my name?
Type Y for YES and N for NO: n
You are good for nothing
```

Fig. 2.25 Use of `getchar` function to read a character from keyboard.

The above program in Fig. 2.25 displays a question of YES/NO type to the user and reads the user's response in a single character (Y or N). If the response is Y or y, it outputs the following message:

```
My name is BUSY BEE
```

However, if the response is N or n, it outputs the following message:

```
You are good for nothing
```

NOTE: There is one line space between the input text and output message.

The `getchar` function may be called successively to read the characters contained in a line of text. For example, the following program segment reads characters from keyboard one after another until the 'Return' key is pressed.

```
____-____-
____-____-
char character;
character = ' ';
while(character != '\n')
{
    character = getchar();
}
____-____-
____-____-
```

Example 2.6 The program shown in Fig. 2.26 requests the user to enter a character and displays a message on the screen telling the user whether the character is an alphabet or digit, or any other special character.

Program

```
#include <stdio.h>
#include <ctype.h>
main()
{
    char character;
    printf("Press any key\n");
    character = getchar();
    if (isalpha(character) > 0)/* Test for letter */
        printf("The character is a letter.");
    else
        if (isdigit (character) > 0)/* Test for digit */
            printf("The character is a digit.");
        else
            printf("The character is not alphanumeric.");
}
```

Output

```
Press any key
h
The character is a letter.
Press any key
5
The character is a digit.
Press any key
*
The character is not alphanumeric.
```

Fig. 2.26 Program to test the character type.

The above program receives in Fig. 2.26 a character from the keyboard and tests whether it is a letter or digit and prints out a message accordingly. These tests are done with the help of the following functions:

```
isalpha(character)
isdigit(character)
```

For example, `isalpha` assumes a value non-zero (TRUE) if the argument `character` contains an alphabet; otherwise it assumes 0 (FALSE). Similar is the case with the function `isdigit`.

C supports many other similar functions, which are given in Table 2.11. These character functions are contained in the file `ctype.h` and, therefore, the following pre-processor directive statement must be included in a program before using these functions:

```
#include <ctype.h>
```

2.17.2 Writing a Character

Like `getchar`, there is an analogous function `putchar` for writing characters one at a time to the terminal. It takes the form as shown further:

TABLE 2.11 Character test functions

Function	Test
<code>isalnum(c)</code>	Is <code>c</code> an alphanumeric character?
<code>isalpha(c)</code>	Is <code>c</code> an alphabetic character?
<code>isdigit(c)</code>	Is <code>c</code> a digit?
<code>islower(c)</code>	Is <code>c</code> lower case letter?
<code>isprint(c)</code>	Is <code>c</code> a printable character?
<code>ispunct(c)</code>	Is <code>c</code> a punctuation mark?
<code>isspace(c)</code>	Is <code>c</code> a white space character?
<code>isupper(c)</code>	Is <code>c</code> an upper case letter?

```
putchar (variable_name);
```

Here, `variable_name` is a type `char` variable containing a character. This statement displays the character contained in the `variable_name` at the terminal. For example, the following statements will display the character `Y` on the screen:

```
answer = 'Y';
putchar (answer);
```

Example 2.7 A program that reads a character from the keyboard and then prints it in reverse case is given in Fig. 2.27. That is, if the input is upper case, the output will be lower case and vice versa.

```
Program
#include <stdio.h>
#include <ctype.h>
main()
{
    char alphabet;
    printf("Enter an alphabet");
    putchar('\n'); /* move to next line */
    alphabet = getchar();
    if (islower(alphabet))
        putchar(toupper(alphabet)); /* Reverse and display */
    else
        putchar(tolower(alphabet)); /* Reverse and display */
}
```



```

Output
Enter an alphabet
a
A
Enter an alphabet
Q
q
Enter an alphabet
z
Z

```

Fig. 2.27 Reading and writing of alphabets in reverse case.

The above program in Fig. 2.27 uses three new functions: `islower`, `toupper`, and `tolower`. The function `islower` is a conditional function and takes the value `TRUE` if the argument is a lowercase alphabet; otherwise it takes the value `FALSE`. The function `toupper` converts the lowercase argument into an uppercase alphabet, while the function `tolower` does the reverse.

2.17.3 Formatted Input

Formatted input refers to the input data that have been arranged in a particular format. For example, consider the following data:

15.75 123 John

This line contains three pieces of data, arranged in a particular form. Such data have to be read conforming to the format of its appearance. For example, the first part of the data should be read into a variable float, the second into int, and the third part into char. This is possible in C using the `scanf` function.

We have already used this input function in a number of examples. Here, we shall explore all the options that are available for reading the formatted data with `scanf` function. The general form of `scanf` is

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

The control string specifies the field format in which the data are to be entered and the arguments `arg1`, `arg2`, ..., `argn` specify the address of locations where the data are stored. Control string and arguments are separated by commas.

Control string (also known as format string) contains field specifications, which direct the interpretation of input data. It may include:

- Field (or format) specifications, consisting the conversion character `%`, a data type character (or type specifier), and an optional number, specifying the field width.
- Blanks, tabs or newlines.
- Blanks, tabs and newlines are ignored. The data type character indicates the type of data that is to be assigned to the variable associated with the corresponding argument. The field width specifier is optional.

Inputting Integer Numbers

The field specification for reading an integer number is:

`%w sd`

The percentage sign (`%`) indicates that a conversion specification follows. `w` is an integer number that specifies the field width of the number to be read and `d`, known as data type character, indicates that the number to be read is in integer mode. Consider the following example:

2.44 Computer Programming

```
scanf ("%2d %5d", &num1, &num2);
```

Suppose the following data are entered at the console:

50 31426

Here, the value 50 is assigned to `num1` and 31426 to `num2`.

Now, suppose the input data are as follows:

31426 50

Now, the variable `num1` will be assigned 31 (because of `%2d`) and `num2` will be assigned 426 (unread part of 31426). The value 50 that is unread will be assigned to the first variable in the next `scanf` call. This kind of errors may be eliminated if we use the field specifications without the field-width specifications.

Input data items must be separated by spaces, tabs or newlines. Punctuation marks do not count as separators. When the `scanf` function searches the input data line for a value to be read, it will always bypass any white space characters.

What happens if we enter a floating point number instead of an integer? The fractional part may be stripped away! Also, `scanf` may skip reading further input.

When the `scanf` reads a particular value, reading of the value will be terminated as soon as the number of characters specified by the field width is reached (if specified) or until a character that is not valid for the value being read is encountered. In the case of integers, valid characters are an optionally signed sequence of digits.

An input field may be skipped by specifying `*` in the place of field width. For example, consider the following statement:

```
scanf ("%d %*d %d", &a, &b)
```

Suppose, you enter the following data as input:

123 456 789

In this case, the value 123 will be assigned to `a` and 789 to `b`. The value 456 will be skipped (because of `*`).

Further, the data type character `d` may be preceded by `'l'` to read long integers and `h` to read short integers.

Example 2.8 Various input formatting options for reading integers are experimented in the program shown in Fig. 2.28.

```
Program
main()
{
    int a,b,c,x,y,z;
    int p,q,r;
    printf("Enter three integer numbers\n");
    scanf ("%d %*d %d",&a,&b,&c);
    printf ("%d %d %d \n\n",a,b,c);
    printf("Enter two 4-digit numbers\n");
```

```

scanf("%2d %4d",&x,&y);
printf("%d %d\n\n", x,y);
printf("Enter two integers\n");
scanf("%d %d", &a,&x);
printf("%d %d \n\n",a,x);
printf("Enter a nine digit number\n");
scanf("%3d %4d %3d",&p,&q,&r);
printf("%d %d %d \n\n",p,q,r);
printf("Enter two three digit numbers\n");
scanf("%d %d",&x,&y);
printf("%d %d",x,y);
}
Output

```

```

Enter three integer numbers
1 2 3
1 3 -3577
Enter two 4-digit numbers
6789 4321
67 89
Enter two integers
44 66
4321 44
Enter a nine-digit number
123456789
66 1234 567
Enter two three-digit numbers
123 456
89 123

```

Fig. 2.28 Reading integers using *scanf*

In the above program in Fig. 2.28, the first *scanf* requests input data for three integer values *a*, *b* and *c*, and accordingly three values 1, 2 and 3 are keyed in. Because of the specification *%*d*, the value 2 has been skipped and 3 is assigned to the variable *b*. Notice that since no data are available for *c*, it contains garbage.

The second *scanf* specifies the format *%2d* and *%4d* for the variables *x* and *y* respectively. Whenever we specify field width for reading integer numbers, the input numbers should not contain more digits than the specified size. Otherwise, the extra digits on the right-hand side will be truncated and assigned to the next variable in the list. Thus, the second *scanf* has truncated the four digit number 6789 and assigned 67 to *x* and 89 to *y*. The value 4321 has been assigned to the first variable in the immediately following *scanf* statement.

NOTE: It is legal to use a non-whitespace character between field specifications. However, the *scanf* expects a matching character in the given location. For example, *scanf("%d-%d", &a, &b);* accepts input like 123-456 to assign 123 to *a* and 456 to *b*.

Inputting Real Numbers

Unlike integer numbers, the field width of real numbers is not to be specified and, therefore, *scanf* reads real numbers using the simple specification *%f* for both the notations, namely, decimal point notation and exponential notation. For example, consider the following statement

```
scanf("%f %f %f", &x, &y, &z);
```

Suppose, the following data are entered as input:

475.89 43.21E-1 678

Here, the value 475.89 will be assigned to *x*, 4.321 to *y* and 678.0 to *z*. The input field specifications may be separated by any arbitrary blank spaces.

If the number to be read is of double type, then the specification should be %lf instead of simple %f. A number may be skipped using %*f specification.

Example 2.9 Reading of real numbers (in both decimal point and exponential notation) is illustrated in Fig. 2.29.

```

Program
main()
{
    float x,y;
    double p,q;
    printf("Values of x and y:");
    scanf("%f %e", &x, &y);
    printf("\n");
    printf("x = %f\n y = %f\n\n", x, y);
    printf("Values of p and q:");
    scanf("%lf %lf", &p, &q);
    printf("\n\n p = %.12lf\n q = %.12e", p,q);
}
Output
Values of x and y:12.3456 17.5e-2
x = 12.345600
y = 0.175000
Values of p and q:4.142857142857 18.5678901234567890
p = 4.142857142857
q = 1.856789012346e+001

```

Fig. 2.29 Reading of real numbers.

Inputting Character Strings

We have already seen how a single character can be read from the terminal using the `getchar` function. The same can be achieved using the `scanf` function also. In addition, a `scanf` function can input strings containing more than one character. Following are the specifications for reading character strings:

%ws or %wc

The corresponding argument should be a pointer to a character array. However, %c may be used to read a single character when the argument is a pointer to a `char` variable.

Example 2.10 Reading of strings using %wc and %ws is illustrated in Fig. 2.30.

Program

```

main()
{
    int no;
    char name1[15], name2[15], name3[15];
    printf("Enter serial number and name one\n");
    scanf("%d %15c", &no, name1);
    printf("%d %15s\n\n", no, name1);
    printf("Enter serial number and name two\n");
    scanf("%d %s", &no, name2);
    printf("%d %15s\n\n", no, name2);
    printf("Enter serial number and name three\n");
    scanf("%d %15s", &no, name3);
    printf("%d %15s\n\n", no, name3);
}

```

Output

```

Enter serial number and name one
1 123456789012345
1 123456789012345r
Enter serial number and name two
2 New York
2      New
Enter serial number and name three
2      York
Enter serial number and name one
1 123456789012
1 123456789012r
Enter serial number and name two
2 New-York
2      New-York
Enter serial number and name three
3 London
3      London

```

Fig. 2.30 Reading of strings

The program in Fig. 2.30 illustrates the use of various field specifications for reading strings. When we use %wc for reading a string, the system will wait until the wth character is keyed in. Note that the specification %s terminates reading at the encounter of a blank space. Therefore, name2 has read only the first part of 'New York' and the second part is automatically assigned to name3. However, during the second run, the string 'New-York' is correctly assigned to name2.

Inputting Mixed Data Types

It is possible to use one scanf statement to input a data line containing mixed mode data. In such cases, care should be exercised to ensure that the input data items match the control specifications in order and type. When an attempt is made to read an item that does not match the type expected, the scanf function does not read any further and immediately returns the values read. For example, consider the following statement:

2.48 Computer Programming

```
scanf ("%d %c %f %s", &count, &code, &ratio, name);
```

Suppose, you enter the following data as input:

15 p 1.575 coffee

Here, the values will be correctly read and assigned to the variables in the order in which they appear. Some systems accept integers in the place of real numbers and vice versa, and the input data are converted to the type specified in the control string.

NOTE: A space before the %c specification in the format string is necessary to skip the white space before p.

Detecting Errors in Input

When a scanf function completes reading its list, it returns the value of number of items that are successfully read. This value can be used to test whether any errors occurred in reading the input. For example, consider the following statement:

```
scanf ("%d %f %s, &a, &b, name);
```

Suppose, the following data is entered at the console:

20 150.25 motor

This will return the value 3 as three input values have been entered in correct order.

However, the value 1 will be returned if the following line is entered:

20 motor 150.25

This is because the function would encounter a string when it was expecting a floating-point value and would, therefore, terminate its scan after reading the first value.

Example 2.11 The program shown in Fig. 2.31 illustrates the testing for correctness of reading of data by scanf function.

```
Program
main()
{
    int a;
    float b;
    char c;
    printf("Enter values of a, b and c\n");
    if (scanf("%d %f %c", &a, &b, &c) == 3)
        printf("a = %d b = %f c = %c\n", a, b, c);
    else
        printf("Error in input.\n");
}
```

Output

```

Enter values of a, b and c
12 3.45 A
a = 12    b = 3.450000    c = A
Enter values of a, b and c
23 78 9
a = 23    b = 78.000000    c = 9
Enter values of a, b and c
8 A 5.25
Error in input.
Enter values of a, b and c
Y 12 67
Error in input.
Enter values of a, b and c
15.75 23 X
a = 15    b = 0.750000    c = 2

```

Fig. 2.31 Detection of errors in *scanf* input.

The function `scanf` is expected to read three items of data and, therefore, when the values for all the three variables are read correctly, the program prints out their values. During the third run, the second item does not match with the type of variable and, therefore, the reading is terminated and the error message is printed. Same is the case with the fourth run.

In the last run, although data items do not match the variables, no error message has been printed. When we attempt to read a real number for an *int* variable, the integer part is assigned to the variable, and the truncated decimal part is assigned to the next variable.

Table 2.12 shows the commonly used `scanf` format codes:

TABLE 2.12 Commonly used `scanf` format codes

Code	Meaning
<code>%c</code>	read a single character
<code>%d</code>	read a decimal integer
<code>%e</code>	read a floating-point value
<code>%f</code>	read a floating-point value
<code>%g</code>	read a floating-point value
<code>%h</code>	read a short integer
<code>%i</code>	read a decimal, hexadecimal or octal integer
<code>%o</code>	read an octal integer
<code>%s</code>	read a string
<code>%u</code>	read an unsigned decimal integer
<code>%x</code>	read a hexadecimal integer
<code>%[..]</code>	read a string of word(s)

The following letters may be used as prefix for certain conversion characters.

- **h**: For short integers
- **l**: For long integers or double
- **L**: For long double

2.17.4 Points to Remember while Using scanf

If we do not plan carefully, some ‘crazy’ things can happen with scanf. Following are some of the general points to keep in mind while writing a scanf statement.

- All function arguments, except the control string, must be pointers to variables.
- Format specifications contained in the control string should match the arguments in order.
- Input data items must be separated by spaces and must match the variables receiving the input in the same order.
- The reading will be terminated, when scanf encounters a ‘mismatch’ of data or a character that is not valid for the value being read.
- When searching for a value, scanf ignores line boundaries and simply looks for the next appropriate character.
- Any unread data items in a line will be considered as part of the data input line to the next scanf call.
- When the field width specifier *w* is used, it should be large enough to contain the input data size.
- Never end the format string with whitespace. It is a fatal error!
- The scanf reads until:
 - A whitespace character is found in a numeric specification, or
 - The maximum number of characters have been read or
 - An error is detected, or
 - The end of file is reached

2.17.5 Formatted Output

We have seen the use of `printf` function for printing captions and numerical results. It is highly desirable that the outputs are produced in such a way that they are understandable and are in an easy-to-use form. It is, therefore, necessary for the programmer to give careful consideration to the appearance and clarity of the output produced by his program.

The `printf` statement provides certain features that can be effectively exploited to control the alignment and spacing of outputs on the terminals. The general form of `printf` statement is:

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

In this code, the control string may comprise the following three types of items:

- Characters that will be printed on the screen as they appear.
- Format specifications that define the output format for display of each item.
- Escape sequence characters such as `\n`, `\t`, and `\b`.

The control string indicates how many arguments follow and what their types are. The arguments *arg1*, *arg2*,, *argn* are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in number, order and type with the format specifications.

A simple format specification has the following form:

```
% w.p type-specifier
```


Here, w is an integer number that specifies the total number of columns for the output value and p is another integer number that specifies the number of digits to the right of the decimal point (of a real number) or the number of characters to be printed from a string. Both w and p are optional. Some examples of formatted printf statement are as under:

```
printf("Programming in C");
printf(" ");
printf("\n");
printf("%d", x);
printf("a = %f\n b = %f", a, b);
printf("sum = %d", 1234);
printf("\n\n");
```

printf never supplies a newline automatically and, therefore, multiple printf statements may be used to build one line of output. A newline can be introduced by the help of a newline character '\n' as shown in some of the examples above.

Output of Integer Numbers

The format specification for printing an integer number is:

```
% w d
```

Here w specifies the minimum field width for the output. However, if a number is greater than the specified field width, it will be printed in full, overriding the minimum specification. d specifies that the value to be printed is an integer. The number is written right-justified in the given field width. Leading blanks will appear as necessary. Table 2.13 illustrates the output of the number 9876 under different formats.

TABLE 2.13 Commonly used scanf format codes

Format	Output
printf("%d", 9876)	9876
printf("%6d", 9876)	9876
printf("%2d", 9876)	9876
printf("%-6d", 9876)	9876
printf("%06d", 9876)	009876

It is possible to force the printing to be left-justified by placing a minus sign directly after the % character, as shown in the fourth example here. It is also possible to pad with zeros the leading blanks by placing a 0 (zero) before the field width specifier, as shown in the last item here. The minus (-) and zero (0) are known as flags.

Long integers may be printed by specifying ld in the place of d in the format specification. Similarly, we may use hd for printing short integers.

Example 2.12 The program in Fig. 2.32 illustrates the output of integer numbers under various formats.

```

Program
    main()
    {
        int m = 12345;
        long n = 987654;
        printf("%d\n",m);
        printf("%10d\n",m);
        printf("%010d\n",m);
        printf("%-10d\n",m);
        printf("%10ld\n",n);
        printf("%10ld\n",-n);
    }

```

Output

```

12345
      12345
0000012345
12345
      987654
- 987654

```

Fig. 2.32 Formatted output of integers.

Output of Real Numbers

The output of a real number may be displayed in decimal notation using the following format specification:

```
% w.p f
```

The integer w indicates the minimum number of positions that are to be used for the display of the value and the integer p indicates the number of digits to be displayed after the decimal point (precision). The value, when displayed, is rounded to p decimal places and printed right-justified in the field of w columns. Leading blanks and trailing zeros will appear as necessary. The default precision is 6 decimal places. The negative numbers will be printed with the minus sign. The number will be displayed in the form $[-] \text{mmm-nnn}$.

We can also display a real number in exponential notation by using the specification:

```
% w.p e
```

The display takes the following form:

```
[ - ] m.nnnne[ ± ]xx
```

Here, the length of the string of n 's is specified by the precision p . The default precision is 6. The field width w should satisfy the condition.

$w^3 p+7$

The value will be rounded off and printed right justified in the field of w columns.

Padding the leading blanks with zeros and printing with left-justification are also possible by using flags 0 or – before the field width specifier w . Table 2.14 illustrates the output of the number $y = 98.7654$ under different format specifications:

TABLE 2.14 Commonly used scanf format codes

Format	Output
<code>printf(“%7.4f”,y)</code>	9 8 . 7 6 5 4
<code>printf(“%7.2f”,y)</code>	9 8 . 7 6
<code>printf(“%-7.2f”,y)</code>	9 8 . 7 7
<code>printf(“%f”,y)</code>	9 8 . 7 6 5 4
<code>printf(“%10.2e”,y)</code>	9 . 8 8 e + 0 1
<code>printf(“%11.4e”,-y)</code>	- 9 . 8 7 6 5 e + 0 1
<code>printf(“%-10.2e”,y)</code>	9 . 8 8 e + 0 1
<code>printf(“%e”,y)</code>	9 . 8 7 6 5 4 0 e + 0 1

Some systems also support a special field specification character that lets the user define the field size at run time. This takes the following form:

`printf(“%*.*f”, width, precision, number);`

In this case, both the field width and the precision are given as arguments which will supply the values for w and p . For example, `printf(“%*.*f”,7,2,number);` is equivalent to `printf(“%7.2f”,number);`. The advantage of this format is that the values for width and precision may be supplied at run time, thus making the format a dynamic one.

Example 2.13 All the options of printing a real number are illustrated in Fig. 2.33.

```
Program
main()
{
    float y = 98.7654;
    printf(“%7.4f\n”, y);
    printf(“%f\n”, y);
    printf(“%7.2f\n”, y);
    printf(“%-7.2f\n”, y);
    printf(“%07.2f\n”, y);
    printf(“%*.*f”, 7, 2, y);
}
```

```

    printf("\n");
    printf("%10.2e\n", y);
    printf("%12.4e\n", -y);
    printf("%-10.2e\n", y);
    printf("%e\n", y);
}
Output

```

```

98.7654
98.765404
98.77
98.77
0098.77
98.77
9.88e+001
-9.8765e+001
9.88e+001
9.876540e+001

```

Fig. 2.33 Formatted output of real numbers

Printing of a Single Character

A single character can be displayed in a desired position using the format:

```
%wC
```

The character will be displayed right-justified in the field of w columns. We can make the display left-justified by placing a minus sign before the integer w . The default value for w is 1.

Printing of Strings

The format specification for outputting strings is similar to that of real numbers. It is of the following form:

```
%w.ps
```

Here, w specifies the field width for display and p instructs that only the first p characters of the string are to be displayed. The display is right-justified. Figure 2.34 shows the effect of variety of specifications in printing a string “NEW DELHI 110001”, containing 16 characters (including blanks).

Specification	Output
	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
%s	N E W D E L H I 1 1 0 0 0 1
%20s	N E W D E L H I 1 1 0 0 0 1
%20.10s	N E W D E L H I
%.5s	N E W D
%-20.10s	N E W D E L H I
%5s	N E W D E L H I 1 1 0 0 0 1

Fig. 2.34 Printing a string through different format specifications.

Mixed Data Output

It is permitted to mix data types in one `printf` statement. For example, the following statement is valid in C.

```
printf("%d %f %s %c", a, b, c, d);
```

As pointed out earlier, `printf` uses its control string to decide how many variables to be printed and what their types are. Therefore, the format specifications should match the variables in number, order and type. If there are not enough variables or if they are of the wrong type, the output results will be incorrect.

Table 2.15 shows some commonly used `printf` format codes:

TABLE 2.15 Commonly used `printf` Format Codes

Code	Meaning
%c	print a single character
%d	print a decimal integer
%e	print a floating point value in exponent form
%f	print a floating point value without exponent
%g	print a floating point value either e-type or f-type depending on
%i	print a signed decimal integer
%o	print an octal integer, without leading zero
%s	print a string
%u	print an unsigned decimal integer
%x	print a hexadecimal integer, without leading Ox

The following letters may be used as prefix for certain conversion characters.

- h: for short integers
- l: for long integers or double
- L: for long double.

Enhancing the Readability of Output

Computer outputs are used as information for analysing certain relationships between variables and for making decisions. Therefore, the correctness and clarity of outputs are of utmost importance. While the correctness depends on the solution procedure, the clarity depends on the way the output is presented. Following are some of the steps we can take to improve the clarity and hence the readability and understandability of outputs.

- Provide enough blank space between two numbers.
- Introduce appropriate headings and variable names in the output.
- Print special messages whenever a peculiar condition occurs in the output.
- Introduce blank lines between the important sections of the output.

2.18 CASE STUDIES

1. Inventory Report

Problem

The ABC Electric Company manufactures four consumer products. Their inventory position on a particular day is given below:

Code	Quantity	Rate (Rs)
F105	275	575.00
H220	107	99.95
I019	321	215.50
M315	89	725.00

It is required to prepare the inventory report table in the following format:

INVENTORY REPORT

Code	Quantity	Rate	Value
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—
Total Value:			—

The value of each item is given by the product of quantity and rate.

Program

The program given in Fig. 2.35 reads the data from the terminal and generates the required output.

```

Program
#define ITEMS 4
main()
{
    /* BEGIN */
    int i, quantity[5];
    float rate[5], value, total_value;
    char code[5][5];      /* READING VALUES */
    i = 1;
    while ( i <= ITEMS)
    {
        printf("Enter code, quantity, and rate:");
        scanf("%s %d %f", code[i], &quantity[i], &rate[i]);
        i++;
    }
    /*.....Printing of Table and Column Headings.....*/
    printf("\n\n");
    printf(" INVENTORY REPORT \n");
    printf("— — — — — \n");
    printf(" Code Quantity Rate Value \n");
    printf("— — — — — \n");
    /*.....Preparation of Inventory Position.....*/
    total_value = 0;
    i = 1;

```

```

while ( i <= ITEMS)
{
    value = quantity[i] * rate[i];
    printf("%5s %10d %10.2f %e\n",code[i],quantity[i],rate[i],value);
    total_value += value;
    i++;
}
/*.....Printing of End of Table.....*/
printf("----- \n");
printf("Total Value = %e\n",total_value);
printf("----- \n");
} /* END */

```

Output

```

Enter code, quantity, and rate:F105 275 575.00
Enter code, quantity, and rate:H220 107 99.95
Enter code, quantity, and rate:I019 321 215.50
Enter code, quantity, and rate:M315 89 725.00

```

INVENTORY REPORT

Code	Quantity	Rate	Value
F105	275	575.00	1.581250e+005
H220	107	99.95	1.069465e+004
I019	321	215.50	6.917550e+004
M315	89	725.00	6.452500e+004
Total Value =		3.025202e+005	

Fig. 2.35 Program for inventory report

2. Reliability Graph

Problem The reliability of an electronic component is given by the following equation:

$$\text{reliability (r)} = e^{-\lambda t}$$

where:

- λ is the component failure rate per hour
- t is the time of operation in hours

A graph is required to determine the reliability at various operating times, from 0 to 3000 hours. The failure rate (λ) is 0.001.

Program The program given in Fig. 2.36 produces a shaded graph. The values of t are self-generated by the for statement

for ($t=0$; $t \leq 3000$; $t = t+150$) in steps of 150.

The integer 50 in the statement $R = (\text{int})(50*r+0.5)$ is a scale factor which converts r to a large value where an integer is used for plotting the curve. Remember r is always less than 1.

```

Problem
#include <math.h>
#define LAMBDA 0.001

```

2.58 Computer Programming

```
main()
{
    double t;
    float r;
    int i, R;
    for (i=1; i<=27; ++i)
    {
        printf("- ");
    }
    printf("\n");
    for (t=0; t<=3000; t+=150)
    {
        r = exp(-LAMBDA*t);
        R = (int)(50*r+0.5);
        printf(" |");
        for (i=1; i<=R; ++i)
        {
            printf("*");
        }
        printf("#\n");
    }
    for (i=1; i<3; ++i)
    {
        printf(" | \n");
    }
}
```

Output

[illegible]

Fig. 2.36 Program to draw reliability graph

2.19 OPERATORS AND EXPRESSIONS

C supports a rich set of built-in operators. We have already used several of them, such as `=`, `+`, `-`, `*`, `&` and `<`. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.

C operators can be classified into a number of categories. They include:

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operators
- Increment and decrement operators
- Conditional operators
- Bitwise operators
- Special operators

An expression is a sequence of operands and operators that reduces to a single value. For example, the following expression will result in the value 25.

`15 + 10`

2.19.1 Arithmetic Operators

C provides all the basic arithmetic operators. They are listed in Table 2.16. The operators `+`, `-`, `*` and `/` all work the same way as they do in other languages. These can operate on any built-in data type allowed in C. The unary minus operator, in effect, multiplies its single operand by `-1`. Therefore, a number preceded by a minus sign changes its sign.

TABLE 2.16 Arithmetic operators

Operator	Meaning
<code>+</code>	Addition or unary plus
<code>-</code>	Subtraction or unary minus
<code>*</code>	Multiplication
<code>/</code>	Division
<code>%</code>	Modulo division

Integer division truncates any fractional part. The modulo division operation produces the remainder of an integer division. Examples of use of arithmetic operators are:

```
a - b
a + b

a * b
a / b
a % b
-a * b
```

2.60 Computer Programming

Here, a and b are variables and are known as operands. The modulo division operator `%` cannot be used on floating point data. Note that C does not have an operator for exponentiation. Older version of C does not support unary plus but ANSI C supports it.

Integer Arithmetic

When both the operands in a single arithmetic expression such as $a+b$ are integers, the expression is called an integer expression, and the operation is called integer arithmetic. Integer arithmetic always yields an integer value. The largest integer value depends on the machine, as pointed out earlier. In the above examples, if a and b are integers, then for $a = 14$ and $b = 4$ we have the following results:

```
a - b =      10
a + b =      18
a * b =      56
a / b =       3 (decimal part truncated)
a % b =       2 (remainder of division)
```

During integer division, if both the operands are of the same sign, the result is truncated towards zero. If one of them is negative, the direction of truncation is implementation dependent. That is,

$6/7 = 0$ and $-6/-7 = 0$

but $-6/7$ may be zero or -1 (machine dependent).

Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend), as shown further:

```
-14 % 3      =      -2
-14 % -3     =      -2
14 % -3      =       2
```

Example 2.14 The program in Fig. 2.37 shows the use of integer arithmetic to convert a given number of days into months and days.

```
Program
main ()
{
    int months, days ;

    printf("Enter days\n") ;
    scanf("%d", &days) ;

    months = days / 30 ;
    days = days % 30 ;
    printf("Months = %d Days = %d", months, days) ;
}

Output
Enter days
265
Months = 8 Days = 25
Enter days
364
Months = 12 Days = 4
Enter days
45
Months = 1 Days = 15
```

Fig. 2.37 Illustration of integer arithmetic.

The variables `months` and `days` are declared as integers. Therefore, the statement `months = days/30;` truncates the decimal part and assigns the integer part to `months`. Similarly, the statement `days = days%30;` assigns the remainder part of the division to `days`. Thus the given number of days is converted into an equivalent number of months and days and the result is printed as shown in the output.

Real Arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation. Since, floating-point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result. If x , y and z are floats, then we will have:

$$\begin{aligned} x &= 6.0/7.0 = 0.857143 \\ y &= 1.0/3.0 = 0.333333 \\ z &= -2.0/3.0 = -0.666667 \end{aligned}$$

The operator `%` cannot be used with real operands.

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression. If either operand is of real type, then only the real operation is performed and the result is always a real number. Thus

$$15/10.0 = 1.5$$

whereas

$$15/10 = 1$$

More about mixed operations will be discussed later when we deal with the evaluation of expressions.

2.19.2 Relational Operators

We often compare two quantities and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of relational operators. We have already used the symbol '`<`', meaning 'less than'.

An expression such as $a < b$ or $1 < 20$ containing a relational operator is termed as a relational expression. The value of a relational expression is either one or zero. It is one if the specified relation is true and zero if the relation is false. For example

$$10 < 20 \text{ is true}$$

but

$$20 < 10 \text{ is false}$$

C supports six relational operators in all. These operators and their meanings are illustrated in Table 2.17.

TABLE 2.17 Relational operators

Operator	Meaning
<code><</code>	is less than
<code><=</code>	is less than or equal to
<code>></code>	is greater than
<code>>=</code>	is greater than or equal to
<code>==</code>	is equal to
<code>!=</code>	is not equal to

2.62 Computer Programming

A simple relational expression contains only one relational operator and takes the following form:

```
ae-1 relational operator ae-2
```

Here, ae-1 and ae-2 are arithmetic expressions, which may be simple constants, variables or combination of them. Following are some examples of simple relational expressions and their values:

```
4.5 <= 10    TRUE
5 < -10      FALSE
-35 >= 0     FALSE
10 < 7+5     TRUE
a+b = c+d    TRUE only if the sum of values of a and b is equal to the sum of values of c and d.
```

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators.

Relational expressions are used in decision-making statements such as **if** and **while** to decide the course of action of a running program.

2.19.3 Logical Operators

In addition to the relational operators, C supports the following three logical operators.

```
&& meaning logical AND
|| meaning logical OR
! meaning logical NOT
```

The logical operators **&&** and **||** are used when we want to test more than one condition and make decisions. An example is:

```
a > b && x == 10
```

An expression of this kind, which combines two or more relational expressions, is termed as a logical expression or a compound relational expression. Like the simple relational expressions, a logical expression also yields a value of one or zero, according to the truth table, shown in Table 2.18. The logical expression given here is true only if $a > b$ is true and $x == 10$ is true. If either (or both) of them are false, the expression is false.

TABLE 2.18 Truth table

op-1	op-2	Value of the expression	
		op-1 && op-2	op-1 op-2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Some examples of the usage of logical expressions are:

1. if (age > 55 && salary < 1000)
2. if (number < 0 || number > 100)

We will see more of them when we discuss decision statements.

NOTE: Relative precedence of the relational and logical operators is as follows:

Highest	!
	> >= < <=
	== !=
	&&
Lowest	

It is important to remember this when we use these operators in compound expressions.

2.19.4 Assignment Operators

Assignment operators are used to assign the result of an expression to a variable. We have seen the usual assignment operator, '='. In addition, C has a set of 'shorthand' assignment operators of the form

```
v op= exp;
```

Here, *v* is a variable, *exp* is an expression and *op* is a C binary arithmetic operator. The operator *op=* is known as the shorthand assignment operator.

The assignment statement *v op= exp;* is equivalent to

```
v = v op (exp);
```

Here, *v* evaluated only once.

Now, consider the following example

```
x += y+1;
```

This expression is same as the following:

```
x = x + (y+1);
```

The shorthand operator *+=* means 'add *y+1* to *x*' or 'increment *x* by *y+1*'. For *y* = 2, the above statement becomes

```
x += 3;
```

When the above statement is executed, 3 is added to *x*. If the old value of *x* is, say 5, then the new value of *x* is 8. Some of the commonly used shorthand assignment operators are illustrated in Table 2.19:

TABLE 2.19 Shorthand Assignment Operators

Statement with simple assignment operator	Statement with shorthand operator
<code>a = a + 1</code>	<code>a += 1</code>
<code>a = a - 1</code>	<code>a -= 1</code>
<code>a = a * (n+1)</code>	<code>a *= n+1</code>
<code>a = a / (n+1)</code>	<code>a /= n+1</code>
<code>a = a % b</code>	<code>a %= b</code>

The use of shorthand assignment operators has the following advantages:

- What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
- The statement is more concise and easier to read.
- The statement is more efficient.

These advantages may be appreciated if we consider a slightly more involved statement like

```
value(5*j-2) = value(5*j-2) + delta;
```

With the help of the `+=` operator, this can also be written as follows:

```
value(5*j-2) += delta;
```

It is easier to read and understand and is more efficient because the expression `5*j-2` is evaluated only once.

Example 2.15 The program in Fig. 2.38 prints a sequence of squares of numbers. Note the use of the shorthand operator `*=`.

```

Program
#define N 100
#define A 2
main()
{
    int a;
    a = A;
    while( a < N )
    {
        printf("%d\n", a);
        a *= a;
    }
}

Output
2
4
16

```

Fig. 2.38 Use of shorthand operator `*=`

The program attempts to print a sequence of squares of numbers starting from 2. The statement `a *= a;` is identical to `a = a*a;` and it replaces the current value of `a` by its square. When the value of `a` becomes equal or greater than `N` (`=100`) the while is terminated. Note that the output contains only three values 2, 4 and 16.

2.19.5 Increment and Decrement Operators

C allows two very useful operators not generally found in other languages. These are the increment and decrement operators:

`++` and `--`

The operator `++` adds 1 to the operand, while `--` subtracts 1. Both are unary operators and take the following form:

`++m;` or `m++;`

`--m;` or `m--;`

`++m;` is equivalent to `m = m+1;` (or `m += 1;`)

`--m;` is equivalent to `m = m-1;` (or `m -= 1;`)

We use the increment and decrement statements in for and while loops extensively. While `++m` and `m++` mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following expressions:

```
m = 5;
y = ++m;
```

In this case, the value of `y` and `m` would be 6. Suppose, if we rewrite the above statements as

```
m = 5;
y = m++;
```

Here, the value of `y` would be 5 and `m` would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.

Similar is the case when we use `++` (or `--`) in subscripted variables. That is, the statement `a[i++] = 10;` is equivalent to the following:

```
a[i] = 10;
i = i+1;
```

The increment and decrement operators can be used in complex statements. For example:

```
m = n++ - j+10;
```

Here, the old value of `n` is used in evaluating the expression. `n` is incremented after the evaluation. Some compilers require a space on either side of `n++` or `++n`.

2.66 Computer Programming

Some of the rules for using ++ and -- operators are:

- Increment and decrement operators are unary operators and thus require variable as their operands.
- When postfix ++ (or --) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
- When prefix ++ (or --) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associativity of ++ and -- operators are the same as those of unary + and unary -.

2.19.6 Conditional Operator

A ternary operator pair '?' ':' is available in C to construct conditional expressions of the following form:

```
exp1 ? exp2 : exp3
```

where exp1, exp2 and exp3 are expressions.

The operator ? : works as follows: exp1 is evaluated first. If it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either exp2 or exp3) is evaluated. For example, consider the following statements.

```
a = 10;  
b = 15;  
x = (a > b) ? a : b;
```

In this example, x will be assigned the value of b. This can also be achieved using the if...else statements as follows:

```
if (a > b)  
    x = a;  
else  
    x = b;
```

Now, consider the evaluation of the following function:

$$y = 1.5x + 3 \text{ for } x \leq 2$$
$$y = 2x + 5 \text{ for } x > 2$$

This can be evaluated using the conditional operator as follows:

```
y = ( x > 2 ) ? ( 2 * x + 5 ) : ( 1.5 * x + 3 );
```

The conditional operator may be nested for evaluating more complex assignment decisions. For example, consider the weekly salary of a salesgirl who is selling some domestic products. If x is the number of products sold in a week, her weekly salary is given by

$$\text{salary} = \begin{cases} 4x + 100 & \text{for } x < 40 \\ 300 & \text{for } x = 40 \\ 4.5x + 150 & \text{for } x > 40 \end{cases}$$

This complex equation can be written using conditional operators as follows:

```
salary = (x != 40) ? ((x < 40) ? (4*x+100) : (4.5*x+150)) : 300;
```

The same can be evaluated using if...else statements as follows:

```
if (x <= 40)
    if (x < 40)
        salary = 4 * x+100;
    else
        salary = 300;
else
    salary = 4.5 * x+150;
```

When the conditional operator is used, the code becomes more concise, and perhaps more efficient. However, the readability is poor. It is better to use if statements when more than a single nesting of conditional operator is required.

2.19.7 Bitwise Operators

C has a distinction of supporting special operators known as bitwise operators for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double. Table 4.20 provides a list the bitwise operators and their meanings.

TABLE 2.20 Bitwise operators

Operator	Meaning
&	bitwise AND
	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

2.19.8 Special Operators

C supports some special operators of interest such as comma operator, sizeof operator, pointer operators (& and *) and member selection operators (. and →).

Comma Operator

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated left to right and the value of the right-most expression is the value of the combined expression. For example, consider the following statement:

2.68 Computer Programming

```
value = (x = 10, y = 5, x+y);
```

This statement first assigns the value 10 to *x*, then assigns 5 to *y*, and finally assigns 15 (i.e. 10 + 5) to *value*. Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Some applications of comma operator are:

- In for loops:

```
for ( n = 1, m = 10, n <=m; n++, m++)
```

- In while loops:

```
while (c = getchar( ), c != '10')
```

- Exchanging values:

```
t = x, x = y, y = t;
```

Size of Operator

The `sizeof` is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier. Some examples are shown further:

```
m = sizeof (sum);  
n = sizeof (long int);  
k = sizeof (235L);
```

The `sizeof` operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Example 2.16 Figure 2.39 shows a program that employs different kinds of operators. The results of their evaluation are also shown for comparison.

Program

```
main()  
{  
    int a, b, c, d;  
    a = 15;  
    b = 10;  
    c = ++a - b;  
    printf("a = %d b = %d c = %d\n", a, b, c);  
    d = b++ + a;  
    printf("a = %d b = %d d = %d\n", a, b, d);  
    printf("a/b = %d\n", a/b);  
}
```

```

        printf("a%%b = %d\n", a%b);
        printf("a *= b = %d\n", a*=b);
        printf("%d\n", (c>d) ? 1 : 0);
        printf("%d\n", (c<d) ? 1 : 0);
    }
Output
a = 16 b = 10 c = 6
a = 16 b = 11 d = 26
a/b = 1
a%b = 5
a *= b = 176
0
1

```

Fig. 2.39 Further illustration of arithmetic operators.

2.19.9 Operator Precedence

Precedence of operators refers to the order in which they are operated in a program. Table 2.21 provides a list of the precedence of operators.

TABLE 2.21 Precedence of operators

Type of operator	Operators	Associativity
Unary operators	+, −, !, ~, ++, --, type, size of	Right to left
Arithmetic operators	*, /, %, +, -	Left to right
Bit-manipulation operators	<<, >>	Left to right
Relational operators	>, <, >=, <=, ==, !=	Left to right
Logical operators	&&,	Left to right
Conditional operators	?, :	Left to right
Assignment operators	=, +=, -=, *=, /=, %=	Right to left

It is important to note the following:

- Precedence rules decide the order in which different operators are applied
- Associativity rule decides the order in which multiple occurrences of the same level operator are applied

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form:

```
variable = expression;
```

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted. Examples of evaluation statements are as follows:

```
x = a * b - c;
y = b / c * a;
z = a - b / c + d;
```

The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables *a*, *b*, *c* and *d* must be defined before they are used in the expressions.

Example 2.17 The program in Fig. 2.40 illustrates the use of variables in expressions and their evaluation. The output of the program also illustrates the effect of presence of parentheses in expressions.

Program

```
main()
{
    float a, b, c, x, y, z;
    a = 9;
    b = 12;
    c = 3;

    x = a - b / 3 + c * 2 - 1;
    y = a - b / (3 + c) * (2 - 1);
    z = a - (b / (3 + c) * 2) - 1;

    printf("x = %f\n", x);
    printf("y = %f\n", y);
    printf("z = %f\n", z);
}
```

Output

```
x = 10.000000
y = 7.000000
z = 4.000000
```

Fig. 2.40 Illustrations of evaluation of expressions

2.19.10 Precedence of Arithmetic Operators

An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C:

High priority * / %

Low priority + -

The basic evaluation procedure includes ‘two’ left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered.

Now, consider the following evaluation statement:

$$x = a - b / 3 + c * 2 - 1$$

When $a = 9$, $b = 12$, and $c = 3$, the statement becomes

$$x = 9 - 12/3 + 3*2 - 1$$

And, it is evaluated as follows

First pass

Step1: $x = 9 - 4 + 3*2 - 1$

Step2: $x = 9 - 4 + 6 - 1$

Second pass

Step3: $x = 5 + 6 - 1$

Step4: $x = 11 - 1$

Step5: $x = 10$

These steps are illustrated in Fig. 2.41. The numbers inside parentheses refer to step numbers.

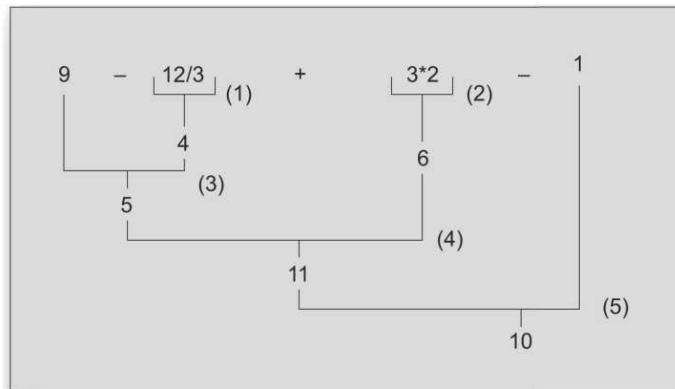


Fig. 2.41 Illustration of hierarchy of operations

However, the order of evaluation can be changed by introducing parentheses into an expression.

Consider the same expression with parentheses as shown below:

$$9 - 12/(3+3)*(2-1)$$

Whenever parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

First pass

Step1: $9 - 12/6 * (2-1)$

Step2: $9 - 12/6 * 1$

Second pass

Step3: $9 - 2 * 1$

Step4: $9 - 2$

Third pass

Step5: 7

This time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remains the same as 5 (i.e. equal to the number of arithmetic operators).

2.72 Computer Programming

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. Just make sure that every opening parenthesis has a matching closing parenthesis.

While parentheses allow us to change the order of priority, we may also use them to improve understandability of the program. When in doubt, we can always add an extra pair just to make sure that the priority assumed is the one we require.

Rules for Evaluation of Expression

1. First, parenthesized sub-expression from left to right are evaluated.
2. If parentheses are nested, the evaluation begins with the innermost sub-expression.
3. The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
4. The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
5. Arithmetic expressions are evaluated from left to right using the rules of precedence.
6. When parentheses are used, the expressions within parentheses assume highest priority.

2.19.11 Some Computational Problems

When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors. We know that the computer gives approximate values for real numbers and the errors due to such approximations may lead to serious problems. For example, consider the following statements:

```
a = 1.0/3.0;  
b = a * 3.0;
```

We know that $(1.0/3.0) \cdot 3.0$ is equal to 1. But there is no guarantee that the value of b computed in a program will equal 1.

Another problem is division by zero. On most computers, any attempt to divide a number by zero will result in abnormal termination of the program. In some cases such a division may produce meaningless results. Care should be taken to test the denominator that is likely to assume zero value and avoid any division by zero.

The third problem is to avoid overflow or underflow errors. It is our responsibility to guarantee that operands are of the correct type and range, and the result may not produce any overflow or underflow.

Example 2.18 The program in Fig. 2.42 shows round-off errors that can occur in computation of floating point numbers.

```
Program  
/*_____ Sum of n terms of 1/n _____-*/  
main()
```

```

{
    float sum, n, term ;
    int count = 1 ;
    sum = 0 ;
    printf("Enter value of n\n") ;
    scanf("%f", &n) ;
    term = 1.0/n ;
    while( count <= n )
    {
        sum = sum + term ;
        count++ ;
    }
    printf("Sum = %f\n", sum) ;
}

```

Output

```

Enter value of n
99
Sum = 1.000001
Enter value of n
143
Sum = 0.999999

```

Fig. 2.42 Round-off errors in floating point computations

We know that the sum of n terms of $1/n$ is 1. However, due to errors in floating point representation, the result is not always 1.

2.19.12 Type Conversions in Expressions

Implicit Type Conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic conversion is known as implicit-type conversion.

During evaluation, it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of the higher type. A typical type conversion process is illustrated in Fig. 2.43.

Figure 2.44 shows the conversion hierarchy for implicit-type conversion in an expression.

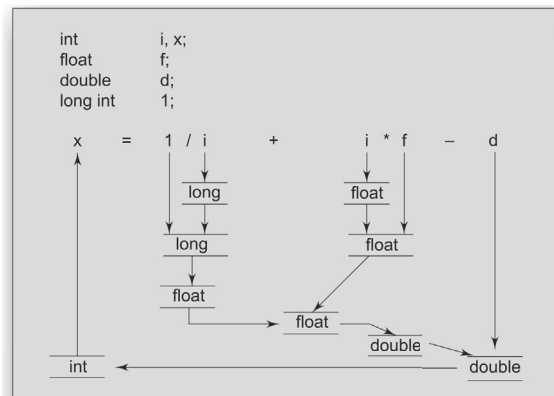


Fig. 2.43 Process of implicit type conversion

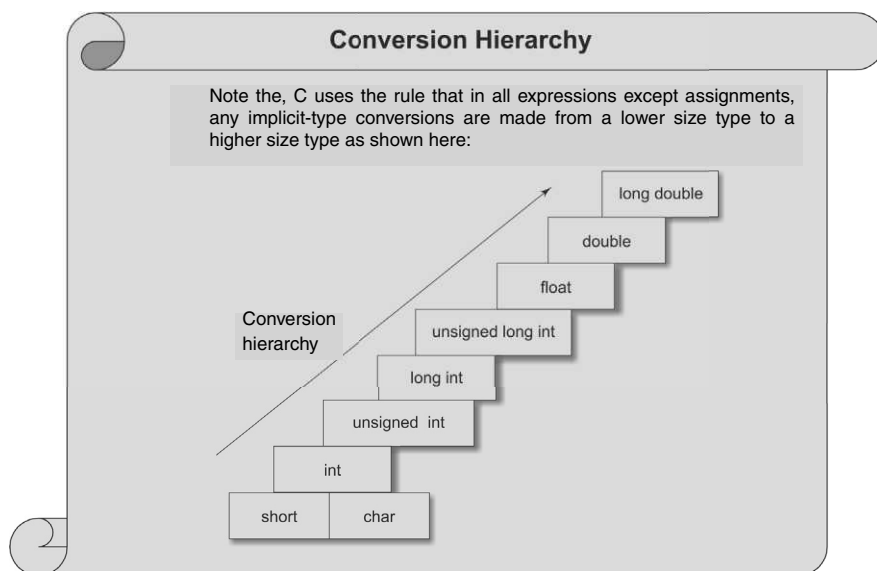


Fig. 2.44 Conversion hierarchy

The sequence of rules that are applied while implicit type conversion is as follows: all `short` and `char` are automatically converted to `int`; then

1. if one of the operands is `long double`, the other will be converted to `long double` and the result will be `long double`;
2. else, if one of the operands is `double`, the other will be converted to `double` and the result will be `double`;
3. else, if one of the operands is `float`, the other will be converted to `float` and the result will be `float`;
4. else, if one of the operands is `unsigned long int`, the other will be converted to `unsigned long int` and the result will be `unsigned long int`;
5. else, if one of the operands is `long int` and the other is `unsigned int`, then
 - (a) if `unsigned int` can be converted to `long int`, the `unsigned int` operand will be converted as such and the result will be `long int`;
 - (b) else, both operands will be converted to `unsigned long int` and the result will be `unsigned long int`;
6. else, if one of the operands is `long int`, the other will be converted to `long int` and the result will be `long int`;
7. else, if one of the operands is `unsigned int`, the other will be converted to `unsigned int` and the result will be `unsigned int`.

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.

- `float` to `int` causes truncation of the fractional part.
- `double` to `float` causes rounding of digits.
- `long int` to `int` causes dropping of the excess higher order bits.

Explicit Conversion

We have just learnt how C performs type conversion automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

ratio = female_number/male_number

Since female_number and male_number are declared as integers in the program, the decimal part of the result of the division would be lost and ratio would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

ratio = (float) female_number/male_number

The operator (float) converts the female_number to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator (float) affect the value of the variable female number. And also, the type of female number remains as int in the other parts of the program.

The process of such a local conversion is known as explicit conversion or casting a value. The general form of a cast is:

(type-name) expression

Here, type-name is one of the standard C data types. The expression may be a constant, variable or an expression. Some examples of casts and their actions are shown in Table 2.22.

TABLE 2.22 Use of casts

Example	Action
<code>x = (int) 7.5</code>	7.5 is converted to integer by truncation.
<code>a = (int) 21.3/(int)4.5</code>	Evaluated as 21/4 and the result would be 5.
<code>b = (double)sum/n</code>	Division is done in floating point mode.
<code>y = (int) (a+b)</code>	The result of <code>a+b</code> is converted to integer.
<code>z = (int)a+b</code>	<code>a</code> is converted to integer and then added to <code>b</code> .
<code>p = cos((double)x)</code>	Converts <code>x</code> to double before using it.

Casting can be used to round-off a given value. For example, consider the following statement:

```
x = (int) (y+0.5);
```

If `y` is 27.6, `y+0.5` is 28.1 and on casting, the result becomes 28, the value that is assigned to `x`. Of course, the expression, being cast is not changed.

2.19.13 Operator Precedence and Associativity

As mentioned earlier each operator, in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of these levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from ‘left to right’ or from ‘right to left’, depending on the level. This is known as the associativity property of an operator. Table 2.23 provides a complete list of operators, their precedence levels, and their rules of association. The groups are listed in the order of decreasing precedence. Rank 1 indicates the highest precedence level and 15 the lowest. The list also includes those operators, which we have not yet been discussed.

TABLE 2.23 Summary of C Operators

Operator	Description	Associativity	Rank
()	Function call	Left to right	1
[]	Array element reference		
+	Unary plus	Right to left	2
−	Unary minus		
++	Increment		
−−	Decrement		
!	Logical negation		
~	Ones complement		
*	Pointer reference (indirection)		
&	Address	Left to right	3
sizeof	Size of an object		
(type)	Type cast (conversion)		
*	Multiplication		
/	Division		
%	Modulus	Left to right	4
+	Addition		
−	Subtraction		
<<	Left shift	Left to right	5
>>	Right shift		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to right	7
!=	Inequality		
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
=	Assignment operators	Right to left	14
* = /= % =			
+ = − = & =			
^ = =			
<< = >> =			
,	Comma operator	Left to right	15

It is very important to note carefully, the order of precedence and associativity of operators. Consider the following conditional statement:

```
if (x == 10 + 15 && y < 10)
```

The precedence rules say that the addition operator has a higher priority than the logical operator (&&) and the relational operators (== and <). Therefore, the addition of 10 and 15 is executed first. This is equivalent to:

```
if (x == 25 && y < 10)
```

The next step is to determine whether x is equal to 25 and y is less than 10. If we assume a value of 20 for x and 5 for y, then

x == 25 is FALSE (0)

y < 10 is TRUE (1)

Note that since the operator < enjoys a higher priority as compared to ==, y < 10 is tested first and then x == 25 is tested.

Finally we get:

if (FALSE && TRUE)

Because one of the conditions is FALSE, the complex condition is FALSE.

In the case of &&, it is guaranteed that the second operand will not be evaluated if the first is zero and in the case of ||, the second operand will not be evaluated if the first is non-zero.

Applying De Morgan's Rule

While designing decision statements, we often come across a situation where the logical NOT operator is applied to a compound logical expression, like !(x&& y || !z). However, a positive logic is always easy to read and comprehend than a negative logic. In such cases, we may apply what is known as De Morgan's rule to make the total expression positive. This rule is as follows:

"Remove the parentheses by applying the NOT operator to every logical expression component, while complementing the relational operators."

That is,

- x becomes !x
- !x becomes x
- && becomes ||
- || becomes &&

Examples:

- !(x && y || !z) becomes !x || !y && z
- !(x <= 0 || !condition) becomes x > 0 && condition

Dangling Else Problem

One of the classic problems encountered when we start using nested if...else statements is the dangling else. This occurs when a matching else is not available for an if. The answer to this problem is very simple. Always match an else to the most recent unmatched if in the current block. In some cases, it is possible that the false condition is not required. In such situations, else statement may be omitted

"else is always paired with the most recent unpaired if"

2.20 CASE STUDIES

1. Salesman's Salary

Problem

A computer manufacturing company has the following monthly compensation policy to their sales-persons:

- Minimum base salary: 1500.00
- Bonus for every computer sold: 200.00
- Commission on the total monthly sales: 2 per cent

Since the prices of computers are changing, the sales price of each computer is fixed at the beginning of every month.

Program

Given the base salary, bonus, and commission rate, the inputs necessary to calculate the gross salary are the price of each computer and the number sold during the month

The gross salary is given by the equation:

Gross salary = base salary + (quantity * bonus rate) + (quantity * Price) * commission rate

A program to compute a sales-person's gross salary is given in Fig. 2.45.

```

Program
#define BASE_SALARY 1500.00
#define BONUS_RATE 200.00
#define COMMISSION 0.02
main()
{
    int quantity;
    float gross_salary, price;
    float bonus, commission;
    printf("Input number sold and price\n");
    scanf("%d %f", &quantity, &price);
    bonus      = BONUS_RATE * quantity;
    commission  = COMMISSION * quantity * price;
    gross_salary = BASE_SALARY + bonus + commission ;
    printf("\n");
    printf("Bonus      = %6.2f\n", bonus);
    printf("Commission  = %6.2f\n", commission);
    printf("Gross salary = %6.2f\n", gross_salary);
}

```

Output

```

Input number sold and price
5 20450.00
Bonus      = 1000.00
Commission  = 2045.00
Gross salary = 4545.00

```

Fig. 2.45 Program of salesman's salary

2. Solution of the Quadratic Equation

An equation of the form is known as the quadratic equation:

$$ax^2 + bx + c = 0$$

The values of x that satisfy the equation are known as the roots of the equation. A quadratic equation has two roots, which are given by the following two formulae:

$$\text{root 1} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$\text{root 2} = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

A program to evaluate these roots is given in Fig. 2.46. The program requests the user to input the values of a , b and c and outputs root 1 and root 2.

```

Program
#include <math.h>
main()
{
    float a, b, c, discriminant, root1, root2;
    printf("Input values of a, b, and c\n");
    scanf("%f %f %f", &a, &b, &c);
    discriminant = b*b - 4*a*c ;
    if(discriminant < 0)
        printf("\n\nROOTS ARE IMAGINARY\n");
    else
    {
        root1 = (-b + sqrt(discriminant))/(2.0*a);
        root2 = (-b - sqrt(discriminant))/(2.0*a);
        printf("\n\nRoot1 = %5.2f\n\nRoot2 = %5.2f\n",
            root1, root2 );
    }
}
Output
Input values of a, b, and c
2 4 -16
Root1 = 2.00
Root2 = -4.00
Input values of a, b, and c
1 2 3
ROOTS ARE IMAGINARY

```

Fig. 2.46 Solution of a quadratic equation

The term $(b^2 - 4ac)$ is called the discriminant. If the discriminant is less than zero, its square roots cannot be evaluated. In such cases, the roots are said to be imaginary numbers and the program outputs an appropriate message.

2.21 DECISION MAKING AND BRANCHING

We have seen that a C program is a set of statements that are normally executed sequentially in the order in which they appear. This happens when no options or no repetitions of certain calculations are necessary.

However, in practice, we have a number of situations where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

C language possesses such decision-making capabilities by supporting the following statements:

- if statement
- switch statement
- goto statement

These statements are popularly known as decision-making statements. Since these statements ‘control’ the flow of execution, they are also known as control statements. We have already used some of these statements in the earlier examples. Here, we will discuss their features, capabilities and applications in detail.

2.21.1 Decision Making with if Statement

The if statement is a powerful decision-making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression. It takes the following form:

```
if (test expression)
```

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is ‘true’ (or non-zero) or ‘false’ (zero), it transfers the control to a particular statement. This point of program has two paths to follow—one for the true condition and the other for the false condition, as shown in Fig. 2.47.

Some examples of decision making, using if statements, are:

- if (bank balance is zero)
 borrow money
- if (room is dark)
 put on lights
- if (code is 1)
 person is male
- if (age is more than 55)
 person is retired

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are:

- Simple if statement
- If...else statement
- Nested if...else statement
- else if ladder.

We will discuss each one of them in the following sections.

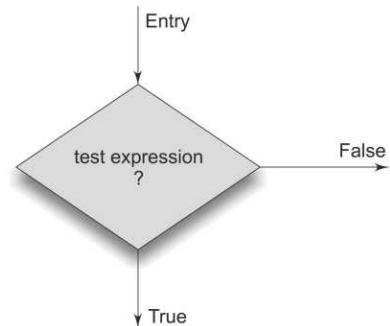


Fig. 2.47 Two-way branching

Simple if Statement

The general form of a simple if statement is

```
if (test expression)
{
    statement-block;
}
statement-x;
```

The `statement-block` may be a single statement or a group of statements. If the `test expression` is true, the `statement-block` will be executed; otherwise the `statement-block` will be skipped and the execution will jump to the `statement-x`. Remember, when the condition is true, both the `statement-block` and the `statement-x` are executed in sequence. This is illustrated in Fig. 2.48.

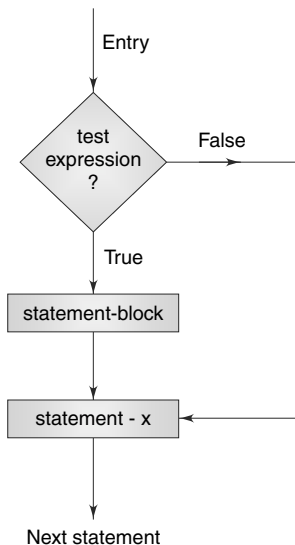


Fig. 2.48 Flow chart of simple if control

Consider the following segment of a program that is written for processing of the marks obtained in an entrance examination.

```
.....
.....
if (category == SPORTS)
{
    marks = marks + bonus_marks;
}
printf("%f", marks);
.....
.....
```

The program tests the type of category of the student. If the student belongs to the `SPORTS` category, then additional `bonus_marks` are added to his marks before they are printed. For others, `bonus_marks` are not added.

Example 2.19 The program given in Fig. 2.49 reads four values a, b, c and d from the terminal and evaluates the ratio of (a+b) to (c-d) and prints the result, if c-d is not equal to zero.

```

Program
main()
{
    int a, b, c, d;
    float ratio;

    printf("Enter four integer values\n");
    scanf("%d %d %d %d", &a, &b, &c, &d);

    if (c-d != 0) /* Execute statement block */
    {
        ratio = (float)(a+b)/(float)(c-d);
        printf("Ratio = %f\n", ratio);
    }
}

```

Output

```

Enter four integer values
12 23 34 45
Ratio = -3.181818

Enter four integer values
12 23 34 34

```

Fig. 2.49 Illustration of simple if statement

The above program has been run for two sets of data to see that the paths function properly. The result of the first run is printed as,

```
Ratio = -3.181818
```

The second run has neither produced any results nor any message. During the second run, the value of (c-d) is equal to zero and, therefore, the statements contained in the `statement-block` are skipped. Since no other statement follows the `statement-block`, program stops without producing any output.

Note the use of float conversion in the statement evaluating the ratio. This is necessary to avoid truncation due to integer division. Remember, the output of the first run -3.181818 is printed correct to six decimal places. The answer contains a round off error. If we wish to have higher accuracy, we must use double or long double data type.

The simple if is often used for counting purposes, as illustrated in the following example.

Example 2.20 The program given in Fig. 2.50 counts the number of boys whose weight is less than 50 kg and height is greater than 170 cm.

Program

```

main()
{
    int count, i;
    float weight, height;

    count = 0;
    printf("Enter weight and height for 10 boys\n");

    for (i =1; i <= 10; i++)
    {
        scanf("%f %f", &weight, &height);
        if (weight < 50 && height > 170)
            count = count + 1;
    }
    printf("Number of boys with weight < 50 kg\n");
    printf("and height > 170 cm = %d\n", count);
}

```

Output

```

Enter weight and height for 10 boys
45  176.5
55  174.2
47  168.0
49  170.7
54  169.0
53  170.5
49  167.0
48  175.0
47  167
51  170
Number of boys with weight < 50 kg
and height > 170 cm = 3

```

Fig. 2.50 Use of if for counting

The above program tests two conditions, one for weight and another for height. This is done using the compound relation

```
if (weight < 50 && height > 170)
```

This would have been equivalently done using two if statements as follows:

```

if (weight < 50)
if (height > 170)
    count = count +1;

```

2.84 Computer Programming

If the value of weight is less than 50, then the following statement is executed, which in turn is another `if` statement. This `if` statement tests height and if the height is greater than 170, then the count is incremented by 1.

The `if...else` Statement

The `if...else` statement is an extension of the simple `if` statement. It takes the following general form:

```
If (test expression)
{
    True-block statement(s)
}
else
{
    False-block statement(s)
}
statement-x
```

If the test expression is true, then the true-block statement(s), immediately following the `if` statements, are executed; otherwise, the false-block statement(s) are executed. In either case, either true-block or false-block will be executed, not both. This is illustrated in Fig. 2.51. In both the cases, the control is transferred subsequently to the statement-x.

Let us consider an example of counting the number of boys and girls in a class. We use code 1 for a boy and 2 for a girl. The program statement to do this may be written as follows:

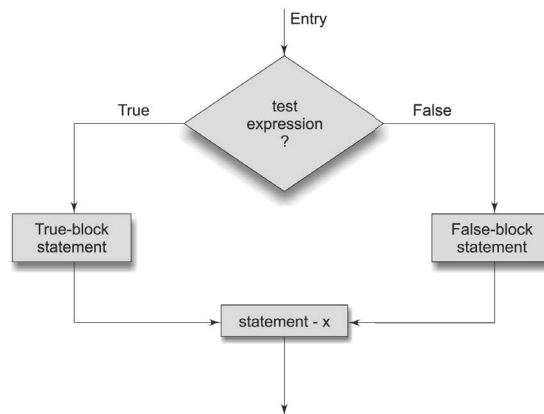


Fig. 2.51 Flow chart of `if...else` control

```
.....
.....
if (code == 1)
    boy = boy + 1;
if (code == 2)
    girl = girl+1;
.....
.....
```

The first test determines whether or not the student is a boy. If yes, the number of boys is increased by 1 and the program continues to the second test. The second test again determines whether the student is a girl. This is unnecessary. Once a student is identified as a boy, there is no need to test again for a girl. A student can be either a boy or a girl, not both. The above program segment can be modified using the `else` clause as follows:

```

.....
.....
if (code == 1)
    boy = boy + 1;
else
    girl = girl + 1;
xxxxxxxxx
.....

```

Here, if the code is equal to 1, the statement `boy = boy + 1;` is executed and the control is transferred to the statement `xxxxxxxx`, after skipping the `else` part. If the code is not equal to 1, the statement `boy = boy + 1;` is skipped and the statement in the `else` part `girl = girl + 1;` is executed before the control reaches the statement `xxxxxxxx`.

Nesting of if...else Statements

When a series of decisions are involved, we may have to use more than one if...else statement in nested form as shown in Fig. 2.52:

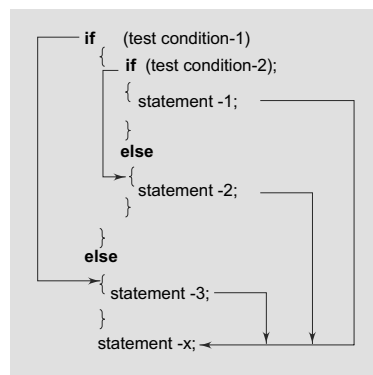


Fig. 2.52 Nested if...else statements

The logic of execution is illustrated in Fig. 2.53. If the condition-1 is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the condition-2 is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

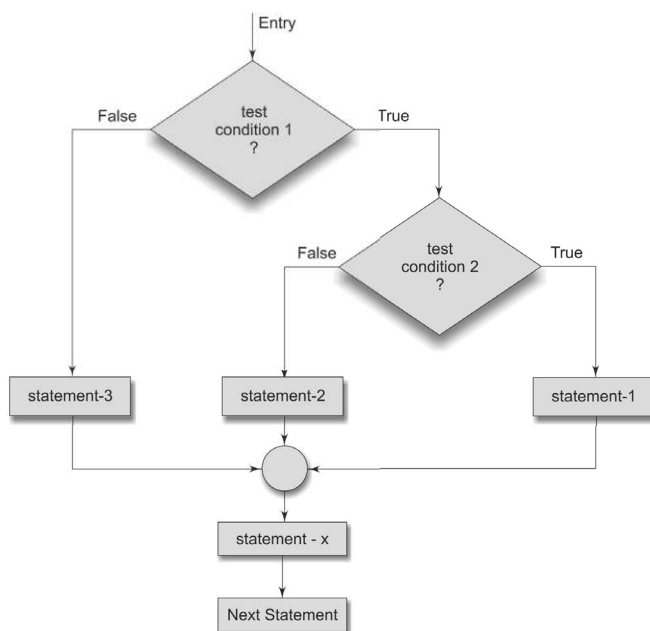


Fig. 2.53 Flow chart of nested if...else statements

Let us now try to understand the concept of nested `if...else` with the help of a real-life example. A commercial bank has introduced an incentive policy of giving bonus to all its deposit holders. The policy is as follows: A bonus of 2 per cent of the balance held on 31st December is given to every one, irrespective of their balance, and 5 per cent is given to female account holders if their balance is more than Rs. 5000. This logic can be coded with the help of nested `if...else` as follows:

```

.....
if (sex is female)
{
    if (balance > 5000)
        bonus = 0.05 * balance;
    else
        bonus = 0.02 * balance;
}
else
{
    bonus = 0.02 * balance;
}
balance = balance + bonus;
.....
.....

```

When nesting, care should be exercised to match every `if` with an `else`. Consider the following alternative to the earlier program (which looks right at the first sight):

```

if (sex is female)
    if (balance > 5000)
        bonus = 0.05 * balance;
    else
        bonus = 0.02 * balance;
    balance = balance + bonus;

```

There is an ambiguity as to over which if the else belongs to. In C, an else is linked to the closest non-terminated if. Therefore, the else is associated with the inner if and there is no else option for the outer if. This means that the computer is trying to execute the statement `balance = balance + bonus;` without really calculating the bonus for the male account holders.

Example 2.21 The program given in Fig. 2.54 selects and prints the largest of the three numbers using nested if...else statements.

```

Program
main()
{
    float A, B, C;
    printf("Enter three values\n");
    scanf("%f %f %f", &A, &B, &C);
    printf("\nLargest value is ");
    if (A>B)
    {
        if (A>C)
            printf("%f\n", A);
        else
            printf("%f\n", C);
    }
    else
    {
        if (C>B)
            printf("%f\n", C);
        else
            printf("%f\n", B);
    }
}

```

```

Output
Enter three values
23445 67379 88843
Largest value is 88843.000000

```

Fig. 2.54 Selecting the largest of three numbers.

The else if Ladder

There is another way of putting ifs together when multipath decisions are involved. A multipath decision is a chain of ifs in which the statement associated with each else is an if. It takes the general form, as shown in Fig. 2.55:

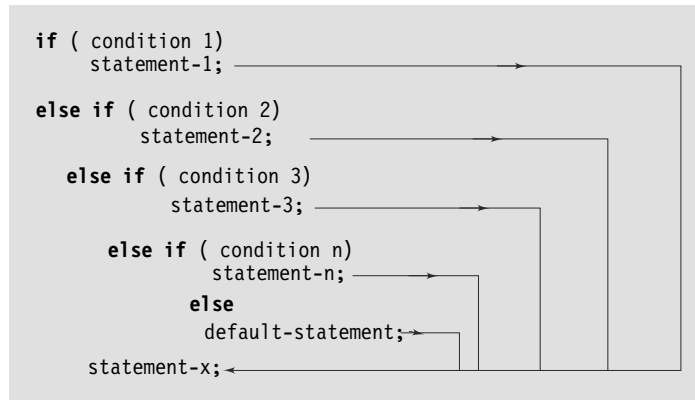


Fig. 2.55 The else if ladder

This construct is known as the `else if` ladder. The conditions are evaluated from the top (of the ladder) downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the `statement-x` (skipping the rest of the ladder). When all the `n` conditions become false, then the final `else` containing the default-statement will be executed. Figure 2.56 shows the logic of execution of `else if` ladder statements.

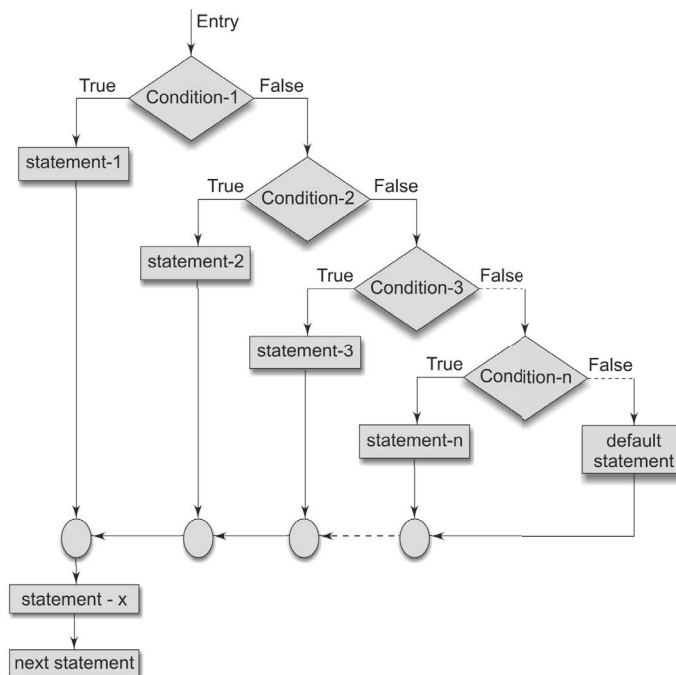


Fig. 2.56 Flow chart of else...if ladder.

Let us consider an example given below:

```

——
——
if (code == 1)
    colour = "RED";
else if (code == 2)
    colour = "GREEN";
    else if (code == 3)
        colour = "WHITE";
    else
        colour = "YELLOW";
——
——

```

Here, code numbers other than 1, 2 or 3 are considered to represent YELLOW colour. The same results can be obtained by using nested if...else statements, as follows:

```

if (code != 1)
    if (code != 2)
        if (code != 3)
            colour = "YELLOW";
        else
            colour = "WHITE";
    else
        colour = "GREEN";
else
    colour = "RED";

```

In such situations, the choice is left to the programmer. However, in order to choose an if structure that is both effective and efficient, it is important that the programmer is fully aware of the various forms of an if statement and the rules governing their nesting.

Rules for Indentation

When using control structures, a statement often controls many other following statements. In such situations, it is a good practice to use indentation to show that the indented statements are dependent on the preceding controlling statement. Some guidelines that could be followed while using indentation are listed as follows:

- Indent statements that are dependent on the previous statements; provide at least three spaces of indentation.
- Align vertically else clause with their matching if clause.
- Use braces on separate lines to identify a block of statements.
- Indent the statements in the block by at least three spaces to the right of the braces.
- Align the opening and closing braces.
- Use appropriate comments to signify the beginning and end of blocks.
- Indent the nested statements as per the above rules.
- Code only one clause or statement on each line.

2.21.2 The switch Statement

We have seen that when one of the many alternatives is to be selected, we can use an `if` statement to control the selection. However, the complexity of such a program increases dramatically when the number of alternatives increases. The program becomes difficult to read and follow. At times, it may confuse even the person who designed it. Fortunately, C has a built-in multiway decision statement known as a `switch`. The `switch` statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form of the `switch` statement is shown further:

```
switch (expression)
{
    case value-1:
        block-1
        break;
    case value-2:
        block-2
        break;
    .....
    .....
    default:
        default-block
        break;
}
statement-x;
```

The expression is an integer expression or characters. `value-1`, `value-2`,...are constants or constant expressions (evaluable to an integral constant) and are known as case labels. Each of these values should be unique within a `switch` statement. `block-1`, `block-2`,...are statement lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that case labels end with a colon (:).

When the `switch` is executed, the value of the expression is successfully compared against the values `value-1`, `value-2`,.... If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

The `break` statement at the end of each block signals the end of a particular case and causes an exit from the `switch` statement, transferring the control to the `statement-x` following the `switch`.

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the `statement-x`. (ANSI C permits the use of as many as 257 case labels)

The selection process of `switch` statement is illustrated in the flow chart shown in Fig. 2.57.

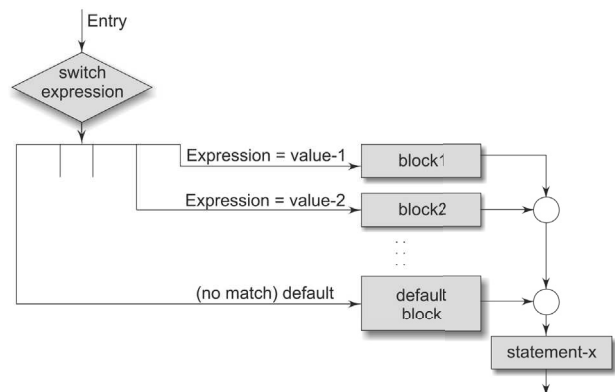


Fig. 2.57 Selection process of the switch statement.

The switch statement is often used for menu selection. For example:

```

____
____
printf(" TRAVEL GUIDE\n\n");
printf(" A Air Timings\n" );
printf(" T Train Timings\n");
printf(" B Bus Service\n" );
printf(" X To skip\n" );
printf("\n Enter your choice\n");
character = getchar();
switch (character)
{
    case 'A' :
        air-display();
        break;
    case 'B' :
        bus-display();
        break;
    case 'T' :
        train-display();
        break;
    default :
        printf(" No choice\n");
}
____
____

```

It is possible to nest the switch statements. That is, a switch may be part of a case statement. ANSI C permits 15 levels of nesting.

Here are some key rules for using switch statement:

- The switch expression must be an integral type.
- Case labels must be constants or constant expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with semicolon.
- The break statement transfers the control out of the switch statement.
- The break statement is optional. That is, two or more case labels may belong to the same statements.
- The default label is optional. If present, it will be executed when the expression does not find a matching case label.
- There can be at most one default label.
- The default may be placed anywhere but usually placed at the end.
- It is permitted to nest switch statements.

2.21.3 The goto Statement

So far we have discussed ways of controlling the flow of execution based on certain specified conditions. Like many other languages, C supports the `goto` statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the `goto` statement in a highly structured language like C, there may be occasions when the use of `goto` might be desirable.

The `goto` requires a label in order to identify the place where the branch is to be made. A label is any valid variable name, and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred. The general forms of `goto` and label statements are shown in Fig. 2.58:



Fig. 2.58 Use of `goto` statement

The `label:` can be anywhere in the program either before or after the `goto label;` statement. During running of a program when a statement like `goto begin;` is met, the flow of control jumps to the statement immediately following the label `begin`. This happens unconditionally.

Note that a `goto` breaks the normal sequential execution of the program. If the `label:` is before the statement `goto label;` a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a backward jump. On the other hand, if the `label:` is placed after the `goto label;`, some statements will be skipped and the jump is known as a forward jump.

A `goto` is often used at the end of a program to direct the control to go to the input statement, to read further data. Consider the following example:

```
main()
{
    double x, y;

    read:
    scanf("%f", &x);
    if (x < 0) goto read;
    y = sqrt(x);
    printf("%f %f\n", x, y);
    goto read;
}
```

This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses two `goto` statements, one at the end, after printing the results to transfer the control back to the input statement, and the other to skip any further computation when the number is negative.

Due to the unconditional `goto` statement at the end, the control is always transferred back to the input statement. In fact, this program puts the computer in a permanent loop known as an infinite loop. The computer

goes round and round until we take some special steps to terminate the loop. To avoid such situations, it is always advisable not to use `goto` statement.

Example 2.22 The program shown in Fig. 2.59 illustrates the use of `goto` statement.

Program

```
#include <math.h>
main()
{
    double x, y;
    int count;
    count = 1;
    printf("Enter FIVE real values in a LINE \n");
read:
    scanf("%lf", &x);
    printf("\n");
    if (x < 0)
        printf("Value - %d is negative\n",count);
    else
    {
        y = sqrt(x);
        printf("%lf\t %lf\n", x, y);
    }
    count = count + 1;
    if (count <= 5)
        goto read;
    printf("\nEnd of computation");
}
```

Output

```
Enter FIVE real values in a LINE
50.70 40 -36 75 11.25
50.750000      7.123903
40.000000      6.324555
Value -3 is negative
75.000000      8.660254
11.250000      3.354102
End of computation
```

Fig. 2.59 Use of the `goto` statement

The program in Fig. 2.59 evaluates the square root for five numbers. The variable `count` keeps the count of numbers read. When `count` is less than or equal to 5, `goto read;` directs the control to the label `read`; otherwise, the program prints a message and stops.

2.22 CASE STUDIES

1. Range of Numbers

Problem

A survey of the computer market shows that personal computers are sold at varying costs by the vendors. The following is the list of costs (in hundreds) quoted by some vendors:

35.00, 40.50, 25.00, 31.25, 68.15,
47.00, 26.65, 29.00 53.45, 62.50

Determine the average cost and the range of values.

Problem analysis

Range is one of the measures of dispersion used in statistical analysis of a series of values. The range of any series is the difference between the highest and the lowest values in the series. That is

$$\text{Range} = \text{highest value} - \text{lowest value}$$

It is therefore necessary to find the highest and the lowest values in the series.

Program: A program to determine the range of values and the average cost of a personal computer in the market is given in Fig. 2.60.

```

Program
main()
{
    int count;
    float value, high, low, sum, average, range;
    sum = 0;
    count = 0;
    printf("Enter numbers in a line : input a NEGATIVE number to end\n");
input:
    scanf("%f", &value);
    if (value < 0) goto output;
    count = count + 1;
    if (count == 1)
        high = low = value;
    else if (value > high)
        high = value;
    else if (value < low)
        low = value;
    sum = sum + value;
    goto input;
output:
    average = sum/count;
    range = high - low;
    printf("\n\n");
    printf("Total values : %d\n", count);
}

```

```

        printf("Highest-value: %f\nLowest-value : %f\n",
               high, low);
        printf("Range      : %f\nAverage : %f\n",
               range, average);
    }
Output
Enter numbers in a line : input a NEGATIVE number to end
35 40.50 25 31.25 68.15 47 26.65 29 53.45 62.50 -1
Total values : 10
Highest-value : 68.150002
Lowest-value : 25.000000
Range : 43.150002
Average : 41.849998

```

Fig. 2.60 Calculation of range of values

When the value is read the first time, it is assigned to two buckets, high and low, through the statement

```
high = low = value;
```

For subsequent values, the value read is compared with high; if it is larger, the value is assigned to high. Otherwise, the value is compared with low; if it is smaller, the value is assigned to low. Note that at a given point, the buckets high and low hold the highest and the lowest values read so far.

The values are read in an input loop created by the goto input; statement. The control is transferred out of the loop by inputting a negative number. This is caused by the following statement:

```
if (value < 0) goto output;
```

2. Pay-Bill Calculations

Problem

A manufacturing company has classified its executives into four levels for the benefit of certain perks. The levels and corresponding perks are shown below:

Level	Perks	
	Conveyance allowance	Entertainment allowance
1	1000	500
2	750	200
3	500	100
4	250	—

An executive's gross salary includes basic pay, house rent allowance at 25% of basic pay and other perks. Income tax is withheld from the salary on a percentage basis as follows:

Gross salary	Tax rate
Gross <= 2000	No tax deduction
2000 < Gross <= 4000	3%
4000 < Gross <= 5000	5%
Gross > 5000	8%

Write a program that will read an executive's job number, level number, and basic pay and then compute the net salary after withholding income tax.

Problem analysis

Gross salary = basic pay + house rent allowance + perks

Net salary = Gross salary – income tax.

The computation of perks depends on the level, while the income tax depends on the gross salary. The major steps are:

1. Read data.
2. Decide level number and calculate perks.
3. Calculate gross salary.
4. Calculate income tax.
5. Compute net salary.
6. Print the results.

Program A program and the results of the test data are given in Fig. 2.61. Note that the last statement should be an executable statement. That is, the label stop: cannot be the last line.

```

Program
#define CA1 1000
#define CA2 750
#define CA3 500
#define CA4 250
#define EA1 500
#define EA2 200
#define EA3 100
#define EA4 0
main()
{
    int level, jobnumber;
    float gross,
        basic,
        house_rent,
        perks,
        net,
        incometax;
input:
    printf("\nEnter level, job number, and basic pay\n");
    printf("Enter 0 (zero) for level to END\n\n");
    scanf("%d", &level);
    if (level == 0) goto stop;

```

```

scanf("%d %f", &jobnumber, &basic);
switch (level)
{
    case 1:
        perks = CA1 + EA1;
        break;
    case 2:
        perks = CA2 + EA2;
        break;
    case 3:
        perks = CA3 + EA3;
        break;
    case 4:
        perks = CA4 + EA4;
        break;
    default:
        printf("Error in level code\n");
        goto stop;
}
house_rent = 0.25 * basic;
gross = basic + house_rent + perks;
if (gross <= 2000)
    incometax = 0;
else if (gross <= 4000)
    incometax = 0.03 * gross;
else if (gross <= 5000)
    incometax = 0.05 * gross;
else
    incometax = 0.08 * gross;
net = gross - incometax;
printf("%d %d %.2f\n", level, jobnumber, net);
goto input;
stop: printf("\n\nEND OF THE PROGRAM");
}

```

Output

```

Enter level, job number, and basic pay
Enter 0 (zero) for level to END
1 1111 4000
1 1111 5980.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
2 2222 3000
2 2222 4465.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
3 3333 2000
3 3333 3007.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
4 4444 1000
4 4444 1500.00

```

```

Enter level, job number, and basic pay
Enter 0 (zero) for level to END
0
END OF THE PROGRAM

```

Fig. 2.61 Pay-bill calculations

2.23 DECISION MAKING AND LOOPING

Looping or iterative statements are used for running a particular set of code for any number of times. Iterative statements consist two parts—head and body. The head of the statement decides the number of times for which the instructions present in the body of the statement are to be executed.

In looping, a sequence of statements are executed until some conditions for termination of the loop are satisfied. Depending on the position of the control statement in the loop, a control structure may be classified either as the entry-controlled loop or as the exit-controlled loop.

The flow charts in Fig. 2.62 illustrate these structures. In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and, therefore, the body is executed unconditionally for the first time. The entry-controlled and exit-controlled loops are also known as pre-test and post-test loops, respectively.

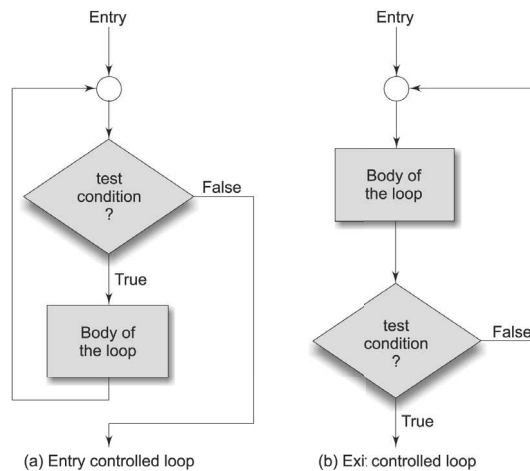


Fig. 2.62 Loop control structures

The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the test condition will eventually transfer the control out of the loop. In case, due to some reason it does not do so, the control sets up an infinite loop and the body is executed over and over again.

A looping process, in general, would include the following four steps:

1. Setting and initialisation of a condition variable.
2. Execution of the statements in the loop.
3. Test for a specified value of the condition variable for execution of the loop.
4. Incrementing or updating the condition variable.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.

The C language provides for three constructs for performing loop operations. They are:

1. The `while` statement.
2. The `do` statement.
3. The `for` statement.

2.23.1 The while Statement

The simplest of all the looping structures in C is the `while` statement. We have used `while` in many of our earlier programs. The basic format of the `while` statement is

```
while (test condition)
{
    body of the loop
}
```

The `while` is an entry-controlled loop statement. The test-condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

An example of `while` statement, which uses the keyboard input, is shown as here:

```
=====
character = ' ';
while (character != 'Y')
    character = getchar();
xxxxxxx;
=====
```

First the character is initialized to ' '. The `while` statement then begins by testing whether character is not equal to Y. Since the character was initialized to ' ', the test is true and the loop statement `character = getchar();` is executed. Each time a letter is keyed in, the test is carried out and the loop statement is executed until the letter Y is pressed.

When Y is pressed, the condition becomes false because character equals Y, and the loop terminates, thus transferring the control to the statement `xxxxxxx;`.

Example 2.23 A program to evaluate the equation $y = x^n$, where n is a non-negative integer, is given in Fig. 2.63.

```

Program
main()
{
    int count, n;
    float x, y;
    printf("Enter the values of x and n : ");
    scanf("%f %d", &x, &n);
    y = 1.0;
    count = 1;          /* Initialisation */
    /* LOOP BEGINS */
    while ( count <= n) /* Testing */
    {
        y = y*x;
        count++;        /* Incrementing */
    }
    /* END OF LOOP */
    printf("\nx = %f; n = %d; x to power n = %f\n",x,n,y);
}

```

Output

```

Enter the values of x and n : 2.5 4
x = 2.500000; n = 4; x to power n = 39.062500
Enter the values of x and n : 0.5 4
x = 0.500000; n = 4; x to power n = 0.062500

```

Fig. 2.63 Program to compute x to the power n using while loop

In this program, the variable `y` is initialised to 1 and then multiplied by `x`, `n` times using the `while` loop. The loop control variable `count` is initialised outside the loop and incremented inside the loop. When the value of `count` becomes greater than `n`, the control exists the loop.

2.23.2 The do Statement

The while loop construct, which we have discussed in the previous section, makes a test of condition before the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions, it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the do statement. This takes the following form:

```

do
{
    body of the loop
}
while (test-condition);

```

On reaching the `do` statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the `test-condition` in the `while` statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the `while` statement.

Since the test-condition is evaluated at the bottom of the loop, the `do...while` construct provides an exit-controlled loop and, therefore, the body of the loop is always executed at least once.

Consider the following example:

```

-----
I = 1;
/*Initializing */
sum = 0;
do
{
    sum = sum + I;
    I = I+2;                /* Incrementing */
}
while(sum < 40 || I < 10);  /* Testing */
printf("%d %d\n", I, sum);
-----

```

This loop will be executed as long as one of the two relations is true.

Example 2.24 A program to print the multiplication table from 1 * 1 to 12 * 10 is given in Fig. 2.64.

```

Program:
#define COLMAX 10
#define ROWMAX 12
main()
{
    int row,column, y;
    row = 1;
    printf(" MULTIPLICATION TABLE \n");
    do /*.....OUTER LOOP BEGINS.....*/
    {
        column = 1;
        do /*.....INNER LOOP BEGINS.....*/
        {
            y = row * column;
            printf("%4d", y);
            column = column + 1;
        }

        while (column <= COLMAX); /*... INNER LOOP ENDS ...*/
        printf("\n");
        row = row + 1;
    }
    while (row <= ROWMAX); /*..... OUTER LOOP ENDS .....*/
}

```

Output									
MULTIPLICATION TABLE									
1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100
11	22	33	44	55	66	77	88	99	110
12	24	36	48	60	72	84	96	108	120

Fig. 2.64 Printing of a multiplication table using do...while loop.

Here, the program contains two `do...while` loops in nested form. The outer loop is controlled by the variable `row` and executed 12 times. The inner loop is controlled by the variable `column` and is executed 10 times, each time the outer loop is executed. That is, the inner loop is executed a total of 120 times, each time printing a value in the table.

Notice that the `printf` of the inner loop does not contain any new line character (`\n`). This allows the printing of all row values in one line. The empty `printf` in the outer loop initiates a new line to print the next row.

2.23.3 The for Statement

Simple 'for' Loops

The `for` loop is another entry-controlled loop that provides a more concise loop control structure. The general form of the `for` loop is

```
for ( initialization ; test-condition ; increment )
{
    body of the loop
}
```

The execution of the `for` statement is as follows:

1. Initialisation of the control variables is done first, using assignment statements such as `i = 1` and `count = 0`. The variables `i` and `count` are known as loop-control variables.
2. The value of the control variable is tested using the `test-condition`. The `test-condition` is a relational expression, such as `i < 10` that determines when the loop will exit. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the `for` statement after evaluating the last statement in the loop. Now, the control variable is incremented using an assignment statement such as `i = i+1` and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the `test-condition`.

Consider the following segment of a program:

```
for ( x = 0 ; x <= 9 ; x = x+1 )
{
    printf("%d", x);
}
printf("\n");
```

This `for` loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the increment section, `x = x+1`.

The `for` statement allows for negative increments as well. For example, the loop discussed earlier can be written as follows:

```
for ( x = 9 ; x >= 0 ; x = x-1 )
    printf("%d", x);
printf("\n");
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example, the following loop will never be executed because the test condition fails at the very beginning itself.

```
for ( x = 9; x < 9; x = x-1 )
    printf("%d", x);
```

One of the important points about the `for` loop is that all the three actions, namely initialisation, testing and incrementing, are placed in the `for` statement itself, thus making them visible to the programmers and users, in one place. The `for` statement and its equivalent of `while` and `do` statements are shown in Table 2.24.

TABLE 2.24 Comparison of the three loops

for	while	do
for ($n=1$; $n \leq 10$; $++n$)	$n = 1$;	$n = 1$;
{	while ($n \leq 10$)	do
_____	{	{
_____	_____	_____
}	_____	_____
	$n = n+1$;	$n = n+1$;
	}	}
		while($n \leq 10$);

Example 2.25 The program in Fig. 2.65 uses a *for* loop to print the “Powers of 2” table for the power 0 to 20, both positive and negative.

```

Program
main()
{
    long int p;
    int n;
    double q;
    printf("-----\n");
    printf(" 2 to power n n 2 to power -n\n");
    printf("-----\n");
    p = 1;
    for (n = 0; n < 21 ; ++n) /* LOOP BEGINS */
    {
        if (n == 0)
            p = 1;
        else
            p = p * 2;
        q = 1.0/(double)p ;
        printf("%10ld %10d %20.12lf\n", p, n, q);
    } /* LOOP ENDS */
    printf("-----\n");
}

```

Output

```

-----
 2 to power n   n   2 to power -n
-----
      1         0   1.000000000000
      2         1   0.500000000000
      4         2   0.250000000000
      8         3   0.125000000000
     16         4   0.062500000000
     32         5   0.031250000000
     64         6   0.015625000000
    128         7   0.007812500000
    256         8   0.003906250000
    512         9   0.001953125000
   1024        10   0.000976562500
   2048        11   0.000488281250
   4096        12   0.000244140625
   8192        13   0.000122070313
  16384        14   0.000061035156
  32768        15   0.000030517578
  65536        16   0.000015258789
 131072        17   0.000007629395
 262144        18   0.000003814697
 524288        19   0.000001907349
1048576        20   0.000000953674
-----

```

Fig. 2.65 Program to print 'Power of 2' table using for loop

Additional Features of for Loop

The for loop in C has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialised at a time in the for statement, as shown further:

```
for (p=1, n=0; n<17; ++n)
```

Note that the initialisation section has two parts $p = 1$ and $n = 1$ separated by a comma.

Like the initialisation section, the increment section may also have more than one part. The multiple arguments in the increment section are separated by commas. For example:

```
for (n=1, m=50; n<=m; n=n+1, m=m-1)
{
    p = m/n;
    printf("%d %d %d\n", n, m, p);
}
```

Further, the test-condition may have any compound relation and the testing need not be limited only to the loop control variable. For example,

```
sum = 0;
for (i = 1; i < 20 && sum < 100; ++i)
{
    sum = sum+i;
    printf("%d %d\n", i, sum);
}
```

Here, the loop is executed as long as both the conditions $i < 20$ and $sum < 100$ are true. The `sum` is evaluated inside the loop.

It is also permissible to use expressions in the assignment statements of initialisation and increment sections. For example,

```
for (x = (m+n)/2; x > 0; x = x/2)
```

Another unique aspect of for loop is that one or more sections can be omitted, if necessary. For instance, the following for declaration is perfectly valid:

```
____
m = 5;
for ( ; m != 100 ; )
{
    printf("%d\n", m);
    m = m+5;
}
____
```

2.106 Computer Programming

Here, both the initialisation and increment sections are omitted in the `for` statement. The initialisation has been done before the `for` statement and the control variable is incremented inside the loop. In such cases, the sections are left 'blank'. However, the semicolons separating the sections must remain. If the test-condition is not present, the `for` statement sets up an 'infinite' loop. Such loops can be broken using `break` or `goto` statements in the loop.

We can also set up time delay loops using the null statement as follows:

```
For ( j = 1000; j > 0; j = j-1)
    ;
```

This loop is executed 1000 times without producing any output; it simply causes a time delay. Notice that the body of the loop contains only a semicolon, known as a null statement.

Nesting of for Loops

Nesting of loops, that is, one `for` statement within another `for` statement, is allowed in C. For example, two loops can be nested as shown in Fig. 2.66:

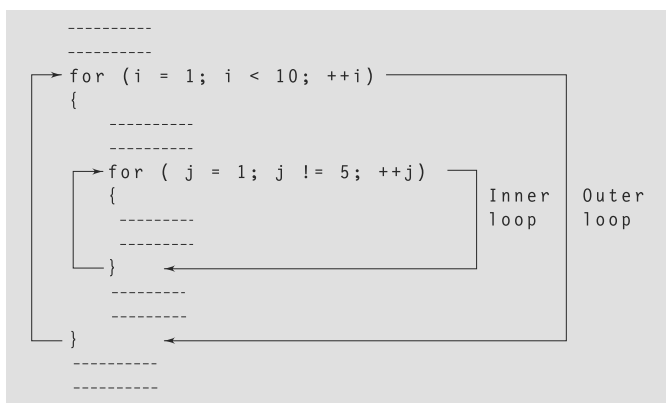


Fig. 2.66 Nesting of for loops

The nesting may continue up to any desired level. The loops should be properly indented so to enable the reader to easily determine which statements are contained within each `for` statement. (ANSI C allows up to 15 levels of nesting. However, some compilers permit more.)

A typical example of nesting of `for` loops is the program to print the multiplication table, as shown further:

```
for (row = 1; row <= ROWMAX ; ++row)
{
    for (column = 1; column <= COLMAX ; ++column)
    {

        y = row * column;
```



```

        printf("%4d", y);
    }
    printf("\n");
}

```

Here, the outer loop controls the rows while the inner loop controls the columns.

2.23.4 Jumps in Loops

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test-condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop, it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. For example, consider the case of searching for a particular name in a list containing, say, 100 names. A program loop written for reading and testing the names 100 times must be terminated as soon as the desired name is found. C permits a jump from one statement to another within a loop as well as a jump out of a loop.

Jumping Out of a Loop

An early exit from a loop can be accomplished by using the `break` statement or the `goto` statement. We have already seen the use of the `break` in the switch statement and the `goto` in the if...else construct. These statements can also be used within while, do or for loops. The use of `break` in loops is illustrated in Fig. 2.67.

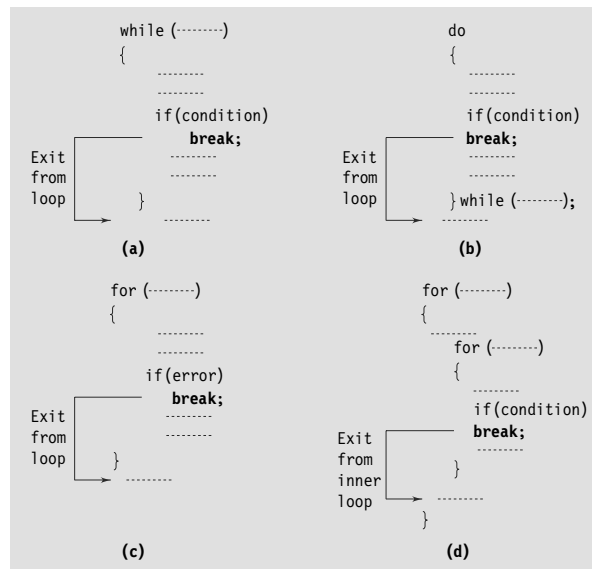


Fig. 2.67 Exiting a loop with `break` statement

When a `break` statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the `break` would only exit from the loop containing it. That is, the `break` will exit only a single loop.

Figure 2.68 shows the flowchart illustrating the use of break statement in a loop.

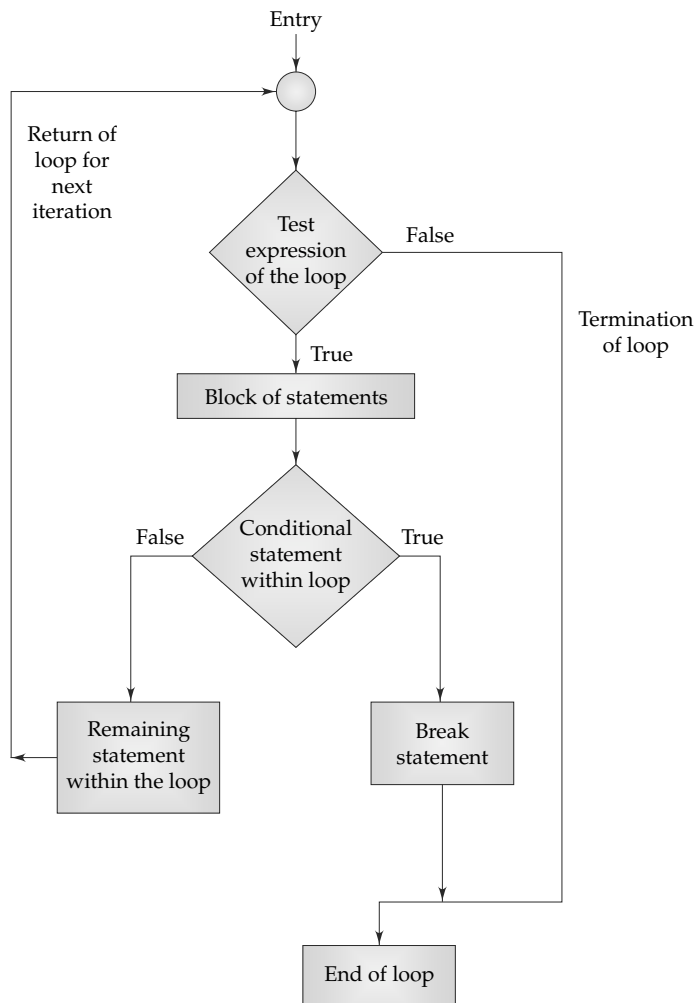


Fig. 2.68 Flowchart for exiting a loop with break statement

Since a `goto` statement can transfer the control to any place in a program, it is useful to provide branching within a loop. Another important use of `goto` is to exit from deeply nested loops when an error occurs. A simple `break` statement would not work here. Figure 4.69 shows the use of `goto` statement for jumping within or outside of a loop.

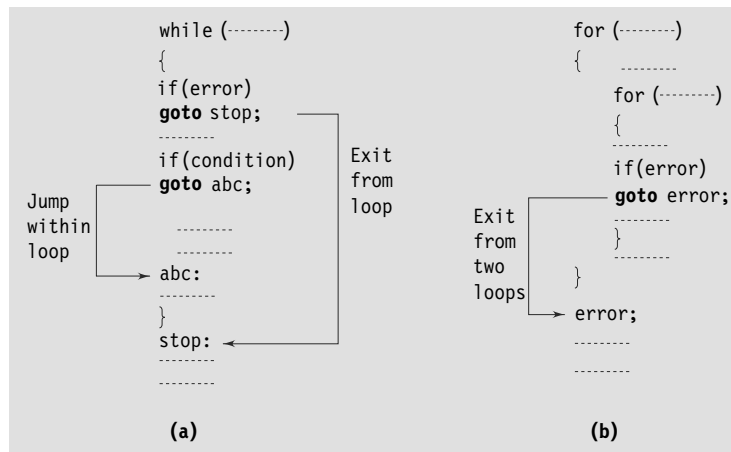


Fig. 2.69 *Jumping within and exiting from the loops with goto statement*

Figure 2.70 shows the flowchart illustrating the use of goto statement in a loop.

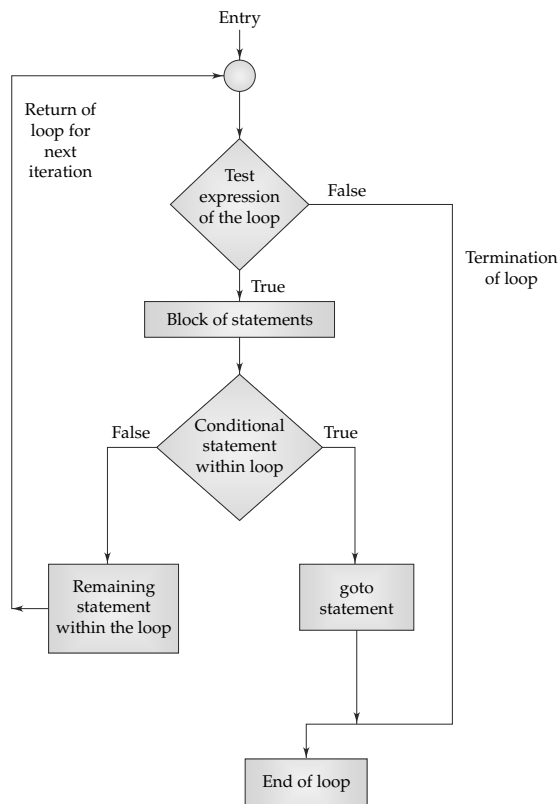


Fig. 2.70 *Flowchart for using goto statement in a loop*

Example 2.26 The program in Fig. 2.71 illustrates the use of the break statement in a C program.

```

Program
main()
{
    int m;
    float x, sum, average;
    printf("This program computes the average of a
        set of numbers\n");
    printf("Enter values one after another\n");
    printf("Enter a NEGATIVE number at the end.\n\n");
    sum = 0;
    for (m = 1 ; m <= 1000 ; ++m)
    {
        scanf("%f", &x);
        if (x < 0)
            break;
        sum += x ;
    }
    average = sum/(float)(m-1);
    printf("\n");

    printf("Number of values = %d\n", m-1);
    printf("Sum                = %f\n", sum);
    printf("Average            = %f\n", average);
}

Output
This program computes the average of a set of numbers
Enter values one after another
Enter a NEGATIVE number at the end.

21 23 24 22 26 22 -1

Number of values = 6
Sum              = 138.000000
Average          = 23.000000

```

Fig. 2.71 Use of break in a program.

The program reads in Fig. 2.71 a list of positive values and calculates their average. The for loop is written to read 1000 values. However, if we want the program to calculate the average of any set of values less than 1000, then we must enter a ‘negative’ number after the last value in the list, to mark the end of input.

Each value, when it is read, is tested to see whether it is a positive number or not. If it is positive, the value is added to the sum; otherwise, the loop terminates. On exit, the average of the values read is calculated and the results are printed out.

Skipping a Part of a Loop

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. For example, in processing of applications for some job, we might like to exclude the processing of data of applicants belonging to a certain category. On reading the category code of an applicant, a test is made to see

whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.

Like the `break` statement, C supports another similar statement called the `continue` statement. However, unlike the `break` that causes the loop to be terminated, the `continue`, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The `continue` statement tells the compiler, “SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION”. The format of the `continue` statement is simple.

```
continue;
```

The use of the `continue` statement in loops is illustrated in Fig. 2.72.

```
while (test-condition)
{
    -----
    if (-----)
        continue;
    -----
    -----
}
```

(a)

```
do
{
    -----
    if (-----)
        continue;
    -----
    -----
}while(test-condition);
```

(b)

```
for (initialization; test condition; increment)
{
    -----
    if (-----)
        continue;
    -----
    -----
}
```

(c)

Fig. 2.72 *Bypassing and continuing in loops.*

Example 2.27 The program shown in Fig. 2.73 illustrates the use of `continue` statement.

```
Program:
#include<stdio.h>
#include<conio.h>
void main()
{
    int x,y;
    clrscr();
    for(x=1;x<=3;x++)
    {
        for(y=1;y<=3;y++)
```

```

        {
            if(x==y)
                continue;
            printf("\n%d\t%d\t",x,y);
        }
    }
    getch();
}

```

Output:

```

1      2
1      3
2      1
2      3
3      1
3      2

```

Fig. 2.73 Use of continue statement

In this above code, two variables, x and y , are taken as an integer data type. Here, when the value of x is equal to y , the control of the program is transferred to the inner for loop using the continue statement.

2.24 JUMPING OUT OF THE PROGRAM

We have just seen that we can jump out of a loop using either the `break` statement or `goto` statement. In a similar way, we can jump out of a program by using the library function `exit()`. In case, due to some reason, we wish to break out of a program and return to the operating system, we can use the `exit()` function, as shown follows:

```

.....
.....
if (test-condition)
exit(0);
.....
.....

```

The `exit()` function takes an integer value as its argument. Normally zero is used to indicate normal termination and a nonzero value to indicate termination due to some error or abnormal condition. The use of `exit()` function requires the inclusion of the header file `<stdlib.h>`.

2.25 STRUCTURED PROGRAMMING

Structured programming is an approach to the design and development of programs. It is a discipline of making a program's logic easy to understand by using only the basic three control structures:

- Sequence (straight line) structure
- Selection (branching) structure
- Repetition (looping) structure

While sequence and loop structures are sufficient to meet all the requirements of programming, the selection structure proves to be more convenient in some situations. The use of structured programming techniques helps ensure well-designed programs that are easier to write, read, debug and maintain compared to those that are unstructured.

Structured programming discourages the implementation of unconditional branching using jump statements such as goto, break and continue. In its purest form, structured programming is synonymous with “goto less programming”. Do not go to goto statement!

2.26 CASE STUDIES

1. Table of Binomial Coefficients

Problem

Binomial coefficients are used in the study of binomial distributions and reliability of multicomponent redundant systems. It is given by

$$B(m,x) = \binom{m}{x} = \frac{m!}{x!(m-x)!}, m \geq x$$

A table of binomial coefficients is required to determine the binomial coefficient for any set of m and x.

Problem Analysis:

The binomial coefficient can be recursively calculated as follows:

$$B(m,0) = 1$$

$$B(m,x) = B(m,x-1) \left[\frac{m-x+1}{x} \right], x = 1,2,3,\dots,m$$

Further,

$$B(0,0) = 1$$

That is, the binomial coefficient is one when either x is zero or m is zero. The program in Fig. 2.74 prints the table of binomial coefficients for m = 10. The program employs one do loop and one while loop.

```

Program
#define MAX 10
main()
{
    int m, x, binom;
    printf(" m x");
    for (m = 0; m <= 10 ; ++m)
        printf("%4d", m);
    printf("\n-----\n");
    m = 0;
    do
    {
        printf("%2d ", m);
        x = 0; binom = 1;
        while (x <= m)
        {
            if(m == 0 || x == 0)
                printf("%4d", binom);
            else

```

```

    {
        binom = binom * (m - x + 1)/x;
        printf("%4d", binom);
    }
    x = x + 1;
}
printf("\n");
m = m + 1;
}
while (m <= MAX);
printf("-----\n");
}

```

Output

mx	0	1	2	3	4	5	6	7	8	9	10

0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

Fig. 2.74 Program to print binomial coefficient table

2. Minimum Cost

Problem

The cost of operation of a unit consists of two components C1 and C2 which can be expressed as functions of a parameter p as follows:

$$C1 = 30 - 8p$$

$$C2 = 10 + p^2$$

The parameter p ranges from 0 to 10. Determine the value of p with an accuracy of + 0.1 where the cost of operation would be minimum.

Problem Analysis

$$\text{Total cost} = C1 + C2 = 40 - 8p + p^2$$

The cost is 40 when p = 0, and 33 when p = 1 and 60 when p = 10. The cost, therefore, decreases first and then increases. The program in Fig. 2.75 evaluates the cost at successive intervals of p (in steps of 0.1) and stops when the cost begins to increase. The program employs break and continue statements to exit the loop.


```

Program
main()
{
    float p, cost, p1, cost1;
    for (p = 0; p <= 10; p = p + 0.1)
    {
        cost = 40 - 8 * p + p * p;
        if(p == 0)
        {
            cost1 = cost;
            continue;
        }
        if (cost >= cost1)
            break;
        cost1 = cost;
        p1 = p;
    }
    p = (p + p1)/2.0;
    cost = 40 - 8 * p + p * p;
    printf("\nMINIMUM COST = %.2f AT p = %.1f\n", cost, p);
}
Output
MINIMUM COST = 24.00 AT p = 4.0

```

Fig. 2.75 Program of minimum cost problem

Example 2.28 Write a program for demonstrating the use of symbolic constants.

Program

```

/*Program for demonstrating use of Symbolic Constants*/
#include <stdio.h>
#include <conio.h>
#define CP 50 /*Defining Symbolic Constant for Cost Price*/

void main( )
{
    int SP, profit;
    clrscr();

    printf("Enter the SELLING PRICE ");
    scanf("%d", &SP);

    profit=SP-CP; /*Using Symbolic Constant CP in an expression*/
    printf("\nThe profit earned is %d", profit);

    getch();
}

```

Output

Enter the SELLING PRICE 65

The profit earned is 15

Example 2.29 Write a program that reads a date in the dd\mm\yyyy format and determines whether the entered date is correct or not.

Program

```
/*Program for demonstrating the use of scanf function*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int dd,mm,yyyy;
    clrscr();

    printf("Enter the date in dd mm yyyy format ");
    scanf("%d %d %d",&dd,&mm,&yyyy); /*Reading the date values*/

    /*Checking whether day, month, and year is correct */
    if(dd<1 || dd>31 || mm<1 || mm>12 || yyyy<0 || (mm==2 && dd>29))
        printf("NO");
    else
        printf("YES");

    getch();
}
```

Output

Enter the date in dd mm yyyy format 22 09 2001
YES

Example 2.30 Write a program to determine whether an input number is even or odd.

Program

```
/*Program for demonstrating use of Modulo Division Operator %*/  
#include <stdio.h>  
#include <conio.h>  
  
void main()  
{  
    int num;  
    clrscr();  
  
    printf("\nEnter a number: ");  
    scanf("%d",&num);/*Reading an Integer*/  
  
    if(num%2==0)/*Using % operator to compute the remainder value*/  
        printf("\n%d is an even number",num);  
    else  
        printf("\n%d is an odd number",num);  
  
    getch();  
}
```

Output

```
Enter a number: 55  
  
55 is an odd number
```

Example 2.31 Write a program to convert degrees value into radians.

Program

```
/*Program for demonstrating real arithmetic*/  
#include <stdio.h>  
#include <conio.h>  
  
void main( )  
{  
    float deg, radian;  
    clrscr();  
  
    printf("Enter the degrees value ");
```

2.118 Computer Programming

```
scanf("%f", &deg);

radian=(deg*3.14)/180.00; /*Using a formula for converting degrees to radians*/
printf("\nThe equivalent value in radians is %.2f", radian);

getch();
}
```

Output

```
Enter the degrees value 120

The equivalent value in radians is 2.09
```

Example 2.32 Write a program for calculating speed.

Program

```
/*Program for demonstrating mixed-mode arithmetic*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int d;
    float t;
    clrscr();
    /*Reading the values of distance and time*/
    printf("\nEnter the distance travelled in Kms: ");
    scanf("%d", &d);
    printf("\nEnter the travel time in hours: ");
    scanf("%f", &t);

    printf("\nSpeed = %.2f Km/h", d/t);/*Mixed-mode arithmetic expression*/
    getch();
}
```

Output

```
Enter the distance travelled in Kms: 380
Enter the travel time in hours: 5.5
Speed = 69.09 Km/h
```

Example 2.33 Write a program for evaluating the logical expression $(10/2-4)\&\&(7\%3)\| (0/10)$.

Program

```
/*Program for demonstrating the use of logical operators*/
#include <stdio.h>
#include <conio.h>

void main()
{
    clrscr();

    if((10/2-4)&&(7%3)|| (0/10))/*Evaluating a logical expression*/
        printf("True");
    else
        printf("False");

    getch();
}
```

Output

```
True
```

Example 2.34 Write a program to demonstrate the use of increment operator.

Program

```
/*Program for demonstrating the use of increment operator*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int a=10;
    clrscr();

    /*Using Increment Operator*/
    printf("a = %d",a);
    printf("\n++a = %d",++a);
    printf("\na = %d",a);
}
```

2.120 Computer Programming

```
printf("\na++ = %d",a++);
printf("\na = %d",a);

getch();
}
```

Output

```
a = 10
++a = 11
a = 11
a++ = 11
a = 12
```

Example 2.35 Write a program to perform bitwise AND and OR operations.

Program

```
/*Program for demonstrating the use of bitwise operators*/
#include <stdio.h>
#include <conio.h>

void main()
{
    char a='A',b='B';
    clrscr();

    printf("a & b = %c",a&b);/*Using bitwise & operator*/
    printf("\na | b = %c",a|b);/*Using bitwise | operator*/

    getch();
}
```

Output

```
a & b = @
a | b = C
```

Example 2.36 Write a program to demonstrate the use of sizeof operator.

```

Program
/*Program for demonstrating the use of sizeof operator*/
#include <stdio.h>
#include <conio.h>

void main()
{
    char a;
    int b;
    long c;
    float d;
    clrscr();

    /*Using the sizeof operator to compute the size of different datatypes*/
    printf("Size of char a = %u",sizeof(a));
    printf("\nSize of int b = %u",sizeof(b));
    printf("\nSize of long c = %u",sizeof(c));
    printf("\nSize of float d = %u",sizeof(d));
    getch();
}

```

Output

```

Size of char a = 1
Size of int b = 2
Size of long c = 4
Size of float d = 4

```

Example 2.37 Write a program to solve the equation $x = a - b/3 + c*2 - 1$.

Program

```

/*Program for demonstrating operator precedence in expression evaluation*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int a=9,b=12,c=3;
    int x;
    clrscr();

```

2.122 Computer Programming

```
x=a-b/3+c*2-1;/*Expression evaluation*/
printf("Value of x = %d",x);/*Printing the resultant value x*/

getch();
}
```

Output

```
Value of x = 10
```

Example 2.38 Write a program to evaluate the expression $A/B*C+D/A$.

Program

```
/*Program for demonstrating operator precedence in expression evaluation*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int A,B,C,D;
    int RESULT;
    clrscr();
    printf("Enter the value of A, B, C and D: ");
    scanf("%d %d %d %d",&A, &B, &C, &D);/*Reading operand values*/

    RESULT= A - B * C + D * A; /*Expression evaluation*/
    printf("RESULT = %d",RESULT);/*Printing the resultant value*/

    getch();
}
```

Output

```
Enter the value of A, B, C and D: 22
5
65
9
RESULT = -105
```

Example 2.39 Write a program to compute logarithmic values.

Program

```

/*Program for demonstrating the use of built-in functions of C*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

void main()
{
    int x;
    float l1,l2;
    clrscr();
    printf("Enter the value of x: ");
    scanf("%d",&x);/*Reading x*/

    l1=log(x);/*Computing natural log*/
    l2=log10(x); /*Computing log to the base 10*/
    printf("log(x) = %.2f\nlog10(x) = %.2f",l1,l2);/*Printing the resultant values*/

    getch();
}

```

Output

```

Enter the value of x: 10
log(x) = 2.30
log10(x) = 1.00

```

Example 2.40 Write a program to compute x^y .

Program

```

/*Program for demonstrating the use of built-in function pow(x,y)*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

void main()
{
    int x,y;
    long z;
    clrscr();

```

2.124 Computer Programming

```
printf("Enter the value of x: ");
scanf("%d",&x);/*Reading the value of x*/
printf("Enter the value of y: ");
scanf("%d",&y);/*Reading the value of y*/

z=pow(x,y);/*Calling the built-in function pow()*/

printf("%d to the power of %d is equal to %ld", x, y, z);

getch();
}
```

Output

```
Enter the value of x: 2
Enter the value of y: 16
2 to the power of 16 is equal to 65536
```

Example 2.41 Write a program for calculating sum of all the numbers between 3 and 20 excluding the multiples of 3.

Program

```
/*Program to demonstrate the use IF Statement*/
#include<stdio.h>
#include<conio.h>

void main()
{
    int i,sum=0;
    clrscr();

    for (i=3;i<20;i++)/*Specifying the looping condition*/
    {
        if(i%3==0)/*Identifying multiples of 3*/
            continue;
        sum=sum+i;
    }

    printf("\nSum of numbers between 3 to 20 excluding multiples of 3 = %d", sum);
    getch();
}
```

Output

```
Sum of numbers between 3 to 20 excluding multiples of 3 = 124
```

Example 2.42 Write a program to determine whether or not a given year is a leap year.

Program

```
/*Program to demonstrate the use IF.....ELSE Statement*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int year;
    clrscr();

    printf("\nEnter the year value: ");
    scanf("%d",&year);/*Reading the year value*/

    if(year%4==0)/*Determining whether year is a multiple of 4*/
        printf("\n\n%d is a leap year",year);
    else
        printf("\n%d is not a leap year",year);

    getch();
}
```

Output

```
Enter the year value: 2005
```

```
2005 is not a leap year
```

Example 2.43 Write a program to determine whether a given value is positive or negative.

Program

```
/*Program for demonstrating the use of IF....ELSE Statement*/
#include <stdio.h>
#include <conio.h>
```

```

void main()
{
    int num;
    clrscr();

    printf("\nEnter a number: ");
    scanf("%d",&num);/*Reading a number*/

    if(num>0)/*Checking positive values*/
        printf("\n%d is a positive number", num);
    else
        printf("\n%d is a negative number", num);

    getch();
}

```

Output

```

Enter a number: 12

12 is a positive number

```

Example 2.44 Write a program to determine the frequency of occurrence of individual digits in a number.

Program

```

/*Program for demonstrating the use of Switch Statement*/
#include <stdio.h>
#include <conio.h>

void main()
{
    long int num1,num2;
    int temp,d1=0,d2=0,d3=0,d4=0,d5=0,d6=0,d7=0,d8=0,d9=0,d0=0;
    clrscr();

    printf("\nEnter the number:");
    scanf("%ld",&num1);

    num2=num1;
    while(num1!=0)

```

```
{
    temp=num1%10;
    switch(temp) /*Using the switch-case decision making construct*/
    {
        case 0:
            d0++; /*Counting number of Zeros*/
            break;
        case 1:
            d1++; /*Counting number of Ones*/
            break;
        case 2:
            d2++; /*Counting number of Twos*/
            break;
        case 3:
            d3++; /*Counting number of Threes*/
            break;
        case 4:
            d4++; /*Counting number of Fours*/
            break;
        case 5:
            d5++; /*Counting number of Fives*/
            break;
        case 6:
            d6++; /*Counting number of Sixes*/
            break;
        case 7:
            d7++; /*Counting number of Sevens*/
            break;
        case 8:
            d8++; /*Counting number of Eights*/
            break;
        case 9:
            d9++; /*Counting number of Nines*/
            break;
    }
    num1=num1/10;
}

/*Displaying the frequency of individual digits in a number*/
printf("\nThe no of 0s in %ld are %d",num2,d0);
printf("\nThe no of 1s in %ld are %d",num2,d1);
printf("\nThe no of 2s in %ld are %d",num2,d2);
printf("\nThe no of 3s in %ld are %d",num2,d3);
```

2.128 Computer Programming

```
printf("\nThe no of 4s in %ld are %d",num2,d4);
printf("\nThe no of 5s in %ld are %d",num2,d5);
printf("\nThe no of 6s in %ld are %d",num2,d6);
printf("\nThe no of 7s in %ld are %d",num2,d7);
printf("\nThe no of 8s in %ld are %d",num2,d8);
printf("\nThe no of 9s in %ld are %d",num2,d9);

getch();
}
```

Output

```
Enter the number:889653442

The no of 0s in 889653442 are 0
The no of 1s in 889653442 are 0
The no of 2s in 889653442 are 1
The no of 3s in 889653442 are 1
The no of 4s in 889653442 are 2
The no of 5s in 889653442 are 1
The no of 6s in 889653442 are 1
The no of 7s in 889653442 are 0
The no of 8s in 889653442 are 2
The no of 9s in 889653442 are 1
```

Example 2.45 Write a program to implement a simple arithmetic calculator.

Program

```
/*Program for demonstrating the use of Switch statement*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int choice;
    float num1, num2;
    clrscr();
    printf("*****Simple Calc*****");/*Displaying Calc options*/
    printf("\n\nChoose a type of operation from the following: ");
    printf("\n\t1.  Addition");
```

```

printf("\n\t2.   Subtraction");
printf("\n\t3.   Multiplication");
printf("\n\t4.   Division\n");
scanf("%d", &choice); /*Reading user's choice*/
printf("\n\nEnter the two operands: ");
scanf("%f %f", &num1, & num2); /*Reading operands*/

/*Using the Switch statement to choose the operation statement*/
switch (choice)
{
case 1:
    printf("\n%.2f + %.2f = %.2lf", num1, num2, num1+num2);
    break;

case 2:
    printf("\n%.2f - %.2f = %.2lf", num1, num2, num1-num2);
    break;

case 3:
    printf("\n%.2f * %.2f = %.2lf", num1, num2, num1*num2);
    break;

case 4:
    printf("\n%.2f / %.2f = %.2lf", num1, num2, num1/num2);
    break;

default:
    printf("\nIncorrect Choice!");
}

getch();
}

```

Output

```
*****Simple Calc*****
```

Choose a type of operation from the following:

1. Addition
2. Subtraction
3. Multiplication
4. Division

2.130 Computer Programming

3

Enter the two operands: 66

6

66.00 * 6.00 = 396.00

Example 2.46 Write a program to calculate the sum of digits of an integer.

Program

```
/*Program for demonstrating the use of while statement*/
#include <stdio.h>
#include <conio.h>

void main()
{
    long num, temp;
    int sum=0;
    clrscr();

    printf("\nEnter an integer value: ");
    scanf("%ld",&num);/*Reading a long integer*/

    temp=num;
    /*Calculating sum of digits*/
    while(temp!=0)
    {
        sum = sum+temp%10;
        temp=temp/10;
    }

    printf("\n\nThe sum of digits of %ld is %d",num,sum);/*Displaying result*/
    getch();
}
```

Output

Enter an integer value: 56988

The sum of digits of 56988 is 36

Example 2.47 Write a program to determine whether or not a given number is an Armstrong.

Program

```
/*Program for demonstrating the use of while statement*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

void main()
{
    int num, temp, sum=0, i;
    clrscr();

    printf("\nEnter a number: ");
    scanf("%d", &num); /*Reading num*/
    temp=num;

    while(temp>0) /*Computing Armstrong value*/
    {
        i=temp%10;
        sum=sum+i*i*i;
        temp=temp/10;
    }

    if(sum==num) /*Checking whether num is Armstrong or not*/
        printf("\n%d is an Armstrong number",num);
    else
        printf("\n%d is not an Armstrong number",num);

    getch();
}
```

Output

```
Enter a number: 371

371 is an Armstrong number
```

Example 2.48 Write a program to determine the Greatest Common Divisor (GCD) of two numbers.

Program

```

/*Program for demonstrating the use of while statement*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

int GCD(int m, int n);/*Declaring the GCD procedure for computing GCD value*/
void main()
{
    int num1,num2;
    clrscr();

    printf("Enter the two numbers whose GCD is to be found: ");
    scanf("%d %d",&num1,&num2);/*Reading the two input numbers*/

    printf("\nGCD of %d and %d is %d\n",num1,num2,GCD(num1,num2));/*Calling the GCD
    procedure*/
    getch();
}

/*Defining the GCD procedure for computing the GCD value for two integer values*/
int GCD(int a, int b)
{
    if(b>a)
        return GCD(b,a);
    if(b==0)
        return a;
    else
        return GCD(b,a%b);
}

```

Output

```

Enter the two numbers whose GCD is to be found: 18
24

GCD of 18 and 24 is 6

```

Example 2.49 Write a program to display the Pascal's triangle.

Program

```
/*Program for demonstrating the use of while statement*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int b,row,x,y,z;
    clrscr();
    b=1;
    y=0;

    printf("Enter the number of rows for the Pascal's triangle:");
    scanf("%d",&row);

    printf("\n*****Pascal's Triangle*****\n");

    while(y<row)
    {
        for(x=40-3*y;x>0;--x)
            printf(" ");

        for(z=0;z<=y;++z)
        {
            if((z==0)||(y==0))
                b=1;
            else
                b=(b*(y-z+1))/z;
            printf("%6d",b);
        }
        printf("\n");
        ++y;
    }

    getch();
}
```

Output

Enter the number of rows for the Pascal's triangle:5

2.134 Computer Programming

*****Pascal's Triangle*****

```

          1
        1 1
      1 2 1
    1 3 3 1
  1 4 6 4 1
```

Example 2.50 Write a program to find the sum of the following series up to 50 terms:

$$-1^3 + 3^3 - 5^3 + 7^3 - 9^3 + 11^3 - \dots$$

Program

```
/*Program for demonstrating the use of while statement*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int b,row,x,y,z;
    clrscr();
    b=1;
    y=0;

    printf("Enter the number of rows for the Pascal's triangle:");
    scanf("%d",&row);

    printf("\n*****Pascal's Triangle*****\n");

    while(y<row)
    {
        for(x=40-3*y;x>0;--x)
            printf(" ");

        for(z=0;z<=y;++z)
        {
            if((z==0)|| (y==0))
                b=1;
            else
                b=(b*(y-z+1))/z;
            printf("%6d",b);
        }
    }
```

```

printf("\n");
++y;
}

getch();
}

```

Output

```
Sum of series = 499850
```

Example 2.51 Write a program to calculate the sum of N terms of the following series:

$$1^2+2^2+3^2+4^2+\dots n^2$$

Program

```

/*Program to demonstrate the use of for statement*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int i,n;
    long sum=0;
    clrscr();

    printf("Enter the value of n = ");
    scanf("%d",&n);/*Reading number of terms in the series, n*/

    for(i=1;i<=n;i++)/*Calculating the sum of the series*/
        sum=sum+i*i;

    printf("Sum of series = %ld",sum);

    getch();
}

```

Output

```
Enter the value of n = 5
Sum of series = 55
```

Example 2.52 Write a program to find the sum of the following series:

$$1 + x + x^2 + x^3 + \dots + x^n.$$

Program

```

/*Program for demonstrating the use of for statement*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

void main()
{

    long sum;
    int n,x,i;
    clrscr();

    printf("Enter the values of x and n:");
    scanf("%d %d",&x,&n);/*Reading values of x and n*/

    if(n<=0 || x<=0)/*Identifying incorrect values*/
    {
        printf("The values must be positive integers. Please try again\n");
        getch();
    }

    else
    {
        sum=1;
        for(i=1;i<=n;i++)
        {
            sum=sum+pow(x,i);/*Calculating the sum of the series*/
        }
        printf("Sum of series=%ld\n",sum);
    }
    getch();
}

```

Output

```

Enter the values of x and n:2
6
Sum of series=127

```

Example 2.53 Write a program to find the sum of the following series:

$$1 + 2 + 2^2 + 2^3 + \dots + 2^n.$$

Program

```
/*Program for demonstrating the use of for statement*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

void main()
{

    long sum;
    int n,i;
    clrscr();

    printf("Enter the value of n: ");
    scanf("%d",&n);/*Reading value of n*/

    sum=1;
    for(i=1;i<=n;i++)
    {
        sum=sum+pow(2,i);/*Calculating the sum of the series*/
    }
    printf("Sum of series = %ld\n",sum);

    getch();
}
```

Output

```
Enter the value of n: 5
Sum of series = 63
```

Example 2.54 Write a program to solve the following series:

$$1 + 1/2 + 1/3 + 1/4 + \dots + 1/n.$$

Program

```
/*Program for demonstrating the use of for statement*/
#include <stdio.h>
```

```
#include <conio.h>

void main()
{
    int n;
    float i;
    double sum;
    clrscr();

    printf("Enter the value of n: ");
    scanf("%d",&n);/*Reading the value of n*/

    sum = 1.0;
    for(i=2.0;i<=n;i++)/*Calculating the sum of the series*/
        sum = sum + 1.0/i;
    printf("\nThe sum of the series 1 + 1/2 + 1/3 +....+1/n = %.8lf",sum);

    getch();
}
```

Output

Enter the value of n: 8

The sum of the series 1 + 1/2 + 1/3 +....+1/n = 2.71785714

Example 2.55 Write a program to determine whether a given number is prime or not.

Program

```
/*Program for demonstrating the use of for statement*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int num,i;
    clrscr();

    printf("\nEnter a number: ");
    scanf("%d",&num);/*Reading a number*/
```



```

for(i=2;i<num;i++)
    if(num%i==0)/*Checking if num is divisible by another number*/
    {
        printf("\n%d is not a prime number",num);
        break;
    }
    else
        ;

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

    getch();
}

```

Output

```

Enter a number: 79

79 is a prime number

```

Example 2.56 Write a program for printing the Fibonacci series.

Program

```

/*Program for demonstrating the use of for statement*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int num1=0, num2=1,len,i,fab;
    clrscr();

    printf("\n\nEnter Length of the Fibonacci Series: ");
    scanf("%d",&len);/*Reading series length*/

    printf("\n<---FIBONACCI SERIES--->");
    printf("\n%d  %d",num1,num2);/*Printing initial values of the series*/

    /*Printing the Fibonacci series*/
    for(i = 1; i <= len-2; i++)

```

2.140 Computer Programming

```
{
    fab=num1 + num2;
    printf("  %d",fab);
    num1=num2;
    num2=fab;
}

getch();
}
```

Output

```
Enter Length of the Fibonacci Series: 6

<----FIBONACCI SERIES---->
0  1  1  2  3  5
```

Example 2.57 Write a program to calculate the sum of the following series:

$1^4 + 3^4 + 5^4 + \dots$ up to 100 terms.

Program

```
/*Program for demonstrating the use of for statement*/
#include<stdio.h>
#include<conio.h>
#include<math.h>

void main()
{
    int i;
    long sum=0;
    clrscr();
    for(i=1;i<=100;i=i+2)
        sum=sum+pow(i,4);/*Calculating the sum of the series*/

    printf("\nSum of the series up to first 100 terms is equal to %ld",sum);
    getch();
}
```

Output

```
Sum of the series up to first 100 terms is equal to 999666690
```

Example 2.58 Write a program for displaying a pyramid of the following form:

$$\begin{array}{cccccccc}
 & & 0 & & & & & \\
 & 1 & 0 & 1 & & & & \\
 & 2 & 1 & 0 & 1 & 2 & & \\
 & 3 & 2 & 1 & 0 & 1 & 2 & 3 \\
 & 4 & 3 & 2 & 1 & 0 & 1 & 2 & 3 & 4 \\
 5 & 4 & 3 & 2 & 1 & 0 & 1 & 2 & 3 & 4 & 5
 \end{array}$$

Program

```

/*Program for demonstrating nesting of for statements*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int num,i,y,x=40;
    clrscr();

    printf("\nEnter a number for \ngenerating the pyramid:\n");
    scanf("%d",&num);

    /*Displaying the pyramid*/
    for(y=0;y<=num;y++)
    {
        gotoxy(x,y+1);
        for(i=0-y;i<=y;i++)
            printf("%3d",abs(i));
        x=x-3;
    }

    getch();
}

```

Output

```

0
Enter a number for      1 0 1
generating the pyramid: 2 1 0 1 2
6                       3 2 1 0 1 2 3
                        4 3 2 1 0 1 2 3 4
                         5 4 3 2 1 0 1 2 3 4 5
                          6 5 4 3 2 1 0 1 2 3 4 5 6

```

Example 2.59 Write a program to calculate the sum of odd numbers between 1 and 50.

Program

```
/*Program to demonstrate the use For Statement*/
#include<stdio.h>
#include<conio.h>

void main()
{
    int i,sum=0;
    clrscr();

    for (i=1;i<50;i++)/*Specifying the looping condition*/
    {
        if(i%2==0)/*Identifying even numbers*/
            continue;
        sum=sum+i;/*Calculating the sum of odd numbers*/
    }

    printf("\nSum of odd numbers between 1 to 50 is = %d", sum);

    getch();
}
```

Output

```
Sum of odd numbers between 1 to 50 is = 625
```



Just Remember

- **Function:** It is a subroutine that may include one or more statements designed to perform a specific task.
- **Global variable:** It is a variable that can be used in more than one function.
- **Function body:** It is a part of a program that contains all the statements between the two braces, i.e. { and }.
- **Newline character:** It instructs the computer to go to the next (new) line.
- **Arguments:** Arguments are the values that are passed to a function as input.
- **Program:** A program contains a sequence of instructions written to perform a specific task.
- **Identifiers:** These are the names of variables, functions and arrays.
- **Constant:** It is a fixed value that does not change during the execution of a program.
- **String constant:** It is a sequence of characters enclosed in double quotes that represents a text string.

- **Variable:** It is a data name that may be used to store a data value.
- **Information:** The processed data generated by a program is called information.
- **Formatted input:** It refers to the input data that have been arranged in a particular format.
- **Control string:** It contains field specifications, which direct the interpretation of input data. It is also known as format string.
- **Formatted output:** It refers to the generated output that has been arranged in a particular format.
- **printf:** It is a function used to print and display output of a program.
- **scanf:** It is a function used to read values entered by the user upon execution of a program.
- **Operator:** It is a symbol that tells the computer to perform certain mathematical or logical computations.
- **Expression:** It is a sequence of operands and operators that reduces to a single value.
- **Integer expression:** When both the operands in a single arithmetic expression are integers, then that expression is termed as integer expression.
- **Real arithmetic:** An arithmetic operation involving only real operands is known as real arithmetic.
- **Relational operators:** These operators are used for making comparisons between two expressions.
- **Logical operators:** These operators are used for testing more than one condition and making decisions.
- **Assignment operators:** These operators are used for assigning the result of an expression to a variable.
- **Bitwise operators:** These operators are used for testing the bits, or shifting them right or left.
- **Comma operator:** It is used to link the related expressions together.
- **sizeof operator:** It is a compile time operator and when used with an operand, it returns the number of bytes the operand occupies.
- **Arithmetic expressions:** It is a combination of variables, constants and operators arranged as per the syntax of the language.
- **Decision-making statements:** These statements control the flow of execution and make decisions to see whether a particular condition has occurred or not.
- **switch statement:** It is built-in multiway decision statement used for testing the value of a given variable against a list of case values.
- **Conditional operator:** It is an operator comprising three operands that is used for making two-way decisions.
- **goto statement:** It is a statement used to transfer the flow of execution unconditionally from one point to another in a program.
- **Program loop:** It consists of two segments, one known as the body of the loop and the other known as the control statement. On the basis of the control statement, the body of the loop is executed repeatedly.
- **Control statement:** It tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.
- **Infinite loop:** It is a permanent loop in which the body is executed over and over again.
- **while statement:** It is an entry-controlled loop statement in which the test-condition is evaluated first and if the condition is true, then the body of the loop is executed.
- **do statement:** It executes the body of the loop before the test is performed.
- **continue statement:** It causes the loop to be continued with the next iteration after skipping further statements in the current iteration.
- **break statement:** It causes the loop to be terminated in which it is enclosed.



Review Questions

True or False

1. Every line in a C program should end with a semicolon.
2. Every C program ends with an END word.
3. `main()` is where the program begins its execution.
4. A line in a program may have more than one statement.
5. The closing brace of the `main()` in a program is the logical end of the program.
6. The purpose of the header file such as `stdio.h` is to store the source code of a program.
7. Comments cause the computer to print the text enclosed between `/*` and `*/` when executed.
8. Syntax errors will be detected by the compiler.
9. Any valid printable ASCII character can be used in an identifier.
10. All variables must be given a type when they are declared.
11. Variable declarations can appear anywhere in a program.
12. ANSI C treats the variables name and Name to be same.
13. Floating point constants, by default, denote float type values.
14. Like variables, constants have a type.
15. Character constants are coded using double quotes.
16. All arithmetic operators have the same level of precedence.
17. A unary expression consists only one operand with no operators.
18. Associativity is used to decide which of several different expressions is evaluated first.
19. An expression statement is terminated with a period.
20. `If` is an iterative control structure.
21. The C standard function that receives a single character from the keyboard is `getchar`.
22. The `scanf` function cannot be used to read a single character from the keyboard.
23. A program stops its execution when a `break` statement is encountered.
24. The `do...while` statement first executes the loop body and then evaluate the loop control expression.
25. An exit-controlled loop is executed a minimum of one time.
26. The three loop expressions used in a `for` loop header must be separated by commas.

Fill in the Blanks

1. Every program statement in a C program must end with a _____.
2. The _____ function is used to display the output on the screen.
3. The _____ header file contains mathematical functions.
4. The `escape sequence character` _____ causes the cursor to move to the next line on the screen.
5. _____ is the largest value that an unsigned `short int` type variable can store.
6. A global variable is also known as _____ variable.
7. A variable can be made constant by declaring it with the qualifier _____ at the time of initialisation.
8. The _____ operator is true only when both the operands are true.
9. _____ operators are used for testing the bits, or shifting them right or left.
10. The _____ statement when executed in a `switch` statement causes immediate exit from the structure.
11. The expression `!(x != y)` can be replaced by the expression _____.
12. The _____ operator returns the number of bytes the operand occupies.

13. The order of evaluation can be changed by using _____ in an expression.
14. _____ is used to determine the order in which different operators in an expression are evaluated.
15. In `do-while` loop, the body is executed at least _____ number of time.
16. The _____ statement is used to skip the remaining part of the statements in a loop.
17. A `for` loop with the no test condition is known as _____ loop.
18. _____ should be avoided as part of structured programming approach.
19. `n++` is equivalent to the expression _____.



Multiple Choice Questions

1. In which year C language was developed?
 - A. 1951
 - B. 1962
 - C. 1972
 - D. 1947
2. C is a _____.
 - A. Structured language
 - B. Object-oriented language
 - C. Machine language
 - D. Assembly language
3. The execution of a program written in a C language begins at:
 - A. `main()`
 - B. `scanf()`
 - C. `printf()`
 - D. `#include<iostream.h>`
4. Which one of the following is an example of a valid file name in C language?
 - A. `system`
 - B. `system.c`
 - C. `system.cpp`
 - D. `system.java`
5. Which of the following is the correct syntax for the `printf` statement?
 - A. `printf('Hello world');`
 - B. `printf("Hello world")`
 - C. `printf("Hello world");`
 - D. `printf{'Hello world'};`
6. Which of the following is an intermediary file generated during the execution of a C program?
 - A. `.c`
 - B. `.obj`
 - C. `.exe`
 - D. `.bak`
7. Which of the following will run successfully in C?
 - A. `main()`
 - B. `main()`
 - C. `main()`
 - D. `main()`
8. Which of the following is the correct form of writing comments?
 - A. `/* comment */`
 - B. `/* comment /*`
 - C. `*/ comment /*`
 - D. `*/comment */`
9. Which one of the following is not a real constant?
 - A. 15.25
 - B. 0.962
 - C. 10
 - D. +24.85
10. Which one of the following is a string constant?
 - A. `'5'`
 - B. `"hello"`
 - C. 25
 - D. None of the above
11. Which one of the following does not represent a variable?
 - A. `%`
 - B. `height`
 - C. `xyl`
 - D. `m_width`
12. The range of values for a `char` data type is
 - A. -128 to 127
 - B. 3.4e-38 to 3.4e+38
 - C. 1.7e-308 to 1.7e+308
 - D. -32,768 to 32,768
13. Which one of the following is a floating-point data type?
 - A. `float`
 - B. `double`
 - C. `long double`
 - D. `int`

- A. int
B. long int
C. double
D. None of the above
14. Which command is used for reading data from keyboard in C language?
A. cout B. cin
C. printf () D. scanf ()
15. Which one of the following commands is used for the purpose of displaying output in C language?
A. scanf ()
B. printf ()
C. system.out.println ()
D. None of the above
16. Which of the following is the trigraph sequence for representing #?
A. ??= B. ??-
C. ??(D. ??>
17. Which of the following is a legible variable name?
A. income_tax B. income-tax
C. income.tax D. income,tax
18. Which of the following is not a string?
A. "S"
B. 'S'
C. "\S\""
D. All of the above are strings
19. Which of the following cannot be used for declaring integer type variables?
A. int B. short int
C. long int D. double int
20. Which of the following will not read an integer number?
A. %d B. %f
C. %ld D. %c
21. Which of the following is not an arithmetic operator?
A. + B. -
C. * D. &
22. Which of the following operators are used for obtaining the remainder in a division operation?
A. /
B. %
C. !
D. None of the above
23. Which of the following is the correct statement for computing logical AND?
A. a<b & x>y
B. a<b && x>y
C. a<b AND x>y
D. None of the above
24. $a += 1$ will result in:
A. $a = a + a$ B. $a = a + 1$
C. $a = 1 + 1$ D. $a = a + (a + 1)$
25. Which of the following is the bitwise left shift operator?
A. < B. <<
C. <<< D. ^



Programming Exercise

- Give a brief description of the history of C language.
- Discuss briefly the characteristics of C.
- What are the advantages and disadvantages of C?
- What are the various keywords used in C?
- Explain different data types in C.
- What is a constant? Explain the constants in C.
- What is a variable? How are the variables declared in C?
- List the different types of control statements in C.
- Give the output of the following code:


```
float c = 34.78650;
printf ("%6.2f" , c);
```

11. Write a program in C to find the value of y using the relation $y = x_2 + 2x - 1$.
12. Write a program in C to find the sum and the average of three numbers.
13. Write a program in C to calculate simple interest.
14. Write a program in C to convert the temperature from $^{\circ}\text{C}$ to $^{\circ}\text{F}$.
15. Write a program in C to evaluate the series $S = 1 + 2 * 1 + 3 * 2 + \dots N * N - 1$.
16. Write a program to determine and print the sum of the following harmonic series for a given value of n :

$$1 + 1/2 + 1/3 + \dots + 1/n$$

The value of n should be given interactively through the terminal.
17. Write a program to read the price of an item in decimal form (like 15.95) and print the output in paise (like 1595 paise).
18. Write a program that prints the even numbers from 1 to 100.
19. Write a program that requests two float-type numbers from the user and then divides the first number by the second and displays the result along with the numbers.
20. The price of 1 kg rice is Rs. 16.75 and 1 kg of sugar is Rs. 15. Write a program to get these

values from the user and display the prices as follows:

```
*** LIST OF ITEMS ***
Item      Price
Rice      Rs 16.75
Sugar     Rs 15.00
```

21. Write program to count and print the number of negative and positive numbers in a given set of numbers. Test your program with a suitable set of numbers. Use `scanf` to read the numbers. Reading should be terminated when the value 0 is encountered.
22. Write a program to do the following:
 - (a) Declare x and y as integer variables and z as a short integer variable.
 - (b) Assign two 6 digit numbers to x and y
 - (c) Assign the sum of x and y to z
 - (d) Output the values of x , y and z

Comment on the output.
23. Write a program to read two floating-point numbers using a `scanf` statement, assign their sum to an integer variable and then output the values of all the three variables.
24. Write a program to illustrate the use of `typedef` declaration in a program.
25. Write a program to illustrate the use of symbolic constants in a real-life application.

UNIT

3A Functions

3A.1 INTRODUCTION

We have mentioned earlier that one of the strengths of C language is C functions. They are easy to define and use. We have used functions in every program that we have discussed so far. However, they have been primarily limited to the three functions, namely, **main**, **printf**, and **scanf**. In this chapter, we shall consider in detail the following:

- How a function is designed?
- How a function is integrated into a program?
- How two or more functions are put together? and
- How they communicate with one another?

C functions can be classified into two categories, namely, *library* functions and *user-defined* functions. **main** is an example of user-defined functions. **printf** and **scanf** belong to the category of library functions. We have also used other library functions such as **sqrt**, **cos**, **strcat**, etc. The main distinction between these two categories is that library functions are not required to be written by us whereas a user-defined function has to be developed by the user at the time of writing a program. However, a user-defined function can later become a part of the C program library. In fact, this is one of the strengths of C language.

3A.2 NEED FOR USER-DEFINED FUNCTIONS

As pointed out earlier, **main** is a specially recognized function in C. Every program must have a **main** function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only **main** function, it leads to a number of problems. The program may become too large and complex and as a result the task of debugging, testing, and maintaining becomes difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These independently coded programs are called *subprograms* that are much easier to understand, debug, and test. In C, such subprograms are referred to as '**functions**'.

3A.2 Computer Programming

There are times when certain type of operations or calculations are repeated at many points throughout a program. For instance, we might use the factorial of a number at several points in the program. In such situations, we may repeat the program statements wherever they are needed. Another approach is to design a function that can be called and used whenever required. This saves both time and space.

This “division” approach clearly results in a number of advantages.

1. It facilitates top-down modular programming as shown in Fig. 3A.1. In this programming style, the high level logic of the overall problem is solved first while the details of each lower-level function are addressed later.
2. The length of a source program can be reduced by using functions at appropriate places. This factor is particularly critical with microcomputers where memory space is limited.
3. It is easy to locate and isolate a faulty function for further investigations.
4. A function may be used by many other programs. This means that a C programmer can build on what others have already done, instead of starting all over again from scratch.

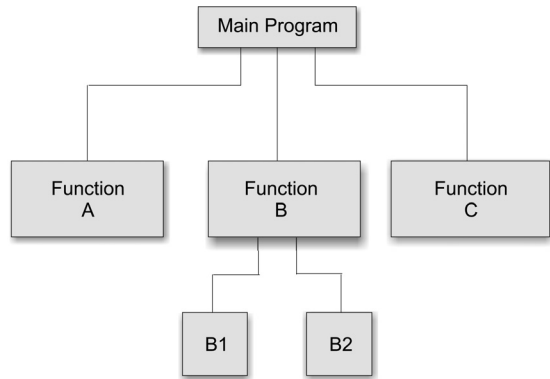


Fig. 3A.1 Top-down modular programming using functions

3A.3 A MULTI-FUNCTION PROGRAM

A function is a self-contained block of code that performs a particular task. Once a function has been designed and packed, it can be treated as a ‘black box’ that takes some data from the main program and returns a value. The inner details of operation are invisible to the rest of the program. All that the program knows about a function is: What goes in and what comes out. Every C program can be designed using a collection of these black boxes known as *functions*.

Consider a set of statements as shown below:

```
void printline(void)
{
    int i;
    for (i=1; i<40; i++)
        printf("-");
    printf("\n");
}
```

The above set of statements defines a function called **printline**, which could print a line of 39-character length. This function can be used in a program as follows:

```
void printline(void); /* declaration */
main( )
{
    printline( );
    printf("This illustrates the use of C functions\n");
    printline();
}
```

```

}
void printline(void)
{
    int i;
    for(i=1; i<40; i++)
        printf("-");
    printf("\n");
}

```

This program will print the following output:

This illustrates the use of C functions

The above program contains two user-defined functions:

main() function

printline() function

As we know, the program execution always begins with the **main** function. During execution of the **main**, the first statement encountered is

```
printline( );
```

which indicates that the function **printline** is to be executed. At this point, the program control is transferred to the function **printline**. After executing the **printline** function, which outputs a line of 39 character length, the control is transferred back to the **main**. Now, the execution continues at the point where the function call was executed. After executing the **printf** statement, the control is again transferred to the **printline** function for printing the line once more.

The **main** function calls the user-defined **printline** function two times and the library function **printf** once. We may notice that the **printline** function itself calls the library function **printf** 39 times repeatedly.

Any function can call any other function. In fact, it can call itself. A 'called function' can also call another function. A function can be called more than once. In fact, this is one of the main features of using functions. Figure 3A.2 illustrates the flow of control in a multi-function program.

Except the starting point, there are no other predetermined relationships, rules of precedence, or hierarchies among the functions that make up a complete program. The functions can be placed in any order. A called function can be placed either before or after the calling function. However, it is the usual practice to put all the called functions at the end. See the box "Modular Programming"

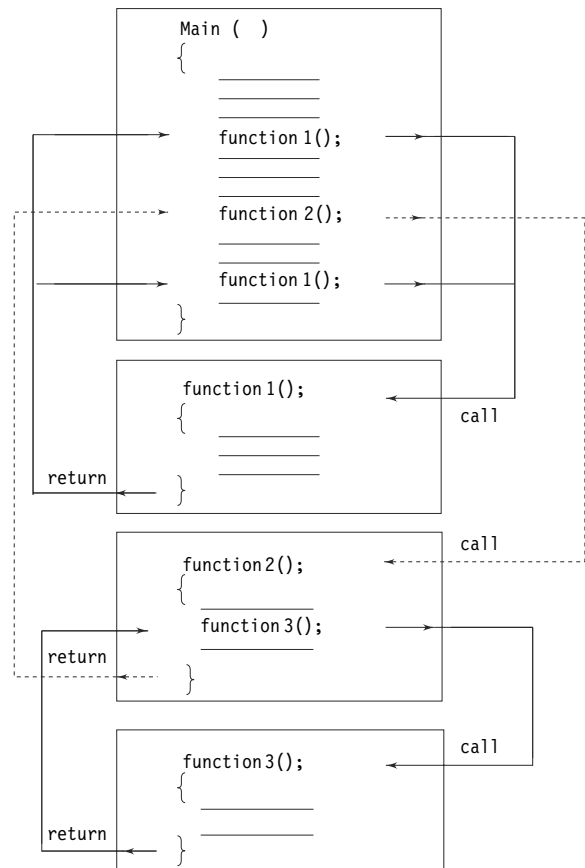


Fig. 3A.2 Flow of control in a multi-function program

Modular Programming

Modular programming is a strategy applied to the design and development of software systems. It is defined as organizing a large program into small, independent program segments called **modules** that are separately named and individually callable *program units*. These modules are carefully integrated to become a software system that satisfies the system requirements. It is basically a "divide-and-conquer" approach to problem solving.

Modules are identified and designed such that they can be organized into a top-down hierarchical structure (similar to an organization chart). In C, each module refers to a function that is responsible for a single task.

Some characteristics of modular programming are:

1. Each module should do only one thing.
2. Communication between modules is allowed only by a calling module.
3. A module can be called by one and only one higher module.
4. No communication can take place directly between modules that do not have calling-called relationship.
5. All modules are designed as *single-entry, single-exit* systems using control structures.

3A.4 ELEMENTS OF USER-DEFINED FUNCTIONS

We have discussed and used a variety of data types and variables in our programs so far. However, declaration and use of these variables were primarily done inside the **main** function. As we mentioned in Chapter 4, functions are classified as one of the derived data types in C. We can therefore define functions and use them like any other variables in C programs. It is therefore not a surprise to note that there exist some similarities between functions and variables in C.

- Both function names and variable names are considered identifiers and therefore they must adhere to the rules for identifiers.
- Like variables, functions have types (such as `int`) associated with them.
- Like variables, function names and their types must be declared and defined before they are used in a program.

In order to make use of a user-defined function, we need to establish three elements that are related to functions.

1. Function definition.
2. Function call.
3. Function declaration.

The *function definition* is an independent program module that is specially written to implement the requirements of the function. In order to use this function we need to invoke it at a required place in the program. This is known as the *function call*. The program (or a function) that calls the function is referred to as the *calling program* or *calling function*. The calling program should declare any function (like declaration of a variable) that is to be used later in the program. This is known as the *function declaration* or *function prototype*.

3A.5 DEFINITION OF FUNCTIONS

A *function definition*, also known as *function implementation* shall include the following elements;

1. function name;
2. function type;
3. list of parameters;
4. local variable declarations;
5. function statements; and
6. a return statement.

All the six elements are grouped into two parts, namely,

- function header (First three elements); and
- function body (Second three elements).

A general format of a function definition to implement these two parts is given below:

```
function_type function_name(parameter list)
{
    local variable declaration;
    executable statement1;
    executable statement2;
    . . . . .
    . . . . .
    return statement;
}
```

The first line **function_type function_name(parameter list)** is known as the *function header* and the statements within the opening and closing braces constitute the *function body*, which is a compound statement.

3A.5.1 Function Header

The function header consists of three parts: the function type (also known as *return type*), the function name and the *formal* parameter list. Note that a semicolon is not used at the end of the function header.

3A.5.2 Name and Type

The *function type* specifies the type of value (*like float or double*) that the function is expected to return to the program calling the function. If the return type is not explicitly specified, C will assume that it is an integer type. If the function is not returning anything, then we need to specify the return type as **void**. Remember, **void** is one of the fundamental data types in C. It is a good programming practice to code explicitly the return type, even when it is an integer. The value returned is the output produced by the function.

The *function name* is any valid C identifier and therefore must follow the same rules of formation as other variable names in C. The name should be appropriate to the task performed by the function. However, care must be exercised to avoid duplicating library routine names or operating system commands.

3A.5.3 Formal Parameter List

The *parameter list* declares the variables that will receive the data sent by the calling program. They serve as input data to the function to carry out the specified task. Since they represent actual input values, they are often referred to as *formal* parameters. These parameters can also be used to send values to the calling programs. This aspect will be covered later when we discuss more about functions. The parameters are also known as *arguments*.

The parameter list contains declaration of variables separated by commas and surrounded by parentheses. Examples:

```
float quadratic (int a, int b, int c) { . . . }
double power (double x, int n) { . . . }
float mul (float x, float y) { . . . }
int sum (int a, int b) { . . . }
```

Remember, there is no semicolon after the closing parenthesis. Note that the declaration of parameter variables cannot be combined. That is, **int sum (int a,b)** is illegal.

3A.6 Computer Programming

A function need not always receive values from the calling program. In such cases, functions have no formal parameters. To indicate that the parameter list is empty, we use the keyword **void** between the parentheses as in

```
void printline (void)
```

```
{  
    ....  
}
```

This function neither receives any input values nor returns back any value. Many compilers accept an empty set of parentheses, without specifying anything as in

```
void printline ( )
```

But, it is a good programming style to use **void** to indicate a null parameter list.

3A.5.4 Function Body

The *function body* contains the declarations and statements necessary for performing the required task. The body enclosed in braces, contains three parts, in the order given below:

1. Local declarations that specify the variables needed by the function.
2. Function statements that perform the task of the function.
3. A **return** statement that returns the value evaluated by the function.

If a function does not return any value (like the **printline** function), we can omit the **return** statement. However, note that its return type should be specified as **void**. Again, it is nice to have a return statement even for **void** functions.

Some examples of typical function definitions are:

```
(a) float mul (float x, float y)  
    {  
        float result;           /* local variable */  
        result = x * y;         /* computes the product */  
        return (result);        /* returns the result */  
    }  
(b) void sum (int a, int b)  
    {  
        printf ("sum = %s", a + b); /* no local variables */  
        return;                  /* optional */  
    }  
(c) void display (void)  
    {  
                                   /* no local variables */  
        printf ("No type, no parameters");  
                                   /* no return statement */  
    }
```

NOTE:

1. When a function reaches its return statement, the control is transferred back to the calling program. In the absence of a return statement, the closing brace acts as a void return.
2. A local variable is a variable that is defined inside a function and used without having any role in the communication between functions.

3A.6 RETURN VALUES AND THEIR TYPES

As pointed out earlier, a function may or may not send back any value to the calling function. If it does, it is done through the **return** statement. While it is possible to pass to the called function any number of values, the called function can only return *one value* per call, at the most.

The **return** statement can take one of the following forms:

```
return;
or
return(expression);
```

The first, the ‘plain’ **return** does not return any value; it acts much as the closing brace of the function. When a **return** is encountered, the control is immediately passed back to the calling function. An example of the use of a simple **return** is as follows:

```
if(error)
return;
```

NOTE: In C99, if a function is specified as returning a value, the return must have value associated with it.

The second form of **return** with an expression returns the value of the expression. For example, the function

```
int mul (int x, int y)
{
    int p;
    p = x*y;
    return(p);
}
```

returns the value of **p** which is the product of the values of **x** and **y**. The last two statements can be combined into one statement as follows:

```
return (x*y);
```

A function may have more than one **return** statements. This situation arises when the value returned is based on certain conditions. For example:

```
if( x <= 0 )
    return(0);
else
    return(1);
```

What type of data does a function return? All functions by default return **int** type data. But what happens if a function must return some other type? We can force a function to return a particular type of data by using a *type specifier* in the function header as discussed earlier.

When a value is returned, it is automatically cast to the function’s type. In functions that do computations using **doubles**, yet return **ints**, the returned value will be truncated to an integer. For instance, the function

```
int product (void)
{
    return (2.5 * 3.0);
}
```

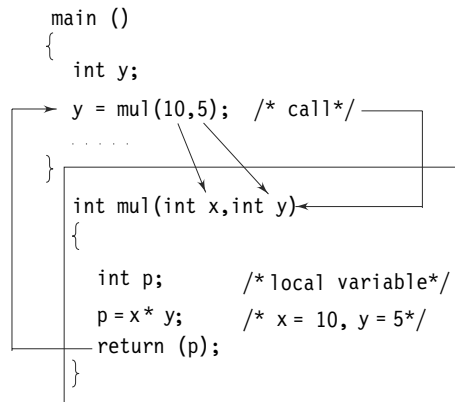
will return the value 7, only the integer part of the result.

3A.7 FUNCTION CALLS

A function can be called by simply using the function name followed by a list of *actual parameters* (or arguments), if any, enclosed in parentheses. Example:

```
main( )
{
    int y;
    y = mul(10,5);      /* Function call */
    printf("%d\n", y);
}
```

When the compiler encounters a function call, the control is transferred to the function **mul()**. This function is then executed line by line as described and a value is returned when a **return** statement is encountered. This value is assigned to **y**. This is illustrated below:



The function call sends two integer values 10 and 5 to the function.

int mul(int x, int y)

which are assigned to **x** and **y** respectively. The function computes the product **x** and **y**, assigns the result to the local variable **p**, and then returns the value 25 to the **main** where it is assigned to **y** again.

There are many different ways to call a function. Listed below are some of the ways the function **mul** can be invoked.

```
mul (10, 5)
mul (m, 5)
mul (10, n)
mul (m, n)
mul (m + 5, 10)
mul (10, mul(m,n))
mul (expression1, expression2)
```

Note that the sixth call uses its own call as its one of the parameters. When we use expressions, they should be evaluated to single values that can be passed as actual parameters.

A function which returns a value can be used in expressions like any other variable. Each of the following statements is valid:

```
printf("%d\n", mul(p,q));
y = mul(p,q) / (p+q);
if (mul(m,n)>total) printf("large");
```

However, a function cannot be used on the right side of an assignment statement. For instance,

```
mul(a,b) = 15;
```

is invalid.

A function that does not return any value may not be used in expressions; but can be called in to perform certain tasks specified in the function. The function **println()** discussed in Section 3A.3 belongs to this category. Such functions may be called in by simply stating their names as independent statements.

Example:

```
main( )
{
    println( );
}
```

Note the presence of a semicolon at the end.

Function Call

A function call is a postfix expression. The operator (.) is at a very high level of precedence. (See Table 3A.8) Therefore, when a function call is used as a part of an expression, it will be evaluated first, unless parentheses are used to change the order of precedence.

In a function call, the function name is the operand and the parentheses set (.) which contains the *actual parameters* is the operator. The actual parameters must match the function's formal parameters in type, order and number. Multiple actual parameters must be separated by commas.

NOTE:

1. If the actual parameters are more than the formal parameters, the extra actual arguments will be discarded.
2. On the other hand, if the actuals are less than the formals, the unmatched formal arguments will be initialized to some garbage.
3. Any mismatch in data types may also result in some garbage values.

3A.8 FUNCTION DECLARATION

Like variables, all functions in a C program must be declared, before they are invoked. A *function declaration* (also known as *function prototype*) consists of four parts.

- Function type (return type).
- Function name.
- Parameter list.
- Terminating semicolon.

They are coded in the following format:

```
Function-type function-name (parameter list);
```

This is very similar to the function header line except the terminating semicolon. For example, **mul** function defined in the previous section will be declared as:

```
int mul (int m, int n); /* Function prototype */
```

Points to note:

1. The parameter list must be separated by commas.
2. The parameter names do not need to be the same in the prototype declaration and the function definition.

3A.10 Computer Programming

3. The types must match the types of parameters in the function definition, in number and order.
4. Use of parameter names in the declaration is optional.
5. If the function has no formal parameters, the list is written as (void).
6. The return type is optional, when the function returns **int** type data.
7. The retype must be **void** if no value is returned.
8. When the declared types do not match with the types in the function definition, compiler will produce an error.

Equally acceptable forms of declaration of **mul** function are:

```
int mul (int, int);  
mul (int a, int b);  
mul (int, int);
```

When a function does not take any parameters and does not return any value, its prototype is written as:

```
void display (void);
```

A prototype declaration may be placed in two places in a program.

1. Above all the functions (including **main**).
2. Inside a function definition.

When we place the declaration above all the functions (in the global declaration section), the prototype is referred to as a *global prototype*. Such declarations are available for all the functions in the program.

When we place it in a function definition (in the local declaration section), the prototype is called a *local prototype*. Such declarations are primarily used by the functions containing them.

The place of declaration of a function defines a region in a program in which the function may be used by other functions. This region is known as the *scope* of the function. (Scope is discussed later in this chapter.) It is a good programming style to declare prototypes in the global declaration section before **main**. It adds flexibility, provides an excellent quick reference to the functions used in the program, and enhances documentation.

Prototypes: Yes or No

Prototype declarations are not essential. If a function has not been declared before it is used, C will assume that its details available at the time of linking. Since the prototype is not available, C will assume that the return type is an integer and that the types of parameters match the formal definitions. If these assumptions are wrong, the linker will fail and we will have to change the program. The moral is that we must always include prototype declarations, preferably in global declaration section.

Parameters Everywhere!

Parameters (also known as arguments) are used in three places:

1. in declaration (prototypes),
2. in function call, and
3. in function definition.

The parameters used in prototypes and function definitions are called *formal parameters* and those used in function calls are called *actual parameters*. Actual parameters used in a calling statement may be simple constants, variables or expressions.

The formal and actual parameters must match exactly in type, order and number. Their names, however, do not need to match.

3A.9 CATEGORY OF FUNCTIONS

A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:

- Category 1: Functions with no arguments and no return values.
- Category 2: Functions with arguments and no return values.
- Category 3: Functions with arguments and one return value.
- Category 4: Functions with no arguments but return a value.
- Category 5: Functions that return multiple values.

In the sections to follow, we shall discuss these categories with examples. Note that, from now on, we shall use the term arguments (rather than parameters) more frequently:

3A.10 NO ARGUMENTS AND NO RETURN VALUES

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function. In effect, there is no data transfer between the calling function and the called function. This is depicted in Fig. 3A.3. The dotted lines indicate that there is only a transfer of control but not data.

As pointed out earlier, a function that does not return any value cannot be used in an expression. It can only be used as an independent statement.

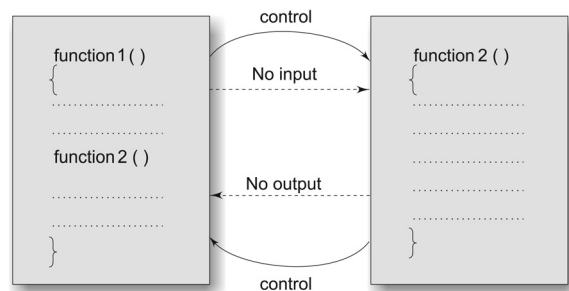


Fig. 3A.3 No data communication between functions

Example 3A.1 Write a program with multiple functions that do not communicate any data between them.

A program with three user-defined functions is given in Fig. 3A.4. **main** is the calling function that calls **prntline** and **value** functions. Since both the called functions contain no arguments, there are no argument declarations. The **prntline** function, when encountered, prints a line with a length of 35 characters as prescribed in the function. The **value** function calculates the value of principal amount after a certain period of years and prints the results. The following equation is evaluated repeatedly:

$$\text{value} = \text{principal}(1 + \text{interest-rate})$$

Program

```
/* Function declaration */
void prntline (void);
void value (void);

main()
{
    prntline();
    value();
}
```

3A.12 Computer Programming

```
        println();
    }

    /* Function1: println( ) */

    void println(void) /* contains no arguments */
    {
        int i ;
        for(i=1; i <= 35; i++)
            printf("%c",'-' );
        printf("\n");
    }

    /* Function2: value( ) */

    void value(void) /* contains no arguments */
    {
        int year, period;
        float inrate, sum, principal;

        printf("Principal amount?");
        scanf("%f", &principal);
        printf("Interest rate? ");
        scanf("%f", &inrate);
        printf("Period? ");
        scanf("%d", &period);

        sum = principal;
        year = 1;
        while(year <= period)
        {
            sum = sum *(1+inrate);
            year = year +1;
        }
        printf("\n%8.2f %5.2f %5d %12.2f\n",
            principal,inrate,period,sum);
    }
}
```

Output

```
-----
Principal amount?      5000
Interest rate?         0.12
Period?                5

5000.00 0.12           5 8811.71
-----
```

Fig. 3A.4 Functions with no arguments and no return values

It is important to note that the function **value** receives its data directly from the terminal. The input data include principal amount, interest rate and the period for which the final value is to be calculated. The **while** loop calculates the final value and the results are printed by the library function **printf**. When the closing brace of **value()** is reached, the control is transferred back to the calling function **main**. Since everything is done by the value itself there is in fact nothing left to be sent back to the called function. Return types of both **println** and **value** are declared as **void**.

Note that no **return** statement is employed. When there is nothing to be returned, the **return** statement is optional. The closing brace of the function signals the end of execution of the function, thus returning the control, back to the calling function.

3A.11 ARGUMENTS BUT NO RETURN VALUES

In Fig. 3A.4 the **main** function has no control over the way the functions receive input data. For example, the function **printline** will print the same line each time it is called. Same is the case with the function **value**. We could make the calling function to read data from the terminal and pass it on to the called function. This approach seems to be wiser because the calling function can check for the validity of data, if necessary, before it is handed over to the called function.

The nature of data communication between the *calling function* and the *called function* with arguments but no return value is shown in Fig. 3A.5.

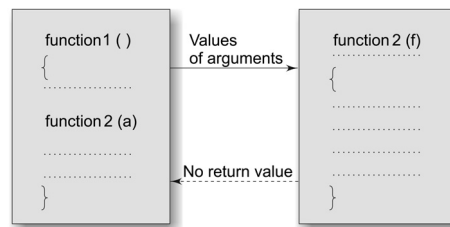


Fig. 3A.5 One-way data communication

We shall modify the definitions of both the called functions to include arguments as follows:

```
void printline(char ch)
```

```
void value(float p, float r, int n)
```

The arguments **ch**, **p**, **r**, and **n** are called the *formal arguments*. The calling function can now send values to these arguments using function calls containing appropriate arguments. For example, the function call

```
value(500,0.12,5)
```

would send the values 500, 0.12 and 5 to the function

```
void value( float p, float r, int n)
```

and assign 500 to **p**, 0.12 to **r** and 5 to **n**. The values 500, 0.12 and 5 are the *actual arguments*, which become the values of the *formal arguments* inside the called function.

The *actual* and *formal* arguments should match in number, type, and order. The values of actual arguments are assigned to the formal arguments on a *one to one* basis, starting with the first argument as shown in Fig. 3A.6.

We should ensure that the function call has matching arguments. In case, the actual arguments are more than the formal arguments ($m > n$), the extra actual arguments are discarded. On the other hand, if the actual arguments are less than the formal arguments, the unmatched formal arguments are initialized to some garbage values. Any mismatch in data type may also result in passing of garbage values. Remember, no error message will be generated.

While the formal arguments must be valid variable names, the actual arguments may be variable names, expressions, or constants. The variables used in actual arguments must be assigned values before the function call is made.

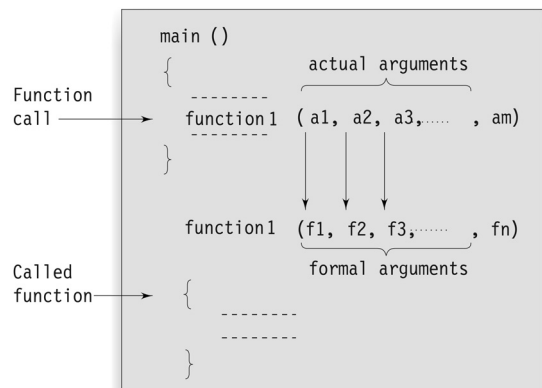


Fig. 3A.6 Arguments matching between the function call and the called function

3A.14 Computer Programming

Remember that, when a function call is made, only *a copy of the values of actual arguments is passed into the called function*. What occurs inside the function will have no effect on the variables used in the actual argument list.

Example 3A.2 Modify the program of Example 3A.1 to include the arguments in the function calls.

The modified program with function arguments is presented in Fig. 3A.7. Most of the program is identical to the program in Fig. 3A.3A. The input prompt and **scanf** assignment statement have been moved from **value** function to **main**. The variables **principal**, **inrate**, and **period** are declared in **main** because they are used in main to receive data. The function call

```
value(principal, inrate, period);
```

passes information it contains to the function **value**.

The function header of **value** has three formal arguments **p,r**, and **n** which correspond to the actual arguments in the function call, namely, **principal**, **inrate**, and **period**. On execution of the function call, the values of the actual arguments are assigned to the corresponding formal arguments. In fact, the following assignments are accomplished across the function boundaries:

```
p = principal;
r = inrate;
n = period;
```

Program

```
/* prototypes */
void printline (char c);
void value (float, float, int);

main( )
{
    float principal, inrate;
    int period;

    printf("Enter principal amount, interest");
    printf(" rate, and period \n");
    scanf("%f %f %d",&principal, &inrate, &period);
    printline('Z');
    value(principal,inrate,period);
    printline('C');
}

void printline(char ch)
{
    int i ;
    for(i=1; i <= 52; i++)
        printf("%c",ch);
    printf("\n");
}

void value(float p, float r, int n)
{
    int year ;
    float sum ;
    sum = p ;
    year = 1;
    while(year <= n)
```


Example 3A.3 In the program presented in Fig. 3A.7 modify the function value, to return the final amount calculated to the main, which will display the required output at the terminal. Also extend the versatility of the function printline by having it to take the length of the line as an argument.

The modified program with the proposed changes is presented in Fig. 3A.9. One major change is the movement of the **printf** statement from **value** to **main**.

Program

```
void printline (char ch, int len);
value (float, float, int);

main( )
{
    float principal, inrate, amount;
    int period;
    printf("Enter principal amount, interest");
    printf("rate, and period\n");
    scanf("%f %f %d", &principal, &inrate, &period);
    printline ('*', 52);
    amount = value (principal, inrate, period);
    printf("\n%f\t%f\t%d\t%f\n\n",principal,
        inrate,period,amount);
    printline('=',52);
}

void printline(char ch, int len)
{
    int i;
    for (i=1;i<=len;i++) printf("%c",ch);
    printf("\n");
}

value(float p, float r, int n) /* default return type */
{
    int year;
    float sum;
    sum = p; year = 1;
    while(year <=n)
    {
        sum = sum * (1+r);
        year = year +1;
    }
    return(sum); /* returns int part of sum */
}
```

Output

```
Enter principal amount, interest rate, and period
5000    0.12 5
*****
5000.000000    0.1200000    5    8811.000000
= = = = =
```

Fig. 3A.9 Functions with arguments and return values

The calculated value is passed on to **main** through statement:

```
return(sum);
```

Since, by default, the return type of **value** function is **int**, the 'integer' value of **sum** at this point is returned to **main** and assigned to the variable **amount** by the functional call

```
amount = value (principal, inrate, period);
```

The following events occur, in order, when the above function call is executed:

1. The function call transfers the control along with copies of the values of the actual arguments to the function **value** where the formal arguments **p**, **r**, and **n** are assigned the actual values of **principal**, **inrate** and **period** respectively.
2. The called function **value** is executed line by line in a normal fashion until the **return(sum);** statement is encountered. At this point, the integer value of **sum** is passed back to the function-call in the **main** and the following indirect assignment occurs:

```
value(principal, inrate, period) = sum;
```

3. The calling statement is executed normally and the returned value is thus assigned to **amount**, a **float** variable.
4. Since **amount** is a **float** variable, the returned integer part of sum is converted to floating-point value. See the output.

Another important change is the inclusion of second argument to **println** function to receive the value of length of the line from the calling function. Thus, the function call

```
println('*', 52);
```

will transfer the control to the function **println** and assign the following values to the formal arguments **ch**, and **len**;

```
ch = '*' ;
len = 52;
```

Returning Float Values

We mentioned earlier that a C function returns a value of the type **int** as the default case when no other type is specified explicitly. For example, the function **value** of Example 3A.3 does all calculations using **floats** but the return statement

```
return(sum);
```

returns only the integer part of **sum**. This is due to the absence of the *type-specifier* in the function header. In this case, we can accept the integer value of **sum** because the truncated decimal part is insignificant compared to the integer part. However, there will be times when we may find it necessary to receive the **float** or **double** type of data. For example, a function that calculates the mean or standard deviation of a set of values should return the function value in either **float** or **double**.

In all such cases, we must explicitly specify the *return type* in both the function definition and the prototype declaration.

If we have a mismatch between the type of data that the called function returns and the type of data that the calling function expects, we will have unpredictable results. We must, therefore, be very careful to make sure that both types are compatible.

Example 3A.4 Write a function **power** that computes **x** raised to the power **y** for integers **x** and **y** and returns double-type value.

Figure 3A.10 shows a **power** function that returns a **double**. The prototype declaration

3A.18 Computer Programming

`double power(int, int);`
appears in **main**, before **power** is called.

Program

```
main( )
{
    int x,y; /*input data */

    double power(int, int); /* prototype declaration*/

    printf("Enter x,y:");

    scanf("%d %d" , &x,&y);
    printf("%d to power %d is %f\n", x,y,power (x,y));
}

double power (int x, int y);
{
    double p;
    p = 1.0 ;      /* x to power zero */

    if(y >=0)
        while(y-- ) /* computes positive powers */
            p *= x;
    else
        while (y++) /* computes negative powers */
            p /= x;
    return(p);     /* returns double type */
}
```

Output

```
Enter x,y:16 2
16 to power 2 is 256.000000

Enter x,y:16 -2
16 to power -2 is 0.003906
```

Fig. 3A.10 Power fuctions: Illustration of return of float values

Another way to guarantee that **power**'s type is declared before it is called in **main** is to define the **power** function before we define **main**. **Power**'s type is then known from its definition, so we no longer need its type declaration in **main**.

3A.13 NO ARGUMENTS BUT RETURNS A VALUE

There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. A typical example is the **getchar** function declared in the header file **<stdio.h>**. We have used this function earlier in a number of places. The **getchar** function has no parameters but it returns an integer type data that represents a character.

We can design similar functions and use in our programs. Example:

```
int get_number(void);
main
```

```

{
    int m = get_number( );
    printf("%d",m);
}
int get_number(void)
{
    int number;
    scanf("%d", &number);
    return(number);
}

```

3A.14 FUNCTIONS THAT RETURN MULTIPLE VALUES

Up till now, we have illustrated functions that return just one value using a return statement. That is because, a return statement can return only one value. Suppose, however, that we want to get more information from a function. We can achieve this in C using the arguments not only to receive information but also to send back information to the calling function. The arguments that are used to “send out” information are called *output parameters*.

The mechanism of sending back information through arguments is achieved using what are known as the *address operator (&)* and *indirection operator (*)*. Let us consider an example to illustrate this.

```

void mathoperation (int x, int y, int *s, int *d);
main( )
{
    int x = 20, y = 10, s, d;
    mathoperation(x,y, &s, &d);

    printf("s=%d\n d=%d\n", s,d);
}
void mathoperation (int a, int b, int *sum, int *diff)
{
    *sum = a+b;
    *diff = a-b;
}

```

The actual arguments **x** and **y** are input arguments, **s** and **d** are output arguments. In the function call, while we pass the actual values of **x** and **y** to the function, we pass the addresses of locations where the values of **s** and **d** are stored in the memory. (That is why, the operator **&** is called the address operator.) When the function is called the following assignments occur:

value of	x to a
value of	y to b
address of	s to sum
address of	d to diff

Note that indirection operator ***** in the declaration of **sum** and **diff** in the header indicates these variables are to store addresses, not actual values of variables. Now, the variables **sum** and **diff** point to the memory locations of **s** and **d** respectively.

(The operator ***** is known as indirection operator because it gives an indirect reference to a variable through its address.)

In the body of the function, we have two statements:

3A.20 Computer Programming

```
* sum   = a+b;  
* diff  = a-b;
```

The first one adds the values **a** and **b** and the result is stored in the memory location pointed to by **sum**. Remember, this memory location is the same as the memory location of **s**. Therefore, the value stored in the location pointed to by **sum** is the value of **s**.

Similarly, the value of **a-b** is stored in the location pointed to by **diff**, which is the same as the location **d**. After the function call is implemented, the value of **s** is **a+b** and the value of **d** is **a-b**. Output will be:

```
s = 30  
d = 10
```

The variables ***sum** and ***diff** are known as *pointers* and **sum** and **diff** as *pointer* variables. Since they are declared as **int**, they can point to locations of **int** type data.

The use of pointer variables as actual parameters for communicating data between functions is called “pass by pointers” or “call by address or reference”.

Rules for Pass by Pointers

1. The types of the actual and formal arguments must be same.
2. The actual arguments (in the function call) must be the addresses of variables that are local to the calling function.
3. The formal arguments in the function header must be prefixed by the indirection operator *****.
4. In the prototype, the arguments must be prefixed by the symbol *****.
5. To access the value of an actual argument in the called function, we must use the corresponding formal argument prefixed with the indirection operator *****.

3A.15 NESTING OF FUNCTIONS

C permits nesting of functions freely. **main** can call **function1**, which calls **function2**, which calls **function3**, and so on. There is in principle no limit as to how deeply functions can be nested.

Consider the following program:

```
float ratio (int x, int y, int z);  
int difference (int x, int y);  
main( )  
{  
    int a, b, c;  
    scanf("%d %d %d", &a, &b, &c);  
    printf("%f \n", ratio(a,b,c));  
}  
  
float ratio(int x, int y, int z)  
{  
    if(difference(y, z))  
        return(x/(y-z));  
    else  
        return(0.0);  
}  
int difference(int p, int q)
```

```

{
    if(p != q)
        return (1);
    else
        return(0);
}

```

The above program calculates the ratio

$$\frac{a}{b - c}$$

and prints the result. We have the following three functions:

```

main()
ratio()
difference()

```

main reads the values of *a*, *b* and *c* and calls the function **ratio** to calculate the value $a/(b-c)$. This ratio cannot be evaluated if $(b-c) = 0$. Therefore, **ratio** calls another function **difference** to test whether the difference $(b-c)$ is zero or not; **difference** returns 1, if *b* is not equal to *c*; otherwise returns zero to the function **ratio**. In turn, **ratio** calculates the value $a/(b-c)$ if it receives 1 and returns the result in **float**. In case, **ratio** receives zero from **difference**, it sends back 0.0 to **main** indicating that $(b-c) = 0$.

Nesting of function calls is also possible. For example, a statement like

```
P = mul(mul(5,2),6);
```

is valid. This represents two sequential function calls. The inner function call is evaluated first and the returned value is again used as an actual argument in the outer function call. If **mul** returns the product of its arguments, then the value of **p** would be 60 ($= 5 \times 2 \times 6$).

Note that the nesting does not mean defining one function within another. Doing this is illegal.

3A.16 RECURSION

When a called function in turn calls another function a process of ‘chaining’ occurs. *Recursion* is a special case of this process, where a function calls itself. A very simple example of recursion is presented below:

```

main( )
{
    printf("This is an example of recursion\n")
    main( );
}

```

When executed, this program will produce an output something like this:

```

This is an example of recursion
This is an example of recursion
This is an example of recursion
This is an ex

```

Execution is terminated abruptly; otherwise the execution will continue indefinitely.

Another useful example of recursion is the evaluation of factorials of a given number. The factorial of a number *n* is expressed as a series of repetitive multiplications as shown below:

factorial of *n* = $n(n-1)(n-2).....1$.

For example,

factorial of 4 = $4 \times 3 \times 2 \times 1 = 24$

3A.22 Computer Programming

A function to evaluate factorial of n is as follows:

```
factorial(int n)
{
    int fact;
    if (n==1)
        return(1);
    else
        fact = n*factorial(n-1);
    return(fact);
}
```

Let us see how the recursion works. Assume n = 3. Since the value of n is not 1, the statement

`fact = n * factorial(n-1);`

will be executed with n = 3. That is,

`fact = 3 * factorial(2);`

will be evaluated. The expression on the right-hand side includes a call to **factorial** with n = 2. This call will return the following value:

`2 * factorial(1)`

Once again, **factorial** is called with n = 1. This time, the function returns 1. The sequence of operations can be summarized as follows:

`fact = 3 * factorial(2)`
`= 3 * 2 * factorial(1)`
`= 3 * 2 * 1`
`= 6`

Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem. When we write recursive functions, we must have an **if** statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function will never return.

3A.17 THE SCOPE, VISIBILITY AND LIFETIME OF VARIABLES

Variables in C differ in behaviour from those in most other languages. For example, in a BASIC program, a variable retains its value throughout the program. It is not always the case in C. It all depends on the 'storage' class a variable may assume.

In C not only do all variables have a data type, they also have a *storage class*. The following variable storage classes are most relevant to functions:

1. Automatic variables.
2. External variables.
3. Static variables.
4. Register variables.

We shall briefly discuss the *scope*, *visibility* and *longevity* of each of the above class of variables. The *scope* of variable determines over what region of the program a variable is actually available for use ('active'). *Longevity* refers to the period during which a variable retains a given value during execution of a program ('alive'). So longevity has a direct effect on the utility of a given variable. The *visibility* refers to the accessibility of a variable from the memory.

The variables may also be broadly categorized, depending on the place of their declaration, as *internal* (local) or *external* (global). Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.

It is very important to understand the concept of storage classes and their utility in order to develop efficient multifunction programs.

3A.17.1 Automatic Variables

Automatic variables are declared inside a function in which they are to be utilized. They are *created* when the function is called and *destroyed* automatically when the function is exited, hence the name automatic. Automatic variables are therefore private (or local) to the function in which they are declared. Because of this property, automatic variables are also referred to as *local* or *internal* variables.

A variable declared inside a function without storage class specification is, by default, an automatic variable. For instance, the storage class of the variable **number** in the example below is automatic.

```
main( )
{
    int number;
    -----
    -----
}
```

We may also use the keyword **auto** to declare automatic variables explicitly.

```
main( )
{
    auto int number;
    -----
    -----
}
```

One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program. This assures that we may declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

Example 3A.5 Write a multifunction to illustrate how automatic variables work.

A program with two subprograms **function1** and **function2** is shown in Fig. 3A.11. **m** is an automatic variable and it is declared at the beginning of each function. **m** is initialized to 10, 100, and 1000 in function1, function2, and **main** respectively.

When executed, **main** calls **function2** which in turn calls **function1**. When **main** is active, $m = 1000$; but when **function2** is called, the **main**'s **m** is temporarily put on the shelf and the new local **m** = 100 becomes active. Similarly, when **function1** is called, both the previous values of **m** are put on the shelf and the latest value of **m** (=10) becomes active. As soon as **function1** ($m=10$) is finished, **function2** ($m=100$) takes over again. As soon it is done, **main** ($m=1000$) takes over. The output clearly shows that the value assigned to **m** in one function does not affect its value in the other functions; and the local value of **m** is destroyed when it leaves a function.

3A.24 Computer Programming

Program

```
void function1(void);
void function2(void);
main( )
{
    int m = 1000;
    function2();

    printf("%d\n",m); /* Third output */
}
void function1(void)
{
    int m = 10;

    printf("%d\n",m); /* First output */
}

void function2(void)
{
    int m = 100;
    function1();
    printf("%d\n",m); /* Second output */
}
```

Output

```
10
100
1000
```

Fig. 3A.11 Working of automatic variables

There are two consequences of the scope and longevity of **auto** variables worth remembering. First, any variable local to **main** will be normally *alive* throughout the whole program, although it is *active* only in **main**. Secondly, during recursion, the nested variables are unique **auto** variables, a situation similar to function-nested **auto** variables with identical names.

3A.17.2 External Variables

Variables that are both *alive* and *active* throughout the entire program are known as *external* variables. They are also known as *global* variables. Unlike local variables, global variables can be accessed by any function in the program. External variables are declared outside a function. For example, the external declaration of integer **number** and float **length** might appear as:

```
int number;
float length = 7.5;
main( )
{
    -----
    -----
}
```

```

function1( )
{
    -----
    -----
}
function2( )
{
    -----
    -----
}

```

The variables **number** and **length** are available for use in all the three functions. In case a local variable and a global variable have the same name, the local variable will have precedence over the global one in the function where it is declared. Consider the following example:

```

int count;
main( )
{
    count = 10;
    -----
    -----
}
function( )
{
    int count = 0;
    -----
    -----
    count = count+1;
}

```

When the **function** references the variable **count**, it will be referencing only its local variable, not the global one. The value of **count** in **main** will not be affected.

Example 3A.6 Write a multifunction program to illustrate the properties of global variables.

A program to illustrate the properties of global variables is presented in Fig. 3A.12. Note that variable **x** is used in all functions but none except **fun2**, has a definition for **x**. Because **x** has been declared ‘above’ all the functions, it is available to each function without having to pass **x** as a function argument. Further, since the value of **x** is directly available, we need not use **return(x)** statements in **fun1** and **fun3**. However, since **fun2** has a definition of **x**, it returns its local value of **x** and therefore uses a **return** statement. In **fun2**, the global **x** is not visible. The local **x** hides its visibility here.

Program

```

int fun1(void);
int fun2(void);
int fun3(void);
int x ;          /* global */
main( )
{

```

```

        x = 10 ;      /* global x */
        printf("x = %d\n", x);
        printf("x = %d\n", fun1());
        printf("x = %d\n", fun2());
        printf("x = %d\n", fun3());
    }
    fun1(void)
    {
        x = x + 10 ;
    }
    int fun2(void)
    {
        int x ;          /* local */
        x = 1 ;
        return (x);
    }
    fun3(void)
    {
        x = x + 10 ;      /* global x */
    }

```

Output

```

x = 10
x = 20
x = 1
x = 30

```

Fig. 3A.12 Illustration of properties of global variables

Once a variable has been declared as global, any function can use it and change its value. Then, subsequent functions can reference only that new value.

Global Variables as Parameters

Since all functions in a program source file can access global variables, they can be used for passing values between the functions. However, using global variables as parameters for passing values poses certain problems.

- The values of global variables which are sent to the called function may be changed inadvertently by the called function.
- Functions are supposed to be independent and isolated modules. This character is lost, if they use global variables.
- It is not immediately apparent to the reader which values are being sent to the called function.
- A function that uses global variables suffers from reusability.

One other aspect of a global variable is that it is available only from the point of declaration to the end of the program. Consider a program segment as shown below:

```

main( )
{
    y = 5;
    . . .
}
int y;      /* global declaration */
func1( )
{

```

```

        y = y+1;
    }

```

We have a problem here. As far as **main** is concerned, **y** is not defined. So, the compiler will issue an error message. Unlike local variables, global variables are initialized to zero by default. The statement

```
y = y+1;
```

in **fun1** will, therefore, assign 1 to **y**.

External Declaration

In the program segment above, the **main** cannot access the variable **y** as it has been declared after the **main** function. This problem can be solved by declaring the variable with the storage class **extern**.

For example:

```

main( )
{
    extern int y;    /* external declaration */
    . . . . .
    . . . . .
}
func1( )
{
    extern int y;    /* external declaration */
    . . . . .
    . . . . .
}
int y;                /* definition */

```

Although the variable **y** has been defined after both the functions, the *external declaration* of **y** inside the functions informs the compiler that **y** is an integer type defined somewhere else in the program. Note that **extern** declaration does not allocate storage space for variables. In case of arrays, the definition should include their size as well.

Example:

```

main( )
{
    int i;
    void print_out(void);
    extern float height [ ];
    . . . . .
    . . . . .
    print_out( );
}
void print_out(void)
{
    extern float height [ ];
    int i;
    . . . . .
    . . . . .
}
float height[SIZE];

```

An **extern** within a function provides the type information to just that one function. We can provide type information to all functions within a file by placing external declarations before any of them.

3A.28 Computer Programming

Example:

```
extern float height[ ];
main( )
{
    int i;
    void print_out(void);
    . . . . .
    . . . . .
    print_out( );
}
void print_out(void)
{
    int i;
    . . . . .
    . . . . .
}
float height[SIZE];
```

The distinction between definition and declaration also applies to functions. A function is defined when its parameters and function body are specified. This tells the compiler to allocate space for the function code and provides type information for the parameters. Since functions are external by default, we declare them (in the calling functions) without the qualifier **extern**. Therefore, the declaration

```
void print_out(void);
```

is equivalent to

```
extern void print_out(void);
```

Function declarations outside of any function behave the same way as variable declarations.

3A.17.3 Static Variables

As the name suggests, the value of static variables persists until the end of the program. A variable can be declared *static* using the keyword **static** like

```
static int x;
static float y;
```

A static variable may be either an internal type or an external type depending on the place of declaration.

Internal static variables are those which are declared inside a function. The scope of internal static variables extend up to the end of the function in which they are defined. Therefore, internal **static** variables are similar to **auto** variables, except that they remain in existence (alive) throughout the remainder of the program. Therefore, internal **static** variables can be used to retain values between function calls. For example, it can be used to count the number of calls made to a function.

Example 3A.7 Write a program to illustrate the properties of a static variable.

The program in Fig. 3A.13 explains the behavior of a static variable.

```
Program
void stat(void);
main ( )
{
```

```

        int i;
        for(i=1; i<=3; i++)
            stat( );
    }
    void stat(void)
    {
        static int x = 0;

        x = x+1;
        printf("x = %d\n", x);
    }

```

Output

```

x = 1
x = 2
x = 3

```

Fig. 3A.13 Illustration of static variable

A static variable is initialized only once, when the program is compiled. It is never initialized again. During the first call to **stat**, **x** is incremented to 1. Because **x** is static, this value persists and therefore, the next call adds another 1 to **x** giving it a value of 2. The value of **x** becomes three when the third call is made.

Had we declared **x** as an **auto** variable, the output would have been:

```

x = 1
x = 1
x = 1

```

This is because each time **stat** is called, the auto variable **x** is initialized to zero. When the function terminates, its value of 1 is lost.

An external **static** variable is declared outside of all functions and is available to all the functions in that program. The difference between a **static** external variable and a simple external variable is that the **static** external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

It is also possible to control the scope of a function. For example, we would like a particular function accessible only to the functions in the file in which it is defined, and not to any function in other files. This can be accomplished by defining ‘that’ function with the storage class **static**.

3A.17.4 Register Variables

We can tell the compiler that a variable should be kept in one of the machine’s registers, instead of keeping in the memory (where normal variables are stored). Since a register access is much faster than a memory access, keeping the frequently accessed variables (e.g. loop control variables) in the register will lead to faster execution of programs. This is done as follows:

```

register int count;

```

Although, ANSI standard does not restrict its application to any particular data type, most compilers allow only **int** or **char** variables to be placed in the register.

Since only a few variables can be placed in the register, it is important to carefully select the variables for this purpose. However, C will automatically convert **register** variables into non-register variables once the limit is reached.

Table 3A.1 summarizes the information on the visibility and lifetime of variables in functions and files.

TABLE 3A.1 Scope and Lifetime of Variables

Storage Class	Where declared	Visibility (Active)	Lifetime (Alive)
None	Before all functions in a file (may be initialized)	Entire file plus other files where variable is declared with extern	Entire program (Global)
extern	Before all functions in a file (cannot be initialized) extern and the file where originally declared as global.	Entire file plus other files where variable is declared	Global
static	Before all functions in a file	Only in that file	Global
None or auto	Inside a function (or a block)	Only in that function or block	Until end of function or block
register	Inside a function or block	Only in that function or block	Until end of function or block
static	Inside a function	Only in that function	Global

3A.17.5 Nested Blocks

A set of statements enclosed in a set of braces is known a *block* or a *compound statement*. Note that all functions including the **main** use compound *statement*. A block can have its own declarations and other statements. It is also possible to have a block of such statements inside the body of a function or another block, thus creating what is known as *nested blocks* as shown below:

```

main ( )
{
    int a = 20;
    int b = 10;
    . . . . .
    {
        int a = 0;
        int c = a + b;
        . . . . .
    }
    b = a;
}

```

Outer block

Inner block

When this program is executed, the value *c* will be 10, not 30. The statement *b = a;* assigns a value of 20 to **b** and not zero. Although the scope of **a** extends up to the end of **main** it is not “visible” inside the inner block where the variable **a** has been declared again. The inner **a** hides the visibility of the outer **a** in the inner block. However, when we leave the inner block, the inner **a** is no longer in scope and the outer **a** becomes visible again.

Remember, the variable **b** is not re-declared in the inner block and therefore it is visible in both the blocks. That is why when the statement

```
int c = a + b;
```

is evaluated, **a** assumes a values of 0 and **b** assumes a value of 10.

Although **main**’s variables are visible inside the nested block, the reverse is not true.

Scope Rules

Scope

The region of a program in which a variable is available for use.

Visibility

The program's ability to access a variable from the memory.

Lifetime

The lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

Rules of use

1. The scope of a global variable is the entire program file.
2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.
3. The scope of a formal function argument is its own function.
4. The lifetime (or longevity) of an **auto** variable declared in **main** is the entire program execution time, although its scope is only the **main** function.
5. The life of an **auto** variable declared in a function ends when the function is exited.
6. A **static** local variable, although its scope is limited to its function, its lifetime extends till the end of program execution.
7. All variables have visibility in their scope, provided they are not declared again.
8. If a variable is redeclared within its scope again, it loses its visibility in the scope of the redeclared variable.

3A.18 MULTIFILE PROGRAMS

So far we have been assuming that all the functions (including the **main**) are defined in one file. However, in real-life programming environment, we may use more than one source files which may be compiled separately and linked later to form an executable object code. This approach is very useful because any change in one file does not affect other files thus eliminating the need for recompilation of the entire program.

Multiple source files can share a variable provided it is declared as an external variable appropriately. Variables that are shared by two or more files are global variables and therefore we must declare them accordingly in one file and then explicitly define them with **extern** in other files. Figure 3A.16 illustrates the use of **extern** declarations in a multifile program.

The function **main** in **file1** can reference the variable **m** that is declared as global in **file2**. Remember, **function1** cannot access the variable **m**. If, however, the **extern int m;** statement is placed before **main**, then both the functions could refer to **m**. This can also be achieved by using **extern int m;** statement inside each function in **file1**.

The **extern** specifier tells the compiler that the following variable types and names have already been declared elsewhere and no need to create storage space for them. It is the responsibility of the *linker* to resolve the reference problem. It is important to note that a multifile global variable should be declared *without* **extern** in one (and only one) of the files. The **extern** declaration is done in places where secondary references are made. If we declare a variable as global in two different files used by a single program, then the linker will have a conflict as to which variable to use and, therefore, issues a warning.

<pre> file1.c main() { extern int m; int i; } </pre>	<pre> file2.c int m /* global variable */ function2() { int i; } </pre>
---	--

<pre> { int j; } </pre>	<pre> { int count; } </pre>
---	---

Fig. 3A.14 Use of extern in a multifile program

The multifile program shown in Fig. 3A.14 can be modified as shown in Fig. 3A.15.

<pre> file1.c int m; /* global variable */ main() { { int i; } function1() { int j; } } </pre>	<pre> file2.c extern int m; function2() { int i; } function3() { int count; } </pre>
--	--

Fig. 3A.15 Another version of a multifile program

When a function is defined in one file and accessed in another, the later file must include a function *declaration*. The declaration identifies the function as an external function whose definition appears elsewhere. We usually place such declarations at the beginning of the file, before all functions. Although all functions are assumed to be external, it would be a good practice to explicitly declare such functions with the storage class **extern**.

3A.19 PREPROCESSOR COMMANDS

Preprocessor commands, as the name suggests, are the instructions that are executed before the source code passes through the compiler. The program that processes the preprocessor command lines or directives is called a preprocessor.

Preprocessor directives are placed in the source program before the main line. Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions (as per the directives) are taken and then the source program is handed over to the compiler.

Preprocessor directives follow special syntax rules that are different from the normal C syntax. They all begin with the symbol # in column one and do not require a semicolon at the end.

Table 3A.2 lists some of the key preprocessor directives and their functions:

TABLE 3A.1 Preprocessor directives

Directive	Function
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies the files to be included
#ifdef	Test for a macro definition
#endif	Specifies the end of #if.
#ifndef	Tests whether a macro is not defined.
#if	Test a compile-time condition
#else	Specifies alternatives when #if test fails.

The preprocessor directives are broadly classified under three categories:

- Macro substitution
- File inclusion
- Conditional inclusion

3A.19.1 Macro Substitution Directives

Macro substitution is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. The preprocessor accomplishes this task under the direction of **#define** statement. This statement, usually known as a macro definition (or simply a macro) takes the following general form:

#define identifier string

If this statement is included in the program at the beginning, then the preprocessor replaces every occurrence of the identifier in the source code by the string. The keyword **#define** is written just as shown (starting from the first column) followed by the identifier and a string, with at least one blank space between them. Note that the definition is not terminated by a semicolon. The string may be any text, while the identifier must be a valid C name.

There are different forms of macro substitution. The most common forms are:

1. Simple macro substitution.
2. Argumented macro substitution.
3. Nested macro substitution.

Simple Macro Substitution

Simple string replacement is commonly used to define constants. Examples of definition of constants are:

```
#define COUNT 100
#define FALSE 0
#define SUBJECTS 6
#define PI 3.1415926
#define CAPITAL "DELHI"
```

Notice that we have written all macros (identifiers) in capitals. It is a convention to write all macros in capitals to identify them as symbolic constants. A definition, such as

```
#define M 5
```

will replace all occurrences of M with 5, starting from the line of definition to the end of the program. However, a macro inside a string does not get replaced. Consider the following two lines:

3A.34 Computer Programming

```
total = M * value;  
printf("M = %d\n", M);
```

These two lines would be changed during preprocessing as follows:

```
total = 5 * value;  
printf("M = %d\n", 5);
```

Notice that the string "**M=%d\n**" is left unchanged.

A macro definition can include more than a simple constant value. It can include expressions as well. Following are valid definitions:

```
#define AREA          5 * 12.46  
#define SIZE         sizeof(int) * 4  
#define TWO-PI       2.0 * 3.1415926
```

Argumented Macro Substitution

The preprocessor permits us to define more complex and more useful form of replacements. It takes the form:

#define	identifier(f1, f2, fn)	string
---------	----------------------------------	--------

Notice that there is no space between the macro *identifier* and the left parentheses. The identifiers f1, f2,fn are the formal macro arguments that are analogous to the formal arguments in a function definition.

There is a basic difference between the simple replacement discussed above and the replacement of macros with arguments. Subsequent occurrence of a macro with arguments is known as a *macro call* (similar to a function call). When a macro is called, the preprocessor substitutes the string, replacing the formal parameters with the actual parameters. Hence, the string behaves like a template.

A simple example of a macro with arguments is

```
#define CUBE(x)      (x*x*x)
```

If the following statement appears later in the program

```
volume = CUBE(side);
```

Then the preprocessor would expand this statement to:

```
volume = (side * side * side );
```

Nested Macro Substitution

Nesting of Macros

We can also use one macro in the definition of another macro. That is, macro definitions may be nested. For instance, consider the following macro definitions.

```
#define M          5  
#define N          M+1  
#define SQUARE(x)  ( x ) * ( x )  
#define CUBE(x)     ( SQUARE (x) * (x) )  
#define SIXTH(x)    ( CUBE(x) * CUBE(x) )
```

The preprocessor expands each **#define** macro, until no more macros appear in the text. For example, the last definition is first expanded into

$$((\text{SQUARE}(x) * (x)) * (\text{SQUARE}(x) * (x)))$$

Since **SQUARE (x)** is still a macro, it is further expanded into

$$((((x)*(x)) * (x)) * (((x) * (x)) * (x)))$$

which is finally evaluated as x^6 .

3A.19.2 File Inclusion Directive

File Inclusion

An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions. This is achieved by the preprocessor directive

#include "filename"

where filename is the name of the file containing the required definitions or functions. At this point, the preprocessor inserts the entire contents of filename into the source code of the program. When the filename is included within the double quotation marks, the search for the file is made first in the current directory and then in the standard directories.

Alternatively this directive can take the form

#include <filename>

without double quotation marks. In this case, the file is searched only in the standard directories.

Nesting of included files is allowed. That is, an included file can include other files. However, a file cannot include itself.

If an included file is not found, an error is reported and compilation is terminated.

Let us assume that we have created the following three files:

SYNTAX.C	contains syntax definitions.
STAT.C	contains statistical functions.
TEST.C	contains test functions.

We can make use of a definition or function contained in any of these files by including them in the program as:

```
#include      <stdio.h>
#include      "SYNTAX.C"
#include      "STAT.C"
#include      "TEST.C"
#define      M      100
main ()
{
    -----
    -----
    -----
}
```

3A.19.3 Conditional Inclusion Directives

Some directives are used to compile a part of the program based on certain conditions. It means that if a specific condition is fulfilled, then the program will be executed otherwise not. These conditional directives are # if, # ifdef, # ifndef, # elif, # else and # endif, as explained below:

3A.36 Computer Programming

#ifdef This directive is used to compile a part of the program only if the macro is defined as a parameter in the program irrespective of its value. The use of **#ifdef** can be explained with the help of the following example:

```
#ifdef max_marks
int marks[max_marks];
#endif
```

In the above example, the second statement will be executed only if the macro `max_marks` has been defined before in the program. If it has not been defined previously, then the `int` statement will not be included in the program execution.

#ifndef This directive is the opposite of **#ifdef** directive. It means that the statements written between **#ifndef** and **#endif** will be executed only if the identifier has not been defined previously. The use of **#ifndef** can be explained with the help of the following example:

```
#ifndef max_marks
#define max_marks 100
#endif
```

According to this directive, the second statement will be executed only if the macro `max_marks` has not been defined earlier.

#if, #else, #elif These directives are used in between a program to allow or prevent the further execution of the program. **#if** is used to check whether the result of a given expression is zero or not. If the result is not zero, then the statements after **#if** are compiled else not. The use of **#if**, **#else** and **#elif** can be explained with the help of the following example:

```
main()
{
    #if AGE<=18

    printf("Juvenile");
    #elif AGE>=18 AGE<=50
    printf("Adult");
    #else
    printf("Aged");
    #endif
}
```

3A.19.4 Additional Preprocessor Directives

C language also supports certain additional preprocessor directives, such as:

- **#error** This directive stops the compilation of the program when the compiler encounters an error (specified as a parameter).
- **#line** This directive allows both the file name and the line number at which the compilation error occurred to be displayed by the compiler.
- **#pragma** This directive allows program implementation related instructions to be given to the compiler.
- **#elif** This directive helps realize an “if..else..if” sequence for testing multiple conditions.



Just Remember

- It is a syntax error if the types in the declaration and function definition do not match.
- It is a syntax error if the number of actual parameters in the function call do not match the number in the declaration statement.
- It is a logic error if the parameters in the function call are placed in the wrong order.
- It is illegal to use the name of a formal argument as the name of a local variable.
- Using **void** as return type when the function is expected to return a value is an error.
- Trying to return a value when the function type is marked **void** is an error.
- Variables in the parameter list must be individually declared for their types. We cannot use multiple declarations (like we do with local or global variables).
- A **return** statement is required if the return type is anything other than **void**.
- If a function does not return any value, the return type must be declared **void**.
- If a function has no parameters, the parameter list must be declared **void**.
- Placing a semicolon at the end of header line is illegal.
- Forgetting the semicolon at the end of a prototype declaration is an error.
- Defining a function within the body of another function is not allowed.
- It is an error if the type of data returned does not match the return type of the function.
- It will most likely result in logic error if there is a mismatch in data types between the actual and formal arguments.
- Functions return integer value by default.
- A function without a return statement cannot return a value, when the parameters are passed by value.
- A function that returns a value can be used in expressions like any other C variable.
- When the value returned is assigned to a variable, the value will be converted to the type of the variable receiving it.
- Function cannot be the target of an assignment.
- A function with void return type cannot be used in the right-hand side of an assignment statement. It can be used only as a stand-alone statement.
- A function that returns a value cannot be used as a stand-alone statement.
- A **return** statement can occur anywhere within the body of a function.
- A function can have more than one return statement.
- A function definition may be placed either after or before the **main** function.
- Where more functions are used, they may be placed in any order.
- A global variable used in a function will retain its value for future use.
- A local variable defined inside a function is known only to that function. It is destroyed when the function is exited.
- A global variable is visible only from the point of its declaration to the end of the program.
- When a variable is redeclared within its scope either in a function or in a block, the original variable is not visible within the scope of the redeclared variable.
- A local variable declared **static** retains its value even after the function is exited.
- Static variables are initialized at compile time and therefore they are initialized only once.
- Use parameter passing by values as far as possible to avoid inadvertent changes to variables of calling function in the called function.
- Although not essential, include parameter names in the prototype declarations for documentation purposes.

- Avoid the use of names that hide names in outer scope.
- The use of functions gives C a modular-based programming approach.
- A function is typically characterised by its name, type and parameters.
- It is a good practice to name a function on the lines of its functionality.
- A function can return multiple values.
- In case of 'pass by reference' function call, the addresses of the variables are passed.
- Make sure that the parameters of a function are declared as pointer types for implementing pass by reference technique.



Multiple Choice Questions

1. The default return type of a function is _____
 (a) void (b) int
 (c) float (d) char
2. Which of the following are the incorrect function declarations?
 (a) `int funct(int a, b);`
 (b) `int funct(int a, int b);`
 (c) `int funct(int , int);`
 (d) `int funct(int ,);`
3. Which of the following is not feasible?
 (a) Functions with no arguments and no return values
 (b) Functions with arguments and no return values
 (c) Functions with no arguments but a return value
 (d) All of the above are feasible
4. 'Call by reference' function call uses the following type of parameter:
 (a) Pointer variables
 (b) Integer variables
 (c) Address variables
 (d) Memory variables
5. Recursion is a situation where:
 (a) A function calls the main function
 (b) A function calls any of the system functions
 (c) A function calls itself
 (d) None of the above
6. Which of the following is not a variable storage class?
 (a) automatic (b) external
 (c) static (d) dynamic
7. Which of the following keywords is used for declaring an external variable?
 (a) external (b) extern
 (c) auto extern (d) ext
8. Which of the following types of variables remain alive for the entire life time of a program?
 (a) extern (b) auto
 (c) static (d) register
9. Which of the following refers to the region of a program where a variable is available for use?
 (a) scope (b) visibility
 (c) life time (d) None of the above
10. The formal arguments in the function header must be prefixed by which of the following indirection operator?
 (a) * (b) +
 (c) - (d) /
11. Which of the following is optional in a function definition?
 (a) Function name
 (b) Function type
 (c) Local variable declaration
 (d) Return statement



Case Study

Calculation of Area under a Curve

One of the applications of computers in numerical analysis is computing the area under a curve. One simple method of calculating the area under a curve is to divide the area into a number of trapezoids of same width and summing up the area of individual trapezoids. The area of a trapezoid is given by

$$\text{Area} = 0.5 * (h_1 + h_2) * b$$

where h_1 and h_2 are the heights of two sides and b is the width as shown in Fig. 3A.16.

The program in Fig. 3A.18 calculates the area for a curve of the function

$$f(x) = x^2 + 1$$

between any two given limits, say, A and B.

Input

Lower limit (A)

Upper limit (B)

Number of trapezoids

Output

Total area under the curve between the given limits.

Algorithm

1. Input the lower and upper limits and the number of trapezoids.
2. Calculate the width of trapezoids.
3. Initialize the total area.
4. Calculate the area of trapezoid and add to the total area.
5. Repeat step-4 until all the trapezoids are completed.
6. Print total area.

The algorithm is implemented in top-down modular form as in Fig. 3A.17.

The evaluation of $f(x)$ has been done using a separate function so that it can be easily modified to allow other functions to be evaluated.

The output for two runs shows that better accuracy is achieved with larger number of trapezoids. The actual area for the limits 0 and 3 is 12 units (by analytical method).

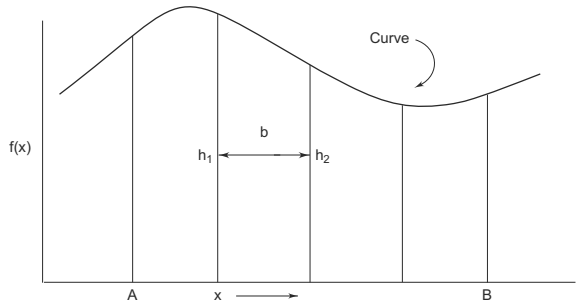


Fig. 3A.16 Area under a curve

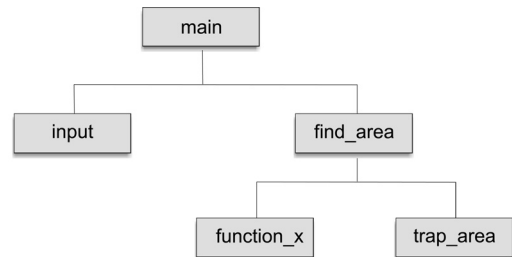


Fig. 3A.17 Modular chart

Program

```

#include <stdio.h>
float  start_point,          /* GLOBAL VARIABLES */
       end_point,
       total_area;
int    numtraps;
  
```

3A.40 Computer Programming

```
main( )
{
    void    input(void);
    float   find_area(float a,float b,int n); /* prototype */

    print("AREA UNDER A CURVE");
    input( );
    total_area = find_area(start_point, end_point, numtraps);
    printf("TOTAL AREA = %f", total_area);
}

void input(void)
{
    printf("\n Enter lower limit:");
    scanf("%f", &start_point);
    printf("Enter upper limit:");
    scanf("%f", &end_point);
    printf("Enter number of trapezoids:");
    scanf("%d", &numtraps);
}

float find_area(float a, float b, int n)
{
    float base, lower, h1, h2; /* LOCAL VARIABLES */
    float function_x(float x); /* prototype */
    float trap_area(float h1,float h2,float base);/*prototype*/

    base = (b-1)/n;
    lower = a;

    for(lower =a; lower <= b-base; lower = lower + base)
    {
        h1  = function_x(lower);
        h1  = function_x(lower + base);
        total_area += trap_area(h1, h2, base);
    }

    return(total_area);
}

float trap_area(float height_1,float height_2,float base)
{
    float area;      /* LOCAL VARIABLE */
    area = 0.5 * (height_1 + height_2) * base;
    return(area);
}

float function_x(float x)
{
    /* F(X) = X * X + 1 */

    return(x*x + 1);
}
```

Output

```
AREA UNDER A CURVE
Enter lower limit: 0
Enter upper limit: 3
Enter number of trapezoids: 30
TOTAL AREA = 12.005000
```

```

AREA UNDER A CURVE
Enter lower limit: 0
Enter upper limit: 3
Enter number of trapezoids: 100
TOTAL AREA = 12.000438

```

Fig. 3A.18 Computing area under a curve



Review Questions

3A.1 State whether the following statements are *true* or *false*.

- (a) C functions can return only one value under their function name.
- (b) A function in C should have at least one argument.
- (c) A function can be defined and placed before the **main** function.
- (d) A function can be defined within the **main** function.
- (e) An user-defined function must be called at least once; otherwise a warning message will be issued.
- (f) Any name can be used as a function name.
- (g) Only a **void** type function can have **void** as its argument.
- (h) When variable values are passed to functions, a copy of them are created in the memory.
- (i) Program execution always begins in the main function irrespective of its location in the program.
- (j) Global variables are visible in all blocks and functions in the program.
- (k) A function can call itself.
- (l) A function without a **return** statement is illegal.
- (m) Global variables cannot be declared as **auto** variables.
- (n) A function prototype must always be placed outside the calling function.
- (o) The return type of a function is **int** by default.

- (p) The variable names used in prototype should match those used in the function definition.
- (q) In parameter passing by pointers, the formal parameters must be prefixed with the symbol ***** in their declarations.
- (r) In parameter passing by pointers, the actual parameters in the function call may be variables or constants.
- (s) In passing arrays to functions, the function call must have the name of the array to be passed without brackets.
- (t) In passing strings to functions, the actual parameter must be name of the string post-fixed with size in brackets.

3A.2 Fill in the blanks in the following statements.

- (a) The parameters used in a function call are called _____.
- (b) A variable declared inside a function is called _____.
- (c) By default, _____ is the return type of a C function.
- (d) In passing by pointers, the variables of the formal parameters must be prefixed with **in** _____ their declaration.
- (e) In prototype declaration, specifying is _____ optional.
- (f) _____ refers to the region where a variable is actually available for use.
- (g) A function that calls itself is known as a _____ function.
- (h) If a local variable has to retain its value between calls to the function, it must be declared as _____.

3A.42 Computer Programming

- (i) A _____ aids the compiler to check the matching between the actual arguments and the formal ones.
- (j) A variable declared inside a function by default assumes _____ storage class.
- 3A.3 The **main** is a user-defined function. How does it differ from other user-defined functions?
- 3A.4 Describe the two ways of passing parameters to functions. When do you prefer to use each of them?
- 3A.5 What is prototyping? Why is it necessary?
- 3A.6 Distinguish between the following:
- (a) Actual and formal arguments
 - (b) Global and local variables
 - (c) Automatic and static variables
 - (d) Scope and visibility of variables
 - (e) & operator and * operator
- 3A.7 Explain what is likely to happen when the following situations are encountered in a program.
- (a) Actual arguments are less than the formal arguments in a function.
 - (b) Data type of one of the actual arguments does not match with the type of the corresponding formal argument.
 - (c) Data type of one of the arguments in a prototype does not match with the type of the corresponding formal parameter in the header line.
 - (d) The order of actual parameters in the function call is different from the order of formal parameters in a function where all the parameters are of the same type.
 - (e) The type of expression used in **return** statement does not match with the type of the function.
- 3A.8 Which of the following prototype declarations are invalid? Why?
- (a) `int (fun) void;`
 - (b) `double fun (void)`
 - (c) `float fun (x, y, n);`
 - (d) `void fun (void, void);`
 - (e) `int fun (int a, b);`
 - (f) `fun (int, float, char);`
 - (g) `void fun (int a, int &b);`
- 3A.9 Which of the following header lines are invalid? Why?
- (a) `float average (float x, float y, float z);`
 - (b) `double power (double a, int n - 1)`
 - (c) `int product (int m, 10)`
 - (d) `double minimum (double x; double y;)`
 - (e) `int mul (int x, y)`
 - (f) `exchange (int *a, int *b)`
 - (g) `void sum (int a, int b, int &c)`
- 3A.10 Find errors, if any, in the following function definitions:
- (a)

```
void abc (int a, int b)
{
    int c;
    . . . .
    return (c);
}
```
 - (b)

```
int abc (int a, int b)
{
    . . . .
    . . . .
}
```
 - (c)

```
int abc (int a, int b)
{
    double c = a + b;
    return (c);
}
```
 - (d)

```
void abc (void)
{
    . . . .
    . . . .
    return;
}
```
 - (e)

```
int abc(void)
{
    . . . .
    . . . .
    return;
}
```

3A.11 Find errors in the following function calls:

- (a) void xyz ();
- (b) xyx (void);
- (c) xyx (int x, int y);
- (d) xyz ();
- (e) xyz () + xyz ();

3A.12 A function to divide two floating point numbers is as follows:

```
divide (float x, float y)
{
    return (x / y);
}
```

What will be the value of the following function calls:

- (a) divide (10, 2)
- (b) divide (9, 2)
- (c) divide (4.5, 1.5)
- (d) divide (2.0, 3.0)

3A.13 What will be the effect on the above function calls if we change the header line as follows:

- (a) int divide (int x, int y)
- (b) double divide (float x, float y)

3A.14 Determine the output of the following program?

```
int prod( int m, int n);
main ( )
{
    int x = 10;
    int y = 20;
    int p, q;
    p = prod (x,y);
    q = prod (p, prod (x,z));
    printf ("%d %d\n", p,q);
}

int prod( int a, int b)
{
    return (a * b);
}
```

3A.15 What will be the output of the following program?

```
void test (int *a);
main ( )
{
    int x = 50;
    test ( &x);
    printf ("%d\n", x);
}

void test (int *a);
{
    *a = *a + 50;
}
```

3A.16 The function **test** is coded as follows:

```
int test (int number)
{
    int m, n = 0;
    while (number)
    {
        m = number % 10;
        if (m % 2)

            n = n + 1;
            number
            = number /10;
    }
    return (n);
}
```

What will be the values of **x** and **y** when the following statements are executed?

```
int x = test (135);
int y = test (246);
```

3A.17 Enumerate the rules that apply to a function call.

3A.18 Summarize the rules for passing parameters to functions by pointers.

3A.19 What are the rules that govern the passing of arrays to function?

3A.20 State the problems we are likely to encounter when we pass global variables as parameters to functions.



Programming Exercises

3A.1 Write a function **exchange** to interchange the values of two variables, say **x** and **y**. Illustrate the use of this function, in a calling function. Assume that **x** and **y** are defined as global variables.

3A.2 Write a function **space(x)** that can be used to provide a space of **x** positions between two output numbers. Demonstrate its application.

3A.3 Use recursive function calls to evaluate

$$f(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

3A.4 An n _order polynomial can be evaluated as follows:

$$P = (\dots(((a_0x+a_1)x+a_2)x+a_3)x+\dots+a_n)$$

Write a function to evaluate the polynomial, using an array variable. Test it using a main program.

3A.5 The Fibonacci numbers are defined recursively as follows:

$$F_1 = 1$$

$$F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}, n > 2$$

Write a function that will generate and print the first n Fibonacci numbers. Test the function for $n = 5, 10$, and 15 .

3A.6 Write a function that will round a floating-point number to an indicated decimal place. For example the number 17.457 would yield the value 17.46 when it is rounded off to two decimal places.

3A.7 Write a function **prime** that returns 1 if its argument is a prime number and returns zero otherwise.

3A.8 Write a function that will scan a character string passed as an argument and convert all lowercase characters into their uppercase equivalents.

3A.9 Develop a top_down modular program to implement a calculator. The program should request the user to input two numbers and display one of the following as per the desire of the user:

(a) Sum of the numbers

(b) Difference of the numbers

(c) Product of the numbers

(d) Division of the numbers

Provide separate functions for performing various tasks such as reading, calculating and displaying. Calculating module should call second level modules to perform the individual mathematical operations. The main function should have only function calls.

3A.10 Develop a modular interactive program using functions that reads the values of three sides of a triangle and displays either its area or its perimeter as per the request of the user. Given the three sides a, b and c .

$$\text{Perimeter} = a + b + c$$

$$\text{Area} = \sqrt{(s-a)(s-b)(s-c)}$$

$$\text{where } s = (a+b+c)/2$$

3A.11 Write a function that can be called to find the largest element of an m by n matrix.

3A.12 Write a function that can be called to compute the product of two matrices of size m by n and n by m . The main function provides the values for m and n and two matrices.

3A.13 Design and code an interactive modular program that will use functions to a matrix of m by n size, compute column averages and row averages, and then print the entire matrix with averages shown in respective rows and columns.

3A.14 Develop a top-down modular program that will perform the following tasks:

(a) Read two integer arrays with unsorted elements.

(b) Sort them in ascending order

(c) Merge the sorted arrays

(d) Print the sorted list

Use functions for carrying out each of the above tasks. The main function should have only function calls.

- 3A.15 Develop your own functions for performing following operations on strings:
- (a) Copying one string to another
 - (b) Comparing two strings
 - (c) Adding a string to the end of another string
- Write a driver program to test your functions.
- 3A.16 Write a program that invokes a function called **find()** to perform the following tasks:
- (a) Receives a character array and a single character.
 - (b) Returns 1 if the specified character is found in the array, 0 otherwise.
- 3A.17 Design a function **locate()** that takes two character arrays **s1** and **s2** and one integer value **m** as parameters and inserts the string **s2** into **s1** immediately after the index **m**.
Write a program to test the function using a real-life situation. (**Hint:** s2 may be a missing word in s1 that represents a line of text.)
- 3A.18 Write a function that takes an integer parameter **m** representing the month number of the year and returns the corresponding name of the month. For instance, if $m = 3$, the month is March.
Test your program.
- 3A.19 In preparing the calendar for a year we need to know whether that particular year is leap year or not. Design a function **leap()** that receives the year as a parameter and returns an appropriate message.
What modifications are required if we want to use the function in preparing the actual calendar?
- 3A.20 Write a function that receives a floating point value **x** and returns it as a value rounded to two nearest decimal places. For example, the value 123.4567 will be rounded to 123.46.
(**Hint:** Seek help of one of the math functions available in math library.)



Key Terms

- **Modular programming:** It is a programming approach of organizing a large program into small, independent program segments known as modules.
- **Program definition:** It is an independent program module that is specially written to implement the requirements of the function.
- **Calling program:** It is the program that calls the function.
- **Function type:** It specifies the type of value that the function is supposed to return.
- **Parameter list:** It declares the variables that will receive the data sent by the calling program.
- **Function body:** It contains the declarations and statements necessary for performing the required task.
- **Local variables:** The variables declared inside a function are known as local variables.
- **Recursion:** It is the process in which a called function in turn calls another function.
- **Internal variables:** These are the variables, which are declared within a particular function.
- **External variables:** These variables are declared outside of any function. These are alive and active throughout the entire program.
- **Block statement:** It is a set of statements enclosed in a set of braces.

UNIT

3B

Arrays

3B.1 INTRODUCTION

So far we have used only the fundamental data types, namely **char**, **int**, **float**, **double** and variations of **int** and **double**. Although these types are very useful, they are constrained by the fact that a variable of these types can store only one value at any given time. Therefore, they can be used only to handle limited amounts of data. In many applications, however, we need to handle a large volume of data in terms of reading, processing and printing. To process such large amounts of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation of data items. C supports a derived data type known as *array* that can be used for such applications.

An array is a *fixed-size* sequenced collection of elements of the same data type. It is simply a grouping of like-type data. In its simplest form, an array can be used to represent a list of numbers, or a list of names. Some examples where the concept of an array can be used:

- List of temperatures recorded every hour in a day, or a month, or a year.
- List of employees in an organization.
- List of products and their cost sold by a store.
- Test scores of a class of students.
- List of customers and their telephone numbers.
- Table of daily rainfall data.

and so on.

Since an array provides a convenient structure for representing data, it is classified as one of the *data structures* in C. Other data structures include structures, lists, queues and trees. A complete discussion of all data structures is beyond the scope of this text.

As we mentioned earlier, an array is a sequenced collection of related data items that share a common name. For instance, we can use an array name *salary* to represent a *set of salaries* of a group of employees in an organization. We can refer to the individual salaries by writing a number called *index* or *subscript* in brackets after the array name. For example,

3B.2 Computer Programming

salary [10]

represents the salary of 10th employee. While the complete set of values is referred to as an array, individual values are called *elements*.

The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. For example, we can use a loop construct, discussed earlier, with the subscript as the control variable to read the entire array, perform calculations, and print out the results.

We can use arrays to represent not only simple lists of values but also tables of data in two, three or more dimensions. In this chapter, we introduce the concept of an array and discuss how to use it to create and apply the following types of arrays.

- One-dimensional arrays
- Two-dimensional arrays
- Multidimensional arrays

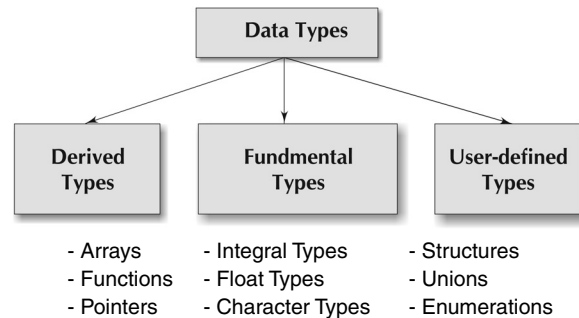
Data Structures

C supports a rich set of derived and user-defined data types in addition to a variety of fundamental types as shown below:

Arrays and structures are referred to as *structured data types* because they can be used to represent data values that have a structure of some sort. Structured data types provide an organizational scheme that shows the relationships among the individual elements and facilitate efficient data manipulations. In programming parlance, such data types are known as *data structures*.

In addition to arrays and structures, C supports creation and manipulation of the following data structures:

- Linked Lists
- Stacks
- Queues
- Trees



3B.2 ONE-DIMENSIONAL ARRAYS

A list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted variable* or a *one-dimensional* array. In mathematics, we often deal with variables that are single-subscripted. For instance, we use the equation

$$A = \frac{\sum_{i=1}^n x_i}{n}$$

to calculate the average of n values of x . The subscripted variable x_i refers to the i th element of x . In C, single-subscripted variable x_i can be expressed as

x[1], x[2], x[3],.....x[n]

The subscript can begin with number 0. That is

x[0]

is allowed. For example, if we want to represent a set of five numbers, say (35,40,20,57,19), by an array variable **number**, then we may declare the variable **number** as follows

```
int number[5];
```

and the computer reserves five storage locations as shown below:

	number[0]
	number[1]
	number[2]
	number[3]
	number[4]

The values to the array elements can be assigned as follows:

```
number[0] = 35;
number[1] = 40;
number[2] = 20;
number[3] = 57;
number[4] = 19;
```

This would cause the array **number** to store the values as shown below:

number[0]	35
number[1]	40
number[2]	20
number[3]	57
number[4]	19

These elements may be used in programs just like any other C variable. For example, the following are valid statements:

```
a = number[0] + 10;
number[4] = number[0] + number [2];
number[2] = x[5] + y[10];
value[6] = number[i] * 3;
```

The subscripts of an array can be integer constants, integer variables like *i*, or expressions that yield integers. *C performs no bounds checking and, therefore, care should be exercised to ensure that the array indices are within the declared limits.*

3B.3 DECLARATION OF ONE-DIMENSIONAL ARRAYS

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is

```
type variable-name[ size ];
```

The *type* specifies the type of element that will be contained in the array, such as **int**, **float**, or **char** and the *size* indicates the maximum number of elements that can be stored inside the array. For example,

```
float height[50];
```

declares the **height** to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly,

```
int group[10];
```

declares the **group** as an array to contain a maximum of 10 integer constants. Remember:

3B.4 Computer Programming

- Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.
- The size should be either a numeric constant or a symbolic constant.

The C language treats character strings simply as arrays of characters. The *size* in a character string represents the maximum number of characters that the string can hold. For instance,

```
char name[10];
```

declares the **name** as a character array (string) variable that can hold a maximum of 10 characters. Suppose we read the following string constant into the string variable **name**.

“WELL DONE”

Each character of the string is treated as an element of the array **name** and is stored in the memory as follows:

'W'
'E'
'L'
'L'
"
'D'
'O'
'N'
'E'
'\0'

When the compiler sees a character string, it terminates it with an additional null character. Thus, the element **name[10]** holds the null character ‘\0’. *When declaring character arrays, we must allow one extra element space for the null terminator.*

Example 3B.1 Write a program using a single-subscripted variable to evaluate the following expressions:

$$\text{Total} = \sum_{i=1}^n x_i^2$$

The values of x_1, x_2, \dots are read from the terminal.

Program in Fig. 3B.1 uses a one-dimensional array **x** to read the values and compute the sum of their squares.

```
Program
main()
{
    int i ;
    float x[10], value, total ;

    /* . . . . .READING VALUES INTO ARRAY . . . . . */

    printf("ENTER 10 REAL NUMBERS\n") ;

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

(Contd.)

```

        {
            scanf("%f", &value) ;
            x[i] = value ;
        }
/* . . . . .COMPUTATION OF TOTAL . . . . . */

total = 0.0 ;
for( i = 0 ; i < 10 ; i++ )
    total = total + x[i] * x[i] ;

/* . . . . PRINTING OF x[i] VALUES AND TOTAL . . . */

printf("\n");
for( i = 0 ; i < 10 ; i++ )
    printf("x[%2d] = %5.2f\n", i+1, x[i]) ;

    printf("\ntotal = %5.2f\n", total) ;
}

```

Output

```

ENTER 10 REAL NUMBERS

1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9 10.10

x[ 1] = 1.10
x[ 2] = 2.20
      x[ 3] = 3.30
      x[ 4] = 4.40
      x[ 5] = 5.50
      x[ 6] = 6.60
      x[ 7] = 7.70
      x[ 8] = 8.80
      x[ 9] = 9.90
      x[10] = 10.10

      Total = 446.86

```

Fig. 3B.1 Program to illustrate one-dimensional array

NOTE: C99 permits arrays whose size can be specified at run time. See Appendix ‘C99 Features’.

3B.4 INITIALIZATION OF ONE-DIMENSIONAL ARRAYS

After an array is declared, its elements must be initialized. Otherwise, they will contain “garbage”. An array can be initialized at either of the following stages:

- At compile time
- At run time

3B.4.1 Compile Time Initialization

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

`type array-name[size] = { list of values };`

The values in the list are separated by commas. For example, the statement

```
int number[3] = { 0,0,0 };
```

will declare the variable **number** as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically. For instance,

```
float total[5] = {0.0,15.75,-10};
```

will initialize the first three elements to 0.0, 15.75, and -10.0 and the remaining two elements to zero.

The *size* may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, the statement

```
int counter[ ] = {1,1,1,1};
```

will declare the **counter** array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

Character arrays may be initialized in a similar manner. Thus, the statement

```
char name[ ] = {'J','o','h','n','\0'};
```

declares the **name** to be an array of five characters, initialized with the string “John” ending with the null character. Alternatively, we can assign the string literal directly as under:

```
char name [ ] = “John”;
```

(Character arrays and strings are discussed in detail in Chapter 10.)

Compile time initialization may be partial. That is, the number of initializers may be less than the declared size. In such cases, the remaining elements are initialized to *zero*, if the array type is numeric and *NULL* if the type is char. For example,

```
int number [5] = {10, 20};
```

will initialize the first two elements to 10 and 20 respectively, and the remaining elements to 0. Similarly, the declaration.

```
char city [5] = {'B'};
```

will initialize the first element to ‘B’ and the remaining four to NULL. It is a good idea, however, to declare the size explicitly, as it allows the compiler to do some error checking.

Remember, however, if we have more initializers than the declared size, the compiler will produce an error. That is, the statement

```
int number [3] = {10, 20, 30, 40};
```

will not work. It is illegal in C.

3B.4.2 Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. For example, consider the following segment of a C program.

```

-----
-----
for (i = 0; i < 100; i = i+1)
{
    if    i < 50
        sum[i] = 0.0;          /* assignment statement */
    else
        sum[i] = 1.0;
}
-----
-----

```

The first 50 elements of the array **sum** are initialized to zero while the remaining 50 elements are initialized to 1.0 at run time.

We can also use a read function such as **scanf** to initialize an array. For example, the statements

```

int x [3];
scanf("%d%d%d", &x[0], &x[1], &x[2]);

```

will initialize array elements with the values entered through the keyboard.

Example 3B.2 Given below is the list of marks obtained by a class of 50 students in an annual examination.

```

43 65 51 27 79 11 56 61 82 09 25 36 07 49 55 63 74 81 49 37
40 49 16 75 87 91 33 24 58 78 65 56 76 67 45 54 36 63 12 21
73 49 51 19 39 49 68 93 85 59

```

Write a program to count the number of students belonging to each of following groups of marks: 0–9, 10–19, 20–29,.....,100.

The program coded in Fig. 3B.2 uses the array **group** containing 11 elements, one for each range of marks. Each element counts those values falling within the range of values it represents.

For any value, we can determine the correct group element by dividing the value by 10. For example, consider the value 59. The integer division of 59 by 10 yields 5. This is the element into which 59 is counted.

```

Program
#define MAXVAL    50
#define COUNTER   11
main()
{
    float          value[MAXVAL];
    int            i, low, high;
    int  group[COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};
    /* . . . . .READING AND COUNTING . . . . . */
    for( i = 0 ; i < MAXVAL ; i++ )
    {
        /*. . . . .READING OF VALUES . . . . . */
        scanf("%f", &value[i]) ;
        /*. . . . .COUNTING FREQUENCY OF GROUPS. . . . . */
    }
}

```

(Contd.)

3B.8 Computer Programming

```
        ++ group[ (int) ( value[i]) / 10] ;
    }
    /* . . . .PRINTING OF FREQUENCY TABLE . . . . . */
    printf("\n");
    printf(" GROUP      RANGE      FREQUENCY\n\n") ;
    for( i = 0 ; i < COUNTER ; i++ )
    {
        low = i * 10 ;
        if(i == 10)
            high = 100 ;

        else
            high = low + 9 ;
        printf(" %2d %3d to %3d %d\n",
                i+1, low, high, group[i] ) ;
    }
}
```

Output

43 65 51 27 79 11 56 61 82 09 25 36 07 49 55 63 74
81 49 37 40 49 16 75 87 91 33 24 58 78 65 56 76 67 (Input data)
45 54 36 63 12 21 73 49 51 19 39 49 68 93 85 59

GROUP	RANGE	FREQUENCY
1	0 to 9	2
2	10 to 19	4
3	20 to 29	4
4	30 to 39	5
5	40 to 49	8
6	50 to 59	8
7	60 to 69	7
8	70 to 79	6
9	80 to 89	4
10	90 to 99	2
11	100 to 100	0

Fig. 3B.2 Program for frequency counting

Note that we have used an initialization statement.

```
int group [COUNTER] = {0,0,0,0,0,0,0,0,0,0};
```

which can be replaced by

```
int group [COUNTER] = {0};
```

This will initialize all the elements to zero.

Searching and Sorting

Searching and sorting are the two most frequent operations performed on arrays. Computer Scientists have devised several data structures and searching and sorting techniques that facilitate rapid access to data stored in lists.

Sorting is the process of arranging elements in the list according to their values, in ascending or descending order. A sorted list is called an *ordered list*. Sorted lists are especially important in list searching because they facilitate rapid search operations. Many sorting techniques are available. The three simple and most important among them are:

- Bubble sort
- Selection sort
- Insertion sort

Other sorting techniques include Shell sort, Merge sort and Quick sort.

Searching is the process of finding the location of the specified element in a list. The specified element is often called the *search key*. If the process of searching finds a match of the search key with a list element value, the search said to be successful; otherwise, it is unsuccessful. The two most commonly used search techniques are:

- Sequential search
- Binary search

A detailed discussion on these techniques is beyond the scope of this text. Consult any good book on data structures and algorithms.

Example 3B.3 Write a program for sorting the elements of an array in descending order.

1. Set the size n of an array.
2. Store n elements in the array.
3. Read and store elements of the array in descending order.
4. Print the elements of array in descending order.

Figure 3B.3 gives a program to implement this algorithm.

Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int *arr,temp,i,j,n;
    clrscr();
    printf("Enter the number of elements in the array:");
    scanf("%d",&n);
    arr=(int*)malloc(sizeof(int)*n);
    for(i=0;i<n;i++)
    {
        printf("Enter a number:");
        scanf("%d",&arr[i]);
    }
    for(i=0;i<n;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(arr[i]<arr[j])
            {
                temp=arr[i];
                arr[i]=arr[j];
                arr[j]=temp;
            }
        }
    }
    printf("Elements of array in descending order are:\n");
    for(i=0;i<n;i++)
```

(Contd.)

3B.10 Computer Programming

```
        printf("%d\n",arr[i]);
        getch();
    }
Output
Enter the number of elements in the array:5
Enter a number:32
Enter a number:43
Enter a number:23
Enter a number:57
Enter a number:47
Elements of array in descending order are:
57
47
43
32
23
```

Fig. 3B.3 Program to sort the elements of an array in descending order

Example 3B.4 Write a program for finding the largest number in an array.

1. Set the size of the array as n .
2. Store n elements in the array.
3. Assign the first element of the array to LARGE.
4. Compare each element in the array with LARGE.
5. If an element of the array is greater than LARGE, then set LARGE=element.
6. Else go to step 4.
7. Repeat this process until it reaches the last element of the array.
8. Print the largest element found in the array.

Figure 3B.4 gives a program to implement this algorithm.

```
Program
#include <stdio.h>
#include <conio.h>
void main()
{
    int *arr,i,j,n,LARGE;
    clrscr();
    printf("Enter the number of elements in the array:");
    scanf("%d",&n);
    arr=(int*)malloc(sizeof(int)*n);
    for(i=0;i<n;i++)
    {
        printf("Enter a number:");
        scanf("%d",&arr[i]);
    }
    LARGE=arr[0];
```

(Contd.)

```

    for(i=1;i<n;i++)
    {
        if(arr[i]>LARGE)
            LARGE=arr[i];
    }
    printf("The largest number in the array is: %d",LARGE);
    getch();
}

```

Output

```

Enter the number of elements in the array:5
Enter a number:32
Enter a number:43
Enter a number:23
Enter a number:57
Enter a number:47
The largest number in the array is:57

```

Fig. 3B.4 Program to find the largest element in an array

Example 3B.5 Write a program for removing the duplicate element in an array.

1. Set the size n of an array.
2. Store n elements in the array.
3. Read and store elements of the array in a sorted order.
4. Compare the first element with the next element.
5. If elements are identical, then replace that element by shift complete array.
6. Else start comparing element with the current element.
7. Repeat this process until it reaches the last element of the array.
8. Print the new array without duplicate elements.

A program to implement this is given in Fig. 3B.5.

Program

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main()
{
    int *arr,i,j,n,x,temp;
    clrscr();
    printf("Enter the number of elements in the array:");
    scanf("%d",&n);
    arr=(int*)malloc(sizeof(int)*n);
    for(i=0;i<n;i++)
    {
        printf("Enter a number:");
        scanf("%d",&arr[i]);
    }
    for(i=0;i<n;i++)

```

(Contd.)

3B.12 Computer Programming

```
{
    for(j=i+1;j<n;j++)
    {
        if(arr[i]>arr[j])
        {
            temp=arr[i];
            arr[i]=arr[j];
            arr[j]=temp;
        }
    }
    printf("\nElements of array after sorting:");
    for(i=0;i<n;i++)
        printf("\n%d",arr[i]);
    i=0;
    j=1;
    while(i<n)
    {
        if(arr[i]==arr[j])
        {
            for(x=j;x<n-1;x++)
                arr[x]=arr[x+1];
            n--;
        }
        else
        {
            i++;
            j++;
        }
    }
    printf("\nElements of array after removing duplicate
elements:");
    for(i=0;i<=n;i++)
        printf("\n%d",arr[i]);
    getch();
}
```

Output

Enter the number of elements in the array:5

Enter a number:3

Enter a number:3

Enter a number:4

Enter a number:6

Enter a number:4

Elements of array after sorting:

3

3

4

4

6

Elements of array after removing duplicate elements:

3

4

6

Fig. 3B.5 Program to sort the elements of an array and removing duplicate elements

Example 3B.6 Write a program for finding the desired k^{th} smallest element in an array.

1. Set the size n of an array.
2. Store n elements in the array.
3. Read and store elements of the array in an ascending order.
4. Print the k^{th} elements of sorted array.

A program is given in Fig. 3B.6.

Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int *arr,i,j,n,temp,k;
    clrscr();
    printf("Enter the number of elements in the array:");
    scanf("%d",&n);
    arr=(int*)malloc(sizeof(int)*n);
    for(i=0;i<n;i++)
    {
        printf("Enter a number:");
        scanf("%d",&arr[i]);
    }
    for(i=0;i<n;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(arr[i]>arr[j])
            {
                temp=arr[i];
                arr[i]=arr[j];
                arr[j]=temp;
            }
        }
    }
    printf("\nElements of array after sorting:");
    for(i=0;i<n;i++)
        printf("\n%d",arr[i]);
    printf("\n\nWhich smallest element do you want to determine?");
    scanf("%d",&k);
    if(k<=n)
        printf("\nDesired smallest element is %d.",arr[k-1]);
    else
        printf("Please enter a valid value for finding the particular
smallest element");
    getch();
}
```

Output

```
Enter the number of elements in the array:5
Enter a number:33
```

(Contd.)

3B.14 Computer Programming

```
Enter a number:32
Enter a number:46
Enter a number:68
Enter a number:47
Elements of array after sorting:
32
33
46
47
68
Which smallest element do you want to determine?3
Desired smallest element is 46.
```

Fig. 3B.6 Program to determine K^{th} smallest element in an array

3B.5 TWO-DIMENSIONAL ARRAYS

So far we have discussed the array variables that can store a list of values. There could be situations where a table of values will have to be stored. Consider the following data table, which shows the value of sales of three items by four sales girls:

	Item1	Item2	Item3
Salesgirl #1	310	275	365
Salesgirl #2	210	190	325
Salesgirl #3	405	235	240
Salesgirl #4	260	300	380

The table contains a total of 12 values, three in each line. We can think of this table as a matrix consisting of four *rows* and three *columns*. Each row represents the values of sales by a particular salesgirl and each column represents the values of sales of a particular item.

In mathematics, we represent a particular value in a matrix by using two subscripts such as v_{ij} . Here v denotes the entire matrix and v_{ij} refers to the value in the i^{th} row and j^{th} column. For example, in the above table v_{23} refers to the value 325.

C allows us to define such tables of items by using two-dimensional arrays. The table discussed above can be defined in C as

`v[4][3]`

Two-dimensional arrays are declared as follows:

```
type array_name [row_size][column_size];
```

Note that unlike most other languages, which use one pair of parentheses with commas to separate array sizes, C places each size in its own set of brackets.

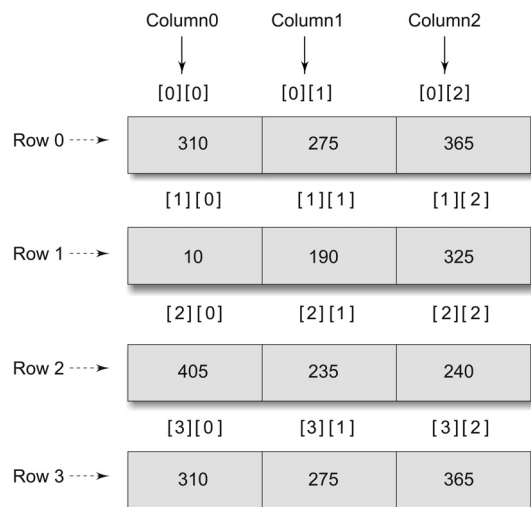


Fig. 3B.7 Representation of a *two-dimensional* array in memory

Two-dimensional arrays are stored in memory, as shown in Fig. 3B.7. As with the single-dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

Example 3B.7 Write a program using a two-dimensional array to compute and print the following information from the table of data discussed above:

- (a) Total value of sales by each girl.
- (b) Total value of each item sold.
- (c) Grand total of sales of all items by all girls.

The program and its output are shown in Fig. 3B.8. The program uses the variable **value** in two-dimensions with the index *i* representing girls and *j* representing items. The following equations are used in computing the results:

$$(a) \text{ Total sales by } m^{\text{th}} \text{ girl} = \sum_{j=0}^2 \text{value}[m][j] \quad (\text{girl_total}[m])$$

$$(b) \text{ Total value of } n^{\text{th}} \text{ item} = \sum_{i=0}^3 \text{value}[i][n] \quad (\text{item_total}[n])$$

$$\begin{aligned} (c) \text{ Grand total} &= \sum_{i=0}^3 \sum_{j=0}^2 \text{value}[i][j] \\ &= \sum_{i=0}^3 \text{girl_total}[i] \\ &= \sum_{j=0}^2 \text{item_total}[j] \end{aligned}$$

Program

```
#define  MAXGIRLS  4
#define  MAXITEMS  3

main()
{
    int  value[MAXGIRLS][MAXITEMS];
    int  girl_total[MAXGIRLS] , item_total[MAXITEMS];
    int  i, j, grand_total;
    /*.....READING OF VALUES AND COMPUTING girl_total ...*/

    printf("Input data\n");
    printf("Enter values, one at a time, row-wise\n\n");

    for( i = 0 ; i < MAXGIRLS ; i++ )
    {
        girl_total[i] = 0;
        for( j = 0 ; j < MAXITEMS ; j++ )
        {
            scanf("%d", &value[i][j]);
            girl_total[i] = girl_total[i] + value[i][j];
        }
    }
}
```

```

/*.....COMPUTING item_total.....*/

for( j = 0 ; j < MAXITEMS ; j++ )
{
    item_total[j] = 0;
    for( i =0 ; i < MAXGIRLS ; i++ )
        item_total[j] = item_total[j] + value[i][j];
}

/*.....COMPUTING grand_total.....*/

grand_total = 0;
for( i =0 ; i < MAXGIRLS ; i++ )
    grand_total = grand_total + girl_total[i];

/* .....PRINTING OF RESULTS.....*/

printf("\n GIRLS TOTALS\n\n");
for( i = 0 ; i < MAXGIRLS ; i++ )
    printf("Salesgirl[%d] = %d\n", i+1, girl_total[i] );
printf("\n ITEM TOTALS\n\n");
for( j = 0 ; j < MAXITEMS ; j++ )
    printf("Item[%d] = %d\n", j+1 , item_total[j] );
printf("\nGrand Total = %d\n", grand_total);
}

```

Output

Input data
Enter values, one at a time, row_wise

310 257 365

210 190 325

405 235 240

260 300 380

GIRLS TOTALS

Salesgirl[1] = 950

Salesgirl[2] = 725

Salesgirl[3] = 880

Salesgirl[4] = 940

ITEM TOTALS

Item[1] = 1185

Item[2] = 1000

Item[3] = 1310

Grand Total = 3495

Fig. 3B.8 Illustration of two-dimensional arrays

Example 3B.8 Write a program to compute and print a multiplication table for numbers 1 to 5 as shown below:

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	.	.	.
4	4	8	.	.	.
5	5	10	.	.	25

The program shown in Fig. 3B.9 uses a two-dimensional array to store the table values. Each value is calculated using the control variables of the nested for loops as follows:

product[i] [j] = row * column

where *i* denotes rows and *j* denotes columns of the product table. Since the indices *i* and *j* range from 0 to 4, we have introduced the following transformation:

row = i+1
column = j+1

Program

```
#define ROWS 5
#define COLUMNS 5
main()
{
    int row, column, product[ROWS][COLUMNS] ;
    int i, j ;
    printf(" MULTIPLICATION TABLE\n\n") ;
    printf(" ") ;
    for( j = 1 ; j <= COLUMNS ; j++ )
        printf("%4d" , j ) ;
    printf("\n") ;
    printf("_____\n");
    for( i = 0 ; i < ROWS ; i++ )
    {
        row = i + 1 ;
        printf("%2d |", row) ;
        for( j = 1 ; j <= COLUMNS ; j++ )
        {
            column = j ;
            product[i][j] = row * column ;
            printf("%4d", product[i][j] ) ;
        }
        printf("\n") ;
    }
}
```

Output

MULTIPLICATION TABLE					
	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

Fig. 3B.9 Program to print multiplication table using two-dimensional array

3B.6 INITIALIZING TWO-DIMENSIONAL ARRAYS

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,

```
int table[2][3] = { 0,0,0,1,1,1};
```

initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can be equivalently written as

```
int table[2][3] = {{0,0,0}, {1,1,1}};
```

by surrounding the elements of the each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below:

```
int table[2][3] = {
                    {0,0,0},
                    {1,1,1}
};
```

Note the syntax of the above statements. Commas are required after each brace that closes off a row, except in the case of the last row.

When the array is completely initialized with all values, explicitly, we need not specify the size of the first dimension. That is, the statement

```
int table [ ] [3] = {
                    { 0, 0, 0},
                    { 1, 1, 1}
};
```

is permitted.

If the values are missing in an initializer, they are automatically set to zero. For instance, the statement

```
int table[2][3] = {
                    {1,1},
                    {2}
};
```

will initialize the first two elements of the first row to one, the first element of the second row to two, and all other elements to zero.

When all the elements are to be initialized to zero, the following short-cut method may be used.

```
int m[3][5] = { {0}, {0}, {0}};
```

The first element of each row is explicitly initialized to zero while other elements are automatically initialized to zero. The following statement will also achieve the same result:

```
int m [3] [5] = { 0, 0};
```

Example 3B.9 A survey to know the popularity of four cars (Ambassador, Fiat, Dolphin and Maruti) was conducted in four cities (Bombay, Calcutta, Delhi and Madras). Each person surveyed was asked to give his city and the type of car he was using. The results, in coded form, are tabulated as follows:

M	1	C	2	B	1	D	3	M	2	B	4
C	1	D	3	M	4	B	2	D	1	C	3
D	4	D	4	M	1	M	1	B	3	B	3
C	1	C	1	C	2	M	4	M	4	C	2
D	1	C	2	B	3	M	1	B	1	C	2
D	3	M	4	C	1	D	2	M	3	B	4

Codes represent the following information:

M – Madras	1 – Ambassador
D – Delhi	2 – Fiat
C – Calcutta	3 – Dolphin
B – Bombay	4 – Maruti

Write a program to produce a table showing popularity of various cars in four cities.

A two-dimensional array **frequency** is used as an accumulator to store the number of cars used, under various categories in each city. For example, the element **frequency** [i][j] denotes the number of cars of type j used in city i. The **frequency** is declared as an array of size 5×5 and all the elements are initialized to zero.

The program shown in Fig. 3B.10 reads the city code and the car code, one set after another, from the terminal. Tabulation ends when the letter X is read in place of a city code.

```

Program
main()
{
    int i, j, car;
    int frequency[5][5] = { {0},{0},{0},{0},{0} };
    char city;
    printf("For each person, enter the city code \n");
    printf("followed by the car code.\n");
    printf("Enter the letter X to indicate end.\n");
    /*. . . . . TABULATION BEGINS . . . . . */
    for( i = 1 ; i < 100 ; i++ )
    {
        scanf("%c", &city );
        if( city == 'X' )
            break;
        scanf("%d", &car );
        switch(city)
        {
            case 'B' : frequency[1][car]++;
                        break;
            case 'C' : frequency[2][car]++;
                        break;
            case 'D' : frequency[3][car]++;
                        break;
        }
    }
}

```

(Contd.)

```

        case 'M' :   frequency[4][car]++;

                                break;

    }
}

/* . . . . TABULATION COMPLETED AND PRINTING BEGINS. . . . */
printf("\n\n");
printf(" POPULARITY TABLE\n\n");
printf("-----\n");
printf("City Ambassador Fiat Dolphin Maruti \n");

printf("-----\n");
for( i = 1 ; i <= 4 ; i++ )
{
    switch(i)
    {
        case 1 : printf("Bombay ") ;
                break ;
        case 2 : printf("Calcutta ") ;
                break ;
        case 3 : printf("Delhi ") ;
                break ;
        case 4 : printf("Madras ") ;
                break ;
    }
    for( j = 1 ; j <= 4 ; j++ )
        printf("%7d", frequency[i][j] ) ;
    printf("\n") ;
}
printf("-----\n");
/* . . . . . PRINTING ENDS. . . . . */
}

```

Output

For each person, enter the city code
followed by the car code.

Enter the letter X to indicate end.

```

M 1 C 2 B 1 D 3 M 2 B 4
C 1 D 3 M 4 B 2 D 1 C 3
D 4 D 4 M 1 M 1 B 3 B 3
C 1 C 1 C 2 M 4 M 4 C 2
D 1 C 2 B 3 M 1 B 1 C 2
D 3 M 4 C 1 D 2 M 3 B 4      X

```

POPULARITY TABLE

City	Ambassador	Fiat	Dolphin	Maruti
Bombay	2	1	3	2
Calcutta	4	5	1	0
Delhi	2	1	3	2
Madras	4	1	1	4

Fig. 3B.10 Program to tabulate a survey data

Memory Layout

The subscripts in the definition of a two-dimensional array represent rows and columns. This format maps the way that data elements are laid out in the memory. The elements of all arrays are stored contiguously in increasing memory locations, essentially in a single list. If we consider the memory as a row of bytes, with the lowest address on the left and the highest address on the right, a simple array will be stored in memory with the first element at the left end and the last element at the right end. Similarly, a two-dimensional array is stored "row-wise", starting from the first row and ending with the last row, treating each row like a simple array. This is illustrated below.

		Column			3 × 3 array
		0	1	2	
row	0	10	20	30	
	1	40	50	60	
	2	70	80	90	

row 0			row 1			row 2		
10	20	30	40	50	60	70	80	90
[0][0]	[0][1]	[0][2]	[1][0]	[1][1]	[1][2]	[2][0]	[2][1]	[2][2]
1	2	3	4	5	6	7	8	9

Memory Layout

For a multi-dimensional array, the order of storage is that the first element stored has 0 in all its subscripts, the second has all of its subscripts 0 except the far right which has a value of 1 and so on.

The elements of a 2 x 3 x 3 array will be stored as under

1	2	3	4	5	6	7	8	9	
000	001	002	010	011	012	020	021	022	..

10	11	12	13	14	15	16	17	18	
100	101	102	110	111	112	120	121	122	..

The far right subscript increments first and the other subscripts increment in order from right to left. The sequence numbers 1, 2,, 18 represents the location of that element in the list

3B.7 MULTI-DIMENSIONAL ARRAYS

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multi-dimensional array is

```
type array_name[s1][s2][s3]...[sm];
```

where s_i is the size of the i th dimension. Some examples are:

```
int survey[3][5][12];
```

```
float table[5][4][5][3];
```

survey is a three-dimensional array declared to contain 180 integer type elements. Similarly **table** is a four-dimensional array containing 300 elements of floating-point type.

The array **survey** may represent a survey data of rainfall during the last three years from January to December in five cities.

If the first index denotes year, the second city and the third month, then the element **survey[2][3][10]** denotes the rainfall in the month of October during the second year in city-3.

3B.22 Computer Programming

Remember that a three-dimensional array can be represented as a series of two-dimensional arrays as shown below:

	<i>month</i>	<i>1</i>	<i>2</i>	<i>.....</i>	<i>12</i>
	<i>city</i>				
Year 1	1				
	.				
	.				
	.				
	5				
	<i>month</i>	<i>1</i>	<i>2</i>	<i>.....</i>	<i>12</i>
	<i>city</i>				
Year 2	1				
	.				
	.				
	.				
	5				

ANSI C does not specify any limit for array dimension. However, most compilers permit seven to ten dimensions. Some allow even more.

3B.8 DYNAMIC ARRAYS

So far, we created arrays at compile time. An array created at compile time by specifying size in the source code has a fixed size and cannot be modified at run time. The process of allocating memory at compile time is known as *static memory allocation* and the arrays that receive static memory allocation are called *static arrays*. This approach works fine as long as we know exactly what our data requirements are.

Consider a situation where we want to use an array that can vary greatly in size. We must guess what will be the largest size ever needed and create the array accordingly. A difficult task in fact! Modern languages like C do not have this limitation. In C it is possible to allocate memory to arrays at run time. This feature is known as *dynamic memory allocation* and the arrays created at run time are called *dynamic arrays*. This effectively postpones the array definition to run time.

Dynamic arrays are created using what are known as *pointer variables* and *memory management functions* **malloc**, **calloc** and **realloc**. These functions are included in the header file **<stdlib.h>**. The concept of dynamic arrays is used in creating and manipulating data structures such as linked lists, stacks and queues.

3B.9 PASSING ARRAYS TO FUNCTIONS

3B.9.1 One-Dimensional Arrays

Like the values of simple variables, it is also possible to pass the values of an array to a function. To pass a one-dimensional array to a called function, it is sufficient to list the name of the array, *without any subscripts*, and the size of the array as arguments. For example, the call

largest(a,n)

will pass the whole array **a** to the called function. The called function expecting this call must be appropriately defined. The **largest** function header might look like:

float largest(float array[], int size)

The function **largest** is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows:

float array[];

The pair of brackets informs the compiler that the argument **array** is an array of numbers. It is not necessary to specify the size of the **array** here.

Let us consider a problem of finding the largest value in an array of elements. The program is as follows:

```
main( )
{
    float largest(float a[ ], int n);
    float value[4] = {2.5,-4.75,1.2,3.67};
    printf("%f\n", largest(value,4));
}

float largest(float a[], int n)
{
    int i;
    float max;
    max = a[0];
    for(i = 1; i < n; i++)
        if(max < a[i])
            max = a[i];
    return(max);
}
```

When the function call **largest(value,4)** is made, the values of all elements of array **value** become the corresponding elements of array **a** in the called function. The **largest** function finds the largest value in the array and returns the result to the **main**.

In C, the name of the array represents the address of its first element. By passing the array name, we are, in fact, passing the address of the array to the called function. The array in the called function now refers to the same array stored in the memory. Therefore, any changes in the array in the called function will be reflected in the original array.

Passing addresses of parameters to the functions is referred to as *pass by address* (or pass by pointers). Note that we cannot pass a whole array by value as we did in the case of ordinary variables.

Example 3B.10 Write a program to calculate the standard deviation of an array of values. The array elements are read from the terminal. Use functions to calculate standard deviation and mean.

Standard deviation of a set of n values is given by

$$S.D = \sqrt{\frac{1}{n} \sum_{i=1}^n (\bar{x} - x_i)^2}$$

where \bar{x} is the mean of the values.

3B.24 Computer Programming

Program

```
#include <math.h>
#define SIZE 5
float std_dev(float a[], int n);
float mean (float a[], int n);

main( )
{
    float value[SIZE];
    int i;

    printf("Enter %d float values\n", SIZE);
    for (i=0 ; i < SIZE ; i++)
        scanf("%f", &value[i]);
    printf("Std.deviation is %f\n", std_dev(value,SIZE));
}

float std_dev(float a[], int n)
{
    int i;
    float x, sum = 0.0;
    x = mean (a,n);
    for(i=0; i < n; i++)
        sum += (x-a[i])*(x-a[i]);
    return(sqrt(sum/(float)n));
}

float mean(float a[],int n)
{
    int i ;
    float sum = 0.0;
    for(i=0 ; i < n ; i++)
        sum = sum + a[i];
    return(sum/(float)n);
}
```

Output

```
Enter 5 float values
35.0 67.0 79.5 14.20 55.75

Std.deviation is 23.231582
```

Fig. 3B.11 *Passing of arrays to a function*

A multifunction program consisting of **main**, **std_dev**, and **mean** functions is shown in Fig. 3B.11. **main** reads the elements of the array **value** from the terminal and calls the function **std_dev** to print the standard deviation of the array elements. **Std_dev**, in turn, calls another function **mean** to supply the average value of the array elements.

Both **std_dev** and **mean** are defined as **floats** and therefore they are declared as **floats** in the global section of the program.

Three Rules to Pass an Array to a Function

1. The function must be called by passing only the name of the array.
2. In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
3. The function prototype must show that the argument is an array.

When dealing with array arguments, we should remember one major distinction. If a function changes the values of the elements of an array, then these changes will be made to the original array that passed to the function. When an entire array is passed as an argument, the contents of the array are not copied into the formal parameter array; instead, information about the addresses of array elements are passed on to the function. Therefore, any changes introduced to the array elements are truly reflected in the original array in the calling function. However, this does not apply when an individual element is passed on as argument. Example 3B.11 highlights these concepts.

Example 3B.11 Write a program that uses a function to sort an array of integers.

A program to sort an array of integers using the function **sort()** is given in Fig. 3A.4. Its output clearly shows that a function can change the values in an array passed as an argument.

Program

```
void sort(int m, int x[ ]);
main()
{
    int i;
    int marks[5] = {40, 90, 73, 81, 35};

    printf("Marks before sorting\n");
    for(i = 0; i < 5; i++)
        printf("%d ", marks[i]);
    printf("\n\n");

    sort (5, marks);

    printf("Marks after sorting\n");
    for(i = 0; i < 5; i++)
        printf("%4d", marks[i]);
    printf("\n");
}

void sort(int m, int x[ ])
{
    int i, j, t;

    for(i = 1; i <= m-1; i++)
        for(j = 1; j <= m-i; j++)
            if(x[j-1] >= x[j])
            {
                t = x[j-1];
                x[j-1] = x[j];
                x[j] = t;
            }
}
```

Output

```

Marks before sorting
40 90 73 81 35

Marks after sorting
35 40 73 81 90

```

Fig. 3B.12 *Sorting of array elements using a function*

3B.9.2 Two-Dimensional Arrays

Like simple arrays, we can also pass multi-dimensional arrays to functions. The approach is similar to the one we did with one-dimensional arrays. The rules are simple.

1. The function must be called by passing only the array name.
2. In the function definition, we must indicate that the array has two-dimensions by including two sets of brackets.
3. The size of the second dimension must be specified.
- 3A. The prototype declaration should be similar to the function header.

The function given below calculates the average of the values in a two-dimensional matrix.

```

double average(int x[][N], int M, int N)
{
    int i, j;
    double sum = 0.0;
    for (i=0; i<M; i++)
        for(j=1; j<N; j++)
            sum += x[i][j];
    return(sum/(M*N));
}

```

This function can be used in a main function as illustrated below:

```

main( )
{
    int M=3, N=2;
    double average(int [ ] [N], int, int);
    double mean;
    int matrix [M][N]=
        {
            {1,2},
            {3,4},
            {5,6}
        };

    mean = average(matrix, M, N);
    . . . . .
    . . . . .
}

```

3B.10 PASSING STRINGS TO FUNCTIONS

The strings are treated as character arrays in C and therefore the rules for passing strings to functions are very similar to those for passing arrays to functions.

Basic rules are:

1. The string to be passed must be declared as a formal argument of the function when it is defined.

Example:

```
void display(char item_name[ ])
{
    .....
    .....
}
```

2. The function prototype must show that the argument is a string. For the above function definition, the prototype can be written as

```
void display(char str[ ]);
```

3. A call to the function must have a string array name without subscripts as its actual argument.

Example:

```
display (names);
```

where **names** is a properly declared string array in the calling function.

We must note here that, like arrays, strings in C cannot be passed by value to functions.

Pass by Value versus Pass by Pointers

The technique used to pass data from one function to another is known as *parameter passing*. Parameter passing can be done in two ways:

- Pass by value (also known as call by value).
- Pass by pointers (also known as call by pointers).

In *pass by value*, values of actual parameters are copied to the variables in the parameter list of the called function. The called function works on the copy and not on the original values of the actual parameters. This ensures that the original data in the calling function cannot be changed accidentally.

In *pass by pointers* (also known as pass by address), the memory addresses of the variables rather than the copies of values are sent to the called function. In this case, the called function directly works on the data in the calling function and the changed values are available in the calling function for its use.

Pass by pointers method is often used when manipulating arrays and strings. This method is also used when we require multiple values to be returned by the called function.

3B.11 MORE ABOUT ARRAYS

What we have discussed in this chapter are the basic concepts of arrays and their applications to a limited extent. There are some more important aspects of application of arrays. They include:

- using pointers for accessing arrays;
- passing arrays as function parameters;
- arrays as members of structures;

3B.28 Computer Programming

- using structure type data as array elements;
- arrays as dynamic data structures; and
- manipulating character arrays and strings.

These aspects of arrays are covered later in the following chapters:

Chapter 11	:	Strings
Chapter 12	:	Functions
Chapter 13	:	Structures
Chapter 14	:	Pointers

Example 3B.12 Write a simple program to implement a one-dimensional array and print its index and corresponding values.

Program

```
/*Program for demonstrating implementation of a simple array*/
#include <stdio.h>
#include <conio.h>

void main( )
{
    int i;
    int arr[5];/*Declaring an array of five elements*/
    clrscr();

    printf("Enter the array values:\n");
    for(i=0;i<5;i++)
        scanf("%d", &arr[i]);/*Reading values to be stored in the array*/

    printf("\nArray arr[5] contains the following values:\n");
    for(i=0;i<5;i++)
        printf("arr[%d] = %d\n", i,arr[i]);/*Printing array values*/

    getch();
}
```

Output

```
Enter the array values:
10
20
30
40
50
```

```

Array arr[5] contains the following values:
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50

```

Example 3B.13 Write a program to generate Fibonacci series using arrays.

Program

```

/*Program for demonstrating the use arrays for generating Fibonacci series*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int arr[50],len,i;
    clrscr();

    printf("\n\nEnter length of the Fibonacci series: ");
    scanf("%d",&len);/*Reading series length*/

    arr[0]=0;/*Initializing 1st element of Fibonacci series*/
    arr[1]=1;/*Initializing 2nd element of Fibonacci series*/

    /*Storing the Fibonacci series values in array arr[]*/
    for(i=2;i<len;i++)
        arr[i]=arr[i-1]+arr[i-2];

    /*Printing the Fibonacci series*/
    printf("\n<---FIBONACCI SERIES--->\n");
    for(i=0;i<len;i++)
        printf("%d ",arr[i]);

    getch();
}

```

Output

```

Enter length of the Fibonacci series: 8

<---FIBONACCI SERIES--->
0 1 1 2 3 5 8 13

```

Example 3B.14 Write a program to perform linear search on a one-dimensional array.

Program

```
/*Program for demonstrating linear search using a simple array*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int arr[10],i,j,element;
    int flag=0;
    clrscr();

    printf("Enter the 10 elements of the array:\n");
    for(i=0;i<10;i++)
        scanf("%d",&arr[i]);/*Reading array values*/

    printf("\n\nEnter the element that you want to search: ");
    scanf("%d",&element);/*Reading the element to be searched*/

    for(j=0;j<10;j++)
        if( arr[j] == element) /*Checking whether the search was successful*/
        {
            printf("\nThe element %d is present at %d position in the list\n",element,j+1);
            flag=1;
            break;
        }
    if(flag==0)/*Checking whether the search was unsuccessful*/
        printf("\nThe element is %d is not present in the list\n",element);

    getch();
}
```

Output

```
Enter the 10 elements of the array:
22
55
1
2
66
69
```

54
78
8
36

Enter the element that you want to search: 54

The element 54 is present at 7 position in the list

Example 3B.15 Write a simple program to read integer elements into an array and compute their sum.

Program

```
/*Program for demonstrating implementation of a simple array*/
#include <stdio.h>
#include <conio.h>

void main( )
{
    int i,sum;
    int arr[5];/*Declaring an array of five elements*/
    clrscr();

    printf("Enter the array values:\n");
    for(i=0;i<5;i++)
        scanf("%d", &arr[i]);/*Reading values to be stored in the array*/

    printf("\nArray arr[5] contains the following elements:\n");
    for(i=0;i<5;i++)
        printf("arr[%d] = %d\n", i,arr[i]);/*Printing array values*/

    sum=0;
    for(i=0;i<5;i++)
        sum=sum+arr[i];/*Computing sum of elements stored in the array*/

    printf("\nSum of array elements = %d",sum); /*Displaying the result*/

    getch();
}
```

3B.32 Computer Programming

Output

Enter the array values:

1
2
3
4
5

Array arr[5] contains the following elements:

arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5

Sum of array elements = 15

Example 3B.16 Write a C program to read in 10 integer numbers and print their average, minimum and maximum numbers.

Program

```
/*Program for demonstrating the use of a simple one-dimensional array*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int a[10], min, max, sum, i;
    float ave;
    clrscr();

    printf("Enter 10 integer elements: ");
    for(i=0;i<=9;i++)
        scanf("%d",&a[i]); /*Reading the input integer elements in an array*/

    min=a[0];
    max=a[0];
    sum=0;

    for(i=0;i<=9;i++)
```



```
sum=sum+a[i];

ave=1.0*sum/10; /*Calculating the average*/

/*Finding minimum and maximum numbers*/
for(i=0;i<=8;i++)
{
    if(a[i+1]>max)
        max=a[i+1];
    if(a[i+1]<min)
        min=a[i+1];
}
/*Displaying the results*/
printf("\nMinimum = %d, Maximum = %d, Average = %.2f",min,max,ave);
getch();
}
```

Output

```
Enter 10 integer elements: 1
2
3
4
5
6
7
8
9
10

Minimum = 1, Maximum = 10, Average = 5.50
```

Example 3B.17 Write a program to read and display a simple 3 X 3 matrix.

Program

```
/*Program for realizing a 3 X 3 matrix using 2-D arrays*/
#include <stdio.h>
#include <conio.h>

void main()
```

3B.34 Computer Programming

```
{
    int i,j,a[3][3];
    clrscr();

    /*Reading matrix elements*/
    printf("Enter the elements of the 3 X 3 matrix:\n");
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
        {
            printf("a[%d][%d] = ",i,j);
            scanf("%d",&a[i][j]);
        }

    /*Printing matrix elements*/
    printf("The various elements contained in the 3 X 3 matrix are:\n");
    for(i=0;i<3;i++)
    {
        printf("\n\t\t");
        for(j=0;j<3;j++)
            printf("%d\t",a[i][j]);
    }

    getch();
}
```

Output

```
Enter the elements of the 3 X 3 matrix:
a[0][0] = 1
a[0][1] = 2
a[0][2] = 3
a[1][0] = 4
a[1][1] = 5
a[1][2] = 6
a[2][0] = 7
a[2][1] = 8
a[2][2] = 9
The various elements contained in the 3 X 3 matrix are:

        1    2    3
        4    5    6
        7    8    9
```

Example 3B.18 Write a program to add two 3 X 3 matrices.

Program

```
/*Program for adding two 3 X 3 matrices using 2-D arrays*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int i,j,a[3][3],b[3][3],c[3][3];
    clrscr();

    printf("Enter the first 3 X 3 matrix:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("a[%d][%d] = ",i,j);
            scanf("%d",&a[i][j]);/*Reading the elements of 1st matrix*/
        }
    }

    printf("Enter the second 3 X 3 matrix:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("b[%d][%d] = ",i,j);
            scanf("%d",&b[i][j]);/*Reading the elements of 2nd matrix*/
        }
    }

    printf("\nThe entered matrices are: \n");
    for(i=0;i<3;i++)
    {
        printf("\n");
        for(j=0;j<3;j++)
            printf("%d\t",a[i][j]);/*Displaying the elements of 1st matrix*/
        printf("\t\t");
        for(j=0;j<3;j++)
            printf("%d\t",b[i][j]);/*Displaying the elements of 2nd matrix*/
    }
}
```

3B.36 Computer Programming

```
for(i=0;i<3;i++)
    for(j=0;j<3;j++)
        c[i][j] =a[i][j]+b[i][j];/*Computing the sum of two matrices*/

printf("\n\nThe sum of the two matrices is shown below: \n");
for(i=0;i<3;i++)
{
    printf("\n\t\t");
    for(j=0;j<3;j++)
        printf("%d\t",c[i][j]);/*Displaying the result*/
}

getch();
}
```

Output

Enter the first 3 X 3 matrix:

a[0][0] = 1
a[0][1] = 2
a[0][2] = 3
a[1][0] = 4
a[1][1] = 5
a[1][2] = 6
a[2][0] = 7
a[2][1] = 8
a[2][2] = 9

Enter the second 3 X 3 matrix:

b[0][0] = 9
b[0][1] = 8
b[0][2] = 7
b[1][0] = 6
b[1][1] = 5
b[1][2] = 4
b[2][0] = 3
b[2][1] = 2
b[2][2] = 1

The entered matrices are:

1	2	3	9	8	7
4	5	6	6	5	4
7	8	9	3	2	1

The sum of the two matrices is shown below:

10	10	10
10	10	10
10	10	10

Example 3B.19 Write a program to subtract two 3 X 3 matrices.

Program

The code for subtracting two 3 X 3 matrices is same as Example 7 with only the following difference:

```
.
.
for(i=0;i<3;i++)
    for(j=0;j<3;j++)
        c[i][j]=a[i][j]-b[i][j];/*Subtracting the two matrices*/
.
.
```

Here, we are performing the subtraction operation instead of addition.

Output

```
Enter the first 3 X 3 matrix:
a[0][0] = 10
a[0][1] = 20
a[0][2] = 30
a[1][0] = 40
a[1][1] = 50
a[1][2] = 60
a[2][0] = 70
a[2][1] = 80
a[2][2] = 90
Enter the second 3 X 3 matrix:
b[0][0] = 5
b[0][1] = 10
b[0][2] = 15
b[1][0] = 20
b[1][1] = 25
b[1][2] = 30
b[2][0] = 35
b[2][1] = 40
b[2][2] = 45
```

3B.38 Computer Programming

The entered matrices are:

10	20	30	5	10	15
40	50	60	20	25	30
70	80	90	35	40	45

The result of subtraction is shown below:

5	10	15
20	25	30
35	40	45

Example 3B.20 Write a program to multiply two 3 X 3 matrices.

Program

```
/*Program for multiplying two 3 X 3 matrices using 2-D arrays*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int i,j,k,a[3][3],b[3][3],c[3][3];
    clrscr();
    printf("Enter the first 3 X 3 matrix:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("a[%d][%d] = ",i,j);
            scanf("%d",&a[i][j]);/*Reading the elements of the 1st matrix*/
        }
    }

    printf("Enter the second 3 X 3 matrix:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("b[%d][%d] = ",i,j);
            scanf("%d",&b[i][j]);/*Reading the elements of the 2nd matrix*/
        }
    }
}
```

```

printf("\nThe entered matrices are: \n");

for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
    {
        printf("%d\t",a[i][j]);/*Displaying the elements of the 1st matrix*/
    }
    printf("\t\t");
    for(j=0;j<3;j++)
    {
        printf("%d\t",b[i][j]);/*Displaying the elements of the 2nd matrix*/
    }
}
/*Multiplying the two matrices*/
for(i=0;i<3;i++)
    for(j=0;j<3;j++)
    {
        c[i][j]=0;
        for(k=0;k<3;k++)
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
    }
printf("\n\nThe product of the two matrices is shown below: \n");

for(i=0;i<3;i++)
{
    printf("\n\t\t");
    for(j=0;j<3;j++)
    {
        printf("%d\t",c[i][j]); /*Displaying the result*/
    }
}

getch();
}

```

Output

```

Enter the first 3 X 3 matrix:
a[0][0] = 1
a[0][1] = 2

```

3B.40 Computer Programming

```
a[0][2] = 3
a[1][0] = 4
a[1][1] = 5
a[1][2] = 6
a[2][0] = 7
a[2][1] = 8
a[2][2] = 9
Enter the second 3 X 3 matrix:
b[0][0] = 9
b[0][1] = 8
b[0][2] = 7
b[1][0] = 6
b[1][1] = 5
b[1][2] = 4
b[2][0] = 3
b[2][1] = 2
b[2][2] = 1
```

The entered matrices are:

1	2	3	9	8	7
4	5	6	6	5	4
7	8	9	3	2	1

The product of the two matrices is shown below:

30	24	18
84	69	54
138	114	90

Example 3B.21 Write a program to generate the transpose of a 3X3 matrix.

Program

```
/*Program for generating the transpose of a 3X3 matrix using 2-D arrays*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int i,j,a[3][3],b[3][3];
```



```
clrscr();
printf("Enter a 3 X 3 matrix:\n");
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        printf("a[%d][%d] = ",i,j);
        scanf("%d",&a[i][j]); /*Reading the elements of the 3X3 matrix*/
    }
}

printf("\nThe entered matrix is: \n");
for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
    {
        printf("%d\t",a[i][j]); /*Displaying the matrix*/
    }
}

for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
        b[i][j]=a[j][i]; /*Computing matrix transpose*/
}
printf("\n\nThe transpose of the matrix is: \n");

for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
    {
        printf("%d\t",b[i][j]); /*Displaying the resultant transposed matrix*/
    }
}

getch();
}
```

Output

Enter a 3 X 3 matrix:

a[0][0] = 1

a[0][1] = 2

a[0][2] = 3

a[1][0] = 4

a[1][1] = 5

a[1][2] = 6

a[2][0] = 7

a[2][1] = 8

a[2][2] = 9

The entered matrix is:

1	2	3
---	---	---

4	5	6
---	---	---

7	8	9
---	---	---

The transpose of the matrix is:

1	4	7
---	---	---

2	5	8
---	---	---

3	6	9
---	---	---

Example 3B.22 Write a simple program to demonstrate the use of functions in C.

Program

```
/*Program for demonstrating use of functions*/
#include <stdio.h>
#include <conio.h>

/*Declaring function prototypes*/
void Func_A(void);
void Func_B(void);
void Func_C(void);

void main( )
{
    clrscr();
    printf("In main function; Calling Func_A()...\n");
```

```
Func_A();/*Calling Function A*/
printf("Back in main()..\n");

getch();
}

void Func_A(void)
{
    printf("Inside Func_A(); Calling Func_B()..\n");
    Func_B();/*Function A calls Function B*/
    printf("Back in Func_A()..\n");
}

void Func_B(void)
{
    printf("Inside Func_B(); Calling Func_C()..\n");
    Func_C(); /*Function B in turn calls Function C*/
    printf("Back in Func_B()..\n");
}

void Func_C(void)
{
    printf("Inside Func_C()..\n"); /*print statement inside Function C*/
}
```

Output

```
In main function; Calling Func_A()..
Inside Func_A(); Calling Func_B()..
Inside Func_B(); Calling Func_C()..
Inside Func_C()..
Back in Func_B()..
Back in Func_A()..
Back in main()..
```

Example 3B.23 Write a program to compute the area of a rectangle.

Program

```
/*No arguments and no return values*/
#include <stdio.h>
#include <conio.h>
```

3B.44 Computer Programming

```
void main()
{
    void area(void); /*Declaring the function prototype for computing area of a rectangle*/
    clrscr();

    area();/*Calling the area function*/
    getch();
}

void area()
{
    float length, breadth;

    printf("Enter the length and breadth of the rectangle:\n");
    scanf("%f %f",&length,&breadth) ;/*Reading the length and breadth values of the rectangle*/

    /*Computing and displaying the area of the rectangle*/
    printf("The area of the rectangle with length %.2f and breadth %.2f is = %.2f", length,breadth,(length*breadth));
}
```

Output

```
Enter the length and breadth of the rectangle:
5
10
The area of the rectangle with length 5.00 and breadth 10.00 is = 50.00
```

Example 3B.24 Write a program to compute the area of a triangle.

Program

```
/*Arguments but no return values*/
#include <stdio.h>
#include <conio.h>

void main()
{
    float base,height;
```

```

void area(float,float); /*Declaring the function prototype for computing area of a
triangle*/
clrscr();

printf("Enter base and height of the triangle:\n");
scanf("%f %f",&base,&height) ;/*Reading the base and height values*/

area(base,height); /*Calling the function by passing base and height as parameters*/

getch();
}

void area(float b,float h)
{
    printf("The area of the triangle with base %.2f and height %.2f is = %.2f",
b,h,(0.5*b*h));/*Displaying the area of the triangle*/
}

```

Output

```

Enter base and height of the triangle:
5
8
The area of the triangle with base 5.00 and height 8.00 is = 20.00

```

Example 3B.25 Write a program that uses functions to determine whether a given number is Armstrong or not.

Program

```

/*Arguments but no return values*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

void main()
{
    int num;
    void arm(int);/*Declaring prototype for the function that computes Armstrong value*/
    clrscr();

    printf("\nEnter a number: ");

```

3B.46 Computer Programming

```
scanf("%d", &num); /*Reading input integer value*/

arm(num); /*Calling the function that assess whether num is Armstrong or not*/
getch();
}

void arm(int n)
{
    int temp, sum=0, i;

    temp=n;

    while(temp>0) /*Computing Armstrong value*/
    {
        i=temp%10;
        sum=sum+i*i*i;
        temp=temp/10;
    }

    if(sum==n) /*Checking whether num is Armstrong or not*/
        printf("\n%d is an Armstrong number",n);
    else
        printf("\n%d is not an Armstrong number",n);
}

Output
Enter a number: 371

371 is an Armstrong number
```

Example 3B.26 Write a program that prints the ASCII value corresponding to a given character.

Program

```
/*Arguments but no return values*/
#include <stdio.h>
#include <conio.h>

void main()
```

```

{
    char ch;
    void ASCII(char); /*Declaring the function prototype for determining ASCII value*/
    clrscr();

    printf("Enter any character:\n");
    scanf("%c",&ch) ;/*Reading input character value*/

    ASCII(ch); /*Calling the ASCII function by passing ch as the parameter*/

    getch();
}

void ASCII(char c)
{
    printf("The ASCII value of %c is %d",c,c);/*Displaying the ASCII value*/
}

```

Output

```

Enter any character:
V
The ASCII value of V is 86

```

Example 3B.27 Write a program to implement a simple arithmetic calculator using functions.

Program

```

/*Arguments with return values*/
#include <stdio.h>
#include <conio.h>

/*Declaring function prototypes for common arithmetic operations*/
float add(float,float);
float sub(float,float);
float mul(float,float);
float div(float,float);

void main()
{
    int choice;

```

3B.48 Computer Programming

```
float num1, num2;
clrscr();
printf("*****Simple Calc*****");/*Displaying CalC options*/
printf("\n\nChoose a type of operation from the following: ");
printf("\n\t1.   Addition");
printf("\n\t2.   Subtraction");
printf("\n\t3.   Multiplication");
printf("\n\t4.   Division\n");
scanf("%d", &choice);/*Reading user's choice*/
printf("\n\nEnter the two operands: ");
scanf("%f %f", &num1, & num2);/*Reading operands*/

/*Using the Switch statement to choose the right operation*/
switch (choice)
{
case 1:
    printf("\n%.2f + %.2f = %.2lf", num1, num2, add(num1,num2)); /*Calling the add
function*/
    break;

case 2:
    printf("\n%.2f - %.2f = %.2lf", num1, num2, sub(num1,num2)); /*Calling the sub
function*/
    break;

case 3:
    printf("\n%.2f * %.2f = %.2lf", num1, num2, mul(num1,num2)); /*Calling the mul
function*/
    break;

case 4:
    printf("\n%.2f / %.2f = %.2lf", num1, num2, div(num1,num2)); /*Calling the div
function*/
    break;

default:
    printf("\nIncorrect Choice!");
}

getch();
}
```



```
/*Writing function definitions*/
float add(float x,float y)
{
    return(x+y);
}

float sub(float x,float y)
{
    return(x-y);
}

float mul(float x,float y)
{
    return(x*y);
}

float div(float x,float y)
{
    return(x/y);
}
```

Output

```
*****Simple Calc*****

Choose a type of operation from the following:
    1.  Addition
    2.  Subtraction
    3.  Multiplication
    4.  Division
3

Enter the two operands: 5
8

5.00 * 8.00 = 40.00
```

Example 3B.28 Write a program that solves the Tower of Hanoi problem using recursion.

Program

```
/*Program demonstrating the use of recursion to solve the Tower of Hanoi problem*/
#include<stdio.h>
#include<conio.h>

void TOH( int, char, char, char);/*Declaring the function prototype*/

void main()
{
    int disk;
    clrscr();

    printf("\nEnter the number of disks: ");
    scanf("%d",&disk);/*Reading the initial number of disks*/

    if(disk<1)
        printf("\n There are no disks to shift");
    else
        printf("\nThere are %d disks in tower 1\n ",disk);

    TOH(disk,'1','2','3');/*Calling the TOH function*/
    printf("\n\nThe %d disks in tower 1 are shifted to tower 2",disk);
    getch();
}

void TOH(int d,char tower1, char tower2, char tower3)
{
    if (d==1)
    {
        printf("\nShift top disk from tower %c to tower %c.", tower1, tower2);
        return;
    }

    TOH(d-1,tower1, tower3, tower2); /*recursive function call*/
    printf("\nShift top disk from tower %c to tower %c.", tower1, tower2);
    TOH(d-1,tower3, tower2, tower1); /*recursive function call*/
}
```

Output

```
Enter the number of disks: 3

There are 3 disks in tower 1

Shift top disk from tower 1 to tower 2.
Shift top disk from tower 1 to tower 3.
Shift top disk from tower 2 to tower 3.
Shift top disk from tower 1 to tower 2.
Shift top disk from tower 3 to tower 1.
Shift top disk from tower 3 to tower 2.
Shift top disk from tower 1 to tower 2.

The 3 disks in tower 1 are shifted to tower 2
```

Example 3B.29 Write a program that uses recursion to generate the Fibonacci series.

Program

```
/*Program demonstrating the use of recursion to generate the Fibonacci series*/
#include<stdio.h>
#include<conio.h>

void main()
{
    int x=0,y=1,n;
    void fib(int,int,int);/*Function prototype*/
    clrscr();

    printf("Enter the number of terms in Fibonacci series: ");
    scanf("%d",&n); /*Reading number of terms in the series*/

    printf("\nThe Fibonacci series is:");
    printf("\n\n%d\t%d",x,y); /*Printing 1st two terms of the series*/

    fib(x,y,n-2); /*Function Call*/
    printf(" ");

    getch();
}
```

3B.52 Computer Programming

```
void fib(int a,int b,int n)
{
    int c;
    if(n==0)
        return;
    n--;
    c=a+b;
    printf("\t%d",c);
    fib(b,c,n);/*recursive function call*/
}
```

Output

Enter the number of terms in Fibonacci series: 8

The Fibonacci series is:

0 1 1 2 3 5 8 13

Example 3B.30 Write a program that uses functions to compute the product of two given matrices.

Program

```
/*Program for multiplying two 3 X 3 matrices using functions*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int i,j,k,a[3][3],b[3][3];
    void multiply (int[][3], int[][3]); /*Declaring the function prototype for matrix
multiplication*/
    clrscr();

    printf("Enter the first 3 X 3 matrix:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("a[%d][%d] = ",i,j);
            scanf("%d",&a[i][j]);/*Reading the elements of the 1st matrix*/
        }
    }
}
```

```

printf("Enter the second 3 X 3 matrix:\n");
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        printf("b[%d][%d] = ",i,j);
        scanf("%d",&b[i][j]);/*Reading the elements of the 2nd matrix*/
    }
}
printf("\nThe entered matrices are: \n");

for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
    {
        printf("%d\t",a[i][j]);/*Displaying the elements of the 1st matrix*/
    }
    printf("\t\t");
    for(j=0;j<3;j++)
    {
        printf("%d\t",b[i][j]);/*Displaying the elements of the 2nd matrix*/
    }
}

multiply(a,b); /*Calling the multiply function to compute the product of a and b*/
getch();
}

void multiply(int x[][3],int y[][3])
{
    int c[3][3],i,j,k;

/*Multiplying the two matrices*/
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
        {
            c[i][j]=0;
            for(k=0;k<3;k++)
                c[i][j]=c[i][j]+x[i][k]*y[k][j];
        }
}

```

3B.54 Computer Programming

```
printf("\n\nThe product of the two matrices is shown below: \n");

for(i=0;i<3;i++)
{
    printf("\n\t\t");
    for(j=0;j<3;j++)
    {
        printf("%d\t",c[i][j]); /*Displaying the result*/
    }
}
}
```

Output

Enter the first 3 X 3 matrix:

a[0][0] = 1
a[0][1] = 2
a[0][2] = 3
a[1][0] = 4
a[1][1] = 5
a[1][2] = 6
a[2][0] = 7
a[2][1] = 8
a[2][2] = 9

Enter the second 3 X 3 matrix:

b[0][0] = 10
b[0][1] = 10
b[0][2] = 10
b[1][0] = 20
b[1][1] = 20
b[1][2] = 20
b[2][0] = 30
b[2][1] = 30
b[2][2] = 30

The entered matrices are:

1	2	3	10	10	10
4	5	6	20	20	20
7	8	9	30	30	30

The product of the two matrices is shown below:

140	140	140
320	320	320
500	500	500

Example 3B.31 Write a program to generate the one's complement of a given binary number.

Program

```

/*Program for demonstrating the passing of a string as a parameter to a function*/
#include <stdio.h>
#include <conio.h>

char b[16]; /*array for storing the result of one's complement*/

void main()
{
    char a[16];
    int i;
    void ones(char[]); /*Declaring the function prototype for generating one's complement*/
    clrscr();

    printf("Enter a binary number: ");
    gets(a); /*Reading the input binary number*/

    ones(a); /*Calling the function by passing input binary string a*/

    printf("\nThe 1's compliment of %s is %s", a,b); /*Displaying the result*/
    getch();
}

void ones(char str[])
{
    int i;
    char C[16];
    for(i=0;str[i]!='\0'; i++)
    {
        if (str[i]!='0' && str[i]!='1')

```

```
{
    printf("\nIncorrect binary number format...the program will quit");
    getch();
    exit(0);
}

/*Changing the binary digits from 0 to 1 and vice versa*/
if (str[i]=='0')
    b[i]='1';
else
    b[i]='0';
}
b[i]='\0';
}
```

Output

Enter a binary number: 11001001

The 1's compliment of 11001001 is 00110110

Example 3B.32

Write a program to compute the two's complement.

Program

```
/*Program for demonstrating the use of global variables*/
#include <stdio.h>
#include <conio.h>
#include <string.h>

char a[16]; /*Declaring a character array as a global variable*/

void main()
{
    void twos(void); /*Declaring the function prototype for computing two's complement*/
    clrscr();
    printf("Enter a binary number: ");
    gets(a); /*Reading a binary string*/

    twos(); /*Calling the function*/
}
```



```

    printf("\n2's compliment = %s",a); /*Printing the result stored in a by the twos
function*/
    getch();
}

void twos(void)
{
    int i,j,k,len;
    len=strlen(a);
    for(k=0;a[k]!='\0'; k++)
    {
        if (a[k]!='0' && a[k]!='1')
        {
            printf("\nIncorrect binary number format...the program will quit");
            getch();
            exit(0);
        }
    }
    /*Computing the two's compliment*/
    for(i=len-1;a[i]!='1'; i--)
    ;
    for(j=i-1;j>=0;j--)
    {
        if(a[j]=='1')
            a[j]='0';
        else
            a[j]='1';
    }
}

```

Output

Enter a binary number: 10100011

2's compliment = 01011101

3B.12 C PROGRAMMING EXAMPLES — BUILT-IN FUNCTIONS**Example 3B.33** Write a program to compute xy .**Program**

```
/*Program for demonstrating the use of built-in function pow()*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

void main()
{
    int x,y;
    long result;
    clrscr();

    printf("\nEnter the values of x and y:\n");
    scanf("%d %d",&x,&y);

    result=pow(x,y); /*Calling the built-in function pow()*/
    printf("\n%d raised to the power of %d = %ld",x, y, result);/*Displaying the re-
    sult*/

    getch();
}
```

Output

```
Enter the values of x and y:
2
6

2 raised to the power of 6 = 64
```

Example 3B.34 Write a program to compute the square root of an integer.**Program**

```
/*Program for demonstrating the use of built-in function pow()*/
#include <stdio.h>
#include <conio.h>
```

```

#include <math.h>

void main()
{
    float x;
    double result;
    clrscr();

    printf("\nEnter the value of x:\n");
    scanf("%f",&x); /*Reading the input value x*/

    result=sqrt(x); /*Calling the built-in function sqrt()*/
    printf("\nThe square root of %.2f is = %.2lf",x, result);/*Displaying the result*/

    getch();
}

```

Output

```

Enter the value of x:
50

The square root of 50.00 is = 7.07

```

Example 3B.35 Write a program that generates random numbers.

Program

```

/*Program for demonstrating the use of built-in function rand()*/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

void main()
{
    clrscr();

    srand(time(NULL)); /*Specifying time value as an input to random number generator*/
    printf("The system generated random number is: %d", rand()); /*Calling the rand()
function to generate the random number*/
}

```

3B.60 Computer Programming

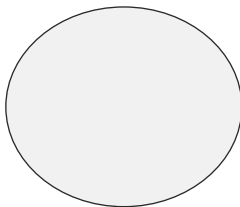
```
    getch();  
}  
  
Output  
The system generated random number is: 18081
```

Example 3B.36 Write a program to draw a circle.

Program

```
/*Program for demonstrating the use of built-in Graphics functions of C*/  
#include<stdio.h>  
#include<conio.h>  
#include<graphics.h>  
  
void main()  
{  
    int gd = DETECT, gm;  
    initgraph(&gd, &gm, "..\\bgi"); /*Initializing the graphics system*/  
  
    circle(320, 225, 50); /*Calling the built in function circle()*/  
    getch();  
    closegraph(); /*Shutting the graphics system*/  
}
```

Output



Example 3B.37 Write a program to draw a rectangle.

Program

```
/*Program for demonstrating the use of built-in graphics functions of C*/  
#include<conio.h>  
#include<graphics.h>
```

```
#include<stdio.h>

void main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "..\\bgi"); /*Initializing the graphics system*/

    rectangle(320, 225, 50,100); /*Calling the built-in function rectangle()*/
    getch();
    closegraph(); /*Shutting the graphics system*/
}
```

Output



Example 3B.38 Write a program to create a 3D bar.

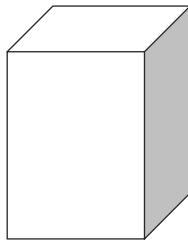
Program

```
/*Program for demonstrating the use of built-in graphics functions of C*/
#include<conio.h>
#include<graphics.h>
#include<stdio.h>

void main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "..\\bgi"); /*Initializing the graphics system*/

    bar3d(150, 50, 250,150, 10, 1); /*Calling the built-in graphics function bar3d()*/
    getch();
    closegraph(); /*Shutting the graphics system*/
}
```

Output



Example 3B.39 Write a program to create a shape and fill it with color.

Program

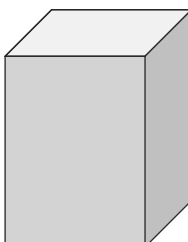
```
/*Program for demonstrating the use of built-in graphics functions of C*/
#include<conio.h>
#include<graphics.h>
#include<stdio.h>

void main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "..\\bgi"); /*Initializing the graphics system*/

    setfillstyle(SOLID_FILL,RED); /*Setting the fill pattern and color*/

    bar3d(150, 50, 250,150, 10, 1); /*Calling the built-in function bar3d()*/
    getch();
    closegraph(); /*Shutting the graphics system*/
}
```

Output





Just Remember

- We need to specify three things, namely, name, type and size, when we declare an array.
- Always remember that subscripts begin at 0 (not 1) and end at size -1.
- Defining the size of an array as a symbolic constant makes a program more scalable.
- Be aware of the difference between the "kth element" and the "element k". The kth element has a subscript k-1, whereas the element k has a subscript of k itself.
- Do not forget to initialize the elements; otherwise they will contain "garbage".
- Supplying more initializers in the initializer list is a compile time error.
- Use of invalid subscript is one of the common errors. An incorrect or invalid index may cause unexpected results.
- When using expressions for subscripts, make sure that their results do not go outside the permissible range of 0 to size -1. Referring to an element outside the array bounds is an error.
- When using control structures for looping through an array, use proper relational expressions to eliminate "off-by-one" errors. For example, for an array of size 5, the following **for** statements are wrong:

```
for (i = 1; i <= 5; i++)
for (i = 0; i <= 5; i++)
for (i = 0; i == 5; i++)
for (i = 0; i < 4; i++)
```

- Referring a two-dimensional array element like `x[i, j]` instead of `x[i][j]` is a compile time error.
- When initializing character arrays, provide enough space for the terminating null character.
- Make sure that the subscript variables have been properly initialized before they are used.
- Leaving out the subscript reference operator `[]` in an assignment operation is compile time error.
- During initialization of multi-dimensional arrays, it is an error to omit the array size for any dimension other than the first.
- An array is just like any other variable with the key exception that it stores multiple values of the same type.
- A loop construct is most commonly used for initializing an array when the input is to be provided by the end user.
- Use multiple arrays for realizing a matrix in a program.



Multiple Choice Questions

- Array is an example of which of the following?
 - Derived types
 - Fundamental types
 - User-defined types
 - None of the above
- Which of the following is not a data structure?
 - Linked list
 - Stack
 - Tree
 - Pointer
- `int a[n]` will reserve how many locations in the memory?
 - n
 - n-1
 - n+1
 - None of the above
- Which of the following is the correct syntax for initialisation of one-dimensional arrays?
 - `num[3]= {0 0 0 };`
 - `num[3]= {0, 0, 0 };`
 - `num[3]= {0; 0 ;0 };`
 - `num[3]= 0`
- Which of the following is the correct syntax for initialisation of two-dimensional arrays?

- (a) `table[2][3]={0,0,0,1,1,1}`
 (b) `table[2][3]={`
 `{0,0,0}`
 `{1,1,1}`
 `}`
 (c) `table[2][3]={0,1}, {0,1}, {0,1}`
 (d) None of the above
6. Which of the following multi-dimensional array declaration is correct for realising a 2×3 matrix?
 (a) `int m[2][3];` (b) `int m[3][2];`
 (c) `int m[3], m[2];` (d) None of the above
7. Which of the following is not the name of a sorting technique?
 (a) Bubble (b) Selection
 (c) Binary (d) Insertion
8. Which of the following is not the name of a searching technique?
 (a) Selection
 (b) Sequential
 (c) Binary
 (d) All of the above are searching techniques
9. In a character string, the last element stores which of the following values?
 (a) Last element of the string
 (b) Blank space
 (c) Null character
 (d) Newline character
10. Which of the following is the correct expression for retrieving the last-row-last-column value of a 5×4 array matrix A?
 (a) `A[4,5]` (b) `A[5,4]`
 (c) `A[5]-> A[4]` (d) `A[4]-> A[5]`



Case Study

1. Median of a List of Numbers

When all the items in a list are arranged in an order, the middle value which divides the items into two parts with equal number of items on either side is called the *median*. Odd number of items have just one middle value while even number of items have two middle values. The median for even number of items is therefore designated as the average of the two middle values.

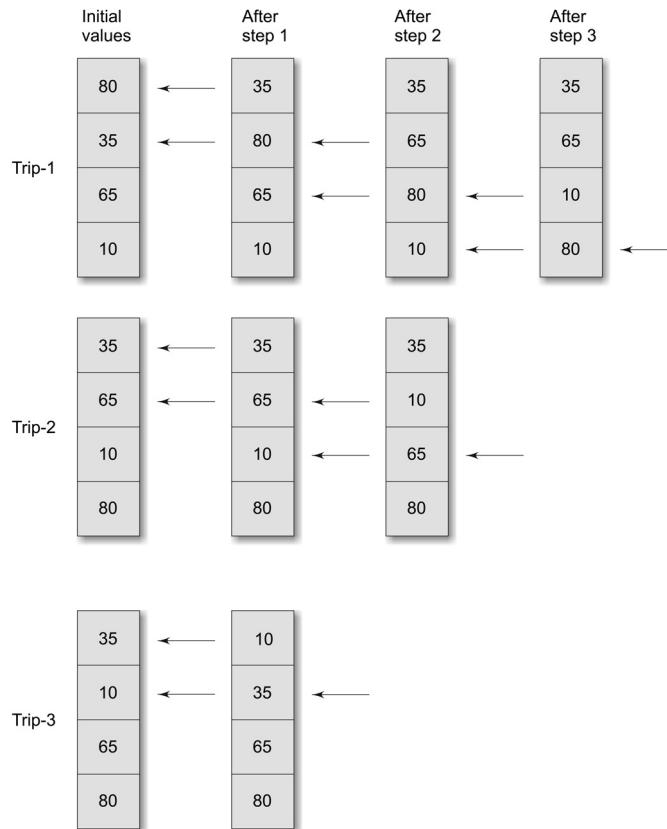
The major steps for finding the median are as follows:

1. Read the items into an array while keeping a count of the items.
2. Sort the items in increasing order.
3. Compute median.

The program and sample output are shown in Fig. 3B.11. The sorting algorithm used is as follows:

1. Compare the first two elements in the list, say `a[1]`, and `a[2]`. If `a[2]` is smaller than `a[1]`, then interchange their values.
2. Compare `a[2]` and `a[3]`; interchange them if `a[3]` is smaller than `a[2]`.
3. Continue this process till the last two elements are compared and interchanged.
4. Repeat the above steps $n-1$ times.

In repeated trips through the array, the smallest elements 'bubble up' to the top. Because of this bubbling up effect, this algorithm is called *bubble sorting*. The bubbling effect is illustrated below for four items.



During the first trip, three pairs of items are compared and interchanged whenever needed. It should be noted that the number 80, the largest among the items, has been moved to the bottom at the end of the first trip. This means that the element 80 (the last item in the new list) need not be considered any further. Therefore, trip-2 requires only two pairs to be compared. This time, the number 65 (the second largest value) has been moved down the list. Notice that each trip brings the smallest value 10 up by one level.

The number of steps required in a trip is reduced by one for each trip made. The entire process will be over when a trip contains only one step. If the list contains n elements, then the number of comparisons involved would be $n(n-1)/2$.

Program

```
#define N 10
main( )
{
    int i,j,n;
    float median,a[N],t;
    printf("Enter the number of items\n");
    scanf("%d", &n);
    /* Reading items into array a */
    printf("Input %d values \n",n);
    for (i = 1; i <= n ; i++)
```

```

        scanf("%f", &a[i]);
/* Sorting begins */
for (i = 1 ; i <= n-1 ; i++)
{
    /* Trip-i begins */
    for (j = 1 ; j <= n-i ; j++)
    {
        if (a[j] <= a[j+1])
        { /* Interchanging values */
            t = a[j];
            a[j] = a[j+1];
            a[j+1] = t;
        }
        else
            continue ;
    }
} /* sorting ends */
/* calculation of median */
if ( n % 2 == 0)
    median = (a[n/2] + a[n/2+1])/2.0 ;
else
    median = a[n/2 + 1];
/* Printing */
for (i = 1 ; i <= n ; i++)
    printf("%f ", a[i]);
printf("\n\nMedian is %f\n", median);
}

```

Output

```

Enter the number of items
5
Input 5 values
1.111 2.222 3.333 4.444 5.555
5.555000 4.444000 3.333000 2.222000 1.111000

Median is 3.333000

Enter the number of items
6
Input 6 values
3 5 8 9 4 6
9.000000 8.000000 6.000000 5.000000 4.000000 3.000000

Median is 5.500000

```

Fig. 3B.11 Program to sort a list of numbers and to determine median**2. Calculation of Standard Deviation**

In statistics, standard deviation is used to measure deviation of data from its mean. The formula for calculating standard deviation of **n** items is

$$s = \sqrt{\text{variance}}$$

where

$$\text{variance} = \frac{1}{n} \sum_{i=1}^n (x_i - m)^2$$

and

$$m = \text{mean} = \frac{1}{n} \sum_{i=1}^n x_i$$

The algorithm for calculating the standard deviation is as follows:

1. Read **n** items.
2. Calculate sum and mean of the items.
3. Calculate variance.
4. Calculate standard deviation.

Complete program with sample output is shown in Fig. 3B.12.

Program

```
#include <math.h>
#define MAXSIZE 100
main( )
{
    int i,n;
    float value [MAXSIZE], deviation,
          sum,sumsqr,mean,variance,stddeviation;
    sum = sumsqr = n = 0 ;
    printf("Input values: input -1 to end \n");
    for (i=1; i< MAXSIZE ; i++)
    {
        scanf("%f", &value[i]);
        if (value[i] == -1)
            break;
        sum += value[i];
        n += 1;
    }
    mean = sum/(float)n;
    for (i = 1 ; i<= n; i++)
    {
        deviation = value[i] - mean;
        sumsqr += deviation * deviation;
    }
    variance = sumsqr/(float)n ;
    stddeviation = sqrt(variance) ;
    printf("\nNumber of items : %d\n",n);
    printf("Mean : %f\n", mean);
    printf("Standard deviation : %f\n", stddeviation);
}
```

Output

```
Input values: input -1 to end
65 9 27 78 12 20 33 49 -1

Number of items : 8

Mean : 36.625000
Standard deviation : 23.510303
```

Fig. 3B.12 Program to calculate standard deviation

3. Evaluating a Test

A test consisting of 25 multiple-choice items is administered to a batch of 3 students. Correct answers and student responses are tabulated as shown below:

	Items																								
Correct answers	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
Student 1																									
Student 2																									
Student 3																									

The algorithm for evaluating the answers of students is as follows:

1. Read correct answers into an array.
2. Read the responses of a student and count the correct ones.
3. Repeat step-2 for each student.
4. Print the results.

A program to implement this algorithm is given in Fig. 3B.13. The program uses the following arrays:

- key[i] - To store correct answers of items
- response[i] - To store responses of students
- correct[i] - To identify items that are answered correctly.

```

Program
#define STUDENTS 3
#define ITEMS 25
main( )
{
    char key[ITEMS+1],response[ITEMS+1];
    int count, i, student,n,
        correct[ITEMS+1];
    /* Reading of Correct answers */
    printf("Input key to the items\n");
    for(i=0; i < ITEMS; i++)
        scanf("%c",&key[i]);
    key[i] = '\0';
    /* Evaluation begins */
    for(student = 1; student <= STUDENTS ; student++)
    {
        /*Reading student responses and counting correct ones*/
        count = 0;
        printf("\n");
        printf("Input responses of student-%d\n",student);
        for(i=0; i < ITEMS ; i++)
            scanf("%c",&response[i]);
        response[i] = '\0';
        for(i=0; i < ITEMS; i++)

```

(Contd.)

```

        correct[i] = 0;
    for(i=0; i < ITEMS ; i++)
        if(response[i] == key[i])
        {
            count = count +1 ;
            correct[i] = 1 ;
        }
    /* printing of results */
    printf("\n");
    printf("Student-%d\n", student);
    printf("Score is %d out of %d\n",count, ITEMS);
    printf("Response to the items below are wrong\n");
    n = 0;
    for(i=0; i < ITEMS ; i++)
        if(correct[i] == 0)
        {
            printf("%d ",i+1);
            n = n+1;
        }
    if(n == 0)
        printf("NIL\n");
    printf("\n");
    } /* Go to next student */
/* Evaluation and printing ends */
}

```

Output

```

Input key to the items
abcdabcdabcdabcdabcdabcd

Input responses of student-1
abcdabcdabcdabcdabcdabcd

Student-1
Score is 25 out of 25
Response to the following items are wrong
NIL

Input responses of student-2
abddcbabcdabcdcccccccc

Student-2
Score is 14 out of 25
Response to the following items are wrong
5 6 7 8 17 18 19 21 22 23 25

Input responses of student-3
aaaaaaaaaaaaaaaaaaaaaaaa

Student-3
Score is 7 out of 25
Response to the following items are wrong
2 3 4 6 7 8 10 11 12 14 15 16 18 19 20 22 23 24

```

Fig. 3B.13 Program to evaluate responses to a multiple-choice test

4. Production and Sales Analysis

A company manufactures five categories of products and the number of items manufactured and sold are recorded product-wise every week in a month. The company reviews its production schedule at every month-end. The review may require one or more of the following information:

- Value of weekly production and sales.
- Total value of all the products manufactured.
- Total value of all the products sold.
- Total value of each product, manufactured and sold.

Let us represent the products manufactured and sold by two two-dimensional arrays M and S respectively. Then,

M =	M11	M12	M13	M14	M15
	M21	M22	M23	M24	M25
	M31	M32	M33	M34	M35
	M41	M42	M43	M44	M45
S =	S11	S12	S13	S14	S15
	S21	S22	S23	S24	S25
	S31	S32	S33	S34	S35
	S41	S42	S43	S44	S45

where M_{ij} represents the number of j th type product manufactured in i th week and S_{ij} the number of j th product sold in i th week. We may also represent the cost of each product by a single dimensional array C as follows:

C =	C1	C2	C3	C4	C5
-----	----	----	----	----	----

where C_j is the cost of j th type product.

We shall represent the value of products manufactured and sold by two value arrays, namely, **Mvalue** and **Svalue**. Then,

$$Mvalue[i][j] = M_{ij} \times C_j$$

$$Svalue[i][j] = S_{ij} \times C_j$$

A program to generate the required outputs for the review meeting is shown in Fig. 3B.14. The following additional variables are used:

$Mweek[i]$ = Value of all the products manufactured in week i

$$= \sum_{j=1}^5 Mvalue[i][j]$$

$Sweek[i]$ = Value of all the products in week i

$$= \sum_{j=1}^5 Svalue[i][j]$$

$Mproduct[j]$ = Value of j th type product manufactured during the month

$$= \sum_{i=1}^4 Mvalue[i][j]$$

Sproduct[j] = Value of jth type product sold during the month

$$= \sum_{i=1}^4 \text{Svalue}[i][j]$$

Mtotal = Total value of all the products manufactured during the month

$$= \sum_{i=1}^4 \text{Mweek}[i] = \sum_{j=1}^5 \text{Mproduct}[j]$$

Stotal = Total value of all the products sold during the month

$$= \sum_{i=1}^4 \text{Sweek}[i] = \sum_{j=1}^5 \text{Sproduct}[j]$$

Program

```
main( )
{
    int M[5][6],S[5][6],C[6],
        Mvalue[5][6],Svalue[5][6],
        Mweek[5], Sweek[5],
        Mproduct[6], Sproduct[6],
        Mtotal, Stotal, i,j,number;
    /*      Input data      */
    printf (" Enter products manufactured week_wise \n");
    printf (" M11,M12,—, M21,M22,— etc\n");
    for(i=1; i<=4; i++)
    for(j=1;j<=5; j++)
        scanf("%d",&M[i][j]);
    printf (" Enter products sold week_wise\n");
    printf (" S11,S12,—, S21,S22,— etc\n");
    for(i=1; i<=4; i++)
    for(j=1; j<=5; j++)
        scanf("%d", &S[i][j]);
    printf(" Enter cost of each product\n");
    for(j=1; j <=5; j++)
        scanf("%d",&C[j]);
    /*      Value matrices of production and sales */
    for(i=1; i<=4; i++)
    for(j=1; j<=5; j++)
    {
        Mvalue[i][j] = M[i][j] * C[j];
        Svalue[i][j] = S[i][j] * C[j];
    }
    /*      Total value of weekly production and sales */
    for(i=1; i<=4; i++)
    {
        Mweek[i] = 0 ;
        Sweek[i] = 0 ;
        for(j=1; j<=5; j++)
        {
            Mweek[i] += Mvalue[i][j];
```

(Contd.)

```

        Sweek[i] += Svalue[i][j];
    }
}
/* Monthly value of product_wise production and sales */
for(j=1; j<=5; j++)
{
    Mproduct[j] = 0 ;
    Sproduct[j] = 0 ;
    for(i=1; i<=4; i++)
    {
        Mproduct[j] += Mvalue[i][j];
        Sproduct[j] += Svalue[i][j];
    }
}
/* Grand total of production and sales values */
Mtotal = Stotal = 0;
for(i=1; i<=4; i++)
{
    Mtotal += Mweek[i];
    Stotal += Sweek[i];
}
/*****
    Selection and printing of information required
*****/
printf("\n\n");
printf(" Following is the list of things you can\n");
printf(" request for. Enter appropriate item number\n");
printf(" and press RETURN Key\n\n");
printf(" 1.Value matrices of production & sales\n");
printf(" 2.Total value of weekly production & sales\n");
printf(" 3.Product_wise monthly value of production &");
printf(" sales\n");
printf(" 4.Grand total value of production & sales\n");
printf(" 5.Exit\n");
number = 0;
while(1)
{
    /* Beginning of while loop */
    printf("\n\n ENTER YOUR CHOICE:");
    scanf("%d",&number);
    printf("\n");
    if(number == 5)
    {
        printf(" GOOD BYE\n\n");
        break;
    }
    switch(number)
    { /* Beginning of switch */
/* VALUE MATRICES */
        case 1:
            printf(" VALUE MATRIX OF PRODUCTION\n\n");
            for(i=1; i<=4; i++)

```



```

        {
            printf(" Week(%d)\t",i);
            for(j=1; j <=5; j++)
                printf("%7d", Mvalue[i][j]);
            printf("\n");
        }
        printf("\n VALUE MATRIX OF SALES\n\n");
        for(i=1; i <=4; i++)
        {
            printf(" Week(%d)\t",i);
            for(j=1; j <=5; j++)
                printf("%7d", Svalue[i][j]);
            printf("\n");
        }
        break;
/* WEEKLY ANALYSIS */
        case 2:
            printf(" TOTAL WEEKLY  PRODUCTION & SALES\n\n");
            printf("                PRODUCTION    SALES\n");
            printf("                -----    —   \n");
            for(i=1; i <=4; i++)
            {
                printf(" Week(%d)\t", i);
                printf("%7d\t%7d\n", Mweek[i], Sweek[i]);
            }
            break;
/* PRODUCT WISE ANALYSIS */
        case 3:
            printf(" PRODUCT_WISE TOTAL PRODUCTION &");
            printf(" SALES\n\n");
            printf("                PRODUCTION SALES\n");
            printf("                ———    —   \n");
            for(j=1; j <=5; j++)
            {
                printf(" Product(%d)\t", j);
                printf("%7d\t%7d\n", Mproduct[j], Sproduct[j]);
            }
            break;
/* GRAND TOTALS */
        case 4:
            printf(" GRAND TOTAL OF PRODUCTION & SALES\n");
            printf("\n Total production = %d\n", Mtotal);
            printf(" Total sales = %d\n", Stotal);
            break;
/* D E F A U L T */
        default :
            printf(" Wrong choice, select again\n\n");
            break;
    } /* End of switch */
} /* End of while loop */
printf(" Exit from the program\n\n");

```

(Contd.)

3B.74 Computer Programming

```
 } /* End of main */
```

Output

```
Enter products manufactured week_wise
```

```
  M11,  M12, ---, M21, M22, --- etc
```

```
 11   15   12   14   13
 13   13   14   15   12
 12   16   10   15   14
 14   11   15   13   12
```

```
Enter products sold week_wise
```

```
  S11,S12,---, S21,S22,--- etc
```

```
 10   13   9   12   11
 12   10   12   14   10
 11   14   10   14   12
 12   10   13   11   10
```

```
Enter cost of each product
```

```
10 20 30 15 25
```

Following is the list of things you can request for. Enter appropriate item number and press RETURN key

- 1.Value matrices of production & sales
- 2.Total value of weekly production & sales
- 3.Product_wise monthly value of production & sales
- 4.Grand total value of production & sales
- 5.Exit

```
ENTER YOUR CHOICE:1
```

```
VALUE MATRIX OF PRODUCTION
```

Week(1)	110	300	360	210	325
Week(2)	130	260	420	225	300
Week(3)	120	320	300	225	350
Week(4)	140	220	450	185	300

```
VALUE MATRIX OF SALES
```

Week(1)	100	260	270	180	275
Week(2)	120	200	360	210	250
Week(3)	110	280	300	210	300
Week(4)	120	200	390	165	250

```
ENTER YOUR CHOICE:2
```

```
TOTAL WEEKLY  PRODUCTION &  SALES
              PRODUCTION    SALES
```

	<u> </u>	<u> </u>
Week(1)	1305	1085
Week(2)	1335	1140
Week(3)	1315	1200
Week(4)	1305	1125

```
ENTER YOUR CHOICE:3
```

```
PRODUCT_WISE TOTAL      PRODUCTION &  SALES
                        PRODUCTION    SALES
```

	<u> </u>	<u> </u>
Product(1)	500	450
Product(2)	1100	940

(Contd.)

```

        Product(3) 1530 1320
        Product(4) 855 765
        Product(5) 1275 1075

ENTER YOUR CHOICE:4

GRAND TOTAL OF PRODUCTION & SALES

Total production = 5260
Total sales      = 4550
ENTER YOUR CHOICE:5
GOOD BYE
Exit from the program

```

Fig. 3B.14 Program for production and sales analysis



Review Questions

- 3B.1 State whether the following statements are *true* or *false*.
- The type of all elements in an array must be the same.
 - When an array is declared, C automatically initializes its elements to zero.
 - An expression that evaluates to an integral value may be used as a subscript.
 - Accessing an array outside its range is a compile time error.
 - A **char** type variable cannot be used as a subscript in an array.
 - An unsigned long int type can be used as a subscript in an array.
 - In C, by default, the first subscript is zero.
 - When initializing a multidimensional array, not specifying all its dimensions is an error.
 - When we use expressions as a subscript, its result should be always greater than zero.
 - In C, we can use a maximum of 4 dimensions for an array.
 - In declaring an array, the array size can be a constant or variable or an expression.
 - The declaration `int x[2] = {1,2,3};` is illegal.
- 3B.2 Fill in the blanks in the following statements.
- The variable used as a subscript in an array is popularly known as _____ variable.
 - An array can be initialized either at compile time or at _____.
 - An array created using **malloc** function at run time is referred to as _____ array.
 - An array that uses more than two subscript is referred to as _____ array.
 - _____ is the process of arranging the elements of an array in order.
- 3B.3 Identify errors, if any, in each of the following array declaration statements, assuming that ROW and COLUMN are declared as symbolic constants:
- `int score (100);`
 - `float values [10,15];`
 - `float average[ROW],[COLUMN];`
 - `char name[15];`
 - `int sum[];`
 - `double salary [i + ROW]`
 - `long int number [ROW]`
 - `int array x[COLUMN];`
- 3B.4 Identify errors, if any, in each of the following initialization statements.

3B.76 Computer Programming

- (a) `int number[] = {0,0,0,0};`
- (b) `float item[3][2] = {0,1,2,3,4,5};`
- (c) `char word[] = {'A', 'R', 'R', 'A', 'Y'};`
- (d) `int m[2,4] = {(0,0,0,0)(1,1,1,1)};`
- (e) `float result[10] = 0;`

3B.5 Assume that the arrays A and B are declared as follows:

```
int A[5][4];
```

```
float B[4];
```

Find the errors (if any) in the following program segments.

- (a) `for (i=1; i<=5; i++)`
`for(j=1; j<=4; j++)`
`A[i][j] = 0;`
- (b) `for (i=1; i<4; i++)`
`scanf("%f", B[i]);`
- (c) `for (i=0; i<=4; i++)`
`B[i] = B[i]+i;`
- (d) `for (i=4; i>=0; i--)`
`for (j=0; j<4; j++)`
`A[i][j] = B[j] + 1.0;`

3B.6 Write a **for** loop statement that initializes all the diagonal elements of an array to one and others to zero as shown below. Assume 5 rows and 5 columns.

1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
.
.
.
.
.
0	0	0	0	0	1

3B.7 We want to declare a two-dimensional integer type array called **matrix** for 3 rows and 5 columns. Which of the following declarations are correct?

- (a) `int maxtrix [3],[5];`
- (b) `int matrix [5] [3];`
- (c) `int matrix [1+2] [2+3];`
- (d) `int matrix [3,5];`
- (e) `int matrix [3] [5];`

3B.8 Which of the following initialization statements are correct?

- (a) `char str1[4] = "GOOD";`
- (b) `char str2[] = "C";`
- (c) `char str3[5] = "Moon";`
- (d) `char str4[] = {'S', 'U', 'N'};`
- (e) `char str5[10] = "Sun";`

3B.9 What is a data structure? Why is an array called a data structure?

3B.10 What is a dynamic array? How is it created? Give a typical example of use of a dynamic array.

3B.11 What is the error in the following program?

```
main ( )
{
    int x ;
    float y [ ] ;
    .....
}
```

3B.12 What happens when an array with a specified size is assigned

- (a) with values fewer than the specified size; and
- (b) with values more than the specified size.

3B.13 Discuss how initial values can be assigned to a multidimensional array.

3B.14 What is the output of the following program?

```
main ( )
{
    int m [ ] = { 1,2,3,4,5 }
    int x, y = 0;
    for (x = 0; x < 5; x++ )
        y = y + m [ x ];
    printf("%d", y) ;
}
```

3B.15 What is the output of the following program?

```
main ( )
{
    chart string [ ] = "HELLO WORLD"
    ;
    int m;
    for (m = 0; string [m] != '\0';
m++ )
        if ( (m%2) == 0)
            printf("%c", string [m] );
}
```



Programming Exercises

- 3B.1 Write a program for fitting a straight line through a set of points (x_i, y_i) , $i = 1, \dots, n$.

The straight line equation is

$$y = mx + c$$

and the values of m and c are given by

$$m = \frac{n \sum (x_i y_i) - (\sum x_i)(\sum y_i)}{n(\sum x_i^2) - (\sum x_i)^2}$$

$$c = \frac{1}{n}(\sum y_i - m \sum x_i)$$

All summations are from 1 to n .

- 3B.2 The daily maximum temperatures recorded in 10 cities during the month of January (for all 31 days) have been tabulated as follows:

City

Day 1 2 3 ----- 10

1			-----	
2				
3				
-				
-				
-				
-				
31				

Write a program to read the table elements into a two-dimensional array **temperature**, and to find the city and day corresponding to

- (a) the highest temperature and
(b) the lowest temperature.

- 3B.3 An election is contested by 5 candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and count the votes cast for each candidate using an array variable **count**. In case, a number read is outside the range 1 to 5, the ballot should be considered as a 'spoilt ballot' and the program should also count the number of spoilt ballots.

- 3B.4 The following set of numbers is popularly known as Pascal's triangle.

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
- - - - -
- - - - -

```

If we denote rows by i and columns by j , then any element (except the boundary elements) in the triangle is given by

$$p_{ij} = p_{i-1, j-1} + p_{i-1, j}$$

Write a program to calculate the elements of the Pascal triangle for 10 rows and print the results.

- 3B.5 The annual examination results of 100 students are tabulated as follows:

Roll No.	Subject 1	Subject 2	Subject 3
.			
.			
.			

Write a program to read the data and determine the following:

- (a) Total marks obtained by each student.
(b) The highest marks in each subject and the Roll No. of the student who secured it.
(c) The student who obtained the highest total marks.

- 3B.6 Given are two one-dimensional arrays A and B which are sorted in ascending order. Write a program to **merge** them into a single sorted array C that contains every item from arrays A and B, in ascending order.

- 3B.7 Two matrices that have the same number of rows and columns can be multiplied to

produce a third matrix. Consider the following two matrices.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{12} & a_{22} & \dots & a_{2n} \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ a_{n1} & \dots & \dots & a_{nn} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{12} & b_{22} & \dots & b_{2n} \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ b_{n1} & \dots & \dots & b_{nn} \end{bmatrix}$$

The product of **A** and **B** is a third matrix **C** of size $n \times n$ where each element of **C** is given by the following equation.

$$C_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

Write a program that will read the values of elements of **A** and **B** and produce the product matrix **C**.

- 3B.8 Write a program that fills a five-by-five matrix as follows:

- Upper left triangle with +1s
- Lower right triangle with -1s
- Right to left diagonal with zeros

Display the contents of the matrix using not more than two **printf** statements

- 3B.9 Selection sort is based on the following idea: Selecting the largest array element and swapping it with the last array element leaves an unsorted list whose size is 1 less than the size of the original list. If we repeat this step again on the unsorted list we will have an ordered list of size 2 and an unsorted list size $n-2$. When we repeat this until the size of the unsorted list becomes one, the result will be a sorted list.

Write a program to implement this algorithm.

- 3B.10 Develop a program to implement the binary search algorithm. This technique compares the search key value with the value of the element that is midway in a “sorted” list. Then;

- (a) If they match, the search is over.
- (b) If the search key value is less than the middle value, then the first half of the list contains the key value.
- (c) If the search key value is greater than the middle value, then the second half contains the key value.

Repeat this “divide-and-conquer” strategy until we have a match. If the list is reduced to one non-matching element, then the list does not contain the key value.

Use the sorted list created in Exercise 10.9 or use any other sorted list.

- 3B.11 Write a program that will compute the length of a given character string.

- 3B.12 Write a program that will count the number occurrences of a specified character in a given line of text. Test your program.

- 3B.13 Write a program to read a matrix of size $m \times n$ and print its transpose.

- 3B.14 Every book published by international publishers should carry an International Standard Book Number (ISBN). It is a 10 character 4 part number as shown below.

0-07-041183-2

The first part denotes the region, the second represents publisher, the third identifies the book and the fourth is the check digit. The check digit is computed as follows:

Sum = $(1 \times \text{first digit}) + (2 \times \text{second digit}) + (3 \times \text{third digit}) + \dots + (9 \times \text{ninth digit})$.

Check digit is the remainder when sum is divided by 11. Write a program that reads a given ISBN number and checks whether it represents a valid ISBN.

- 3B.15 Write a program to read two matrices **A** and **B** and print the following:

- (a) $A + B$; and
- (b) $A - B$.



Key Terms

- **Array:** It is a fixed-size sequenced collection of elements of the same data type.
- **Structured data types:** These data types provide an organizational scheme that shows the relationships among the individual elements and facilitate efficient data manipulations.
- **One-dimensional array:** In this array, a list of items is given one variable name using only one subscript.
- **Sorting:** It is the process of arranging elements in the list according to their values, in ascending or descending order.
- **Searching:** It is the process of finding the location of the specified element in a list.
- **Two-dimensional array:** It is used to store a table of values with two dimensions.
- **Multi-dimensional array:** It is used to store data with three or more dimensions.
- **Static memory allocation:** It is the process of allocating memory at compile time.
- **Static arrays:** These are the arrays, which receive static memory allocation.
- **Dynamic memory allocation:** It is the process of allocating memory at run time.
- **Dynamic arrays:** These are the arrays created at run time.

UNIT

4_A Pointers

4A.1 INTRODUCTION

A pointer is a derived data type in C. It is built from one of the fundamental data types available in C. Pointers contain memory addresses as their values. Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

Pointers are undoubtedly one of the most distinct and exciting features of C language. It has added power and flexibility to the language. Although they appear little confusing and difficult to understand for a beginner, they are a powerful tool and handy to use once they are mastered.

Pointers are used frequently in C, as they offer a number of benefits to the programmers. They include:

1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

Of course, the real power of C lies in the proper use of pointers. In this chapter, we will examine the pointers in detail and illustrate how to use them in program development.

4A.2 UNDERSTANDING POINTERS

The computer's memory is a sequential collection of *storage cells* as shown in Fig. 4A.1. Each cell, commonly known as a *byte*, has a number called *address* associated with it. Typically, the addresses are numbered

4A.2 Computer Programming

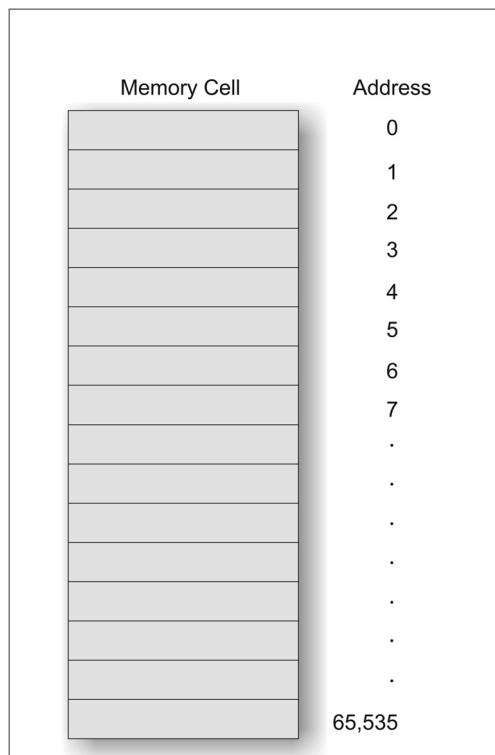


Fig. 4A.1 Memory organisation

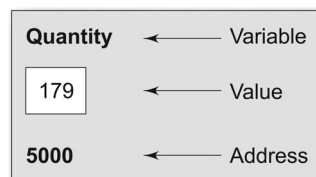


Fig. 4A.2 Representation of a variable

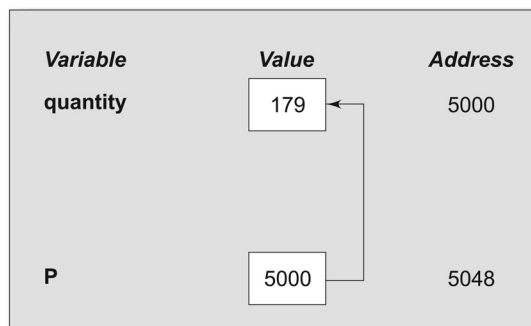


Fig. 4A.3 Pointer variable

consecutively, starting from zero. The last address depends on the memory size. A computer system having 64 K memory will have its last address as 65,535.

Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number. Consider the following statement

```
int quantity = 179;
```

This statement instructs the system to find a location for the integer variable **quantity** and puts the value 179 in that location. Let us assume that the system has chosen the address location 5000 for **quantity**. We may represent this as shown in Fig. 4A.2. (Note that the address of a variable is the address of the first byte occupied by that variable.)

During execution of the program, the system always associates the name **quantity** with the address 5000. (This is something similar to having a house number as well as a house name.) We may have access to the value 179 by using either the name **quantity** or the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables, that can be stored in memory, like any other variable. Such variables that hold memory addresses are called *pointer variables*. A pointer variable is, therefore, nothing but a variable that contains an address, which is a location of another variable in memory.

Remember, since a pointer is a variable, its value is also stored in the memory in another location. Suppose, we assign the address of **quantity** to a variable **p**. The link between the variables **p** and **quantity** can be visualized as shown in Fig. 4A.3. The address of **p** is 5048.

Since the value of the variable **p** is the address of the variable **quantity**, we may access the value of **quantity** by using the value of **p** and therefore, we say that the variable **p** ‘points’ to the variable **quantity**. Thus, **p** gets the name ‘pointer’. (We are not really concerned about the actual values of pointer variables. They may be different everytime we run the program. What we are concerned about is the relationship between the variables **p** and **quantity**.)

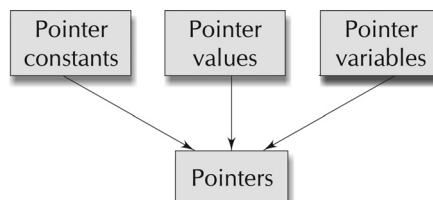
Underlying Concepts of Pointers

Pointers are built on the three underlying concepts as illustrated below:

Memory addresses within a computer are referred to as *pointer constants*. We cannot change them; we can only use them to store data values. They are like house numbers.

We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as *pointer value*. The pointer value (i.e. the address of a variable) may change from one run of the program to another.

Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a *pointer variable*.



4A.3 ACCESSING THE ADDRESS OF A VARIABLE

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. How can we then determine the address of a variable? This can be done with the help of the operator **&** available in C. We have already seen the use of this *address operator* in the **scanf** function. The operator **&** immediately preceding a variable returns the address of the variable associated with it. For example, the statement

```
p = &quantity;
```

would assign the address 5000 (the location of **quantity**) to the variable **p**. The **&** operator can be remembered as ‘address of’.

The **&** operator can be used only with a simple variable or an array element. The following are illegal uses of address operator:

1. **&125** (pointing at constants).
2. `int x[10];`
&x (pointing at array names).
3. **&(x+y)** (pointing at expressions).

If **x** is an array, then expressions such as

&x[0] and &x[i+3]

are valid and represent the addresses of 0th and (i+3)th elements of **x**.

Example 4A.1 Write a program to print the address of a variable along with its value.

The program shown in Fig. 4A.4, declares and initializes four variables and then prints out these values with their respective storage locations. Note that we have used **%u** format for printing address values. Memory addresses are unsigned integers.

4A.4 Computer Programming

```
Program
main()
{
    char    a;
    int     x;
    float   p, q;

    a  = 'A';
    x  = 125;
    p  = 10.25, q = 18.76;
    printf("%c is stored at addr %u.\n", a, &a);
    printf("%d is stored at addr %u.\n", x, &x);
    printf("%f is stored at addr %u.\n", p, &p);
    printf("%f is stored at addr %u.\n", q, &q);
}
```

Output

```
A is stored at addr 4436.
125 is stored at addr 4434.
10.250000 is stored at addr 4442.
18.760000 is stored at addr 4438.
```

Fig. 4A.4 Accessing the address of a variable

4A.4 DECLARING POINTER VARIABLES

In C, every variable must be declared for its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

***data_type* **pt_name*;**

This tells the compiler three things about the variable ***pt_name***.

1. The asterisk (*) tells that the variable ***pt_name*** is a pointer variable.
2. ***pt_name*** needs a memory location.
3. ***pt_name*** points to a variable of type *data_type*.

For example,

```
int *p;          /* integer pointer */
```

declares the variable **p** as a pointer variable that points to an integer data type. Remember that the type **int** refers to the data type of the variable being pointed to by **p** and not the type of the value of the pointer. Similarly, the statement

```
float *x;        /* float pointer */
```

declares **x** as a pointer to a floating-point variable.

The declarations cause the compiler to allocate memory locations for the pointer variables **p** and **x**. Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown:



Pointer Declaration Style

Pointer variables are declared similarly as normal variables except for the addition of the unary `*` operator. This symbol can appear anywhere between the type name and the pointer variable name. Programmers use the following styles:

```
int*      p;      /* style 1 */
int       *p;     /* style 2 */
int      * p;     /* style 3 */
```

However, the style2 is becoming increasingly popular due to the following reasons:

1. This style is convenient to have multiple declarations in the same statement. Example:

```
int *p, x, *q;
```

2. This style matches with the format used for accessing the target values. Example:

```
int x, *p, y;
x = 10;
p = &x;
y = *p;          /* accessing x through p */
*p = 20;         /* assigning 20 to x */
```

We use in this book the style 2, namely,

```
int *p;
```

4A.5 INITIALIZATION OF POINTER VARIABLES

The process of assigning the address of a variable to a pointer variable is known as *initialization*. As pointed out earlier, all uninitialized pointers will have some unknown values that will be interpreted as memory addresses. They may not be valid addresses or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce erroneous results. It is therefore important to initialize pointer variables carefully before they are used in the program.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable. Example:

```
int quantity;
int *p;          /* declaration      */
p = &quantity;   /* initialization */
```

We can also combine the initialization with the declaration. That is,

```
int *p = &quantity;
```

is allowed. The only requirement here is that the variable **quantity** must be declared before the initialization takes place. Remember, this is an initialization of **p** and not ***p**.

We must ensure that the pointer variables always point to the corresponding type of data. For example,

```
float a, b;
int x, *p;
p = &a;      /* wrong */
b = *p;
```

will result in erroneous output because we are trying to assign the address of a **float** variable to an **integer pointer**. When we declare a pointer to be of **int** type, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step. For example,

```
int x, *p = &x;      /* three in one */
```

4A.6 Computer Programming

is perfectly valid. It declares **x** as an integer variable and **p** as a pointer variable and then initializes **p** to the address of **x**. And also remember that the target variable **x** is declared first. The statement

```
int *p = &x, x;
```

is not valid.

We could also define a pointer variable with an initial value of NULL or 0 (zero). That is, the following statements are valued

```
int *p = NULL;  
int *p = 0;
```

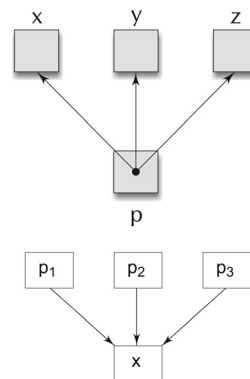
Pointer Flexibility

Pointers are flexible. We can make the same pointer to point to different data variables in different statements. Example;

```
int x, y, z, *p;  
.....  
p = &x;  
.....  
p = &y;  
.....  
p = &z;  
.....
```

We can also use different pointers to point to the same data variable. Example.

```
int x;  
int *p1 = &x;  
int *p2 = &x;  
int *p3 = &x;  
.....  
.....
```



With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

```
int *p = 5360;           / *absolute address */
```

4A.6 ACCESSING A VARIABLE THROUGH ITS POINTER

Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer? This is done by using another unary operator ***** (asterisk), usually known as the *indirection operator*. Another name for the indirection operator is the *dereferencing operator*. Consider the following statements:

```
int quantity, *p, n;  
quantity = 179;  
p = &quantity;  
n = *p;
```

The first line declares **quantity** and **n** as integer variables and **p** as a pointer variable pointing to an integer. The second line assigns the value 179 to **quantity** and the third line assigns the address of **quantity** to the pointer variable **p**. The fourth line contains the indirection operator *****. When the operator ***** is placed before a pointer variable in an expression (on the right-hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address. In this case, ***p** returns the value of the variable **quantity**,

because **p** is the address of **quantity**. The ***** can be remembered as ‘value at address’. Thus the value of **n** would be 179. The two statements

```
p = &quantity;
n = *p;
```

are equivalent to

```
n = *&quantity;
```

which in turn is equivalent to

```
n = quantity;
```

In C, the assignment of pointers and addresses is always done symbolically, by means of symbolic names. You cannot access the value stored at the address 5368 by writing `*5368`. It will not work. Example 4A.2 illustrates the distinction between pointer value and the value it points to.

Example 4A.2 Write a program to illustrate the use of indirection operator ‘*****’ to access the value pointed to by a pointer.

The program and output are shown in Fig. 4A.5. The program clearly shows how we can access the value of a variable using a pointer. You may notice that the value of the pointer **ptr** is 4104 and the value it points to is 10. Further, you may also note the following equivalences:

```
x = *(&x) = *ptr = y
&x = &*ptr
```

Program

```
main()
{
    int    x, y;
    int    *ptr;
    x = 10;
    ptr = &x;
    y = *ptr;
    printf("Value of x is %d\n\n",x);
    printf("%d is stored at addr %u\n", x, &x);
    printf("%d is stored at addr %u\n", *&x, &x);
    printf("%d is stored at addr %u\n", *ptr, ptr);
    printf("%d is stored at addr %u\n", ptr, &ptr);
    printf("%d is stored at addr %u\n", y, &y);
    *ptr = 25;
    printf("\nNow x = %d\n",x);
}
```

Output

```
Value of x is 10
10 is stored at addr 4104
10 is stored at addr 4104
10 is stored at addr 4104
4104 is stored at addr 4106
10 is stored at addr 4108
Now x = 25
```

Fig. 4A.5 Accessing a variable through its pointer

4A.8 Computer Programming

The actions performed by the program are illustrated in Fig. 4A.6. The statement **ptr = &x** assigns the address of **x** to **ptr** and **y = *ptr** assigns the value pointed to by the pointer **ptr** to **y**.

Note the use of the assignment statement

```
*ptr = 25;
```

This statement puts the value of 25 at the memory location whose address is the value of **ptr**. We know that the value of **ptr** is the address of **x** and therefore, the old value of **x** is replaced by 25. This, in effect, is equivalent to assigning 25 to **x**. This shows how we can change the value of a variable *indirectly* using a pointer and the *indirection operator*.

4A.7 CHAIN OF POINTERS

It is possible to make a pointer to point to another pointer, thus creating a chain of pointers as shown.

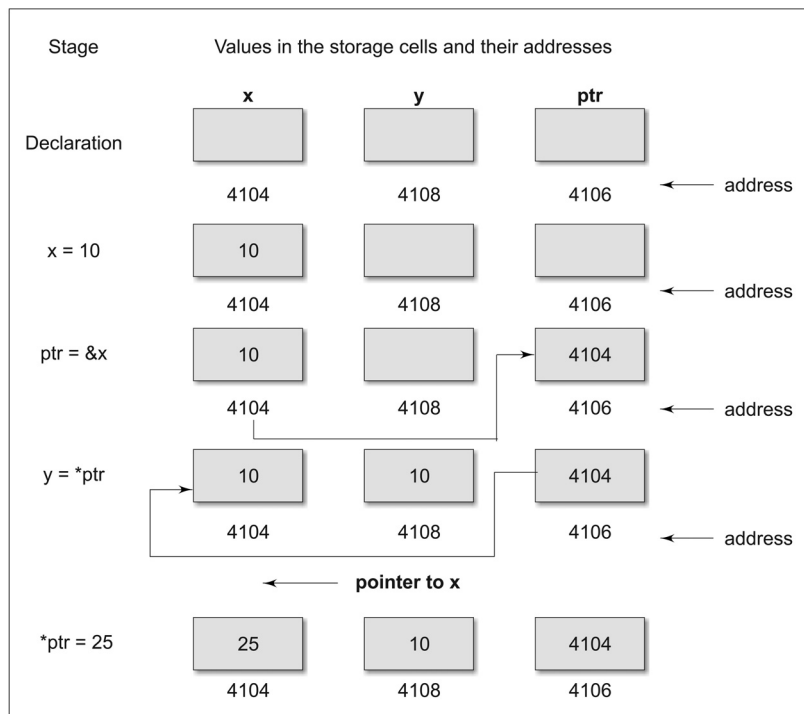
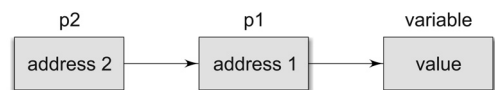


Fig. 4A.6 Illustration of pointer assignments

Here, the pointer variable **p2** contains the address of the pointer variable **p1**, which points to the location that contains the desired value. This is known as *multiple indirections*.

A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name. Example:

```
int **p2;
```

This declaration tells the compiler that **p2** is a pointer to a pointer of **int** type. Remember, the pointer **p2** is not a pointer to an integer, but rather a pointer to an integer pointer.

We can access the target value indirectly pointed to by pointer to a pointer by applying the indirection operator twice. Consider the following code:

```
main ( )
{
    int x, *p1,    **p2;
    x = 100;
    p1 = &x;      /* address of    x */
    p2 = &p1      /* address of    p1 */
    printf ("%d", **p2);
}
```

This code will display the value 100. Here, **p1** is declared as a pointer to an integer and **p2** as a pointer to a pointer to an integer.

4A.8 POINTER EXPRESSIONS

Like other variables, pointer variables can be used in expressions. For example, if **p1** and **p2** are properly declared and initialized pointers, then the following statements are valid.

```
y = *p1 * *p2;           same as (*p1) * (*p2)
sum = sum + *p1;
z = 5* - *p2/ *p1;       same as (5 * (- (*p2)))/(*p1)
*p2 = *p2 + 10;
```

Note that there is a blank space between / and * in the item3 above. The following is wrong.

```
z = 5* - *p2 /*p1;
```

The symbol /* is considered as the beginning of a comment and therefore the statement fails.

C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another. $p1 + 4$, $p2 - 2$ and $p1 - p2$ are all allowed. If **p1** and **p2** are both pointers to the same array, then **p2 - p1** gives the number of elements between **p1** and **p2**.

We may also use short-hand operators with the pointers.

```
p1++;
-p2;
sum += *p2;
```

In addition to arithmetic operations discussed above, pointers can also be compared using the relational operators. The expressions such as **p1 > p2**, **p1 == p2**, and **p1 != p2** are allowed. However, any comparison of pointers that refer to separate and unrelated variables makes no sense. Comparisons can be used meaningfully in handling arrays and strings.

We may not use pointers in division or multiplication. For example, expressions such as

p1 / p2 or p1 * p2 or p1 / 3

are not allowed. Similarly, two pointers cannot be added. That is, $p1 + p2$ is illegal.

Example 4A.3 Write a program to illustrate the use of pointers in arithmetic operations.

The program in Fig. 4A.7 shows how the pointer variables can be directly used in expressions. It also illustrates the order of evaluation of expressions. For example, the expression

4* - *p2 / *p1 + 10

is evaluated as follows:

((4 * (-(*p2))) / (*p1)) + 10

4A.10 Computer Programming

When $*p1 = 12$ and $*p2 = 4$, this expression evaluates to 9. Remember, since all the variables are of type `int`, the entire evaluation is carried out using the integer arithmetic.

Program

```
main()
{
    int a, b, *p1, *p2, x, y, z;
    a = 12;
    b = 4;
    p1 = &a;
    p2 = &b;
    x = *p1 * *p2 - 6;
    y = 4* - *p2 / *p1 + 10;
    printf("Address of a = %u\n", p1);
    printf("Address of b = %u\n", p2);
    printf("\n");
    printf("a = %d, b = %d\n", a, b);
    printf("x = %d, y = %d\n", x, y);
    *p2 = *p2 + 3;
    *p1 = *p2 - 5;
    z = *p1 * *p2 - 6;
    printf("\na = %d, b = %d,", a, b);
    printf(" z = %d\n", z);
}
```

Output

```
Address of a = 4020
Address of b = 4016
a = 12, b = 4
x = 42, y = 9
a = 2, b = 7, z = 8
```

Fig. 4A.7 Evaluation of pointer expressions

4A.9 POINTER INCREMENTS AND SCALE FACTOR

We have seen that the pointers can be incremented like

```
p1 = p2 + 2;
```

```
p1 = p1 + 1;
```

and so on. Remember, however, an expression like

```
p1++;
```

will cause the pointer **p1** to point to the next value of its type. For example, if **p1** is an integer pointer with an initial value, say 2800, then after the operation **p1 = p1 + 1**, the value of **p1** will be 2802, and not 2801. That is, when we increment a pointer, its value is increased by the 'length' of the data type that it points to. This length called the *scale factor*.

For an IBM PC, the length of various data types are as follows:

characters	1 byte
integers	2 bytes

floats	4 bytes
long integers	4 bytes
doubles	8 bytes

The number of bytes used to store various data types depends on the system and can be found by making use of the **sizeof** operator. For example, if **x** is a variable, then **sizeof(x)** returns the number of bytes needed for the variable. (Systems like Pentium use 4 bytes for storing integers and 2 bytes for short integers.)

Rules of Pointer Operations

The following rules apply when performing operations on pointer variables.

1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
4. An integer value may be added or subtracted from a pointer variable.
6. When two pointers point to the same array, one pointer variable can be subtracted from another.
7. When two pointers point to the objects of the same data types, they can be compared using relational operators.
8. A pointer variable cannot be multiplied by a constant.
9. Two pointer variables cannot be added.
10. A value cannot be assigned to an arbitrary address (i.e. `&x = 10;` is illegal).

4A.10 POINTERS AND ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element. Suppose we declare an array **x** as follows:

```
int x[5] = {1, 2, 3, 4, 5};
```

Suppose the base address of **x** is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows:

Elements	→	x[0]	x[1]	x[2]	x[3]	x[4]
Value	→	1	2	3	4	5
Address	→	1000	1002	1004	1006	1008

The name **x** is defined as a constant pointer pointing to the first element, **x[0]** and therefore the value of **x** is 1000, the location where **x[0]** is stored. That is,

```
x = &x[0] = 1000
```

If we declare **p** as an integer pointer, then we can make the pointer **p** to point to the array **x** by the following assignment:

```
p = x;
```

This is equivalent to

```
p = &x[0];
```

Now, we can access every value of **x** using **p++** to move from one element to another. The relationship between **p** and **x** is shown as:

4A.12 Computer Programming

```
p = &x[0] (= 1000)
p+1 = &x[1] (= 1002)
p+2 = &x[2] (= 1004)
p+3 = &x[3] (= 1006)
p+4 = &x[4] (= 1008)
```

You may notice that the address of an element is calculated using its index and the scale factor of the data type. For instance,

$$\begin{aligned}\text{address of } \mathbf{x[3]} &= \text{base address} + (3 \times \text{scale factor of } \mathbf{int}) \\ &= 1000 + (3 \times 2) = 1006\end{aligned}$$

When handling arrays, instead of using array indexing, we can use pointers to access array elements. Note that $\mathbf{*(p+3)}$ gives the value of $\mathbf{x[3]}$. The pointer accessing method is much faster than array indexing.

Example 4A.4 illustrates the use of pointer accessing method.

Example 4A.4 Write a program using pointers to compute the sum of all elements stored in an array.

The program shown in Fig. 4A.8 illustrates how a pointer can be used to traverse an array element. Since incrementing an array pointer causes it to point to the next element, we need only to add one to \mathbf{p} each time we go through the loop.

Program

```
main()
{
    int *p, sum, i;
    int x[5] = {5,9,6,3,7};
    i = 0;
    p = x;    /* initializing with base address of x */
    printf("Element   Value   Address\n\n");
    while(i < 5)
    {
        printf(" x[%d] %d %u\n", i, *p, p);
        sum = sum + *p;    /* accessing array element */
        i++, p++;          /* incrementing pointer */
    }
    printf("\n Sum      = %d\n", sum);
    printf("\n &x[0]    = %u\n", &x[0]);
    printf("\n p        = %u\n", p);
}
```

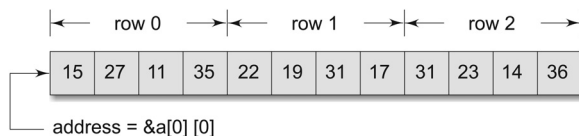
Output

Element	Value	Address
x[0]	5	166
x[1]	9	168
x[2]	6	170
x[3]	3	172
x[4]	7	174
Sum	= 55	
&x[0]	= 166	
p	= 176	

Fig. 4A.8 Accessing one-dimensional array elements using the pointer

4A.14 Computer Programming

The elements of **a** will be stored as:



If we declare **p** as an **int** pointer with the initial address of **&a[0][0]**, then

a[i][j] is equivalent to ***(p+4 × i+j)**

You may notice that, if we increment **i** by 1, the **p** is incremented by 4, the size of each row. Then the element **a[2][3]** is given by ***(p+2 × 4+3) = *(p+11)**.

This is the reason why, when a two-dimensional array is declared, we must specify the size of each row so that the compiler can determine the correct storage mapping.

4A.11 POINTERS AND CHARACTER STRINGS

Strings are treated like character arrays and therefore, they are declared and initialized as follows:

```
char str [5] = "good";
```

The compiler automatically inserts the null character **'\0'** at the end of the string. C supports an alternative method to create strings using pointer variables of type **char**. Example:

```
char *str = "good";
```

This creates a string for the literal and then stores its address in the pointer variable **str**.

The pointer **str** now points to the first character of the string "good" as:

We can also use the run-time assignment for giving values to a string pointer. Example

```
char * string1;  
string1 = "good";
```

Note that the assignment

```
string1 = "good";
```

is not a string copy, because the variable **string1** is a pointer, not a string.

C does not support copying one string to another through the assignment operation.)

We can print the content of the string **string1** using either **printf** or **puts** functions as follows:

```
printf("%s", string1);  
puts (string1);
```

Remember, although **string1** is a pointer to the string, it is also the name of the string. Therefore, we do not need to use indirection operator ***** here.

Like in one-dimensional arrays, we can use a pointer to access the individual characters in a string. This is illustrated by Example 4A.5.

Example 4A.5 Write a program using pointers to determine the length of a character string.

A program to count the length of a string is shown in Fig. 4A.10. The statement

```
char *cptr = name;
```

declares **cptr** as a pointer to a character and assigns the address of the first character of **name** as the initial value. Since a string is always terminated by the null character, the statement

```
while(*cptr != '\0')
```

is true until the end of the string is reached.

When the **while** loop is terminated, the pointer **cptr** holds the address of the null character. Therefore, the statement

```
length = cptr - name;
```

gives the length of the string **name**.

The output also shows the address location of each character. Note that each character occupies one memory cell (byte).

Program

```
main()
{
    char *name;
    int length;
    char *cptr = name;
    name = "DELHI";
    printf ("%s\n", name);
    while(*cptr != '\0')
    {
        printf("%c is stored at address %u\n", *cptr, cptr);
        cptr++;
    }
    length = cptr - name;
    printf("\nLength of the string = %d\n", length);
}
```

Output

```
DELHI
D is stored at address 54
E is stored at address 55
L is stored at address 56
H is stored at address 57
I is stored at address 58
Length of the string = 5
```

Fig. 4A.10 String handling by pointers

In C, a constant character string always represents a pointer to that string. And therefore the following statements are valid:

```
char *name;
name = "Delhi";
```

These statements will declare **name** as a pointer to character and assign to **name** the constant character string "Delhi". You might remember that this type of assignment does not apply to character arrays. The statements like

```
char name[20];
name = "Delhi";
```

do not work.

4A.12 ARRAY OF POINTERS

One important use of pointers is in handling of a table of strings. Consider the following array of strings:

```
char name [3][25]:
```

This says that the **name** is a table containing three names, each with a maximum length of 25 characters (including null character). The total storage requirements for the **name** table are 75 bytes.

[illegible]

We know that rarely the individual strings will be of equal lengths. Therefore, instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length. For example,

```
char *name[3] = {
    "New Zealand",
    "Australia",
    "India"
};
```

declares **name** to be an *array of three pointers* to characters, each pointer pointing to a particular name as:

This declaration allocates only 28 bytes, sufficient to hold all the characters as shown

N	e	w		Z	e	a	l	a	n	d	\0
A	u	s	t	r	a	l	i	a	\0		
I	n	d	i	a	\0						

The following statement would print out all the three names:

```
for(i = 0; i <= 2; i++)
    printf("%s\n", name[i]);
```

To access the j th character in the i th name, we may write as

```
* (name[i]+j)
```

The character arrays with the rows of varying length are called ‘ragged arrays’ and are better handled by pointers.

Remember the difference between the notations ***p[3]** and **(*p)[3]**. Since ***** has a lower precedence than **[]**, ***p[3]** declares **p** as an array of 3 pointers while **(*p)[3]** declares **p** as a pointer to an array of three elements.

4A.13 POINTERS AS FUNCTION ARGUMENTS

We have seen earlier that when an array is passed to a function as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements. If `x` is an array, when we call `sort(x)`, the address of `x[0]` is passed to the function `sort`. The function uses this address for manipulating the array elements. Similarly, we can pass the address of a variable as an argument to a function in the normal fashion. We used this method when discussing functions that return multiple values.

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of variables is known as ‘*call by reference*’. (You know, the process of passing the actual value of variables is known as “call by value”.) The function which is called by ‘reference’ can change the value of the variable used in the call.

Consider the following code:

```
main()
{
    int x;
    x = 20;
    change(&x);          /* call by reference or address */
    printf("%d\n",x);
}
change(int *p)
{
    *p = *p + 10;
}
```

When the function **change()** is called, the address of the variable **x**, not its value, is passed into the function **change()**. Inside **change()**, the variable **p** is declared as a pointer and therefore **p** is the address of the variable **x**. The statement,

$$*p = *p + 10;$$

means ‘add 10 to the value stored at the address **p**’. Since **p** represents the address of **x**, the value of **x** is changed from 20 to 30. Therefore, the output of the program will be 30, not 20.

Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function. Note that this mechanism is also known as “*call by address*” or “*pass by pointers*”

NOTE: C99 adds a new qualifier **restrict** to the pointers passed as function parameters.

Example 4A.6 Write a function using pointers to exchange the values stored in two locations in the memory.

The program in Fig. 4A.11 shows how the contents of two locations can be exchanged using their address locations. The function **exchange()** receives the addresses of the variables **x** and **y** and exchanges their contents.

```
Program
void exchange (int *, int *);      /* prototype */
main()
{
    int x, y;
    x = 100;
    y = 200;
    printf("Before exchange : x = %d   y = %d\n\n", x, y);
    exchange(&x,&y);               /* call */
    printf("After exchange  : x = %d   y = %d\n\n", x, y);
}
exchange (int *a, int *b)
{
    int t;
    t = *a;      /* Assign the value at address a to t */
    *a = *b;     /* put b into a */
    *b = t;      /* put t into b */
}
```

Output

```
Before exchange : x = 100   y = 200
After exchange  : x = 200   y = 100
```

Fig. 4A.11 Passing of pointers as function parameters

You may note the following points:

1. The function parameters are declared as pointers.
2. The dereferenced pointers are used in the function body.
3. When the function is called, the addresses are passed as actual arguments.

The use of pointers to access array elements is very common in C. We have used a pointer to traverse array elements in Example 4A.4. We can also use this technique in designing user-defined functions discussed in Chapter 4. Let us consider the problem sorting an array of integers discussed in Example 12.6.

The function **sort** may be written using pointers (instead of array indexing) as shown:

```
void sort (int m, int *x)
{
    int i j, temp;
    for (i=1; i<= m-1; i++)
        for (j=1; j<= m-1; j++)
            if (*(x+j-1) >= *(x+j))
            {
                temp = *(x+j- 1);
                *(x+j-1) = *(x+j);
                *(x+j) = temp;
            }
}
```

Note that we have used the pointer *x* (instead of array *x[]*) to receive the address of array passed and therefore the pointer *x* can be used to access the array elements (as pointed out in Section 4A.10). This function can be used to sort an array of integers as follows:

```
.....
int score[4] = {45, 90, 71, 83};
.....
sort(4, score); /* Function call */
.....
```

The calling function must use the following prototype declaration.

```
void sort (int, int *);
```

This tells the compiler that the formal argument that receives the array is a pointer, not array variable.

Pointer parameters are commonly employed in string functions. Consider the function **copy** which copies one string to another.

```
copy(char *s1, char *s2)
{
    while( (*s1++ = *s2++) != '\0')
        ;
}
```

This copies the contents of **s2** into the string **s1**. Parameters **s1** and **s2** are the pointers to character strings, whose initial values are passed from the calling function. For example, the calling statement

```
copy(name1, name2);
```

will assign the address of the first element of **name1** to **s1** and the address of the first element of **name2** to **s2**.

Note that the value of ***s2++** is the character that **s2** pointed to before **s2** was incremented. Due to the postfix ++, **s2** is incremented only after the current value has been fetched. Similarly, **s1** is incremented only after the assignment has been completed.

Each character, after it has been copied, is compared with '\0' and therefore copying is terminated as soon as the '\0' is copied.

4A.14 FUNCTIONS RETURNING POINTERS

We have seen so far that a function can return a single value by its name or return multiple values through pointer parameters. Since pointers are a data type in C, we can also force a function to return a pointer to the calling function. Consider the following code:

```
int *larger (int *, int *);    /* prototype */
main ( )
{
    int a = 10;
    int b = 20;
    int *p;
    p = larger(&a, &b);      /*Function call */
    printf ("%d", *p);
}
int *larger (int *x, int *y)
{
    if (*x>*y)
        return (x);        /* address of a */
    else
        return (y);        /* address of b */
}
```

The function **larger** receives the addresses of the variables **a** and **b**, decides which one is larger using the pointers **x** and **y** and then returns the address of its location. The returned value is then assigned to the pointer variable **p** in the calling function. In this case, the address of **b** is returned and assigned to **p** and therefore the output will be the value of **b**, namely, 20.

Note that the address returned must be the address of a variable in the calling function. It is an error to return a pointer to a local variable in the called function.

4A.15 POINTERS TO FUNCTIONS

A function, like a variable, has a type and an address location in the memory. It is therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

```
type (*fptr) ();
```

This tells the compiler that **fptr** is a pointer to a function, which returns *type* value. The parentheses around ***fptr** are necessary. Remember that a statement like

```
type *gptr();
```

would declare **gptr** as a function returning a pointer to *type*.

4A.20 Computer Programming

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer. For example, the statements

```
double mul(int, int);
double (*p1)();
p1 = mul;
```

declare **p1** as a pointer to a function and **mul** as a function and then make **p1** to point to the function **mul**. To call the function **mul**, we may now use the pointer **p1** with the list of parameters. That is,

```
(*p1)(x,y) /* Function call */
```

is equivalent to

```
mul(x,y)
```

Note the parentheses around ***p1**.

Example 4A.7 Write a program that uses a function pointer as a function argument.

A program to print the function values over a given range of values is shown in Fig. 4A.12. The printing is done by the function **table** by evaluating the function passed to it by the **main**.

With **table**, we declare the parameter **f** as a pointer to a function as follows:

```
double (*f)();
```

The value returned by the function is of type **double**. When **table** is called in the statement

```
table (y, 0.0, 2, 0.5);
```

we pass a pointer to the function **y** as the first parameter of **table**. Note that **y** is not followed by a parameter list.

During the execution of **table**, the statement

```
value = (*f)(a);
```

calls the function **y** which is pointed to by **f**, passing it the parameter **a**. Thus the function **y** is evaluated over the range 0.0 to 2.0 at the intervals of 0.5.

Similarly, the call

```
table (cos, 0.0, PI, 0.5);
```

passes a pointer to **cos** as its first parameter and therefore, the function **table** evaluates the value of **cos** over the range 0.0 to PI at the intervals of 0.5.

Program

```
#include <math.h>
#define PI 3.1415926
double y(double);
double cos(double);
double table (double(*f)(), double, double, double);

main()
{   printf("Table of y(x) = 2*x*x-x+1\n\n");
    table(y, 0.0, 2.0, 0.5);
    printf("\nTable of cos(x)\n\n");
    table(cos, 0.0, PI, 0.5);
}

double table(double(*f)(),double min, double max, double step)
{   double a, value;
```

```

    for(a = min; a <= max; a += step)
    {
        value = (*f)(a);
        printf("%5.2f  %10.4f\n", a, value);
    }
}

double y(double x)
{
    return(2*x*x-x+1);
}

```

Output

Table of $y(x) = 2x^2 - x + 1$

0.00	1.0000
0.50	1.0000
1.00	2.0000
1.50	4.0000
2.00	7.0000

Table of $\cos(x)$

0.00	1.0000
0.50	0.8776
1.00	0.5403
1.50	0.0707
2.00	-0.4161
2.50	-0.8011
3.00	-0.9900

Fig. 4A.12 Use of pointers to functions**Compatibility and Casting**

A variable declared as a pointer is not just a *pointer type* variable. It is also a pointer to a *specific* fundamental data type, such as a character. A pointer therefore always has a type associated with it. We cannot assign a pointer of one type to a pointer of another type, although both of them have memory addresses as their values. This is known as *incompatibility* of pointers.

All the pointer variables store memory addresses, which are compatible, but what is not compatible is the underlying data type to which they point to. We cannot use the assignment operator with the pointers of different types. We can however make explicit assignment between incompatible pointer types by using **cast** operator, as we do with the fundamental types. Example:

```

int x;
char *p;
p = (char *) & x;

```

In such cases, we must ensure that all operations that use the pointer **p** must apply casting properly.

We have an exception. The exception is the void pointer (void *). The void pointer is a **generic pointer** that can represent any pointer type. All pointer types can be assigned to a void pointer and a void pointer can be assigned to any pointer without casting. A void pointer is created as follows:

```
void *vp;
```

Remember that since a void pointer has no object type, it cannot be de-referenced.

4A.22 Computer Programming

The **void pointer**, also known as the generic pointer, is a special type of pointer that can be pointed at objects of any data type. A void pointer is declared like a normal pointer, using the void keyword as the pointer's type. However, because the void pointer does not know what type of object it is pointing to, it can not be dereferenced. Rather, the void pointer must first be explicitly cast to another pointer type before it is dereferenced.

```
#include <stdio.h>
void main()
{
    int a=10;
    float b=35.75;
    void *ptr; // Declaring a void pointer
    ptr=&a; // Assigning address of integer to void pointer.
    printf("The value of integer variable is= %d",*( (int*) ptr) );// (int*)ptr - is
    used for type casting. Where as *((int*)ptr) dereferences the typecasted void
    pointer variable.
    ptr=&b; // Assigning address of float to void pointer.
    printf("The value of float variable is= %f",*( (float*) ptr) );
}
```

it is not possible to do pointer arithmetic on a void pointer

```
void *ptr;
int a;
ptr=&a;
ptr++; // This statement is invalid and will result in an error because 'ptr' is a void pointer variable.
```

4A.16 INTRODUCTION

Most often we face situations in programming where the data is dynamic in nature. That is, the number of data items keep changing during execution of the program. For example, consider a program for processing the list of customers of a corporation. The list grows when names are added and shrinks when names are deleted. When list grows we need to allocate more memory space to the list to accommodate additional data items. Such situations can be handled more easily and effectively by using what is known as *dynamic data structures* in conjunction with *dynamic memory management* techniques.

Dynamic data structures provide flexibility in adding, deleting or rearranging data items at run time. Dynamic memory management techniques permit us to allocate additional memory space or to release unwanted space at run time, thus, optimizing the use of storage space. This chapter discusses the concept of *linked lists*, one of the basic types of dynamic data structures. Before we take up linked lists, we shall briefly introduce the dynamic storage management functions that are available in C. These functions would be extensively used in processing linked lists.

4A.17 DYNAMIC MEMORY ALLOCATION

C language requires the number of elements in an array to be specified at compile time. But we may not be able to do so always. Our initial judgement of size, if it is wrong, may cause failure of the program or wastage of memory space.

Many languages permit a programmer to specify an array's size at run time. Such languages have the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory at run time is known as *dynamic memory allocation*. Although C does not inherently have this facility, there are four library routines known as “memory management functions” that can be used for allocating and freeing memory during program execution. They are listed in Table 4A.1. These functions help us build complex application programs that use the available memory intelligently.

Memory Allocation Process

Before we discuss these functions, let us look at the memory allocation process associated with a C program. Figure 4A.13 shows the conceptual view of storage of a C program in memory.

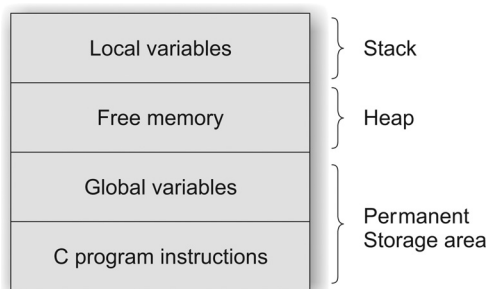


Fig. 4A.13 Storage of a C program

TABLE 4A.1 Memory Allocation Functions

Function	Task
malloc	Allocates request size of bytes and returns a pointer to the first byte of the allocated space.
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
free	Frees previously allocated space.
realloc	Modifies the size of previously allocated space.

The program instructions and global and static variables are stored in a region known as *permanent storage area* and the local variables are stored in another area called *stack*. The memory space that is located between these two regions is available for dynamic allocation during execution of the program. This free memory region is called the *heap*. The size of the heap keeps changing when program is executed due to creation and death of variables that are local to functions and blocks. Therefore, it is possible to encounter memory “overflow” during dynamic allocation process. In such situations, the memory allocation functions mentioned above return a NULL pointer (when they fail to locate enough memory requested).

4A.18 ALLOCATING A BLOCK OF MEMORY: MALLOC

A block of memory may be allocated using the function **malloc**. The **malloc** function reserves a block of memory of specified size and returns a pointer of type **void**. This means that we can assign it to any type of pointer. It takes the following form:

```
ptr = (cast-type *) malloc(byte-size);
```

ptr is a pointer of type *cast-type*. The **malloc** returns a pointer (of *cast-type*) to an area of memory with size *byte-size*.

4A.24 Computer Programming

Example:

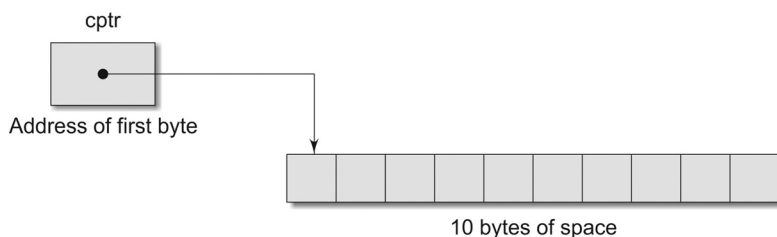
```
x = (int *) malloc (100 *sizeof(int));
```

On successful execution of this statement, a memory space equivalent to “100 times the size of an **int**” bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer **x** of type of **int**.

Similarly, the statement

```
cptr = (char*) malloc(10);
```

allocates 10 bytes of space for the pointer **cptr** of type **char**. This is illustrated as:



Note that the storage space allocated dynamically has no name and therefore its contents can be accessed only through a pointer.

We may also use **malloc** to allocate space for complex data types such as structures.

Example:

```
st_var = (struct store *)malloc(sizeof(struct store));
```

where, **st_var** is a pointer of type **struct store**

Remember, the **malloc** allocates a block of contiguous bytes. The allocation can fail if the space in the heap is not sufficient to satisfy the request. If it fails, it returns a **NULL**. We should therefore check whether the allocation is successful before using the memory pointer. This is illustrated in the program in Fig. 4A.14.

Example 4A.8 Write a program that uses a table of integers whose size will be specified interactively at run time.

The program is given in Fig. 4A.14. It tests for availability of memory space of required size. If it is available, then the required space is allocated and the address of the first byte of the space allocated is displayed. The program also illustrates the use of pointer variable for storing and accessing the table values.

```
Program
#include <stdio.h>
#include <stdlib.h>
#define NULL 0

main()
{
    int *p, *table;
    int size;
    printf("\nWhat is the size of table?");
    scanf("%d",&size);
    printf("\n")
    /*-----Memory allocation ----- */
    if((table = (int*)malloc(size *sizeof(int))) == NULL)
    {
        printf("No space available \n");
    }
}
```



```

        exit(1);
    }
    printf("\n Address of the first byte is %u\n", table);
    /* Reading table values*/
    printf("\nInput table values\n");

    for (p=table; p<table + size; p++)
        scanf("%d",p);
    /* Printing table values in reverse order*/
    for (p = table + size -1; p >= table; p --)
        printf("%d is stored at address %u \n",*p,p);
}

```

Output

```

What is the size of the table? 5
Address of the first byte is 2262
Input table values
11 12 13 14 15 15
15 is stored at address 2270
14 is stored at address 2268
13 is stored at address 2266
12 is stored at address 2264
11 is stored at address 2262

```

Fig. 4A.14 Memory allocation with **malloc**

4A.19 ALLOCATING MULTIPLE BLOCKS OF MEMORY: CALLOC

calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. While **malloc** allocates a single block of storage space, **calloc** allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general form of **calloc** is:

```
ptr = (cast-type *) calloc (n, elem-size);
```

The above statement allocates contiguous space for n blocks, each of size *elem-size* bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

The following segment of a program allocates space for a structure variable:

```

. . . .
. . . .
struct student
{
    char name[25];
    float age;
    long int id_num;
};
typedef struct student record;
record *st ptr;
int class_size = 30;

st_ptr=(record *)calloc(class_size, sizeof(record));

```

```

: : :
: : :

```

record is of type **struct student** having three members: **name**, **age** and **id_num**. The **calloc** allocates memory to hold data for 30 such records. We must be sure that the requested memory has been allocated successfully before using the **st_ptr**. This may be done as follows:

```

if(st_ptr == NULL)
{
    printf("Available memory not sufficient");
    exit(1);
}

```

4A.20 RELEASING THE USED SPACE: FREE

Compile-time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic run-time allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited.

When we no longer need the data we stored in a block of memory, and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the **free** function:

```
free (ptr);
```

ptr is a pointer to a memory block, which has already been created by **malloc** or **calloc**. Use of an invalid pointer in the call may create problems and cause system crash. We should remember two things here:

1. It is not the pointer that is being released but rather what it points to.
2. To release an array of memory that was allocated by **calloc** we need only to release the pointer once. It is an error to attempt to release elements individually.



Just Remember

- Only an address of a variable can be stored in a pointer variable.
- Do not store the address of a variable of one type into a pointer variable of another type.
- The value of a variable cannot be assigned to a pointer variable.
- A pointer variable contains garbage until it is initialized. Therefore we must not use a pointer variable before it is assigned, the address of a variable.
- Remember that the definition for a pointer variable allocates memory only for the pointer variable, not for the variable to which it is pointing.
- If we want a called function to change the value of a variable in the calling function, we must pass the address of that variable to the called function.
- When we pass a parameter by address, the corresponding formal parameter must be a pointer variable.
- It is an error to assign a numeric constant to a pointer variable.
- It is an error to assign the address of a variable to a variable of any basic data types.
- It is an error to assign a pointer of one type to a pointer of another type without a cast (with an exception of void pointer).
- A proper understanding of a precedence and associativity rules is very important in pointer applications. For example, expressions like

*p++, *p[], (*p)[], (p).member should be carefully used.

- When an array is passed as an argument to a function, a pointer is actually passed. In the header function, we must declare such arrays with proper size, except the first, which is optional.
- A very common error is to use (or not to use) the address operator (&) and the indirection operator (*) in certain places. Be careful. The compiler may not warn such mistakes.
- Pointers should be used very carefully as they involve direct memory access.
- A pointer-to-a-pointer reference should be used carefully as it involves usage of two indirection operators which could be confusing.
- A pointer variable can be assigned value of another pointer variable.
- A pointer variable can be initialized with a NULL value.
- You can define a pointer to a function that can be used as an argument in another function.
- Use the **sizeof** operator to determine the size of a linked list.
- When using memory allocation functions **malloc** and **calloc**, test for a NULL pointer return value. Print appropriate message if the memory allocation fails.
- Never call memory allocation functions with a zero size.
- Release the dynamically allocated memory when it is no longer required to avoid any possible “memory leak”.
- Using a pointer after its memory has been released is an error.
- It is an error to assign the return value from **malloc** or **calloc** to anything other than a pointer.
- It is a logic error to set a pointer to NULL before the node has been released. The node is irretrievably lost.
- It is an error to declare a self-referential structure without a structure tag.
- It is an error to release individually the elements of an array created with **calloc**.
- It is a logic error to fail to set the link field in the last node to null.



Multiple Choice Questions

1. Pointer is an example of which of the following type?
 - (a) Derived type
 - (b) Fundamental type
 - (c) User-defined type
 - (d) None of the above
2. An integer pointer:
 - (a) Points to an another integer value
 - (b) Points to the address of another integer value.
 - (c) Points to itself
 - (d) None of the above
3. In the expression *ptr=&a, what does & signify
 - (a) Address of a
 - (b) Address of ptr
 - (c) Value of a
 - (d) None of the above
4. Which of the following expressions in C is used for accessing the address of a variable var?
 - (a) &var
 - (b) *var
 - (c) &(*var)
 - (d) *(&var)
5. Which of the following is a syntactically correct pointer declaration?
 - (a) float *x;
 - (b) float* x;
 - (c) float * x;
 - (d) All of the above are correct
6. Which of the following expressions will give the value stored in variable x?
 - (a) x
 - (b) *x
 - (c) *&x
 - (d) &x
7. If a1=&x and a2 =&a1, what will be the output generated by the expression **a2?

4A.28 Computer Programming

- (a) Address of a2
(b) Address of a1
(c) Value of x
(d) Address of x
8. If $a1 = 2$ and $a2 = \&a1$ then what does $a2++$ depict (consider the address value of a1 to be 3802)
(a) 3 (b) 3803
(c) 3804 (d) 3802
9. What will be the expression for obtaining the address of the i th element of an array A?
(a) $A[i]$ (b) $\&A[i]$
(c) $*A[i]$ (d) $A[\&i]$
10. What is the size of an integer & float pointer?
(a) 2 & 4 (b) 4 & 4
(c) 2 & 2 (d) 1 & 1
11. Linked list uses type of memory allocation
(a) static (b) random
(c) dynamic (d) compile time
12. The number of extra pointers required to reverse a singly linked list is
(a) 1 (b) 2
(c) 3 (d) 4
13. The number of extra pointers required to reverse a double linked list is
(a) 1 (b) 2
(c) 3 (d) 4
14. The functions used for memory allocation are
(a) malloc (b) calloc
(c) a and b (d) none of the above
15. Linked lists use _____ type of structures.
(a) nested (b) self-referential
(c) simple (d) unions
16. _____ cannot be used to represent linked lists.
(a) arrays (b) structures
(c) unions (d) all the above
17. $\text{calloc}(m,n)$ is equivalent to
(a) $\text{malloc}(m*n,0)$
(b) $\text{memset}(0,m*n)$
(c) $\text{ptr}=\text{malloc}(m*n)$
(d) $\text{malloc}(m/n)$



Case Study

1. Processing of Examination Marks

Marks obtained by a batch of students in the Annual Examination are tabulated as follows:

Student name	Marks obtained
S. Laxmi	45 67 38 55
V.S. Rao	77 89 56 69
-	- - - -

It is required to compute the total marks obtained by each student and print the rank list based on the total marks.

The program in Fig. 4A.15 stores the student names in the array **name** and the marks in the array **marks**. After computing the total marks obtained by all the students, the program prepares and prints the rank list. The declaration

```
int marks[STUDENTS][SUBJECTS+1];
```

defines **marks** as a pointer to the array's first row. We use **rowptr** as the pointer to the row of **marks**. The **rowptr** is initialized as follows:

```
int (*rowptr)[SUBJECTS+1] = array;
```

Note that **array** is the formal argument whose values are replaced by the values of the actual argument **marks**. The parentheses around ***rowptr** makes the **rowptr** as a pointer to an array of **SUBJECTS+1** integers. Remember, the statement

```
int *rowptr[SUBJECTS+1];
```

would declare **rowptr** as an array of **SUBJECTS+1** elements.

When we increment the **rowptr** (by **rowptr+1**), the incrementing is done in units of the size of each row of **array**, making **rowptr** point to the next row. Since **rowptr** points to a particular row, **(*rowptr)[x]** points to the xth element in the row.

Program

```
#define STUDENTS 5
#define SUBJECTS 4
#include <string.h>

main()
{
    char name[STUDENTS][20];
    int marks[STUDENTS][SUBJECTS+1];

    printf("Input students names & their marks in four subjects\n");
    get_list(name, marks, STUDENTS, SUBJECTS);
    get_sum(marks, STUDENTS, SUBJECTS+1);
    printf("\n");
    print_list(name, marks, STUDENTS, SUBJECTS+1);
    get_rank_list(name, marks, STUDENTS, SUBJECTS+1);
    printf("\nRanked List\n\n");
    print_list(name, marks, STUDENTS, SUBJECTS+1);
}

/* Input student name and marks */
get_list(char *string [ ],
          int array [ ] [SUBJECTS +1], int m, int n)
{
    int i, j, (*rowptr)[SUBJECTS+1] = array;
    for(i = 0; i < m; i++)
    {
        scanf("%s", string[i]);
        for(j = 0; j < SUBJECTS; j++)
            scanf("%d", &(*(rowptr + i))[j]);
    }
}

/* Compute total marks obtained by each student */
get_sum(int array [ ] [SUBJECTS +1], int m, int n)
{
    int i, j, (*rowptr)[SUBJECTS+1] = array;
    for(i = 0; i < m; i++)
    {
        (*(rowptr + i))[n-1] = 0;
        for(j = 0; j < n-1; j++)
            (*(rowptr + i))[n-1] += (*(rowptr + i))[j];
    }
}

/* Prepare rank list based on total marks */
get_rank_list(char *string [ ],
```

4A.30 Computer Programming

```
                int array [ ] [SUBJECTS + 1]
                int m,
                int n)
{
    int i, j, k, (*rowptr)[SUBJECTS+1] = array;
    char *temp;

    for(i = 1; i <= m-1; i++)
        for(j = 1; j <= m-i; j++)
            if( (*(rowptr + j-1))[n-1] < (*(rowptr + j))[n-1])
            {
                swap_string(string[j-1], string[j]);

                for(k = 0; k < n; k++)
                    swap_int(&(*(rowptr + j-1))[k], &(*(rowptr+j))[k]);
            }
}
/*      Print out the ranked list      */
print_list(char *string[ ],
            int array [ ] [SUBJECTS + 1],
            int m,
            int n)
{
    int i, j, (*rowptr)[SUBJECTS+1] = array;
    for(i = 0; i < m; i++)
    {
        printf("%-20s", string[i]);
        for(j = 0; j < n; j++)
            printf("%5d", (*(rowptr + i))[j]);
        printf("\n");
    }
}
/*      Exchange of integer values      */
swap_int(int *p, int *q)
{
    int temp;
    temp = *p;
    *p = *q;
    *q = temp;
}

/*      Exchange of strings      */
swap_string(char s1[ ], char s2[ ])
{
    char swaparea[256];
    int i;
    for(i = 0; i < 256; i++)
        swaparea[i] = '\0';
    i = 0;
    while(s1[i] != '\0' && i < 256)
    {
```

```

        swaparea[i] = s1[i];
        i++;
    }
    i = 0;
    while(s2[i] != '\0' && i < 256)
    {
        s1[i] = s2[i];
        s1[++i] = '\0';
    }
    i = 0;
    while(swaparea[i] != '\0')
    {
        s2[i] = swaparea[i];
        s2[++i] = '\0';
    }
}

```

Output

Input students names & their marks in four subjects

S.Laxmi	45	67	38	55
V.S.Rao	77	89	56	69
A.Gupta	66	78	98	45
S.Mani	86	72	0	25
R.Daniel	44	55	66	77

S.Laxmi	45	67	38	55	205
V.S.Rao	77	89	56	69	291
A.Gupta	66	78	98	45	287
S.Mani	86	72	0	25	183
R.Daniel	44	55	66	77	242

Ranked List

V.S.Rao	77	89	56	69	291
A.Gupta	66	78	98	45	287
R.Daniel	44	55	66	77	242
S.Laxmi	45	67	38	55	205
S.Mani	86	72	0	25	183

Fig. 4A.15 Preparation of the rank list of a class of students

2. Inventory Updating

The price and quantity of items stocked in a store changes every day. They may either increase or decrease. The program in Fig. 4A.16 reads the incremental values of price and quantity and computes the total value of the items in stock.

The program illustrates the use of structure pointers as function parameters. **&item**, the address of the structure **item**, is passed to the functions **update()** and **mul()**. The formal arguments **product** and **stock**, which receive the value of **&item**, are declared as pointers of type **struct stores**.

4A.32 Computer Programming

Program

```
struct stores
{
    char  name[20];
    float price;
    int   quantity;
};

main()
{
    void update(struct stores *, float, int);
    float      p_increment, value;
    int        q_increment;

    struct stores item = {"XYZ", 25.75, 12};
    struct stores *ptr = &item;

    printf("\nInput increment values:");
    printf(" price increment and quantity increment\n");
    scanf("%f %d", &p_increment, &q_increment);

    /* - - - - - */
    update(&item, p_increment, q_increment);
    /* - - - - - */
    printf("Updated values of item\n\n");
    printf("Name      : %s\n", ptr->name);
    printf("Price       : %f\n", ptr->price);
    printf("Quantity    : %d\n", ptr->quantity);

    /* - - - - - */
    value = mul(&item);
    /* - - - - - */
    printf("\nValue of the item = %f\n", value);
}

void update(struct stores *product, float p, int q)
{
    product->price += p;
    product->quantity += q;
}

float mul(struct stores *stock)
{
    return(stock->price * stock->quantity);
}
```

Output

```
Input increment values: price increment and quantity increment
10 12
Updated values of item
```



```

Name      : XYZ
Price     : 35.750000
Quantity  : 24

Value of the item = 858.000000

```

Fig. 4A.16 Use of structure pointers as function parameters



Review Questions

- 4A.1 State whether the following statements are *true* or *false*.
- Pointer constants are the addresses of memory locations.
 - Pointer variables are declared using the address operator.
 - The underlying type of a pointer variable is void.
 - Pointers to pointers is a term used to describe pointers whose contents are the address of another pointer.
 - It is possible to cast a pointer to float as a pointer to integer.
 - An integer can be added to a pointer.
 - A pointer can never be subtracted from another pointer.
 - When an array is passed as an argument to a function, a pointer is passed.
 - Pointers cannot be used as formal parameters in headers to function definitions.
 - Value of a local variable in a function can be changed by another function.
- 4A.2 Fill in the blanks in the following statements:
- A pointer variable contains as its value the _____ of another variable.
 - The _____ operator is used with a pointer to de-reference the address contained in the pointer.
 - The _____ operator returns the value of the variable to which its operand points.
 - The only integer that can be assigned to a pointer variable is _____.
 - The pointer that is declared as _____ cannot be de-referenced.
- 4A.3 What is a pointer?
- 4A.4 How is a pointer initialized?
- 4A.5 Explain the effects of the following statements:
- `int a, *b = &a;`
 - `int p, *p;`
 - `char *s;`
 - `a = (float *) &x;`
 - `double(*f)();`
- 4A.6 If **m** and **n** have been declared as integers and **p1** and **p2** as pointers to integers, then state errors, if any, in the following statements.
- `p1 = &m;`
 - `p2 = n;`
 - `*p1 = &n;`
 - `p2 = &* &m;`
 - `m = p2 - p1;`
 - `p1 = &p2;`
 - `m = *p1 + *p2++;`
- 4A.7 Distinguish between `(*m)[5]` and `*m[5]`.
- 4A.8 Find the error, if any, in each of the following statements:
- `int x = 10;`
 - `int *y = 10;`
 - `int a, *b = &a;`
 - `int m;`
`int **x = &m;`
- 4A.9 Given the following declarations:
- ```

int x = 10, y = 10;
int *p1 = &x, *p2 = &y;

```

#### 4A.34 Computer Programming

What is the value of each of the following expressions?

- (a)  $(*p1)++$
- (b)  $—(*p2)$
- (c)  $*p1 + (*p2) —$
- (d)  $++(*p2) - *p1$

4A.10 Describe typical applications of pointers in developing programs.

4A.11 What are the arithmetic operators that are permitted on pointers?

4A.12 What is printed by the following program?

```
int m = 100';
int * p1 = &m;
int **p2 = &p1;
printf(“%d”, **p2);
```

4A.13 What is wrong with the following code?

```
int **p1, *p2;
p2 = &p1;
```

4A.14 Assuming **name** as an array of 15 character length, what is the difference between the following two expressions?

- (a)  $name + 10$ ; and
- (b)  $*(name + 10)$ .

4A.15 What is the output of the following segment?

```
int m[2];
*(m+1) = 100;
*m = *(m+1);
printf(“%d”, m [0]);
```

4A.16 What is the output of the following code?

```
int m [2];
int *p = m;
m [0] = 100 ;
```

```
m [1] = 200 ;
printf(“%d %d”, ++*p, *p);
```

4A.17 What is the output of the following program?

```
int f(char *p);
main ()
{
 char str[] = “ANSI”;
 printf(“%d”, f(str));
}
int f(char *p)
{
 char *q = p;
 while (*++p)
 ;
 return (p-q);
}
```

4A.18 Given below are two different definitions of the function **search( )**

- (a) `void search (int* m[ ], int x)`  
{  
}  
(b) `void search (int ** m, int x)`  
{  
}

Are they equivalent? Explain.

4A.19 Do the declarations

```
char s [5] ;
char *s;
represent the same? Explain.
```

4A.20 Which one of the following is the correct way of declaring a pointer to a function? Why?

- (a) `int ( *p) (void) ;`
- (b) `int *p (void);`



### Programming Exercises

4A.1 Write a program using pointers to read in an array of integers and print its elements in reverse order.

4A.2 We know that the roots of a quadratic equation of the form

$$ax^2 + bx + c = 0$$

are given by the following equations:

$$x_1 = \frac{-b + \text{square-root}(b^2 - 4ac)}{2a}$$

$$x_2 = \frac{-b - \text{square-root}(b^2 - 4ac)}{2a}$$

Write a function to calculate the roots. The function must use two pointer parameters, one to receive the coefficients a, b, and c, and the other to send the roots to the calling function.

- 4A.3 Write a function that receives a sorted array of integers and an integer value, and inserts the value in its correct place.
- 4A.4 Write a function using pointers to add two matrices and to return the resultant matrix to the calling function.
- 4A.5 Using pointers, write a function that receives a character string and a character as argument and deletes all occurrences of this character in the string. The function should return the corrected string with no holes.
- 4A.6 Write a function **day\_name** that receives a number n and returns a pointer to a character string containing the name of the corresponding day. The day names should be kept in a **static** table of character strings local to the function.
- 4A.7 Write a program to read in an array of names and to sort them in alphabetical order.

Use **sort** function that receives pointers to the functions **strcmp** and **swap.sort** in turn should call these functions via the pointers.

- 4A.8 Given an array of sorted list of integer numbers, write a function to search for a particular item, using the method of *binary search*. And also show how this function may be used in a program. Use pointers and pointer arithmetic.

**(Hint:** In binary search, the target value is compared with the array's middle element. Since the table is sorted, if the required value is smaller, we know that all values greater than the middle element can be ignored. That is, in one attempt, we eliminate one half the list. This search can be applied recursively till the target value is found.)

- 4A.9 Write a function (using a pointer parameter) that reverses the elements of a given array.
- 4A.10 Write a function (using pointer parameters) that compares two integer arrays to see whether they are identical. The function returns 1 if they are identical, 0 otherwise.



## Key Terms

- **Pointer:** It is a derived data type that contains memory address as its value.
- **Memory:** It is a sequential collection of storage cells.
- **Pointer variables:** These variables hold memory addresses and are stored in the memory.
- **Call by reference:** It is the process of calling a function using pointers to pass the address of the variables.
- **Call by value:** It is the process of passing the actual value of variables.



# UNIT

---

# 4B

# Strings

---

## 4B.1 INTRODUCTION

A string is a sequence of characters that is treated as a single data item. We have used strings in a number of examples in the past. Any group of characters (except double quote sign) defined between double quotation marks is a string constant. Example:

“Man is obviously made to think.”

If we want to include a double quote in the string to be printed, then we may use it with a back slash as shown below.

“\” Man is obviously made to think,\” said Pascal.”

For example,

```
printf (“\” Well Done !”\”);
```

will output the string

“ Well Done !”

while the statement

```
printf(“ Well Done !”);
```

will output the string

Well Done !

Character strings are often used to build meaningful and readable programs. The common operations performed on character strings include:

- Reading and writing strings.
- Combining strings together.
- Copying one string to another.
- Comparing strings for equality.
- Extracting a portion of a string.

In this chapter, we shall discuss these operations in detail and examine library functions that implement them.

## 4B.2 DECLARING AND INITIALIZING STRING VARIABLES

C does not support strings as a data type. However, it allows us to represent strings as character arrays. In C, therefore, a string variable is any valid C variable name and is always declared as an array of characters. The general form of declaration of a string variable is:

```
char string_name[size];
```

The *size* determines the number of characters in the *string\_name*. Some examples are:

```
char city[10];
char name[30];
```

When the compiler assigns a character string to a character array, it automatically supplies a *null* character (`'\0'`) at the end of the string. Therefore, the *size* should be equal to the maximum number of characters in the string *plus* one.

Like numeric arrays, character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms:

```
char city [9] = " NEW YORK ";
char city [9]={ 'N', 'E', 'W', ' ', 'Y', 'O', 'R', 'K', '\0' };
```

The reason that **city** had to be 9 elements long is that the string NEW YORK contains 8 characters and one element space is provided for the null terminator. Note that when we initialize a character array by listing its elements, we must supply explicitly the null terminator.

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized. For example, the statement

```
char string [] = { 'G', 'O', 'O', 'D', '\0' };
```

defines the array **string** as a five element array.

We can also declare the size much larger than the string size in the initializer. That is, the statement.

```
char str[10] = "GOOD";
```

is permitted. In this case, the computer creates a character array of size 10, places the value "GOOD" in it, terminates with the null character, and initializes all other elements to NULL. The storage will look like:

|   |   |   |   |    |    |    |    |    |    |
|---|---|---|---|----|----|----|----|----|----|
| G | O | O | D | \0 | \0 | \0 | \0 | \0 | \0 |
|---|---|---|---|----|----|----|----|----|----|

However, the following declaration is illegal.

```
char str2[3] = "GOOD";
```

This will result in a compile time error. Also note that we cannot separate the initialization from declaration. That is,

```
char str3[5];
str3 = "GOOD";
```

is not allowed. Similarly,

```
char s1[4] = "abc";
char s2[4];
s2 = s1; /* Error */
```

is not allowed. An array name cannot be used as the left operand of an assignment operator.

### Terminating Null Character

You must be wondering, “why do we need a terminating null character?” As we know, a string is not a data type in C, but it is considered a data structure stored in an array. The string is a variable-length structure and is stored in a fixed-length array. The array size is not always the size of the string and most often it is much larger than the string stored in it. Therefore, the last element of the array need not represent the end of the string. We need some way to determine the end of the string data and the null character serves as the “end-of-string” marker.

## 4B.3 READING STRINGS FROM TERMINAL

### 4B.3.1 Using scanf Function

The familiar input function **scanf** can be used with **%s** format specification to read in a string of characters. Example:

```
char address[10]
scanf("%s", address);
```

The problem with the **scanf** function is that it terminates its input on the first white space it finds. A white space includes blanks, tabs, carriage returns, form feeds, and new lines. Therefore, if the following line of text is typed in at the terminal,

NEW YORK

then only the string “NEW” will be read into the array **address**, since the blank space after the word ‘NEW’ will terminate the reading of string.

The **scanf** function automatically terminates the string that is read with a null character and therefore the character array should be large enough to hold the input string plus the null character. Note that unlike previous **scanf** calls, in the case of character arrays, the ampersand (&) is not required before the variable name.

The **address** array is created in the memory as shown below:

|   |   |   |    |   |   |   |   |   |   |
|---|---|---|----|---|---|---|---|---|---|
| N | E | W | \0 | ? | ? | ? | ? | ? | ? |
| 0 | 1 | 2 | 3  | 4 | 5 | 6 | 7 | 8 | 9 |

Note that the unused locations are filled with garbage.

If we want to read the entire line “NEW YORK”, then we may use two character arrays of appropriate sizes. That is,

```
char adr1[5], adr2[5];
scanf("%s %s", adr1, adr2);
```

with the line of text

NEW YORK

will assign the string “NEW” to **adr1** and “YORK” to **adr2**.

### Example 4B.1 Write a program to read a series of words from a terminal using scanf function.

The program shown in Fig. 4B.1 reads four words and displays them on the screen. Note that the string ‘Oxford Road’ is treated as *two words* while the string ‘Oxford-Road’ as *one word*.

## 4B.4 Computer Programming

### Program

```
main()
{
 char word1[40], word2[40], word3[40], word4[40];
 printf("Enter text : \n");
 scanf("%s %s", word1, word2);
 scanf("%s", word3);
 scanf("%s", word4);

 printf("\n");
 printf("word1 = %s\nword2 = %s\n", word1, word2);
 printf("word3 = %s\nword4 = %s\n", word3, word4);
}
```

### Output

```
Enter text :
Oxford Road, London M17ED

word1 = Oxford
word2 = Road,
word3 = London
word4 = M17ED

Enter text :
Oxford-Road, London-M17ED United Kingdom
word1 = Oxford-Road
word2 = London-M17ED
word3 = United
word4 = Kingdom
```

**Fig. 4B.1** Reading a series of words using **scanf** function

We can also specify the field width using the form **%ws** in the **scanf** statement for reading a specified number of characters from the input string . Example:

```
scanf("%ws", name);
```

Here, two things may happen.

1. The width **w** is equal to or greater than the number of characters typed in. The entire string will be stored in the string variable.
2. The width **w** is less than the number of characters in the string. The excess characters will be truncated and left unread.

Consider the following statements:

```
char name[10];
scanf("%5s", name);
```

The input string RAM will be stored as:

|   |   |   |    |   |   |   |   |   |   |
|---|---|---|----|---|---|---|---|---|---|
| R | A | M | \0 | ? | ? | ? | ? | ? | ? |
| 0 | 1 | 2 | 3  | 4 | 5 | 6 | 7 | 8 | 9 |

The input string KRISHNA will be stored as:

|   |   |   |   |   |    |   |   |   |   |
|---|---|---|---|---|----|---|---|---|---|
| K | R | I | S | H | \0 | ? | ? | ? | ? |
| 0 | 1 | 2 | 3 | 4 | 5  | 6 | 7 | 8 | 9 |



### 4B.3.2 Reading a Line of Text

We have seen just now that **scanf** with **%s** or **%ws** can read only strings without whitespaces. That is, they cannot be used for reading a text containing more than one word. However, C supports a format specification known as the *edit set conversion code* **%[. .]** that can be used to read a line containing a variety of characters, including whitespaces. Recall that we have used this conversion code in Chapter 5. For example, the program segment

```
char line [80];
scanf("%[^\n]", line);
printf("%s", line);
```

will read a line of input from the keyboard and display the same on the screen. We would very rarely use this method, as C supports an intrinsic string function to do this job. This is discussed in the next section.

### 4B.3.3 Using *getchar* and *gets* Functions

We have discussed in Chapter 5 as to how to read a single character from the terminal, using the function **getchar**. We can use this function repeatedly to read successive single characters from the input and place them into a character array. Thus, an entire line of text can be read and stored in an array. The reading is terminated when the newline character (**'\n'**) is entered and the null character is then inserted at the end of the string. The **getchar** function call takes the form:

```
char ch;
ch = getchar();
```

Note that the **getchar** function has no parameters.

---

**Example 4B.2** Write a program to read a line of text containing a series of words from the terminal.

---

The program shown in Fig. 4B.2 can read a line of text (up to a maximum of 80 characters) into the string **line** using **getchar** function. Every time a character is read, it is assigned to its location in the string **line** and then tested for *newline* character. When the *newline* character is read (signalling the end of line), the reading loop is terminated and the *newline* character is replaced by the null character to indicate the end of character string.

When the loop is exited, the value of the index **c** is one number higher than the last character position in the string (since it has been incremented after assigning the new character to the string). Therefore the index value **c-1** gives the position where the *null* character is to be stored.

Program

```
#include <stdio.h>
main()
{
 char line[81], character;
 int c;
 c = 0;
 printf("Enter text. Press <Return> at end\n");
 do
 {
 character = getchar();
```

(Contd.)

```

 line[c] = character;
 c++;
 }
 while(character != '\n');
 c = c - 1;
 line[c] = '\0';
 printf("\n%s\n", line);
}

```

Output

```

Enter text. Press <Return> at end
Programming in C is interesting.
Programming in C is interesting.
Enter text. Press <Return> at end
National Centre for Expert Systems, Hyderabad.
National Centre for Expert Systems, Hyderabad.

```

**Fig. 4B.2** Program to read a line of text from terminal

Another and more convenient method of reading a string of text containing whitespaces is to use the library function **gets** available in the `<stdio.h>` header file. This is a simple function with one string parameter and called as under:

```
gets (str);
```

**str** is a string variable declared properly. It reads characters into **str** from the keyboard until a new-line character is encountered and then appends a null character to the string. Unlike **scanf**, it does not skip whitespaces. For example the code segment

```

char line [80];
gets (line);
printf ("%s", line);

```

reads a line of text from the keyboard and displays it on the screen. The last two statements may be combined as follows:

```
printf("%s", gets(line));
```

*(Be careful not to input more character that can be stored in the string variable used. Since C does not check array-bounds, it may cause problems.)*

C does not provide operators that work on strings directly. For instance we cannot assign one string to another directly. For example, the assignment statements.

```

string = "ABC";
string1 = string2;

```

are not valid. If we really want to copy the characters in **string2** into **string1**, we may do so on a character-by-character basis.

---

**Example 4B.3** Write a program to copy one string into another and count the number of characters copied.

---

The program is shown in Fig. 4B.3. We use a **for** loop to copy the characters contained inside **string2** into the **string1**. The loop is terminated when the *null* character is reached. Note that we are again assigning a null character to the **string1**.

## Program

```

main()
{
 char string1[80], string2[80];
 int i;
 printf("Enter a string \n");
 printf("?");
 scanf("%s", string2);
 for(i=0 ; string2[i] != '\0'; i++)
 string1[i] = string2[i];
 string1[i] = '\0';
 printf("\n");
 printf("%s\n", string1);
 printf("Number of characters = %d\n", i);
}

```

## Output

```

Enter a string
?Manchester
Manchester
Number of characters = 10
Enter a string
?Westminster
Westminster
Number of characters = 11

```

Fig. 4B.3 Copying one string into another

## 4B.4 WRITING STRINGS TO SCREEN

### 4B.4.1 Using printf Function

We have used extensively the **printf** function with %s format to print strings to the screen. The format %s can be used to display an array of characters that is terminated by the null character. For example, the statement

```
printf("%s", name);
```

can be used to display the entire contents of the array **name**.

We can also specify the precision with which the array is displayed. For instance, the specification

```
%10.4
```

indicates that the *first four* characters are to be printed in a field width of 10 columns.

However, if we include the minus sign in the specification (e.g., %-10.4s), the string will be printed left-justified. The Example 4B.4 illustrates the effect of various %s specifications.

---

**Example 4B.4** Write a program to store the string “United Kingdom” in the array country and display the string under various format specifications.

---

The program and its output are shown in Fig. 4B.4. The output illustrates the following features of the %s specifications.

## 4B.8 Computer Programming

1. When the field width is less than the length of the string, the entire string is printed.
2. The integer value on the right side of the decimal point specifies the number of characters to be printed.
3. When the number of characters to be printed is specified as zero, nothing is printed.
4. The minus sign in the specification causes the string to be printed left-justified.
5. The specification `%.ns` prints the first `n` characters of the string.

Program

```
main()
{
 char country[15] = "United Kingdom";
 printf("\n\n");
 printf("*123456789012345*\n");
 printf("---- \n");
 printf("%15s\n", country);
 printf("%5s\n", country);
 printf("%15.6s\n", country);
 printf("%-15.6s\n", country);
 printf("%15.0s\n", country);
 printf("%.3s\n", country);
 printf("%s\n", country);
 printf("---- \n");
}
```

Output

```
123456789012345

United Kingdom
United Kingdom
 United
United
Uni
United Kingdom

```

**Fig. 4B.4** Writing strings using `%s` format

The **printf** on UNIX supports another nice feature that allows for variable field width or precision. For instance

```
printf("%*.*s\n", w, d, string);
```

prints the first **d** characters of the string in the field width of **w**.

This feature comes in handy for printing a sequence of characters. Example 4B.5 illustrates this.

---

**Example 4B.5** Write a program using for loop to print the following output:

```
C
CP
CPr
CPro
....
....
```

```

CProgramming
CProgramming
.....
.....
CPro
CPr
CP
C

```

---

The outputs of the program in Fig. 4B.5, for variable specifications **%12.\*s**, **%.\*s**, and **%\*.1s** are shown in Fig. 4B.6, which further illustrates the variable field width and the precision specifications.

#### Program

```

main()
{
 int c, d;
 char string[] = "CProgramming";
 printf("\n\n");
 printf("-----\n");
 for(c = 0 ; c <= 11 ; c++)
 {
 d = c + 1;
 printf("|%-12.*s|\n", d, string);
 }
 printf("|_____| \n");
 for(c = 11 ; c >= 0 ; c--)
 {
 d = c + 1;
 printf("|%-12.*s|\n", d, string);
 }
 printf("-----\n");
}

```

#### Output

```

C
CP
CPr
CPro
CProg
CProgr
CProgra
CProgram
CProgramm
CProgrammi
CProgrammin
CProgramming
CProgramming
CProgrammin
CProgrammi
CProgramm

```

## 4B.10 Computer Programming

```
CProgram
CProgra
CProgr
CProg
CPro
CPr
CP
C
```

**Fig. 4B.5** Illustration of variable field specifications by printing sequences of characters

|              |              |           |
|--------------|--------------|-----------|
| C            | C            | C         |
| CP           | CP           | C         |
| CPr          | CPr          | C         |
| CPro         | CPro         | C         |
| CProg        | CProg        | C         |
| CProgr       | CProgr       | C         |
| CProgra      | CProgra      | C         |
| CProgram     | CProgram     | C         |
| CProgramm    | CProgramm    | C         |
| CProgrammi   | CProgrammi   | C         |
| CProgrammin  | CProgrammin  | C         |
| CProgramming | CProgramming | C         |
| <hr/>        |              |           |
| CProgramming | CProgramming | C         |
| CProgrammin  | CProgrammin  | C         |
| CProgrammi   | CProgrammi   | C         |
| CProgramm    | CProgramm    | C         |
| CProgram     | CProgram     | C         |
| CProgra      | CProgra      | C         |
| CProgr       | CProgr       | C         |
| CProg        | CProg        | C         |
| CPro         | CPro         | C         |
| CPr          | CPr          | C         |
| CP           | CP           | C         |
| C            | C            | C         |
| (a) %12.*s   | (b) %.*s     | (c) %*.1s |

**Fig. 4B.6** Further illustrations of variable specifications

### 4B.4.2 Using putchar and puts Functions

Like **getchar**, C supports another character handling function **putchar** to output the values of character variables. It takes the following form:

```
char ch = 'A';
putchar (ch);
```

The function **putchar** requires one parameter. This statement is equivalent to:

```
printf("%c", ch);
```

We have used **putchar** function in Chapter 5 to write characters to the screen. We can use this function repeatedly to output a string of characters stored in an array using a loop. Example:

```
char name[6] = "PARIS"
for (i=0, i<5; i++)
 putchar(name[i];
putchar('\n');
```

Another and more convenient way of printing string values is to use the function **puts** declared in the header file `<stdio.h>`. This is a one parameter function and invoked as under:

```
puts (str);
```

where **str** is a string variable containing a string value. This prints the value of the string variable **str** and then moves the cursor to the beginning of the next line on the screen. For example, the program segment

```
char line [80];
gets (line);
puts (line);
```

reads a line of text from the keyboard and displays it on the screen. Note that the syntax is very simple compared to using the **scanf** and **printf** statements.

## 4B.5 ARITHMETIC OPERATIONS ON CHARACTERS

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into an integer value by the system. The integer value depends on the local character set of the system.

To write a character in its integer representation, we may write it as an integer. For example, if the machine uses the ASCII representation, then,

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

will display the number 97 on the screen.

It is also possible to perform arithmetic operations on the character constants and variables. For example,

```
x = 'z'-1;
```

is a valid statement. In ASCII, the value of **'z'** is 122 and therefore, the statement will assign the value 121 to the variable **x**.

We may also use character constants in relational expressions. For example, the expression

```
ch >= 'A' && ch <= 'Z'
```

would test whether the character contained in the variable **ch** is an upper-case letter.

We can convert a character digit to its equivalent integer value using the following relationship:

```
x = character - '0';
```

where **x** is defined as an integer variable and **character** contains the character digit. For example, let us assume that the **character** contains the digit **'7'**,

Then,

```
x = ASCII value of '7' - ASCII value of '0'
 = 55 - 48
 = 7
```

The C library supports a function that converts a string of digits into their integer values. The function takes the form

## 4B.12 Computer Programming

```
x = atoi(string);
```

**x** is an integer variable and **string** is a character array containing a string of digits. Consider the following segment of a program:

```
number = "1988";
year = atoi(number);
```

**number** is a string variable which is assigned the string constant "1988". The function **atoi** converts the string "1988" (contained in **number**) to its numeric equivalent 1988 and assigns it to the integer variable **year**. String conversion functions are stored in the header file <std.lib.h>.

---

**Example 4B.6** Write a program which would print the alphabet set a to z and A to Z in decimal and character form.

---

The program is shown in Fig. 4B.7. In ASCII character set, the decimal numbers 65 to 90 represent upper case alphabets and 97 to 122 represent lower case alphabets. The values from 91 to 96 are excluded using an **if** statement in the **for** loop.

Program

```
main()
{
 char c;
 printf("\n\n");
 for(c = 65 ; c <= 122 ; c = c + 1)
 {
 if(c > 90 && c < 97)
 continue;
 printf("|%4d - %c ", c, c);
 }
 printf("\n");
}
```

Output

```
| 65 - A | 66 - B | 67 - C | 68 - D | 69 - E | 70 - F
| 71 - G | 72 - H | 73 - I | 74 - J | 75 - K | 76 - L
| 77 - M | 78 - N | 79 - O | 80 - P | 81 - Q | 82 - R
| 83 - S | 84 - T | 85 - U | 86 - V | 87 - W | 88 - X
| 89 - Y | 90 - Z | 97 - a | 98 - b | 99 - c | 100 - d
| 101 - e | 102 - f | 103 - g | 104 - h | 105 - i | 106 - j
| 107 - k | 108 - l | 109 - m | 110 - n | 111 - o | 112 - p
| 113 - q | 114 - r | 115 - s | 116 - t | 117 - u | 118 - v
| 119 - w | 120 - x | 121 - y | 122 - z |
```

Fig. 4B.7 Printing of the alphabet set in decimal and character form

## 4B.6 PUTTING STRINGS TOGETHER

Just as we cannot assign one string to another directly, we cannot join two strings together by the simple arithmetic addition. That is, the statements such as

```
string3 = string1 + string2;
string2 = string1 + "hello";
```

are not valid. The characters from **string1** and **string2** should be copied into the **string3** one after the other. The size of the array **string3** should be large enough to hold the total characters.



The process of combining two strings together is called *concatenation*. Example 4B.7 illustrates the concatenation of three strings.

---

**Example 4B.7** The names of employees of an organization are stored in three arrays, namely `first_name`, `second_name`, and `last_name`. Write a program to concatenate the three parts into one string to be called `name`.

---

The program is given in Fig. 4B.8. Three **for** loops are used to copy the three strings. In the first loop, the characters contained in the **first\_name** are copied into the variable **name** until the *null* character is reached. The *null* character is not copied; instead it is replaced by a *space* by the assignment statement

```
name[i] = ' ';
```

Similarly, the **second\_name** is copied into **name**, starting from the column just after the space created by the above statement. This is achieved by the assignment statement

```
name[i+j+1] = second_name[j];
```

If **first\_name** contains 4 characters, then the value of **i** at this point will be 4 and therefore the first character from **second\_name** will be placed in the *fifth cell* of **name**. Note that we have stored a space in the *fourth cell*.

In the same way, the statement

```
name[i+j+k+2] = last_name[k];
```

is used to copy the characters from **last\_name** into the proper locations of **name**.

At the end, we place a null character to terminate the concatenated string **name**. In this example, it is important to note the use of the expressions **i+j+1** and **i+j+k+2**.

```

Program
main()
{
 int i, j, k ;
 char first_name[10] = {"VISWANATH"} ;
 char second_name[10] = {"PRATAP"} ;
 char last_name[10] = {"SINGH"} ;
 char name[30] ;
 /* Copy first_name into name */
 for(i = 0 ; first_name[i] != '\0' ; i++)
 name[i] = first_name[i] ;
 /* End first_name with a space */
 name[i] = ' ' ;
 /* Copy second_name into name */
 for(j = 0 ; second_name[j] != '\0' ; j++)
 name[i+j+1] = second_name[j] ;
 /* End second_name with a space */
 name[i+j+1] = ' ' ;
 /* Copy last_name into name */
 for(k = 0 ; last_name[k] != '\0' ; k++)
 name[i+j+k+2] = last_name[k] ;
 /* End name with a null character */
 name[i+j+k+2] = '\0' ;
 printf("\n\n") ;
 printf("%s\n", name) ;
}

```

Output

```
VISWANATH PRATAP SINGH
```

**Fig. 4B.8** Concatenation of strings

## 4B.7 COMPARISON OF TWO STRINGS

Once again, C does not permit the comparison of two strings directly. That is, the statements such as

```
if(name1 == name2)
if(name == "ABC")
```

are not permitted. It is therefore necessary to compare the two strings to be tested, character by character. The comparison is done until there is a mismatch or one of the strings terminates into a null character, whichever occurs first. The following segment of a program illustrates this.

```
i=0;
while(str1[i] == str2[i] && str1[i] != '\0'
 && str2[i] != '\0')
 i = i+1;
if (str1[i] == '\0' && str2[i] == '\0')
 printf("strings are equal\n");
else
 printf("strings are not equal\n");
```

## 4B.8 STRING-HANDLING FUNCTIONS

Fortunately, the C library supports a large number of string-handling functions that can be used to carry out many of the string manipulations discussed so far. Following are the most commonly used string-handling functions.

| Function | Action                         |
|----------|--------------------------------|
| strcat() | concatenates two strings       |
| strcmp() | compares two strings           |
| strcpy() | copies one string over another |
| strlen() | finds the length of a string   |

We shall discuss briefly how each of these functions can be used in the processing of strings.

### 4B.8.1 strcat() Function

The **strcat** function joins two strings together. It takes the following form:

```
strcat(string1, string2);
```

**string1** and **string2** are character arrays. When the function **strcat** is executed, **string2** is appended to **string1**. It does so by removing the null character at the end of **string1** and placing **string2** from there. The string at **string2** remains unchanged. For example, consider the following three strings:

will result in:

|         |   |   |   |   |   |   |   |   |   |    |   |   |   |
|---------|---|---|---|---|---|---|---|---|---|----|---|---|---|
|         | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 0 | 1 | 2 |
| Part1 = | V | E | R | Y |   | G | O | O | D | \0 |   |   |   |

|         |   |   |   |   |    |   |   |
|---------|---|---|---|---|----|---|---|
|         | 0 | 1 | 2 | 3 | 4  | 5 | 6 |
| Part2 = | G | O | O | D | \0 |   |   |

Execution of the statement

```
strcat(part1, part2);
```

will result in:

|         | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 0 | 1 | 2 |
|---------|---|---|---|---|---|---|---|---|---|----|---|---|---|
| Part1 = | V | E | R | Y |   | G | O | O | D | \0 |   |   |   |

|         | 0 | 1 | 2 | 3 | 4  | 5 | 6 |
|---------|---|---|---|---|----|---|---|
| Part2 = | G | O | O | D | \0 |   |   |

while the statement  
will result in:

|         | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8  | 9 | 0 | 1 | 2 |
|---------|---|---|---|---|---|---|---|---|----|---|---|---|---|
| Part1 = | V | E | R | Y |   | B | A | D | \0 |   |   |   |   |

|         | 0 | 1 | 2 | 3  | 4 | 5 | 6 |
|---------|---|---|---|----|---|---|---|
| Part3 = | B | A | D | \0 |   |   |   |

We must make sure that the size of **string1** (to which **string2** is appended) is large enough to accommodate the final string.

**strcat** function may also append a string constant to a string variable. The following is valid:

```
strcat(part1, "GOOD");
```

C permits nesting of **strcat** functions. For example, the statement

```
strcat(strcat(string1, string2), string3);
```

is allowed and concatenates all the three strings together. The resultant string is stored in **string1**.

## 4B.8.2 strcmp() Function

The **strcmp** function compares two strings identified by the arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first nonmatching characters in the strings. It takes the form:

```
strcmp(string1, string2);
```

**string1** and **string2** may be string variables or string constants. Examples are:

```
strcmp(name1, name2);
strcmp(name1, "John");
strcmp("Rom", "Ram");
```

Our major concern is to determine whether the strings are equal; if not, which is alphabetically above. The value of the mismatch is rarely important. For example, the statement

```
strcmp("their", "there");
```

will return a value of  $-9$  which is the numeric difference between ASCII "i" and ASCII "r". That is, "i" minus "r" in ASCII code is  $-9$ . If the value is negative, **string1** is alphabetically above **string2**.

## 4B.8.3 strcpy() Function

The **strcpy** function works almost like a string-assignment operator. It takes the form:

## 4B.16 Computer Programming

```
strcpy(string1, string2);
```

and assigns the contents of **string2** to **string1**. **string2** may be a character array variable or a string constant. For example, the statement

```
strcpy(city, "DELHI");
```

will assign the string "DELHI" to the string variable **city**. Similarly, the statement

```
strcpy(city1, city2);
```

will assign the contents of the string variable **city2** to the string variable **city1**. The size of the array **city1** should be large enough to receive the contents of **city2**.

### 4B.8.4 strlen() Function

This function counts and returns the number of characters in a string. It takes the form

```
n = strlen(string);
```

Where **n** is an integer variable, which receives the value of the length of the **string**. The argument may be a string constant. The counting ends at the first null character.

---

**Example 4B.8** s1, s2, and s3 are three string variables. Write a program to read two string constants into s1 and s2 and compare whether they are equal or not. If they are not, join them together. Then copy the contents of s1 to the variable s3. At the end, the program should print the contents of all the three variables and their lengths.

---

The program is shown in Fig. 4B.9. During the first run, the input strings are "New" and "York". These strings are compared by the statement

```
x = strcmp(s1, s2);
```

Since they are not equal, they are joined together and copied into **s3** using the statement

```
strcpy(s3, s1);
```

The program outputs all the three strings with their lengths.

During the second run, the two strings **s1** and **s2** are equal, and therefore, they are not joined together. In this case all the three strings contain the same string constant "London".

Program

```
#include <string.h>
main()
{ char s1[20], s2[20], s3[20];
 int x, l1, l2, l3;
 printf("\n\nEnter two string constants \n");
 printf("?");
 scanf("%s %s", s1, s2);
 /* comparing s1 and s2 */
 x = strcmp(s1, s2);
 if(x != 0)
 { printf("\n\nStrings are not equal \n");
 strcat(s1, s2); /* joining s1 and s2 */
 }
 else
 printf("\n\nStrings are equal \n");
 /* copying s1 to s3
```

```

 strcpy(s3, s1);
/* Finding length of strings */
 l1 = strlen(s1);
 l2 = strlen(s2);
 l3 = strlen(s3);
/* output */
 printf("\ns1 = %s\t length = %d characters\n", s1, l1);
 printf("s2 = %s\t length = %d characters\n", s2, l2);
 printf("s3 = %s\t length = %d characters\n", s3, l3);
}

```

Output

```

Enter two string constants
? New York

Strings are not equal
s1 = NewYork length = 7 characters
s2 = York length = 4 characters
s3 = NewYork length = 7 characters

Enter two string constants
? London London

Strings are equal
s1 = London length = 6 characters
s2 = London length = 6 characters
s3 = London length = 6 characters

```

**Fig. 4B.9** Illustration of string handling functions

#### Other String Functions

The header file **<string.h>** contains many more string manipulation functions. They might be useful in certain situations.

#### 4B.8.5 strncpy() Function

In addition to the function **strcpy** that copies one string to another, we have another function **strncpy** that copies only the left-most *n* characters of the source string to the target string variable. This is a three-parameter function and is invoked as follows:

```
strncpy(s1, s2, 5);
```

This statement copies the first 5 characters of the source string **s2** into the target string **s1**. Since the first 5 characters may not include the terminating null character, we have to place it explicitly in the 6th position of **s2** as shown below:

```
s1[6] = '\0';
```

Now, the string **s1** contains a proper string.

#### 4B.8.6 strncmp() Function

A variation of the function **strcmp** is the function **strncmp**. This function has three parameters as illustrated in the function call below:

```
strncmp (s1, s2, n);
```

this compares the left-most *n* characters of **s1** to **s2** and returns.

- (a) 0 if they are equal;
- (b) negative number, if s1 sub-string is less than s2; and
- (c) positive number, otherwise.

### 4B.8.7 strncat() Function

This is another concatenation function that takes three parameters as shown below:

```
strncat (s1, s2, n);
```

This call will concatenate the left-most **n** characters of **s2** to the end of **s1**. Example:

S1: 

|   |   |   |   |    |  |  |  |  |  |  |  |
|---|---|---|---|----|--|--|--|--|--|--|--|
| B | A | L | A | \0 |  |  |  |  |  |  |  |
|---|---|---|---|----|--|--|--|--|--|--|--|

S2: 

|   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|----|
| G | U | R | U | S | A | M | Y | \0 |
|---|---|---|---|---|---|---|---|----|

After **strncat (s1, s2, 4);** execution:

S1: 

|   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|----|
| B | A | L | A | G | U | R | U | \0 |
|---|---|---|---|---|---|---|---|----|

### 4B.8.8 strstr() Function

It is a two-parameter function that can be used to locate a sub-string in a string. This takes the forms:

```
strstr (s1, s2);
```

```
strstr (s1, "ABC");
```

The function **strstr** searches the string **s1** to see whether the string **s2** is contained in **s1**. If yes, the function returns the position of the first occurrence of the sub-string. Otherwise, it returns a NULL pointer. Example.

```
if (strstr (s1, s2) == NULL)
 printf("substring is not found");
else
 printf("s2 is a substring of s1");
```

We also have functions to determine the existence of a character in a string. The function call

```
strchr(s1, 'm');
```

will locate the first occurrence of the character 'm' and the call

```
strrchr(s1, 'm');
```

will locate the last occurrence of the character 'm' in the string **s1**.

#### Warnings

- When allocating space for a string during declaration, remember to count the terminating null character.
- When creating an array to hold a copy of a string variable of unknown size, we can compute the size required using the expression **strlen (stringname) + 1**.
- When copying or concatenating one string to another, we must ensure that the target (destination) string has enough space to hold the incoming characters. Remember that no error message will be available even if this condition is not satisfied. The copying may overwrite the memory and the program may fail in an unpredictable way.
- When we use **strncpy** to copy a specific number of characters from a source string, we must ensure to append the null character to the target string, in case the number of characters is less than or equal to the source string.

## 4B.9 TABLE OF STRINGS

We often use lists of character strings, such as a list of the names of students in a class, list of the names of employees in an organization, list of places, etc. A list of names can be treated as a table of strings and a two-dimensional character array can be used to store the entire list. For example, a character array **student[30][15]** may be used to store a list of 30 names, each of length not more than 15 characters. Shown below is a table of five cities:

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| C | h | a | n | d | i | g | a | r | h |
| M | a | d | r | a | s |   |   |   |   |
| A | h | m | e | d | a | b | a | d |   |
| H | y | d | e | r | a | b | a | d |   |
| B | o | m | b | a | y |   |   |   |   |

This table can be conveniently stored in a character array **city** by using the following declaration:

```
char city[][]
{
 "Chandigarh",
 "Madras",
 "Ahmedabad",
 "Hyderabad",
 "Bombay"
};
```

To access the name of the *i*th city in the list, we write

```
city[i-1]
```

and therefore **city[0]** denotes "Chandigarh", **city[1]** denotes "Madras" and so on. This shows that once an array is declared as two-dimensional, it can be used like a one-dimensional array in further manipulations. That is, the table can be treated as a column of strings.

---

**Example 4B.9** Write a program that would sort a list of names in alphabetical order.

---

A program to sort the list of strings in alphabetical order is given in Fig. 4B.10. It employs the method of bubble sorting described in Case Study 1 in the previous chapter.

Program

```
#define ITEMS 5
#define MAXCHAR 20
main()
{
 char string[ITEMS][MAXCHAR], dummy[MAXCHAR];
 int i = 0, j = 0;
 /* Reading the list */
 printf ("Enter names of %d items \n ",ITEMS);
 while (i < ITEMS)
 scanf ("%s", string[i++]);
 /* Sorting begins */
 for (i=1; i < ITEMS; i++) /* Outer loop begins */
```

```

 {
 for (j=1; j <= ITEMS-i ; j++) /*Inner loop begins*/
 {
 if (strcmp (string[j-1], string[j]) > 0)
 { /* Exchange of contents */
 strcpy (dummy, string[j-1]);
 strcpy (string[j-1], string[j]);
 strcpy (string[j], dummy);
 }
 } /* Inner loop ends */
 } /* Outer loop ends */
 /* Sorting completed */
 printf ("\nAlphabetical list \n\n");
 for (i=0; i < ITEMS ; i++)
 printf ("%s", string[i]);
 }

```

Output

```

Enter names of 5 items
London Manchester Delhi Paris Moscow
Alphabetical list
Delhi
London
Manchester
Moscow
Paris

```

**Fig. 4B.10** *Sorting of strings in alphabetical order*

Note that a two-dimensional array is used to store the list of strings. Each string is read using a **scanf** function with **%s** format. Remember, if any string contains a white space, then the part of the string after the white space will be treated as another item in the list by the **scanf**. In such cases, we should read the entire line as a string using a suitable algorithm. For example, we can use **gets** function to read a line of text containing a series of words. We may also use **puts** function in place of **scanf** for output.

## 4B.10 OTHER FEATURES OF STRINGS

Other aspects of strings we have not discussed in this chapter include:

- Manipulating strings using pointers.
- Using string as function parameters.
- Declaring and defining strings as members of structures.

These topics will be dealt with later when we discuss functions, structures and pointers.

## 4B.11 STRING / DATA CONVERSION

It may be required for us to convert strings to equivalent data values (of int or float type) and vice versa. This can be achieved using certain built-in functions of C, as described in Table 4B.1:



**TABLE 4B.1** String / Data Conversion Functions

| Function       | Description                                                                                                                                                                                                                                                                                                                                     | Syntax                                                                                                                 |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|
| atoi()         | Converts a string <i>s</i> to an equivalent integer value                                                                                                                                                                                                                                                                                       | int atoi (const char *s);                                                                                              |
| atol()         | Converts a string <i>s</i> to an equivalent long integer value                                                                                                                                                                                                                                                                                  | long atoi (const char *s);                                                                                             |
| atof()         | Converts a string <i>s</i> to an equivalent floating-point value                                                                                                                                                                                                                                                                                | float atoi (const char *s);                                                                                            |
| ecvt(), fcvt() | Convert floating point numbers to equivalent null-terminated strings<br><br>ecvt and fcvt differ in their interpretation of the argument <i>ndig</i> . In case of ecvt, it specifies the total number of characters in the equivalent string; however in case of fcvt it specifies the equivalent number of characters after the decimal point. | char *ecvt(double value, int ndig, int *dec, int *sign)<br><br>char *fcvt(double value, int ndig, int *dec, int *sign) |

**Example 4B.9** Write a program to demonstrate the conversion of a string to its equivalent integer and floating-point value.

**Program**

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

void main()
{
 char *str = "834.41";
 clrscr();

 printf("String = %s \n\nInteger Data Value = %d\n\n
 Float Data Value = %.2f", str, atoi(str),atof(str));
 getch();
}
```

**Output**

String = 834.41

Integer Data Value = 834

Float Data Value = 834.41



## Just Remember

- Character constants are enclosed in single quotes and string constants are enclosed in double quotes.
- Allocate sufficient space in a character array to hold the null character at the end.
- Avoid processing single characters as strings.
- Using the address operator **&** with a **string** variable in the **scanf** function call is an error.
- It is a compile time error to assign a string to a character variable.
- Using a string variable name on the left of the assignment operator is illegal.
- When accessing individual characters in a string variable, it is logical error to access outside the array bounds.
- Strings cannot be manipulated with operators. Use string functions.
- Do not use string functions on an array **char** type that is not terminated with the null character.
- Do not forget to append the null character to the target string when the number of characters copied is less than or equal to the source string.
- Be aware the return values when using the functions **strcmp** and **strncmp** for comparing strings.
- When using string functions for copying and concatenating strings, make sure that the target string has enough space to store the resulting string. Otherwise memory overwriting may occur.
- The header file `<stdio.h>` is required when using standard I/O functions.
- The header file `<ctype.h>` is required when using character handling functions.
- The header file `<stdlib.h>` is required when using general utility functions.
- The header file `<string.h>` is required when using string manipulation functions.
- A string is nothing but a character array terminated with a null character.
- `gets` and `puts` are quite handy string i/o functions.
- Character arithmetic operations are actually performed on the ASCII values of the operand characters.
- Use `strlen` function to determine the length of a string.



## Multiple Choice Questions

- Which of the following is used to represent the end of a string?
  - Blank space
  - Null character
  - Newline character
  - Last element of the string
- Which of the following is used to display a string on the I/O console?
 

|        |        |
|--------|--------|
| (a) %s | (b) %c |
| (c) %d | (d) %f |
- Which of the following is true for `getchar`?
  - Read a string of characters
  - Read a character
  - Read the characters until `\n` is encountered
  - None of the above
- What will be the result of the following character arithmetic expression?  
`X = 'A' - 2;`

|        |        |
|--------|--------|
| (a) 63 | (b) 64 |
| (c) 65 | (d) 66 |
- Which of the following is used to determine the length of a string?
 

|                         |                         |
|-------------------------|-------------------------|
| (a) <code>strlen</code> | (b) <code>strcmp</code> |
| (c) <code>strcpy</code> | (d) <code>strcat</code> |

6. Which of the following header files are required to be included for performing string operations?
  - (a) string.h                      (b) conio.h
  - (c) stdio.h                      (d) ctype.h
7. What value will strlen function return for the string {'R','a','m','/'0'}
8. During concatenation, which of the following situation will occur if the target string is of lesser space then the two source strings?
  - (a) 3                              (b) 4
9. Which of the following is the correct syntax for copying a string S1 into S2?
  - (a) strcpy(S2,S1);    (b) strcpy(S1,S2);
  - (c) strcmp(S1,S2);    (d) strcmp(S2,S1);
10. Which of the following should be used for printing a “ inside the printf statement?
  - (a) “”””                      (b) “\
  - (c) \”                              (d) /”



## Case Study

### 1. Counting Words in a Text

One of the practical applications of string manipulations is counting the words in a text. We assume that a word is a sequence of any characters, except escape characters and blanks, and that two words are separated by one blank character. The algorithm for counting words is as follows:

1. Read a line of text.
2. Beginning from the first character in the line, look for a blank. If a blank is found, increment words by 1.
3. Continue steps 1 and 2 until the last line is completed.

The implementation of this algorithm is shown in Fig. 4B.6. The first **while** loop will be executed once for each line of text. The end of text is indicated by pressing the ‘Return’ key an extra time after the entire text has been entered. The extra ‘Return’ key causes a newline character as input to the last line and as a result, the last line contains only the null character.

The program checks for this special line using the test

```
if (line[0] == '\0')
```

and if the first (and only the first) character in the line is a null character, then counting is terminated. Note the difference between a null character and a blank character.

Program

```
#include <stdio.h>
main()
{
 char line[81], ctr;
 int i,c,
 end = 0,
 characters = 0,
 words = 0,
 lines = 0;
 printf("KEY IN THE TEXT.\n");
 printf("GIVE ONE SPACE AFTER EACH WORD.\n");
```

```

printf("WHEN COMPLETED, PRESS 'RETURN'.\n\n");
while(end == 0)
{
 /* Reading a line of text */
 c = 0;
 while((ctr=getchar()) != '\n')
 line[c++] = ctr;
 line[c] = '\0';
 /* counting the words in a line */
 if(line[0] == '\0')
 break ;
 else
 {
 words++;
 for(i=0; line[i] != '\0';i++)
 if(line[i] == ' ' || line[i] == '\t')
 words++;
 }
 /* counting lines and characters */
 lines = lines +1;
 characters = characters + strlen(line);
}
printf ("\n");
printf("Number of lines = %d\n", lines);
printf("Number of words = %d\n", words);
printf("Number of characters = %d\n", characters);
}

```

**Output**

```

KEY IN THE TEXT.
GIVE ONE SPACE AFTER EACH WORD.
WHEN COMPLETED, PRESS 'RETURN'.

Admiration is a very short-lived passion.
Admiration involves a glorious obliquity of vision.
Always we like those who admire us but we do not
like those whom we admire.
Fools admire, but men of sense approve.

Number of lines = 5
Number of words = 36
Number of characters = 205

```

**Fig. 4B.11** Counting of characters, words and lines in a text

The program also counts the number of lines read and the total number of characters in the text. Remember, the last line containing the null string is not counted.

After the first **while** loop is exited, the program prints the results of counting.

## 2. Processing of a Customer List

Telephone numbers of important customers are recorded as follows:

| Full name             | Telephone number |
|-----------------------|------------------|
| Joseph Louis Lagrange | 869245           |
| Jean Robert Argand    | 900823           |
| Carl Freidrich Gauss  | 806788           |
| -----                 | -----            |
| -----                 | -----            |

It is desired to prepare a revised alphabetical list with surname (last name) first, followed by a comma and the initials of the first and middle names. For example,

Argand,J.R

We create a table of strings, each row representing the details of one person, such as first\_name, middle\_name, last\_name, and telephone\_number. The columns are interchanged as required and the list is sorted on the last\_name. Figure 4B.12 shows a program to achieve this.

Program

```
#define CUSTOMERS 10

main()
{
 char first_name[20][10], second_name[20][10],
 surname[20][10], name[20][20],
 telephone[20][10], dummy[20];

 int i,j;

 printf("Input names and telephone numbers \n");
 printf("?");
 for(i=0; i < CUSTOMERS ; i++)
 {
 scanf("%s %s %s %s", first_name[i],
 second_name[i], surname[i], telephone[i]);

 /* converting full name to surname with initials */
 strcpy(name[i], surname[i]);
 strcat(name[i], ",");
 dummy[0] = first_name[i][0];
 dummy[1] = '\0';
 strcat(name[i], dummy);
 strcat(name[i], ".");
 dummy[0] = second_name[i][0];
 dummy[1] = '\0';
 strcat(name[i], dummy);
 }

 /* Alphabetical ordering of surnames */
 for(i=1; i <= CUSTOMERS-1; i++)
 for(j=1; j <= CUSTOMERS-i; j++)
 if(strcmp (name[j-1], name[j]) > 0)
```

#### 4B.26 Computer Programming

```
 {
 /* Swaping names */
strcpy(dummy, name[j-1]);
strcpy(name[j-1], name[j]);
strcpy(name[j], dummy);

 /* Swaping telephone numbers */
strcpy(dummy, telephone[j-1]);
strcpy(telephone[j-1], telephone[j]);
strcpy(telephone[j], dummy);
 }

 /* printing alphabetical list */
printf("\nCUSTOMERS LIST IN ALPHABETICAL ORDER \n\n");
for(i=0; i < CUSTOMERS ; i++)
 printf(" %-20s\t %-10s\n", name[i], telephone[i]);
}
```

##### Output

```
Input names and telephone numbers
?Gottfried Wilhelm Leibniz 711518
Joseph Louis Lagrange 869245
Jean Robert Argand 900823
Carl Freidrich Gauss 806788
Simon Denis Poisson 853240
Friedrich Wilhelm Bessel 719731
Charles Francois Sturm 222031
George Gabriel Stokes 545454
Mohandas Karamchand Gandhi 362718
Josian Willard Gibbs 123145
```

##### CUSTOMERS LIST IN ALPHABETICAL ORDER

|               |        |
|---------------|--------|
| Argand, J.R   | 900823 |
| Bessel, F.W   | 719731 |
| Gandhi, M.K   | 362718 |
| Gauss, C.F    | 806788 |
| Gibbs, J.W    | 123145 |
| Lagrange, J.L | 869245 |
| Leibniz, G.W  | 711518 |
| Poisson, S.D  | 853240 |
| Stokes, G.G   | 545454 |
| Sturm, C.F    | 222031 |

**Fig. 4B.12** Program to alphabetize a customer list



## Review Questions

4B.1 State whether the following statements are *true* or *false*.

- (a) When initializing a string variable during its declaration, we must include the null character as part of the string constant, like "GOOD\0".
- (b) The **gets** function automatically appends the null character at the end of the string read from the keyboard.
- (c) When reading a string with **scanf**, it automatically inserts the terminating null character.
- (d) String variables cannot be used with the assignment operator.
- (e) We cannot perform arithmetic operations on character variables.
- (f) We can assign a character constant or a character variable to an **int** type variable.
- (g) The function **scanf** cannot be used in any way to read a line of text with the white-spaces.
- (h) The ASCII character set consists of 128 distinct characters.
- (i) In the ASCII collating sequence, the uppercase letters precede lowercase letters.
- (j) In C, it is illegal to mix character data with numeric data in arithmetic operations.
- (k) The function **getchar** skips white-space during input.
- (l) In C, strings cannot be initialized at run time.
- (m) The input function **gets** has one string parameter.
- (n) The function call **strcpy(s2, s1)**; copies string s2 into string s1.
- (o) The function call **strcmp("abc", "ABC")**; returns a positive number.

4B.2 Fill in the blanks in the following statements.

- (a) We can use the conversion specification \_\_\_\_\_ in **scanf** to read a line of text.
- (b) We can initialize a string using the string manipulation function \_\_\_\_\_.
- (c) The function **strncat** has \_\_\_\_\_ parameters.
- (d) To use the function **atoi** in a program, we must include the header file \_\_\_\_\_.
- (e) The function \_\_\_\_\_ does not require any conversion specification to read a string from the keyboard.
- (f) The function \_\_\_\_\_ is used to determine the length of a string.
- (g) The \_\_\_\_\_ string manipulation function determines if a character is contained in a string.
- (h) The function \_\_\_\_\_ is used to sort the strings in alphabetical order.
- (i) The function call **strcat (s2, s1)**; appends \_\_\_\_\_ to \_\_\_\_\_.
- (j) The **printf** may be replaced by function \_\_\_\_\_ for printing strings.

4B.3 Describe the limitations of using **getchar** and **scanf** functions for reading strings.

4B.4 Character strings in C are automatically terminated by the *null* character. Explain how this feature helps in string manipulations.

4B.5 Strings can be assigned values as follows:

- (a) During type declaration  
char string[ ] = {"....."};
- (b) Using **strcpy** function  
strcpy(string, ".....");
- (c) Reading using **scanf** function  
scanf("%s", string);
- (d) Reading using **gets** function  
gets(string);

Compare them critically and describe situations where one is superior to the others.

#### 4B.28 Computer Programming

4B.6 Assuming the variable **string** contains the value "The sky is the limit.", determine what output of the following program segments will be.

- (a) `printf("%s", string);`
- (b) `printf("%25.10s", string);`
- (c) `printf("%s", string[0]);`
- (d) `for (i=0; string[i] != "."; i++)  
    printf("%c", string[i]);`
- (e) `for (i=0; string[i] != '\0'; i++;)  
    printf("%d\n", string[i]);`
- (f) `for (i=0; i <= strlen(string); ;)`

```
{
 string[i++] = i;
 printf("%s\n", string[i]);
}
```

- (g) `printf("%c\n", string[10] + 5);`
- (h) `printf("%c\n", string[10] + 5')`

4B.7 Which of the following statements will correctly store the concatenation of strings **s1** and **s2** in string **s3**?

- (a) `s3 = strcat (s1, s2);`
- (b) `strcat (s1, s2, s3);`
- (c) `strcat (s3, s2, s1);`
- (d) `strcpy (s3, strcat (s1, s2));`
- (e) `strcmp (s3, strcat (s1, s2));`
- (f) `strcpy (strcat (s1, s2), s3);`

4B.8 What will be the output of the following statement?

```
printf ("%d", strcmp ("push", "pull"));
```

4B.9 Assume that **s1**, **s2** and **s3** are declared as follows:

```
char s1[10] = "he", s2[20] = "she",
s3[30], s4[30];
```

What will be the output of the following statements executed in sequence?

```
printf("%s", strcpy(s3, s1));
printf("%s", strcat(strcat(strcpy(s4,
s1), "or"), s2));
printf("%d %d",
strlen(s2)+strlen(s3), strlen(s4));
```

4B.10 Find errors, if any, in the following code segments;

- (a) `char str[10]  
    strncpy(str, "GOD", 3);  
    printf("%s", str);`
- (b) `char str[10];  
    strcpy(str, "Balagurusamy");`
- (c) `if strstr("Balagurusamy", "guru") ==  
    0;  
    printf("Substring is found");`
- (d) `char s1[5], s2[10],  
    gets(s1, s2);`

4B.11 What will be the output of the following segment?

```
char s1[] = "Kolkotta" ;
char s2[] = "Pune" ;
strcpy (s1, s2) ;
printf("%s", s1) ;
```

4B.12 What will be the output of the following segment?

```
char s1[] = "NEW DELHI" ;
char s2[] = "BANGALORE" ;
strncpy (s1, s2, 3) ;
printf("%s", s1) ;
```

4B.13 What will be the output of the following code?

```
char s1[] = "Jabalpur" ;
char s2[] = "Jaipur" ;
printf(strncmp(s1, s2, 2));
```

4B.14 What will be the output of the following code?

```
char s1[] = "ANIL KUMAR GUPTA";
char s2[] = "KUMAR";
printf (strstr (s1, s2));
```

4B.15 Compare the working of the following functions:

- (a) `strcpy` and `strncpy`;
- (b) `strcat` and `strncat`; and
- (c) `strcmp` and `strncmp`.





## Programming Exercises

- 4B.1 Write a program, which reads your name from the keyboard and outputs a list of ASCII codes, which represent your name.
- 4B.2 Write a program to do the following:
- To output the question “Who is the inventor of C ?”
  - To accept an answer.
  - To print out “Good” and then stop, if the answer is correct.
  - To output the message ‘try again’, if the answer is wrong.
  - To display the correct answer when the answer is wrong even at the third attempt and stop.
- 4B.3 Write a program to extract a portion of a character string and print the extracted string. Assume that *m* characters are extracted, starting with the *n*th character.
- 4B.4 Write a program which will read a text and count all occurrences of a particular word.
- 4B.5 Write a program which will read a string and rewrite it in the alphabetical order. For example, the word STRING should be written as GINRST.
- 4B.6 Write a program to replace a particular word by another word in a given string. For example, the word “PASCAL” should be replaced by “C” in the text “It is good to program in PASCAL language.”
- 4B.7 A Maruti car dealer maintains a record of sales of various vehicles in the following form:
- | Vehicle type | Month of sales | Price  |
|--------------|----------------|--------|
| MARUTI-800   | 02/01          | 210000 |
| MARUTI-DX    | 07/01          | 265000 |
| GYPSY        | 04/02          | 315750 |
| MARUTI-VAN   | 08/02          | 240000 |
- Write a program to read this data into a table of strings and output the details of a particular vehicle sold during a specified period. The program should request the user to input the vehicle type and the period (starting month, ending month).
- 4B.8 Write a program that reads a string from the keyboard and determines whether the string is a *palindrome* or not. (A string is a palindrome if it can be read from left and right with the same meaning. For example, Madam and Anna are palindrome strings. Ignore capitalization).
- 4B.9 Write program that reads the cost of an item in the form RRRR.PP (Where RRRR denotes Rupees and PP denotes Paise) and converts the value to a string of words that expresses the numeric value in words. For example, if we input 125.75, the output should be “ONE HUNDRED TWENTY FIVE AND PAISE SEVENTY FIVE”.
- 4B.10 Develop a program that will read and store the details of a list of students in the format
- | Roll No. | Name  | Marks obtained |
|----------|-------|----------------|
| .....    | ..... | .....          |
| .....    | ..... | .....          |
| .....    | ..... | .....          |
- and produce the following output lists:
- Alphabetical list of names, roll numbers and marks obtained.
  - List sorted on roll numbers.
  - List sorted on marks (rank-wise list)
- 4B.11 Write a program to read two strings and compare them using the function **strcmp()** and print a message that the first string is equal, less, or greater than the second one.
- 4B.12 Write a program to read a line of text from the keyboard and print out the number of occurrences of a given substring using the function **strstr()**.
- 4B.13 Write a program that will copy *m* consecutive characters from a string *s1* beginning at position *n* into another string *s2*.

#### 4B.30 Computer Programming

4B.14 Write a program to create a directory of students with roll numbers. The program should display the roll number for a specified name and vice-versa.

4B.15 Given a string  
`char str [ ] = "123456789";`

Write a program that displays the following:

```
1
2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```



### Key Terms

---

- **String:** It is a sequence of characters that is treated as a single data item.
- **strcat:** It is a function used to join two strings together.
- **strcmp:** It is a function used to compare two strings passed as arguments. It generates a value of 0 if they are equal.
- **strcpy:** It is a function used to copy one string into another.
- **strstr:** It is a two-parameter function that is used to locate a sub-string in a string.

# 5 Structures and Unions

---

### 5.1 INTRODUCTION

We have seen that arrays can be used to represent a group of data items that belong to the same type, such as **int** or **float**. However, we cannot use an array if we want to represent a collection of data items of different types using a single name. Fortunately, C supports a constructed data type known as *structures*, a mechanism for packing data of different types. A structure is a convenient tool for handling a group of logically related data items. For example, it can be used to represent a set of attributes, such as `student_name`, `roll_number` and `marks`. The concept of a structure is analogous to that of a ‘record’ in many other languages. More examples of such structures are:

|           |   |                                 |
|-----------|---|---------------------------------|
| time      | : | seconds, minutes, hours         |
| date      | : | day, month, year                |
| book      | : | author, title, price, year      |
| city      | : | name, country, population       |
| address   | : | name, door-number, street, city |
| inventory | : | item, stock, value              |
| customer  | : | name, telephone, city, category |

Structures help to organize complex data in a more meaningful way. It is a powerful concept that we may often need to use in our program design. This chapter is devoted to the study of structures and their applications in program development. Another related concept known as *unions* is also discussed.

### 5.2 DEFINING A STRUCTURE

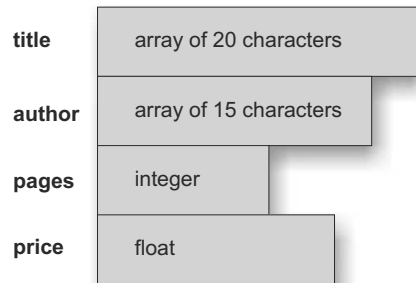
Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

## 5.2 Computer Programming

```
struct book_bank
{
 char title[20];
 char author[15];
 int pages;
 float price;
};
```

The keyword **struct** declares a structure to hold the details of four data fields, namely **title**, **author**, **pages**, and **price**. These fields are called *structure elements* or *members*. Each member may belong to a different type of data. **book\_bank** is the name of the structure and is called the *structure tag*. The tag name may be used subsequently to declare variables that have the tag's structure.

Note that the above definition has not declared any variables. It simply describes a format called *template* to represent information as shown below:



The general format of a structure definition is as follows:

```
struct tag_name
{
 data_type member1;
 data_type member2;

};
```

In defining a structure you may note the following syntax:

1. The template is terminated with a semicolon.
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as **book\_bank** can be used to declare structure variables of its type, later in the program.

### Arrays vs Structures

Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in a number of ways.

1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is derived data type whereas a structure is a programmer-defined one.
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

## 5.3 DECLARING STRUCTURE VARIABLES

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of any other data types. It includes the following elements:

1. The keyword **struct**.
2. The structure tag name.
3. List of variable names separated by commas.
4. A terminating semicolon.

For example, the statement

```
struct book_bank, book1, book2, book3;
```

declares **book1**, **book2**, and **book3** as variables of type **struct book\_bank**.

Each one of these variables has four members as specified by the template. The complete declaration might look like this:

```
struct book_bank
{
 char title[20];
 char author[15];
 int pages;
 float price;
};
struct book_bank book1, book2, book3;
```

Remember that the members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as **book1**. When the compiler comes across a declaration statement, it reserves memory space for the structure variables. It is also allowed to combine both the structure definition and variables declaration in one statement.

The declaration

```
struct book_bank
{
 char title[20];
 char author[15];
 int pages;
 float price;
} book1, book2, book3;
```

is valid. The use of tag name is optional here. For example:

```
struct
{

} book1, book2, book3;
```

declares **book1**, **book2**, and **book3** as structure variables representing three books, but does not include a tag name. However, this approach is not recommended for two reasons.

1. Without a tag name, we cannot use it for future declarations:
2. Normally, structure definitions appear at the beginning of the program file, before any variables or functions are defined. They may also appear before the **main**, along with macro definitions, such as **#define**. In such cases, the definition is *global* and can be used by other functions as well.

### Type-Defined Structures

We can use the keyword **typedef** to define a structure as follows:

```
typedef struct
{
 type member1;
 type member2;

} type_name;
```

The `type_name` represents structure definition associated with it and therefore can be used to declare structure variables as shown below:

```
type_name variable1, variable2,;
```

Remember that (1) the name `type_name` is the type definition name, not a variable and (2) we cannot define a variable with `typedef` declaration.

## 5.4 ACCESSING STRUCTURE MEMBERS

We can access and assign values to the members of a structure in a number of ways. As mentioned earlier, the members themselves are not variables. They should be linked to the structure variables in order to make them meaningful members. For example, the word **title**, has no meaning whereas the phrase ‘title of book3’ has a meaning. The link between a member and a variable is established using the *member operator* ‘.’ which is also known as ‘dot operator’ or ‘period operator’. For example,

`book1.price`

is the variable representing the price of **book1** and can be treated like any other ordinary variable. Here is how we would assign values to the members of **book1**:

```
strcpy(book1.title, "BASIC");
strcpy(book1.author, "Balagurusamy");
book1.pages = 250;
book1.price = 120.50;
```

We can also use **scanf** to give the values through the keyboard.

```
scanf("%s\n", book1.title);
scanf("%d\n", &book1.pages);
```

are valid input statements.

---

**Example 5.1** Define a structure type, struct personal that would contain person name, date of joining and salary. Using this structure, write a program to read this information for one person from the keyboard and print the same on the screen.

---

Structure definition along with the program is shown in Fig. 5s.1. The **scanf** and **printf** functions illustrate how the member operator ‘.’ is used to link the structure members to the structure variables. The variable name with a period and the member name is used like an ordinary variable.

```
Program
struct personal
{
 char name[20];
 int day;
```

```

 char month[10];
 int year;
 float salary;
 };
 main()
 {
 struct personal person;
 printf("Input Values\n");
 scanf("%s %d %s %d %f",
 person.name,
 &person.day,
 person.month,
 &person.year,
 &person.salary);
 printf("%s %d %s %d %f\n",
 person.name,
 person.day,
 person.month,
 person.year,
 person.salary);
 }

```

Output

```

Input Values
M.L.Goel 10 January 1945 4500
M.L.Goel 10 January 1945 4500.00

```

**Fig. 5.1** Defining and accessing structure members

## 5.5 STRUCTURE INITIALIZATION

Like any other data type, a structure variable can be initialized at compile time.

```

main()
{
 struct
 {
 int weight;
 float height;
 }
 student = {60, 180.75};

}

```

This assigns the value 60 to **student. weight** and 180.75 to **student. height**. There is a one-to-one correspondence between the members and their initializing values.

A lot of variation is possible in initializing a structure. The following statements initialize two structure variables. Here, it is essential to use a tag name.

```

main()
{
 struct st_record

```

## 5.6 Computer Programming

```
{
 int weight;
 float height;
};
struct st_record student1 = { 60, 180.75 };
struct st_record student2 = { 53, 170.60 };
.....
.....
}
```

Another method is to initialize a structure variable outside the function as shown below:

```
struct st_record
{
 int weight;
 float height;
} student1 = {60, 180.75};
main()
{
 struct st_record student2 = {53, 170.60};

}
```

C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variables.

Note that the compile-time initialization of a structure variable must have the following elements:

1. The keyword **struct**.
2. The structure tag name.
3. The name of the variable to be declared.
4. The assignment operator =.
5. A set of values for the members of the structure variable, separated by commas and enclosed in braces.
6. A terminating semicolon.

### Rules for Initializing Structures

There are a few rules to keep in mind while initializing structure variables at compile-time.

1. We cannot initialize individual members inside the structure template.
2. The order of values enclosed in braces must match the order of members in the structure definition.
3. It is permitted to have a partial initialization. We can initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the end of the list.
4. The uninitialized members will be assigned default values as follows:
  - Zero for integer and floating point numbers.
  - '\0' for characters and strings.



## 5.6 COPYING AND COMPARING STRUCTURE VARIABLES

Two variables of the same structure type can be copied the same way as ordinary variables. If **person1** and **person2** belong to the same structure, then the following statements are valid:

```
person1 = person2;
person2 = person1;
```

However, the statements such as

```
person1 == person2
person1 != person2
```

are not permitted. C does not permit any logical operations on structure variables. In case, we need to compare them, we may do so by comparing members individually.

---

**Example 5.2** Write a program to illustrate the comparison of structure variables.

---

The program shown in Fig. 5.2 illustrates how a structure variable can be copied into another of the same type. It also performs member-wise comparison to decide whether two structure variables are identical.

```
program
 struct class
 {
 int number;
 char name[20];
 float marks;
 };

 main()
 {
 int x;
 struct class student1 = {111,"Rao",72.50};
 struct class student2 = {222,"Reddy", 67.00};
 struct class student3;

 student3 = student2;

 x = ((student3.number == student2.number) &&
 (student3.marks == student2.marks)) ? 1 : 0;

 if(x == 1)
 {
 printf("\nstudent2 and student3 are same\n\n");
 printf("%d %s %f\n", student3.number,
 student3.name,
 student3.marks);
 }
 else
 printf("\nstudent2 and student3 are different\n\n");
 }
}
```

Output

```
student2 and student3 are same

222 Reddy 67.000000
```

**Fig. 5.2** Comparing and copying structure variables

### Word Boundaries and Slack Bytes

Computer stores structures using the concept of “word boundary”. The size of a word boundary is machine dependent. In a computer with two bytes word boundary, the members of a structure are stored *left\_aligned* on the word boundary, as shown below. A character data takes one byte and an integer takes two bytes. One byte between them is left unoccupied. This unoccupied byte is known as the *slack byte*.

When we declare structure variables, each one of them may contain slack bytes and the values stored in such slack bytes are undefined. Due to this, even if the members of two variables are equal, their structures do not necessarily compare equal. C, therefore, does not permit comparison of structures. However, we can design our own function that could compare individual members to decide whether the structures are equal or not.

## 5.7 OPERATIONS ON INDIVIDUAL MEMBERS

As pointed out earlier, the individual members are identified using the member operator, the *dot*. A member with the *dot operator* along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operators. Consider the program in Fig. 5.2. We can perform the following operations:

```
if (student1.number == 111)
 student1.marks += 10.00;
float sum = student1.marks + student2.marks;
student2.marks * = 0.5;
```

We can also apply increment and decrement operators to numeric type members. For example, the following statements are valid:

```
student1.number ++;
++ student1.number;
```

The precedence of the *member operator* is higher than all *arithmetic* and *relational* operators and therefore no parentheses are required.

### Three Ways to Access Members

We have used the dot operator to access the members of structure variables. In fact, there are two other ways. Consider the following structure:

```
typedef struct
{
 int x;
 int y;
} VECTOR;
VECTOR v, *ptr;
ptr = & n;
```

The identifier **ptr** is known as **pointer** that has been assigned the address of the structure variable **n**. Now, the members can be accessed in three ways:

- using dot notation : **n.x**
- using indirection notation : **(\*ptr).x**
- using selection notation : **ptr -> x**

The second and third methods will be considered in Chapter 14.

## 5.8 ARRAYS OF STRUCTURES

We use structures to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. For example:

```
struct class student[100];
```

defines an array called **student**, that consists of 100 elements. Each element is defined to be of the type **struct class**. Consider the following declaration:

```
struct marks
{
 int subject1;
 int subject2;
 int subject3;
};
main()
{
 struct marks student[3] =
 {{45,68,81}, {75,53,69}, {57,36,71}};
```

This declares the **student** as an array of three elements **student[0]**, **student[1]**, and **student[2]** and initializes their members as follows:

```
student[0].subject1 = 45;
student[0].subject2 = 65;
....
....
student[2].subject3 = 71;
```

Note that the array is declared just as it would have been with any other array. Since **student** is an array, we use the usual array-accessing methods to access individual elements and then the member operator to access members. Remember, each element of **student** array is a structure variable with three members.

An array of structures is stored inside the memory in the same way as a multi-dimensional array. The array **student** actually looks as shown in Fig. 5.3.

|                       |    |
|-----------------------|----|
| student [0].subject 1 | 45 |
| .subject 2            | 68 |
| .subject 3            | 81 |
| student [1].subject 1 | 75 |
| .subject 2            | 53 |
| .subject 3            | 69 |
| student [2].subject 1 | 57 |
| .subject 2            | 36 |
| .subject 3            | 71 |

**Fig. 5.3** The array *student* inside memory

---

**Example 5.3** For the student array discussed above, write a program to calculate the subject-wise and student-wise totals and store them as a part of the structure.

---

The program is shown in Fig. 5.4. We have declared a four-member structure, the fourth one for keeping the student-totals. We have also declared an **array** total to keep the subject-totals and the grand-total. The grand-total is given by **total.total**. Note that a member name can be any valid C name and can be the same as an existing structure variable name. The linked name **total.total** represents the **total** member of the structure variable **total**.

```

Program
 struct marks
 {
 int sub1;
 int sub2;
 int sub3;
 int total;
 };
 main()
 {
 int i;
 struct marks student[3] = {{45,67,81,0},
 {75,53,69,0},
 {57,36,71,0}};

 struct marks total;
 for(i = 0; i <= 2; i++)
 {
 student[i].total = student[i].sub1 +
 student[i].sub2 +
 student[i].sub3;
 total.sub1 = total.sub1 + student[i].sub1;
 total.sub2 = total.sub2 + student[i].sub2;
 total.sub3 = total.sub3 + student[i].sub3;
 total.total = total.total + student[i].total;
 }
 printf(" STUDENT TOTAL\n\n");
 for(i = 0; i <= 2; i++)
 printf("Student[%d] %d\n", i+1, student[i].total);
 printf("\n SUBJECT TOTAL\n\n");
 printf("%s %d\n%s %d\n%s %d\n",
 "Subject 1 ", total.sub1,
 "Subject 2 ", total.sub2,
 "Subject 3 ", total.sub3);

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

```

Output

| STUDENT    | TOTAL |
|------------|-------|
| Student[1] | 193   |
| Student[2] | 197   |

|                   |       |
|-------------------|-------|
| Student[3]        | 164   |
| SUBJECT           | TOTAL |
| Subject 1         | 177   |
| Subject 2         | 156   |
| Subject 3         | 221   |
| Grand Total = 554 |       |

**Fig. 5.4** Arrays of structures: Illustration of subscripted structure variables

## 5.9 ARRAYS WITHIN STRUCTURES

C permits the use of arrays as structure members. We have already used arrays of characters inside a structure. Similarly, we can use single-dimensional or multi-dimensional arrays of type **int** or **float**. For example, the following structure declaration is valid:

```
struct marks
{
 int number;
 float subject[3];
} student[2];
```

Here, the member **subject** contains three elements, **subject[0]**, **subject[1]** and **subject[2]**. These elements can be accessed using appropriate subscripts. For example, the name

```
student[1].subject[2];
```

would refer to the marks obtained in the third subject by the second student.

**Example 5.4** Rewrite the program of Example 5.3 using an array member to represent the three subjects.

The modified program is shown in Fig. 5.5. You may notice that the use of array name for subjects has simplified in code.

```
Program
main()
{
 struct marks
 {
 int sub[3];
 int total;
 };
 struct marks student[3] =
 {45,67,81,0,75,53,69,0,57,36,71,0};

 struct marks total;
 int i,j;

 for(i = 0; i <= 2; i++)
 {
 for(j = 0; j <= 2; j++)
 {
```

## 5.12 Computer Programming

```
 student[i].total += student[i].sub[j];
 total.sub[j] += student[i].sub[j];
 }
 total.total += student[i].total;
}
printf("STUDENT TOTAL\n\n");
for(i = 0; i <= 2; i++)
 printf("Student[%d] %d\n", i+1, student[i].total);

printf("\nSUBJECT TOTAL\n\n");
for(j = 0; j <= 2; j++)
 printf("Subject-%d %d\n", j+1, total.sub[j]);

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

}
```

Output

| STUDENT    | TOTAL |
|------------|-------|
| Student[1] | 193   |
| Student[2] | 197   |
| Student[3] | 164   |

| STUDENT       | TOTAL |
|---------------|-------|
| Student-1     | 177   |
| Student-2     | 156   |
| Student-3     | 221   |
| Grand Total = | 554   |

**Fig. 5.5** Use of subscripted members arrays in structures

## 5.10 STRUCTURES WITHIN STRUCTURES

Structures within a structure means *nesting* of structures. Nesting of structures is permitted in C. Let us consider the following structure defined to store information about the salary of employees.

```
struct salary
{
 char name;
 char department;
 int basic_pay;
 int dearness_allowance;
 int house_rent_allowance;
 int city_allowance;
}
employee;
```

This structure defines name, department, basic pay and three kinds of allowances. We can group all the items related to allowance together and declare them under a substructure as shown below:

```

struct salary
{
 char name;
 char department;
 struct
 {
 int dearness;
 int house_rent;
 int city;
 }
 allowance;
}
employee;

```

The salary structure contains a member named **allowance**, which itself is a structure with three members. The members contained in the inner structure namely **dearness**, **house\_rent**, and **city** can be referred to as:

```

employee.allowance.dearness
employee.allowance.house_rent
employee.allowance.city

```

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most) with the member using dot operator. The following are invalid:

**employee.allowance** (actual member is missing)

**employee.house\_rent** (inner structure variable is missing)

An inner structure can have more than one variable. The following form of declaration is legal:

```

struct salary
{

 struct
 {
 int dearness;

 }
 allowance,
 arrears;
}
employee[100];

```

The inner structure has two variables, **allowance** and **arrears**. This implies that both of them have the same structure template. Note the comma after the name **allowance**. A base member can be accessed as follows:

```

employee[1].allowance.dearness
employee[1].arrears.dearness

```

We can also use tag names to define inner structures. Example:

```

struct pay
{

```

## 5.14 Computer Programming

```
 int dearness;
 int house_rent;
 int city;
 };
 struct salary
 {
 char name;
 char department;
 struct pay allowance;
 struct pay arrears;
 };
 struct salary employee[100];
```

**pay** template is defined outside the **salary** template and is used to define the structure of **allowance** and **arrears** inside the **salary** structure.

It is also permissible to nest more than one type of structures.

```
 struct personal_record
 {
 struct name_part name;
 struct addr_part address;
 struct date date_of_birth;

 };
 struct personal_record person1;
```

The first member of this structure is **name**, which is of the type **struct name\_part**. Similarly, other members have their structure types.

**NOTE:** C permits nesting up to 15 levels. However, C99 allows 63 levels of nesting.

## 5.11 POINTERS AND STRUCTURES

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variables. Suppose **product** is an array variable of **struct** type. The name **product** represents the address of its zeroth element. Consider the following declaration:

```
 struct inventory
 {
 char name[30];
 int number;
 float price;
 } product[2], *ptr;
```

This statement declares **product** as an array of two elements, each of the type **struct inventory** and **ptr** as a pointer to data objects of the type **struct inventory**. The assignment

```
 ptr = product;
```



would assign the address of the zeroth element of **product** to **ptr**. That is, the pointer **ptr** will now point to **product[0]**. Its members can be accessed using the following notation.

```
ptr -> name
ptr -> number
ptr -> price
```

The symbol  $\rightarrow$  is called the *arrow operator* (also known as *member selection operator*) and is made up of a minus sign and a greater than sign. Note that **ptr $\rightarrow$**  is simply another way of writing **product[0]**.

When the pointer **ptr** is incremented by one, it is made to point to the next record, i.e., **product[1]**. The following **for** statement will print the values of members of all the elements of **product** array.

```
for(ptr = product; ptr < product+2; ptr++)
 printf ("%s %d %f\n", ptr->name, ptr->number, ptr->price);
```

We could also use the notation

```
(*ptr).number
```

to access the member **number**. The parentheses around **\*ptr** are necessary because the member operator '**.**' has a higher precedence than the operator '**\***'.

---

### Example 5.5 Write a program to illustrate the use of structure pointers.

---

A program to illustrate the use of a structure pointer to manipulate the elements of an array of structures is shown in Fig. 5.6. The program highlights all the features discussed above. Note that the pointer **ptr** (of type **struct invent**) is also used as the loop control index in **for** loops.

```
Program
struct invent
{
 char *name[20];
 int number;
 float price;
};
main()
{
 struct invent product[3], *ptr;
 printf("INPUT\n\n");
 for(ptr = product; ptr < product+3; ptr++)
 scanf("%s %d %f", ptr->name, &ptr->number, &ptr->price);
 printf("\nOUTPUT\n\n");
 ptr = product;
 while(ptr < product + 3)
 {
 printf("%-20s %5d %10.2f\n",
 ptr->name,
 ptr->number,
 ptr->price);
 ptr++;
 }
}
```

Output

```
INPUT
Washing_machine 5 7500
```

## 5.16 Computer Programming

|                 |    |         |
|-----------------|----|---------|
| Electric_iron   | 12 | 350     |
| Two_in_one      | 7  | 1250    |
| OUTPUT          |    |         |
| Washing machine | 5  | 7500.00 |
| Electric_iron   | 12 | 350.00  |
| Two_in_one      | 7  | 1250.00 |

**Fig. 5.6** Pointer to structure variables

While using structure pointers, we should take care of the precedence of operators.

The operators ‘->’ and ‘.’, and () and [] enjoy the highest priority among the operators. They bind very tightly with their operands. For example, given the definition

```
struct
{
 int count;
 float *p; /* pointer inside the struct */
} ptr; /* struct type pointer */
```

then the statement

```
++ptr->count;
```

increments **count**, not **ptr**. However,

```
(++ptr)->count;
```

increments **ptr** first, and then links **count**. The statement

```
ptr++ -> count;
```

is legal and increments **ptr** after accessing **count**.

The following statements also behave in the similar fashion.

|                       |                                                              |
|-----------------------|--------------------------------------------------------------|
| <b>*ptr-&gt;p</b>     | Fetches whatever <b>p</b> points to.                         |
| <b>*ptr-&gt;p++</b>   | Increments <b>p</b> after accessing whatever it points to.   |
| <b>(*ptr-&gt;p)++</b> | Increments whatever <b>p</b> points to.                      |
| <b>*ptr++-&gt;p</b>   | Increments <b>ptr</b> after accessing whatever it points to. |

In the previous chapter, we discussed about passing of a structure as an argument to a function. We also saw an example where a function receives a copy of an entire structure and returns it after working on it. As we mentioned earlier, this method is inefficient in terms of both, the execution speed and memory. We can overcome this drawback by passing a pointer to the structure and then using this pointer to work on the structure members. Consider the following function:

```
print_invent(struct invent *item)
{
 printf("Name: %s\n", item->name);
 printf("Price: %f\n", item->price);
}
```

This function can be called by

```
print_invent(&product);
```

The formal argument **item** receives the address of the structure **product** and therefore it must be declared as a pointer of type **struct invent**, which represents the structure of **product**.

## 5.12 STRUCTURES AND FUNCTIONS

We know that the main philosophy of C language is the use of functions. And therefore, it is natural that C supports the passing of structure values as arguments to functions. There are three methods by which the values of a structure can be transferred from one function to another.

1. The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.
2. The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function. All compilers may not support this method of passing the entire structure as a parameter.
3. The third approach employs a concept called *pointers* to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way arrays are passed to function. This method is more efficient as compared to the second one.

In this section, we discuss in detail the second method, while the third approach using pointers is discussed in the next chapter, where pointers are dealt in detail.

The general format of sending a copy of a structure to the called function is:

```
function_name(structure_variable_name);
```

The called function takes the following form:

```
data_type function_name(struct_type st_name)
{

 return(expression);
}
```

The following points are important to note:

1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as **struct** with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same **struct** type.
3. The **return** statement is necessary only when the function is returning some data back to the calling function. The *expression* may be any simple variable or structure variable or an expression using simple variables.
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called functions must be declared in the calling function appropriately.

---

**Example 5.6** Write a simple program to illustrate the method of sending an entire structure as a parameter to a function.

---

## 5.18 Computer Programming

A program to update an item is shown in Fig. 5.7. The function **update** receives a copy of the structure variable **item** as one of its parameters. Note that both the function **update** and the formal parameter **product** are declared as type **struct stores**. It is done so because the function uses the parameter **product** to receive the structure variable **item** and also to return the updated values of **item**.

The function **mul** is of type **float** because it returns the product of **price** and **quantity**. However, the parameter **stock**, which receives the structure variable **item** is declared as type **struct stores**.

The entire structure returned by **update** can be copied into a structure of identical type. The statement

```
item = update(item, p_increment, q_increment);
```

replaces the old values of **item** by the new ones.

### Program

```
/* Passing a copy of the entire structure */
struct stores
{
 char name[20];
 float price;
 int quantity;
};
struct stores update (struct stores product, float p, int q);
float mul (struct stores stock);
main()
{
 float p_increment, value;
 int q_increment;

 struct stores item = {"XYZ", 25.75, 12};

 printf("\nInput increment values:");
 printf(" price increment and quantity increment\n");
 scanf("%f %d", &p_increment, &q_increment);

/* - - - - - */
 item = update(item, p_increment, q_increment);
/* - - - - - */
 printf("Updated values of item\n\n");
 printf("Name : %s\n", item.name);
 printf("Price : %f\n", item.price);
 printf("Quantity : %d\n", item.quantity);
/* - - - - - */
 value = mul(item);
/* - - - - - */

 printf("\nValue of the item = %f\n", value);
}
struct stores update(struct stores product, float p, int q)
{
 product.price += p;
 product.quantity += q;
 return(product);
}
```

```

 }
 float mul(struct stores stock)
 {
 return(stock.price * stock.quantity);
 }

```

**Output**

```

Input increment values: price increment and quantity increment
10 12
Updated values of item
Name : XYZ
Price : 35.750000
Quantity : 24
Value of the item = 858.000000

```

**Fig. 5.7** Using structure as a function parameter

You may notice that the template of **stores** is defined before **main()**. This has made the data type **struct stores** as *global* and has enabled the functions **update** and **mul** to make use of this definition.

**Passing Structure Through Pointers**

If the number of members in a structure object is large it is beneficial to pass the structure object by address. This method of passing fixed data movement irrespective of the size of the structure object. Since the structure object is passed by address, the changes made in the objects pointed by the formal parameters in the called function are reflected back to the calling function.

```

#include <stdio.h>
#include <string.h>

struct tag{
 /* the structure type */
 char lname[20]; /* last name */
 char fname[20]; /* first name */
 int age; /* age */
 float rate; /* e.g. 12.75 per hour */
};

struct tag my_struct; /* define the structure */
void show_name(struct tag *p); /* function prototype */

int main(void)
{
 struct tag *st_ptr; /* a pointer to a structure */
 st_ptr = &my_struct; /* point the pointer to my_struct */
 strcpy(my_struct.lname, "Jensen");
 strcpy (my_struct.fname, "Ted");
 printf("%s", my_struct.fname);
 printf("%s", my_struct.lname);
 my_struct.age = 63;
 show_name(st_ptr); /* pass the pointer */
}

```

## 5.20 Computer Programming

```
 return 0;
}

void show_name(struct tag *p)
{
 printf("%s", p->fname); /* p points to a structure */
 printf("%s", p->lname);
 printf("%d", p->age);
}
```

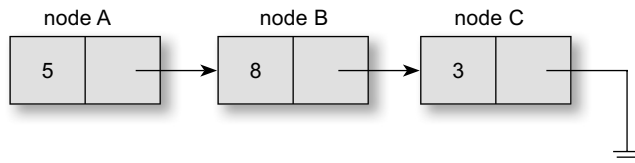
### Self Referential Structure

Self referential structure is a Structure that has a pointer to itself. Pointer stores the address of the Structure of same type.

```
struct struct_name
{
 datatype datatype_name;
 struct_name * pointer_name;
}
```

Self-Referential Structure allow to create data structures that contains references to data of the same type as themselves.

```
struct node {
 int value;
 struct node *next;
};
```



## 5.13 UNIONS

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword union as follows:

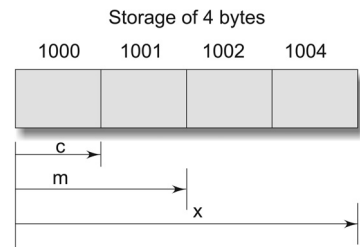
```
union item
{
 int m;
 float x;
 char c;
} code;
```

This declares a variable **code** of type **union item**. The union contains three members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above, the member **x** requires 4 bytes which is the largest among the members. Figure 5.8 shows how all the three variables share the same address. This assumes that a float variable requires 4 bytes of storage.

To access a union member, we can use the same syntax that we use for structure members. That is,

```
code.m
code.x
code.c
```



**Fig. 5.8** Sharing of a storage locating by union members

are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the statements such as

```
code.m = 379;
code.x = 7859.36;
printf("%d", code.m);
```

would produce erroneous output (which is machine dependent).

In effect, a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supersedes the previous member's value.

Unions may be used in all places where a structure is allowed. The notation for accessing a union member which is nested inside a structure remains the same as for the nested structures.

Unions may be initialized when the variable is declared. But, unlike structures, it can be initialized only with a value of the same type as the first union member. For example, with the preceding, the declaration

```
union item abc = {100};
```

is valid but the declaration

```
union item abc = {10.75};
```

is invalid. This is because the type of the first member is **int**. Other members can be initialized by either assigning values or reading from the keyboard.

## 5.14 SIZE OF STRUCTURES

We normally use structures, unions, and arrays to create variables of large sizes. The actual size of these variables in terms of bytes may change from machine to machine. We may use the unary operator **sizeof** to tell us the size of a structure (or any variable). The expression

```
sizeof(struct x)
```

will evaluate the number of bytes required to hold all the members of the structure **x**. If **y** is a simple structure variable of type **struct x**, then the expression

```
sizeof(y)
```

would also give the same answer. However, if **y** is an array variable of type **struct x**, then

```
sizeof(y)
```

would give the total number of bytes the array **y** requires.

## 5.22 Computer Programming

This kind of information would be useful to determine the number of records in a database. For example, the expression

`sizeof(y)/sizeof(x)`

would give the number of elements in the array `y`.

### 5.15 BIT FIELDS

So far, we have been using integer fields of size 16 bits to store data. There are occasions where data items require much less than 16 bits space. In such cases, we waste memory space. Fortunately, C permits us to use small *bit fields* to hold data items and thereby to pack several data items in a word of memory. Bit fields allow direct manipulation of string of a string of preselected bits as if it represented an integral quantity.

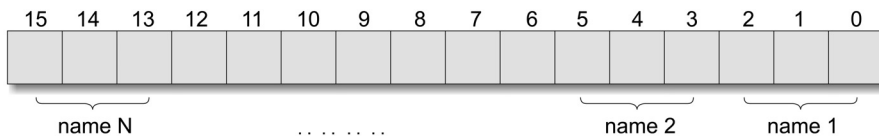
A *bit field* is a set of adjacent bits whose size can be from 1 to 16 bits in length. A word can therefore be divided into a number of bit fields. The name and size of bit fields are defined using a structure. The general form of bit field definition is:

```
struct tag-name
{
 data-type name1: bit-length;
 data-type name2: bit-length;

 data-type nameN: bit-length;
}
```

The *data-type* is either **int** or **unsigned int** or **signed int** and the *bit-length* is the number of bits used for the specified name. Remember that a **signed** bit field should have at least 2 bits (one bit for sign). Note that the field name is followed by a colon. The *bit-length* is decided by the range of value to be stored. The largest value that can be stored is  $2^{n-1}$ , where **n** is bit-length.

The internal representation of bit fields is machine dependent. That is, it depends on the size of **int** and the ordering of bits. Some machines store bits from left to right and others from right to left. The sketch below illustrates the layout of bit fields, assuming a 16-bit word that is ordered from right to left.



There are several specific points to observe:

1. The first field always starts with the first bit of the word.
2. A bit field cannot overlap integer boundaries. That is, the sum of lengths of all the fields in a structure should not be more than the size of a word. In case, it is more, the overlapping field is automatically forced to the beginning of the next word.
3. There can be unnamed fields declared with size. Example:

**Unsigned** : *bit-length*

Such fields provide padding within the word.

4. There can be unused bits in a word.
5. We cannot take the address of a bit field variable. This means we cannot use **scanf** to read values into bit fields. We can neither use pointer to access the bit fields.



6. Bit fields cannot be arrayed.
7. Bit fields should be assigned values that are within the range of their size. If we try to assign larger values, behavior would be unpredicted.

Suppose, we want to store and use personal information of employees in compressed form, this can be done as follows:

```
struct personal
{
 unsigned sex : 1
 unsigned age : 7
 unsigned m_status : 1
 unsigned children : 3
 unsigned : 4
} emp;
```

This defines a variable name **emp** with four bit fields. The range of values each field could have is follows:

| <i>Bit field</i> | <i>Bit length</i> | <i>Range of value</i>  |
|------------------|-------------------|------------------------|
| sex              | 1                 | 0 or 1                 |
| age              | 7                 | 0 or 127 ( $2^7 - 1$ ) |
| m_status         | 1                 | 0 or 1                 |
| children         | 3                 | 0 to 7 ( $2^3 - 1$ )   |

Once bit fields are defined, they can be referenced just as any other structure-type data item would be referenced. The following assignment statements are valid.

```
emp.sex = 1;
emp.age = 50;
```

Remember, we cannot use **scanf** to read values into a bit field. We may have to read into a temporary variable and then assign its value to the bit field. For example:

```
scanf("%d %d", &AGE, &CHILDREN);
emp.age = AGE;
emp.children = CHILDREN;
```

One restriction in accessing bit fields is that a pointer cannot be used. However, they can be used in normal expressions like any other variable. For example:

```
sum = sum + emp.age;
if(emp.m_status).;
printf("%d\n", emp.age);
```

are valid statements.

It is possible to combine normal structure elements with bit field elements. For example:

```
struct personal
{
 char name[20]; /* normal variable */
 struct addr address; /* structure variable */
 unsigned sex : 1;
 unsigned age : 7;


```

## 5.24 Computer Programming

```
 }
 emp[100];
```

This declares **emp** as a 100 element array of type **struct personal**. This combines normal variable name and structure type variable **address** with bit fields.

Bit fields are packed into words as they appear in the definition. Consider the following definition.

```
 struct pack
 {
 unsigned a:2;
 int count;
 unsigned b : 3;
 };
```

Here, the bit field **a** will be in one word, the variable **count** will be in the second word and the bit field **b** will be in the third word. The fields **a** and **b** would not get packed into the same word.'

## 5.16 TYPEDEF

C supports a feature known as “type definition” that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables . It takes the general form:

```
typedef type identifier;
```

Where *type* refers to an existing data type and “identifier” refers to the “new” name given to the data type. The existing data type may belong to any class of type, including the user-defined ones. Remember that the new type is ‘new’ only in name, but not the data type. **typedef** cannot create a new type. Some examples of type definition are:

```
typedef int units;
typedef float marks;
```

## 5.17 ENUM

Enum is a user-defined enumerated data type supported by ANSI C.

It is defined as follows:

```
enum identifier {value1, value2, ... valuen};
```

The “identifier” is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as *enumeration constants*). After this definition, we can declare variables to be of this ‘new’ type as below:

```
enum identifier v1, v2, ... vn;
```

The enumerated variables *v1, v2, ... vn* can only have one of the values *value1, value2, ... valuen*. The assignments of the following types are valid:

```
v1 = value3;
v5 = value1;
```

An example:

```
enum day {Monday, Tuesday, ... Sunday};
```

```
enum day week_st, week_end;
week_st = Monday;
week_end = Friday;
if(week_st == Tuesday)
week_end = Saturday;
```

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant value1 is assigned 0, value2 is assigned 1, and so on. However, the automatic assignments can be overridden by assigning values explicitly to the enumeration constants. For example:

```
enum day {Monday = 1, Tuesday, ... Sunday};
```

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement. Example:

```
enum day {Monday, ... Sunday} week_st, week_end;
```

---

**Example 5.7** Write a program to print the marks obtained by two students using structures.

---

### Program

```
/*Program for demonstrating structure member initialization*/
#include void main ()
{
 struct student/*Declaring the student structure*/
 {
 int marks1, marks2, marks3;
 };

 struct student std1 = {55,66,80};/*Initializing marks for student 1*/
 struct student std2<stdio.h>
#include <conio.h>

 = {60,85,78};/*Initializing marks for student 2*/
 clrscr();

 /*Displaying marks for student 1*/
 printf("Marks obtained by 1st student: %d, %d and %d",std1.marks1, std1.marks2,
std1.marks3);

 /*Displaying marks for student 2*/
 printf("\nMarks obtained by 2nd student: %d, %d and %d",std2.marks1, std2.marks2,
std2.marks3);

 getch();
}
```

## 5.26 Computer Programming

### Output

```
Marks obtained by 1st student: 55, 66 and 80
Marks obtained by 2nd student: 60, 85 and 78
```

---

**Example 5.8** Write a program that uses a simple structure for storing different students' details.

---

### Program

```
/*Program for creating a simple student structure*/
#include <stdio.h>
#include <conio.h>

void main ()
{
 int num, i=0;
 struct student /*Declaring the student structure*/
 {
 char name[30];
 int rollno;
 int t_marks;
 };
 struct student std[10];
 clrscr();

 /*Reading the number of students for which details are to be entered*/
 printf("Enter the number of students: ");
 scanf("%d",&num);

 /*Reading the student details*/
 for(i=0;i<num;i++)
 {
 printf("\nEnter the details for %d student",i+1);
 printf("\n\n Name ");
 scanf("%s",std[i].name);
 printf("\n Roll No. ");
 scanf("%d",&std[i].rollno);
 printf("\n Total Marks ");
 scanf("%d",&std[i].t_marks);
 }

 /*Displaying the student details*/
 printf("\n Press any key to display the student details!");
```

```

 getch();

 for(i=0;i<num;i++)
 printf("\nstudent %d \n Name %s \n Roll No. %d \n Total Marks %d\n",i+1,std[i].
name, std[i].rollno, std[i].t_marks);

 getch();
}

```

## Output

```

Enter the number of students: 2

Enter the details for 1 student

Name Ajay

Roll No. 2

Total Marks 343

Enter the details for 2 student

Name Arun

Roll No. 6

Total Marks 325

Press any key to display the student details!
student 1
Name Ajay
Roll No. 2
Total Marks 343

student 2
Name Arun
Roll No. 6
Total Marks 325

```

---

**Example 5.9** Write a simple program to demonstrate the process of defining a structure variable and assigning values to its members.

---

## Program

```

/*Program to demonstrate how to define a structure and assign values to its members*/
#include <stdio.h>
#include <conio.h>

```

## 5.28 Computer Programming

```
void main()
{
 struct personal/*Defining a structure*/
 {
 char name[20];
 int day;
 char month[10];
 int year;
 float salary;
 };

 struct personal person;/*Declaring a structure variable*/
 clrscr();

 /*Reading values for structure members*/
 printf("Enter the Values (name, day, month, year, salary): \n");
 scanf("%s %d %s %d %f",person.name,&person.day,person.month,&person.year,&person.
 salary);

 /*Displaying the values*/
 printf("%s %d %s %d, %.2f\n",person.name,person.day,person.month,person.year,person.
 salary);

 getch();
}
```

### Output

```
Enter the Values (name, day, month, year, salary):
Arun
22
9
1971
12000
Arun 22 9 1971, 12000.00 X
```

---

**Example 5.10** Write a program to realize the concept of complex numbers using structures and also print the sum of two complex number variables.

---

### Program

```
/*Program for realizing complex numbers using structures*/
#include <stdio.h>
```

```

#include <conio.h>
#include <math.h>

void main()
{
 struct complex /*Declaring the complex number datatype using structure*/
 {
 double real; /*Real part*/
 double img; /*Imaginary part*/
 };
 struct complex c1, c2, c3;
 clrscr();
 /*Reading the 1st complex number*/
 printf("\n Enter two Complex Numbers (x+iy):\n\n Real Part of First Number: ");
 scanf("%lf",&c1.real);
 printf("\n Imaginary Part of First Number: ");
 scanf("%lf",&c1.img);

 /*Reading the 2nd complex number*/
 printf("\n Real Part of Second Number: ");
 scanf("%lf",&c2.real);
 printf("\n Imaginary Part of Second Number: ");
 scanf("%lf",&c2.img);

 /*Performing complex number addition*/
 c3.real=c1.real+c2.real;
 c3.img=c1.img+c2.img;
 printf("\n\n%.2lf+(%.2lf)i + %.2lf+(%.2lf)i = %.2lf+(%.2lf)i", c1.real, c1.img,
 c2.real, c2.img, c3.real, c3.img);

 getch();
}

```

## Output

```

Enter two Complex Numbers (x+iy):

Real Part of First Number: 1

Imaginary Part of First Number: 2

Real Part of Second Number: 3

```

### 5.30 Computer Programming

Imaginary Part of Second Number: 4

$$1.00+(2.00)i + 3.00+(4.00)i = 4.00+(6.00)i$$

**Example 5.11** Write a program to read the marks obtained by a student in three subjects and compute its sum and average.

#### Program

```
/*Program for demonstrating operations on individual structure members*/
#include <stdio.h>
#include <conio.h>

void main ()
{
 struct student/*Declaring the student structure*/
 {
 int marks1, marks2, marks3,sum;
 float ave;
 };
 struct student std1;
 clrscr();

 /*Reading marks for the student*/
 printf("Enter the marks obtained by the student in three subjects: ");
 scanf("%d %d %d",&std1.marks1, &std1.marks2, &std1.marks3);

 std1.sum = std1.marks1 + std1.marks2 + std1.marks3;
 std1.ave = (std1.marks1 + std1.marks2 + std1.marks3)/3;

 /*Displaying sum and average*/
 printf("Sum = %d\nAverage = %.2f",std1.sum, std1.ave);

 getch();
}
```

#### Output

```
Enter the marks obtained by the student in three subjects: 55
65
89
Sum = 209
Average = 69.00
```



---

**Example 5.12** Implement the problem in Example 1 using pointer notation.

---

**Program**

```
/*Program for demonstrating structure member initialization using pointers*/
#include <stdio.h>
#include <conio.h>

void main ()
{
 struct student /*Declaring the student structure*/
 {
 int marks1, marks2, marks3;
 } s1, s2;
 struct student *std1, *std2; /*Declaring pointer to structure*/
 clrscr();
 std1 = &s1;
 std2 = &s2;

 /*Assigning values to structure members using pointer notation*/
 std1->marks1 = 55;
 std1->marks2 = 66;
 std1->marks3 = 80;
 std2->marks1 = 60;
 std2->marks2 = 85;
 std2->marks3 = 78;

 /*Displaying marks for student 1*/
 printf("Marks obtained by 1st student: %d, %d and %d", std1->marks1, std1->marks2,
 std1->marks3);

 /*Displaying marks for student 2*/
 printf("\nMarks obtained by 2nd student: %d, %d and %d", std2->marks1, std2->marks2,
 std2->marks3);

 getch();
}
```

**Output**

```
Marks obtained by 1st student: 55, 66 and 80
Marks obtained by 2nd student: 60, 85 and 78
```

---

**Example 5.13** Implement the problem in Example 5.11 using pointer notation.

---

## 5.32 Computer Programming

### Program

```
/*Program for demonstrating operations on individual structure members using pointer notation*/
#include <stdio.h>
#include <conio.h>

void main ()
{
 struct student /*Declaring the student structure*/
 {
 int marks1, marks2, marks3, sum;
 float ave;
 } s1;
 struct student *std1; /*Declaring pointer to structure*/
 clrscr();
 std1 = &s1;

 /*Reading marks for the student*/
 printf("Enter the marks obtained by the student in three subjects: ");
 scanf("%d %d %d", &s1.marks1, &s1.marks2, &s1.marks3);

 /*Performing arithmetic operations on structure members*/
 std1->sum = std1->marks1 + std1->marks2 + std1->marks3;
 std1->ave = (std1->marks1 + std1->marks2 + std1->marks3)/3;

 /*Displaying sum and average*/
 printf("Sum = %d\nAverage = %.2f", std1->sum, std1->ave);

 getch();
}
```

### Output

```
Enter the marks obtained by the student in three subjects: 69
89
55
Sum = 213
Average = 71.00
```

---

### Example 5.14 Implement the following student information fields using structures:

---

```
Roll No
Name (First, Middle, Last)
DOB (Day, Month, Year)
Course (Elective1, Elective2)
```

**Program**

```
/*Program for demonstrating nesting of structures*/
#include <stdio.h>
#include <conio.h>

void main ()
{
 struct student/*Declaring the nested structure*/
 {
 int roll_no;
 struct name
 {
 char First[20];
 char Middle[20];
 char Last[20];
 }st_name;
 struct dob
 {
 int day;
 int month;
 int year;
 }st_dob;
 struct course
 {
 char elective1[20];
 char elective2[20];
 }st_course;
 };
 struct student std1;
 clrscr();

 /*Initializing structure variable std1*/
 std1.roll_no=21;
 strcpy(std1.st_name.First,"Tulsi");
 strcpy(std1.st_name.Middle,"K");
 strcpy(std1.st_name.Last,"Shanta");
 std1.st_dob.day=2;
 std1.st_dob.month=2;
 std1.st_dob.year=1981;
 strcpy(std1.st_course.elective1,"Mechanics");
 strcpy(std1.st_course.elective2,"Animation");
```

### 5.34 Computer Programming

```
/*Printing the values of std1*/
printf("\nRoll No.: %d",std1.roll_no);
printf("\nName: %s %s %s",std1.st_name.First,std1.st_name.Middle,std1.st_name.Last);
printf("\nDate of Birth (DD MM YYYY): %d %d %d",std1.st_dob.day,std1.st_dob.
month,std1.st_dob.year);
printf("\nCourse Electives: %s & %s",std1.st_course.elective1,std1.st_course.elec-
tive2);

getch();
}
```

#### Output

```
Roll No.: 21
Name: Tulsi K Shanta
Date of Birth (DD MM YYYY): 2 2 1981
Course Electives: Mechanics & Animation
```

#### Example 5.15 Implement the following employee information fields using structures:

Employee ID  
Name (First, Middle, Last)  
DOJ,  
Gross Salary (HRA, BASIC, Special Allowance).

#### Program

```
/*Program for demonstrating nesting of structures*/
#include <stdio.h>
#include <conio.h>

void main ()
{
 struct employee/*Declaring the nested structure*/
 {
 int emp_id;
 struct name
 {
 char First[20];
 char Middle[20];
 char Last[20];
 }emp_name;
 char doj[20];
 struct G_Sal
 {
```

```

 float basic;
 float hra;
 float spl_allow;
}emp_sal;
};
struct employee empl;
clrscr();

/*Initializing structure variable empl*/
empl.emp_id=37;
strcpy(empl.emp_name.First,"N");
strcpy(empl.emp_name.Middle,"Siva");
strcpy(empl.emp_name.Last,"Kumar");
strcpy(empl.doj,"22/10/2004");
empl.emp_sal.basic=17432.00;
empl.emp_sal.hra=10032.00;
empl.emp_sal.spl_allow=5000.00;

/*Printing the values of empl*/
printf("\nEmp ID: %d",empl.emp_id);
printf("\nName: %s %s %s",empl.emp_name.First,empl.emp_name.Middle,empl.emp_name.
Last);
printf("\nDate of Joining (DD MM YYYY): %s",empl.doj);
printf("\nGross Salary: %.2f",empl.emp_sal.basic+empl.emp_sal.hra+empl.emp_sal.spl_
allow);

getch();
}

```

## Output

```

Emp ID: 37
Name: N Siva Kumar
Date of Joining (DD MM YYYY): 22/10/2004
Gross Salary: 32464.00

```

---

**Example 5.16** Write a simple program to demonstrate the use of unions.

---

## Program

```

/*Program for demonstrating the use of unions*/
#include <stdio.h>
#include <conio.h>

```

### 5.36 Computer Programming

```
void main ()
{
 union student/*Declaring union*/
 {
 int roll_no;
 char result;
 }st1,st2;
 clrscr();

 /*Initializing union variables*/
 st1.roll_no=20;
 st2.result='P';

 /*Accessing and printing the values correctly*/
 printf("\nRoll NO: %d",st1.roll_no);
 printf("\nResult: %c",st2.result);

 printf("\n\n");

 /*Accessing and printing the values incorrectly*/
 printf("\nRoll NO: %d",st2.roll_no);
 printf("\nResult: %c",st1.result);

 getch();
}
```

### Output

```
Roll NO: 20
Result: P

Roll NO: 12880
Result: P
```

---

**Example 5.17** Write a program to display the size of a structure variable.

---

### Program

```
/*Program for demonstrating the use of sizeof operator with structures*/
#include <stdio.h>
#include <conio.h>

void main ()
{
 struct s/*Declaring a structure*/
```

```

{
 int a;
 char b;
 float c;
 long d;
}s1;
clrscr();

/*Printing the size of structure variable s1*/
printf("\nSize of (s1) = %d",sizeof(s1));

getch();
}

```

## Output

```
Size of (s1) = 11
```

---

**Example 5.18** Write a program that uses the sizeof operator to differentiate between structures and unions.

---

## Program

```

/*Program for differentiating between structure and union*/
#include <stdio.h>
#include <conio.h>

void main ()
{
 struct s/*Declaring a structure*/
 {
 int a;
 char b;
 float c;
 long d;
 }s1;

 union u/*Declaring a union*/
 {
 int a;
 char b;
 float c;
 long d;
 }u1;
 clrscr();
}

```

### 5.38 Computer Programming

```
/*Printing the sizes of structure and union variables*/
printf("\nSize of (s1) = %d",sizeof(s1));
printf("\nSize of (u1) = %d",sizeof(u1));

getch();
}
```

#### Output

```
Size of (s1) = 11
Size of (u1) = 4
```

## 5.18 COMMAND LINE ARGUMENTS

What is a command line argument? It is a parameter supplied to a program when the program is invoked. This parameter may represent a filename the program should process. For example, if we want to execute a program to copy the contents of a file named **X\_FILE** to another one named **Y\_FILE**, then we may use a command line like

C > PROGRAM X\_FILE Y\_FILE

where **PROGRAM** is the filename where the executable code of the program is stored. This eliminates the need for the program to request the user to enter the filenames during execution. How do these parameters get into the program?

We know that every C program should have one **main** function and that it marks the beginning of the program. But what we have not mentioned so far is that it can also take arguments like other functions. In fact **main** can take two arguments called **argc** and **argv** and the information contained in the command line is passed on to the program through these arguments, when **main** is called up by the system.

The variable **argc** is an argument counter that counts the number of arguments on the command line. The **argv** is an argument vector and represents an array of character pointers that point to the command line arguments. The size of this array will be equal to the value of **argc**. For instance, for the command line given above, **argc** is three and **argv** is an array of three pointers to strings as shown below:

```
argv[0] -> PROGRAM
argv[1] -> X_FILE
argv[2] -> Y_FILE
```

In order to access the command line arguments, we must declare the main function and its parameters as follows:

```
main(int argc, char *argv[])
{

}
```

The first parameter in the command line is always the program name and therefore **argv[0]** always represents the program name.



---

**Example 5.19** Write a program that will receive a filename and a line of text as command line arguments and write the text to the file.

---

Figure 5.9 shows the use of command line arguments. The command line is

F14\_7 TEXT AAAAAA BBBB BB CCCCCC DDDDDD EEEEE E FFFFFF GGGGGG

Each word in the command line is an argument to the **main** and therefore the total number of arguments is 9.

The argument vector **argv[1]** points to the string TEXT and therefore the statement

```
fp = fopen(argv[1], "w");
```

opens a file with the name TEXT. The **for** loop that follows immediately writes the remaining 7 arguments to the file TEXT.

**Program**

```
#include <stdio.h>

main(int argc, char *argv[])
{
 FILE *fp;
 int i;
 char word[15];

 fp = fopen(argv[1], "w"); /* open file with name argv[1] */
 printf("\nNo. of arguments in Command line = %d\n\n",argc);
 for(i = 2; i < argc; i++)
 fprintf(fp,"%s ", argv[i]); /* write to file argv[1] */
 fclose(fp);

 /* Writing content of the file to screen */

 printf("Contents of %s file\n\n", argv[1]);
 fp = fopen(argv[1], "r");
 for(i = 2; i < argc; i++)
 {
 fscanf(fp,"%s", word);
 printf("%s ", word);
 }

 fclose(fp);
 printf("\n\n");

 /* Writing the arguments from memory */

 for(i = 0; i < argc; i++)
 printf("%*s \n", i*5,argv[i]);
}
```

**Output**

```
C>F14_7 TEXT AAAAAA BBBB BB CCCCC DDDDDD EEEEE FFFFFF GGGGG
```

## 5.40 Computer Programming

```
No. of arguments in Command line = 9

Contents of TEXT file

AAAAAA BBBBBB CCCCCC DDDDDD EEEEE EFFFFF GGGGGG

C:\C\F12_7.EXE
TEXT
 AAAAAA
 BBBBBB
 CCCCCC
 DDDDDD
 EEEEE
 FFFFFF
 GGGGGG
```

**Fig. 5.9** Use of command line arguments

---

**Example 5.20** Write a simple program to open and close a file; or print an error message in case of unsuccessful operation.

---

```
Program
/*Program for opening and closing a file*/
/*Source.txt file is placed at ../bin/ location*/
#include <stdio.h>
#include <conio.h>

void main()
{
 FILE *fs; /*Declaring file access pointer*/
 char ch;
 clrscr();

 fs = fopen("Source.txt", "r"); /*Opening a file*/
 if (fs == NULL)
 {
 printf("Source file cannot be opened."); /*Displaying error message in-
 case of unsuccessful opening of the file*/
 getch();
 exit(0);
 }
 else
 printf("\nFile Source.txt successfully opened");

 fclose(fs); /*Closing the file*/
 printf("\nFile Source.txt successfully closed");

 printf("\nPrint any key to end the program execution");
 getch();
}
```

**Output**

```
File Source.txt successfully opened
File Source.txt successfully closed
Print any key to end the program execution
```

**Example 5.21** Write a program to copy the contents of one file into another.

```
Program

/*Program for copying contents of one file into another*/
/*Source.txt file is placed at ../bin/ location*/
#include <stdio.h>
#include <conio.h>

void main()
{
 FILE *fs,*ft; /*Declaring file access pointers*/
 char ch;
 clrscr();

 fs = fopen("Source.txt","r"); /*Opening the source file in read mode*/
 if(fs==NULL)
 {
 printf("Source file cannot be opened.");
 exit(0);
 }
 ft = fopen("Destination.txt","w"); /*Opening the target file in write
mode*/
 if (ft==NULL)
 {
 printf("Target file cannot be opened.");
 fclose(fs);
 exit(0);
 }
 while(1)
 {
 ch=fgetc(fs); /*Reading contents of Source.txt character by character*/
 if (ch==EOF)
 break;
 else
 fputc(ch,ft); /*Copying Source.txt file contents into Destination.txt
one character at a time*/
 }
 /*Closing the files*/
 fclose(fs);
 fclose(ft);
 printf("\nFile copy operation performed successfully");
 printf("\nYou can confirm the same by checking the Destination.txt
file");
 getch();
}
```

## 5.42 Computer Programming

### Output

```
File copy operation performed successfully
You can confirm the same by checking the Destination.txt file
```

### Example 5.22 Write a program to count the number of characters in a file.

```
Program
/*Program for counting the number of characters present in a file*/
/*Source.txt file is placed at ../bin/ location*/
#include <stdio.h>
#include <conio.h>

void main()
{
 FILE *fs; /*Declaring the file access pointer*/
 char ch;
 long count=0; /*Declaring the count variable*/
 clrscr();

 fs = fopen("Source.txt","r"); /*Opening the source file*/
 if(fs==NULL)
 {
 printf("Source file cannot be opened.");
 exit(0);
 }

 while(1)
 {
 ch=fgetc(fs);
 if (ch==EOF)
 break;
 else
 count=count+1; /*Counting the number of characters present in the
source file*/
 }

 fclose(fs);
 printf("\nThe number of characters in %s is %ld", "Source.txt", count);
 getch();
}
```

### Output

```
The number of characters in Source.txt is 41
```

### Example 5.23 Write a program to read the contents of one file and write them in reverse order in another file.

```
Program
/*Program for reading file contents and printing its reverse*/
/*Source.txt file is placed at ../bin/ location*/
```

```

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

void main()
{
 FILE *fs,*ft; /*Declaring file access pointers*/
 char str[100];
 int i;
 clrscr();

 fs = fopen("Source.txt","r"); /*Opening the source file in read mode*/
 if(fs==NULL)
 {
 printf("Source file cannot be opened.");
 exit(0);
 }
 ft = fopen("Target.txt","w"); /*Opening the target file in write mode*/
 if (ft==NULL)
 {
 printf("Target file cannot be opened.");
 fclose(fs);
 exit(0);
 }

 i=0;
 while(1)
 {
 str[i]=fgetc(fs); /*Reading contents of Source.txt character by character*/
 if (str[i]==EOF)
 break;
 else
 i=i+1;
 }

 for(i=strlen(str)-2;i>=0;i--)
 fputc(str[i],ft); /*Writing the reverse of file contents*/

 /*Closing the files*/
 fclose(fs);
 fclose(ft);

 printf("\nContents of Source.txt successfully printed in reverse order in Target.txt");
 printf("\nYou can confirm the same by checking the Target.txt file");
 getch();
}

```

## Output

```

Contents of Source.txt successfully printed in reverse order in Target.
txt
You can confirm the same by checking the Target.txt file

```

---

**Example 5.24** Write a program to read a list of integers from one file and copy the same in reverse order in another file.

---

```
Program
/*Program for demonstrating the use of getw and putw*/
#include <stdio.h>
#include <conio.h>
/*
This program uses source file named S1, which contains the following elements:
1
2
3
4
5
*/

void main()
{
 FILE *fs,*ft,*fp; /*Declaring file access pointers*/
 int i,j,arr[10],num;
 clrscr();

 fs = fopen("S1","r"); /*Opening the source file in read mode*/
 if (fs==NULL)
 {
 printf("Source file cannot be opened.");
 exit(0);
 }

 ft = fopen("S2","w"); /*Opening the destination file in write mode*/
 if (ft==NULL)
 {
 printf("Target file cannot be opened.");
 fclose(fs);
 exit(0);
 }

 i=0;

 while((arr[i]=getw(fs))!=EOF) /*Reading integers from source file and
 storing them in array arr[]*/

 i=i+1;
 fclose(fs);

 for(j=i-1;j>=0;j--)
 putw(arr[j],ft); /*Copying the source integer list into target file in
 reverse order*/
 fclose(ft);

 /*Verifying the contents of the target file*/
 fp = fopen("T2","r");
 if (fp==NULL)
```

```

 {
 printf("Target file cannot be opened.");
 exit(0);
 }
 printf("Contents of the newly written file are:\n");
 while((num=getw(fp))!=EOF)
 printf("%d\n",num);
 fclose(fp);

 getch();
}

```

## Output

```

Contents of the newly written file are:
5
4
3
2
1

```

---

**Example 5.25** Write a program for reading integers from a file and writing square of those integers into another file.

---

```

Program
/*Program for demonstrating the use of fprintf and fscanf*/
/*Source.txt file is placed at ../bin/ location; assuming that it contains 10 integer values*/
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

void main()
{
 FILE *fs,*ft;/*Declaring file access pointers*/
 int c;
 clrscr();

 fs=fopen("Source.txt","r");/*Opening the source file*/
 if(fs==NULL)
 {

 printf("Cannot open source file");
 exit(1);
 }

 ft=fopen("Target.txt","w");/*Opening the target file*/
 if(ft==NULL)
 {
 printf("Cannot open target file");
 fclose(fs);
 exit(0);
 }
}

```

## 5.46 Computer Programming

```
 }
 for(;fscanf(fs,"%d",&c)!=EOF;)*Using fscanf for reading integers from
file*/
 fprintf(ft," %d ",(c*c));/*Using fprintf for writing integers to a
file*/

 printf("The square of the 10 integers contained in source file have
been computed and copied to the target file. You can open the Target.txt
file for validation");
 fclose(fs);
 fclose(ft);

 getch();
}
```

### Output

The square of the 10 integers contained in source file have been computed and copied to the target file. You can open the Target.txt file for validation

**Example 5.26** Write a program that reads list of integers from two different files and merges and stores them in a single file.

```
Program
/*Program for demonstrating the use of fseek*/
/*Source1.txt and Source2.txt files containing integer lists are placed
at ../bin/ location*/
#include<stdio.h>
#include<conio.h>

void main()
{
 FILE *fs,*ft; /*Declaring file access pointers*/
 int c;
 clrscr();

 fs=fopen("Source1.txt","r");/*Opening the 1st source file*/
 if(fs==NULL)
 {
 printf("Cannot open source file");
 exit(1);
 }

 ft=fopen("Target.txt","w");/*Opening the target file*/
 if(ft==NULL)
 {
 printf("Cannot open target file");
 fclose(fs);
 exit(0);
 }

 for(;fscanf(fs,"%d",&c)!=EOF;)*Using fscanf for reading integers from
file*/
```



```

 fprintf(ft," %d ",(c));/*Using fprintf for writing integers to a
file*/

/*Closing the file streams*/
fclose(fs);
fclose(ft);

fs=fopen("Source2.txt","r");/*Opening the 2nd source file*/
if(fs==NULL)
{
 printf("Cannot open source file");
 exit(1);
}

ft=fopen("Target.txt","r+");/*Opening the target file again*/
if(ft==NULL)
{
 printf("Cannot open target file");
 fclose(fs);
 exit(0);
}

fseek(ft,0L,2);/*Using fseek to move to the end of the file*/

for(;fscanf(fs,"%d",&c)!=EOF;){/*Reading integers from file*/
 fprintf(ft," %d ",(c));/*Writing integers to the file*/

/*Closing the file streams*/
fclose(fs);
fclose(ft);
printf("Files merged successfully; you can check the output by opening
Target.txt file");

 getch();
}

```

## Output

```
Files merged successfully; you can check the output by opening Target.
txt file
```

## Example 5.27 Write a program to open a file using command line arguments and display its contents.

```

Program
/* Program for demonstrating the use of command line arguments*/
/*Program file name: fileopen.c*/
/*Assuming source file Source.txt is placed inside the ../bin/ loca-
tion*/
#include <stdio.h>
#include <conio.h>

void main(int argc, char *argv[])
{

```

## 5.48 Computer Programming

```
char ch;
FILE *fp;

if(argc!=2)/*Checking the number of arguments given at command line*/
{
puts("Improper number of arguments.");
exit(0);
}

fp = fopen(argv[1],"r"); /*Opening the file in read mode*/
if(fp == NULL)
{
puts("File cannot be opened.");
exit(0);
}

printf("Contents of file are:\n");
while(1)
{
ch=fgetc(fp);/*Reading contents of Source.txt character by character*/
if (ch==EOF)
break;
else
printf("%c",ch); /*Displaying the file contents*/
}

fclose(fp);/*Closing the source file*/
getch();
}
```

### Output

```
D:\TC\BIN>fileopen
Improper number of arguments.

D:\TC\BIN>fileopen Source.txt
Contents of file are:
Hey! We are inside the Source.txt file...
```

## 5.19 APPLICATION OF COMMAND LINE ARGUMENTS

The key application of command line arguments is run time specification of data. That means the programmer must not statically include all the required data within the program but the same can be specified during runtime as well. Thus, the use of command line arguments creates a generalized version of an otherwise specific program.

For example, consider a situation where you are required to copy the contents of one file into another. As long as the source and target files remain the same, we can specify the names of the files statically within the program code. But, if we want the program to copy any source file contents into any target file, then we must use the concept of command line arguments that allow the end users of the program to specify the source and target file names dynamically during runtime. As we can see here, the use of command line arguments turns

the static file copy program with limited functionality into a dynamic file copy tool. Similarly, we can put the concept of command line arguments into use wherever data is required to be specified dynamically.

---

**Example 5.28** Write a program to copy the contents of one file into another using the command line arguments.

---

### Program

```
/*Program for copying contents of one file into another*/
#include <stdio.h>
#include <conio.h>

void main(int argc, char *argv[]) /*Specifying command-line arguments*/
{
 FILE *fs,*ft; /*Declaring file access pointers*/
 char ch;
 clrscr();

 if(argc!=3) /*Checking the number of arguments given at command line*/
 {
 puts("Improper number of arguments.");
 exit(0);
 }

 fs = fopen(argv[1],"r"); /*Opening the source file in read mode*/
 if(fs==NULL)
 {
 printf("Source file cannot be opened.");
 exit(0);
 }
 ft = fopen(argv[2],"w"); /*Opening the target file in write mode*/
 if (ft==NULL)
 {
 printf("Target file cannot be opened.");
 fclose(fs);
 exit(0);
 }
 while(1)
 {
 ch=fgetc(fs); /*Reading contents of source file character by character*/
 if (ch==EOF)
 break;
 else
```

```

 fputc(ch,ft);/*Copying file contents into destination file one character at a
time*/
}
/*Closing the files*/
fclose(fs);
fclose(ft);
printf("\nFile copy operation performed successfully");
printf("\nYou can confirm the same by checking the destination file");
getch();
}

```

## Output

```

Type EXIT to return to Turbo C++. . .Microsoft(R) Windows DOS
(C)Copyright Microsoft Corp 1990-2001.

D:\TC\BIN>filecopy source.txt destination.txt

File copy operation performed successfully
You can confirm the same by checking the destination file

```



## Just Remember

- Remember to place a semicolon at the end of definition of structures and unions.
- We can declare a structure variable at the time of definition of a structure by placing it after the closing brace but before the semicolon.
- Do not place the structure tag name after the closing brace in the definition. That will be treated as a structure variable. The tag name must be placed before the opening brace but after the keyword **struct**.
- When we use **typedef** definition, the *type\_name* comes after the closing brace but before the semicolon.
- We cannot declare a variable at the time of creating a **typedef** definition. We must use the *type\_name* to declare a variable in an independent statement.
- It is an error to use a structure variable as a member of its own **struct** type structure.
- Assigning a structure of one type to a structure of another type is an error.
- Declaring a variable using the tag name only (without the keyword **struct**) is an error.
- It is an error to compare two structure variables.
- It is illegal to refer to a structure member using only the member name.
- When structures are nested, a member must be qualified with all levels of structures nesting it.
- When accessing a member with a pointer and dot notation, parentheses are required around the pointer, like *(\*ptr).number*.
- The selection operator (*->*) is a single token. Any space between the symbols *-* and *>* is an error.
- When using **scanf** for reading values for members, we must use address operator **&** with non-string members.

- Forgetting to include the array subscript when referring to individual structures of an array of structures is an error.
- A union can store only one of its members at a time. We must exercise care in accessing the correct member. Accessing a wrong data is a logic error.
- It is an error to initialize a union with data that does not match the type of the first member.
- Always provide a structure tag name when creating a structure. It is convenient to use tag name to declare new structure variables later in the program.
- Use short and meaningful structure tag names.
- Avoid using same names for members of different structures (although it is not illegal).
- Passing structures to functions by pointers is more efficient than passing by value. (Passing by pointers are discussed in Chapter 12.)
- We cannot take the address of a bit field. Therefore, we cannot use **scanf** to read values in bit fields. We can neither use pointer to access the bit fields.
- Bit fields cannot be arrayed.
- Structures and unions give the programmer the liberty of creating user-defined complex data types.
- The size of a structure variable can be computed by adding the storage space of each of its member variables.
- The initialization of a structure variable can also be done at the time of its declaration by using the assignment operator '=' and placing the initial values inside braces, separated by commas.
- The uninitialized structure members are assigned the value zero for integer and float and '\0' for characters and strings.
- Nesting of structures is possible by declaring a structure inside another structure.



## Multiple Choice Questions

1. A structure stores which of the following?
  - (a) Multiple values of the same type
  - (b) Multiple values of different types
  - (c) Multiple values of the same user-defined type
  - (d) None of the above
2. Which of the following is the correct syntax for declaring a structure?
  - (a) struct  
(  
Datatype 1  
Datatype 2  
);
  - (b) struct  
{  
Datatype 1  
Datatype 2  
}
  - (c) struct  
{  
Datatype 1  
Datatype 2  
};
  - (d) struct;  
{  
Datatype 1  
Datatype 2  
}
3. Which of the following is the incorrect way of declaring structure type variables?
  - (a) struct book  
{  
Datatypes  
}book1, book2;
  - (b) struct book  
{

## 5.52 Computer Programming

- ```
Datatypes
};
Struct book book1, book2;
(c) struct book1, book2
{
    Datatype
};
(d) All of the above are correct
```
4. Which of the following is the correct way of assigning a value to the 'name' field of a structure 'book'?
- (a) book.name (b) book->name
(c) book(name) (d) None of the above
5. The uninitialized integer data type of a structure contains which of the following default values?
- (a) garbage (b) zero
(c) one (d) None of the above
6. The uninitialized character data type of a structure contains which of the following default values?
- (a) garbage (b) zero
(c) '\0' (d) None of the above
7. Which of the following expressions are correct for accessing the 'num' variable value of the ith element of a structure array 'student'?
- (a) student[i].num (b) student.num[i] (c) student[i]->num (d) None of the above
8. C allows nesting of structures till which level?
- (a) 13 (b) 14
(c) 15 (d) 16
9. In the following union declaration, what will be the size of a union variable?
- ```
union item
{
 int a;
 float b;
 char c;
} item 1;
```
- (a) 1      (b) 2  
(c) 4      (d) 8
10. Which of the following is the correct way of representing the selection statement?
- (a) ->  
(b) '  
(c) ->  
(d) All of the above are correct
11. Which of the following is the correct way of declaring a variable of bit size 1?
- (a) unsigned b:1;  
(b) unsigned b(1);  
(c) unsigned b->sizeof(1)  
(d) None of the above



## Case Study

### Book Shop Inventory

A book shop uses a personal computer to maintain the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher, stock position, etc. Whenever a customer wants a book, the shopkeeper inputs the title and author of the book and the system replies whether it is in the list or not. If it is not, an appropriate message is displayed. If book is in the list, then the system displays the book details and asks for number of copies. If the requested copies are available, the total cost of the books is displayed; otherwise the message "Required copies not in stock" is displayed.

A program to accomplish this is shown in Fig. 5.10. The program uses a template to define the structure of the book. Note that the date of publication, a member of **record** structure, is also defined as a structure.

When the title and author of a book are specified, the program searches for the book in the list using the function

**look\_up(table, s1, s2, m)**

The parameter **table** which receives the structure variable **book** is declared as type **struct record**. The parameters **s1** and **s2** receive the string values of **title** and **author** while **m** receives the total number of books in the list. Total number of books is given by the expression

**sizeof(book)/sizeof(struct record)**

The search ends when the book is found in the list and the function returns the serial number of the book. The function returns **-1** when the book is not found. Remember that the serial number of the first book in the list is zero. The program terminates when we respond "NO" to the question

Do you want any other book?

Note that we use the function

**get(string)**

to get title, author, etc. from the terminal. This enables us to input strings with spaces such as "C Language". We cannot use **scanf** to read this string since it contains two words.

Since we are reading the quantity as a string using the **get(string)** function, we have to convert it to an integer before using it in any expressions. This is done using the **atoi()** function.

**Programs**

```
#include <stdio.h>
#include <string.h>
struct record
{
 char author[20];
 char title[30];
 float price;
 struct
 {
 char month[10];
 int year;
 }
 date;
 char publisher[10];
 int quantity;
};
int look_up(struct record table[],char s1[],char s2[],int m);
void get (char string []);
main()
{
 char title[30], author[20];
 int index, no_of_records;
 char response[10], quantity[10];
 struct record book[] = {
 {"Ritche","C Language",45.00,"May",1977,"PHI",10},
 {"Kochan","Programming in C",75.50,"July",1983,"Hayden",5},
 {"Balagurusamy","BASIC",30.00,"January",1984,"TMH",0},
 {"Balagurusamy","COBOL",60.00,"December",1988,"Macmillan",25}
 };

 no_of_records = sizeof(book)/ sizeof(struct record);
```

## 5.54 Computer Programming

```
do
{
 printf("Enter title and author name as per the list\n");
 printf("\nTitle: ");
 get(title);
 printf("Author: ");
 get(author);
 index = look_up(book, title, author, no_of_records);
 if(index != -1) /* Book found */
 {
 printf("\n%s %s %.2f %s %d %s\n\n",
 book[index].author,
 book[index].title,
 book[index].price,
 book[index].date.month,
 book[index].date.year,
 book[index].publisher);

 printf("Enter number of copies:");
 get(quantity);
 if(atoi(quantity) < book[index].quantity)
 printf("Cost of %d copies = %.2f\n",atoi(quantity),
 book[index].price * atoi(quantity));
 else
 printf("\nRequired copies not in stock\n\n");
 }
 else
 printf("\nBook not in list\n\n");
 printf("\nDo you want any other book? (YES / NO):");
 get(response);
}
while(response[0] == 'Y' || response[0] == 'y');
printf("\n\nThank you. Good bye!\n");
}

void get(char string [])
{
 char c;
 int i = 0;
 do
 {
 c = getchar();
 string[i++] = c;
 }
 while(c != '\n');
 string[i-1] = '\0';
}

int look_up(struct record table[],char s1[],char s2[],int m)
{
 int i;
 for(i = 0; i < m; i++)
```



```

 if(strcmp(s1, table[i].title) == 0 &&
 strcmp(s2, table[i].author) == 0)
 return(i); /* book found */
 return(-1); /* book not found */
}

```

### Output

```

Enter title and author name as per the list
Title: BASIC
Author: Balagurusamy
Balagurusamy BASIC 30.00 January 1984 TMH
Enter number of copies:5
Required copies not in stock
Do you want any other book? (YES / NO):y
Enter title and author name as per the list
Title: COBOL
Author: Balagurusamy
Balagurusamy COBOL 60.00 December 1988 Macmillan
Enter number of copies:7
Cost of 7 copies = 420.00

Do you want any other book? (YES / NO):y
Enter title and author name as per the list
Title: C Programming
Author: Ritche

Book not in list
Do you want any other book? (YES / NO):n

Thank you. Good bye!

```

**Fig. 5.10** Program of bookshop inventory



## Review Questions

- 5.1 State whether the following statements are *true* or *false*.
- A **struct** type in C is a built-in data type.
  - The tag name of a structure is optional.
  - Structures may contain members of only one data type.
  - A structure variable is used to declare a data type containing multiple fields.
  - It is legal to copy a content of a structure variable to another structure variable of the same type.
  - Structures are always passed to functions by printers.
  - Pointers can be used to access the members of structure variables.
  - We can perform mathematical operations on structure variables that contain only numeric type members.
  - The keyword **typedef** is used to define a new data type.
  - In accessing a member of a structure using a pointer *p*, the following two are equivalent:

## 5.56 Computer Programming

(*\*p*).member\_name and *p* → member\_name

- (k) A union may be initialized in the same way a structure is initialized.
- (l) A union can have another union as one of the members.
- (m) A structure cannot have a union as one of its members.
- (n) An array cannot be used as a member of a structure.
- (o) A member in a structure can itself be a structure.

5.2 Fill in the blanks in the following statements:

- (a) The \_\_\_\_\_ can be used to create a synonym for a previously defined data type.
- (b) A \_\_\_\_\_ is a collection of data items under one name in which the items share the same storage.
- (c) The name of a structure is referred to as \_\_\_\_\_.
- (d) The selection operator → requires the use of a \_\_\_\_\_ to access the members of a structure.
- (e) The variables declared in a structure definition are called its \_\_\_\_\_.

5.3 A structure tag name **abc** is used to declare and initialize the structure variables of type **struct abc** in the following statements. Which of them are incorrect? Why? Assume that the structure **abc** has three members, **int**, **float** and **char** in that order.

- (a) struct a,b,c;
- (b) struct abc a,b,c
- (c) abc x,y,z;
- (d) struct abc a[ ];
- (e) struct abc a = { };
- (f) struct abc = b, { 1+2, 3.0, "xyz" }
- (g) struct abc c = {4,5,6};
- (h) struct abc a = 4, 5.0, "xyz";

5.4 Given the declaration

```
struct abc a,b,c;
```

which of the following statements are legal?

- (a) scanf ("%d", &a);
- (b) printf ("%d", b);
- (c) a = b;
- (d) a = b + c;
- (e) if (a>b)

. . . . .

5.5 Given the declaration

```
struct item_bank
{
 int number;
 double cost;
};
```

which of the following are correct statements for declaring one dimensional array of structures of type **struct item\_bank**?

- (a) int item\_bank items[10];
- (b) struct items[10] item\_bank;
- (c) struct item\_bank items (10);
- (d) struct item\_bank items [10];
- (e) struct items item\_bank [10];

5.6 Given the following declaration

```
typedef struct abc
{
 char x;
 int y;
 float z[10];
} ABC;
```

State which of the following declarations are invalid? Why?

- (a) struct abc v1;
- (b) struct abc v2[10];
- (c) struct ABC v3;
- (d) ABC a,b,c;
- (e) ABC a[10];

5.7 How does a structure differ from an array?

5.8 Explain the meaning and purpose of the following:

- (a) Template
- (b) **struct** keyword

(c) **typedef** keyword

(d) **sizeof** operator

(e) Tag name

5.9 Explain what is wrong in the following structure declaration:

```
struct
{
 int number;
 float price;
}
main()
{

}
```

5.10 When do we use the following?

(a) Unions

(b) Bit fields

(c) The **sizeof** operator

5.11 What is meant by the following terms?

(a) Nested structures

(b) Array of structures

Give a typical example of use of each of them.

5.12 Given the structure definitions and declarations

```
struct abc
{
 int a;
 float b;
};
struct xyz
{
 int x;
 float y;
};
abc a1, a2;
xyz x1, x2;
```

find errors, if any, in the following statements:

(a) `a1 = x1;`

(b) `abc.a1 = 10.75;`

(c) `int m = a + x;`

(d) `int n = x1.x + 10;`

(e) `a1 = a2;`

(f) `if (a.a1 > x.x1) ...`

(g) `if (a1.a < x1.x) ...`

(h) `if (x1 != x2) ...`

5.13 Describe with examples, the different ways of assigning values to structure members.

5.14 State the rules for initializing structures.

5.15 What is a 'slack byte'? How does it affect the implementation of structures?

5.16 Describe three different approaches that can be used to pass structures as function arguments.

5.17 What are the important points to be considered when implementing bit-fields in structures?

5.18 Define a structure called **complex** consisting of two floating-point numbers **x** and **y** and declare a variable **p** of type **complex**. Assign initial values 0.0 and 1.1 to the members.

5.19 What is the error in the following program?

```
typedef struct product
{
 char name [10];
 float price ;
} PRODUCT products [10];
```

5.20 What will be the output of the following program?

```
main ()
{
 union x
 {
 int a;
 float b;
 double c ;
 };
 printf("%d\n", sizeof(x));
 a.x = 10;
 printf("%d%f%f\n", a.x, b.x, c.x);
 c.x = 1.23;
 printf("%d%f%f\n", a.x, b.x, c.x);
}
```



## Programming Exercises

- 5.1 Define a structure data type called **time\_struct** containing three members integer **hour**, integer **minute** and integer **second**. Develop a program that would assign values to the individual members and display the time in the following form:  
16:40:51
- 5.2 Modify the above program such that a function is used to input values to the members and another function to display the time.
- 5.3 Design a function **update** that would accept the data structure designed in Exercise 5.1 and increments time by one second and returns the new time. (If the increment results in 60 seconds, then the second member is set to zero and the minute member is incremented by one. Then, if the result is 60 minutes, the minute member is set to zero and the hour member is incremented by one. Finally when the hour becomes 24, it is set to zero.)
- 5.4 Define a structure data type named **date** containing three integer members **day**, **month** and **year**. Develop an interactive modular program to perform the following tasks;
  - To read data into structure members by a function
  - To validate the date entered by another function
  - To print the date in the format  
April 29, 2002  
by a third function.

The input data should be three integers like 29, 4, and 2002 corresponding to day, month and year. Examples of invalid data:  
31, 4, 2002 – April has only 30 days  
29, 2, 2002 – 2002 is not a leap year
- 5.5 Design a function **update** that accepts the **date** structure designed in Exercise 5.4 to increment the date by one day and return the new date. The following rules are applicable:
  - If the date is the last day in a month, month should be incremented
  - If it is the last day in December, the year should be incremented
  - There are 29 days in February of a leap year
- 5.6 Modify the input function used in Exercise 5.4 such that it reads a value that represents the date in the form of a long integer, like 19450815 for the date 15-8-1945 (August 15, 1945) and assigns suitable values to the members **day**, **month** and **year**.  
Use suitable algorithm to convert the long integer 19450815 into year, month and day.
- 5.7 Add a function called **nextdate** to the program designed in Exercise 5.4 to perform the following task;
  - Accepts two arguments, one of the structure **date** containing the present date and the second an integer that represents the number of days to be added to the present date.
  - Adds the days to the present date and returns the structure containing the next date correctly.

Note that the next date may be in the next month or even the next year.
- 5.8 Use the **date** structure defined in Exercise 5.4 to store two dates. Develop a function that will take these two dates as input and compares them.
  - It returns 1, if the **date1** is earlier than **date2**
  - It returns 0, if **date1** is later date
- 5.9 Define a structure to represent a vector (a series of integer values) and write a modular program to perform the following tasks:
  - To create a vector
  - To modify the value of a given element
  - To multiply by a scalar value
  - To display the vector in the form (10, 20, 30, . . . . .)

- 5.10 Add a function to the program of Exercise 5.9 that accepts two vectors as input parameters and return the addition of two vectors.
- 5.11 Create two structures named **metric** and **British** which store the values of distances. The **metric** structure stores the values in meters and centimeters and the **British** structure stores the values in feet and inches. Write a program that reads values for the structure variables and adds values contained in one variable of **metric** to the contents of another variable of **British**. The program should display the result in the format of feet and inches or metres and centimetres as required.
- 5.12 Define a structure named **census** with the following three members:
- A character array city [ ] to store names
  - A long integer to store population of the city
  - A float member to store the literacy level
- Write a program to do the following:
- To read details for 5 cities randomly using an array variable
  - To sort the list alphabetically
  - To sort the list based on literacy level
- To sort the list based on population
  - To display sorted lists
- 5.13 Define a structure that can describe a hotel. It should have members that include the name, address, grade, average room charge, and number of rooms. Write functions to perform the following operations:
- To print out hotels of a given grade in order of charges
  - To print out hotels with room charges less than a given value
- 5.14 Define a structure called **cricket** that will describe the following information:
- player name  
team name  
batting average
- Using **cricket**, declare an array **player** with 50 elements and write a program to read the information about all the 50 players and print a team-wise list containing names of players with their batting average.
- 5.15 Design a structure **student\_record** to contain name, date of birth and total marks obtained. Use the **date** structure designed in Exercise 5.4 to represent the date of birth. Develop a program to read data for 10 students in a class and list them rank-wise.



## Key Terms

- **Array:** It is a collection of related data elements of same type.
- **Structure:** It is a collection of related data elements of different types.
- **Dot operator:** It is a member operator used to identify the individual members in a structure.
- **Union:** It is a collection of many members of different types.
- **Bit field:** It is a set of adjacent bits that holds data items and packs several data items in a word of memory.
- **Command line argument:** It is a parameter supplied to a program when the program is invoked.



# 6 File Management in C

---

### 6.1 INTRODUCTION

Until now we have been using the functions such as **scanf** and **printf** to read and write data. These are console oriented I/O functions, which always use the terminal (keyboard and screen) as the target place. This works fine as long as the data is small. However, many real-life problems involve large volumes of data and in such situations, the console oriented I/O operations pose two major problems.

1. It becomes cumbersome and time consuming to handle large volumes of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of *files* to store data. A file is a place on the disk where a group of related data is stored. Like most other languages, C supports a number of functions that have the ability to perform basic file operations, which include:

- naming a file,
- opening a file,
- reading data from a file,
- writing data to a file, and
- closing a file.

There are two distinct ways to perform file operations in C. The first one is known as the *low-level* I/O and uses UNIX system calls. The second method is referred to as the *high-level* I/O operation and uses functions in C's standard I/O library. We shall discuss in this chapter, the important file handling functions that are available in the C library. They are listed in Table 6.1.

**TABLE 6.1** High Level I/O Functions

| Function name | Operation                                                                    |
|---------------|------------------------------------------------------------------------------|
| fopen()       | * Creates a new file for use.                                                |
|               | * Opens an existing file for use.                                            |
| fclose()      | * Closes a file which has been opened for use.                               |
| getc()        | * Reads a character from a file.                                             |
| putc()        | * Writes a character to a file.                                              |
| fprintf()     | * Writes a set of data values to a file.                                     |
| fscanf()      | * Reads a set of data values from a file.                                    |
| getw()        | * Reads an integer from a file.                                              |
| putw()        | * Writes an integer to a file.                                               |
| fseek()       | * Sets the position to a desired point in the file.                          |
| ftell()       | * Gives the current position in the file (in terms of bytes from the start). |
| rewind()      | * Sets the position to the beginning of the file.                            |

There are many other functions. Not all of them are supported by all compilers. You should check your C library before using a particular I/O function.

## 6.2 TYPES OF FILES

As already explained, files are used for the storage and retrieval of data by a C program. Depending upon the format in which data is stored, files are primarily categorized into two types:

- **Text file** As the name suggests, a text file stores textual information like alphabets, numbers, special symbols, etc. In actuality, the ASCII code of textual characters is stored in the text files. But, since data is stored in a storage device in the binary format, the text file contents are first converted in the binary form before actually being stored in the storage device. In other words, we can say that the text files store the ASCII encrypted information. A text file can store different character sets such as:
  - Upper case English alphabets (A to Z)
  - Lowercase English alphabets (a to z)
  - Numeric characters (like 1, 3, 5, etc.)
  - Punctuation characters (like :, ; , " , ' , ? , etc.)
  - Special characters (like \$, %, etc.)

Some of the examples of text files include C source code files and files with .txt extension. C language supports various operations for manipulating data stored in a text file. Some of these operations include creating a new file, opening an existing file, reading a file, writing into the file, etc. These operations are performed with the help of inbuilt functions of C, explained later in the chapter.

- **Binary file** As the name suggests, a binary file stores the information in the binary form, i.e., in the same format as it is stored in the memory. Thus, the use of binary file eliminates the need of data conversion from text to binary format for storage purpose. However, one of the main drawbacks of binary file is that the data stored in a binary file is not in human understandable form. Any file which stores the data in the form of bytes, i.e., 8-bit representation is known as binary file. Every executable file generated by the C compiler is a binary file. Apart from .exe files, examples of binary files include video stream files, image files, etc. C language supports binary file operations such as **read**, **write** and **append** with the help of various inbuilt functions.



## 6.3 DEFINING AND OPENING A FILE

If we want to store data in a file in the secondary memory, we must specify certain things about the file, to the operating system. They include:

1. Filename.
2. Data structure.
3. Purpose.

*Filename* is a string of characters that make up a valid filename for the operating system. It may contain two parts, a *primary name* and an *optional period* with the extension. Examples:

```
Input.data
store
PROG.C
Student.c
Text.out
```

*Data structure* of a file is defined as **FILE** in the library of standard I/O function definitions. Therefore, all files should be declared as type **FILE** before they are used. **FILE** is a defined data type.

When we open a file, we must specify what we want to do with the file. For example, we may write data to the file or read the already existing data.

Following is the general format for declaring and opening a file:

```
FILE *fp;
fp = fopen("filename", "mode");
```

The first statement declares the variable **fp** as a “pointer to the data type **FILE**”. As stated earlier, **FILE** is a structure that is defined in the I/O library. The second statement opens the file named *filename* and assigns an identifier to the **FILE** type pointer **fp**. This pointer, which contains all the information about the file is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The mode does this job. Mode can be one of the following:

- r** open the file for reading only.
- w** open the file for writing only.
- a** open the file for appending (or adding) data to it.

Note that both the filename and mode are specified as strings. They should be enclosed in double quotation marks.

When trying to open a file, one of the following things may happen:

1. When the mode is ‘writing’, a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is ‘appending’, the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the purpose is ‘reading’, and if it exists, then the file is opened with the current contents safe otherwise an error occurs.

Consider the following statements:

## 6.4 Computer Programming

```
FILE *p1, *p2;
p1 = fopen("data", "r");
p2 = fopen("results", "w");
```

The file **data** is opened for reading and **results** is opened for writing. In case, the **results** file already exists, its contents are deleted and the file is opened as a new file. If **data** file does not exist, an error will occur.

Many recent compilers include additional modes of operation. They include:

- r+** The existing file is opened to the beginning for both reading and writing.
- w+** Same as **w** except both for reading and writing.
- a+** Same as **a** except both for reading and writing.

We can open and use a number of files at a time. This number however depends on the system we use.

### 6.4 CLOSING A FILE

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. In case, there is a limit to the number of files that can be kept open simultaneously, closing of unwanted files might help open the required files. Another instance where we have to close a file is when we want to reopen the same file in a different mode. The I/O library supports a function to do this for us. It takes the following form:

```
fclose(file_pointer);
```

This would close the file associated with the **FILE** pointer **file\_pointer**. Look at the following segment of a program.

```
.....
.....
FILE *p1, *p2;
p1 = fopen("INPUT", "w");
p2 = fopen("OUTPUT", "r");
.....
.....
fclose(p1);
fclose(p2);
.....
```

This program opens two files and closes them after all operations on them are completed. Once a file is closed, its file pointer can be reused for another file.

As a matter of fact all files are closed automatically whenever a program terminates. However, closing a file as soon as you are done with it is a good programming habit.

### 6.5 INPUT/OUTPUT OPERATIONS ON FILES

Once a file is opened, reading out of or writing to it is accomplished using the standard I/O routines that are listed in Table 6.1.

### 6.5.1 The *getc* and *putc* Functions

The simplest file I/O functions are **getc** and **putc**. These are analogous to **getchar** and **putchar** functions and handle one character at a time. Assume that a file is opened with mode **w** and file pointer **fp1**. Then, the statement

```
putc(c, fp1);
```

writes the character contained in the character variable **c** to the file associated with **FILE** pointer **fp1**. Similarly, **getc** is used to read a character from a file that has been opened in read mode. For example, the statement

```
c = getc(fp2);
```

would read a character from the file whose file pointer is **fp2**.

The file pointer moves by one character position for every operation of **getc** or **putc**. The **getc** will return an end-of-file marker EOF, when end of the file has been reached. Therefore, the reading should be terminated when EOF is encountered.

---

**Example 6.1** Write a program to read data from the keyboard, write it to a file called INPUT, again read the same data from the INPUT file, and display it on the screen.

---

A program and the related input and output data are shown in Fig. 6.1. We enter the input data via the keyboard and the program writes it, character by character, to the file **INPUT**. The end of the data is indicated by entering an **EOF** character, which is *control-Z* in the reference system. (This may be control-D in other systems.) The file INPUT is closed at this signal.

```
Program
#include <stdio.h>

main()
{
 FILE *f1;
 char c;
 printf("Data Input\n\n");
 /* Open the file INPUT */
 f1 = fopen("INPUT", "w");

 /* Get a character from keyboard */
 while((c=getchar()) != EOF)

 /* Write a character to INPUT */
 putc(c,f1);
 /* Close the file INPUT */
 fclose(f1);
 printf("\nData Output\n\n");
 /* Reopen the file INPUT */
 f1 = fopen("INPUT","r");

 /* Read a character from INPUT*/
 while((c=getc(f1)) != EOF)

 /* Display a character on screen */
 printf("%c",c);
```

## 6.6 Computer Programming

```
 /* Close the file INPUT */
 fclose(f1);
 }

Output

Data Input
This is a program to test the file handling
features on this system^Z

Data Output
This is a program to test the file handling
features on this system
```

**Fig. 6.1** Character oriented read/write operations on a file

The file INPUT is again reopened for reading. The program then reads its content character by character, and displays it on the screen. Reading is terminated when **getc** encounters the end-of-file mark EOF.

Testing for the end-of-file condition is important. Any attempt to read past the end of file might either cause the program to terminate with an error or result in an infinite loop situation.

### 6.5.2 The **getw** and **putw** Functions

The **getw** and **putw** are integer-oriented functions. They are similar to the **getc** and **putc** functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general forms of **getw** and **putw** are:

```
putw(integer, fp);
getw(fp);
```

Example 6.2 illustrates the use of **putw** and **getw** functions.

---

**Example 6.2** A file named DATA contains a series of integer numbers. Code a program to read these numbers and then write all 'odd' numbers to a file to be called ODD and all 'even' numbers to a file to be called EVEN.

---

The program is shown in Fig. 6.2. It uses three files simultaneously and therefore, we need to define three-file pointers **f1**, **f2** and **f3**.

First, the file DATA containing integer values is created. The integer values are read from the terminal and are written to the file **DATA** with the help of the statement

```
putw(number, f1);
```

Notice that when we type -1, the reading is terminated and the file is closed. The next step is to open all the three files, **DATA** for reading, **ODD** and **EVEN** for writing. The contents of **DATA** file are read, integer by integer, by the function **getw(f1)** and written to **ODD** or **EVEN** file after an appropriate test. Note that the statement

```
(number = getw(f1)) != EOF
```

reads a value, assigns the same to **number**, and then tests for the end-of-file mark.

Finally, the program displays the contents of ODD and EVEN files. It is important to note that the files **ODD** and **EVEN** opened for writing are closed before they are reopened for reading.

**Program**

```

#include <stdio.h>
main()
{
 FILE *f1, *f2, *f3;
 int number, i;

 printf("Contents of DATA file\n\n");
 f1 = fopen("DATA", "w"); /* Create DATA file */
 for(i = 1; i <= 30; i++)
 {
 scanf("%d", &number);
 if(number == -1) break;
 putw(number, f1);
 }
 fclose(f1);

 f1 = fopen("DATA", "r");
 f2 = fopen("ODD", "w");
 f3 = fopen("EVEN", "w");

 /* Read from DATA file */
 while((number = getw(f1)) != EOF)
 {
 if(number %2 == 0)
 putw(number, f3); /* Write to EVEN file */
 else
 putw(number, f2); /* Write to ODD file */
 }
 fclose(f1);
 fclose(f2);
 fclose(f3);

 f2 = fopen("ODD", "r");
 f3 = fopen("EVEN", "r");
 printf("\n\nContents of ODD file\n\n");
 while((number = getw(f2)) != EOF)
 printf("%4d", number);
 printf("\n\nContents of EVEN file\n\n");
 while((number = getw(f3)) != EOF)
 printf("%4d", number);

 fclose(f2);
 fclose(f3);
}

```

**Output**

```

Contents of DATA file
111 222 333 444 555 666 777 888 999 000 121 232 343 454 565 -1

```

```

Contents of ODD file
111 333 555 777 999 121 343 565

Contents of EVEN file
222 444 666 888 0 232 454

```

Fig. 6.2 Operations on integer data

### 6.5.3 The **fprintf** and **fscanf** Functions

So far, we have seen functions, that can handle only one character or integer at a time. Most compilers support two other functions, namely **fprintf** and **fscanf**, that can handle a group of mixed data simultaneously.

The functions **fprintf** and **fscanf** perform I/O operations that are identical to the familiar **printf** and **scanf** functions, except of course that they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general form of **fprintf** is

```
fprintf(fp, "control string", list);
```

where *fp* is a file pointer associated with a file that has been opened for writing. The *control string* contains output specifications for the items in the list. The *list* may include variables, constants and strings. Example:

```
fprintf(f1, "%s %d %f", name, age, 7.5);
```

Here, **name** is an array variable of type **char** and **age** is an **int** variable.

The general format of **fscanf** is

```
fscanf(fp, "control string", list);
```

This statement would cause the reading of the items in the list from the file specified by *fp*, according to the specifications contained in the *control string*. Example:

```
fscanf(f2, "%s %d", item, &quantity);
```

Like **scanf**, **fscanf** also returns the number of items that are successfully read. When the end of file is reached, it returns the value **EOF**.

---

**Example 6.3** Write a program to open a file named INVENTORY and store in it the following data:

| Item name | Number | Price | Quantity |
|-----------|--------|-------|----------|
| AAA-1     | 111    | 17.50 | 115      |
| BBB-2     | 125    | 36.00 | 75       |
| C-3       | 247    | 31.75 | 104      |

Extend the program to read this data from the file INVENTORY and display the inventory table with the value of each item.

---

The program is given in Fig. 6.3. The filename INVENTORY is supplied through the keyboard. Data is read using the function **fscanf** from the file **stdin**, which refers to the terminal and it is then written to the file that is being pointed to by the file pointer **fp**. Remember that the file pointer **fp** points to the file INVENTORY.

After closing the file INVENTORY, it is again reopened for reading. The data from the file, along with the item values are written to the file **stdout**, which refers to the screen. While reading from a file, care should be taken to use the same format specifications with which the contents have been written to the file....é

**Program**

```

#include <stdio.h>

main()
{
 FILE *fp;
 int number, quantity, i;
 float price, value;
 char item[10], filename[10];

 printf("Input file name\n");
 scanf("%s", filename);
 fp = fopen(filename, "w");
 printf("Input inventory data\n\n");
 printf("Item name Number Price Quantity\n");
 for(i = 1; i <= 3; i++)
 {
 fscanf(stdin, "%s %d %f %d",
 item, &number, &price, &quantity);
 fprintf(fp, "%s %d %.2f %d",
 item, number, price, quantity);
 }
 fclose(fp);
 fprintf(stdout, "\n\n");

 fp = fopen(filename, "r");

 printf("Item name Number Price Quantity Value\n");
 for(i = 1; i <= 3; i++)
 {
 fscanf(fp, "%s %d %f %d", item, &number, &price, &quantity);
 value = price * quantity;
 fprintf(stdout, "%-8s %7d %8.2f %8d %11.2f\n",
 item, number, price, quantity, value);
 }
 fclose(fp);
}

```

**Output**

```

Input file name
INVENTORY
Input inventory data

Item name Number Price Quantity
AAA-1 111 17.50 115
BBB-2 125 36.00 75
C-3 247 31.75 104

Item name Number Price Quantity Value
AAA-1 111 17.50 115 2012.50
BBB-2 125 36.00 75 2700.00
C-3 247 31.75 104 3302.00

```

**Fig. 6.3** Operations on mixed data types

## 6.6 ERROR HANDLING DURING I/O OPERATIONS

It is possible that an error may occur during I/O operations on a file. Typical error situations include:

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Attempting to write to a write-protected file.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output. Fortunately, we have two status-inquiry library functions; **feof** and **ferror** that can help us detect I/O errors in the files.

The **feof** function can be used to test for an end of file condition. It takes a **FILE** pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read, and returns zero otherwise. If **fp** is a pointer to file that has just been opened for reading, then the statement

```
if(feof(fp))
 printf("End of data.\n");
```

would display the message "End of data." on reaching the end of file condition.

The **ferror** function reports the status of the file indicated. It also takes a **FILE** pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise. The statement

```
if(ferror(fp) != 0)
 printf("An error has occurred.\n");
```

would print the error message, if the reading is not successful.

We know that whenever a file is opened using **fopen** function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a NULL pointer. This facility can be used to test whether a file has been opened or not. Example:

```
if(fp == NULL)
 printf("File could not be opened.\n");
```

---

**Example 6.4** Write a program to illustrate error handling in file operations.

---

The program shown in Fig. 6.4 illustrates the use of the **NULL** pointer test and **feof** function. When we input filename as TETS, the function call

```
fopen("TETS", "r");
```

returns a **NULL** pointer because the file TETS does not exist and therefore the message "Cannot open the file" is printed out.

Similarly, the call **feof(fp2)** returns a non-zero integer when the entire data has been read, and hence the program prints the message "Ran out of data" and terminates further reading.

```
Program
#include <stdio.h>

main()
{
 char *filename;
```



```

FILE *fp1, *fp2;
int i, number;

fp1 = fopen("TEST", "w");
for(i = 10; i <= 100; i += 10)
 putw(i, fp1);

fclose(fp1);

printf("\nInput filename\n");

open_file:
scanf("%s", filename);

if((fp2 = fopen(filename, "r")) == NULL)
{
 printf("Cannot open the file.\n");
 printf("Type filename again.\n\n");
 goto open_file;
}
else
for(i = 1; i <= 20; i++)
{
 number = getw(fp2);
 if(feof(fp2))
 {
 printf("\nRan out of data.\n");
 break;
 }
 else
 printf("%d\n", number);
}

fclose(fp2);
}

```

**Output**

```

Input filename
TETS
Cannot open the file.
Type filename again.

```

```

TEST
10
20
30
40
50
60
70
80

```

```

90
100

Ran out of data.
```

Fig. 6.4 Illustration of error handling in file operations

## 6.7 RANDOM ACCESS TO FILES

So far we have discussed file functions that are useful for reading and writing data sequentially. There are occasions, however, when we are interested in accessing only a particular part of a file and not in reading the other parts. This can be achieved with the help of the functions **fseek**, **ftell**, and **rewind** available in the I/O library.

**ftell** takes a file pointer and return a number of type **long**, that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form:

```
n = ftell(fp);
```

**n** would give the relative offset (in bytes) of the current position. This means that **n** bytes have already been read (or written).

**rewind** takes a file pointer and resets the position to the start of the file. For example, the statement

```
rewind(fp);
n = ftell(fp);
```

would assign **0** to **n** because the file position has been set to the start of the file by **rewind**. Remember, the first byte in the file is numbered as 0, second as 1, and so on. This function helps us in reading a file more than once, without having to close and open the file. Remember that whenever a file is opened for reading or writing, a **rewind** is done implicitly.

**fseek** function is used to move the file position to a desired location within the file. It takes the following form:

```
fseek(file_ptr, offset, position);
```

*file\_ptr* is a pointer to the file concerned, *offset* is a number or variable of type long, and *position* is an integer number. The *offset* specifies the number of positions (bytes) to be moved from the location specified by *position*. The *position* can take one of the following three values:

| Value | Meaning           |
|-------|-------------------|
| 0     | Beginning of file |
| 1     | Current position  |
| 2     | End of file       |

The offset may be positive, meaning move forwards, or negative, meaning move backwards.

Examples in Table 6.2 illustrate the operations of the **fseek** function:

**TABLE 6.2** Operations of fseek Function

| Statement       | Meaning                                                                                      |
|-----------------|----------------------------------------------------------------------------------------------|
| fseek(fp,0L,0); | Go to the beginning.<br>(Similar to rewind)                                                  |
| fseek(fp,0L,1); | Stay at the current position.<br>(Rarely used)                                               |
| fseek(fp,0L,2); | Go to the end of the file, past the last character of the file.                              |
| fseek(fp,m,0);  | Move to (m+1)th byte in the file.                                                            |
| fseek(fp,m,1);  | Go forward by m bytes.                                                                       |
| fseek(fp,-m,1); | Go backward by m bytes from the current position.                                            |
| fseek(fp,-m,2); | Go backward by m bytes from the end. (Positions the file to the mth character from the end.) |

When the operation is successful, **fseek** returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and **fseek** returns -1 (minus one). It is good practice to check whether an error has occurred or not, before proceeding further.

**Example 6.5** Write a program that uses the functions ftell and fseek.

A program employing **ftell** and **fseek** functions is shown in Fig. 6.5. We have created a file **RANDOM** with the following contents:

```
Position ----> 0 1 2 ... 25
Character stored ----> A B C ... Z
```

We are reading the file twice. First, we are reading the content of every fifth position and printing its value along with its position on the screen. The second time, we are reading the contents of the file from the end and printing the same on the screen.

During the first reading, the file pointer crosses the end-of-file mark when the parameter **n** of **fseek(fp,n,0)** becomes 30. Therefore, after printing the content of position 30, the loop is terminated.

For reading the file from the end, we use the statement

```
fseek(fp,-1L,2);
```

to position the file pointer to the last character. Since every read causes the position to move forward by one position, we have to move it back by two positions to read the next character. This is achieved by the function

```
fseek(fp, -2L, 1);
```

in the while statement. This statement also tests whether the file pointer has crossed the file boundary or not. The loop is terminated as soon as it crosses it.

```
Program
#include <stdio.h>
main()
{
 FILE *fp;
 long n;
 char c;
 fp = fopen("RANDOM", "w");
 while((c = getchar()) != EOF)
```

## 6.14 Computer Programming

```
 putc(c,fp);

 printf("No. of characters entered = %ld\n", ftell(fp));
 fclose(fp);
 fp = fopen("RANDOM","r");
 n = 0L;

 while(feof(fp) == 0)
 {
 fseek(fp, n, 0); /* Position to (n+1)th character */
 printf("Position of %c is %ld\n", getc(fp),ftell(fp));
 n = n+5L;
 }
 putchar('\n');

 fseek(fp,-1L,2); /* Position to the last character */
 do
 {
 putchar(getc(fp));
 }
 while(!fseek(fp,-2L,1));
 fclose(fp);
}
```

Output

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
No. of characters entered = 26
Position of A is 0
Position of F is 5
Position of K is 10
Position of P is 15
Position of U is 20
Position of Z is 25
Position of is 30

ZYXWVUTSRQPONMLKJIHGFEDCBA
```

**Fig. 6.5** Illustration of *fseek* and *ftell* functions

---

**Example 6.6** Write a program to append additional items to the file INVENTORY created in Example 6.3 and print the total contents of the file.

---

The program is shown in Fig. 6.6. It uses a structure definition to describe each item and a function **append()** to add an item to the file.

On execution, the program requests for the filename to which data is to be appended. After appending the items, the position of the last character in the file is assigned to **n** and then the file is closed.

The file is reopened for reading and its contents are displayed. Note that reading and displaying are done under the control of a **while** loop. The loop tests the current file position against **n** and is terminated when they become equal.

**Program**

```

#include <stdio.h>

struct invent_record
{
 char name[10];
 int number;
 float price;
 int quantity;
};

main()
{
 struct invent_record item;
 char filename[10];
 int response;
 FILE *fp;
 long n;
 void append (struct invent_record *x, file *y);
 printf("Type filename:");
 scanf("%s", filename);

 fp = fopen(filename, "a+");
 do
 {
 append(&item, fp);
 printf("\nItem %s appended.\n", item.name);
 printf("\nDo you want to add another item\
 (1 for YES /0 for NO)?");
 scanf("%d", &response);
 } while (response == 1);

 n = ftell(fp); /* Position of last character */
 fclose(fp);

 fp = fopen(filename, "r");

 while(ftell(fp) < n)
 {
 fscanf(fp, "%s %d %f %d",
 item.name, &item.number, &item.price, &item.quantity);
 fprintf(stdout, "%-8s %7d %8.2f %8d\n",
 item.name, item.number, item.price, item.quantity);
 }
 fclose(fp);
}

void append(struct invent_record *product, File *ptr)
{
 printf("Item name:");
 scanf("%s", product->name);
 printf("Item number:");
 scanf("%d", &product->number);
}

```

```

 printf("Item price:");
 scanf("%f", &product->price);
 printf("Quantity:");
 scanf("%d", &product->quantity);
 fprintf(ptr, "%s %d %.2f %d",
 product->name,
 product->number,
 product->price,
 product->quantity);
}

```

**Output**

```

Type filename:INVENTORY
Item name:XXX
Item number:444
Item price:40.50
Quantity:34
Item XXX appended.
Do you want to add another item(1 for YES /0 for NO)?1
Item name:YYY
Item number:555
Item price:50.50
Quantity:45
Item YYY appended.
Do you want to add another item(1 for YES /0 for NO)?0
AAA-1 111 17.50 115
BBB-2 125 36.00 75
C-3 247 31.75 104
XXX 444 40.50 34
YYY 555 50.50 45

```

**Fig. 6.6** Adding items to an existing file**Just Remember**

- Do not try to use a file before opening it.
- Remember, when an existing file is open using 'w' mode, the contents of file are deleted.
- When a file is used for both reading and writing, we must open it in 'w+' mode.
- EOF is integer type with a value -1. Therefore, we must use an integer variable to test EOF.
- It is an error to omit the file pointer when using a file function.
- It is an error to open a file for reading when it does not exist.
- It is an error to try to read from a file that is in write mode and vice versa.
- It is an error to attempt to place the file marker before the first byte of a file.
- It is an error to access a file with its name rather than its file pointer.
- It is a good practice to close all files before terminating a program.
- Use getw and putw for reading and writing an integer to a file.
- You must choose an appropriate file access mode depending upon the type of file operation that you want to perform.
- It is always advisable to implement an error-handling mechanism through the use of ferror function.
- Use the fseek function to move to any desired location in a file.



## Multiple Choice Questions

1. Which of the following tasks are not performed by the fopen function?
  - (a) Creating a new file which does not already exist
  - (b) Opening an existing file
  - (c) Deleting the existing file and creating a new one with the same name
  - (d) Opening an existing file in append mode
2. Which of the following is true for getc function?
  - (a) Read a string from the file
  - (b) Read a character from the file
  - (c) Read a character from the console
  - (d) Read a string from the console
3. Which of the following tasks are performed by the rewind function?
  - (a) Set the file pointer to the end of the file
  - (b) Set the file pointer to the beginning of the file
  - (c) Set the file pointer to any desired position in the file
  - (d) Erase all the file contents and place the file pointer at the beginning
4. Which of the following are not i/o functions in C?
  - (a) fscanf
  - (b) fseek
  - (c) ftel
  - (d) forward
5. What is the functionality of a+ file access mode?
  - (a) Open the file for appending data to it
  - (b) Open the file in read-write mode for appending data to it
  - (c) Append the contents of one file into another
  - (d) None of the above
6. Which of the following is true for putc?
  - (a) Put one character at a time into the file
  - (b) Put one string at a time into the file
  - (c) Put one character at a time into the console
  - (d) Put one string at a time into the console
7. Which of the following functions is used to read an integer value from a file?
  - (a) getw
  - (b) getc
  - (c) gets
  - (d) geti
8. Which of the following file-handling functions is analogous to the standard i/o function printf?
  - (a) fprintf
  - (b) fprintf
  - (c) fprintfs
  - (d) fprintfline
9. What will the function call fseek(fp,m,0) do?
  - (a) Move to the mth byte in the file
  - (b) Move to the (m-1)th byte in the file
  - (c) Move to the (m+1)th byte in the file
  - (d) Move to a location which is m positions ahead of the current file pointer position
10. Command line functions are:
  - (a) Passed at run time
  - (b) Passed at compile time
  - (c) Passed to the main function during preprocessing of the program
  - (d) Passed during dynamic memory allocation
11. End of file is represented by which of the following?
  - (a) eof
  - (b) EOF
  - (c) '/0'
  - (d) Any garbage value
12. What is the type and value of EOF?
  - (a) int, 0
  - (b) int, 1
  - (c) int, -1
  - (d) char, 1
13. Which of the following is a commonly used function for error-handling?
  - (a) ferr
  - (b) ferror
  - (c) feof
  - (d) None of the above
14. Which of the following is not a file access mode?
  - (a) r+
  - (b) w+
  - (c) a-
  - (d) a+
15. In C, files can be manipulated in which of the following modes?
  - (a) text
  - (b) int
  - (c) binary
  - (d) All of the above



## Review Questions

- 6.1 State whether the following statements are *true* or *false*.
- A file must be opened before it can be used.
  - All files must be explicitly closed.
  - Files are always referred to by name in C programs.
  - Using **fseek** to position a file beyond the end of the file is an error.
  - Function **fseek** may be used to seek from the beginning of the file only.
- 6.2 Fill in the blanks in the following statements.
- The mode \_\_\_\_\_ is used for opening a file for updating.
  - The function \_\_\_\_\_ may be used to position a file at the beginning.
  - The function \_\_\_\_\_ gives the current position in the file.
  - The function \_\_\_\_\_ is used to write data to randomly accessed file.
- 6.3 Describe the use and limitations of the functions **getc** and **putc**.
- 6.4 What is the significance of EOF?
- 6.5 When a program is terminated, all the files used by it are automatically closed. Why is it then necessary to close a file during execution of the program?
- 6.6 Distinguish between the following functions:
- getc and getchar
  - printf and fprintf
  - feof and ferror
- 6.7 How does an append mode differ from a write mode?
- 6.8 What are the common uses of **rewind** and **ftell** functions?
- 6.9 Explain the general format of **fseek** function?
- 6.10 What is the difference between the statements **rewind(fp)**; and **fseek(fp,0L,0);**?
- 6.11 Find error, if any, in the following statements:
- ```
FILE fptr;
fptr = fopen ("data", "a+");
```
- 6.12 What does the following statement mean?
- ```
FILE(*p) (void)
```
- 6.13 What does the following statement do?
- ```
While ( (c = getchar( ) != EOF
)
    putc(c, fl);
```
- 6.14 What does the following statement do?
- ```
While ((m = getw(fl)) != EOF)
 printf("%5d", m);
```
- 6.15 What does the following segment do?
- ```
. . . .
for (i = 1; i <= 5; i++ )
{
    fscanf(stdin, "%s", name);
    fprintf(fp, "%s", name);
}
. . . .
```
- 6.16 What is the purpose of the following functions?
- feof()
 - ferror()
- 6.17 Give examples of using **feof** and **ferror** in a program.
- 6.18 Can we read from a file and write to the same file without resetting the file pointer? If not, why?
- 6.19 When do we use the following functions ?
- free()
 - rewind()
- 6.20 Describe an algorithm that will append the contents of one file to the end of another file.



Programming Exercises

- 6.1 Write a program to copy the contents of one file into another.
- 6.2 Two files DATA1 and DATA2 contain sorted lists of integers. Write a program to produce a third file DATA which holds a single sorted, merged list of these two lists. Use command line arguments to specify the file names.
- 6.3 Write a program that compares two files and returns 0 if they are equal and 1 if they are not.
- 6.4 Write a program that appends one file at the end of another.
- 6.5 Write a program that reads a file containing integers and appends at its end the sum of all the integers.
- 6.6 Write a program that prompts the user for two files, one containing a line of text known as source file and other, an empty file known as target file and then copies the contents of source file into target file.
Modify the program so that a specified character is deleted from the source file as it is copied to the target file.
- 6.7 Write a program that requests for a file name and an integer, known as offset value. The program then reads the file starting from the location specified by the offset value and prints the contents on the screen.
Note: If the offset value is a positive integer, then printing skips that many lines. If it is a negative number, it prints that many lines from the end of the file. An appropriate error message should be printed, if anything goes wrong.
- 6.8 Write a program to create a sequential file that could store details about five products. Details include product code, cost and number of items available and are provided through keyboard.
- 6.9 Write a program to read the file created in Exercise 14.8 and compute and print the total value of all the five products.
- 6.10 Rewrite the program developed in Exercise 14.8 to store the details in a random access file and print the details of alternate products from the file. Modify the program so that it can output the details of a product when its code is specified interactively.



Key Terms

- **Filename:** It is a string of characters that make up a valid filename for the operating system.
- **ftell:** It is a function, which takes a file pointer and returns a number of type long that corresponds to the current position.
- **rewind:** It is a function, which takes a file pointer and resets the position to the start of the file.
- **fseek:** It is a function used to move the file position to a desired location within the file.

7

Sorting and Searching Techniques

7.1 INTRODUCTION

Information retrieval is one of the key features expected out of a computerized system. Given a telephone number getting the name and address of the subscriber, given a policy number showing the policy details are all some of the widely used retrieval practices. Basically, the process of retrieval involves searching a collection of well-organized data in a faster manner which must be capable of throwing the result almost instantaneously to the querying person. In order to be faster, the retrieval process has to be very efficient. Also, the data must be so arranged that it assists the retrieval process. The retrieval as we have been talking so far is called searching while the organizing of data is achieved through sorting. We will discuss these two aspects in this chapter.

7.2 SORTING

As we saw in the last section, the term sorting means bringing some orderliness in the data, which are otherwise not ordered. Consider a simple scenario—you are going to the railway station to board a train. The train, which you are supposed to board, is at platform 10. Assume that the railway station does not follow an ordered platform numbering scheme. Because of this the platforms are numbered in the fashion—8, 5, 2, 1, etc. Will it not take more time for you to locate the platform, which you are looking for? On the other hand, if the platform numbers are arranged like 1, 2, 3, etc. finding the 10th platform is an easier task. Thus, an ordered data makes the searching process efficient and easier. The term sorting refers to the process of bringing orderliness in the data.

Many sorting methods are advocated. Each method has its own advantages and disadvantages. Often a designer or a developer is puzzled as to which method is best and efficient. The efficiency could be measured in terms of memory usage, time taken to sort, CPU usage, etc. We are more concerned about the quickness of the algorithm. The next section answers this question.

7.2.1 Sorting Efficiency

The efficiency of sorting algorithms is measured using the O-notation (called Big O). The letter O stands for “order of”. Let us understand what we mean by this notation. Consider an algorithm, which can sort n numbers in 50 seconds on a Pentium II PC. The same algorithm can run in 40 seconds on a Pentium III PC. Here the reduction in the time is not due to an increase in the efficiency of the algorithm but due to the configurational change in the PC in which it is getting executed. Hence the reduction in time is not attributable to the efficiency of the algorithm but to that of the machine. Hence we need an altogether different scale to measure the efficiency of an algorithm. The answer is the O-notation. It is intended to measure the performance of the algorithm based on the number of inputs it receives.

Let us understand some mathematics behind the O-notation. Consider a general quadratic equation of the form $an^2 + bn + c = 0$. In this form, let us assume the values of a , b and c to be 0.5, 0.25 and 0.12 respectively. The equation now becomes:

$$f(n) = 0.5n^2 + 0.25n + 0.12$$

What will happen to this function, as n gets larger and larger? This would mean that n would contribute more to the expression yielding higher values. You can also eventually see that the term n^2 will significantly contribute to the value of $f(n)$. Hence the terms n and the constant are not very important. The O-notation depends on this.

The notation identifies the most contributing term in the expression and throws out the other terms. We categorize the O-notation based on these dominant terms to create various classes of functions to measure an algorithm's efficiency. Some of the widely used classes are $O(1)$ – Constant, $O(n)$ – linear, $O(n^k)$ —polynomial, $O(2^n)$ —Exponential and $O(\log n)$ – logarithmic. The corresponding tabular form is shown in Table 7.1.

TABLE 7.1 O-notation analysis

n	O(1)	O(n)	O(log n)	O(n log n)	O(n²)	O(n³)
1	1	1	1	1	1	1
2	1	2	1	2	4	8
4	1	4	2	8	16	64
8	1	8	3	24	64	512
16	1	16	4	64	256	4096

Let us now understand what Table 7.1 means?

- **O(1)**—this means that an algorithm will run for a constant amount of time irrespective of the size of input it receives. Whatever be the size of the input you give, the algorithm will run for the same amount of time to give the result. Some examples are push to stack, pop from stack.
- **O(n)**—this means that the amount of time the algorithm will take to complete is directly proportional to the size of the input. If the size of the input is large, the time taken to complete the process will increase. Consider a typical case of finding the sum of numbers in an array. If the array size increases, the time taken to compute the sum also increases.
- **O(n²)**—this means that the amount of time the algorithm will take to complete increases by n^2 if you increase the size of the input by n . Certain sorting algorithms (which we will see shortly) like bubble sort, insertion sort have this.

- $O(2^n)$ —this means that the amount of time the algorithm will take to complete doubles each time you increase the input.

You will agree to the fact that an algorithm with $O(\log n)$ has the best efficiency factor. However, it is always not possible to get an algorithm with such efficiency. You will understand this statement as you read the chapter.

The sorting algorithms, which we will discuss in this chapter, are divided into two categories based on the O-notation efficiency. These are $O(n^2)$ and $O(n \log n)$. As you can see we do not have a category of sorting algorithms with $O(\log n)$ efficiency. The algorithms, which we will discuss, are:

- $O(n)$ – bubble sort, insertion sort, selection sort and shell sort
- $O(n \log n)$ – heap sort, merge sort and quick sort

7.2.2 Exchange Sorting–Bubble Sort

The simplest and the oldest sorting technique is the bubble sort. However this is the most inefficient algorithm. Let us understand this method.

The method takes two elements at a time. It compares these two elements. If the first element is less than the second element, they are left undisturbed. If the first element is greater than the second element, then they are swapped. The procedure continues with the next two elements, goes and ends when all the elements are sorted. Consider the list 74, 39, 35, 97, 84.

Pass 1: (first element is compared with all other elements)

Note that the first element may change during the process.

- (1) Compare 74 and 39. Since $74 > 39$, they are swapped. The array now changes like 39, 74, 35, 97, 84.
- (2) Compare 39 and 35. Since $39 > 35$, they are swapped. The array now changes like 35, 39, 74, 97, 84.
- (3) Compare 35 and 97. Since $35 < 97$, they are undisturbed. The array is now 35, 39, 74, 97, 84.
- (4) Compare 35 and 84. Since $35 < 84$, they are undisturbed. The array is now 35, 39, 74, 97, 84.

At the end of this pass, the first element will be in the correct place. The second pass begins with the second element and is compared with all other elements. Note that the number of comparisons will be reduced by one since the first element is not compared any more.

Pass 2: (second element is compared)

- (1) Compare 39 and 74. Since $39 < 74$, they are undisturbed. The array is now 35, 39, 74, 97, 84.
- (2) Compare 39 and 97. Since $39 < 97$, they are undisturbed. The array is now 35, 39, 74, 97, 84.
- (3) Compare 39 and 84. Since $39 < 84$, they are undisturbed. The array is now 35, 39, 74, 97, 84.

This pass did not bring any change in the array elements. The next pass starts.

Pass 3: (third element is compared)

- (1) Compare 74 and 97. Since $74 < 97$, they are undisturbed. The array is now 35, 39, 74, 97, 84.
- (2) Compare 74 and 84. Since $74 < 84$, they are undisturbed. The array is now 35, 39, 74, 97, 84.

This pass also did not bring any change in the array elements. The next pass starts.

Pass 4: (fourth element is compared)

- (1) Compare 97 and 84. Since $97 > 84$, they are swapped. The array is now 35, 39, 74, 84, 97.

The sorting ends here. Note that the last number is not compared with any more numbers.

7.4 Computer Programming

1. C program for bubble sort

Let us assume that an array named `arr[]` will hold the array to be sorted and `n` is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature. The required C program is shown in Fig. 7.1.

```
#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500
void bubblesort(int arr[], int maxsize);
int arr[n],n;
int main()
{
    int i;
    printf("\nHow many arr you want to sort: ");
    scanf("%d",&n);

    printf("\nEnter the values one by one: ");
    for (i = 0; i < n; i++)
    {
        printf ("\nEnter element %i :",i);
        scanf("%d",&arr[i]);
    }
    printf("\nArray before sorting:\n");
    for (i = 0; i < n; i++)
        printf("[%i], ",arr[i]);
    printf ("\n");
    bubblesort(arr, n);
    printf("\nArray after sorting:\n");
    for (i = 0; i < n; i++)
        printf("[%i], ", arr[i]);
    }
void bubblesort(int arr[], int n)
{
    int i, j, temp;
    for (i = 0; i < n - 1; i++)
    {
        for (j = i; j < n; j++)
        {
            if (arr[i] > arr[j])
            {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

Fig. 7.1 Bubble sort–C program

At the end of the execution, the array `arr[]` will be sorted in the ascending order. If you need to perform a descending order sorting, replace the `>` symbol with `<` symbol in the comparison statement.

2. Efficiency of bubble sort

Let us now analyze the efficiency of the bubble sort. You can see from the algorithm that each time (each pass), the number of elements scanned for comparison reduces by 1. Thus, we have:

Number of comparisons in the first pass = $(n - 1)$

Number of comparisons in the second pass = $(n - 2)$

Number of comparisons in the last pass = 1

Thus, the total number of comparisons at the end of the algorithm would have been:

$$(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1) / 2 = n^2 / 2 + O(n) = O(n^2)$$

Hence the order of the bubble sort algorithm is $O(n^2)$. A sample output showing the result of the program is given below:

```
How many elements you want to sort: 5
```

```
Enter the values one by one:
```

```
Enter element 0 :74
```

```
Enter element 1 :39
```

```
Enter element 2 :35
```

```
Enter element 3 :97
```

```
Enter element 4 :84
```

```
Array before sorting:
```

```
[74], [39], [35], [97], [84],
```

```
Array after sorting:
```

```
[35], [39], [74], [84], [97],
```

7.2.3 Exchange Sorting–Quick Sort

This method, invented by Hoare, is considered to be a fast method to sort the elements. The method is also called partition exchange sorting. The method is based on divide-and-conquer technique, i.e. the entire list is divided into various partitions and sorting is applied again and again on the partitions.

In this method, the list is divided into two, based on an element called the pivot element. Usually, the first element is considered to be the pivot element. Now, move the pivot element into its correct position in the list. The elements to the left of the pivot are less than the pivot while the elements to the right of the pivot are greater than the pivot. The process is reapplied to each of these partitions. This process proceeds till we get the sorted list of elements. Let us understand by an example. Consider the list 74, 39, 35, 32, 97, 84.

- (1) We will take 74 as the pivot and move it to position so that the new list becomes—39, 35, 32, 74, 97, 84. Note that the elements to the left are lesser than 74. Also, these elements (39, 35 and 32) are yet to be sorted. Similarly, the elements 97 and 84, which are right to the pivot, are yet to be sorted.
- (2) Now take the partitioned list—39, 35, 32. Let us take 39 as the pivot. Moving it to the correct position gives—35, 32, 39. The partition to the left is 35, 32. There is no right partition. Reapply the process on the partition, we get 32, 35. Thus, we have 32, 35, 39.

7.6 Computer Programming

- (3) Apply the process to the right partition of 74 which is 97, 84. Taking 97 as the pivot and positioning it, we get 74, 84, 97.
- (4) Assembling all the elements from each partition, we get the sorted list.

1. C program for quick sort

The quick sort is a very good example for recursive programming. Let us assume that an array named `elements[]` will hold the array to be sorted and `maxsize` is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature. The required C program is shown in Fig. 7.2.

```
#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500
void quickSort(int elements[], int maxsize);
void sort(int elements[], int left, int right);
int elements[MAXSIZE];
int main()
{
    int i, maxsize;
    printf("\nHow many elements you want to sort: ");
    scanf("%d",&maxsize);
    printf("\nEnter the values one by one: ");
    for (i = 0; i < maxsize; i++)
    {
        printf ("\nEnter element %i :",i);
        scanf("%d",&elements[i]);
    }
    printf("\nArray before sorting:\n");
    for (i = 0; i < maxsize; i++)
        printf("[%i], ",elements[i]);
    printf ("\n");
    quickSort(elements, maxsize);
    printf("\nArray after sorting:\n");
    for (i = 0; i < maxsize; i++)
        printf("[%i], ", elements[i]);
    }
void quickSort(int elements[], int maxsize)
{
    sort(elements, 0, maxsize - 1);
}
void sort(int elements[], int left, int right)
{
    int pivot, l, r;
    l = left;
    r = right;
    pivot = elements[left];
    while (left < right)
    {
        while ((elements[right] >= pivot) && (left < right))
            right--;
        if (left != right)
```



```

        {
            elements[left] = elements[right];
            left++;
        }
        while ((elements[left] <= pivot) && (left < right))
            left++;
        if (left != right)
        {
            elements[right] = elements[left];
            right--;
        }
        elements[left] = pivot;
        pivot = left;
        left = l;
        right = r;
        if (left < pivot)
            sort(elements, left, pivot - 1);
        if (right > pivot)
            sort(elements, pivot + 1, right);
    }

```

Fig. 7.2 Quicksort C program

A sample output showing the result of the program is given below:

```

How many elements you want to sort: 6

Enter the values one by one:
Enter element 0 :74

Enter element 1 :39

Enter element 2 :32

Enter element 3 :35

Enter element 4 :97

Enter element 5 :84

Array before sorting:
[74], [39], [32], [35], [97], [84],

Array after sorting:
[32], [35], [39], [74], [84], [97],

```

2. Efficiency of quick sort

Let us now analyze the efficiency of the quick sort. The selection of the pivot plays a vital role in determining the efficiency of the quick sort. The main reasons for this are—the pivot may partition the list into two so that one partition is much bigger than the other and the partition may be in an unsorted fashion (which is possible

7.8 Computer Programming

to the maximum). We will analyze the quick sort from the comparison point of view. We will consider two possible cases here.

Case I

In this case we will make two vital assumptions. These are:

1. The pivot, which we choose, will always be swapped into exactly the middle of the partition. To put it simple, the pivot (after swapping into its correct position) will have an equal number of elements both to its left and right.
2. The number of elements in the list is a power of 2. This means, if there are x elements in the array, we say that $x = 2^y$. This can be rewritten as $y = \log_2 x$.

In this situation, the first of the pass will have x comparisons. When the first pass is completed, the list will be partitioned into two equal halves, each with $x/2$ elements (which is obvious since we assumed that the pivot will partition the list exactly into two). In the next pass, we will have four partitions, each with equal number of elements (which will be $x/4$). Proceeding in this way, we will get the following table.

TABLE 7.2

Pass	Number of comparisons
1	x
2	$2 * (x / 2)$
3	$4 * (x / 4)$
4	$8 * (x / 8)$
X	$X * (x / x)$

Thus, the total number of comparisons would be $O(x) + O(x) + \dots + y$ terms. This is equivalent to $O(x * y)$. Let us substitute $y = \log_2 x$, so that we get $O(x \log x)$. Thus the efficiency of quick sort is $O(x \log x)$.

Case II

Let us now forgo one of our assumptions, which we made earlier. We will assume that the pivot partitions the list into two so that one of the partitions has no elements while the other has all the other elements. This is the worst case possible. Here, the total number of comparisons at the end of the sort would have been:

$(x - 1) + (x - 2) + \dots + 2 + 1 = \frac{1}{2}(x - 1) * x = \frac{1}{2}(x^2) - \frac{1}{2}(x) = O(x^2)$. Thus, the efficiency of the quick sort in its worst case is $O(x^2)$.

7.2.4 Selection Sort

We will discuss the straight selection sort (also called push-down sorting) in this section. As the name suggests, the first element of the list is selected. It is compared repeatedly with all the elements. If any element is found to be lesser than the selected element, these two are swapped. This procedure is repeated till the entire array is sorted. Let us understand this with a simple example. Consider the list 74, 39, 35, 32, 97, 84. Table 7.3 gives the status of the list after each pass is completed.

TABLE 7.3

Pass	List after pass
1	32, 39, 35, 74, 97, 84
2	32, 35, 39, 74, 97, 84
3	32, 35, 39, 74, 97, 84
4	32, 35, 39, 74, 97, 84
5	32, 35, 39, 74, 84, 97

1. C program for selection sort

Let us assume that an array named `elements[]` will hold the array to be sorted and `maxsize` is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature. The required C program is shown in Fig. 7.3.

```

#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500
void selection(int elements[], int maxsize);
int elements[MAXSIZE],maxsize;
int main()
{
    int i;
    printf("\nHow many elements you want to sort: ");
    scanf("%d",&maxsize);

    printf("\nEnter the values one by one: ");
    for (i = 0; i < maxsize; i++)
    {
        printf ("\nEnter element %i :",i);
        scanf("%d",&elements[i]);
    }

    printf("\nArray before sorting:\n");
    for (i = 0; i < maxsize; i++)
        printf("[%i], ",elements[i]);

    printf ("\n");
    selection(elements, maxsize);

    printf("\nArray after sorting:\n");
    for (i = 0; i < maxsize; i++)
        printf("[%i], ", elements[i]);
}

void selection(int elements[], int array_size)
{
    int i, j, k;
    int min, temp;
    for (i = 0; i < maxsize-1; i++)
    {
        min = i;
        for (j = i+1; j < maxsize; j++)

```

```

        {
            if (elements[j] < elements[min])
            min = j;
        }
        temp = elements[i];
        elements[i] = elements[min];
        elements[min] = temp;
    }
}

```

Fig. 7.3 C program for selection sorting

2. Efficiency of selection sort

Let us now analyze the efficiency of the selection sort. You can easily understand from the algorithm that the first pass of the program does $(\text{maxsize} - 1)$ comparisons. The next pass does $(\text{maxsize} - 2)$ comparisons and so on. Thus, the total number of comparisons at the end of the sort would be :

$$(\text{maxsize} - 1) + (\text{maxsize} - 2) + \dots + 1 = \text{maxsize} * (\text{maxsize} - 1) / 2 = O(\text{maxsize}^2)$$

Note that this efficiency is same as the worst case of quick sort.

7.2.5 Merge Sort

The merge sort technique sorts a given set of values by combining two sorted arrays into one larger sorted array.

Consider the sorted array, **A**, which contains **p** elements, and the sorted array **B**, containing **q** elements. The merge sort technique combines the elements of **A** and **B** into a single sorted array **C** with **p + q** elements.

The total number of comparisons in the merge sort technique to sort n data-items of an array is $\log n$. The merge sort technique requires at most $\log n$ passes, so the complexity of the merge sort is $O(n \log n)$. Consider an example, to merge two sorted arrays into a single sorted array by using the merge sort technique. The data items stored in an array *A*, are:

$$A = \{56, 78\}$$

The data items stored in an array, *B*, are:

$$B = \{45, 67, 89\}$$

The data items of the *C* array after merging the two sorted arrays, *A* and *B*, are:

$$C = \{45, 56, 67, 78, 89\}$$

The first data item of the *A* array is compared with the first data item of the *B* array. If the first data item of *A* is smaller than the first data item of *B*, then that data item from *A* is moved to the new array, *C*. If the data item of *B* is smaller than the data item of *A*, then it is moved to the array, *C*. This comparing of data items continues until one of the arrays ends.

The following code shows the implementation of merge sort technique.

```

#include<stdio.h>
#include<conio.h>
void smerge_sort(int *a, int *b, int *c, int n1, int n2, int *n);
void main()

```

```

{
    int n1=0, n2=0, n=0, i=0;
    int a[30], b[30], c[60];
    clrscr();
    printf("\nEnter two sorted arrays\n");
    printf("\n\t\tDetails of 1st array:");
    printf("\nNumber of elements n1: ");
    scanf("%d",&n1);
    printf("Enter elements: ");
    for(i=0;i<n1;i++)
        scanf("%d",&a[i]);
    printf("\n\t\tDetails of 2nd array:");
    printf("\nNumber of elements n2: ");
    scanf("%d",&n2);
    printf("Enter elements: ");
    for(i=0;i<n2;i++)
        scanf("%d",&b[i]);
    smerge_sort(a, b, c, n1, n2, &n);
    printf("\nResultant array after merge sort.\n");
    for(i=0;i<n;i++)
        printf("%d ",c[i]);
    getch();
}

void smerge_sort(int *a, int *b, int *c, int n1, int n2, int *n)
{
    int i=0, j=0, k=0;
    while(i<n1 && j<n2)
    {
        if(a[i]<b[j])
        {
            c[k]=a[i];
            i++;
        }
        else
        {
            c[k]=b[j];
            j++;
        }
        k++;
    }
    while(i<n1)
    {
        c[k]=a[i];
        i++;
        k++;
    }
    while(j<n2)
    {
        c[k]=b[j];
        j++;
        k++;
    }
    *n=k;
}

```

Fig. 7.4 C program for merge sort

7.12 Computer Programming

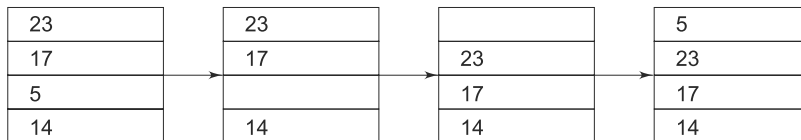
The output of the above code is:

```
Enter two sorted arrays
    Details of 1st array:
Number of elements n1:5
Enter elements: 5 10 14 19 30
    Details of 2nd array:
Number of elements n2: 6
Enter elements: 2 23 28 44 49 60
Resultant array after merge sort.
2 5 10 14 19 23 28 30 44 49 60
```

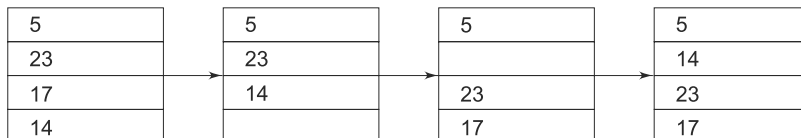
7.2.6 Simple Insertion Sort

We will discuss the insertion sort in this section. An example of an insertion sort occurs while playing cards. To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence. This is illustrated with a simple example in Fig. 7.5. Consider the list – 23, 17, 5, 14.

Stage 1



Stage 2



Stage 1

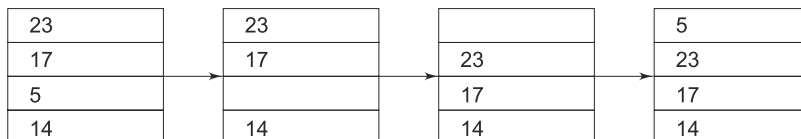


Fig. 7.5 Simple insertion sort

Look at how the sorting proceeds in various stages. At each stage pushing the remaining elements down creates an empty space.

1. C program for insertion sort

Let us assume that an array named `elements[]` will hold the array to be sorted and `maxsize` is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature. The required C program is shown in Fig. 7.6.

```

#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500
void insertionsort(int elements[], int maxsize);
int elements[MAXSIZE],maxsize;
int main()
{
    int i;
    printf("\nHow many elements you want to sort: ");
    scanf("%d",&maxsize);

    printf("\nEnter the values one by one: ");
    for (i = 0; i < maxsize; i++)
    {
        printf ("\nEnter element %i :",i);
        scanf("%d",&elements[i]);
    }

    printf("\nArray before sorting:\n");
    for (i = 0; i < maxsize; i++)
        printf("[%i], ",elements[i]);
    printf ("\n");
    insertionsort(elements, maxsize);
    printf("\nArray after sorting:\n");
    for (i = 0; i < maxsize; i++)
        printf("[%i], ", elements[i]);
}
void insertionsort(int elements[], int maxsize)
{
    int i, j, index;
    for (i=1; i < maxsize; i++)
    {
        index = elements[i];
        j = i;
        while ((j > 0) && (elements[j-1] > index))
        {
            elements[j] = elements[j-1];
            j = j - 1;
        }
        elements[j] = index;
    }
}

```

Fig. 7.6 C program for insertion sort

2. Efficiency of insertion sort

Let us now analyze the efficiency of the selection sort. Assuming there are n elements in the array, we must pass through $n - 1$ entry. For each entry, we may need to examine and shift a maximum of $n - 1$ other entries, resulting in a efficiency of $O(n^2)$.

7.2.7 Shell Sort

We will discuss the shell sort in this section. The shell sort is based on the insertion sort. The given list of elements is broken down in few smaller lists. After this, the insertion sort is applied on such smaller lists.

7.14 Computer Programming

These smaller lists are again recombined. Another partition is made and the procedure is repeated. The breaking down of the list into smaller lists is by selecting elements at specific locations. This is determined by a factor called increment. Consider the list – 23, 12, 6, 34, 13, 7, 44. We may initially take the increment as 3. This would break the list as 23, 34, 44; 12, 13 and 6, 7. The first list is done using elements[0], elements[3] and elements[6] where the increment is 3. This list is sorted using insertion sort. The next list is elements[1] and elements[4] where the increment is the same 3. The final list is elements[2] and elements[5] with the same increment of 3. After sorting them, the sublists are recombined.

As a next stage, the increment is changed (reduced by a suitable amount) and the process is repeated. The process continues till the increment is 1. The choice of increment is left to our choice. We can have increments such as 7, 5, 3 and 1 or 6, 4, 1, etc.

1. C program for shell sort

Let us assume that an array named elements[] will hold the array to be sorted and maxsize is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature. The required C program is shown in Fig. 7.7. Note that we have taken the increments as 7, 2 and 1. The successive increments are deduced by the formula $\text{increment} / 2$.

```
#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500
void shellsort(int elements[], int maxsize);
int elements[MAXSIZE], maxsize;
int main()
{
    int i;
    printf("\nHow many elements you want to sort: ");
    scanf("%d", &maxsize);
    printf("\nEnter the values one by one: ");
    for (i = 0; i < maxsize; i++)
    {
        printf("\nEnter element %i :", i);
        scanf("%d", &elements[i]);
    }
    printf("\nArray before sorting:\n");
    for (i = 0; i < maxsize; i++)
        printf("[%i], ", elements[i]);
    printf("\n");
    shellsort(elements, maxsize);
    printf("\nArray after sorting:\n");
    for (i = 0; i < maxsize; i++)
        printf("[%i], ", elements[i]);
}
void shellsort(int elements[], int maxsize)
{
    int i, j, increment, temp;
    increment = 7;
    while (increment > 0)
```



```

{
for (i=0; i < maxsize; i++)
{
j = i;
temp = elements[i];
while ((j >= increment) && (elements[j-increment] > temp))
{
elements[j] = elements[j - increment];
j = j - increment;
}
elements[j] = temp;
}
if (increment/3 != 0)
increment = increment/3;
else if (increment == 1)
increment = 0;
else
increment = 1;
}
}

```

Fig. 7.7 C program for shell sort

7.3 SEARCHING

All of us use searching in one way or the other in our daily life. Any information gathering process or retrieval process is basically a searching process. We already saw that searching is more efficient if the data is available in an ordered fashion (sorted). We saw in the last few sections some of the widely used methods of sorting. Let us now look into two of the widely used search methods—the linear search and binary search techniques.

7.3.1 Linear Search

The linear search or the sequential searching is most simple searching method. It does not expect the list to be sorted also. The key, which is to be searched, is compared with each element of the list one by one. If a match exists, the search is terminated. If the end of the list is reached, it means that the search has failed and the key has no matching element in the list.

1. C program for linear search

Figure 7.8 shows the C program for linear search.

```

#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500
int linearsearch(int elements[], int maxsize, int key);
int elements[MAXSIZE], maxsize, key;
int main()
{
int i, pos;
printf("\nHow many elements you have in the list: ");
scanf("%d",&maxsize);
printf("\nEnter the key to be searched: ");

```

```

scanf("%d",&key);
printf("\nEnter the values one by one: ");
for (i = 0; i < maxsize; i++)
{
    printf ("\nEnter element %i :",i);
    scanf("%d",&elements[i]);
}
pos = -1;
pos = linearsearch(elements, maxsize, key);
if (pos != -1)
    printf("\nThe key %d is found at position %d", key, pos);
else
    printf("\nThe key is not found in the list");
}
int linearsearch(int elements[], int maxsize, int key)
{
    int i, j, temp;
    for (i = 0; i < maxsize - 1; i++)
    {
        if (key == elements[i])
            return i;
    }
    return -1;
}

```

Fig. 7.8 C program for linear search

2. Efficiency of linear search

To find the number of key comparisons for a successful match, we can add the number required for each comparison and divide by the total number of elements in the list. This would give:

$$\frac{1 + 2 + 3 + \dots + n}{n} = \frac{n(n+1)}{2n}$$

7.3.2 Binary Search

The binary search is yet another simple searching method. However, in the binary search, the algorithm expects the list to be sorted. You can apply any one of the sorting methods before using the binary search algorithm.

How does the algorithm operates? The given list is divided into two equal halves. The given key is compared with the middle element of the list. Now three situations may occur:

- (a) The middle element matches with the key—the search will end peacefully here.
- (b) The middle element is greater than the key—then the value which we are searching is (possibly) in the first half of the list
- (c) The middle element is lower than the key—then the value which we are searching is (possibly) in the second half of the list

Now the list is searched either in the first half or in the second half as got by the result of conditions (b) and (c) given above. This process is repeated till we get the key or the search fails because the key does not exist.

1. C program for iterative binary search

Figure 7.9 gives the C program for iterative binary search.

```
#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500

int binsearch(int elements[], int maxsize, int key);
void bubblesort(int elements[], int maxsize);
int elements[MAXSIZE], maxsize, key;

int main()
{
    int i, pos;
    printf("\nHow many elements you have in the list: ");
    scanf("%d",&maxsize);
    printf("\nEnter the key to be searched: ");
    scanf("%d",&key);

    printf("\nEnter the values one by one: ");
    for (i = 0; i < maxsize; i++)
    {
        printf ("\nEnter element %i :",i);
        scanf("%d",&elements[i]);
    }

    bubblesort(elements, maxsize);
    pos = -1;
    pos = binsearch(elements, maxsize, key);
    if (pos != -1)
        printf("\nThe key %d is found at position %d", key, pos);
    else
        printf("\nThe key is not found in the list");
}

void bubblesort(int elements[], int maxsize)
{
    int i, j, temp;
    for (i = 0; i < maxsize - 1; i++)
    {
        for (j = i; j < maxsize; j++)
        {
            if (elements[i] > elements[j])
            {
                temp = elements[i];
                elements[i] = elements[j];
                elements[j] = temp;
            }
        }
    }
}

int binsearch(int elements[], int maxsize, int key)
{
    int i, first, middle, last;
    first = 0;
    last = maxsize - 1;
```

```

while (last >= first)
{
    middle = (first + last ) / 2;
    if (key > elements[middle])
        first = middle + 1;
    else if (key < elements[middle])
        last = middle - 1;
    else
        return middle;
}
return -1;
}

```

Fig. 7.9 C program for iterative binary search

In the above program, a bubble sort function is also added to get a sorted list before applying binary search on the list. Instead of the bubble sort, you can adopt any other sort routine, which we have already discussed in previous sections.

In the program, you can infer that we have used the variable ‘middle’ to locate our key. All the comparisons are based on the element, which is at the position pointed by the variable ‘middle’. This variable is altered continuously till we get the result. If the search fails, the function returns a negative value.

2. C program for recursive binary search

We can write a recursive version of the binary search also very easily. Figure 7.10 gives program for such a version of binary search.

```

#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500
int binsearch(int elements[], int maxsize, int key, int first,
int last);
void bubblesort(int elements[], int maxsize);
int elements[MAXSIZE], maxsize, key;
int main()
{
    int i, pos;
    printf("\nHow many elements you have in the list: ");
    scanf("%d",&maxsize);
    printf("\nEnter the key to be searched: ");
    scanf("%d",&key);
    printf("\nEnter the values one by one: ");
    for (i = 0; i < maxsize; i++)
    {
        printf ("\nEnter element %i :",i);
        scanf("%d",&elements[i]);
    }
    bubblesort(elements, maxsize);
    pos = -1;
    pos = binsearch(elements, maxsize, key, 0, maxsize - 1);
    if (pos != -1)

```

```

        printf("\nThe key %d is found at position %d", key, pos);
    else
        printf("\nThe key is not found in the list");
    }
void bubblesort(int elements[], int maxsize)
{
    int i, j, temp;
    for (i = 0; i < maxsize - 1; i++)
    {
        for (j = i; j < maxsize; j++)
        {
            if (elements[i] > elements[j])
            {
                temp = elements[i];
                elements[i] = elements[j];
                elements[j] = temp;
            }
        }
    }
}
int binsearch(int elements[], int maxsize, int key, int first,
int last)
{
    int i, middle;
    middle = -1;
    if (first <= last)
    {
        middle = (first + last) / 2;
        if (key > elements[middle])
            middle = binsearch(elements, maxsize, key, middle + 1, last);
        else if (key < elements[middle])
            middle = binsearch(elements, maxsize, key, first, middle - 1);
    }
    return middle;
}

```

Fig. 7.10 C program for recursive binary search

7.3.3 Indexed Sequential Search

Yet another searching technique is the indexed sequential search. This is widely employed in various file handling and database systems. The method is based on a “short representation” of the list. This “short representation” is called as index. How this index is formed? Many ways do exist to form this index. One of the widely used ways is to extract every eighth element from the list and make that as a part of the index.

Whenever a particular key is to be searched, it is first searched in the index. The appropriate location from the index is taken and that location is sequentially searched. Figure 7.11 explains this scenario. In the figure, every fifth record is taken as an example.

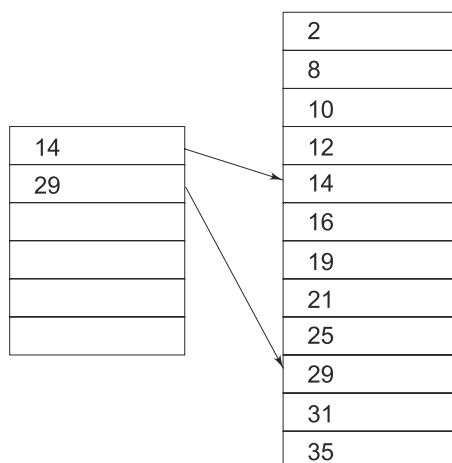


Fig. 7.11 Indexed sequential search

In Fig. 7.11, assume that we are searching for 16. The index is first searched. Note that you may not get an exact match for the key in the index. You will get only the range, which you may use for searching the main list. In this example, we get the range that 16 is between 14 and 29. You may perform a sequential search in this range to get at the key value.

Example 7.1 Program to implement a simple hash function in C.

Program

```

/*Program for implementing a Hash function*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int hash(int); /*Defining the hash function*/
    int k;
    clrscr();

    printf("\n\n----HASH TABLE----\n");
    printf("\nInput Key\tOutput Index");
    for(k=1000;k<=1025;k++) /*Specifying input key values*/
        printf("\n%d\t\t %d",k,hash(k)); /*Calling hash function to generate indices*/

    getch();
}

```

```
int hash(int l)
{
    return(l-1000);
}
```

Output

-----HASH TABLE-----

Input Key	Output Index
1000	0
1001	1
1002	2
1003	3
1004	4
1005	5
1006	6
1007	7
1008	8
1009	9
1010	10
1011	11
1012	12
1013	13
1014	14
1015	15
1016	16
1017	17
1018	18
1019	19
1020	20
1021	21
1022	22
1023	23
1024	24
1025	25



Just Remember

- Sorting means rearranging a given list in an orderly manner. Searching is the process of locating a particular key from a list.
- The efficiency of a sorting algorithm is expressed in terms of the O-notation.
- A sorting algorithm with $O(\log n)$ is supposed to be a highly efficient sorting algorithm.
- The efficiency of bubble sort, insertion sort, selection sort and shell sort is $O(n)$. The efficiency of heap sort, merge sort and quick sort $O(n \log n)$.
- Quick sort designed by Hoare is a very efficient sorting algorithm.
- Binary search may be combined with the indexed sequential searching to give better efficiency.
- Always remember that searching in a sorted list is more efficient than searching in an unsorted list.



Multiple Choice Questions

- The time complexity of binary search in average case is
 - $O(n)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(\log n)$
- The time complexity of bubble sort in best case is
 - $O(n)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(\log n)$
- The time complexity of selection sort in best case is
 - $O(n)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(\log n)$
- The time complexity of insertion sort in best case is
 - $O(n)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(\log n)$
- The time complexity of quick sort in best case is
 - $O(n)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(\log n)$
- The time complexity of heap sort in best case is
 - $O(n)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(\log n)$
- The time complexity of merge sort in best case is
 - $O(n)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(\log n)$
- The best sorting technique among the following is
 - quick
 - heap
 - merge
 - bubble
- In worst case quick sort behaves like
 - insertion
 - heap
 - selection
 - bubble
- The time complexity of bubble sort in worst case is
 - $O(n)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(\log n)$
- The time complexity of selection sort in worst case is
 - $O(n)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(\log n)$
- The time complexity of insertion sort in worst case is
 - $O(n)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(\log n)$
- The time complexity of quick sort in worst case is
 - $O(n)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(\log n)$

14. The time complexity of heap sort in worst case is
 - (a) $O(n)$
 - (b) $O(n^2)$
 - (c) $O(n \log n)$
 - (d) $O(\log n)$
15. The time complexity of merge sort in worst case is
 - (a) $O(n)$
 - (b) $O(n^2)$
 - (c) $O(n \log n)$
 - (d) $O(\log n)$
16. Quick sort is an application of
 - (a) partition exchange sort
 - (b) partition sort
 - (c) greedy method
 - (d) divide and conquer
17. Merge sort is an application of
 - (a) greedy method
 - (b) divide and conquer
 - (c) a and b
 - (d) none
18. The space complexity of quick sort in average case is
 - (a) 0
 - (b) $O(n)$
 - (c) $O(n \log n)$
 - (d) $O(\log n)$
19. The space complexity of bubble sort in worst case is
 - (a) 0
 - (b) $O(n)$
 - (c) $O(n \log n)$
 - (d) $O(\log n)$
20. Binary search is effective only when the elements are in
 - (a) ascending order
 - (b) descending order
 - (c) a and b
 - (d) jumbled order



Review Questions

- 7.1 Discuss bubble sort, write a C program and show its efficiency.
- 7.2 Discuss the quick sort. Derive the efficiency of the quick sort in the best and worst cases.
- 7.3 What is a heap? Show how would you perform a heap sort using a heap? Work out the efficiency factor of the heap sort.
- 7.4 Show that the efficiency of quick sort in its worst case is the same as that of simple insertion sort.
- 7.5 Show that shell sort is based on the insertion sort.
- 7.6 Discuss the linear search method.
- 7.7 Explain the binary searching with a C program.
- 7.8 What is an index? How can an index speed up the search process?
- 7.9 State whether the following statements are True or False
 - (a) Sorting and searching are always tightly interrelated.
 - (b) Quick sort's efficiency relies on the pivot.
 - (c) Bubble sort even though simple is very efficient.
 - (d) Selection sort is based on binary tree.
 - (e) Shell sort is based on quick sort.
 - (f) Quick sort uses a divide-and-conquer approach.
 - (g) A sorting method whose efficiency is $O(2^n)$ means that the amount of time the algorithm will take to complete doubles each time you increase the input.
 - (h) A sorting method with $O(\log n)$ is the most efficient method.
 - (i) Sorting is mandatory for linear searching.
 - (j) Binary search can be done on an ordered list only.
 - (k) Indexed sequential search can be applied to file handling only and not to list.
 - (l) The choice of constructing the index by picking every third number from the main list can be changed.

8 Data Structures

8.1 INTRODUCTION

In the field of computing, we have formulated many ways to handle data efficiently. So far, we have seen how to use variables to store data. However, variables are not feasible when handling a huge amount of data.

Consider a situation when we want to access information, such as student name, roll no, section name, sex, address, subjects, and marks obtained in each subject for students in a specific state of the country. With a single variable, it is an unfeasible task to store such a large amount of related data. Also, we need an organized medium of storage to handle any correlated information. Thus, the concept of data structure is introduced.

Arrays are considered as good example for implementing simple data structures. Arrays can be effectively used for random access of fixed amount of data. However, if the data structure requires frequent insertions and deletions then arrays are not much efficient because even a single insertion or deletion operation forces many other data elements to be moved to new locations. Also, there is always an upper limit to the number of elements that can be stored in the array-based data structure.

Dynamic data structures such as linked lists, which are based on dynamic memory management techniques, remove these limitations of static array-based data structures. They provide the flexibility in adding, deleting or rearranging data items at run time. For example, consider a program for processing the list of customers of a corporation. The list grows when names are added and shrinks when names are deleted. When the list grows, we need to allocate more memory space to the list to accommodate additional data items. With arrays, there will always be a restriction to the maximum number of elements that can be accommodated into the list; but with linked lists there is no such concern. The use of dynamic memory management techniques by linked list allows it to allocate additional memory space or to release unwanted space at run time, thus, optimizing the use of storage space.

There are certain linear data structures (e.g. stacks and queues) that permit the insertion and deletion operations only at the beginning or at end of the list, not in the middle. Such data structures have significant importance in systems processes such as compilation and program control.

8.2 Computer Programming

Linked lists, stacks and queues are the major focus areas of this chapter along with their array and linked implementations. But before we start, let's explore the abstract data type model on which these data structures are based.

8.2 ABSTRACT DATA TYPES

Abstract Data Type (ADT) is a specification for the type of values that a data type can store and the operations that can be performed on those values. It is just a specification without any details on the implementation part. ADT is basically a logical encapsulation of the set of data values along with their associated operations. That means, the outside world can interact with the data type only through the interfaced set of operations, the implementation details of which are well encapsulated or hidden inside the logical ADT unit. Figure 8.1 shows this abstraction:

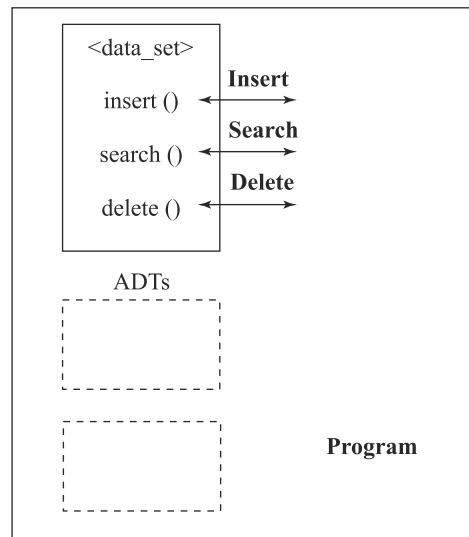


Fig. 8.1 Representation of ADT

The above figure shows an ADT containing a dataset and three functions to perform operations on the data set. The outside program can interact with the ADT only through these functions. The code implementation of the functions, though present within the same program, is logically segregated from rest of the program instructions. In the above figure, the only way for the program to use or manipulate the ADT is through the latter's three subroutines, insert, search and delete.

8.3 LINEAR LIST

We know that a list refers to a set of items organized sequentially. An array is an example of list. In an array, the sequential organization is provided implicitly by its index. We use the index for accessing and manipulation of array elements. One major problem with the arrays is that the size of an array must be specified precisely at the beginning. As pointed out earlier, this may be a difficult task in many real-life applications.

A completely different way to represent a list is to make each item in the list part of a structure that also contains a "link" to the structure containing the next item, as shown in Fig. 8.4. This type of list is called a *linked list* because it is a list whose order is given by links from one item to the next.

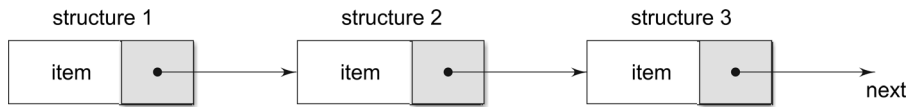


Fig. 8.2 A linked list

Each structure of the list is called a *node* and consists of two fields, one containing the item, and the other containing the address of the next item (a pointer to the next item) in the list. A linked list is therefore a collection of structures ordered not by their physical placement in memory (like an array) but by logical links that are stored as part of the data in the structure itself. The link is in the form of a pointer to another structure of the same type. Such a structure is represented as follows:

```

struct node
{
    int item;
    struct node *next;
};
  
```

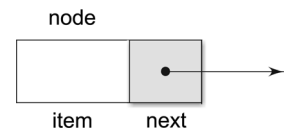
The first member is an integer **item** and the second a pointer to the next node in the list as shown below. Remember, the **item** is an integer here only for simplicity, and could be any complex data type.

Such structures, which contain a member field that points to the same structure type are called *self-referential* structure.

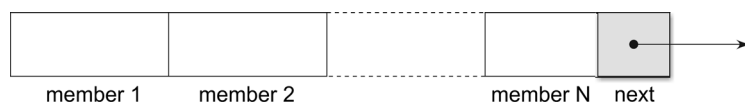
A node may be represented in general form as follows:

```

struct tag-name
{
    type member1;
    type member2;
    . . . .
    . . . .
    struct tag-name *next;
};
  
```



The structure may contain more than one item with different data types. However, one of the items must be a pointer of the type **tag-name**.



8.4 Computer Programming

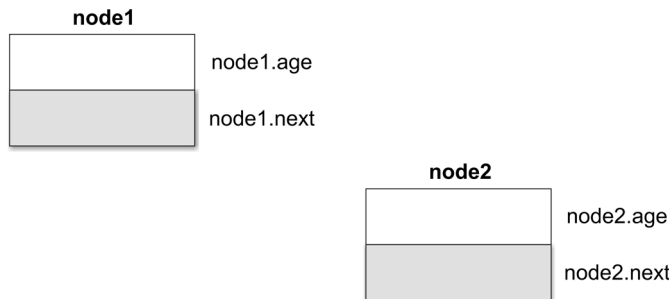
Let us consider a simple example to illustrate the concept of linking. Suppose we define a structure as follows:

```
struct link_list  
{  
  
    float age;  
  
    struct link_list *next;  
  
};
```

For simplicity, let us assume that the list contains two nodes **node1** and **node2**. They are of type **struct link_list** and are defined as follows:

```
struct link_list node1, node2;
```

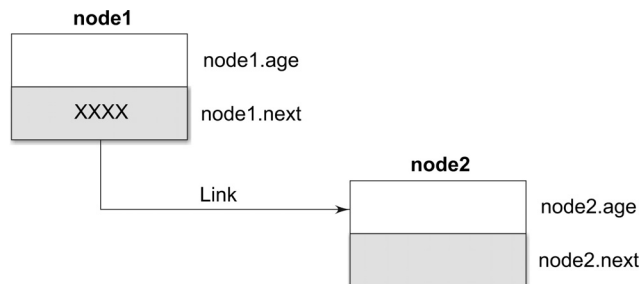
This statement creates space for two nodes each containing two empty fields as shown:



The **next** pointer of **node1** can be made to point to **node2** by the statement

```
node1.next = &node2;
```

This statement stores the address of **node2** into the field **node1.next** and thus establishes a “link” between **node1** and **node2** as shown:

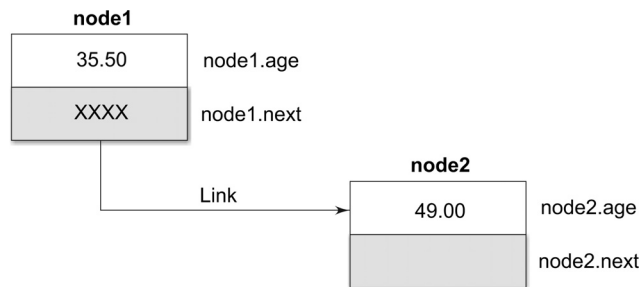


“xxxx” is the address of **node2** where the value of the variable **node2.age** will be stored. Now let us assign values to the field age.

```
node1.age = 35.50;
```

```
node2.age = 49.00;
```

The result is as follows:



We may continue this process to create a linked list of any number of values.

For example:

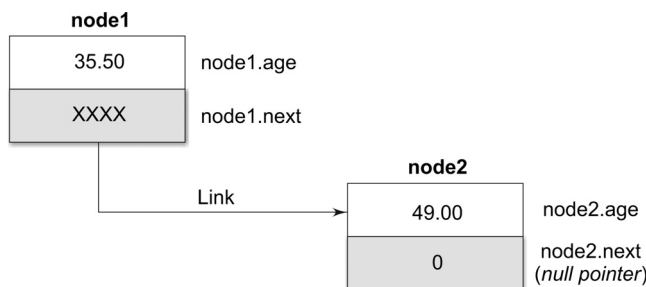
```
node2.next = &node3;
```

would add another link provided **node3** has been declared as a variable of type **struct link list**.

No list goes on forever. Every list must have an end. We must therefore indicate the end of a linked list. This is necessary for processing the list. C has a special pointer value called **null** that can be stored in the **next** field of the last node. In our two-node list, the end of the list is marked as follows:

```
node2.next = 0;
```

The final linked list containing two nodes is as shown:



The value of the age member of **node2** can be accessed using the **next** member of **node1** as follows:

```
printf("%f\n", node1.next->age);
```

8.3.1 Linked Lists Operations

Since a linked list is an example of an ADT, we must define the associated functions that are required to manipulate the linked list structure. The typical operations performed on a linked list are

- Insert
- Delete
- Search
- Print

Insert

It involves adding an element into the linked list and resetting the link pointers wherever required. Inserting an element into a list has three possibilities:

- Insertion at the beginning of the list

8.6 Computer Programming

- Insertion at the end of the list
- Insertion anywhere in the middle of the list

The first two situations can be easily realized with the help of 'first' and 'last' pointers pointing to the first and last elements in the list. However, the third situation of inserting an element in the middle requires a search operation to be performed for identifying the insertion location. Figure 8.3 (a) and (b) shows the insertion of an element between two existing elements of the linked list:

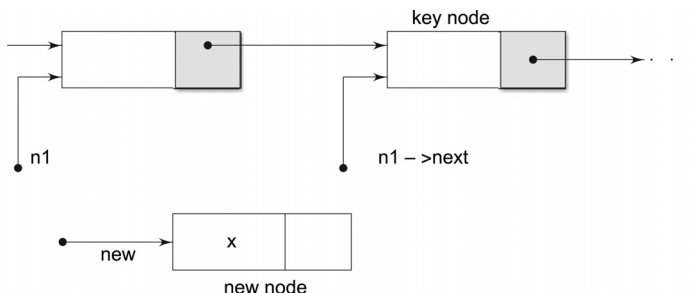


Fig. 8.3 (a) Creating a new element

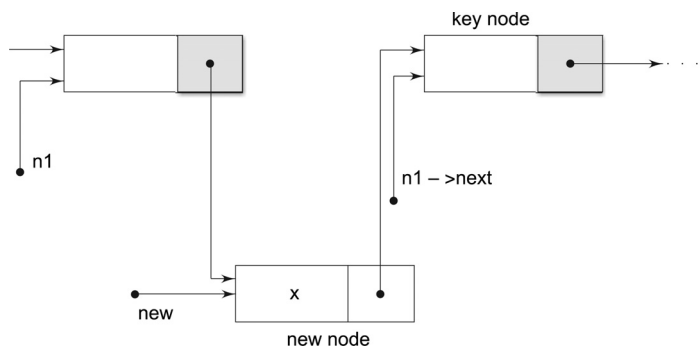


Fig. 8.3 (b) Inserting the newly created element

Example 8.1 Write a function in C to insert an element at the end of a linked list.

Program

```
/*Insert function*/
void insert(int value)
{
    /*Dynamically declaring a list element*/
    struct list *ptr = (struct list*)malloc(sizeof(struct list));
    /*Assigning value to the newly allocated list element*/
    ptr->element = value;

    /*Updating linked list pointers*/
```



```

if(first==NULL)
{
    first = last = ptr;
    ptr->next=NULL;
}
else
{
    last->next = ptr;
    ptr->next = NULL;
    last = ptr;
}
}

```

Delete

It involves removing an element from the linked list and resetting the link pointers wherever required. Deleting an element from the list has three possibilities:

- Deletion from the beginning of the list
- Deletion from the end of the list
- Deletion of a specific element from the list

The first two situations can be easily realized with the help of 'first' and 'last' pointers pointing to the first and last elements in the list. However, the third situation of deleting a specific element first requires a search operation to be performed for locating that element. Figure 8.4 (a) and (b) shows the deletion of an element from the middle of the list:

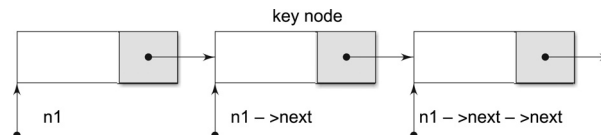


Fig. 8.4 (a) Searching the element to be deleted

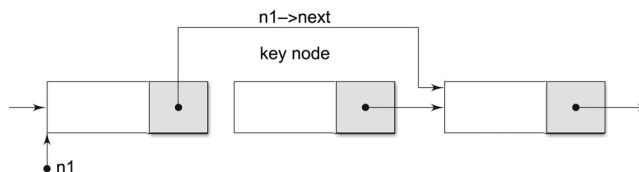


Fig. 8.4 (b) Deleting the element

Example 8.2 Write a function in C to delete a specific element from a linked list.

8.8 Computer Programming

Program

```
/*Delete function*/
int delete(int value)
{
    struct list *loc,*temp;
    int i;
    i=value;

    loc=search(i); /*Searching the element to be deleted*/

    if(loc==NULL) /*Element not found*/
        return(-9999);

    if(loc==first) /*Element is found at first location in the list*/
    {
        if(first==last) /*Found element is the only element in the list*/
            first=last=first->next;
        else
            first=first->next;
        return(value);
    }

    for(temp=first;temp->next!=loc;temp=temp->next)
        ;
    temp->next=loc->next; /*Deleting the element and updating the link pointer */
    if(loc==last)
        last=temp;
    return(loc->element);
}
```

Search

It involves traversing the linked list to find a specific element. The pointers between elements help in traversing the linked list from start till end.

Example 8.3 Write a function in C to search a specific element in the linked list.

Program

```
/*Search function*/
struct list *search (int value)
{
    struct list *ptr;
```

```

    if(first==NULL) /*Checking whether the list is empty*/
        return(NULL);

    if(first==last && first->element==value) /*Condition where list has only one element*/
        return(first);

    /*Traversing the linked list*/
    for(ptr=first;ptr!=last;ptr=ptr->next)
        if(ptr->element==value)
            return(ptr); /*Returning the location where element is found*/

    if(last->element==value)
        return(last);
    else
        return(NULL); /*Returning NULL value in case of unsuccessful search*/
}

```

Print

It involves displaying all the linked list elements on the console. The pointers between elements help in traversing the linked list from start till end.

Example 8.4 Write a function in C to display all the linked list elements.

Program

```

/*Display function*/
void display()
{
    struct list *ptr;

    if(first==NULL) /*Checking whether the linked list is empty*/
    {
        printf("\n\tList is Empty!!");
        return;
    }

    printf("\nElements present in the Linked list are:\n");
    if(first==last) /*Only one element present in the list*/
    {
        printf("\t%d",first->element);
        return;
    }
}

```

8.10 Computer Programming

```
/*Printing the linked list elements*/
for(ptr=first;ptr!=last;ptr=ptr->next) /*Traversing the linked list*/
    printf("\t%d",ptr->element);
printf("\t%d",last->element);
}
```

8.3.2 Implementation

Example 8.5 Write a program to implement a linked list and perform its common operations.

The following program implements a dynamic linked list in C. It uses the insert (Example 8.1), delete (Example 8.2), search (Example 8.3) and display (Example 8.4) functions for realizing the common linked list operations.

Program

```
/*Program for implementing a simple linked list*/
#include<stdio.h>
#include<conio.h>

struct list /*Declaring the structure of the linked list element*/
{
    int element;
    struct list *next; /*Linked list element pointing to another element in the list*/
};

/*Declaring the linked list pointers*/
struct list *first = NULL;
struct list *last = NULL;

void insert(int); /*Declaring a function prototype for inserting an element into the linked list*/
int delete(int); /*Declaring a function prototype for deleting an element from the linked list*/
void display(void); /*Declaring a function prototype for displaying the elements of the linked list*/
struct list *search (int); /*Declaring a function prototype for searching an element in the linked list*/

void main()
{
    int num1, num2, choice;
    struct list *loc;
```

```

while(1)
{
    clrscr();
    /*Creating an interactive interface for performing standard linked list operations*/
    printf("\n\nSelect an option\n");
    printf("\n1 - Insert an element into the linked list");
    printf("\n2 - Delete an element from the linked list ");
    printf("\n3 - Search an element in the linked list ");
    printf("\n4 - Display the elements of the linked list");
    printf("\n5 - Exit");

    printf("\n\nEnter your choice: ");
    scanf("%d", &choice);

    switch(choice)
    {
        case 1:
        {
            printf("\nEnter the element to be inserted into the linked list: ");
            scanf("%d",&num1);
            insert(num1); /*Inserting an element*/
            getch();
            break;
        }

        case 2:
        {
            printf("\nEnter the element to be deleted from the linked list: ");
            scanf("%d",&num1);
            num2=delete(num1); /*Deleting an element*/
            if(num2==-9999)
                printf("\n\t%d element not present in the linked list\n\t",num1);
            else
                printf("\n\t%d element removed from the linked list\n\t",num2);
            getch();
            break;
        }

        case 3:
        {

```

8.12 Computer Programming

```
printf("\nEnter the element to be searched in the linked list: ");
scanf("%d",&num1);
loc=search(num1); /*Searching an element*/
if(loc==NULL)
    printf("\n\t%d element not present in the linked list\n\t",num1);
else
    printf("\n\t%d element is present before element %d in the linked list\n\t",num1,(loc->next)->element);

    getch();
    break;
}

case 4:
{
    display(); /*Displaying linked list elements*/
    getch();
    break;
}

case 5:
{
    exit(1);
    break;
}

default:
{
    printf("\nInvalid choice.");
    getch();
    break;
}
}
}
```

Output

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list

- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

Enter your choice: 1

Enter the element to be inserted into the linked list: 1

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list
- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

Enter your choice: 4

Elements present in the Linked list are:

1

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list
- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

Enter your choice: 1

Enter the element to be inserted into the linked list: 2

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list
- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

8.14 Computer Programming

Enter your choice: 1

Enter the element to be inserted into the linked list: 3

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list
- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

Enter your choice: 1

Enter the element to be inserted into the linked list: 4

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list
- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

Enter your choice: 4

Elements present in the Linked list are:

1 2 3 4

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list
- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

Enter your choice: 2

Enter the element to be deleted from the linked list: 5

5 element not present in the linked list

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list
- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

Enter your choice: 2

Enter the element to be deleted from the linked list: 3

3 element removed from the linked list

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list
- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

Enter your choice: 4

Elements present in the Linked list are:

1 2 4

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list
- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

Enter your choice: 3

Enter the element to be searched in the linked list: 3

3 element not present in the linked list

8.16 Computer Programming

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list
- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

Enter your choice: 3

Enter the element to be searched in the linked list: 2

2 element is present before element 4 in the linked list

Select an option

- 1 - Insert an element into the linked list
- 2 - Delete an element from the linked list
- 3 - Search an element in the linked list
- 4 - Display the elements of the linked list
- 5 - Exit

Enter your choice: 5

8.3.3 Linked List with Header

Till now, we have seen a simple linked list which contains exactly the same number of nodes as there are elements in the list. However, there is another type of linked list which contains a special node present at the beginning of the list called header node. It is exactly similar to any other node in the list with the exception that its data field is empty. The link field of the header node points to the first element in the list, and if there are no elements then it points to null. The start pointer however always points to the header node. Thus, the first node in such linked lists is always the one being pointed by start or head node. The objective of implementing a header linked list is to segregate the actual list elements and have a separate pointer mechanism for tracking the beginning of the list.

Figure 8.5 shows the representation of a header linked list:

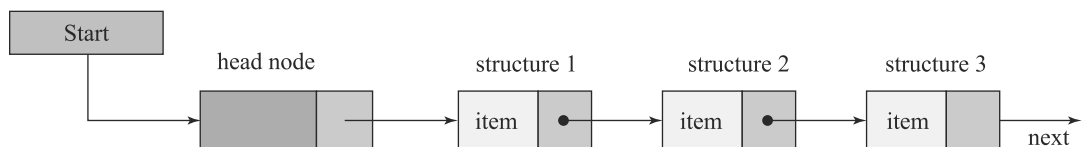


Fig. 8.5 Header linked list

Implementation of Header Linked Lists

If we look at Fig. 8.5 above from implementation perspective, we'll notice that the main difference between a simple linked list and a header linked list is the start pointer. In case of linked lists without header, start was itself a node (first node in the list) containing a value, but in case of header linked lists, it is a pointer to the first node. Thus, the implementation of header linked lists can be simply derived from a normal linked list implementation by just replacing start with start -> next.

Example 8.6 Write a function in C to search a specific element in a header linked list.

Program

```
/*Search function*/
struct list *search (int value)
{
    struct list *ptr;

    if(first->next==NULL) /*Checking whether the list is empty*/
        return(NULL);

    if(first->next==last && (first->next)->element==value) /*Condition where list has
only one element*/
        return(first->next);

    /*Traversing the linked list*/
    for(ptr=first->next;ptr!=last;ptr=ptr->next)
        if(ptr->element==value)
            return(ptr); /*Returning the location where element is found*/

    if(last->element==value)
        return(last);
    else
        return(NULL); /*Returning NULL value in case of unsuccessful search*/
}
```

Similarly, we can also realize the other function implementations of a header linked list.

8.4 STACKS

A stack is a linear data structure in which a data item is inserted and deleted at one end. A stack is called a Last In First Out (LIFO) structure because the data item that is inserted last into the stack is the first data item to be deleted from the stack. To understand this, imagine a pile of books. Usually, we pick up the topmost book from the pile. Similarly, the *topmost* item (i.e. the item which made the entry last) is the first one to be picked up/removed from the stack.

8.18 Computer Programming

Stacks are extensively used in computer applications. Their most notable use is in system software (such as compilers, operating systems, etc.). For example, when one C function calls another function, and passes some parameters, these parameters are passed using the stack. In fact, even many compilers store the local variables inside a function on the stack.

For a real-life use of stacks, imagine a magazines library. New magazines keep arriving in the library, and keep getting registered. Whenever a member walks in, she asks for the *latest* magazine that is available. The librarian takes a look at the topmost magazine, and offers it to the member. As and when any magazine is issued to a member, it is removed from the list of the available magazines. Of course, to keep things simple, we shall ignore the cases where the members do not want the latest magazine, but want something else. A stack best implements this arrangement.

8.4.1 Stack Operations

In general, writing a value to the stack is called as a *push* operation, whereas reading a value from it is called as a *pop* operation. Like in the case of queues, a pop (i.e. read) operation in the case of a stack is also destructive. That is, once an item is popped from the stack, it is no longer available.

Figure 8.6 specifies what happens when the push and pop operations get executed. Note how the new items get added to the end of the stack, and also how elements get removed from there. We assume that initially the stack is empty.

Operation	Contents of the stack after the operation
Push (A)	A
Push (B)	A B
Pop	A
Push (C)	A C
Push (D)	A C D
Push (E)	A C D E
Pop	A C D
Pop	A C
Pop	A
Pop	Empty

Fig. 8.6 How a stack works

8.5 STACK IMPLEMENTATION

A typical implementation of the push operation checks if there is still any room left in the stack, and if there is any, it adds the received item at the top of the stack, and increments the stack counter by one. Similarly, the implementation of pop operation checks whether or not the stack is already empty, if it is not, it returns the top element of the stack and decrements the stack counter by one.

We can implement stacks by using arrays or linked lists. The advantages or disadvantages of array or linked implementations of stacks are the same that are associated with such types of data structures. However, both types of implementations have their own usage in specific situations.

8.5.1 Array Implementation of Stacks

The array implementation of stacks involves allocating a fixed size array in the memory. Both stack operations (push and pop) are made on this array with a constant check being made to ensure that the array does not go out of bounds.

Push Operation

The push operation involves checking whether or not the stack pointer is pointing at the upper bound of the array. If it is not, the new item is pushed (inserted) into the stack.

Example 8.7 Write a function in C to implement the push operation under array representation of stacks.

Program

```
/*Push function*/
void push(int element)
{
    if(top==99) /*Checking whether the stack is full*/
    {
        printf("Stack is Full.\n");
        getch();
        exit(1);
    }
    top=top+1; /*Incrementing stack pointer*/
    stack[top]=element; /*Inserting the new element*/
}
```

Pop Operation

The pop operation involves checking whether or not the stack pointer is already pointing at NULL (empty stack). If it is not, the item that is being currently pointed is popped (removed) from the stack and the stack pointer is reset accordingly.

Example 8.8 Write a function in C to implement the pop operation under array representation of stacks.

Program

```
/*Pop function*/
int pop()
{
    int element;
    if(top== -1) /*Checking whether the stack is empty*/
    {
        printf("\n\tStack is Empty.\n");
    }
}
```

8.20 Computer Programming

```
    getch();
    exit(1);
}
return(stack[top--]); /*Returning the top element and decrementing the stack pointer*/
}
```

Implementation

Example 8.9 Write a program to implement a stack using arrays and perform its common operations.

The following program implements a stack using arrays in C. It uses the push (Example 8.8) and pop (Example 8.7) functions for realizing the common stack operations.

Program

```
/*Program for demonstrating implementation of stacks using arrays*/
#include <stdio.h>
#include <conio.h>

int stack[100]; /*Declaring a 100 element stack array*/
int top=-1; /*Declaring and initializing the stack pointer*/

void push(int); /*Declaring a function prototype for inserting an element into the stack*/
int pop(); /*Declaring a function prototype for removing an element from the stack*/
void display(); /*Declaring a function prototype for displaying the elements of a stack*/

void main()
{
    int choice;
    char num1=0,num2=0;
    while(1)
    {
        clrscr();
        /*Creating an interactive interface for performing stack operations*/
        printf("Select a choice from the following:");
        printf("\n[1] Push an element into the stack");
        printf("\n[2] Pop out an element from the stack");
        printf("\n[3] Display the stack elements");
        printf("\n[4] Exit\n");
        printf("\n\tYour choice: ");
        scanf("%d",&choice);
```

```

switch(choice)
{
    case 1:
    {
        printf("\n\tEnter the element to be pushed into the stack: ");
        scanf("%d",&num1);
        push(num1); /*Inserting an element*/
        break;
    }

    case 2:
    {
        num2=pop(); /*Removing an element*/
        printf("\n\t%d element popped out of the stack\n\t",num2);
        getch();
        break;
    }

    case 3:
    {
        display(); /*Displaying stack elements*/
        getch();
        break;
    }

    case 4:
        exit(1);
        break;

    default:
        printf("\nInvalid choice!\n");
        break;
    }
}

void display()
{
    int i;
    printf("\n\tThe various stack elements are:\n\t");
    for(i=top;i>=0;i--)
        printf("%d\t",stack[i]); /*Printing stack elements*/
}

```

8.22 Computer Programming

Output

Select a choice from the following:

- [1] Push an element into the stack
- [2] Pop out an element from the stack
- [3] Display the stack elements
- [4] Exit

Your choice: 1

Enter the element to be pushed into the stack: 1

Select a choice from the following:

- [1] Push an element into the stack
- [2] Pop out an element from the stack
- [3] Display the stack elements
- [4] Exit

Your choice: 1

Enter the element to be pushed into the stack: 2

Select a choice from the following:

- [1] Push an element into the stack
- [2] Pop out an element from the stack
- [3] Display the stack elements
- [4] Exit

Your choice: 1

Enter the element to be pushed into the stack: 3

Select a choice from the following:

- [1] Push an element into the stack
- [2] Pop out an element from the stack
- [3] Display the stack elements
- [4] Exit

Your choice: 3

The various stack elements are:

3 2 1

Select a choice from the following:

- [1] Push an element into the stack
- [2] Pop out an element from the stack
- [3] Display the stack elements
- [4] Exit

Your choice: 2

3 element popped out of the stack

Select a choice from the following:

- [1] Push an element into the stack
- [2] Pop out an element from the stack
- [3] Display the stack elements
- [4] Exit

Your choice: 3

The various stack elements are:

2 1

Select a choice from the following:

- [1] Push an element into the stack
- [2] Pop out an element from the stack
- [3] Display the stack elements
- [4] Exit

Your choice: 4

8.5.2 Linked Implementation of Stacks

The linked implementation of stacks involves dynamically allocating memory space at run time while performing stack operations. Since, the allocation of memory space is dynamic, the stack consumes only that much amount of space as is required for holding its elements. This is contrary to array-based stacks which occupy a fixed memory space even if there are no elements present.

Push Operation

The push operation under linked implementation of stacks involves the following tasks:

- Reserving memory space of the size of a stack element in memory
- Storing the pushed (inserted) value at the new location
- Linking the new element with existing stack
- Updating the stack pointer

Example 8.10 Write a function in C to implement the push operation under linked representation of stacks.

Program

```
/*Push function*/
void push(int value)
{
    struct stack *ptr;
    ptr=(struct stack*)malloc(sizeof(struct stack)); /*Dynamically declaring a stack element*/

    ptr->element=value; /*Assigning value to the newly allocated stack element*/

    /*Updating stack pointers*/
    ptr->next=top;
    top=ptr;
    return;
}
```

Pop Operation

The pop operation under linked implementation of stacks involves the following tasks:

- Checking whether the stack is empty
- Retrieving the top element of the stack
- Updating the stack pointer
- Returning the retrieved (popped) value

Example 8.11 Write a function in C to implement the pop operation under linked representation of stacks.

Program

```
/*Pop function*/
int pop()
{
    if(top==NULL) /*Checking whether the stack is empty*/
    {
        printf("\n\STACK is Empty.");
        getch();
        exit(1);
    }
    else
```

```

{
    int temp=top->element; /* Retrieving the top element*/
    top=top->next; /*Updating the stack pointer*/
    return (temp); /*Returning the popped value*/
}
}

```

Implementation

Example 8.12 Write a program to implement a stack using linked lists and perform its common operations.

The following program implements a stack using linked lists in C. It uses the push (Example 8.10) and pop (Example 8.11) functions for realizing the common stack operations.

Program

```

/*Program for demonstrating implementation of stacks using linked list*/
#include <stdio.h>
#include <conio.h>

struct stack /*Declaring the structure for stack elements*/
{
    int element;
    struct stack *next; /*Stack element pointing to another stack element*/
}*top;

void push(int); /*Declaring a function prototype for inserting an element into the
stack*/
int pop(); /*Declaring a function prototype for removing an element from the stack*/
void display(); /*Declaring a function prototype for displaying the elements of a
stack*/

void main()
{
    int num1, num2, choice;

    while(1)
    {
        clrscr();
        /*Creating an interactive interface for performing stack operations*/
        printf("Select a choice from the following:");
        printf("\n[1] Push an element into the stack");
        printf("\n[2] Pop out an element from the stack");
    }
}

```

8.26 Computer Programming

```
printf("\n[3] Display the stack elements");
printf("\n[4] Exit\n");
printf("\n\tYour choice: ");
scanf("%d",&choice);

switch(choice)
{
    case 1:
    {
        printf("\n\tEnter the element to be pushed into the stack: ");
        scanf("%d",&num1);
        push(num1); /*Inserting an element*/
        break;
    }

    case 2:
    {
        num2=pop(); /*Removing an element*/
        printf("\n\t%d element popped out of the stack\n\t",num2);
        getch();
        break;
    }

    case 3:
    {
        display(); /*Displaying stack elements*/
        getch();
        break;
    }

    case 4:
    {
        exit(1);
        break;
    }

    default:
    {
        printf("\nInvalid choice!\n");
        break;
    }
}
```

```

void display()
{
    struct stack *ptr1=NULL;
    ptr1=top;
    printf("\nThe various stack elements are:\n");
    while(ptr1!=NULL)
    {
        printf("%d\t",ptr1->element); /*Printing stack elements*/
        ptr1=ptr1->next;
    }
}

```

Output

The output of the above program is same as the output of the program shown in Example 8.9.

8.6 APPLICATIONS OF STACKS

Stack operations are implemented in machines to do some basic tasks, such as evaluating expressions, handling dynamic memory allocation, etc.

Consider an arithmetic expression, $a+b*c$. In this expression, the addition operation is not evaluated first. This is because, operators are evaluated in the order of their precedence in the expression. So the entire equation is examined to determine whether there is any operator with higher precedence. After examining the expression, back tracking is done to evaluate the first operator to obtain the result. The back tracking operation can be best implemented by the stack operations. The various stack-oriented notations are:

- Infix
- Prefix
- Postfix

8.6.1 Infix Notation

As we know, an ordinary mathematical expression is called infix notation. When using infix notation, the operators are placed between the operands in an expression. While denoting an expression in infix notation, we use parenthesis to specify the order in which the operations are to be performed. Otherwise, the precedence rules will be followed to obtain an unambiguous result of the expression.

Example 8.13 An example of an infix notation is, $(A/B)+(C-D)*E$.

8.6.2 Prefix Notation

In the case of prefix notation, the operators are placed before the operands in a mathematical expression. This notation is also known as polish notation.

Example 8.14 The prefix notation for the equation, $A/B+C-D*E$, is $+/AB*-CDE$.

8.6.3 Conversion from Infix to Prefix Notation

The following algorithm is used to convert an infix expression to prefix expression.

Algorithm

1. First, initialize the stack to be empty and reverse the given input string.
2. For each character in the input string
 - If the input string is a right parenthesis, push it onto the stack
 - If the input string is an operand, append to the output.
 - else
 - If the stack is empty or the operator has higher priority than the operator on the top of the stack or the top of the stack is a right parenthesis,
 - then
 - Push the operator onto the stack
 - else
 - Pop the operator from the stack and append to the output.
3. If the input string is a left parenthesis, pop operators from the stack and append all the operators to the output until the right parenthesis is encountered. Pop the right parenthesis from the stack and discard it.
4. If the end of the input string is encountered, then iterate the loop until the stack is not empty. Pop the stack, and append the remaining input string to the output and reverse the output string.

Example 8.15 Consider the conversion of the infix expression, $A*B/(C-D)+E*(F-G)$, to its equivalent prefix notation. First, to convert the infix expression to the prefix notation, reverse the given input string as follows:
 $)G-F(*E+)D-C(/B*A$

Input String	Stack Operation	Postfix Notation
))	
G)	G
-)-	G
F)-	GF
(Empty	GF-
*	*	GF-
E	*	GF-E
+	+	GF-E*
)	+))	GF-E*
D	+))	GF-E*D
-	+))-	GF-E*D
C	+))-	GF-E*DC
(+	GF-E*DC-
/	+/	GF-E*DC-
B	+/	GF-E*DC-B
*	+/*	GF-E*DC-BA
A	+/*	GF-E*DC-BA
	Empty	GF-E*DC-BA*/+

Now, reverse the output string, GF-E*DC-BA*/+, as +/*AB-CD*E-FG to obtain the prefix notation. Thus, the given infix expression is converted to its equivalent prefix notation. Conversion of an infix to prefix expression is shown in the following program.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
#define MAX 50
char output[MAX];
char stack[MAX];
char input[MAX];
char *s, *t;
int top;
int l ;
/*Initializing Function*/
void Initialize(void);
void SetExpression(char *);
char PopFromStack(void);
void PushOnStack(char);
int priority(char);
void ConvertToPrefix(void);
void main()
{
    clrscr();
    Initialize();
    printf("\nEnter an infix expression: ");
    gets(input);
    SetExpression(input);
    strrev(s);
    ConvertToPrefix();
    strrev(output);
    printf("\nThe Prefix expression is: ");
    puts(output);
    getch();
}
void Initialize(void)
{
    top = -1;
    strcpy(output, "");
    strcpy(stack, "");
    l = 0;
}
void SetExpression(char *str)
{
    s = str;
    l = strlen(s);
    *(output + l) = '\0';
    t = output;
}
/*Adding operator to the stack */
```

8.30 Computer Programming

```
void PushOnStack(char c)
{
    if (top == MAX - 1)
        printf("\nStack is full.\n");
    else
    {
        top++;
        stack[top] = c;
    }
}

/*Popping an operator from the stack*/
char PopFromStack(void)
{
    if (top == -1)
        return -1;
    else
    {
        char item = stack[top];
        top--;
        return item;
    }
}

/* Returning the priority of the operator */
int priority(char c)
{
    if (c == '^') return 3;
    if (c == '*' || c == '/' || c == '%')
        return 2;
    else if (c == '+' || c == '-') return 1;
    else return 0;
}

/* Converting the Infix expression to Prefix*/
void ConvertToPrefix (void)
{
    char opr;
    while (*s)
    {
        if (*s == ' ' || *s == '\t')
        {
            s++;
            continue;
        }
        if (isdigit (*s) || isalpha (*s))
        {
            while (isdigit (*s) || isalpha (*s))
            {
                *t = *s;
                s++;
                t++;
            }
        }
    }
}
```



```

    }
    if (*(s) == ')')
    {
        PushOnStack(*(s));
        s++;
    }
    if (*(s) == '*' || *(s) == '+' || *(s) == '/' || *(s) == '%' || *(s)
== '-' || *(s)
== '^')
    {
        if (top != -1)
        {
            opr = PopFromStack ();
            while(priority (opr)>priority (*(s)))
            {
                *(t) = opr;
                t++;
                opr = PopFromStack();
            }
            PushOnStack(opr);
            PushOnStack(*(s));
        }
        else PushOnStack ( *(s));
        s++ ;
    }
    if (*(s) == '(')
    {
        opr = PopFromStack ();
        while (opr != ')')
        {
            *(t) = opr;
            t++;
            opr = PopFromStack ();
        }
        s++;
    }
}
while (top != -1)
{
    opr = PopFromStack ();
    *(t) = opr;
    t++;
}
t++;
}

```

8.6.4 Evaluation of Prefix Expression

Stacks are also used to evaluate a prefix expression. To evaluate a prefix expression, consider the following steps:

8.32 Computer Programming

- Reverse the given input string.
- If the input string is an operand, then push it onto the stack.
- If the input string is an operator, then the first two operands on the stack are evaluated using this operator by popping them from the stack and the result is also placed onto the stack.

Example 8.16 Consider the evaluation of a prefix expression, $+/63*-432$. To do this, reverse the input string as $234-*36/+$ shown in Fig. 8.7.

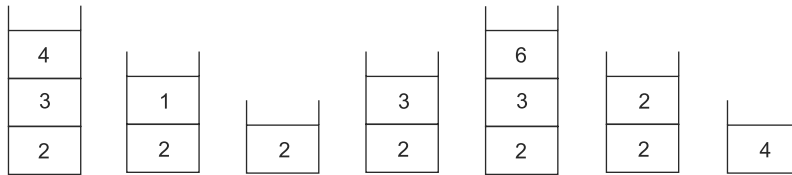


Fig. 8.7 Reversing the input string

8.6.5 Postfix Notation

When we use postfix notation for a mathematical expression, the operators are placed after the operands, i.e. the operators are preceded by the operands. Postfix notation is also known as Reverse Polish Notation (RPN). The advantage of using prefix and postfix notations is that the use of parenthesis and operator precedence rules is avoided completely.

Example 8.17 The postfix notation for the expression, $A/B+C-D*E$, is $AB/CD-E*+$.

8.6.6 Conversion from Infix to Postfix Notation

The following algorithm is used to convert an infix expression to an equivalent postfix expression:

Algorithm

1. First, initialize the stack to be empty.
2. For each character in the input string,
If input string is an operand, append to the output.
If the input string is a left parenthesis, push it onto the stack
else
If stack empty or the operator has higher priority than the operator on the top of the stack or the top of the stack is opening parenthesis
then
Push the operator onto the stack
else
Pop the operator from the stack and append to the output.
3. If the input string is a closing parenthesis, pop operators from the stack and append the operators to the output until an opening parenthesis is encountered. Pop the opening parenthesis from the stack and discard it.
4. If the end of the input string is encountered, then iterate the loop until the stack is not empty. Pop the stack and append the remaining input string to the output.

Example 8.18 Consider the conversion of the infix expression, $A*B/(C-D)+E*(F-G)$, to its equivalent postfix notation:

Input String	Stack Operation	Postfix Notation
A	Empty	A
*	*	A
B	*	AB
/	/	AB*
(/(AB*
C	/(AB*C
-	/(-	AB*C
D	/(-	AB*CD
)	/	AB*CD-
+	+	AB*CD-/
E	+	AB*CD-/E
*	+*	AB*CD-/E
(+*(AB*CD-/E
F	+*(AB*CD-/EF
-	+*(-	AB*CD-/EF
G	+*(-	AB*CD-/EFG
)	Empty	AB*CD-/EFG-*+

The given infix expression is converted to its equivalent postfix notation, as shown in the following program.

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

#define MAX 50
char output[MAX];
char stack[MAX];
char input[MAX];
char *s, *t ;
int top;
int l ;
/*Initializing Function*/
void Initialize(void);
void SetExpression(char *);
char PopFromStack(void);
void PushOnStack(char);
int priority(char);
void ConvertToPostfix(void);
```

8.34 Computer Programming

```
void main()
{
    clrscr();
    Initialize();
    printf("\nEnter an infix expression: ");
    gets(input);
    SetExpression(input);
    ConvertToPostfix();
    printf("\nThe postfix expression is: ");
    puts(output);
    getch();
}

void Initialize(void)
{
    top = -1;
    strcpy(output, "");
    strcpy(stack, "");
    l = 0;
}

void SetExpression(char *str)
{
    s = str;
    l = strlen(s);
    *(output + l) = '\0';
    t = output;
}

/*Adding operator to the stack */
void PushOnStack ( char c)
{
    if (top == MAX - 1)
        printf ("\nStack is full.\n");
    else
    {
        top++;
        stack[top] = c;
    }
}

/* Popping an operator from the stack */
char PopFromStack (void)
{
    if (top == -1) /* Stack is empty*/
        return -1;
    else
    {
        char item = stack[top];
        top--;
        return item;
    }
}
```

```

/* Returning the priority of the operator */
int priority (char c)
{
    if (c == '^') return 3 ;
    if (c == '*' || c == '/' || c == '%') return 2;
    else if (c == '+' || c == '-') return 1;
    else return 0;
}
/* Converting the infix expression to postfix */
void ConvertToPostfix (void)
{
    char opr;
    while (*(s) )
    {
        if (*(s) == ' ' || *(s) == '\t')
        {
            s++;
            continue;
        }
        if (isdigit (*(s) ) || isalpha (*(s) ))
        {
            while (isdigit (*(s) ) || isalpha (*(s) ))
            {
                *(t) = *(s);
                s++;
                t++;
            }
        }
        if (*(s) == '(')
        {
            PushOnStack ( *(s));
            s++;
        }
        if (*(s) == '*' || *(s) == '+' || *(s) == '/' || *(s) == '%' || *(s) == '-' || *(s) == '^')
        {
            if (top != -1)
            {
                opr = PopFromStack ();
                while (priority (opr) >=
                    priority (*(s) ))
                {
                    *(t) = opr;
                    t++;
                    opr =
                        PopFromStack ();
                }
                PushOnStack ( opr);
            }
        }
    }
}

```

8.36 Computer Programming

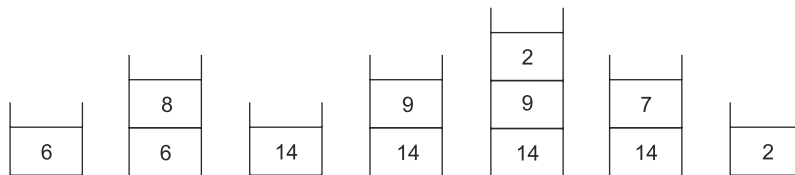
```
        PushOnStack ( *(s) );
    }
    else PushOnStack ( *(s));
    s++;
}
if (*(s) == ')')
{
    opr = PopFromStack ();
    while (opr != '(')
    {
        *(t) = opr;
        t++;
        opr = PopFromStack ();
    }
    s++;
}
while (top != -1)
{
    opr = PopFromStack ();
    *(t) = opr;
    t++;
}
t++;
}
```

8.6.7 Evaluation of Postfix Expression

Stacks are used to evaluate the postfix expression. To evaluate the postfix expression, consider the following steps:

- If the input string is an operand, then push it onto the stack.
- If the input string is an operator, then the first two operands on the stack are evaluated using this operator by popping them from the stack and the result is also placed onto the stack.

Example 8.19 Consider the evaluation of postfix expression 68+92-/.



8.7 QUEUES

A **queue** is very similar to the way we (are supposed to) queue up at train reservation counters or at bank cashiers' desks. A queue is a linear, sequential list of items that are accessed in the order *First In First Out (FIFO)*. That is, the first item inserted in a queue is also the first one to be accessed, the second item inserted in a queue is also the second one to be accessed, and the last one to be inserted is also the last one to be accessed. We cannot store/access the items in a queue arbitrarily or in any random fashion.

Suppose that we are creating a queue of items. For this purpose, let us assume two very simple functions, *write* and *read*. The *write* function adds a new item to the queue, whereas the *read* item reads one item from the queue. If we go on operating with these two functions with a few insertions and a few retrievals, the resulting values of the queue would look as shown in Fig. 8.9.

Operation	Contents of the queue
write (a)	a
write (b)	a b
write (c)	a b c
read (returns a)	b c
write (d)	b c d
read (returns b)	c d
read (returns c)	d
write (e)	d e

Fig. 8.9 Queue operations

As the figure shows, the operations on a queue are FIFO. Also, note that when one item is read from the queue, it is destroyed. Therefore, a *read* operation on a queue is destructive, unlike what happens with other data structures, such as linked lists. If the programmer needs to implement a non-destructive reading of a queue, the values, as they are read (and therefore, automatically removed) from the queue, must be stored somewhere else for later retrieval.

8.7.1 Queue Operations

In order to implement the *write* and *read* operations of a queue, two pointers, *start* and *end* are required. One pointer (*start*) points at the current start of the queue (i.e. at the item that is the first in the queue at any point in time). The other pointer (*end*) points at the current end of the queue (i.e. at the next storage location available in the queue for a new item). Thus, insertions and retrievals from a queue, as discussed earlier, would have the effects on the pointers as shown in Fig. 8.10.

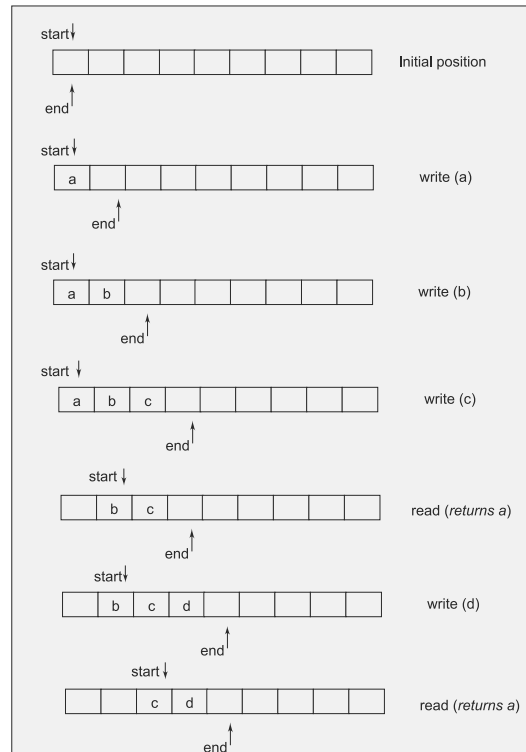


Fig. 8.10 Pointer movements because of queue operations—Part I

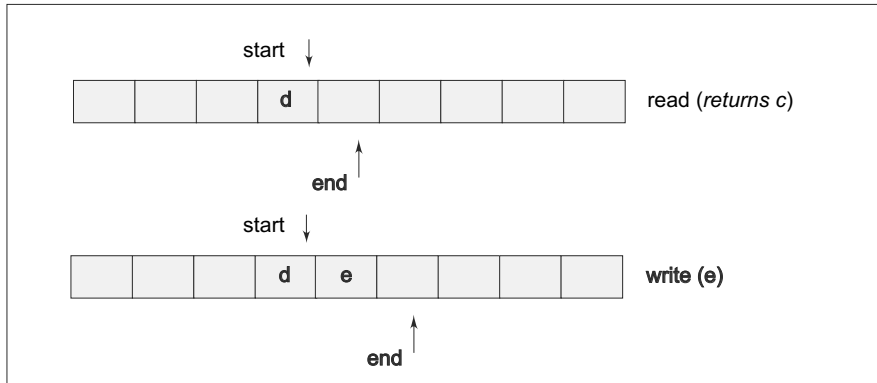


Fig. 8.10 Pointer movements because of queue operations–Part II

8.8 QUEUE IMPLEMENTATION

A typical implementation of the write or add operation checks if there is still room left in the queue, and if there is any, it adds the received item at the end of the queue, and increments the end counter by one. Similarly, the implementation of read or remove operation checks whether or not the queue is already empty, if it is not, it returns the starting element of the queue and makes the start pointer point to the next element in the queue.

We can implement queues by using arrays or linked lists.

8.8.1 Array Implementation of Queues

The array implementation of queues involves allocating a fixed size array in the memory. Both queue operations (add and remove) are made on this array with a constant check being made to ensure that the array does not go out of bounds.

Add Operation

The add operation involves checking whether or not the end pointer is pointing at the upper bound of the array. If it is not, the new item is added at the end of the queue and the end pointer is incremented by one.

Example 8.19 Write a function in C to implement the add operation under array representation of queues.

Program

```
/*Add function*/
void add(int value)
{
    if(front==-1) /*Adding element in an empty queue*/
    {
        front = rear = front+1;
        queue[front] = value;
        return;
    }
}
```



```

if(rear==99) /*Checking whether the queue is full*/
    printf("\tQueue is full\n");
else
{
    rear = rear +1;
    queue[rear]=value; /*Adding element at the end*/
}
}

```

Remove Operation

The remove operation involves checking whether or not the start pointer is already pointing at NULL (empty queue). If it is not, the item that is being currently pointed by the start pointer is removed from the queue and the start pointer is reset to the next element in the queue.

Example 8.20 Write a function in C to implement the remove operation under array representation of queues.

Program

```

/*Remove function*/
int rem()
{
    int i;

    if(front==-1) /*Checking whether the queue is empty*/
        return(-9999);

    else
    {
        i=queue[front]; /*Removing the element at the start*/
        front = front+1;
        return(i);
    }
}

```

Implementation

Example 8.21 Write a program to implement a queue using arrays and perform its common operations.

The following program implements a queue using arrays in C. It uses the add (Example 8.20) and rem (Example 8.21) functions for realizing the common queue operations.

8.40 Computer Programming

Program

```
/*Program for implementing queue using arrays*/
#include<stdio.h>
#include<conio.h>

int queue[100]; /*Declaring a 100 element queue array*/
int front=-1;
int rear=-1; /*Declaring and initializing the queue pointers*/

void add(int); /*Declaring a function prototype for adding an element into the queue*/
int rem(); /*Declaring a function prototype for removing an element from the queue*/
void display(void); /*Declaring a function prototype for displaying the elements of a queue*/

void main()
{
    int num1, num2, choice;
    while(1)
    {
        clrscr();
        /*Creating an interactive interface for performing queue operations*/
        printf("\n\nSelect an option\n");
        printf("\n1 - Insert an element into the Queue");
        printf("\n2 - Remove an element from the Queue ");
        printf("\n3 - Display all the elements in the Queue");
        printf("\n4 - Exit");

        printf("\n\nEnter your choice: ");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
            {
                printf("\nEnter the element to be inserted into the queue ");
                scanf("%d",&num1);
                add(num1); /*Adding an element*/
                getch();
                break;
            }
        }
    }
}
```

```
case 2:
{
    num2=rem(); /*Removing an element*/
    if(num2==-9999)
        printf("\n\tQueue is empty!!");
    else
        printf("\n\t%d element removed from the queue\n\t",num2);
    getch();
    break;
}

case 3:
{
    display(); /*Displaying the queue elements*/
    getch();
    break;
}

case 4:
{
    exit(1);
    break;
}

default:
{
    printf("\nInvalid choice.");
    getch();
    break;
}
}

void display()
{
    int k=front;

    if(front==-1)
    {
        printf("\n\tQueue is Empty!!");
    }
}
```

8.42 Computer Programming

```
    return;
}

else
{
    printf("\nElements present in the Queue are:\n");
    /*Printing queue elements*/
    while(k!=rear)
    {
        printf("\t%d",queue[k]);
        k=k+1;
    }
    printf("\t%d",queue[rear]);
}
}
```

Output

Select an option

- 1 - Insert an element into the Queue
- 2 - Remove an element from the Queue
- 3 - Display all the elements in the Queue
- 4 - Exit

Enter your choice: 3

Queue is Empty!!

Select an option

- 1 - Insert an element into the Queue
- 2 - Remove an element from the Queue
- 3 - Display all the elements in the Queue
- 4 - Exit

Enter your choice: 1

Enter the element to be inserted into the queue 1

Select an option

- 1 - Insert an element into the Queue
- 2 - Remove an element from the Queue
- 3 - Display all the elements in the Queue
- 4 - Exit

Enter your choice: 1

Enter the element to be inserted into the queue 2

Select an option

- 1 - Insert an element into the Queue
- 2 - Remove an element from the Queue
- 3 - Display all the elements in the Queue
- 4 - Exit

Enter your choice: 1

Enter the element to be inserted into the queue 3

Select an option

- 1 - Insert an element into the Queue
- 2 - Remove an element from the Queue
- 3 - Display all the elements in the Queue
- 4 - Exit

Enter your choice: 3

Elements present in the Queue are:

1 2 3

Select an option

- 1 - Insert an element into the Queue
- 2 - Remove an element from the Queue
- 3 - Display all the elements in the Queue
- 4 - Exit

8.44 Computer Programming

```
Enter your choice: 2

        1 element removed from the queue

Select an option

1 - Insert an element into the Queue
2 - Remove an element from the Queue
3 - Display all the elements in the Queue
4 - Exit

Enter your choice: 3

Elements present in the Queue are:
        2        3

Select an option

1 - Insert an element into the Queue
2 - Remove an element from the Queue
3 - Display all the elements in the Queue
4 - Exit

Enter your choice: 4
```

8.8.2 Linked Implementation of Queues

The linked implementation of queues involves dynamically allocating memory space at run time while performing queue operations. Since, the allocation of memory space is dynamic, the queue consumes only that much amount of space as is required for holding its elements. This is contrary to array-based queues which occupy a fixed memory space even if there are no elements present. Moreover, the linked implementation of queues also removes the disadvantage of simple array-based queues in which the starting locations of the array go wasted and are not reallocated if there are a series of queue removal operations.

Add Operation

The add operation under linked implementation of queues involves the following tasks:

- Reserving memory space of the size of a queue element in memory
- Storing the added value at the new location
- Linking the new element with the last element of the queue
- Updating the end pointer

Example 8.22 Write a function in C to implement the add operation under linked representation of queues.

Program

```

/*Add function*/
void add(int value)
{
    struct queue *ptr = (struct queue*)malloc(sizeof(struct queue));/*Dynamically de-
    claring a queue element*/

    ptr->element = value; /*Assigning value to the newly allocated queue element*/

    if(front==NULL) /*Adding element in an empty queue*/
    {
        front = rear = ptr;
        ptr->next=NULL;
    }

    /*Updating queue pointers*/
    else
    {
        rear->next = ptr;
        ptr->next = NULL;
        rear = ptr;
    }
}

```

Remove Operation

The remove operation under linked implementation of queues involves the following tasks:

- Checking whether the queue is empty
- Retrieving the start element of the queue
- Updating the start pointer
- Returning the retrieved (removed) value

Example 8.23 Write a function in C to implement the remove operation under linked representation of queues.

Program

```

/*Remove function*/
int rem()
{
    int i;

    if(front==NULL) /*Checking whether the queue is empty*/

```

```

    return(-9999);

else
{
    i=front->element; /*removing element from the start*/
    front = front->next;
    return(i);
}
}

```

Implementation

Example 8.24 Write a program to implement a queue using linked lists and perform its common operations.

The following program implements a queue using linked lists in C. It uses the add (Example 8.23) and rem (Example 8.24) functions for realizing the common queue operations.

Program

```

/*Program for implementing queue using linked list*/
#include<stdio.h>
#include<conio.h>

struct queue /*Declaring the structure for queue elements*/
{
    int element;
    struct queue *next; /*Queue element pointing to another queue element*/
};

struct queue *front=NULL;
struct queue *rear = NULL;

void add(int); /*Declaring a function prototype for adding an element into the queue*/
int rem(); /*Declaring a function prototype for removing an element from the queue*/
void display(void); /*Declaring a function prototype for displaying the elements of a queue*/

void main()
{
    int num1, num2, choice;
    while(1)

```



```

{
    clrscr();
    /*Creating an interactive interface for performing queue operations*/
    printf("\n\nSelect an option\n");
    printf("\n1 - Insert an element into the Queue");
    printf("\n2 - Remove an element from the Queue ");
    printf("\n3 - Display all the elements in the Queue");
    printf("\n4 - Exit");

    printf("\n\nEnter your choice: ");
    scanf("%d", &choice);

    switch(choice)
    {
        case 1:
        {
            printf("\nEnter the element to be inserted into the queue ");
            scanf("%d",&num1);
            add(num1); /*Adding an element*/
            getch();
            break;
        }

        case 2:
        {
            num2=rem(); /*Removing an element*/
            if(num2==-9999)
                printf("\n\tQueue is empty!!");
            else
                printf("\n\t%d element removed from the queue\n\t",num2);
            getch();
            break;
        }

        case 3:
        {
            display(); /*Displaying queue elements*/
            getch();
            break;
        }
    }
}

```

```

    case 4:
    {
        exit(1);
        break;
    }

    default:
    {
        printf("\nInvalid choice.");
        getch();
        break;
    }
}
}

void display()
{
    struct queue *ptr=front;
    if(front==NULL)
    {
        printf("\n\tQueue is Empty!!");
        return;
    }

    else
    {
        printf("\nElements present in the Queue are:\n");
        /*Printing queue elements*/
        while(ptr!=rear)
        {
            printf("\t%d",ptr->element);
            ptr=ptr->next;
        }
        printf("\t%d",rear->element);
    }
}

```

Output

The output of the above program is same as the output of the program shown in Example 8.22.



Just Remember

- A queue is called as a First In First Out (FIFO) structure. In contrast, a stack is called as a Last In First Out (LIFO) structure.
- The most notable use of stacks is in system software (such as compilers, operating systems, etc).
- A pop (i.e. read) operation in the case of a stack is destructive. That is, once an item is popped from the stack, it is no longer available.
- A queue is a linear, sequential list of items that are accessed in the order *First In First Out (FIFO)*.
- A list/queue can also be circular in which case, it is called as a **circular linked list** or a **circular queue**.
- Arrays and lists are called linear data structures.
- The ordinary mathematical expression is called infix notation.
- In prefix notation the operators are placed before the operands.
- In postfix notation the operators are placed after the operands.
- Use the **sizeof** operator to determine the size of a linked list.
- When using memory allocation functions **malloc** and **calloc**, test for a NULL pointer return value. Print appropriate message if the memory allocation fails.
- Never call memory allocation functions with a zero size.
- Release the dynamically allocated memory when it is no longer required to avoid any possible “memory leak”.
- Using **free** function to release the memory not allocated dynamically with **malloc** or **calloc** is an error.
- Use of a invalid pointer with **free** may cause problems and, sometimes, system crash.
- Using a pointer after its memory has been released is an error.
- It is an error to assign the return value from **malloc** or **calloc** to anything other than a pointer.
- It is a logic error to set a pointer to NULL before the node has been released. The node is irretrievably lost.
- It is an error to declare a self-referential structure without a structure tag.
- It is an error to release individually the elements of an array created with **calloc**.
- It is a logic error to fail to set the link filed in the last node to null.



Multiple Choice Questions

- The prefix equivalent for the postfix $ab+cd+*$ is
 - $a+b*c+d$
 - $+ab*+cd$
 - $*+ab+cd$
 - $*++abcd$
- The postfix equivalent for the prefix $*++abcd$ is
 - $ab+c+d*$
 - $abcd++*$
 - $ab+cd+*$
 - $ab+c*d+$
- The infix equivalent to the postfix expression $abc+d-*e\%/f/$ is
 - $a+b*c-d\%/f/f$
 - $a*(b+c-d)\%/e/f$
 - $a*b+c-d\%/e/f$
 - $a*(b-c+d)\%/e/f$
- Evaluate the expression $2*3/5+6-4$
 - 1
 - 2
 - 3
 - 4
- The value of the prefix expression $+/*2-5\ 6\ 4\ 3$ is
 - 1
 - 2
 - 3
 - 4

8.50 Computer Programming

6. The value of the postfix expression $1\ 4\ +\ 3\ /\ 2\ *\ 6\ 4\ \% -$ is
 - (a) 1
 - (b) -1
 - (c) 0
 - (d) 4
7. Towers of Hanoi is an application of
 - (a) Stack
 - (b) Queue
 - (c) Linked list
 - (d) Dequeue
8. The data structure used in railway reservation is
 - (a) Stacks
 - (b) Queues
 - (c) Priority queues
 - (d) Binary tree
9. The data structure applicable for a fully packed bus is
 - (a) Stacks
 - (b) Queues
 - (c) Priority queues
 - (d) Binary tree
10. The recursive functions are evaluated using
 - (a) Stacks
 - (b) Queues
 - (c) Priority queues
 - (d) Binary tree
11. The nested loops are evaluated using
 - (a) Stacks
 - (b) Queues
 - (c) Structures
 - (d) Binary tree
12. The data structure used in resource sharing systems is
 - (a) Stacks
 - (b) Queues
 - (c) Arrays
 - (d) Binary tree
13. Which of the following is not a linear data structure?
 - (a) Stacks
 - (b) Queues
 - (c) Linked list
 - (d) Binary tree
14. In evaluation of postfix expression the data structure used is
 - (a) Stacks
 - (b) Queues
 - (c) Arrays
 - (d) Binary tree
15. Linked list uses type of memory allocation
 - (a) static
 - (b) random
 - (c) dynamic
 - (d) compile time
16. The number of extra pointers required to reverse a singly linked list is
 - (a) 1
 - (b) 2
 - (c) 3
 - (d) 4
17. The number of extra pointers required to reverse a double linked list is
 - (a) 1
 - (b) 2
 - (c) 3
 - (d) 4
18. The functions used for memory allocation are
 - (a) malloc
 - (b) calloc
 - (c) a and b
 - (d) none of the above
19. Linked lists use _____ type of structures.
 - (a) nested
 - (b) self-referential
 - (c) simple
 - (d) unions
20. _____ cannot be used to represent linked lists.
 - (a) arrays
 - (b) structures
 - (c) unions
 - (d) all the above
21. $\text{calloc}(m,n)$ is equivalent to
 - (a) $\text{malloc}(m*n,0)$
 - (b) $\text{memset}(0,m*n)$
 - (c) $\text{ptr}=\text{malloc}(m*n)$
 - (d) $\text{malloc}(m/n)$



Review Questions

- 8.1 Explain in detail the push and pop operations of stack with a real life example.
- 8.2 Describe how items in a queue are accessed.
- 8.3 Describe about circular queue.
- 8.4 Define infix and prefix notation.
- 8.5 Explain about the evaluation of postfix notation.
- 8.6 State whether the following statements are True or False
 - (a) Items in a stack exit in the same order in which they had entered the stack.
 - (b) Stacks are rarely used in computer applications.
 - (c) Writing a value to the stack is called as a *pop* operation, whereas reading a value from it is called as a *push* operation.
 - (d) Once an item is popped from the stack, it is no longer available.

- (e) Pop operation increments the stack counter by one to see if the stack is empty.
 - (f) The stack can be implemented using the dynamic memory management techniques.
 - (g) Dynamically allocated memory can only be accessed using pointers.
 - (h) **calloc** is used to change the memory allocation previously allocated with **malloc**
 - (i) Only one call to free is necessary to release an entire array allocated with **calloc**
 - (j) Memory should be freed when it is no longer required.
 - (k) To ensure that it is released, allocated memory should be freed before the program ends.
 - (l) The link field in a linked list always points to successor.
 - (m) The first step in adding a node to a linked list is to allocate memory for the next node.
- 8.7 Fill in the blanks in the following statements.
- (a) Function _____ is used to dynamically allocate memory to arrays.
 - (b) A _____ is an ordered collection of data in which each element contains the location of the next element.
 - (c) Data structures which contain a member field that points to the same structure type are called _____ structures.
 - (d) A _____ identifies the last logical node in a linked list.
 - (e) Stacks are referred to as _____
- 8.8 What is a linked list? How is it represented?
- 8.9 What is dynamic memory allocation? How does it help in building complex programs?
- 8.10 What is the principal difference between the functions **malloc** and **calloc**
- 8.11 Find errors, if any, in the following memory management statements:

- (a) `*ptr = (int *)malloc(m, sizeof(int));`
- (b) `table = (float *)calloc(100);`
- (c) `node = free(ptr);`

8.12 Why a linked list is called a dynamic data structure? What are the advantages of using linked lists over arrays?

8.13 Describe different types of linked lists.

8.14 Identify errors, if any, in the following structure definition statements:

```
struct
{
    char name[30]
    struct *next;
};
typedef struct node;
```

8.15 The following code is defined in a header file *list.h*

```
typedef struct
{
    char name[15];
    int age;
    float weight;
}DATA;
struct linked_list
{
    DATA person;
    Struct linked_list *next;
};
typedef struct linked_list NODE;
typedef NODE *NDPTR;
```

Explain how could we use this header file for writing programs.

8.16 What does the following code achieve?

```
int * p ;
p = malloc (sizeof (int) ) ;
```

8.17 What does the following code do?

```
float *p;
p = calloc (10,sizeof(float) ) ;
```

8.18 What is the output of the following code?

```
int i, *ip ;
ip = calloc ( 4, sizeof(int) );
for ( i = 0 ; i < 4 ; i++)
    *ip++ = i * i;
for ( i = 0 ; i < 4 ; i++)
    printf( "%d\n", *--ip );
```

8.19 What is printed by the following code?

```
int *p;
```

8.52 Computer Programming

```
p = malloc (sizeof (int) );
*p = 100 ;
p = malloc (sizeof (int) );
*p = 111;
printf("%d", *p);
```

8.20 What is the output of the following segment?

```
struct node
{
    int m ;
    struct node *next;
} x, y, z, *p;
x.m = 10 ;
```

```
y.m = 20 ;
z.m = 30 ;
x.next = &y;
y.next = &z;
z.next = NULL;
p = x.next;
while (p != NULL)
{
    printf("%d\n", p -> m);
    p = p -> next;
}
```



Programming Exercises

- 8.1 Rewrite the function print() using iterative technique in for loop.
- 8.2 Write a menu driven program to create a linked list of a class of students and perform the following operations:
 - (a) Write out the contents of the list.
 - (b) Edit the details of a specified student.
 - (c) Count the number of students above a specified age and weight.Make use of the header file defined in Review Question 8.15.
- 8.3 Write recursive and non-recursive functions for reversing the elements in a linear list. Compare the relative efficiencies of them.
- 8.4 Write an interactive program to create linear linked lists of customer names and their telephone numbers. The program should be menu driven and include features for adding a new customer and deleting an existing customer.
- 8.5 Modify the above program so that the list is always maintained in the alphabetical order of customer names.
- 8.6 Develop a program to combine two sorted lists to produce a third sorted lists which contains one occurrence of each of the elements in the original lists.
- 8.7 Write a program to create a circular linked list so that the input order of data items maintained. Add function to carry out the following operations on circular linked list.
 - (a) Count the number of nodes
 - (b) Write out contents
 - (c) Locate and write the contents of a given node
- 8.8 Write a program to construct an ordered doubly linked list and write out the contents of a specified node.
- 8.9 Write a function that would traverse a linear singly linked list in reverse and write out the contents in reverse order.
- 8.10 Given two ordered singly linked lists, write a function that will merge them into a third ordered list.
- 8.11 Write a function that takes a pointer to the first node in a linked list as a parameter and returns a pointer to the last node. NULL should be returned if the list is empty.
- 8.12 Write a function that counts and returns the total number of nodes in a linked list.
- 8.13 Write a function that takes a specified node of a linked list and makes it as its last node.
- 8.14 Write a function that computes and returns the length of a circular list.
- 8.15 Write functions to implement the following tasks for a doubly linked list.
 - (a) To insert a node.
 - (b) To delete a node.
 - (c) To find a specified node.

APPENDIX

A

A.1 VOID POINTERS

Every pointer is associated with a certain data type, for instance an integer pointer points to an integer type variable while a character pointer points to a character type variable. However, there could be situations when the data type for a pointer is not known beforehand. In such cases, a generic pointer is declared that may point to any data type, i.e. int, char or float. Such a generic pointer is termed as void pointer. It points to a raw memory address which may contain any data type value.

Example A.1 Write a program to demonstrate the use of void pointers.

Program

```
/*Program demonstrating void pointers*/
#include <stdio.h>
#include <conio.h>

void main()
{
    int a=10;
    char b='T';
    void *v1, *v2;
    clrscr();

    v1=&a; /*Assigning integer type address to void pointer*/
    v2=&b; /*Assigning character type address to void pointer*/
```

A.2 Computer Programming

```
printf("a = %d\n",*(int *)v1);
printf("b = %c\n",*(char *)v2);
getch();

}
```

Output

```
a = 10
b = T
```

A.2 LEVEL DISK INPUT/OUTPUT

Low-level disk i/o is another way of working with files stored in the disk that is faster as compared to normal high-level file i/o. It uses an intermediate buffer to store the data during i/o operations. The use of buffer greatly speeds up the read and write operations. The size of the buffer is specified programmatically based on requirements. io.h header file contains the functions that support low-level disk i/o, thus it needs to be included at the time of writing such programs. The key functions contained in io.h are:

- **open:** Opens a file and returns an integer reference id pertaining to the file. All successive references to the file are made by using this id. It returns -1 if a file is not successfully opened. Similar to fopen function, the open functions also uses certain flags to specify the mode in which a file is to be opened. For instance, the O_APPEND flag opens the file in append mode while O_RDWR flag opens the file in read/write mode.
- **close:** Closes an already opened file. It returns -1 in case of unsuccessful file close operation.
- **write:** Helps to write into a file. It takes as parameter a pointer to the buffer location. It returns an integer value specifying the number of bytes written. A return value of -1 specifies unsuccessful write operation.
- **read:** Helps to read the contents of a file. The read function also takes as parameter a pointer to the buffer location. The data read from the disk is placed into this buffer. The function returns an integer value specifying the number of bytes written into the buffer. A return value of -1 specifies unsuccessful read operation.

Example A.2 Write a program to demonstrate low-level disk i/o operations.

Program

```
/*Program to demonstrate low-level disk i/o*/
#include <stdio.h>
#include <conio.h>
#include <io.h>

void main()
```



```

{
    char f_name[12] = "Test.txt"; /*Declaring character array containing file name*/
    char buffr[20]; /*Declaring buffer of size 20 characters*/
    int i,r;

    r=open(f_name,O_CREATE,O_TEXT); /*Opening the file Test.txt*/
    if(r==-1) /*Checking whether file is successfully opened*/
    {
        printf("File can not be opened");
        getch();
        exit();
    }
    else
    {
        printf("Enter the text to be written into the file: ");
        for(i=0;i<=18;i++)
            buffr[i]=getche(); /*Reading input from console*/
        buffr[i]='\0';
        write(r,buffr,20); /*Performing disk write operation*/
        close(r);
        printf("\nFile successfully written!!");
        getch();
    }
}

```

Output

```

Enter the text to be written into the file: This is sample text
File successfully written!!

```

A.3 THREE BASIC OPERATIONS

The tea-making algorithm and flow chart discussed earlier were quite simple. One step followed another in a sequential fashion. However, things are not so simple in real life! There are so many *ifs* and *buts*. For example, consider the following statements:

- If it is 9 am, I must go to the office.
- If it is raining, take your umbrella with you.
- Read each line and say it loudly until the end of this page.

How do we take care of such things in an algorithm and a flow chart? And how many different kinds of situations we must cater to? This section attempts to answer these questions.

In general, the steps in an algorithm can be divided in three basic categories as listed below:

- **Sequence**—A series of steps that we perform one after the other

A.4 Computer Programming

- **Selection**—Making a choice from multiple available options
- **Iteration**—Performing repetitive tasks

These three basic categories of activities combined in different ways can form the basis for describing *any* algorithm. It might sound surprising initially. But it is true. Think of any situation in our daily life and try to fit it in one of the three categories: it works!

Let us now look at each of the categories in more detail.

Sequence

A sequence is a series of steps that we follow in any algorithm without any break, i.e. unconditionally. The algorithm for making tea described in Fig. A.1 belongs to this category. Figure A.3 describes another algorithm in this category for ‘boiling water’. What this means is that we have *exploded* further the step or instruction 1 in the algorithm for making tea given in Fig. A.1. We can explode all such steps in Fig. A.1 in the following way.

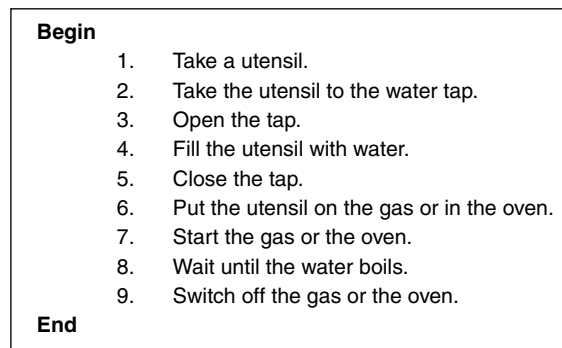


Fig. A.3 Algorithm for boiling water

We will not draw the flow chart for boiling water. It should be clear by now that it is a matter of writing all of the above 9 steps in rectangles one after the other; each connected to its successor by an arrow.

Selection

We will quickly realize that only ‘sequence’ is not good enough to express an algorithm. Quite a few of our actions depend on certain events. Thus, there is a need to be able to make a choice from many available options. Therefore, there is a process of selection. A selection statement generally takes the form as shown in Fig. A.4.

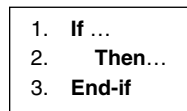


Fig. A.4 Selection

We take so many decisions, small and big, in our daily life without even realizing it. For example, if it is hot, we switch on the fan. We can depict this selection process as shown in Fig. A.5.

1. **If** it is hot
2. **Then** switch on the fan
3. **End-if**

Fig. A.5 Example of a selection

Note that *End-if* is an additional step that indicates the end of the selection process. A question may be asked: is *End-if* really necessary? Let us illustrate this by the following selection process that does not have an *End-if*. Refer to Fig. A.6.

1. **If** the guest wants tea
2. **Then** make tea
3. Offer biscuits

Fig. A.6 Importance of *End-if*

What do we do if the guest does not want tea? Do we offer him biscuits? It is not clear from the selection process described above. It can be argued and proved either way. That is, we are not sure whether the *Offer biscuits* portion is a part of our *If* condition or not. A miserly person would always say that he considers *Offer biscuits* as a part of the condition, and hence would only offer biscuits to someone who wants tea! Hence, it is always better to end a selection process with an *End-if* to avoid any confusion at least for our algorithms. Different computer programming languages have different conventions.

The position and placement of *End-if* instructions can change the meaning of the entire algorithm as shown in Fig. A.7 (a) and A.7 (b). Let us study the algorithm shown in Fig. A.7 (a). If the guest wants tea, the algorithm will execute step 2 after step 1 (i.e. make tea), then fall through step 3 and then step 4 (i.e. offer biscuits). Thus, if the guest wants tea, he gets the biscuits, too! Now, let us trace the algorithm if the guest does not want tea. In this case, the algorithm will follow step 3 after step 1 (i.e. skip the tea) and then fall through step 4 (i.e. offer biscuits). Thus, biscuits will be offered regardless of tea. If we study algorithm shown in Fig. A.7 (b), we will notice that biscuits are offered only with tea. If the guest does not want tea, the algorithm will follow step 1 and directly step 4. The semicolon at the end of step 2 indicates that tea and biscuits are offered together. You will notice that the positioning of *End-if* (at step 3 or 4) has made all the difference! The Figs. A.8 (a) and A.8 (b) show the corresponding flow charts.

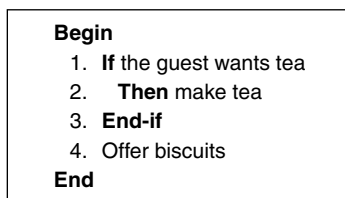


Fig. A.7(a) Offer biscuits to all guests

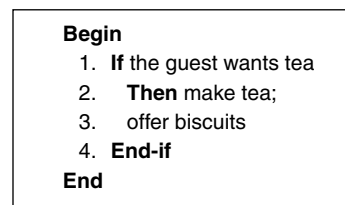


Fig. A.7(b) Offer biscuits only to guests who want tea

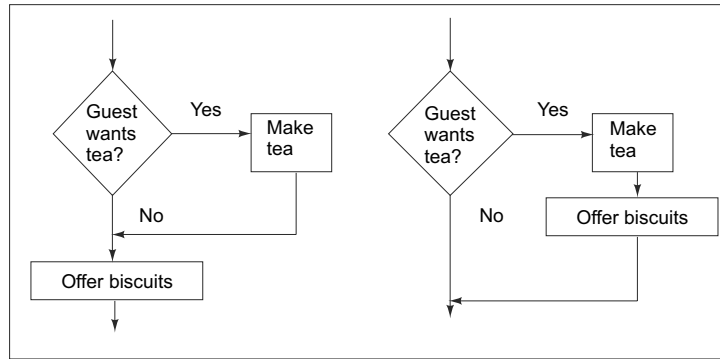


Fig. A.8(a)

Fig. A.8(b)

In both of these cases (a and b), we have not specified the action if the guest wants coffee. Let us modify our algorithm and also the flow chart to accommodate that possibility. Note that we are offering coffee as an alternative to tea. We will be generous enough to offer biscuits in either case. Since we want to offer the biscuits irrespective of the acceptance/rejection of tea/coffee offer, we should consider only the first algorithm for modification, viz. Fig. A.7(a). The modified algorithm and its corresponding flow chart are shown in Figs. A.9 and A.10 respectively.

Note that we offer biscuits regardless of the acceptance of the tea/coffee offer. That is exactly what we had wanted. Consider the following algorithm as shown in Fig. A.11(a). Figure A.11(b) shows the corresponding flow chart.

Can it achieve the same objective as the algorithm shown in Fig. A.9? A close examination will reveal that it is *not* the same. For instance, if the guest rejects both tea and coffee, the algorithm shown in Fig. A.9 offers the guest biscuits regardless. However, the algorithm shown in Fig. A.11(a) does not offer anything! How easily things that seem similar can mislead us, as they actually are different. These are the things that one needs to be careful of while writing an algorithm. Otherwise, it leads to the programming errors called as **bugs**, which programmers refer to, all the time.

We will notice some problem in the algorithm and flow charts depicted in Figs A.9–A.11. After a guest chooses to have tea, the algorithm still asks if the guest wants coffee. Unless a guest wants both simultaneously, this is completely wasteful. We can avoid this by introducing a **goto** instruction. Using this, we can rewrite the algorithm of Fig. A.9 as shown in Figs. A.12 (a) and A.12 (b).

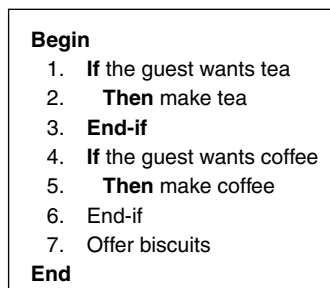


Fig. A.9 Algorithm for offering tea/coffee and biscuits

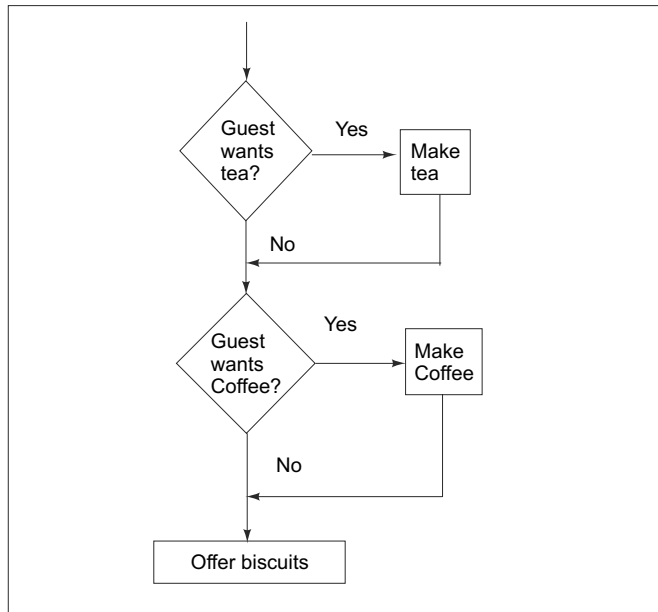


Fig. A.10 Flow chart for offering tea/coffee and biscuits

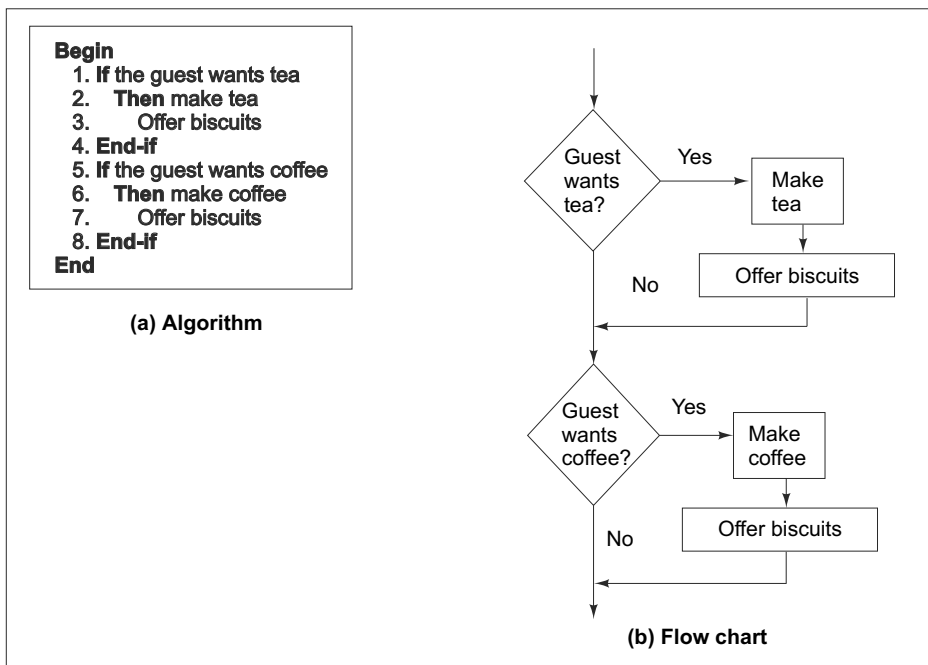


Fig. A.11 Offering biscuits only if guest wants tea /coffee

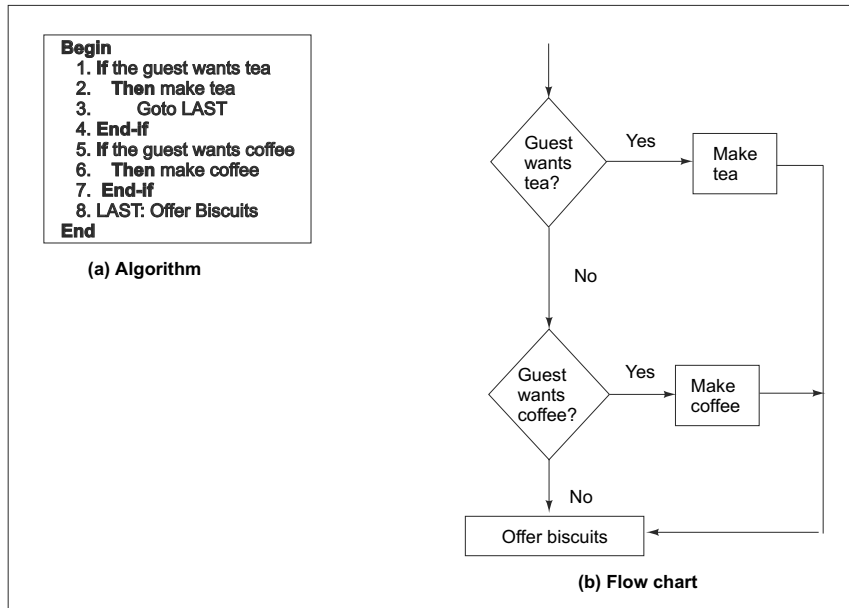


Fig. A.12 Improved algorithm to avoid unnecessary step

In this algorithm, *LAST* is called **label** for an instruction. *Offer biscuits* (step 8). This allows the *goto* instruction to branch to a specific instruction. However, the *goto* makes a program difficult to understand and therefore, modify. So, generally, *gotos* are avoided. If you avoid such a *goto* in a program, you can read the program starting at the top and ending at the bottom without branching up or down. This is called as **goto-less** or **top-down** or **structured** programming method. Such programs are easy to understand and therefore, easy to maintain. We will see how the same algorithm can be written in a structured fashion later.

The compound conditions

It is very easy to group different conditions into one. We generally use words like *and* and/or *or* to join sentences. In the same way, we can combine two or more conditions into a single **compound condition**. For example, take a look at the algorithm shown in Fig. A.13.

```

1. If it is a weekday
2.   and it is 7 am or more
3.   and you are feeling ok
4. Then
5.   Take breakfast
6.   Go to work
7. End-if
  
```

Fig. A.13 Compound conditions with and

In this algorithm, you are doing two things together, viz. taking breakfast and going to work. However, you perform these only if all the three conditions are met, viz. it has to be a weekday, it has to be at least 7 am and you have to be feeling good. If any of these is not valid, you will not perform both of these actions. For instance, if it is 7 am, and you are feeling ok, but if it is a weekend, you will not perform both the actions. But then if you take the algorithm very seriously in real life, you will notice a bug. You will not go to work on a weekend all right, but you will also have to skip your breakfast! So, be careful!

Similarly, the algorithm in Fig. A.14 illustrates the use of *or* word to join two or more conditions. (Note that we are now assuming implicit *Begin* and *End* statements for our algorithms.)

```

1. If it is a weekday
2.   or work is pending
3.   or boss is angry
4. Then
5.   Take breakfast
6.   Go to work
7. End-if

```

Fig. A.14 Compound conditions with *or*

In this case, even if any of the conditions is met, you have breakfast and go to work. For instance, if your boss is angry, then even if it is a holiday and there is actually no pending work, you will still perform these two actions. Only if all the three conditions are not met, you skip having breakfast and going to work.

The nested conditions

Sometimes, things are not very easy to express. With a variety of situations to worry about, it becomes really complicated. For example, if something depends on still something else to happen (or otherwise), there will be an *if* within another *if*. We will call it a **nested condition**. Let us look at an example.

```

1. If you are feeling ok
2. Then go to work
3. Else
4.   If you have fever
5.     Then go to the doctor
6.     Else
7.       Just relax
8.   End-if
9. End-if

```

Fig. A.15 Nested conditions

You will notice that there is an *If* at the beginning (step 1) and corresponding *End-if* at the bottom (step 9), which denotes the end of that *If* condition. Within this, you will notice another *If* and *End-if* pair written in the same columns (steps 4 and 8). The *Else* provides an alternative path of execution. Let us trace the algorithm if you are ok. In this case, the algorithm executes step 2, skips steps 3 through 8 and goes directly to step 9. If you are not ok but you do not have fever, the algorithm goes through steps 1, 3, 4, 6, 7, 8 and 9. You will not go to work and you will just relax, but you will not consult a doctor even if you have stomach pain. This is a bug!

If you are not ok and if you have fever, you will go through steps 1, 3, 4, 5, 8 and 9. In this case, you will not go to work, you will not consult the doctor, but you will not relax. So, there is another bug. Try removing it!

Let us rewrite the first algorithm of Fig. A.6 using nested conditions. This will make the benefit of *else* clearer.

```

1. If the guest wants tea
2.   Then make tea
3.   Else
4.     If the guest wants coffee
5.       Then make coffee
6.       End-if
7.   End-if
8. Offer biscuits

```

Fig. A.16 The 'tea-coffee' algorithm using nested conditions

A.10 Computer Programming

As can be seen, if the guest accepts the offer for tea, we would simply make tea and offer biscuits without bothering if he wants coffee (unless the guest wants both!). This does not happen in Fig. A.6. It might not make a big difference if we have just one or two conditions to check. However, as the number of conditions increases, it is desired that the algorithm be as compact and efficient as possible.

As a further illustration, let us write an algorithm to find the largest of any three given numbers. It is very easy to write an algorithm to find the larger of two numbers. Assuming that the two numbers are *a* and *b*, we simply need to compare the two. However, when there are three numbers, things are a bit complicated. The algorithm is as shown in Fig. A.17.

```
1. If a > b
2.   Then If a > c
3.     Then choose a
4.     Else choose c
5.   End-if
6. Else If b > c
7.   Then choose b
8.   Else choose c
9. End-if
10. End-if
```

Fig. A.17 Selecting the largest of three numbers

To ascertain that the algorithm indeed works as expected, we might assign values as 15, 2 and 21 to *a*, *b* and *c* respectively. When we imagine these numbers in place of *a*, *b* and *c* in the above algorithm and check if the algorithm works correctly, it is called as a **pencil run** of our algorithm. It is also termed as a **walkthrough** of an algorithm. It is always a good practice to do a walkthrough of an algorithm with a variety of values. That is, we should take *a*, *b* and *c* such that once *a* is the greatest of them all (e.g. *a* = 7, *b* = 2 and *c* = 1), then *b* (e.g. *a* = 13, *b* = 20, *c* = 10) and then *c* (e.g. *a* = 18, *b* = 24, *c* = 49). This will make sure that our algorithm gives the correct result in each case. When we are doing a walkthrough of our algorithm in this fashion, we are actually checking if it works fine.

Testing and debugging

What the above discussion means is, we are checking to see if our algorithm gives the desired result. Hence, we are **testing** our algorithm. A set of values of *a*, *b* and *c* that we use for testing (e.g. *a* = 18, *b* = 24, *c* = 49) is called as a **test case** as we test our algorithm for a possibility (or case) at a given time. A group of test cases makes up the **test data**.

If for any one of such conditions, our algorithm gives an incorrect result, we say that there is a **bug** in the algorithm—it is another name for an error. It is necessary then to correct this bug, that is, change the algorithm accordingly and do a walkthrough again. Now the algorithm should give the desired result. This process of removing a bug from an algorithm is called as **debugging**.

To understand this, let us deliberately introduce a bug in our algorithm. The modified algorithm to find out the largest of three numbers is shown in Fig. A.18. The statements in *Italics* are interchanged from Fig. A.17.

```
1. If a > b
2.   Then If a > c
3.     Then choose a
4.     Else choose c
5.   End-if
6. Else If b > c
7.   Then choose c
8.   Else choose b
9. End-if
10. End-if
```

Fig. A.18 A bug in an algorithm

Let us test our modified algorithm as shown in Fig. A.18 with values as $a=1$, $b=5$ and $c=9$. You will realize that our algorithm gives a result that b is the largest of the three. This is, of course, incorrect. This is how debugging helps us in identifying and correcting mistakes.

Indentation and Algorithm maintenance

A question: why write *end-if* exactly below *if*? Why to start some sentences at the beginning and some after leaving some spaces? The algorithm could as well be written as shown in Fig. A.19.

```

1. If the guest wants tea
2. Then make tea
3. Else
4. If the guest wants coffee
5. Then make coffee
6. End-if
7. End-if
8. Offer biscuits

```

Fig. A.19 Indentation

Well, the guest might end up having to drink coffee when he actually wanted tea! The obvious problem with this style of writing is that it is not easy to follow. That is, it is not **indented**. Further, if someone is to change it (add another drink, for example), it is not easy to see immediately where to make changes. That, in software terms means an algorithm that is difficult to **maintain**.

Iteration

Let us imagine a different situation. Suppose that I have asked my friend to wait for me until I arrive. Then we two would go to see a movie. How would my friend write an algorithm for this situation? At first, it might appear that one should be able to do this with the help of (a series of) If-Then-Else statements. Let us say the friend decides to check my arrival every two minutes. So, the algorithm in Fig. A.20 should suffice.

```

1. If I arrive
2. Then we go for a movie
3. Else wait for two minutes
4. If I arrive
5. Then we go for a movie
6. Else wait for two minutes
7. If I arrive
8. Then...

```

Fig. A.20 Need for iteration

Wait a minute. How many times should this be written? If I arrive two hours hence, for example, the above instructions would have to be written 60 times! But if I come after four hours from now, they would be repeated 120 times!! This is like imagining what if the clocks were not round. Think that a clock to be a straight line. After 12, it is 1 all right, but the hourly clock hand moves straight (for simplicity, disregard the seconds and minutes). So, we have:

```

1 2 3 4 5 6 7 8 9 10 11 12 1 2 3 4 5 6 7 8 9 10 11 12

```

As can be seen, just for one day, we would need 24 sets of 1-12. We have the same problems with our algorithm:

A.12 Computer Programming

- It is too long and repetitive
- We do not have any control over it as we are not sure when it will end

An iteration algorithm would solve these problems. There are many ways to write this algorithm. Figure A.21 shows all of them. Let us see draw the corresponding flow chart also.

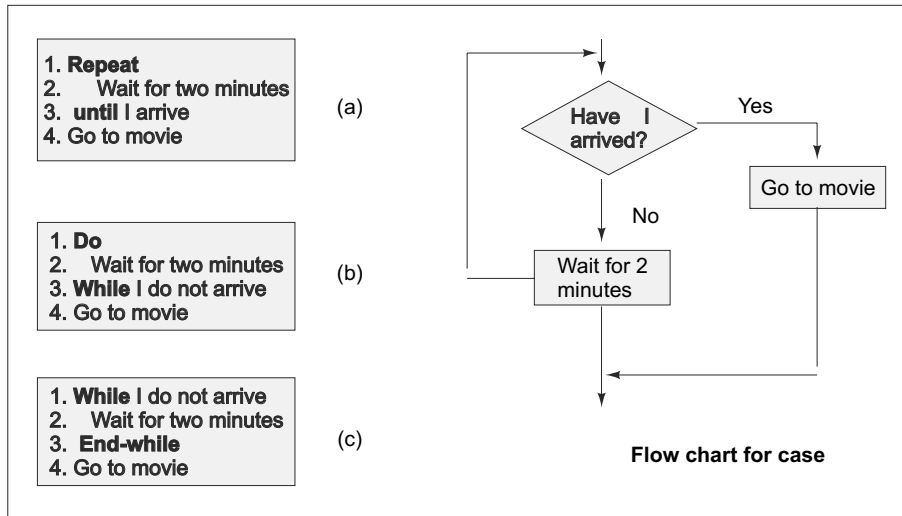


Fig. A.21 Iteration

Repeat-until and Do-while (cases a and b) are essentially the same. Both check the success/failure using common terms like *until* and *while*. However, they differ significantly from (c) (While-End-while). We shall now see, why. What happens if I arrive before the algorithm was executed – that is, at the beginning of the algorithm? In (a) and (b), my friend will still wait for two minutes! The algorithms first ask my friend to wait for two minutes and then check if I have arrived. On the other hand, case (c) first checks to see if I have arrived. Only then does it ask my friend to wait for two minutes. Note that the flow chart is drawn to suit style (c). We could as well draw it for the first two (wait first and then check). It must be pointed out that algorithms (a) and (b) are not bad at all. In some situations we could prefer case (a) or (b). In others, we would opt for (c).

To summarize, sequence, selection and iteration form the building blocks for writing any algorithm. These three basic types of instructions can deal with virtually any situation and can put it in words. There are many different ways to write the same algorithm. Our aim should be to make it as compact and understandable as possible without any loss of meaning.

Code.No: 09AIEC01

R09

SET-1

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

I B.TECH - REGULAR EXAMINATIONS, JUNE - 2011

C PROGRAMMING AND DATA STRUCTURES

(COMMON TO CE, EEE, ME, ECE, CSE, CHEM, EIE, BME, AE, AME, BT, ECOMPE, ETM, IT, ICE, MCT, MMT, MIE, MIM)

Time: 3hours

Max.Marks:75

**Answer any FIVE questions
All questions carry equal marks**

1. (a) What is an algorithm? Write an algorithm to read five integers and find out if the values are in ascending order.
(b) Draw a flow chart to read ten integer values and print the sum of squares of the values. [8+7]
2. (a) Write minimal C expressions for the following:
 - (i) $2x^4 + 3x^3 - 4x^2 + 7x - 10$
 - (ii) Digit at the 10's place of the given positive integer x (for example, digit at the 10's place in 3458 is 5)
 - (iii) True if the given positive integer x is odd, false otherwise
 - (iv) Add x to y , and then decrement x
 - (v) True if $5 \leq a \leq 10$, false otherwise
 - (vi) Fourth bit from the right if the number x is treated in binary representation.(b) Write a complete C program to print all the prime numbers between 1 and n . Where ' n ' is the value supplied by the user. [6+9]
3. (a) Explain the following storage classes with examples: auto, register, extern.
(b) Explain how two dimensional arrays can be used to represent matrices. Write C code to perform matrix addition and matrix multiplication. [9+9]
4. (a) Consider the function *maxpos* that has two parameters: *int maxpos(int arr[], int n)* n is greater than or equal to 1, but less than or equal to the size of the array *arr*. Code the function *maxpos* to return the position of the first maximum value among the first n elements of the array *arr*.
(b) What are command line arguments? Illustrate their use with a simple C program. [9+6]
5. Explain the following with example:
 - (a) Nested structures
 - (b) Array of structures
 - (c) Unions[5+5+5]
6. (a) List and explain the Streams functions for text files along with their prototypes.
(b) Write a complete C program to copy data from one file to another file. The name of the source file and the name of the destination file are supplied by the user. [6+9]

SET 1.2 Computer Programming

7. (a) Explain bubble sort with the algorithm or a C program.
(b) Illustrate the results of bubble sort for each pass, for the following initial array of elements:
68 67 99 33 122 200 [9+6]
8. (a) Explain what is stack and the operations performed on stack.
(b) Explain how a stack be implemented using arrays. [7+8]

ANSWERS

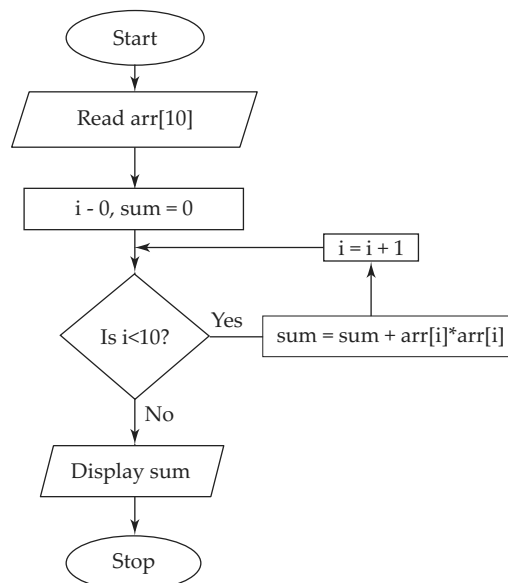
1 (a). Algorithm

An algorithm is a sequence of steps written in the form of English phrases that specify the tasks that are performed while solving a problem. It involves identifying the variable names and types that would be used for solving the problem. Algorithms help a programmer in breaking down the solution of a problem into a number of sequential steps. Corresponding to each step a statement is written in a programming language; all these statements are collectively termed as a program.

Algorithm for Determining Ascending Order

```
Step 1 - Start
Step 2 - Read a 5 element array (arr[])
Step 3 - Initialize the looping counter i = 0
Step 4 - Repeat Steps 5-7 while i<4
Step 5 - If arr[i] <= arr[i+1] goto Step 7 else goto Step 6
Step 6 - Display message "Array elements are not in ascending order" and go to Step 9
Step 7 - i = i + 1
Step 8 - Display message "Array elements are in ascending order"
Step 9 - Stop
```

1 (b). Flowchart



2(a). Here we are assuming that the resultant expression values are stored in the 'result' variable.

(i).

```
result = 2*x*x*x*x + 3*x*x*x - 4*x*x + 7*x - 10
```

(ii).

```
x = x / 10  
result = x % 10
```

(iii).

```
if(x % 2 == 1)  
    printf("True");  
else  
    printf("False");
```

(iv).

```
y = y + x-;
```

(v).

```
if(a>=5 && a<=10)  
    printf("True");  
else  
    printf("False");
```

(vi).

```
result = 8&x;  
if(result>0)  
    printf("bit = 1");  
else  
    printf("bit = 0");
```

2 (b).

Program

```
#include <stdio.h>  
#include <conio.h>  
  
void main()  
{  
    int prime(int num);  
    int n,i;
```

SET 1.4 Computer Programming

```
int temp;

printf("Enter the value of n: ");
scanf("%d",&n);

printf("Prime numbers between 1 and %d are:\n",n);
for(i=2;i<=n;i++)
{
    temp=prime(i);
    if(temp==-99)
        continue;
    else
        printf("%d\t",i);
}

getch();
}

int prime(int num)
{
    int j;
    for(j=2;j<num;j++)
    {
        if(num%j==0)
            return(-99);
        else
            ;
    }

    if(j==num)
        return(num);
}
```

Output

```
Enter the value of n: 20
Prime numbers between 1 and 20 are:
2      3      5      7      11     13     17     19
```

3 (a). Storage Classes Refer Section 3A.17

3 (b). Using 2-D Arrays for Matrix representation Refer Section 3B.5

Program for Matrix Addition Refer Section 3B.11 (Example 3B.18)

Program for Matrix Multiplication Refer Section 3B.11 (Example 3B.20)

4 (a).

Program

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

int maxpos(int [], int);
void main()
{
    int n;
    int arr[10] = {12, 3, 45, 8, 19, 10, 7, 23, 29, 5};
    printf("Enter the value of n: ");
    scanf("%d",&n);

    if(n>=1 && n<=10)
        ;
    else
    {
        printf("Invalid value of n...Press any key to terminate the program..");
        getch();
        exit(0);
    }

    printf("Within the first %d elements of array, the first maximum value is
    stored at index %d",n,maxpos(arr,n));
    getch();
}

int maxpos(int a[],int N)
{
    int i;
    int max=0,index;
    for(i=0;i<N;i++)
        if(a[i]>max)
        {
            max=a[i];
            index = i;
        }
    return(index);
}
```

SET 1.6 Computer Programming

Output

```
Enter the value of n: 5
Within the first 5 elements of array, the first maximum value is stored at index 2
```

4 (b). Command Line Arguments Refer Section 5.18

5 (a). Nested Structures Refer Section 5.10

5 (b). Array of Structures Refer Section 5.8

5 (c). Unions Refer Section 5.13

6 (a). Stream functions for text file manipulation

Name	Prototype	Description
fgets	char *fgets(char*, int, FILE*)	Reads a string from an input stream
fputs	int fputs(const char*, FILE*)	Writes a string to an output stream
fscanf	int fscanf (FILE*, "format string", arg_list)	Analogous to scanf function, fscanf is used for reading data from the input stream in the specified format
fprintf	int fprintf (FILE*, "format string", arg_list)	Analogous to printf function, fprintf is used for writing data to the output stream in the specified format

6 (b). Program Refer Section 5.19 (Example 5.28)

7 (a). Bubble Sort Refer Section 7.2.2

7 (b). Applying Bubble Sort

<i>Initial Array:</i>	68	67	99	33	122	200
<i>Pass 1:</i>	33	68	99	67	122	200
<i>Pass 2:</i>	33	67	99	68	122	200
<i>Pass 3:</i>	33	67	68	99	122	200
<i>Pass 4:</i>	33	67	68	99	122	200
<i>Pass 5:</i>	33	67	68	99	122	200 (<i>Sorted Array</i>)

8 (a). Stack and its Operations Refer Section 8.4

8 (b). Array Implementation of Stacks Refer Section 8.5.1

Code.No: 09AIEC01

R09

SET-2

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

I B.TECH - REGULAR EXAMINATIONS, JUNE - 2011

C PROGRAMMING AND DATA STRUCTURES

**(COMMON TO CE, EEE, ME, ECE, CSE, CHEM, EIE, BME, AE, AME, BT, ECOMPE,
ETM, IT, ICE, MCT, MMT, MIE, MIM)**

Time: 3hours

Max.Marks:75

**Answer any FIVE questions
All questions carry equal marks**

1. (a) What is an algorithm? Write an algorithm to find out if a given number is a prime.
(b) Draw a flow chart to read ten positive integers and how many are multiples of 7. [8+7]
2. (a) Write minimal C expressions for the following:
 - (i) $3x^4 + x^3 - 4x^2 + 7x$
 - (ii) Maximum of the values of three variables a, b and c
 - (iii) Digit at the 100's place of the given positive integer x (for example, digit at the 100's place in 3458 is 4)
 - (iv) True if the given positive integer x is even, false otherwise
 - (v) Increment x , and then add to z
 - (vi) True if the given positive integer x is a multiple of 3 and 7, false otherwise.(b) What are the bitwise operators in C? Explain the same with examples. [6+9]
3. (a) What is recursion? Write a complete C program that reads a positive integer, calculate the factorial of the number using recursion, and print the result.
(b) Explain the facilities provided by the C preprocessor with examples. [8+7]
4. (a) Write a complete C program that reads a string and prints if it is a palindrome or not.
(b) Explain about memory allocation functions in C. [8+7]
5. Explain the following with example:
 - (a) Pointers to structures
 - (b) Self referential structures
 - (c) Unions[5+5+5]
6. (a) Explain the different modes that can be provided as a parameter to the *fopen()* function.
(b) Write a complete C program for the following: There are two input files named "first.dat" and "second.dat". The files are to be merged. That is, copy the content of "first.dat" and then the content of "second.dat" to a new file named "result.dat". [6+9]
7. (a) Write a C program or algorithm to sort an array of integers in ascending order using insertion sort.
(b) Illustrate the results of insertion sort for each pass, for the following initial array of elements:
68 57 99 33 122 200 [6+6]

8. What is a singly linked list? Explain with C code how the insertion, deletion and searching operations are performed on a singly linked list. [15]

ANSWERS

1 (a). Algorithm

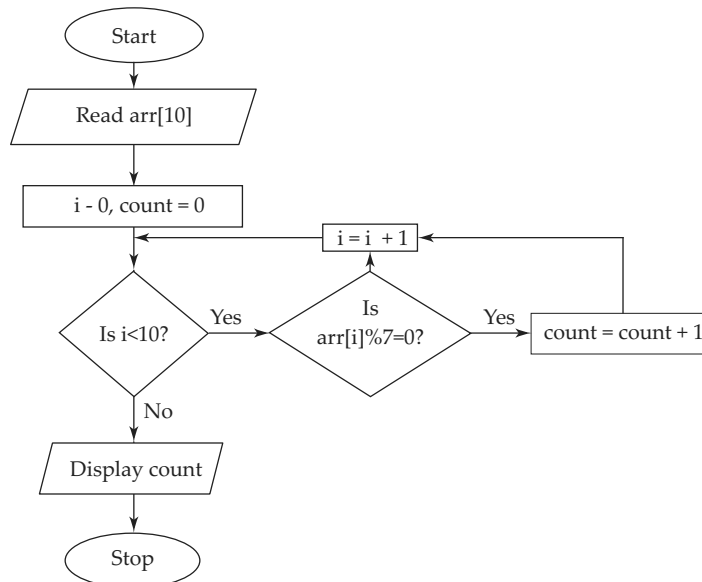
An algorithm is a sequence of steps written in the form of English phrases that specify the tasks that are performed while solving a problem. It involves identifying the variable names and types that would be used for solving the problem. Algorithms help a programmer in breaking down the solution of a problem into a number of sequential steps. Corresponding to each step a statement is written in a programming language; all these statements are collectively termed as a program.

Algorithm for Determining if the Given Number is Prime

```

Step 1 - Start
Step 2 - Accept a number from the user (num)
Step 3 - Initialize looping counter i = 2
Step 4 - Repeat Step 5 while i < num
Step 5 - If remainder of num divided by i (num%i) is Zero then goto Step 6 Else increment i and goto Step 4
Step 6 - Display "num is not a prime number" and break from the loop
Step 7 - If i = num then goto Step 8 Else goto Step 9
Step 8 - Display "num is a prime number"
Step 9 - Stop
    
```

1 (b). Flowchart



2(a). Here we are assuming that the resultant expression values are stored in the 'result' variable.

(i).

```
result = 3*x*x*x*x + x*x*x - 4*x*x + 7*x
```

(ii).

```
if(a>b)
    if(a>c)
        printf("%d is largest",a);
    else
        printf("%d is largest",c);
else
    if(b>c)
        printf("%d is largest",b);
    else
        printf("%d is largest",c);
```

(iii).

```
x = x / 100;
result = x % 10;
```

(iv).

```
if(x % 2 == 0)
    printf("True");
else
    printf("False");
```

(v).

```
z = z + ++x;
```

(vi).

```
if(x%3==0 && x%7==0)
    printf("True");
else
    printf("False");
```

2 (b). **Bitwise Operators** Refer Section 2.19.7

3 (a). **Recursion** Refer Section 3A.16

SET 2.4 Computer Programming

Program

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

void main()
{
    long unsigned x;
    int y;
    long unsigned fact(int);

    printf("\nEnter a number: ");
    scanf(" %d", &y);
    x = fact(y);

    printf("\nThe factorial of %d is %lu", y,x);
    getch();
}

long unsigned fact(int n)
{
    long unsigned f;
    if( n == 0 )
        return(1);
    else
        f = n*fact(n - 1);
    return(f);
}
```

Output

```
Enter a number: 5

The factorial of 5 is 120
```

3 (b). C Preprocessor Refer Section 3A.19

4 (a).

Program

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main()
{
```

```

char chk='t', str[30];
int len,left,right;
printf("\nEnter a string: ");
scanf("%s", &str);
len=strlen(str);
left=0;
right=len-1;
while(left < right && chk=='t')
{
    if(str[left]==str[right])
        ;
    else
        chk='f';
    left++;
    right--;
}
if(chk=='t')
    printf("\nThe sting %s is a palindrome",str);
else
    printf("\nThe sting %s is not a palindrome",str);
getch();
}

```

Output

```

Enter a string: nitin

The sting nitin is a palindrome

```

4 (b). Memory Allocation Functions**Malloc**

Malloc function is used to allocate a block of memory. It reserves a block of memory of specified size and returns a pointer of type void. This means that any type of pointer can be assigned to it.

The syntax of using malloc function is shown below:

```
ptr = (cast-type *) malloc (byte-size);
```

In the above syntax:

- ptr is a pointer of type cast-type
- byte-size is the size of the memory that is reserved.
- malloc returns a pointer (of cast-type) to an area of memory with size byte-size.

Example:

```
x = (int *) malloc (100*sizeof(int));
```

SET 2.6 Computer Programming

Calloc

Calloc function is used for allocating memory space at run time for storing derived types such as arrays and structures. While malloc allocates a single block of storage space, calloc allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The syntax of calloc is:

```
ptr = (cast-type *) calloc (n, elem-size);
```

In the above syntax:

- calloc allocates contiguous space of n blocks, each of size elem-size bytes.
- All bytes are initialized to zero
- Pointer to the first byte is allocated to ptr.
- If there is not enough memory space, NULL pointer is returned.

Example:

```
st_ptr = (record *) calloc (30, sizeof(record));
```

5 (a). Pointers to structures Refer Section 5.11

5 (b). Self-referential structures

A self-referential structure is a structure that contains member pointers pointing to itself. That means, it declares one or more of its members as pointers to itself. Such types of structures are commonly used during linked implementation of data structures. For example, consider the following structure declaration representing a queue node:

```
struct queue
{
    int info;
    struct queue *next;
};
```

In the above declaration, next is a pointer to the same structure of which it is a part of. Logically, next is being used here to point to the next node in the queue.

5 (c). Unions Refer Section 5.13

6 (a). fopen() Modes Refer Section 6.3

6 (b).

Program

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

/*first.dat and second.dat are the two source files containing integer values*/

void main()
{
    FILE *fs,*ft;/*Declaring file access pointers*/
    int c;
```

```
fs=fopen("first.dat","r");/*Opening the 1st source file*/
if(fs==NULL)
{
    printf("Cannot open source file");
    exit(1);
}

ft=fopen("result.dat","w");/*Opening the target file*/
if(ft==NULL)
{
    printf("Cannot open target file");
    fclose(fs);
    exit(0);
}

for(;fscanf(fs,"%d",&c)!=EOF;)
    fprintf(ft," %d ",(c));

/*Closing the file streams*/
fclose(fs);
fclose(ft);

fs=fopen("second.dat","r");/*Opening the 2nd source file*/
if(fs==NULL)
{
    printf("Cannot open source file");
    exit(1);
}

ft=fopen("result.dat","r+");/*Opening the target file again*/
if(ft==NULL)
{
    printf("Cannot open target file");
    fclose(fs);
    exit(0);
}

fseek(ft,0L,2);/*Using fseek to move to the end of the file*/

for(;fscanf(fs,"%d",&c)!=EOF;)
    fprintf(ft," %d ",(c));
```

SET 2.8 Computer Programming

```
/*Closing the file streams*/
fclose(fs);
fclose(ft);
printf("Files merged successfully; you can check the output by opening result.
dat file");

getch();
}
```

Output

```
Files merged successfully; you can check the output by opening result.dat file
```

7 (a). Insertion Sort Refer Section 7.2.6

7 (b). Applying Insertion Sort

<i>Initial Array:</i>	68	57	99	33	122	200
<i>Pass 1:</i>	57	68	99	33	122	200
<i>Pass 2:</i>	57	68	99	33	122	200
<i>Pass 3:</i>	33	57	68	99	122	200
<i>Pass 3:</i>	33	57	68	99	122	200
<i>Pass 3:</i>	33	57	68	99	122	200 (<i>Sorted Array</i>)

8. Linked List and its Operations Refer Section 8.3

Code.No: 09AIEC01

R09

SET-3

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

I B.TECH - REGULAR EXAMINATIONS, JUNE - 2011

C PROGRAMMING AND DATA STRUCTURES

**(COMMON TO CE, EEE, ME, ECE, CSE, CHEM, EIE, BME, AE, AME, BT, ECOMPE,
ETM, IT, ICE, MCT, MMT, MIE, MIM)**

Time: 3hours

Max.Marks:75

**Answer any FIVE questions
All questions carry equal marks**

1. (a) Write an algorithm to find out all the factors of a given positive integer.
(b) What is a flow chart? Draw a flow chart to read ten integers and print the sum of squares of all ten values. [8+7]
2. (a) Write minimal C expressions for the following:
 - (i) $x^3 - 4x^2 + 7x - 12$
 - (ii) Absolute value of (a-b)
 - (iii) Remainder when unsigned integer variable x is divided by 8, using bitwise operators
 - (iv) True if the given positive integer x is even and is also a multiple of 7, false otherwise
 - (v) Minimum of the values of three variables a, b and c.
 - (vi) True if the given character variable c represents a numeral (that is '0' ... '9'), false otherwise.(b) Write a complete C program that reads a value in the range 1 to 12 and print the name of that month and the next month: Print error for any other input value. (For example, print "May followed by June" if the input is 5. Note that December is followed by January). [6+9]
3. (a) What is recursion? Write a complete C program that reads a positive integer N, compute the first N Fibonacci numbers using recursion and print the results. Illustrate how the results are computed when the value of N is 4?
(b) Explain how matrices can be represented using two dimensional arrays. Explain with code how Transpose of a matrix can be done. [8+7]
4. (a) Write a complete C program that displays the position or index in the string S where the string T begins. The program displays -1 if S does not contain T. For example, if S is "information processing" and T is "process", the value displayed is 12. The string S and T are supplied by the user.
(b) Explain the following.
 - (i) Array of pointers
 - (ii) Malloc function [9+6]
5. Explain the following with example:
 - (a) Self referential structures
 - (b) Typedef
 - (c) Enumerated types [9+5+5]

SET 3.2 Computer Programming

6. (a) Explain what is a text file and what is a binary file.
(b) Write a complete C program for finding the number of words in the given text file. Assume that the words are separated by blanks or tabs. [6+9]
7. (a) Write an algorithm or a C program for sorting integers in ascending order using selection sort.
(b) Illustrate the results for each pass of selection sort, for the following initial array of elements: [9+6]
23 78 45 8 32 56
8. Explain what is a queue and operations performed on queue. Provide C code for the same. [15]

ANSWERS

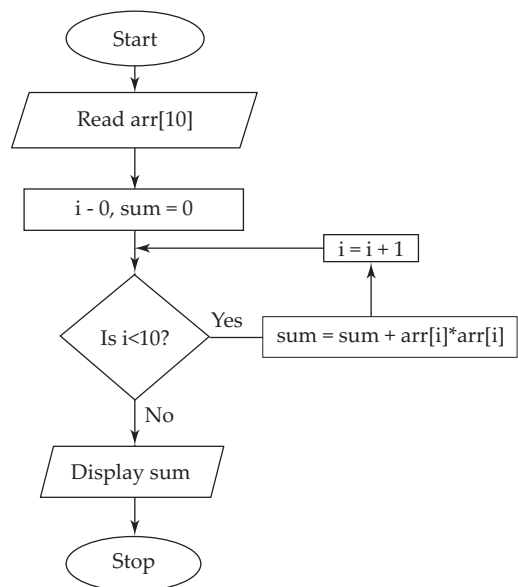
1 (a). Algorithm to Find the Factors of a Given Number

```
Step 1 - Start
Step 2 - Accept a number from the user (num)
Step 3 - Initialize looping counter i = 1
Step 4 - Repeat Step 5-7 while i <= num/2
Step 5 - If remainder of num divided by i (num % i) is Zero then goto Step 6
else goto Step 7
Step 6 - Display i
Step 7 - Set i = i + 1
Step 9 - Stop
```

1 (b). Flowchart

A flowchart can be defined as the pictorial representation of a process, which describes the sequence and flow of control and information in the process. The flow of information is represented in a flowchart in a step-by-step form. Different symbols are used for depicting different activities in the flowchart. Flowcharts are mainly used for developing business workflows and solving problems using computers.

Flowchart



2(a). Here we are assuming that the resultant expression values are stored in the 'result' variable.

(i).

```
result = x*x*x - 4*x*x + 7*x - 12
```

(ii).

```
result = abs(a-b);
```

(iii).

```
result = x & 7;
```

(iv).

```
if(x%2==0 && x%7==0)
    printf("True");
else
    printf("False");
```

(v).

```
if(a<b)
    if(a<c)
        printf("%d is smallest",a);
    else
        printf("%d is smallest",c);
else
    if(b<c)
        printf("%d is smallest",b);
    else
        printf("%d is smallest",c);
```

(vi).

```
if(c>=48 && c<=57)
    printf("True");
else
    printf("False");
```

2 (b).

Program

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

SET 3.4 Computer Programming

```
void main()
{
    char month[12][20] = {"January","February","March","April","May","June","July",
    "August","September","October","November","December"};
    int i;

    printf("Enter the month value: ");
    scanf("%d",&i);

    if(i<1 || i>12)
    {
        printf("Incorrect Value!!\nPress any key to terminate the program...");
        getch();
        exit(0);
    }

    if(i!=12)
        printf("%s followed by %s",month[i-1],month[i]);
    else
        printf("%s followed by %s",month[i-1],month[0]);

    getch();
}
```

Output

```
Enter the month value: 6
June followed by July
```

3 (a). Recursion Refer Section 3A.16

Program

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

void main()
{
    int x=0,y=1,n;
    void fib(int,int,int);/*Function prototype*/

    printf("Enter the number of terms in Fibonacci series: ");
    scanf("%d",&n); /*Reading number of terms in the series*/
```

```

printf("\nThe Fibonacci series is:");
printf("\n\n%d\t%d",x,y); /*Printing 1st two terms of the series*/

fib(x,y,n-2); /*Function Call*/
printf(" ");

getch();
}

void fib(int a,int b,int n)
{
    int c;
    if(n==0)
        return;
    n--;
    c=a+b;
    printf("\t%d",c);
    fib(b,c,n);/*recursive function call*/
}

```

Output

```

Enter the number of terms in Fibonacci series: 4

The Fibonacci series is:

0      1      1      2

```

Here, the value of n is 4. Thus, four terms of the Fibonacci series are required to be printed. The program is designed in such a way that the first two terms are printed by the main function by default. Then, the remaining n-2 terms are printed by calling the recursive function fib(). In this case, the remaining two terms are printed by the fib() function.

**3 (b). Using 2-D Arrays for Matrix representation
Program for Finding transpose of a Matrix**

Refer Section 3B.5

Refer Section 3B.11 (Example 3B.21)

4 (a).

Program

```

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

void main()

```

SET 3.6 Computer Programming

```
{
    char S[30],T[30];
    char *pos,*ptr;

    ptr=&S[0];
    printf("Enter String S: ");
    gets(S);

    printf("Enter String T: ");
    gets(T);

    pos=strstr(S,T);

    if(pos==NULL)
        printf("\n\n-1");
    else
        printf("Position of T in S is %d",pos-ptr);
    getch();
}
```

Output

```
Enter String S: Computer Programming
Enter String T: Program
Position of T in S is 9
```

4 (b). (i) Array of Pointers Refer Section 4A.12

4 (b). (ii) Malloc Function

Malloc function is used to allocate a block of memory. It reserves a block of memory of specified size and returns a pointer of type void. This means that any type of pointer can be assigned to it.

The syntax of using malloc function is shown below:

```
ptr = (cast-type *) malloc (byte-size);
```

In the above syntax:

- ptr is a pointer of type cast-type
- byte-size is the size of the memory that is reserved.
- malloc returns a pointer (of cast-type) to an area of memory with size byte-size.

Example:

```
x = (int *) malloc (100*sizeof(int));
```

5 (a) Self referential structures

A self-referential structure is a structure that contains member pointers pointing to itself. That means, it declares one or more of its members as pointers to itself. Such types of structures are commonly used during linked implementation of data structures. For example, consider the following structure declaration representing a queue node:

```
struct queue
{
    int info;
    struct queue *next;
};
```

In the above declaration, next is a pointer to the same structure of which it is a part of. Logically, next is being used here to point to the next node in the queue.

5 (b) Typedef Refer Section 5.16**5 (c) Enumerated types** Refer Section 5.17**6 (a) Text and Binary File** Refer Section 6.2**6 (b) Program** Refer Section 5.18**7 (a) Selection Sort** Refer Section 7.2.4**7 (b) Applying Selection Sort**

<i>Initial Array:</i>	23	78	45	8	32	56
<i>Pass 1:</i>	8	78	45	23	32	56
<i>Pass 2:</i>	8	23	45	78	32	56
<i>Pass 3:</i>	8	23	32	78	45	56
<i>Pass 4:</i>	8	23	32	45	78	56
<i>Pass 5:</i>	8	23	32	45	56	78

8. Queue and its Operations Refer Section 8.7 and 8.8.

Code.No: 09AIEC01

R09

SET-4

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

I B.TECH - REGULAR EXAMINATIONS, JUNE - 2011

C PROGRAMMING AND DATA STRUCTURES

**(COMMON TO CE, EEE, ME, ECE, CSE, CHEM, EIE, BME, AE, AME, BT, ECOMPE,
ETM, IT, ICE, MCT, MMT, MIE, MIM)**

Time: 3hours

Max.Marks:75

**Answer any FIVE questions
All questions carry equal marks**

1. (a) Write an algorithm to read ten positive integers and find out how many are perfect squares (such as 49, 81). You may assume that the input values read are in the range 1 to 10000.
(b) List the various steps in software development. [8+7]
2. (a) Write minimal C expressions for the following:
 - (i) $x^3 - 3x^2 + 3x - 1$
 - (ii) Digit at the 10's place of the given positive integer x (for example, digit at the 10's place in 3458 is 5)
 - (iii) True if the given positive integer x is a multiple of both 17 and 11, false otherwise.
 - (iv) Remainder when unsigned integer variable x is divided by 8, using bitwise operators
 - (v) True if $25 > a \geq 10$, false otherwise
 - (vi) Second bit from the right if the number x is treated in binary representation(b) Write a complete C program to read ten integers and find:
 - (i) The number of even integers and their sum, and
 - (ii) The number of odd integers and their sum[6+9]
3. (a) Write a complete C program to perform these functions:
 - (i) to return the factorial of the given number using recursion, and
 - (ii) to return the factorial of the given number using iteration(b) Write a complete C program to do the following: Read data to fill a two dimensional array *int table* [4] [4]. Then print the sum of each column and sum of each row. [8+7]
4. (a) Write the C function *int minpos (float x[], int n)* that returns the position of the first minimum value among the first *n* elements of the given array *x*.
(b) Explain the use of functions *strcpy ()* and *strcmp ()* [9+6]
5. (a) Explain how complex numbers can be represented using structures. Write two C functions: one to return the sum of two complex numbers passed as parameters, and another to return the product of two complex numbers passed as parameters.

SET 4.2 Computer Programming

- (b) Explain the following with example:
(i) Enumerated types (ii) Unions [9+6]
6. Write a complete C program to reverse the first n characters in a file. The file name and the value n are specified on the command line. Incorporate validation of arguments: that is, the program should check that the number of arguments passed and also the value of n are meaningful. [15]
7. (a) Write an algorithm or program for binary search to find a given integer in an array of integers.
(b) Illustrate the results of bubble sort for each pass, for the following initial array of elements:
44 36 57 19 25 89 28 [8+7]
8. (a) Explain the properties of the abstract data structure stack.
(b) Explain the algorithm to convert infix expression to postfix expression. [7+8]

ANSWERS

1 (a). Algorithm to Find Perfect Squares

```
Step 1 - Start
Step 2 - Accept 10 integers from the user and store them in an array (arr[])
Step 3 - Initialize looping counter i = 0
Step 4 - Repeat Steps 5-15 while i <= 9
Step 5 - If arr[i]<0 OR arr[i]>10000 goto Step 6 else goto Step 7
Step 6 - Display message "arr[i] is out of range", increment i by 1 and goto step 4
Step 7 - Initialize looping counter j = 1
Step 9 - Repeat Step 10-12 while j < arr[i]
Step 10 - if arr[i]%j = 0 goto Step 11 else increment j by 1 and goto Step 9
Step 11 - if j*j = arr[i] goto Step 12 else increment j by 1 and goto Step 9
Step 12 - Display "arr[i] is a perfect square", increment i by 1 and goto step 4
Step 13 - if j = arr[i] goto Step 14 else goto Step 15
Step 14 - Display message "arr[i] is not a perfect square"
Step 15 - Increment i by 1 and goto Step 4
Step 16 - Stop
```

1 (b). Software Development Steps Refer Section 1.6

2 (a) Here we are assuming that the resultant expression values are stored in the 'result' variable.

(i).

```
result = x*x*x - 3*x*x + 3*x - 1
```

(ii).

```
x = x / 10;
result = x % 10;
```

(iii).

```
if(x%17==0 && x%11==0)
    printf("True");
else
    printf("False");
```

(iv).

```
result = x & 7;
```

(v).

```
if(a<25 && a>=10)
    printf("True");
else
    printf("False");
```

(vi).

```
result = 2&x;
if(result>0)
    printf("bit = 1");
else
    printf("bit = 0");
```

2 (b).

Program

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

void main()
{
    int i;
    int counte,counto,sume,sumo;
    int arr[10];
    counte=counto=sume=sumo=0;

    printf("Enter 10 integers: \n");
    for(i=0;i<=9;i++)
    {
        printf("arr[%d] = ",i);
        scanf("%d",&arr[i]);
    }
}
```

SET 4.4 Computer Programming

```
for(i=0;i<=9;i++)
{
    if(arr[i]%2==0)
    {
        counte=counte+1;
        sume=sume+arr[i];
    }
    else
    {
        counto=counto+1;
        sumo=sumo+arr[i];
    }
}

printf("There are %d even numbers and their sum is %d\n",counte,sume);
printf("There are %d odd numbers and their sum is %d",counto,sumo);
getch();
}
```

Output

```
Enter 10 integers:
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
arr[5] = 6
arr[6] = 7
arr[7] = 8
arr[8] = 9
arr[9] = 10
There are 5 even numbers and their sum is 30
There are 5 odd numbers and their sum is 25
```

3 (a) (i). Factorial Program Using Recursion

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

void main()
{
    long unsigned x;
```

```
int y;
long unsigned fact(int);

printf("\nEnter a number: ");
scanf("%d", &y);
x = fact(y);

printf("\nThe factorial of %d is %lu", y,x);
getch();
}

long unsigned fact(int n)
{
    long unsigned f;
    if( n == 0 )
        return(1);
    else
        f = n*fact(n - 1);
    return(f);
}
```

Output

```
Enter a number: 5

The factorial of 5 is 120
```

3 (a) (ii). Factorial Program Using Iteration

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

void main()
{
    long unsigned x=1;
    int y,i;

    printf("Enter a number: ");
    scanf("%d", &y);
    for(i=y;i>=1;i--)
        x=x*i;
    printf("\nThe factorial of %d is %lu", y,x);
    getch();
}
```

SET 4.6 Computer Programming

```
}
```

Output

```
Enter a number: 5

The factorial of 5 is 120
```

3 (b).

Program

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

void main()
{
    int i,j,table[4][4];
    int sumr,sumc;

    printf("Enter the elements of the 4 X 4 array:\n");
    for(i=0;i<4;i++)
        for(j=0;j<4;j++)
        {
            printf("table[%d][%d] = ",i+1,j+1);
            scanf("%d",&table[i][j]);
        }

    printf("The various elements contained in the 4 X 4 array are:\n");
    for(i=0;i<4;i++)
    {
        printf("\n\t\t\t");
        for(j=0;j<4;j++)
            printf("%d\t",table[i][j]);
    }

    printf("\n\nSum of each column and row is:\n");

    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
            sumr=sumr+table[i][j];
        printf("Row %d = %d\n",i+1,sumr);
        sumr=0;
    }
}
```

```
}  
printf("\n\n");  
  
for(j=0;j<4;j++)  
{  
    for(i=0;i<4;i++)  
        sumc=sumc+table[i][j];  
    printf("Column %d = %d\n",j+1,sumc);  
    sumc=0;  
}  
  
getch();  
}
```

Output

```
table[3][3] = 3  
table[3][4] = 3  
table[4][1] = 4  
table[4][2] = 4  
table[4][3] = 4  
table[4][4] = 4  
The various elements contained in the 4 X 4 array are:
```

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Sum of each column and row is:

Row 1 = 4
Row 2 = 8
Row 3 = 12
Row 4 = 16

Column 1 = 10
Column 2 = 10
Column 3 = 10
Column 4 = 10

SET 4.8 Computer Programming

4 (a).

Program

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

int minpos(float [], int);
void main()
{
    int n;
    float x[10] = {12.5, 3.0, 45.1, 8.2, 19.3, -10.0, 7.8, 23.7, 29.9, 5.2};
    printf("Enter the value of n: ");
    scanf("%d",&n);

    if(n>=1 && n<=10)
        ;
    else
    {
        printf("Invalid value of n...Press any key to terminate the program..");
        getch();
        exit(0);
    }

    printf("Within the first %d elements of array, the first minimum value is
    stored at index %d",n,minpos(x,n));
    getch();
}

int minpos(float a[],int N)
{
    int i,index;
    float min=9999.99;
    for(i=0;i<N;i++)
        if(a[i]<min)
        {
            min=a[i];
            index = i;
        }
    return(index);
}
```


Output

```
Enter the value of n: 5
Within the first 5 elements of array, the first minimum value is stored at index 1
```

4 (b).

strcpy() Refer Section 4B.8.3

strcmp() Refer Section 4B.8.6

5 (a). Representing Complex Numbers Using Structures

A complex number has two parts: real and imaginary. Structures can be used to realize complex numbers in C, as shown below:

```
struct complex /*Declaring the complex number datatype using structure*/
{
    double real;/*Real part*/
    double img;/*Imaginary part*/
};
```

Function to return the sum of two complex numbers

```
struct complex add(struct complex c1, struct complex c1)
{
    struct complex c3;
    c3.real=c1.real+c2.real;
    c3.img=c1.img+c2.img;
    return(c3);
}
```

Function to return the product of two complex numbers

```
struct complex product(struct complex c1, struct complex c1)
{
    struct complex c3;
    c3.real=c1.real*c2.real-c1.img*c2.img;
    c3.img=c1.real*c2.img+c1.img*c2.real;
    return(c3);
}
```

5 (b) (i). Enumerated Types Refer Section 5.17

5 (b) (ii). Unions Refer Section 5.13

SET 4.10 Computer Programming

6.

Program

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

void main(int argc, char *argv[])
{
    FILE *fs;
    char str[100];
    int i,n,j;

    if(argc!=3)/*Checking the number of arguments given at command line*/
    {
        puts("Improper number of arguments.");
        exit(0);
    }

    n=atoi(argv[2]);
    fs = fopen(argv[1],"r");/*Opening the source file in read mode*/
    if(fs==NULL)
    {
        printf("Source file cannot be opened.");
        exit(0);
    }

    i=0;
    while(1)
    {
        if(str[i]=fgetc(fs)!=EOF)/*Reading contents of file character by character*/
            i=i+1;
        else
            break;
    }
    fclose(fs);

    fs = fopen(argv[1],"w");/*Opening the file in write mode*/
    if(n<0||n>strlen(str))
    {
        printf("Incorrect value of n. Program will terminate...\n\n");
        getch();
    }
}
```

```

    exit(1);
}

j=strlen(str);
for(i=1;i<=n;i++)
{
    fputc(str[j],fs);
    j--;
}
fclose(fs);

printf("\n%d characters of the file successfully printed in reverse order",n);
getch();
}

```

Output

```

D:\TC\BIN\program source.txt 5
5 characters of the file successfully printed in reverse order

```

7 (a). Binary Search Refer Section 7.3.2**7 (b). Applying Bubble Sort**

<i>Initial Array:</i>	44	36	57	19	25	89	28
<i>Pass 1:</i>	19	44	57	36	25	89	28
<i>Pass 2:</i>	19	25	57	44	36	89	28
<i>Pass 3:</i>	19	25	28	57	44	89	36
<i>Pass 4:</i>	19	25	28	36	57	89	44
<i>Pass 5:</i>	19	25	28	36	44	89	57
<i>Pass 6:</i>	19	25	28	36	44	57	89 (<i>Sorted Array</i>)

8 (a). Stack Properties

The various properties of stack data structure are:

- It allows insertion and deletion of elements only at one end called stack top.
- It is based on the Last-In-First-Out (LIFO) principle.
- It has significant importance in systems processes such as compilation and program control
- Two key operations associated with stacks are push and pop.
- It can be implemented either through arrays or linked lists.

8 (b). Infix to Postfix Conversion Refer Section 8.6.6

Code.No: 09AIEC01

R09

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
I B.TECH - REGULAR EXAMINATIONS, MAY/JUNE - 2012
COMPUTER PROGRAMMING AND DATA STRUCTURES
(COMMON TO ALL BRANCHES)

Time: 3hours

Max.Marks: 75

Answer any FIVE questions
All questions carry equal marks

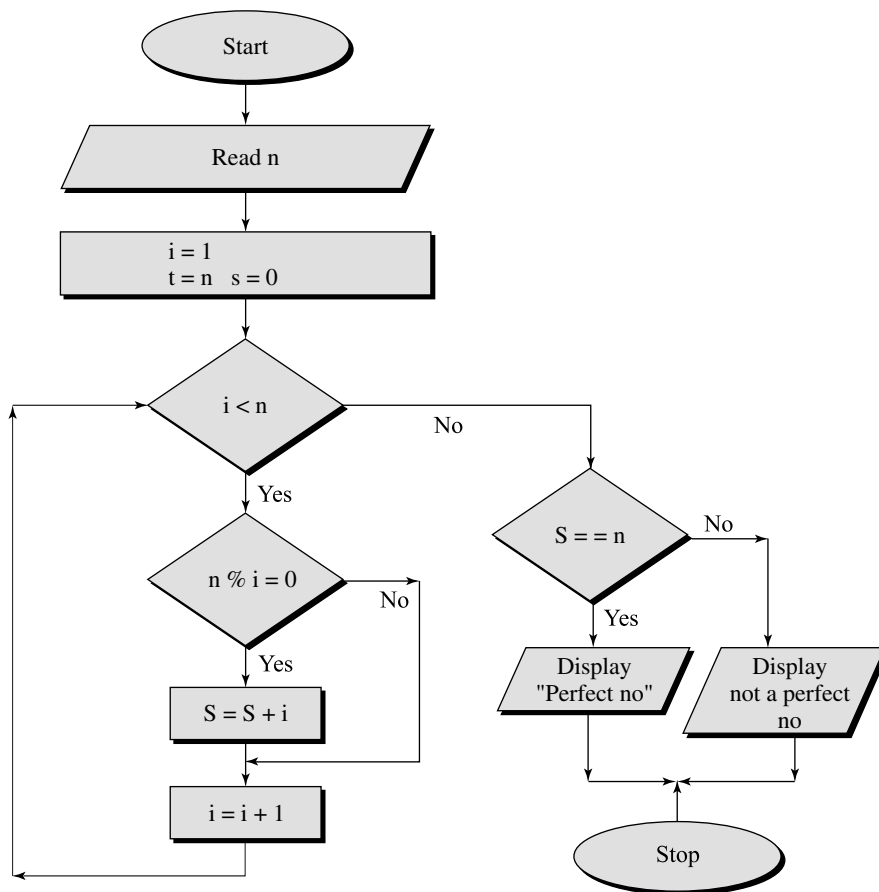
1. (a) Explain and specify the interactions between various components that support the basic functionality of a computer.
(b) Draw the flow chart to check whether a given number is perfect or not.
(c) Describe various categories of computing environments. [6+4+5]
2. (a) What is type conversion? Explain briefly about implicit and explicit type conversions.
(b) An integer is divisible by 9 if the sum of its digit is also divisible by 9. Write a C program that prompts the user to input in integer. The program should then output the number and a message stating whether the number is divisible by 9. [8+9]
3. (a) What is a storage class? Explain various storage classes in C with examples.
(b) Explain the differences between call-by-value and call-by-reference with suitable examples. [8+7]
4. (a) Explain how strings are declared and initialized in C?
(b) What are the arithmetic operators that are permitted on pointers?
(c) Write a 'C' program to reverse the string passed as an argument that cannot be altered. [4+4+7]
5. (a) What is a self referential structure? How it differs from nested structures. Explain with an illustrative example.
(b) Write a short note on typedef and enum. [9+6]
6. (a) What are the ways to set the file pointer randomly in a file? Explain.
(b) Write a 'C' program to copy the content of one file to another file. [8+7]
7. (a) Write a C program for binary search using recursion to find a given integer in an array of n elements.
(b) Illustrate the results for each pass of selection sort, for the following array of elements 2, 3, 78, 5, 46, 32, 56, 8, 100, 9. [8+7]
8. (a) Write a 'C' program to demonstrate the operations of a circular queue using arrays.
(b) Explain the procedure for converting infix expression to postfix expression with an example routine. [8+7]

ANSWERS

1. (a) Refer Section 1.2, Page 1.2

1. (b)

Flowchart



1. (c) Refer Section 1.4, Page 1.15

2. (a) Refer Section 2.9.12, Page 2.73

2. (b)

```

#include<stdio.h>
main()
{
    int s=0, n, no;
    printf("Enter the number");
    scanf("%d", &n);

```

```
no=n
while (no! = 0)
{
    s + = (no%10);
    no = no/10;
}
if (s%9 == 0)
{
    printf("%d is divisible by 9", n);
}
else
{
    printf("%d is not divisible by 9", n);
}
}
```

3. (a) Refer Section 3A.17, Page 3A.22

3. (b) **Call by value:** In this method, the value of a variable is passed to a function. The passed value is copied into another variable in the parameter list of the function. Thus any change made to the new variable's value inside the function is not reflected back in the main function. The following code explains this.

```
void change (int x)
{
    x = 100
}
void main()
{
    int num = 50;
    printf ("The number before function call %d", num);
    change (num);
    printf ("The number after the function call %d; num)
}
```

Output: The number before function call 50

The number after the function call 50

Call by reference:

In this method, the memory address of the variables rather than the copies of values are passed to the function. Thus any changes made to the values stored at the memory address stay. Permanent and get reflected back to the main function. The following code explains the same.

```
void change (int *p)
{
    *p=100;
}
```

SET 1.4 Computer Programming

```
void main()
{
    int n=160;
    print f ("The number before function call %d", n);
    change (&n);
    print f ("The number after the function call %d", n);
}
```

Output: The number before function call 160;
The number after the function call 100;

4. (a) Refer Section 4B.2, Page 4B.2

4. (b) Refer Sections 4A.8 and 4A.9

4. (c)

```
char*reverse (char *s);
void main()
{ char s[100], rs;
  printf("Enter the string");
  scant("%s", s);
  rs=reverse (s);
  printf("The reverse of string %s is %s", s, rs);
}
char* reverse (char *ps)
{    char rs[100];
  int n = strlen (ps);
  int i, f;
  for (i=n; j=0; i>0 i--)
  {rs[jtt] = ps[i];
  }
  rs[j] = '\0';
  return rs;
}
```

5. (a) Structures which contains a member field that points to the same structure field that points to the same structure type are called self referential structure.

```
struct node
{
    int item;
    struct node *next
};
```

Here node is having a pointer *next which points to the same node structure type.

Structure within structure means nesting of structures.

```
struct salary
{
    char name;
    char dept;
```



```
struct {
    int dearners;
    int h-rent;
    int city;
} allowance
} employee
```

The salary structure contains a member named allowance which itself is a structure with 3 members. The members contained in the inner structure namely dearness can be referred to as employee.allowance.dearness.

The inner structure can have many variables also.

Tag names can also be used to define

inner structure

```
struct pay
{
    int dearness;
    int house-rent;
    int city;
} allowance
} employee
```

The salary structure contains a member named allowance which itself is a structure with 3 members. The members contained in the inner structure namely dearness can be referred to as employee.allowance.dearness.

The inner structure can have many variables.

Tag names can also be used to define inner structure.

```
struct pay
{
    int dearness;
    int house-rent;
    int city;
} ;
struct salary
{
    char name;
    char dept;
    struct pay allowance;
};
```

5. (b) Refer Sections 5.16 and 5.17, Page 5.20

6. (a) Refer Section 6.7, Page 6.12

6. (b) Refer Example 6.9 Page 6.19

7. (a) Refer Section 7.3.2, Page 7.16

7. (b) 2, 3, 78, 5, 46, 32, 56, 8, 100, 9
is the original array

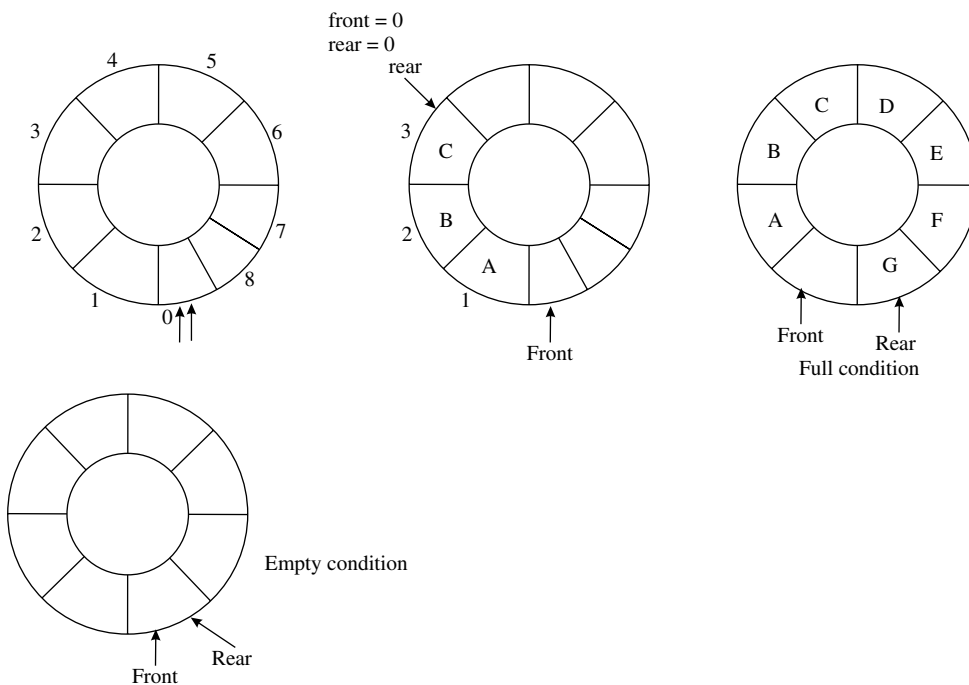
After pass-1 array content is

2 3 78 5 46 32 56 8 100 9

SET 1.6 Computer Programming

Pass 2										
Pass 3	2	3	78	5	46	32	56	8	100	9
Pass 4	2	3	5	78	46	32	56	8	100	9
Pass 5	2	3	5	8	46	32	56	78	100	9
Pass 6	2	3	5	8	9	32	56	78	100	46
Pass 7	2	3	5	8	9	32	56	78	100	46
Pass 8	2	3	5	8	9	32	46	78	100	56
Pass 9	2	3	5	8	9	32	46	56	78	100
	2	3	5	8	9	32	46	56	78	100

8. (a)



```
#include<stdio.h>
#define size 100
int queue [100].
int front = -1;
int rear = -1
void add (int);    || add an ele to queue
int rem()         || removes the element
```

```
void display(void) || Display the element
void main()
{   int n1,n2, choice;
    while(1)
    {printf("\n select an option ");
      printf("\n 1-Insert element");
      printf("\n 2-Remove element");
      printf("\n 3-Display");
      printf("\n 4-Exit");
      printf("Enter your choice");
      scans ("%d"; & choice);
      switch(choice)
      {
        Case 1:   printf("\n Enter the element to be inserted");
                  scanf("%d", &n1)
                  add(n1);
                  break;
        Case 2:
                  n2 = rem();
                  printf("%d is the deleted element", n2);
                  break;
        Case 3:
                  display();
                  break;
        Case 4:
                  exit(1);
                  break
      }
    }
}
void display()
{
    int i;
    if (front == rear)
        printf("queue empty");
    else
        for (i=front+1; i!=rear; i=(i+1)%size)
            printf("%d", queue[i]);
}
void add (int value)
{
    if (front == rear + 1)%size)
        printf("queue is full)
    else
        rear = (rear + 1) % size
        queue [rear] = value
        return
    }
}
int rem()
{   int t;
    if (front==rear)
```

SET 1.8 *Computer Programming*

```
        { printf("queue is empty");  
          return(-9999);  
        }  
    else front = (front + 1)% size)  
    {    t = queue [front]  
        return t;  
    }  
}
```

8. (b) Refer Section 8.6.6, Page 8.32